Smart Configurator
User's Manual: RX API Reference

Target Device
RX Family

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

Readers
The target readers of this manual are the application system engineers who use the Code Generator and need to understand its function.

Purpose
The purpose of this manual is to explain the user for understanding and using the Code Generator functions.
We aim to help their system development including their hardware and software.

Organization
This manual can be broadly divided into the following units.
1. GENERAL
2. OUTPUT FILES
3. API FUNCTIONS

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** ...................................................................................................................................................... 6  
   1.1 Overview ............................................................................................................................................................ 6  
   1.2 Features ............................................................................................................................................................... 6  

2. **OUTPUT FILES** ............................................................................................................................................. 7  
   2.1 Description .......................................................................................................................................................... 7  

3. **INITIALIZATION** .......................................................................................................................................... 20  

4. **API FUNCTIONS** ......................................................................................................................................... 21  
   4.1 Overview .......................................................................................................................................................... 21  
   4.2 Function Reference ........................................................................................................................................... 22  
      4.2.1 Common .................................................................................................................................................. 24  
      4.2.2 8-Bit timer ................................................................................................................................................ 34  
      4.2.3 Buses ........................................................................................................................................................ 42  
      4.2.4 Clock Frequency Accuracy Measurement Circuit ................................................................................... 50  
      4.2.5 Comparator .............................................................................................................................................. 59  
      4.2.6 Compare Match Timer ............................................................................................................................. 66  
      4.2.7 Complementary PWM Mode Timer ........................................................................................................ 76  
      4.2.8 Continuous Scan Mode S12AD ................................................................................................................ 86  
      4.2.9 CRC Calculator ...................................................................................................................................... 101  
      4.2.10 D/A Converter ....................................................................................................................................... 111  
      4.2.11 Data Operation Circuit ........................................................................................................................... 120  
      4.2.12 Data Transfer Controller ........................................................................................................................ 130  
      4.2.13 Dead Time Compensation Counter ........................................................................................................ 136  
      4.2.14 DMA Controller ..................................................................................................................................... 147  
      4.2.15 Event Link Controller ............................................................................................................................ 161  
      4.2.16 General PWM Timer (GPT) .................................................................................................................. 171  
      4.2.17 Group Scan Mode S12AD ..................................................................................................................... 188  
      4.2.18 I2C Master Mode ................................................................................................................................. 205  
      4.2.19 I2C Slave Mode ................................................................................................................................... 238  
      4.2.20 Interrupt Controller ................................................................................................................................ 254  
      4.2.21 Low Power Consumption ...................................................................................................................... 270  
      4.2.22 Low Power Timer ................................................................................................................................. 282  
      4.2.23 Normal Mode Timer .............................................................................................................................. 288  
      4.2.24 Phase Counting Mode Timer ............................................................................................................... 296
4.2.25 Port Output Enable ................................................................................................................................. 307
4.2.26 I/O Port .................................................................................................................................................. 318
4.2.27 Programmable Pulse Generator ............................................................................................................. 322
4.2.28 PWM Mode Timer ................................................................................................................................... 326
4.2.29 Realtime Clock ...................................................................................................................................... 334
4.2.30 Remote Control Signal Receiver ........................................................................................................... 359
4.2.31 SCI/SCIF Asynchronous Mode ............................................................................................................. 375
4.2.32 SCI/SCIF Clock Synchronous Mode ..................................................................................................... 399
4.2.33 Single Scan Mode S12AD ..................................................................................................................... 422
4.2.34 Smart Card Interface Mode ................................................................................................................... 437
4.2.35 SPI Clock Synchronous Mode ............................................................................................................. 452
4.2.36 SPI Operation Mode .............................................................................................................................. 474
4.2.37 Voltage Detection Circuit ..................................................................................................................... 489
4.2.38 Watchdog Timer .................................................................................................................................... 496
4.2.39 Continuous Scan Mode DSAD .............................................................................................................. 504
4.2.40 Single Scan Mode DSAD ...................................................................................................................... 515
4.2.41 Delta-Sigma Modulator Interface .......................................................................................................... 526
4.2.42 Analog Front End .................................................................................................................................. 535
4.2.43 Motor ..................................................................................................................................................... 538
4.2.44 LCD Controller ...................................................................................................................................... 551

Revision Record ............................................................................................................................................... 558
1. GENERAL

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

1.1 Overview

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

1.2 Features

The features of the Code Generator are as follows.

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

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

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

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

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

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

This chapter explains the file output by the Code Generator.

2.1 Description

The Code Generator outputs the following files.

Table 2.1 Output File List (1/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>&lt;workspaceName&gt;.c</td>
<td>main</td>
</tr>
<tr>
<td></td>
<td>dbsct.c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>resetprg.c</td>
<td>PowerON_Reset</td>
</tr>
<tr>
<td></td>
<td>sbrk.c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vecttbl.c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vecttbl.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hwsetup.c</td>
<td>hardware_setup</td>
</tr>
<tr>
<td></td>
<td>hwsetup.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r_cg_hardware_setup.c</td>
<td>r_undefined_exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_Systeminit</td>
</tr>
<tr>
<td>r_cg_macrodriver.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_cg_userdefine.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_smc_entry.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_smc_cgc.c</td>
<td>R_CGC_Create</td>
<td></td>
</tr>
<tr>
<td>r_smc_cgc_user.c</td>
<td>R_CGC_Create_UserInit</td>
<td></td>
</tr>
<tr>
<td>r_smc_cgc.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_smc_interrup.c</td>
<td>R_Interrupt_Create</td>
<td></td>
</tr>
<tr>
<td>r_smc_interrupt.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pin.c</td>
<td>R_Pins_Create</td>
<td></td>
</tr>
<tr>
<td>Pin.h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.2 Output File List (2/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit timer</td>
<td><code>&lt;Config_TMR0&gt;.c</code></td>
<td>R_&lt;Config_TMR0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_TMR0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_TMR0&gt;_user.c</code></td>
<td>R_&lt;Config_TMR0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_TMR0&gt;_cmimn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_TMR0&gt;_cmuin_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_TMR0&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td>Buses</td>
<td><code>&lt;Config_BSC&gt;.c</code></td>
<td>R_&lt;Config_BSC&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_BSC&gt;_Error_Monitoring_Start</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_BSC&gt;_user.c</code></td>
<td>R_&lt;Config_BSC&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_BSC&gt;_buserr_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_BSC&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td>Clock Frequency</td>
<td><code>&lt;Config_CAC&gt;.c</code></td>
<td>R_&lt;Config_CAC&gt;_Create</td>
</tr>
<tr>
<td>Accuracy Measurement</td>
<td></td>
<td>R_&lt;Config_CAC&gt;_Start</td>
</tr>
<tr>
<td>Circuit</td>
<td><code>&lt;Config_CAC&gt;_user.c</code></td>
<td>R_&lt;Config_CAC&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CAC&gt;_mendf_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CAC&gt;_mendi_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CAC&gt;_ferrf_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CAC&gt;_ferri_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CAC&gt;_ovff_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CAC&gt;_ovfi_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_CAC&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td>Comparator</td>
<td><code>&lt;Config_CMPB0&gt;.c</code></td>
<td>R_&lt;Config_CMPB0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CMPB0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_CMPB0&gt;_user.c</code></td>
<td>R_&lt;Config_CMPB0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CMPB0&gt;_cmpbn_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_CMPB0&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td>Compare Match Timer</td>
<td><code>&lt;Config_CMT0&gt;.c</code></td>
<td>R_&lt;Config_CMT0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CMT0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_CMT0&gt;_user.c</code></td>
<td>R_&lt;Config_CMT0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CMT0&gt;_cmmin_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_CMT0&gt;_cmwin_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_CMT0&gt;.h</code></td>
<td></td>
</tr>
</tbody>
</table>
## Table 2.3 Output File List (3/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complementary PWM Mode Timer</strong></td>
<td><code>&lt;Config_MTU3_MTU4&gt;.c</code></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU3_MTU4&gt;_user.c</code></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU3_MTU4&gt;_tgimn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU3_MTU4&gt;_cj_tgimj_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU3_MTU4&gt;_cj_tcivj_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU3_MTU4&gt;.h</code></td>
<td>—</td>
</tr>
<tr>
<td><strong>Continuous Scan Mode S12AD</strong></td>
<td><code>&lt;Config_S12AD0&gt;.c</code></td>
<td>R_&lt;Config_S12AD0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Get_ValueResult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareValue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareAValue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareBValue</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_S12AD0&gt;_user.c</code></td>
<td>R_&lt;Config_S12AD0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_compare_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptB</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_S12AD0&gt;.h</code></td>
<td>—</td>
</tr>
<tr>
<td><strong>CRC Calculator</strong></td>
<td><code>&lt;Config_CRC&gt;.c</code></td>
<td>R_&lt;Config_CRC&gt;_SetCRC8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CRC&gt;_SetCRC16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CRC&gt;_SetCCITT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CRC&gt;_SetCRC32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CRC&gt;_SetCRC32C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CRC&gt;_Input_Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_CRC&gt;_Get_Result</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_CRC&gt;.h</code></td>
<td>—</td>
</tr>
<tr>
<td><strong>D/A Converter</strong></td>
<td><code>&lt;Config_DA&gt;.c</code></td>
<td>R_&lt;Config_DA&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DA&gt;_n_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DA&gt;_n_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DA&gt;_n_Set_ConversionValue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DA&gt;_Sync_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DA&gt;_Sync_Stop</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DA&gt;_user.c</code></td>
<td>R_&lt;Config_DA&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DA&gt;.h</code></td>
<td>—</td>
</tr>
</tbody>
</table>
### Table 2.4  Output File List (4/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Data Operation Circuit</strong></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DOC&gt;.c</code></td>
<td>R_&lt;Config_DOC&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DOC&gt;_SetMode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DOC&gt;_WriteData</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DOC&gt;_GetResult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DOC&gt;_ClearFlag</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DOC&gt;_user.c</code></td>
<td>R_&lt;Config_DOC&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_DOC&gt;_dopcf_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_DOC&gt;_dopci_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DOC&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Data Transfer Controller</strong></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DTC&gt;.c</code></td>
<td>R_&lt;Config_DTC&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DTC&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DTC&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DTC&gt;_user.c</code></td>
<td>R_&lt;Config_DTC&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DTC&gt;.h</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Dead Time Compensaion Counter</strong></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU5&gt;.c</code></td>
<td>R_&lt;Config_MTU5&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU5&gt;_U5_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU5&gt;_U5_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU5&gt;_V5_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU5&gt;_V5_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU5&gt;_W5_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU5&gt;_W5_Stop</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU5&gt;_user.c</code></td>
<td>R_&lt;Config_MTU5&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU5&gt;_tgimn_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU5&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>DMA Controller</strong></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DMAC0&gt;.c</code></td>
<td>R_&lt;Config_DMAC0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DMAC0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DMAC0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DMAC0&gt;_Set_SoftwareTrigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DMAC0&gt;_Clear_SoftwareTrigger</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DMAC0&gt;_user.c</code></td>
<td>R_&lt;Config_DMAC0&gt;_Create_Userinit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_DMAC0&gt;_dmacni_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_dmacn_callback_transfer_end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_dmacn_callback_transfer_escape_end</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DMAC0&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_cg_dmac_user.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_dmac_dmac74i_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_dmacn_callback_transfer_end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_dmacn_callback_transfer_escape_end</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DMAC0&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_ELC&gt;.c</code></td>
<td>R_&lt;Config_ELC&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_ELC&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_ELC&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_ELC&gt;_GenerateSoftwareEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_ELC&gt;_Set_PortBuffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_ELC&gt;_Get_PortBuffer</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_ELC&gt;_user.c</code></td>
<td>R_&lt;Config_ELC&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_ELC&gt;_elsrni_interrupt</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_ELC&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td>Peripheral Function</td>
<td>File Name</td>
<td>API Function Name</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td><strong>General PWM Timer</strong></td>
<td><code>&lt;Config_GPT0&gt;.c</code></td>
<td><code>R_&lt;Config_GPT0&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_GPT0&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_GPT0&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_GPT0&gt;_user.c</code></td>
<td><code>R_&lt;Config_GPT0&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_GPT0&gt;_gtcimn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_GPT0&gt;_gtcivn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_GPT0&gt;_gtcinn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_GPT0&gt;_gdten_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_GPT0&gt;.h</code></td>
<td>---</td>
</tr>
<tr>
<td></td>
<td><code>r_cg_gpt_user.c</code></td>
<td><code>r_gpt_etgin_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_gpt_etgip_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>r_cg_gpt.h</code></td>
<td>---</td>
</tr>
</tbody>
</table>

| **Group Scan Mode S12AD** | `<Config_S12AD0>.c` | `R_<Config_S12AD0>_Create` |
|                         |                     | `R_<Config_S12AD0>_Start`    |
|                         |                     | `R_<Config_S12AD0>_Stop`     |
|                         | `<Config_S12AD0>_user.c` | `R_<Config_S12AD0>_Create_UserInit` |
|                         |                     | `r_<Config_S12AD0>_interrupt` |
|                         |                     | `r_<Config_S12AD0>_compare_interrupt` |
|                         |                     | `r_<Config_S12AD0>_compare_interruptA` |
|                         |                     | `r_<Config_S12AD0>_compare_interruptB` |
|                         |                     | `r_<Config_S12AD0>_groupb_interrupt` |
|                         |                     | `r_<Config_S12AD0>_groupc_interrupt` |
|                         | `<Config_S12AD0>.h` | ---                                    |
### Table 2.6 Output File List (6/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I2C Master Mode</strong></td>
<td><code>&lt;Config_RIIC0&gt;.c</code></td>
<td>R_&lt;Config_RIIC0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Master_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Master_Send_Without_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Master_Receive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_IIC_StartCondition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_IIC_StopCondition</td>
</tr>
<tr>
<td><code>&lt;Config_RIIC0&gt;_user.c</code></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_error_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_transmitend_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_callback_error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_callback_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_callback_receiveend</td>
</tr>
<tr>
<td><code>&lt;Config_RIIC0&gt;.h</code> &amp; <code>&lt;Config_SCI0&gt;.c</code></td>
<td></td>
<td>--</td>
</tr>
<tr>
<td><code>&lt;Config_SCI0&gt;_user.c</code></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmitend_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
</tr>
<tr>
<td><code>&lt;Config_SCI0&gt;.h</code> &amp; <code>&lt;Config_RIIC0&gt;.h</code></td>
<td></td>
<td>--</td>
</tr>
<tr>
<td><strong>I2C Slave Mode</strong></td>
<td><code>&lt;Config_RIIC0&gt;.c</code></td>
<td>R_&lt;Config_RIIC0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Slave_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Slave_Receive</td>
</tr>
<tr>
<td><code>&lt;Config_RIIC0&gt;_user.c</code></td>
<td></td>
<td>R_&lt;Config_RIIC0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_error_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_transmitend_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_callback_error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_callback_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RIIC0&gt;_callback_receiveend</td>
</tr>
<tr>
<td><code>&lt;Config_RIIC0&gt;.h</code> &amp; <code>&lt;Config_SCI0&gt;.h</code></td>
<td></td>
<td>--</td>
</tr>
</tbody>
</table>
Table 2.7  Output File List (7/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt Controller</strong></td>
<td><code>&lt;Config_ICU&gt;.c</code></td>
<td><code>R_&lt;Config_ICU&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_IRQn_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_IRQn_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_Software_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_Software_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_SoftwareInterrupt_Generate</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_Software2_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_ICU&gt;_Software2_Stop</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_ICU&gt;_user.c</code></td>
<td><code>R_&lt;Config_ICU&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_ICU&gt;_irqn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_ICU&gt;_software_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_ICU&gt;_software2_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_ICU&gt;_nmi_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_ICU&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td><strong>Low Power Consumption</strong></td>
<td><code>&lt;Config_LPC&gt;.c</code></td>
<td><code>R_&lt;Config_LPC&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_AllModuleClockStop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_Sleep</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_DeepSleep</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_SoftwareStandby</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_DeepSoftwareStandby</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_ChangeOperatingPowerControl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_ChangeSleepModeReturnClock</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPC&gt;_SetVOLSR_PGAVLS</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_LPC&gt;_user.c</code></td>
<td><code>R_&lt;Config_LPC&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_LPC&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td><strong>Low Power Timer</strong></td>
<td><code>&lt;Config_LPT&gt;.c</code></td>
<td><code>R_&lt;Config_LPT&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPT&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_LPT&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_LPT&gt;_user.c</code></td>
<td><code>R_&lt;Config_LPT&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_LPT&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td><strong>Normal Mode Timer</strong></td>
<td><code>&lt;Config_MTU0&gt;.c</code></td>
<td><code>R_&lt;Config_MTU0&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_MTU0&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_MTU0&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU0&gt;_user.c</code></td>
<td><code>R_&lt;Config_MTU0&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_MTU0&gt;_tgimm_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_MTU0&gt;_tgimm_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_MTU0&gt;_tcivn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_MTU0&gt;_tcinv_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_MTU0&gt;.h</code></td>
<td></td>
</tr>
<tr>
<td>Peripheral Function</td>
<td>File Name</td>
<td>API Function Name</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Phase Counting Mode</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td>&lt;Config_MTU1&gt;.c</td>
<td>R_&lt;Config_MTU1&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU1&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU1&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU1_MTU2&gt;_MTU_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU1_MTU2&gt;_MTU_Stop</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU1&gt;_user.c</td>
<td>R_&lt;Config_MTU1&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tcinu_interrupt</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU1&gt;.h</td>
<td>R_&lt;Config_MTU1&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU1&gt;_tcinu_interrupt</td>
</tr>
<tr>
<td><strong>Port Output Enable</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Config_POE&gt;.c</td>
<td>R_&lt;Config_POE&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_POE&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_POE&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_POE&gt;_Set_HiZ_MTU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_POE&gt;_Clear_HiZ_MTU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_POE&gt;_Set_HiZ_GPTn</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_POE&gt;_user.c</td>
<td>R_&lt;Config_POE&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_POE&gt;_oein_interrupt</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_POE&gt;.h</td>
<td>R_&lt;Config_POE&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_POE&gt;_oein_interrupt</td>
</tr>
<tr>
<td><strong>I/O Port</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Config_PORT&gt;.c</td>
<td>R_&lt;Config_PORT&gt;_Create</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_PORT&gt;_user.c</td>
<td>R_&lt;Config_PORT&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_PORT&gt;.h</td>
<td>R_&lt;Config_PORT&gt;_Create_UserInit</td>
</tr>
<tr>
<td><strong>Programmable Pulse</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Generator</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Config_PPG0&gt;.c</td>
<td>R_&lt;Config_PPG0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_PPG0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_PPG0&gt;.h</td>
<td>R_&lt;Config_PPG0&gt;_Create_UserInit</td>
</tr>
<tr>
<td><strong>PWM Mode Timer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU0&gt;.c</td>
<td>R_&lt;Config_MTU0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU0&gt;_user.c</td>
<td>R_&lt;Config_MTU0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU0&gt;.h</td>
<td>R_&lt;Config_MTU0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tginn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tcinvn_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU0&gt;_tcinvn_interrupt</td>
</tr>
</tbody>
</table>
## Table 2.9 Output File List (9/13)

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>File Name</th>
<th>API Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Realtime Clock</strong></td>
<td><code>&lt;Config_RTC&gt;.c</code></td>
<td><code>R_&lt;Config_RTC&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Restart</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Restart_BinaryCounter</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_CalendarCounterValue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Get_CalendarCounterValue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Get_CalendarTimeCaptureValue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_CalendarCounterValue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Get_CalendarTimeCaptureValue</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_RTCOUTOn</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_RTCOUTOff</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_CalendarAlarm</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_BinaryAlarm</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_ConstPeriodInterruptOn</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_ConstPeriodInterruptOff</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_CarryInterruptOn</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Set_CarryInterruptOff</code></td>
</tr>
<tr>
<td><code>&lt;Config_RTC&gt;_user.c</code></td>
<td></td>
<td><code>R_&lt;Config_RTC&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_RTC&gt;_alm_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_RTC&gt;_prd_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_RTC&gt;_cup_interrupt</code></td>
</tr>
<tr>
<td><code>&lt;Config_RTC&gt;.h</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Remote Control Signal Receiver</strong></td>
<td><code>&lt;Config_REMC0&gt;.c</code></td>
<td><code>R_&lt;Config_REMC0&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_REMC0&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_REMC0&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>R_&lt;Config_REMC0&gt;_Read</code></td>
</tr>
<tr>
<td><code>&lt;Config_REMC0&gt;_user.c</code></td>
<td></td>
<td><code>R_&lt;Config_REMC0&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_remin中断</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_comparematch</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_receiveerror</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_receiveend</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_bufferfull</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_header</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_data0</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_data1</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>r_&lt;Config_REMC0&gt;_callback_specialdata</code></td>
</tr>
<tr>
<td><code>&lt;Config_REMC0&gt;.h</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peripheral Function</td>
<td>File Name</td>
<td>API Function Name</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>SCI/SCIF Asynchronous Mode</td>
<td>&lt;Config_SCI0&gt;.c</td>
<td>R_&lt;Config_SCI0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Receive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Multiprocessor_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Multiprocessor_Receive</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_SCI0&gt;_user.c</td>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmitend_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_receiveerror_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_teif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_txif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_rcxf_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_drif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_erif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_brif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_receiverror</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_error</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_SCI0&gt;.h</td>
<td></td>
</tr>
<tr>
<td>SCI/SCIF Clock Synchronous Mode</td>
<td>&lt;Config_SCI0&gt;.c</td>
<td>R_&lt;Config_SCI0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Receive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Serial_Send_Receive</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_SCI0&gt;_user.c</td>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmitend_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_receiveerror_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_teif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_txif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_rcxf_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_drif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_erif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_brif_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_receiverror</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_error</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_SCI0&gt;.h</td>
<td></td>
</tr>
<tr>
<td>Peripheral Function</td>
<td>File Name</td>
<td>API Function Name</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Single Scan Mode S12AD</td>
<td><code>&lt;Config_S12AD0&gt;.c</code></td>
<td>R_&lt;Config_S12AD0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Get_ValueResult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareValue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareAValue</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_S12AD0&gt;_user.c</code></td>
<td>R_&lt;Config_S12AD0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_compare_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptB</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_S12AD0&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td>Smart Card Interface Mode</td>
<td><code>&lt;Config_SCI0&gt;.c</code></td>
<td>R_&lt;Config_SCI0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_SmartCard_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_SCI0&gt;_SmartCard_Receive</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_SCI0&gt;_user.c</code></td>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_SCI0&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td>SPI Clock Synchronous Mode</td>
<td><code>&lt;Config_RSPI0&gt;.c</code></td>
<td>R_&lt;Config_RSPI0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_Send_Receive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_SPI_Master_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_SPI_Master_Send_Receive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_SPI_Slave_Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_RSPI0&gt;_SPI_Slave_Send_Receive</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_user.c</code></td>
<td>R_&lt;Config_RSPI0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_receive_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_transmit_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_error_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_idle_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_transmitend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_receiveerror_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_RSPI0&gt;_callback_receiveend</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_RSPI0&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td>Smart Configurator. 2. OUTPUT FILES</td>
<td>Table 2.11 Output File List (11/13)</td>
<td></td>
</tr>
<tr>
<td>Peripheral Function</td>
<td>File Name</td>
<td>API Function Name</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SPI Operation Mode</strong></td>
<td><code>&lt;Config_RSPI0&gt;.c</code></td>
<td><code>&lt;Config_RSPI0&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_Send</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_Send_Receive</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_user.c</code></td>
<td><code>&lt;Config_RSPI0&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_receive_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_transmit_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_error_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_idle_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_callback_receiveend</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_RSPI0&gt;_callback_transmitend</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_RSPI0&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td><strong>Voltage Detection Circuit</strong></td>
<td><code>&lt;Config_LVD1&gt;.c</code></td>
<td><code>&lt;Config_LVD1&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_LVD1&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_LVD1&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_LVD1&gt;_user.c</code></td>
<td><code>&lt;Config_LVD1&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_LVD1&gt;_lvdn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_LVD1&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td><strong>Watchdog Timer</strong></td>
<td><code>&lt;Config_WDT&gt;.c</code></td>
<td><code>&lt;Config_WDT&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_WDT&gt;_Restart</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_WDT&gt;_user.c</code></td>
<td><code>&lt;Config_WDT&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_WDT&gt;_wuni_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_WDT&gt;_iwuni_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_WDT&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td><strong>Continuous Scan Mode DSAD</strong></td>
<td><code>&lt;Config_DSAD0&gt;.c</code></td>
<td><code>&lt;Config_DSAD0&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Set_SoftwareTrigger</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Get_ValueResult</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Chm_Set_DisconnectDetection</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_user.c</code></td>
<td><code>&lt;Config_DSAD0&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_adin_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_scanendn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DSAD0&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td><strong>Single Scan Mode DSAD</strong></td>
<td><code>&lt;Config_DSAD0&gt;.c</code></td>
<td><code>&lt;Config_DSAD0&gt;_Create</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Start</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Stop</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Set_SoftwareTrigger</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Get_ValueResult</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_Chm_Set_DisconnectDetection</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_user.c</code></td>
<td><code>&lt;Config_DSAD0&gt;_Create_UserInit</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_adin_interrupt</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;Config_DSAD0&gt;_scanendn_interrupt</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;Config_DSAD0&gt;.h</code></td>
<td>-</td>
</tr>
<tr>
<td>Peripheral Function</td>
<td>File Name</td>
<td>API Function Name</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Delta-Sigma</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modulator Interface</td>
<td>&lt;Config_DSMIF0&gt;.c</td>
<td>R_&lt;Config_DSMIF0&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DSMIF0&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_DSMIF0&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_DSMIF0&gt;_user.c</td>
<td>R_&lt;Config_DSMIF0&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_DSMIF0&gt;_ocdin_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_DSMIF0&gt;_scdin_interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_DSMIF0&gt;_sumein_interrupt</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_DSMIF0&gt;.h</td>
<td></td>
</tr>
<tr>
<td>Analog Front End</td>
<td>&lt;Config_AFE&gt;.c</td>
<td>R_&lt;Config_AFE&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Config_AFE&gt;_user.c</td>
<td>R_&lt;Config_AFE&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_AFE&gt;.h</td>
<td></td>
</tr>
<tr>
<td>Motor</td>
<td>&lt;Config_MTU3_MTU4&gt;.c</td>
<td>R_&lt;Config_MTU3_MTU4&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_StartTimerCount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_StopTimerCount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_StartTimerCtrl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_StopTimerCtrl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_UpdDuty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_StartAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_StopAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_MTU3_MTU4&gt;_AdcGetConvVal</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU3_MTU4&gt;_user.c</td>
<td>R_&lt;Config_MTU3_MTU4&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU3_MTU4&gt;_CrestInterrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_&lt;Config_MTU3_MTU4&gt;_ad_interrupt</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_MTU3_MTU4&gt;.h</td>
<td></td>
</tr>
<tr>
<td>LCD Controller</td>
<td>&lt;Config_LCD&gt;.c</td>
<td>R_&lt;Config_LCD&gt;_Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_LCD&gt;_Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_LCD&gt;_Stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_LCD&gt;_Voltage_On</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R_&lt;Config_LCD&gt;_Voltage_Off</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_LCD&gt;_user.c</td>
<td>R_&lt;Config_LCD&gt;_Create_UserInit</td>
</tr>
<tr>
<td></td>
<td>&lt;Config_LCD&gt;.h</td>
<td></td>
</tr>
</tbody>
</table>
3. INITIALIZATION

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

- **PowerOn_Reset_PC**
  - Sets interrupt table register
  - Sets exception table register
  - Sets single-precision floating-point status word

- **mcu_clock_setup**
  - Initializes system clock
  - Initializes CLKOUT pin output

- **User's pre process when worm start**
  - Initializes C runtime environment

- **bsp_ram_initialize**
  - Initializes RAM

- **init_iolib**
  - Initializes C library

- **bsp_interrupt_open**
  - Initializes callback function array

- **bsp_maapped_interrupt_open**
  - Initializes mapped interrupt

- **bsp_register_protect_open**
  - Initializes variables needed for register protection functionality

- **hardware_setup**
  - Configures the ROM cache function
  - Configures the port and pin direction
  - Configures interrupts used

- **R_Systeminit**
  - Initializes peripheral functions

- **bsp_non_existent_port_init**
  - Initializes non existent pins

- **bsp_adc_initial_configure**
  - Configures the ADC initial settings

- **bsp_bsc_initial_configure**
  - Configures the BUS initial settings

- **Sets processor status word**
  - Enable the bus error interrupt

Process of blue box is FIT board support package module. Please refer to the application note (R01AN1685) for details.
4. API FUNCTIONS
This chapter describes the API functions output that are output by the Code Generator.

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

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

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

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

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

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

Remarks
In the generated code by the smart configurator, the for statement, the while statement, the do-while statement (loop processing) are used in register setting reflected waiting process etc. If fail-safe processing for infinite loop is required, check the generated code and add processing.
4.2 Function Reference
This section describes the API functions output by the Code Generator, using the following notation format.

Figure 4.1 Notation Format of API Functions

1. Name
   Indicates the name of the API function.

2. Outline
   Outlines the functions of the API function.

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

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

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
</tr>
</tbody>
</table>

(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.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Macro</td>
<td>(b) Description of return value</td>
</tr>
<tr>
<td>Macro of return value</td>
<td>Description of return value</td>
</tr>
</tbody>
</table>
4.2.1 Common

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

Table 4.1 Common API Functions

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

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

[Syntax]
```c
void r_undefined_exception ( void );
```

[Argument(s)]
None.

[Return value]
None.
**PowerON_Reset**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**hardware_setup**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_Systeminit

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
R_CGC_Create

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

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

[Syntax]

void R_CGC_Create ( void );

[Argument(s)]

None.

[Return value]

None.
**R_CGC_Create_UserInit**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_Interrupt_Create

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function makes settings for the multi-function pin controller.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
main

This is the main function.

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

[Syntax]
void main ( void );

[Argument(s)]
None.

[Return value]
None.
4.2.2 8-Bit timer

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

Table 4.2 API Functions: [8-bit timer]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_TMR0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the 8-bit timer.</td>
</tr>
<tr>
<td>R_&lt;Config_TMR0&gt;_Start</td>
<td>Starts counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_TMR0&gt;_Stop</td>
<td>Stops counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_TMR0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the 8-bit timer.</td>
</tr>
<tr>
<td>r_&lt;Config_TMR0&gt;_cmimn_interrupt</td>
<td>Executes processing in response to compare match interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_TMR0&gt;_ovin_interrupt</td>
<td>Executes processing in response to overflow interrupts.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling the 8-bit timer.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
The API function starts counting by the counter.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**R_<Config_TMR0>_Stop**

This API function stops counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_TMR0>_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
The API function executes processing in response to compare match interrupts.

**[Syntax]**

```c
static void r_ Config_TMR0_ _cmimn_ interrupt ( void );
```

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to overflow interrupts.

**[Syntax]**

```c
static void r_<Config_TMR0>_ovi_interrupt ( void );
```

Remark \( n \) is the channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Use timer as One-shot timer.

[API setting example]

```c
r_cg_main.c

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

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

    r_cg_tmr_user.c

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

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

Table 4.3 API Functions: [Buses]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_BSC&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the bus.</td>
</tr>
<tr>
<td>R_&lt;Config_BSC&gt;_Error_Monitoring_Start</td>
<td>Enables the detection of bus errors due to access to illegal addresses.</td>
</tr>
<tr>
<td>R_&lt;Config_BSC&gt;_Error_Monitoring_Stop</td>
<td>Disables the detection of bus errors due to access to illegal addresses.</td>
</tr>
<tr>
<td>R_&lt;Config_BSC&gt;_InitializeSDRAM</td>
<td>Initializes the SDRAM controller.</td>
</tr>
<tr>
<td>R_&lt;Config_BSC&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the bus.</td>
</tr>
<tr>
<td>r_&lt;Config_BSC&gt;_buserr_interrupt</td>
<td>Executes processing in response to bus errors due to access to illegal addresses.</td>
</tr>
</tbody>
</table>
**R_<Config_BSC>_Create**

This API function executes initialization processing that is required before controlling the bus.  
**Remark**  
This API function is called from `R_Systeminit` before the main() function is executed.

**[Syntax]**

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

**[Argument(s)]**
None.

**[Return value]**
None.
R_<Config_BSC>_Error_Monitoring_Start

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

[Syntax]
void R_<Config_BSC>_Error_Monitoring_Start ( void );

[Argument(s)]
None.

[Return value]
None.
This API function disables the detection of bus errors due to access to illegal addresses.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
R_<Config_BSC>_InitializeSDRAM

This API function initializes the SDRAM controller.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
R_<Config_BSC>_Create_UserInit

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

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

[Syntax]

void R_<Config_BSC>_Create_UserInit ( void );

[Argument(s)]

None.

[Return value]

None.
### r_<Config_BSC>_buserr_interrupt

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

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

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

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

### Syntax

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

### Argument(s)

None.

### Return value

None.
Usage example

Acquiring the address to which access caused a bus error:

main.c

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

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

Config_BSC_user.c

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

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

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

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

Table 4.4 API Functions: [Clock Frequency Accuracy Measurement Circuit]

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

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function starts measurement of the clock frequency accuracy.

**[Syntax]**

```c
void R_{Config_CAC}_Start ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
This API function stops measurement of the clock frequency accuracy.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_CAC>_Create_UserInit**

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

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

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to measurement end interrupts.

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

**[Syntax]**

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

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

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to measurement end interrupts.

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

**Syntax**

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

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

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

**Argument(s)**
None.

**Return value**
None.
This API function executes processing in response to overflow interrupts.

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

**[Syntax]**

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

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

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

**[Argument(s)]**
None.

**[Return value]**
None.
Usage example

Counting the number of frequency errors:

main.c

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

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

Config_CAC_user.c

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

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

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

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

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{&lt;Config_CMPB0&gt;}_Create$</td>
<td>The Code Generator outputs the following API functions for the comparator.</td>
</tr>
<tr>
<td>$R_{&lt;Config_CMPB0&gt;}_Start$</td>
<td>The Code Generator outputs the following API functions for the comparator.</td>
</tr>
<tr>
<td>$R_{&lt;Config_CMPB0&gt;}_Stop$</td>
<td>The Code Generator outputs the following API functions for the comparator.</td>
</tr>
<tr>
<td>$R_{&lt;Config_CMPB0&gt;}_Create_UserInit$</td>
<td>The Code Generator outputs the following API functions for the comparator.</td>
</tr>
<tr>
<td>$r_{&lt;Config_CMPB0&gt;}_cmpbn_interrupt$</td>
<td>The Code Generator outputs the following API functions for the comparator.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling the comparator.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**R_config_CMPB0_Start**

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

**Syntax**

```c
void R_config_CMPB0_Start ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function stops comparison of the analog input voltage with the reference voltage.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_CMPB0>_Create_UserInit**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to comparator Bn interrupts.

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

[Syntax]

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

Remark
n is a channel number.

[Argument(s)]
None.

[Return value]
None.
Usage example

Setting a flag when the result of comparison changes:

main.c

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

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

Config_CMPB0_user.c

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

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

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

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

**Table 4.6 API Functions: [Compare Match Timer]**

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_{Config_CMT0}&gt;_Create</td>
<td>Executes initialization processing that is required before controlling a compare match timer (CMT or CMTW).</td>
</tr>
<tr>
<td>R_{Config_CMT0}&gt;_Start</td>
<td>Starts counting by the counter.</td>
</tr>
<tr>
<td>R_{Config_CMT0}&gt;_Stop</td>
<td>Stops counting by the counter.</td>
</tr>
<tr>
<td>R_{Config_CMT0}&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for a compare match timer (CMT or CMTW).</td>
</tr>
<tr>
<td>r_{Config_CMT0}&gt;_cmin_interrupt</td>
<td>Executes processing in response to compare match interrupts (CM{in}).</td>
</tr>
<tr>
<td>r_{Config_CMT0}&gt;_cmwin_interrupt</td>
<td>Executes processing in response to compare match interrupts (CMW{in}).</td>
</tr>
<tr>
<td>r_{Config_CMT0}&gt;_icmin_interrupt</td>
<td>Executes processing in response to input capture interrupts.</td>
</tr>
<tr>
<td>r_{Config_CMT0}&gt;_ocmin_interrupt</td>
<td>Executes processing in response to output compare interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_CMT0>_Create**

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

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

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function starts counting by the counter.

**[Syntax]**

```c
void R_Config_CMT0_Start ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R\_<Config\_CMT0>\_Stop**

This API function stops counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_CMT0>_Create_UserInit

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

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

[Syntax]
```c
void R_<Config_CMT0>_Create_UserInit ( void );
```

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to compare match interrupts (CMI\textsubscript{n}).

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

**[Syntax]**

```
void r_\<Config\_CMT0\>_cmin\_interrupt ( void );
```

**Remark** \( n \) is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare match interrupts (CMWi).

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

**[Syntax]**

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

**Remark**

- $n$ is a channel number.

**[Argument(s)]**
- None.

**[Return value]**
- None.
### r\_<Config\_CMT0>_icmin\_interrupt

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

**[Syntax]**

```
void r\_<Config\_CMT0>_icmin\_interrupt ( void );
```

**Remark**

\( n \) is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to output compare interrupts.

**Syntax**
```c
void r_<Config_CMT0>_ocmin_interrupt ( void );
```

Remark  
*n* is a channel number.

**Argument(s)**

None.

**Return value**

None.
Usage example

With the timer operating in a one-shot manner:

main.c

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

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

Config_CMT0_user.c

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

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

Table 4.7 API Functions: [Complementary PWM Mode Timer]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>R_&lt;Config_MTU3_MTU4&gt;_Create</code></td>
<td>Executes initialization processing that is required before controlling a timer in complementary PWM mode.</td>
</tr>
<tr>
<td><code>R_&lt;Config_MTU3_MTU4&gt;_Start</code></td>
<td>Starts counting by the counter.</td>
</tr>
<tr>
<td><code>R_&lt;Config_MTU3_MTU4&gt;_Stop</code></td>
<td>Stops counting by the counter.</td>
</tr>
<tr>
<td><code>R_&lt;Config_MTU3_MTU4&gt;_Create_UserInit</code></td>
<td>Executes user-specific initialization processing for a timer in complementary PWM mode.</td>
</tr>
<tr>
<td><code>r_&lt;Config_MTU3_MTU4&gt;_tgimn_interrupt</code></td>
<td>Executes processing in response to compare match interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_MTU3_MTU4&gt;_cj_tgimj_interrupt</code></td>
<td>Executes processing in response to compare match interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_MTU3_MTU4&gt;_cj_tcivj_interrupt</code></td>
<td>Executes processing in response to underflow interrupts.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling a timer in complementary PWM mode.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function stops counting by the counter.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
R_<Config_MTU3_MTU4>_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to compare match interrupts.

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

**[Syntax]**

```c
void r_<Config_MTU3_MTU4>_tginn_interrupt ( void );
```

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare match interrupts.

**Remark**

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

**[Syntax]**

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

**Remark**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare match interrupts.

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

**[Syntax]**

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

**Remark**

\( j \) is channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

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

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

    while (1U)
    {
        nop();
    }
}
```
```c
#define __xxxx_TCDRB_VALUE
#define __xxxx_TDDRB_VALUE

Config_MTU6_MTU7_user.c

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

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

static void r_Config_MTU6_MTU7_tgib6_interrupt(void)
{
    /* Start user code for r_Config_MTU6_MTU7_tgib6_interrupt. Do not edit comment generated here */
    gu_pulse_u += g_pulse_dir_u;
    if(gu_pulse_u == _xxxx_TCDRB_VALUE)
    {
        g_pulse_dir_u = -1;
    }
    else if(gu_pulse_u == _xxxx_TDDRB_VALUE)
    {
        g_pulse_dir_u = 1;
    }
    MTU6.TGRB = gu_pulse_u;
    MTU6.TGRD = gu_pulse_u;
    /* End user code. Do not edit comment generated here */
}
```
4.2.8 Continuous Scan Mode S12AD

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

Table 4.8 API Functions: [Continuous Scan Mode S12AD]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the S12AD in continuous scan mode.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Start</td>
<td>Starts A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Stop</td>
<td>Stops A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Get_ValueResult</td>
<td>Gets the result of conversion.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Set_CompareValue</td>
<td>Sets the compare level.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Set_CompareAValue</td>
<td>Sets the compare level for window A.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Set_CompareBValue</td>
<td>Sets the compare level for window B.</td>
</tr>
<tr>
<td>R_&lt;_Config_S12AD0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the S12AD in continuous scan mode.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_interrupt</td>
<td>Executes processing in response to scan end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_compare_interrupt</td>
<td>Executes processing in response to compare interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptA</td>
<td>Executes processing in response to compare interrupts for window A.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptB</td>
<td>Executes processing in response to compare interrupts for window B.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling the continuous scan mode S12AD.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_S12AD0>_Start**

This API function starts A/D conversion.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_S12AD0>_Stop**

This API function stops A/D conversion.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_{<Config_S12AD0}_Get_ValueResult

This API function gets the result of conversion.

**[Syntax]**

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

**[Argument(s)]**

For RX130 or RX230/RX231

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

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

[Return value]
None.
This API function sets the compare level.

**[Syntax]**

```c
void R_Config_S12AD0_Set_CompareValue ( uint16_t reg_value0, uint16_t reg_value1);
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
**R_Config_S12AD0_Set_CompareAValue**

This API function sets the compare level for window A.

**[Syntax]**

```c
void R_Config_S12AD0_Set_CompareAValue ( uint16_t reg_value0, uint16_t reg_value1);
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
R_<Config_S12AD0>_Set_CompareBValue

This API function sets the compare level for window B.

[Syntax]
void R_<Config_S12AD0>_Set_CompareBValue ( uint16_t reg_value0, uint16_t reg_value1);

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

[Return value]
None.
This API function executes user-specific initialization processing for the continuous scan mode S12AD.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to scan end interrupts.

**[Syntax]**
```c
void r_<Config_S12AD0>_interrupt ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
This API function executes processing in response to compare interrupts.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
`r_<Config_S12AD0>_compare_interruptA`

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare interrupts for window B.

**r_<Config_S12AD0>_compare_interruptB**

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

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

main.c

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

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

Config_S12AD0_user.c

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

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

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

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

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

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

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

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function initializes the CRC calculator in preparation for 16-bit CRC calculation (polynomial: \( X^{16} + X^{15} + X^2 + 1 \)).

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function initializes the CRC calculator in preparation for 16-bit CRC calculation (polynomial: $X^{16} + X^{12} + X^{5} + 1$).

**Syntax**

```c
void R_Config_CRC_SetCCITT ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function initializes the CRC calculator in preparation for 32-bit CRC calculation (polynomial: \( X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1 \)).

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function initializes the CRC calculator in preparation for 32-bit CRC calculation (polynomial: \(X^{32} + X^{28} + X^{27} + X^{26} + X^{25} + X^{23} + X^{22} + X^{20} + X^{19} + X^{18} + X^{14} + X^{13} + X^{11} + X^{10} + X^9 + X^8 + X^6 + 1\)).

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_CRC>_Input_Data

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

[Syntax]

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

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

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t data;</td>
<td>Initial value of the data from which the CRC is to be calculated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint32_t data;</td>
<td>Initial value of the data from which the CRC is to be calculated</td>
</tr>
</tbody>
</table>

Remark  The sizes of the arguments vary with the device group

[Return value]

None.
**R_<Config_CRC>_Get_Result**

This API function gets the result of the operation.

**[Syntax]**

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

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

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint16_t * const result;</td>
<td>Pointer to the location where the result of the operation is to be stored</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint32_t * const result;</td>
<td>Pointer to the location where the result of the operation is to be stored</td>
</tr>
</tbody>
</table>

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

**[Return value]**

None.
Usage example 1

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

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

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

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

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

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

    /*** Transmit "tx_buf" /***/
}
```
Usage example2

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

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

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

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

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

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

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

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

Table 4.10 API Functions: [D/A Convertor]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_DA&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the D/A converter.</td>
</tr>
<tr>
<td>R_&lt;Config_DA&gt;_Start</td>
<td>Starts D/A conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DA&gt;_Stop</td>
<td>Stops D/A conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DA&gt;_Set_ConversionValue</td>
<td>Sets a value for D/A conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DA&gt;_Sync_Start</td>
<td>Starts synchronous D/A conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DA&gt;_Sync_Stop</td>
<td>Stops synchronous D/A conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DA&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the D/A converter.</td>
</tr>
</tbody>
</table>
R_<Config_DA>_Create

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
This API function starts D/A conversion.

**Syntax**

```c
void R_<Config_DA>_n_Start ( void );
```

Remark  

$n$ is a channel number.

**Argument(s)**

None.

**Return value**

None.
This API function stops D/A conversion.

**[Syntax]**

```c
void R_<Config_DA>_n_Stop ( void );
```

**Remark**

$n$ is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function sets a value for D/A conversion.

**Syntax**
```c
void R_<Config_DA>_n_Set_ConversionValue ( uint16_t reg_value );
```

**Remark**

\( n \) is a channel number.

**Argument(s)**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value;</td>
<td>Value to be D/A converted</td>
</tr>
</tbody>
</table>

**Return value**

None.
This API function starts synchronous D/A conversion.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_DA>_Sync_Stop

This API function stops synchronous D/A conversion.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
R_<Config_DA>_Create_UserInit

This API function executes user-specific initialization processing for the D/A converter.
Remark  This API function is called from R_<Config_DA>_Create as a callback routine.

[Syntax]

void R_<Config_DA>_Create_UserInit ( void );

[Argument(s)]
None.

[Return value]
None.
Usage example

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

main.c

```c
#include "r_smc_entry.h"
void main(void)
{
    /* Set the DA0 converter value */
    R_<Config_DA>_0_Set_ConversionValue(1000U);

    /* Set the DA1 converter value */
    R_<Config_DA>_1_Set_ConversionValue(2000U);

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

    while (1U)
    {
        nop();
    }
}
```
4.2.11 Data Operation Circuit

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

Table 4.11 API Functions: [Data Operation Circuit]

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

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
**R_<Config_DOC>_SetMode**

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

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

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

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

**[Syntax]**

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

**[Argument(s)]**

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

**[Return value]**

None.

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

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

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

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

**[Syntax]**

```c
void R_<Config_DOC>_SetMode ( doc_mode_t mode, type value1, type value 2 );
```
### [Argument(s)]

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

### [Return value]

None.
R_<Config_DOC>_WriteData

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

[Syntax]

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

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type data;</td>
<td>Input value for use in the operation, the “type” can be “uint16_t” or “uint32_t”</td>
</tr>
</tbody>
</table>

[Return value]

None.
R_<Config_DOC>_GetResult

This API function gets the result of the operation.

[Syntax]
void R_<Config_DOC>_GetResult ( type * const data );

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>type * const data;</td>
<td>Pointer to the location where the result of the operation is to be stored; the “type” can be “uint16_t” or “uint32_t”</td>
</tr>
</tbody>
</table>

[Return value]
None.
This API function clears the data operation circuit flag.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**R_<Config_DOC>_Create_UserInit**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to data operation circuit interrupts.

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

**[Syntax]**

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

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

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

**[Argument(s)]**
None.

**[Return value]**
None.
Usage example

Adding an array of values in data addition mode and getting the result of addition in response to the interrupt when it has exceeded "FFFFh";
changing the operating mode to data comparison mismatch mode and generating an interrupt when a value other than "000h" is detected in the array:

main.c

```c
#include "r_smc_entry.h"
extern volatile uint16_t data[16];

void main(void)
{
    uint8_t cnt;
    while (1U)
    {
        for (cnt = 0; cnt < 16U; cnt++)
        {
            /* Write new data to compare */
            R_<Config_DOC>_WriteData(data[cnt]);
        }
    }
}
```

Config_DOC_user.c

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

void r_<Config_DOC>_dopci_interrupt(void)
{
    /* Start user code for r_<Config_DOC>_dopci_interrupt. Do not edit comment generated here */
    /* Get result */
    R_<Config_DOC>_GetResult((uint16_t *)&result);
    /* Configure the operation mode of DOC */
    R_<Config_DOC>_SetMode(COMPARE_MISMATCH, 0x0000);
    /* Clear DOPCI flag */
    R_<Config_DOC>_ClearFlag();
    /* End user code. Do not edit comment generated here */
}
```
4.2.12 Data Transfer Controller

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

Table 4.12 API Functions: [Data Transfer Controller]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_DTC&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the data transfer controller.</td>
</tr>
<tr>
<td>R_&lt;Config_DTC&gt;_Start</td>
<td>Enables activation of the data transfer controller.</td>
</tr>
<tr>
<td>R_&lt;Config_DTC&gt;_Stop</td>
<td>Disables activation of the data transfer controller.</td>
</tr>
<tr>
<td>R_&lt;Config_DTC&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the data transfer controller.</td>
</tr>
</tbody>
</table>
R_<Config_DTC>_Create

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
**R_<Config_DTC>_Start**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_DTC>_Stop

This API function disables activation of the data transfer controller.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
4.2.12.1  R_<Config_DTC>_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
4.2.12.2 Usage example

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

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

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

    while (1U)
    {
        nop();
    }
}
```
4.2.13 Dead Time Compensation Counter

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

Table 4.13 API Functions: [Dead Time Compensation Counter]

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

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_MTU5>_U5_Start**

This API function starts U phase counting by the counter.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
R_<Config_MTU5>_U5_Stop

This API function stops U phase counting by the counter.

[Syntax]
void R_<Config_MTU5>_U5_Stop ( void );

[Argument(s)]
None.

[Return value]
None.
This API function starts V phase counting by the counter.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**R_<Config_MTU5>_V5_Stop**

This API function stops V phase counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_MTU5>_W5_Start

This API function starts W phase counting by the counter.

[Syntax]
void R_<Config_MTU5>_W5_Start ( void );

[Argument(s)]
None.

[Return value]
None.
**R_<Config_MTU5>_W5_Stop**

This API function stops W phase counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_MTU5>_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
This API function executes processing in response to input capture interrupts.

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

**[Syntax]**

```c
void r_<Config_MTUS>_tginmn_interrupt ( void );
```

Remark    \( n \) is a channel number and \( m \) is the number of a timer general register.

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

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

main.c

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

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

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

Config_MTU5_user.c

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

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

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

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

**[Syntax]**

```c
void R_Config_DMAC0_Create ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts the DMAC waiting for DMA activation triggers.

[Syntax]
```c
void R_<Config_DMAC0>_Start ( void );
```

[Argument(s)]
None.

[Return value]
None.
**R_<Config_DMAC0>_Stop**

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function sets a software transfer request.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_DMAC0>_Clear_SoftwareTrigger

This API function clears a software transfer request.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
R_{Config\_DMAC0\_Create\_UserInit}

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

Remark This API function is called from R_{Config\_DMAC0\_Create} as a callback routine.

[Syntax]

```c
void R_{Config\_DMAC0\_Create\_UserInit} ( void );
```

[Argument(s)]

None.

[Return value]

None.
This API function executes processing in response to transfer end interrupts from channel n.

Syntax

void r_<Config_DMAC0>_dmacni_interrupt ( void );

Remark

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

[Argument(s)]

None.

[Return value]

None.
This API function executes processing in response to transfer end interrupts from channel n.

**Remark**  This API function is called as a callback routine from \texttt{r\_<Config\_DMAC0\>_dmacni\_interrupt}, which is the interrupt handler for transfer end interrupts from channel n.

**[Syntax]**

\begin{verbatim}
void r_dmacn_callback_transfer_end ( void );
\end{verbatim}

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to transfer escape end interrupts from channel \( n \).

**Remark**  This API function is called as a callback routine from \( r_{<Config\_DMAC0>_\_dmacn\_interrupt} \), which is the interrupt handler for transfer escape end interrupts from channel \( n \).

### [Syntax]

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

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

### [Argument(s)]

- None.

### [Return value]

- None.
This API function executes processing in response to transfer end interrupts from channels 4 to 7.

[Syntax]
```c
void r_dmac_dmac74i_interrupt ( void );
```

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to transfer end interrupts from channel \( n \).

Remark  This API function is called as a callback routine from \( r_{\text{dmac}} \_\text{dmac74i\_interrupt} \), which is the interrupt handler for transfer end interrupts from channel \( n \).

\[ \text{Syntax} \]

\[
\text{void } r_{\text{dmacn\_callback\_transfer\_end}} ( \text{ void } );
\]

Remark  \( n \) is a channel number. \( (n = 4 \text{ to } 7) \)

\[ \text{Argument(s)} \]

None.

\[ \text{Return value} \]

None.
This API function executes processing in response to transfer escape end interrupts from channel n.

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

### Syntax

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

**Remark**

- n is a channel number. \((n = 4 \text{ to } 7)\)

### Argument(s)

- None.

### Return value

- None.
Usage example

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

```c
#include "r_smc_entry.h"

void main(void)
{
    /* Start CMT channel 0 counter */
    R_Config_CMT0_Start();

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

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

```c
Config_DMAC0_user.c

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

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

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

```c
Config_DMAC0.h

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

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

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

#### Table 4.15 API Functions: [Event Link Controller]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_Create}</td>
<td>Executes initialization processing that is required before controlling the event link controller.</td>
</tr>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_Start}</td>
<td>Starts interlinked operation of peripheral functions.</td>
</tr>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_Stop}</td>
<td>Stops interlinked operation of peripheral functions.</td>
</tr>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_GenerateSoftwareEvent}</td>
<td>Generates a software event.</td>
</tr>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_Set_PortBuffern}</td>
<td>Sets the value of a port buffer.</td>
</tr>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_Get_PortBuffern}</td>
<td>Gets the value of a port buffer.</td>
</tr>
<tr>
<td>\texttt{R_&lt;Config_ELC&gt;_Create_UserInit}</td>
<td>Executes user-specific initialization processing for the event link controller.</td>
</tr>
<tr>
<td>\texttt{r_&lt;Config_ELC&gt;_elsmi_interrupt}</td>
<td>Executes processing in response to event link interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_ELC>_Create**

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

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

**[Syntax]**
```c
void R_<Config_ELC>_Create ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
This API function starts interlinked operation of peripheral functions.

[Syntax]

```c
void R_Config_ELC_Start ( void );
```

[Argument(s)]
None.

[Return value]
None.
This API function stops interlinked operation of peripheral functions.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function generates a software event.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_ELC>_Set_PortBuffer**

This API function sets the value of a port buffer.

**[Syntax]**

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

**Remark**  
$n$ is a port number.

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t  value;</td>
<td>Value to be written to the port buffer</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
**R_<Config_ELC>_Get_PortBuffer**

This API function gets the value of a port buffer.

**[Syntax]**

```
void R_<Config_ELC>_Get_PortBuffern ( uint8_t * const value );
```

Remark  

\( n \) is a port number.

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const value;</td>
<td>Pointer to the location where the read value is to be stored</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
R_<Config_ELC>_Create_UserInit

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

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

[Syntax]
void R_<Config_ELC>_Create_UserInit ( void );

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to event link interrupts.

[Syntax]
```
void r_<Config_ELC>_elsrn_interrupt ( void );
```

Remark

\( n \) is the number of an event link setting register.

[Argument(s)]

None.

[Return value]

None.
Usage example

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

main.c

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

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

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

Config_ELC_user.c

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

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

### Table 4.16 API Functions: [General PWM Timer (GPT)]

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_GPT0>_Start

This API function starts counting by the counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
R_<Config_GPT0>_Stop

This API function stops counting by the counter.

[Syntax]

void R_<Config_GPT0>_Stop ( void );

[Argument(s)]

None.

[Return value]

None.
R_<Config_GPT0>_HardwareStart

This API function enables interrupts.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
**R_<Config_GPT0>_HardwareStop**

This API function disables interrupts.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function enables interrupts due to the input of a falling-edge or rising-edge external trigger.

**Syntax**

```c
void R_Config_GPT0_ETGI_Start ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function disables interrupts due to the input of a falling-edge or rising-edge external trigger.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_GPT0>_Software_Clear

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
R_<Config_GPT0>_Create_UserInit

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

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

[Syntax]

void     R_<Config_GPT0>_Create_UserInit ( void );

[Argument(s)]

None.

[Return value]

None.
This API function executes processing in response to input capture interrupts or compare match interrupts.

**[Syntax]**

```c
void r_<Config_GPT0>_gtcimn_interrupt ( void );
```

**Remark**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to overflow interrupts.

**[Syntax]**

```c
void r_<Config_GPT0>_gtcivn_interrupt ( void );
```

**Remark**

\( n \) is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to underflow interrupts.

**[Syntax]**

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

**Remark**  
$n$ is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to dead time error interrupts.

**Syntax**

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

*Remark*  
$n$ is a channel number.

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to interrupts due to the input of a falling-edge external trigger.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to interrupts due to the input of a rising-edge external trigger.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Acquiring a captured value:

main.c

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

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

Config_GPT0_user.c

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

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

```c
#include "r_smc_entry.h"

void main(void)
{
    /* Start GPT channel 0 counter */
    R_Config_GPT0_Start();

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

```c
volatile uint16_t g_gpt0_capture_value;

static void r_Config_GPT0_gtcia0_interrupt(void)
{
    /* Start user code for r_Config_GPT0_gtcia0_interrupt. Do not edit comment generated here */
    g_gpt0_capture_value = GPT0.GTCCRA;
    /* End user code. Do not edit comment generated here */
}
```
### 4.2.17 Group Scan Mode S12AD

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

**Table 4.17 API Functions: [Group Scan Mode S12AD]**

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

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
R_<Config_S12AD0>_Start

This API function starts A/D conversion.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function stops A/D conversion.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_S12AD0>_Get_ValueResult

This API function gets the result of conversion.

**[Syntax]**

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

**[Argument(s)]**

For RX130 or RX230/RX231

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

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

[Return value]  
None.
This API function sets the compare level.

**[Syntax]**

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

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
R_<Config_S12AD0>_Set_CompareAValue

This API function sets the compare level for window A.

**[Syntax]**
```c
void R_<Config_S12AD0>_Set_CompareAValue ( uint16_t reg_value0, uint16_t reg_value1);
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

**[Return value]**
None.
R_<Config_S12AD0>_Set_CompareBValue

This API function sets the compare level for window B.

[Syntax]
void R_<Config_S12AD0>_Set_CompareBValue ( uint16_t reg_value0, uint16_t reg_value1);

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

[Return value]
None.
R_<Config_S12AD0>_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to scan end interrupts.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare interrupts.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare interrupts for window A.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to compare interrupts for window B.

[Syntax]
```c
void r_<Config_S12AD0>_compare_interruptB ( void );
```

[Argument(s)]
None.

[Return value]
None.
r_<Config_S12AD0>_groupb_interrupt

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
r_<Config_S12AD0>_groupc_interrupt

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

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

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

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

```
Config_S12AD0_user.c
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_s12ad0_ch000_value;
volatile uint16_t g_s12ad0_ch001_value;
/* End user code. Do not edit comment generated here */

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

static void r_Config_S12AD0_groupb_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_groupb_interrupt. Do not edit comment generated here */
    /* Get result from the AD0 channel 1 (AN001) converter */
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, (uint16_t *)&g_s12ad0_ch001_value);
    /* End user code. Do not edit comment generated here */
}
### 4.2.18 I2C Master Mode

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

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

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_Create</code></td>
<td>Executes initialization processing that is required before controlling I2C communications in master mode.</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_Start</code></td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_Stop</code></td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_Master_Send</code></td>
<td>Starts master transmission.</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_Master_Send_Without_Stop</code></td>
<td>Starts master transmission (with no stop condition being issued at the end of transmission).</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_Master_Receive</code></td>
<td>Starts master reception.</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_IIC_StartCondition</code></td>
<td>Issues a start condition.</td>
</tr>
<tr>
<td><code>R_&lt;Config_RIIC0&gt;_IIC_StopCondition</code></td>
<td>Issues a stop condition.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_error_interrupt</code></td>
<td>Executes processing in response to communication error interrupts or communication event generation interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_receive_interrupt</code></td>
<td>Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_transmit_interrupt</code></td>
<td>Executes processing in response to transmit data empty interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_transmitend_interrupt</code></td>
<td>Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_callback_error</code></td>
<td>Executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_callback_transmitend</code></td>
<td>Executes processing specific to the detection of a stop condition in master transmission among the sources of communication error interrupts or communication event generation interrupts. In master transmission without issuing a stop condition, this API function executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_RIIC0&gt;_callback_receiveend</code></td>
<td>Executes processing specific to the detection of a stop condition in master reception among the sources of communication error interrupts or communication event generation interrupts.</td>
</tr>
<tr>
<td>API Function Name</td>
<td>Function</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling I2C communications in master mode.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Start</td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Stop</td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_IIC_Master_Send</td>
<td>Starts master transmission.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_IIC_Master_Receive</td>
<td>Starts master reception.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_IIC_StartCondition</td>
<td>Issues a start condition.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_IIC_StopCondition</td>
<td>Issues a stop condition.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for I2C communications in master mode.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
<td>Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
<td>Executes processing in response to transmit data empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_transmitend_interrupt</td>
<td>Executes processing in response to interrupts on completion of generation of a start condition/restart condition/stop condition.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
<td>Executes processing specific to the detection of a stop condition in master transmission among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition. When the transmit data empty interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
<td>Executes processing specific to the detection of a stop condition in master reception among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition. When the receive data full interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.</td>
</tr>
</tbody>
</table>
R_<Config_RIIC0>_Create

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

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

[Syntax]

    void     R_<Config_RIIC0>_Create ( void );

[Argument(s)]

  None.

[Return value]

  None.
This API function starts serial communications.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_RIIC0>_Stop**

This API function stops serial communications.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_RIIC0>_Master_Send

This API function starts master transmission.

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

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

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

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

[Syntax]

```c
MD_STATUS  R_<Config_RIIC0>_Master_Send ( uint16_t  \text{adr}, uint8_t * const \text{tx_buf}, uint16_t \text{tx_num} );
```

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t \text{adr};</td>
<td>Slave address [15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]</td>
</tr>
<tr>
<td>I</td>
<td>uint8_t * const \text{tx_buf};</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t \text{tx_num};</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>The bus is busy.</td>
</tr>
<tr>
<td>MD_ERROR2</td>
<td>The specification of argument \text{adr} is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_RIIC0>_Master_Send_Without_Stop

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

Remark1. This API function executes the master transmission of the slave address specified by the argument \(adr\) and the R/W# bit to slave devices, and then repeats the master transmission of single bytes of data from the buffer specified by the argument \(tx\_buf\) the number of times specified by the argument \(tx\_num\).

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

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

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

[Syntax]

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

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t adr;</td>
<td>Slave address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![Slave address (0 to 1023)]</td>
</tr>
<tr>
<td>I</td>
<td>uint8_t * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>The bus is busy.</td>
</tr>
<tr>
<td>MD_ERROR2</td>
<td>The specification of argument (adr) is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_RIIC0>_Master_Receive

This API function starts master reception.

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

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

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

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

**[Syntax]**

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

**[Argument(s)]**

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

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR2</td>
<td>The specification of argument <code>adr</code> is invalid.</td>
</tr>
<tr>
<td>MD_ERROR3</td>
<td>The specification of argument <code>adr</code> is invalid (10-bit addresses are not supported in master reception).</td>
</tr>
<tr>
<td>MD_ERROR4</td>
<td>The bus is busy (timeout or loss in arbitration has been detected).</td>
</tr>
<tr>
<td>MD_ERROR5</td>
<td>The bus is busy (no stop condition has been detected).</td>
</tr>
</tbody>
</table>
This API function issues a start condition.

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

**[Syntax]**

\begin{verbatim}
void R_<Config_RIIC0>_IIC_StartCondition ( void );
\end{verbatim}

**[Argument(s)]**

None.

**[Return value]**

None.
This API function issues a stop condition.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_RIIC0>_Create_UserInit

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

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

[Syntax]
void R_<Config_RIIC0>_Create_UserInit ( void );

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to communication error interrupts or communication event generation interrupts.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
r_<Config_RIIC0>_receive_interrupt

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to transmit data empty interrupts.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to transmit end interrupts.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.

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

**[Syntax]**
```c
void r_<Config_RIIC0>_callback_error ( MD_STATUS status );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| I   | MD_STATUS status; | Interrupt sources
|     |                 | MD_ERROR1: Detection of loss in arbitration                      |
|     |                 | MD_ERROR2: Detection of timeout                                  |
|     |                 | MD_ERROR3: Detection of NACK                                      |
|     |                 | MD_ERROR4: Detection of communication sequence error             |

**[Return value]**

None.
This API function executes processing specific to the detection of a stop condition in master transmission among the sources of communication error interrupts or communication event generation interrupts.

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

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

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

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
r_<Config_RIIC0>_callback_receiveend

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

Remark1. This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.
Remark2. To execute master reception, call R_<Config_RIIC0>_IIC_Master_Receive.

[Syntax]
void r_<Config_RIIC0>_callback_receiveend ( void );

[Argument(s)]
None.

[Return value]
None.
R_<Config_SCI0>_Create

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts serial communications.

[Syntax]
```c
void R_<Config_SCI0>_Start ( void );
```

[Argument(s)]
None.

[Return value]
None.
This API function stops serial communications.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
R_<Config_SCI0>_IIC_Master_Send

This API function starts master transmission.

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

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

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

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

[Syntax]

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

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t <code>adr</code>;</td>
<td>Slave address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slave Address (0 to 127)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(0)</td>
</tr>
<tr>
<td>I</td>
<td>uint8_t <code>tx_buf</code>;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t <code>tx_num</code>;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

[Return value]

None.
This API function starts master reception.

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

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

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

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

**[Syntax]**

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

**[Argument(s)]**

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

**[Return value]**

None.
**R_<Config_SCI0>_IIC_StartCondition**

This API function issues a start condition.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
### R_<Config_SCI0>_IIC_StopCondition

This API function issues a stop condition.

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

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
R_<Config_SCI0>_Create_UserInit

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

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
This API function executes processing in response to receive data full interrupts.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to transmit data empty interrupts.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to interrupts on completion of generation of a start condition/restart condition/stop condition.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function executes processing specific to the detection of a stop condition in master transmission among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition.

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

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

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

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

### Syntax

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

### Argument(s)

None.

### Return value

None.
This API function executes processing specific to the detection of a stop condition in master reception among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition.

Remark1. This API function is called from r_Config_RIICn_transmitend_interrupt as a callback routine.
Remark2. To execute master reception, call R_<Config_SCI0>_IIC_Master_Receive.

When the receive data full interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.
Remark3. This API function is called from r_<Config_SCI0>_receive_interrupt as a callback routine.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example 1

For RIIC, repeating master transmission four times:

```c
#include "r_smc_entry.h"
extern volatile uint8_t g_riic0_tx_buf[2];

void main(void)
{
    /* Start the RIIC0 I2C Bus Interface */
    R_Config_RIIC0_Start();

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

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

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

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

static void r_Config_RIIC0_callback_transmitend(void)
{
    /* Start user code for r_Config_RIIC0_callback_transmitend. Do not edit comment generated here */
    if (++g_riic0_tx_cnt < 4U)
    {
        g_riic0_tx_buf[0] = g_riic0_tx_cnt;
        g_riic0_tx_buf[1] += 0x01;
        /* Send RIIC0 data to slave device [Slave assress : 160 (10-bit address mode)] */
        R_Config_RIIC0_Master_Send(0x00A0, (uint8_t *)g_riic0_tx_buf, 2U);
    }
    else
    {
        /* Stop the RIIC0 I2C Bus Interface */
        R_Config_RIIC0_Stop();
    }
    /* End user code. Do not edit comment generated here */
}
```
Usage example 2
For SCI simple I2C mode, repeating master transmission four times:

**main.c**

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

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

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

**Config_SCI0_user.c**

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_sci0_tx_buf[2];
volatile uint8_t g_sci0_tx_cnt;
/* End user code. Do not edit comment generated here */
void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_cnt = 0U;
    g_sci0_tx_buf[0] = g_riic0_tx_cnt;
    g_sco0_tx_buf[1] = 0x01;
    /* End user code. Do not edit comment generated here */
}
static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated here */
    if (++g_sci0_tx_cnt < 4U)
    {
        g_sci0_tx_buf[0] = g_sci0_tx_cnt;
        g_sci0_tx_buf[1] += 0x01;

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

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

Table 4.20 API Functions: [I2C Slave Mode]

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

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts serial communications.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function stops serial communications.

[Syntax]
```c
void R_<Config_RIIIC0>_Stop ( void );
```

[Argument(s)]
None.

[Return value]
None.
**R_<Config_RIIC0>_Slave_Send**

This API function starts slave transmission.

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

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

**[Syntax]**

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

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
</tbody>
</table>
R_<Config_RIIC0>_Slave_Receive

This API function starts slave reception.

Remark1. This API function repeats the slave reception of single bytes of data the number of times specified by the argument \( rx\_num \), storing the data in the buffer specified by the argument \( rx\_buf \).

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

[Syntax]

\[
\text{MD\_STATUS} \quad \text{R}_{\_}\text{<Config\_RIIC0>\_Slave\_Receive} \left( \text{uint8\_t * const } rx\_buf, \text{uint16\_t } rx\_num \right);
\]

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t rx_num;</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
</tbody>
</table>
This API function executes user-specific initialization processing for I2C communications in slave mode.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
r_<Config_RIIC0>_error_interrupt

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to receive data full interrupts.

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to transmit data empty interrupts.

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to transmit end interrupts.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.

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

[Syntax]
```c
void r_<Config_RIIC0>_callback_error ( MD_STATUS status );
```

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>MD_STATUS status;</td>
<td>Interrupt sources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MD_ERROR1: Detection of loss in arbitration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MD_ERROR2: Detection of timeout</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MD_ERROR3: Detection of NACK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MD_ERROR4: Detection of communication sequence error</td>
</tr>
</tbody>
</table>

[Return value]
None.
This API function executes processing specific to the detection of a stop condition in slave transmission among the sources of communication error interrupts or communication event generation interrupts.

Remark1. This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.
Remark2. To execute slave transmission, call R_<Config_RIIC0>_Slave_Send.

**[Syntax]**

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

**[Argument(s)]**
None.

**[Return value]**
None.
This API function executes processing specific to the detection of a stop condition in slave reception among the sources of communication error interrupts or communication event generation interrupts.

**Remark1.** This API function is called from \( r_{<Config\_RIIC0>\_error\_interrupt} \) as a callback routine.

**Remark2.** To execute slave reception, call \( R_{<Config\_RIIC0>\_Slave\_Receive} \).

**[Syntax]**

\[
\text{void } r_{<Config\_RIIC0>\_callback\_receiveend}(\text{void});
\]

**[Argument(s)]**

None.

**[Return value]**

None.
4.2.19.1 Usage example

Repeating slave reception four times:

main.c

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

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

    while (1U)
    {
        nop();
    }
}
```
Config_RIIC2_user.c

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

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

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

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

Table 4.21 API Functions: [Interrupt Controller]

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

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

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

[Syntax]
void R_<Config_ICU>_Create ( void );

[Argument(s)]
None.

[Return value]
None.
This API function enables the detection of an external pin interrupt.

**[Syntax]**

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

Remark: \( n \) is the number of an IRQ pin.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function disables the detection of an external pin interrupt.

**[Syntax]**
void R_<Config_ICU>_IRQn_Stop ( void );

Remark  \( n \) is the number of an IRQ pin.

**[Argument(s)]**
None.

**[Return value]**
None.
This API function enables the detection of a software interrupt.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function disables the detection of a software interrupt.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_ICU>_SoftwareInterrupt_Generate**

This API function generates a software interrupt.

Remark  
\( r_{<Config_ICU>\_software\_interrupt} \) is called in response to this API function.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_ICU>_Software2_Start

This API function enables the detection of software interrupt 2.

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
R_<Config_ICU>_Software2_Stop

This API function disables the detection of software interrupt 2.

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
**R\_<Config_ICU>_SoftwareInterrupt2_Generate**

This API function generates software interrupt 2.

**Remark**  
\textit{r\_<Config_ICU>_software2_interrupt} is called in response to this API function.

**Syntax**

\[
\text{void } \text{R\_<Config_ICU>_SoftwareInterrupt2_Generate ( void );}
\]

**Argument(s)**

None.

**Return value**

None.
This API function executes user-specific initialization processing for the interrupt controller.

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

[Syntax]

```c
void R_<Config.ICU>_Create_UserInit ( void );
```

[Argument(s)]
None.

[Return value]
None.
r_<Config_ICU>_irqn_interrupt

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

**[Syntax]**

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

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

**[Argument(s)]**

None.

**[Return value]**

None.
r_<Config_ICU>_software_interrupt

This API function executes processing in response to software interrupts.

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to software interrupts 2.

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

### Syntax

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

### Argument(s)

None.

### Return value

None.
This API function executes processing in response to NMI pin interrupts.

[Syntax]
```c
void r_<Config_ICU>_nmi_interrupt ( void );
```

[Argument(s)]
None.

[Return value]
None.
Usage example

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

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

    /* Enable sleep mode */
    R_Config_LPC_Sleep();

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

```c
#include "r_smc_entry.h"
/* Start user code for include. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */
static void r_Config_ICU_irq0_interrupt(void)
{
    /* Start user code for r_Config_ICU_irq0_interrupt. Do not edit comment generated here */
    /* Allow sleep mode return clock to be changed */
    R_Config_LPC_ChangeSleepModeReturnClock(RETURN_MAIN_CLOCK);
    /* End user code. Do not edit comment generated here */
}
```
### 4.2.21 Low Power Consumption

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

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

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

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

**[Syntax]**
```c
void R_<Config_LPC>_Create ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
R_<Config_LPC>_AllModuleClockStop

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
</tbody>
</table>
R_<Config_LPC>_Sleep

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

[Syntax]

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

[Argument(s)]
None.

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>Abnormal end</td>
</tr>
</tbody>
</table>
R_<Config_LPC>_DeepSleep

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
</tbody>
</table>
### R_<Config_LPC>_SoftwareStandby

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

#### [Syntax]

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

#### [Argument(s)]

None.

#### [Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>Abnormal end</td>
</tr>
</tbody>
</table>
This API function places the MCU in deep software standby mode as a low-power state.

**Syntax**

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

**Argument(s)**

None.

**Return value**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
</tbody>
</table>
R<Config_LPC>_ChangeOperatingPowerControl

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

**Syntax**

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

**Argument(s)**

For RX130 or RX230/RX231

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>operating_mode_t mode;</td>
<td>Operating power control mode type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HIGH_SPEED: High-speed operating mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIDDLE_SPEED: Medium-speed operating mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW_SPEED: Low-speed operating mode</td>
</tr>
</tbody>
</table>

For other devices

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>operating_mode_t mode;</td>
<td>Operating power control mode type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HIGH_SPEED: High-speed operating mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW_SPEED1: Low-speed operating mode 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW_SPEED2: Low-speed operating mode 2</td>
</tr>
</tbody>
</table>

**Return value**

For RX130 or RX230/RX231

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>The transition to the low-speed operating mode did not end normally.</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument mode is invalid.</td>
</tr>
</tbody>
</table>

For other devices

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>The transition to the low-speed operating mode 1 did not end normally.</td>
</tr>
<tr>
<td>MD_ERROR2</td>
<td>The transition to the low-speed operating mode 2 did not end normally.</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument mode is invalid.</td>
</tr>
</tbody>
</table>
**R_<Config_LPC>_ChangeSleepModeReturnClock**

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

**[Syntax]**

```c
MD_STATUS R_<Config_LPC>_ChangeSleepModeReturnClock ( return_clock_t clock );
```

**[Argument(s)]**

For RX130 or RX230/RX231

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>return_clock_t clock;</td>
<td>Clock source type</td>
</tr>
<tr>
<td></td>
<td>RETURN_LOCO:</td>
<td>Low-speed on-chip oscillator</td>
</tr>
<tr>
<td></td>
<td>RETURN_HOCO:</td>
<td>High-speed on-chip oscillator</td>
</tr>
<tr>
<td></td>
<td>RETURN_MAIN_CLOCK:</td>
<td>Main clock oscillator</td>
</tr>
<tr>
<td></td>
<td>RETURN_DISABLE:</td>
<td>The clock source is not switched.</td>
</tr>
</tbody>
</table>

For other devices

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>return_clock_t clock;</td>
<td>Clock source type</td>
</tr>
<tr>
<td></td>
<td>RETURN_HOCO:</td>
<td>High-speed on-chip oscillator</td>
</tr>
<tr>
<td></td>
<td>RETURN_MAIN_CLOCK:</td>
<td>Main clock oscillator</td>
</tr>
<tr>
<td></td>
<td>RETURN_DISABLE:</td>
<td>The clock source is not switched.</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>The specification of the clock source that is currently set is invalid.</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument clock is invalid.</td>
</tr>
</tbody>
</table>
This API function executes user-specific initialization processing for the low power consumption.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function sets the PGA operating condition setting (VOLSR,PGAVLS bit).

**[Syntax]**

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

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t pgavls_bit;</td>
<td>PGAVLS bit value&lt;br&gt; 0u : The AVCC voltage is at least 4.0 V,&lt;br&gt;and the pseudo-differential input of the PGAs are enabled and negative voltages is to be input to the pins.&lt;br&gt;1u : The AVCC voltage is lower than 4.0 V, or negative voltages are not to be input to the pins.</td>
</tr>
</tbody>
</table>

None.

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>Abnormal end</td>
</tr>
</tbody>
</table>
Usage example

Refer to "Example" in Interrupt Controller.
4.2.22 Low Power Timer

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

Table 4.23 API Functions: [Low Power Timer]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_LPT&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the low-power timer.</td>
</tr>
<tr>
<td>R_&lt;Config_LPT&gt;_Start</td>
<td>Starts counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_LPT&gt;_Stop</td>
<td>Stops counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_LPT&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the low-power timer.</td>
</tr>
</tbody>
</table>
**R_<Config_LPT>_Create**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_LPT>_Stop

This API function stops counting by the counter.

[Syntax]
void R_<Config_LPT>_Stop ( void );

[Argument(s)]
None.

[Return value]
None.
**R_<Config_LPT>_Create_UserInit**

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

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

**[Syntax]**
```c
void R_<Config_LPT>_Create_UserInit ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
Usage example

None.
4.2.23 Normal Mode Timer

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

Table 4.24 API Functions: [Normal Mode Timer]

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

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_MTU0>_Start

This API function starts counting by the counter.

[Syntax]
void R_<Config_MTU0>_Start ( void );

[Argument(s)]
None.

[Return value]
None.
R_<Config_MTU0>_Stop

This API function stops counting by the counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
**R_<Config_MTU0>_Create_UserInit**

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

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to input capture or compare match interrupts.

**[Syntax]**

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

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

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

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to overflow interrupts.

**[Syntax]**

```c
void r_<Config_MTU0>_tcivn_interrupt ( void );
```

```c
void r_<Config_MTU0>_tcinv_interrupt ( void );
```

**Remark1.** \( n \) is a channel number.

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

With the timer operating in a one-shot manner:

main.c

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

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

<Config_MTU0_user.c

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

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

**Table 4.25 API Functions: [Phase Counting Mode Timer]**

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_MTU1&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the phase counting mode timer (MTU or TPU).</td>
</tr>
<tr>
<td>R_&lt;Config_MTU1&gt;_Start</td>
<td>Starts counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_MTU1&gt;_Stop</td>
<td>Stops counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_MTU1_MTU2&gt;_MTU_Start</td>
<td>Starts all MTU channels’ counters simultaneously for 32-bit cascading operation</td>
</tr>
<tr>
<td>R_&lt;Config_MTU1_MTU2&gt;_MTU_Stop</td>
<td>Stops all MTU channels’ counters simultaneously for 32-bit cascading operation</td>
</tr>
<tr>
<td>R_&lt;Config_MTU1&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the phase counting mode timer (MTU or TPU).</td>
</tr>
<tr>
<td>r_&lt;Config_MTU1&gt;_tgimn_interrupt</td>
<td>Executes processing in response to input capture or compare match interrupts.</td>
</tr>
<tr>
<td>(The name of this API function varies with the resource.)</td>
<td></td>
</tr>
<tr>
<td>r_&lt;Config_MTU1&gt;_tginm_interrupt</td>
<td>(The name of this API function varies with the resource.)</td>
</tr>
<tr>
<td>r_&lt;Config_MTU1&gt;_tcivn_interrupt</td>
<td>Executes processing in response to overflow interrupts.</td>
</tr>
<tr>
<td>(The name of this API function varies with the resource.)</td>
<td></td>
</tr>
<tr>
<td>r_&lt;Config_MTU1&gt;_tcinv_interrupt</td>
<td>Executes processing in response to underflow interrupts.</td>
</tr>
<tr>
<td>(The name of this API function varies with the resource.)</td>
<td></td>
</tr>
<tr>
<td>r_&lt;Config_MTU1&gt;_tcinvu_interrupt</td>
<td></td>
</tr>
</tbody>
</table>
**R_<Config_MTU1>_Create**

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

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

**[Syntax]**

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

**[Argument(s)]**
None.

**[Return value]**
None.
R_<Config_MTU1>_Start

This API function starts counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_MTU1>_Stop**

This API function stops counting by the counter.

**[Syntax]**

```c
void R_<Config_MTU1>_Stop ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts MTU channels’ counter simultaneously for 32-bit cascading operation, other channels related to the ones used for cascading operation are configurable via cascading mode GUI.

**[Syntax]**

```c
void R_<Config_MTU1_MTU2>_MTU_Start ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_MTU1_MTU2>_MTU_Stop

This API function stops MTU channels’ counter simultaneously for 32-bit cascading operation, other channels related to the ones used for cascading operation are configurable via cascading mode GUI.

[Syntax]

```
void R_<Config_MTU1_MTU2>_MTU_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.
**R_<Config_MTU1>_Create_UserInit**

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

**Remark**  This API function is called from **R_<Config_MTU1>_Create** as a callback routine.

**[Syntax]**

```c
void R_<Config_MTU1>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to input capture or compare match interrupts.

[Syntax]
```c
void r_<Config_MTU1>_tgi\(n\)_interrupt ( void );
void r_<Config_MTU1>_tgim\(m\)_interrupt ( void );
```

Remark1. \(n\) is a channel number and \(m\) is the number of a timer general register.
Remark2. The name of this API function varies with the resource.

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to overflow interrupts.

**[Syntax]**

```c
void r_<Config_MTU1>_tcivn_interrupt ( void );
void r_<Config_MTU1>_tcinv_interrupt ( void );
```

Remark1. \( n \) is a channel number.
Remark2. The name of this API function varies with the resource.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to underflow interrupts.

**[Syntax]**

```c
void r_<Config_MTU1>_tciniu_interrupt ( void );
```

Remark1. \( n \) is a channel number.
Remark2. The name of this API function varies with the resource.

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Determining the direction of rotation from the 2-phase pulses input from a rotary encoder:

main.c

```c
#include "r_smc_entry.h"
void main(void)
{
  /* Start the MTU1 channel counter */
  R_Config_MTU1_Start();

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

Config_MTU1_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_mtu1_dir;
/* End user code. Do not edit comment generated here */

void R_Config_MTU1_Create_UserInit(void)
{
  /* Start user code for user init. Do not edit comment generated here */
  /* Clear state */
  g_mtu1_dir = 0U;
  /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU1_tgia1_interrupt(void)
{
  /* Start user code for r_Config_MTU1_tgia1_interrupt. Do not edit comment generated here */
  /* Set CW state */
  g_mtu1_dir = 1U;
  /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU1_tgib1_interrupt(void)
{
  /* Start user code for r_Config_MTU1_tgib1_interrupt. Do not edit comment generated here */
  /* Set CCW state */
  g_mtu1_dir = 2U;
  /* End user code. Do not edit comment generated here */
}```
4.2.25 Port Output Enable

The Code Generator outputs the following API functions for the port output enable module.

Table 4.26 API Functions: [Port Output Enable]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.&lt;Config_POE&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the port output enable module.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Start</td>
<td>Enables output enable interrupts.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Stop</td>
<td>Disables output enable interrupts.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Set_HiZ_MTUn</td>
<td>Places the pins of the MTUn in the high-impedance state.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Clear_HiZ_MTUn</td>
<td>Releases the pins of the MTUn from the high-impedance state.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Set_HiZ_GPTn</td>
<td>Places the pins of the GPTn in the high-impedance state.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Clear_HiZ_GPTn</td>
<td>Releases the pins of the GPTn from the high-impedance state.</td>
</tr>
<tr>
<td>R.&lt;Config_POE&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the port output enable module.</td>
</tr>
<tr>
<td>r.&lt;Config_POE&gt;_oein_interrupt</td>
<td>Executes processing in response to output enable interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_POE>_Create**

This API function executes initialization processing that is required before controlling the port output enable module.

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

**[Syntax]**

```c
void R_<Config_POE>_Create ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function enables output enable interrupts.

**Syntax**

```c
void R_<Config_POE>_Start ( void );
```

**Argument(s)**

None.

**Return value**

None.
R_<Config_POE>_Stop

This API function disables output enable interrupts.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
This API function places the pins of the MTUn in the high-impedance state.

[Syntax]

```c
void R_Config_POE_Set_HiZ_MTUn ( void );
```

[Remark]  
$n$ is a channel number.

[Argument(s)]
None.

[Return value]
None.
This API function releases the pins of the MTUn from the high-impedance state.

**[Syntax]**

```c
void R_Config_POE_Clear_HiZ_MTUn ( void );
```

**Remark**

*n* is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function places the pins of the GPTn in the high-impedance state.

**[Syntax]**

```c
void R_<Config_POE>_Set_HiZ_GPTn ( void );
```

**Remark**

$n$ is a channel number.

**[Argument(s)]**

None.

**[Return value]**

None.
This API function releases the pins of the GPTn from the high-impedance state.

**Syntax**

```c
void R_Config_POE_Clear_HiZ_GPTn ( void );
```

Remark  

\( n \) is a channel number.

**Argument(s)**

None.

**Return value**

None.
This API function executes user-specific initialization processing for the port output enable module.

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

**[Syntax]**

```c
void R_<Config_POE>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to output enable interrupts.

**Syntax**

```c
void r_<Config_POE>_oein_interrupt ( void );
```

Remark  

\( n \) is the number of an output enable interrupt source.

**Argument(s)**

None.

**Return value**

None.
Usage example

Placing the pins of the MTU0 in the high-impedance state in response to an output enable interrupt:

main.c
```c
#include "r_smc_entry.h"
void main(void)
{
    /* Start the POE module */
    R_Config_POE_Start();

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

Config_POE_user.c
```c
void r_Config_POE_oei1_interrupt(void)
{
    /* Start user code for r_Config_POE_oei1_interrupt. Do not edit comment generated here */
    /* Stop the POE module */
    R_Config_POE_Stop();

    /* Place MTU0 pins in high-impedance */
    R_Config_POE_Set_HiZ_MTU0();
    /* End user code. Do not edit comment generated here */
}
```
### 4.2.26 I/O Port

The Code Generator outputs the following API functions for the I/O ports.

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config.PORT&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the I/O ports.</td>
</tr>
<tr>
<td>R_&lt;Config.PORT&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the I/O ports.</td>
</tr>
</tbody>
</table>
**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]**

```c
void R_<Config_PORT>_Create ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes user-specific initialization processing for the I/O ports.

**Remark**
This API function is called from \texttt{R\_<Config\_PORT\>_Create} as a callback routine.

**[Syntax]**
\begin{verbatim}
void \texttt{R\_<Config\_PORT\>_Create\_UserInit} ( void );
\end{verbatim}

**[Argument(s)]**
None.

**[Return value]**
None.
Usage example

None.
4.2.27 Programmable Pulse Generator

The Code Generator outputs the following API functions for the programmable pulse generator.

Table 4.28 API Functions: [Programmable Pulse Generator]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_PPG0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the programmable pulse generator.</td>
</tr>
<tr>
<td>R_&lt;Config_PPG0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the programmable pulse generator.</td>
</tr>
</tbody>
</table>
**R_<Config_PPG0>_Create**

This API function executes initialization processing that is required before controlling the programmable pulse generator.

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

**[Syntax]**

```c
void R_<Config_PPG0>_Create ( void );
```

**[Argument(s)]**  
None.

**[Return value]**  
None.
This API function executes user-specific initialization processing for the programmable pulse generator.

**Remark**  This API function is called from R_<Config_PPG0>_Create as a callback routine.

**[Syntax]**

```c
void R_<Config_PPG0>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

None.
### 4.2.28 PWM Mode Timer

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

#### Table 4.29 API Functions: [PWM Mode Timer]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_MTU0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the PWM mode timer (MTU or TPU).</td>
</tr>
<tr>
<td>R_&lt;Config_MTU0&gt;_Start</td>
<td>Starts counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_MTU0&gt;_Start</td>
<td>Stops counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_MTU0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the PWM mode timer (MTU or TPU).</td>
</tr>
<tr>
<td>r_&lt;Config_MTU0&gt;_tgimn_interrupt</td>
<td>Executes processing in response to compare match interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_MTU0&gt;_tgimn_interrupt</td>
<td>(The name of this API function varies with the resource.)</td>
</tr>
<tr>
<td>r_&lt;Config_MTU0&gt;_tcivn_interrupt</td>
<td>Executes processing in response to overflow interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_MTU0&gt;_tcinv_interrupt</td>
<td>(The name of this API function varies with the resource.)</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling a timer (MTU or TPU) in PWM mode.

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

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
This API function starts counting by the counter.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_MTU0>_Stop

This API function stops counting by the counter.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
R_{Config\_MTU0}\_Create\_UserInit

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

Remark This API function is called from R_{Config\_MTU0}\_Create as a callback routine.

[Syntax]

\[
\text{void } R_{\text{Config}\_\text{MTU0}}\_\text{Create\_UserInit} ( \text{void} );
\]

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to compare match interrupts.

**[Syntax]**

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

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

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

**[Argument(s)]**

- None.

**[Return value]**

- None.
This API function executes processing in response to overflow interrupts.

**[Syntax]**

```c
void r_<Config_MTU0>_tcivn_interrupt ( void );

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

**Remark1.** $n$ is a channel number.

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Output of ten PWM pulses:

main.c

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

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

Config_MTU0_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_mtu0_cnt;
/* End user code. Do not edit comment generated here */
void R_Config_MTU0_Create_UserInit(void)
{
        /* Start user code for user init. Do not edit comment generated here */
        g_mtu0_cnt = 0U;
        /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU0_tgia0_interrupt(void)
{
        /* Start user code for r_Config_MTU0_tgia0_interrupt. Do not edit comment generated here */
        if ((++g_mtu0_cnt) > 9U)
        {
                /* Stop MTU channel 0 counter */
                R_Config_MTU0_Stop();
        }
        /* End user code. Do not edit comment generated here */
}
```
4.2.29 Realtime Clock

The Code Generator outputs the following API functions for the realtime clock.

Table 4.30 API Functions: [Realtime Clock]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_RTC&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the realtime clock.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Start</td>
<td>Starts counting by the counters.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Stop</td>
<td>Stops counting by the counters.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Restart</td>
<td>Stops the counters, initializes the counter values, and then restarts the counters in calendar count mode.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Restart_BinaryCounter</td>
<td>Stops the counters, clears the counter values to 0, and then restarts the counters in binary count mode.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_CalendarCounterValue</td>
<td>Specifies calendar values.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Get_CalendarCounterValue</td>
<td>Gets the calendar values from the counter registers.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Get_CalendarTimeCaptureValue</td>
<td>Gets the calendar values from the capture registers.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_BinaryCounterValue</td>
<td>Specifies a value for binary counting.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Get_BinaryCounterValue</td>
<td>Gets the binary counter value from the counter registers.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Get_BinaryTimeCaptureValue</td>
<td>Gets the binary counter value from the capture registers.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_RTCOUTOn</td>
<td>Specifies the frequency of output through the RTCOUT pin and starts the output.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_RTCOUTOff</td>
<td>Stops output through the RTCOUT pin.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_CalendarAlarm</td>
<td>Specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Calendar count mode)</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_BinaryAlarm</td>
<td>Specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Binary count mode)</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_ConstPeriodInterruptOn</td>
<td>Specifies the interval of periodic interrupts and enables the detection of the interrupts.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_ConstPeriodInterruptOff</td>
<td>Disables the detection of periodic interrupts.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_CarryInterruptOn</td>
<td>Enables the detection of carry interrupts.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Set_CarryInterruptOff</td>
<td>Disables the detection of carry interrupts.</td>
</tr>
<tr>
<td>R_&lt;Config_RTC&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the realtime clock.</td>
</tr>
<tr>
<td>r_&lt;Config_RTC&gt;_alm_interrupt</td>
<td>Executes processing in response to alarm interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RTC&gt;_prd_interrupt</td>
<td>Executes processing in response to periodic interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RTC&gt;_cup_interrupt</td>
<td>Executes processing in response to carry interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_RTC>_Create**

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

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

**[Syntax]**

```c
void R_<Config_RTC>_Create ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
R_<Config_RTC>_Start

This API function starts counting by the counters.

[Syntax]

```c
void R_<Config_RTC>_Start ( void );
```

[Argument(s)]
None.

[Return value]
None.
**R_<Config_RTC>_Stop**

This API function stops counting by the counters.

**[Syntax]**

```c
void R_<Config_RTC>_Stop ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_RTC>_Restart**

This API function stops the counters, initializes the counter values, and then restarts the counters in calendar count mode.

**[Syntax]**

```c
void R_<Config_RTC>_Restart ( rtc_calendarcounter_value_t counter_write_val );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>rtc_calendarcounter_value_t counter_write_val</td>
<td>Initial values (year, month, date, day-of-week, hour, minute, and second)</td>
</tr>
</tbody>
</table>

*Remark*  The following shows the structure that holds the initial value arguments, `rtc_calendarcounter_value_t`.

```c
typedef struct {
    uint8_t rseccnt;  /* Second */
    uint8_t rmincnt;  /* Minute */
    uint8_t rhrcnt;  /* Hour */
    uint8_t rdaycnt;  /* Date */
    uint8_t rwkcnt;  /* Day-of-week (0: Sunday; 6: Saturday) */
    uint8_t rmoncnt;  /* Month */
    uint16_t ryrcnt;  /* Year */
} rtc_calendarcounter_value_t;
```

*[Return value]*

None.
This API function stops the counters, clears the counter values to 0, and then restarts the counters in binary count mode.

**[Syntax]**

```c
void R_<Config_RTC>_Restart_BinaryCounter ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function specifies calendar values.

**[Syntax]**

```c
void R_<Config_RTC>_Set_CalendarCounterValue ( rtc_calendarcounter_value_t counter_write_val );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>rtc_calendarcounter_value_t</td>
<td>Calendar values (year, month, date, day-of-week, hour, minute, and second)</td>
</tr>
</tbody>
</table>

Remark The following shows the structure that holds the calendar value arguments, `rtc_calendarcounter_value_t`.

```c
typedef struct {
    uint8_t rseccnt; /* Second */
    uint8_t rmincnt; /* Minute */
    uint8_t rhrcnt; /* Hour */
    uint8_t rdaycnt; /* Date */
    uint8_t rwkcnt; /* Day-of-week (0: Sunday; 6: Saturday) */
    uint8_t rmoncnt; /* Month */
    uint16_t ryrcnt; /* Year */
} rtc_calendarcounter_value_t;
```

**[Return value]**

None.
This API function gets the calendar values from the counter registers.

**[Syntax]**

```c
void R_<Config_RTC>_Get_CalendarCounterValue ( rtc_calendarcounter_value_t * const counter_read_val );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>rtc_calendarcounter_value_t * const counter_read_val;</td>
<td>Pointer to the location where the acquired calendar values (year, month, date, day-of-week, hour, minute, and second) are to be stored</td>
</tr>
</tbody>
</table>

**Remark**

The following shows the structure that holds the calendar value arguments, rtc_calendarcounter_value_t.

```c
typedef struct {
    uint8_t rseccnt; /* Second */
    uint8_t rmincnt; /* Minute */
    uint8_t rhrcnt; /* Hour */
    uint8_t rdaycnt; /* Date */
    uint8_t rwkcnt; /* Day-of-week (0: Sunday; 6: Saturday) */
    uint8_t rmoncnt; /* Month */
    uint16_t ryrcnt; /* Year */
} rtc_calendarcounter_value_t;
```

**[Return value]**

None.
This API function gets the calendar values from the capture registers.

**Syntax**

```c
void R_<Config_RTC>_Get_CalendarTimeCaptureValue ( rtc_calendarcounter_value_t * const counter_read_val);
```

Remark  $n$ is a channel number.

**Argument(s)**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>rtc_calendarcounter_value_t * const counter_read_val;</td>
<td>Pointer to the location where the acquired calendar values (year, month, date, day-of-week, hour, minute, and second) are to be stored</td>
</tr>
</tbody>
</table>

Remark  The following shows the structure that holds the calendar value arguments, rtc_calendarcounter_value_t:

```c
typedef struct {
  uint8_t rseccnt; /* Second */
  uint8_t rmincnt; /* Minute */
  uint8_t rhrcnt; /* Hour */
  uint8_t rdaycnt; /* Date */
  uint8_t rwkcnt; /* Day-of-week (0: Sunday; 6: Saturday) */
  uint8_t rmoncnt; /* Month */
  uint16_t ryrcnt; /* Year */
} rtc_calendarcounter_value_t;
```

**Return value**

None.
R_<Config_RTC>_Set_BinaryCounterValue

This API function specifies a value for binary counting.

[Syntax]

```c
void R_<Config_RTC>_Set_BinaryCounterValue ( uint32_t counter_write_val );
```

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint32_t counter_write_val;</td>
<td>Binary counter value</td>
</tr>
</tbody>
</table>

[Return value]

None.
R_<Config_RTC>_Get_BinaryCounterValue

This API function gets the binary counter value from the counter registers.

**[Syntax]**

```c
void R_<Config_RTC>_Get_BinaryCounterValue ( uint32_t * const counter_read_val );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint32_t * const counter_read_val</td>
<td>Pointer to the location where the acquired binary counter value is to be stored</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
This API function gets the binary counter value from the capture registers.

**[Syntax]**

`void R_<Config_RTC>_Get_BinaryTimeCaptureValueN ( uint32_t * const counter_read_val );`

*Remark*  
$n$ is a channel number.

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint32_t * const counter_read_val;</td>
<td>Pointer to the location where the acquired binary counter value is to be stored</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
This API function specifies the frequency of output through the RTCOUT pin and starts the output.

**Syntax**

```c
void R_<Config_RTC>_Set_RTCOUTOn ( rtc_rtcout_period_t rtcout_freq );
```

**Argument(s)**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>rtc_rtcout_period_t rtcout_freq;</td>
<td>Frequency of output through RTCOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTCOUT_1HZ: 1 Hz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTCOUT_64HZ: 64 Hz</td>
</tr>
</tbody>
</table>

**Return value**

None.
R_<Config_RTC>_Set_RTCOUTOff

This API function stops output through the RTCOUT pin.

[Syntax]
void R_<Config_RTC>_Set_RTCOUTOff ( void );

[Argument(s)]
None.

[Return value]
None.
**R_<Config_RTC>_Set_CalendarAlarm**

This API function specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Calendar count mode)

**[Syntax]**

```c
void R_<Config_RTC>_Set_CalendarAlarm ( rtc_calendar_alarm_enable_t alarm_enable,
rtc_calendar_alarm_value_t alarm_val );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>rtc_calendar_alarm_enable_t alarm_enable;</td>
<td>Comparison flags (year, month, date, day-of-week, hour, minute, and second)</td>
</tr>
<tr>
<td>I</td>
<td>rtc_calendar_alarm_value_t alarm_val;</td>
<td>Calendar values (year, month, date, day-of-week, hour, minute, and second)</td>
</tr>
</tbody>
</table>

Remark 1. The following shows the structure that holds the comparison flag arguments, `rtc_calendar_alarm_enable_t`.

```c
typedef struct {
    uint8_t sec_enb; /* Second (0x0: Not compared; 0x80: Compared) */
    uint8_t min_enb; /* Minute (0x0: Not compared; 0x80: Compared) */
    uint8_t hr_enb; /* Hour (0x0: Not compared; 0x80: Compared) */
    uint8_t day_enb; /* Date (0x0: Not compared; 0x80: Compared) */
    uint8_t wk_enb; /* Day-of-week (0x0: Not compared; 0x80: Compared) */
    uint8_t mon_enb; /* Month (0x0: Not compared; 0x80: Compared) */
    uint8_t yr_enb; /* Year (0x0: Not compared; 0x80: Compared) */
} rtc_calendar_alarm_enable_t;
```

Remark 2. The following shows the structure that holds the calendar value arguments, `rtc_calendar_alarm_value_t`.

```c
typedef struct {
    uint8_t rsecar; /* Second */
    uint8_t rminar; /* Minute */
    uint8_t rhrar; /* Hour */
    uint8_t rdayar; /* Date */
    uint8_t rwkar; /* Day-of-week (0: Sunday; 6: Saturday) */
    uint8_t rmonar; /* Month */
    uint16_t ryrar; /* Year */
} rtc_calendar_alarm_value_t;
```

**[Return value]**

None.
R\_<Config_RTC>_Set_BinaryAlarm

This API function specifies the conditions for generating alarm interrupts and enables the detection of
the interrupts. (Binary count mode)

[Syntax]

void R\_<Config_RTC>_Set_BinaryAlarm ( uint32_t alarm_enable, uint32_t alarm_val );

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint32_t alarm_enable;</td>
<td>Comparison flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0: Not compared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1: Compared</td>
</tr>
<tr>
<td>I</td>
<td>uint32_t alarm_val;</td>
<td>Binary counter value</td>
</tr>
</tbody>
</table>

[Return value]

None.
R_<Config_RTC>_Set_ConstPeriodInterruptOn

This API function specifies the interval of periodic interrupts and enables the detection of the interrupts.

**[Syntax]**

```c
void R_<Config_RTC>_Set_ConstPeriodInterruptOn ( rtc_int_period_t period );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>rtc_int_period_t period;</td>
<td>Interval of periodic interrupts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_2_SEC: 2 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_SEC: 1 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_2_SEC: 1/2 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_4_SEC: 1/4 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_8_SEC: 1/8 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_16_SEC: 1/16 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_32_SEC: 1/32 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_64_SEC: 1/64 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_128_SEC: 1/128 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PES_1_256_SEC: 1/256 second</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
R_<Config_RTC>_Set_ConstPeriodInterruptOff

This API function disables the detection of periodic interrupts.

[Syntax]

```c
void R_<Config_RTC>_Set_ConstPeriodInterruptOff ( void );
```

[Argument(s)]

None.

[Return value]

None.
This API function enables the detection of carry interrupts.

**Syntax**

```c
void R_<Config_RTC>_Set_CarryInterruptOn ( void );
```

**Argument(s)**

None.

**Return value**

None.
R_{Config_RTC}_Set_CarryInterruptOff

This API function disables the detection of carry interrupts.

[Syntax]

void R_{Config_RTC}_Set_CarryInterruptOff ( void );

[Argument(s)]
None.

[Return value]
None.
**R_{<Config_RTC>_Create_UserInit}**

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

**Remark** This API function is called from R_{<Config_RTC>_Create} as a callback routine.

**[Syntax]**

```c
void R_{<Config_RTC>_Create_UserInit} ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to alarm interrupts.

Remark  This API function is called as the interrupt handler for alarm interrupts, which occur when the conditions specified in \texttt{R\_<Config_RTC>_Set_CalendarAlarm} are satisfied.

### Syntax

\[
\begin{align*}
\text{void} & \quad \text{r\_<Config_RTC>_alm_interrupt} ( \text{void} );
\end{align*}
\]

### Argument(s)
None.

### Return value
None.
This API function executes processing in response to periodic interrupts.

**Remark**  
This API function is called as the interrupt handler for periodic interrupts, which occur when the period specified by the parameter period in `R_<Config_RTC>_Set_ConstPeriodInterruptOn` has elapsed.

**Syntax**

```c
void r_<Config_RTC>_prd_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to carry interrupts.

**Remark**  This API function is called as the interrupt handler for carry interrupts, which occur in response to carrying to the second counter (RSECCNT) or binary counter 0 (BCNT0), or to carrying to the 64-Hz counter (R64CNT) while the counter (R64CNT) is being read.

**[Syntax]**

```c
void r_<Config_RTC>_cup_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Using alarm interrupts to implement virtual processing for leap second correction (turning back the clock from 23:59:59 to 23:59:58 on a scheduled day):

```
main.c

#include "r_smc_entry.h"

void main(void)
{
    /* Start RTC counter */
    R_Config_RTC_Start();

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

Config_RTC_user.c

/* Start user code for include. Do not edit comment generated here */
 volatile rtc_calendarcounter_value_t counter_val;
/* End user code. Do not edit comment generated here */

static void r_Config_RTC_alm_interrupt(void)
{
    /* Start user code for r_Config_RTC_alm_interrupt. Do not edit comment generated here */
    /* Disable ALM interrupt */
    IEN(RTC, ALM) = 0U;

    /* Get RTC calendar counter value */
    R_Config_RTC_Get_CalendarCounterValue((rtc_calendarcounter_value_t *)&counter_val);

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

    /* Set RTC calendar counter value */
    R_Config_RTC_Set_CalendarCounterValue(counter_val);
    /* End user code. Do not edit comment generated here */
}
4.2.30 Remote Control Signal Receiver

The Code Generator outputs the following API functions for the remote control signal receiver.

Table 4.31 API Functions: [Remote Control Signal Receiver]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_REMC0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the remote control signal receiver.</td>
</tr>
<tr>
<td>R_&lt;Config_REMC0&gt;_Start</td>
<td>Starts operation of the remote control signal receiver.</td>
</tr>
<tr>
<td>R_&lt;Config_REMC0&gt;_Stop</td>
<td>Stops operation of the remote control signal receiver.</td>
</tr>
<tr>
<td>R_&lt;Config_REMC0&gt;_Read</td>
<td>Specifies the location where the received data are to be stored and the number of bytes to be received.</td>
</tr>
<tr>
<td>R_&lt;Config_REMC0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the remote control signal receiver.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_remcin_interrupt</td>
<td>Executes processing in response to REMC interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_comparematch</td>
<td>Executes processing in response to compare match interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_receiveerror</td>
<td>Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_receiveend</td>
<td>Executes processing in response to data reception complete interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_bufferfull</td>
<td>Executes processing in response to receive buffer full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_header</td>
<td>Executes processing in response to header pattern match interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_data0</td>
<td>Executes processing in response to data &quot;0&quot; pattern match interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_data1</td>
<td>Executes processing in response to data &quot;1&quot; pattern match interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_REMC0&gt;_callback_specialdata</td>
<td>Executes processing in response to special data pattern match interrupts.</td>
</tr>
</tbody>
</table>
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**

```c
void R_<Config_REMC0>_Create ( void );
```

**Argument(s)**

None.

**Return value**

None.
R_<Config_REMC0>_Start

This API function starts operation of the remote control signal receiver.

[Syntax]
void R_<Config_REMC0>_Start ( void );

[Argument(s)]
None.

[Return value]
None.
R_<Config_REMC0>_Stop

This API function stops operation of the remote control signal receiver.

[Syntax]
void R_<Config_REMC0>_Stop ( void );

[Argument(s)]
None.

[Return value]
None.
**R_<Config_REMC0>_Read**

This API function specifies the location where the received data are to be stored and the number of bytes to be received.

**Remark**
This API function specifies the location where the received data read by the REMC interrupt routine at the end of data reception are to be stored.

**[Syntax]**

```c
MD_STATUS R_<Config_REMC0>_Read ( uint8_t * const rx_buf, uint8_t rx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const rx_buf</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint8_t rx_num</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ERROR1</td>
<td>The specification of argument rx_num is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_REMC0>_Create_UserInit

This API function executes user-specific initialization processing for the remote control signal receiver.

Remark  This API function is called from R_<Config_REMC0>_Create as a callback routine.

[Syntax]
void  R_<Config_REMC0>_Create_UserInit ( void );

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to REMC interrupts.

**Syntax**

```c
void r_<Config_REMC0>_remcin_interrupt ( void );
```

Remark  

*n* is a channel number.

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to compare match interrupts.

**Remark** This API function is called as a callback routine from `r_<Config_REMC0>_remcin_interrupt`, which is the interrupt handler for REMC interrupts.

**[Syntax]**

```c
void r_<Config_REMC0>_callback_comparematch ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to receive error interrupts.

**Remark**
This API function is called as a callback routine from `r_<Config_REMC0>_remcin_interrupt`, which is the interrupt handler for REMC interrupts.

### Syntax

```
void r_<Config_REMC0>_callback_receiveerror ( void );
```

#### Argument(s)
None.

#### Return value
None.
This API function executes processing in response to data reception complete interrupts.

Remark  This API function is called as a callback routine from 
\texttt{r\_\&lt;Config\_REMC0\>_remcin\_interrupt},
which is the interrupt handler for REMC interrupts.

\begin{verbatim}
void  r\_\&lt;Config\_REMC0\>_callback\_receiveend ( void );
\end{verbatim}

[Argument(s)]

None.

[Return value]

None.
This API function executes processing in response to receive buffer full interrupts.

Remark This API function is called as a callback routine from `r_<Config_REMC0>_remcin_interrupt`, which is the interrupt handler for REMC interrupts.

**Syntax**

```c
void r_<Config_REMC0>_callback_bufferfull ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to header pattern match interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```c
void r_<Config_REMC0>_callback_header ( void );
```

[Argument(s)]

None.

[Return value]

None.
**r_<Config_REMC0>_callback_data0**

This API function executes processing in response to data "0" pattern match interrupts.

**Remark**
This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

**[Syntax]**

```c
void r_<Config_REMC0>_callback_data0 ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_REMC0>_callback_data1**

This API function executes processing in response to data "1" pattern match interrupts.

**Remark**  This API function is called as a callback routine from `r_<Config_REMC0>_remcin_interrupt`, which is the interrupt handler for REMC interrupts.

**[Syntax]**

```c
void r_<Config_REMC0>_callback_data1 ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_REMC0>_callback_specialdata**

This API function executes processing in response to special data pattern match interrupts.

**Remark**
This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

**[Syntax]**

```c
void r_<Config_REMC0>_callback_specialdata ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
Usage example

Stopping operation of the remote control signal receiver at the end of data reception:

main.c

```c
#include "r_smc_entry.h"
extern volatile uint8_t g_remc0_rx_buf[8];
void main(void)
{
    /* Start the REMC0 operation */
    R_Config_REMC0_Start();

    /* Read data from receive data buffer */
    R_Config_REMC0_Read((uint8_t *)g_remc0_rx_buf, 8U);

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

Config_REMC0_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_remc0_rx_buf[8];
/* End user code. Do not edit comment generated here */
static void r_Config_REMC0_callback_receiveend(void)
{
    /* Start user code for r_Config_REMC0_callback_receiveend. Do not edit comment generated here */
    /* Stop the REMC0 operation */
    R_Config_REMC0_Stop();
    /* End user code. Do not edit comment generated here */
}
```
### 4.2.31 SCI/SCIF Asynchronous Mode

The Code Generator outputs the following API functions for the SCI/SCIF asynchronous mode

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_SCI0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the SCI/SCIF asynchronous mode.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Start</td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Stop</td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Serial_Send</td>
<td>Starts transmission. (Asynchronous mode)</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Serial_Receive</td>
<td>Starts reception. (Asynchronous mode)</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Serial_Multiprocessor_Send</td>
<td>[SCI/RSCI] Starts transmission. (Multi-processor communications function)</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Serial_Multiprocessor_Receive</td>
<td>[SCI/RSCI] Starts reception. (Multi-processor communications function)</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the SCI/SCIF asynchronous mode.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_transmitend_interrupt</td>
<td>[SCI/RSCI] Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
<td>[SCI/RSCI] Executes processing in response to transmit data empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
<td>[SCI/RSCI] Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_receiveerror_interrupt</td>
<td>[SCI/RSCI] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_teif_interrupt</td>
<td>[SCIF] Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_txif_interrupt</td>
<td>[SCIF] Executes processing in response to transmit FIFO data empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_rxif_interrupt</td>
<td>[SCIF] Executes processing in response to receive FIFO data full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_drif_interrupt</td>
<td>[SCIF] Executes processing in response to receive data ready interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_erif_interrupt</td>
<td>[SCIF] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_brif_interrupt</td>
<td>[SCIF] Executes processing in response to break interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
<td>[SCI/RSCI] Executes processing in response to transmit end interrupts or transmit data empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
<td>[SCI/RSCI] Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_receiveerror</td>
<td>[SCI/RSCI] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_error</td>
<td>[SCIF] Executes processing in response to receive error interrupts or break interrupts.</td>
</tr>
</tbody>
</table>
R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling the SCI/SCIF asynchronous mode.

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
R_<Config_SCI0>_Start

This API function starts serial communications.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
**R_<Config_SCI0>_Stop**

This API function stops serial communications.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts transmission. (Asynchronous mode)

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

Remark2.  Calling `R_<Config_SCI0>_Start` is required before this API function is called to execute transmission.

**[Syntax]**

```c
MD_STATUS     R_<Config_SCI0>_Serial_Send ( uint8_t * const tx_buf, uint16_t tx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td><code>uint8_t * const tx_buf;</code></td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td><code>uint16_t tx_num;</code></td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
This API function starts reception. (Asynchronous mode)

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

**Remark2.** Calling `R_<Config_SCI0>_Start` is required before this API function is called to execute reception.

**Syntax**

```c
MD_STATUS  R_<Config_SCI0>_Serial_Receive ( uint8_t * const rx_buf, uint16_t rx_num );
```

**Argument(s)**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t rx_num;</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

**Return value**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>rx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
**R_<Config_SCI0>_Serial_Multiprocessor_Send**

[SCI/RSCI] This API function starts transmission. (Multi-processor communications function)

**Remark1.** In multi-processor communications, a transmission cycle consists of an ID transmission cycle to specify a receiving station, and a data transmission cycle to transmit data to the specified receiving station.

**Remark2.** In the ID transmission cycle, this API function repeats the transmission of single bytes of data from the buffer specified by the argument `id_buf` the number of times specified by the argument `id_num`.

**Remark3.** In the data transmission cycle, this API function repeats the transmission of single bytes of data from the buffer specified by the argument `tx_buf` the number of times specified by the argument `tx_num`.

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

**[Syntax]**

```
MD_STATUS R_<Config_SCI0>_Serial_Multiprocessor_Send (uint8_t * const id_buf,
uint16_t id_num, uint8_t * const tx_buf, uint16_t tx_num);
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t * const id_buf;</td>
<td>Pointer to the buffer where the IDs to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t id_num;</td>
<td>Number of IDs to be transmitted</td>
</tr>
<tr>
<td>I</td>
<td>uint8_t * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_SCI0>_Serial_Multiprocessor_Receive

(SCI/RSCI) This API function starts reception. (Multi-processor communications function)

Remark1. In multi-processor communications, a receiving station with an ID matching a received ID receives the data that follows the ID.

Remark2. This API function repeats the reception of single bytes of data the number of times specified by the argument rx_num, storing the data in the buffer specified by the argument rx_buf.

Remark3. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

Syntax

```
MD_STATUS     R_<Config_SCI0>_Serial_Multiprocessor_Receive ( uint8_t * const rx_buf,
                      uint16_t  rx_num );
```

Argument(s)

```
<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const rx_buf</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t  rx_num</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>
```

Return value

```
<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument rx_num is invalid.</td>
</tr>
</tbody>
</table>
```
R_<Config_SCI0>_Create_UserInit

This API function executes user-specific initialization processing for the SCI/SCIF asynchronous mode.

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

[Syntax]

void R_<Config_SCI0>_Create_UserInit ( void );

[Argument(s)]
None.

[Return value]
None.
r_<Config_SCI0>_transmitend_interrupt

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

[Syntax]
void r_<Config_SCI0>_transmitend_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
r_<Config_SCI0>_transmit_interrupt

[SCI/RSCI] This API function executes processing in response to transmit data empty interrupts.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
r_<Config_SCI0>_receive_interrupt

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.

Receive interrupt request (SCI0.SCR.BIT.RIE=0 / RSCI10.SCR0.BIT.RIE=0) will be disabled inside this handler. If this caused any problem, please edit the source code at your discretion.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
r_<Config_SCI0>_receiveerror_interrupt

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

[Syntax]
void r_<Config_SCI0>_receiveerror_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
`r_<Config_SCI0>_teif_interrupt`

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

**Syntax**

```c
void r_<Config_SCI0>_teif_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
r_<Config_SCI0>_txif_interrupt

[SCIF] This API function executes processing in response to transmit FIFO data empty interrupts.

[Syntax]

    void     r_<Config_SCI0>_txif_interrupt ( void );

[Argument(s)]

    None.

[Return value]

    None.
r_<Config_SCI0>_rxif_interrupt

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.

[Syntax]
void r_<Config_SCI0>_rxif_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
r_<Config_SCI0>_drif_interrupt

[SCIF] This API function executes processing in response to receive data ready interrupts.

[Syntax]

```c
void r_<Config_SCI0>_drif_interrupt ( void );
```

[Argument(s)]
None.

[Return value]
None.
r_<Config_SCI0>_erif_interrupt

[SCIF] This API function executes processing in response to receive error interrupts.

Remark  This API function is called as the interrupt handler for interrupts due to framing errors or parity errors.

[Syntax]
void r_<Config_SCI0>_erif_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
r_<Config_SCI0>_brif_interrupt

[SCIF] This API function executes processing in response to break interrupts.
Remark      This API function is called as the interrupt handler for interrupts due to break signal
detection or overrun errors.

[Syntax]
void r_<Config_SCI0>_brif_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
**r_<Config_SCI0>_callback_transmitend**

[SCI/RSCI] This API function executes processing in response to transmit end interrupts or transmit data empty interrupts.

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

[SCIF] This API function executes processing in response to transmit end interrupts or transmit FIFO data empty interrupts.

Remark2. This API function is called from `r_<Config_SCI0>_teif_interrupt` or `r_<Config_SCI0>_txif_interrupt` as a callback routine.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_SCI0>_callback_receiveend**

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.

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

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.

Remark2. This API function is called from `r_<Config_SCI0>_rxif_interrupt` as a callback routine.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_SCI0>_callback_receiveerror**

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

Remark  This API function is called from `r_<Config_SCI0>_receiveerror_interrupt` as a callback routine.

**[Syntax]**

```c
void r_<Config_SCI0>_callback_receiveerror ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_SCI0>_callback_error**

[SCIF] This API function executes processing in response to receive error interrupts or break interrupts.

**Remark** This API function is called from `r_<Config_SCI0>_erif_interrupt` or `r_<Config_SCI0>_brif_interrupt` as a callback routine.

**[Syntax]**

```c
void r_<Config_SCI0>_callback_error ( scif_error_type_t error_type );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>scif_error_type_t error_type;</td>
<td>Interrupt sources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RECEIVE_ERROR: Framing error or parity error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OVERRUN_ERROR: Overrun error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BREAK_DETECT: Break signal detection</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
Usage example

Transmitting the upper-case character "A":

main.c

```c
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf;
void main(void)
{
    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Transmit SCI0 data */
    R_Config_SCI0_Serial_Send((uint8_t *)&g_sci0_tx_buf, 1U);

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

Config_SCI0_user.c

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

void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_buf = 'A';
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated here */
    /* Stop the SCI0 channel */
    R_Config_SCI0_Stop();
    /* End user code. Do not edit comment generated here */
}
```
### 4.32 SCI/SCIF Clock Synchronous Mode

The Code Generator outputs the following API functions for the SCI/SCIF clock synchronous mode.

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Create</code></td>
<td>Executes initialization processing that is required before controlling the SCI/SCIF clock synchronous mode.</td>
</tr>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Start</code></td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Stop</code></td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Serial_Send</code></td>
<td>Starts transmission. (Clock synchronous mode)</td>
</tr>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Serial_Receive</code></td>
<td>Starts reception. (Clock synchronous mode)</td>
</tr>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Serial_Send_Receive</code></td>
<td>Starts transmission and reception. (Clock synchronous mode)</td>
</tr>
<tr>
<td><code>R_&lt;Config_SCI0&gt;_Create_UserInit</code></td>
<td>Executes user-specific initialization processing for the SCI/SCIF clock synchronous mode.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_transmitend_interrupt</code></td>
<td>[SCI/RSCI] Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_transmit_interrupt</code></td>
<td>[SCI/RSCI] Executes processing in response to transmit data empty interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_receive_interrupt</code></td>
<td>[SCI/RSCI] Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_receiveerror_interrupt</code></td>
<td>[SCI/RSCI] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_teif_interrupt</code></td>
<td>[SCIF] Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_txif_interrupt</code></td>
<td>[SCIF] Executes processing in response to transmit FIFO data empty interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_rxif_interrupt</code></td>
<td>[SCIF] Executes processing in response to receive FIFO data full interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_erif_interrupt</code></td>
<td>[SCIF] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_brif_interrupt</code></td>
<td>[SCIF] Executes processing in response to break interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_callback_transmitend</code></td>
<td>[SCI/RSCI] Executes processing in response to transmit end interrupts or transmit data empty interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_callback_receiveend</code></td>
<td>[SCI/RSCI] Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_callback_receiveerror</code></td>
<td>[SCI/RSCI] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td><code>r_&lt;Config_SCI0&gt;_callback_error</code></td>
<td>[SCIF] Executes processing in response to receive error interrupts or break interrupts.</td>
</tr>
</tbody>
</table>
R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling the SCI/SCIF clock synchronous mode.

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function starts serial communications.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
This API function stops serial communications.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**R_<Config_SCI0>_Serial_Send**

This API function starts transmission. (Clock synchronous mode)

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

**Remark2.** Calling `R_<Config_SCI0>_Start` is required before this API function is called to execute transmission.

**[Syntax]**

```
MD_STATUS     R_<Config_SCI0>_Serial_Send ( uint8_t * const tx_buf, uint16_t tx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_SCI0>_Serial_Receive

This API function starts reception. (Clock synchronous mode)

Remark1. This API function repeats the reception of single bytes of data the number of times specified by the argument \( rx\_num \), storing the data in the buffer specified by the argument \( rx\_buf \).

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

[Syntax]

\[
\text{MD\_STATUS} \quad \text{R}_<\text{Config\_SCI0}>\_\text{Serial\_Receive} ( \text{uint8\_t} * \text{const} \ rx\_buf, \text{uint16\_t} \ rx\_num );
\]

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const ( rx_buf )</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t ( rx_num )</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument ( rx_num ) is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_SCI0>_Serial_Send_Receive

This API function starts transmission and reception. (Clock synchronous mode)

Remark1. To transmit data, this API function repeats the transmission of single bytes of data from the buffer specified by the argument `tx_buf` the number of times specified by the argument `tx_num`.

Remark2. To receive data, this API function repeats the reception of single bytes of data the number of times specified by the argument `rx_num`, storing the data in the buffer specified by the argument `rx_buf`.

Remark3. Calling `R_<Config_SCI0>_Start` is required before this API function is called to execute reception.

[Syntax]

```c
MD_STATUS R_<Config_SCI0>_Serial_Send_Receive ( uint8_t * const tx_buf, uint16_t tx_num, uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
<tr>
<td>O</td>
<td>uint8_t * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t rx_num;</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
R_\textlt{Config\_SCI0\gt}_\textlt{Create\_UserInit}

This API function executes user-specific initialization processing for the SCI/SCIF clock synchronous mode.

Remark This API function is called from R_\textlt{Config\_SCI0\gt}_\textlt{Create} as a callback routine.

[Syntax]

\begin{verbatim}
void R_\textlt{Config\_SCI0\gt}_\textlt{Create\_UserInit} ( \textlt{void\gt};
\end{verbatim}

[Argument(s)]

None.

[Return value]

None.
r_<Config_SCI0>_transmitend_interrupt

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
**r_<Config_SCI0>_transmit_interrupt**

[SCI/RSCI] This API function executes processing in response to transmit data empty interrupts.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**r_<Config_SCI0>_receive_interrupt**

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.

Receive interrupt request (SCI0.SCR.BIT.RIE=0 / RSCI10.SCR0.BIT.RIE=0) will be disabled inside this handler. If this caused any problem, please edit the source code at your discretion.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
[SCI/RSCI] This API function executes processing in response to receive error interrupts.

**Syntax**

```c
void r_<Config_SCI0>_receiveerror_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
r_<Config_SCI0>_teif_interrupt

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

[Syntax]
void r_<Config_SCI0>_teif_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
r_<Config_SCI0>_txif_interrupt

[SCIF] This API function executes processing in response to transmit FIFO data empty interrupts.

[Syntax]

```c
void r_<Config_SCI0>_txif_interrupt ( void );
```

[Argument(s)]
None.

[Return value]
None.
`r_<Config_SCI0>_rxif_interrupt`

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.

**[Syntax]**

```c
void r_<Config_SCI0>_rxif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.
**r_<Config_SCI0>_erif_interrupt**

[SCIF] This API function executes processing in response to receive error interrupts.

**Remark**  
This API function is called as the interrupt handler for interrupts due to framing errors or parity errors.

**[Syntax]**

```c
void r_<Config_SCI0>_erif_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_SCI0>_brif_interrupt**

[SCIF] This API function executes processing in response to break interrupts.

**Remark**  
This API function is called as the interrupt handler for interrupts due to break signal detection or overrun errors.

**[Syntax]**

```c
void r_<Config_SCI0>_brif_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
r_<Config_SCI0>_callback_transmitend

[SCI/RSCI] This API function executes processing in response to transmit end interrupts or transmit data empty interrupts.

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

[SCIF] This API function executes processing in response to transmit end interrupts or transmit FIFO data empty interrupts.

Remark2. This API function is called from r_<Config_SCI0>_teif_interrupt or r_<Config_SCI0>_txif_interrupt as a callback routine.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
r_<Config_SCI0>_callback_receiveend

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.
Remark1. This API function is called from r_<Config_SCI0>_receive_interrupt as a callback routine.

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.
Remark2. This API function is called from r_<Config_SCI0>_rxif_interrupt as a callback routine.

[Syntax]
void r_<Config_SCI0>_callback_receiveend ( void );

[Argument(s)]
None.

[Return value]
None.
**r_<Config_SCI0>_callback_receiveerror**

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

**Remark**  This API function is called from **r_<Config_SCI0>_receiveerror_interrupt** as a callback routine.

**[Syntax]**

```c
void r_<Config_SCI0>_callback_receiveerror ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
[SCIF] This API function executes processing in response to receive error interrupts or break interrupts.

Remark This API function is called from `r_<Config_SCI0>_erif_interrupt` or `r_<Config_SCI0>_brif_interrupt` as a callback routine.

**Syntax**

```c
void r_<Config_SCI0>_callback_error ( scif_error_type_t error_type );
```

**Argument(s)**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>scif_error_type_t error_type;</td>
<td>Interrupt sources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RECEIVE_ERROR: Framing error or parity error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OVERRUN_ERROR: Overrun error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BREAK_DETECT: Break signal detection</td>
</tr>
</tbody>
</table>

**Return value**

None.
Usage example

Transmitting received data:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf;
extern volatile uint8_t g_sci0_rx_buf;
void main(void)
{
    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Transmit and receive SCI0 data */
    R_Config_SCI0_Serial_Send_Receive((uint8_t *)&g_sci0_tx_buf, 1U, (uint8_t *)&g_sci0_rx_buf, 1U);

    while (1U)
    {
        nop();
    }
}
```
Config_SCI0_user.c
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_sci0_tx_buf;
volatile uint8_t g_sci0_rx_buf;
/* End user code. Do not edit comment generated here */

void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_buf = 0U;
    g_sci0_rx_buf = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated here */
    /* Transmit and receive SCI0 data */
    R_Config_SCI0_Serial_Send_Receive((uint8_t *)&g_sci0_tx_buf, 1U, (uint8_t *)&g_sci0_rx_buf, 1U);
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_receiveend(void)
{
    /* Start user code for r_Config_SCI0_callback_receiveend. Do not edit comment generated here */
    g_sci0_tx_buf = g_sci0_rx_buf;
    g_sci0_rx_buf = 0U;
    /* End user code. Do not edit comment generated here */
}
4.2.33 Single Scan Mode S12AD

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

Table 4.34 API Functions: [Single Scan Mode S12AD]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the single scan mode S12AD.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Start</td>
<td>Starts A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Stop</td>
<td>Stops A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Get_ValueResult</td>
<td>Gets the result of conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareValue</td>
<td>Sets the compare level.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareAValue</td>
<td>Sets the compare level for window A.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Set_CompareBValue</td>
<td>Sets the compare level for window B.</td>
</tr>
<tr>
<td>R_&lt;Config_S12AD0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the single scan mode S12AD.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_interrupt</td>
<td>Executes processing in response to scan end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_compare_interrupt</td>
<td>Executes processing in response to compare interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptA</td>
<td>Executes processing in response to compare interrupts for window A.</td>
</tr>
<tr>
<td>r_&lt;Config_S12AD0&gt;_compare_interruptB</td>
<td>Executes processing in response to compare interrupts for window B.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling the single scan mode S12AD.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function starts A/D conversion.

**Syntax**

```c
void R_Config_S12AD0_Start ( void );
```

**Argument(s)**

None.

**Return value**

None.
R_<Config_S12AD0>_Stop

This API function stops A/D conversion.

[Syntax]
void R_<Config_S12AD0>_Stop ( void );

[Argument(s)]
None.

[Return value]
None.
R_<Config_S12AD0>_Get_ValueResult

This API function gets the result of conversion.

[Syntax]

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

[Argument(s)]

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

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>ad_channel_t channel;</td>
<td>Channel number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL0: Input channel AN000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL1: Input channel AN001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL2: Input channel AN002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL3: Input channel AN003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL4: Input channel AN004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL5: Input channel AN005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL6: Input channel AN006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL7: Input channel AN007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL8: Input channel AN008</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL9: Input channel AN009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL10: Input channel AN010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL11: Input channel AN011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL12: Input channel AN012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL13: Input channel AN013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL14: Input channel AN014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL15: Input channel AN015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL16: Input channel AN016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL17: Input channel AN017</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL18: Input channel AN018</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL19: Input channel AN019</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCHANNEL20: Input channel AN020</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADTEMPSENSOR: Extended analog input (temperature sensor output)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADINTERREFVOLT: Extended analog input (internal reference voltage)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADSELFDIAGNOSIS: Result of self-diagnosis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDATADUPLICATION: Double-trigger mode result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDATADUPLICATIONA: Double-trigger mode A result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDATADUPLICATIONB: Double-trigger mode B result</td>
</tr>
</tbody>
</table>

| O   | uint16_t * const buffer; | Pointer to the location where the acquired results of conversion are to be stored |

[Return value]
None.
R_<Config_S12AD0>_Set_CompareValue

This API function sets the compare level.

[Syntax]

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

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

[Return value]

None.
**R\_<Config\_S12AD0>_Set\_CompareAValue**

This API function sets the compare level for window A.

**[Syntax]**

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

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
R_<Config_S12AD0>_Set_CompareBValue

This API function sets the compare level for window B.

[Syntax]

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

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint16_t reg_value0;</td>
<td>Value to be set in compare register 0</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t reg_value1;</td>
<td>Value to be set in compare register 1</td>
</tr>
</tbody>
</table>

[Return value]

None.
This API function executes user-specific initialization processing for the S12AD in single scan mode.

Remark This API function is called from \texttt{R\textunderscore <Config\textunderscore S12AD0\textunderscore Create} as a callback routine.

[Syntax]

\begin{verbatim}
void \texttt{R\textunderscore <Config\textunderscore S12AD0\textunderscore Create\textunderscore UserInit} ( void );
\end{verbatim}

[Argument(s)]

None.

[Return value]

None.
### r_<Config_S12AD0>_interrupt

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare interrupts.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare interrupts for window A.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to compare interrupts for window B.

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Getting the result of A/D conversion:

main.c

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

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

Config_S12AD0_user.c

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

static void r_Config_S12AD0_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, (uint16_t *)&g_s12ad0_ch000_value);
    /* End user code. Do not edit comment generated here */
}
4.2.34 Smart Card Interface Mode

The Code Generator outputs the following API functions for the smart card interface mode.

Table 4.35 API Functions: [SCI/RSCI in Smart Card Interface Mode]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_SCI0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the smart card interface mode.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Start</td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Stop</td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_SmartCard_Send</td>
<td>Starts transmission. (Smart card interface mode)</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_SmartCard_Receive</td>
<td>Starts reception. (Smart card interface mode)</td>
</tr>
<tr>
<td>R_&lt;Config_SCI0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the smart card interface mode.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_transmit_interrupt</td>
<td>Executes processing in response to transmit data empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_receive_interrupt</td>
<td>Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_receiveerror_interrupt</td>
<td>Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_transmitend</td>
<td>Executes processing in response to transmit data empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_receiveend</td>
<td>Executes processing in response to receive data full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_SCI0&gt;_callback_receiveerror</td>
<td>Executes processing in response to receive error interrupts.</td>
</tr>
</tbody>
</table>
R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling the smart card interface.

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
This API function starts serial communications.

[Syntax]
void R_<Config_SCI0>_Start ( void );

[Argument(s)]
None.

[Return value]
None.
R_<Config_SCI0>_Stop

This API function stops serial communications.

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function starts transmission. (Smart card interface mode)

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

Remark2. Calling `R_<Config_SCI0>_Start` is required before this API function is called to execute transmission.

### Syntax

```c
MD_STATUS R_<Config_SCI0>_SmartCard_Send ( uint8_t * const tx_buf, uint16_t tx_num );
```

### Argument(s)

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>uint8_t * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

### Return value

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
**R_<Config_SCI0>_SmartCard_Receive**

This API function starts reception. (Smart card interface mode)

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

Remark2. Calling `R_<Config_SCI0>_Start` is required before this API function is called to execute reception.

**[Syntax]**

```c
MD_STATUS R_<Config_SCI0>_SmartCard_Receive ( uint8_t * const rx_buf, uint16_t rx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint8_t * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t rx_num;</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>rx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
This API function executes user-specific initialization processing for the smart card interface.

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to transmit data empty interrupts.

**Syntax**

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

**Argument(s)**
None.

**Return value**
None.
This API function executes processing in response to receive data full interrupts.

**Syntax**

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

**Argument(s)**

None.

**Return value**

None.
**r_<Config_SCI0>_receiveerror_interrupt**

This API function executes processing in response to receive error interrupts.

**[Syntax]**

```c
void r_<Config_SCI0>_receiveerror_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to transmit data empty interrupts.

Remark  
This API function is called from `r_<Config_SCI0>_transmit_interrupt` as a callback routine.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.
r_<Config_SCI0>_callback_receiveend

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

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

[Syntax]

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

[Argument(s)]
None.

[Return value]
None.
This API function executes processing in response to receive error interrupts.

Remark this API function is called from `r_<Config_SCI0>_receiveerror_interrupt` as a callback routine.

**Syntax**

```c
void r_<Config_SCI0>_callback_receiveerror ( void );
```

**Argument(s)**

None.

**Return value**

None.
Usage example

Transmitting one byte, and then switching to the reception of one byte:

```c
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf;
void main(void)
{
    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Transmit SCI0 data */
    R_Config_SCI0_SmartCard_Send((uint8_t *)&g_sci0_tx_buf, 1U);

    while (1U)
    {
        nop();
    }
}
```
void R_Config_SCI0_Create_UserInit(void) {
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_buf = 'A';
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_transmitend(void) {
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated here */
    /* Stop the SCI0 channel */
    R_Config_SCI0_Stop();

    /* Initializes SCI0 */
    R_Config_SCI0_Create();

    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Receive SCI0 data */
    R_Config_SCI0_SmartCard_Receive((uint8_t *)&g_sci0_rx_buf, 1U);
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_receiveend(void) {
    /* Start user code for r_Config_SCI0_callback_receiveend. Do not edit comment generated here */
    /* Stop the SCI0 channel */
    R_Config_SCI0_Stop();
    /* End user code. Do not edit comment generated here */
}
### 4.2.35 SPI Clock Synchronous Mode

The Code Generator outputs the following API functions for the SPI clock synchronous mode (RSPI/SCI/RSCI).

**Table 4.36 API Functions: [RSPI/SCI/RSCI in SPI Clock Synchronous Mode]**

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_RSPI0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the SPI clock synchronous mode (RSPI/SCI/RSCI).</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_Start</td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_Stop</td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_Send</td>
<td>[RSPI] Starts transmission.</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_Send_Receive</td>
<td>[RSPI] Starts transmission and reception.</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_SPI_Master_Send</td>
<td>[SCI/RSCI] Starts master transmission. (Simple SPI mode)</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_SPI_Master_Send_Receive</td>
<td>[SCI/RSCI] Starts master transmission and reception. (Simple SPI mode)</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_SPI_Slave_Send</td>
<td>[SCI/RSCI] Starts slave transmission. (Simple SPI mode)</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_SPI_Slave_Send_Receive</td>
<td>[SCI/RSCI] Starts slave transmission and reception. (Simple SPI mode)</td>
</tr>
<tr>
<td>R_&lt;Config_RSPI0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the SPI clock synchronous mode (RSPI/SCI/RSCI).</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_receive_interrupt</td>
<td>Executes processing in response to receive buffer full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_transmit_interrupt</td>
<td>Executes processing in response to transmit buffer empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_error_interrupt</td>
<td>[RSPI] Executes processing in response to error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_idle_interrupt</td>
<td>[RSPI] Executes processing in response to RSPI idle interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_transmitend_interrupt</td>
<td>[SCI/RSCI] Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_receiveerror_interrupt</td>
<td>[SCI/RSCI] Executes processing in response to receive error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_callback_receiveend</td>
<td>Executes processing in response to receive buffer full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_callback_transmitend</td>
<td>[RSPI] Executes processing in response to RSPI idle interrupts. [SCI/RSCI] Executes processing in response to transmit end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_callback_error</td>
<td>[RSPI] Executes processing in response to error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSPI0&gt;_callback_receiveerror</td>
<td>[SCI/RSCI] Executes processing in response to receive error interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_RSPI0>_Create**

This API function executes initialization processing that is required before controlling the clock synchronous SPI mode (RSPI/SCI/RSCI).

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

**[Syntax]**

```plaintext
void R_<Config_RSPI0>_Create ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_RSPI0>_Start**

This API function starts serial communications.

**[Syntax]**

```c
void R_<Config_RSPI0>_Start ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function stops serial communications.

**[Syntax]**

```
void R_<Config_RSPI0>_Stop ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_RSPI0>_Send**

*RSPI* This API function starts transmission.

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

**Remark2.** Calling `R_<Config_RSPI0>_Start` is required before this API function is called to execute transmission.

**[Syntax]**
```
MD_STATUS R_<Config_RSPI0>_Send ( type * const tx_buf, uint16_t tx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td><code>type * const tx_buf;</code></td>
<td>Pointer to the buffer where the data to be transmitted are stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td><code>uint16_t tx_num;</code></td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
**R_<Config_RSPI0>_Send_Receive**

[RSPI] This API function starts transmission and reception.

**Remark1.** To transmit data, this API function repeats the transmission of single bytes of data from the buffer specified by the argument `tx_buf` the number of times specified by the argument `tx_num`.

**Remark2.** To receive data, this API function repeats the reception of single bytes of data the number of times specified by the argument `tx_num`, storing the data in the buffer specified by the argument `rx_buf`.

**Remark3.** Calling `R_<Config_RSPI0>_Start` is required before this API function is called to execute transmission and reception.

**[Syntax]**

```c
MD_STATUS    R_<Config_RSPI0>_Send_Receive ( type * const tx_buf, uint16_t tx_num, 
    type * const rx_buf);
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted and received</td>
</tr>
<tr>
<td>O</td>
<td>type * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_RSPI0>_SPI_Master_Send

[SCI/RSCI] This API function starts master transmission. (Simple SPI mode)

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

Remark2. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute master transmission.

[Syntax]

```
MD_STATUS    R_<Config_RSPI0>_SPI_Master_Send (type * const tx_buf, uint16_t tx_num);
```  

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument tx_num is invalid.</td>
</tr>
</tbody>
</table>
**R_<Config_RSPi0>_SPI_Master_Send_Receive**

[SCI/RSCI] This API function starts master transmission and reception. (Simple SPI mode)

**Remark1.** To transmit data as a master, this API function repeats the master transmission of single bytes of data from the buffer specified by the argument `tx_buf` the number of times specified by the argument `tx_num`.

**Remark2.** To receive data as a master, this API function repeats the master reception of single bytes of data the number of times specified by the argument `rx_num`, storing the data in the buffer specified by the argument `rx_buf`.

**Remark3.** Calling `R_<Config_RSPi0>_Start` is required before this API function is called to execute transmission and reception.

**[Syntax]**

```c
MD_STATUS R_<Config_RSPi0>_SPI_Master_Send_Receive ( type * const tx_buf, uint16_t tx_num, type * const rx_buf, uint16_t rx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
<tr>
<td>O</td>
<td>type * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t rx_num;</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
**R_<Config_RSPI0>_SPI_Slave_Send**

[SCI/RSCI] This API function starts slave transmission. (Simple SPI mode)

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

Remark2. Calling `R_<Config_RSPI0>_Start` is required before this API function is called to execute slave transmission.

**[Syntax]**

```c
MD_STATUS    R_<Config_RSPI0>_SPI_Slave_Send (type * const tx_buf, uint16_t tx_num);
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <code>tx_num</code> is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_RSPi0>_SPI_Slave_Send_Receive

[SCI/RSCI] This API function starts slave transmission and reception. (Simple SPI mode)

Remark1. To transmit data as a slave, this API function repeats the slave transmission of single bytes of data from the buffer specified by the argument tx_buf the number of times specified by the argument tx_num.

Remark2. To receive data as a slave, this API function repeats the slave reception of single bytes of data the number of times specified by the argument rx_num, storing the data in the buffer specified by the argument rx_buf.

Remark3. Calling R_<Config_RSPi0>_Start is required before this API function is called to execute transmission and reception.

[Syntax]

MD_STATUS R_<Config_RSPi0>_SPI_Slave_Send_Receive ( type * const tx_buf, uint16_t tx_num, type * const rx_buf, uint16_t rx_num );

[Argument(s)]

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
<tr>
<td>O</td>
<td>type * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t rx_num;</td>
<td>Number of bytes to be received</td>
</tr>
</tbody>
</table>

[Return value]

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument tx_num is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_RSPI0>_Create_UserInit

This API function executes user-specific initialization processing for the clock synchronous SPI mode (RSPI/SCI/RSCI).

Remark This API function is called from R_<Config_RSPI0>_Create as a callback routine.

[Syntax]

```c
void R_<Config_RSPI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.
**r_<Config_RSPI0>_receive_interrupt**

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

Receive interrupt request (SCI0.SCR.BIT.RIE=0 / RSCI10.SCR0.BIT.RIE=0) or Receive Buffer full interrupt request (RSPI0.SPCR.BIT.SPRIE = 0) will be disabled inside this handler. If this caused any problem, please edit the source code at your discretion.

**[Syntax]**

```c
void _r_<Config_RSPI0>_receive_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to transmit buffer empty interrupts.

**Syntax**

```c
void r_Config_RSPI0_transmit_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
r_<Config_RSPi0>_error_interrupt

[RSPI] This API function executes processing in response to error interrupts.

[Syntax]

```c
void r_<Config_RSPi0>_error_interrupt ( void );
```

[Argument(s)]
None.

[Return value]
None.
r_<Config_RSPI0>_idle_interrupt

[RSPI] This API function executes processing in response to RSPI idle interrupts.

[Syntax]

```c
void r_<Config_RSPI0>_idle_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.
Function: \texttt{r_\langle Config\_RSPI0\rangle\_transmitend\_interrupt}

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

[Syntax]

\begin{verbatim}
void r_\langle Config\_RSPI0\rangle\_transmitend\_interrupt ( void );
\end{verbatim}

[Argument(s)]
None.

[Return value]
None.
r_<Config_RSPi0>_receiveerror_interrupt

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

[Syntax]
void r_<Config_RSPi0>_receiveerror_interrupt ( void );

[Argument(s)]
None.

[Return value]
None.
**r_<Config_RSPI0>_callback_receiveend**

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

**Remark**  This API function is called from `r_<Config_RSPI0>_receive_interrupt` as a callback routine.

**[Syntax]**

```
void r_<Config_RSPI0>_callback_receiveend ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to transmit buffer empty interrupts.
Remark1. This API function is called from `r_<Config_RSPI0>_transmit_interrupt` as a callback routine.

This API function executes processing in response to RSPI idle interrupts.
Remark2. This API function is called from `r_<Config_RSPI0>_idle_interrupt` as a callback routine.

This API function executes processing in response to transmit end interrupts.
Remark3. This API function is called from `r_<Config_RSPI0>_transmitend_interrupt` as a callback routine.

**Syntax**
```c
void r_<Config_RSPI0>_callback_transmitend ( void );
```

**Argument(s)**
None.

**Return value**
None.
**r_<Config_RSPI0>_callback_error**

**[RSPI]** This API function executes processing in response to error interrupts.

**Remark1.** This API function is called from `r_<Config_RSPI0>_error_interrupt` as a callback routine.

**[Syntax]**

```c
void r_<Config_RSPI0>_callback_error ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_RSPI0>_callback_receiveerror**

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

**Remark1.** This API function is called from `r_<Config_RSPI0>_receiveerror_interrupt` as a callback routine.

**[Syntax]**

```c
void r_<Config_RSPI0>_callback_receiveerror ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Stopping communications after completing transmission and reception of the same number of bytes:

main.c

```c
#include "r_smc_entry.h"
extern volatile uint32_t g_rspi0_tx_buf[4];
extern volatile uint32_t g_rspi0_rx_buf[4];
void main(void)
{
    /* Start the RSPI0 module operation */
    R_Config_RSPI0_Start();

    /* Send and receive RSPI0 data */
    R_Config_RSPI0_Send_Receive((uint32_t *)g_rspi0_tx_buf, 4U, (uint32_t *)g_rspi0_rx_buf);

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

Config_RSP10_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_rspi0_tx_buf[4];
volatile uint32_t g_rspi0_rx_buf[4];
/* End user code. Do not edit comment generated here */

void R_Config_RSP10_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_rspi0_tx_buf[0] = 0x000000FF;
    g_rspi0_tx_buf[1] = 0x0000FF00;
    g_rspi0_tx_buf[2] = 0x00FF0000;
    g_rspi0_tx_buf[3] = 0xFF000000;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_RSP10_callback_receiveend(void)
{
    /* Start user code for r_Config_RSP10_callback_receiveend. Do not edit comment generated here */
    /* Stop the RSPI0 module operation */
    R_Config_RSP10_Stop();
    /* End user code. Do not edit comment generated here */
```
4.2.36 SPI Operation Mode

The Code Generator outputs the following API functions for the SPI operation mode.

Table 4.37 API Functions: [SPI Operation Mode]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_RSP10&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the SPI operation mode.</td>
</tr>
<tr>
<td>R_&lt;Config_RSP10&gt;_Start</td>
<td>Starts serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_RSP10&gt;_Stop</td>
<td>Stops serial communications.</td>
</tr>
<tr>
<td>R_&lt;Config_RSP10&gt;_Send</td>
<td>Starts transmission.</td>
</tr>
<tr>
<td>R_&lt;Config_RSP10&gt;_Send_Receive</td>
<td>Starts transmission and reception.</td>
</tr>
<tr>
<td>R_&lt;Config_RSP10&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the SPI operation mode.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_receive_interrupt</td>
<td>Executes processing in response to receive buffer full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_transmit_interrupt</td>
<td>Executes processing in response to transmit buffer empty interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_error_interrupt</td>
<td>Executes processing in response to error interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_idle_interrupt</td>
<td>Executes processing in response to RSPI idle interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_callback_receiveend</td>
<td>Executes processing in response to receive buffer full interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_callback_transmitend</td>
<td>Executes processing in response to transmit buffer empty interrupts or RSPI idle interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_RSP10&gt;_callback_error</td>
<td>Executes processing in response to error interrupts.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling the SPI operation mode.

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

**[Syntax]**
```c
void R_<Config_RSPI0>_Create ( void );
```

**[Argument(s)]**
None.

**[Return value]**
None.
R_<Config_RSPI0>_Start

This API function starts serial communications.

[Syntax]

```c
void R_<Config_RSPI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.
R_<Config_RSPi0>_Stop

This API function stops serial communications.

[Syntax]
void R_<Config_RSPi0>_Stop ( void );

[Argument(s)]
None.

[Return value]
None.
**R_<Config_RSPI0>_Send**

This API function starts transmission.

**Remark 1.** This API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

**Remark 2.** Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission.

**[Syntax]**

```c
MD_STATUS R_<Config_RSPI0>_Send ( type * const tx_buf, uint16_t tx_num );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument <em>tx_num</em> is invalid.</td>
</tr>
</tbody>
</table>
R_<Config_RSPI0>_Send_Receive

This API function starts transmission and reception.

Remark1. To transmit data, this API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. To receive data, this API function repeats the reception of single bytes of data the number of times specified by the argument *tx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission and reception.

**[Syntax]**

```c
MD_STATUS R_<Config_RSPI0>_Send_Receive ( type * const tx_buf, uint16_t tx_num, type * const rx_buf );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>type * const tx_buf;</td>
<td>Pointer to the buffer where the data to be transmitted are stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
<tr>
<td>I</td>
<td>uint16_t tx_num;</td>
<td>Number of bytes to be transmitted and received</td>
</tr>
<tr>
<td>O</td>
<td>type * const rx_buf;</td>
<td>Pointer to the buffer where the received data are to be stored, the &quot;type&quot; can be &quot;uint32_t&quot;, &quot;uint16_t&quot; and &quot;uint8_t&quot; based on the buffer access width set on the GUI</td>
</tr>
</tbody>
</table>

**[Return value]**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_OK</td>
<td>Normal end</td>
</tr>
<tr>
<td>MD_ARGERROR</td>
<td>The specification of argument tx_num is invalid.</td>
</tr>
</tbody>
</table>
This API function executes user-specific initialization processing for the SPI operation mode.

**Remark** This API function is called from `R_<Config_RSPI0>_Create` as a callback routine.

**[Syntax]**

```c
void R_<Config_RSPI0>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to receive buffer full interrupts.

**Syntax**

```c
void r_<Config_RSPI0>_receive_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to transmit buffer empty interrupts.

**Syntax**

```c
void r_<Config_RSPI0>_transmit_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
This API function executes processing in response to error interrupts.

**[Syntax]**

```c
void r_<Config_RSP10>_error_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to RSPI idle interrupts.

**[Syntax]**

```c
void    r_<Config_RSPI0>_idle_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to receive buffer full interrupts.

Remark This API function is called from r_<Config_RSPI0>_receive_interrupt as a callback routine.

[Syntax]

```c
void r_<Config_RSPI0>_callback_receiveend ( void );
```

[Argument(s)]
None.

[Return value]
None.
**r_<Config_RSPI0>_callback_transmitend**

This API function executes processing in response to transmit buffer empty interrupts or RSPI idle interrupts.

**Remark**  This API function is called from **r_<Config_RSPI0>_transmit_interrupt** or **r_<Config_RSPI0>_idle_interrupt** as a callback routine.

**[Syntax]**

```c
void r_<Config_RSPI0>_callback_transmitend ( void );
```

**[Argument(s)]**

- None.

**[Return value]**

- None.
**r_<Config_RSPI0>_callback_error**

This API function executes processing in response to error interrupts.

**Remark**  This API function is called from `r_<Config_RSPI0>_error_interrupt` as a callback routine.

**[Syntax]**

```c
void r_<Config_RSPI0>_callback_error ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Stopping communications after completing transmission and reception of the same number of bytes:

main.c

```c
#include "r_smc_entry.h"
extern volatile uint32_t g_rspi0_tx_buf[4];
extern volatile uint32_t g_rspi0_rx_buf[4];
void main(void)
{
    /* Start the RSPI0 module operation */
    R_Config_RSPI0_Start();

    /* Send and receive RSPI0 data */
    R_Config_RSPI0_Send_R receive((uint32_t *)g_rspi0_tx_buf, 4U, (uint32_t *)g_rspi0_rx_buf);
    while (1U)
    {
        nop();
    }
}
```

Config_RSPI0_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_rspi0_tx_buf[4];
volatile uint32_t g_rspi0_rx_buf[4];
/* End user code. Do not edit comment generated here */

void R_Config_RSPI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_rspi0_tx_buf[0] = 0x000000FF;
    g_rspi0_tx_buf[1] = 0x0000FF00;
    g_rspi0_tx_buf[2] = 0x00FF0000;
    g_rspi0_tx_buf[3] = 0xFF000000;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_RSPI0_callback_receiveend(void)
{
    /* Start user code for r_Config_RSPI0_callback_receiveend. Do not edit comment generated here */
    /* Stop the RSPI0 module operation */
    R_Config_RSPI0_Stop();
    /* End user code. Do not edit comment generated here */
}
4.2.37 Voltage Detection Circuit

The Code Generator outputs the following API functions for the voltage detection circuit.

Table 4.38 API Functions: [Voltage Detection Circuit]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_LVD1&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the voltage detection circuit.</td>
</tr>
<tr>
<td>R_&lt;Config_LVD1&gt;_Start</td>
<td>Starts monitoring of the voltage.</td>
</tr>
<tr>
<td>R_&lt;Config_LVD1&gt;_Stop</td>
<td>Stops monitoring of the voltage.</td>
</tr>
<tr>
<td>R_&lt;Config_LVD1&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the voltage detection circuit.</td>
</tr>
<tr>
<td>r_&lt;Config_LVD1&gt;_lvdn_interrupt</td>
<td>Executes processing in response to voltage monitoring interrupts.</td>
</tr>
</tbody>
</table>
R_<Config_LVD1>_Create

This API function executes initialization processing that is required before controlling the voltage detection circuit.

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

[Syntax]
void R_<Config_LVD1>_Create ( void );

[Argument(s)]
None.

[Return value]
None.
This API function starts monitoring of the voltage.

**[Syntax]**

```
void     R_<Config_LVD1>_Start ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_LVD1>_Stop**

This API function stops monitoring of the voltage.

**[Syntax]**

```c
void R_<Config_LVD1>_Stop ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_LVD1>_Create_UserInit**

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

**Remark**
This API function is called from R_<Config_LVD1>_Create as a callback routine.

**[Syntax]**

```c
void R_<Config_LVD1>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to voltage monitoring interrupts.

**r_<Config_LVD1>_lvdn_interrupt**

**[Syntax]**

```c
void r_<Config_LVD1>_lvdn_interrupt ( void );
```

**Remark**

\( n \) is a circuit number.

**[Argument(s)]**

None.

**[Return value]**

None.
Usage example

Generating a software reset in response to a voltage monitoring interrupt:

main.c

```c
#include "r_smc_entry.h"
void main(void)
{
    /* Start the LVD1 operation */
    R_Config_LVD1_Start();

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

Config_LVD1_user.c

```c
void r_Config_LVD1_lvd1_interrupt(void)
{
    /* Start user code for r_Config_LVD1_lvd1_interrupt. Do not edit comment generated here */
    /* Software reset */
    SYSTEM.PRCR.WORD = 0xA502;
    SYSTEM.SWRR = 0xA501;
    /* End user code. Do not edit comment generated here */
}
```
4.2.38 Watchdog Timer

The Code Generator outputs the following API functions for the watchdog timer (WDT or IWDT).

Table 4.39 API Functions: [Watchdog Timer]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_WDT&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the watchdog timer (WDT or IWDT).</td>
</tr>
<tr>
<td>R_&lt;Config_WDT&gt;_Restart</td>
<td>Clears the counter in the watchdog timer, and then restarts counting by the counter.</td>
</tr>
<tr>
<td>R_&lt;Config_WDT&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the watchdog timer (WDT or IWDT).</td>
</tr>
<tr>
<td>r_&lt;Config_WDT&gt;_wuni_interrupt</td>
<td>Executes processing in response to maskable interrupts or non-maskable interrupts (WUNI).</td>
</tr>
<tr>
<td>r_&lt;Config_WDT&gt;_iwuni_interrupt</td>
<td>Executes processing in response to maskable interrupts or non-maskable interrupts (IWUNI).</td>
</tr>
<tr>
<td>r_&lt;Config_WDT&gt;_nmi_interrupt</td>
<td>Executes processing in response to non-maskable interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_WDT>_Create**

This API function executes initialization processing that is required before controlling the watchdog timer (WDT or IWDT).

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

**[Syntax]**

```c
void R_<Config_WDT>_Create ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function clears the counter in the watchdog timer, and then restarts counting by the counter.

**[Syntax]**

```c
void R_<Config_WDT>_Restart ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes user-specific initialization processing for the watchdog timer (WDT or IWDT).

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

**[Syntax]**

```c
void R_<Config_WDT>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_WDT>_wuni_interrupt**

This API function executes processing in response to maskable interrupts or non-maskable interrupts (WUNI).

**Remark**  This API function is called as the interrupt handler for maskable interrupts or non-maskable interrupts due to underflows or refresh errors of the down-counter.

**[Syntax]**

```c
void r_<Config_WDT>_wuni_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**r_<Config_WDT>_iwuni_interrupt**

This API function executes processing in response to maskable interrupts or non-maskable interrupts (IWUNI).

**Remark** This API function is called as the interrupt handler for maskable interrupts or non-maskable interrupts due to underflows or refresh errors of the down-counter.

**[Syntax]**

```c
void r_<Config_WDT>_iwuni_interrupt ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to non-maskable interrupts.

**Remark** This API function is called as the interrupt handler for non-maskable interrupts due to underflows or refresh errors of the down-counter.

**Syntax**

```c
void r_<Config_WDT>_nmi_interrupt ( void );
```

**Argument(s)**

None.

**Return value**

None.
Usage example

Refreshing the counter value on every loop of the main function and issuing a software reset in response to an underflow of the counter:

main.c

```c
#include "r_smc_entry.h"
void main(void)
{
    while (1U)
    {
        /* Restarts WDT module */
        R_Config_WDT_Restart();
    }
}
```

Config_WDT_user.c

```c
void r_Config_WDT_wuni_interrupt(void)
{
    /* Start user code for r_Config_WDT_wuni_interrupt. Do not edit comment generated here */
    /* Software reset */
    SYSTEM.PRCR.WORD = 0xA502;
    SYSTEM.SWRR = 0xA501;
    /* End user code. Do not edit comment generated here */
}
4.2.39 Continuous Scan Mode DSAD

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

Table 4.40 API Functions: [Continuous Scan Mode DSAD]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the continuous scan mode DSAD.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Start</td>
<td>Starts waiting trigger of A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Stop</td>
<td>Stops A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Set_SoftwareTrigger</td>
<td>Starts A/D conversion by software trigger.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Get_ValueResult</td>
<td>Gets the result of conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Chm_Set_DisconnectDetection</td>
<td>Sets disconnect detection assist.</td>
</tr>
<tr>
<td>r_&lt;Config_DSAD0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the continuous scan mode DSAD.</td>
</tr>
<tr>
<td>r_&lt;Config_DSAD0&gt;_adin_interrupt</td>
<td>Executes processing in response to conversion end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_DSAD0&gt;_scanendn_interrupt</td>
<td>Executes processing in response to scan end interrupts.</td>
</tr>
</tbody>
</table>
**R_<Config_DSAD0>_Create**

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

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

**[Syntax]**

```c
void R_<Config_DSAD0>_Create ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
This API function starts waiting trigger of A/D conversion.

**[Syntax]**

```c
void R(Config_DSAD0)_Start ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
This API function stops A/D conversion.

[Syntax]
void R_<Config_DSAD0>_Stop ( void );

[Argument(s)]
None

[Return value]
None
This API function starts A/D conversion by software trigger.

**[Syntax]**
```c
void R_Config_DSAD0_Set_SoftwareTrigger ( void );
```

**[Argument(s)]**
- None

**[Return value]**
- None
**R_<Config_DSAD0>_Get_ValueResult**

This API function gets the result of conversion.

**[Syntax]**

```c
void R_<Config_DSAD0>_Get_ValueResult ( uint32_t * const buffer );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint32_t * const buffer;</td>
<td>Pointer to the location where the acquired results of conversion are to be stored</td>
</tr>
</tbody>
</table>

**[Return value]**

None
This API function sets the disconnect detection assist.

**[Syntax]**
```c
void  R_<Config_DSAD0>_Chm_Set_DisconnectDetection ( bool pos, bool neg );
```

**Remark1.**  $m$ is channel numbers.

**Remark2.**  Do not call this API during A/D conversion.

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>bool pos;</td>
<td>Enable to disconnect detection assist for positive input signal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>True : Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>False : Disable</td>
</tr>
<tr>
<td>I</td>
<td>bool neg;</td>
<td>Enable to disconnect detection assist for negative input signal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>True : Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>False : Disable</td>
</tr>
</tbody>
</table>

**[Return value]**

None
R_\text{Config\_DSAD0\_Create\_UserInit}

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

Remark This API function is called from R_\text{Config\_DSAD0\_Create} as a callback routine.

[Syntax]

\begin{verbatim}
void R_\text{Config\_DSAD0\_Create\_UserInit} ( void );
\end{verbatim}

[Argument(s)]
None

[Return value]
None
This API function executes processing in response to conversion end interrupts.

[Syntax]
```c
void r_<Config_DSAD0>_adin_interrupt ( void );
```

[Argument(s)]
None

[Return value]
None
This API function executes processing in response to scan end interrupts.

**[Syntax]**

```c
void r_<Config_DSAD0>_scanendn_interrupt ( void );
```

**Remark**

- `n` is unit numbers.

**[Argument(s)]**

- None

**[Return value]**

- None
Usage example

Getting the result value when scan end interrupt occurs.

main.c

```c
#include "r_smc_entry.h"
void main(void)
{
    /* Start the DSAD0 converter */
    R_Config_DSAD0_Start();

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

Config_DSAD0_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_dsad0_value;
/* End user code. Do not edit comment generated here */

void r_Config_DSAD0_scanend0_interrupt(void)
{
    /* Start user code for r_Config_DSAD0_scanend0_interrupt. Do not edit comment generated here */
    /* Get result from the DSAD0 converter */
    R_Config_DSAD0_Get_ValueResult((uint32_t *)&g_dsad0_value);
    /* End user code. Do not edit comment generated here */
}
4.2.40 Single Scan Mode DSAD

The Code Generator outputs the following API functions for the single scan mode DSAD.

Table 4.41 API Functions [Single Scan Mode DSAD]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the single scan mode DSAD.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Start</td>
<td>Starts waiting trigger of A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Stop</td>
<td>Stops A/D conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Set_SoftwareTrigger</td>
<td>Starts A/D conversion by software trigger.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Get_ValueResult</td>
<td>Gets the result of conversion.</td>
</tr>
<tr>
<td>R_&lt;Config_DSAD0&gt;_Chm_Set_DisconnectDetection</td>
<td>Sets disconnect detection assist.</td>
</tr>
<tr>
<td>r_&lt;Config_DSAD0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the single scan mode DSAD.</td>
</tr>
<tr>
<td>r_&lt;Config_DSAD0&gt;_adin_interrupt</td>
<td>Executes processing in response to conversion end interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_DSAD0&gt;_scanendn_interrupt</td>
<td>Executes processing in response to scan end interrupts.</td>
</tr>
</tbody>
</table>
R_<Config_DSAD0>_Create

This API function executes initialization processing that is required before controlling the single scan mode DSAD.

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

[Syntax]
void R_<Config_DSAD0>_Create ( void );

[Argument(s)]
None

[Return value]
None
R_<Config_DSAD0>_Start

This API function starts waiting trigger of A/D conversion.

[Syntax]
void R_<Config_DSAD0>_Start ( void );

[Argument(s)]
None

[Return value]
None
This API function stops A/D conversion.

**[Syntax]**

```c
void R_Config_DSAD0_Stop ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
This API function starts A/D conversion by software trigger.

**[Syntax]**

```c
void R_<Config_DSAD0>_Set_SoftwareTrigger ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
This API function gets the result of conversion.

**[Syntax]**

```c
void R_Config_DSAD0_Get_ValueResult ( uint32_t * const buffer );
```

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint32_t * const buffer;</td>
<td>Pointer to the location where the acquired results of conversion are to be stored</td>
</tr>
</tbody>
</table>

**[Return value]**

None
This API function sets the disconnect detection assist.

**[Syntax]**

```c
void R_<Config_DSAD0>_Chm_Set_DisconnectDetection ( bool pos, bool neg );
```

Remark1.  $m$ is channel numbers.
Remark2.  Do not call this API during A/D conversion.

**[Argument(s)]**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| I   | bool pos; | Enable to disconnect detection assist for positive input signal.  
|     |          | True : Enable  
|     |          | False : Disable |
| I   | bool neg; | Enable to disconnect detection assist for negative input signal.  
|     |          | True : Enable  
|     |          | False : Disable |

**[Return value]**

None
R_<Config_DSAD0>_Create_UserInit

This API function executes user-specific initialization processing for the single scan mode DSAD.

Remark This API function is called from R_<Config_DSAD0>_Create as a callback routine.

[Syntax]
void R_<Config_DSAD0>_Create_UserInit ( void );

[Argument(s)]
None

[Return value]
None
This API function executes processing in response to conversion end interrupts.

**Syntax**

```c
void r_<Config_DSAD0>_adin_interrupt ( void );
```

**Remark**

\( n \) is unit numbers.

**Argument(s)**

None

**Return value**

None
This API function executes processing in response to scan end interrupts.

**[Syntax]**

```
void r_<Config_DSAD0>_scanendn_interruptB ( void );
```

**[Remark]**

$n$ is unit numbers.

**[Argument(s)]**

None

**[Return value]**

None
Usage example

Getting the result value when scan end interrupt occurs.

main.c

```c
#include "r_smc_entry.h"
void main(void)
{
    /* Start the DSAD0 converter */
    R_Config_DSAD0_Start();

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

Config_DSAD0_user.c

```c
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_dsad0_value;
/* End user code. Do not edit comment generated here */

void r_Config_DSAD0_scanend0_interrupt(void)
{
    /* Start user code for r_Config_DSAD0_scanend0_interrupt. Do not edit comment generated here */
    /* Get result from the DSAD0 converter */
    R_Config_DSAD0_Get_ValueResult((uint32_t *)&g_dsad0_value);
    /* End user code. Do not edit comment generated here */
}
```
4.2.41  Delta-Sigma Modulator Interface

The Code Generator outputs the following API functions for the delta-sigma modulator interface.

Table 4.42  API Functions [Delta-Sigma Modulator Interface]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_DSMIF0&gt;_Create</td>
<td>Executes initialization processing that is required before controlling the delta-sigma modulator interface.</td>
</tr>
<tr>
<td>R_&lt;Config_DSMIF0&gt;_Start</td>
<td>Starts conversion</td>
</tr>
<tr>
<td>R_&lt;Config_DSMIF0&gt;_Stop</td>
<td>Stops conversion</td>
</tr>
<tr>
<td>R_&lt;Config_DSMIF0&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for the delta-sigma modulator interface.</td>
</tr>
<tr>
<td>r_&lt;Config_DSMIF0&gt;_ocdin_interrupt</td>
<td>Executes processing in response to overcurrent detection interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_DSMIF0&gt;_scdin_interrupt</td>
<td>Executes processing in response to short-circuit detection interrupts.</td>
</tr>
<tr>
<td>r_&lt;Config_DSMIF0&gt;_sumein_interrupt</td>
<td>Executes processing in response to current sum error detection interrupts.</td>
</tr>
</tbody>
</table>
**R(Config_DSMIF0)_Create**

This API function executes initialization processing that is required before controlling the delta-sigma modulator interface.

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

**[Syntax]**

```c
void R_Config_DSMIF0_Create ( void );
```

**[Argument(s)]**
None

**[Return value]**
None
This API function starts conversion.

[Syntax]

```c
void R_<Config_DSMIF0>_Start ( void );
```

[Argument(s)]

None

[Return value]

None
This API function stops conversion.

**[Syntax]**

```c
void R_Config_DSMIF0_Stop ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
**R_<Config_DSMIF0>_Create_UserInit**

This API function executes user-specific initialization processing for the delta-sigma modulator interface.

**Remark**  This API function is called from R_<Config_DSMIF0>_Create as a callback routine.

**[Syntax]**

```c
void R_<Config_DSMIF0>_Create_UserInit ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
This API function executes processing in response to overcurrent detection interrupts.

**[Syntax]**

```c
void r_<Config_DSIF0>_ocdi_n_interrupt ( void );
```

**Remark**

$n$ is unit numbers.

**[Argument(s)]**

None

**[Return value]**

None
This API function executes processing in response to short-circuit detection interrupts.

**Syntax**

```c
void r_<Config_DSMIF0>_scdin_interrupt ( void );
```

Remark

*n* is unit numbers.

**Argument(s)**

None

**Return value**

None
This API function executes processing in response to current sum error detection interrupts.

[Syntax]
```c
void r_<Config_DSMIF0>_sumein_interrupt ( void );
```

Remark  
$n$ is unit numbers.

[Argument(s)]
None

[Return value]
None
Usage example

Stops conversion when over current detection.

main.c
#include "r_smc_entry.h"
void main(void)
{
    /* Start the DSMIF0 filtering */
    R_Config_DSMIF0_Start();

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

Config_DSMIF_user.c
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_s12ad0_ch000_value;
/* End user code. Do not edit comment generated here */
void r_Config_DSMIF0_ocdi0_interrupt(void)
{
    /* Start user code for r_Config_DSMIF0_ocdi0_interrupt. Do not edit comment generated here */
    /* Stop the DSMIF0 convert */
    R_Config_DSMIF0_Stop();
    /* End user code. Do not edit comment generated here */
4.2.42 Analog Front End

The Code Generator outputs the following API functions for the analog front end.

Table 4.43 API Functions [Delta-Sigma Modulator Interface]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{R}_{\text{Config_AFE}}$ Create</td>
<td>Executes initialization processing that is required before controlling the analog front end.</td>
</tr>
<tr>
<td>$\text{R}_{\text{Config_AFE}}$ Create UserInit</td>
<td>Executes user-specific initialization processing for the analog front end.</td>
</tr>
</tbody>
</table>
This API function executes initialization processing that is required before controlling the analog front end.

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

**[Syntax]**

```c
void R_Config_AFE_Create ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
**R_<Config_AFE>_Create_UserInit**

This API function executes user-specific initialization processing for the analog front end.

**Remark**  
This API function is called from \( R_{<Config_AFE>\_Create} \) as a callback routine.

**[Syntax]**

```c
void R_<Config_AFE>_Create_UserInit ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
### 4.2.43 Motor

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

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_Create</td>
<td>Executes initialization processing for MTU (complementary PWM mode) and S12AD (single scan mode) used for motor.</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_StartTimerCount</td>
<td>Starts timer count</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_StopTimerCount</td>
<td>Stops timer count</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_StartTimerCtrl</td>
<td>Starts timer pulse output</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_StopTimerCtrl</td>
<td>Stops timer pulse output</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_UpdDuty</td>
<td>Updates the buffer of duty register</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_StartAD</td>
<td>Starts A/D converter</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_StopAD</td>
<td>Stops A/D converter</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_AdcGetConvVal</td>
<td>Gets the A/D conversion result</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for motor configuration.</td>
</tr>
<tr>
<td>R_&lt;Config_MTU3_MTU4&gt;_CrestInterrupt</td>
<td>Executes processing in response to crest interrupt</td>
</tr>
<tr>
<td>r_&lt;Config_MTU3_MTU4&gt;_ad_interrupt</td>
<td>Executes processing in response to A/D conversion end interrupt</td>
</tr>
</tbody>
</table>
**R_<Config_MTU3_MTU4>_Create**

This API function executes initialization for MTU (complementary PWM mode) and S12AD (single scan mode) used for motor.

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

**[Syntax]**

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

**[Argument(s)]**

None

**[Return value]**

None
This API function starts timer count.

**[Syntax]**

```c
void R_<Config_MTU3_MTU4>_StartTimerCount ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function stops timer count.

**Syntax**

```c
void R_Config_MTU3_MTU4_StopTimerCount ( void );
```

**Argument(s)**

None.

**Return value**

None.
**R_<Config_MTU3_MTU4>_StartTimerCtrl**

This API function starts timer pulse output.

**[Syntax]**

```c
void R_<Config_MTU3_MTU4>_StartTimerCtrl ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
This API function stops timer pulse output.

**[Syntax]**

```c
void R_<Config_MTU3_MTU4>_StopTimerCtrl ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
**R_<Config_MTU3_MTU4>_UpdDuty**

This API function updates the buffer of duty register.

**[Syntax]**

3-Phase Brushless DC Motor

```c
void     R_<Config_MTU3_MTU4>_UpdDuty (uint16_t duty_u, uint16_t duty_v, uint16_t duty_w);
```

2-Phase Stepping Motor

```c
void     R_<Config_MTU3_MTU4>_UpdDuty (uint16_t duty_a, uint16_t duty_b);
```

**[Argument(s)]**

3-Phase Brushless DC Motor

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint16_t duty_u</td>
<td>U phase duty register value</td>
</tr>
<tr>
<td>O</td>
<td>uint16_t duty_v</td>
<td>V phase duty register value</td>
</tr>
<tr>
<td>O</td>
<td>uint16_t duty_w</td>
<td>W phase duty register value</td>
</tr>
</tbody>
</table>

2-Phase Stepping Motor

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>uint16_t duty_a</td>
<td>A phase duty register value</td>
</tr>
<tr>
<td>O</td>
<td>uint16_t duty_b</td>
<td>B phase duty register value</td>
</tr>
</tbody>
</table>

**[Return value]**

None.
This API function starts A/D converter.

**Syntax**

```c
void R_<Config_MTU3_MTU4>_StartAD ( void );
```

**Argument(s)**

None.

**Return value**

None.
R_<Config_MTU3_MTU4>_StopAD

This API function stops A/D converter.

[Syntax]
void R_<Config_MTU3_MTU4>_StopAD ( void );

[Argument(s)]
None.

[Return value]
None.
**R_<Config_MTU3_MTU4>_AdcGetConvVal**

This API function gets the A/D conversion result.

**[Syntax]**

**3-Phase Brushless DC Motor**

```c
void     R_<Config_MTU3_MTU4>_AdcGetConvVal ( r_mtr_adc_tb *mtr_ad_data );
```

**2-Phase Stepping Motor**

```c
void     R_<Config_MTU3_MTU4>_AdcGetConvVal ( r_mtr_adc_ts *mtr_ad_data );
```

**[Argument(s)]**

**3-Phase Brushless DC Motor**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>r_mtr_adc_tb *mtr_ad_data</td>
<td>Pointer to the location where the read value is to be stored</td>
</tr>
</tbody>
</table>

Structure Definition:
```c
typedef struct {
    uint16_t   u2_iu_ad;
    uint16_t   u2_iv_ad;
    uint16_t   u2_iw_ad;
    uint16_t   u2_vdc_ad;
    uint16_t   u2_vphase_u_ad;
    uint16_t   u2_vphase_v_ad;
    uint16_t   u2_vphase_w_ad;
} r_mtr_adc_tb;
```

**2-Phase Stepping Motor**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>r_mtr_adc_ts *mtr_ad_data</td>
<td>Pointer to the location where the read value is to be stored</td>
</tr>
</tbody>
</table>

Structure Definition:
```c
typedef struct {
    uint16_t   u2_ia_ad;
    uint16_t   u2_ib_ad;
    uint16_t   u2_vdc_ad;
    uint16_t   u2_vphase_a_ad;
    uint16_t   u2_vphase_b_ad;
} r_mtr_adc_ts;
```

**[Return value]**

None.
**R_<Config_MTU3_MTU4>_Create_UserInit**

This API function executes user-specific initialization processing for motor configuration.

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

**[Syntax]**

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

**[Argument(s)]**

None.

**[Return value]**

None.
This API function executes processing in response to crest interrupt (TGIA3 or TGIA6).

**Remark**
This API function is called as the interrupt handler for TGRA3/TGRA6 compare match interrupts, which occur when the current counter value (the value of the timer counter (TCNT)) matches a specified value (the value of the timer general register (TGRA3 or TGRA6)).

**Syntax**

```c
void r_<Config_MTU3_MTU4>_CrestInterrupt ( void );
```

**Argument(s)**
None.

**Return value**
None.
This API function executes processing in response to A/D conversion end interrupt.

Remark Interrupt of one A/D converter unit that has smallest unit number in S12AD units selected on GUI is enabled.

[Syntax]

```c
void r_<Config_MTU3_MTU4>_ad_interrupt ( void );
```

[Argument(s)]
None.

[Return value]
None.
4.2.44 LCD Controller

The Code Generator outputs the following API functions for the LCD Controller.

Table 4.45 API Functions [LCD Controller]

<table>
<thead>
<tr>
<th>API Function Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_&lt;Config_LCD&gt;_Create</td>
<td>Executes initialization processing for LCD Controller.</td>
</tr>
<tr>
<td>R_&lt;Config_LCD&gt;_Start</td>
<td>Starts LCD display</td>
</tr>
<tr>
<td>R_&lt;Config_LCD&gt;_Stop</td>
<td>Stops LCD display</td>
</tr>
<tr>
<td>R_&lt;Config_LCD&gt;_Voltage_On</td>
<td>Enables voltage boost circuit or capacitor split circuit</td>
</tr>
<tr>
<td>R_&lt;Config_LCD&gt;_Voltage_Off</td>
<td>Disables voltage boost circuit or capacitor split circuit</td>
</tr>
<tr>
<td>R_&lt;Config_LCD&gt;_Create_UserInit</td>
<td>Executes user-specific initialization processing for LCD Controller configuration.</td>
</tr>
</tbody>
</table>
**R_<Config_LCD>_Create**

This API function executes initialization for LCD Controller.

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

**[Syntax]**

```c
void R_<Config_LCD>_Create ( void );
```

**[Argument(s)]**

None

**[Return value]**

None
R_<Config_LCD>_Start

This API function starts LCD display.

[Syntax]

```c
void R_<Config_LCD>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.
R_<Config_LCD>_Stop

This API function stops LCD display.

[Syntax]

```c
void R_<Config_LCD>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.
R_<Config_LCD>_Voltage_On

This API function enables voltage boost circuit or capacitor split circuit.

[Syntax]
void R_<Config_LCD>_Voltage_On ( void );

[Argument(s)]
None.

[Return value]
None.
R_\text{<Config_LCD>_Voltage_Off}

This API function disables voltage boost circuit or capacitor split circuit.

[Syntax]

\begin{verbatim}
void R_\text{<Config_LCD>_Voltage_Off ( void );}
\end{verbatim}

[Argument(s)]

None.

[Return value]

None.
**R_<Config_LCD>_Create_UserInit**

This API function executes user-specific initialization processing for LCD Controller configurations.

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

**[Syntax]**

```c
void R_<Config_LCD>_Create_UserInit ( void );
```

**[Argument(s)]**

None.

**[Return value]**

None.
## Revision Record

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>July 01, 2018</td>
<td>1</td>
<td>First Edition issued</td>
</tr>
<tr>
<td>1.01</td>
<td>July 10, 2018</td>
<td>19</td>
<td>Modified remarks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>Modified section name</td>
</tr>
<tr>
<td>1.02</td>
<td>Nov. 05, 2019</td>
<td>19</td>
<td>Added initialization page</td>
</tr>
<tr>
<td></td>
<td></td>
<td>480–</td>
<td>Added new components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>508</td>
<td>- Continuous Scan Mode DSAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Single Scan Mode DSAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Delta-Sigma Modulator Interface</td>
</tr>
<tr>
<td>1.03</td>
<td>Jan. 20, 2020</td>
<td>11,</td>
<td>Added file lists into output files table for components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12,</td>
<td>- General PWM Timer (GPT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18–19</td>
<td>- Low Power Consumption</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Continuous Scan Mode DSAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Single Scan Mode DSAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Delta-Sigma Modulator Interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>166,</td>
<td>Added API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>174</td>
<td>- Software counter clear</td>
</tr>
<tr>
<td></td>
<td></td>
<td>253,</td>
<td>Added API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>263</td>
<td>- Set PGA operating condition setting</td>
</tr>
<tr>
<td>1.04</td>
<td>Jul. 20, 2020</td>
<td>200–</td>
<td>Modified section structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>232</td>
<td>- Sparated API for RIIC and SCI simple I2C mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>207</td>
<td>Modified remarks</td>
</tr>
<tr>
<td>1.05</td>
<td>Oct. 20, 2020</td>
<td>502</td>
<td>Added new API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>513</td>
<td>- Set disconnect detection assist setting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>527–</td>
<td>Added new components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>529</td>
<td>- Analog front end</td>
</tr>
<tr>
<td>1.05</td>
<td>Oct. 20, 2020</td>
<td>530–</td>
<td>Added new components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>542</td>
<td>- Motor</td>
</tr>
<tr>
<td>1.06</td>
<td>Feb. 16, 2022</td>
<td>137–</td>
<td>Splitted the Start and Stop API by U, V and W phases</td>
</tr>
<tr>
<td></td>
<td></td>
<td>142</td>
<td>Modified the API name and parameter description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>219</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>248</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>299</td>
<td>Added new API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>300</td>
<td>- Start/Stop MTU channels’ counters simultaneously</td>
</tr>
<tr>
<td></td>
<td></td>
<td>455–</td>
<td>Modified the data type description for transmit and receive buffer parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>460</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>476</td>
<td>Modified the data type description for transmit and receive buffer parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>1.07</td>
<td>Aug. 01, 2022</td>
<td>122–</td>
<td>Updated DOC component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>125</td>
<td>- Added three parameters API description for the R_&lt;Config_DOC&gt;SetMode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Updated the value type with &quot;uint32_t&quot; support</td>
</tr>
<tr>
<td>Rev.</td>
<td>Date</td>
<td>Page</td>
<td>Summary</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.07</td>
<td>Aug. 01, 2022</td>
<td>206</td>
<td>Added RSCI support information for the simple I2C mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>375 - 418</td>
<td>- Added RSCI support information for the SCI/SCIF Asynchronous Mode component and SCI/SCIF Clock Synchronous Mode component</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Updated the description of r_&lt;Config_SCI0&gt;_receive_interrupt API for the SCI/SCIF Asynchronous Mode component and SCI/SCIF Clock Synchronous Mode component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>437</td>
<td>Added RSCI support information for Smart Card Interface component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>452 - 472</td>
<td>- Added RSCI support information for the simple SPI mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added new API (Executes processing in response to receive error interrupts)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Updated the description of r_&lt;Config_RSPI0&gt;_receive_interrupt API for the SPI Clock Synchronous Mode component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>551 - 557</td>
<td>- Added new component</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- LCD controller</td>
</tr>
</tbody>
</table>