RA Family

IEC60730/60335 Self test Library for RA MCU (CM33 Class-C)

R01AN8014EJ0103
Rev.1.03
Oct.29.25

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:
  - Renesas RA Family（Arm® Cortex®-M33）※ See **Table a** for series and groups.
- Development environment:
  - GNU-GCC ARM Embedded Toolchain13.2.1.arm-13-7 / Renesas e2 studio 2025-04.1 (25.4.1)

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®-M33**	**Arm® Cortex®-M33**
	Series	**RA6**	**RA4**
	Group	**RA6M4**	**RA4M3**
Test Function	CPU	○	○
	ROM	○	○
	RAM	○	○
	Clock	○	○
	Independent Watchdog Timer (IWDT)	○	○

About Support for Arm® TrustZone®

This self-test library is assumed to be executed in the "secure area" (hereinafter referred to as "Safety Part") in Arm® TrustZone®.The code of the self-test library is generated by "Flat Project" of RA Project Generator (PG)*.

*: For more information on RA Project Generator, see the RA FSP (Flexible Software Package) documentation.

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.

RA4 and RA6 series (with Arm® Cortex®-M33）self-test library implement functions of the following main self-testing.

- Instruction decoding

  Verify that the corresponding instruction of Arm Cortex-M33 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 "

  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 WALKPAT 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 time-slot and logical monitoring”

Contents

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 instruction test and CPU register 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®-M33(CM33)
Instruction	Profile		ARMv8-M Mainline
	Instruction set		Cortex-M33 Instruction Set
	DSP		SIMD only
	FPU		Single and double precision instructions
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	✓
	Floating-point Status Control Register	FPSCR	✓
	Application Program Status Register	APSR	✓

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	
		CONTROL	Control Register	
		EPSR	Execution Program Status Register	
		FAULTMASK	Fault Mask Register	
		FPSCR	Floating-point Status and Control Register	✓
		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"**.

See the "Arm® Cortex®-M33 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)	*	21	BIC (immediate)	*
2	ADC (register)	✓	22	BIC (register)	✓
3	ADD (SP plus immediate)	✓	23	BKPT	
4	ADD (SP plus register)	*	24	BL	✓
5	ADD (immediate)	*	25	BLX, BLXNS	✓
6	ADD (immediate, to PC)	*	26	BX, BXNS	✓
7	ADD (register)	✓	27	CBNZ, CBZ	✓
8	ADR	✓	28	CDP, CDP2	
9	AND (immediate)	*	29	CINC	N/A
10	AND (register)	✓	30	CINV	N/A
11	ASR (immediate)	✓	31	CLREX	✓
12	ASR (register)	*	32	CLRM	N/A
13	ASRL (immediate)	N/A	33	CLZ	✓
14	ASRL (register)	N/A	34	CMN (immediate)	*
15	ASRS (immediate)	*	35	CMN (register)	✓
16	ASRS (register)	✓	36	CMP (immediate)	*
17	B	✓	37	CMP (register)	✓
18	BF, BFX, BFL, BFLX, BFCSEL	N/A	38	CNEG	N/A
19	BFC	✓	39	CPS	
20	BFI	✓	40	CSDB	N/A

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	CSEL	N/A	71	LDC, LDC2 (literal)	N/A
42	CSET	N/A	72	LDM, LDMIA, LDMFD	✓
43	CSETM	N/A	73	LDMDB, LDMEA	✓
44	CSINC	N/A	74	LDR (immediate)	✓
45	CSINV	N/A	75	LDR (literal)	*
46	CSNEG	N/A	76	LDR (register)	✓
47	CX1	N/A	77	LDRB (immediate)	✓
48	CX1D	N/A	78	LDRB (literal)	*
49	CX2	N/A	79	LDRB (register)	*
50	CX2D	N/A	80	LDRBT	✓
51	CX3	N/A	81	LDRD (immediate)	✓
52	CX3D	N/A	82	LDRD (literal)	*
53	DBG		83	LDREX	✓
54	DMB		84	LDREXB	✓
55	DSB		85	LDREXH	✓
56	EOR (immediate)	*	86	LDRH (immediate)	✓
57	EOR (register)	✓	87	LDRH (literal)	✓
58	ESB	N/A	88	LDRH (register)	*
59	FLDMDBX, FLDMIAX		89	LDRHT	✓
60	FSTMDBX, FSTMIAX		90	LDRSB (immediate)	*
61	ISB		91	LDRSB (literal)	✓
62	IT	✓	92	LDRSB (register)	✓
63	LCTP	N/A	93	LDRSBT	✓
64	LDA	✓	94	LDRSH (immediate)	✓
65	LDAB	✓	95	LDRSH (literal)	*
66	LDAEX	✓	96	LDRSH (register)	✓
67	LDAEXB	✓	97	LDRSHT	✓
68	LDAEXH	✓	98	LDRT	✓
69	LDAH	✓	99	LE, LETP	N/A
70	LDC, LDC2 (immediate)	N/A	100	LSL (immediate)	✓

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	LSL (register)	*	131	PKHBT, PKHTB	✓
102	LSLL (immediate)	N/A	132	PLD (literal)	
103	LSLL (register)	N/A	133	PLD, PLDW (immediate)	
104	LSLS (immediate)	*	134	PLD, PLDW (register)	
105	LSLS (register)	✓	135	PLI (immediate, literal)	
106	LSR (immediate)	✓	136	PLI (register)	
107	LSR (register)	*	137	POP (multiple registers)	✓
108	LSRL (immediate)	N/A	138	POP (single register)	✓
109	LSRS (immediate)	*	139	PSSBB	N/A
110	LSRS (register)	✓	140	PUSH (multiple registers)	✓
111	MCR, MCR2		141	PUSH (single register)	✓
112	MCRR, MCRR2		142	QADD	✓
113	MLA	✓	143	QADD16	✓
114	MLS	✓	144	QADD8	✓
115	MOV (immediate)	✓	145	QASX	✓
116	MOV (register)	*	146	QDADD	✓
117	MOV, MOVS (register-shifted register)	*	147	QDSUB	✓
118	MOVT	✓	148	QSAX	✓
119	MRC, MRC2		149	QSUB	✓
120	MRRC, MRRC2		150	QSUB16	✓
121	MRS	✓	151	QSUB8	✓
122	MSR (register)	✓	152	RBIT	✓
123	MUL	✓	153	REV	✓
124	MVN (immediate)	*	154	REV16	✓
125	MVN (register)	✓	155	REVSH	✓
126	NOP		156	ROR (immediate)	✓
127	ORN (immediate)	*	157	ROR (register)	*
128	ORN (register)	✓	158	RORS (immediate)	*
129	ORR (immediate)	*	159	RORS (register)	✓
130	ORR (register)	✓	160	RRX	✓

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	RRXS	✓	191	SMUAD, SMUADX	✓
162	RSB (immediate)	✓	192	SMULBB, SMULBT, SMULTB, SMULTT	✓
163	RSB (register)	*	193	SMULL	✓
164	SADD16	✓	194	SMULWB, SMULWT	✓
165	SADD8	✓	195	SMUSD, SMUSDX	✓
166	SASX	✓	196	SQRSHR (register)	N/A
167	SBC (immediate)	*	197	SQRSHRL (register)	N/A
168	SBC (register)	✓	198	SQSHL (immediate)	N/A
169	SBFX	✓	199	SQSHLL (immediate)	N/A
170	SDIV	✓	200	SRSHR (immediate)	N/A
171	SEL	✓	201	SRSHRL (immediate)	N/A
172	SEV		202	SSAT	✓
173	SG		203	SSAT16	✓
174	SHADD16	✓	204	SSAX	✓
175	SHADD8	✓	205	SSBB	N/A
176	SHASX	✓	206	SSUB16	✓
177	SHSAX	✓	207	SSUB8	✓
178	SHSUB16	✓	208	STC, STC2	N/A
179	SHSUB8	✓	209	STL	✓
180	SMLABB, SMLABT, SMLATB, SMLATT	✓	210	STLB	✓
181	SMLAD, SMLADX	✓	211	STLEX	✓
182	SMLAL	✓	212	STLEXB	✓
183	SMLALBB, SMLALBT, SMLALTB, SMLALTT	✓	213	STLEXH	✓
184	SMLALD, SMLALDX	✓	214	STLH	✓
185	SMLAWB, SMLAWT	✓	215	STM, STMIA, STMEA	✓
186	SMLSD, SMLSDX	✓	216	STMDB, STMFD	✓
187	SMLSLD, SMLSLDX	✓	217	STR (immediate)	✓
188	SMMLA, SMMLAR	✓	218	STR (register)	✓
189	SMMLS, SMMLSR	✓	219	STRB (immediate)	✓
190	SMMUL, SMMULR	✓	220	STRB (register)	✓

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	STRBT	✓	251	UBFX	✓
222	STRD (immediate)	✓	252	UDF	
223	STREX	✓	253	UDIV	✓
224	STREXB	✓	254	UHADD16	✓
225	STREXH	✓	255	UHADD8	✓
226	STRH (immediate)	✓	256	UHASX	✓
227	STRH (register)	✓	257	UHSAX	✓
228	STRHT	✓	258	UHSUB16	✓
229	STRT	✓	259	UHSUB8	✓
230	SUB (SP minus immediate)	✓	260	UMAAL	✓
231	SUB (SP minus register)	*	261	UMLAL	✓
232	SUB (immediate)	✓	262	UMULL	✓
233	SUB (immediate, from PC)	*	263	UQADD16	✓
234	SUB (register)	*	264	UQADD8	✓
235	SVC		265	UQASX	✓
236	SXTAB	✓	266	UQRSHL (register)	N/A
237	SXTAB16	✓	267	UQRSHLL (register)	N/A
238	SXTAH	✓	268	UQSAX	✓
239	SXTB	✓	269	UQSHL (immediate)	N/A
240	SXTB16	✓	270	UQSHLL (immediate)	N/A
241	SXTH	✓	271	UQSUB16	✓
242	TBB, TBH	✓	272	UQSUB8	✓
243	TEQ (immediate)	*	273	URSHR (immediate)	N/A
244	TEQ (register)	✓	274	URSHRL (immediate)	N/A
245	TST (immediate)	*	275	USAD8	✓
246	TST (register)	✓	276	USADA8	✓
247	TT, TTT, TTA, TTAT		277	USAT	✓
248	UADD16	✓	278	USAT16	✓
249	UADD8	✓	279	USAX	✓
250	UASX	✓	280	USUB16	✓

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	USUB8	✓	301	VAND	N/A
282	UXTAB	✓	302	VBIC (immediate)	N/A
283	UXTAB16	✓	303	VBIC (register)	N/A
284	UXTAH	✓	304	VBRSR	N/A
285	UXTB	✓	305	VCADD (floating-point)	N/A
286	UXTB16	✓	306	VCADD	N/A
287	UXTH	✓	307	VCLS	N/A
288	VABAV	N/A	308	VCLZ	N/A
289	VABD (floating-point)	N/A	309	VCMLA (floating-point)	N/A
290	VABD	N/A	310	VCMP (floating-point)	N/A
291	VABS (floating-point)	N/A	311	VCMP (vector)	N/A
292	VABS (vector)	N/A	312	VCMP	✓
293	VABS	✓	313	VCMPE	✓
294	VADC	N/A	314	VCMUL (floating-point)	N/A
295	VADD (floating-point)	N/A	315	VCTP	N/A
296	VADD (vector)	N/A	316	VCVT (between double-precision and single-precision)	N/A
297	VADD	✓	317	VCVT (between floating-point and fixed-point) (vector)	N/A
298	VADDLV	N/A	318	VCVT (between floating-point and fixed-point)	✓
299	VADDV	N/A	319	VCVT (between floating-point and integer)	N/A
300	VAND (immediate)	N/A	320	VCVT (between single and half-precision floating-point)	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	VCVT (floating-point to integer)	✓	346	VFNMA	✓
322	VCVT (from floating-point to integer)	N/A	347	VFNMS	✓
323	VCVT (integer to floating-point)	✓	348	VHADD	N/A
324	VCVTA	✓	349	VHCADD	N/A
325	VCVTB		350	VHSUB	N/A
326	VCVTM	✓	351	VIDUP, VIWDUP	N/A
327	VCVTN	✓	352	VINS	N/A
328	VCVTP	✓	353	VLD2	N/A
329	VCVTR	✓	354	VLD4	N/A
330	VCVTT		355	VLDM	✓
331	VCX1 (vector)	N/A	356	VLDR (System Register)	N/A
332	VCX1	N/A	357	VLDR	✓
333	VCX2 (vector)	N/A	358	VLDRB, VLDRH, VLDRW	N/A
334	VCX2	N/A	359	VLDRB, VLDRH, VLDRW, VLDRD (vector)	N/A
335	VCX3 (vector)	N/A	360	VLLDM	
336	VCX3	N/A	361	VLSTM	
337	VDDUP, VDWDUP	N/A	362	VMAX, VMAXA	N/A
338	VDIV	✓	363	VMAXNM	✓
339	VDUP	N/A	364	VMAXNM, VMAXNMA (floating-point)	N/A
340	VEOR	N/A	365	VMAXNMV, VMAXNMAV (floating-point)	N/A
341	VFMA (vector by scalar plus vector, floating-point)	N/A	366	VMAXV, VMAXAV	N/A
342	VFMA	✓	367	VMIN, VMINA	N/A
343	VFMA, VFMS (floating-point)	N/A	368	VMINNM	✓
344	VFMAS (vector by vector plus scalar, floating-point)	N/A	369	VMINNM, VMINNMA (floating-point)	N/A
345	VFMS	✓	370	VMINNMV, VMINNMAV (floating-point)	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	VMINV, VMINAV	N/A	386	VMOV (general-purpose register to vector lane)	N/A
372	VMLA (vector by scalar plus vector)	N/A	387	VMOV (half of doubleword register to single general-purpose register)	N/A
373	VMLA	✓	388	VMOV (immediate) (vector)	N/A
374	VMLADAV	N/A	389	VMOV (immediate)	✓
375	VMLALDAV	N/A	390	VMOV (register) (vector)	N/A
376	VMLALV	N/A	391	VMOV (register)	✓
377	VMLAS (vector by vector plus scalar)	N/A	392	VMOV (single general-purpose register to half of doubleword register)	N/A
378	VMLAV	N/A	393	VMOV (two 32-bit vector lanes to two general-purpose registers)	N/A
379	VMLS	✓	394	VMOV (two general-purpose registers to two 32-bit vector lanes)	N/A
380	VMLSDAV	N/A	395	VMOV (vector lane to general-purpose register)	N/A
381	VMLSLDAV	N/A	396	VMOVL	N/A
382	VMOV (between general-purpose register and half-precision register)	N/A	397	VMOVN	N/A
383	VMOV (between general-purpose register and single-precision register)	✓	398	VMOVX	N/A
384	VMOV (between two general-purpose registers and a doubleword register)	N/A	399	VMRS	✓
385	VMOV (between two general-purpose registers and two single-precision registers)	✓	400	VMSR	✓

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	VMUL (floating-point)	N/A	431	VQDMLSDH, VQRDMLSDH	N/A
402	VMUL (vector)	N/A	432	VQDMULH, VQRDMULH	N/A
403	VMUL	✓	433	VQDMULL	N/A
404	VMULH, VRMULH	N/A	434	VQMOVN	N/A
405	VMULL (integer)	N/A	435	VQMOVUN	N/A
406	VMULL (polynomial)	N/A	436	VQNEG	N/A
407	VMVN (immediate)	N/A	437	VQRSHL	N/A
408	VMVN (register)	N/A	438	VQRSHRN	N/A
409	VNEG (floating-point)	N/A	439	VQRSHRUN	N/A
410	VNEG (vector)	N/A	440	VQSHL, VQSHLU	N/A
411	VNEG	✓	441	VQSHRN	N/A
412	VNMLA	✓	442	VQSHRUN	N/A
413	VNMLS	✓	443	VQSUB	N/A
414	VNMUL	✓	444	VREV16	N/A
415	VORN (immediate)	N/A	445	VREV32	N/A
416	VORN	N/A	446	VREV64	N/A
417	VORR (immediate)	N/A	447	VRHADD	N/A
418	VORR	N/A	448	VRINT (floating-point)	N/A
419	VPNOT	N/A	449	VRINTA	✓
420	VPOP	✓	450	VRINTM	✓
421	VPSEL	N/A	451	VRINTN	✓
422	VPST	N/A	452	VRINTP	✓
423	VPT (floating-point)	N/A	453	VRINTR	✓
424	VPT	N/A	454	VRINTX	✓
425	VPUSH	✓	455	VRINTZ	✓
426	VQABS	N/A	456	VRMLALDAVH	N/A
427	VQADD	N/A	457	VRMLALVH	N/A
428	VQDMLADH, VQRDMLADH	N/A	458	VRMLSLDAVH	N/A
429	VQDMLAH, VQRDMLAH (vector by scalar plus vector)	N/A	459	VRSHL	N/A
430	VQDMLASH, VQRDMLASH (vector by vector plus scalar)	N/A	460	VRSHR	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	VRSHRN	N/A	474	VST4	N/A
462	VSBC	N/A	475	VSTM	✓
463	VSCCLRM	N/A	476	VSTR (System Register)	N/A
464	VSEL	✓	477	VSTR	✓
465	VSHL	N/A	478	VSTRB, VSTRH, VSTRW	N/A
466	VSHLC	N/A	479	VSTRB, VSTRH, VSTRW, VSTRD (vector)	N/A
467	VSHLL	N/A	480	VSUB (floating-point)	N/A
468	VSHR	N/A	481	VSUB (vector)	N/A
469	VSHRN	N/A	482	VSUB	✓
470	VSLI	N/A	483	WFE	
471	VSQRT	✓	484	WFI	
472	VSRI	N/A	485	WLS, DLS, WLSTP, DLSTP	N/A
473	VST2	N/A	486	YIELD	

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 CPU Software API Source File

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_1.asm r_cpu_diag_4_2.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 r_cpu_diag_9.asm r_cpu_diag_10.asm r_cpu_diag_11.asm r_cpu_diag_12.asm r_cpu_diag_13.asm r_cpu_diag_14_1.asm r_cpu_diag_14_2.asm r_cpu_diag_15_1.asm r_cpu_diag_15_2.asm r_cpu_diag_15_3.asm r_cpu_diag_15_4.asm r_cpu_diag_15_5.asm r_cpu_diag_15_6.asm r_cpu_diag_16.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_1.h r_cpu_diag_4_2.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 r_cpu_diag_9.h r_cpu_diag_10.h r_cpu_diag_11.h r_cpu_diag_12.h r_cpu_diag_13.h r_cpu_diag_14_1.h r_cpu_diag_14_2.h r_cpu_diag_15_1.h r_cpu_diag_15_2.h r_cpu_diag_15_3.h r_cpu_diag_15_4.h r_cpu_diag_15_5.h r_cpu_diag_15_6.h r_cpu_diag_16.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

Directive Name	Remarks
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_1 *1	When set to “1”, the CPU test function: R_CPU_Diag4_1 is constructed.
BUILD_R_CPU_DIAG_4_2 *1	When set to “1”, the CPU test function: R_CPU_Diag4_2 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.
BUILD_R_CPU_DIAG_9	When set to “1”, the CPU test function: R_CPU_Diag9 is constructed.
BUILD_R_CPU_DIAG_10	When set to “1”, the CPU test function: R_CPU_Diag10 is constructed.
BUILD_R_CPU_DIAG_11	When set to “1”, the CPU test function: R_CPU_Diag11 is constructed.
BUILD_R_CPU_DIAG_12	When set to “1”, the CPU test function: R_CPU_Diag12 is constructed.
BUILD_R_CPU_DIAG_13	When set to “1”, the CPU test function: R_CPU_Diag13 is constructed.
BUILD_R_CPU_DIAG_14_1 *1	When set to “1”, the CPU test function: R_CPU_Diag14_1 is constructed.
BUILD_R_CPU_DIAG_14_2 *1	When set to “1”, the CPU test function: R_CPU_Diag14_2 is constructed.
BUILD_R_CPU_DIAG_15_1 *1	When set to “1”, the CPU test function: R_CPU_Diag15_1 is constructed.
BUILD_R_CPU_DIAG_15_2 *1	When set to “1”, the CPU test function: R_CPU_Diag15_2 is constructed.
BUILD_R_CPU_DIAG_15_3 *1	When set to “1”, the CPU test function: R_CPU_Diag15_3 is constructed.
BUILD_R_CPU_DIAG_15_4 *1	When set to “1”, the CPU test function: R_CPU_Diag15_4 is constructed.
BUILD_R_CPU_DIAG_15_5 *1	When set to “1”, the CPU test function: R_CPU_Diag15_5 is constructed.
BUILD_R_CPU_DIAG_15_6 *1	When set to “1”, the CPU test function: R_CPU_Diag15_6 is constructed.
BUILD_R_CPU_DIAG_16 *1	When set to “1”, the CPU test function: R_CPU_Diag16 is constructed.

Notes: 1. See **Table 1-15**.
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 5	R_CPU_Diag4_1 R_CPU_Diag4_2	Memory access (Load/Store) without exclusive operations
5	6	R_CPU_Diag5	Memory access (Load/Store) with exclusive and privileged operations
6	7	R_CPU_Diag6	System related operations
7	8 9 10	R_CPU_Diag7_1 R_CPU_Diag7_2 R_CPU_Diag7_3	Registers R0 - R12, MSP(R13), LR(R14), and APSR diagnostic operations
8	11	R_CPU_Diag8	Multiply-accumulate and multiply-subtract operations (MAC and MSB)
9	12	R_CPU_Diag9	Combined arithmetic operations
10	13	R_CPU_Diag10	Saturating and rounding operations
11	14	R_CPU_Diag11	Floating-point four basic arithmetic, absolute value and comparison operations
12	15	R_CPU_Diag12	Floating-point multiply-accumulate and multiply-subtract operation
13	16	R_CPU_Diag13	Floating-point rounding and data type conversion
14	17 18	R_CPU_Diag14_1 R_CPU_Diag14_2	Floating-point memory access and data transfer diagnostic operations
15	19 20 21 22 23 24	R_CPU_Diag15_1 R_CPU_Diag15_2 R_CPU_Diag15_3 R_CPU_Diag15_4 R_CPU_Diag15_5 R_CPU_Diag15_6	Registers S0 - S31 and FPSCR
16	25	R_CPU_Diag16	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

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2">void CPU_Test_ClassC(void)</td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">Perform the CPU tests in the following order:
1. Save the current stack limit register.
SaveMspPt = __get_MSPLIM();
SavePspPt = __get_PSPLIM();
2. Disable CPU stack pointer monitoring function
__set_MSPLIM(0);
__set_PSPLIM(0);
3. Pass the parameters and call function R_CPU_Diag
4. Check the value of argument "result"
5. If the result is OK, return to step 3 above. (perform the next test)
When all defined CPU tests are completed, go to step 6 below.
If an error is detected, the external function CPU_Test_ErrorHandler will be called
Check each test for more details.
6. Restore the stack pointer limit register saved in step 1.
7. CPU_Test_PC
8. Exit the function once all tests are performed
If not, all tests performed, the external function CPU_Test_ErrorHandler will be called.</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>const uint32_t forceFail</td><td>Forced FAIL Option
The default value is fixed at "1" (N/A).
* If you want to test the forced FAIL, change the value to fixed at "0".</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2">void CPU_Test_PC(void)</td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">This function tests the program counter (PC) register.
This checks that the PC is working reliably.
The function returns the inverted value of the specified parameter so that it can verify that the function was executed actually. This return value is checked for correctness.
If an error is detected, the external function CPU_Test_ErrorHandler is called.</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

■ 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 **Table 1.16** for the argument index, test number, and test function. 1. Set "resultTemp" to the initial value. When the test function is performed, the test result is saved in "resultTemp". 2. 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. 3. Perform the function of the corresponding CPU test according to the value of the argument “index”. 4. Set the test result to "* result" and exit the function.	
Input Parameters	
uint32_t index	CPU Test No (Refer to **Table 1.16**) 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

Syntax	
static void norm_null(const uint32_t forceFail, int32_t *result)	
Description	
This function is a dummy function of the CPU test function excluded from compilation by the directive. Set the test result to PASS.	
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 (1: PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_0.asm File

Syntax	
void R_CPU_Diag0(const uint32_t forceFail, int32_t *result)	
Description	
1. Addition instructions test Execute each instruction of ADCS(register), ADDS (register), SADD16, SADD8, UADD16, UADD8, SHADD16, SHADD8 and check that the local signature matches the expected global signature **2. Subtraction instructions test** Execute each instruction of SBCS (register), SUBS (immediate), RSBS (immediate), SSUB16, SSUB8, USUB16, USUB8, SHSUB16, SHSUB8 and check that the local signature matches the expected global signature **3. Multiplication instructions test** Execute each instruction of MULS, SMULL, SMULWB, SMMULR, SMULTB, UMULL and check that the local signature matches the expected global signature **4. Division instructions test** Execute each instruction of SDIV, UDIV and check that the local signature matches the expected global signature **5. Addition and subtraction for stack pointer test** Execute each instruction of SUB (SP minus immediate), ADD (SP plus immediate), SUB.W (SP minus immediate), ADD.W (SP plus immediate) and check that the local signature matches the expected global signature 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".	
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	
1. **Sign extension** Execute each instruction of SXTB T1, SXTAB16 T1, SXTAH T1, SXTB T1, SXTB16 T1, SXTH T1 and check that the local signature matches the expected global signature 2. **Zero extension** Execute each instruction of UXTAB T1, UXTAB16 T1, UXTAH T1, UXTB T1, UXTB16 T1, UXTH T1 and check that the local signature matches the expected global signature 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".	
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	
1. **Branch** Execute each instruction of ADR T1, ADR T3, BEQ T1, B T2, BL T1, BLX T1, BX T1, CBZ T1, IT EQ T1, TBB T1, TBH T1 to check that the local signature matches the expected global signature 2. **Logical test** Execute each instruction of TEQ T1, TST T1 and check that the local signature matches the expected global signature 3. **Logical operation** Execute each instruction of ANDS T1, ORRS T1, ORNS T1, EORS T1, MVNS T1 and check that the local signature matches the expected global signature 4. **Comparison** Execute each instruction of CMN T1, CMP T1 and check that the local signature matches the expected global signature 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".	
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	
1. **Bit manipulation** Execute each instruction of ASR (immediate) T3, ASRS (register) T1, BFC T1, BFI T1, BICS (register) T1, LSL (immediate) T3, LSLS (register) T1, LSR (immediate) T3, LSRS (register) T1, ROR (immediate) T3, RORS (register) T1, RRX T3, RRXS T3, CLZ T1, RBIT T1, SBFX T1, UBFX T1 and check that the local signature matches the expected global signature 2. **Data manipulation** Execute each instruction of REV T1, REV16 T1, REVSH T1, SEL T1, PKHBT T1 and check that the local signature matches the expected global signature 3. **Data transfer** Execute each instruction of MOVS (immediate) T1, MOVT T1, MRS T1, MSR (register) T1 and check that the local signature matches the expected global signature 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".	
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_1.asm File

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2">void R_CPU_Diag4_1(const uint32_t forceFail, int32_t *result)</td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">**1. LDR and STR**
Execute each instruction of
LDR (immediate) T2, STR (immediate) T2,
LDR (immediate) T3, STR (immediate) T3,
LDR (immediate) T4, STR (immediate) T4, (post-indexed),
LDR (immediate) T4, STR (immediate) T4, (negative immediate),
LDR (immediate) T4, STR (immediate) T4, (pre-indexed),
LDR (register) T2, STR (register) T2
and check that the local signature matches the expected global signature

2. LDRH and STRH
Execute each instruction of
LDRH (immediate) T1, STRH (immediate) T1,
LDRSH (register) T1, STRH (register) T1,
LDRSH (immediate) T1, STRH (immediate) T2,
LDRSH (immediate) T2, STRH (immediate) T3, (post-indexed),
LDRSH (immediate) T2, STRH (immediate) T3, (negative immediate),
LDRSH (immediate) T2, STRH (immediate) T3, (pre-indexed),
LDRSH (register) T2, STRH (register) T2
and check that the local signature matches the expected global signature

3. LDRB and STRB
Execute each instruction of
LDRSB (register) T1, STRB (register) T1 ,
LDRB (immediate) T1, STRB (immediate) T1
and check that the local signature matches the expected global signature

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".</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td>const uint32_t forceFail</td><td>Forced FAIL Option
When set to 0, the function fails forcibly.
0 : Enabled
Others : Disabled</td></tr>
<tr><td>int32_t *result</td><td>Pointer to store Test result</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>int32_t *result</td><td>Test result (0: FAIL / 1: PASS)</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

■ r_cpu_diag_4_2.asm File

Syntax	
void R_CPU_Diag4_2(const uint32_t forceFail, int32_t *result)	
Description	
4. LDRD and STRD Execute each instruction of LDRD (immediate) T1, STRD (immediate) T1, (post-indexed), LDRD (immediate) T1, STRD (immediate) T1, (immediate), LDRD (immediate) T1, STRD (immediate) T1, (pre-indexed) and check that the local signature matches the expected global signature **5. LDM and STM** Execute each instruction of LDM and STM, LDM T3, STMDB T2, LDM T2, STM T2, LDMDB T1, STM T2 and check that the local signature matches the expected global signature **6. LDA and STL** Execute each instruction of LDA T1, STL T1, LDAH T1, STLH T1, LDAB T1, STLB T1 and check that the local signature matches the expected global signature **7. LDRH / LDRSB (literal)** Execute each instruction of LDRH (literal) T1, LDRSB (literal) T1 and check that the local signature matches the expected global signature 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".	
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	
1. LDAEX and STLEX Execute each instruction of LDAEX T1, STLEX T1, LDAEXH T1, STLEXH T1, LDAEXB T1, STLEXB T1 and check that the local signature matches the expected global signature **2. LDREX and STREX** Execute each instruction of LDREX T1, STREX T1, LDREXH T1, STREXH T1, LDREXB T1, STREXB T1 and check that the local signature matches the expected global signature **3. LDRT and STRT** Execute each instruction of LDRT T1, STRT T1, LDRHT T1, STRHT T1, LDRSHT T1, STRHT T1, LDRBT T1, STRBT T1, LDRSBT T1, STRBT T1 and check that the local signature matches the expected global signature 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".	
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	
1 **PUSH and POP** After executing the PUSH instruction using R4, R5, R6, R7, R8, R9, execute POP instruction and check that the values in each pair of registers R4&R7, R5&R8, R6&R9 match the expected values. 2 **Other (miscellaneous) operations** Execute each instruction of CLREX T1 and check that the local signature matches the expected global signature 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".	
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	
1. **Detecting “0” fixed fault for status and control registers** 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") 2 **Detecting “1” fixed fault for status and control registers** 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) 3 **Detecting “0” fixed fault for general purpose registers** 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") 4 **Detecting “1” fixed fault for general purpose registers** 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) 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".	
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

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2">void R_CPU_Diag7_2(const uint32_t forceFail, int32_t *result)</td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">5 Detecting coupling fault for general purpose registers between any two bits
Perform the following tests for the R0^R12 and R14 registers
—Nearest neighbor coupling(Test pattern : 0x55555555)
—Next nearest neighbor coupling(Test pattern : 0x33333333)
—4-fold neighbor coupling(Test pattern : 0x0f0f0f0f)
—8-fold neighbor coupling(Test pattern : 0x00ff00ff)
—16-fold neighbor coupling(Test pattern : 0x0000ffff)

1. The procedure is as follows
2. Set each of the above test patterns to R0, write to R1, and check if it matches R0.
3. If they match, change the register written in 1 above in the order of R2 to R14 and perform.
4. Set each of the above test patterns to R14, write to R0, and confirm that it matches R0.
5. If they match, perform the following test pattern.
6. When all is completed, move to the following test

6. Detecting coupling fault for general purpose registers between any two registers
—Detecting R7, R8, R9, R10, R11, R12, LR(R14) coupling fault (Using A's pattern)
—Detecting R0, R1, R2, R3, R4, R5, R6 coupling fault (Using B's pattern)
The procedure is as follows.
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.
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.
3. Complete the test.

R13 (SP) is excluded from this test.
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".</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td>const uint32_t forceFail</td><td>Forced FAIL Option
When set to 0, the function fails forcibly.
0 : Enabled
Others : Disabled</td></tr>
<tr><td>int32_t *result</td><td>Pointer to store Test result</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>int32_t *result</td><td>Test result (0: FAIL / 1: PASS)</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

■ r_cpu_diag_7_3.asm File

Syntax	
void R_CPU_Diag7_3(const uint32_t forceFail, int32_t *result)	
Description	
7. Detecting "0" fixed fault for MSP(R13) After writing "0xfffffffc" 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") **8. Detecting "1" fixed fault for MSP(R13)** 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") **9. Detecting coupling fault for MSP(R13) between any two bits** 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) The procedure is as follows. 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 **10. Detecting coupling fault between MSP(R13) to other general-purpose registers** ―Detect coupling error of SP(R13), R2 ―Detect coupling error of SP(R13), R3 The procedure is as follows. 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 matches. 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. 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".	
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_8.asm File

Syntax	
void R_CPU_Diag8(const uint32_t forceFail, int32_t *result)	
Description	
1. **Multiply accumulate (MAC)** Execute each instruction of MLA T1, SMLAL T1, SMLALBB T1, SMLALD T1, UMAAL T1, UMLAL T1, SMMLA T1, SMLADX T1, SMLATT T1, SMLAWB T1 and check that the local signature matches the expected global signature 2. **Multiply subtract (MSB)** Execute each instruction of MLS T1, SMLSLD T1, SMMLSR T1, SMLSD T1 and check that the local signature matches the expected global signature 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".	
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_9.asm File

Syntax	
void R_CPU_Diag9(const uint32_t forceFail, int32_t *result)	
Description	
1 **Addition and subtraction** Execute each instruction of SASX T1, SSAX T1, UASX T1, USAX T1 and check that the local signature matches the expected global signature 2 **Addition and halving** Execute each instruction of UHADD16 T1, UHADD8 T1 and check that the local signature matches the expected global signature 3 **Subtraction and halving** Execute each instruction of UHSUB16 T1, UHSUB8 T1 and check that the local signature matches the expected global signature 4 **Addition, subtraction and halving** Execute each instruction of SHASX T1, SHSAX T1, UHASX T1, UHSAX T1 and check that the local signature matches the expected global signature 5 **Dual multiplication** Execute each instruction of SMUAD T1, SMUSDX T1 and check that the local signature matches the expected global signature 6 **Absolute difference** Execute each instruction of USAD8 T1, USADA8 T1 and check that the local signature matches the expected global signature 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".	
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_10.asm File

Syntax	
void R_CPU_Diag10(const uint32_t forceFail, int32_t *result)	
Description	
1 **Saturating** Execute each instruction of SSAT T1, SSAT16 T1, USAT T1, USAT16 T1 and check that the local signature matches the expected global signature 2 **Saturate addition** Execute each instruction of QADD T1, QADD16 T1, QADD8 T1, UQADD16 T1, UQADD8 T1, QDADD T1 and check that the local signature matches the expected global signature 3 **Saturate subtraction** Execute each instruction of QSUB T1, QSUB16 T1, QSUB8 T1, QDSUB T1, UQSUB16 T1, UQSUB8 T1 and check that the local signature matches the expected global signature 4 **Saturate addition and subtraction** Execute each instruction of QASX T1, QSAX T1, UQASX T1, UQSAX T1 and check that the local signature matches the expected global signature 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".	
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_11.asm File

Syntax	
void R_CPU_Diag11(const uint32_t forceFail, int32_t *result)	
Description	
1 **Four basic arithmetic instructions test** Execute each instruction of VADD T2, VSUB T2, VMUL T2, VNMUL T2, VDIV T1 and check that the local signature matches the expected global signature 2. **Absolute, compare, negative, minimum and maximum instructions test** Execute each instruction of VABS T2, VCMP T1, VCMPE T1, VNEG T2, VMAXNM T2, VMINNM T2 and check that the local signature matches the expected global signature 3. **Conditional select instructions test** Execute each instruction of VSELGE T1, VSELGT T1, VSELEQ T1, VSELVS T1 and check that the local signature matches the expected global signature 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".	
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_12.asm File

Syntax	
void R_CPU_Diag12(const uint32_t forceFail, int32_t *result)	
Description	
1. **Multiply accumulate (MAC)** Execute each instruction of VMLA T2, VNMLA T1, VFMA T2, VFNMA T1 and check that the local signature matches the expected global signature 2 **Multiply subtract (MSB)** Execute each instruction of VMLS T2, VNMLS T1, VFMS T2, VFNMS T1 and check that the local signature matches the expected global signature 3 **Square root** Execute each instruction of VSQRT (minus) T1, VSQRT (zero) T1, VSQRT (plus) T1 and check that the local signature matches the expected global signature. 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".	
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_13.asm File

Syntax	
void R_CPU_Diag13(const uint32_t forceFail, int32_t *result)	
Description	
1. **Floating-point rounding** Execute each instruction of VRINTA T1, VRINTM T1, VRINTN T1, VRINTP T1, VRINTR (RN mode) T1, VRINTR (RP mode) T1 VRINTR (RM mode) T1, VRINTR (RZ mode) T1, VRINTX T1, VRINTZ T1 and check that the local signature matches the expected global signature 2 **Floating-point conversion** Execute each instruction of VCVT (between float and fix) F32 to S32, T1 <fbits = 31>, VCVT (between float and fix) F32 to U32, T1<fbits = 16>, VCVT (between float and fix) S32 to F32, T1<fbits = 24>, VCVT (between float and fix) U32 to F32, T1<fbits = 8>, VCVT (float to int) F32 to S32, T1, VCVT (float to int) F32 to U32, T1, VCVT (int to float), T1, VCVTA T1, VCVTM T1, VCVTN T1, VCVTP T1, VCVTP T1 and check that the local signature matches the expected global signature. 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".	
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_14_1.asm File

Syntax

void R_CPU_Diag14_1(const uint32_t forceFail, int32_t *result)

Description

1. VPOP T2 and VPUSH T2

Perform the following test:

- Verify VPOP after VPUSH using single register

The procedure is as follows:

1. Set values to register R4, R5, then write data to registers S1, S0
2. Use VPUSH instruction to push the register S1 onto the stack
3. Use VPOP instruction to pop the register S0 from the stack
4. Verify that the values in register S0 and S1 match the expected value via R5 and R4

—Verify VPOP after VPUSH using multiple registers

The procedure are as follows

1. Set data to S4~S7 and S0~S4
2. Use VPUSH instruction to push registers S4~S7 onto the stack
3. Use VPOP instruction to restore registers S0~S4 from the stack
4. Verify that the expected values match for each registers pair via R4-R7: S0 and S4, S1 and S5, S2 and S6, S3 and S7

2. VLDR/VLDM T2 and VSTR/VSTM T2

Perform the following test:

—Verify VLDR after VSTR using single register

The procedure is as follows:

1. Write data to register S1, S0
2. Store register S1 to the stack using VSTR instruction
3. Load register S0 from the stack using VLDR instruction
4. Verify that the values in S0 and S1 registers match the expected values via R4, R5

—Verify VLDM after VSTM using multiple registers

The procedure is as follows:

1. Set data to S4~S7 and S0~S4
2. Store S4~S7 registers to the stack using VSTM instruction
3. Load the stack into registers S0~S4 using VLDR instruction
4. Verify the register values matches expected values for pairs via R4-R7: S0 and S4, S1 and S5, S2 and S6, S3 and S7

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".

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_14_2.asm File

Syntax

void R_CPU_Diag14_2(const uint32_t forceFail, int32_t *result)

Description

3. VMOV

The following tests are performed

—VMOV (general-purpose register to single-precision register)

The procedure is as follows

1. Set data to S0, R4 respectively
2. Execute "VMOV S0, R4"
3. Verify the match with the expected value in each register S0 and R4 via R5

—VMOV (single-precision register to general-purpose register)

The procedure is as follows

1. Set data to S0(=R5), R4 respectively
2. Execute "VMOV R4, S0"
3. Verify the match with the expected value in each register S0 and R4 via R5

—VMOV (two general-purpose register to two single-precision register)

The procedure is as follows

1. Set data to S0, S1, R5, R4 respectively
2. Execute "VMOV S0, S1, R4, R5"
3. Verify the match with the expected value in each register S0 and R4, S1 and R5 via R6

—VMOV (two single-precision register to two general-purpose register)

The procedure is as follows

1. Set data to S0(=R6), S1(=R7)
2. Execute "VMOV R4, R5, S0, S1"
3. Verify the match with the expected value in each register S0 and R4, S1 and R5 via R6, R7

—VMOV (an immediate constant into the destination floating-point register)

The procedure is as follows

1. Set data to S0(=R6) and R4 respectively (set R4 to the expected value from step 2 below: the floating-point format of #9
2. Execute "VMOV.F32 S0, #9"
3. Verify the match with the expected value in each register S0 and R4 via R5

—VMOV (a single-precision register to another single-precision register)

The procedure is as follows

1. Set data to S0(=R6), S1(=R4)
2. Execute "VMOV.F32 S0, S1"
3. Verify the match with the expected value in each register S0 and S1 via R5, R4

4 VMRS

The following test are performed

—VMRS (FPSCR to general-purpose register with {FPSCR N, Z, C, V} = {1, 1, 1, 1})

The procedure is as follows

1. Set the data to R4, R5(=FPSCR) respectively
   (The setting values are {FPSCR N, Z, C, V} = {1, 1, 1, 1})
2. Execute “VMRS R4, FPSCR”
3. Verify the values in R5 and FPSCR registers match the expected values via R4, R5

—VMRS (FPSCR to general-purpose register with {FPSCR N, Z, C, V} = {0, 0, 0, 0})

The procedure is as follows

1. Set data to R4 and R5(=FPSCR) respectively
   (The setting values are {FPSCR N, Z, C, V} = {0, 0, 0, 0})
2. Execute "VMRS R4, FPSCR"
3. Verify that the values in R5 and FPSCR registers match the expected values via R4, R5

－VMRS (FPSCR to APSR with {FPSCR N, Z, C, V} = {1, 1, 1, 1})
The procedure is as follows

1. Set data to R4(=APSR), R5(=FPSCR) respectively
   (The setting values are {FPSCR N, Z, C, V} = {1, 1, 1, 1})
2. Set "VMRS APSR_nzcv, FPSCR"
3. Verify the match with expected values in registers APSR and FPSCR via R4 and R5
   *Verify the values of N, Z, C, V flags for registers APSR and FPSC

－VMRS (FPSCR to APSR with {FPSCR N, Z, C, V} = {0, 0, 0, 0})
The procedure is as follows

1. Set data to R4(=APSR), R5(=FPSCR) respectively
   (The setting values are {FPSCR N, Z, C, V} = {0, 0, 0, 0})
2. Execute "VMRS APSR_nzcv, FPSCR"
3. Verify that the values in register APSR and FPSCR match the expected values via R4, R5
   *Verify the values of N, Z, C, V flags in APSR and FPSCR

5. VMSR

The following tests are performed

－VMSR (general-purpose register to FPSCR with {APSR N, Z, C, V} = {1, 1, 1, 1})
The procedure is as follows

1. Set data to R5(= FPSCR), R4 respectively
   (The setting values are {FPSCR N, Z, C, V} = {1, 1, 1, 1})
2. Execute "VMSR FPSCR, R4"
3. Verify that R5 and R4 matches via R5 and R4
   *Verify the values of N, Z, C, V flags of FPSCR

－VMSR (general-purpose register to FPSCR with {FPSCR N, Z, C, V} = {0, 0, 0, 0})
The procedure is as follows

1. Set data to R5(= FPSCR), R4 respectively
   (The setting values are {FPSCR N, Z, C, V} = {0, 0, 0, 0})
2. Execute "VMSR FPSCR, R4"
3. Compare R5 and R4 and verify the R5 matches R4
   *Verify the values of N, Z, C, V flags of FPSCR

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".

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_15_1.asm File

<table>
<tr><td colspan="2">Syntax</td></tr>
<tr><td colspan="2">void R_CPU_Diag15_1(const uint32_t forceFail, int32_t *result)</td></tr>
<tr><td colspan="2">Description</td></tr>
<tr><td colspan="2">1. Detecting “0” fixed fault for FPU status and control registers
After writing "1" to the corresponding bit in FPSCR register using R7 and R8 (0xf7c0009f), execute a read operation and verify that it matches the expected value (Confirm that it is not fixed at "0")
2. Detecting “1” fixed fault for FPU status and control registers
After writing "0" to the corresponding bit in FPSCR register using R7 and R8 (0x00000000), execute a read operation and verify that it matches the expected value (Confirm that it is not fixed at "1")
3. Detecting “0” fixed fault for FPU data registers
After writing "0xffffffff" to each register (S0-S1) using R7 and R8, execute a read operation and verify that the values match the expected values (Confirm that the values are not fixed at "0")
4. Detecting “1” fixed fault for FPU data registers
After writing "0x00000000" to each register (S0-S1) using R7 and R8, execute a read operation and verify that the values match the expected values (Confirm that the values are not fixed at "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".</td></tr>
<tr><td colspan="2">Input Parameters</td></tr>
<tr><td>const uint32_t forceFail</td><td>Forced FAIL Option
When set to 0, the function fails forcibly.
0 : Enabled
Others : Disabled</td></tr>
<tr><td>int32_t *result</td><td>Pointer to store Test result</td></tr>
<tr><td colspan="2">Output Parameters</td></tr>
<tr><td>int32_t *result</td><td>Test result (0: FAIL / 1: PASS)</td></tr>
<tr><td colspan="2">Return Values</td></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

■ r_cpu_diag_15_2.asm File

Syntax	
void R_CPU_Diag15_2(const uint32_t forceFail, int32_t *result)	
Description	
5. Detecting coupling fault for single-precision registers between any two bits The following tests will be performed —Nearest neighbor coupling(Test pattern : 0x55555555) —Next nearest neighbor coupling(Test pattern : 0x33333333) The procedure is as follows: 1. Set each test pattern to R7 as above 2. Using R7, R8, write the test pattern to each register of the single-precision register (S0-S31), then read them out 3. Confirm that the values in each register of R7 and R8 match the expected value. 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".	
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_15_3.asm File

Syntax	
void R_CPU_Diag15_3(const uint32_t forceFail, int32_t *result)	
Description	
6. Detecting coupling fault for general purpose registers between any two bits The following tests will be performed —4-fold neighbor coupling(Test pattern : 0x0f0f0f0f) —8-fold neighbor coupling(Test pattern : 0x00ff00ff) The procedure is as follows: 1. Set each test pattern to R7 as above 2. Using R7, R8, write the test pattern to each register of the single-precision register (S0-S31), then read them out 3. Confirm that the values in each register of R7 and R8 match the expected value. 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".	
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_15_4.asm ファイル

Syntax	
void R_CPU_Diag15_4(const uint32_t forceFail, int32_t *result)	
Description	
7. Detecting coupling fault for general purpose registers between any two bits The following tests will be performed －16-fold neighbor coupling(Test pattern : 0x0000ffff) The procedure is as follows: 1. Set each test pattern to R7 as above 2. Using R7, R8, write the test pattern to each register of the single-precision register (S0-S31), then read them out 3. Confirm that the values in each register of R7 and R8 match the expected value. 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".	
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_15_5.asm File

Syntax

void R_CPU_Diag15_5(const uint32_t forceFail, int32_t *result)

Description

8. Detecting coupling fault for FPU data registers between any two registers

The procedure is as follows:

－Detecting S16, S17, S18, S19, S20, S21, S22, S23 coupling fault (Using A's pattern)

[A’s pattern]

R4 = 0x55555555
R5 = 0xAAAAAAAA
R6 = 0x00000000
R7 = 0xFFFFFFFF
R8 = 0x33333333
R9 = 0xCCCCCCCC
R10 = 0x5555AAAA
R11 = 0xAAAA5555

The procedure is as follows:

1. Set each test pattern for R4～R11, transfer R4 to S0, R5 to S1, ..., R11 to S7
2. Transfer S0 to S16, S1 to S17, ..., S7 to S23
3. Read S16~S23 via R12 and verify that the values match the expected values for R4～R11

－Detecting S24, S25, S26, S27, S28, S29, S30, S31 coupling fault(Using B's pattern)

[B’s pattern]

R4 = 0xFFFF0000
R5 = 0x0000FFFF
R6 = 0x3333CCCC
R7 = 0xCCCC3333
R8 = 0xFFAA5533
R9 = 0x3355AAFF
R10 = 0xFEDCBA98
R11 = 0x76543210

The procedure is as follows:

1. Set each test pattern for R4～R11, transfer R4 to S9, R5 to S10, ..., R11 to S18
2. Transfer S9 to S24, S10 to S25, ..., S8 to S31
3. Read S24～S31 via R12 and verify that the values match the expected values for R4～R11

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".

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_15_6.asm File

Syntax

void R_CPU_Diag15_6(const uint32_t forceFail, int32_t *result)

Description

9. Detecting coupling fault for FPU data registers between any two registers

The following tests will be performed

－Detecting S0, S1, S2, S3, S4, S5, S6, S7 coupling fault (Using C's pattern)

[C’s pattern]

R4 = 0x44444444
R5 = 0x99999999
R6 = 0x00000000
R7 = 0xFFFFFFFF
R8 = 0x22222222
R9 = 0xBBBBBBBB
R10 = 0x4444BBBB
R11 = 0xBBBB4444

The procedure is as follows:

1. Set each test pattern for R4～R11, transfer R4 to S18, ..., R9 to S23, R10 to S16,R11 to S17
2. Transfer S18 to S0, ..., S23 to S5, S16 to S6, S17 to S7
3. Read S0～S7 via R12 and verify that the values match the expected values for R4～R11

－Detecting S8, S9, S10, S11, S12, S13, S14, S15 coupling fault(Using D's pattern)

[D’s pattern]

R4 = 0xEEEE1111
R5 = 0x1111EEEE
R6 = 0x2222DDDD
R7 = 0xDDDD2222
R8 = 0xEEBB6622
R9 = 0x2266BBEE
R10 = 0xBA98FEDC
R11 = 0x32107654

The procedure is as follows

1. Set each test pattern for R4～R11, transfer R4 to S27, ..., R8 to S31, R9 to S24, R10 to S25, R11 to S26
2. Transfer S27 to S8, ..., S31 to S12, S24 to S13, S25 to S14, S26 to S15
3. Read S8～S15 via R12 and verify that the value match the expected values for R4～R11

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".

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_16.asm File

Syntax	
void R_CPU_Diag16(const uint32_t forceFail, int32_t *result)	
Description	
Perform CPU register test using WALKPAT algorithm on General-purpose registers (R0-12, R14). (For details on the algorithm, refer to Section **1.3.3(2)** WALKPAT) Save the test results in “resultTemp”. (0: FAIL /1: PASS) The test patterns used are as follows ◆ Test Patterns 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)	
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 **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 bit
- Initial Value = 0xFFFFFFFF
- XOR with h'FFFFFFFF is performed on the output CRC

1.2.2 Multi Checksum

In the ROM test, the ROM area to be tested is divided into 128K 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 1MB, it is stored at addresses 0xFFFE0 to 0xFFFFF when building.

In addition, the self-test library divides the process into 128Kbytes 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.

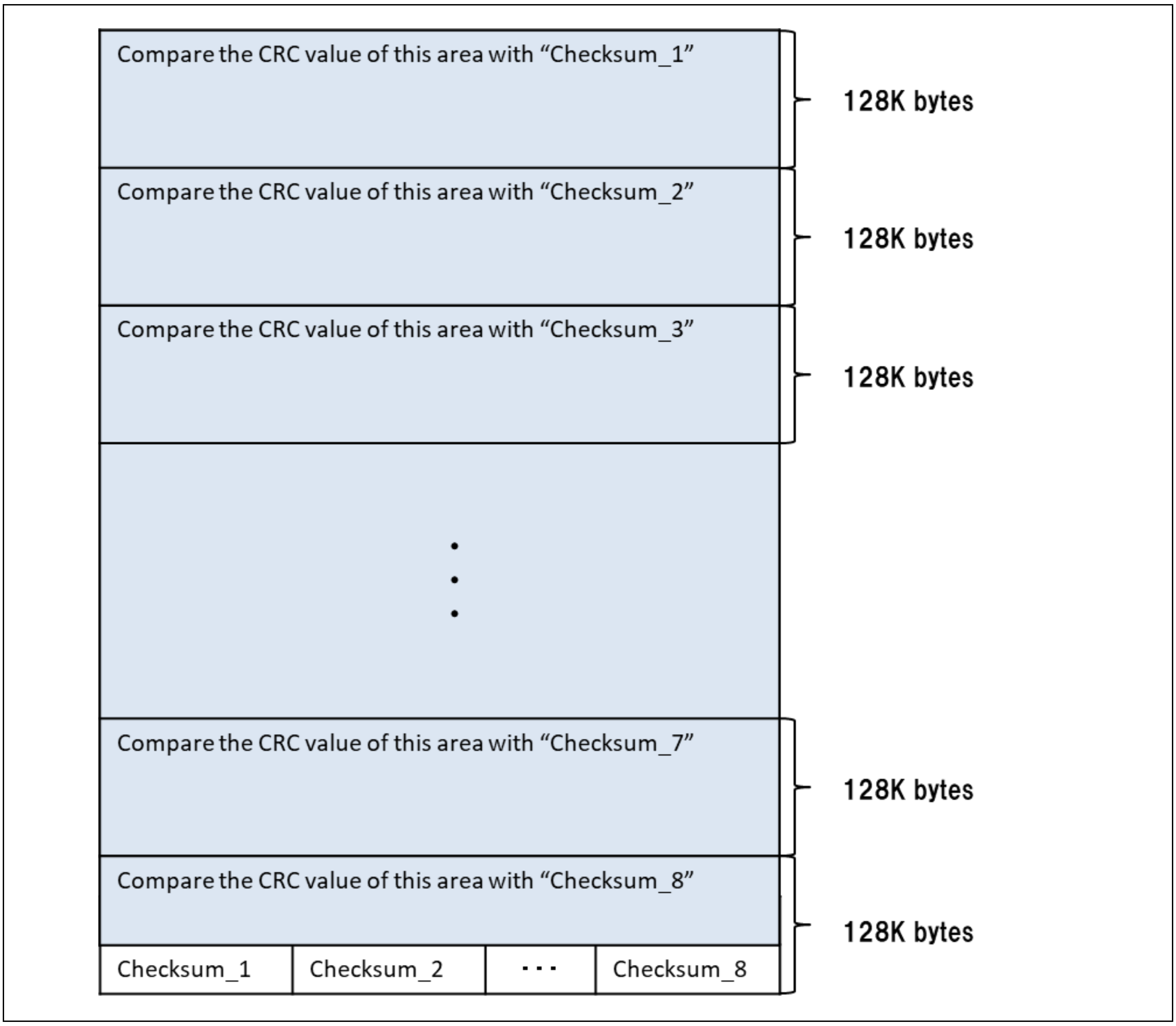


Figure 1-1 Code FLASH block diagram on ROM test

1.2.3 CRC Software API

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

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

Table 1-17 CRC Software API Source File

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

Syntax	
bool_t CRC_Verify(const uint32_t ui32_NewCRCValue, const uint32_t ui32_AddrRefCRC)	
Description	
This function compares a new CRC value with a reference CRC by supplying address where reference CRC is stored.	
Input Parameters	
const uint32_t ui32_NewCRCValue	Value of calculated new CRC value.
const uint32_t ui32_AddrRefCRC	Address where 32-bit reference CRC value is stored.
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : True = Passed, 0 : False = Failed

■ crc.c File

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

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

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

■ crc.c File

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

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.

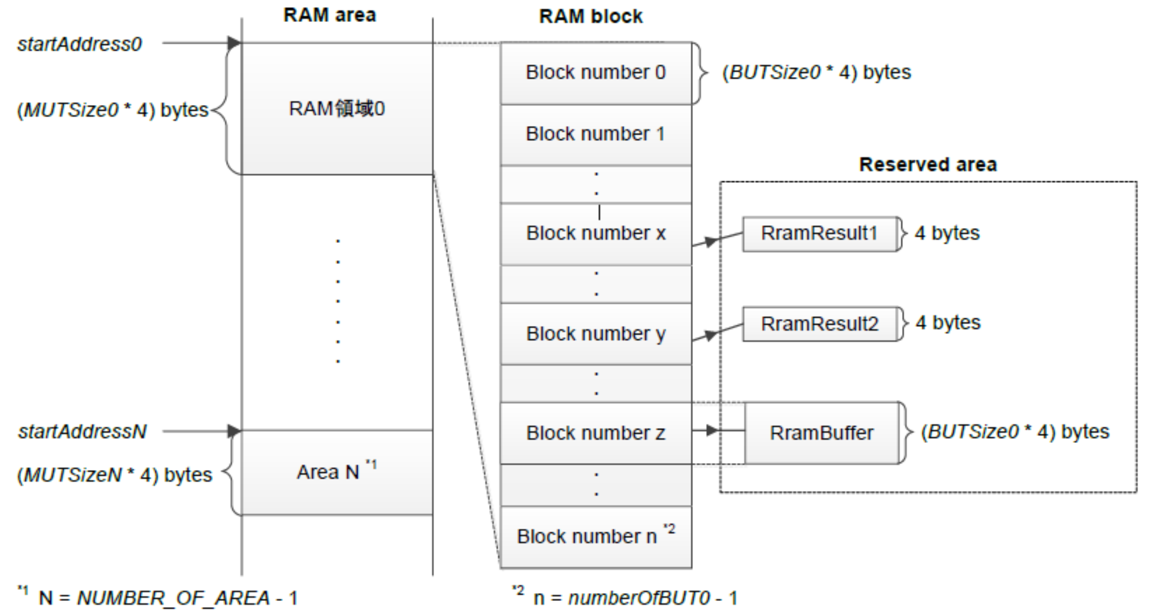


Figure 1-2 RAM Block Configuration (example)

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.

Specially, the files “fsp_ra6m4.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.

◆ Relavant definition section (highlighted in blue) within the “fsp_ra6m4.ld(or fsp_ra4m3.ld)”

The specific modification procedure involves duplicating the automatically generated “fsp_gen.ld” file, adding the above highlighted blue section and saving it as either “fsp_ra6m4.ld” or “fsp_ra4m3.ld”.

```
/***** 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)

```
/*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.*/
/*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 building.

◆ Applicable parts for the generated MAP File (“RA6M4.map(or RA4M3.map)”)

```
.ram_test_flash$$
      0x20000000      0x318 load address 0x0000004c
      0x20000000                __ram_from_flash$$Base = .
      0x0000004c                __ram_from_flash$$Load = LOADADDR (__ram_from_flash$$)
      0x20000000                . = ALIGN (0x4)
      0x20000000                __RramBuffer_start = .
 *(RAM_TEST_BUFFER*)
 RAM_TEST_BUFFER
      0x20000000      0x300 ./SelfTestLib/src/RA_SelfTests.o
      0x20000000                RramBuffer
      0x20000100                RAM_Test_dummy1
      0x200001fc                RramResult1
      0x20000200                RAM_Test_dummy2
      0x200002fc                RramResult2
      0x20000300                __RramBuffer_stop = .
```

RramBuffer → RAM Buffer for temporarilly saved data : RamBuffer[]

RramResult1 → result variables : RramResult1

RramResult2 → result variables : RramResult2

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.

{⇕(w0);⇑(r0,w1,r1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇕(r0)}

Notation		
	{}: Sequence	⇑ : increasing addressing
	() : March element	⇓ : decreasing addressing
	wx : write x	⇕ : either ⇑ or ⇓
	rx : read x	

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	$2n^2$

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 (refer to **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	
r_ram_diag_config.h	Definition of RAM Test Directive.
r_ram_diag_config.inc	Definition of RAM Test execution pattern.
r_ram_diag.c	Definition of RAM Test API function.
r_ram_diag.h	Declaration of RAM Test API function.
r_ram_marchc.asm	Definition of Extended March C- algorithm function.
r_ram_marchc.h	Declaration of Extended March C- algorithm function.
r_ram_walpat.asm	Definition of WALKPAT algorithm function.
r_ram_walpat.h	Declaration of WALKPAT algorithm function.

Table 1-20 Directives for Software Configuration for RAM Test

Directive Name	
NUMBER_OF_AREA	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
startAddressN [*1]	Start address to the RAM area under test
MUTSizeN [*1]	Size of RAM area under test (N) in double word.
numberOfBUTN [*1]	Number of RAM blocks under test.
BUTSizeN [*1]	Size of RAM block under test (N) in double word. Calculated by BUTSizeN = MUTSizeN / numberOfBUTN
RAM_BUFFER_SIZE	Size of buffer (RramBuffer) under test in double word.

*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

This function verifies RAM.

Test results can be confirmed via the return values in the result variables (RramResult1, RramResult2).

If Test result is PASS :
RramResult1 = 1 and RramResult2 = 1

If Test result is FAIL :
Other than above

Perform the RAM tests in the following order:

1. Check if the RAM block is a valid area by the arguments "area" and "index".
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 and end addresses are saved in sAdr, eAdr)
3. The function of the corresponding algorithm is called by the argument “algorithm”
   ・Extended March C (algorithm = RAM_ALG_MARCHC): function R_RAM_Diag_MarchC()
   ・WALKPAT (algorithm = RAM_ALG_WALPAT): function R_RAM_Diag_Walpat()
   Note:
   The argument “destructive” selects whether the data is destructive or non-destructive
   (In the case of the destructive test, the RAM block is cleared to “0” after the test)
4. Return to the called function

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 should be smaller than the directive numberOfBUTN. (See **Table 1-20**) Return 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	
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)**) In the case of non-destructive test, the current data of the test area is saved in the specified RamBuffer area. The test results are stored below. - RramResult1 (0 : FAIL / 1 : PASS) - RramResult2 (0 : FAIL / 1 : PASS) The test patterns used are the following (See “r_ramdiag_config.inc”): ◆ Test Patterns 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)	
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

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2">void R_RAM_Diag_walpat(uint32_t start, uint32_t end, uint32_t destructive)</td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">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(2)**)
In the case of non-destructive test, the current data of the test area is saved in the specified RamBuffer area.

The test results are stored below.
- RramResult1 (0 : FAIL / 1 : PASS)
- RramResult2 (0 : FAIL / 1 : PASS)

The test patterns used are the following (See “r_ramdiag_config.inc”):
◆ Test Patterns
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)</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td>uint32_t start</td><td>Start address of the block under test</td></tr>
<tr><td>uint32_t end</td><td>End address of the block under test</td></tr>
<tr><td>uint32_t destructive</td><td>Specify type of the Memory test
0: Non-destructive test
1: Destructive test</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>RramResult1</td><td>0 : FAIL / 1 : PASS</td></tr>
<tr><td>RramResult2</td><td>0 : FAIL / 1 : PASS</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

1.4 Clock

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

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

1.4.1 Main Clock Frequency Monitoring by CAC

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

1. Must select the reference clock (using the ref_clock input parameter)
2. Provide the frequency of the target and reference clocks in Hz

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

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

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

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

1.4.2 Oscillation Stop Detection of Main Clock

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

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

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

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

1.4.3 Clock Software API

The software API source files related to Clock test are shown in **Table 1-21**.

Table 1-21 Clock Source File

File Name	
clock_monitor.h	Declaration of Clock Test API function.
clock_monitor.c	Clock test implementation part

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

■ clock_monitor.c File

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

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

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

Syntax	
bool_t CAC_Err_Detect_Test(void)	
Description	
When the power is turned on, it checks that the frequency error detection by the CAC function and the interrupt by the overflow error detection are operating normally. Returns “TRUE” if each interrupt occurrence can be confirmed within a certain period (counted by software loop).	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : True = Passed(Each interrupt occurrences was confirmed) 0 : False = Failed(Could not confirm any interrupt occurrences)

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 example of configuration). The option setting memory is a series of registers that can be used to select the state of the microcontroller after reset and is located in the code flash area.

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

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

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	
NONE	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 the IWDT timed out or failed to update correctly, otherwise False.

Syntax	
bool_t IWDT_Err_Detect_Test(void)	
Description	
When the power is turned on, it check that the interrupt by the detection of counter underflow for IWDT function is operating normally. Returns “TRUE” if NMI interrupt occurrence by detecting IWDT counter underflow can be confirmed within a certain period of time (counted by software loop). Set f_IWDT_ERROR_TEST to "1" and determine if f_IWDT_ERROR_TEST becomes "0" within a certain period of time. Note that the user must create a process to set f_IWDT_ERROR_TEST to "0" when the IWDT underflow/refresh error interrupt status flag is "1" in NMI_Handler_callback(). For details, refer to “**2.5 Independent** Watchdog Timer (IWDT)”	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : TRUE = PASS(NMI interrupt occurrences can be confirmed)、 0 : FALSE = FAIL(NMI interrupt occurrences can not be confirmed)

Syntax	
bool_t IWDT_Err_Test_Judge(void)	
Description	
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. 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. Return "FALSE" in other case than above. Note that this function must be called within NMI_Handler_callback(). For details, refer to 2.5.2 Example of registering and writing an NMI interrupt callback function.	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
`bool_t`	"TRUE" if by IWDT error test , otherwise FALSE

2. Example Usage

This section gives 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, the function CPU_Test_ClassC can be used, as it is for the power-on tests, to automatically run 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.

For example, if the CPU core is CM33 and not using FPU, FPU-related tests from the CPU test completion can be excluded. (Set “0” for the directives from “BUILD_R_CPU_DIAG_11” to “BUILD_R_CPU_DIAG_15_6” in Table 1-15)

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

- Definition 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  (1)
#define BUILD_R_CPU_DIAG_4_2  (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)
#define BUILD_R_CPU_DIAG_9    (1)
#define BUILD_R_CPU_DIAG_10   (1)
#define BUILD_R_CPU_DIAG_11   (1)
#define BUILD_R_CPU_DIAG_12   (1)
#define BUILD_R_CPU_DIAG_13   (1)
#define BUILD_R_CPU_DIAG_14_1 (1)
#define BUILD_R_CPU_DIAG_14_2 (1)
#define BUILD_R_CPU_DIAG_15_1 (1)
#define BUILD_R_CPU_DIAG_15_2 (1)
#define BUILD_R_CPU_DIAG_15_3 (1)
#define BUILD_R_CPU_DIAG_15_4 (1)
#define BUILD_R_CPU_DIAG_15_5 (1)
#define BUILD_R_CPU_DIAG_15_6 (1)
#define BUILD_R_CPU_DIAG_16   (1)
```

2.2 ROM

ROM test, it compares the calculated CRC value of the range under test with a pre-stored reference CRC value. (32-bit CRC32 Polynomial uses "CRC-32")

It achieves "99.6% coverage of all information errors." by CRC calculator and comparing to CRC value by software.

A reference CRC value must be stored to a ROM area that is not included in the CRC calculation. The way the reference CRC value is calculated depends on your development environment.

In addition, this sample software performs divided processing to reduce the processing load of the ROM test and supports Multi Checksum.

The CRC module incorporated into the RA MCU must be initialized before use by calling the CRC_Init function. When dividing and processing, please initialize only the first time of divided 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 (*1) introduced below to calculate the reference CRC value. The user uses this tool to write the CRC value for reference in ROM in advance and compares it with this value in the self-test.

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

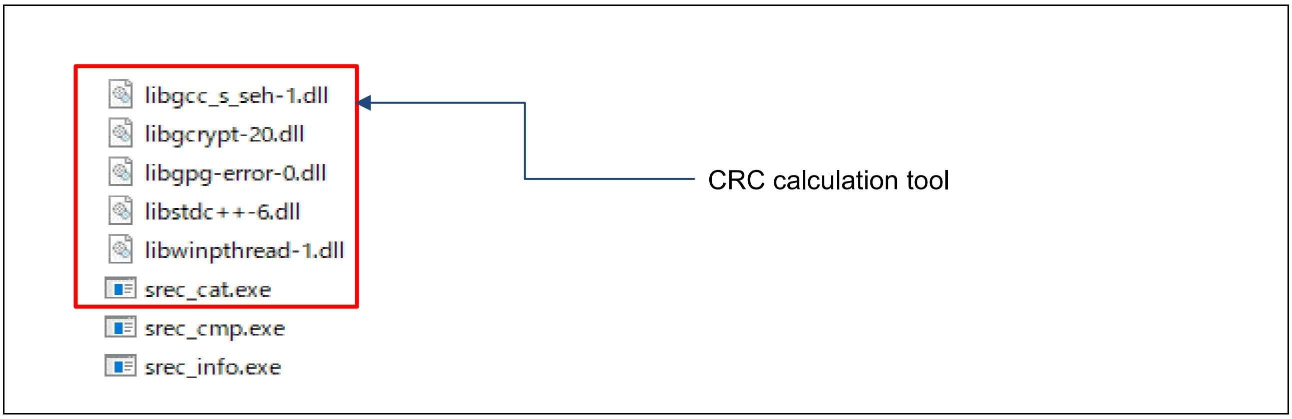
- SRecord Web Site (SRecord v1.65)

  http://srecord.sourceforge.net/

- CRC Checksum Generation with "SRecord" Tools for GNU and Eclips

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

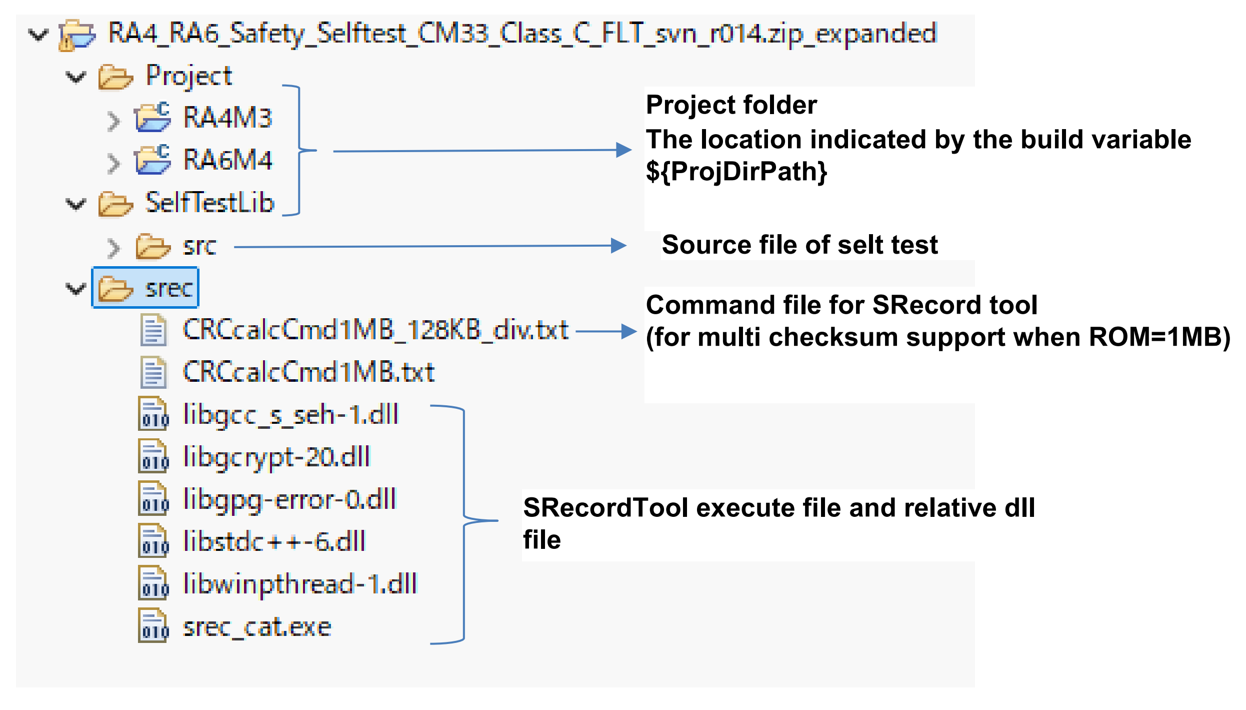
After unzipping the downloaded ZIP file, the following program are extracted to "\srecord-1.65.0-win64.zip\srecord-1.65.0-win64\bin"



libgcc_s_seh-1.dll
libgcrypt-20.dll
libgpg-error-0.dll
libstdc++-6.dll
libwinpthread-1.dll
srec_cat.exe
srec_cmp.exe
srec_info.exe

Figure 2-1 SRecord Tool Contents

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



Project folder
The location indicated by the build variable ${ProjDirPath}

Source file of selt test

Command file for SRecord tool
(for multi checksum support when ROM=1MB)

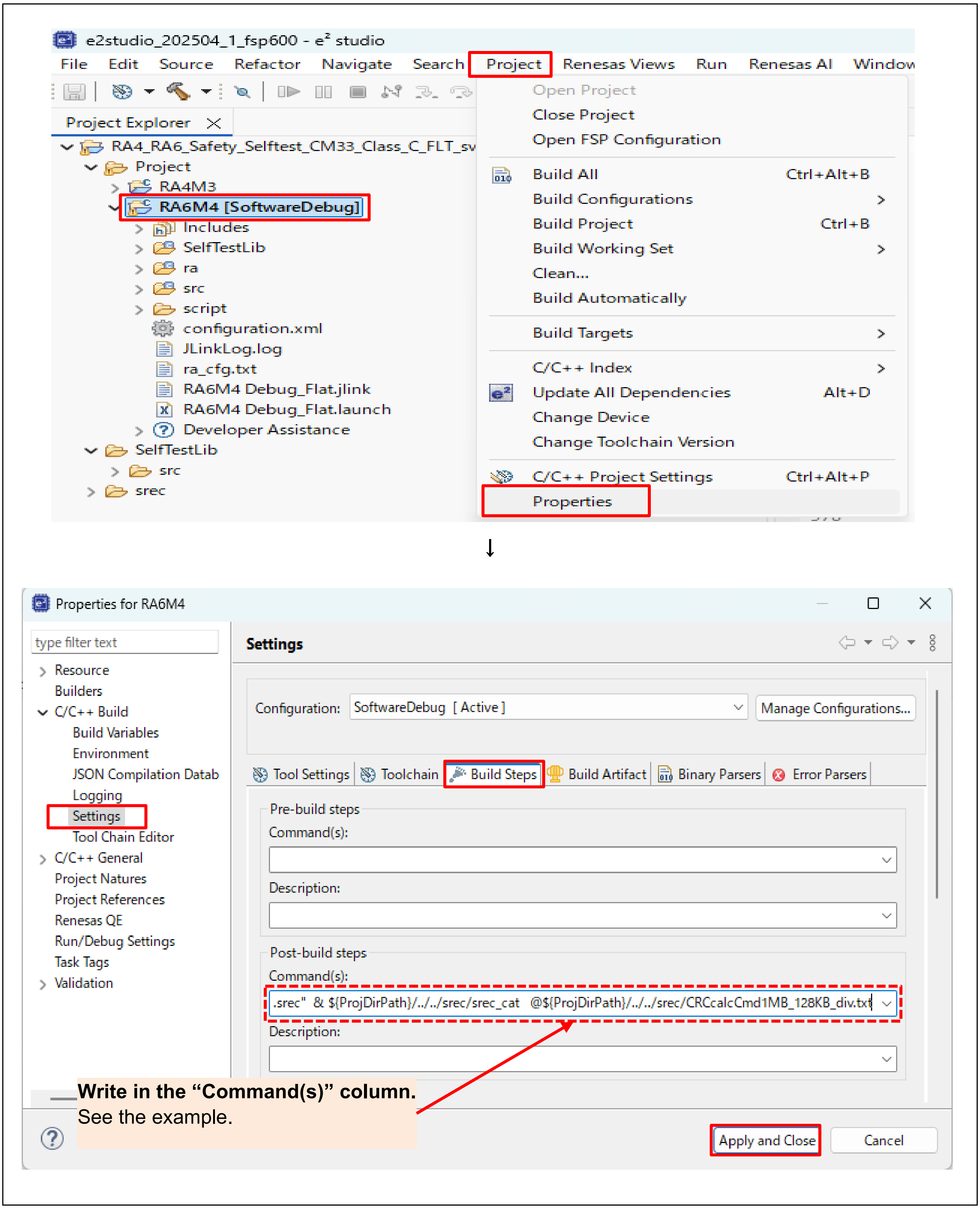
SRecordTool execute file and relative dll file

Figure 2-2 Folder Configuration Example

◆ **Setting in the project**

Open “Project”⇒”Properties” of e^{2}studio, in “Post-build steps” use command objcopy to generate S-record file from the * .elf file generated.

Note that the converted file name is "Original.srec" here, which will be the input for the SRecord tool.



Configuration: SoftwareDebug [Active]

Pre-build steps

Command(s):

Description:

Post-build steps

Command(s):

```
.srec"  & ${ProjDirPath}/../../srec/srec_cat   @${ProjDirPath}/../../srec/CRCcalcCmd1MB_128KB_div.txt
```

Description:

Write in the “Command(s)” column.
See the example.

Figure 2-3 Output S-Record file and start SRecord tool (example of RA6M4 project)

In the “Post-build steps” of the “Build Steps” tab in **Figure 2-4,** describe as follows. * () indicates in e^{2}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/CRCcalcCmd1MB_128KB_div.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:

“**CRCcalcCmd1MB_128KB_div.txt**” (when divided processing is enable)

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

■ CRCcalcCmd1MB_128KB_div.txt file contents (example)

```
# CRC calculate
Original.srec                     # Read srec file
-fill 0xFF 0x00000 0x100000    # 1MB ROM fill by 0xFF
#
-crop 0xE0000 0xFFFE0             # CRC calculate area (Test area 0xE0000 - 0xFFFE0  : 128KB -
32bye) for debug
-STM32-le  0x0FFFFC      # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFFC.
-crop 0xFFFFC 0x100000            # Keep CRC area(0xFFFFC - 0xFFFFF)
Original.srec                     # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0xC0000 0xE0000             # CRC calculate area (Test area 0xC0000 - 0xDFFFF : 128KB) for
debug
-STM32-le  0x0FFFF8      # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFF8.
-crop 0xFFFD8 0x100000            # Keep CRC area(0xFFFD8 - 0xFFFFF)
Original.srec                     # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0xA0000 0xC0000            # CRC calculate area (Test area 0xA0000 - 0xBFFFF : 128KB) for
debug
-STM32-le  0x0FFFF4     # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFF4.
-crop 0xFFFD4 0x100000           # Keep CRC area(0xFFFD4 - 0xFFFFF)
Original.srec                    # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0x80000 0xA0000            # CRC calculate area (Test area 0x80000 - 0x9FFFF : 128KB) for
debug
-STM32-le  0x0FFFF0     # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFF0.
-crop 0xFFFD0 0x100000           # Keep CRC area(0xFFFD0 - 0xFFFFF)
Original.srec                    # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0x60000 0x80000            # CRC calculate area (Test area 0x60000 - 0x7FFFF : 128KB) for
debug
-STM32-le  0x0FFFEC     # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFEC.
-crop 0xFFFCC 0x100000           # Keep CRC area(0xFFFCC - 0xFFFFF)
Original.srec                    # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0x40000 0x60000            # CRC calculate area (Test area 0x40000 - 0x5FFFF : 128KB) for
debug
-STM32-le  0x0FFFE8     # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFE8.
-crop 0xFFFC8 0x100000           # Keep CRC area(0xFFFC8 - 0xFFFFF)
Original.srec                    # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0x20000 0x40000            # CRC calculate area (Test area 0x20000 - 0x3FFFF : 128KB) for
debug
-STM32-le  0x0FFFE4     # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFE4.
-crop 0xFFFC4 0x100000           # Keep CRC area(0xFFFC8 - 0xFFFFF)
Original.srec                    # Read srec file
#
-fill 0xFF 0x00000 0xF0000     # 0-0xF0000 ROM fill by 0xFF
-crop 0x00000 0x20000            # CRC calculate area (Test area 0x0 - 0x1FFFF : 128KB) for debug
-STM32-le  0x0FFFE0     # The algorithm used by the STM32 hardware unit is just a CRC32, and
store CRC Value at 0xFFFE0.
-crop 0xFFFC0 0x100000           # Keep CRC area(0xFFFC0 - 0xFFFFF)
Original.srec                    # Read srec file
#
-fill 0xFF 0x000000 0x0FFFE0    # -fill 0xFF from 0x0 to 0xFFFE0
-Output addcrc.srec           # Output of S-record file including CRC value
```

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

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

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

Right-click on the top of the project tree and select “Debug” → “Debug Configuration”.

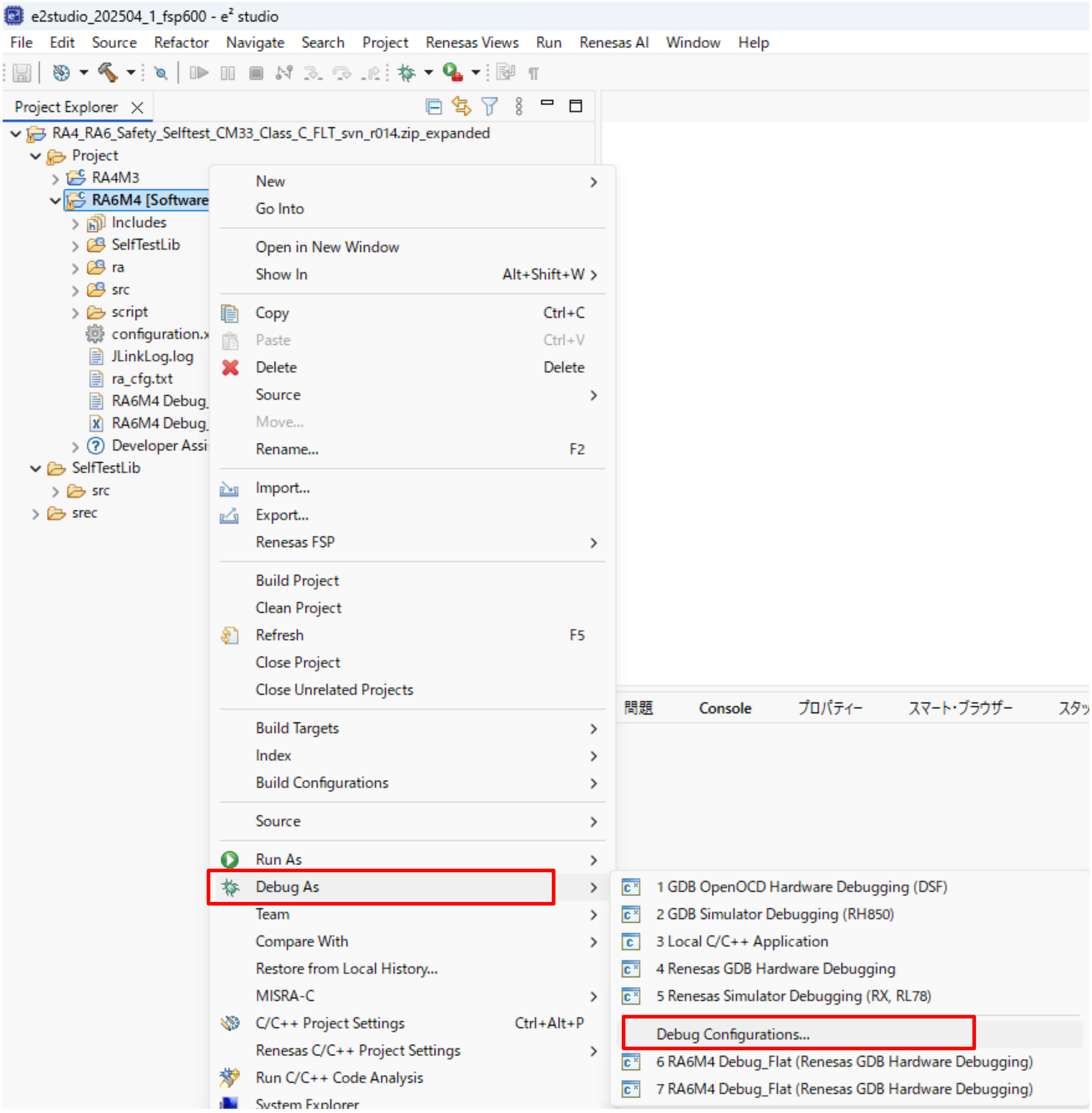
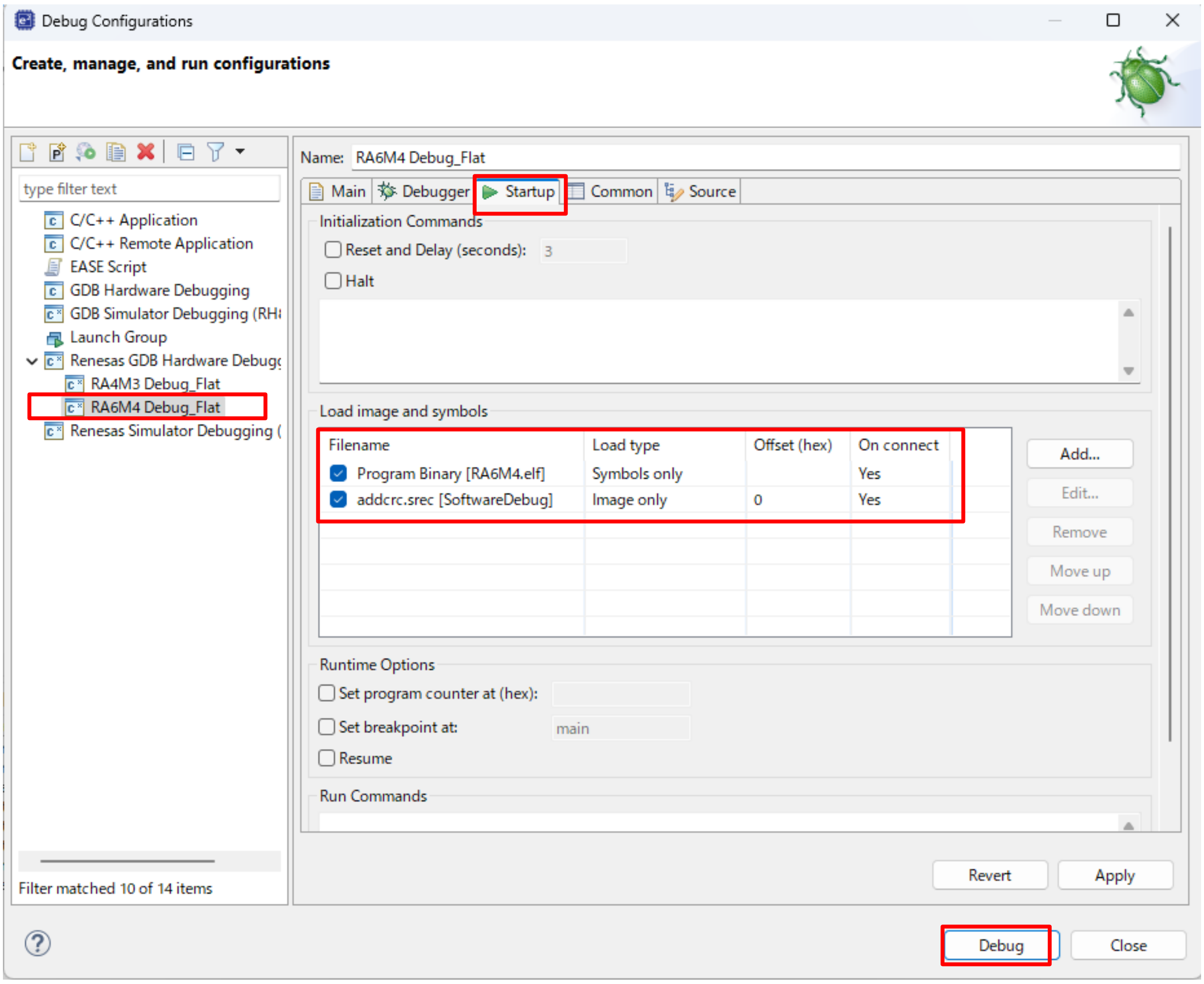


Figure 2-4 Select Debug Configuration of the Project (Example of RA6M4 project)

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.



Debug Configurations

Create, manage, and run configurations

Name: RA6M4 Debug_Flat

Initialization Commands

☐ Reset and Delay (seconds): 3

☐ Halt

Load image and symbols

Filename	Load type	Offset (hex)	On connect
☑ Program Binary [RA6M4.elf]	Symbols only		Yes
☑ addcrc.srec [SoftwareDebug]	Image only	0	Yes

Runtime Options

☐ Set program counter at (hex):

☐ Set breakpoint at: main

☐ Resume

Run Commands

Filter matched 10 of 14 items

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 “**RA_SelfTests.c**” file of the sample software (for RA6M4) (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.*/
#if(DIV_AREA==1)
●   #define CRC_ADDRESS 0x000FFFE0UL  // Flash ROM 1MB(8div * 128KB)
//#define CRC_ADDRESS 0x000BFFC0  // Flash ROM 768KB
//#define CRC_ADDRESS 0x0007FFC0  // Flash ROM 512KB
#else
              . . .
#endif
```

It stores the precomputed checksum with the above settings.

When divided processing is enabled. (DIV_AREA=1): Store in the area of address 0xFFFE0 to FFFFF.

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

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.

When testing RAM, keep the following points in mind:

1. Include r_ram_diag.h
2. Modify the directives in r_ram_diag_config.h as needed (refer to **Table 1-20**)
3. Disable ECC and S-Cache and perform the test
4. Define the required parameters 1for R_RAM_Diag (refer to **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

The monitoring of the main clock is set up with a single function call to ClockMonitor_Init.

Example:

```
#define TARGET_CLOCK_FREQUENCY_HZ         (50000000) // 50 MHz
#define REFERENCE_CLOCK_FREQUENCY_HZ      (15000)    // 15 kHz

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

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

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

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

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

When using FSP (Flexible Software Package) with e² studio, the ICU configuration can be set in the "Interrupts" tab of the RA Configuration Editor.

Table 2-1 Setting of IELSRn Register related to CAC

MCU	Event Name	IELSRn.IELS[8:0]
RA6M4, RA4M3	CAC_FERRI	0x09E
	CAC_OVFI	0x0A0

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

// NVIC settings related to CAC

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

NVIC settings related to Frequency error interrupt

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

NVIC settings related to Overflow error interrupt

If oscillation stop is detected, an NMI interrupt occurs. In this sample software, as shown in the following, the prepared in advance error handling function ("Clock_Stop_Detection()") is executed in the NMI interrupt callback function (NMI_Handler_callback).

```
static void NMI_Handler_callback(bsp_grp_irq_t irq)
{
      switch(irq){
            case BSP_GRP_IRQ_IWDT_ERROR      :
                                                   . . .
                  break;
            case BSP_GRP_IRQ_LVD1            :
            case BSP_GRP_IRQ_LVD2            :
                  break;
            case BSP_GRP_IRQ_OSC_STOP_DETECT :
                  Clock_Stop_Detection();
                  break;
            case BSP_GRP_IRQ_TRUSTZONE       :
                  . . .
                  break;
            default:
                break;

      }
}
```

2.5 Indenpend Watchdog Timer (IWDT)

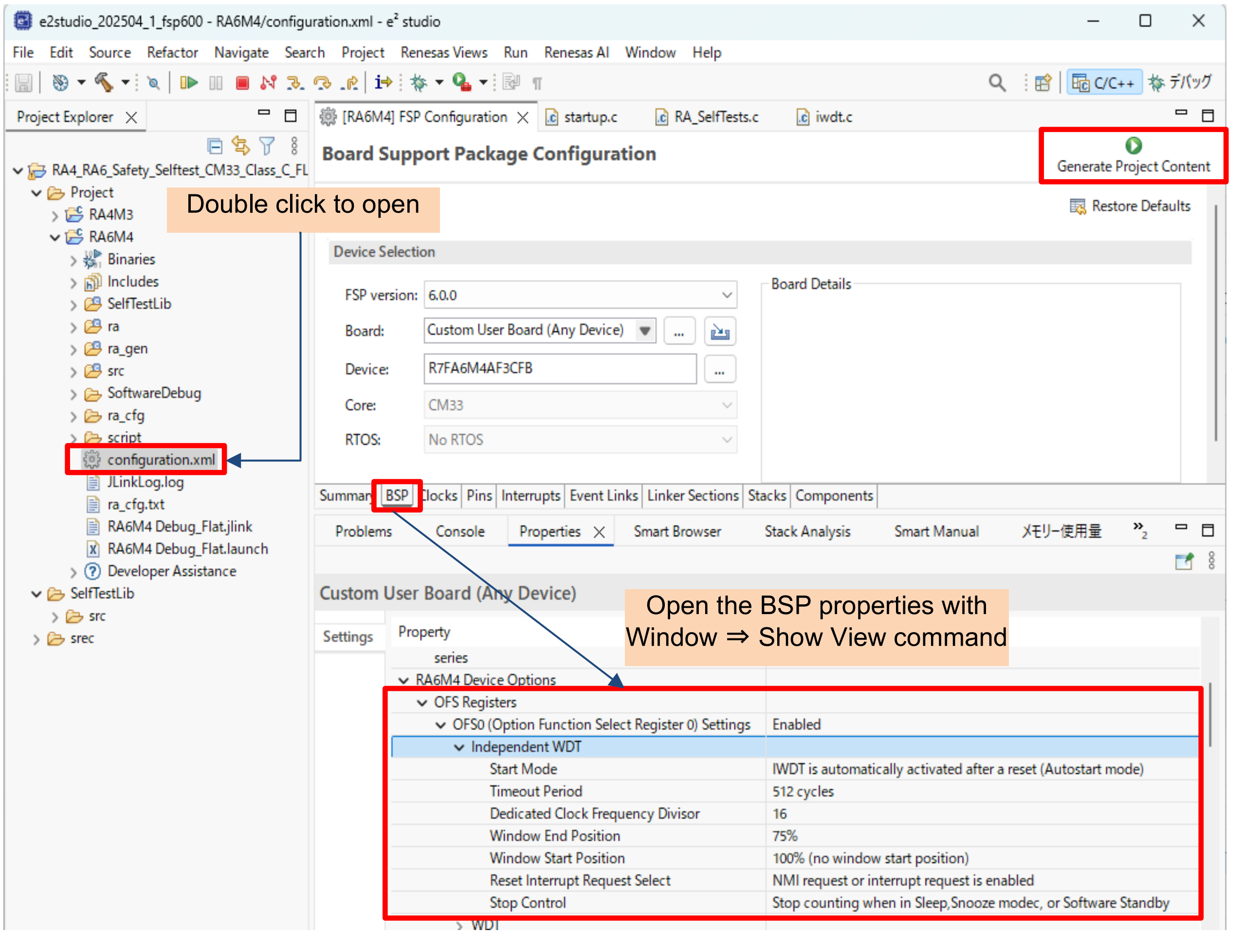
2.5.1 OFS0 Register Setting Example (IWDT related)

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 2-2 OFS0 Register Setting Example (IWDT related)

Item	OFS0 Register Setting (For example)
IWDT Start Mode Select (IWDTSTRT)	0: Automatically activate IWDT after a reset (auto start mode)
IWDT Timeout Period Select (IWDTTOPS[1:0])	10b：512 cycle (0x01FF)
IWDT-Dedicated Clock Frequency Division Ratio Select (IWDTCKS[3:0])	0010b：1/16
IWDT Window End Position Select (IWDTRPES[1:0])	00b：75%
IWDT Window Start Position Select (IWDTRPSS[1:0])	11b：100%
IWDT Reset Interrupt Request Select (IWDTRSTIRQS)	0: Enable non-maskable interrupt request or interrupt request
IWDT Stop Control (IWDTSTPCTL)	1: Stop counting when in Sleep, Snooze, or Software Standby mode.

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



Double click to open

Open the BSP properties with Window ⇒ Show View command

FSP version: 6.0.0
Board: Custom User Board (Any Device)
Device: R7FA6M4AF3CFB
Core: CM33
RTOS: No RTOS

Custom User Board (Any Device)

Property	
series	
RA6M4 Device Options	
OFS Registers	
OFS0 (Option Function Select Register 0) Settings	Enabled
Independent WDT	
Start Mode	IWDT is automatically activated after a reset (Autostart mode)
Timeout Period	512 cycles
Dedicated Clock Frequency Divisor	16
Window End Position	75%
Window Start Position	100% (no window start position)
Reset Interrupt Request Select	NMI request or interrupt request is enabled
Stop Control	Stop counting when in Sleep,Snooze modec, or Software Standby

Figure 2-6. Example of OFS0 Register Setting by Using FSP with e^2 studio (Example of RA6M4)

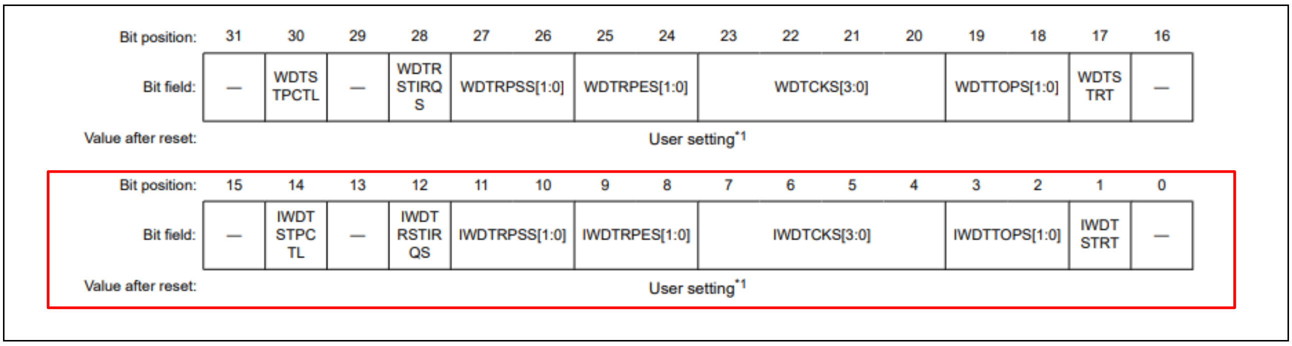
When the "Generate Project Content" button is clicked, the contents set in the property will be reflected in the definition of the corresponding symbol in the following file.

- Applicable file

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

- Applcable symbol (excerpt)

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



Bit position:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Bit field:	—	WDTSTPCTL	—	WDTRSTIRQS	WDTRPSS[1:0]		WDTRPES[1:0]		WDTCKS[3:0]				WDTTOPS[1:0]		WDTSTRT	—
Value after reset:	User setting*1															

Bit position:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit field:	—	IWDTSTPCTL	—	IWDTRSTIRQS	IWDTRPSS[1:0]		IWDTRPES[1:0]		IWDTCKS[3:0]				IWDTTOPS[1:0]		IWDTSTRT	—
Value after reset:	User setting*1															

Figure 2-7. Option Function Select Register 0 (OFS0)

For details on the IWDT, refer to RA MCU Hardware User's Manual, section "**25. 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();
```

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, it 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: It 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 (NMI_Handler_callback) (blue text)

```
 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;
               case BSP_GRP_IRQ_OSC_STOP_DETECT :
                       Clock_Stop_Detection();
                       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
RA FSP (Flexible Software Package):	www.renesas.com/FSP
RA Support Forum:	www.renesas.com/ra/forum
Renesas Support: :	www.renesas.com/support

Reference Documents

[1] Arm® Cortex®-M33 Devices Generic User Guide Revision: r1p0
  - 2.1.3 Core registers
  - Chapter 3:The Cortex®-M33 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	Sep 30, 2025	—	First edition issued
1.01	Oct 1, 2025	69	Added to “Table 1-18 RAM Test Algorithm Characteristics”
		84	Added to "2.2 ROM"
1.02	Oct. 16, 2025	2	Added to “About Support for Arm® TrustZone® “
1.03	Oct. 29.2025	78	Added API to 1.5.1
		98-99	Corrected the description and code example for 2.5.2

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

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/.

© 2025 Renesas Electronics Corporation. All rights reserved.