User's Manual

Smart Configurator

User's Manual: RL78 API Reference

RENESAS MCU
RL78 Family

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

Renesas Electronics
www.renesas.com

Rev.1.04 Jul 2023

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.

© 2023 Renesas Electronics Corporation. All rights reserved.

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:	$\overline{XXX}$ (overscore over pin or signal name)
Note:	Footnote for item marked with Note in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numeric representation:	Decimal ... XXXX Hexadecimal ... 0xXXXX

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

TABLE OF CONTENTS

1. GENERAL........8

1.1 Overview........8

1.2 Features........8

2. OUTPUT FILES........9

2.1 Description........9

3. INITIALIZATION........30

4. API FUNCTIONS........31

4.1 Overview........31

4.2 Function Reference........32

4.2.1 General........33

4.2.2 Port........132

4.2.3 Delay Counter........138

4.2.4 Divider Function........148

4.2.5 External Event Counter (Timer Array Unit)........155

4.2.6 External Event Counter (Timer RJ)........164

4.2.7 Input Pulse High-/Low-Level Width Measurement (Timer Array Unit)........171

4.2.8 Input Pulse High-/Low-Level Width Measurement (Timer RJ)........179

4.2.9 PWM Output (Timer Array Unit)........187

4.2.10 PWM Output (Timer RDn using PWM mode/ Extended PWM mode)........195

4.2.11 PWM Output (Timer RD0 and RD1 using PWM mode/ PWM3 mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode)........203

4.2.12 PWM Output (Timer RG using PWM mode/ PWM2 mode)........213

4.2.13 PWM Output (Timer KB using standalone mode (period controlled by TKBCRn0 register)/standalone mode (period controlled by external trigger input)/interleave PFC output mode)........220

4.2.14 PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (1 slave)........245

4.2.15 PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (2 slaves)........270

4.2.16 Input Pulse Interval/Period Measurement (Timer Array Unit)........295

4.2.17 Input Pulse Interval/Period Measurement (Timer RJ)........303

4.2.18 Interval Timer (Timer Array Unit)........311

4.2.19 Interval Timer (Timer RJ)........324

4.2.20 Interval Timer (12-bit Interval Timer) 331
4.2.21 One-Shot Pulse Output 338
4.2.22 Square Wave Output (Timer Array Unit) 348
4.2.23 Square Wave Output (Timer RJ) 357
4.2.24 Interval Timer (32-bit Interval Timer using 8-bit counter mode) 364
4.2.25 Interval Timer (32-bit Interval Timer using 16-bit counter mode) 372
4.2.26 Interval Timer (32-bit Interval Timer using 32-bit counter mode) 382
4.2.27 Input Capture Function (Timer RD) 390
4.2.28 Input Capture Function (Timer RG) 399
4.2.29 Input Capture Function (Timer RX) 408
4.2.30 Output Compare Function (Timer RD) 416
4.2.31 Output Compare Function (Timer RG) 423
4.2.32 Three -phase PWM Output (Timer RD) 430
4.2.33 PWM option unit A (Timer RD) 439
4.2.34 Phase Counting Mode 444
4.2.35 Clock Output/Buzzer Output Controller 454
4.2.36 Real-Time Clock 460
4.2.37 A/D Converter 481
4.2.38 12 Bit A/D Single Scan 502
4.2.39 12 Bit A/D Continuous Scan 510
4.2.40 12 Bit A/D Group Scan 518
4.2.41 D/A Converter 529
4.2.42 Data Transfer Controller 536
4.2.43 Comparator 542
4.2.44 Programmable Gain Amplifier 549
4.2.45 SPI (CSI) Communication 555
4.2.46 UART Communication (Serial array unit) 568
4.2.47 UART Communication (Serial Interface UARTA) 585
4.2.48 UART Communication (LIN/UART module) 603
4.2.49 DALI Communication (Control devices) 617
4.2.50 DALI Communication (Control gear) 641
4.2.51 IIC Communication (Master mode) (Serial Array Unit) 661
4.2.52 IIC Communication (Master mode) (Serial Interface IICA) 675
4.2.53 IIC Communication (Slave mode) (Serial Interface IICA) 689
4.2.54 Interrupt Countroller 704
4.2.55 Voltage Detector 711
4.2.56 Snooze Mode Sequencer 717
4.2.57 Key Interrupt 732

4.2.58 Remote Control Signal Receiver 739
4.2.59 Watchdog Timer 755
4.2.60 Logic and Event Link Controller 761
4.2.61 Event Link Controller 768

Appendix API Function Comparison Table 773

Revision Record 779

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 (1/21)

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_linker_script.ld	—
	r_cg_port.h	—
	r_cg_pclbuz.h	—
	r_cg_kr.h	—
	r_cg_wdt.h	—
	r_cg_intc.h	—
	r_cg_sms.h	—
	r_cg_elc.h	—
	r_cg_dtc_common.c	R_DTC_Set_PowerOn R_DTC_Set_PowerOff
	r_cg_dtc_common.h	—
	r_cg_dtc.h	—
	r_cg_tau_common.c	R_TAU*m*_Create R_TAU*m*_Set_PowerOn R_TAU*m*_Set_PowerOff R_TAU*m*_Set_Reset R_TAU*m*_Release_Reset
	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_Set_PowerOn R_ITL_Set_PowerOff R_ITL_Set_Reset R_ITL_Release_Reset
	r_cg_itl_common_user.c	r_itl_interrupt
	r_cg_itl_common.h	—
	r_cg_itl.h	—

Table 2-2 Output File List (2/21)

Component / Folder Name	File Name	API Function Name
General	r_cg_trd_common.c	R_TRD_Create R_TRD_Set_PowerOn R_TRD_Set_PowerOff R_TRD_Set_Reset R_TRD_Release_Reset R_PWMOPA_Set_PowerOn R_PWMOPA_Set_PowerOff R_PWMOPA_Set_Reset R_PWMOPA_Release_Reset R_TRD_ForcedOutput_Enable R_TRD_ForcedOutput_Disable
	r_cg_trd_common.h	—
	r_cg_trd.h	—
	r_cg_trj_common.c	R_TRJ_Set_PowerOn R_TRJ_Set_PowerOff R_TRJ_Set_Reset R_TRJ_Release_Reset
	r_cg_trj_common.h	—
	r_cg_trj.h	—
	r_cg_trg_common.c	R_TRG_Set_PowerOn R_TRG_Set_PowerOff R_TRG_Set_Reset R_TRG_Release_Reset
	r_cg_trg_common.h	—
	r_cg_trg.h	—
	r_cg_trx_common.c	R_TRX_Set_PowerOn R_TRX_Set_PowerOff R_TRX_Set_Reset R_TRX_Release_Reset
	r_cg_trx_common.h	—
	r_cg_trx.h	—
	r_cg_tkb_common.c	R_TKB_Create R_TKB_Set_PowerOn R_TKB_Set_PowerOff R_TKB_Set_Reset R_TKB_Release_Reset
	r_cg_tkb_common.h	—
	r_cg_tkb.h	—
	r_cg_rtc_common.c	R_RTC_Set_PowerOn R_RTC_Set_PowerOff
	r_cg_rtc_common.h	—
	r_cg_rtc.h	—

Table 2-3 Output File List (3/21)

Component / Folder Name	File Name	API Function Name
General	r_cg_it_common.c	R_IT_Set_PowerOn R_IT_Set_PowerOff
	r_cg_it_common.h	—
	r_cg_it.h	—
	r_cg_ad_common.c	R_ADC_Set_PowerOn R_ADC_Set_PowerOff R_ADC_Set_Reset R_ADC_Release_Reset
	r_cg_ad_common.h	—
	r_cg_ad.h	—
	r_cg_da_common.c	R_DAC_Create R_DAC_Set_PowerOn R_DAC_Set_PowerOff R_DAC_Set_Reset R_DAC_Release_Reset
	r_cg_da_common.h	—
	r_cg_da.h	—
	r_cg_comp_common.c	R_COMP_Create R_COMP_Set_PowerOn R_COMP_Set_PowerOff R_COMP_Set_Reset R_COMP_Release_Reset
	r_cg_comp_common.h	—
	r_cg_comp.h	—
	r_cg_pgacomp_common.c	R_PGACOMP_Create R_PGACOMP_Set_PowerOn R_PGACOMP_Set_PowerOff R_PGACOMP_Set_Reset R_PGACOMP_Release_Reset
	r_cg_pgacomp_common.h	—
	r_cg_pgacomp.h	—
	r_cg_sau_common.c	R_SAU*m*_Create R_SAU*m*_Set_PowerOn R_SAU*m*_Set_PowerOff R_SAU*m*_Set_Reset R_SAU*m*_Release_Reset R_SAU*m*_Set_SnoozeOn R_SAU*m*_Set_SnoozeOff
	r_cg_sau_common.h	—
	r_cg_sau.h	—
	r_cg_uarta_common.c	R_UARTA_Create R_UARTA_Set_PowerOn R_UARTA_Set_PowerOff
	r_cg_uarta_common.h	—
	r_cg_uarta.h	—

Table 2-4 Output File List (4/21)

Component / Folder Name	File Name	API Function Name
General	r_cg_iica_common.c	R_IICAn_Set_PowerOn R_IICAn_Set_PowerOff R_IICAn_Set_Reset R_IICAn_Release_Reset
	r_cg_iica_common.h	—
	r_cg_iica.h	—
	r_cg_rlin3_common.c	R_RLIN3n_Set_PowerOn R_RLIN3n_Set_PowerOff
	r_cg_rlin3_common.h	—
	r_cg_rlin3.h	—
	r_cg_dali_common.c	R_DALI_Set_PowerOn R_DALI_Set_PowerOff R_DALI_Set_Reset R_DALI_Release_Reset
	r_cg_dali_common.h	—
	r_cg_dali.h	—
	r_cg_lvd_common.c	R_LVD_Start_Interrupt R_LVD_Stop_Interrupt
	r_cg_lvd_common_user.c	r_lvd_interrupt
	r_cg_lvd_common.h	—
	r_cg_lvd.h	—
	r_cg_remc_common.c	R_REMC_Set_PowerOn R_REMC_Set_PowerOff R_REMC_Set_Reset R_REMC_Release_Reset
	r_cg_remc_common.h	—
	r_cg_remc.h	—
Ports	{Config_PORT}.c	R_{Config_PORT}_Create R_{Config_PORT}_ReadPmnValues R_{Config_PORT}_ReadDigitalOutputLevel
	{Config_PORT}_user.c	R_{Config_PORT}_Create_UserInit
	{Config_PORT}.h	—

Table 2-5 Output File List (5/21)

Component / Folder Name	File Name	API Function Name
Delay Counter	{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	—
Divider Function	{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	—
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	—
External Event Counter (Timer RJ)	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_ interrupt
	{Config_TRJ*n*}.h	—
Input Pulse High-/Low-Level Width 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*}_Create_UserInit r_{Config_TAU*m*_*n*}_ interrupt
	{Config_TAU*m*_*n*}.h	—

Table 2-6 Output File List (6/21)

Component / Folder Name	File Name	API Function Name
Input Pulse High-/Low-Level Width Measurement (Timer RJ)	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop R_{Config_TRJ*n*}_Get_PulseWidth
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_ interrupt
	{Config_TRJ*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	—
PWM Output (Timer RD using PWM mode/ Extended PWM mode)	{Config_TRD*n*}.c	R_{Config_TRD*n*}_Create R_{Config_TRD*n*}_Start R_{Config_TRD*n*}_Stop R_{Config_TRD*n*}_Set_TRD*n*_ReloadTrigger
	{Config_TRD*n*}_user.c	R_{Config_TRD*n*}_Create_UserInit r_{Config_TRD*n*}_trd*n*_interrupt
	{Config_TRD*n*}.h	—
PWM Output (Timer RD0 and RD1 using PWM mode/ PWM3 mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode)	{Config_TRD0_TRD1}.c	R_{Config_TRD0_TRD1}_Create R_{Config_TRD0_TRD1}_Start R_{Config_TRD0_TRD1}_Stop R_{Config_TRD0_TRD1}_Set_TRD*n*_ReloadTrigger R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger
	{Config_TRD0_TRD1}_user.c	R_{Config_TRD0_TRD1}_Create_UserInit r_{Config_TRD0_TRD1}_trd*n*_interrupt
	{Config_TRD0_TRD1}.h	—
PWM Output (Timer RG using PWM mode/ PWM2 mode)	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt
	{Config_TRG}.h	—

Table 2-7 Output File List (7/21)

Component / Folder Name	File Name	API Function Name
PWM Output (Timer KB using standalone mode (period controlled by TKBCRn0 register)/standalone mode (period controlled by external trigger input)/interleave PFC output mode)	{Config_TKB*n*}.c	R_{Config_TKB*n*}_Create R_{Config_TKB*n*}_Start R_{Config_TKB*n*}_Stop R_{Config_TKB*n*}_Set_BatchOverwriteRequestOn R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Start R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Stop R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Start R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Stop R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Start R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Start R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Stop R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Start R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Stop R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Start R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Stop
	{Config_TKB*n*}_user.c	R_{Config_TKB*n*}_Create_UserInit r_{Config_TKB*n*}_terminated0_interrupt r_{Config_TKB*n*}_terminated1_interrupt r_{Config_TKB*n*}_activated0_interrupt r_{Config_TKB*n*}_activated1_interrupt r_{Config_TKB*n*}_end_count_interrupt
	{Config_TKB*n*}.h	—

Table 2-8 Output File List (8/21)

Component / Folder Name	File Name	API Function Name
PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (1 slave)	{Config_TKB0_TKB*n*}.c	R_{Config_TKB0_TKB*n*}_Create R_{Config_TKB0_TKB*n*}_Start R_{Config_TKB0_TKB*n*}_Stop R_{Config_TKB0_TKB*n*}_TKB*m*_Set_BatchOverwriteRequestOn R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Start R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Start R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*0_DitheringFunction_Start R_{Config_TKB0_TKB*n*}_TKBO*m*0_DitheringFunction_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*1_DitheringFunction_Start R_{Config_TKB0_TKB*n*}_TKBO*m*1_DitheringFunction_Stop
	{Config_TKB0_TKB*n*}_user.c	R_{Config_TKB0_TKB*n*}_Create_UserInit r_{Config_TKB0_TKB*n*}_tkb*m*_terminated0_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_terminated1_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_activated0_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_activated1_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_end_count_interrupt
	{Config_TKB0_TKB*n*}.h	—

Table 2-9 Output File List (9/21)

Component / Folder Name	File Name	API Function Name
PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (2 slaves)	{Config_TKB0_TKB1_TKB2}.c	R_{Config_TKB0_TKB1_TKB2}_Create R_{Config_TKB0_TKB1_TKB2}_Start R_{Config_TKB0_TKB1_TKB2}_Stop R_{Config_TKB0_TKB1_TKB2}_TKB*n*_Set_BatchOverwriteRequestOn R_{Config_TKB_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_DitheringFunction_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_DitheringFunction_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_DitheringFunction_Start R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Stop
	{Config_TKB0_TKB1_TKB2}_user.c	R_{Config_TKB0_TKB1_TKB2}_Create_UserInit r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated0_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated1_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated0_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated1_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_end_count_interrupt
	{Config_TKB0_TKB1_TKB2}.h	—

Table 2-10 Output File List (10/21)

Component / Folder Name	File Name	API Function Name
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	—
Input Pulse Interval/Period Measurement (Timer RJ)	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop R_{Config_TRJ*n*}_Get_PulseWidth
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_ interrupt
	{Config_TRJ*n*}.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	—
Interval Timer (Timer RJ)	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_interrupt
	{Config_TRJ*n*}.h	—
Interval Timer (12-bit Interval Timer)	{Config_IT}.c	R_{Config_IT}_Create R_{Config_IT}_Start R_{Config_IT}_Stop
	{Config_IT}_user.c	R_{Config_IT}_Create_UserInit r_{Config_IT}_interrupt
	{Config_IT}.h	—

Table 2-11 Output File List (11/21)

Component / Folder Name	File Name	API Function Name
One-Shot Pulse Output	{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	—
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	—
Square Wave Output (Timer RJ)	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_interrupt
	{Config_TRJ*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_SoftwareTriggerOn 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 2-12 Output File List (12/21)

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	—
Input Capture Function (Timer RD)	{Config_TRD*n*}.c	R_{Config_TRD*n*}_Create R_{Config_TRD*n*}_Start R_{Config_TRD*n*}_Stop R_{Config_TRD*n*}_Get_PulseWidth
	{Config_TRD*n*}_user.c	R_{Config_TRD*n*}_Create_UserInit r_{Config_TRDn}_trd*n*_interrupt
	{Config_TRD*n*}.h	—
Input Capture Function (Timer RG)	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop R_{Config_TRG}_Get_PulseWidth
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt
	{Config_TRG}.h	—
Input Capture Function (Timer RX)	{Config_TRX}.c	R_{Config_TRX}_Create R_{Config_TRX}_Start R_{Config_TRX}_Stop R_{Config_TRX}_Get_BuffValue
	{Config_TRX}_user.c	R_{Config_TRX}_Create_UserInit r_{Config_TRX}_interrupt
	{Config_TRX}.h	—
Output Compare Function (Timer RD)	{Config_TRD*n*}.c	R_{Config_TRD*n*}_Create R_{Config_TRD*n*}_Start R_{Config_TRD*n*}_Stop
	{Config_TRD*n*}_user.c	R_{Config_TRD*n*}_Create_UserInit r_{Config_TRDn}_trdn_interrupt
	{Config_TRD*n*}.h	—

Table 2-13 Output File List (13/21)

Component / Folder Name	File Name	API Function Name
Output Compare Function (Timer RG)	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt
	{Config_TRG}.h	—
Three -phase PWM Output (Timer RD)	{Config_TRD0_TRD1}.c	R_{Config_TRD0_TRD1}_Create R_{Config_TRD0_TRD1}_Start R_{Config_TRD0_TRD1}_Stop R_{Config_TRD0_TRD1}_Set_TRD_ReloadTrigger
	{Config_TRD0_TRD1}_user.c	R_{Config_TRD0_TRD1}_Create_UserInit r_{Config_TRD0_TRD1}_trd0_Interrupt r_{Config_TRD0_TRD1}_trd1_Interrupt
	{Config_TRD0_TRD1}.h	—
PWM option unit A (Timer RD)	{Config_PWMOPA}.c	R_{Config_PWMOPA}_Create R_{Config_PWMOPA}_Software_Release
	{Config_PWMOPA }_user.c	R_{Config_PWMOPA}_Create_UserInit
	{Config_PWMOPA }.h	—
Phase counting mode	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop R_{Config_TRG}_Get_MeasurementCapture
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt r_{Config_TRG}_clear_interrupt r_{Config_TRG}_capture_interrupt
	{Config_TRG}.h	—
Clock Output/Buzzer Output Controller	{Config_PCLBUZ*n*}.c	R_{Config_PCLBUZ*n*}_Create R_{Config_PCLBUZ*n*}_Start R_{Config_PCLBUZ*n*}_Stop
	{Config_PCLBUZ*n*}_user.c	R_{Config_PCLBUZ*n*}_Create_UserInit
	{Config_PCLBUZ*n*}.h	—

Table 2-14 Output File List (14/21)

Component / Folder Name	File Name	API Function Name
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	—
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}_ADS*n*_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 R_{Config_ADC}_ADS*n*_Get_Result_10bit R_{Config_ADC}_ADS*n*_Get_Result_8bit R_{Config_ADC}_ADS*n*_Get_Result_12bit
	{Config_ADC}_user.c	R_{Config_ADC}_Create_UserInit r_{Config_ADC}_interrupt r_{Config_ADC}_ad*n*_interrupt
	{Config_ADC}.h	—

Table 2-15 Output File List (15/21)

Component / Folder Name	File Name	API Function Name
12 Bit A/D Single Scan	{Config_S12ADn}.c	R_{Config_S12ADn}_Create R_{Config_S12ADn}_Start R_{Config_S12ADn}_Stop R_{Config_S12ADn}_Get_ValueResult
	{Config_S12ADn}_user.c	R_{Config_S12ADn}_Create_UserInit r_{Config_S12ADn}_interrupt
	{Config_S12ADn}.h	—
12 Bit A/D Continuous Scan	{Config_S12ADn}.c	R_{Config_S12ADn}_Create R_{Config_S12ADn}_Start R_{Config_S12ADn}_Stop R_{Config_S12ADn}_Get_ValueResult
	{Config_S12ADn}_user.c	R_{Config_S12ADn}_Create_UserInit r_{Config_S12ADn}_interrupt
	{Config_S12ADn}.h	—
12 Bit A/D Group Scan	{Config_S12ADn}.c	R_{Config_S12ADn}_Create R_{Config_S12ADn}_Start R_{Config_S12ADn}_Stop R_{Config_S12ADn}_Get_ValueResult
	{Config_S12ADn}_user.c	R_{Config_S12ADn}_Create_UserInit r_{Config_S12ADn}_interrupt r_{Config_S12ADn}_groupb_interrupt
	{Config_S12ADn}.h	—
D/A Converter	{Config_DACn}.c	R_{Config_DACn}_Create R_{Config_DACn}_Start R_{Config_DACn}_Stop R_{Config_DACn}_Set_ConversionValue
	{Config_DACn}_user.c	R_{Config_DACn}_Create_UserInit
	{Config_DACn}.h	—
Data Transfer Controller	{Config_DTC}.c	R_{Config_DTC}_Create R_{Config_DTCDn}_Start R_{Config_DTCDn}_Stop
	{Config_DTC}_user.c	R_{Config_DTC}_Create_UserInit
	{Config_DTC}.h	—
Comparator	{Config_COMPn}.c	R_{Config_COMPn}_Create R_{Config_COMPn}_Start R_{Config_COMPn}_Stop
	{Config_COMPn}_user.c	R_{Config_COMPn}_Create_UserInit r_{Config_COMPn}_interrupt
	{Config_COMPn}.h	—

Table 2-16 Output File List (16/21)

Component / Folder Name	File Name	API Function Name
Programmable Gain Amplifier	{Config_PGA}.c	R_{Config_PGA}_Create R_{Config_PGADn}_Start R_{Config_PGADn}_Stop
	{Config_PGA}_user.c	R_{Config_PGA}_Create_UserInit
	{Config_PGA}.h	—
SPI (CSI) Communication	{Config_CSIp}.c	R_{Config_CSIp}_Create R_{Config_CSIp}_Start R_{Config_CSIp}_Stop R_{Config_CSIp}_Send R_{Config_CSIp}_Receive R_{Config_CSIp}_Send_Receive
	{Config_CSIp}_user.c	R_{Config_CSIp}_Create_UserInit r_{Config_CSIp}_interrupt r_{Config_CSIp}_callback_sendend r_{Config_CSIp}_callback_receiveend r_{Config_CSIp}_callback_error
	{Config_CSIp}.h	—
UART Communication (Serial array unit)	{Config_UARTq}.c	R_{Config_UARTq}_Create R_{Config_UARTq}_Start R_{Config_UARTq}_Stop R_{Config_UARTq}_Send R_{Config_UARTq}_Receive R_{Config_UARTq}_Loopback_Enable R_{Config_UARTq}_Loopback_Disble
	{Config_UARTq}_user.c	R_{Config_UARTq}_Create_UserInit r_{Config_UARTq}_interrupt_send r_{Config_UARTq}_interrupt_receive r_{Config_UARTq}_interrupt_error r_{Config_UARTq}_callback_sendend r_{Config_UARTq}_callback_receiveend r_{Config_UARTq}_callback_error r_{Config_UARTq}_callback_softwareoverrun
	{Config_UARTq}.h	—

Table 2-17 Output File List (17/21)

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	—
UART Communication (LIN/UART module)	{Config_RLIN3*n*}.c	R_{Config_RLIN3*n*}_Create R_{Config_RLIN3*n*}_Start R_{Config_RLIN3*n*}_Stop R_{Config_RLIN3*n*}_Send R_{Config_RLIN3*n*}_Receive
	{Config_RLIN3*n*}_user.c	R_{Config_RLIN3*n*}_Create_UserInit r_{Config_RLIN3*n*}_interrupt_send r_{Config_RLIN3*n*}_interrupt_receive r_{Config_RLIN3*n*}_interrupt_error r_{Config_RLIN3*n*}_callback_sendend r_{Config_RLIN3*n*}_callback_receiveend r_{Config_RLIN3*n*}_callback_error
	{Config_RLIN3*n*}.h	—

Table 2-18 Output File List (18/21)

Component / Folder Name	File Name	API Function Name
DALI Communication (Control devices)	{Config_DALI}.c	R_{Config_DALI}_Create R_{Config_DALI}_Start R_{Config_DALI}_Stop R_{Config_DALI}_SoftwareReset R_{Config_DALI}_EnableForceActiveState R_{Config_DALI}_DisableForceActiveState R_{Config_DALI}_GetStatus R_{Config_DALI}_Send R_{Config_DALI}_GetReceiveFrame
	{Config_DALI}_user.c	R_{Config_DALI}_Create_UserInit r_{Config_DALI}_interrupt_send r_{Config_DALI}_interrupt_receive r_{Config_DALI}_interrupt_error r_{Config_DALI}_interrupt_falling_edge_detection r_{Config_DALI}_interrupt_power_down_detection r_{Config_DALI}_interrupt_collision_detection r_{Config_DALI}_interrupt_stop_bit_detection r_{Config_DALI}_callback_sendend r_{Config_DALI}_callback_receiveend r_{Config_DALI}_callback_error
	{Config_DALI}.h	—
DALI Communication (Control gear)	{Config_DALI}.c	R_{Config_DALI}_Create R_{Config_DALI}_Start R_{Config_DALI}_Stop R_{Config_DALI}_SoftwareReset R_{Config_DALI}_EnableForceActiveState R_{Config_DALI}_DisableForceActiveState R_{Config_DALI}_GetStatus R_{Config_DALI}_Send R_{Config_DALI}_GetReceiveFrame
	{Config_DALI}_user.c	R_{Config_DALI}_Create_UserInit r_{Config_DALI}_interrupt_error r_{Config_DALI}_interrupt_falling_edge_detection r_{Config_DALI}_interrupt_power_down_detection r_{Config_DALI}_interrupt_stop_bit_detection r_{Config_DALI}_callback_sendend r_{Config_DALI}_callback_receiveend r_{Config_DALI}_callback_error
	{Config_DALI}.h	—

Table 2-19 Output File List (19/21)

Component / Folder Name	File Name	API Function Name
IIC 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	—
IIC Communication (Master mode) (Serial Interface IICA)	{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	—
IIC Communication (Slave mode) (Serial Interface IICA)	{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	—

Table 2-20 Output File List (20/21)

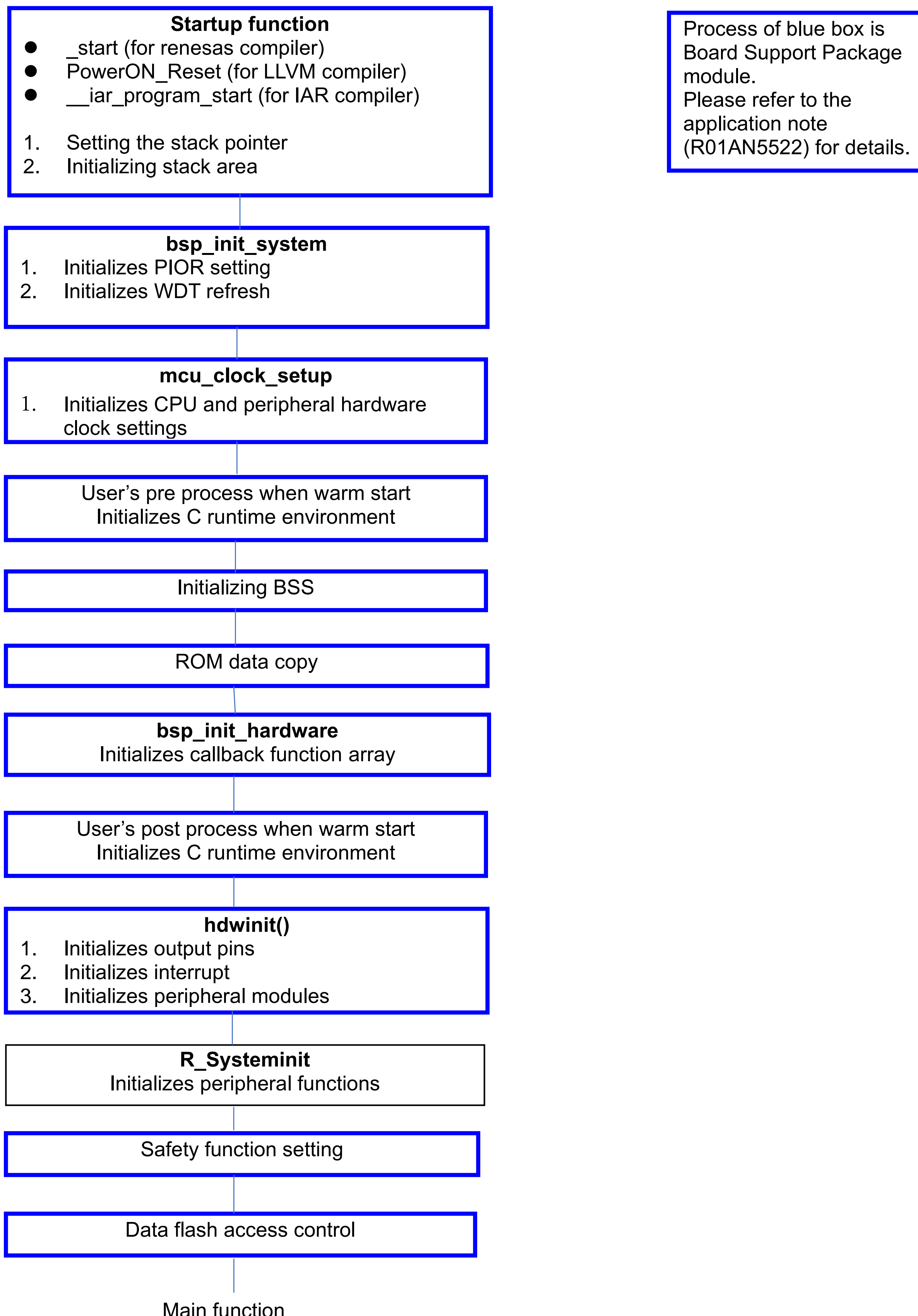
Component / Folder Name	File Name	API Function Name
Interrupt Controller	{Config_INTC}.c	R_{Config_INTC}_Create R_{Config_INTC}_INTP*n*_Start R_{Config_INTC}_INTP*n*_Stop
	{Config_INTC}_user.c	R_{Config_INTC}_Create_UserInit r_{Config_INTC}_intp*n*_interrupt
	{Config_INTC}.h	—
Voltage Detector	{Config_LVD*n*}.c	R_{Config_LVD*n*}_Create R_{Config_LVD1} _Start R_{Config_LVD1} _Stop
	{Config_LVD*n*}_user.c	R_{Config_LVD*n*}_Create_UserInit
	{Config_LVD*n*}.h	—
Snooze Mode Sequencer	{Config_SMS}.c	R_{Config_SMS}_Create R_{Config_SMS}_Start R_{Config_SMS}_Stop R_{Config_SMS}_GetStatus R_{Config_SMS}_GetReturn R_{Config_SMS}_TriggerWait_Disable R_{Config_SMS}_TriggerWait_Enable R_{Config_SMS}_Set_PowerOn R_{Config_SMS}_Set_PowerOff R_{Config_SMS}_Set_Reset R_{Config_SMS}_Release_Reset
	{Config_SMS}_user.c	R_{Config_SMS}_Create_UserInit r_{Config_SMS}_interrupt
	{Config_SMS}.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	—

Table 2-21 Output File List (21/21)

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	—
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	—
Logic and Event Link Controller	{Config_*xxx*}.c	R_{Config_*xxx*}_Create R_{Config_*xxx*}_Start R_{Config_*xxx*}_Stop
	{Config_*xxx*}_user.c	R_{Config_*xxx*}_Create_UserInit r_{Config_*xxx*}_interrupt
	{Config_*xxx*}.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	—

3. INITIALIZATION

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



Process of blue box is Board Support Package module.
Please refer to the application note (R01AN5522) for details.

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.

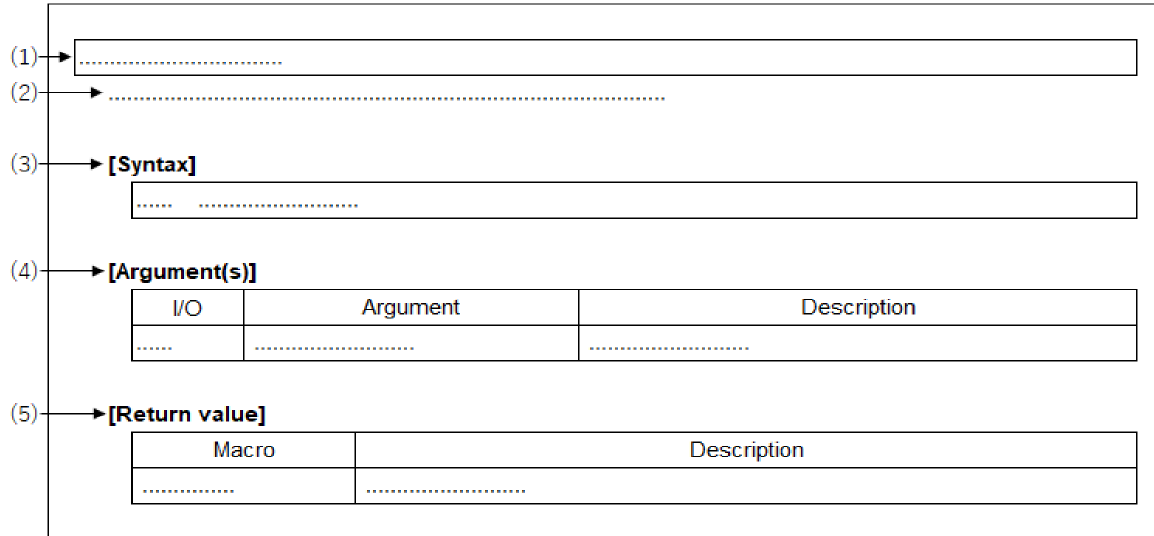
- Names of elements in enumeration specifiers “enum”
  These are in all-capital letters.

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: (1/3)

API Function Name	Peripheral Name	Description
main	-	Main function.
R_Systeminit	-	Executes initialization processing that is required before controlling various peripheral modules.
R_DTC_Set_PowerOn	Data Transfer Controller	Starts the clock supply for DTC.
R_DTC_Set_PowerOff		Stops the clock supply for DTC.
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*_Set_PowerOn		Starts the clock supply for TAU*m*.
R_TAU*m*_Set_PowerOff		Stops the clock supply for TAU*m*.
R_TAU*m*_Set_Reset		Sets TAU*m* module in reset state.
R_TAU*m*_Release_Reset		Releases TAU*m* module from reset state.
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 INTITL interrupt.
R_ITL_Stop Interrupt		Stops INTITL interrupt.
R_ITL_Set_PowerOn		Starts the clock supply for 32-bits IT.
R_ITL_Set_PowerOff		Stops the clock supply for 32-bits IT.
R_ITL_Set_Reset		Sets 32-bits IT module in reset state.
R_ITL_Release_Reset		Releases 32-bits IT module from reset state.
r_itl_interrupt		Executes processing in response to 32-bit interval timer interrupt (INTITL).
R_TRD_Create	Timer RD	Executes initialization processing that is required before controlling TRD (enables TRD input clock supply and initializes TRD module).
R_TRD_Set_PowerOn		Starts the clock supply for TRD.
R_TRD_Set_PowerOff		Stops the clock supply for TRD.
R_TRD_Set_Reset		Sets TRD module in reset state.
R_TRD_Release_Reset		Releases TRD module from reset state.
R_PWMOPA_Set_PowerOn		Starts the clock supply for PWMOPA.
R_PWMOPA_Set_PowerOff		Stops the clock supply for PWMOPA.
R_PWMOPA_Set_Reset		Sets PWMOPA module in reset state.
R_PWMOPA_Release_Reset		Releases PWMOPA module from reset state.
R_TRD_ForcedOutput_Enable		Enables TRD pulse output forced cutoff.
R_TRD_ForcedOutput_Disable		Disables TRD pulse output forced cutoff.

Table 4-2 API Functions: (2/3)

API Function Name	Peripheral Name	Description
R_TRJ_Set_PowerOn	Timer RJ	Starts the clock supply for TRJ.
R_TRJ_Set_PowerOff		Stops the clock supply for TRJ.
R_TRJ_Set_Reset		Sets TRJ module in reset state.
R_TRJ_Release_Reset		Releases TRJ module from reset state.
R_TRG_Set_PowerOn	Timer RG	Starts the clock supply for TRG.
R_TRG_Set_PowerOff		Stops the clock supply for TRG.
R_TRG_Set_Reset		Sets TRG module in reset state.
R_TRG_Release_Reset		Releases TRG module from reset state.
R_TRX_Set_PowerOn	Timer RX	Starts the clock supply for TRX.
R_TRX_Set_PowerOff		Stops the clock supply for TRX.
R_TRX_Set_Reset		Sets TRX module in reset state.
R_TRX_Release_Reset		Releases TRX module from reset state.
R_TKB_Create	Timer KB	Executes initialization processing that is required before controlling TKB (enables TKB input clock supply and initializes TKB module).
R_TKB_Set_PowerOn		Starts the clock supply for TKB.
R_TKB_Set_PowerOff		Stops the clock supply for TKB.
R_TKB_Set_Reset		Sets TKB module in reset state.
R_TKB_Release_Reset		Releases TKB module from reset state.
R_RTC_Set_PowerOn	Realtime Clock	Starts the clock supply for RTC.
R_RTC_Set_PowerOff		Stops the clock supply for RTC.
R_IT_Set_PowerOn	12-bit interval timer	Starts the clock supply for 12-bit interval timer.
R_IT_Set_PowerOff		Stops the clock supply for 12-bit interval timer.
R_ADC_Set_PowerOn	A/D Converter	Starts the clock supply for AD converter.
R_ADC_Set_PowerOff		Stops the clock supply for AD converter.
R_ADC_Set_Reset		Sets AD converter module in reset state.
R_ADC_Release_Reset		Releases AD converter module from reset state.
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_Set_PowerOn		Starts the clock supply for DA converter.
R_DAC_Set_PowerOff		Stops the clock supply for DA converter.
R_DAC_Set_Reset		Sets DA converter module in reset state.
R_DAC_Release_Reset		Releases DA converter module from reset state.
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_Set_PowerOn		Starts the clock supply for comparator.
R_COMP_Set_PowerOff		Stops the clock supply for comparator.
R_COMP_Set_Reset		Sets comparator module in reset state.
R_COMP_Release_Reset		Releases comparator module from reset state.

Table 4-3 API Functions: (3/3)

API Function Name	Peripheral Name	Description
R_PGACOMP_Create	Comparator and Programmable Gain Amplifier	Executes initialization processing that is required before controlling the COMP and PGA module (enables input clock supply and initializes COMP and PGA module).
R_PGACOMP_Set_PowerOn		Starts the clock supply for COMP and PGA.
R_PGACOMP_Set_PowerOff		Stops the clock supply for COMP and PGA.
R_PGACOMP_Set_Reset		Sets COMP and PGA module in reset state.
R_PGACOMP_Release_Reset		Releases COMP and PGA module from reset state.
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*_Set_PowerOn		Starts the clock supply for SAU*m*.
R_SAU*m*_Set_PowerOff		Stops the clock supply for SAU*m*.
R_SAU*m*_Set_Reset		Sets SAU*m* module in reset state.
R_SAU*m*_Release_Reset		Releases SAU*m* module from reset state.
R_SAU*m*_Set_SnoozeOn		Enables SAU*m* wakeup function.
R_SAU*m*_Set_SnoozeOff		Disables SAU*m* wakeup function.
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_Set_PowerOn		Starts the clock supply for UARTA0/UARTA1.
R_UARTA_Set_PowerOff		Stops the clock supply for UARTA0/UARTA1.
R_IICA*n*_Set_PowerOn	Serial Interface IICA	Starts the clock supply for IICA*n*.
R_IICA*n*_Set_PowerOff		Stops the clock supply for IICA*n*.
R_IICA*n*_Set_Reset		Sets IICA*n* module in reset state.
R_IICA*n*_Release_Reset		Releases IICA*n* module from reset state.
R_RLIN3n_Set_PowerOn	LIN/UART module	Starts the clock supply for RLIN3*n*.
R_RLIN3n_Set_PowerOff		Stops the clock supply for RLIN3*n*.
R_DALI_Set_PowerOn	Digital Addressable Lighting Interface	Starts the clock supply for DALI.
R_DALI_Set_PowerOff		Stops the clock supply for DALI.
R_DALI_Set_Reset		Sets DALI module in reset state.
R_DALI_Release_Reset		Releases DALI module from reset state.
R_LVD_Start_Interrupt	Voltage Detector	Starts INTLVI interrupt.
R_LVD_Stop_Interrupt		Stops INTLVI interrupt.
r_lvd_interrupt		Executes processing in response to INTLVI interrupt.
R_REMC_Set_PowerOn	Remote Control Signal Receiver	Starts the clock supply for REMC.
R_REMC_Set_PowerOff		Stops the clock supply for REMC.
R_REMC_Set_Reset		Sets REMC module in reset state.
R_REMC_Release_Reset		Releases REMC module from reset state.

main

This API function implements main function.

Remark When using SmartConfigurator stand-alone mode or using with CS+, please note to add the following code manually:

1) add “#include "r_smc_entry.h"”
2) add “EI()” in main()

[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_Set_PowerOn

This API function starts the clock supply for DTC.

[Syntax]

```
void    R_DTC_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DTC_Set_PowerOff

This API function stops the clock supply for DTC.

[Syntax]

```
void    R_DTC_Set_PowerOff(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*_Set_PowerOn

This API function starts the clock supply for TAU*m*.

[Syntax]

```
void    R_TAUm_ Set_PowerOn(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_TAU*m*_Set_PowerOff

This API function stops the clock supply for TAU*m*.

[Syntax]

```
void    R_TAUm_ Set_PowerOff(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_TAU*m*_Set_Reset

This API function sets TAU*m* module in reset state.

[Syntax]

```
void    R_TAUm_ Set_Reset(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_TAU*m*_Release_Reset

This API function releases TAU*m* module from reset state.

[Syntax]

```
void    R_TAUm_Release_Reset(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 INTITL 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 INTITL interrupt.

[Syntax]

```
void    R_ ITL_Stop_Interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Set_PowerOn

This API function starts the clock supply for 32-bits IT.

[Syntax]

```
void    R_ ITL_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Set_PowerOff

This API function stops the clock supply for 32-bits IT.

[Syntax]

```
void    R_ ITL_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Set_Reset

This API function sets 32-bits IT module in reset state.

[Syntax]

```
void     R_ ITL_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Release_Reset

This API function releases 32-bits IT module from reset state.

[Syntax]

```
void    R_ ITL_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

r_itl_interrupt

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

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

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_itl_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void     r_itl_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_Create

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

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

[Syntax]

```
void R_TRD_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_Set_PowerOn

This API function starts the clock supply for TRD.

[Syntax]

```
void    R_TRD_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_Set_PowerOff

This API function stops the clock supply for TRD.

[Syntax]

```
void    R_TRD_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_Set_Reset

This API function sets TRD module in reset state.

[Syntax]

```
void    R_TRD_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_Release_Reset

This API function releases TRD module from reset state.

[Syntax]

```
void    R_TRD_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PWMOPA_Set_PowerOn

This API function starts the clock supply for PWMOPA.

[Syntax]

```
void      R_PWMOPA_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PWMOPA_Set_PowerOff

This API function stops the clock supply for PWMOPA.

[Syntax]

```
void    R_PWMOPA_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PWMOPA_Set_Reset

This API function sets PWMOPA module in reset state.

[Syntax]

```
void    R_PWMOPA_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PWMOPA_Release_Reset

This API function releases PWMOPA module from reset state.

[Syntax]

```
void    R_PWMOPA_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_ForcedOutput_Enable

This API function enables TRD pulse output forced cutoff. It can't be called during timer counter is running. Please call it before R_{Config_TRD*n*}_Start() or R_{Config_TRD0_TRD1}_Start().

[Syntax]

```
void    R_TRD_ForcedOutput_Enable(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRD_ForcedOutput_Disable

This API function disables TRD pulse output forced cutoff. It can't be called during timer counter is running. Please call it after R_{Config_TRD*n*}_Stop() or R_{Config_TRD0_TRD1}_Stop().

[Syntax]

```
void	R_TRD_ForcedOutput_Disable(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRJ_Set_PowerOn

This API function starts the clock supply for TRJ.

[Syntax]

```
void       R_TRJ_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRJ_Set_PowerOff

This API function stops the clock supply for TRJ.

[Syntax]

```
void    R_TRJ_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRJ_Set_Reset

This API function sets TRJ module in reset state.

[Syntax]

```
void    R_TRJ_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRJ_Release_Reset

This API function releases TRJ module from reset state.

[Syntax]

```
void    R_TRJ_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRG_Set_PowerOn

This API function starts the clock supply for TRG.

[Syntax]

```
void    R_TRG_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRG_Set_PowerOff

This API function stops the clock supply for TRG.

[Syntax]

```
void      R_TRG_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRG_Set_Reset

This API function sets TRG module in reset state.

[Syntax]

```
void    R_TRG_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRG_Release_Reset

This API function releases TRG module from reset state.

[Syntax]

```
void        R_TRG_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRX_Set_PowerOn

This API function starts the clock supply for TRX.

[Syntax]

```
void    R_TRX_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRX_Set_PowerOff

This API function stops the clock supply for TRX.

[Syntax]

```
void    R_TRX_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRX_Set_Reset

This API function sets TRX module in reset state.

[Syntax]

```
void        R_TRX_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TRX_Release_Reset

This API function releases TRX module from reset state.

[Syntax]

```
void    R_TRX_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TKB_Create

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

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

[Syntax]

```
void    R_TKB_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TKB_Set_PowerOn

This API function starts the clock supply for TKB.

[Syntax]

```
void    R_TKB_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TKB_Set_PowerOff

This API function stops the clock supply for TKB.

[Syntax]

```
void    R_TKB_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TKB_Set_Reset

This API function sets TKB module in reset state.

[Syntax]

```
void        R_TKB_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TKB_Release_Reset

This API function releases TKB module from reset state.

[Syntax]

```
void    R_TKB_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_RTC_Set_PowerOn

This API function starts the clock supply for RTC.

[Syntax]

```
void      R_RTC_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_RTC_Set_PowerOff

This API function stops the clock supply for RTC.

[Syntax]

```
void    R_RTC_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_IT_Set_PowerOn

This API function starts the clock supply for 12-bit interval timer.

[Syntax]

```
void    R_IT_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_IT_Set_PowerOff

This API function stops the clock supply for 12-bit interval timer.

[Syntax]

```
void    R_IT_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ADC_Set_PowerOn

This API function starts the clock supply for AD converter.

[Syntax]

```
void      R_ADC_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ADC_Set_PowerOff

This API function stops the clock supply for AD converter.

[Syntax]

```
void	R_ADC_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ADC_Set_Reset

This API function sets AD converter module in reset state.

[Syntax]

```
void     R_ADC_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ADC_Release_Reset

This API function releases AD converter module from reset state.

[Syntax]

```
void        R_ADC_Release_Reset(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_Set_PowerOn

This API function starts the clock supply for DA converter.

[Syntax]

```
void	R_DAC_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DAC_Set_PowerOff

This API function stops the clock supply for DA converter.

[Syntax]

```
void      R_DAC_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DAC_Set_Reset

This API function sets DA converter module in reset state.

[Syntax]

```
void        R_DAC_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DAC_Release_Reset

This API function releases DA converter module from reset state.

[Syntax]

```
void    R_DAC_Release_Reset(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_Set_PowerOn

This API function starts the clock supply for comparator.

[Syntax]

```
void       R_COMP_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_COMP_Set_PowerOff

This API function stops the clock supply for comparator.

[Syntax]

```
void    R_COMP_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_COMP_Set_Reset

This API function sets comparator module in reset state.

[Syntax]

```
void    R_COMP_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_COMP_Release_Reset

This API function releases comparator module from reset state.

[Syntax]

```
void    R_COMP_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PGACOMP_Create

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

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

[Syntax]

```
void    R_PGACOMP_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PGACOMP_Set_PowerOn

This API function starts the clock supply for comparator module and PGA module.

[Syntax]

```
void    R_PGACOMP_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PGACOMP_Set_PowerOff

This API function stops the clock supply for comparator module and PGA module.

[Syntax]

```
void    R_PGACOMP_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PGACOMP_Set_Reset

This API function sets comparator module and PGA module in reset state.

[Syntax]

```
void      R_PGACOMP_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_PGACOMP_Release_Reset

This API function releases comparator module and PGA module from reset state.

[Syntax]

```
void    R_PGACOMP_Release_Reset(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*_Set_PowerOn

This API function starts the clock supply for SAU*m*.

[Syntax]

```
void	R_SAUm_Set_PowerOn(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Set_PowerOff

This API function stops the clock supply for SAU*m*.

[Syntax]

```
void    R_SAUm_Set_PowerOff(void);
```

Remark   *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Set_Reset

This API function sets SAU*m* module in reset state.

[Syntax]

```
void    R_SAUm_Set_Reset(void);
```

Remark   *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Release_Reset

This API function releases SAU*m* module from reset state.

[Syntax]

```
void    R_SAUm_Release_Reset(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_Set_PowerOn

This API function starts the clock supply for UARTA0/UARTA1.

[Syntax]

```
void      R_UARTA_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_UARTA_Set_PowerOff

This API function stops the clock supply for UARTA0/UARTA1.

[Syntax]

```
void    R_UARTA_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_IICA*n*_Set_PowerOn

This API function starts the clock supply for IICA*n*.

[Syntax]

```
void    R_IICAn_Set_PowerOn(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_IICA*n*_Set_PowerOff

This API function stops the clock supply for IICA*n*.

[Syntax]

```
void    R_IICAn_Set_PowerOff(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_IICA*n*_Set_Reset

This API function sets IICA*n* module in reset state.

[Syntax]

```
void    R_IICAn_Set_Reset(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_IICA*n*_Release_Reset

This API function releases IICA*n* module from reset state.

[Syntax]

```
void    R_IICAn_Release_Reset(void);
```

Remark   *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_RLIN3*n*_Set_PowerOn

This API function starts the clock supply for RLIN3*n*.

[Syntax]

```
void    R_RLIN3n_Set_PowerOn(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_RLIN3*n*_Set_PowerOff

This API function stops the clock supply for RLIN3*n*.

[Syntax]

```
void    R_RLIN3n_Set_PowerOff(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_DALI_Set_PowerOn

This API function starts the clock supply for DALI.

[Syntax]

```
void      R_DALI_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DALI_Set_PowerOff

This API function stops the clock supply for DALI.

[Syntax]

```
void        R_DALI_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DALI_Set_Reset

This API function sets DALI module in reset state.

[Syntax]

```
void    R_DALI_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DALI_Release_Reset

This API function releases DALI module from reset state.

[Syntax]

```
void        R_DALI_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_LVD_Start_Interrupt

This API function starts INTLVI interrupt.

[Syntax]

```
void    R_LVD_Start_Interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_LVD_Stop_Interrupt

This API function stops INTLVI interrupt.

[Syntax]

```
void    R_LVD_Stop_Interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_lvd_interrupt

This API function executes processing in response to INTLVI interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_lvd_interrupt(void);
```

For LLVM toolchain:

```
void      r_lvd_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_lvd_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_REMC_Set_PowerOn

This API function starts the clock supply for REMC.

[Syntax]

```
void    R_REMC_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_REMC_Set_PowerOff

This API function stops the clock supply for REMC.

[Syntax]

```
void    R_REMC_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_REMC_Set_Reset

This API function sets REMC module in reset state.

[Syntax]

```
void    R_REMC_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_REMC_Release_Reset

This API function releases REMC module from reset state.

[Syntax]

```
void    R_REMC_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using R_*Xxxx*_Set_PowerOn(), R_ *Xxxx*_Set_PowerOff(), R_ *Xxxx*_Set_Reset(), R_ *Xxxx*_Release_Reset(), 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(ITL) as an example)

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
        EI();

        // To enable INTITL interrupt which is shared among ITL each channels
        R_ITL_Start_Initerrupt();

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

        // To Disable INTITL interrupt which is shared among ITL each channels
        R_ITL_Stop_Initerrupt();

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

        // To use ITL again, supplies input clock.
        R_Config_ ITL_Set_PowerOn();

        // To set ITL in the reset state
        R_ ITL_Set_Reset();

        // To use ITL again, release ITL from the reset state
        R_ ITL_ Release_Reset();

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

}
```

4.2.2 Port

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

Table 4-4 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}_ReadPmnValues		Specifies the value in the output latch for a port is read when the pin is in output mode.
R_{Config_PORT}_ReadDigitalOutputLevel		Specifies the output level on a port pin is read when the pin is in output mode.
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}_ReadPmnValues

This API function specifies the value in the output latch for a port is read when the pin is in output mode.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_PORT}_ReadDigitalOutputLevel

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}_ReadDigitalOutputLevel(void);
```

[Argument(s)]

None.

[Return value]

None.

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"

void main(void);

void main(void)
{
        EI();
        R_Config_PORT_ ReadDigitalOutputLevel ();
}
```

4.2.3 Delay Counter

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

Table 4-5 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 (INTTM*mn*).

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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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

void main(void);

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

        R_Config_TAU0_1_Lower8bits_Start();
        R_Config_TAU0_1_Set_SoftwareTriggerOn();
        while (ch0_run_count < 20);
        R_Config_TAU0_1_Lower8bits_Stop();
}
```

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 */
}
```

4.2.4 Divider Function

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

Table 4-6 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 (INTTM*mn*).

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* 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*}_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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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

void main(void);

void main(void)
{
        EI();
        R_Config_TAU0_0_Start();
        while( ch0_run_count < 20);
        R_Config_TAU0_0_Stop();
}
```

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 */
}
```

4.2.5 External Event Counter (Timer Array Unit)

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

Table 4-7 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 (INTTM*mn*).

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* 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

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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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

void main(void);

void main(void)
{
        EI();
        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();
}
```

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 */
}
```

4.2.6 External Event Counter (Timer RJ)

Below is a list of API functions output by the Smart Configurator for external event counter (input to the TRJIO*n* pin) use.

Table 4-8 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRJ*n*}_Create	Timer RJ*n*	Executes initialization processing that is required before controlling the Timer RJ*n* module in external event counter mode.
R_{Config_TRJ*n*}_Start		Starts the TRJ*n* counter.
R_{Config_TRJ*n*}_Stop		Stops the TRJ*n* counter.
R_{Config_TRJ*n*}_Create_UserInit		Executes user-specific initialization processing for the TRJ*n*.
r_{Config_TRJ*n*}_ interrupt		Executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

R_{Config_TRJ*n*}_Create

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

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Start

This API function starts the TRJ*n* counter.

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Stop

This API function stops the TRJn counter.

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRJ*n*}_interrupt

This API function executes processing in response to to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows..

Remark This API function is called as the interrupt handler for TRJ*n* interrupts (INTTRJ*n*), which occur when the counter underflows.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TRJ0 counting as 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

void main(void);

void main(void)
{
        EI();
        R_Config_TRJ0_Start();
        while (ch0_run_count < 20);
        R_Config_TRJ0_Stop();
}
```

Config_TRJ0_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_TRJ0_interrupt (void)
{
        /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
        ch0_run_count ++;
        /* End user code. Do not edit comment generated here */
}
```

4.2.7 Input Pulse High-/Low-Level Width Measurement (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 4-9 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 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 (INTTM*mn*).

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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

void main(void)
{
        EI();
        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);

}
```

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 */
        /* Set the flag */
        tau_interrupt_flag = 1U;
        /* End user code. Do not edit comment generated here */
}
```

4.2.8 Input Pulse High-/Low-Level Width Measurement (Timer RJ)

Below is a list of API functions output by the Smart Configurator for input pulse high-/low-level width of an external signal (input to the TRJIO*n* pin) measurement use.

Table 4-10 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRJ*n*}_Create	Timer RJ*n*	Executes initialization processing that is required before controlling the TRJn module in pulse high-/low-level width measurement mode.
R_{Config_TRJ*n*}_Start		Starts the TRJ*n* counter.
R_{Config_TRJ*n*}_Stop		Stops the TRJ*n* counter.
R_{Config_TRJ*n*}_Get_PulseWidth		Measures TRJ*n* input pulse width.
R_{Config_TRJ*n*}_Create_UserInit		Executes user-specific initialization processing for the TRJ*n*.
r_{Config_TRJ*n*}_ interrupt		Executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

R_{Config_TRJ*n*}_Create

This API function executes initialization processing that is required before controlling the TRJ*n* module in input pulse high-/low-level width of an external signal (input to the TRJIO*n* pin) measurement mode.

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

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Start

This API function starts the TRJ*n* counter.

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Stop

This API function stops the TRJ*n* counter.

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Get_PulseWidth

This API function measures TRJ*n* input pulse width.

[Syntax]

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

Remark *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_TRJ*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRJ*n*}_interrupt

This API function executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

Remark This API function is called as the interrupt handler for TRJn counter underflows interrupts (INTTRJ*n*), which occur when the measurement of the active width of the external input (TRJIO*n*) is completed in pulse width measurement mode.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TRJ0 input pulse width from TRJIO0 pin:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
	EI();
	interrupt_flag = 0;
	R_Config_TRJ0_Start();
	while(interrupt_flag == 0);
	R_Config_TRJ0_Stop();
	R_Config_TRJ0_Get_PulseWidth(&width);

}
```

Config_TRJ0_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 */

static void __near r_Config_TRJ0_interrupt (void)
{
	...
	/* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
	/* Set the flag */
	interrupt_flag = 1U;
	/* End user code. Do not edit comment generated here */
}
```

4.2.9 PWM Output (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for PWM output 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 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 (INTTM*mn*).
r_{Config_TAU*m*_*n*}_channel*p*_interrupt		Executes processing in response to timer channel*p* count end interrupt (INTTM*mp*)

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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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 channelmn count end interrupt (INTTMmp).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

void main(void)
{
        EI();
        tau_pwm_count = 0;
        R_Config_TAU0_0_Start();
        while (tau_interrupt_flag < 10);
        R_Config_TAU0_0_Stop();
}
```

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 */
}
```

4.2.10 PWM Output (Timer RDn using PWM mode/ Extended PWM mode)

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

Table 4-12 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRD*n*}_Create	Timer RD	Executes initialization processing that is required before controlling the TRD*n* module in PWM mode/ Extended PWM mode.
R_{Config_TRD*n*}_Start		Starts the TRD*n* counter.
R_{Config_TRD*n*}_Stop		Stops the TRD*n* counter.
R_{Config_TRD*n*}_Set_TRD*n*_ReloadTrigger		Generates TRD*n* buffer registers reload trigger in Extended PWM mode.
R_{Config_TRD*n*}_Create_UserInit		Executes user-specific initialization processing for the TRD*n*.
r_{Config_TRD*n*}_trd*n*_interrupt		Executes processing in response to timer RDn count compare match interrupt (INTTRDn).

R_{Config_TRD*n*}_Create

This API function executes initialization processing that is required before controlling the TRD*n* module in PWM mode/ Extended PWM mode.

Remark This API function is called from R_TRD_Create.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Start

This API function starts the TRD*n* counter.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Stop

This API function stops the TRD*n* counter.

[Syntax]

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

Remark   *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Set_TRDn_ReloadTrigger

This API function generates TRDn buffer registers reload trigger in Extended PWM mode.

[Syntax]

```
MD_STATUS R_{Config_TRDn}_Set_TRDn_ReloadTrigger (st_extpwm_buffer_registers_t * buffer);
```

Remark *n* is 0, 1.

[Argument(s)]

I/O	Argument(s)	Description
I	st_extpwm_buffer_registers_t * buffer;	buffer registers value

Remark Below is shown the structure st_extpwm_buffer_registers_t.

```
typedef struct {
    uint16_t trdgrcn;
    uint16_t trdgrdn;
    uint16_t trdcmpdn;
} st_extpwm_buffer_registers_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Waiting for reload trigger status

R_{Config_TRD*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRD*n*}_trdn_interrupt

This API function executes processing in response to timer RD*n* count compare match interrupt (INTTRD*n*).

Remark This API function is called as the interrupt handler for count compare match interrupt (INTTRD*n*), which occur when the content of the TRD*n* register matches content of the TRDGR*hn* (h = A, B, C, or D) register or TRD*n* register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRDn}_trdn_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TRDn}_trdn_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TRDn}_trdn_interrupt(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TRD0 to output three PWM waveforms pulses:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
	EI();
	R_Config_TRD0_Start();
	while (trd_pwm_count < 10);
	R_Config_TRD0_Stop();
}
```

Config_TRD0_user.c

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

static void __near r_Config_TRD0_trd0_interrupt (void)
{
	/* Start user code for r_Config_TRD0_trd0_interrupt. Do not edit comment generated here */
	trd_pwm_count ++;
	/* End user code. Do not edit comment generated here */
}
```

4.2.11 PWM Output (Timer RD0 and RD1 using PWM mode/ PWM3 mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode)

Below is a list of API functions output by the Smart Configurator for outputing two PWM waveforms in PWM3 mode/ Timer KB3 PWM Output Gate mode or four PWM waveforms use in Timer KB3 PWM Output Gate mode or six PWM waveforms use in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.

Table 4-13 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRD0_TRD1}_Create	Timer RD	Executes initialization processing that is required before controlling the TRD0 module in PWM3 mode or the TRD*n* module in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.
R_{Config_TRD0_TRD1}_Start		Starts the TRD0 counter in PWM3 mode or the TRD*n* counter in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.
R_{Config_TRD0_TRD1}_Stop		Stops the TRD0 counter in PWM3 mode or the TRD*n* counter in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.
R_{Config_TRD0_TRD1}_Set_TRD*n*_ReloadTrigger		Generates TRD*n* buffer registers reload trigger in Extended PWM mode.
R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger		Generates TRD0 buffer registers reload trigger in Timer KB3 PWM Output Gate mode.
R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger		Generates TRD1 buffer registers reload trigger in Timer KB3 PWM Output Gate mode.
R_{Config_TRD0_TRD1}_Create_UserInit		Executes user-specific initialization processing for the TRD0_TRD1.
r_{Config_TRD0_TRD1}_trd*n*_interrupt		Executes processing in response to timer RDn count compare match interrupt (INTTRDn).

R_{Config_TRD0_TRD1}_Create

This API function executes initialization processing that is required before controlling the TRD0 module in PWM3 mode or the TRD*n* module in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.

Remark This API function is called from R_TRD_Create.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD0_TRD1}_Start

This API function starts the TRD0 counter in PWM3 mode or the TRD*n* counter in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD0_TRD1}_Stop

This API function stops the TRD0 counter in PWM3 mode or the TRD*n* counter in PWM mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD0_TRD1}_Set_TRDn_ReloadTrigger

This API function generates TRD*n* buffer registers reload trigger in Extended PWM mode.

[Syntax]

```
MD_STATUS R_{Config_TRD0_TRD1}_Set_TRDn_ReloadTrigger (st_extpwm_buffer_registers_t * buffer);
```

Remark *n* is 0, 1.

[Argument(s)]

I/O	Argument(s)	Description
I	st_extpwm_buffer_registers_t * buffer;	buffer registers value

Remark Below is shown the structure st_extpwm_buffer_registers_t.

```
typedef struct {
        uint16_t trdgrcn;
        uint16_t trdgrdn;
        uint16_t trdcmpdn;
} st_extpwm_buffer_registers_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Waiting for reload trigger status

R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger

This API function generates TRD0 buffer registers reload trigger in Timer KB3 PWM Output Gate mode.

[Syntax]

```
MD_STATUS R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger
(st_kb3pwm_ch0_buffer_registers_t * buffer);
```

[Argument(s)]

I/O	Argument(s)	Description
I	st_kb3pwm_ch0_buffer_registers_t * buffer;	buffer registers value

Remark Below is shown the structure st_kb3pwm_ch1_buffer_registers_t.

```
typedef struct {
        uint16_t trdgra0;
        uint16_t trdgrb0;
        uint16_t trdcmpb0;
} st_kb3pwm_ch0_buffer_registers_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Waiting for reload trigger status

R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger

This API function generates TRD1 buffer registers reload trigger in Timer KB3 PWM Output Gate mode.

[Syntax]

```
MD_STATUS R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger
(st_kb3pwm_ch1_buffer_registers_t * buffer);
```

[Argument(s)]

I/O	Argument(s)	Description
I	st_kb3pwm_ch1_buffer_registers_t * buffer;	buffer registers value

Remark Below is shown the structure st_kb3pwm_ch1_buffer_registers_t.

```
typedef struct {
        uint16_t trdgra1;
        uint16_t trdgrb1;
        uint16_t trdcmpa1;
        uint16_t trdcmpb1;
        uint16_t trdcmpe1;
 } st_kb3pwm_ch1_buffer_registers_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Waiting for reload trigger status

R_{Config_TRD0_TRD1}_Create_UserInit

This API function executes user-specific initialization processing for the TRD0_TRD1.

Remark This API functions is called from R_{Config_TRD0_TRD1}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRD0_TRD1}_trdn_interrupt

This API function executes processing in response to timer TRD*n* count compare match interrupt (INTTRD*n*).

Remark This API function is called as the interrupt handler for count compare match interrupt (INTTRD*n*), which occur when the content of the TRD*n* register matches content of the TRDGR*jn* (j = A, B, C, or D) register in PWM mode/ PWM3 mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode or TRD0 register overflow in PWM mode/ PWM3 mode/ Extended PWM mode.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRD0_TRD1}_trdn_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TRD0_TRD1}_trdn_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TRD0_TRD1}_trdn_interrupt(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TRD0_TRD1 to output six PWM waveforms pulses:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t trd_kb3_pwm_count;

void main(void);

void main(void)
{
    EI();
    R_Config_TRD0_TRD1_Start();
    while (trd_kb3_pwm_count < 20);
    R_Config_TRD0_TRD1_Stop();
}
```

Config_TRD0_TRD1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t trd_kb3_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRD0_TRD1_trd0_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd0_interrupt. Do not edit comment generated here */
    trd_kb3_pwm_count ++;
    /* End user code. Do not edit comment generated here */
}

static void __near r_Config_TRD0_TRD1_trd1_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd1_interrupt. Do not edit comment generated here */
    trd_kb3_pwm_count ++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.12 PWM Output (Timer RG using PWM mode/ PWM2 mode)

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

Table 4-14 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRG}_Create	Timer RG	Executes initialization processing that is required before controlling the TRG module in PWM mode/ PWM2 mode.
R_{Config_TRG}_Start		Starts the TRG counter.
R_{Config_TRG}_Stop		Stops the TRG counter.
R_{Config_TRG}_Create_UserInit		Executes user-specific initialization processing for the TRG.
r_{Config_TRG}_interrupt		Executes processing in response to timer RG count compare match interrupt (INTTRG).

R_{Config_TRG}_Create

This API function executes initialization processing that is required before controlling the TRG module in PWM mode/ PWM2 mode.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Start

This API function starts the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Stop

This API function stops the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRG}_interrupt

This API function executes processing in response to timer RG count compare match interrupt (INTTRG).

Remark This API function is called as the interrupt handler for count compare match interrupt (INTTRG), which occur when the content of the TRG register matches content of the TRGGR*h* (h = A, B, C, or D) register or TRG register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRG}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TRG to output PWM waveforms pulses:

(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
    EI();
    R_Config_TRG_Start();
    while (trg_pwm_count < 10);
    R_Config_TRG_Stop();
}
```

Config_TRG_user.c

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

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

4.2.13 PWM Output (Timer KB using standalone mode (period controlled by TKBCRn0 register)/standalone mode (period controlled by external trigger input)/interleave PFC output mode)

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

Table 4-15 API Functions: (1/2)

API Function Name	Peripheral Name	Description
R_{Config_TKB*n*}_Create	Timer KB	Executes initialization processing that is required before controlling the TKB*n* module in TKB using standalone mode (period controlled by TKBCRn0 register)/standalone mode (period controlled by external trigger input)/interleave PFC output mode.
R_{Config_TKB*n*}_Start		Starts the TKB*n* counter.
R_{Config_TKB*n*}_Stop		Stops the TKB*n* counter.
R_{Config_TKB*n*}_Set_BatchOverwriteRequestOn		Sets TKB*n* batch overwrite request function.
R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Start		Starts TKB*n* TKBO*n*0 forced output stop function 1.
R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Stop		Stops TKB*n* TKBO*n*0 forced output stop function 1.
R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Start		Starts TKB*n* TKBO*n*1 forced output stop function 1.
R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Stop		Stops TKB*n* TKBO*n*1 forced output stop function 1.
R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Start		Starts TKB*n* TKBO*n*0 smooth start function.
R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop		Stops TKB*n* TKBO*n*0 smooth start function.
R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Start		Starts TKB*n* TKBO*n*1 smooth start function.
R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Stop		Stops TKB*n* TKBO*n*1 smooth start function.

Table 4-16 API Functions: (2/2)

API Function Name	Peripheral Name	Description
R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Start	Timer KB	Starts TKB*n* TKBO*n*0 dithering function.
R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Stop		Stops TKB*n* TKBO*n*0 dithering function.
R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Start		Starts TKB*n* TKBO*n*1 dithering function.
R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Stop		Stops TKB*n* TKBO*n*1 dithering function.
R_{Config_TKB*n*}_Create_UserInit		Executes user-specific initialization processing for the TKB*n*.
r_{Config_TKB*n*}_terminated0_interrupt		Executes processing in response to timer KB*n* TKBO*n*0 forced output stop termination interrupt (INTTMKBSTP*n*0).
r_{Config_TKB*n*}_terminated1_interrupt		Executes processing in response to timer KB*n* TKBO*n*1 forced output stop termination interrupt (INTTMKBSTP*n*1).
r_{Config_TKB*n*}_activated0_interrupt		Executes processing in response to timer KB*n* TKBO*n*0 forced output stop activation interrupt (INTTMKBSTR*n*0).
r_{Config_TKB*n*}_activated1_interrupt		Executes processing in response to timer KB*n* TKBO*n*1 forced output stop activation interrupt (INTTMKBSTR*n*1).
r_{Config_TKB*n*}_end_count_interrupt		Executes processing in response to timer KB*n* count compare match interrupt (INTTMKB*n*).

R_{Config_TKB*n*}_Create

This API function executes initialization processing that is required before controlling the TKB*n* module in standalone mode (period controlled by TKBCR*n*0 register)/standalone mode (period controlled by external trigger input)/interleave PFC output mode.

Remark This API function is called from R_TKB_Create.

[Syntax]

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

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_Start

This API function starts the TKB*n* counter.

[Syntax]

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

Remark   *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_Stop

This API function stops the TKB*n* counter.

[Syntax]

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

Remark   *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_Set_BatchOverwriteRequestOn

This API function sets TKB*n* batch overwrite request function.

[Syntax]

```
void    R_{Config_TKBn}_Set_BatchOverwriteRequestOn(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Start

This API function starts TKB*n* TKBO*n*0 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Stop

This API function stops TKB*n* TKBO*n*0 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Start

This API function starts TKB*n* TKBO*n*1 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn1_Forced_Output_Stop_Function1_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Stop

This API function stops TKB*n* TKBO*n*1 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn1_Forced_Output_Stop_Function1_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Start

This API function starts TKB*n* TKBO*n*0 smooth start function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn0_SmoothStartFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop

This API function stops TKB*n* TKBO*n*0 smooth start function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn0_SmoothStartFunction_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Start

This API function starts TKB*n* TKBO*n*1 smooth start function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn1_SmoothStartFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Stop

This API function stops TKB*n* TKBO*n*1 smooth start function.

[Syntax]

```
void	R_{Config_TKBn}_TKBOn1_SmoothStartFunction_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Start

This API function starts TKB*n* TKBO*n*0 dithering function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn0_DitheringFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Stop

This API function stops TKB*n* TKBO*n*0 dithering function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn0_DitheringFunction_Stop(void);
```

Remark   *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Start

This API function starts TKB*n* TKBO*n*1 dithering function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn1_DitheringFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Stop

This API function stops TKB*n* TKBO*n*1 dithering function.

[Syntax]

```
void    R_{Config_TKBn}_TKBOn1_DitheringFunction_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB*n*}_terminated0_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*0 forced output stop termination interrupt (INTTMKBSTP*n*0).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TKBn}_terminated0_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TKBn}_terminated0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TKBn}_terminated0_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB*n*}_terminated1_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*1 forced output stop termination interrupt (INTTMKBSTP*n*1).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TKBn}_terminated1_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TKBn}_terminated1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TKBn}_terminated1_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB*n*}_activated0_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*0 forced output stop activation interrupt (INTTMKBSTR*n*0).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TKBn}_activated0_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TKBn}_activated0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TKBn}_activated0_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB*n*}_activated1_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*1 forced output stop activation interrupt (INTTMKBSTR*n*1).

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_{Config_TKBn}_activated1_interrupt(void);
```

For LLVM toolchain:

```
void     r_{Config_TKBn}_activated1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void     r_{Config_TKBn}_activated1_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB*n*}_end_count_interrupt

This API function executes processing in response to timer KB*n* count compare match interrupt (INTTMKBn).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKBn}_end_count_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKBn}_end_count_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKBn}_end_count_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TKB1 to output PWM waveforms pulses:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
	EI();
	R_Config_TKB1_Start();
	while (tkb_pwm_count < 10);
	R_Config_TKB1_Stop();
}
```

Config_TKB1_user.c

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

static void __near r_Config_TKB1_end_count_interrupt (void)
{
	/* Start user code for r_Config_TKB1_end_count_interrupt. Do not edit comment generated here */
	tkb_pwm_count ++;
	/* End user code. Do not edit comment generated here */
}
```

4.2.14 PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (1 slave)

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

Table 4-17 API Functions: (1/2)

API Function Name	Peripheral Name	Description
R_{Config_TKB0_TKB*n*}_Create	Timer KB	Executes initialization processing that is required before controlling the TKB0 and TKB*n* modules in TKB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master).
R_{Config_TKB0_TKB*n*}_Start		Starts the TKB0 and TKB*n* counters.
R_{Config_TKB0_TKB*n*}_Stop		Stops the TKB0 and TKB*n* counters.
R_{Config_TKB0_TKB*n*}_TKB*m*_Set_BatchOverwriteRequestOn		Sets TKB*m* batch overwrite request function.
R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Start		Starts TKB*m* TKBO*m*0 forced output stop function 1.
R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Stop		Stops TKB*m* TKBO*m*0 forced output stop function 1.
R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Start		Starts TKB*m* TKBO*m*1 forced output stop function 1.
R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Stop		Stops TKB*m* TKBO*m*1 forced output stop function 1.
R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Start		Starts TKB*m* TKBO*m*0 smooth start function.
R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Stop		Stops TKB*m* TKBO*m*0 smooth start function.
R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Start		Starts TKB*m* TKBO*m*1 smooth start function.
R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Stop		Stops TKB*m* TKBO*m*1 smooth start function.

Table 4-18 API Functions: (2/2)

API Function Name	Peripheral Name	Description
R_{Config_TKB0_TKB*n*}_TKBO*m*0_DitheringFunction_Start	Timer KB	Starts TKB*m* TKBO*m*0 dithering function.
R_{Config_TKB0_TKB*n*}_TKBO*m*0_DitheringFunction_Stop		Stops TKB*m* TKBO*m*0 dithering function.
R_{Config_TKB0_TKB*n*}_TKBO*m*1_DitheringFunction_Start		Starts TKB*m* TKBO*m*1 dithering function.
R_{Config_TKB0_TKB*n*}_TKBO*m*1_DitheringFunction_Stop		Stops TKB*m* TKBO*m*1 dithering function.
R_{Config_TKB0_TKB*n*}_Create_UserInit		Executes user-specific initialization processing for the TKB0 and TKB*n*.
r_{Config_TKB0_TKB*n*}_tkb*m*_terminated0_interrupt		Executes processing in response to timer KB*m* TKBO*m*0 forced output stop termination interrupt (INTTMKBSTP*m*0).
r_{Config_TKB0_TKB*n*}_tkb*m*_terminated1_interrupt		Executes processing in response to timer KB*m* TKBO*m*1 forced output stop termination interrupt (INTTMKBSTP*m*1).
r_{Config_TKB0_TKB*n*}_tkb*m*_activated0_interrupt		Executes processing in response to timer KB*m* TKBO*m*0 forced output stop activation interrupt (INTTMKBSTR*m*0).
r_{Config_TKB0_TKB*n*}_tkb*m*_activated1_interrupt		Executes processing in response to timer KB*m* TKBO*m*1 forced output stop activation interrupt (INTTMKBSTR*m*1).
r_{Config_TKB0_TKB*n*}_tkb*m*_end_count_interrupt		Executes processing in response to timer KB*m* count compare match interrupt (INTTMKB*m*).

R_{Config_TKB0_TKB*n*}_Create

This API function executes initialization processing that is required before controlling the TKB0 and TKB*n* modules in simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master).

Remark This API function is called from R_TKB_Create.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_Create(void);
```

Remark *n* is 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_Start

This API function starts the TKB0 and TKB*n* counters.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_Start(void);
```

Remark *n* is 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_Stop

This API function stops the TKB0 and TKB*n* counters.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_Stop(void);
```

Remark   *n* is 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKB*m*_Set_BatchOverwriteRequestOn

This API function sets TKB*m* batch overwrite request function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBm_Set_BatchOverwriteRequestOn(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Start

This API function starts TKB*m* TKBO*m*0 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_Forced_Output_Stop_Function1_Start(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Stop

This API function stops TKB*m* TKBO*m*0 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_Forced_Output_Stop_Function1_Stop(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Start

This API function starts TKB*m* TKBO*m*1 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_Forced_Output_Stop_Function1_Start(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Stop

This API function stops TKB*m* TKBO*m*1 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_Forced_Output_Stop_Function1_Stop(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Start

This API function starts TKB*m* TKBO*m*0 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_SmoothStartFunction_Start(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Stop

This API function stops TKB*m* TKBO*m*0 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_SmoothStartFunction_Stop(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Start

This API function starts TKB*m* TKBO*m*1 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_SmoothStartFunction_Start(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Stop

This API function stops TKB*m* TKBO*m*1 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_SmoothStartFunction_Stop(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*0_DitheringFunction_Start

This API function starts TKB*m* TKBO*m*0 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_DitheringFunction_Start(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*0_DitheringFunction_Stop

This API function stops TKB*m* TKBO*m*0 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_DitheringFunction_Stop(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*1_DitheringFunction_Start

This API function starts TKB*m* TKBO*m*1 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_DitheringFunction_Start(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_TKBO*m*1_DitheringFunction_Stop

This API function stops TKB*m* TKBO*m*1 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_DitheringFunction_Stop(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TKB0 and TKB*n*.

Remark This API functions is called from R_{Config_TKB0_TKB*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TKB0_TKBn}_Create_UserInit(void);
```

Remark *n* is 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB*n*}_tkb*m*_terminated0_interrupt

This API function executes processing in response to timer KB*m* TKBO*m*0 forced output stop termination interrupt (INTTMKBSTP*m*0).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB*n*}_tkb*m*_terminated1_interrupt

This API function executes processing in response to timer KB*m* TKBO*m*1 forced output stop termination interrupt (INTTMKBSTP*m*1).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TKB0_TKBn}_tkbm_terminated1_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TKB0_TKBn}_tkbm_terminated1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TKB0_TKBn}_tkbm_terminated1_interrupt(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB*n*}_tkb*m*_activated0_interrupt

This API function executes processing in response to timer KB*m* TKBO*m*0 forced output stop activation interrupt (INTTMKBSTR*m*0).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB*n*}_tkb*m*_activated1_interrupt

This API function executes processing in response to timer KB*m* TKBO*m*1 forced output stop activation interrupt (INTTMKBSTR*m*1).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_activated1_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKBn}_tkbm_activated1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_activated1_interrupt(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB*n*}_tkb*m*_end_count_interrupt

This API function executes processing in response to timer KB*m* count compare match interrupt (INTTMKBm).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_end_count_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKBn}_tkbm_end_count_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_end_count_interrupt(void);
```

Remark *n* is 1, 2. *m* is 0, *n*.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TKB0 and TKB1 to output PWM waveforms pulses:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
        EI();
        R_Config_TKB0_TKB1_Start();
        while (tkb_pwm_count < 10);
        R_Config_TKB0_TKB1_Stop();
}
```

Config_TKB0_TKB1_user.c

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

static void __near r_Config_TKB0_TKB1_tkb0_end_count_interrupt (void)
{
        /* Start user code for r_Config_TKB0_TKB1_tkb0_end_count_interrupt. Do not edit comment
generated here */
        tkb_pwm_count ++;
        /* End user code. Do not edit comment generated here */
}
```

4.2.15 PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (2 slaves)

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

Table 4-19 API Functions: (1/2)

API Function Name	Peripheral Name	Description
R_{Config_TKB0_TKB1_TKB2}_Create	Timer KB	Executes initialization processing that is required before controlling the TKB0, TKB1 and TKB2 modules in TKB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master).
R_{Config_TKB0_TKB1_TKB2}_Start		Starts the TKB0, TKB1 and TKB2 counters.
R_{Config_TKB0_TKB1_TKB2}_Stop		Stops the TKB0, TKB1 and TKB2 counters.
R_{Config_TKB0_TKB1_TKB2}_TKB*n* _Set_BatchOverwriteRequestOn		Sets TKB*n* batch overwrite request function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0 _Forced_Output_Stop_Function1_Start		Starts TKB*n* TKBO*n*0 forced output stop function 1.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0 _Forced_Output_Stop_Function1_Stop		Stops TKB*n* TKBO*n*0 forced output stop function 1.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1 _Forced_Output_Stop_Function1_Start		Starts TKB*n* TKBO*n*1 forced output stop function 1.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1 _Forced_Output_Stop_Function1_Stop		Stops TKB*n* TKBO*n*1 forced output stop function 1.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0 _SmoothStartFunction_Start		Starts TKB*n* TKBO*n*0 smooth start function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0 _SmoothStartFunction_Stop		Stops TKB*n* TKBO*n*0 smooth start function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1 _SmoothStartFunction_Start		Starts TKB*n* TKBO*n*1 smooth start function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1 _SmoothStartFunction_Stop		Stops TKB*n* TKBO*n*1 smooth start function.

Table 4-20 API Functions: (2/2)

API Function Name	Peripheral Name	Description
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_DitheringFunction_Start	Timer KB	Starts TKB*n* TKBO*n*0 dithering function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_DitheringFunction_Stop		Stops TKB*n* TKBO*n*0 dithering function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_DitheringFunction_Start		Starts TKB*n* TKBO*n*1 dithering function.
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_DitheringFunction_Stop		Stops TKB*n* TKBO*n*1 dithering function.
R_{Config_TKB0_TKB1_TKB2}_Create_UserInit		Executes user-specific initialization processing for the TKB0, TKB1 and TKB2.
r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated0_interrupt		Executes processing in response to timer KB*n* TKBO*n*0 forced output stop termination interrupt (INTTMKBSTP*n*0).
r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated1_interrupt		Executes processing in response to timer KB*n* TKBO*n*1 forced output stop termination interrupt (INTTMKBSTP*n*1).
r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated0_interrupt		Executes processing in response to timer KB*n* TKBO*n*0 forced output stop activation interrupt (INTTMKBSTR*n*0).
r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated1_interrupt		Executes processing in response to timer KB*n* TKBO*n*1 forced output stop activation interrupt (INTTMKBSTR*n*1).
r_{Config_TKB0_TKB1_TKB2}_tkb*n*_end_count_interrupt		Executes processing in response to timer KB*n* count compare match interrupt (INTTMKB*n*).

R_{Config_TKB0_TKB1_TKB2}_Create

This API function executes initialization processing that is required before controlling the TKB0, TKB1 and TKB2 modules in simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master).

Remark This API function is called from R_TKB_Create.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_Start

This API function starts the TKB0, TKB1 and TKB2 counters.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_Stop

This API function stops the TKB0, TKB1 and TKB2 counters.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKB*n*_Set_BatchOverwriteRequestOn

This API function sets TKB*n* batch overwrite request function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBn_Set_BatchOverwriteRequestOn(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Start

This API function starts TKB*n* TKBO*n*0 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_Forced_Output_Stop_Function1_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Stop

This API function stops TKB*n* TKBO*n*0 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_Forced_Output_Stop_Function1_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Start

This API function starts TKB*n* TKBO*n*1 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_Forced_Output_Stop_Function1_Start(void);
```

Remark   *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Stop

This API function stops TKB*n* TKBO*n*1 forced output stop function 1.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_Forced_Output_Stop_Function1_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Start

This API function starts TKB*n* TKBO*n*0 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_SmoothStartFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Stop

This API function stops TKB*n* TKBO*n*0 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_SmoothStartFunction_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Start

This API function starts TKB*n* TKBO*n*1 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_SmoothStartFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Stop

This API function stops TKB*n* TKBO*n*1 smooth start function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_SmoothStartFunction_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_DitheringFunction_Start

This API function start TKB*n* TKBO*n*0 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_DitheringFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_DitheringFunction_Stop

This API function stops TKB*n* TKBO*n*0 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_DitheringFunction_Stop(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_DitheringFunction_Start

This API function start TKB*n* TKBO*n*1 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Start(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_DitheringFunction_Stop

This API function stops TKB*n* TKBO*n*1 dithering function.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Stop(void);
```

Remark   *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TKB0_TKB1_TKB2}_Create_UserInit

This API function executes user-specific initialization processing for the TKB0, TKB1 and TKB2.

Remark This API functions is called from R_{Config_TKB0_TKB1_TKB2}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TKB0_TKB1_TKB2}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated0_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*0 forced output stop termination interrupt (INTTMKBSTP*n*0).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated0_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated0_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated1_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*1 forced output stop termination interrupt (INTTMKBSTP*n*1).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated1_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated1_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated0_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*0 forced output stop activation interrupt (INTTMKBSTR*n*0).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated0_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated0_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated1_interrupt

This API function executes processing in response to timer KB*n* TKBO*n*1 forced output stop activation interrupt (INTTMKBSTR*n*1).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated1_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated1_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_end_count_interrupt

This API function executes processing in response to timer KB*n* count compare match interrupt (INTTMKBn).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_end_count_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_end_count_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_end_count_interrupt(void);
```

Remark *n* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TKB0, TKB1 and TKB2 to output PWM waveforms pulses:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
	EI();
	R_Config_TKB0_TKB1_TKB2_Start();
	while (tkb_pwm_count < 10);
	R_Config_TKB0_TKB1_TKB2_Stop();
}
```

Config_TKB0_TKB1_TKB2_user.c

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

static void __near r_Config_TKB0_TKB1_TKB2_tkb0_end_count_interrupt (void)
{
	/* Start user code for r_Config_TKB0_TKB1_TKB2_tkb0_end_count_interrupt. Do not edit comment
generated here */
	tkb_pwm_count ++;
	/* End user code. Do not edit comment generated here */
}
```

4.2.16 Input Pulse Interval/Period Measurement (Timer Array Unit)

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

Table 4-21 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 (INTTM*mn*).

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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

void main(void)
{
	EI();
	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);
}
```

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 */
	/* Set the flag */
	tau_interrupt_flag = 1U;
	/* End user code. Do not edit comment generated here */
}
```

4.2.17 Input Pulse Interval/Period Measurement (Timer RJ)

Below is a list of API functions output by the Smart Configurator for input pulse period of an external signal (input to the TRJIO*n* pin) measurement use.

Table 4-22 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRJ*n*}_Create	Timer RJ*n*	Executes initialization processing that is required before controlling the TRJn module in input pulse width measurement mode.
R_{Config_TRJ*n*}_Start		Starts the TRJ*n* counter.
R_{Config_TRJ*n*}_Stop		Stops the TRJ*n* counter.
R_{Config_TRJ*n*}_Get_PulseWidth		Measures TRJ*n* input pulse width.
R_{Config_TRJ*n*}_Create_UserInit		Executes user-specific initialization processing for the TRJ*n*.
r_{Config_TRJ*n*}_ interrupt		Executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

R_{Config_TRJ*n*}_Create

This API function executes initialization processing that is required before controlling the TRJ*n* module in input pulse width of an external signal (input to the TRJIO*n* pin) measurement mode.

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Start

This API function starts the TRJ*n* counter.

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Stop

This API function stops the TRJ*n* counter.

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Get_PulseWidth

This API function measures TRJ*n* input pulse width.

[Syntax]

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

Remark *n* is 0.

[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_TRJ*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRJ*n*}_interrupt

This API function executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

Remark This API function is called as the interrupt handler for capture interrupts (INTTRJ*n*), which occur when the measurement of the active width of the external input (TRJIO*n*) is completed in pulse width measurement mode.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TRJn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_TRJn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TRJ0 input pulse period:

(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
    EI();
    R_Config_TAU0_0_Start();    //Output square wave from TAU0_0
    R_Config_TRJ0_Start();

    while (trj_interrupt_flag < 20);
    R_Config_TRJ0_Get_PulseWidth(&width);

    while (1U);
}
```

Config_TRJ0_user.c

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

static void __near r_Config_TRJ0_interrupt (void)
{
    ...
    /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
    trj_interrupt_flag++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.18 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 4-23 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 (INTTM*mn*).
r_{Config_TAU*m*_*n*}_higher8bits_interrupt		Executes processing in response to timer channel*n* count end interrupt (INTTM*mn*H) (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 (INTTM*mn*).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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 (INTTM*mn*H) (at higher 8-bit timer operation).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

void main(void)
{
        EI();
        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();
}
```

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 */
}
```

4.2.19 Interval Timer (Timer RJ)

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

Table 4-24 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRJ*n*}_Create	Timer RJ*n*	Executes initialization processing that is required before controlling the Timer RJ*n* module in interval timer mode.
R_{Config_TRJ*n*}_Start		Starts the TRJ*n* counter.
R_{Config_TRJ*n*}_Stop		Stops the TRJ*n* counter.
R_{Config_TRJ*n*}_Create_UserInit		Executes user-specific initialization processing for the TRJ*n*.
r_{Config_TRJ*n*}_ interrupt		Executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

R_{Config_TRJ*n*}_Create

This API function executes initialization processing that is required before controlling the TRJ*n* module in inverval timer mode.

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Start

This API function starts the TRJ*n* counter.

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Stop

This API function stops the TRJ*n* counter.

[Syntax]

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

Remark   *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRJ*n*}_interrupt

This API function executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

Remark This API function is called as the interrupt handler for TRJ*n* interrupts (INTTRJ*n*), which occur when the count value reaches 0000H and the next count source is input, the counter underflows.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TRJ0 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 uint8_t ch0_run_count;

void main(void);

void main(void)
{
        EI();
        R_Config_TRJ0_Start();
        while (ch0_run_count < 20);
        R_Config_TRJ0_Stop();
}
```

Config_TRJ0_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_TRJ0_interrupt (void)
{
        /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
        ch0_run_count ++;
        P0_bit.no0 = ~P0_bit.no0;
        /* End user code. Do not edit comment generated here */
}
```

4.2.20 Interval Timer (12-bit Interval Timer)

Below is a list of API functions output by the Smart Configurator for interval timer (12-bit Interval Timer) use.

Table 4-25 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_IT}_Create	12-bit Interval Timer	Executes initialization processing that is required before controlling the 12-bit interval timer module.
R_{Config_IT}_Start		Starts the 12-bit interval timer counter.
R_{Config_IT}_Stop		Stops the 12-bit interval timer counter.
R_{Config_IT}_Create_UserInit		Executes user-specific initialization processing for the 12-bit interval timer.
r_{Config_IT}_ interrupt		Executes processing in response to the interrupt (INTIT) when 12-bit interval timer counter underflows.

R_{Config_IT}_Create

This API function executes initialization processing that is required before controlling the 12-bit interval timer module in interval timer mode.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_IT}_Start

This API function starts the 12-bit interval timer counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_IT}_Stop

This API function stops the 12-bit interval timer counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_IT}_Create_UserInit

This API function executes user-specific initialization processing for the 12-bit interval timer.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_IT}_interrupt

This API function executes processing in response to the interrupt (INTIT) when 12-bit interval timer counter is same as compare value.

Remark This API function is called as the interrupt handler for 12-bit interval timer interrupts (INTIT), which occur when the count value reaches compare value.

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_{Config_IT}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void     r_{Config_IT}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using 12-bit Interval Timer 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 uint8_t count;

void main(void);

void main(void)
{
        EI();
        R_Config_IT_Start();
        while (count < 20);
        R_Config_IT_Stop();
}
```

Config_IT_user.c

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

static void __near r_Config_IT_interrupt (void)
{
        /* Start user code for r_Config_IT_interrupt. Do not edit comment generated here */
        count ++;
        P0_bit.no0 = ~P0_bit.no0;
        /* End user code. Do not edit comment generated here */
}
```

4.2.21 One-Shot Pulse Output

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

Table 4-26 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*}_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*}_channel*n*_interrupt		Executes processing in response to timer channel*n* count end interrupt (INTTM*mn*).
r_{Config_TAU*m*_*n*}_channel*p*_interrupt		Executes processing in response to timer channel*p* count end interrupt (INTTM*mp*).

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*}_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*}_channel*n*_interrupt

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

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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 (INTTMmp).

Remark1. In one-shot pulse output function, this API function is called as the interrupt handler for count end interrupt (INTTMmp), 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 (INTTMmp), which occur when the valid capture edge is detected, and the current counter value (TCRmp) is transferred to timer data register mp (TDRmp).

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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

void main(void);

void main(void)
{
	EI();
	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();

}
```

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 */
}
```

4.2.22 Square Wave Output (Timer Array Unit)

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

Table 4-27 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* count end interrupt (INTTM*mn*).

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* 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

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 (INTTMmn).

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

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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

void main(void);

void main(void)
{
        EI();
        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();

}
```

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 */
}
```

4.2.23 Square Wave Output (Timer RJ)

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

Table 4-28 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRJ*n*}_Create	Timer RJ*n*	Executes initialization processing that is required before controlling the Timer RJ*n* module in square wave output mode.
R_{Config_TRJ*n*}_Start		Starts the TRJ*n* counter.
R_{Config_TRJ*n*}_Stop		Stops the TRJ*n* counter.
R_{Config_TRJ*n*}_Create_UserInit		Executes user-specific initialization processing for the TRJ*n*.
r_{Config_TRJ*n*}_ interrupt		Executes processing in response to the interrupt (INTTRJ*n*) when TRJ*n* counter underflows.

R_{Config_TRJ*n*}_Create

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

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Start

This API function starts the TRJ*n* counter.

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Stop

This API function stops the TRJ*n* counter.

[Syntax]

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

Remark   *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRJ*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRJ*n*}_interrupt

This API function executes processing in response to the interrupt (INTTRJn) when TRJn counter underflows.

Remark This API function is called as the interrupt handler for TRJn interrupts (INTTRJ*n*), which occur when the count value reaches 0000H and the next count source is input, the counter underflows.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TRJ0 counting to output an inverted pulse from pins TRJIO0 and TRJO0 for a user-defined period:

(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
	EI();
	R_Config_TRJ0_Start();
	while (ch0_run_count < 20);
	R_Config_TRJ0_Stop();
}
```

Config_TRJ0_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_TRJ0_interrupt (void)
{
	/* Start user code for r_Config_ TRJ0_interrupt. Do not edit comment generated here */
	ch0_run_count ++;
	/* End user code. Do not edit comment generated here */
}
```

4.2.24 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 4-29 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 (INTITL)

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 (INTITL).

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]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

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

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 */
        interrupt_flag ++;
        P0 = ~P0;
        /* End user code. Do not edit comment generated here */
}
```

4.2.25 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 4-30 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 (INTITL).

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 (INTITL).

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]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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"

void main(void);

void main(void)
{
        EI();
        R_Config_ITL000_ITL001_Set_OperationMode();
        /* Capture setting */
        ITLCC0 |= _80_ITL_CAPTURE_ENABLE;
        CAPF0CR = 1U;
        ITLCC0 |= _00_ITL_CAPTURE_COUNTER_RETAIN;
        ITLCC0 &= _FC_ITL_CAPTURE_TRIGGER_CLEAR;
        ITLCC0 |= _00_ITL_CAPTURE_TRIGGER_SOFTWARE;
        R_Config_ITL000_ITL0011_Start();
        R_Config_ITL000_ITL0011_Set_SoftwareTriggerOn();
}
```

Config_ITL000_user.c

```
volatile uint16_t value = 0U;

void r_Config_ITL000_callback_shared_interrupt(void)
{
    R_Config_ITL000_ITL0011_Get_CaptureValue (&value);
}
```

4.2.26 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 4-31 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 (INTITL).

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 (INTITL).

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]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

void main(void)
{
        EI();
        R_Config_ITL000_ITL001_ITL012_ITL013_Start();
        while (itl_run_count < 20);
        R_Config_ITL000_ITL001_ITL012_ITL013_Stop();
}
```

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 */
}
```

4.2.27 Input Capture Function (Timer RD)

Below is a list of API functions output by the Smart Configurator for Input Capture Function use.

Table 4-32 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRD*n*}_Create	Timer RD	Executes initialization processing that is required before controlling the TRD*n* module in Input Capture Function mode.
R_{Config_TRD*n*}_Start		Starts the TRD*n* counter.
R_{Config_TRD*n*}_Stop		Stops the TRD*n* counter.
R_{Config_TRD*n*}_Get_PulseWidth		Measures TRD*n* input pulse width.
R_{Config_TRD*n*}_Create_UserInit		Executes user-specific initialization processing for the TRD*n*.
r_{Config_TRD*n*}_trd*n*_interrupt		Executes processing in response to timer RD*n* capture interrupt (INTTRD*n*).

R_{Config_TRD*n*}_Create

This API function executes initialization processing that is required before controlling the TRD*n* module in Input Capture Function mode.

Remark This API function is called from R_TRD_Create.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Start

This API function starts the TRD*n* counter.

[Syntax]

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

Remark   *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Stop

This API function stops the TRD*n* counter.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Get_PulseWidth

This API function calculates the TRD*n* pulse width.

[Syntax]

```
MD_STATUS R_{Config_TRDn}_Get_PulseWidth (uint32_t * const active_width, uint32_t * const inactive_width, e_timer_channel_t channel);
```

Remark *n* is 0, 1.

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * const active_width;	The high-level width
O	uint32_t * const inactive_width;	The low-level width
I	e_timer_channel_t channel	The TRDIO*ji* pin (i = 0 or 1, j = A, B, C, or D) external signal and ELC signal input.

Remark Below is shown the structure e_timer_channel_t.

```
typedef enum
{
        TMCHANNELA,
        TMCHANNELB,
        TMCHANNELC,
        TMCHANNELD,
        TMCHANNELELC
} e_timer_channel_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Counter doesn’t work as capture mode.
MD_ARGERROR	Argument input error.

R_{Config_TRD*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRD*n*}_trd*n*_interrupt

This API function executes processing in response to timer RD*n* capture interrupt (INTTRD*n*).

Remark This API function is called as the interrupt handler for capture interrupts (INTTRD*n*), which occur when the valid capture edge of TRDIO*jn* (j = A, B, C, or D) input is detected, or TRD*n* register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TRDn}_trdn_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TRDn}_trdn_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TRDn}_trdn_interrupt(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TRD0 input pulse width:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

typedef struct {
	uint32_t active_width;
	uint32_t inactive_width;
} TRD_PulseWidth_t;

TRD_PulseWidth_t trd_a;
TRD_PulseWidth_t trd_b;
TRD_PulseWidth_t trd_d;

static void delay_ms(uint32_t time_ms)
{
	uint32_t i = 0;
	while(time_ms--) {
		for(i = 0; i < 156; I++) {
			NOP();
		}
	}
}

void main(void)
{
	EI();

	P0 = 0x00U;
	R_Config_TAU0_0_Start();
	R_Config_TAU0_1_Start();
	R_Config_TAU0_2_Start();
	R_Config_TRD0_Start();

	delay_ms(2000);
	R_Config_TRD0_Get_PulseWidth(&trd_a.active_width, &trd_a.inactive_width, TMCHANNELA);
	R_Config_TRD0_Get_PulseWidth(&trd_b.active_width, &trd_b.inactive_width, TMCHANNELB);
	R_Config_TRD0_Get_PulseWidth(&trd_d.active_width, &trd_d.inactive_width, TMCHANNELC);

	while(1);
}
```

Config_TRD0_user.c

```
static void __near r_Config_TRD0_interrupt(void)
{
    ...
      /* Start user code for r_Config_TRD0_interrupt. Do not edit comment generated here */

    /* TRDGRA0 input capture interrupt */
    P0_bit.no0 = ~P0_bit.no0;

    /* TRDGRB0 input capture interrupt */
    P0_bit.no1 = ~P0_bit.no1;

    /* TRDGRD0 input capture interrupt */
    P0_bit.no2 = ~P0_bit.no2;
    /* End user code. Do not edit comment generated here */
}
```

4.2.28 Input Capture Function (Timer RG)

Below is a list of API functions output by the Smart Configurator for Input Capture Function use.

Table 4-33 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRG}_Create	Timer RG	Executes initialization processing that is required before controlling the TRG module in Input Capture Function mode.
R_{Config_TRG}_Start		Starts the TRG counter.
R_{Config_TRG}_Stop		Stops the TRG counter.
R_{Config_TRG}_Get_PulseWidth		Measures TRG input pulse width.
R_{Config_TRG}_Create_UserInit		Executes user-specific initialization processing for the TRG.
r_{Config_TRG}_interrupt		Executes processing in response to timer RG capture interrupt (INTTRG).

R_{Config_TRG}_Create

This API function executes initialization processing that is required before controlling the TRG module in Input Capture Function mode.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Start

This API function starts the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Stop

This API function stops the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Get_PulseWidth

This API function calculates the TRG pulse width.

[Syntax]

```
MD_STATUS R_{Config_TRG}_Get_PulseWidth(uint32_t * const active_width, uint32_t * const inactive_width, e_trg_channel_t channel);
```

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * const active_width;	The high-level width
O	uint32_t * const inactive_width;	The low-level width
I	e_trg_channel_t channel	The TRGIOA, TRGIOB and ELC signal input.

Remark Below is shown the structure e_trg_channel_t.

```
typedef enum
{
    TRG_CHANNELA,
    TRG_CHANNELB,
    TRG_CHANNELELC
} e_trg_channel_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Counter doesn’t work as capture mode.
MD_ARGERROR	Argument input error.

R_{Config_TRG}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRG}_interrupt

This API function executes processing in response to timer RG capture interrupt (INTTRG).

Remark This API function is called as the interrupt handler for capture interrupts (INTTRG), which occur when the valid capture edge of TRGIOA and TRGIOB input is detected, or TRG register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRG}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TRG input pulse width:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

typedef struct {
	uint32_t active_width;
	uint32_t inactive_width;
} TRG_PulseWidth_t;

TRG_PulseWidth_t trg_a;
TRG_PulseWidth_t trg_b;

static void delay_ms(uint32_t time_ms)
{
	uint32_t i = 0;
	while(time_ms--) {
		for(i = 0; i < 156; I++) {
			NOP();
		}
	}
}

void main(void)
{
	EI();

	P0 = 0x00U;
	R_Config_TAU0_0_Start();
	R_Config_TAU0_1_Start();
	R_Config_TRG_Start();

	delay_ms(2000);
	R_Config_TRG_Get_PulseWidth(&trg_a.active_width, &trg_a.inactive_width, TMCHANNELA);
	R_Config_TRG_Get_PulseWidth(&trg_b.active_width, &trg_b.inactive_width, TMCHANNELB);

	while(1);
}
```

Config_TRG_user.c

```
static void __near r_Config_TRG_interrupt(void)
{
        ...
          /* Start user code for r_Config_TRG_interrupt. Do not edit comment generated here */

        /* TRGGRA input capture interrupt */
        P0_bit.no0 = ~P0_bit.no0;

        /* TRGGRB input capture interrupt */
        P0_bit.no1 = ~P0_bit.no1;
        /* End user code. Do not edit comment generated here */
}
```

4.2.29 Input Capture Function (Timer RX)

Below is a list of API functions output by the Smart Configurator for Input Capture Function use.

Table 4-34 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRX}_Create	Timer RX	Executes initialization processing that is required before controlling the TRX module in Input Capture Function mode.
R_{Config_TRX}_Start		Starts the TRX counter.
R_{Config_TRX}_Stop		Stops the TRX counter.
R_{Config_TRX}_Get_BufferValue		Gets the TRX buffer value.
R_{Config_TRX}_Create_UserInit		Executes user-specific initialization processing for the TRX.
r_{Config_TRX}_interrupt		Executes processing in response to timer RX capture interrupt (INTTRX).

R_{Config_TRX}_Create

This API function executes initialization processing that is required before controlling the TRX module in Input Capture Function mode.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRX}_Start

This API function starts the TRX counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRX}_Stop

This API function stops the TRX counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRX}_Get_BufferValue

This API function gets the TRX buffer value.

[Syntax]

```
void R_{Config_TRX}_Get_BufferValue(uint32_t * const value);
```

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * value;	Buffer value

[Return value]

None.

R_{Config_TRX}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRX}_interrupt

This API function executes processing in response to timer RX capture interrupt (INTTRX).

Remark This API function is called as the interrupt handler for capture interrupts (INTTRX), which occur when comparator interrupt signal is detected, or TRX register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TRX}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_TRX}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TRX input pulse width:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

volatile uint8_t comp_flag = 0U;

void main(void);

void main(void)
{
        EI();
        R_Config_TRX_Start();
        R_Config_COMP0_Start();

        // The first interrupt is dummy value, no need to store in trx_buffer
        while(comp_flag != 1U);
        comp_flag = 0U;

        for (char i = 0; i < 10; i++)
        {
                while(comp_flag != 1U);
                R_Config_TRX_Get_BufferValue(trx_buffer + i);
                comp_flag = 0U;
        }

        while(1);
}
```

Config_COPM0_user.c

```
extern volatile uint8_t comp_flag;
static void __near r_Config_COMP2_interrupt(void)
{
        ...
         /* Start user code for r_Config_TRX_interrupt. Do not edit comment generated here */
         comp_flag = 1U;
         /* End user code. Do not edit comment generated here */
}
```

4.2.30 Output Compare Function (Timer RD)

Below is a list of API functions output by the Smart Configurator for Output Compare Function mode use.

Table 4-35 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRD*n*}_Create	Timer RD	Executes initialization processing that is required before controlling the TRD*n* module in Output Compare Function mode.
R_{Config_TRD*n*}_Start		Starts the TRD*n* counter.
R_{Config_TRD*n*}_Stop		Stops the TRD*n* counter.
R_{Config_TRD*n*}_Create_UserInit		Executes user-specific initialization processing for the TRD*n*.
r_{Config_TRD*n*}_trd*n*_interrupt		Executes processing in response to timer TRD*n* count compare match interrupt (INTTRD*n*).

R_{Config_TRD*n*}_Create

This API function executes initialization processing that is required before controlling the TRD*n* module in Output Compare Function mode.

Remark This API function is called from R_TRD_Create.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Start

This API function starts the TRD*n* counter.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Stop

This API function stops the TRD*n* counter.

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRD*n*}_trd*n*_interrupt

This API function executes processing in response to timer TRD*n* count compare match interrupt (INTTRD*n*).

Remark This API function is called as the interrupt handler for count compare match interrupt (INTTRD*n*), which occur when the content of the TRD*n* register matches content of the TRDGR*jn* (j = A, B, C, or D) register or TRD*n* register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRDn}_trdn_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_TRDn}_trdn_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_TRDn}_trdn_interrupt(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TRD1 to output an arbitrary level from the TRDIOj1 pin:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
        EI();
        R_Config_TRD1_Start();
        while (ch1_run_count < 20);
        R_Config_TRD1_Stop();
}
```

Config_TRD1_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_TRD1_trd1_interrupt (void)
{
        /* Start user code for r_Config_ TRD1_trd1_interrupt. Do not edit comment generated here */
        ch1_run_count ++;
        /* End user code. Do not edit comment generated here */
}
```

4.2.31 Output Compare Function (Timer RG)

Below is a list of API functions output by the Smart Configurator for Output Compare Function mode use.

Table 4-36 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRG}_Create	Timer RG	Executes initialization processing that is required before controlling the TRG module in Output Compare Function mode.
R_{Config_TRG}_Start		Starts the TRG counter.
R_{Config_TRG}_Stop		Stops the TRG counter.
R_{Config_TRG}_Create_UserInit		Executes user-specific initialization processing for the TRG.
r_{Config_TRG}_TRG_interrupt		Executes processing in response to timer TRG count compare match interrupt (INTTRG).

R_{Config_TRG}_Create

This API function executes initialization processing that is required before controlling the TRG module in Output Compare Function mode.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Start

This API function starts the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Stop

This API function stops the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRG}_interrupt

This API function executes processing in response to timer TRG count compare match interrupt (INTTRG).

Remark This API function is called as the interrupt handler for count compare match interrupt (INTTRG), which occur when the content of the TRG register matches content of the TRGGR*j* (j = A, B, C, or D) register or TRG register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRG}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TRG to output an arbitrary level from the TRGIOj pin:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
        EI();
        R_Config_TRG_Start();
        while (ch1_run_count < 20);
        R_Config_TRG_Stop();
}
```

Config_TRG_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_TRG_interrupt (void)
{
        /* Start user code for r_Config_ TRG_interrupt. Do not edit comment generated here */
        ch1_run_count ++;
        /* End user code. Do not edit comment generated here */
}
```

4.2.32 Three -phase PWM Output (Timer RD)

Below is a list of API functions output by the Smart Configurator for Three -phase PWM output (for Timer RD using reset synchronous PWM mode/ complementary PWM mode/ extended complementary PWM mode) use.

Table 4-37 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRD0_TRD1}_Create	Timer RD	Executes initialization processing that is required before controlling the TRD0 and TRD1 module in reset synchronous PWM mode/ complementary PWM mode/ extended complementary PWM mode.
R_{Config_TRD0_TRD1}_Start		Starts the TRD0 and TRD1 counter.
R_{Config_TRD0_TRD1}_Stop		Stops the TRD0 and TRD1 counter.
R_{Config_ TRD0_TRD1}_Set_TRD_ReloadTrigger		Generates TRD0 and TRD1 buffer registers reload trigger.
R_{Config_TRD0_TRD1}_Create_UserInit		Executes user-specific initialization processing for the TRD0_TRD1.
r_{Config_TRD0_TRD1}_trd0_interrupt		Executes processing in response to timer RD0 count compare match interrupt (INTTRD0).
r_{Config_TRD0_TRD1}_trd1_interrupt		Executes processing in response to timer RD1 count compare match interrupt (INTTRD1).

R_{Config_TRD0_TRD1}_Create

This API function executes initialization processing that is required before controlling the TRD0 module in reset synchronous PWM mode or the TRD0 and TRD1 module in complementary PWM mode extended complementary PWM mode.

Remark This API function is called from R_TRD_Create.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD0_TRD1}_Start

This API function starts the TRD0 counter in reset synchronous PWM mode or the TRD0 and TRD1 counter in complementary PWM mode / extended complementary PWM mode.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD0_TRD1}_Stop

This API function stops the TRD0 counter in reset synchronous PWM mode or the TRD0 and TRD1 counter in complementary PWM mode/ extended complementary PWM mode.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRD0_TRD1}_Set_TRD_ReloadTrigger

This API function generates TRD0 and TRD1 buffer registers reload trigger in Extended complementary PWM mode.

[Syntax]

```
MD_STATUS R_{Config_ TRD0_TRD1}_Set_TRD_ReloadTrigger (st_extpwm_buffer_registers_t *
buffer);
```

[Argument(s)]

I/O	Argument(s)	Description
I	st_extpwm_buffer_registers_t * buffer;	buffer registers value

Remark Below is shown the structure st_extpwm_buffer_registers_t.

```
typedef struct {
    uint16_t trdgrd0;
    uint16_t trdcmpd0;
    uint16_t trdgrc1;
    uint16_t trdcmpc1;
    uint16_t trdgrd1;
    uint16_t trdcmpd1;
    uint16_t trdadtb0;
    uint16_t trdadtb1;
} st_extcompwm_buffer_registers_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR	Waiting for reload trigger status

R_{Config_TRD0_TRD1}_Create_UserInit

This API function executes user-specific initialization processing for the TRD0_TRD1.

Remark This API functions is called from R_{Config_TRD0_TRD1}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TRD0_TRD1}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRD0_TRD1}_trd0_interrupt

This API function executes processing in response to timer TRD0 count compare match interrupt (INTTRD0) in reset synchronous PWM mode/ complementary PWM mode or response to interrupt request signal 0 (INTTRD0) with decimation control in extended complementary PWM mode mode.

Remark 1. In reset synchronous PWM mode:
This API function is called as the interrupt handler for count compare match interrupt (INTTRD0), which occur when the content of the TRD0 register matches content of the TRDGR*j*0 (j = A, B, C, or D) register or TRD0 register overflow.

Remark 2. In complementary PWM mode mode:
This API function is called as the interrupt handler for count compare match interrupt (INTTRD0), which occur when the content of the TRD0 register matches content of the TRDGR*j*0 (j = A, B, C, or D) register.

Remark 3. In extended complementary PWM mode mode:
This API function is called as the interrupt handler for interrupt request signal 0 (INTTRD0), which occur when TRD1 register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_{Config_TRD0_TRD1}_trd0_interrupt(void);
```

For LLVM toolchain:

```
void     r_{Config_TRD0_TRD1}_trd0_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void     r_{Config_TRD0_TRD1}_trd0_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRD0_TRD1}_trd1_interrupt

This API function executes processing in response to timer TRD1 count compare match interrupt (INTTRD1) or timer TRD1 interrupt request signal 1 (INTTRD1) with decimation control in extended complementary PWM mode.

Remark1 In reset synchronous PWM mode:
This API function is called as the interrupt handler for count compare match interrupt (INTTRD1), which occur when the content of the TRD1 register matches content of the TRDGRA1 and TRDGRB1 register.

Remark2 In complementary PWM mode mode:
This API function is called as the interrupt handler for count compare match interrupt (INTTRD1), which occur when the content of the TRD1 register matches content of the TRDGR*j*1 (j = A, B, C, or D) register or TRD1 register underflow.

Remark3 In extended complementary PWM mode mode:
This API function is called as the interrupt handler for interrupt request signal 1 (INTTRD1), which occur when TRD1 register underflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near       r_{Config_TRD0_TRD1}_trd1_interrupt(void);
```

For LLVM toolchain:

```
void       r_{Config_TRD0_TRD1}_trd1_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void       r_{Config_TRD0_TRD1}_trd1_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for outputting three normal-phases and three counter-phases of the symmetric or asymmetric PWM waveform with the same period in extended complementary PWM mode:
(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
        EI();
        R_Config_TRD0_TRD1_Start();
        while (trd_pwm_count < 20);
        R_Config_TRD0_TRD1_Stop();
}
```

Config_TRD0_TRD1_user.c

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

static void __near r_Config_TRD0_TRD1_trd0_interrupt (void)
{
        /* Start user code for r_Config_TRD0_trd0_interrupt. Do not edit comment generated here */
        trd_pwm_count ++;
        /* End user code. Do not edit comment generated here */
}

static void __near r_Config_TRD0_TRD1_trd1_interrupt (void)
{
        /* Start user code for r_Config_TRD0_trd1_interrupt. Do not edit comment generated here */
        trd_pwm_count ++;
        /* End user code. Do not edit comment generated here */
}
```

4.2.33 PWM option unit A (Timer RD)

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

Table 4-38 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_PWMOPA}_Create	Timer RD	Executes initialization processing that is required before controlling the PWM option unit A.
R_{Config_PWMOPA}_Software_Release		Releases output by software.
R_{Config_PWMOPA}_Create_UserInit		Executes user-specific initialization processing for the PWM option unit A.

R_{Config_ PWMOPA }_Create

This API function executes initialization processing that is required before controlling the PWM option unit A.

Remark This API function is called from R_TRD_Create.

[Syntax]

```
void R_{Config_ PWMOPA }_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_PWMOPA}_Software_Release

This API function releases output by software.

[Syntax]

```
void    R_{Config_PWMOPA}_Software_Release(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ PWMOPA }_Create_UserInit

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

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

[Syntax]

```
void    R_{Config_ PWMOPA }_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for pulse output from the timer RD output pin TRDIOji (j = A, B, C, D; i = 0, 1) can release forced cutoff by software trigger and pulse output is resumed:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
	EI();
	R_Config_PWMOPA_Software_Release();

	while(1);
}
```

4.2.34 Phase Counting Mode

Below is a list of API functions output by the Smart Configurator for detecting a phase difference between external input signals from two pins TRGCLKA and TRGCLKB and the TRG counter is incremented or decremented.

Table 4-39 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TRG}_Create	Timer RG	Executes initialization processing that is required before controlling the TRG module in phase counting mode.
R_{Config_TRG}_Start		Starts the TRG counter.
R_{Config_TRG}_Stop		Stops the TRG counter.
R_{Config_TRG}_Get_MeasurementCapture		Gets TRG measurement capture value to calculate phase change times on TRGCLKA and TRGCLKB.
R_{Config_TRG}_Create_UserInit		Executes user-specific initialization processing for the TRG.
r_{Config_TRG}_interrupt		Executes processing in response to timer RG count compare match interrupt (INTTRG).
r_{Config_TRG}_clear_interrupt		Executes processing in response to timer RG count compare match counter clearing and Z-signal detection counter clearing interrupt (INTGCR).
r_{Config_TRG}_capture_interrupt		Executes processing in response to timer RG TRGPMC count compare match interrupt (INTPMC).

R_{Config_TRG}_Create

This API function executes initialization processing that is required before controlling the TRG module in phase counting mode.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Start

This API function starts the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Stop

This API function stops the TRG counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_TRG}_Get_MeasurementCapture

This API function get TRG measurement capture value to calculate phase change times on TRGCLKA and TRGCLKB.

[Syntax]

```
void R_{Config_TRG}_Get_MeasurementCapture(uint16_t * const capture_value);
```

[Argument(s)]

I/O	Argument(s)	Description
O	uint16_t * const capture_value;	Measurement capture value

[Return value]

None.

R_{Config_TRG}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRG}_interrupt

This API function executes processing in response to timer RG count compare match interrupt (INTTRG).

Remark This API function is called as the interrupt handler for count compare match interrupt (INTTRG), which occur when the content of the TRG register matches content of the TRGGR*h* (h = A, B, C, or D) register or TRG register overflow.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_TRG}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRG}_clear_interrupt

This API function executes processing in response to timer RG count compare match counter clearing and Z-signal detection counter clearing interrupt (INTGCR).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TRG}_clear_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TRG}_clear_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TRG}_clear_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_TRG}_capture_interrupt

This API function executes processing in response to timer RG TRGPMC count compare match interrupt (INTPMC).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_TRG}_capture_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_TRG}_capture_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_TRG}_capture_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for Phase Counting Mode clearing by TRGGRA compare match:

(Blue code is user code.)

main.c

```
#include "r_cg_macrodriver.h"
#include "Config_TAU0_0.h"
#include "Config_TAU0_3.h"
#include "Config_TRG.h"

extern uint8_t count;

void main(void);

void main(void)
{
        EI();

        //support external signal input to the TRGCLKA
        R_Config_TAU0_0_Start();
        //support external signal input to the TRGCLKB
        R_Config_TAU0_3_Start();
        R_Config_TRG_Start();

        while (count > 0U);
        R_Config_TRG_Stop();
}
```

Config_TRG_user.c

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

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

4.2.35 Clock Output/Buzzer Output Controller

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

Table 4-40 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_PCLBUZ*n*}_Create	Clock Output/Buzzer Output Controller	Executes initialization processing that is required before controlling the PCLBUZ*n* module.
R_{Config_PCLBUZ*n*}_Start		Starts the PCLBUZ*n* module.
R_{Config_PCLBUZ*n*}_Stop		Stops the PCLBUZ*n* module.
R_{Config_PCLBUZ*n*}_Create_UserInit		Executes user-specific initialization processing for the PCLBUZ*n*.

R_{Config_PCLBUZ*n*}_Create

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

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

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_PCLBUZ*n*}_Start

This API function starts the PCLBUZn converter.

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_PCLBUZ*n*}_Stop

This API function stops the PCLBUZn converter.

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_PCLBUZ*n*}_Create_UserInit

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

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

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using clock output/buzzer output controller 0:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
        EI();
        R_Config_PCLBUZ0_Start();
}
```

4.2.36 Real-Time Clock

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

Table 4-41 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 INTRTC interrupt.
r_{Config_RTC}_callback_constperiod		Executes processing in response to INTRTC fixed-cycle interrupt.
r_{Config_RTC}_callback_alarm		Executes processing in response to INTRTC 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 constant period of INTRTC

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 INTRTC interrupt.

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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 INTRTC 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 INTRTC 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"

void main(void);

void main(void)
{
        EI();
        R_Config_RTC_Set_AlarmOn();
        R_ Config_RTC_Start();
}
```

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 intetrrupt 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;

void main(void);

void main(void)
{
        EI();
        R_Config_RTC_Set_ConstPeriodInterruptOn(ONEHOUR);
        R_ Config_RTC_Start();
}
```

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 */
}
```

4.2.37 A/D Converter

Below is a list of API functions output by the Smart Configurator for A/D converter use.

Table 4-42 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}_ADS*n*_Set_ADChannel		Selects analog input channel (Only when selecting "AD Advanced Mode").
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}_ADS*n*_Get_Result_10bit		Returns the high 10 bits conversion result in the buffer (Only when selecting "AD Advanced Mode").
R_{Config_ADC}_ADS*n*_Get_Result_8bit		Returns the high 8 bits conversion result in the buffer (Only when selecting "AD Advanced Mode").
R_{Config_ADC}_ADS*n*_Get_Result_12bit		Returns the low 12 bits conversion result in the buffer (Only when selecting "AD Advanced Mode").
R_{Config_ADC}_Create_UserInit		Executes user-specific initialization processing for the AD converter.
r_{Config_ADC}_interrupt		Executes processing in response to INTAD interrupt.
r_{Config_ADC}_ad*n*_interrupt		Executes processing in response to INTAD*n* interrupt (Only when selecting "AD Advanced Mode").

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}_ADSn_Set_ADChannel

This API function selects analog input channel (Only when selecting "AD Advanced Mode").

[Syntax]

```
MD_STATUS    R_{Config_ADC}_ADSn_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}_ADS*n*_Get_Result_10bit

This API function returns the high 10 bits conversion result in the buffer (Only when selecting "AD Advanced Mode").

[Syntax]

```
void    R_{Config_ADC}_ADSn_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}_ADS*n*_Get_Result_8bit

This API function returns the high 8 bits conversion result in the buffer (Only when selecting "Advanced Mode").

[Syntax]

```
void    R_{Config_ADC}_ADSn_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}_ADS*n*_Get_Result_12bit

This API function returns the low 12 bits conversion result in the buffer (Only when selecting "Advanced Mode").

[Syntax]

```
void    R_{Config_ADC}_ADSn_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 INTAD interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_ADC}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_ADC}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_ADC}_ad*n*_interrupt

This API function executes processing in response to INTAD*n* interrupt (Only when selecting "AD Advanced Mode").

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_ADC}_adn_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_ADC}_adn_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_ADC}_adn_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;

void main(void);

void main(void)
{
       EI();
       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();
}
```

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 */
       /* Set the flag */
       adc_Interrupt_flag = 1U;
       /* End user code. Do not edit comment generated here */
}
```

4.2.38 12 Bit A/D Single Scan

Below is a list of API functions output by the Smart Configurator for 12 Bit A/D Single Scan use.

Table 4-43 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_S12AD*n*}_Create	12-bit A/D converter	Executes initialization processing that is required before controlling the 12-bit A/D converter module.
R_{Config_S12AD*n*}_Start		Starts the 12-bit AD converter.
R_{Config_S12AD*n*}_Stop		Stops the12-bit AD converter.
R_{Config_S12AD*n*}_Get_ValueResult		Returns 12 bits conversion result in the buffer.
R_{Config_S12AD*n*}_Create_UserInit		Executes user-specific initialization processing for the 12-bit A/D converter.
r_{Config_S12AD*n*}_interrupt		Executes processing in response to INTAD interrupt.

R_{Config_S12ADn}_Create

This API function executes initialization processing that is required before controlling the 12-bit A/D converter module.

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

[Syntax]

```
void    R_{Config_S12ADn}_Create(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Start

This API function starts the 12 Bit A/D converter.

[Syntax]

```
void    R_{Config_S12ADn}_Start(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Stop

This API function stops the 12 Bit A/D converter.

[Syntax]

```
void    R_{Config_S12ADn}_Stop(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Get_ValueResult

This API function returns the 12 bits conversion result in the buffer.

[Syntax]

```
void    R_{Config_S12ADn}_Get_ValueResult (e_ad_channel_t channel, uint16_t * const buffer);
```

Remark *n* is 0.

[Argument(s)]

I/O	Argument(s)	Description
I	e_ad_channel_t channel;	The channel of data register to be read
I	uint16_t * const buffer;	The address where to write the conversion result

Remark Below is shown 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,
 ADCHANNEL15, ADCHANNEL16, ADCHANNEL17, ADCHANNEL18, ADCHANNEL19,
 ADCHANNEL20, ADCHANNEL21, ADCHANNEL22, ADCHANNEL23, ADCHANNEL24,
 ADCHANNEL25, ADCHANNEL26, ADCHANNEL27, ADCHANNEL28, ADCHANNEL29,
 ADCHANNEL30, ADINTERREFVOLT, ADSELFDIAGNOSIS
} e_ad_channel_t;
```

[Return value]

None.

R_{Config_S12ADn}_Create_UserInit

This API function executes user-specific initialization processing for the 12 Bit A/D converter.

Remark This API functions is called from R_{Config_S12ADn}_Create as a callback routine.

[Syntax]

```
void    R_{Config_S12ADn}_Create_UserInit(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_S12ADn}_interrupt

This API function executes processing in response to INTAD interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_S12ADn}_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_S12ADn}_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_S12ADn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting the 12-bit A/D conversion result of single scan mode:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

extern volatile uint8_t interrupt_flag;
uint16_t AD_buffer_0 = 0;
uint16_t AD_buffer_30 = 0;

void main(void)
{
   EI();

   interrupt_flag = 0;
   R_Config_S12AD0_Start();

   while(   interrupt_flag != 1 );
   R_Config_S12AD0_Get_ValueResult (ADCHANNEL0, &AD_buffer_0);
   R_Config_S12AD0_Get_ValueResult (ADCHANNEL30, &AD_buffer_30);
   interrupt_flag = 2;

   while(1);
}
```

Config_S12AD0_user.c

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

static void __near r_Config_S12AD0_interrupt(void)
{
       /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
        interrupt_flag = 1;
       /* End user code. Do not edit comment generated here */
}
```

4.2.39 12 Bit A/D Continuous Scan

Below is a list of API functions output by the Smart Configurator for 12 Bit A/D Continuous Scan use.

Table 4-44 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_S12AD*n*}_Create	12-bit A/D converter	Executes initialization processing that is required before controlling the 12-bit A/D converter module.
R_{Config_S12ADn}_Start		Starts the 12-bit AD converter.
R_{Config_S12ADn}_Stop		Stops the12-bit AD converter.
R_{Config_S12ADn}_Get_ValueResult		Returns 12 bits conversion result in the buffer.
R_{Config_S12ADn}_Create_UserInit		Executes user-specific initialization processing for the 12-bit A/D converter.
r_{Config_S12ADn}_interrupt		Executes processing in response to INTAD interrupt.

R_{Config_S12ADn}_Create

This API function executes initialization processing that is required before controlling the 12-bit A/D module.

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

[Syntax]

```
void    R_{Config_S12ADn}_Create(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Start

This API function starts the 12 Bit A/D converter.

[Syntax]

```
void    R_{Config_S12ADn}_Start(void);
```

Remark   *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Stop

This API function stops the 12 Bit A/D converter.

[Syntax]

```
void    R_{Config_S12ADn}_Stop(void);
```

Remark   *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Get_ValueResult

This API function returns the 12 bits conversion result in the buffer.

[Syntax]

```
void    R_{Config_S12ADn}_Get_ValueResult (e_ad_channel_t channel, uint16_t * const buffer);
```

Remark *n* is 0.

[Argument(s)]

I/O	Argument(s)	Description
I	e_ad_channel_t channel;	The channel of data register to be read
I	uint16_t * const buffer;	The address where to write the conversion result

Remark Below is shown 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,
ADCHANNEL15, ADCHANNEL16, ADCHANNEL17, ADCHANNEL18, ADCHANNEL19,
ADCHANNEL20, ADCHANNEL21, ADCHANNEL22, ADCHANNEL23, ADCHANNEL24,
ADCHANNEL25, ADCHANNEL26, ADCHANNEL27, ADCHANNEL28, ADCHANNEL29,
ADCHANNEL30, ADSELFDIAGNOSIS
} e_ad_channel_t;
```

[Return value]

None.

R_{Config_S12ADn}_Create_UserInit

This API function executes user-specific initialization processing for the 12 Bit A/D converter.

Remark This API functions is called from R_{Config_S12ADn}_Create as a callback routine.

[Syntax]

```
void    R_{Config_S12ADn}_Create_UserInit(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_S12ADn}_interrupt

This API function executes processing in response to INTAD interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_S12ADn}_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_S12ADn}_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_S12ADn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting the 12-bit A/D conversion result of continuous scan mode by a software trigger input:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

extern volatile uint8_t interrupt_flag;
uint16_t AD_buffer_3 = 0;
uint16_t AD_buffer_28 = 0;
uint8_t continues_num = 0;

void main(void)
{
   EI();

   interrupt_flag = 0;
   continues_num = 0;
   R_Config_S12AD0_Start();
   R_Config_TAU0_0_Start();
   R_Config_TAU0_0_Set_SoftwareTriggerOn();

   while( 1 ){
        if( interrupt_flag == 1 ){
             interrupt_flag = 0;
             continues_num ++;
              R_Config_S12AD0_Get_ValueResult (ADCHANNEL3, &AD_buffer_3);
              R_Config_S12AD0_Get_ValueResult (ADCHANNEL28, &AD_buffer_28);

               if( continues_num >=3 ){
                     R_Config_S12AD0_Stop();
                     break;
               }
        }
   };
   interrupt_flag = 2;

   while(1);
}
```

Config_S12AD0_user.c

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

static void _near r_Config_S12AD0_interrupt(void)
{

    /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
    interrupt_flag = 1;
    /* End user code. Do not edit comment generated here */
}
```

4.2.40 12 Bit A/D Group Scan

Below is a list of API functions output by the Smart Configurator for 12 Bit A/D Group Scan use.

Table 4-45 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_S12AD*n*}_Create	12-bit A/D converter	Executes initialization processing that is required before controlling the 12-bit A/D converter module.
R_{Config_S12AD*n*}_Start		Starts the 12-bit AD converter.
R_{Config_S12AD*n*}_Stop		Stops the12-bit AD converter.
R_{Config_S12AD*n*}_Get_ValueResult		Returns 12 bits conversion result in the buffer.
R_{Config_S12AD*n*}_Create_UserInit		Executes user-specific initialization processing for the 12-bit A/D converter.
r_{Config_S12AD*n*}_interrupt		Executes processing in response to INTAD interrupt.
r_{Config_S12AD*n*}_groupb_interrupt		Executes processing in response to INTADGB interrupt.

R_{Config_S12ADn}_Create

This API function executes initialization processing that is required before controlling the 12-bit A/D module.

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

[Syntax]

```
void    R_{Config_S12ADn}_Create(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Start

This API function starts the 12 Bit A/D converter.

[Syntax]

```
void    R_{Config_S12ADn}_Start(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Stop

This API function stops the 12 Bit A/D converter.

[Syntax]

```
void        R_{Config_S12ADn}_Stop(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

R_{Config_S12ADn}_Get_ValueResult

This API function returns the 12 bits conversion result in the buffer.

[Syntax]

```
void    R_{Config_S12ADn}_Get_ValueResult (e_ad_channel_t channel, uint16_t * const buffer);
```

Remark *n* is 0.

[Argument(s)]

I/O	Argument(s)	Description
I	e_ad_channel_t channel;	The channel of data register to be read
I	uint16_t * const buffer;	The address where to write the conversion result

Remark Below is shown 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,
ADCHANNEL15, ADCHANNEL16, ADCHANNEL17, ADCHANNEL18, ADCHANNEL19,
ADCHANNEL20, ADCHANNEL21, ADCHANNEL22, ADCHANNEL23,
ADCHANNEL24, ADCHANNEL25, ADCHANNEL26, ADCHANNEL27, ADCHANNEL28,
ADCHANNEL29, ADCHANNEL30, ADINTERREFVOLT, ADSELFDIAGNOSIS
} e_ad_channel_t;
```

[Return value]

None.

R_{Config_S12ADn}_Create_UserInit

This API function executes user-specific initialization processing for the 12 Bit A/D converter.

Remark This API functions is called from R_{Config_S12ADn}_Create as a callback routine.

[Syntax]

```
void    R_{Config_S12ADn}_Create_UserInit(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_S12ADn}_interrupt

This API function executes processing in response to INTAD interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_S12ADn}_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_S12ADn}_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_S12ADn}_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

r_{Config_S12ADn}_ groupb_interrupt

This API function executes processing in response to INTADGB interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_S12ADn}_groupb_interrupt(void);
```

For LLVM toolchain:

```
void      r_{Config_S12ADn}_groupb_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_S12ADn}_groupb_interrupt(void);
```

Remark *n* is 0.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting the 12-bit A/D conversion result of group scan mode by a synchronous trigger from the timer function:
(Blue code is user code.)

main.c

```
#include "r_cg_macrodriver.h"

void main(void);

extern volatile uint8_t interrupt_flag_GA; //    Group A interrupt flag
extern volatile uint8_t interrupt_flag_GB; //    Group B interrupt flag
uint16_t GB_ANI0_1;    // Group B, ANI0 buf, first reading
uint16_t GB_ANI1_1;
uint16_t GB_ANI2_1;
uint16_t GA_ANI3_1;    // Group A, ANI3 buf, first reading
uint16_t GA_ANI4_1;
uint16_t GB_ANI0_2;    // Group B, ANI0 buf, second reading
uint16_t GB_ANI1_2;
uint16_t GB_ANI2_2;
uint16_t GA_ANI3_2;    // Group A, ANI3 buf, second reading
uint16_t GA_ANI4_2;
uint16_t GB_ANI0_3;    // Group B, ANI0 buf, third reading
uint16_t GB_ANI1_3;
uint16_t GB_ANI2_3;
uint16_t GA_ANI3_3;    // Group A, ANI3 buf, third reading
uint16_t GA_ANI4_3;

void main(void)
{
        EI();
        RAMSAR = 0x9F;

        interrupt_flag_GA = 0;
        interrupt_flag_GB = 0;

        GB_ANI0_1 = 0;    // Group B, ANI0 buf, first reading
        GB_ANI1_1 = 0;
        GB_ANI2_1 = 0;
        GA_ANI3_1 = 0;    // Group A, ANI3 buf, first reading
        GA_ANI4_1 = 0;
        GB_ANI0_2 = 0;    // Group B, ANI0 buf, second reading
        GB_ANI1_2 = 0;
        GB_ANI2_2 = 0;
        GA_ANI3_2 = 0;    // Group A, ANI3 buf, second reading
        GA_ANI4_2 = 0;

        R_Config_S12AD0_Start();
        R_Config_TAU0_0_Start();
        R_Config_TAU0_4_Start();
        R_Config_TRD0_Start();

        R_Config_TAU0_4_Set_SoftwareTriggerOn();    // trigger Group B
        // input 1v
        while(1){
                // state 1
                if(interrupt_flag_GB == 1){    // first reading, when Group B triggered in
                        // Group B (Group B conversion completed)
                        R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, &GB_ANI0_1);
```

```
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, &GB_ANI1_1);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL2, &GB_ANI2_1);
            // Group A (Group A not start)
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL3, &GA_ANI3_1);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL4, &GA_ANI4_1);
            interrupt_flag_GB = 2;    // read once,    Group B conversion completed

            R_Config_TAU0_0_Set_SoftwareTriggerOn();    // trigger Group A
        }
        // state 2
        if(interrupt_flag_GA == 1){ // second reading
            // Group B (Group B not start)
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, &GB_ANI0_2);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, &GB_ANI1_2);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL2, &GB_ANI2_2);
            // Group A (Group A start)
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL3, &GA_ANI3_2);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL4, &GA_ANI4_2);
            interrupt_flag_GA = 2;    // read once,    Group A conversion completed
        }

        if((interrupt_flag_GA == 2) && (interrupt_flag_GB == 2)){
            break;    // Complete all conversions.
        }
    }
    R_Config_S12AD0_Stop();    // stop conversion
    GB_ANI0_3 = 0;             // wait for restart
    GB_ANI1_3 = 0;
    GB_ANI2_3 = 0;
    GA_ANI3_3 = 0;
    GA_ANI4_3 = 0;
    interrupt_flag_GA = 0;
    interrupt_flag_GB = 0;     // state 3
    // input 0.6v
    R_Config_S12AD0_Start(); // Group scan restart
    R_Config_TAU0_0_Set_SoftwareTriggerOn();    // trigger Group A again

    while(interrupt_flag_GA != 1);
    // Group B (Group B not start)
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, &GB_ANI0_3);
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, &GB_ANI1_3);
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL2, &GB_ANI2_3);
    // Group A (Group A start)
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL3, &GA_ANI3_3);
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL4, &GA_ANI4_3);

    while(1);
}
```

Config_S12AD0_user.c

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

static void __near r_Config_S12AD0_interrupt(void)
{
        /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
        interrupt_flag_GA = 1;
        /* End user code. Do not edit comment generated here */
}
static void __near r_Config_S12AD0_groupb_interrupt(void)
{
        /* Start user code for r_Config_S12AD0_groupb_interrupt. Do not edit comment generated here */
        interrupt_flag_GB = 1;
        /* End user code. Do not edit comment generated here */
}
```

4.2.41 D/A Converter

Below is a list of API functions output by the Smart Configurator for D/A converter use.

Table 4-46 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_DACn}_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"

void main(void);

void main(void)
{
        EI();
        R_Config_DAC0_Set_ConversionValue(0xF0);
        R_Config_DAC0_Start();

}
```

4.2.42 Data Transfer Controller

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

Table 4-47 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_DTCD*n*}_Start		Starts DTCD*n* module operation.
R_{Config_DTCD*n*}_Stop		Stops DTCD*n* 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_DTCD*n*}_Start

This API function starts DTCD*n* module operation

[Syntax]

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

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_DTCD*n*}_Stop

This API function stops DTCD*n* module operation

[Syntax]

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

Remark *n* is the channel number.

[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"

void main(void);

void main(void)
{
        EI();
        R_DTCD0_Start();
        R_Config_RTC_Start();

        while(dtc_controldata_0.dtcct != 0);

        R_Config_RTC_Stop();
        R_DTCD0_Stop();
}
```

4.2.43 Comparator

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

Table 4-48 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 INTCMP*n* 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 or R_PGACOMP_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 INTCMPn interrupt.

Remark This API function is called as the interrupt handler for comparator interrupts, which occur when an active edge of the comparator output is detected.

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_{Config_COMPn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;

void main(void);

void main(void)
{
        EI();
        comp0_trig_flag = 0U;
        /* Start comparator 0 */
        R_Config_COMP0_Start ();
        while(comp0_trig_flag != 1U);
        comp1_trig_flag = 0U;
        R_Config_COMP0_Stop ();
}
```

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 */
}
```

4.2.44 Programmable Gain Amplifier

Below is a list of API functions output by the Smart Configurator for programmable gain amplifier use.

Table 4-49 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_PGA}_Create	Programmable Gain Amplifier	Executes initialization processing that is required before controlling the PGA module.
R_{Config_PGA}_Start		Starts the PGA.
R_{Config_PGA}_Stop		Stops the PGA.
R_{Config_PGA}_Create_UserInit		Executes user-specific initialization processing for the PGA.

R_{Config_PGA}_Create

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

Remark This API function is called from R_PGACOMP_Create.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_PGA}_Start

This API function starts the PGA.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_PGA}_Stop

This API function stops the PGA.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_PGA}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for outputting an amplifying signal when inputing a signal:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
        EI();
        /* Start comparator 0 */
        R_Config_PGA_Start ();
        while(1U);
}
```

4.2.45 SPI (CSI) Communication

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

Table 4-50 API Functions:

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

R_{Config_CSI*p*}_Create

This API function executes initialization processing that is required before controlling the CSI*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, 30, 31.

[Syntax]

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

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

R_{Config_CSI*p*}_Start

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

[Syntax]

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

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

R_{Config_CSI*p*}_Stop

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

[Syntax]

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

Remark   *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

R_{Config_CSI*p*}_Send

This API function sends CSI*p* data.

[Syntax]

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

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[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_CSI*p*}_Receive

This API function receives CSIp data.

[Syntax]

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

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[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_CSI*p*}_Send_Receive

Sends and receives CSI*p* data.

[Syntax]

```
MD_STATUS R_{Config_CSIp}_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, 30, 31.

[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_CSI*p*}_Create_UserInit

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

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

[Syntax]

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

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

r_{Config_CSI*p*}_interrupt

This API function executes processing in response to transfer end interrupt/buffer empty interrupt (INTCSI*p*).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_CSIp}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_CSIp}_interrupt(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

r_{Config_CSI*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_CSI*p*}_interrupt corresponding to the CSI*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_CSIp}_callback_sendend(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

r_{Config_CSI*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_CSI*p*}_interrupt corresponding to the CSI*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_CSIp}_callback_receiveend(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

r_{Config_CSI*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_CSI*p*}_interrupt corresponding to the CSI*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_CSIp}_callback_error(uint8_t err_type);
```

Remark *p* is 00, 01, 10, 11, 20, 21, 30, 31.

[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 CSI00 send data and CSI11 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;

void main(void);

void main(void)
{
	EI();
	R_Config_CSI11_Start();
	R_Config_CSI00_Start();
	R_Config_CSI11_Receive(rx_buf, sizeof(rx_buf));
	R_Config_CSI00_Send(tx_buf, sizeof(tx_buf));
	while(transmitend_flag != 1U);
	transmitend_flag = 0U;
	while(receiveend_flag != 1U);
	receiveend_flag = 0U;
	R_Config_CSI00_Stop();
	R_Config_CSI11_Stop();
}
```

Config_CSI00_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_CSI00_callback_sendend(void)
{
	/* Start user code for r_Config_CSI00_callback_sendend. Do not edit comment generated here */
	transmitend_flag = 1U;
	/* End user code. Do not edit comment generated here */
}
```

Config_CSI11_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_CSI11_callback_receiveend (void)
{
	/* Start user code for r_Config_CSI11_callback_receiveend. Do not edit comment generated here */
	receiveend_flag = 1U;
	/* End user code. Do not edit comment generated here */
}
```

4.2.46 UART Communication (Serial array unit)

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

Table 4-51 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, 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTq}_Start

This API function starts UART*q* module operation.

[Syntax]

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

Remark *q* is 0, 1, 2, 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UART*q*}_Stop

This API function stops UART*q* module operation.

[Syntax]

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

Remark   *q* is 0, 1, 2, 3.

[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, 3.

[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, 3.

[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, 3.

[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, 3.

[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, 3.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTq}_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]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

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

Remark *q* is 0, 1, 2, 3.

[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]

For CCRL78 toolchain:

```
static void __near      r_{Config_UARTq}_interrupt_receive(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_UARTq}_interrupt_receive(void);
```

Remark *q* is 0, 1, 2, 3.

[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]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

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

Remark *q* is 0, 1, 2, 3.

[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, 3.

[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, 3.

[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, 3.

[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, 3.

[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;

void main(void);

void main(void)
{
        EI();
        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 */
}
```

4.2.47 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 4-52 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 (INTUT*n*).
r_{Config_UARTA*n*}_interrupt_receive		Executes processing in response to UARTA*n* reception transfer end interrupt (INTUR*n*).
r_{Config_UARTA*n*}_interrupt_error		Executes processing in response to UARTA*n* reception communication error interrupt (INTURE*n*).
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 (INTUTn).

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_UARTAn}_interrupt_send(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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 (INTURn).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_UARTAn}_interrupt_receive(void);
```

For LLVM toolchain:

```
void      r_{Config_UARTAn}_interrupt_receive(void);
```

For IAR toolchain:

```
__interrupt static 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 (INTUREn).

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_{Config_UARTAn}_interrupt_error(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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;
void main(void);

void main(void)
{
        EI();
        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();
}
```

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 */
}
```

4.2.48 UART Communication (LIN/UART module)

Below is a list of API functions output by the Smart Configurator for UART Communication (LIN/UART module) use.

Table 4-53 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_RLIN3*n*}_Create	LIN/UART module	Executes initialization processing that is required before controlling the RLIN3*n* module.
R_{Config_RLIN3*n*}_Start		Starts RLIN3*n* module operation.
R_{Config_RLIN3*n*}_Stop		Stops RLIN3*n* module operation.
R_{Config_RLIN3*n*}_Send		Sends RLIN3*n* data.
R_{Config_RLIN3*n*}_Receive		Receives RLIN3*n* data.
R_{Config_RLIN3*n*}_Create_UserInit		Executes user-specific initialization processing for the RLIN3*n*.
r_{Config_RLIN3*n*}_interrupt_send		Executes processing in response to RLIN3*n* transmit end interrupt (in single-transfer mode) or buffer empty interrupt (in continuous transfer mode).
r_{Config_RLIN3*n*}_interrupt_receive		Executes processing in response to RLIN3*n* receive end interrupt.
r_{Config_RLIN3*n*}_interrupt_error		Executes processing in response to RLIN3*n* error interrupt.
r_{Config_RLIN3*n*}_callback_sendend		Executes processing in response to transmit end interrupt.
r_{Config_RLIN3*n*}_callback_receiveend		Executes processing in response to receive end interrupt.
r_{Config_RLIN3*n*}_callback_error		Executes processing in response to receive error interrupt.

R_{Config_RLIN3n}_Create

This API function executes initialization processing that is required before controlling the RLIN3*n* module.

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

[Syntax]

```
void    R_{Config_RLIN3n}_Create(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_RLIN3n}_Start

This API function starts RLIN3*n* module operation.

[Syntax]

```
void    R_{Config_RLIN3n}_Start(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_RLIN3n}_Stop

This API function stops RLIN3*n* module operation.

[Syntax]

```
void    R_{Config_RLIN3n}_Stop(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

R_{Config_RLIN3n}_Send

This API function sends RLIN3*n* data.

[Syntax]

```
MD_STATUS    R_{Config_RLIN3n}_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *n* is 0, 1.

[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_RLIN3n}_Receive

This API function receives RLIN3*n* data.

[Syntax]

```
MD_STATUS    R_{Config_RLIN3n}_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *n* is 0, 1.

[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_RLIN3n}_Create_UserInit

This API function executes user-specific initialization processing for the RLIN3*n*.

Remark This API functions is called from R_{Config_RLIN3n}_Create as a callback routine.

[Syntax]

```
void    R_{Config_RLIN3n}_Create_UserInit(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_RLIN3n}_interrupt_send

This API function executes processing in response to RLIN3*n* transmit end interrupt (in single-transfer mode) or buffer empty interrupt (in continuous transfer mode).

[Syntax]

For CCRL78 toolchain:

```
static void __near r_{Config_RLIN3n}_interrupt_send(void);
```

For LLVM toolchain:

```
void r_{Config_RLIN3n}_interrupt_send(void);
```

For IAR toolchain:

```
__interrupt static void r_{Config_RLIN3n}_interrupt_send(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_RLIN3n}_interrupt_receive

This API function executes processing in response to RLIN3*n* receive end interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_RLIN3n}_interrupt_receive(void);
```

For LLVM toolchain:

```
void      r_{Config_RLIN3n}_interrupt_receive(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_RLIN3n}_interrupt_receive(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_RLIN3n}_interrupt_error

This API function executes processing in response to RLIN3*n* error interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_RLIN3n}_interrupt_error(void);
```

For LLVM toolchain:

```
void      r_{Config_RLIN3n}_interrupt_error(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_RLIN3n}_interrupt_error(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_RLIN3n}_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_RLIN3n}_interrupt_send corresponding to the RLIN3*n* 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_RLIN3n}_callback_sendend(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_RLIN3n}_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_RLIN3n}_interrupt_receive corresponding to the RLIN3*n* 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_RLIN3n}_callback_receiveend(void);
```

Remark *n* is 0, 1.

[Argument(s)]

None.

[Return value]

None.

r_{Config_RLIN3n}_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_RLIN3n}_interrupt_error corresponding to the RLIN3*n* 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_RLIN3n}_callback_error(uint8_t err_type);
```

Remark *n* is 0, 1.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t err_type;	Error type info Bit2: Overrun error Bit3: Framing error Bit4: Expansion bit detection flag Bit5: ID match flag Bit6: Parity error Bit0, 1, 7: 0

[Return value]

None.

Usage example

This is an example for RLIN30 receive data and RLIN31 send data:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf0[] = {0x7a, 0x85, 0xbc, 0x26, 0x01, 0x4f};
uint8_t rx_buf0[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
volatile uint8_t sendend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void);

void main(void)
{
	EI();

	R_Config_RLIN30_Start();
	R_Config_RLIN31_Start();

	sendend_flag = 0U;
	receiveend_flag = 0U;
	R_Config_RLIN30_Receive(rx_buf0, sizeof(rx_buf0));
	R_Config_RLIN31_Send(tx_buf0, sizeof(tx_buf0));
	while((1U != sendend_flag) || (1U != receiveend_flag));

	R_Config_RLIN30_Stop();
	R_Config_RLIN31_Stop();
	while(1);
}
```

Config_RLIN31_user.c

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

static void r_Config_RLIN31_callback_sendend(void)
{
	/* Start user code for r_Config_RLIN31_callback_sendend. Do not edit comment generated here */
	sendend_flag = 1U;
	/* End user code. Do not edit comment generated here */
}
```

Config_RLIN30_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_RLIN30_callback_receiveend (void)
{
	/* Start user code for r_Config_RLIN30_callback_receiveend. Do not edit comment generated here
*/
	receiveend_flag = 1U;
	/* End user code. Do not edit comment generated here */
}
```

4.2.49 DALI Communication (Control devices)

Below is a list of API functions output by the Smart Configurator for DALI communication (Control devices) use.

Table 4-54 API Functions: (1/2)

API Function Name	Peripheral Name	Description
R_{Config_DALI}_Create	DALI	Executes initialization processing that is required before controlling the DALI communication (Control devices).
R_{Config_DALI}_Start		Starts DALI communication (Control devices) operation.
R_{Config_DALI}_Stop		Stops DALI communication (Control devices) operation.
R_{Config_DALI}_SoftwareReset		Resets DALI communication (Control devices) operation.
R_{Config_DALI}_EnableForceActiveState		Enable DALITxD0 assertion and assertion level (active state) is low. The output from the DALITxD0 pin is driven low.
R_{Config_DALI}_DisableForceActiveState		Disable DALITxD0 assertion. Internal data for transmission are output from the DALITxD0 pin.
R_{Config_DALI}_GetStatus		Gets the state of the DALI communication (Control devices).
R_{Config_DALI}_Send		Sends frame data.
R_{Config_DALI}_GetReceiveFrame		Receives frame data and frame length.
R_{Config_DALI}_Create_UserInit		Executes user-specific initialization processing for the DALI communication (Control devices).
r_{Config_DALI}_interrupt_send		Executes processing in response to DALI communication (Control devices) transmission 32bit-data completion interrupt (INTTD).
r_{Config_DALI}_interrupt_receive		Executes processing in response to DALI communication (Control devices) reception 32bit-data completion interrupt (INTRD).
r_{Config_DALI}_interrupt_error		Executes processing in response to DALI communication (Control devices) reception communication error interrupt (INTED).

Table 4-55 API Functions: (2/2)

API Function Name	Peripheral Name	Description
r_{Config_DALI}_interrupt_falling_edge_detection		Executes processing in response to DALI communication (Control devices) falling edge detection interrupt (INTFED).
r_{Config_DALI}_interrupt_power_down_detection		Executes processing in response to DALI communication (Control devices) power down detection interrupt (INTBPD).
r_{Config_DALI}_interrupt_collision_detection		Executes processing in response to DALI communication (Control devices) collision detection interrupt (INTCLD).
r_{Config_DALI}_interrupt_stop_bit_detection		Executes processing in response to DALI communication (Control devices) stop bit detection interrupt (INTSDD).
r_{Config_DALI}_callback_sendend		Executes processing in response to DALI communication (Control devices) stop bit detection interrupt.
r_{Config_DALI}_callback_receiveend		Executes processing in response to DALI communication (Control devices) stop bit detection interrupt.
r_{Config_DALI}_callback_error		Executes processing in response to DALI communication (Control devices) reception communication error interrupt.

R_{Config_DALI}_Create

This API function executes initialization processing that is required before controlling the DALI communication (Control devices).

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_Start

This API function starts DALI communication (Control devices) operation.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_Stop

This API function stops DALI communication (Control devices) operation.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_SoftwareReset

This API function resets DALI communication (Control devices) operation.

[Syntax]

```
void    R_{Config_DALI}_SoftwareReset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_EnableForceActiveState

This API function enable DALITxD0 assertion and assertion level (active state) is low. The output from the DALITxD0 pin is driven low.

[Syntax]

```
void    R_{Config_DALI}_EnableForceActiveState(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_DisableForceActiveState

This API function disable DALITxD0 assertion. Internal data for transmission are output from the DALITxD0 pin.

[Syntax]

```
void    R_{Config_DALI}_DisableForceActiveState(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_GetStatus

This API function gets the state of the DALI communication (Control devices).

[Syntax]

```
void    R_{Config_DALI}_GetStatus(uint16_t * const status);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t * const status;	DALI status register buffer pointer

[Return value]

None

R_{Config_DALI}_Send

This API function sends frame data. The frame length is set in the GUI and it is a fixed value.

[Syntax]

```
void    R_{Config_DALI}_Send(uint16_t * const tx_buf);
```

[Argument(s)]

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

Remark Below is shown the relationship between frame length and tx_buf length.

Frame length	tx_buf[] length
8 bits	1
16 bits	1
17 bits	2
20 bits	2
24 bits	2
32 bits	2
64 bits	4
128 bits	8
256 bits	16

[Return value]

None

R_{Config_DALI}_GetReceivedFrame

This API function receives frame data and frame length.

[Syntax]

```
MD_STATUS R_{Config_DALI}_GetReceivedFrame(uint32_t * const rx_buf, uint16_t * const rx_num);
```

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * const rx_buf;	Receive buffer pointer
I	uint16_t * const rx_num;	Buffer frame length

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error

R_{Config_DALI}_Create_UserInit

This API function executes user-specific initialization processing for the DALI communication (Control devices).

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_send

This API function executes processing in response to DALI communication (Control devices) transmission each 32bit-data completion interrupt (INTTD).

Remark This API function is only useful when transmission data length is larger then 32bit.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_DALI}_interrupt_send(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_DALI}_interrupt_send(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_receive

This API function executes processing in response to DALI communication (Control devices) reception each 32bit-data completion interrupt (INTRD).

Remark This API function is only useful when reception data length is larger then 32bit.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_receive(void);
```

For LLVM toolchain:

```
void      r_{Config_DALI}_interrupt_receive(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_receive(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_error

This API function executes processing in response to DALI communication (Control devices) reception communication error interrupt (INTED).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_error(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_error(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_falling_edge_detection

This API function executes processing in response to DALI communication (Control devices) falling edge detection interrupt (INTFED).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

For LLVM toolchain:

```
void      r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_power_down_detection

This API function executes processing in response to DALI communication (Control devices) power down detection interrupt (INTBPD).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_power_down_detection(void);
```

For LLVM toolchain:

```
void      r_{Config_DALI}_interrupt_power_down_detection(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_power_down_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_collision_detection

This API function executes processing in response to DALI communication (Control devices) collision detection interrupt (INTCLD).

[Syntax]

For CCRL78 toolchain:

```
static void __near	r_{Config_DALI}_interrupt_collision_detection(void);
```

For LLVM toolchain:

```
void	r_{Config_DALI}_interrupt_collision_detection(void);
```

For IAR toolchain:

```
__interrupt static void	r_{Config_DALI}_interrupt_collision_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_stop_bit_detection

This API function executes processing in response to DALI communication (Control devices) stop bit detection interrupt (INTSDD).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

For LLVM toolchain:

```
void      r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_callback_sendend

This API function executes processing in response to DALI communication (Control devices) stop bit detection interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_DALI}_interrupt_stop_bit_detection corresponding to the DALI communication (Control devices) stop bit detection 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_DALI}_callback_sendend(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_callback_receiveend

This API function executes processing in response to DALI communication (Control devices) stop bit detection interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_DALI}_interrupt_stop_bit_detection corresponding to the DALI communication (Control devices) stop bit detection 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_DALI}_callback_receiveend(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_callback_error

This API function executes processing in response to DALI communication (Control devices) reception communication error interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_DALI}_interrupt_error corresponding to the DALI communication (Control devices) 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_DALI}_callback_error(uint16_t err_type);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t err_type;	Error type value: Bit0: Manchester framing error Bit1: Overrun error Bit2: Frame size violation error Bit3: Bit timing violation error Bit4 to Bit7: 0

[Return value]

None.

Usage example

This is an example for DALI communication (Control devices) transmit a 16bit frame data and DALI communication (Control gear) receieve data:
(Blue code is user code.)

main.c for DALI communication (Control devices)

```
#include "r_smc_entry.h"

uint16_t tx_buf0[] = {0xFF66};
volatile uint8_t sendend_flag = 0U;
void main(void);

void main(void)
{
        EI();
        R_Config_DALI_Device_Start();
        R_Config_DALI_Send(tx_buf0);
        while(1U != sendend_flag);
        R_Config_DALI_Device_Stop();
}
```

main.c for DALI communication (Control gear)

```
#include "r_smc_entry.h"

uint8_t rx_buf0[];
uint8_t rx_buf1[100];
uint8_t p_rx_num = 0U;
uint8_t rx_num = 0U;
volatile uint8_t receiveend_flag = 0U;
void main(void);

void main(void)
{
        EI();
        R_Config_DALI_Gear_Start();
        while(1U != receiveend_flag);
        if(R_Config_DALI1_GetReceivedFrame(&rx_buf0,&p_rx_num) == MD_OK)
        {
                rx_buf1[rx_num]= rx_buf0;
                rx_num++;
        }

        R_Config_ DALI1_Gear_Stop();
}
```

Config_DALI_Device_user.c

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

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

Config_DALI1_Gear_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_DALI_Gear_callback_receiveend (void)
{
    /* Start user code for r_Config_DALI_Gear_callback_receiveend. Do not edit comment generated
here */
    receiveend_flag++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.50 DALI Communication (Control gear)

Below is a list of API functions output by the Smart Configurator for DALI communication (Control gear) use.

Table 4-56 API Functions: (1/2)

API Function Name	Peripheral Name	Description
R_{Config_DALI}_Create	DALI	Executes initialization processing that is required before controlling the DALI communication (Control gear).
R_{Config_DALI}_Start		Starts DALI communication (Control gear) operation.
R_{Config_DALI}_Stop		Stops DALI communication (Control gear) operation.
R_{Config_DALI}_SoftwareReset		Resets DALI communication (Control gear) operation.
R_{Config_DALI}_EnableForceActiveState		Enable DALITxD0 assertion and assertion level (active state) is low. The output from the DALITxD0 pin is driven low.
R_{Config_DALI}_DisableForceActiveStat e		Disable DALITxD0 assertion. Internal data for transmission are output from the DALITxD0 pin.
R_{Config_DALI}_GetStatus		Gets the state of the DALI communication (Control gear).
R_{Config_DALI}_Send		Sends frame data.
R_{Config_DALI}_GetReceiveFrame		Receives frame data and frame length.
R_{Config_DALI}_Create_UserInit		Executes user-specific initialization processing for the DALI communication (Control gear).
r_{Config_DALI}_interrupt_error		Executes processing in response to DALI communication (Control gear) reception communication error interrupt (INTED).

Table 4-57 API Functions: (2/2)

API Function Name	Peripheral Name	Description
r_{Config_DALI}_interrupt_falling_edge_detection		Executes processing in response to DALI communication (Control gear) falling edge detection interrupt (INTFED).
r_{Config_DALI}_interrupt_power_down_detection		Executes processing in response to DALI communication (Control gear) power down detection interrupt (INTBPD).
r_{Config_DALI}_interrupt_stop_bit_detection		Executes processing in response to DALI communication (Control gear) stop bit detection interrupt (INTSDD).
r_{Config_DALI}_callback_sendend		Executes processing in response to DALI communication (Control gear) stop bit detection interrupt.
r_{Config_DALI}_callback_receiveend		Executes processing in response to DALI communication (Control gear) stop bit detection interrupt.
r_{Config_DALI}_callback_error		Executes processing in response to DALI communication (Control gear) reception communication error interrupt.

R_{Config_DALI}_Create

This API function executes initialization processing that is required before controlling the DALI communication (Control gear).

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_Start

This API function starts DALI communication (Control gear) operation.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_Stop

This API function stops DALI communication (Control gear) operation.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_SoftwareReset

This API function resets DALI communication (Control gear) operation.

[Syntax]

```
void    R_{Config_DALI}_SoftwareReset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_EnableForceActiveState

This API function enable DALITxD0 assertion and assertion level (active state) is low. The output from the DALITxD0 pin is driven low.

[Syntax]

```
void    R_{Config_DALI}_EnableForceActiveState(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_DisableForceActiveState

This API function disable DALITxD0 assertion. Internal data for transmission are output from the DALITxD0 pin.

[Syntax]

```
void    R_{Config_DALI}_DisableForceActiveState(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DALI}_GetStatus

This API function gets the state of the DALI communication (Control gear).

[Syntax]

```
void     R_{Config_DALI}_GetStatus(uint16_t * const status);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t * const status;	DALI status register buffer pointer

[Return value]

None

R_{Config_DALI}_Send

This API function sends frame data. The frame length is fixed to 8 bits.

Remark This API function sets the data from the buffer specified in argument *tx_buf* to register TDR1L. And enter r_{Config_DALI}_interrupt_stop_bit_detection when register TDR1L transmission is completed.

[Syntax]

```
void      R_{Config_DALI}_Send(uint8_t tx_buf);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t tx_buf;	Transfer buffer

[Return value]

None

R_{Config_DALI}_GetReceivedFrame

This API function receives frame data and frame length.

[Syntax]

```
MD_STATUS    R_{Config_DALI}_GetReceivedFrame(uint32_t * const rx_buf, uint16_t * const rx_num);
```

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * const rx_buf;	Receive buffer pointer
I	uint16_t * const rx_num;	Buffer frame length

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error

R_{Config_DALI}_Create_UserInit

This API function executes user-specific initialization processing for the DALI communication (Control gear).

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_error

This API function executes processing in response to DALI communication (Control gear) reception communication error interrupt (INTED).

[Syntax]

For CCRL78 toolchain:

```
static void __near       r_{Config_DALI}_interrupt_error(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void       r_{Config_DALI}_interrupt_error(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_falling_edge_detection

This API function executes processing in response to DALI communication (Control gear) falling edge detection interrupt (INTFED).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

For LLVM toolchain:

```
void      r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_power_down_detection

This API function executes processing in response to DALI communication (Control gear) power down detection interrupt (INTBPD).

[Syntax]

For CCRL78 toolchain:

```
static void __near	r_{Config_DALI}_interrupt_power_down_detection(void);
```

For LLVM toolchain:

```
void	r_{Config_DALI}_interrupt_power_down_detection(void);
```

For IAR toolchain:

```
__interrupt static void	r_{Config_DALI}_interrupt_power_down_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_interrupt_stop_bit_detection

This API function executes processing in response to DALI communication (Control gear) stop bit detection interrupt (INTSDD).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

For LLVM toolchain:

```
void      r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

For IAR toolchain:

```
__interrupt static void      r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_callback_sendend

This API function executes processing in response to DALI communication (Control gear) stop bit detection interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_DALI}_interrupt_stop_bit_detection corresponding to the DALI communication (Control devices) stop bit detection 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_DALI}_callback_sendend(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_callback_receiveend

This API function executes processing in response to DALI communication (Control gear) stop bit detection interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_DALI}_interrupt_stop_bit_detection corresponding to the DALI communication (Control devices) stop bit detection 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_DALI}_callback_receiveend(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_DALI}_callback_error

This API function executes processing in response to DALI communication (Control gear) reception communication error interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_DALI}_interrupt_error corresponding to the DALI communication (Control gear) 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_DALI}_callback_error(uint16_t err_type);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t err_type;	Error type value: Bit0: Manchester framing error Bit1: Overrun error Bit2: Frame size violation error Bit3: Bit timing violation error Bit4 to Bit7: 0

[Return value]

None.

Usage example

Refer to DALI Communication (Control devices) mode Usage example.

4.2.51 IIC Communication (Master mode) (Serial Array Unit)

Below is a list of API functions output by the Smart Configurator for IIC communication (master mode) (serial array unit) use.

Table 4-58 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 INTIIC*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, 30, 31.

[Syntax]

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

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICr}_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_IICr}_StartCondition(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[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, 30, 31.

[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, 30, 31.

[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_IICr}_Master_Send(uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[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_IICr}_Master_Receive(uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[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, 30, 31.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IIC*r*}_interrupt

This API function executes processing in response to INTIIC*r* transfer end interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_IICr}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void    r_{Config_IICr}_interrupt(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[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_IICr}_callback_master_sendend(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[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, 30, 31.

[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_IICr}_callback_error(MD_STATUS flag);
```

Remark *r* is 00, 01, 10, 11, 20, 21, 30, 31.

[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;

void main(void);

void main(void)
{
        EI();
        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();
}
```

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 */
}
```

4.2.52 IIC Communication (Master mode) (Serial Interface IICA)

Below is a list of API functions output by the Smart Configurator for IIC communication (master mode) (serial interface IICA) use.

Table 4-59 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 (INTIICAn).
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 (INTIICA*n*).

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_IICAn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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_IICAn}_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_IICAn}_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_IICAn}_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;

void main(void);

void main(void)
{
    EI();
    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();
}
```

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 */
}
```

4.2.53 IIC Communication (Slave mode) (Serial Interface IICA)

Below is a list of API functions output by the Smart Configurator for IIC communication (slave mode) (serial interface IICA) use.

Table 4-60 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 (INTIICA*n*).
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 (INTIICA*n*).

[Syntax]

For CCRL78 toolchain:

```
static void __near     r_{Config_IICAn}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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_IICAn}_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_IICAn}_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.

4.2.54 Interrupt Countroller

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

Table 4-61 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_INTC}_Create	Interrupt function	Executes initialization processing that is required before controlling the INTC module.
R_{Config_INTC}_INTP*n*_Start		Clears INTP*n* interrupt flag and enables interrupt.
R_{Config_INTC}_INTP*n*_Stop		Disables INTP*n* interrupt and clears interrupt flag.
R_{Config_INTC}_Create_UserInit		Executes user-specific initialization processing for the INTC module.
r_{Config_INTC}_intp*n*_interrupt		Executes processing in response to INTP*n* interrupt.

R_{Config_INTC}_Create

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_INTC}_INTP*n*_Start

This API function clears INTP*n* interrupt flag and enables interrupt.

[Syntax]

```
void    R_{Config_INTC}_INTPn_Start(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_INTC}_INTP*n*_Stop

This API function disables INTP*n* interrupt and clears interrupt flag.

[Syntax]

```
void    R_{Config_INTC}_INTPn_Stop(void);
```

Remark   *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_INTC}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_INTC}_intp*n*_interrupt

This API function executes processing in response to INTP*n* interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near    r_{Config_INTC}_intpn_interrupt(void);
```

For LLVM toolchain:

```
void    r_{Config_INTC}_intpn_interrupt(void);
```

For IAR toolchain:

```
__interrupt static void    r_{Config_INTC}_intpn_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 INTP0 valid edge input:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t intp0_int_flag;

void main(void);

void main(void)
{
        EI();
        intp0_int_flag = 0U;
        R_Config_INTC_INTP0_Start ();
        while(intp0_int_flag != 1U);
        R_Config_ INTC_INTP0_Stop ();
}
```

Config_INTC_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t intp0_int_flag = 0U
/* End user code. Do not edit comment generated here */

static void __near r_Config_INTC_intp0_interrupt(void)
{
        /* Start user code for r_Config_INTC_intp0_interrupt. Do not edit comment generated here */
        /* Set the flag */
        intp0_int_flag = 1U;
        /* End user code. Do not edit comment generated here */
}
```

4.2.55 Voltage Detector

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

Table 4-62 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_LVD1}_Start		Starts the voltage detector operation.
R_{Config_LVD1}_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_LVD1}_Start

This API function starts the voltage detector operation.

[Syntax]

```
void      R_{Config_LVD1}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_LVD1}_Stop

This API function stops the voltage detector operation.

[Syntax]

```
void     R_{Config_LVD1}_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"

void main(void);

void main(void)
{
        EI();
        R_LVD_Start_Interrupt();
        R_ Config_LVD1_Start();
}
```

r_cg_lvd_common_user.c

```
static void __near r_lvd_interrupt(void)
{
        /* Start user code for r_lvd_interrupt. Do not edit comment generated here */
        /*Clear Flag*/
        DLVD1FCLR = 1U;
        /* End user code. Do not edit comment generated here */
}
```

4.2.56 Snooze Mode Sequencer

Below is a list of API functions output by the Smart Configurator for snooze mode sequencer use.

Table 4-63 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_SMS}_Create	Snooze Mode Sequencer	Executes initialization processing that is required before controlling the SMS module, including configure SMS, copy the SMS instructions, and copy the SMS data.
R_{Config_SMS}_Start		Sets SMS data from arguments and starts SMS module operation.
R_{Config_SMS}_Stop		Stops SMS module operation.
R_{Config_SMS}_GetStatus		Checks SMS wakeup status.
R_{Config_SMS}_GetReturn		Returns SMS data.
R_{Config_SMS}_TriggerWait_Enable		Enables trigger wait operation.
R_{Config_SMS}_TriggerWait_Disable		Disables trigger wait operation.
R_{Config_SMS}_Set_PowerOn		Starts the clock supply for SMS module.
R_{Config_SMS}_Set_PowerOff		Stops the clock supply for SMS module.
R_{Config_SMS}_Set_Reset		Sets SMS module in reset state.
R_{Config_SMS}_Release_Reset		Releases SMS module from reset state.
R_{Config_SMS}_Create_UserInit		Executes user-specific initialization processing for the SMS module.
r_{Config_SMS}_interrupt		Executes processing in response to INTSMSE interrupt.

R_{Config_SMS}_Create

This API function executes initialization processing that is required before controlling the SMS module, including configure SMS, copy the SMS instructions and copy the SMS data.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_Start

This API function sets SMS data from arguments and starts SMS module operation.

[Syntax]

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

```
void    R_{Config_SMS}_Start(uint16_t arg1, uint16_t arg2, ……, uint16_t argn);
```

Remark 1. The arguments of this API function varies according to Start Block setting.
For example, if there are three arguments in Start Block setting, this API will be
R_{Config_SMS}_Start(uint16_t arg1, uint16_t arg2, uint16_t arg3).

Remark 2. $n \leq 14$.

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t arg*n*;	SMS start data (n<=14)

Remark n = 1 - 14.

[Return value]

None.

R_{Config_SMS}_Stop

This API function stops SMS module operation.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_GetStatus

This API function checks SMS wakeup status.

[Syntax]

```
uint8_t     R_{Config_SMS}_GetStatus(void);
```

[Argument(s)]

None.

[Return value]

Macro	Description
uint8_t g_sms_finish_flag	SMS finish flag 0 – SMS not wakeup 1 – SMS wakeup

R_{Config_SMS}_GetReturn

This API function returns SMS data.

[Syntax]

```
void    R_{Config_SMS}_GetReturn(void);
```

```
void    R_{Config_SMS}_GetReturn(uint16_t *p_ret1, uint16_t *p_ret2, ……, uint16_t *p_retn);
```

Remark 1. The arguments of this API function vary according to Wake Up Block setting.
For example, if there are two arguments in Wake Up Block setting, this API will be R_{Config_SMS}_GetReturn(uint16_t *p_ret1, uint16_t *p_ret2).

Remark 2. $n \leq 14$.

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t *p_ret*n*;	SMS data (n<=14)

Remark n = 1 - 14.

[Return value]

None.

R_{Config_SMS}_TriggerWait_Enable

This API function enables trigger wait operation.

[Syntax]

```
void    R_{Config_SMS}_TriggerWait_Enable(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_TriggerWait_Disable

This API function disables trigger wait operation.

[Syntax]

```
void    R_{Config_SMS}_TriggerWait_Disable(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_Set_PowerOn

This API function starts the clock supply for SMS module.

[Syntax]

```
void    R_{Config_SMS}_Set_PowerOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_Set_PowerOff

This API function stops the clock supply for SMS module.

[Syntax]

```
void    R_{Config_SMS}_Set_PowerOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_Set_Reset

This API function sets SMS module in reset state.

[Syntax]

```
void      R_{Config_SMS}_Set_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_Release_Reset

This API function releases SMS module from reset state.

[Syntax]

```
void R_{Config_SMS}_Release_Reset(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_SMS}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_SMS}_interrupt

This API function executes processing in response to INTSMSE interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_SMS}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{Config_SMS}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using SMS to read A/D conversion result:
(Blue code is user code.)

main.c

```
  #include "r_smc_entry.h"

uint8_t g_sms_finish_flag;
uint16_t adc_single_chl = 0x7890;

void main(void);

void main(void)
{
        EI();

        // Set SMS as sequencer stopped state
        R_Config_ SMS_TriggerWait_Enable();
        R_Config_SMS_Start ((uint16_t)&adc_single_chl, (uint16_t)&adc_single_chl);
        // Set SMS as sequencer operating state
        R_Config_ SMS_TriggerWait_Disable();
        STOP ();
        while (g_sms_finish_flag != 1);
        g_sms_finish_flag = 0;
        R_Config_SMS_Stop ();
}
```

Config_SMS_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_ t    g_sms_finish_flag;
/* End user code. Do not edit comment generated here */

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

4.2.57 Key Interrupt

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

Table 4-64 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 INTKR interrupt flag and enables interrupt.
R_{Config_KR}_Stop		Disables INTKR 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 INTKR 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 INTKR 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 INTKR 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 INTKR interrupt.

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

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

[Argument(s)]

None.

[Return value]

None.

Usage example

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

(Blue code is user code.)

main.c

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

void main(void);

void main(void)
{
        EI();
        kr_int_flag = 0U;
        R_Config_KR_Start ();
        while(kr_int_flag != 1U);
        R_Config_ KR_Stop ();
}
```

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 */
}
```

4.2.58 Remote Control Signal Receiver

Below is a list of API functions output by the Smart Configurator for remote control signal receiver use.

Table 4-65 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 INTREMC 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 INTREMC interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{Config_REMC}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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];

void main(void);

void main(void)
{
    EI();
    /* 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);
}
```

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 */
}
```

4.2.59 Watchdog Timer

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

Table 4-66 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 INTWDTI 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 INTWDTI interrupt.

Remark This API function is called as the interrupt handler for maskable interrupts when 75% of the overflow time + 1/4 fIL is reached.

[Syntax]

For CCRL78 toolchain:

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

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static 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"

void main(void);

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

4.2.60 Logic and Event Link Controller

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

Table 4-67 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_*xxx*}_Create	Logic and Event Link Controller	Executes initialization processing that is required before controlling the ELCL module.
R_{Config_*xxx*}_Start		Enables ELCL output.
R_{Config_*xxx*}_Stop		Disables ELCL output.
R_{Config_xxx}_Create_UserInit		Executes user-specific initialization processing for the ELCL.
r_{Config_*xxx*}_interrupt		Executes processing in response to INTELCL interrupt.

Note1. "*xxx*" is ELCL module name.

Note2. r_{Config_*xxx*}_interrupt function is generated only when ELCL output signal used as INTELCL in ELCL function GUI.

R_{Config_*xxx*}_Create

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_*xxx*}_Start

This API function clears ELCL interrupt flag and enables ELCL output.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_*xxx*}_Stop

This API function disables INTELCL interrupt and clears interrupt flag, disable ELCL output.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_{Config_*xxx*}_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

r_{Config_*xxx*}_interrupt

This API function executes processing in response to INTELCL interrupt.

[Syntax]

For CCRL78 toolchain:

```
static void __near      r_{ Config_xxx}_interrupt(void);
```

For LLVM toolchain:

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

For IAR toolchain:

```
__interrupt static void      r_{ Config_xxx}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using ELCL to implement 2 port signal AND logic:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
        EI();

        R_Config_AND_Start();
}
```

4.2.61 Event Link Controller

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

Table 4-68 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}_Stop		Disables ELC output
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}_Stop

This API function stops the ELC event resources.

[Syntax]

```
void    R_{Config_ELC}_Stop(uint32_t event);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint32_t event	Event resources to be stopped (bit n of ELSELRn)

[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 stop “Key return signal detection” event generation:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
        uint32_t stp_event = 0x00000020;
        EI();
        R_Config_ELC_Stop(stp_even);

        while(1);
}
```

Appendix API Function Comparison Table

This chapter compares the API functions which are output by the Code Generator with the API functions which are output by the Smart Configurator. The user who used to use the Code Generator tool can understand which the corresponding API functions are generated by Smart Configurator.

Table 5-1 Code Generator and Smart Configurator API Comparison List (1/6)

Peripheral Function	Code Generator API Function Name	Smart Configurator API Function Name
Common	main	main
	R_MAIN_UserserInit	-
	hdwinit	-
	R_Systeminit	R_Systeminit
	low_level_init	-
	HardwareSetup	-
Clock generator	R_CGC_Create	-
	R_CGC_Set_ClockMode	-
	R_CGC_Create_UserInit	-
	R_CGC_Get_ResetSource	-
Port functions	R_PORT_Create	R_{Config_PORT}_Create
	R_PORT_Create_UserInit	R_{Config_PORT}_Create_UserInit
Timer array unit	R_TAU*m*_Create	R_TAU*m*_Create
	R_TAU*m*_Channel*n*_Start	R_{Config_TAU*m*_*n*}_Start
	R_TAU*m*_Channel*n*_Higher8bits_Start	R_{Config_TAU*m*_*n*}_Higher8bits_Start
	R_TAU*m*_Channel*n*_Lower8bits_Start	R_{Config_TAU*m*_*n*}_Lower8bits_Start
	R_TAU*m*_Channel*n*_Stop	R_{Config_TAU*m*_*n*}_Stop
	R_TAU*m*_Channel*n*_Higher8bits_Stop	R_{Config_TAU*m*_*n*}_Higher8bits_Stop
	R_TAU*m*_Channel*n*_Lower8bits_Stop	R_{Config_TAU*m*_*n*}_Lower8bits_Stop
	R_TAU*m*_Reset	R_TAU*m*_Set_Reset
	R_TAU*m*_Set_PowerOff	R_TAU*m*_Set_PowerOff
	R_TAU*m*_Channel*n*_Get_PulseWidth	R_{Config_TAU*m*_*n*}_Get_PulseWidth
	R_TAU*m*_Channel*n*_Set_SoftwareTriggerOn	R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn
	R_TAU*m*_Create_UserInit	R_{Config_TAU*m*_*n*}_Create_UserInit
	r_taum_channel*n*_interrupt	r_{Config_TAU*m*_*n*}_interrupt
	r_taum_channel*n*_higher8bits_interrupt	r_{Config_TAU*m*_*n*}_higher8bits_interrupt
Real-time clock	R_RTC_Create	R_{Config_RTC}_Create
	R_RTC_Start	R_{Config_RTC}_Start
	R_RTC_Stop	R_{Config_RTC}_Stop
	R_RTC_Set_PowerOff	R_RTC_Set_PowerOff
	R_RTC_Set_HourSystem	R_{Config_RTC}_Set_HourSystem
	R_RTC_Set_CounterValue	R_{Config_RTC}_Set_CounterValue
	R_RTC_Get_CounterValue	R_{Config_RTC}_Get_CounterValue
	R_RTC_Set_ConstPeriodInterruptOn	R_{Config_RTC}_Set_ConstPeriodInterruptOn
	R_RTC_Set_ConstPeriodInterruptOff	R_{Config_RTC}_Set_ConstPeriodInterruptOff
	R_RTC_Set_AlarmOn	R_{Config_RTC}_Set_AlarmOn
	R_RTC_Set_AlarmOff	R_{Config_RTC}_Set_AlarmOff
	R_RTC_Set_AlarmValue	R_{Config_RTC}_Set_AlarmValue
	R_RTC_Get_AlarmValue	R_{Config_RTC}_Get_AlarmValue
	R_RTC_Set_RTC1HZOn	R_{Config_RTC}_Set_RTC1HZOn
	R_RTC_Set_RTC1HZOff	R_{Config_RTC}_Set_RTC1HZOff
	R_RTC_Create_UserInit	R_{Config_RTC}_Create_UserInit
	r_rtc_interrupt	r_{Config_RTC}_interrupt
	r_rtc_callback_constperiod	r_{Config_RTC}_callback_constperiod
	r_rtc_callback_alarm	r_{Config_RTC}_callback_alarm

Table 5-2 Code Generator and Smart Configurator API comprison List (2/6)

Peripheral Function	Code Generator API Function Name	Smart Configurator API Function Name
Clock output/buzzer output controller	R_PCLBUZ*n*_Create	R_{Config_PCLBUZ*n*}_Create
	R_PCLBUZ*n*_Start	R_{Config_PCLBUZ*n*}_Start
	R_PCLBUZ*n*_Stop	R_{Config_PCLBUZ*n*}_Stop
	R_PCLBUZ*n*_Create_UserInit	R_{Config_PCLBUZ*n*}_Create_UserInit
Watchdog timer	R_WDT_Create	R_{Config_WDT}_Create
	R_WDT_Restart	R_{Config_WDT}_Restart
	R_WDT_Create_UserInit	R_{Config_WDT}_Create_UserInit
	r_wdt_interrupt	r_{Config_WDT}_interrupt
A/D converter	R_ADC_Create	R_{Config_ADC}_Create
	R_ADC_Start	R_{Config_ADC}_Start
	R_ADC_Stop	R_{Config_ADC}_Stop
	R_ADC_Set_OperationOn	R_{Config_ADC}_Set_OperationOn
	R_ADC_Set_OperationOff	R_{Config_ADC}_Set_OperationOff
	R_ADC_Reset	R_ADC_Set_Reset
	R_ADC_Set_PowerOff	R_ADC_Set_PowerOff
	R_ADC_Set_ADChannel	R_{Config_ADC}_Set_ADChannel
	R_ADC_Set_SnoozeOn	R_{Config_ADC}_Set_SnoozeOn
	R_ADC_Set_SnoozeOff	R_{Config_ADC}_Set_SnoozeOff
	R_ADC_Set_TestChannel	R_{Config_ADC}_Set_TestChannel
	R_ADC_Get_Result	R_{Config_ADC}_Get_Result_10bit
	R_ADC_Get_Result_8bit	R_{Config_ADC}_Get_Result_8bit
	R_ADC_Create_UserInit	R_{Config_ADC}_Create_UserInit
	r_adc_interrupt	r_{Config_ADC}_interrupt
D/A converter	R_DAC_Create	R_DAC_Create
	R_DAC*n*_Start	R_{Config_DAC*n*}_Start
	R_DAC*n*_Stop	R_{Config_DAC*n*}_Stop
	R_DAC_Set_PowerOff	R_DAC_Set_PowerOff
	R_DAC*n*_Set_ConversionValue	R_{Config_DAC*n*}_Set_ConversionValue
	R_DAC_Reset	R_DAC_Set_Reset
	R_DAC*n*_Create_UserInit	R_{Config_DAC*n*}_Create_UserInit
Comparator	R_COMP_Create	R_COMP_Create
	R_COMP*n*_Start	R_{Config_COMP*n*}_Start
	R_COMP*n*_Stop	R_{Config_COMP*n*}_Stop
	R_COMP_Reset	R_COMP_Set_Reset
	R_COMP_Set_PowerOff	R_COMP_Set_PowerOff
	R_COMP_Create_UserInit	R_{Config_COMP*n*}_Create_UserInit
	r_comp*n*_interrupt	r_{Config_COMP*n*}_interrupt
Programmable gain amplifier	R_PGA_Create	R_{Config_PGA}_Create
	R_PGA_Start	R_{Config_PGA}_Start
	R_PGA_Stop	R_{Config_PGA}_Stop
	R_PGA_Create_UserInit	R_{Config_PGA}_Create_UserInit

Table 5-3 Code Generator and Smart Configurator API Comparison List (3/6)

Peripheral Function	Code Generator API Function Name	Smart Configurator API Function Name
Serial array unit	R_SAU*m*_Create	R_SAU*m*_Create
	R_SAU*m*_Reset	R_SAU*m*_Set_Reset
	R_SAU*m*_Set_PowerOff	R_SAU*m*_Set_PowerOff
	R_SAU*m*_Set_SnoozeOn	R_SAU*m*_Set_SnoozeOn
	R_SAU*m*_Set_SnoozeOff	R_SAU*m*_Set_SnoozeOff
	R_UART*n*_Create	R_{Config_UART*q*}_Create
	R_UART*n*_Start	R_{Config_UART*q*}_Start
	R_UART*n*_Stop	R_{Config_UART*q*}_Stop
	R_UART*n*_Send	R_{Config_UART*q*}_Send
	R_UART*n*_Receive	R_{Config_UART*q*}_Receive
	R_CSI*mn*_Create	R_{Config_CSI*p*}_Create
	R_CSI*mn*_Start	R_{Config_CSI*p*}_Start
	R_CSI*mn*_Stop	R_{Config_CSI*p*}_Stop
	R_CSI*mn*_Send	R_{Config_CSI*p*}_Send
	R_CSI*mn*_Receive	R_{Config_CSI*p*}_Receive
	R_CSI*mn*_Send_Receive	R_{Config_CSI*p*}_Send_Receive
	R_IIC*mn*_Create	R_{Config_IIC*r*}_Create
	R_IIC*mn*_StartCondition	R_{Config_IIC*r*}_StartCondition
	R_IIC*mn*_StopCondition	R_{Config_IIC*r*}_StopCondition
	R_IIC*mn*_Stop	R_{Config_IIC*r*}_Stop
	R_IIC*mn*_Master_Send	R_{Config_IIC*r*}_Master_Send
	R_IIC*mn*_Master_Receive	R_{Config_IIC*r*}_Master_Receive
	R_SAU*m*_Create_UserInit	-
	r_uart*n*_interrupt_send	r_{Config_UART*q*}_interrupt_send
	r_uart*n*_interrupt_receive	r_{Config_UART*q*}_interrupt_receive
	r_uart*n*_interrupt_error	r_{Config_UART*q*}_interrupt_error
	r_uart*n*_callback_sendend	r_{Config_UART*q*}_callback_sendend
	r_uart*n*_callback_receiveend	r_{Config_UART*q*}_callback_receiveend
	r_uart*n*_callback_error	r_{Config_UART*q*}_callback_error
	r_uart*n*_callback_softwareoverrun	r_{Config_UART*q*}_callback_softwareoverrun
	r_csi*mn*_interrupt	r_{Config_CSI*p*}_interrupt
	r_csi*mn*_callback_sendend	r_{Config_CSI*p*}_callback_sendend
	r_csi*mn*_callback_receiveend	r_{Config_CSI*p*}_callback_receiveend
	r_csi*mn*_callback_error	r_{Config_CSI*p*}_callback_error
	r_iic*mn*_interrupt	r_{Config_IIC*r*}_interrupt
	r_iic*mn*_callback_master_sendend	r_{Config_IIC*r*}_callback_master_sendend
	r_iic*mn*_callback_master_receiveend	r_{Config_IIC*r*}_callback_master_receiveend
	r_iic*mn*_callback_master_error	r_{Config_IIC*r*}_callback_master_error

Table 5-4 Code Generator and Smart Configurator API Comparison List (4/6)

Peripheral Function	Code Generator API Function Name	Smart Configurator API Function Name
Serial interface IICA	R_IICA*n*_Create	R_{Config_IICA*n*}_Create
	R_IICA*n*_StopCondition	R_{Config_IICA*n*}_StopCondition
	R_IICA*n*_Stop	R_{Config_IICA*n*}_Stop
	R_IICA*n*_Reset	R_IICA*n*_Set_Reset
	R_IICA*n*_Set_PowerOff	R_IICA*n*_Set_PowerOff
	R_IICA*n*_Master_Send	R_{Config_IICA*n*}_Master_Send
	R_IICA*n*_Master_Receive	R_{Config_IICA*n*}_Master_Receive
	R_IICA*n*_Slave_Send	R_{Config_IICA*n*}_Slave_Send
	R_IICA*n*_Slave_Receive	R_{Config_IICA*n*}_Slave_Receive
	R_IICA*n*_Set_SnoozeOn	-
	R_IICA*n*_Set_SnoozeOff	-
	R_IICA*n*_Set_WakeupOn	R_{Config_IICA*n*}_Set_WakeupOn
	R_IICA*n*_Set_WakeupOff	R_{Config_IICA*n*}_Set_WakeupOff
	R_IICA*n*_Create_UserInit	R_{Config_IICA*n*}_Create_UserInit
	r_iica*n*_interrupt	r_{Config_IICA*n*}_interrupt
	r_iica*n*_callback_master_sendend	r_{Config_IICA*n*}_callback_master_sendend
	r_iica*n*_callback_master_receiveend	r_{Config_IICA*n*}_callback_master_receiveend
	r_iica*n*_callback_master_error	r_{Config_IICA*n*}_callback_master_error
	r_iica*n*_callback_slave_sendend	r_{Config_IICA*n*}_callback_slave_sendend
	r_iica*n*_callback_slave_receiveend	r_{Config_IICA*n*}_callback_slave_receiveend
	r_iica*n*_callback_slave_error	r_{Config_IICA*n*}_callback_slave_error
	r_iica*n*_callback_getstopcondition	r_{Config_IICA*n*}_callback_getstopcondition
Data transfer controller	R_DTC_Create	R_{Config_DTC}_Create
	R_DTC*n*_Start	R_{Config_DTCD*n*}_Start
	R_DTC*n*_Stop	R_{Config_DTCD*n*}_Stop
	R_DTC_Set_PowerOff	R_DTC_Set_PowerOff
	R_DTC_Create_UserInit	R_{Config_DTC}_Create_UserInit
Event link controller	R_ELC_Create	R_{Config_ELC}_Create
	R_ELC_Stop	R_{Config_ELC}_Stop
	R_ELC_Create_UserInit	R_{Config_ELC}_Create_UserInit
Interrupt functions	R_INTC_Create	R_{Config_INTC}_Create
	R_INTC*n*_Start	R_{Config_INTC}_INTP*n*_Start
	R_INTC*n*_Stop	R_{Config_INTC}_INTP*n*_Stop
	R_INTC_Create_UserInit	R_{Config_INTC}_Create_UserInit
	r_intcn_interrupt	r_{Config_INTC}_intp*n*_interrupt
Key interrupt function	R_KEY_Create	R_{Config_KR}_Create
	R_KEY_Start	R_{Config_KR}_Start
	R_KEY_Stop	R_{Config_KR}_Stop
	R_KEY_Create_UserInit	R_{Config_KR}_Create_UserInit
	r_key_interrupt	r_{Config_KR}_interrupt
Voltage detector	R_LVD_Create	R_{Config_LVD*n*}_Create
	R_LVD_InterruptMode_Start	R_LVD_Start_Interrupt
	R_LVD_Create_UserInit	R_{Config_LVD*n*}_Create_UserInit
	r_lvd_interrupt	r_lvd_interrupt

Table 5-5 Code Generator and Smart Configurator API Comparison List (5/6)

Peripheral Function	Code Generator API Function Name	Smart Configurator API Function Name
Timer RD	R_TMRD*n*_Create	R_TRD_Create
	R_TMRD*n*_Start	R_{Config_TRD*n*}_Start
	R_TMRD*n*_Stop	R_{Config_TRD*n*}_Stop
	R_TMRD*n*_Set_PowerOff	R_TRD_Set_PowerOff
	R_TMRD*n*_ForcedOutput_Start	R_TRD_ForcedOutput_Enable
	R_TMRD*n*_ForcedOutput_Stop	R_TRD_ForcedOutput_Disable
	R_TMRD*n*_Get_PulseWidth	R_{Config_TRD*n*}_Get_PulseWidth
	R_TMRD_PWMOP_ForcedOutput_Stop	R_{Config_PWMOPA}_Software_Release
	R_TMRD_PWMOP_Set_PowerOff	R_PWMOPA_Set_PowerOff
	R_TMRD*n*_Create_UserInit	R_{Config_TRD*n*}_Create_UserInit
	r_tmrdn_interrupt	r_{Config_TRD*n*}_trd*n*_interrupt
Timer RJ	R_TMRJ*n*_Create	R_{Config_TRJ*n*}_Create
	R_TMRJ*n*_Create_UserInit	R_{Config_TRJ*n*}_Create_UserInit
	r_tmrjn_interrupt	r_{Config_TRJ*n*}_ interrupt
	R_TMRJ*n*_Start	R_{Config_TRJ*n*}_Start
	R_TMRJ*n*_Stop	R_{Config_TRJ*n*}_Stop
	R_TMRJ*n*_Set_PowerOff	R_TRJ_Set_PowerOff
	R_TMRJ*n*_Get_PulseWidth	R_{Config_TRJ*n*}_Get_PulseWidth
12-bit interval timer	R_IT_Create	R_{Config_IT}_Create
	R_IT_Create_UserInit	R_{Config_IT}_Create_UserInit
	r_it_interrupt	r_{Config_IT}_ interrupt
	R_IT_Start	R_{Config_IT}_Start
	R_IT_Stop	R_{Config_IT}_Stop
	R_IT_Set_PowerOff	R_IT_Set_PowerOff
Timer RG	R_TMRG*n*_Create	R_{Config_TRG}_Create
	R_TMRG*n*_Create_UserInit	R_{Config_TRG}_Create_UserInit
	r_tmrgn_interrupt	r_{Config_TRG}_ interrupt
	R_TMRG*n*_Start	R_{Config_TRG}_Start
	R_TMRG*n*_Stop	R_{Config_TRG}_Stop
	R_TMRG*n*_Set_PowerOff	R_TRG_Set_PowerOff
	R_TMRG*n*_Get_PulseWidth	R_{Config_TRG}_Get_PulseWidth
Timer RX	R_TMRX_Create	R_{Config_TRX}_Create
	R_TMRX_Create_UserInit	R_{Config_TRX}_Create_UserInit
	r_tmrx_interrupt	r_{Config_TRX}_ interrupt
	R_TMRX_Start	R_{Config_TRX}_Start
	R_TMRX_Stop	R_{Config_TRX}_Stop
	R_TMRX_Set_PowerOff	R_TRX_Set_PowerOff
	R_TMRX_Get_BufferValue	R_{Config_TRX}_Get_BufferValue

Table 5-6 Code Generator and Smart Configurator API Comparison List (6/6)

Peripheral Function	Code Generator API Function Name	Smart Configurator API Function Name
Timer KB	R_KB*n*_Create	R_{Config_TKB*n*}_Create
	R_KB*n*_Start	R_{Config_TKB*n*}_Start
	R_KB*n*_Stop	R_{Config_TKB*n*}_Stop
	R_KB*n*_Simultaneous_Start	-
	R_KB*n*_Simultaneous_Stop	-
	R_KB*n*_Synchronous_Start	-
	R_KB*n*_Synchronous_Stop	-
	R_KB*n*_TKBO*m*0_SmoothStartFunction_Start	R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Start
	R_KB*n*_TKBO*m*0_SmoothStartFunction_Stop	R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop
	R_KB*n*_TKBO*m*1_SmoothStartFunction_Start	R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Start
	R_KB*n*_TKBO*m*1_SmoothStartFunction_Stop	R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Stop
	R_KB*n*_TKBO*m*0_DitheringFunction_Start	R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Start
	R_KB*n*_TKBO*m*0_DitheringFunction_Stop	R_{Config_TKB*n*}_TKBO*n*0_DitheringFunction_Stop
	R_KB*n*_TKBO*m*1_DitheringFunction_Start	R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Start
	R_KB*n*_TKBO*m*1_DitheringFunction_Stop	R_{Config_TKB*n*}_TKBO*n*1_DitheringFunction_Stop
	R_KB*n*_Set_BatchOverwriteRequestOn	R_{Config_TKB*n*}_Set_BatchOverwriteRequestOn
	R_KB*n*_TKBO*m*0_Forced_Output_Stop_Function1_Start	R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Start
	R_KB*n*_TKBO*m*0_Forced_Output_Stop_Function1_Stop	R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Stop
	R_KB*n*_TKBO*m*1_Forced_Output_Stop_Function1_Start	R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Start
	R_KB*n*_TKBO*m*1_Forced_Output_Stop_Function1_Stop	R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Stop
	R_KB*n*_Set_PowerOff	R_{Config_TKB*n*}_Set_PowerOff
	R_KB*n*_Create_UserInit	R_{Config_TKB*n*}_Create_UserInit
	r_kb*n*_interrupt	r_{Config_TKB*n*}_end_count_interrupt

Revision Record

Rev.	Section	Description
1.00	—	First Edition issued
1.01	Section 2.1 Description	Table 2.8 Output File List (8/10): Add a new function r_{Config_UARTA*n*}_PollingEnd_UserCode() Remove two functions r_{Config_UARTA*n*}_send_1byte() and R_{Config_UARTA*n*}_Send_Polling() from {Config_UARTA*n*}_user.c.
		Table 2.10 Output File List (10/10): update Logic and Event Link Controller API: 1.Added new API r_{Config_xxx}_interrupt() 2.Update remark info for R_{Config_xxx}_Create() 3.Update usage example to remove R_Config_AND_Create() calling in main()
	Section 4.2 Function Reference	4.2.23 UART Communication (Serial Interface UARTA): Add a new function r_{Config_UARTA*n*}_PollingEnd_UserCode() Remove two functions r_{Config_UARTA*n*}_send_1byte() and R_{Config_UARTA*n*}_Send_Polling() from {Config_UARTA*n*}_user.c.
		4.2.33 Logic and Event Link Controller: 1.Added new API r_{Config_xxx}_interrupt() 2.Update remark info for R_{Config_xxx}_Create() 3.Update usage example to remove R_Config_AND_Create() calling in main()
1.02	all	Add remark for all callback functions
	Section 4.2 Function Reference	4.2.1 General: Update Table 4-1 and Table 4-2
		Add 4.2.6 External Event Counter (Timer RJ)
		Add 4.2.8 Input Pulse High-/Low-Level Width Measurement (Timer RJ)
		Add 4.2.10 PWM Output (Timer RDn using PWM mode/ Extended PWM mode)
		Add 4.2.11 PWM Output (Timer RD0 and RD1 using PWM mode/ PWM3 mode/ Extended PWM mode)
		4.2.12 Input Pulse Interval/Period Measurement (Timer Array Unit): Update chapter name
		Add 4.2.13 Input Pulse Interval/Period Measurement (Timer RJ)
		Add 4.2.15 Interval Timer (Timer RJ)
		Add 4.2.18 Square Wave Output (Timer RJ)
		Add 4.2.22 Input Capture Function (Timer RD)
		Add 4.2.23 Output Compare Function (Timer RD)
		Add 4.2.24 Three -phase PWM Output (Timer RD)
		Add 4.2.25 PWM option unit A (Timer RD)
		Add 4.2.29 12 Bit A/D Single Scan
		Add 4.2.30 12 Bit A/D Continuous Scan

Rev.	Section	Description
1.02	Section 4.2 Function Reference	Add 4.2.31 12 Bit A/D Group Scan
		Add 4.2.38 UART Communication (LIN/UART module)
		Add 4.2.49 Event Link Controller
		4.2.41 IIC Communication (Slave mode) (Serial Interface IICA): Add remark for R_{Config_IICAn}_Slave_Send/R_{Config_IICAn}_Slave_Receive/r_{Config_IICAn}_slave_handler
	Appendix API Function Comparison Table	Add Table 5-5 Code Generator and Smart Configurator API Comparison List (5/5)
1.03	Section 2.1 Description	Update Table 2-2 Output File List (2/14), Table 2-6 Output File List (6/14), Table 2-14 Output File List (14/14)
	Section 4.2 Function Reference	4.2.1 General: 1.Update Table 4-1 and Table 4-2; 2.Update R_ITL_Start_Interrupt; 3.Add R_TRD_ForcedOutput_Enable, R_TRD_ForcedOutput_disable, R_IT_Set_PowerOn and R_IT_Set_PowerOff
		Add 4.2.16 Interval Timer (12-bit Interval Timer)
		4.2.17 One-Shot Pulse Output: Add R_{Config_TAUm_n}_Get_PulseWidth
		4.2.20 Interval Timer (32-bit Interval Timer using 8-bit counter mode): Update R_{Config_ITLn}_Start
		4.2.21 Interval Timer (32-bit Interval Timer using 16-bit counter mode): Update R_{Config_ITLn_ITLm}_Start
		4.2.22 Interval Timer (32-bit Interval Timer using 32-bit counter mode): Update R_{Config_ITL000_ITL001_ITL012_ITL013}_Start
		Update 4.2.49 Logic and Event Link Controller
	Appendix API Function Comparison Table	Update Table 5-5 Code Generator and Smart Configurator API Comparison List (5/5)
1.04	Section 2.1 Description	Update Table 2-1 to Table 2-21
	Section 4.2 Function Reference	4.2.1 General: Add R_TRD_Set_Reset, R_TRD_Release_Reset, R_PWMOPA_Set_Reset, R_PWMOPA_Release_Reset, R_TRJ_Set_Reset, R_TRJ_Release_Reset, R_TRG_Set_PowerOn, R_TRG_Set_PowerOff, R_TRG_Set_Reset, R_TRG_Release_Reset, R_TRX_Set_PowerOn, R_TRX_Set_PowerOff, R_TRX_Set_Reset, R_TRX_Release_Reset, R_TKB_Create, R_TKB_Set_PowerOn, R_TKB_Set_PowerOff, R_TKB_Set_Reset, R_TKB_Release_Reset, R_PGACOMP_Create, R_PGACOMP_Set_PowerOn, R_PGACOMP_Set_PowerOff, R_PGACOMP_Set_Reset, R_PGACOMP_Release_Reset, R_DALI_Set_PowerOn, R_DALI_Set_PowerOff, R_DALI_Set_Reset, R_DALI_Release_Reset
		4.2.7 Input Pulse High-/Low-Level Width Measurement (Timer Array Unit): Update the hyperlink in Remark of R_{Config_TAUm_n}_Create_UserInit
		Add 4.2.11 PWM Output (Timer RD0 and RD1 using PWM mode/ PWM3 mode/ Extended PWM mode/ Timer KB3 PWM Output Gate mode)
		Add 4.2.12 PWM Output (Timer RG using PWM mode/ PWM2 mode)

Rev.	Section	Description
1.04	Section 4.2 Function Reference	Add 4.2.13 PWM Output (Timer KB using standalone mode (period controlled by TKBCRn0 register)/standalone mode (period controlled by external trigger input)/interleave PFC output mode)
		Add 4.2.14 PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (1 slave)
		Add 4.2.15 PWM Output (Timer KB using simultaneous start/stop mode (period controlled by TKBCRn0 register)/simultaneous start/stop mode (period controlled by external trigger input)/synchronous start/clear mode (period controlled by master)) (2 slaves)
		4.2.16 Input Pulse Interval/Period Measurement (Timer Array Unit): Update the hyperlink in Remark of R_{Config_TAUm_n}_Create_UserInit
		4.2.18 Interval Timer (Timer Array Unit): Update the content of example
		4.2.22 Square Wave Output (Timer Array Unit): Update the hyperlink in Remark of R_{Config_TAUm_n}_Create_UserInit and the content of example
		Add 4.2.28 Input Capture Function (Timer RG)
		Add 4.2.29 Input Capture Function (Timer RX)
		Add 4.2.31 Output Compare Function (Timer RG)
		Add 4.2.34 Phase Counting Mode
		4.2.37 A/D Converter: Add API
		4.2.42 Data Transfer Controller: Update the hyperlink in R_{Config_DTC}_Create
		4.2.43 Comparator: Update description of R_{Config_COMPn}_Create
		Add 4.2.44 Programmable Gain Amplifier
		4.2.46 UART Communication (Serial array unit): Update the description of R_{Config_UARTq}_Send and the content of example
		4.2.47 UART Communication (Serial Interface UARTA): Update the description of R_{Config_UARTq}_Send and the hyperlink in Remark of R_{Config_UARTAn}_Create_UserInit
		4.2.48 UART Communication (LIN/UART module): Update the description of R_{Config_RLIN3n}_Send and the hyperlink in Remark of r_{Config_RLIN3n}_callback_receiveend
		Add 4.2.49 DALI Communication (Control devices)
		Add 4.2.50 DALI Communication (Control gear)
	Appendix API Function Comparison Table	Update Table 5-2 Code Generator and Smart Configurator API comprison List (2/6), Table 5-5 Code Generator and Smart Configurator API comprison List (5/6), Table 5-6 Code Generator and Smart Configurator API comprison List (6/6)

Smart Configurator User's Manual: RL78 API Reference

Publication Date:	Rev.1.00	Apr 01, 2021
	Rev.1.04	Jul 20, 2023

Published by:	Renesas Electronics Corporation

Smart Configurator

R20UT4852EC0103