RA Family
IEC 60730/60335 Self Test Library for RA MCU (CM4_CM23)

R01AN5540EJ0101
Rev.1.01
May.31.2024

Introduction

Today, as automatic electronic controls systems continue to expand into many diverse applications, the requirement of reliability and safety are becoming an ever increasing factor in system design.

For example, the introduction of the IEC60730 safety standard for household appliances requires manufactures to design automatic electronic controls that ensure safe and reliable operation of their products.

The IEC60730 standard covers all aspects of product design but Annex H is of key importance for design of Microcontroller based control systems. This provides three software classifications for automatic electronic controls:

1. Class A: Control functions, which are not intended to be relied upon for the safety of the equipment.
   Examples: Room thermostats, humidity controls, lighting controls, timers, and switches.
2. Class B: Control functions, which are intended to prevent unsafe operation of the controlled equipment.
   Examples: Thermal cut-offs and door locks for laundry equipment.
3. Class C: Control functions, which are intended to prevent special hazards
   Examples: Automatic burner controls and thermal cut-outs for closed.

Appliances such as washing machines, dishwashers, dryers, refrigerators, freezers, and Cookers/Stoves will tend to fall under the classification of Class B.

This Application Note provides guidelines of how to use flexible sample software routines to assist with compliance with IEC60730 class B safety standards. These routines have been certified by VDE Test and Certification Institute GmbH and a copy of the Test Certificate is available in the download package for this Application Note (See Note 1 below).

Although these routines were developed using IEC60730 compliance as a basis, they can be implemented in any system for self testing of Renesas MCUs.

The software routines provided are to be used after reset and also during the program execution. The end user has the flexibility of how to integrate these routines into their overall system design but this document and the accompanying sample code provide an example of how to do this.

It is worth noting that the definition of error handling routines is demanded to the user as well as interrupt handler routines. Since errors that are covered by the software routines are very critical (e.g. PC failure) and the correct SW functionality cannot be assured it is strongly recommended to the user to not only rely on SW error handling, but to also use HW safety mechanisms, e.g. the utilization of the Independent Watchdog (IWDT).

Note 1. This document is based on the European Norm EN60335-1:2002/A1:2004 Annex R, in which the Norm IEC 60730-1 (EN60730-1:2000) is used in some points. The *Annex R* of the mentioned Norm contains just a single sheet that jumps to the IEC 60730-1 for definitions, information and applicable paragraphs.

Target

- Device:
  - Renesas RA Family (Arm® Cortex®-M4, Arm® Cortex®-M23) * See next page for series and groups.
- Development environment (one of the following):
  - GNU-GCC ARM Embedded 9.2.1.20191025 / e^2 studio 7.8.0
  - IAR / EWARM Version 8.50.1

The term "RA MCU" used in this document refers to the following products.

Table 1. RA MCU Self-Test Function List

		CPU Core	Arm® Cortex®-M4		Arm® Cortex®-M23	
		Series	RA6	RA4	RA2	
		Group	RA6M1 RA6M2 RA6M3 RA6T1	RA4M1 RA4W1	RA2A1	RA2L1 RA2E1 RA2E2 RA2E3
Test Function	CPU		○	○	○(*1)	○(*1)
	ROM		○	○	○	○
	RAM		○	○	○	○
	Clock		○	○	○	○
	Independent Watchdog Timer (IWDT)		○	○	○	○
	Voltage		○	○	○	○
	ADC	ADC12	○	–	–	○
		ADC14	–	○	–	–
		ADC16	–	–	○	–
	Temperature		○	○	○	○
	Port Output Enable (POE)		○	○	○	○
	GPIO		○	○	○	○

*1: Excluding FPU test

Table of Contents

1. Tests

1.1 CPU

This section describes CPU tests routines. (Reference: *IEC 60730: 1999+A1:2003 Annex H - Table H.11.12.1 CPU*)

This software tests the following CPU registers.

Table 1.1. List of Registers to Be Tested

CPU Core		Arm® Cortex®-M4		Arm® Cortex®-M23	
Group		RA6M1 RA6M2 RA6M3 RA6T1	RA4M1 RA4W1	RA2A1	RA2L1 RA2E1 RA2E2 RA2E3
Register to be Tested	General-purpose Register	R0-R12		R0-R12	
	Control Register	MSP, PSP, LR, PSR(ASPR, IPSR, EPSR), PRIMASK, CONTROL, FAULTMASK, BASEPRI		MSP, PSP, LR, PSR(APSR, IPSR, EPSR), PRIMASK, CONTROL	
	Program Counter	PC		PC	
	FPU Extension Register (S0 - S31)	S0-S31		(Not applicable)	
	FPU Control Register (*1)	CPACR, FPCCR, FPCAR, FPSCR, FPDSCR		(Not applicable)	

*1: Even if the register names are the same, the bit field configuration may differ depending on the device.

The source file cpu_test.c provides implementation of the CPU test using C language and relies on assembly language function to access the registers (that is, `CPU_Test_Control`). File cpu_test_coupling.c is also required to use the coupling test version of the General Purpose Registers. Coupling test relies on assembly language functions:

- `TestGPRsCouplingStart_A`
- `TestGPRsCouplingR1_R3_A`
- `TestGPRsCouplingR4_R6_A`
- `TestGPRsCouplingR7_R9_A`
- `TestGPRsCouplingR10_R12_A`
- `TestGPRsCouplingR0_A`
- `TestGPRsCouplingStart_B`
- `TestGPRsCouplingR1_R3_B`
- `TestGPRsCouplingR4_R6_B`
- `TestGPRsCouplingR7_R9_B`
- `TestGPRsCouplingR10_R12_B`
- `TestGPRsCouplingR0_B`
- `TestGPRsCouplingEnd`

Alternatively, `CPU_Test_General_Low`, `CPU_Test_General_High` assembly language functions are used to test General-purpose registers.

The cpu_test.c source file relies also on `FPU_Control` assembly language function to access the FPU control registers. File fpu_test_coupling.c is also required if using the coupling test version of the FPU extension registers.

- `TestFPUCouplingStart_A`
- `TestFPUCouplingS0_S3_A`
- `TestFPUCouplingS4_S7_A`
- `TestFPUCouplingS8_S11_A`
- `TestFPUCouplingS12_S15_A`
- `TestFPUCouplingS16_S19_A`
- `TestFPUCouplingS20_S23_A`
- `TestFPUCouplingS24_S27_A`
- `TestFPUCouplingS28_S31_A`

- `TestFPUCouplingStart_B`
- `TestFPUCouplingS0_S3_B`
- `TestFPUCouplingS4_S7_B`
- `TestFPUCouplingS8_S11_B`
- `TestFPUCouplingS12_S15_B`
- `TestFPUCouplingS16_S19_B`
- `TestFPUCouplingS20_S23_B`
- `TestFPUCouplingS24_S27_B`
- `TestFPUCouplingS28_S31_B`
- `TestFPUCouplingEnd`

Alternatively, `FPU_Exten` assembly language function is used to test FPU extension registers

The header file cpu_test.h header file provides the interface to the CPU tests.

These tests are testing such fundamental aspects of the CPU operation; the API functions do not have return values to indicate the result of a test. Instead the user of these tests must create an error handling function with the following declaration:

```
extern void CPU_Test_ErrorHandler(void);
```

The CPU test will jump to this function if an error is detected. This function must not return.

All the test functions follow the rules of register preservation following a C function call. Therefore the user can call these functions like any normal C function without any additional responsibilities for saving register values beforehand.

1.1.1 CPU Software API

Table 1.2. Software API Source Files

File Name
cpu_test.h fpu_test.h
cpu_test.c cpu_test_coupling.c fpu_test_coupling.c
TestGPRsCouplingStart_A.asm TestGPRsCouplingR0_A.asm TestGPRsCouplingR1_R3_A.asm TestGPRsCouplingR4_R6_A.asm TestGPRsCouplingR7_R9_A.asm TestGPRsCouplingR10_R12_A.asm TestGPRsCouplingStart_B.asm TestGPRsCouplingR0_B.asm TestGPRsCouplingR1_R3_B.asm TestGPRsCouplingR4_R6_B.asm TestGPRsCouplingR7_R9_B.asm TestGPRsCouplingR10_R12_B.asm TestGPRsCouplingEnd.asm CPU_Test_Control.asm CPU_Test_General_Low.asm CPU_Test_General_High.asm fpu_control.asm fpu_exten.asm TestFPUCouplingStart_A.asm TestFPUCouplingS0_S3_A.asm TestFPUCouplingS4_S7_A.asm TestFPUCouplingS8_S11_A.asm TestFPUCouplingS12_S15_A.asm TestFPUCouplingS16_S19_A.asm TestFPUCouplingS20_S23_A.asm TestFPUCouplingS24_S27_A.asm TestFPUCouplingS28_S31_A.asm TestFPUCouplingStart_B.asm TestFPUCouplingS0_S3_B.asm TestFPUCouplingS4_S7_B.asm TestFPUCouplingS8_S11_B.asm TestFPUCouplingS12_S15_B.asm TestFPUCouplingS16_S19_B.asm TestFPUCouplingS20_S23_B.asm TestFPUCouplingS24_S27_B.asm TestFPUCouplingS28_S31_B.asm TestFPUCouplingEnd.asm

■ cpu_test.c File

Syntax	
`void CPU_Test_All(void)`	
Description	

Runs through all the tests detailed below in the following order:
1. If using Coupling General Purpose Registers tests (*1. see below):
`CPU_Test_GPRsCouplingPartA`
`CPU_Test_GPRsCouplingPartB`
2. If not using Coupling General Purpose Registers test:
`CPU_Test_General_Low`
`CPU_Test_General_High`
3. `CPU_Test_Control`
4. `CPU_Test_PC`
5. If using Coupling FPU extension registers tests (*2. see below):
`FPU_Test_FPUCouplingPartA`
`FPU_Test_FPUCouplingPartB`
6. If not using Coupling FPU extension registers test:
`FPU_Exten`
7. `FPU_Control`
It is the calling function's responsibility to ensure that the processor is in Privileged Mode. If this function is called in unprivileged mode, the test will fail as some of the register bits are not accessible in unprivileged mode. In addition, since the `CPU_Test_Control` function tests stack pointer registers (that is, MSP and PSP), then it is necessary to disable stack pointer monitoring (MSPMPUCTL.ENABLE = 0, PSPMPUCTL.ENABLE = 0) before running the `CPU_Test_All` function and restore its setting after function return.
It is also the calling function's responsibility to ensure no interrupts occur during this test.
If an error is detected then external function `CPU_Test_ErrorHandler` will be called.
See the individual tests for a full description.
*1. A `#define USE_TEST_GPRS_COUPLING` in the code is used to select which functions will be used to test the General Purpose Registers.
*2. A `#define USE_TEST_FPU_COUPLING` in the code is used to select which functions will be used to test the FPU extension registers.	

Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax
`void CPU_Test_PC(void)`
Description
This function tests the Program Counter (PC) register. This provides a confidence check that the PC is working. It tests that the PC is working by calling a function that is located in its own section so that it can be located away from this function, so that when it is called more of the PC Register bits are required for it to work. So that this function can be sure that the function has actually been executed it returns the inverse of the supplied parameter. This return value is checked for correctness. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.
Input Parameters
`NONE` | N/A
Output Parameters
`NONE` | N/A
Return Values
`NONE` | N/A

■ CPU_Test_General_Low.asm File

Syntax
`void CPU_Test_General_Low(void)`
Description
Tests general-purpose registers R0, R1, R2, R3, R4, R5, R6 and R7. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to disable exception during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.
Input Parameters
`NONE` | N/A
Output Parameters
`NONE` | N/A
Return Values
`NONE` | N/A

■ CPU_Test_General_High.asm File

Syntax	
`void CPU_Test_General_High(void)`	
Description	
Tests general-purpose registers R8, R9, R10, R11 and R12. These are the general purpose registers. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to disable exceptions during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

■ CPU_Test_Control.asm File

Syntax
`void CPU_Test_Control(void)`
Description
Tests control registers (Refer to "Table 1.1. List of Registers to Be Tested" because it depends on the device). This test assumes registers R1 to R4 are working. Generally the test procedure for each register is as follows. For each register: 1. Write h'55555555 to. 2. Read back and check value equals h'55555555. 3. Write h'AAAAAAAA to. 4. Read back and check value equals h'AAAAAAAA. Note however that there are some cases where restrictions on specific bits within a register do not allow this procedure. For these cases other test values have been chosen. It is the calling function's responsibility to ensure that the processor is in Privileged Mode. If this function is called in Unprivileged Mode the test will fail as some of the register bits are not accessible in Unprivileged Mode. It is also the calling function's responsibility to disable exceptions during this test. Note: FAULTMASK and PRIMASK are not tested since this test requires exceptions be disabled. Thus they are not activated during the test modifying FAULTMASK and PRIMASK. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.
Input Parameters
`NONE` — N/A
Output Parameters
`NONE` — N/A
Return Values
`NONE` — N/A

■ cpu_test_coupling.c File

Syntax	
`void CPU_Test_GPRsCouplingPartA(void)`	
Description	
Tests general-purpose registers R0 to R12. Coupling faults between the registers are detected. The general-purpose register test consists of Part A and Part B, and this function is Part A. It is the calling function's responsibility to ensure no interrupts occur during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void CPU_Test_GPRsCouplingPartB(void)`	
Description	
Tests general-purpose registers R0 to R12. Coupling faults between the registers are detected. The general-purpose register test consists of Part A and Part B, and this function is Part B. It is the calling function's responsibility to ensure no interrupts occur during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

■ fpu_test_coupling.c File

Syntax	
`void FPU_Test_FPUCouplingPartA (void)`	
Description	
Tests FPU extension registers S0 to S31. Coupling faults between the registers are detected. The FPU extension registers test consists of Part A and Part B, and this function is Part A. It is the calling function's responsibility to ensure no interrupts occur during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void FPU_Test_FPUCouplingPartB(void)`	
Description	
Tests FPU extension registers S0 to S31. Coupling faults between the registers are detected. The FPU extension registers test consists of Part A and Part B, and this function is Part B. It is the calling function's responsibility to ensure no interrupts occur during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

■ fpu_exten.asm File

Syntax	
`void FPU_Exten(void)`	
Description	
Test FPU extension registers S0 to S31. Registers are tested in pairs. For each pair of registers: 1. Write h'55555555 to both. 2. Read both and check they are equal. 3. Write h'AAAAAAAA to both. 4. Read both and check they are equal. It is the calling function's responsibility to disable exception during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

■ fpu_control.asm File

Syntax	
`void FPU_Control(void)`	
Description	
Tests FPU control registers (Refer to "Table 1.1. List of Registers to Be Tested" because it depends on the device). This test assumes registers R1 to R10 are working. Generally the test procedure for each register is as follows. For each register: 1. Write h'55555555 to. 2. Read back and check value equals h'55555555. 3. Write h'AAAAAAAA to. 4. Read back and check value equals h'AAAAAAAA. Note however that there are some cases where restrictions on specific bits within a register do not allow this procedure. For these cases other test values have been chosen. It is the calling function's responsibility to ensure that the processor is in Privileged Mode. If this function is called in Unprivileged Mode the test will fail as some of the register bits are not accessible in Unprivileged Mode. It is also the calling function's responsibility to disable exceptions during this test. If an error is detected then external function `CPU_Test_ErrorHandler` will be called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

1.2 ROM

This section describes the ROM/Flash memory test using CRC routines. (Reference: *IEC 60730: 1999+A1:2003 Annex H – H2.19.4.1 CRC – Single Word*)

CRC is a fault/error control technique which generates a single word or checksum to represent the contents of memory. A CRC checksum is the remainder of a binary division with no bit carry (XOR used instead of subtraction) of the message bit stream, by a predefined (short) bit stream of length n + 1. which represents the coefficients of a polynomial with degree n. Before the division, n zeros are appended to the message stream. CRCs are often used because they are simple to implement in binary hardware and are easy to analyze mathematically.

The ROM test can be achieved by generating a CRC value for the contents of the ROM and saving it.

During the memory self-test, the same CRC algorithm is used to generate another CRC value, which is compared with the saved CRC value. The technique recognizes all one-bit errors and a high percentage of multi-bit errors.

The complicated part of using CRCs is if you need to generate a CRC value that will then be compared with other CRC values produced by other CRC generators. This proves difficult because there are a number of factors that can change the resulting CRC value even if the basic CRC algorithm is the same. This includes the combination of the order that the data is supplied to the algorithm, the assumed bit order in any look-up table used and the required order of the bits of the actual CRC value. This complication has arisen because big- and little-endian systems were developed to work together that employed serial data transfers where bit order became important. Also, some debuggers implement a software break on ROM, in which case the contents of ROM may be rewritten during debugging.

The method of calculating the reference CRC value depends on the toolchain used. For the detailed procedure, refer to Section 2.2 ROM in 2.Example Usage

1.2.1 CRC32 Algorithm

The RA MCU includes a CRC module that includes support for the CRC32. This software set the CRC module to produce a 32-bit CRC32.

- Polynomial = 0x04C11DB7 ($x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$)
- Width = 32 bits
- Initial value = 0xFFFFFFFF
- XOR with h'FFFFFFFF is performed on the output CRC

1.2.2 CRC Software API

The functions in the reminder of this section are used to calculate a CRC value and verify its correctness against a value stored in ROM.

All software is written in ANSI C. The renesas.h header file includes definition of RA MCU registers.

Table 1.3. CRC Software API Source Files

File Name
crc.h crc_verify.h
crc.c CRC_Verify.c

■ CRC_Verify.c File

Syntax	
`bool_t CRC_Verify(const uint32_t ui32_NewCRCValue, const uint32_t ui32_AddrRefCRC)`	
Description	
This function compares a new CRC value with a reference CRC by supplying address where reference CRC is stored.	
Input Parameters	
`const uint32_t ui32_NewCRCValue`	Value of calculated new CRC value.
`const uint32_t ui32_AddrRefCRC`	Address where 32 bit reference CRC value is stored.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Passed, False = Failed

■ crc.c File

Syntax	
`void CRC_Init(void)`	
Description	
Initializes the CRC module. This function must be called before any of the other CRC functions can be.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`uint32_t CRC_Calculate(const uint32_t* pui32_Data, uint32_t ui32_Length)`	
Description	
This function calculates the CRC of a single specified memory area.	
Input Parameters	
`const uint32_t* pui32_Data`	Pointer to start of memory to be tested.
`uint32_t ui32_Length`	Length of the data in long words.
Output Parameters	
`NONE`	N/A
Return Values	
`Uint32_t`	The 32-bit calculated CRC32 value.

The following functions are used when the memory area cannot simply be specified by a start address and length. They provide a way of adding memory areas in ranges/sections. This can also be used if function `CRC_Calculate` takes too long in a single function call.

■ crc.c File

Syntax	
`void CRC_Start(void)`	
Description	
Prepares the module for starting to receive data. Call this once prior to using function `CRC_AddRange`.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void CRC_AddRange( const uint32_t* pui32_Data, uint32_t ui32_Length)`	
Description	
Use this function rather than `CRC_Calculate` to calculate the CRC on data made up of more than one address range. Call `CRC_Start` first then `CRC_AddRange` for each address range required and then call `CRC_Result` to get the CRC value.	
Input Parameters	
`const uint32_t* pui32_Data`	Pointer to start of memory range to be tested.
`uint32_t ui32_Length`	Length of the data in long words.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`uint32_t CRC_Result(void)`	
Description	
Calculates the CRC value for all the memory ranges added using function `CRC_AddRange` since `CRC_Start` was called.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`uint32_t`	The calculated CRC32 value.

1.3 RAM

March tests are a family of tests that are well recognized as an effective way of testing RAM.

A March test consists of a finite sequence of March elements. A March element is a finite sequence of operations applied to every cell in the memory array before proceeding to the next cell.

In general, the more March elements the algorithm consists of, the better its fault coverage will be but at the expense of a slower execution time.

The algorithms themselves are destructive (they do not preserve the current RAM values) but the supplied test functions provide a non-destructive option so that memory contents can be preserved. This is achieved by copying the memory to a supplied buffer before running the actual algorithm and then restoring the memory from the buffer at the end of the test. The API includes an option for automatically testing the buffer as well as the RAM test area.

The area of RAM being tested cannot be used for anything else while it is being tested. This makes the testing of RAM used for the stack particularly difficult. To help with this problem the API includes functions which can be used for testing the stack.

The following section introduces the specific March Tests. Following that is the specification of the software APIs.

1.3.1 RAM Test Algorithms

(1) March C

The March C algorithm (van de Goor 1991) consists of 6 March elements with a total of 10 operations. It detects the following faults.

1. Stuck-At Faults (SAF)
   - The logic value of a cell or a line is always 0 or 1.
2. Transition Faults (TF)
   - A cell or a line that fails to undergo a 0→1 or a 1→0 transition.
3. Coupling Faults (CF)
   - A write operation to one cell changes the content of a second cell.
4. Address Decoder Faults (AF)

   Any fault that affects the address decoder.
   - With a certain address, no cell will be accessed.
   - A certain cell is never accessed.
   - With a certain address, multiple cells are accessed simultaneously.
   - A certain cell can be accessed by multiple addresses.

These are the 6 March elements.

1. Write all zeros to array.
2. Starting at lowest address, read zeros, write ones, increment up array bit by bit.
3. Starting at lowest address, read ones, write zeros, increment up array bit by bit.
4. Starting at highest address, read zeros, write ones, decrement down array bit by bit.
5. Starting at highest address, read ones, write zeros, decrement down array bit by bit.
6. Read all zeros from array.

(2) March X

Note: This algorithm has not been implemented for the RA MCU and is only presented here for information as it relates to the March X WOM version below.

The March X algorithm consists of 4 March elements with a total of 6 operations. It detects the following faults.

1. Stuck-At Faults (SAF)
2. Transition Faults (TF)
3. Inversion Coupling Faults (CFin)
4. Address Decoder Faults (AF)

These are the 4 March elements.

1. Write all zeros to array.
2. Starting at lowest address, read zeros, write ones, increment up array bit by bit.
3. Starting at highest address, read ones, write zeros, decrement down array bit by bit.
4. Read all zeros from array.

(3) March X (Word-Oriented Memory Version)

The March X Word-Oriented Memory (WOM) algorithm has been created from a standard March X algorithm in two stages. First, the standard March X is converted from using a single-bit data pattern to using a data pattern equal to the memory access width. At this stage the test is primarily detecting inter-word faults including Address Decoder faults. The second stage is to add an additional two March elements. The first uses a data pattern of alternating high/low bits then the second uses the inverse. The addition of these elements is to detect intra-word coupling faults.

These are the 6 March elements.

1. Write all zeros to array.
2. Starting at lowest address, read zeros, write ones, increment up array word by word.
3. Starting at highest address, read ones, write zeros, decrement down word by word.
4. Starting at lowest address, read zeros, write h’AAs, increment up array word by word.
5. Starting at highest address, read h’AAs, write h’55s, decrement down word by word.
6. Read all h’55s from array.

1.3.2 RAM Software API

Two implementations of the RAM tests are available.

(a) Standard implementation.

(b) HW (Hardware) implementation:
This version uses the DOC (Data Operation Circuit) and optionally a DTC (Data Transfer Controller) to help perform the tests.

Both implementations share the same core API but the 'HW' implementation has some additional functions. Please see details in Section 1.3.2 1.1.1(3).

(1) March C API

This test can be configured to use 8-, 16- or 32-bit RAM accesses.

This is achieved by #defining `RAMTEST_MARCH_C_ACCESS_SIZE` in the header file to be one of the following.

1. `RAMTEST_MARCH_C_ACCESS_SIZE_8BIT`
2. `RAMTEST_MARCH_C_ACCESS_SIZE_16BIT`
3. `RAMTEST_MARCH_C_ACCESS_SIZE_32BIT`

Sometimes limiting the maximum size of RAM that can be tested with a single function call can speed the test up as well as reducing stack and code size. This is done by limiting the size of the variable used to hold the number of 'words' that the test area contains. The 'word' size is the selected access width.

This is achieved by `#defining RAMTEST_MARCH_C_MAX_WORDS` in the header file to be one of the following.

1. `RAMTEST_MARCH_C_MAX_WORDS_8BIT` (Max words in test area is 0xFF)
2. `RAMTEST_MARCH_C_MAX_WORDS_16BIT` (Max words in test area is 0xFFFF)
3. `RAMTEST_MARCH_C_MAX_WORDS_32BIT` (Max words in test area is 0xFFFFFFFF)

Table 1.4. March C API Source Files

Implementation	Header Files	Source Files
Standard	ramtest_march_c.h	ramtest_march_c.c
HW	ramtest_march_c.h	ramtest_march_c_HW.c
	ramtest_march_HW.h	ramtest_march_HW.c

Select either the standard implementation or the HW implementation fileset for the compiler build target.

The source is written in ANSI C and uses renesas.h header file to access peripheral registers.

Note: The API allows just a single word to be tested with a function call. However, for coupling faults to be tested between words, it is important to use the functions to test a data range bigger than one word.

■ ramtest_march_c.c File

Syntax	
`bool_t RamTest_March_C( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
RAM memory test using March C (Goor 1991) algorithm.	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned with the selected memory access width and be a value greater or equal to ui32_StartAddr.
`void* const p_RAMSafe`	For a destructive memory test, set to NULL. For a non-destructive memory test, set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

Syntax	
`bool_t RamTest_March_C_Extra( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
Non Destructive RAM memory test using March C (Goor 1991) algorithm. This function differs from the `RamTest_March_C` function by testing the 'RAMSafe' buffer before using it. If the test of the 'RAMSafe' buffer fails then the test will be aborted and the function will return false.	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned with the selected memory access width and be a value greater or equal to `ui32_StartAddr`.
`void* const p_RAMSafe`	Set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

(2) March X WOM API

This test can be configured to use 8-, 16- or 32-bit RAM accesses.

This is achieved by `#defining RAMTEST_MARCH_X_WOM_ACCESS_SIZE` in the header file to be one of the following.

- `RAMTEST_MARCH_X_WOM_ACCESS_SIZE_8BIT`
- `RAMTEST_MARCH_X_WOM_ACCESS_SIZE_16BIT`
- `RAMTEST_MARCH_X_WOM_ACCESS_SIZE_32BIT`

In order to speed up the run time of the test you can choose to limit the maximum size of RAM that can be tested with a single function call. This is done by limiting the size of the variable used to hold the number of 'words' that the test area contains. The 'word' size is the same as the selected access width.

This is achieved by `#defining RAMTEST_MARCH_X_WOM_MAX_WORDS` in the header file to be one of the following.

- `RAMTEST_MARCH_X_WOM_MAX_WORDS_8BIT` (Max words in test area is 0xFF)
- `RAMTEST_MARCH_X_WOM_MAX_WORDS_16BIT` (Max words in test area is 0xFFFF)
- `RAMTEST_MARCH_X_WOM_MAX_WORDS_32BIT` (Max words in test area is 0xFFFFFFFF)

Table 1.5. March X WOM Source Files

Implementation	Header Files	Source Files
Standard	ramtest_march_x_wom.h	ramtest_march_x_wom.c
HW	ramtest_march_x_wom.h	ramtest_march_x_wom_HW.c
	ramtest_march_HW.h	ramtest_march_HW.c

Select either the standard implementation or the HW implementation fileset for the compiler build target.

The source is written in ANSI C and uses renesas.h header file to access peripheral registers.

Note: The API allows just a single word to be tested with a function call. However, for coupling faults to be tested between words it is important to use the functions to test a data range bigger than one word.

■ ramtest_march_x_wom.c File

Syntax	
`bool_t RamTest_March_X_WOM(const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
RAM memory test based on March X algorithm converted for WOM.	
Input Parameters	
`const uint32_t ui32_StartAddr`	Address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`const uint32_t ui32_EndAddr`	Address of the last word of RAM to be tested. This must be aligned with the selected memory access width and be a value greater or equal to `ui32_StartAddr`.
`void* const p_RAMSafe`	For a destructive memory test, set to NULL.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

Syntax	
`bool_t RamTest_March_X_WOM_Extra( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
Non-Destructive RAM memory test based on March X algorithm converted for WOM. This function differs from the `RamTest_March_X_WOM` function by testing the 'RAMSafe' buffer before using it. If the test of the 'RAMSafe' buffer fails then the test will be aborted and the function will return false.	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned with the selected memory access width and be a value greater or equal to `ui32_StartAddr`.
`void* const p_RAMSafe`	Set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

(3) March C and March X WOM HW Implementation Specific API.

The 'HW' implementations of the March C and the March X WOM tests use the DOC (Data Operation Circuit) and optionally a DTC to help perform the tests. The DTC is used to initialize the RAM under test and the DOC is used to compare values read back from RAM with expected values.

It is the user's responsibility to ensure that nothing else accesses the DOC or chosen DTC during the RAM tests.

Table 1.6. HW Implementation Files

Test	Header Files	Source Files
(1) March C	ramtest_march_c.h	ramtest_march_c_HW.c
(2) March X WOM	ramtest_march_x_wom.h	ramtest_march_x_wom_HW.c
HW Initialization (Common)	ramtest_march_HW.h	ramtest_march_HW.c

The optional use of the DTC is controlled using the following #defines.

Table 1.7. Defining DTC Options

File Containing Definitions	#define	Meaning if #defined
ramtest_march_HW.h	RAMTEST_USE_DTC	Initialize and use DTC

■ ramtest_march_HW.c File (Initialization)

Syntax	
`void RamTest_March_HW_Init(void)`	
Description	
Initialize the hardware (DOC and optionally DTC) used by the 'HW' implementations of the RAM tests. The DTC is only used if `RAMTEST_USE_DTC` is defined. Call this function before using any other RAM Test function that uses a HW implementation.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`bool_t RamTest_March_HW_PreTest(void)`	
Description	
This may be used to check if the hardware (DOC and DTC) are functioning correctly before using. A quick functional test of the DOC and (if `RAMTEST_USE_DTC` is defined) the DTC is performed.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test failed.

Syntax	
`bool_t RamTest_March_HW_Is_Init(void)`	
Description	
Checks if `RamTest_March_HW_Init` has been called. This is used by specific RAM tests to check that the HW has been initialized before trying to use it. A user does not have to use this function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

■ ramtest_march_c_HW.c File

Syntax	
`bool_t RamTest_March_C_DTC(const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
[HW implementations version] RAM memory test using March C (Goor 1991) algorithm.	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be aligned with the selected memory access width and be a value greater or equal to ui32_StartAddr.
`void* const p_RAMSafe`	For a destructive memory test, set to NULL. For a non-destructive memory test, set to the start of a buffer that is large enough to copy the contents of the test area into it and that is aligned with the selected memory access width.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

■ ramtest_march_x_won_HW.c File

Syntax	
`bool_t RamTest_March_X_WOM_DTC(const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
[HW implementations version] RAM memory test based on March X algorithm converted for WOM.	
Input Parameters	
`const uint32_t ui32_StartAddr`	Address of the first word of RAM to be tested. This must be aligned with the selected memory access width.
`const uint32_t ui32_EndAddr`	Address of the last word of RAM to be tested. This must be aligned with the selected memory access width and be a value greater or equal to `ui32_StartAddr`.
`void* const p_RAMSafe`	For a destructive memory test, set to NULL.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

(4) RAM Test Stack API

This API enables a RAM test to be performed on an area of RAM that includes the stack. As the function that performs the RAM test requires a stack these functions will, re-locate the stack to a supplied new RAM area allowing the original stack area to be tested. Three functions are provided that can be called depending upon which stack (Main or Process) is in the test area or if both are.

It is the calling function's responsibility to ensure that the processor is in Privileged Mode. If this function is called in unprivileged mode the test will fail as some of the register bits are not accessible in unprivileged mode.

Note: The stack testing functions make use of one of the March RAM tests presented previously by passing it in as a function pointer. If using a test that requires initialization before use it is the user's responsibility to ensure this has been done before trying to use the test by calling one of these functions.

Table 1.8. RAM Test Stack API Source Files

File Name
ramtest_stack.h
ramtest_stack.c
StartBothTestAssembly.asm StartMainTestAssembly.asm StartProcTestAssembly.asm

■ ramtest_stack.c File

Syntax	

`bool_t RamTest_Stack_Main(const uint32_t ui32_StartAddr,`
`const uint32_t ui32_EndAddr,`
`void* const p_RAMSafe,`
`const uint32_t ui32_NewMSP,`
`const TEST_FUNC fpTest_Func)`	

Description	
RAM test of an area that includes the Main Stack (but not includes the Process stack).	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be compatible with the requirements of `fpTest_Func`.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be compatible with the requirements of `fpTest_Func`.
`void* const p_RAMSafe`	Set to the start of a buffer that is the same size as the test RAM area. This must be compatible with the requirements of `fpTest_Func`.
`const uint32_t ui32_NewUSP`	New Stack pointer value for the Main stack to be relocated to.
`const TEST_FUNC fpTest_Func`	Function pointer of type `TEST_FUNC` to the actual memory test to be used. `Typedef bool_t(*TEST_FUNC)( uint32_t, uint32_t, void*);` For example, `'RamTest_March_X_WOM'`.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

Syntax	

`bool_t RamTest_Stack_Proc(const uint32_t ui32_StartAddr,`
`const uint32_t ui32_EndAddr,`
`void* const p_RAMSafe,`
`const uint32_t ui32_NewPSP,`
`const TEST_FUNC fpTest_Func)`	

Description	
RAM test of an area that includes the Process stack (but not includes the Main stack).	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be compatible with the requirements of the `fpTest_Func`.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be compatible with the requirements of the `fpTest_Func`.
`void* const p_RAMSafe`	Set to the start of a buffer that is the same size as the test RAM area. This must be compatible with the requirements of the `fpTest_Func`.
`const uint32_t ui32_NewPSP`	New Stack pointer value for the Process stack to be relocated to.
`const fpTest_Func`	Function pointer of type `TEST_FUNC` to the actual memory test to be used. `Typedef bool_t(*TEST_FUNC)(uint32_t, uint32_t, void*);` For example `'RamTest_March_X_WOM'`.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

Syntax	

`bool_t RamTest_Stacks(const uint32_t ui32_StartAddr,`
`const uint32_t ui32_EndAddr,`
`void* const p_RAMSafe,`
`const uint32_t ui32_NewPSP,`
`const uint32_t ui32_NewMSP,`
`const TEST_FUNC fpTest_Func)`	

Description	
RAM test of an area that includs both the Main stack and the Process stack.	
Input Parameters	
`const uint32_t ui32_StartAddr`	The address of the first word of RAM to be tested. This must be compatible with the requirements of the `fpTest_Func`.
`const uint32_t ui32_EndAddr`	The address of the last word of RAM to be tested. This must be compatible with the requirements of the `fpTest_Func`.
`void* const p_RAMSafe`	Set to the start of a buffer that is the same size as the test RAM area. This must be compatible with the requirements of the `fpTest_Func`.
`const uint32_t ui32_NewPSP`	New Stack pointer value for the Process stack to be relocated to.
`const uint32_t ui32_NewMSP`	New Stack pointer value for the Main stack to be relocated to.
`const TEST_FUNC fpTest_Func`	Function pointer of type `TEST_FUNC` to the actual memory test to be used. `Typedef bool_t(*TEST_FUNC)(const uint32_t, const uint32_t, void* const);` For example: `RamTest_March_X_WOM`
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test or parameter check failed.

1.4 Clock

The RA MCU has a Clock Frequency Accuracy Measurement Circuit (CAC). The CAC counts the pulses of the target clock within the time generated by the reference clock and generates an interrupt request if the number of pulses is outside the acceptable range.

The main clock oscillator also has an oscillation stop detection circuit.

1.4.1 Main Clock Frequency Monitoring by CAC

Either one of Main, SUB_CLOCK, HOCO, MOCO, LOCO, IWDTCLK, and PCLKB or an External clock on the CACREF pin can be used as a reference clock source.

(a) When using an external reference clock:

- `#define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK` in clock_monitor.h file.
- Be sure to provide target and reference clocks frequency in Hz.

(b) When using one of the internal clock source:

- Ensure `CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK` is not defined.
- Be sure to select the reference clock (through ref_clock input parameter).
- Be sure to provide target and reference clocks frequency in Hz.

If the frequency of the main clock deviates during runtime from a configured range, two types of interrupt can be generated: frequency error interrupt or an overflow interrupt. The user of this module must enable these two kinds of interrupt and handle them. See Section 2.4 for an example of interrupt activation. The allowable frequency range can be adjusted using.

```
/* Percentage tolerance of main clock allowed before an error is reported.*/
#define CLOCK_TOLERANCE_PERCENT 10
```

When using the internal clock as the reference clock, the reference clock division ratio in the CAC circuit (RCDS [1: 0] in the CACR2 register) is fixed at 1/128 in the test function.

The division ratio of the target clock (TCSS [1: 0] in the CACR1 register) is selected from 1/1, 1/4, 1/8, 1/32 by calculation in the test function based on the input parameters. However, no matter which division ratio is applied, an error occurs if the calculation result is not within the range that can be set in the 16-bit wide "CAC Upper-Limit and Lower-Limit Value Setting Register".

1.4.2 Oscillation Stop Detection of Main Clock

The main clock oscillator of the RA MCU has an oscillation stop detection circuit. If the main clock stops, the Middle-Speed On-Chip oscillator (MOCO) will automatically be used instead and an NMI interrupt will be generated.

In the `ClockMonitor_Init` function, when the main clock oscillator stop bit (MOSTP) in the main clock oscillator control register (MOSCCR) is 0 (main clock oscillator operation), oscillation stop detection and NMI is enabled as follows.

- Oscillation stop detection control register (OSTDCR)
  - Oscillation stop detection function enable bit (OSTDE): Enable
  - Oscillation stop detection interrupt enable bit (OSTDIE): Enable
- ICU non-maskable interrupt enable register (NMIER)
  - Oscillation stop detection interrupt enable bit (OSTEN): Enable

The user of this module must handle the NMI interrupt and check the `NMISR.OSTST` (Oscillation Stop Detection Interrupt Status Flag) bit.

Table 1.9. Clock Source Files

File Name
clock_monitor.h
clock_monitor.c

The test module relies on the renesas.h header file to access to peripheral registers.

■ clock_monitor.c File

There are two versions of the `ClockMonitor_Init` function.

(a) `ClockMonitor_Init` Function When Using an External Reference Clock. (If `CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK` Is Defined.)

Syntax

```
void ClockMonitor_Init( clock_source_t target_clock,
                        uint32_t MainClockFrequency,
                        uint32_t ExternalRefClockFrequency,
                        CLOCK_MONITOR_CACREF_PIN ePin,
                        CLOCK_MONITOR_ERROR_CALL_BACK CallBack)
```

Description

1. Start monitoring the target clock selected through `target_clock` input parameter using the CAC module and the CACREF pin as a reference clock.
2. The CACREF pin can be selected by SW (for details, refer to Section 2.4 Clock in Chapter 2 Example Usage). It is the user's responsibility to select the terminals based on the system configuration.
3. Enables Oscillation Stop Detection and configures an NMI to be generated if detected.

Input Parameters

`clock_source_t target_clock`	Target clock monitored by CAC. The clock shall be one among Main clock, Sub clock, HOCO clock, MOCO clock, LOCO clock, IWDTCLK clock and PCLKB clock.
`uint32_t MainClockFrequency`	Target clock expected frequency in Hz. (The parameter name is MainClockFrequency, but it is the frequency of the target clock specified by target_clock.)
`uint32_t ExternalRefClockFrequency`	External reference clock frequency in Hz.
`CLOCK_MONITOR_CACREF_PIN ePin`	The pin to use for CACREF.
`CLOCK_MONITOR_ERROR_CALL_BACK CallBack`	A function that is called when the target clock is out of tolerance or when this function fails to properly configure the CAC circuit from the input parameters.

Output Parameters

`NONE`	N/A

Return Values

`NONE`	N/A

(b) `ClockMonitor_Init` Function When Using One of the Internal Clock Source for Reference Clock. (If `CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK` Is Not Defined.)

Syntax	

`void ClockMonitor_Init( clock_source_t target_clock,`
`clock_source_t ref_clock,`
`uint32_t target_clock_frequency,`
`uint32_t ref_clock_frequency,`
`CLOCK_MONITOR_ERROR_CALL_BACK CallBack)`	

Description	
1. Start monitoring the target clock selected through `target_clock` input parameter using the CAC module and the reference clock selected through `ref_clock` input parameter. 2. Enables Oscillation Stop Detection and configures an NMI to be generated if detected.	
Input Parameters	
`clock_source_t target_clock`	Target clock monitored by CAC. The clock shall be one of Main clock, Sub clock, HOCO clock, MOCO clock, LOCO clock, IWDTCLK clock, and PCLKB clock.
`clock_source_t ref_clock`	The reference clock to be used by CAC to monitor the target clock. The clock shall be one of Main clock, Sub clock, HOCO clock, MOCO clock, LOCO clock, IWDTCLK clock, and PCLKB clock.
`uint32_t target_clock_frequency`	The target clock frequency in Hz
`uint32_t ref_clock_frequency`	The reference clock frequency in Hz.
`CLOCK_MONITOR_ERROR_CALL_BACK CallBack`	A function that is called when the target clock is out of tolerance or when this function fails to properly configure the CAC circuit from the input parameters.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`extern void cac_ferrf_isr(void)`	
Description	
CAC frequency error interrupt handler. This function calls the callback function registered by the `ClockMonitor_Init` function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`extern void cac_ovff_isr(void)`	
Description	
CAC overflow error interrupt handler. This function calls the callback function registered by the `ClockMonitor_Init` function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

1.5 Independent Watchdog Timer (IWDT)

A watchdog timer is used to detect abnormal program execution. If a program is not running as expected, the watchdog timer will not be refreshed by software as required and will therefore detect an error.

The Independent Watchdog Timer (IWDT) module of the RA MCU is used for this. It includes a windowing feature so that the refresh must happen within a specified 'window' rather than just before a specified time. It can be configured to generate an internal reset or a NMI interrupt if an error is detected.

All the configurations for IWDT can be done through the Option Function Select Register 0 (OFS0) in Option-Setting Memory whose settings are controlled by the user (see Section 2.5 for an example of configuration). The option setting memory is a series of registers that can be used to select the state of the microcontroller after reset and is located in the code flash area.

A function is provided to be used after a reset to decide if the IWDT has caused the reset.

The test module relies on the renesas.h header file to access to peripheral registers.

Table 1.10. Independent Watchdog Timer Source Files

File Name
iwdt.h
iwdt.c

Syntax	
`void IWDT_Init (void)`	
Description	
Initialize the independent watchdog timer. After calling this, the `IWDT_Kick` function must then be called at the correct time to prevent a watchdog timer error. Note: If configured to produce an interrupt then this will be the Non Maskable Interrupt (NMI). This must be handled by user code which must check the NMISR.IWDTST flag.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void IWDT_Kick(void)`	
Description	
Refresh the watchdog timer count.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`bool_t IWDT_DidReset(void)`	
Description	
Returns true if the IWDT has timed out or not been refreshed correctly. This can be called after a reset to decide if the watchdog timer caused the reset.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
`bool_t`	True if watchdog timer has timed out, otherwise false.

1.6 Voltage

The RA MCU has a Low Voltage Detection (LVD) module. This can be used to detect the power supply voltage (VCC) falling below a specified voltage. The supplied sample code demonstrates using Voltage Detection Circuit 1 to generate a NMI interrupt when VCC drops below a specified level. The hardware is also capable of generating a reset but this behavior is not supported in the sample code.

The test module relies on the renesas.h header file to access to peripheral registers.

Table 1.11. Voltage Source Files

File Name
voltage.h
voltage.c

Syntax	
`void VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL eVoltage)`	
Description	
Initialize and start voltage monitoring. An NMI will be generated if VCC falls below the specified voltage. Note: The Non-Maskable Interrupt (NMI) must be handled by user code which must check the NMISR.LVDST flag. Note: The voltage threshold eVoltage shall be set to a value lower than the nominal Vcc one.	
Input Parameters	
`VOLTAGE_MONITOR_LEVEL eVoltage`	The specified low voltage level. See declaration of enumerated type `VOLTAGE_MONITOR_LEVEL` in voltage.h for details.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

1.7 ADC

The ADC has a diagnostic mode that can be used to test the ADC. The diagnostic mode can be configured so that a test is performed every time the ADC is used normally for a conversion. The diagnostic reference voltage and hence the expected result is automatically rotated between zero, half scale, and full scale (zero, +full scale, -full scale for RA2A1).

The ADC (A/D converter) built into each RA microcomputer and the diagnostic reference voltage are shown below.

Table 1.12. ADC in RA MCU

Group	RA6M1 RA6M2 RA6M3 RA6T1	RA4M1 RA4W1	RA2L1 RA2E1 RA2E2 RA2E3	RA2A1
ADC	ADC12	ADC14	ADC14	ADC16
Number of units	2	1	1	1
Diagnostic reference voltage	Zero / Half Scale / Full Scale			Zero / +Full Scale / -Full Scale

The diagnostic SW provides an AD conversion for the diagnostic reference voltage.

There are 2 units of ADC12 of RA6M1/RA6M2/RA6M3/RA6T1 (Unit 0 and Unit 1). When testing unit 1, use the function with "`_u1`" in its name.

The test module relies on the renesas.h header file to access to peripheral registers.

Table 1.13. ADC Source Files

File Name
test_adc.h
test_adc.c

Syntax	
`void Test_ADC_Init(void)`	
Description	
Initialize the ADC module (Unit 0). This must be called before using any other ADC functions.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void Test_ADC_Init_u1(void)`	
Description	
Initializes ADC module Unit 1. This must be called before using any other ADC functions.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`bool_t Test_ADC_Wait(void)`	
Description	
This function waits while the AD conversion is being performed by the ADC module (Unit 0). This test does not preserve ADC configuration and is therefore suitable as a power-on test rather than as a run-time periodic test.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test failed.

Syntax	
`bool_t Test_ADC_Wait_u1(void)`	
Description	
This function waits while AD conversion is being performed by ADC module Unit 1. This test does not preserve ADC configuration and is therefore suitable as a power-on test rather than as a run-time periodic test.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test failed.

Syntax	
`void Test_ADC_Start(const ADC_ERROR_CALL_BACK Callback)`	
Description	
Set up the ADC module (Unit 0) so that diagnostic tests are performed each time the ADC is used. The diagnostic reference voltage (See "Table 1.12. ADC in RA MCU") is automatically rotated. User code must now call the `Test_ADC_CheckResult` function either periodically or following every ADC completion to check the diagnostic result.	
Input Parameters	
`const ADC_ERROR_CALL_BACK Callback`	Function to call if an error is detected. Note: This function will only get called if `Test_ADC_CheckResult` is called with parameter `bCallErrorHandler` set true.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void Test_ADC_Start_u1(const ADC_ERROR_CALL_BACK Callback)`	
Description	
Set up ADC module Unit 1 so that diagnostic tests are performed each time the ADC is used. The diagnostic reference voltage (See "Table 1.12. ADC in RA MCU") is automatically rotated. User code must now call the `Test_ADC_CheckResult_u1` function either periodically or following every ADC completion to check the diagnostic result.	
Input Parameters	
`const ADC_ERROR_CALL_BACK Callback`	Function to call if an error is detected. Note: This function will only get called if `Test_ADC_CheckResult_u1` is called with parameter `bCallErrorHandler` set true.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`bool_t Test_ADC_CheckResult(const bool_t bCallErrorHandler)`	
Description	
Check that the ADC module (Unit 0) diagnostic result is as expected. This must be called after `Test_ADC_Start` and then be called periodically or whenever an ADC conversion completes. Note: The actual result is allowed to be with a certain tolerance of the expected result. See `ADC_TOLERANCE` in test_adc.c for details.	
Input Parameters	
`const bool_t bCallErrorHandler`	Set true to call the error call-back function supplied to function `Test_ADC_Start`, otherwise false.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test failed.

Syntax	
`bool_t Test_ADC_CheckResult_u1(const bool_t bCallErrorHandler)`	
Description	
Check that the ADC module Unit 1 diagnostic result is as expected. This must be called after `Test_ADC_Start_u1` and then be called periodically or whenever an ADC conversion completes. Note: The actual result is allowed to be with a certain tolerance of the expected result. See `ADC_TOLERANCE` in test_adc.c for details.	
Input Parameters	
`const bool_t bCallErrorHandler`	Set true to call the error call-back function supplied to function `Test_ADC_Start_u1`, otherwise false.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Test passed, False = Test failed.

1.8 Temperature

The RA MCU has a Temperature Sensor module that can monitor the MCU temperature. The ADC module unit 1 is also required in conjunction with the Temperature Sensor.

The test module relies on the renesas.h header file to access to peripheral registers.

Table 1.14. Temperature Source Files

File Name
temperature.h
temperature.c

Syntax	

`void Temperature_Init( uint16_t Temperature_ADC_Value_Min,`
`uint16_t Temperature_ADC_Value_Max,`
`TEMPERATURE_ERROR_CALL_BACK Error_callback)`	

Description	
Initializes the Temperature Sensor and enables the ADC module. Specify an allowed temperature range in terms of ADC output values. After calling this function, the `Temperature_Start` function must be called periodically to perform an ADC conversion on the Temperature Sensor output and then the remaining functions must be used to check the result. Please note that the sensor temperature slope is negative, thus a temperature increase implies a decrease of the ADC converted value, while a temperature decrease implies an increase of ADC converted value. For details, refer to the "Temperature Sensor" and "A/D Converter" chapters in the “RA MCU User’s Manual: Hardware”.	
Input Parameters	
`uint16_t Temperature_ADC_Value_Min`	Specify the minimum value that the ADC should output when reading the temperature sensor.
`uint16_t Temperature_ADC_Value_Max`	Specify the maximum value that the ADC should output when reading the temperature sensor.
`TEMPERATURE_ERROR_CALL_BACK Error_callback`	This function will be called by function `Temperature_CheckResult` if the temperature (ADC value) is outside the specified allowable range.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void Temperature_Start(void)`	
Description	
Starts an ADC conversion to read the temperature. This will use the ADC module, destroying its current settings. It is the user's responsibility to ensure this behavior is not a problem. Following this function use function `Temperature_Read_Wait` or `Temperature_CheckResult`.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void Temperature_Wait_Finish (void)`	
Description	
This function blocks until a temperature conversion, started by `Temperature_Start`, has completed.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`uint16_t Temperature_Read_Wait (void)`	
Description	
This function blocks until a temperature conversion, started by `Temperature_Start`, has completed and then returns the ADC value.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
`uint16_t`	ADC output value

Syntax	
`bool_t Temperature_CheckResult(bool_t bCallErrorHandler)`	
Description	
This function blocks until a temperature conversion, started by `Temperature_Start`, has completed and then checks if the ADC value is within the range specified in `Temperature_Init`.	
Input Parameters	
`bool_t bCallErrorHandler`	Set true to get the callback registered in `Temperature_Init` called if the temperature falls outside the specified limits, otherwise set false.
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = Result falls within specified limits, False = Result falls outside specified limits.

1.9 Port Output Enable (POE)

The Port Output Enable for the GPT (POEG) module can be used to place General PWM Timer (GPT) output pins in the output disable state in one of the following ways.

- Input level detection of the GTETRGn pins
- Output-disable request from the GPT
- Comparator interrupt request detection
- Oscillation stop detection of the clock generation circuit
- Register settings

This software provides the setting of certain pins (GTIOC0A and GTIOC0B) of GPT channel 0 into the high-impedance state when a rising edge on GTETRGn input pin is detected or when oscillation stop is detected in the above conditions.

The number of GTETR Gn input terminals (number of groups) of POEG varies depending on the device.

Table 1.15. GTETRGn Pin Symbol 'n' in Each MCU

MCU	GTETRGn Pin
RA6M1,RA6M2,RA6M3,RA6T1	n = A, B, C, D
RA2A1	n = A, B
RA2L1, RA2E1, RA2E2, RA2E3	n = A, B
RA4M1,RA4W1	n = A, B

The POEG group is user-selected. The user should set the I/O port to be used as GTETRGn in the POE.h file according to the system configuration. Also, specify the selected POEG group in the input parameter "group" of the `GPT_Init` function and `POE_Init` function.

In addition, the user must enable handling of interrupts generated by POEG. See Section 2.9 for more information on configuring the GTETRGn input pin and enabling POE interrupts.

In this software, only GPT channel 0 (GTIOC0A pin and GTIOC0B pin) is the target of output prohibition control of the General PWM Timer (GPT) by POEG.

Table 1.16. GPT Channel Targeted for Output Prohibition Operation by This Software

MCU	GPT Chanel	
RA2L1,RA2E1,RA2E2, RA2E3, RA2A1, RA4M1,RA4W1	GPT320	General PWMTimer 0 (32-bit)
RA6M1,RA6M2,RA6M3, RA6T1	GPT32EH0	General PWMTimer 0 (32-bit Enhanced High Resolution)

The test module relies on the renesas.h header file to access to peripheral registers.

Table 1.17. Port Output Enable Source Files

File Name
POE.h GPT.h
POE.c GPT.c

■ GPT.c File

Syntax	
`void GPT_init(POE_group_t group)`	
Description	
This function configures GPT channel 0 output prohibition control in association with the POEG group specified by the input parameter ‘group’. Note: Describe the settings related to the PWM waveform output in the user software. (Example: Assignment of GTIOC0A pin and GTIOC0B pin to I/O port, selection of GTIOCA pin/GTIOCB pin disable value setting by General PWM Timer I/O Control Register (GTIOR), etc.)	
Input Parameters	
`POE_group_t group`	POE group to associate the GPT channel 0.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

■ POE.c File

Syntax	
`void POE_Init(POE_CALL_BACK Callback, POE_group_t group)`	
Description	
This function configures the POEG group specified by the input parameter ‘group’. In the following cases, POEG puts the GTIOC0A and GTIOC0B pins of GPT channel 0 into a high impedance state. 1. When the rising edge on the GTETRGn input pin is detected. (At the same time, an interrupt occurs.) 2. When oscillation stop is detected. (At the same time, NMI occurs.)	
Input Parameters	
`POE_CALL_BACK Callback`	Function to call if a rising edge on the GTETRGn input pin is detected.
`POE_group_t group`	POEG group composed of this function.
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void POE_ClearFlags_ga(void)`	
Description	
This function clears the Port Input Detection Flag (PIDF), the Detection Flag for GPT or ACMPHS Output-Disable Request (IOCF), the Oscillation Stop Detection Flag (OSTPF), and the Software stop flag(SSF) in the POEG Group A Setting Register (POEGGA). This will release the pins from the high-impedance state.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void POE_ClearFlags_gb(void)`	
Description	
This function clears the Port Input Detection Flag (PIDF), the Detection Flag for GPT or ACMPHS Output-Disable Request (IOCF), the Oscillation Stop Detection Flag (OSTPF), and the Software stop flag(SSF) in the POEG Group B Setting Register (POEGGB). This will release the pins from the high-impedance state.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`void POE_ClearFlags_gc(void)`	
Description	
This function clears the Port Input Detection Flag (PIDF), the Detection Flag for GPT or ACMPHS Output-Disable Request (IOCF), the Oscillation Stop Detection Flag (OSTPF), and the Software stop flag(SSF) in the POEG Group C Setting Register (POEGGC). This will release the pins from the high-impedance state.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void POE_ClearFlags_gd(void)`	
Description	
This function clears the Port Input Detection Flag (PIDF), the Detection Flag for GPT or ACMPHS Output-Disable Request (IOCF), the Oscillation Stop Detection Flag (OSTPF), and the Software stop flag(SSF) in the POEG Group D Setting Register (POEGGD). This will release the pins from the high-impedance state.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`extern void poeg_group_event0_isr(void)`	
Description	
An interrupt handler for an output disable request for POEG group A. Call the callback function registered by the `POE_Init` function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`extern void poeg_group_event1_isr(void)`	
Description	
An interrupt handler for an output disable request for POEG group B. Call the callback function registered by the `POE_Init` function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`extern void poeg_group_event2_isr(void)`	
Description	
An interrupt handler for an output disable request for POEG group C. Call the callback function registered by the `POE_Init` function.	
Input Parameters	
`NONE`	N/A
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

Syntax	
`extern void poeg_group_event3_isr(void)`	
Description	
An interrupt handler for an output disable request for POEG group D. Call the callback function registered by the `POE_Init` function.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

1.10 GPIO

The GPIO readback level detection function is a function to read the digital output level of a pin when the port is in output mode. This makes it possible to diagnose defects in the terminals.

The GPIO readback level detection function checks the pins in the following procedure.

1. In port control register 1 (PCNTR1), set the PDRn bit to 1 to set it as an output port.
2. In the port control register 3 (PCNTR3), setting the POSRn bit to 1 outputs High, and setting the PORRn bit to 1 outputs Low.
3. In port control register 2 (PCNTR2), read the state of the each pin with the PIDR bit.

The port to be tested is specified in the gpio_config.h header file.

Table 1.18. GPIO Source File

File Name
gpio.h gpio_config.h
gpio.c

Syntax	
`void GPIO_Start(const GPIO_ERROR_CALL_BACK Callback)`	
Description	
This function uses the GPIO readback level detection function to read the digital output level of the pin when the port is in output mode and diagnose the pin failure. The port to be tested is specified in the gpio_config.h header file.	
Input Parameters	
`const GPIO_ERROR_CALL_BACK Callback`	A function to be called when a pin defect is detected (the read value of the port is different from the expected value)
Output Parameters	
`NONE`	N/A
Return Values	
`NONE`	N/A

2. Example Usage

This section gives to the user some useful suggestions about how to apply the released software.

Self testing can be divided into two patterns:

(a) Power-on Test

These are tests run once following a reset. They should be run as soon as possible but especially if start-up time is important it may be permissible to run some initialization code before running all the tests so that for example a faster main clock can be selected.

(b) Periodic Test

These are tests that are run regularly throughout normal program operation. This document does not provide a judgment of how often a particular test should be ran. How the scheduling of the periodic tests is performed is up to the user depending upon how their application is structured.

The following sections provide an example of how each test type should be used.

2.1 CPU

If a fault is detected by any of the CPU tests then a user supplied function called `CPU_Test_ErrorHandler` will be called. As any error in the CPU is very serious the aim of this function should be to get to a safe state, where software execution is not relied upon, as soon as possible.

2.1.1 Power-On

All the CPU tests should be run as soon as possible following a reset.

Note: The function must be called before the device is put in Unprivileged mode.

The function `CPU_Test_All` can be used to automatically run all the CPU tests.

2.1.2 Periodic

To test the CPU periodically, the function `CPU_Test_All` can be used, as it is for the power-on tests, to automatically run all CPU tests. Alternatively, to reduce the amount of testing done in a single function call, the user can choose to call each of the individual CPU test functions in turn each time the CPU periodic test is scheduled.

2.2 ROM

The ROM is tested by calculating a CRC value (CRC32) of its contents and comparing with a reference CRC value that must be added to a specific location in the ROM not included in the CRC calculation.

The CRC module must be initialized before use with a call to the `CRC_Init` function.

Ensure that all ROM sections used are included in the CRC calculation that both IAR and the CRC Test code use so that the results will match.

2.2.1 Reference CRC Value Calculation in Advance

(1) When Using GNU/e² studio

Since the GNU tool does not have a CRC calculation function, use the SRecord tool (*1) introduced below to calculate the reference CRC value. The user uses this tool to write the CRC value for reference in ROM in advance, and compares it with this value in the self-test.

*1: SRecord is an open source project on SourceForge. See below for details.

- SRecord Web Site (SRecord v1.64)
  http://srecord.sourceforge.net/
- CRC Checksum Generation with “SRecord” Tools for GNU and Eclips
  https://gcc-renesas.com/wiki/index.php?title=CRC_Checksum_Generation_with_%E2%80%98SRecord%E2%80%99_Tools_for_GNU_and_Eclipse

After unzipping the downloaded ZIP file, the following folders will be expanded.

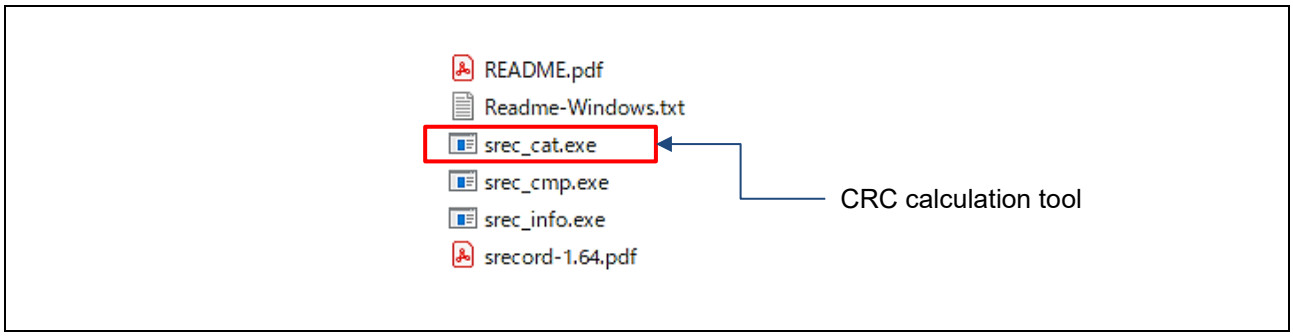

Figure 2.1 SRecord Tool Contents

An example of the folder structure of the project and SRecord tool is shown below.

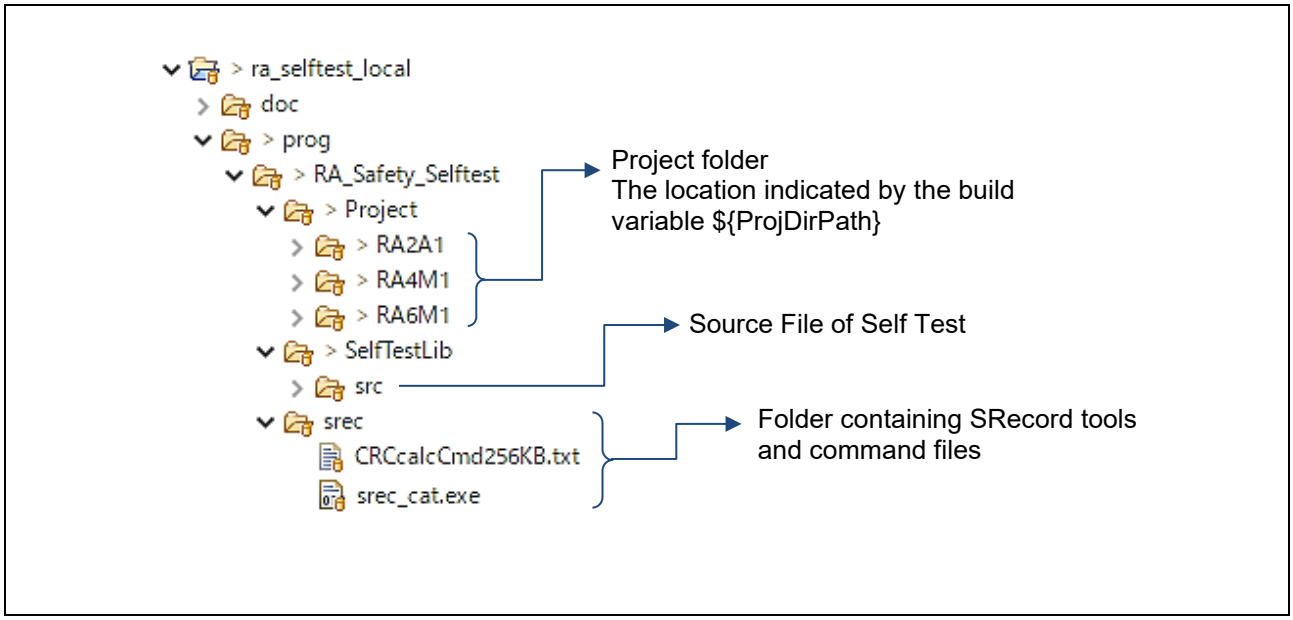

Figure 2.2 Folder Configuration Example

Open "project" -> "property" of e^2 studio, and use the "objcopy" command in the step after building to generate an S record file from the .elf file. Here, the converted file name is Original.srec. This file is the input for the SRecord tool.

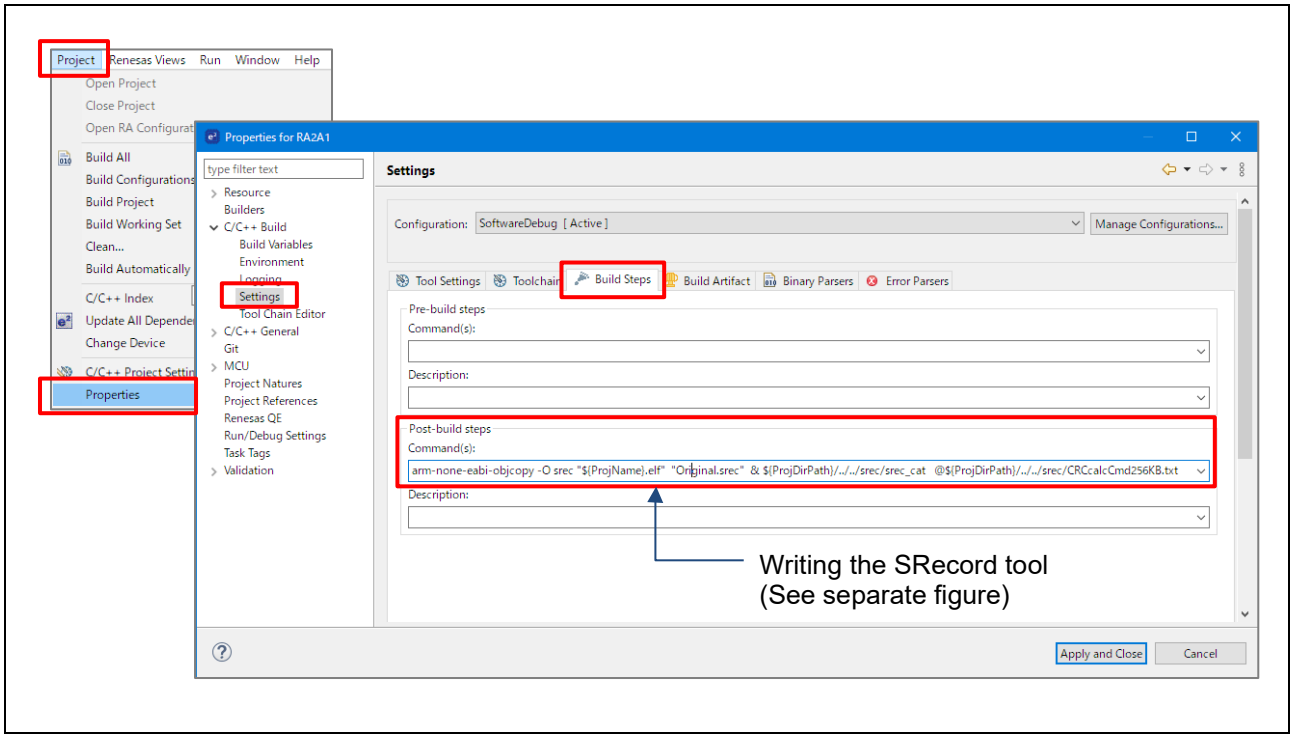

Figure 2.3 Output SRecord File and Start SRecord Tool

In the Build Steps tab in Figure 3, describe as follows.

■ Example of Command(s): entry (write on one line without line breaks)

```
arm-none-eabi-objcopy -O srec "${ProjName}.elf" "Original.srec" &
${ProjDirPath}/../../srec/srec_cat @${ProjDirPath}/../../srec/CRCcalcCmd256KB.txt
```

The part before "&" on the first line indicates the generation of the S record file, and the description of "srec_cat @command file" indicates the start of the srec_cat tool.

An example of the description of "CRCcalcCmd256KB.txt" as a command file is shown below.

■ Contents of CRCcalcCmd256KB.txt file (example)

```
Original.srec                 # Read srec file
-fill 0xFF 0x00000 0x040000   # 256KB ROM fill by 0xFF
-crop 0x00000 0x03FFFC        # Keep CRC calculate area
-STM32-le 0x03FFFC            # Calculate and output CRC value
-crop 0x03FFFC 0x040000       # Keep CRC area
Original.srec                 # Read srec file again
-fill 0xFF 0x000000 0x03FFFC  # fill 0xFF 0x0000 0x0FFFC
-Output addcrc.srec           # Output of S-record file including CRC value
```

If the ROM capacity varies depending on the device, change the address setting according to the device.

Also, when debugging, some ROMs rewrite the contents of ROM due to a software break. In that case, it is necessary to set the operation target area to something other than the debug area.

With the above operation, addcrc.srec (S record file with CRC calculation result added to the end of program code) can be created in the build configuration folder under the project folder, so download it to the target board.

Right-click on the top of the project tree and select "Debug as" → "Debug Configuration".

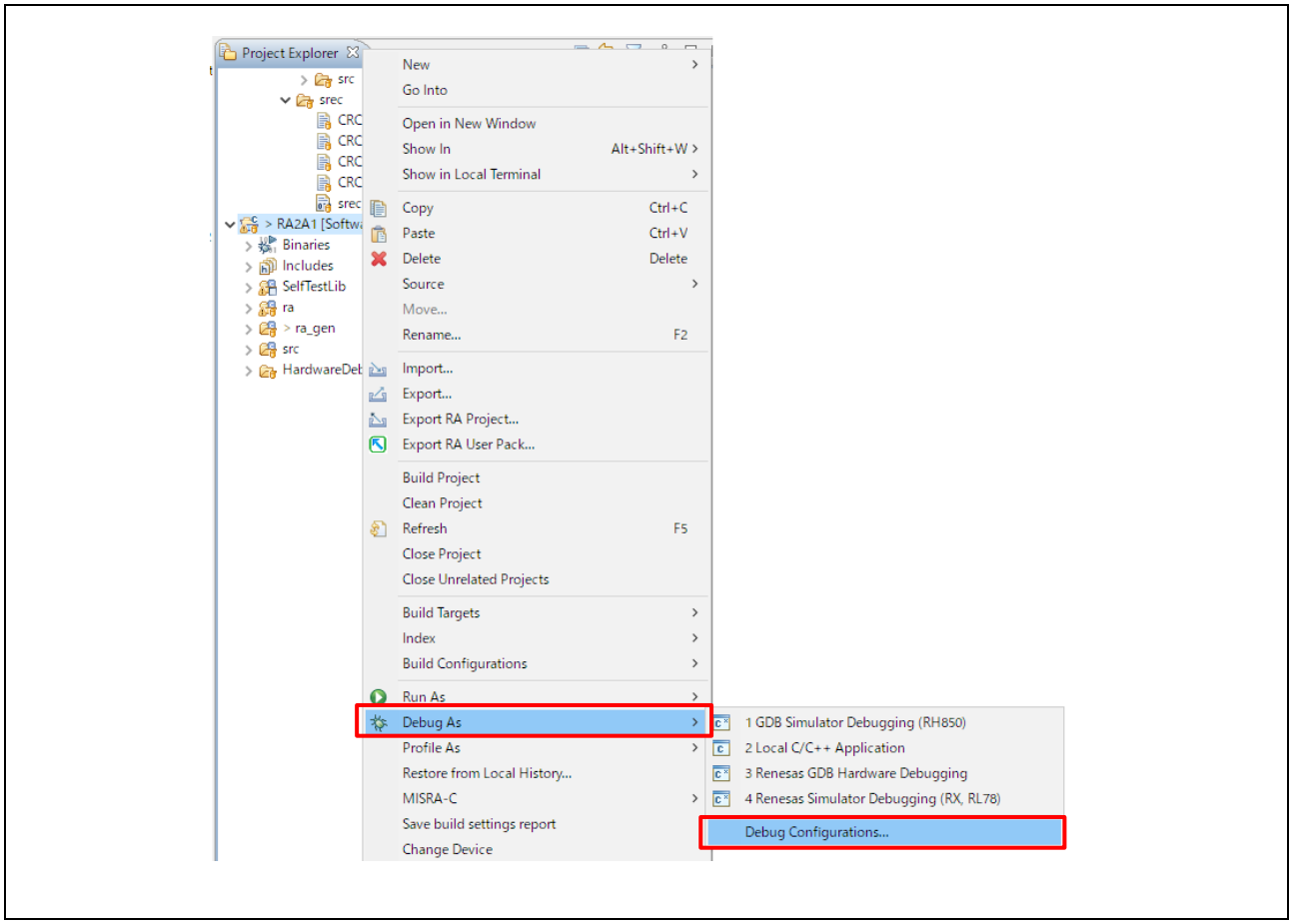

Figure 2.4 Select Debug Configuration of the Project

When the debug configuration dialog is displayed, select the "Startup" tab and select the build configuration to use. Only the symbol information is read from the ELF file, and the program image including the CRC calculation value is set to be read from addcrc.srec.

Click the "Debug" button to download the CRC calculation value to the target.

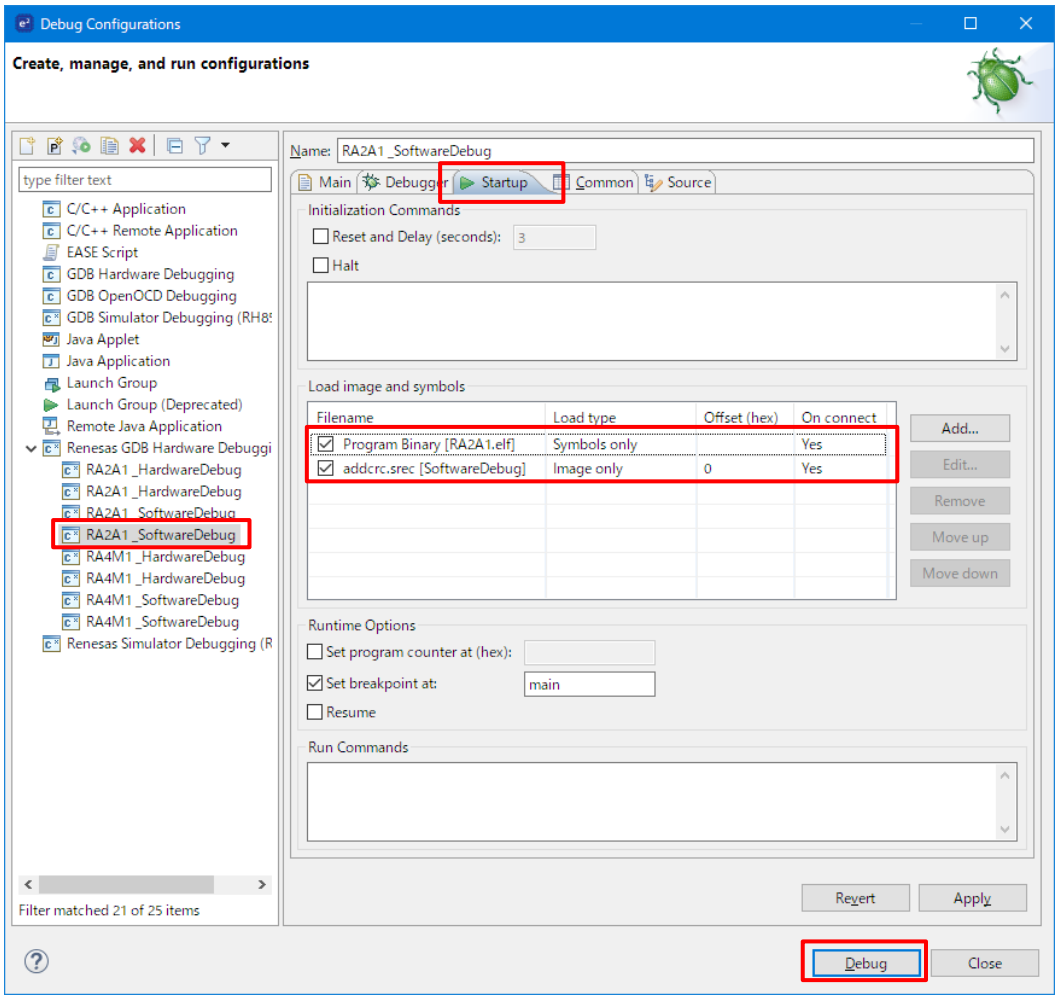

Figure 2.5 Load Image and Symbol Setting Example

(2) When Using IAR/EWARM

The IAR for ARM Toolchain can be used to calculate and add a CRC value to the built file at a location specified by the user. Select Project → Options, then select the Checksum tab in the Linker option and set the CRC generation settings.

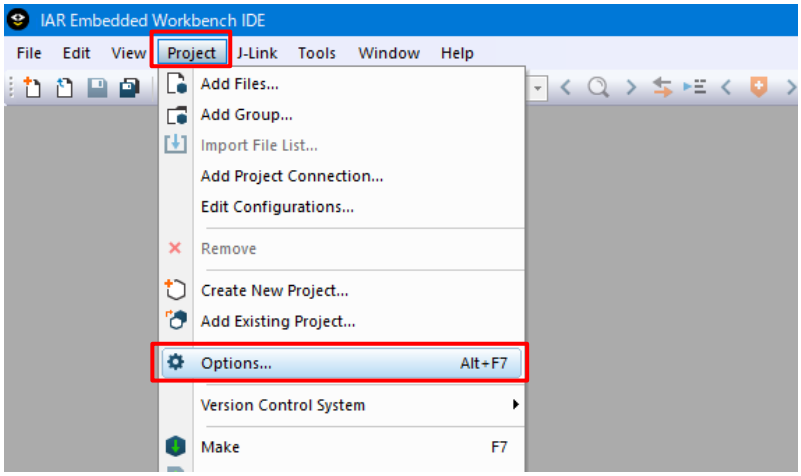

Figure 2.6 Select Project → Options...

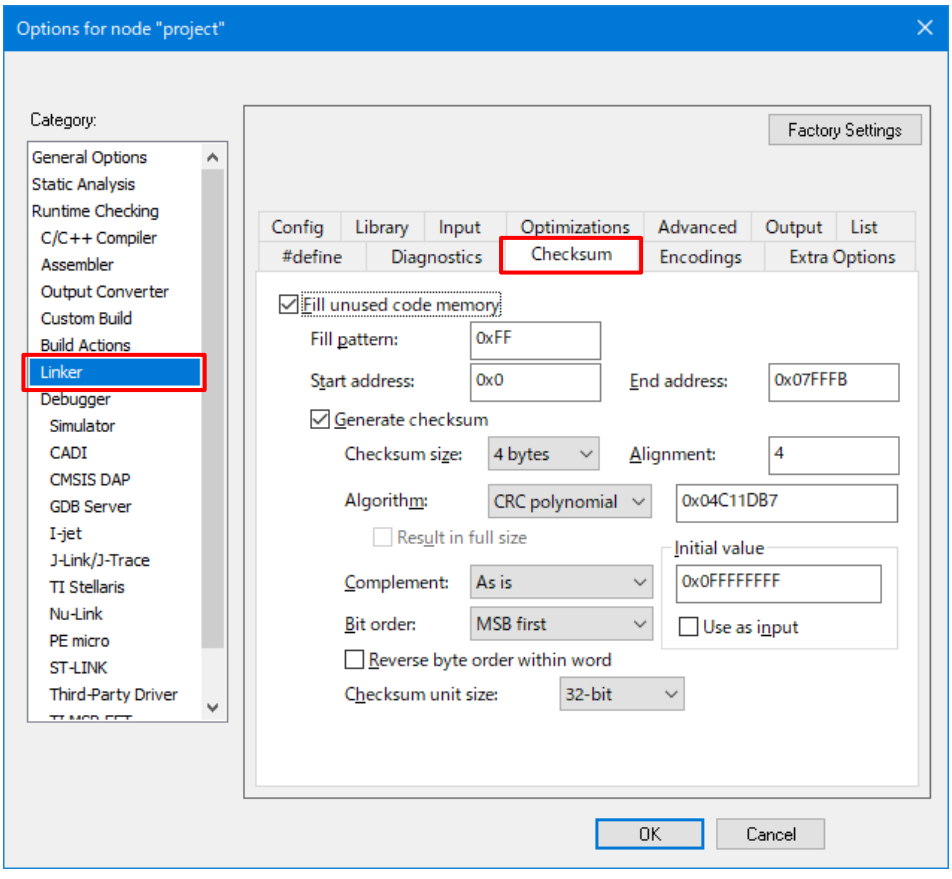

Figure 2.7 Adding Reference CRC Example (Set the Parameters According to the MCU)

2.2.2 Power-On

All the ROM memory used must be tested at power-on.

If this area is one contiguous block then function `CRC_Calculate` can be used to calculate and return a calculated CRC value.

If the ROM used is not in one contiguous block then the following procedure must be used.

1. Call CRC_Start.
2. Call CRC_AddRange for each area of memory to be included in the CRC calculation.
3. Call CRC_Result to get the calculated CRC value.

The calculated CRC value can then be compared with the reference CRC value stored in the ROM using function `CRC_Verify`.

It is a user's responsibility to ensure that all ROM areas used by their project are included in the CRC calculations.

2.2.3 Periodic

It is suggested that the periodic testing of ROM is done using the `CRC_AddRange` method, even if the ROM is contiguous. This allows the CRC value to be calculated in sections so that no single function call takes too long. Follow the procedure as specified for the power-on tests and ensure that each address range is small enough that a call to `CRC_AddRange` does not take too long.

2.3 RAM

It is very important to realize that the area of RAM that needs to be tested may change dramatically depending upon your project's memory map.

If you are using the 'HW' versions of the RAM Tests (where the DOC and possibly DTC are used), then you must call function `RamTest_March_HW_Init` prior to running the test.

When testing RAM, keep the following points in mind:

1. RAM being tested cannot be used for anything else including the current stack.
2. Any non-destructive test requires a RAM buffer where memory contents can be safely copied to and restored from.
3. There are two stacks, Main and Process. Stack tests can test either or both of the two stacks. Any test of the stack requires a RAM buffer where the stack can be relocated to.

2.3.1 Power-On

At power-on, a full destructive test can be performed on the RAM other than the stack. The stack must be tested with a non-destructive test. However, if startup time is very important, it might be possible to fine tune this so that only the area of stack used before the power-on RAM test is performed using the slower non-destructive test and the rest of the stack tested with a destructive test.

2.3.2 Periodic

All periodic tests must be non-destructive. It is assumed that the periodic tests are called from an interrupt handler and therefore the device is in privileged mode.

2.4 Clock

The monitoring of the main clock is set up with a single function call to `ClockMonitor_Init`. There are two versions of this file depending on the choice between using an external or internal reference clock as decided by the following #define:

```
#define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK
```

For example:

```
#ifdef CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK
#define MAIN_CLOCK_FREQUENCY_HZ         (12000000) // 12 MHz
#define EXTERNAL_REF_CLOCK_FREQUENCY_HZ (15000)    // 15kHz

ClockMonitor_Init(MAIN, MAIN_CLOCK_FREQUENCY_HZ, EXTERNAL_REF_CLOCK_FREQUENCY_HZ,
eCLOCK_MONITOR_CACREF_A, CAC_Error_Detected_Loop);

#else

#define TARGET_CLOCK_FREQUENCY_HZ         (12000000) // 12 MHz
#define REFERENCE_CLOCK_FREQUENCY_HZ      (15000)    // 15kHz

ClockMonitor_Init(MAIN, IWDTCLK, TARGET_CLOCK_FREQUENCY_HZ,
REFERENCE_CLOCK_FREQUENCY_HZ, CAC_Error_Detected_Loop);
/*NOTE: The IWDTCLK clock must be enabled before starting the clock monitoring.*/

#endif
```

When using an external reference clock as the reference clock, the user can specify the CACREF pin to use with the input parameter of the `ClockMonitor_Init` function (in the above example, `eCLOCK_MONITOR_CACREF_A` is specified).

The relationship between the terminals and input parameters of each device of the RA MCU is shown below. The user decides which terminal to use according to the system configuration.

Table 2.1 CACREF Pin and Input Parameter (`CLOCK_MONITOR_CACREF_PIN ePin`)

MCU	Terminal (Port Number) That Can Be Specified for CACREF	Symbol of Input Parameter "ePin"
RA6M1,RA6M2, RA6M3,RA6T1	P402	eCLOCK_MONITOR_CACREF_A
	P600	eCLOCK_MONITOR_CACREF_B
	P708	eCLOCK_MONITOR_CACREF_C
RA2L1,RA2E1,RA2E2, RA2E3, RA4M1,RA4W1	P204	eCLOCK_MONITOR_CACREF_A
	P400	eCLOCK_MONITOR_CACREF_B
RA2A1	P302	eCLOCK_MONITOR_CACREF_A
	P407	eCLOCK_MONITOR_CACREF_B

The `ClockMonitor_Init` function can be called as soon as the main clock has been configured and the IWDT has been enabled. See Section 2.5 for enabling the IWDT.

The clock monitoring is then performed by hardware and so there is nothing that needs to be done by software during the periodic tests.

In order to enable interrupt generation by the CAC, both Interrupt Controller Unit (ICU) and Nested Vectored Interrupt Controller (NVIC) should be configured in order to handle it.

In the interrupt controller unit (ICU), set the event signal number corresponding to CAC frequency error interrupt and CAC overflow in the ICU event link setting register (IELSRn) (except RA2L1,RA2E1,RA2E2, RA2E3).

When using FSP (Flexible Software Package) with e^2 studio, the ICU configuration can be set in the "Interrupts" tab of the RA Configuration Editor.

In the case of IAR/EWARM, ICU configuration can be set in the "Interrupts" tab by using "RA Smart Configurator" for IAR/EWARM.

Table 2.2 Setting of IELSRn Register Related to CAC (RA2A1, RA4M1/RA4W1, RA6M1/RA6M2/RA6M3/RA6T1)

MCU	Event Name	IELSRn.IELS
RA6M1,RA6M2, RA6M3,RA6T1	CAC_FERRI	0x87
	CAC_OVFI	0x89
RA4M1,RA4W1	CAC_FERRI	0x47
	CAC_OVFI	0x49
RA2A1	CAC_FERRI	0x35
	CAC_OVFI	0x37

Table 2.3 Setting of IELSRn Register Related to CAC (RA2L1/RA2E1/RA2E2/RA2E3)

MCU	Event Name	IELSRn	IELS[4:0]
RA2L1,RA2E1, RA2E2,RA2E3	CAC_FERRI	group1 (IELSR1/9/17/25)	0x0B
		group5 (IELSR5/13/21/29)	
	CAC_OVFI	group3 (IELSR3/11/19/27)	0x08
		group7 (IELSR7/15/23/31)	

The nested vector interrupt controller (NVIC) is set by the `test_main` function in the RA_SelfTests.c file. Where `NVIC_SetPriority` and `NVIC_EnableIRQ` are CMSIS functions provided by FSP, and *`CAC_FREQUENCY_ERROR_IRQn`* and *`CAC_OVERFLOW_IRQn`* are IRQ numbers generated by the FSP.

```
// NVIC settings related to CAC

/* CAC frequency error  ISR priority */
NVIC_SetPriority(CAC_FREQUENCY_ERROR_IRQn,0);
/* CAC frequency error  ISR enable */
NVIC_EnableIRQ(CAC_FREQUENCY_ERROR_IRQn);

/* CAC overflow  ISR priority */
NVIC_SetPriority(CAC_OVERFLOW_IRQn,0);
/* CAC overflow  ISR enable */
NVIC_EnableIRQ(CAC_OVERFLOW_IRQn);
```

If oscillation stop is detected, an NMI interrupt is generated. User code must handle this NMI interrupt and check the `NMISR.OSTST` flag as shown in this example:

```
if(1 == R_ICU->NMISR_b.OSTST)
{
   Clock_Stop_Detection();

   /*Clear OSTST bit by writing 1 to NMICLR.OSTCLR bit*/
   R_ICU->NMICLR_b.OSTCLR = 1;
}
```

The `OSTDCR.OSTDF` status bit can then be read to determine the status of the main clock.

2.5 Independent Watchdog Timer (IWDT)

In order to configure the Independent Watchdog Timer, it is necessary to set the OFS0 register in Option-Setting Memory. For example, suppose the Option-Setting Memory is set as follows.

Table 2.4 OFS0 Register Setting Example (IWDT Related)

Item	OFS0 Register Setting (For Example)
IWDT Start Mode Select (IWDTSTRT)	1: Disable IWDT after a reset
IWDT Timeout Period Select (IWDTTOPS[1:0])	11b: 2048 cycles
IWDT-Dedicated Clock Frequency Division Ratio Select (IWDTCKS[3:0])	1111b: 1/128
IWDT Window End Position Select (IWDTRPES[1:0])	11b: 0% (no window end position setting)
IWDT Window Start Position Select (IWDTRPSS[1:0])	11b: 100% (no window start position setting)
IWDT Reset Interrupt Request Select (IWDTRSTIRQS)	0: Enable non-maskable interrupt request or interrupt request
IWDT Stop Control (IWDTSTPCTL)	1: Stop counting when in Sleep, Snooze, or Software Standby mode.

When using FSP (Flexible Software Package) with e^2 studio, the "Option-Setting Memory" settings can be done in the property of the "BSP" tab of the configuration.

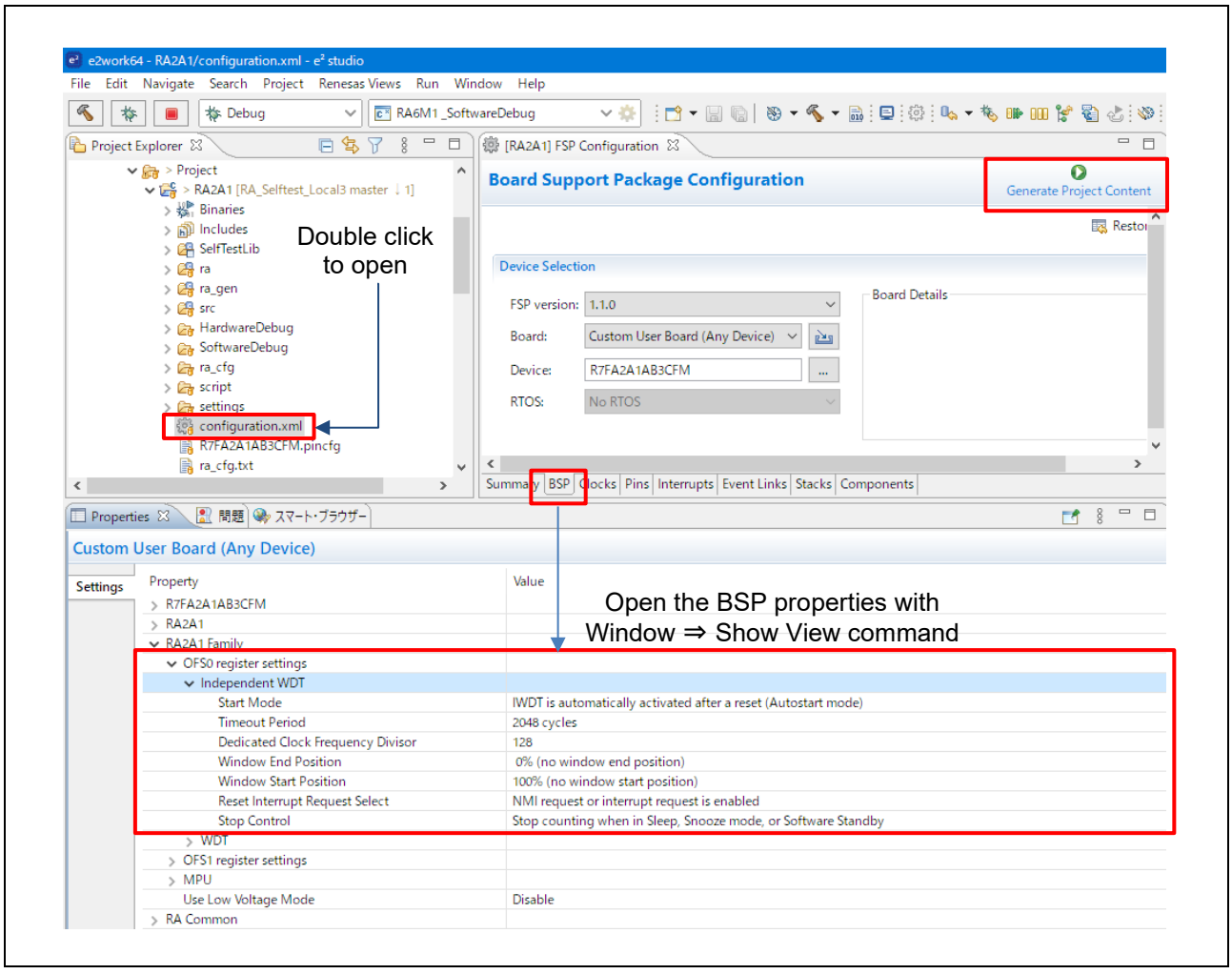

Figure 2.8. Example of OFS0 Register Setting by Using FSP with e^2 studio

In the case of IAR/EWARM, the "Option Setting Memory" can be set in the properties of the "BSP" tab using the "RA Smart Configurator" for IAR/EWARM.

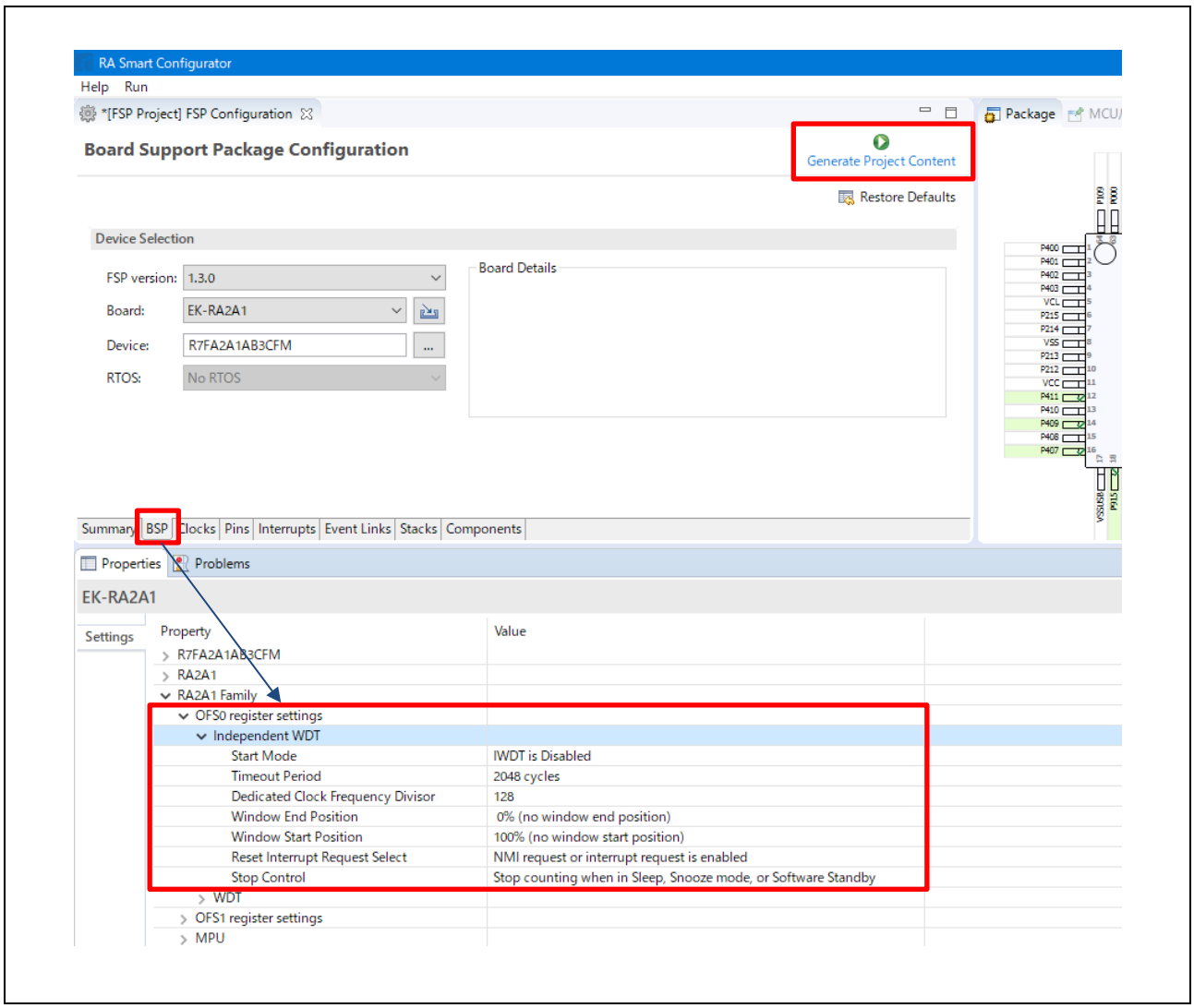

Figure 2.9 Example of OFS0 Register Setting by Using RA Smart Configurator for IAR/EWARM

When the "Generate Project Content" button is clicked, the contents set in the property will be reflected in the definition of the corresponding symbol in the following file . (For details, refer to "Renesas Flexible Software Package (FSP) User's Manual".)

- Applicable file
  ..*project-name*\ra_cfg\fsp_cfg\bsp**bsp_mcu_family_cfg.h**

- Applicable symbol (Excerpt)

```
#define OFS_SEQ1 0xA001A001 | (0 << 1) | (3 << 2)
#define OFS_SEQ2 (15 << 4) | (3 << 8) | (3 << 10)
#define OFS_SEQ3 (0 << 12) | (1 << 14) | (1 << 17)
```

: :

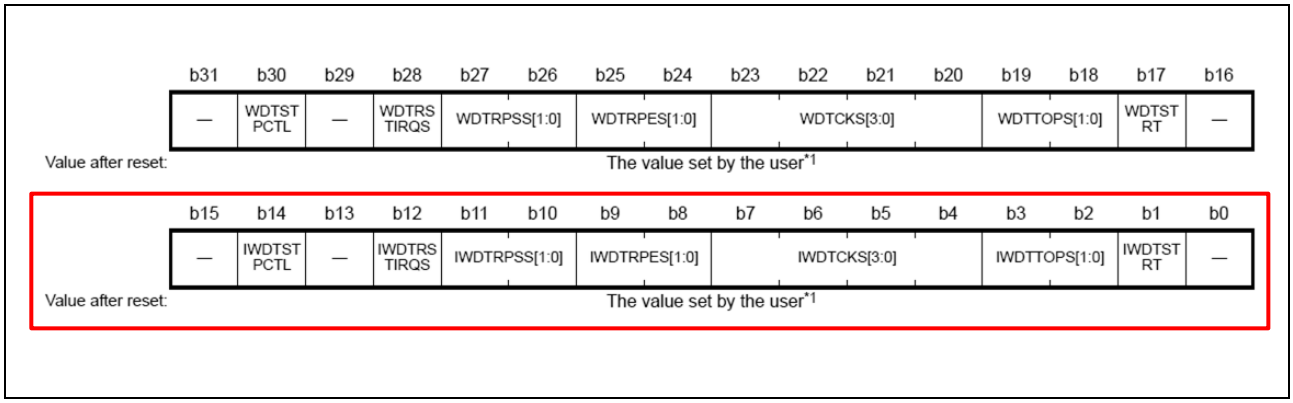

Figure 2.10 Option Function Select Register 0 (OFS0)

The Independent Watchdog Timer should be initialized as soon as possible following a reset with a call to `IWDT_Init`:

```
/*Setup the Independent WDT.*/
IWDT_Init();
```

After this, the watchdog timer must be refreshed regularly enough to prevent the watchdog timer timing out and performing a reset. Note, if using windowing the refresh must not just be regular enough but also timed to match the specified window. A watchdog timer refresh is called by calling this:

```
/*Regularly kick the watchdog to prevent it performing a reset. */
IWDT_Kick();
```

If the watchdog timer has been configured to generate an NMI on error detection then the user must handle the resulting interrupt.

If the watchdog timer has been configured to perform a reset on error detection then following a reset the code should check if the IWDT caused the reset by calling IWDT_DidReset:

```
if(TRUE == IWDT_DidReset())
{
   /*todo: Handle a watchdog reset.*/
   while(1){
      /*DO NOTHING*/
   }
}
```

2.6 Voltage

The Low Voltage Detection (LVD) module is configured to monitor the main supply voltage with a call to the `VoltageMonitor_Init` function. This should be set up as soon as possible following a power on reset.

Please note to set the LVD1SR.DET bit to 0 both before calling `VoltageMonitor_Init` function and in NMI routine, see section of Low Voltage Detection module of "RA MCU User's Manual: Hardware" for further details.

Please note to set a voltage threshold eVoltage lower than the Vcc nominal value.

The following example sets up the voltage monitor to generate an NMI if the voltage drops below 2.99 V.

```
VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL_2_99);
```

If a low voltage condition is detected, an NMI interrupt will be generated that the user must handle:

```
/*Low Voltage LVD1*/
if(1 == R_ICU->NMISR_b.LVD1ST)
{
   Voltage_Test_Failure();

   /*Clear LVD1ST bit by writing 1 to NMICLR.LVD1CLR bit*/
   R_ICU->NMICLR_b.LVD1CLR = 1;
}
```

2.7 ADC

The ADC module has a built in diagnostic mode which allows various reference voltages to be tested against. To account for allowed inaccuracies, the expected result is allowed to fall within a tolerance defined using:

```
#define ADC_TOLERANCE 8    // The value for ADC16
```

This value is set as the maximum absolute accuracy that the ADC is rated to. In a calibrated system this tolerance could be tightened.

The ADC test module must be initialized with a call to `Test_ADC_Init`.

Since ADC12 of RA6M1/RA6M2/RA6M3/RA6T1 has 2 units (unit 0 and unit 1), use the function with "_u1" in the name when testing unit 1.

2.7.1 Power-On

At power-on, the ADC module can be tested using the `Test_ADC_Wait` function. The return value of this function must be checked for the result.

2.7.2 Periodic

The periodic testing should start with a single call to `Test_ADC_Start`. Following that the ADC module will perform a reference conversion each time it is used. The reference voltage is rotated between 0 V, VREF/2 and VREF (For RA2A1, 0V, +VREF, -VREF).

The result of these reference conversions must be checked periodically using a call to `Test_ADC_CheckResult`.

2.8 Temperature

When testing the MCU temperature, it is important to remember that the ADC module will be used. Therefore if the user's code also uses the ADC to monitor analog pins it is important that resource sharing of the ADC module is carefully considered.

The temperature sensor must be initialized before use with a call to `Temperature_Init`. This function must be passed the allowable range of temperatures expressed in terms of the ADC output. See the "RA MCU User's Manual: Hardware" for details on how to calculate or find by experiment these values.

```
/*Temperature Sensor*/
Temperature_Init( TEMPERATURE_ADC_MIN,
                  TEMPERATURE_ADC_MAX,
                  Temperature_Test_Failure);
```

2.8.1 Power-On

Temperature test procedure at power-on will be the same as that explained for the periodic tests.

2.8.2 Periodic

Periodically the use of the ADC module must be taken over by the temperature sensor. To make a temperature reading, the user calls this function.

```
/*Start ADC reading temperature sensor output.*/
Temperature_Start();
```

The result can then be checked against the allowable range supplied in the Temperature_Init function with a call to:

```
/*The registered Error callback will be called if there is an error.*/
Temperature_CheckResult(TRUE);
```

To avoid the periodic test blocking the SW application for too long, it can be arranged so that each time the periodic test is scheduled it actually checks the result of the temperature test started on the previous scheduled test and then start a new conversion.

The user's code can use functions `Temperature_Is_Finished` or `Temperature_Wait_Finish` to determine when the application can resume using the ADC to read analog pins.

2.9 Port Output Enable (POE)

The GPT port output enable (POEG) is a function that disables the output pin of the General PWM Timer (GPT). See Section 1.9 for output prohibition conditions and the POEG groups that can be used with each MCU.

The POE initialization and start-up can be made using the following call.

```
/* Group A or B (A to D for RA6M1) is selected by the user */
POE_Init(POE_Event_Detected, GROUP_A);
```

Note that the POEG group choice is up to the user. The user must carefully study the description of `POE_Init` and consult the "RA MCU User's Manual: Hardware" to determine if the sample configuration of the POE meets the requirements of the user's system.

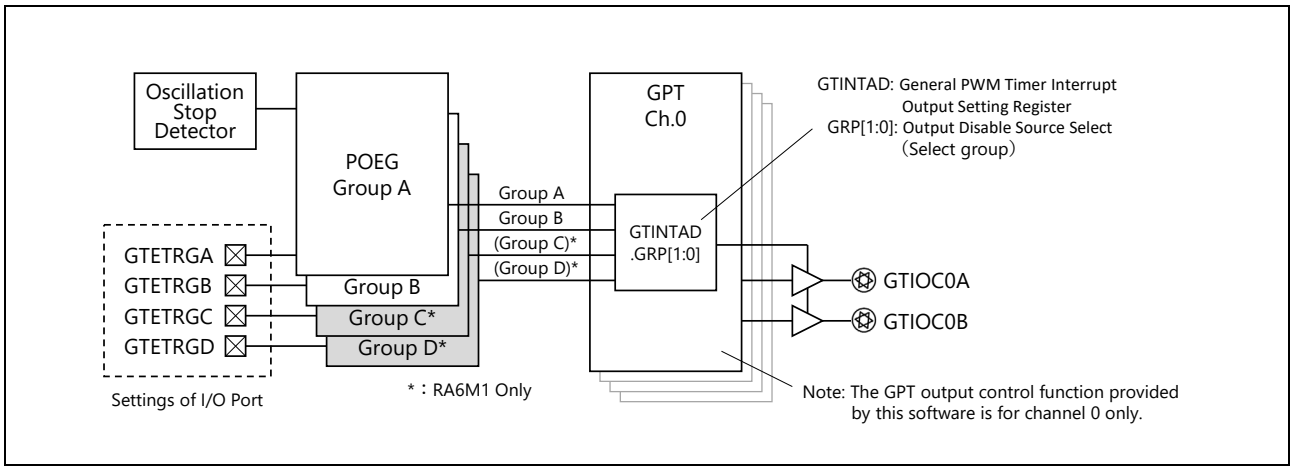

Figure 2.11 Outline of the Relationship Between POEG and GPT (General PWM Timer)

2.9.1 Settings of I/O Port

Depending upon the pins used in the user's system, the renesas.h header file may need to be adapted for the desired behavior. In addition, the pin must be set as the pin used as GTETRGn of the General PWM Timer (GPT).

POEG is an I/O port control register that controls output inhibition only when specified as the GPT pin by setting the PmnPFS.PMR and PmnPFS.PSEL[4:0] bits. If the pin is designated as a general purpose I/O pin, POEG does not perform output disable control.

- PmnPFS.PMR = 1 : Use pins as input/output ports for peripheral functions
- PmnPFS.PSEL[4:0] = 00010b : Specified as a GPT pin

Table 2.5. Settings of I/O Port (Pins That Can Be Specified as GTETRGn)

MCU	GTETRGn	Pins That Can be Specified as GTETRGn (Note)
RA2A1	GTETRG A	P400, P213
	GTETRG B	P109, P212
RA2L1,RA2E1, RA2E2, RA2E3	GTETRG A	P100, P105, P213, P401, P503
	GTETRG B	P101, P104, P212, P504
RA4M1,RA4W1	GTETRG A	P100, P105, P213
	GTETRG B	P101, P104, P212
RA6M1,RA6M2, RA6M3,RA6T1	GTETRG A	P100, P105, P401
	GTETRG B	P101, P104
	GTETRG C	P213, P503
	GTETRG D	P212, P504

Note: The target I/O port may not exist depending on the MCU package. Also, make sure that the pin is not used for any other purpose in the system.

2.9.2 Settings of Interrupt

In order to enable interrupt generation by the POE, both Interrupt Controller Unit (ICU) and Nested Vectored Interrupt Controller (NVIC) must be configured to handle it.

In the interrupt controller unit (ICU), set the event number corresponding to the POE group event in the ICU event link setting register (IELSRn) (except RA2L1,RA2E1,RA2E2, RA2E3).

When using FSP (Flexible Software Package) with e^2 studio, the ICU configuration can be set in the "Interrupts" tab of the RA Configuration Editor.

In the case of IAR/EWARM, ICU configuration can be set in the "Interrupts" tab by using "RA Smart Configurator" for IAR/EWARM.

Table 2.6 IELSRn Register Settings Related to POEG (RA2A1, RA4M1/RA4W1, RA6M1/RA6M2/RA6M3/RA6T1)

MCU	Event Name	IELSRn.IELS
RA2A1	POEG_GROUP0	0x041
	POEG_GROUP1	0x042
RA4M1,RA4W1	POEG_GROUP0	0x055
	POEG_GROUP1	0x056
RA6M1,RA6M2, RA6M3,RA6T1	POEG_GROUP0	0x09A
	POEG_GROUP1	0x09B
	POEG_GROUP2	0x09C
	POEG_GROUP3	0x09D

Table 2.7 IELSRn Register Settings Related to POEG (RA2L1/RA2E1/RA2E2/RA2E3)

MCU	Event Name	IELSRn	IELS[4:0]
RA2L1, RA2E1, RA2E2, RA2E3	POEG_GROUP0	group2 (IELSR2/10/18/26)	0x0B
		group6 (IELSR6/14/22/30)	
	POEG_GROUP1	group3 (IELSR3/11/19/27)	0x0B
		group7 (IELSR7/15/23/31)	

The nested vector interrupt controller (NVIC) is set by the `test_main` function in the RA_SelfTests.c file. Where `NVIC_SetPriority` and `NVIC_EnableIRQ` are CMSIS functions provided by FSP. *POEG0_EVENT_IRQn*, *POEG1_EVENT_IRQn*, *POEG2_EVENT_IRQn* and *POEG3_EVENT_IRQn* are FSP-generated IRQ numbers.

```
// NVIC setting example of interrupt related to POE group

/* POEG0 port output disable ISR priority */
NVIC_SetPriority(POEG0_EVENT_IRQn,0);
/* POEG0 port output disable ISR enable */
NVIC_EnableIRQ(POEG0_EVENT_IRQn);
/* POEG1 port output disable ISR priority */
NVIC_SetPriority(POEG1_EVENT_IRQn,0);
/* POEG1 port output disable ISR enable */
NVIC_EnableIRQ(POEG1_EVENT_IRQn);

#if defined(_MCU_RA6M1)
    /* POEG2 port output disable ISR priority */
    NVIC_SetPriority(POEG2_EVENT_IRQn,0);
    /* POEG2 port output disable ISR enable */
    NVIC_EnableIRQ(POEG2_EVENT_IRQn);
    /* POEG3 port output disable ISR priority */
    NVIC_SetPriority(POEG3_EVENT_IRQn,0);
    /* POEG3 port output disable ISR enable */
    NVIC_EnableIRQ(POEG3_EVENT_IRQn);
#endif
```

2.10 GPIO

The specification of the port to be tested in each MCU is defined by the g_GPIO_Test_Port structure in the gpio_config.h header file (port m, bit n).

```
// Port specification example
static const struct {
   uint16_t m;
   uint16_t n;
} g_GPIO_Test_Port[GPIO_TEST_NUM] =
{
#if defined(_MCU_RA2A1)
   { 0 ,  0 } ,    // PORT000
   { 1 ,  3 } ,    // PORT103
   { 2 ,  5 }      // PORT205
#elif defined(_MCU_RA4M1)
   { 0 ,  8 } ,    // PORT008
   { 1 ,  7 } ,    // PORT107
   { 2 ,  6 }      // PORT206
#else
   { 0 ,  8 } ,    // PORT008
   { 1 , 14 } ,    // PORT114
   { 2 ,  6 }      // PORT206
#endif
};
```

3. Benchmarking

3.1 RA4M1 Measurement Results

The environment in which the benchmark measurement was performed is shown below.

Table 3.1 Measurement Environment

Item	Contents
Device	R7FA4M1AB3CFP
Toolchain	IAR Embedded Workbench for ARM
Toolchain Version	v.8.50.124811
In-circuit debbuger	SEGGER J-link on board
Development board	EK-RA4M1 v1 (製品型名:RTK7EKA4M11S00001BU)

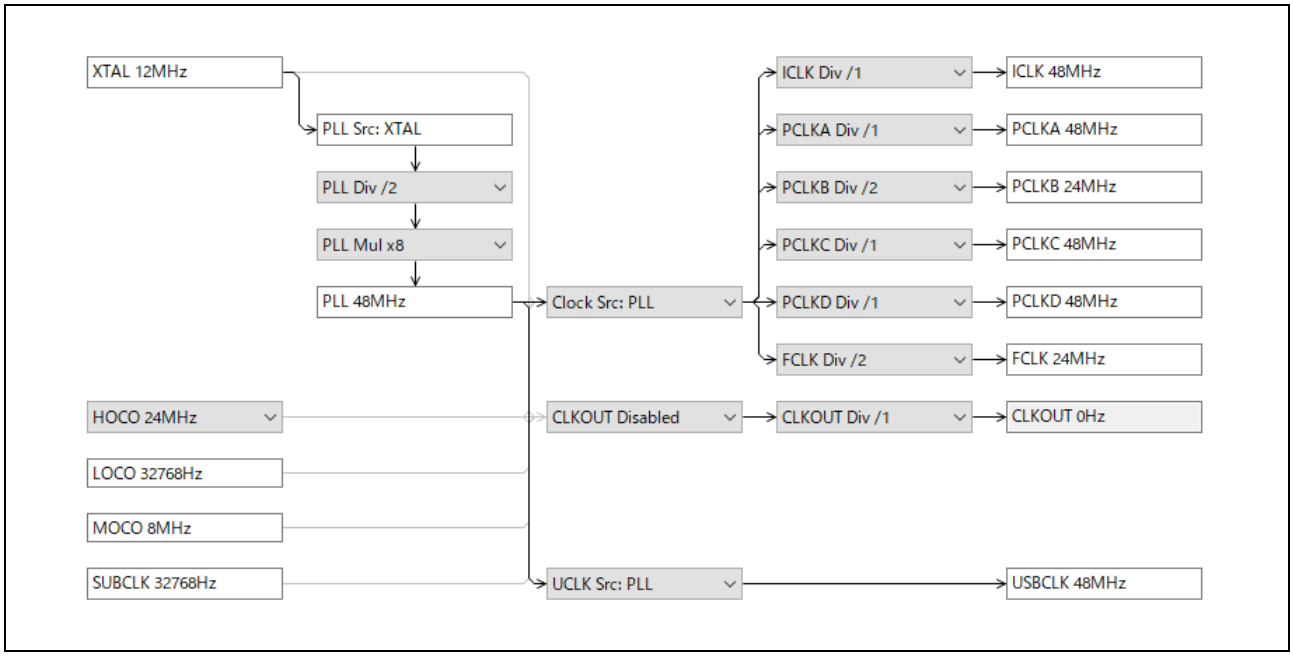

Figure 3.1 Clock Configuration

Table 3.2 IAR/EWARM Build Options

Category			Contents
General Options	Target	Device	Renesas R7FA4M1AB
C/C++ Complier	Language 1	Language	C
		C-dialect	Standard C
		Language Conformance	Standard with IAR extension
	Language 2	Plain ‘char’ is	Unsigned
		Floating-point semantics	Strict conformance
	Optimizations	Optimization Level	None

3.1.1 CPU

Table 3.3 CPU Test Results [ICLK: 48MHz]

Measurement		Result		Unit
		Non-Coupling Test (CPU only)	Coupling Test (CPU only)	
ROM Usage		2280	26174	bytes
RAM Usage		-	-	bytes
Stack Usage		-	-	bytes
CPU_Test_All	Clock Cycle Count	1075	10080	-
	Time Measured	22.40	210.00	µs

3.1.2 ROM

Table 3.4 Test Results for CRC32 [ICLK: 48MHz]

Measurement		Result	Unit
ROM Usage		114	bytes
RAM Usage		-	bytes
Stack Usage		-	bytes
CRC_Init CRC_Calculate (256KB ROM overall) CRC_Verify	Clock Cycle Count	783360	-
	Time Measured	16.32	ms

3.1.3 RAM

The measurement results of the tests performed with 8-bit and 32-bit access widths are shown below.

(1) March C

Table 3.5 March C Test Results (8-bit Access, 32-bit Word Limit) [ICLK: 48MHz]

Measurement			Result (Normal)	Result (HW)	Unit
ROM Usage			606	1182	bytes
RAM Usage			-	-	bytes
Stack Usage			84	84	bytes
Stack Usage Extra			116	116	bytes
Clock Cycle Count	Destructive	1024 bytes	1159680	1056000	-
	Non-destructive	1024 bytes	1178880	1063680	-
	Extra	1024 bytes	2342400	2119680	-
Time Measured	Destructive	1024 bytes	24.16	22.00	ms
	Non-destructiv	1024 bytes	24.56	22.16	ms
	Extra	1024 bytes	48.80	44.16	ms

Table 3.6 March C Test Results (32-bit Access, 32-bit Word Limit) [ICLK: 48MHz]

Measurement			Result (Normal)	Result (HW)	Unit
ROM Usage			636	1228	bytes
RAM Usage			-	-	bytes
Stack Usage			84	84	bytes
Stack Usage Extra			116	116	bytes
Clock Cycle Count	Destructive	1024 bytes	906240	879360	-
	Non-destructiv	1024 bytes	910080	883200	-
	Extra	1024 bytes	1820160	1766400	-
Time Measured	Destructive	1024 bytes	18.88	18.32	ms
	Non-destructiv	1024 bytes	18.96	18.40	ms
	Extra	1024 bytes	37.92	36.80	ms

(2) March X WOM

Table 3.7 March X WOM Test Results (8-bit Access, 32-bit Word Limit) [ICLK: 48MHz]

Measurement			Result (Normal)	Result (HW)	Unit
ROM Usage			462	1036	bytes
RAM Usage			-	-	bytes
Stack Usage			76	72	bytes
Stack Usage Extra			108	104	bytes
Clock Cycle Count	Destructive	1024 bytes	110592	94848	-
	Non-destructiv	1024 bytes	127104	99456	-
	Extra	1024 bytes	237120	193920	-
Time Measured	Destructive	1024 bytes	2.304	1.976	ms
	Non-destructiv	1024 bytes	2.648	2.072	ms
	Extra	1024 bytes	4.940	4.040	ms

Table 3.8 March X WOM test results (32-bit Access, 32-bit Word Limit) [ICLK: 48MHz]

Measurement			Result (Normal)	Result (HW)	Unit
ROM Usage			462	1036	bytes
RAM Usage			-	-	bytes
Stack Usage			76	72	bytes
Stack Usage Extra			108	104	bytes
Clock Cycle Count	Destructive	1024 bytes	24576	29568	-
	Non-destructiv	1024 bytes	28800	33984	-
	Extra	1024 bytes	53376	63360	-
Time Measured	Destructive	1024 bytes	0.512	0.616	ms
	Non-destructiv	1024 bytes	0.600	0.708	ms
	Extra	1024 bytes	1.112	1.320	ms

(3) Stack Test

Table 3.9 Stack Test Results [ICLK: 48MHz]

Measurement		Result (Normal)	Result (HW)	Unit
ROM Usage		344	344	bytes
RAM Usage		-	-	bytes
`RamTest_Stack_Main` (Only Stack Relocation)	Stack Usage	48	48	bytes
	Clock Cycle Count	126720	98880	-
	Time Measured	2.640	2.060	ms
`RamTest_Stacks` (Only Stack Relocation)	Stack Usage	64	64	bytes
	Clock Cycle Count	910080	887040	-
	Time Measured	18.960	18.480	ms

Website and Support

Visit the following URLs to learn about the key elements of the RA MCU, download tools, components, and related documentation, and get support.

- RA Product Information: www.renesas.com/ra
- RA FSP (Flexible Software Package): www.renesas.com/FSP
- RA Support Forum: www.renesas.com/ra/forum
- Renesas Support: www.renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Dec 18, 2020	–	First edition
1.01	May 31, 2024	2,5,39,47,61,2,68,69	Add applicable products

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

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

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

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

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

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

(Rev.4.0-1 November 2017)

Corporate Headquarters

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

www.renesas.com

Trademarks

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

Contact information

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