

## RA Family

R01AN8239EJ0100

Rev.1.00

Mar 26, 2026

## IEC 60730/60335 Self Test Library for RA0 MCU (CM23 Class-C)

### 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.

This Application Note provides guidelines of how to use flexible sample software routines to assist with compliance with IEC60730 class C 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.

The software routines provided are to be used after reset and also during the program execution. This document and the accompanying sample code provide an example of how to do this.

## Target Device

- Device:
  - Renesas RA Family (Arm® Cortex®-M23) \* See **Table a** for series and groups.
- Development environment:
  - GNU-GCC ARM Embedded Toolchain 13.2.1.arm-13-7 / Renesas e<sup>2</sup> studio 2025-10 (25.10.0)

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

**Table a. RA MCU Self-Test Function List**

CPU Core		Arm® Cortex®-M23
Series		RA0
Group		RA0E1, RA0E2, RA0L1
Test Function	CPU	○
	ROM	○
	RAM	○
	Clock	○
	Independent Watchdog Timer (IWDT)	○

**Self-test library overview**

The self-test library consists of instruction decoding, CPU registers, internal memory, watchdog timer, and monitoring functions for the system clock.

As described below, the anomaly monitoring process provides an application program interface (API) for each module that monitors. Use each function according to the purpose.

The self-test library functions are divided into modules according to IEC60730Class-C. The anomaly monitoring process can be performed standalone by selecting each test function in turn.

The RA0 series (with Arm® Cortex®-M23) self-test library implement functions of the following main self-testing.

- Instruction decoding  
Verify that the corresponding instruction of Arm Cortex-M23 works properly according to the specifications.  
See "IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H2.18.5 equivalence class test".
  
- CPU Register  
Test the CPU registers listed in "Table 1.1 CPU Test target (Overview) ".  
The internal data path is verified during the normal operation test of the above registers.  
See "IEC 60730-1:2013+A1:2015+A2:2020 Annex H - Table H.11.12.7 1.CPU".
  
- Invariable memory  
Test the internal Flash memory of the MCU.  
See "IEC 60730-1:2013+A1:2015+A2:2020 Annex H - H2.19.4.2 CRC – double word".
  
- Variable memory  
Test Internal SRAM  
The RAM test uses the Word-oriented Memory Test and the Extended March C-algorithm.  
See "IEC 60730-1:2013+A1:2015+A2:2020 Annex H - H.2.19.7 walkpat memory test".
  
- System Clock  
Test the operation and frequency of the system clock based on the reference clock source (this test requires an independent internal or external reference clock).  
See "IEC Reference - IEC 60730-1:2013+A1:2015+A2:2020 Annex H - H2.18.10.1 Frequency monitoring".
  
- CPU/Program Counter(PC)  
To confirm that the program is executing the sequence within the specified time, it is confirmed using the built-in watchdog timer that operates with a clock independent of the CPU.  
See "IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H2.18.10.3 independent time-slot and logical monitoring".

## Contents

1. Test .....	7
1.1 CPU .....	7
1.1.1 CPU Register Test and CPU Instruction Test .....	7
1.1.2 Test Error .....	20
1.1.3 CPU Software API .....	21
1.2 ROM .....	41
1.2.1 CRC32 Algorithm .....	41
1.2.2 Multi Checksum .....	41
1.2.3 CRC Software API .....	42
1.3 RAM .....	46
1.3.1 RAM Block Configuration .....	46
1.3.2 Reserved Area .....	47
1.3.3 RAM Test Algorithm .....	49
1.3.4 RAM Software API .....	52
1.4 Clock .....	56
1.4.1 Monitoring the system clock frequency using TAU0 ch5 (Input pulse interval measurement function) .....	56
1.4.2 Clock Test Software API .....	57
1.5 Independent Watchdog Timer (IWDT) .....	59
1.5.1 IWDT Software API .....	60
2. Example Usage .....	63
2.1 CPU .....	64
2.1.1 Power-On .....	64
2.1.2 Periodic .....	64
2.1.3 Preparation for CPU testing .....	64
2.2 ROM .....	66
2.2.1 Reference CRC Value Calculation in Advance .....	66
2.2.2 Setting for the support Multi-checksum .....	73
2.2.3 Power-On .....	74
2.2.4 Periodic .....	74
2.3 RAM .....	75
2.3.1 Power-On .....	75
2.3.2 Periodic .....	75
2.4 Clock .....	76
2.5 Independent Watchdog Timer (IWDT) .....	79
2.5.1 OFS0 Register Setting Example (IWDT related) .....	79
2.5.2 Example of registering and writing an NMI interrupt callback function .....	81
Website and Support .....	83

Reference Document .....83

Revision History .....84

## 1. Test

### 1.1 CPU

The objective of the CPU test is to detect random permanent faults from CPU core.

Main functions of CPU Test are described below.

- CPU instruction test
- CPU register test

#### 1.1.1 CPU Register Test and CPU Instruction Test

**Table 1.16** describes the outline of each test of the CPU test performed by this self-test library.

The related registers and instruction codes are tested by executing each test, and by checking the execution results, CPU fault can be detected.

Test target (Overview) are CPU instructions and registers listed in **Table 1.1**.

**Table 1.1 CPU Test target (Overview)**

Test target			Arm® Cortex®-M23(CM23)
Instruction	Profile		ARMv8-M Baseline
	Instruction set		Cortex-M23 Instruction Set
	DSP		N/A
	FSP		N/A
Register	General purpose registers	R0 – R12	✓
	Stack Pointer	SP(R13)	✓
	Link Register	LR(R14)	✓
	Program Counter	PC(R15)	✓
	Single-precision Floating-point Registers	S0 – S31	N/A
	Floating-point Status Control Register	FPSCR	N/A
	Application Program Status Register	APSR	✓

N/A: Not available

The list of the Armv8-M registers and their test support status is listed in the below "**Table 1.2 - Table 1.3**".

See the "Arm@v8-M Architecture Reference Manual" ([Reference Document \[2\]](#)) for detailed information on each register.

[Notation]

- ✓ : To be tested  
 (blank) : Not to be tested  
 N/A : Not applicable

**Table 1.2 Armv8-M Registers Tested/Not Tested by CPU Test (1 of 2)**

No.	Component	Register	Description	Tested by CPU test
1	Special and general-purpose registers	APSR	Application Program Status Register	✓
		BASEPRI	Base Priority Mask Register	N/A
		CONTROL	Control Register	
		EPSR	Execution Program Status Register	
		FAULTMASK	Fault Mask Register	N/A
		FPSCR	Floating-point Status and Control Register	N/A
		IPSR	Interrupt Program Status Register	
		LO_BRANCH_INFO	Loop and branch tracking information	N/A
		LR(R14)	Link Register	✓
		MSPLIM	Main Stack Pointer Limit Register	
		PC(R15)	Program Counter	✓
		PRIMASK	Exception Mask Register	
		PSPLIM	Process Stack Pointer Limit Register	
		Rn (R0 - R12)	General-Purpose Register n	✓
		SP (R13)	Current Stack Pointer Register	✓
		SP	Stack Pointer (Non-secure)	
		S0 – S31	Single-precision Floating-point Registers	✓
		VPR	Vector Predication Status and Control Register	N/A
XPSR	Combined Program Status Registers			

**Table 1.3 Armv8-M Registers Tested/Not Tested by CPU Test (2 of 2)**

No.	Component	Register	Tested by CPU test
2	Payloads	All registers	
3	Instrumentation Macrocell	All registers	
4	Data Watchpoint and Trace	All registers	
5	Flash Patch and Breakpoint	All registers	
6	Performance Monitoring Unit	All registers	N/A
7	Reliability, Availability and Serviceability Extension Fault Status Register (Registers starting at address 0xE0005000)	All registers	N/A
8	Implementation Control Block	All registers	
9	SysTick Timer	All registers	
10	Nested Vectored Interrupt Controller	All registers	
11	System Control Block	All registers	
12	Memory Protection Unit	All registers	
13	Security Attribution Unit	All registers	
14	Debug Control Block	All registers	
15	Software Interrupt Generation	All registers	
16	Reliability, Availability and Serviceability Extension Fault Status Register (Registers starting at address 0xE000EF04)	All registers	
17	Floating-Point Extension	All registers	
18	Cache Maintenance Operations	All registers	
19	Debug Identification Block	All registers	
20	Implementation Control Block (NS alias)	All registers	
21	SysTick Timer (NS alias)	All registers	
22	Nested Vectored Interrupt Controller (NS alias)	All registers	
23	System Control Block (NS alias)	All registers	
24	Memory Protection Unit (NS alias)	All registers	
25	Debug Control Block (NS alias)	All registers	
26	Software Interrupt Generation (NS alias)	All registers	
27	Reliability, Availability and Serviceability Extension Fault Status Register (NS Alias)	All registers	
28	Floating-Point Extension (NS alias)	All registers	
29	Cache Maintenance Operations (NS alias)	All registers	
30	Debug Identification Block (NS alias)	All registers	
31	Trace Port Interface Unit	All registers	

The list of the Armv8-M instructions and their test support status is listed in the below "Table 1.4 - Table 1.13"<sup>\*1</sup>.

See the "Arm® Cortex®-M23 Devices Generic User Guide " ([Reference Document \[1\]](#)) for detailed information on each instruction.

Note that the main purpose is not to test individual instructions, but to detect random permanent failure of the CPU core.

[Notation]

✓ : To be tested

(blank) : Not to be tested

N/A : Not applicable

\* : Not tested but fault is detected in conjunction with the other instruction (The mnemonic of target instruction is tested by the other instruction encoding (see "Arm®v8-M Architecture Reference Manual") and instruction encoding of the target instruction is tested by the other instruction)

Please note the primary aim is not to test individual instructions but to detect random permanent failure of the CPU core.

**Table 1.4** Armv8-M Instructions Tested/Not Tested by CPU Test (1 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
1	ADC (immediate)	N/A	21	BFC	N/A
2	ADC (register)	✓	22	BFI	N/A
3	ADD (SP plus immediate)	✓	23	BIC (immediate)	N/A
4	ADD (SP plus register)	*	24	BIC (register)	✓
5	ADD (immediate)	*	25	BKPT	
6	ADD (immediate, to PC)	*	26	BL	✓
7	ADD (register)	✓	27	BLX, BLXNS	✓
8	ADR	✓	28	BTI	N/A
9	AND (immediate)	N/A	29	BX, BXNS	✓
10	AND (register)	✓	30	BXAUT	N/A
11	ASR (immediate)	N/A	31	CBNZ, CBZ	✓
12	ASR (register)	N/A	32	CDP, CDP2	N/A
13	ASRL (immediate)	N/A	33	CINC	N/A
14	ASRL (register)	N/A	34	CINV	N/A
15	ASRS (immediate)	*	35	CLREX	✓
16	ASRS (register)	✓	36	CLRM	N/A
17	AUT	N/A	37	CLZ	N/A
18	AUTG	N/A	38	CMN (immediate)	N/A
19	B	✓	39	CMN (register)	✓
20	BF, BFX, BFL, BFLX, BFCSEL	N/A	40	CMP (immediate)	*

**Table 1.5** Armv8-M Instructions Tested/Not Tested by CPU Test (2 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
41	CMP (register)	✓	71	LDAEXB	✓
42	CNEG	N/A	72	LDAEXH	✓
43	CPS		73	LDAH	✓
44	CSDB	N/A	74	LDC, LDC2 (immediate)	N/A
45	CSEL	N/A	75	LDC, LDC2 (literal)	N/A
46	CSET	N/A	76	LDM, LDMIA, LDMFD	✓
47	CSETM	N/A	77	LDMDB, LDMEA	N/A
48	CSINC	N/A	78	LDR (immediate)	✓
49	CSINV	N/A	79	LDR (literal)	*
50	CSNEG	N/A	80	LDR (register)	✓
51	CX1	N/A	81	LDRB (immediate)	✓
52	CX1D	N/A	82	LDRB (literal)	N/A
53	CX2	N/A	83	LDRB (register)	*
54	CX2D	N/A	84	LDRBT	N/A
55	CX3	N/A	85	LDRD (immediate)	N/A
56	CX3D	N/A	86	LDRD (literal)	N/A
57	DBG	N/A	87	LDREX	✓
58	DMB		88	LDREXB	✓
59	DSB		89	LDREXH	✓
60	EOR (immediate)	N/A	90	LDRH (immediate)	✓
61	EOR (register)	✓	91	LDRH (literal)	N/A
62	ESB	N/A	92	LDRH (register)	*
63	FLDMDBX, FLDMIAX	N/A	93	LDRHT	N/A
64	FSTMDBX, FSTMIAX	N/A	94	LDRSB (immediate)	N/A
65	ISB		95	LDRSB (literal)	N/A
66	IT	N/A	96	LDRSB (register)	✓
67	LCTP	N/A	97	LDRSBT	N/A
68	LDA	✓	98	LDRSH (immediate)	N/A
69	LDAB	✓	99	LDRSH (literal)	N/A
70	LDAEX	✓	100	LDRSH (register)	✓

**Table 1.6** Armv8-M Instructions Tested/Not Tested by CPU Test (3 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
101	LDRSHT	N/A	131	ORN (immediate)	N/A
102	LDRT	N/A	132	ORN (register)	N/A
103	LE, LETP	N/A	133	ORR (immediate)	N/A
104	LSL (immediate)	N/A	134	ORR (register)	✓
105	LSL (register)	N/A	135	PAC	N/A
106	LSLL (immediate)	N/A	136	PACBTI	N/A
107	LSLL (register)	N/A	137	PACG	N/A
108	LSLS (immediate)	*	138	PKHBT, PKHTB	N/A
109	LSLS (register)	✓	139	PLD (literal)	N/A
110	LSR (immediate)	N/A	140	PLD, PLDW (immediate)	N/A
111	LSR (register)	N/A	141	PLD, PLDW (register)	N/A
112	LSRL (immediate)	N/A	142	PLI (immediate, literal)	N/A
113	LSRS (immediate)	*	143	PLI (register)	N/A
114	LSRS (register)	✓	144	POP (multiple registers)	✓
115	MCR, MCR2	N/A	145	POP (single register)	N/A
116	MCRR, MCRR2	N/A	146	PSSBB	N/A
117	MLA	N/A	147	PUSH (multiple registers)	✓
118	MLS	N/A	148	PUSH (single register)	N/A
119	MOV (immediate)	✓	149	QADD	N/A
120	MOV (register)	*	150	QADD16	N/A
121	MOV, MOVS (register-shifted register)	*	151	QADD8	N/A
122	MOVT	✓	152	QASX	N/A
123	MRC, MRC2	N/A	153	QDADD	N/A
124	MRRC, MRRC2	N/A	154	QDSUB	N/A
125	MRS	✓	155	QSAX	N/A
126	MSR (register)	✓	156	QSUB	N/A
127	MUL	✓	157	QSUB16	N/A
128	MVN (immediate)	N/A	158	QSUB8	N/A
129	MVN (register)	✓	159	RBIT	N/A
130	NOP		160	REV	✓

**Table 1.7** Armv8-M Instructions Tested/Not Tested by CPU Test (4 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
161	REV16	✓	191	SMLALD, SMLALDX	N/A
162	REVSH	✓	192	SMLAWB, SMLAWT	N/A
163	ROR (immediate)	N/A	193	SMLSD, SMLSDX	N/A
164	ROR (register)	N/A	194	SMLSLD, SMLSLDX	N/A
165	RORS (immediate)	N/A	195	SMMLA, SMMLAR	N/A
166	RORS (register)	✓	196	SMMLS, SMMLSR	N/A
167	RRX	N/A	197	SMMUL, SMMULR	N/A
168	RRXS	N/A	198	SMUAD, SMUADX	N/A
169	RSB (immediate)	✓	199	SMULBB, SMULBT, SMULTB, SMULTT	N/A
170	RSB (register)	N/A	200	SMULL	N/A
171	SADD16	N/A	201	SMULWB, SMULWT	N/A
172	SADD8	N/A	202	SMUSD, SMUSDX	N/A
173	SASX	N/A	203	SQRSHR (register)	N/A
174	SBC (immediate)	N/A	204	SQRSHRL (register)	N/A
175	SBC (register)	✓	205	SQSHL (immediate)	N/A
176	SBFX	N/A	206	SQSHLL (immediate)	N/A
177	SDIV	✓	207	SRSR (immediate)	N/A
178	SEL	N/A	208	SRSRHL (immediate)	N/A
179	SEV		209	SSAT	N/A
180	SG		210	SSAT16	N/A
181	SHADD16	N/A	211	SSAX	N/A
182	SHADD8	N/A	212	SSBB	N/A
183	SHASX	N/A	213	SSUB16	N/A
184	SHSAX	N/A	214	SSUB8	N/A
185	SHSUB16	N/A	215	STC, STC2	N/A
186	SHSUB8	N/A	216	STL	✓
187	SMLABB, SMLABT, SMLATB, SMLATT	N/A	217	STLB	✓
188	SMLAD, SMLADX	N/A	218	STLEX	✓
189	SMLAL	N/A	219	STLEXB	✓
190	SMLALBB, SMLALBT, SMLALTB, SMLALTT	N/A	220	STLEXH	✓

**Table 1.8** Armv8-M Instructions Tested/Not Tested by CPU Test (5 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
221	STLH	✓	251	TEQ (register)	N/A
222	STM, STMIA, STMEA	✓	252	TST (immediate)	N/A
223	STMDB, STMFD	N/A	253	TST (register)	✓
224	STR (immediate)	✓	254	TT, TTT, TTA, TTAT	
225	STR (register)	*	255	UADD16	N/A
226	STRB (immediate)	✓	256	UADD8	N/A
227	STRB (register)	✓	257	UASX	N/A
228	STRBT	N/A	258	UBFX	N/A
229	STRD (immediate)	N/A	259	UDF	
230	STREX	✓	260	UDIV	✓
231	STREXB	✓	261	UHADD16	N/A
232	STREXH	✓	262	UHADD8	N/A
233	STRH (immediate)	✓	263	UHASX	N/A
234	STRH (register)	✓	264	UHSAX	N/A
235	STRHT	N/A	265	UHSUB16	N/A
236	STRT	N/A	266	UHSUB8	N/A
237	SUB (SP minus immediate)	✓	267	UMAAL	N/A
238	SUB (SP minus register)	N/A	268	UMLAL	N/A
239	SUB (immediate)	✓	269	UMULL	N/A
240	SUB (immediate, from PC)	N/A	270	UQADD16	N/A
241	SUB (register)	*	271	UQADD8	N/A
242	SVC		272	UQASX	N/A
243	SXTAB	N/A	273	UQRSHL (register)	N/A
244	SXTAB16	N/A	274	UQRSHLL (register)	N/A
245	SXTAH	N/A	275	UQSAX	N/A
246	SXTB	✓	276	UQSHL (immediate)	N/A
247	SXTB16	N/A	277	UQSHLL (immediate)	N/A
248	SXTH	✓	278	UQSUB16	N/A
249	TBB, TBH	N/A	279	UQSUB8	N/A
250	TEQ (immediate)	N/A	280	URSHR (immediate)	N/A

**Table 1.9** Armv8-M Instructions Tested/Not Tested by CPU Test (6 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
281	URSHRL (immediate)	N/A	301	VADC	N/A
282	USAD8	N/A	302	VADD (floating-point)	N/A
283	USADA8	N/A	303	VADD (vector)	N/A
284	USAT	N/A	304	VADD	N/A
285	USAT16	N/A	305	VADDLV	N/A
286	USAX	N/A	306	VADDV	N/A
287	USUB16	N/A	307	VAND (immediate)	N/A
288	USUB8	N/A	308	VAND	N/A
289	UXTAB	N/A	309	VBIC (immediate)	N/A
290	UXTAB16	N/A	310	VBIC (register)	N/A
291	UXTAH	N/A	311	URSHRL (immediate)	N/A
292	UXTB	✓	312	USAD8	N/A
293	UXTB16	N/A	313	USADA8	N/A
294	UXTH	✓	314	USAT	N/A
295	VABAV	N/A	315	USAT16	N/A
296	VABD (floating-point)	N/A	316	USAX	N/A
297	VABD	N/A	317	USUB16	N/A
298	VABS (floating-point)	N/A	318	USUB8	N/A
299	VABS (vector)	N/A	319	UXTAB	N/A
300	VABS	N/A	320	UXTAB16	N/A

**Table 1.10** Armv8-M Instructions Tested/Not Tested by CPU Test (7 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
321	UXTAH	N/A	346	VDUP	N/A
322	UXTB	✓	347	VEOR	N/A
323	UXTB16	N/A	348	VFMA (vector by scalar plus vector, floating-point)	N/A
324	UXTH	✓	349	VFMA	N/A
325	VABAV	N/A	350	VFMA, VFMS (floating-point)	N/A
326	VABD (floating-point)	N/A	351	VFMAS (vector by vector plus scalar, floating-point)	N/A
327	VABD	N/A	352	VFMS	N/A
328	VABS (floating-point)	N/A	353	VFNMA	N/A
329	VABS (vector)	N/A	354	VFNMS	N/A
330	VABS	N/A	355	VHADD	N/A
331	VADC	N/A	356	VHCADD	N/A
332	VADD (floating-point)	N/A	357	VHSUB	N/A
333	VADD (vector)	N/A	358	VIDUP, VIWDUP	N/A
334	VADD	N/A	359	VINS	N/A
335	VADDLV	N/A	360	VLD2	N/A
336	VADDV	N/A	361	VLD4	N/A
337	VAND (immediate)	N/A	362	VLDM	N/A
338	VAND	N/A	363	VLDR (System Register)	N/A
339	VBIC (immediate)	N/A	364	VLDR	N/A
340	VBIC (register)	N/A	365	VLDRB, VLDRH, VLDRW	N/A
341	VCX2 (vector)	N/A	366	VLDRB, VLDRH, VLDRW, VLDRD (vector)	N/A
342	VCX3	N/A	367	VLLDM	N/A
343	VCX3 (vector)	N/A	368	VLSTM	N/A
344	VDDUP, VDWDUP	N/A	369	VMAX, VMAXA	N/A
345	VDIV	N/A	370	VMAXNM	N/A

**Table 1.11** Armv8-M Instructions Tested/Not Tested by CPU Test (8 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
371	VMAXNM, VMAXNMA (floating-point)	N/A	386	VMLS	N/A
372	VMAXNMV, VMAXNMAV (floating-point)	N/A	387	VMLSDAV	N/A
373	VMAXV, VMAXAV	N/A	388	VMLSLDAV	N/A
374	VMIN, VMINA	N/A	389	VMOV (between general-purpose register and half-precision register)	N/A
375	VMINNM	N/A	390	VMOV (between general-purpose register and single-precision register)	N/A
376	VMINNM, VMINNMA (floating-point)	N/A	391	VMOV (between two general-purpose registers and a doubleword register)	N/A
377	VMINNMV, VMINNMAV (floating-point)	N/A	392	VMOV (between two general-purpose registers and two single-precision registers)	N/A
378	VMINV, VMINAV	N/A	393	VMOV (general-purpose register to vector lane)	N/A
379	VMLA (vector by scalar plus vector)	N/A	394	VMOV (half of doubleword register to single general-purpose register)	N/A
380	VMLA	N/A	395	VMOV (immediate) (vector)	N/A
381	VMLADAV	N/A	396	VMOV (immediate)	N/A
382	VMLALDAV	N/A	397	VMOV (register) (vector)	N/A
383	VMLALV	N/A	398	VMOV (register)	N/A
384	VMLAS (vector by vector plus scalar)	N/A	399	VMOV (single general-purpose register to half of doubleword register)	N/A
385	VMLAV	N/A	400	VMOV (two 32-bit vector lanes to two general-purpose registers)	N/A

**Table 1.12** Armv8-M Instructions Tested/Not Tested by CPU Test (9 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
401	VMOV (two general-purpose registers to two 32-bit vector lanes)	N/A	431	VPT	N/A
402	VMOV (vector lane to general-purpose register)	N/A	432	VPUSH	N/A
403	VMOVL	N/A	433	VQABS	N/A
404	VMOVN	N/A	434	VQADD	N/A
405	VMOVX	N/A	435	VQDMLADH, VQRDMLADH	N/A
406	VMRS	N/A	436	VQDMLAH, VQRDMLAH (vector by scalar plus vector)	N/A
407	VMSR	N/A	437	VQDMLASH, VQRDMLASH (vector by vector plus scalar)	N/A
408	VMUL (floating-point)	N/A	438	VQDMLSDH, VQRDMLSDH	N/A
409	VMUL (vector)	N/A	439	VQDMULH, VQRDMULH	N/A
410	VMUL	N/A	440	VQDMULL	N/A
411	VMULH, VRMULH	N/A	441	VQMOVN	N/A
412	VMULL (integer)	N/A	442	VQMOVUN	N/A
413	VMULL (polynomial)	N/A	443	VQNEG	N/A
414	VMVN (immediate)	N/A	444	VQRSHL	N/A
415	VMVN (register)	N/A	445	VQRSHRN	N/A
416	VNEG (floating-point)	N/A	446	VQRSHRUN	N/A
417	VNEG (vector)	N/A	447	VQSHL, VQSHLU	N/A
418	VNEG	N/A	448	VQSHRN	N/A
419	VNMLA	N/A	449	VQSHRUN	N/A
420	VNMLS	N/A	450	VQSUB	N/A
421	VNMUL	N/A	451	VREV16	N/A
422	VORN (immediate)	N/A	452	VREV32	N/A
423	VORN	N/A	453	VREV64	N/A
424	VORR (immediate)	N/A	454	VRHADD	N/A
425	VORR	N/A	455	VRINT (floating-point)	N/A
426	VPNOT	N/A	456	VRINTA	N/A
427	VPOP	N/A	457	VRINTM	N/A
428	VPSEL	N/A	458	VRINTN	N/A
429	VPST	N/A	459	VRINTP	N/A
430	VPT (floating-point)	N/A	460	VRINTR	N/A

**Table 1.13** Armv8-M Instructions Tested/Not Tested by CPU Test (10 of 10)

No.	Instruction	Tested by CPU test	No.	Instruction	Tested by CPU test
461	VRINTX	N/A	478	VSQRT	N/A
462	VRINTZ	N/A	479	VSRI	N/A
463	VRMLALDAVH	N/A	480	VST2	N/A
464	VRMLALVH	N/A	481	VST4	N/A
465	VRMLSLDAVH	N/A	482	VSTM	N/A
466	VRSHL	N/A	483	VSTR (System Register)	N/A
467	VRSHR	N/A	484	VSTR	N/A
468	VRSHRN	N/A	485	VSTRB, VSTRH, VSTRW	N/A
469	VSBC	N/A	486	VSTRB, VSTRH, VSTRW, VSTRD (vector)	N/A
470	VSCCLRM	N/A	487	VSUB (floating-point)	N/A
471	VSEL	N/A	488	VSUB (vector)	N/A
472	VSHL	N/A	489	VSUB	N/A
473	VSHLC	N/A	490	WFE	
474	VSHLL	N/A	491	WFI	
475	VSHR	N/A	492	WLS, DLS, WLSTP, DLSTP	N/A
476	VSHRN	N/A	493	YIELD	
477	VSLI	N/A			

### 1.1.2 Test Error

The CPU test will jump to this function if an error is detected.

This error handling function is the structure of closed loop and should 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.

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

### 1.1.3 CPU Software API

The software API source files related to CPU testing are shown in **Table 1.14**.

When the CPU Test API is executed, the related CPU registers and instructions codes are tested.

A CPU fault can be detected by checking the execution result output to the argument.

It needs to set the configuration of CPU tests before compiling your code. The CPU test configuration directive and each CPU test is shown in **Table 1.15** and **Table 1.16**.

For details, refer to “**2.1.3 Preparation for CPU testing**”.

**Table 1.14 Source files of CPU Software API**

File Name	Remarks
r_cpu_diag_config.h	Definition of CPU Test Directive.
cpu_test.c	CPU test implementation part
r_cpu_diag_0.asm r_cpu_diag_1.asm r_cpu_diag_2.asm r_cpu_diag_3.asm r_cpu_diag_4.asm r_cpu_diag_5.asm r_cpu_diag_6.asm r_cpu_diag_7_1.asm r_cpu_diag_7_2.asm r_cpu_diag_7_3.asm r_cpu_diag_8.asm	Definition of CPU Test core function.  Note: Please note that some tests consist of multiple files like r_cpu_diag_7_1.asm, r_cpu_diag_7_2.asm.
r_cpu_diag_0.h r_cpu_diag_1.h r_cpu_diag_2.h r_cpu_diag_3.h r_cpu_diag_4.h r_cpu_diag_5.h r_cpu_diag_6.h r_cpu_diag_7_1.h r_cpu_diag_7_2.h r_cpu_diag_7_3.h r_cpu_diag_8.h	Declaration of CPU Test core function.
r_cpu_diag.c	Definition of CPU Test API function.
r_cpu_diag.h	Declaration of CPU Test API function.
r_cpu_diag.inc	Definition of Assembler macro.

**Table 1.15 Directives for Software Configuration for CPU Test**

File Name	Description
BUILD_R_CPU_DIAG_0	When set to "1", the CPU test function: R_CPU_Diag0 is constructed.
BUILD_R_CPU_DIAG_1	When set to "1", the CPU test function: R_CPU_Diag1 is constructed.
BUILD_R_CPU_DIAG_2	When set to "1", the CPU test function: R_CPU_Diag2 is constructed.
BUILD_R_CPU_DIAG_3	When set to "1", the CPU test function: R_CPU_Diag3 is constructed.
BUILD_R_CPU_DIAG_4	When set to "1", the CPU test function: R_CPU_Diag4 is constructed.
BUILD_R_CPU_DIAG_5	When set to "1", the CPU test function: R_CPU_Diag5 is constructed.
BUILD_R_CPU_DIAG_6	When set to "1", the CPU test function: R_CPU_Diag6 is constructed.
BUILD_R_CPU_DIAG_7_1 *1	When set to "1", the CPU test function: R_CPU_Diag7_1 is constructed.
BUILD_R_CPU_DIAG_7_2 *1	When set to "1", the CPU test function: R_CPU_Diag7_2 is constructed.
BUILD_R_CPU_DIAG_7_3 *1	When set to "1", the CPU test function: R_CPU_Diag7_3 is constructed.
BUILD_R_CPU_DIAG_8	When set to "1", the CPU test function: R_CPU_Diag8 is constructed.

Notes: 1. See **Table 1.16**

Please note that some tests have multiple directives like BUILD\_R\_CPU\_DIAG\_7\_1, BUILD\_R\_CPU\_DIAG\_7\_2.

**Table 1.16 CPU Test Target**

Test No	index *1	Function name *2	Objective of the Test
0	0	R_CPU_Diag0	Four basic arithmetic operations (add, sub, mul and div)
1	1	R_CPU_Diag1	Sign/Zero extension operations
2	2	R_CPU_Diag2	Branch, logical, comparison and conditional operations
3	3	R_CPU_Diag3	Bit manipulation and data transfer operations
4	4	R_CPU_Diag4	Memory access (Load/Store) without exclusive operations
5	5	R_CPU_Diag5	Memory access (Load/Store) with exclusive and privileged operations
6	6	R_CPU_Diag6	System related operations
7	7	R_CPU_Diag7_1	Registers R0 - R12, MSP(R13), LR(R14), and APSR diagnostic operation
	8	R_CPU_Diag7_2	
	9	R_CPU_Diag7_3	
8	10	R_CPU_Diag8	CPU register test using WALKPAT algorithm

Notes: 1. Test is required for all indexes when the test spans over multiple indexes.

2. See **Table 1.15** for software configuration directives for code generation of each function.

## ■ cpu\_test.c File

Syntax	
void CPU_Test_ClassC(void)	
Description	
<p>Perform the CPU tests in the following order:</p> <ol style="list-style-type: none"> <li>1. Pass parameters and call function R_CPU_Diag</li> <li>2. Check the value of the argument "result"</li> <li>3. If the result is OK, return to step 1. above (perform the next test) When all the CPU tests are completed, go to step 4 If an error is detected, the external function CPU_Test_ErrorHandler will be called Check individual tests for more information.</li> <li>4. CPU_Test_PC</li> <li>5. Finish the function</li> </ol>	
Input Parameters	
NONE	N/A
Output Parameters	
const uint32_t forceFail	<p>Forced FAIL Option</p> <p>The default value is fixed at "1" (N/A).</p> <p>* If you want to test the forced FAIL, change the value to fixed at "0".</p>
Return Values	
NONE	N/A

<b>Syntax</b>	
void CPU_Test_PC(void)	
<b>Description</b>	
<p>This function tests the program counter (PC) register.</p> <p>Test PC operation by calling a separate function.</p> <p>Call the separate function (TestPC_TestFunction) that returns the inverted value of the specified parameter and check the return value.</p> <p>This confirms that PC is definitely operating.</p> <p>If an error is detected, the external function CPU_Test_ErrorHandler is called.</p>	
<b>Input Parameters</b>	
NONE	N/A
<b>Output Parameters</b>	
NONE	N/A
<b>Return Values</b>	
NONE	N/A

## ■ r\_cpu\_diag.c File

Syntax	
void R_CPU_Diag(uint32_t index, const uint32_t forceFail, int32_t *result)	
Description	
Use the index argument to execute the test function that corresponds to the CPU test number. See <b>Table 1.16</b> for the argument index, test number, and test function	
<ol style="list-style-type: none"> <li>Set "resultTemp" to the initial value. When the test function is performed, the test result is saved in "resultTemp".</li> <li>Check if the value of the argument "Index" is valid. If it is invalid, it exits the process after setting "FAIL(=0)" in the test result.</li> <li>Perform the function of the corresponding CPU test according to the value of the argument "index".</li> <li>Set the test result to "* result" and exit the function.</li> </ol>	
Input Parameters	
uint32_t index	CPU Test No (Refer to <b>Table 1.16</b> ) Returns FAIL when argument value is invalid.
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

<b>Syntax</b>	
static void norm_null(const uint32_t forceFail, int32_t *result)	
<b>Description</b>	
This function is a dummy function of the CPU test function excluded from compilation by the directive. Set the test result to PASS.	
<b>Input Parameters</b>	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
<b>Output Parameters</b>	
int32_t *result	Test result (1: PASS)
<b>Return Values</b>	
NONE	N/A

## ■ r\_cpu\_diag\_0.asm File

Syntax	
void R_CPU_Diag0(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. Addition instructions test</b> Execute each instruction of ADCS (register), ADDS (register) and check the match with the expected value of local signature and global signature.</p> <p><b>2. Subtraction instructions test</b> Execute each instruction of SBCS (register), SUBS (immediate), RSBS (immediate) and check the match with the expected value of local signature and global signature.</p> <p><b>3. Multiplication instructions test</b> Execute each instruction of MULS and check the match with the expected value of local signature and global signature.</p> <p><b>4. Division instructions test</b> Execute each instruction of SDIV, UDIV and check the match with the expected value of local signature and global signature.</p> <p><b>5. Addition and subtraction for stack pointer test</b> Execute each instruction of SUB (SP minus immediate), ADD (SP plus immediate) and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_1.asm File

Syntax	
void R_CPU_Diag1(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. Sign extension</b> Execute each instruction of SXTB T1, SXTH T1 and check the match with the expected value of local signature and global signature.</p> <p><b>2. Zero extension</b> Execute each instruction of UXTB T1, UXTH T1 and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_2.asm File

Syntax	
void R_CPU_Diag2(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. Branch</b> Execute each instruction of ADR T1, BEQ T1, B T2, BL T1, BLX T1, BX T1, CBZ T1 and check the match with the expected value of local signature and global signature.</p> <p><b>2. Logical test</b> Execute each instruction of TST T1 and check the match with the expected value of local signature and global signature.</p> <p><b>3. Logical operation</b> Execute each instruction of ANDS T1, ORRS T1, EORS T1, MVNS T1 and check the match with the expected value of local signature and global signature.</p> <p><b>4. Comparison</b> Execute each instruction of CMN T1, CMP T1 and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_3.asm File

Syntax	
void R_CPU_Diag3(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. Bit manipulation</b> Execute each instruction of ASRS (register) T1, BICS (register) T1, LSL (register) T1, LSR (register) T1, ROR (register) T1 and check the match with the expected value of local signature and global signature.</p> <p><b>2. Data manipulation</b> Execute each instruction of REV T1, REV16 T1, REVSH T1 and check the match with the expected value of local signature and global signature.</p> <p><b>3. Data transfer</b> Execute each instruction of MOVS (immediate) T1, MOVT T1, MRS T1, MSR (register) T1 and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0 : FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_4.asm File

Syntax	
void R_CPU_Diag4(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. LDR and STR</b> Execute each instruction of LDR (immediate) T2, STR (immediate) T2 and check the match with the expected value of local signature and global signature.</p> <p><b>2. LDRH and STRH</b> Execute each instruction of LDRH (immediate) T1, STRH (immediate) T1, LDRSH (register) T1, STRH (register) T1 and check the match with the expected value of local signature and global signature.</p> <p><b>3. LDRB and STRB</b> Execute each instruction of LDRSB (register) T1, STRB (register) T1, LDRB (immediate) T1, STRB (immediate) T1 and check the match with the expected value of local signature and global signature.</p> <p><b>4. LDM and STM</b> Execute each instruction of LDM and STM, LDM T3, STMDB T2 and check the match with the expected value of local signature and global signature.</p> <p><b>5. LDA and STL</b> Execute each instruction of LDA T1, STL T1, LDAH T1, STLH T1, LDAB T1, STLB T1 and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_5.asm File

Syntax	
void R_CPU_Diag5(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. LDAEX and STLEX</b> Execute each instruction of LDAEX T1, STLEX T1, LDAEXH T1, STLEXH T1, LDAEXB T1, STLEXB T1 and check the match with the expected value of local signature and global signature.</p> <p><b>2. LDREX and STREX</b> Execute each instruction of LDREX T1, STREX T1, LDREXH T1, STREXH T1, LDREXB T1, STREXB T1 and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_6.asm File

Syntax	
void R_CPU_Diag6(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. PUSH and POP</b></p> <p>After executing the PUSH instruction using R1, R2, R3, R4, R5, R6, execute the POP instruction and check the match with the expected value in each register of R1 and R4, R2 and R5, and R3 and R6.</p> <p><b>2. Other (miscellaneous) operations</b></p> <p>Execute each instruction of CLREX T1 and check the match with the expected value of local signature and global signature.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## ■ r\_cpu\_diag\_7\_1.asm File

Syntax	
void R_CPU_Diag7_1(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>1. Detecting “0” fixed fault for status and control registers</b> After writing "1" to the corresponding bit of the APSR register using R4 and R5, execute reading and check the match between each register of R4 and R5 and the expected value. (Confirm that it is not fixed to "0")</p> <p><b>2. Detecting “1” fixed fault for status and control registers</b> After writing "0" to the corresponding bit of the APSR register using R4 and R5, execute reading and confirm the match between each resist of R4 and R5 and the expected value. (Confirm that "1" is not fixed)</p> <p><b>3. Detecting “0” fixed fault for general purpose registers</b> After writing ALL "1" to R0 to R12 and LR (R14), execute reading and check that the registers of R0 to R12 and LR (R14) match the expected value. (Confirm that it is not fixed to "0")</p> <p><b>4. Detecting “1” fixed fault for general purpose registers</b> After writing ALL "0" to R0 to R12 and LR (R14), execute reading and check that the registers of R0 to R12 and LR (R14) match the expected value. (Confirm that "1" is not fixed)</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

■ r\_cpu\_diag\_7\_2.asm File

Syntax	
void R_CPU_Diag7_2(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>5. Detecting tests for the R0-R12 and R14 registers.</b></p> <ul style="list-style-type: none"> <li>– Nearest neighbor coupling( Test pattern : 0x55555555)</li> <li>– Next nearest neighbor coupling(Test pattern : 0x33333333)</li> <li>– 4-fold neighbor coupling(Test pattern : 0x0f0f0f0f)</li> <li>– 8-fold neighbor coupling(Test pattern : 0x00ff00ff)</li> <li>– 16-fold neighbor coupling(Test pattern : 0x0000ffff)</li> </ul> <p><b>The procedure is as follows.</b></p> <ol style="list-style-type: none"> <li>(1) Set each of the above test patterns to R0, write to R1, and check if it matches R0.</li> <li>(2) If they match, change the register written in 1 above in the order of R2 to R14 and perform.</li> <li>(3) Set each of the above test patterns to R14, write to R0, and confirm that it matches R0.</li> <li>(4) If they match, perform the following test pattern.</li> <li>(5) When all is completed, move to the following test.</li> </ol> <p><b>6. Detecting coupling fault for general purpose registers between any two registers</b></p> <ul style="list-style-type: none"> <li>– Detecting R7, R8, R9, R10, R11, R12, LR(R14) coupling fault (Using A's pattern)</li> <li>– Detecting R0, R1, R2, R3, R4, R5, R6 coupling fault (Using B's pattern)</li> </ul> <p><b>The procedure is as follows.</b></p> <ol style="list-style-type: none"> <li>(1) Set test patterns for R0 to R6, write R0 to R7, R1 to R8, ..., R6 to R14, and confirm each value of R0 and R7, R1 and R8, ..., R6 and R14 is matched.</li> <li>(2) Set test patterns for R7 to R14, write R8 to R0, R9 to R1, ..., R7 to R6, and confirm each value of R8 and R0, R9 and R1, ..., R7 and R6 is matched.</li> <li>(3) Complete the test.</li> </ol> <p>Note that R13 (SP) is excluded from this test.</p> <p>If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

■ r\_cpu\_diag\_7\_3.asm File

Syntax	
void R_CPU_Diag7_3(const uint32_t forceFail, int32_t *result)	
Description	
<p><b>7. Detecting "0" fixed fault for MSP(R13)</b> After writing "0xfffffc" to the SP (R13) register using R5, execute reading and confirm that R5 and SP (R13) match the expected value. (Confirm that not fixed to "0")</p> <p><b>8. Detecting "1" fixed fault for MSP(R13)</b> After writing "0x00000000" to the SP (R13) register using R5, execute reading and confirm that R5 and SP (R13) match the expected value. (Confirm that not fixed to "1")</p> <p><b>9. Detecting coupling fault for MSP(R13) between any two bits</b> Perform the following tests for R13(SP)            – Nearest neighbor coupling(Test pattern : 0x55555554)            – Next nearest neighbor coupling(Test pattern : 0x33333330)            – 4-fold neighbor coupling(Test pattern : 0x0f0f0f0c)            – 8-fold neighbor coupling(Test pattern : 0x00ff00fc)            – 16-fold neighbor coupling(Test pattern : 0x0000fffc)  <b>The procedure is as follows.</b>            (1) Set each of the above test patterns to R5, write to R13 (SP), and confirm that it matches R5.            (2) If they match, carry out the next test pattern.            (3) When all is completed, move to the following test</p> <p><b>10. Detecting coupling fault between MSP(R13) to other general purpose registers</b>            – Detecting SP, R2 coupling fault            – Detecting SP, R3 coupling fault  <b>The procedure is as follows.</b>            (1) Set test patterns for R6 and R7, write R6 to SP (R13) and R7 to R2, and check that the values of R6 and SP (R13) and R7 and R2 match.            (2) Set test patterns for R6 and R7, write R7 to SP (R13) and R6 to R3, and check that the values of R7 and SP (R13) and R6 and R3 match.            (3) Finish the test.</p> <p>Note that Bit0 and 1 of R13 (SP) are fixed to "0". If it matches the expected value, set PASS (0x0001) to "resultTemp", and if it does not match the expected value, set FAIL (0x0000) to "resultTemp".</p>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	

int32_t *result	Test result (0: FAIL / 1: PASS)
<b>Return Values</b>	
NONE	N/A

■ r\_cpu\_diag\_8.asm File

Syntax	
void R_CPU_Diag8(const uint32_t forceFail, int32_t *result)	
Description	
<p>CPU register test process with the WALKPAT algorithm to the General-Purpose Registers (R0-12, R14). (see 1.3.3(2)WALKPAT about the WALKPAT algorithm)</p> <p>The test result is saved in "resultTemp" ( 0 : FAIL / 1 : PASS)</p> <p>The test patterns used are the following (See "r_ramdiag_config.inc"):</p> <p>◆Test patterns</p> <pre> pattern0 : 00000000000000000000000000000000 (0x00000000) pattern0n : 11111111111111111111111111111111 (0xFFFFFFFF) pattern1 : 00000000000000001111111111111111 (0x0000FFFF) pattern1n : 11111111111111110000000000000000 (0xFFFF0000) pattern2 : 00000000111111110000000011111111 (0x00FF00FF) pattern2n : 11111111000000001111111100000000 (0xFF00FF00) pattern3 : 00001111000011110000111100001111 (0x0F0F0F0F) pattern3n : 11110000111100001111000011110000 (0xF0F0F0F0) pattern4 : 00110011001100110011001100110011 (0x33333333) pattern4n : 11001100110011001100110011001100 (0xCCCCCCCC) pattern5 : 01010101010101010101010101010101 (0x55555555) pattern5n : 10101010101010101010101010101010 (0xAAAAAAAA)                     </pre>	
Input Parameters	
const uint32_t forceFail	Forced FAIL Option When set to 0, the function fails forcibly. 0 : Enabled Others : Disabled
int32_t *result	Pointer to store Test result
Output Parameters	
int32_t *result	Test result (0: FAIL / 1: PASS)
Return Values	
NONE	N/A

## 1.2 ROM

This section describes the ROM/Flash memory test using CRC calculator. (Reference: IEC 60730-1:2013 + A1 : 2015+A2:2020 Annex H – H2.19.4.2 CRC – Double 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 CRC32. This software sets 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  $0xFFFFFFFF$  is performed on the output CRC.

### 1.2.2 Multi Checksum

In the ROM test, the ROM area to be tested is divided into 16K bytes as shown in **Figure 1.1**, and the CRC is calculated and stored in a specific area.

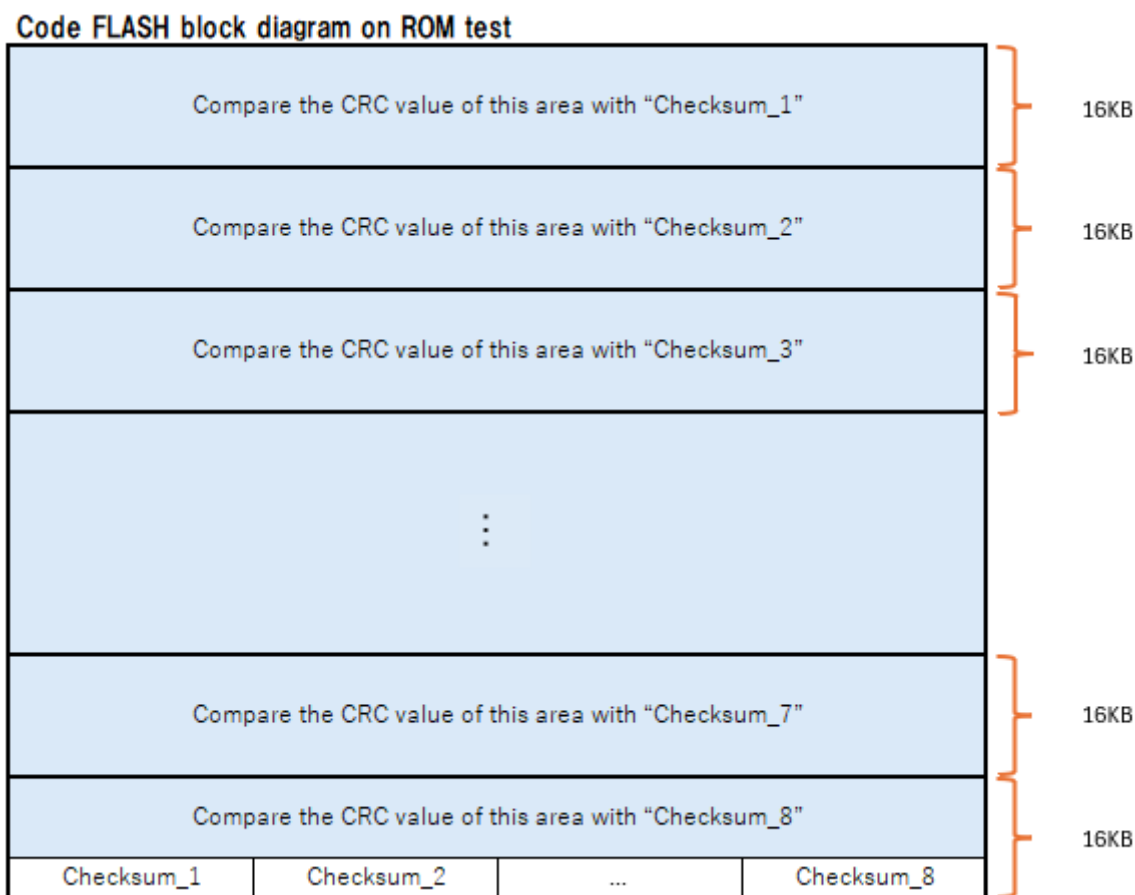
Because this sample software is a product with a code flash memory of 128KB, it is stored at addresses  $0x1FFE0$  to  $0x1FFFF$  when building. \* For products with 256KB memory, change the address accordingly.

In addition, the self-test library divides the process into 16Kbytes each, and after performing the CRC calculation process, it checks for a match with the CRC value stored in the above specified area to determine the ROM test result.

By editing "RA\_SelfTests.c" in the sample project, you can change the enable setting for split processing.

(For details, refer to "**2.2.2 Setting for the support Multi-checksum**".)

The sample project targets the code FLASH area, excluding the checksum storage area.



**Figure 1.1 Code FLASH block diagram on ROM test**

**1.2.3 CRC Software API**

The functions in the remainder 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 1.17 CRC Software API Source Files**

File Name	
crc.h	Defining ROM test API functions
crc_verify.h	Defining ROM test API functions
crc.c	Implementation part of ROM test
CRC_Verify.c	Implementation part of ROM test

### ■ CRC\_Verify.c File

<b>Syntax</b>	
bool_t CRC_Verify(const uint32_t ui32_NewCRCValue, const uint32_t ui32_AddrRefCRC)	
<b>Description</b>	
This function compares a new CRC value with a reference CRC by supplying address where reference CRC is stored.	
<b>Input Parameters</b>	
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.
<b>Output Parameters</b>	
NONE	N/A
<b>Return Values</b>	
bool_t	1 : True = Passed, 0 : False = Failed

### ■ crc.c File

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

<b>Syntax</b>	
uint32_t CRC_Calculate(const uint32_t* pui32_Data, uint32_t ui32_Length)	
<b>Description</b>	
This function calculates the CRC of a single specified memory area.	
<b>Input Parameters</b>	
const uint32_t* pui32_Data	Pointer to start of memory to be tested.
uint32_t ui32_Length	Length of the data in long words.
<b>Output Parameters</b>	
NONE	N/A
<b>Return Values</b>	
Uin32_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	
Prepare the module is 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	
Read-out value from the CRC Data Output Register (CRCDOR) is bit-reversed and returned as the return value.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
uint32_t	The calculated CRC32 value.

### 1.3 RAM

This section describes the RAM test and the two test algorithms used.

The objective of the RAM test is to detect random permanent faults from MCU built-in SRAM.

Key features of the RAM Test are as follows,

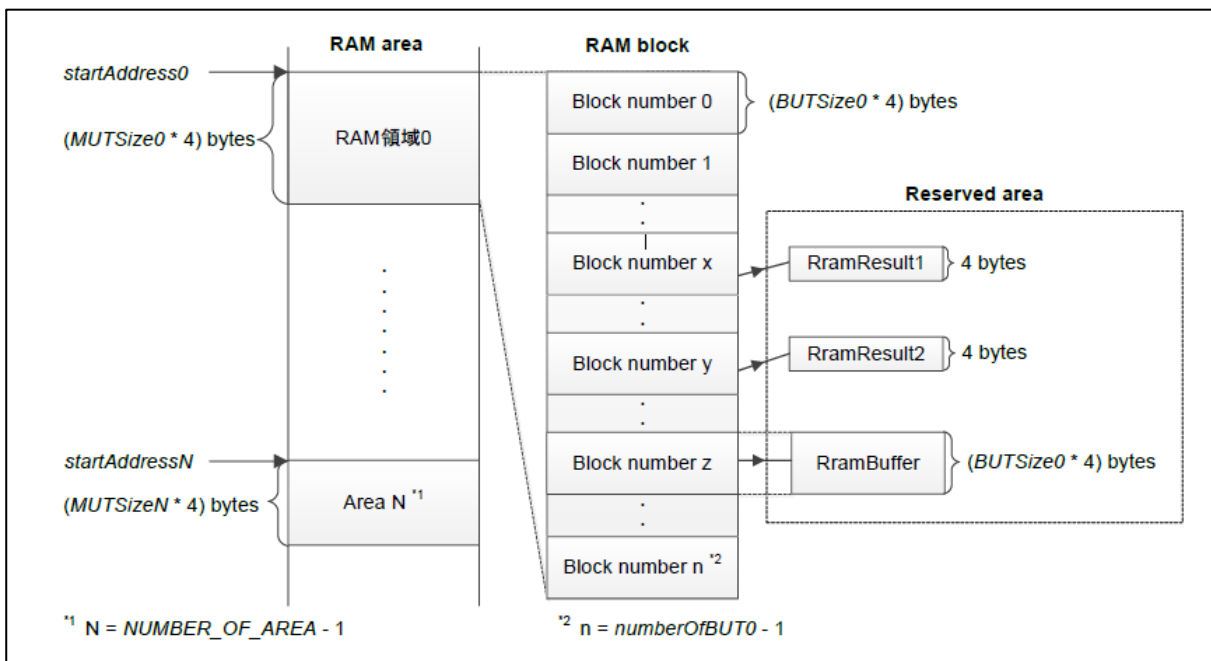
- Whole memory check including stack(s)
- Block-wise implementation of the test
- Supports two test algorithms (Extended March C-, WALKPAT)
- Supports two test types (Destructive / Non-destructive testing)

#### 1.3.1 RAM Block Configuration

Target of the RAM Test is RAM block in the RAM area.

RAM area and RAM block under test are configured by directives described in **Table 1.20**.

**Figure 1.2** shows how RAM area 0 is divided by n block. Directives are indicated by italics.



**Figure 1.2 RAM Block Configuration (example)**

### 1.3.2 Reserved Area

For the RAM test, the user must allocate the following reserved areas to RAM blocks in the Secure area.

1. Buffer (RramBuffer)

In non-destructive test, data value in the RAM block under test is temporarily saved to this buffer. The user shall reserve a specific RAM block for this buffer.

2. Test result variable (RramResult1)

3. Test result variable (RramResult2)

The test result variable is allocated to two different RAM blocks.

By storing copies of test results in two different blocks, a fault can be detected even if one of the variables cannot be stored in the faulting block.

Reserved areas are pre-defined in this software.

Specifically, the files "fsp.ld", "RA\_SelfTests.c", and "r\_ram\_diag\_config.h" define the items related to the reserved area (data save buffer, result variables).

The parts of each definition in this sample software are described below.

◆ Definition parts in the "fsp\_ra0e2.ld" (blue text)

The specific modification procedure involves duplicating the automatically generated "fsp\_gen.ld" file, adding the blue section below, and saving it as "fsp\_ra0e2.ld"

Note: The contents of the "fsp.ld" file vary depending on the FSP version. Please modify it according to the system.

```

/***** RAM memory section allocations *****/
/* ram initialized from flash */
__ram_from_flash$$ :
{
    __ram_from_flash$$Base = .; __ram_from_flash$$Load = LOADADDR(__ram_from_flash$$);
/* RAM BUFFER for RAM Test */
    . = ALIGN(4);
    __RramBuffer_start = .;
        KEEP(*(RAM_TEST_BUFFER*))
    __RramBuffer_stop = .;
/*
*/
    /* section.ram.from_flash */
    *(.ram_from_flash)
    /* section.ram.code_from_flash */
    *(.ram_code_from_flash)
    *(.data*)
    *(vtable)
    __ram_from_flash$$Limit = .;
}> RAM AT > FLASH

```

## ◆ Definition parts in “RA\_Self Tests.c” file (blue text)

```

/--> For RAM test of Class-C
/*Number of bytes to test each time the RAM periodic test is run.*/
/*NOTE: The periodic RAM test requires a safe buffer of the same size as the test size.*/
#define RAM_TEST_BUFFER_SIZE      RAM_BUFFER_SIZE

/*The periodic RAM (including Stack) tests requires a buffer.
   Locate it in its own section after (higher address than) the stacks.*/
volatile uint32_t RramBuffer[RAM_TEST_BUFFER_SIZE] __attribute__((section("RAM_TEST_BUFFER")));
volatile uint32_t RAM_Test_dummy1[RAM_TEST_BUFFER_SIZE-1] __attribute__((section("RAM_TEST_BUFFER")));
volatile uint32_t RramResult1 __attribute__((section("RAM_TEST_BUFFER")));
volatile uint32_t RAM_Test_dummy2[RAM_TEST_BUFFER_SIZE-1] __attribute__((section("RAM_TEST_BUFFER")));
volatile uint32_t RramResult2 __attribute__((section("RAM_TEST_BUFFER")));

```

## ◆ Definition parts in the “r\_ram\_diag\_config.h” file (blue text)

```

/* RAM test buffer size (Expressed in double words) */
/* Note: Set the maximum RAM block size of all RAM areas */
#define RAM_BUFFER_SIZE      (BUTSize0)

```

It is possible to check the location of the "reserved area" with the MAP file generated after build.

## ◆ Applicable parts for the generated MAP file (“RA0E2.map”)

```

ram_from_flash$$
0x20004000 0x31c load address 0x00000440
0x20004000      __ram_from_flash$$Base = .
0x20004000      __ram_from_flash$$Load = LOADADDR (__ram_from_flash$$)
0x20004000      . = ALIGN (0x4)
0x20004000      __RramBuffer_start = .
*(RAM_TEST_BUFFER*)
RAM_TEST_BUFFER
0x20004000 0x300 ./SelfTestLib/src/RA_SelfTests.o
0x20004000      RramBuffer      → RAM Buffer for temporarily saved data : RramBuffer[]
0x20004100      RAM_Test_dummy1
0x200041fc      RramResult1    → result variables : RramResult1
0x20004200      RAM_Test_dummy2
0x200042fc      RramResult2    → result variables : RramResult2
0x20004300      __RramBuffer_stop = .

```

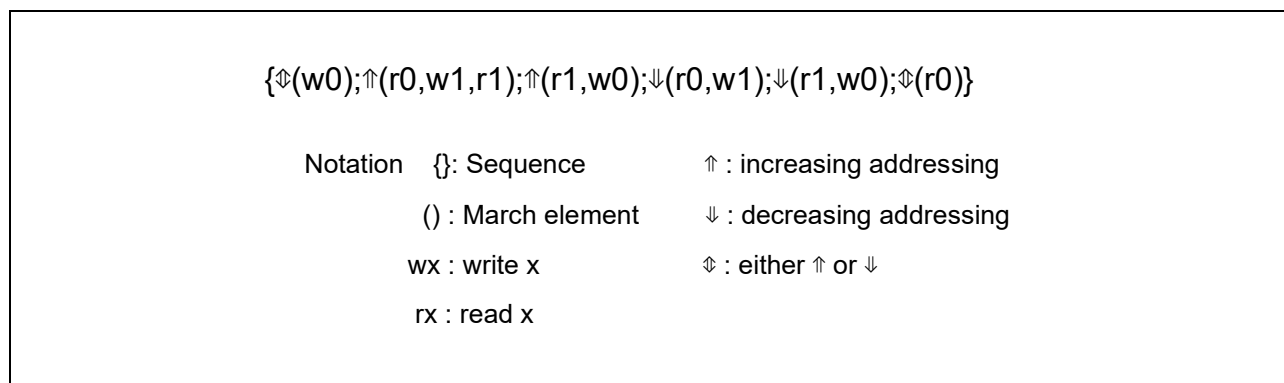
Note: The address to be placed depends on the definition contents of the ld file to be used.

### 1.3.3 RAM Test Algorithm

#### (1) Extended March C-

Extended March C- is one of the March test algorithms used for RAM testing.

The algorithm is represented in **Figure 1.3**.



**Figure 1.3** Extended March C- Algorithm

## (2) WALKPAT

WALKPAT (stands for Walking Pattern) is one of the test algorithms used for RAM testing.

The algorithm is represented in **Figure 1.4**.

```
Write 0 in all cells;
For i=0 to n-1
{
  complement cell[i];
  For j=0 to n-1, j != i
  {
    read cell[j];
  }
  read cell[i];
  complement cell[i];
}
Write 1 in all cells;
For i=0 to n-1
{
  complement cell[i];
  For j=0 to n-1, j != i
  {
    read cell[j];
  }
  read cell[i];
  complement cell[i];
}
```

**Figure 1.4** WALKPAT Algorithm

(3) Algorithm Characteristics

Table 1.18 shows characteristics of two test algorithms available for the RAM Test.

**Table 1.18 RAM Test Algorithm Characteristics**

Fault models and complexity	Extended March C-	WALKPAT
Address Faults (AF)	✓	✓
Stuck At faults (SAF)	✓	✓
Transactional Faults (TF)	✓	✓
Coupling Faults (CF)	✓	✓
Stuck-Open Faults (SOF)	✓	N/A
Data Retention Faults (DRF)	✓	N/A
Sense Amplifier Recovery Faults (SARF)	N/A	✓
Dynamic cross links	✓	✓
Complexity	11n	$\sqrt{2n^2}$

n = the number of addressing cells of the memory

The following algorithm descriptions are related to 1-bit word memory, but they can be applied to m-bit memory.

m-bit memories can be dealt with by repeating each algorithm for a number of times determined by:

$$\lceil \log_2 m \rceil + 1$$

Since m=32bit for this software, the algorithm will be repeated 6 times, and the following 6 different patterns are applied.

- #1: 00000000000000000000000000000000
- #2: 00000000000000001111111111111111
- #3: 00000000111111110000000011111111
- #4: 00001111000011110000111100001111
- #5: 00110011001100110011001100110011
- #6: 01010101010101010101010101010101

### 1.3.4 RAM Software API

The software API source files related to RAM testing are shown in **Table 1.19**.

When RAM Test API is executed, specified one RAM block of RAM area is tested. An RAM fault can be detected by checking the execution result output to the argument.

Before compiling the code, it is necessary to change the RAM block under test and reservation area (see **1.3.2**).

**Table 1.20** shows directive for configuration. The directive can be found in the `r_ram_diag_config.h`

**Table 1.19 RAM Software API source file**

File Name	
<code>r_ram_diag_config.h</code>	Definition of RAM Test Directive.
<code>r_ram_diag_config.inc</code>	Definition of RAM Test execution pattern.
<code>r_ram_diag.c</code>	Definition of RAM Test API function.
<code>r_ram_diag.h</code>	Declaration of RAM Test API function.
<code>r_ram_marchc.asm</code>	Definition of Extended March C- algorithm function.
<code>r_ram_marchc.h</code>	Declaration of Extended March C- algorithm function.
<code>r_ram_walpat.asm</code>	Definition of WALKPAT algorithm function.
<code>r_ram_walpat.h</code>	Declaration of WALKPAT algorithm function.

**Table 1.20 Directives for Software Configuration for RAM Test**

Directive Name	
<code>NUMBER_OF_AREA</code>	Number of RAM area under test (1-8). Shall be set to 1 except for the following case. - multiple RAM areas under test are sporadically allocated - there are multiple RAM blocks under test and each block size is not the same
<code>startAddressN</code> *1	Start address to the RAM area under test
<code>MUTSizeN</code> *1	Size of RAM area under test (N) in double word.
<code>numberOfBUTN</code> *1	Number of RAM blocks under test.
<code>BUTSizeN</code> *1	Size of RAM block under test (N) in double word. Calculated by $BUTSizeN = MUTSizeN / numberOfBUTN$
<code>RAM_BUFFER_SIZE</code>	Size of buffer (RramBuffer) under test in double word.

Note: 1. N=0~(NUMBER\_OF\_AREA – 1)

## ■ r\_ram\_diag.c File

Syntax	
void R_RAM_Diag(uint32_t area, uint32_t index, uint32_t algorithm, uint32_t destructive)	
Description	
<p>This function verifies RAM.</p> <p>Test result can be checked by the return value in result variable (RramResult1, RramResult2)</p> <p>If Test result is PASS :</p> <p style="padding-left: 20px;">RramResult1 = 1 and RramResult2 = 1</p> <p>If Test result is FAIL :</p> <p style="padding-left: 20px;">Other than above</p> <p>Perform the RAM tests in the following order:</p> <ol style="list-style-type: none"> <li>1. Check if the RAM block is a valid area by the arguments "area" and "index".</li> <li>2. Use the macro functions (R_RAM_BLK_SADR, R_RAM_BLK_EADR) to calculate the start and end addresses of the RAM block under test. (The calculated start address and end address are saved in sAdr and eAdr.)</li> <li>3. The function of the corresponding algorithm is called by the argument "algorithm". For Extended March C- (algorithm = RAM_ALG_MARCHC): R_RAM_Diag_MarchC () function For WALKPAT (algorithm = RAM_ALG_WALPAT) : R_RAM_Diag_Walpat () function</li> </ol> <p>Note:</p> <p>The argument "destructive" selects whether the data is destructive or non-destructive. (In the case of the destruction test, the RAM block is cleared to "0" after the test.)</p> <ol style="list-style-type: none"> <li>4. Return to the called function.</li> </ol>	
Input Parameters	
uint32_t area	Number of RAM area Shall be smaller than the directive NUMBER_OF_AREA. Returns 0 (FAIL) when the value is invalid.
uint32_t index	RAM block index of RAM area set in "area" RAM block index starts with 0. It will be smaller than the directive numberOfBUTN. (See <b>Table 1.20</b> ) Returns 0 (FAIL) when the value is invalid.
uint32_t algorithm	Specify the algorithm. 0(RAM_ALG_MARCHC): Extended March C- 1(RAM_ALG_WALPAT): WALKPAT *WALKPAT when the value is other than 0."
uint32_t destructive	Specify type of the Memory test. 0: Non-destructive test 1: Destructive test Non-destructive test when invalid value is set. RAM block is cleared to 0 after destructive test. Notice: RAM block is always cleared to 0 when the block with buffer, regardless of test type.
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

■ r\_ram\_marchc.asm File

Syntax	
void R_RAM_Diag_MarchC(uint32_t start, uint32_t end, uint32_t destructive)	
Description	
<p>Performs RAM test processing by the "Extended March C-"algorithm for the RAM block specified by the arguments start and end. (See 1.3.3(1) )</p> <p>In the case of non-destructive test, the current data of the test area is saved in the specified RamBuffer area.</p> <p>The test results are stored below.</p> <ul style="list-style-type: none"> <li>- RramResult1 ( 0 : FAIL / 1 : PASS)</li> <li>- RramResult2 ( 0 : FAIL / 1 : PASS)</li> </ul> <p>The test patterns used are the following (See "r_ramdiag_config.inc"):</p> <p>◆ Test Pattern</p> <pre> pattern0 : 00000000000000000000000000000000 (0x00000000) pattern0n : 11111111111111111111111111111111 (0xFFFFFFFF) pattern1 : 00000000000000001111111111111111 (0x0000FFFF) pattern1n : 11111111111111110000000000000000 (0xFFFF0000) pattern2 : 00000001111111100000000111111111 (0x00FF00FF) pattern2n : 11111110000000011111111000000000 (0xFF00FF00) pattern3 : 00001111000011110000111100001111 (0x0F0F0F0F) pattern3n : 11110000111100001111000011110000 (0xF0F0F0F0) pattern4 : 00110011001100110011001100110011 (0x33333333) pattern4n : 11001100110011001100110011001100 (0xCCCCCCCC) pattern5 : 01010101010101010101010101010101 (0x55555555) pattern5n : 10101010101010101010101010101010 (0xAAAAAAAA) </pre>	
Input Parameters	
uint32_t start	Start address of the block under test
uint32_t end	End address of the block under test
uint32_t destructive	Specify type of the Memory test. 0: Non-destructive test 1: Destructive test
Output Parameters	
RramResult1	0 : FAIL / 1 : PASS
RramResult2	0 : FAIL / 1 : PASS
Return Values	
NONE	N/A

■ r\_ram\_walpat.asm file

Syntax	
void R_RAM_Diag_walpat(uint32_t start, uint32_t end, uint32_t destructive)	
Description	
<p>Performs RAM test processing by the "WALKPAT" algorithm for the RAM block specified by the arguments start and end. (See <b>1.3.3(2)</b> )</p> <p>In the case of non-destructive test, the current data of the test area is saved in the specified RamBuffer area.</p> <p>The test results are stored below.</p> <ul style="list-style-type: none"> <li>- RramResult1 ( 0 : FAIL / 1 : PASS)</li> <li>- RramResult2 ( 0 : FAIL / 1 : PASS)</li> </ul> <p>The test patterns used are the following (See "r_ramdiag_config.inc"):</p> <p>◆Test patterns</p> <pre> pattern0 : 00000000000000000000000000000000 (0x00000000) pattern0n : 11111111111111111111111111111111 (0xFFFFFFFF) pattern1 : 00000000000000001111111111111111 (0x0000FFFF) pattern1n : 11111111111111110000000000000000 (0xFFFF0000) pattern2 : 00000000111111110000000011111111 (0x00FF00FF) pattern2n : 11111111000000001111111100000000 (0xFF00FF00) pattern3 : 00001111000011110000111100001111 (0x0F0F0F0F) pattern3n : 11110000111100001111000011110000 (0xF0F0F0F0) pattern4 : 00110011001100110011001100110011 (0x33333333) pattern4n : 11001100110011001100110011001100 (0xCCCCCCCC) pattern5 : 01010101010101010101010101010101 (0x55555555) pattern5n : 10101010101010101010101010101010 (0xAAAAAAAA) </pre>	
Input Parameters	
uint32_t start	Start address of the block under test
uint32_t end	End address of the block under test
uint32_t destructive	Specify type of the Memory test. 0: Non-destructive test 1: Destructive test
Output Parameters	
RramResult1	0 : FAIL / 1 : PASS
RramResult2	0 : FAIL / 1 : PASS
Return Values	
NONE	N/A

## 1.4 Clock

This test uses the interval measurement function of input pulses via Channel 5 of MCU's built-in Timer Array Unit (TAU) to verify that the system clock frequency is within the allowable range. If it is outside the range, it detects a frequency abnormality and performs error handling.

### 1.4.1 Monitoring the system clock frequency using TAU0 ch5 (Input pulse interval measurement function)

Counting starts on the rising edge of the input pulse signal selected by Timer Input Select Register 0 (TIS0.TIS[2:0]). The timer's count value is captured on the rising edge of the next pulse. This method measures the interval between input pulses.

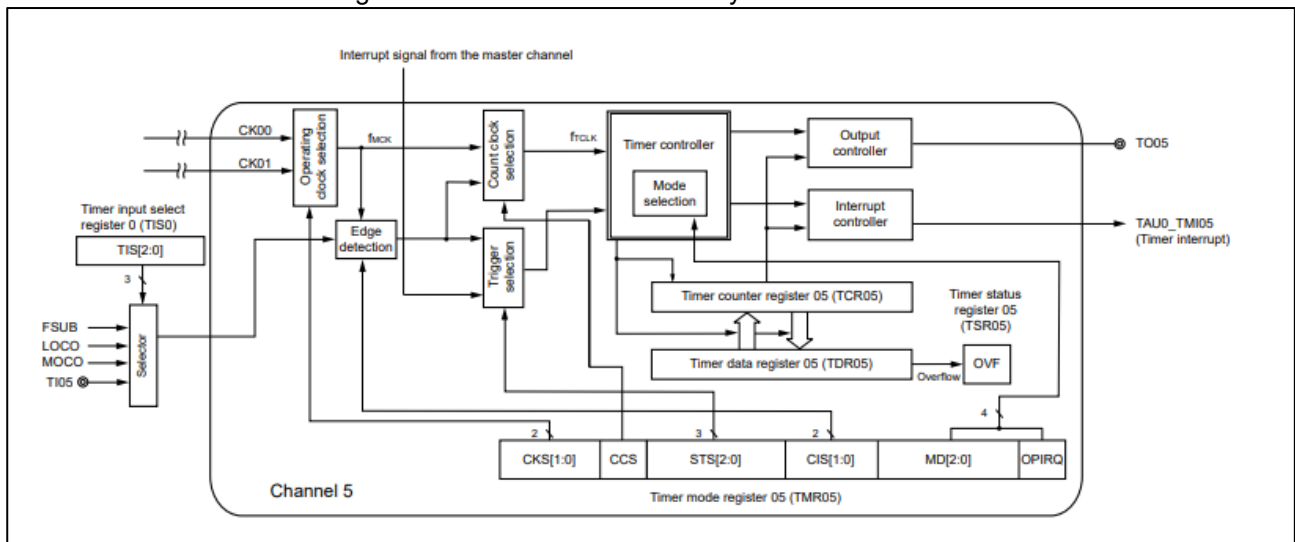
Select "LOCO" for the timer input chosen by Timer Input Select Register 0 (TIS0.TIS[2:0]), and select "PCLKB (no division)" for the operating clock of Channel 5 in the related register (\*1).

Note: 1. The corresponding registers and settings are TMR05.CCS=0, TMR05.CKS[1:0]=b'00, TPS0.PRS0[3:0]=0x0.

In RA0E2, PCLKB (Peripheral Module Clock) and the system clock (ICLK) share the same clock source.

Since the system clock (ICLK) cannot be directly frequency-detected, the frequency detection is performed on PCLKB, which shares the same clock source.

Below is the internal block diagram for channel 5 of timer array unit 0 used in this instance.



**Figure 1.5 Internal block diagram of channel 5 of timer array unit 0**

In the sample software, when a valid edge on the LOCO input is detected, PCLKB starts counting as the operating clock. Upon detecting the next valid edge, the value of the TCR05 register is captured into Timer Data Register 05 (TDR05), and TAU0\_TMI05 is generated. (TCR05 register restarts counting from 0x0000.)

\*For details, refer to the chapter "Timer Array Unit (TAU)" in the User's Manual Hardware Edition.

Additionally, within the TAU0 ch5 interrupt, it determines whether the system clock frequency is within the set range. If it is outside the range, error handling is executed.

For usage examples of this test module, refer to **Chapter 2.4**.

### 1.4.2 Clock Test Software API

**Table 1.21 Clock Test Software API source file**

File Name
clock_monitor.h
clock_monitor.c

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

#### ■ clock\_monitor.c File

Syntax	
void ClockMonitor_Init( CLOCK_MONITOR_ERROR_CALL_BACK Callback)	
Description	
<ol style="list-style-type: none"> <li>1. Register error detection processing function.</li> <li>2. Initialize TAU0(ch5) module and start monitoring PCLKB (system clock as clock source)</li> <li>3. If LOCO was stopped, start oscillation.</li> </ol>	
Input Parameters	
CLOCK_MONITOR_ERROR_CALL_BACK Callback	Function called when PCLKB is out of tolerance
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
void tau0_tmi05_int (void)	
Description	
<p>TAU0 ch5 capture completion interrupt handling. The interrupt occurs when capture is completed. The function determines whether the captured input pulse falls within the valid range. If it is outside the valid range, it calls the callback function registered in the function ClockMonitor_Init.</p> <p>※ The valid range is defined by and “hwMAXTIME” and “hwMINTIME” in clock_monitor.h.</p> <p>The behavior depends on the value of the variables “f_CLOCK_FREQ_ERR” and “f_Clock_TAU5_Test”            If “f_CLOCK_FREQ_ERR” is other than “0”, the capture result is ignored and no validity check is performed            If “f_Clock_TAU5_Test” is other than “0”, the callback result is not called even if the result indicates an invalid range.            These two variables are set by the function TAU5_Err_Detect_Test() to verify TAU0 ch5 interrupt operation.</p> <p>If only using this function, configure “f_CLOCK_FREQ_ERR” and “f_Clock_TAU5_Test” according to your usage conditions.</p>	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
bool_t TAU5_Err_Detect_Test(void)	
Description	
<p>Verify that TAU ch5 interrupt function operates correctly upon Power-On            Within the function, setting ‘f_Clock_TAU5_Test = 1’ and ‘f_CLOCK_FREQ_ERR = 0xff’ indicates this is the “TAU0 ch5 interrupt operation test” within the TAU ch5 interrupt handler.            If interrupt are confirmed within a fixed timer period (counted by a software loop), it returns “TRUE”            If interrupts cannot be confirmed , it returns “FALSE”.</p> <p>Note that the function stops the timer operation of TAU0 ch5 after the operation check.</p>	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : TRUE = PASS(Confirm the occurrence of each interrupt) 0 : FALSE = FAIL(Unable to confirm)

## 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 an 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.

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

### 1.5.1 IWDT Software API

The software API source files related to IWDT testing are shown in **Table 1.22**.

**Table 1.22 Independent Watchdog Timer source file**

File Name	
iwdt.h	Declaration of IWDT Test API function.
iwdt.c	IWDT test implementation part

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	
bool_t	N/A

Syntax	
bool_t IWDT_DidReset(void)	
Description	
Returns true if the IWDT has timed out or not been refreshed correctly.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	True if watchdog timer has timed out, otherwise false.

Syntax	
bool_t IWDT_Err_Detect_Test(void)	
Description	
<p>When the power is turned on, it check that the interrupt by the detection of counter underflow for IWDT function is operating normally.</p> <p>Returns "TRUE" if NMI interrupt occurrence by detecting IWDT counter underflow can be confirmed within a certain period of time (counted by software loop).</p> <p>Set f_IWDT_ERROR_TEST to "1" and determine if f_IWDT_ERROR_TEST becomes "0" within a certain period of time.</p> <p>Note:</p> <p>If the watchdog timer is configured to execute a reset upon error detection (OFS0.IWDTRSTIRQS=1), do not use this function to verify normal operation.</p>	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : True = Passed(NMI interrupt occurrences was occurred) 0 : False = Failed(Could not be confirmed NMI interrupt occurrences)

<b>Syntax</b>	
bool_t IWDT_Err_Test_Judge(void)	
<b>Description</b>	
<p>Check whether the problem is caused by counter underflow detection during IWDT error test at power on, or whether it is caused by some other factor.</p> <p>If NMI interrupt has occurred due to IWDT underflow confirmed by IWDT error test, return "TRUE", set "f_IWDT_ERROR_TEST" to "0" and clear the IWDT underflow flag.</p> <p>Return "FALSE" in other case than above.</p> <p>Note that this function must be called within NMI_Handler_callback().</p> <p>For details, refer to 2.5.2 Example of registering and writing an NMI interrupt callback function.</p>	
<b>Input Parameters</b>	
NONE	N/A
<b>Output Parameters</b>	
NONE	N/A
<b>Return Values</b>	
bool_t	"TRUE" if by IWDT error test , otherwise FALSE

## 2. Example Usage

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

Self test can be divided into two patterns:

(a) Power-On Test

These tests are 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 tests are run regularly throughout normal program operation. This document does not provide a judgment of how often a particular test should be run. 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 test 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

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

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

### 2.1.2 Periodic

To test the CPU periodically, use the function `CPU_Test_ClassC` similar to the Power-On test.

Calling it regularly allows automatic execution of CPU tests.

Alternatively, to reduce the amount of testing done in a single function call, the user can select by "`r_cpu_diag_config.h`".

### 2.1.3 Preparation for CPU testing

The following describes the preparation for CPU testing.

It configures the CPU test via directive settings before compiling your code.

See **Table 1.15** for the relationship between directives and each CPU test.

Directives are used to define what tests will be included in or excluded from the compilation.

The directive can be found in the `r_cpu_diag_config.h` file.

The sample software is set to build all CPU tests.

If set the directives to "0"(an excluded from test), the empty function called `norm_null()` is executed.

The next page shows where to set the directives that make up the CPU test.

◆ Define parts in the “r\_cpu\_diag\_config.h” file (blue text)

If “1” is set in the following settings, it will be subject to test execution, and if “0” is set, it will not be subject to test execution.

```
/******  
*****  
* Macro definitions  
*****  
*****/  
/* ==== Define build options ==== */  
#define BUILD_R_CPU_DIAG_0    (1)  
#define BUILD_R_CPU_DIAG_1    (1)  
#define BUILD_R_CPU_DIAG_2    (1)  
#define BUILD_R_CPU_DIAG_3    (1)  
#define BUILD_R_CPU_DIAG_4    (1)  
#define BUILD_R_CPU_DIAG_5    (1)  
#define BUILD_R_CPU_DIAG_6    (1)  
#define BUILD_R_CPU_DIAG_7_1  (1)  
#define BUILD_R_CPU_DIAG_7_2  (1)  
#define BUILD_R_CPU_DIAG_7_3  (1)  
#define BUILD_R_CPU_DIAG_8    (1)
```

## 2.2 ROM

ROM test compares the calculated CRC value for the tested range with the pre-stored reference CRC value. (A 32-bit CRC32 polynomial is used.)

The CRC comparator and software comparison process achieve “99.6% coverage of all information errors.”

The reference CRC value must be stored in a ROM area not included in the CRC calculation. The method for calculating the reference CRC value varies depending on the development environment.

Furthermore, this sample software performs split processing to reduce the processing load of the ROM test and supports Multi Checksum.

The CRC module built into the RA MCU must be initialized before use by calling the CRC\_Init function. When performing split processing, initialize only during the first instance of split processing.

### 2.2.1 Reference CRC Value Calculation in Advance

Since the GNU tool does not have a CRC calculation function, use the SRecord tool (\*) 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.

Note: SRecord is an open source project on SourceForge. Check below contents for details.

- SRecord Web Site (SRecord v1.65)

<http://srecord.sourceforge.net/>

- CRC Checksum Generation with “SRecord” Tools for GNU and Eclipse

<https://sourceforge.net/projects/srecord/files/srecord-win32/1.65/>

After unzipping the downloaded ZIP file, the following programs are extracted to “\srecord-1.65.0-win64.zip\srecord-1.65.0-win64\bin”

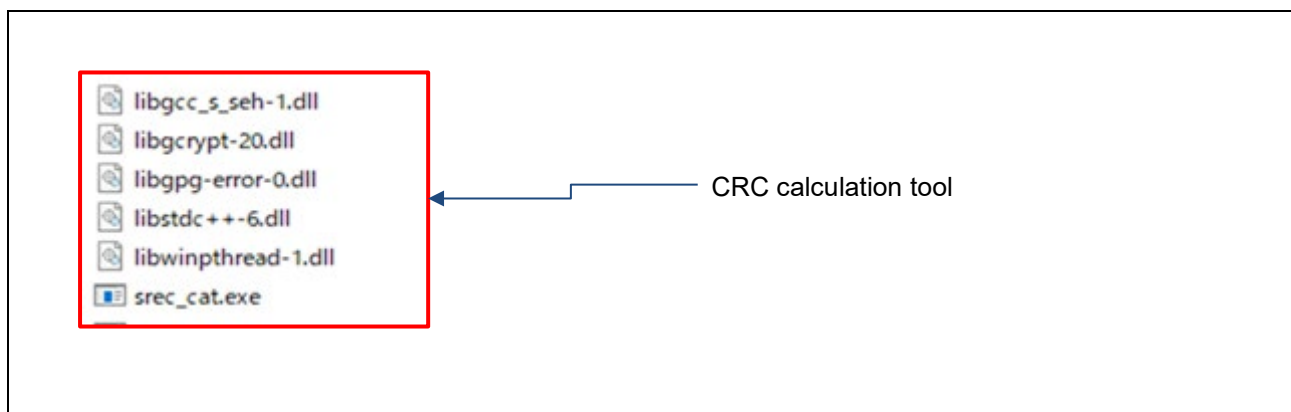


Figure 2.1 SRecord Tool Contents

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

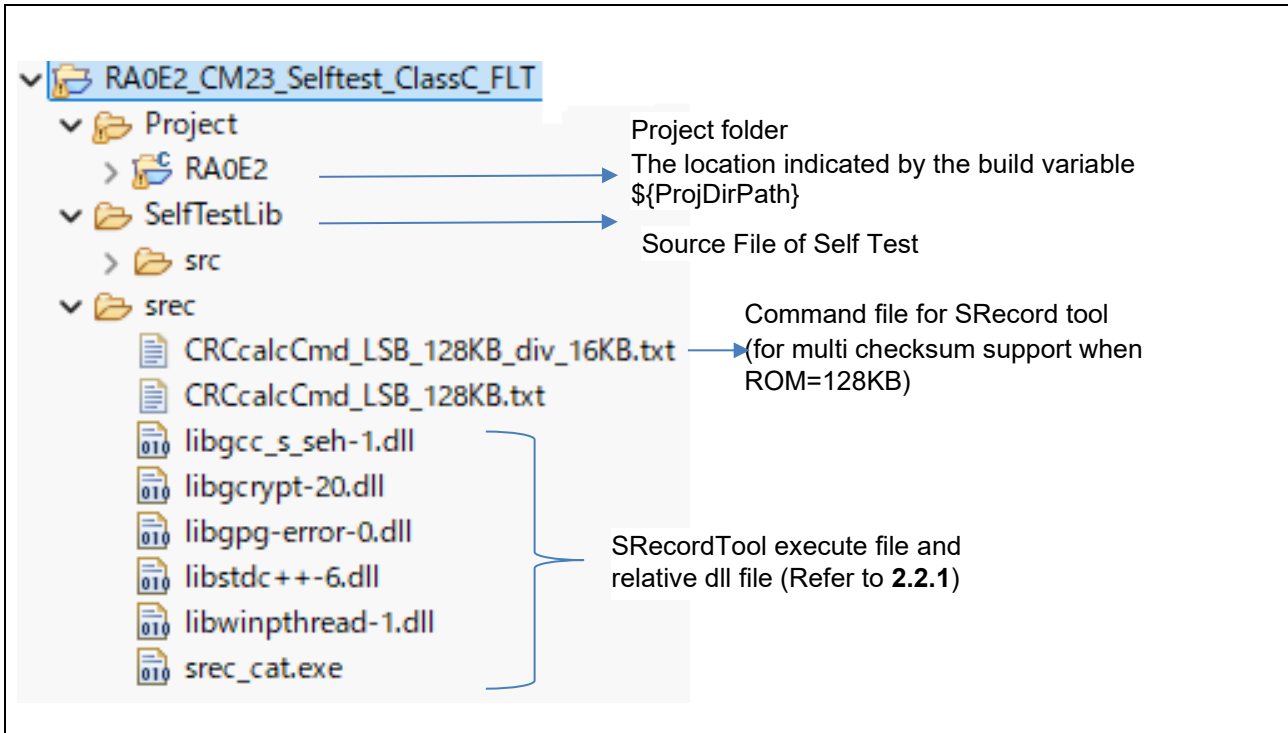


Figure 2.2 Folder Configuration Example

Next, the settings for using the tool in this sample project will be explained to pre-write reference CRC values into ROM.

◆ Setting in the project

Open e<sup>2</sup>studio menu bar: Project ⇒ Properties. In the ‘Settings’ section, under the ‘Build Steps’ tab, use the objcopy command in the ‘Build Steps’ field to generate an SRecord file from the generated \*.elf file.

For reference purposes in the following examples, the filename before writing the CRC value is Original.srec. This file becomes the input for the SRecord tool.

For details, refer to **Figure 2.3** and the later section “Example Entry for the Command(s): Field in ‘Post-build steps’”. \*( ) indicates the English version of e<sup>2</sup>studio.

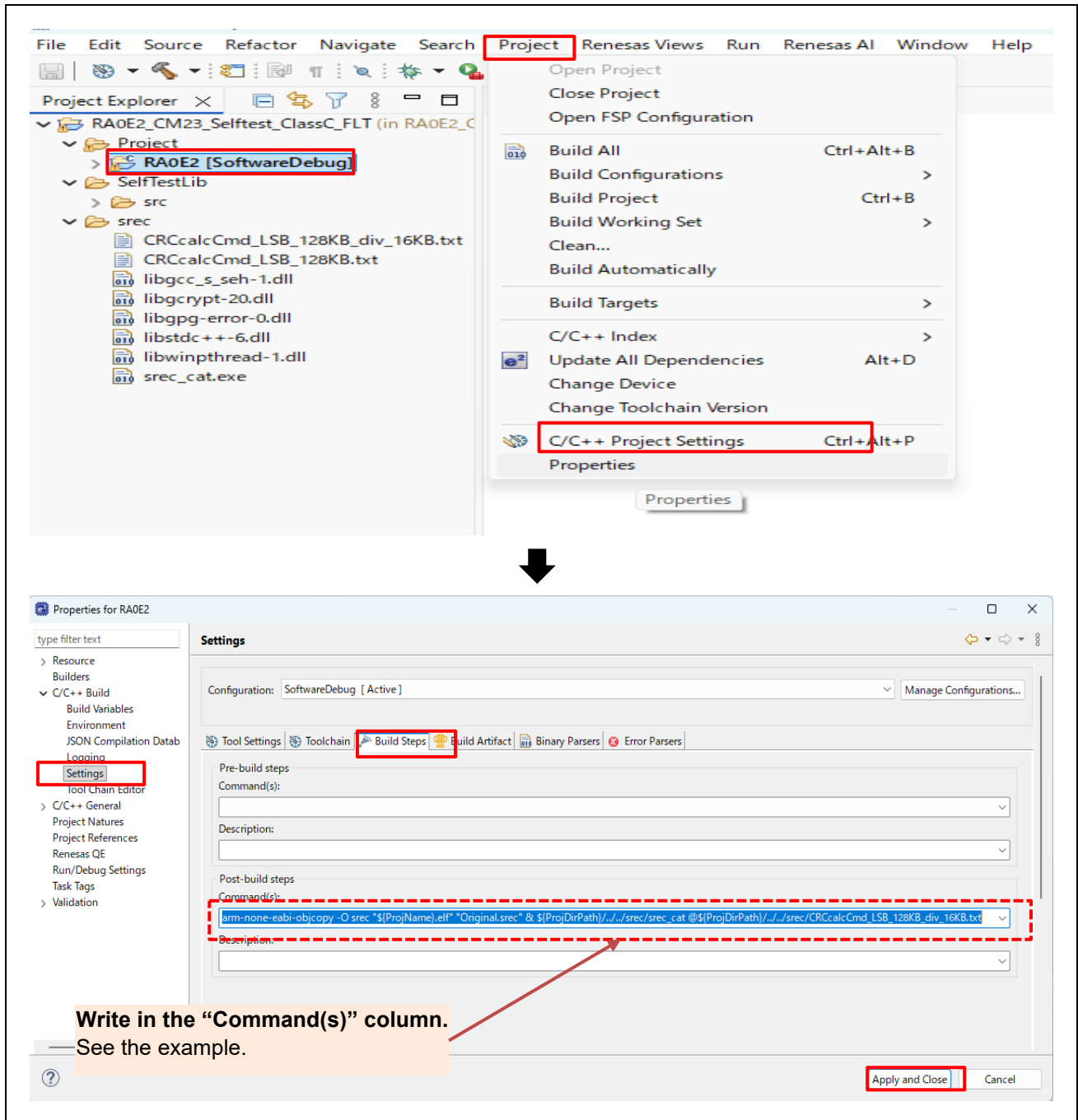


Figure 2.3 Output S-Record file and start SRecord tool

In the "Post-build steps" of the "Build Steps" tab in **Figure 2.3**, describe as follows. \* () indicates in e<sup>2</sup>studio English version.

■□ Example of entry in the Command (s) column of "Post-build steps" (write on one line without line breaks) [when divided processing is enabled (DIV\_AREA=1)] ※please use this settings.

```
arm-none-eabi-objcopy -O srec "${ProjName}.elf" "Original.srec" & ${ProjDirPath}/../srec/srec_cat  
@${ProjDirPath}/../srec/CRCcalcCmd_LSB_128KB_div_16KB.txt
```

Until before the "&" in the first line above mean that the S-record file is generated.

The format "srec\_cat @ command file" on the second line is the launch of the srec\_cat tool.

The description example is shown about the following Command files:

"CRCcalcCmd\_LSB\_128KB\_div\_16KB.txt"(when divided processing is enabled)

Also, please refer to "2.2.2 Setting for the support Multi-checksum" for setting of split processing.

■ CRCcalcCmd\_LSB\_128KB\_div\_16KB.txt file contents (example)

```

# CRC calculate
Original.srec
-fill 0xFF 0x00000 0x020000
# Area No.8
-crop 0x1C000 0x1FFE0
-CRC32_Little_Endian 0x01FFFC -CCITT
-crop 0x1FFFC 0x200000
Original.srec
# Area No.7
-fill 0xFF 0x00000 0x20000
-crop 0x18000 0x1C000
-CRC32_Little_Endian 0x01FFF8 -CCITT
-crop 0x1FFF8 0x200000
Original.srec
# Area No.6
-fill 0xFF 0x00000 0x18000
-crop 0x14000 0x18000
-CRC32_Little_Endian 0x01FFF4 -CCITT
-crop 0x1FFF4 0x200000
Original.srec
# Area No.5
-fill 0xFF 0x00000 0x14000
-crop 0x10000 0x14000
-CRC32_Little_Endian 0x01FFF0 -CCITT
-crop 0x1FFF0 0x20000
Original.srec
# Area No.4
-fill 0xFF 0x00000 0x10000
-crop 0xC000 0x10000
-CRC32_Little_Endian 0x01FFEC -CCITT
-crop 0x1FFEC 0x20000
Original.srec
# Area No.3
-fill 0xFF 0x00000 0xC000
-crop 0x8000 0xC000
-CRC32_Little_Endian 0x01FFE8 -CCITT
-crop 0x1FFE8 0x20000
Original.srec
# Area No.2
-fill 0xFF 0x00000 0x8000
-crop 0x4000 0x8000
-CRC32_Little_Endian 0x01FFE4 -CCITT
-crop 0x1FFE4 0x20000
Original.srec
# Area No.1
-fill 0xFF 0x00000 0x4000
-crop 0x0000 0x4000
-CRC32_Little_Endian 0x01FFE0 -CCITT
-crop 0x1FFE0 0x20000
Original.srec
#
-fill 0xFF 0x00000 0x01FFE0
-Output_Block_Size 16
-Output addcrc.srec -Motorola -address-length=4

# Read srec file
# 128KB ROM fill by 0xFF

# CRC calculate area (Test area 0x1C000 - 0x1FFDF : 16KB-16B) for debug
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFFC.
# Keep CRC area(0x1FFFC - 0x1FFFF)
# Read srec file

# 0-0x1C000 ROM fill by 0xFF
# CRC calculate area (Test area 0x180000 - 0x1BFFF : 16KB)
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFF8.
# Keep CRC area(0x1FFF8 - 0x1FFFF)
# Read srec file

# 0-0x18000 ROM fill by 0xFF
# CRC calculate area (Test area 0x14000 - 0x17FFF : 16KB)
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFF4.
# Keep CRC area(0x1FFF4 - 0x1FFFF)
# Read srec file

# 0-0x14000 ROM fill by 0xFF
# CRC calculate area (Test area 0x10000 - 0x13FFF : 16KB)
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFF0.
# Keep CRC area(0x1FFF0 - 0x1FFFF)
# Read srec file

# 0-0x10000 ROM fill by 0xFF
# CRC calculate area (Test area 0xC0000 - 0xFFFF : 16KB)
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFEC.
# Keep CRC area(0x1FFEC - 0x1FFFF)
# Read srec file

# 0-0xC000 ROM fill by 0xFF
# CRC calculate area (Test area 0x8000 - 0x1BFF : 16KB)
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFE8.
# Keep CRC area(0x1FFE8 - 0x1FFFF)
# Read srec file

# 0-0x8000 ROM fill by 0xFF
# CRC calculate area (Test area 0x4000 - 0x7FFF : 16KB)
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFE4.
# Keep CRC area(0x1FFE4 - 0x1FFFF)
# Read srec file

# 0-0x4000 ROM fill by 0xFF
# CRC calculate area (Test area 0x0000 - 0x3FFF : 16KB) for debug
#(chg) CRC32 LSB (with initial value : 0xFFFFFFFF), and store CRC Value at 0x1FFE0.
# Keep CRC area(0x1FFE0 - 0x1FFFF)
# Read srec file

# -fill 0xFF from 0x0 to 0x1FFE0
#(chg) Set Block size
#(chg) 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 generated in the build configuration folder under the project folder, so download it to the target board.

From the Menu bar, select “Run” and then “Debug Configurations”.

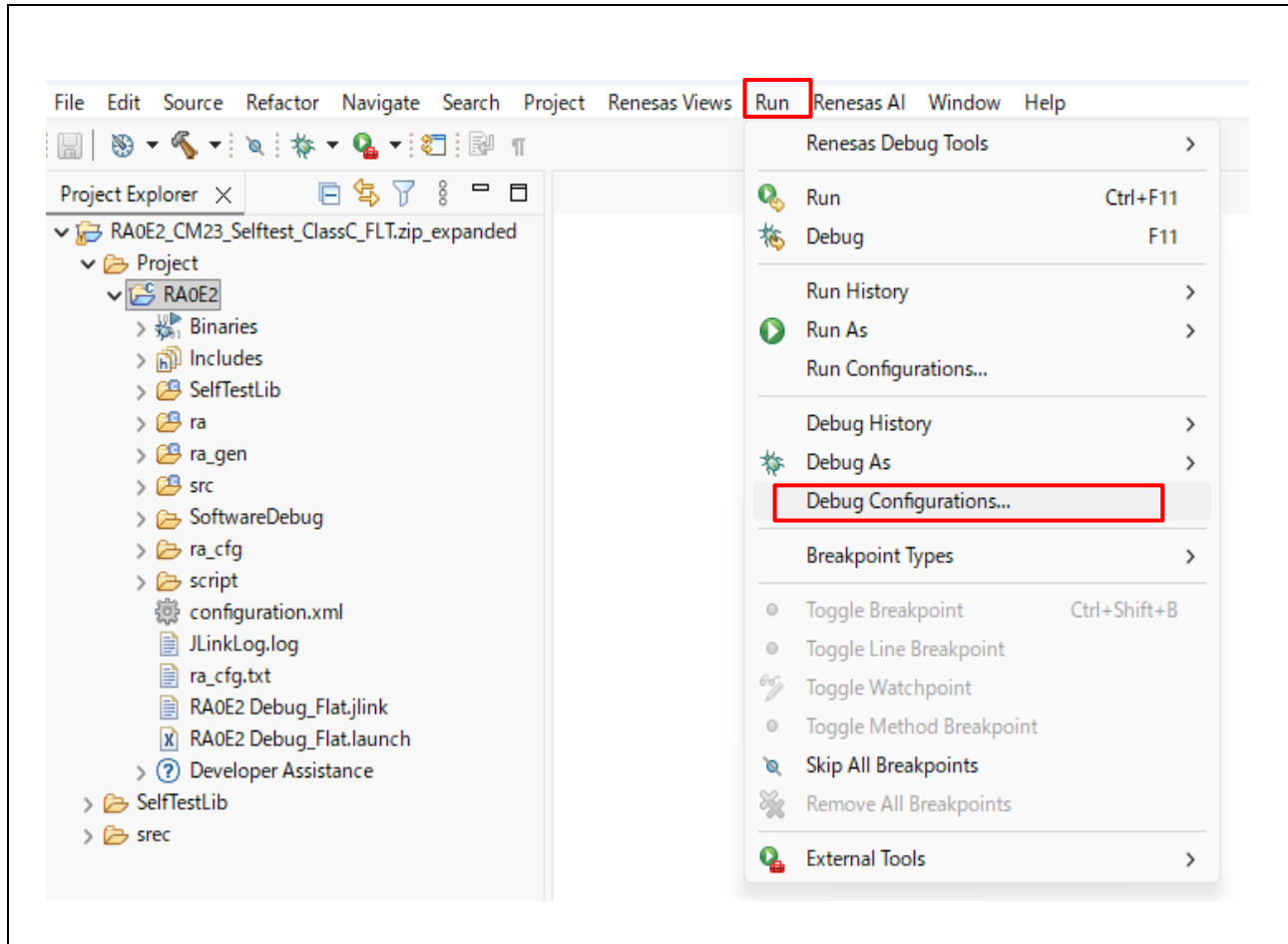


Figure 2.4 Select Debug Configuration of the Project

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

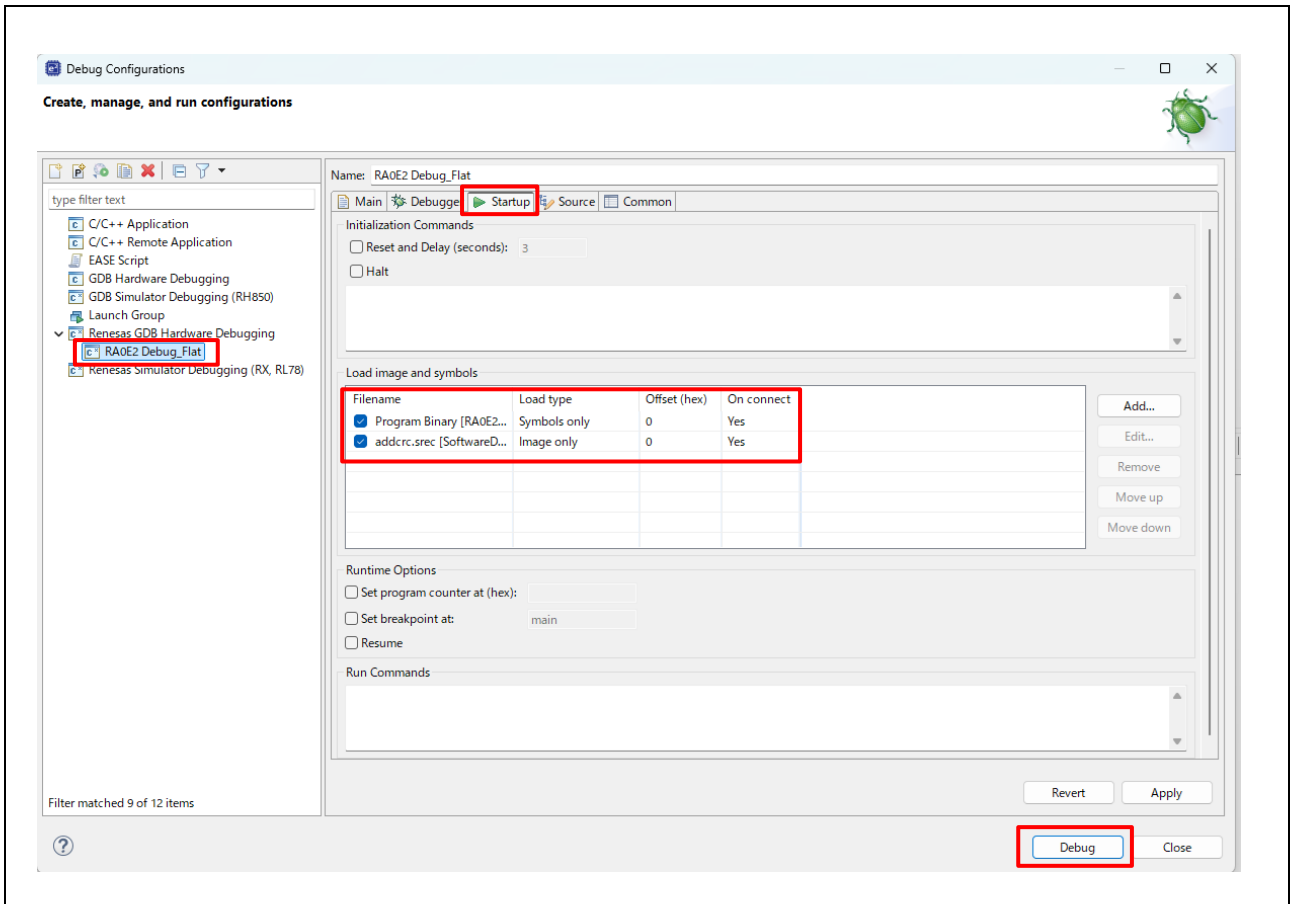


Figure 2.5 Load Image and Symbol Setting Example

## 2.2.2 Setting for the support Multi-checksum

It takes some time to test all areas in one ROM test. As measure, it is possible to divide the processing with the following settings.

Edit and set "RA\_Self Tests.c" including this sample software. Divided processing is enabled by default.

◆ Setting part in the "RA\_SelfTests.c" file of the sample software (for RA0E2) (blue text)

Set whether to enable or disable split processing below

```
#define DIV_AREA 1           // 0:Not divide  1:Do divide
```

The reference addresses for pre-computed CRC values are defined below

```
/* The address where the 32bit reference CRC value will be stored.
   The linker must be configured to generate a CRC value and store it at this location. */
#define DIV_AREA 1           // 0:Not divide  1:Do divide
#if(DIV_AREA==1)
#define CRC_ADDRESS 0x0001FFE0 // Flash ROM 128KB *The area from 0x1FFE0 to 0xFFFFF is stored Calurated CRC Value.
#endif
```

It stores the precomputed checksum with the above settings.

When divided processing is enabled. (DIV\_AREA=1) : Store in the area of address 0x1FFE0 to 1FFFF.

For the stored method, refer to "2.2.1 Reference CRC Value Calculation in Advance"

### 2.2.3 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 the user's responsibility to ensure that the ROM area to be tested is included in the CRC calculation.

### 2.2.4 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.

1. When testing RAM, keep the following points in mind:
2. Include `r_ram_diag.h`.
3. Modify the directives in `r_ram_diag_config.h` as needed (see **Table 1.20**).
4. Define the required parameters for `R_RAM_Diag` (see **1.3.4**), pass the parameters and call the function `R_RAM_Diag`.
5. For non-destructive tests, allocate a buffer (`RramBuffer`) and set the protected data to be stored in other blocks

### 2.3.1 Power-On

At power on, a RAM test is performed.

First performing the RAM test with the Extended March C-algorithm, then performing the RAM test with the WALKPAT algorithm.

It is possible to choose a destructive test.

If startup time is very important, make fine adjustments such as limiting the area to be tested and the test algorithm to be used.

### 2.3.2 Periodic

All periodic tests must be non-destructive.

In the periodic RAM test, select "Extended March C-" or "WALKPAT" as the algorithm to be used. (\* Select "WALKPAT" in the sample project)

Also, if the test target area is wide, the processing time will be long, so it will be necessary to divide the RAM blocks according to the system.

## 2.4 Clock

To monitor the system clock, configure TAU0 Channel 5 related settings in the FSP Configuration settings screen.

The “Stacks Configuration” screen below allows to verify settings related to the capture function for TAU0 Channel 5.

The procedure is as follows.

1. Double-click the configuration.xml file in the red box in Project Explorer
2. FSP Configuration screen appears → Select the “Stacks” tab in the red box to display the “Stacks Configuration” screen
3. Select “g\_timer05 Timer, Independent Channel, ... ” under Thread to select TAU0 ch5
4. Select “Properties” tab.

The settings (“Settings”) for “Stacks Configuration” screen are displayed as shown below.

Next, select “General” and “Input” for “Common”, “Module g\_timer05 Timer...” to display the detail settings.

The red dotted box indicates the main TAU0 ch5 configuration items.

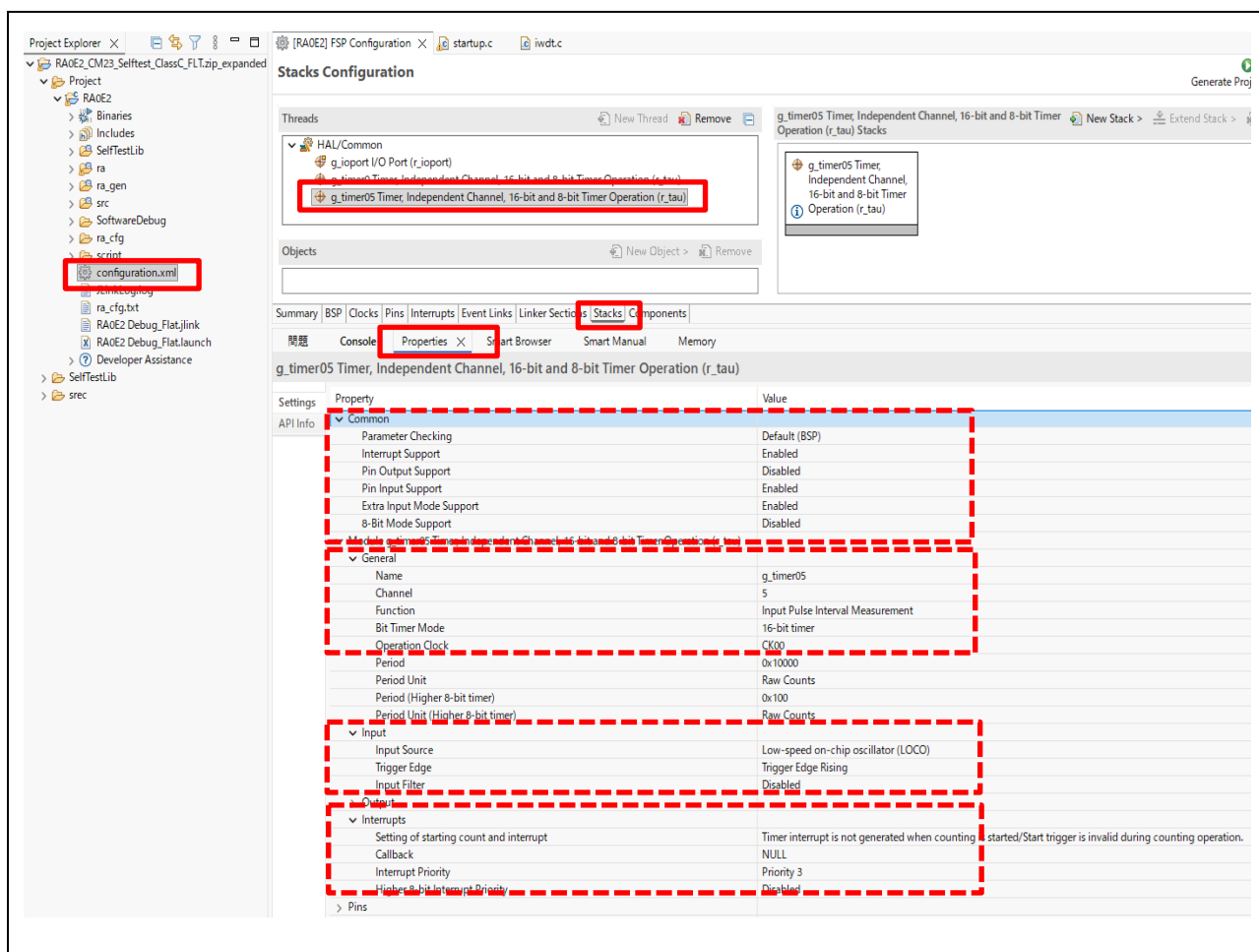


Figure 2.6 FSP Configuration Setting (Stacks Configuration Tab: TAU0 ch5)

For detailed specifications of the TAU0 ch5, refer to the chapter “Timer Array Unit (TAU)” in the User's Manual: Hardware.

Next, in the “Interrupts Configuration” tab below, you can verify the user name of function and event number assigned to the capture interrupt for TAU0 Channel 5.

For detailed specifications regarding interrupts, refer to the chapter “Interrupt Controller Unit (ICU)” in the User’s Manual: Hardware.

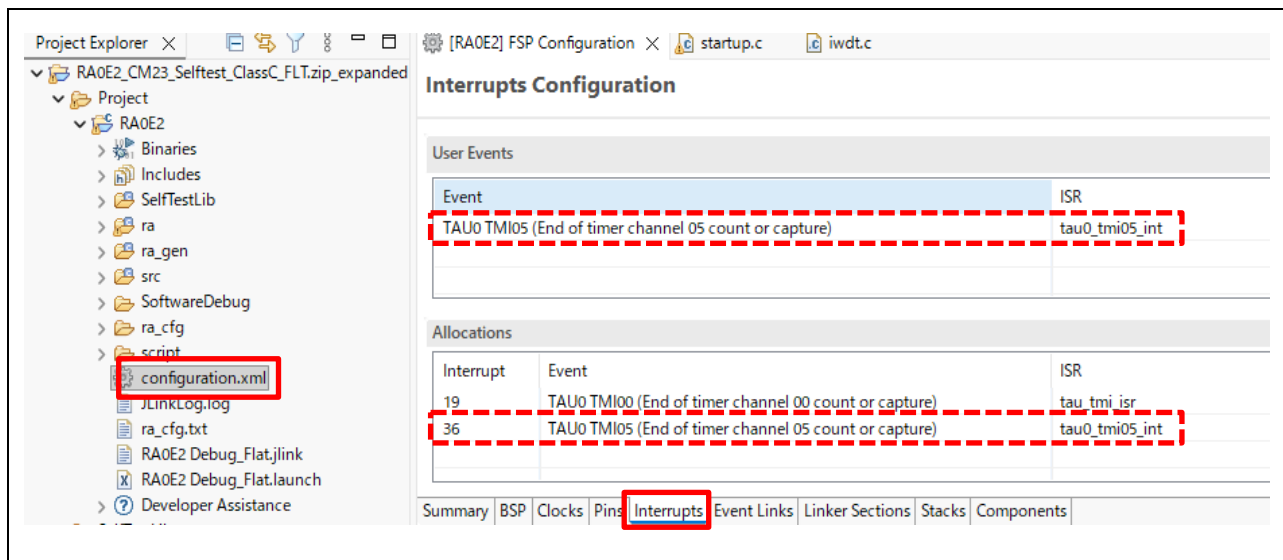


Figure 2.7 FSP Configuration Setting (Interrupts Configuration Tab: TAU0 ch5)

In the sample software, calling the “ClockMonitor\_Init” function reflects the above settings in the relevant registers.

[Example]

```
// TAU0 ch5 initial setting
// In RA0, reflect the Stack setting in FSP Configuration to each TAU0 ch5 related register.
// At the same time, set the error functions.
```

```
ClockMonitor_Init ( Clock_Test_Failure );
```

In the sample project, the FSP Configuration settings are reflected in the TAU0 ch5-related registers within this function.

Next, the clock frequency tolerance range is defined in the following part of "clock\_monitor.h".

In this sample software, "LOCO" is set for the TAU0 ch5 timer input, and "PCLKB" (ICLK was selected to HOCO) is set for the counter operation clock.

If it change the target clock mentioned above, please change the following definitions to match the selected clock.

\*The blue dotted boxes below indicate where the upper and lower limits of the acceptable range are set in this sample software, while the text in green indicates supplementary comments.

[The part of definition in "clock\_monitor.h"]

```
/* Hardware clock test reference values */
/*;hwMAXTIME= HOCO_FREQUENCY_MAX / LOCO_FREQUENCY_MIN */
/*;hwMINTIME= HOCO_FREQUENCY_MIN / LOCO_FREQUENCY_MAX
/*
;HOCO_FREQUENCY_MIN: Value considering -1% error of HOCO frequency (31.68MHz)
;LOCO_FREQUENCY_MAX: Value considering +15% error of low-speed on-chip oscillator frequency (37.68KHz)
;HOCO_FREQUENCY_MAX: Value considering +1% error of HOCO frequency (32.32MHz)
;LOCO_FREQUENCY_MIN: Value considering -15% error of low-speed on-chip oscillator frequency (27.85KHz)
;Change the lower and upper limits of the allowable range for pulse interval measurement as appropriate depending on the system.
*/
// egde trig clock ; LOCO , count clock : PCLKB
```

```
#define hwMAXTIME          (0x0489)    // 1161
#define hwMINTIME          (0x0349)    // 841
```

Also, this sample software uses the "TAU5\_Err\_Detect\_Test" function to verify that the TAU0 ch5 interrupt operates correctly when P-ON starts up.

## 2.5 Independent Watchdog Timer (IWDT)

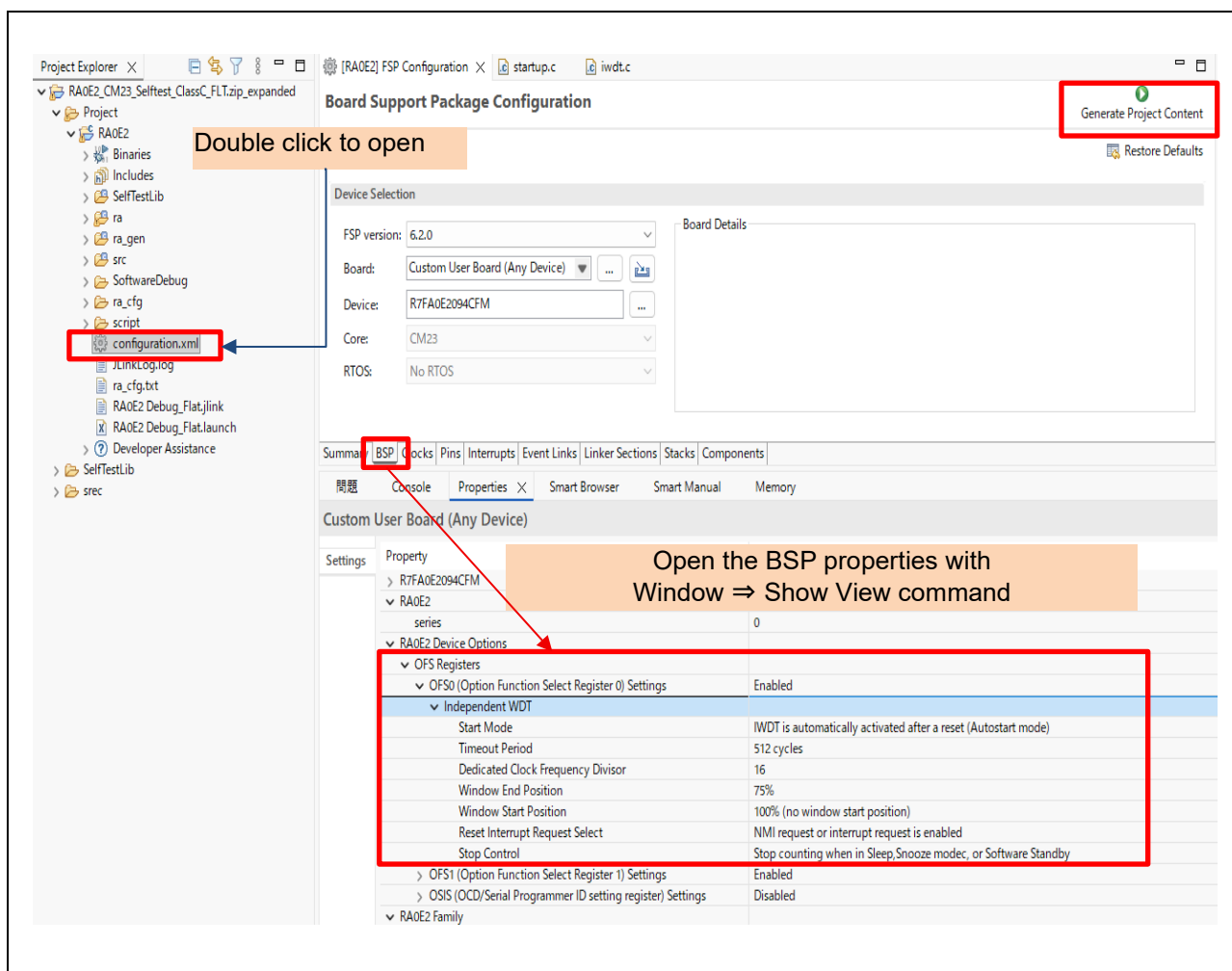
### 2.5.1 OFS0 Register Setting Example (IWDT related)

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 2.1 OFS0 Register Setting Example (IWDT related)**

Item	OFS0 register Setting (example)
IWDT Start Mode Select (IWDTSTRT)	0: Automatically activate IWDT after a reset (auto start mode)
IWDT Timeout Period Select (IWDTTOPS[1:0])	01b : 512 cycles
IWDT-Dedicated Clock Frequency Division Ratio Select (IWDTCKS[3:0])	0010b : 1/16
IWDT Window End Position Select (IWDRPES[1:0])	00b : 75%
IWDT Window Start Position Select (IWDRPSS[1:0])	11b : 100%
IWDT Reset Interrupt Request Select (IWDRSTIRQS)	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<sup>2</sup> 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<sup>2</sup> studio**

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.

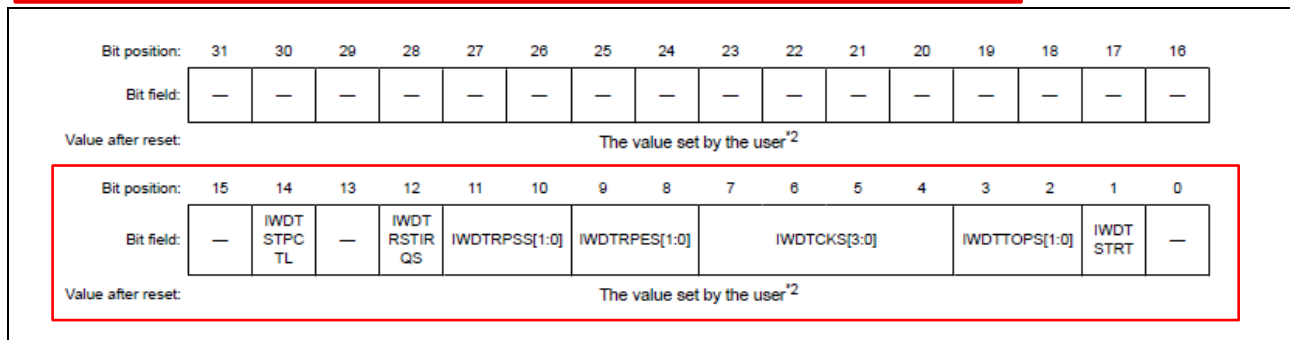
※Please note that the file names and symbols may differ depending on the FSP version used

- Applicable files

..\project-name\ra\_cfg\fsp\_cfg\bsp\bsp\_mcu\_ofs\_cfg.

- Applicable symbol (excerpt)

```
#define OFS_IWDT (0xFFFFA001 | 0 << 1 | 1 << 2 | 2 << 4 | 0 << 8 | 3 << 10 | 0 << 12 | 1 << 14)
#define BSP_CFG_OPTION_SETTING_OFS0 (OFS_IWDT)
```



**Figure 2.9 Option Function Select Register 0 (OFS0)**

For details on the IWDT, refer to the RA MCU User's Manual: Hardware, "Section 20. Independent Watchdog Timer (IWDT)".

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.

### 2.5.2 Example of registering and writing an NMI interrupt callback function

It check whether the IWDT operates normally at Power ON startup on API function "IWDT\_Err\_Detect\_Test()".

For that, user is necessary to prepare the processing that call the function of "IWDT\_Err\_Test\_Judge()" if the cause of the interrupt is an IWDT underflow in the NMI interrupt callback function (NMI\_Handler\_callback).

Users can register callbacks using the BSP API function "R\_BSP\_GroupIrqWrite()" provided by FSP (Flexible Software Package).

By doing this, you can enable notification of one or more group interrupts.

When an NMI interrupt occurs, the NMI handler checks to see if there is a callback registered for the interrupt source, and if so, calls the registered callback function.

For more information, refer to the RA FSP (Flexible Software Package) documentation below.

[Renesas Flexible Software Package \(FSP\) Documentation](#)

Refer to "MCU Board Support Package" – "◆ R\_BSP\_GroupIrqWrite()"

Note: If the watchdog timer is configured to execute a reset (OFS0.IWDTRSTIRQS=1) when an error is detected, do not use API function IWDT\_Err\_Detect\_Test() to check for correct operation.

The following describes the registration and description example of the NMI interrupt callback function (NMI\_Handler\_callback).

© Register NMI interrupt callback function

This is a description example when registering a callback function of the sample software "RA\_SelfTest.c". Please register according to the user's system.

```
for (bsp_grp_irq_t irq = BSP_GRP_IRQ_IWDT_ERROR; irq <= BSP_GRP_IRQ_CACHE_PARITY; irq++){
    R_BSP_GroupIrqWrite(irq , NMI_Handler_callback);
}
```

©Description example of generating an IWDT interrupt factor in the NMI interrupt callback function

```
static void NMI_Handler_callback(bsp_grp_irq_t irq)
{
    /*Read NMISR register to discover NMI cause.*/
    switch(irq){
        case BSP_GRP_IRQ_IWDT_ERROR :
            if( FALSE == IWDT_Err_Test_Judge() )
            {
                Watchdog_Test_Failure();
            }
            break;
        default:
            Error_Detected_Loop(ERROR_NMI_OTHER);
            /*Should not return from an NMI*/
            while(1){;}
    }
}
```

## 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](http://www.renesas.com/ra)
- RA (Flexible Software Package): [www.renesas.com/FSP](http://www.renesas.com/FSP)
- RA Support Forum: [www.renesas.com/ra/forum](http://www.renesas.com/ra/forum)
- Renesas Support: [www.renesas.com/support](http://www.renesas.com/support)

## Reference Document

- (1) Arm® Cortex®-M23 Devices Generic User Guide Revision: r1p0
  - 2.1.4 Core registers
  - Chapter 3: The Cortex-M23 Instruction Set
- (2) Arm®v8-M Architecture Reference Manual
  - D1.1 Register index
  - C2.4 Alphabetical list of instructions

**Revision History**

Rev.	Date	Description	
		Page	Summary
1.00	Mar.26.2026	-	First edition

# 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 be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

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

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

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

(Rev.5.0-1 October 2020)

## Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,  
Koto-ku, Tokyo 135-0061, Japan  
[www.renesas.com](http://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/](http://www.renesas.com/contact/).