RAファミリ
RA MCUのためのIEC60730/60335セルフテスト・ライブラリ (CM33 Class-C)

R01AN6974JJ0100
Rev.1.00
2023.6.30

概要

今日、自動電子制御システムが多くの多様なアプリケーションに拡大し続けているため、信頼性と安全性の要件は、システム設計においてますます増大する要素になりつつあります。

たとえば、家電製品向けの IEC60730 安全規格を導入するには、製造業者が製品の安全で信頼性の高い動作を保証する自動電子制御を設計する必要があります。

IEC60730 規格は製品設計のすべての側面をカバーしていますが、Annex H はマイクロコントローラベースの制御システムの設計にとって非常に重要です。これにより、自動電子制御用の 3 つのソフトウェア分類が提供されます。

1. クラス A： 装置の安全性に寄与することを意図したものではない制御機能
   例：部屋のサーモスタット、湿度制御、照明制御、タイマ、スイッチ
2. クラス B： 装置の安全でない操作を防止するための制御機能
   例：洗濯機のサーマルカットオフおよびドアロック
3. クラス C： 特別な危険を防止するための制御機能
   例：自動バーナー制御と閉動作のためのサーマルカットアウト

このアプリケーションノートでは、柔軟なサンプルソフトウェアルーチンを使用して、IEC60730 クラス C 安全規格への準拠を支援する方法のガイドラインを示します。これらのルーチンは VDE Test and Certification Institute GmbH によって認定されており、テスト証明書のコピーは、このアプリケーションノートのダウンロードパッケージで入手できます。

提供されるソフトウェアルーチンは、リセット後およびプログラムの実行中に使用されます。このドキュメントとそれに付随するサンプルコードは、これを行う方法の例を提供します。

ターゲット

- デバイス：
  - ルネサス RA ファミリ MCU（Arm® Cortex®-M33）※シリーズとグループは下記**表 a** を参照
- 開発環境：
  - GNU-GCC ARM Embedded Toolchain 10.3.1.20210824 / Renesas e² studio 2021-10(21.10.0)

本書において「RA MCU」と表記している場合は、以下の製品のことを指します。

表 a. RA ファミリ MCU セルフテスト機能リスト

	CPU コア	Arm® Cortex®-M33
	シリーズ	RA6
	グループ	RA6M4
テスト機能	CPU	○
	ROM	○
	RAM	○
	クロック	○
	独立ウォッチドッグタイマ (IWDT)	○

Arm® TrustZone® への対応について

本セルフテストライブラリは、Arm® TrustZone® における「セキュア領域」(以降、Safety Part)で実行されることを前提としており、RA Project Generator (※) の "TrustZone Secure Project"でセルフテストライブラリのコード生成を実施しています。

また、RA Project Generator (※) の "TrustZone Non-Secure Project"で「非セキュア領域」(以降、Non-Safety Part) で実行するサンプルプログラムを含めた最終コード作成を実施しています。

※RA Project Generator の詳細は、RA FSP (Flexible Software Package) のドキュメントを参照ください。

また、RA Arm® TrustZone® ツールの詳細については、以下のリンクを参照してください。

https://www.renesas.com/jp/ja/document/apn/ra-arm-trustzone-tooling-primer.

セルフテストライブラリの概要

セルフテストライブラリは、命令デコード、CPU レジスタ、内部メモリ、ウォッチドッグ・タイマおよびシステム・クロックを対象とする監視関数で構成されます。

以降で説明するように、異常監視処理には監視を行う各モジュールのアプリケーション・プログラム・インタフェース（API）が用意されています。各関数は用途に応じて使用します。

セルフテストライブラリ関数は、IEC60730Class-C に準じてモジュール別に分かれています。異常監視処理は、各テスト関数を順番に選択してスタンドアロンで実行することができます。

また、Arm TrustZone 対応マイコン内部を安全部(セキュア領域)と非安全部(非セキュア領域)に分離する方式を採用し、本セルフテストライブラリは安全部(セキュア領域)に実装されることを想定しています。

RA6 シリーズ（Arm® Cortex®-M33 搭載）のセルフテストライブラリには以下の主なセルフテストを実施する関数があります。

- 命令デコード

Arm　Cortex-M33 の該当する命令に対して仕様に沿って正常に動作するかを検証します。

IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H2.18.5 equivalence class test を参照してください。

- CPU レジスタ

「表 1-1 CPU Test target 」に記載された CPU レジスタをテストします。
内部データ・パスは、以上のレジスタの正常動作テストの中で検証します。
IEC 60730-1:2013+A1:2015+A2:2020 Annex H - Table H.11.12.7　1.CPU を参照してください。

- 不変メモリ

MCU の内部 Flash メモリをテストします。
IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H2.19.4.2 CRC – double word を参照してください。

- 可変メモリ

内部 SRAM をテストします。
RAM テストでは、WALKPAT アルゴリズムと Extended March C-アルゴリズムを使用します。
IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H.2.19.7　walkpat memory test を参照してください。

- システム・クロック

基準クロック・ソースを元にしてシステム・クロックの動作および周波数をテストします（このテストには内部または外部の独立した基準クロックが必要です）。
IEC Reference - IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H2.18.10.1 Frequency monitoring を参照してください。

- CPU／プログラムカウンタ

プログラムが規定時間内でシーケンスを実行してることを確認するために、CPU とは独立したクロックで動作する内蔵ウォッチドッグ・タイマを用いて確認しています。
IEC 60730-1:2013+A1:2015+A2:2020 Annex H – H2.18.10.3 time-slot and logical monitoring を参照してください。

セルフテストライブラリ及びテストサンブルソフトの S/W マッピングについて

本サンプルプログラムは以下のように TrustZone Secure project、TrustZone non-secure project の 2 つのプロジェクトを作成し、Non-safety part(**非安全部**)と Safety Part(**安全部**)に振り分けてプログラムを配置します。

なお、本セルフテストライブラリは Safety part(安全部)に配置する必要があります。

◆非安全部（Non safety part）：　非セキュア(non-secure)領域
■ユーザーアプリケーション
■P-ON 起動時の非安全部(Non-Safety part)の初期化処理

◆安全部（Safety part）：　セキュア(secure)領域
■P-ON 起動時の安全部(Safety part)の初期化処理
■セルフテストライブラリ関連初期設定(周期タイマ(AGT5)、割込みなど)
■P-ON 起動時の各セルフテスト(CPU, RAM, ROM, クロックの各テスト)
■定周期の各セルフテスト　(CPU, RAM, ROM の各テスト)

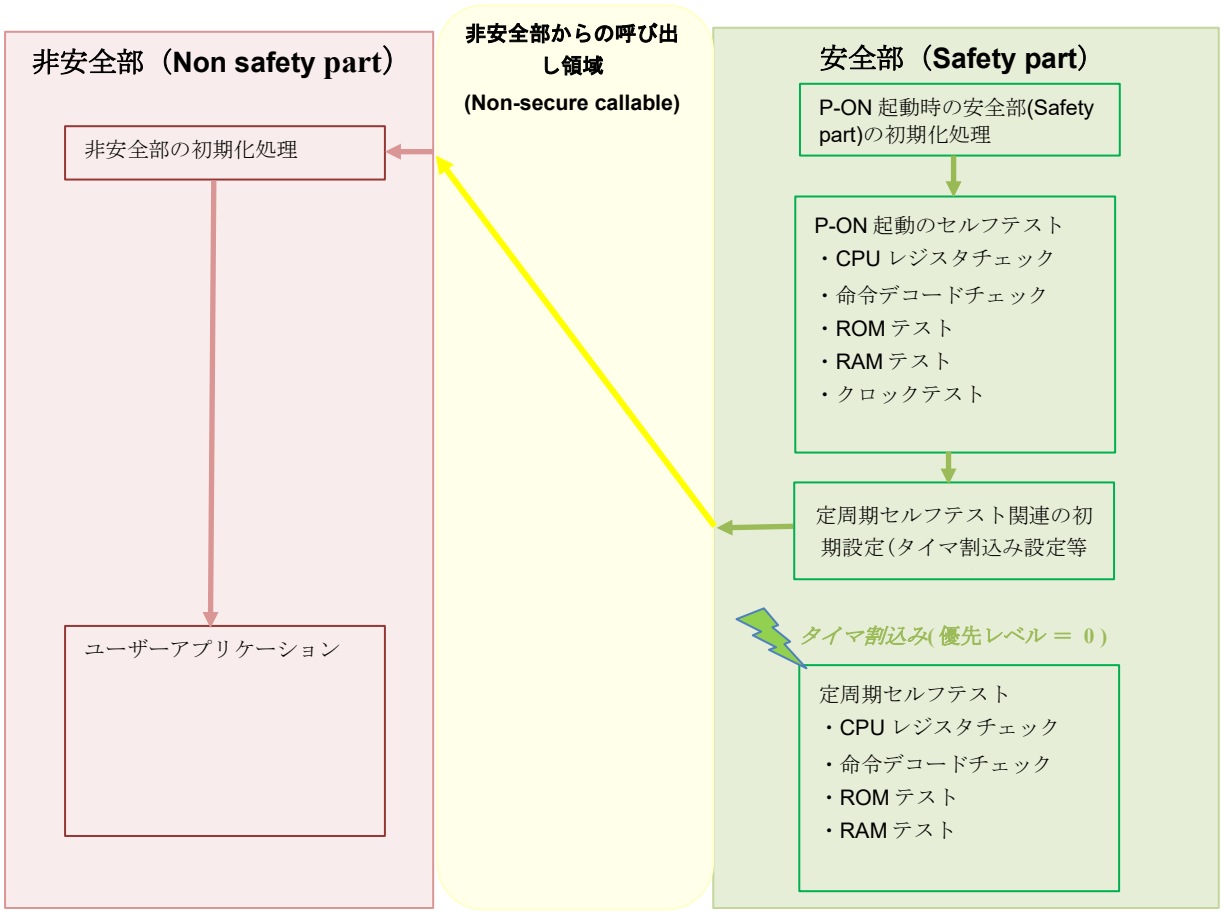

図 a　例.セルフテストライブラリ処理の非安全部、安全部への配置イメージ

※非安全部から安全部の関数を呼び出す場合については下記「Renesas RA Family RA Arm® TrustZone® Tooling Primer」を参照ください

＜参考 URL＞：

https://www.renesas.com/jp/ja/document/apn/ra-arm-trustzone-tooling-primer?language=en&r=1353811

目次

1. テスト

1.1 CPU

CPU テストの目的は、CPU コアからランダムな永続的な障害を検出することです。
CPU テストの主な機能は以下のとおりです。

- ●CPU命令テスト(CPU instruction test)
- ●CPUレジスタテスト(CPU register test)

1.1.1 CPU レジスタテストと CPU 命令テスト

本セルフテストライブラリで実施する CPU テストの各テスト概要について**表 1-16** に記載しております。
各テストを実行することで関連するレジスタや命令コードをテストし、その 実行結果を確認することで CPU の故障を検出できます。

テスト対象(概要)は下記の**表 1-1** にリストされている CPU 命令とレジスタです。

表 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
	FSP		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	✓

下記の**表 1-2～表 1-3** は Armv8-M レジスタの一覧とテスト対応状況を示します。

なお、各レジスタの詳細内容は“Arm®v8-M Architecture Reference Manual”(参考文書[2])を参照ください。

[表記]

✓　　: テスト対象

(空白) : テスト対象外

N/A　: 適用外です。

表 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	

表 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	

下記の**表 1-4** ～**表 1-13** は Armv8-M の命令一覧とテスト対応状況を示します。
なお、各命令の詳細内容は“Arm® Cortex®-M33 Devices Generic User Guide” (参考文書[1])を参照ください。

主な目的は、個々の命令をテストすることではなく、CPU コアのハードウェア障害を検出することであることに注意してください。

[表記]

✓　　: テスト対象

(空白) : テスト対象外

N/A　: 適用外です。

表 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

表 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)	✓

表 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	✓

表 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)	✓

表 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	✓

表 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

表 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

表 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	✓

表 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

表 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　テストエラー

エラーが検出された場合、CPU テストは下記の関数にジャンプします。
このエラー処理関数は閉ループ処理になっているため、return してはいけません。
すべてのテスト関数は、C 関数呼び出し後のレジスタ保存の規則に従います。したがって、ユーザはこれらの関数を通常の C 関数のように呼び出すことができ、事前にレジスタ値を保存するいかなる責任もありません。

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

1.1.3　CPU ソフトウェア API

CPU テストに関連するソフトウェア API ソースファイルは**表 1-14** の通りです。
CPU テスト API を実行すると、関連する CPU レジスタや命令コードがテストされます。
引数に出力された実行結果を確認することで、CPU 障害を検出できます。

コードをコンパイルする前に CPU テストを構成します。**表 1-15** 及び**表 1-16** に CPU テスト構成のディレクティブと各 CPU テストを示します。
詳細については”**2.1.3 CPU テストの事前準備**”を参照ください。

表 1-14 CPU ソフトウェア API ソースファイル

ファイル名	備考
r_cpu_diag_config.h	CPU テストディレクティブの定義
cpu_test.c	CPU テスト実装部
r_cpu_diag_0.asm r_cpu_diag_1.asm r_cpu_diag_2.asm r_cpu_diag_3.asm r_cpu_diag_4.asm r_cpu_diag_5.asm r_cpu_diag_6.asm r_cpu_diag_7_1.asm r_cpu_diag_7_2.asm r_cpu_diag_7_3.asm r_cpu_diag_8.asm 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	CPU テストコア機能の定義 注： 一部のテストは、 r_cpu_diag_7_1.asm、 r_cpu_diag_7_2.asm などの複数のファイルで構成されていることに 注意してください。
r_cpu_diag_0.h r_cpu_diag_1.h r_cpu_diag_2.h r_cpu_diag_3.h r_cpu_diag_4.h r_cpu_diag_5.h r_cpu_diag_6.h r_cpu_diag_7_1.h r_cpu_diag_7_2.h r_cpu_diag_7_3.h r_cpu_diag_8.h 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	CPU テストコア機能の宣言

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	
r_cpu_diag.c	CPU テスト API 関数の定義
r_cpu_diag.h	CPU テスト API 関数の宣言
r_cpu_diag.inc	アセンブラマクロの定義

表 1-15 Directives for Software Configuration for CPU Test

ディレクテブ名	説明
BUILD_R_CPU_DIAG_0	“1”に設定すると、CPU テスト関数：R_CPU_Diag0 が構築されます。
BUILD_R_CPU_DIAG_1	“1”に設定すると、CPUテスト関数：R_CPU_Diag1が構築されます。
BUILD_R_CPU_DIAG_2	“1”に設定すると、CPUテスト関数：R_CPU_Diag2が構築されます。
BUILD_R_CPU_DIAG_3	“1”に設定すると、CPUテスト関数：R_CPU_Diag3が構築されます。
BUILD_R_CPU_DIAG_4_1 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag4_1が構築されます。
BUILD_R_CPU_DIAG_4_2 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag4_2が構築されます。
BUILD_R_CPU_DIAG_5	“1”に設定すると、CPUテスト関数：R_CPU_Diag5が構築されます。
BUILD_R_CPU_DIAG_6	“1”に設定すると、CPUテスト関数：R_CPU_Diag6が構築されます。
BUILD_R_CPU_DIAG_7_1 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag7_1が構築されます。
BUILD_R_CPU_DIAG_7_2 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag7_2が構築されます。
BUILD_R_CPU_DIAG_7_3 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag7_3が構築されます。
BUILD_R_CPU_DIAG_8	“1”に設定すると、CPUテスト関数：R_CPU_Diag8が構築されます。
BUILD_R_CPU_DIAG_9	“1”に設定すると、CPUテスト関数：R_CPU_Diag9が構築されます。
BUILD_R_CPU_DIAG_10	“1”に設定すると、CPUテスト関数：R_CPU_Diag10が構築されます。
BUILD_R_CPU_DIAG_11	“1”に設定すると、CPUテスト関数：R_CPU_Diag11が構築されます。
BUILD_R_CPU_DIAG_12	“1”に設定すると、CPUテスト関数：R_CPU_Diag12が構築されます。
BUILD_R_CPU_DIAG_13	“1”に設定すると、CPUテスト関数：R_CPU_Diag13が構築されます。
BUILD_R_CPU_DIAG_14_1 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag14_1が構築されます。
BUILD_R_CPU_DIAG_14_2 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag14_2が構築されます。
BUILD_R_CPU_DIAG_15_1 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag15_1が構築されます。
BUILD_R_CPU_DIAG_15_2 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag15_2が構築されます。
BUILD_R_CPU_DIAG_15_3 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag15_3が構築されます。
BUILD_R_CPU_DIAG_15_4 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag15_4が構築されます。
BUILD_R_CPU_DIAG_15_5 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag15_5が構築されます。
BUILD_R_CPU_DIAG_15_6 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag15_6が構築されます。
BUILD_R_CPU_DIAG_16 *1	“1”に設定すると、CPUテスト関数：R_CPU_Diag16が構築されます。

*1
表 1-16 参照
一部のテストには、BUILD_R_CPU_DIAG_7_1、BUILD_R_CPU_DIAG_7_2 などの複数のディレクティブがあることに注意してください。

表 1-16 CPU Test Target

Test No	index *1	Function name *2	Objective of the Test (テストの目的)
0	0	R_CPU_Diag0	Four basic arithmetic operations (add, sub, mul and div) 4 つの基本的な算術演算（add、sub、mul、および div）
1	1	R_CPU_Diag1	Sign/Zero extension operations Sign/Zero extension 操作(※SXTA and UXTA 命令)
2	2	R_CPU_Diag2	Branch, logical, comparison and conditional operations 分岐、論理、比較、および条件付き操作(※ADR 命令)
3	3	R_CPU_Diag3	Bit manipulation and data transfer ビット操作とデータ転送
4	4 5	R_CPU_Diag4_1 R_CPU_Diag4_2	Memory access (Load/Store) without exclusive 排他的でないメモリアクセス（ロード/ストア）
5	6	R_CPU_Diag5	Memory access (Load/Store) with exclusive and privileged 排他的および特権付きのメモリアクセス（ロード/ストア）
6	7	R_CPU_Diag6	System related システム関連
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 R0-R12, SP(R13), LR(R14), APSR　各レジスタ
8	11	R_CPU_Diag8	Multiply-accumulate and multiply-subtract operations (MAC and MSB) 積和演算と乗算減算演算（MAC および MSB）
9	12	R_CPU_Diag9	Combined arithmetic operations 結合された算術演算
10	13	R_CPU_Diag10	Saturating and rounding operations 飽和および丸め操作
11	14	R_CPU_Diag11	Floating-point four basic arithmetic, absolute value and comparison operations 浮動小数点 4 つの基本的な算術演算、絶対値演算、および比較演算
12	15	R_CPU_Diag12	Floating-point multiply-accumulate and multiply-subtract operation 浮動小数点の積和演算と乗算-減算演算
13	16	R_CPU_Diag13	Floating-point rounding and data type conversion 浮動小数点の丸めとデータ型変換
14	17 18	R_CPU_Diag14_1 R_CPU_Diag14_2	Floating-point memory access and data transfer 浮動小数点メモリアクセスとデータ転送
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 S0 – S31, FPSCR　各レジスタ
16	25	R_CPU_Diag16	CPU register test using WALKPAT WALKPAT アルゴリズムを使用した CPU レジスタテスト

*1）複数のインデックスにまたがる場合は、すべてのインデックスでテストが必要です。

*2）各関数をコード生成するためのソフトウエア構成ディレクティブについては**表 1-15** を参照。

■ cpu_test.c ファイル

Syntax
`void CPU_Test_ClassC(void)`

Description

次の順序で CPU テストを実行します。
 	
1. 現在のスタックミットレジスタを退避します。
`SaveMspPt = __get_MSPLIM();`
`SavePspPt = __get_PSPLIM();`
2. CPU スタックポインタ監視機能を無効にします。
`__set_MSPLIM(0);`
`__set_PSPLIM(0);`
3. パラメータを渡し、関数 R_CPU_Diag を呼び出します。
 	
4. 引数「result」の値を確認します。
 	
5. 結果が OK の場合、上記 3.へ戻ります。(次のテストを実施)。
定義された全ての CPU テストが完了したら下記 6 へ。
なお、エラーが検出された場合、外部関数 CPU_Test_ErrorHandler が呼び出されます。
詳細については、個々のテストを参照してください。
 	
6. 上記 1 で退避した CPU スタックポインタリミットレジスタを復帰します。
 	
7. CPU_Test_PC
 	
8. 全てのテストが実施されたなら関数を終了します。
実施されたなかった場合は外部関数 CPU_Test_ErrorHandler が呼び出されます。

Input Parameters	
NONE	N/A
Output Parameters	
`const uint32_t forceFail`	強制 FAIL オプション “1”(N/A)に固定 ※強制 FAIL にしたい場合、”0”固定に変更してください。
Return Values	
NONE	N/A

Syntax	
`void CPU_Test_PC(void)`	
Description	
この関数は、プログラムカウンタ（PC）レジスタをテストします。 これにより、PC が確実に動作していることを確認します。 この関数は、関数が実際に実行されたことを確認できるように、指定されたパラメータの反転値を返します。この戻り値が正しいかどうかがチェックされます。 エラーが検出された場合、外部関数 `CPU_Test_ErrorHandler` が呼び出されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

■ r_cpu_diag.c ファイル

Syntax	
void R_CPU_Diag(uint32_t index, const uint32_t forceFail, int32_t *result)	
Description	
引数 index を使用して、CPU テスト番号に該当するテスト関数を実行します。 引数 index とテスト番号、該当テスト関数については**表 1-16** を参照してください。 1. “resultTemp”に初期値を設定します。 テスト関数を実行すると、テスト結果が「resultTemp」に保存されます。 2. 引数 Index の値が有効がどうかをチェックします。 無効の場合、テスト結果に FAIL(=0)を設定して終了します。 3. 引数 index の値に従い、該当する CPU テストのテスト関数を実行します。 4. 結果を*result へ設定し、処理を終了します。	
Input Parameters	
uint32_t index	CPU テスト番号(**表 1-16** 参照) 引数の値が無効な場合は FAIL を返します。
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

Syntax	
`uint32_t R_CPU_Diag_GetVersion(void)`	
Description	
この関数は、CPU テストソフトウェアのバージョン情報を返します。 バージョンは、”r_cpu_diag.h”ファイルで定義	
Input Parameters	
`NONE`	N/A
Output Parameters	
`uint32_t version`	CPU テスト　ソフトウエアバージョン (0xXXXXYYYY → XXXX : Major, YYYY: Minor)
Return Values	
`uint32_t`	0xXXXXYYYY → XXXX : Major, YYYY: Minor

Syntax	
`static void norm_null(const uint32_t forceFail, int32_t *result)`	
Description	
この関数は、ディレクティブでコンパイルから除外された CPU テスト関数のダミー関数です。 テスト結果を PASS に設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的に FAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_0.asm ファイル

Syntax	
void R_CPU_Diag0(const uint32_t forceFail, int32_t *result)	
Description	
1 Addition instructions test ADCS(register), ADDS (register), SADD16, SADD8, UADD16, UADD8, SHADD16, SHADD8 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Subtraction instructions test** SBCS (register), SUBS (immediate), RSBS (immediate), SSUB16, SSUB8, USUB16, USUB8, SHSUB16, SHSUB8 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Multiplication instructions test** MULS, SMULL, SMULWB, SMMULR, SMULTB, UMULL の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **4 Division instructions test** SDIV, UDIV の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **5 Addition and subtraction for stack pointer test** SUB (SP minus immediate), ADD (SP plus immediate), SUB.W (SP minus immediate), ADD.W (SP plus immediate) の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ：N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_1.asm ファイル

Syntax	
`void R_CPU_Diag1(const uint32_t forceFail, int32_t *result)`	
Description	
1 Sign extension SXTAB T1, SXTAB16 T1, SXTAH T1, SXTB T1, SXTB16 T1, SXTH T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Zero extension** UXTAB T1, UXTAB16 T1, UXTAH T1, UXTB T1, UXTB16 T1, UXTH T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的に FAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_2.asm ファイル

Syntax	
`void R_CPU_Diag2(const uint32_t forceFail, int32_t *result)`	
Description	
1 Branch ADR T1, ADR T3, BEQ T1, B T2, BL T1, BLX T1, BX T1, CBZ T1, IT EQ T1, TBB T1, TBH T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Logical test** TEQ T1, TST T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Logical operation** ANDS T1, ORRS T1, ORNS T1, EORS T1, MVNS T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **4 Comparison** CMN T1, CMP T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_3.asm ファイル

Syntax	
`void R_CPU_Diag3(const uint32_t forceFail, int32_t *result)`	
Description	
1 Bit manipulation 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 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Data manipulation** REV T1, REV16 T1, REVSH T1, SEL T1, PKHBT T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Data transfer** MOVS (immediate) T1, MOVT T1, MRS T1, MSR (register) T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_4_1.asm ファイル

Syntax
`void R_CPU_Diag4_1(const uint32_t forceFail, int32_t *result)`
Description
1 LDR and STR 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 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 LDRH and STRH** 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 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 LDRB and STRB** LDRSB (register) T1, STRB (register) T1 , LDRB (immediate) T1, STRB (immediate) T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。

Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_4_2.asm ファイル

Syntax	
void R_CPU_Diag4_2(const uint32_t forceFail, int32_t *result)	
Description	
4 LDRD and STRD LDRD (immediate) T1, STRD (immediate) T1, (post-indexed) , LDRD (immediate) T1, STRD (immediate) T1, (immediate) , LDRD (immediate) T1, STRD (immediate) T1, (pre-indexed) の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **5 LDM and STM** LDM and STM , LDM T3, STMDB T2 , LDM T2, STM T2 , LDMDB T1, STM T2 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **6 LDA and STL** LDA T1, STL T1 , LDAH T1, STLH T1 , LDAB T1, STLB T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **7 LDRH / LDRSB (literal)** LDRH (literal) T1 , LDRSB (literal) T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_5.asm ファイル

Syntax	
`void R_CPU_Diag5(const uint32_t forceFail, int32_t *result)`	
Description	
1 LDAEX and STLEX LDAEX T1, STLEX T1 , LDAEXH T1, STLEXH T1 , LDAEXB T1, STLEXB T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 LDREX and STREX** LDREX T1, STREX T1 , LDREXH T1, STREXH T1 , LDREXB T1, STREXB T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 LDRT and STRT** LDRT T1, STRT T1 , LDRHT T1, STRHT T1 , LDRSHT T1, STRHT T1 , LDRBT T1, STRBT T1 , LDRSBT T1, STRBT T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_6.asm ファイル

Syntax	
void R_CPU_Diag6(const uint32_t forceFail, int32_t *result)	
Description	
1 PUSH and POP R4, R5, R6, R7, R8, R9 を使用して PUSH 命令後に POP 命令を実行し R4 と R7、R5 と R8、R6 と R9 の各レジスタで期待値との一致を確認する。 **2 Other (miscelaneous) operations** CLREX T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_7_1.asm ファイル

Syntax
`void R_CPU_Diag7_1(const uint32_t forceFail, int32_t *result)`

Description
1 Detecting “0” fixed fault for status and control registers (ステータスおよび制御レジスタの「0」固定障害の検出) R4, R5 を使用して APSR レジスタの該当ビットへ”1”を書き込み後、読み出しを実行し R4 と R5 の各レジスタと期待値との一致を確認する。("0”固定になっていないことの確認) **2 Detecting “1” fixed fault for status and control registers** (ステータスおよび制御レジスタの「1」固定障害の検出) R4, R5 を使用して APSR レジスタの該当ビットへ”0”を書き込み後、読み出しを実行し R4 と R5 の各レジスタと期待値との一致を確認する。("1”固定になっていないことの確認) **3 Detecting “0” fixed fault for general purpose registers** (汎用レジスタの「0」固定障害の検出) R0～R12、LR(R14)へ ALL”1”を書き込み後、読み出しを実行し R0～R12、LR(R14)の各レジスタと期待値との一致を確認する。("0”固定になっていないことの確認) **4 Detecting “1” fixed fault for general purpose registers** (汎用レジスタの「1」固定障害の検出) R0～R12、LR(R14)へ ALL”0”を書き込み後、読み出しを実行し R0～R12、LR(R14)の各レジスタと期待値との一致を確認する。("1”固定になっていないことの確認) 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。

Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ

Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)

Return Values	
`NONE`	N/A

■ r_cpu_diag_7_2.asm ファイル

Syntax

```
void R_CPU_Diag7_2(const uint32_t forceFail, int32_t *result)
```

Description

5 Detecting coupling fault for general purpose registers between any two bits : 任意の 2 ビット間の汎用レジスタの結合障害の検出

R0-R12, R14 レジスタに次のテストを実施します

－**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**)**

手順は、以下の通り

1.上記の各テストパターンを R0 に設定し、R1 へ書き込み、R0 と一致確認する。
2.一致すれば書き込み対象レジスタを R2 から R14 まで順に実施する。
3. 上記の各テストパターンを R14 に設定し、R0 へ書き込み、R0 と一致確認する。
4.一致すれば次のテストパターンを実施する。
5.全て終了すれば下記のテストへ移行する。

6. Detecting coupling fault for general purpose registers between any two registers : 任意の 2 つのレジスタ間の汎用レジスタの結合障害の検出)

－R7、R8、R9、R10、R11、R12、LR（R14）結合障害の検出
－R0、R1、R2、R3、R4、R5、R6 の結合障害の検出

手順は、以下の通り

1.R0～R6 に各々テストパターンを設定し、R0 を R7 へ、R1 を R8 へ、…、R6 を R14 へ書き込み、R0 と R7、R1 と R8、…、R6 と R14 の値の一致を各々確認する。
2.R7～R14 に各々テストパターンを設定し、R8 を R0 へ、R9 を R1 へ、…、R7 を R6 へ書き込み、R8 と R0、R9 と R1、…、R7 と R6 の値の一致を各々確認する。
3.テストを終了する。

なお、R13(SP)は本テストでは対象外です。

期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。

Input Parameters

`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ

Output Parameters

`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)

Return Values

`NONE`	N/A

■ r_cpu_diag_7_3.asm ファイル

Syntax
`void R_CPU_Diag7_3(const uint32_t forceFail, int32_t *result)`

Description

7. Detecting "0" fixed fault for MSP(R13) : MSP（R13）の「0」固定障害の検出
R5 を使用して SP(R13)レジスタへ"0xfffffffc"を書き込み後、読み出しを実行し
R5 と SP(R13)を期待値との一致を確認する。("0"固定になっていないことの確認)

8. Detecting "1" fixed fault for MSP(R13) : MSP（R13）の「1」固定障害の検出
R5 を使用して SP(R13)レジスタへ"0x00000000"を書き込み後、読み出しを実行し
R5 と SP(R13)を期待値との一致を確認する。("1"固定になっていないことの確認)

9. Detecting coupling fault for MSP(R13) between any two bits : 任意の 2 ビット間の MSP（R13）の結合障害の検出
R13(SP)に次のテストを実施します
－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**)**
手順は、以下の通り
1.上記の各テストパターンを R5 に設定し、R13(SP)へ書き込み、R5 と一致確認する。
2.一致すれば次のテストパターンを実施する。
3.全て終了すれば下記のテストへ移行する。

10. Detecting coupling fault between MSP(R13) to other general purpose registers : MSP（R13）と他の汎用レジスタ間の結合障害の検出
－SP(R13)、R2 カップリング障害の検出
－SP(R13)、R3 カップリング障害の検出
手順は、以下の通り
1.R6, R7 に各々テストパターンを設定し、R6 を SP(R13)へ、R7 を R2 へ書き込み、
R6 と SP(R13)、R7 と R2 の値の一致を各々確認する。
2. R6, R7 に各々テストパターンを設定し、R7 を SP(R13)へ、R6 を R3 へ書き込み、
R7 と SP(R13)、R6 と R3 の値の一致を各々確認する。
3.テストを終了する。

なお、R13(SP)の bit0,1 は"0"固定です。
期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を" resultTemp"へ設定します。

Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_8.asm ファイル

Syntax	
`void R_CPU_Diag8(const uint32_t forceFail, int32_t *result)`	
Description	
1 Multiply accumulate (MAC) MLA T1, SMLAL T1, SMLALBB T1, SMLALD T1, UMAAL T1, UMLAL T1, SMMLA T1, SMLADX T1, SMLATT T1, SMLAWB T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Multiply subtract (MSB)** MLS T1, SMLSLD T1, SMMLSR T1, SMLSD T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_9.asm ファイル

Syntax	
void R_CPU_Diag9(const uint32_t forceFail, int32_t *result)	
Description	
1 Addition and subtraction SASX T1, SSAX T1, UASX T1, USAX T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Addition and halving** UHADD16 T1, UHADD8 T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Subtraction and halving** UHSUB16 T1, UHSUB8 T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **4 Addition, subtraction and halving** SHASX T1, SHSAX T1, UHASX T1, UHSAX T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **5 Dual multiplication** SMUAD T1, SMUSDX T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **6 Absolute difference** USAD8 T1, USADA8 T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_10.asm ファイル

Syntax
`void R_CPU_Diag10(const uint32_t forceFail, int32_t *result)`
Description
1 Saturating SSAT T1, SSAT16 T1, USAT T1, USAT16 T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Saturate addition** QADD T1, QADD16 T1, QADD8 T1, UQADD16 T1, UQADD8 T1, QDADD T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Saturate subtraction** QSUB T1, QSUB16 T1, QSUB8 T1, QDSUB T1, UQSUB16 T1, UQSUB8 T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **4 Saturate addition and subtraction** QASX T1, QSAX T1, UQASX T1, UQSAX T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。

Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_11.asm ファイル

Syntax	
`void R_CPU_Diag11(const uint32_t forceFail, int32_t *result)`	
Description	
1 Four basic arithmetic instructions test VADD T2, VSUB T2, VMUL T2, VNMUL T2, VDIV T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Absolute, compare, negative, minimum and maximum instructions test** VABS T2, VCMP T1, VCMPE T1, VNEG T2, VMAXNM T2, VMINNM T2 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Conditional select instructions test** 3-1 VSELGE T1, VSELGT T1, VSELEQ T1, VSELVS T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的に FAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_12.asm ファイル

Syntax	
`void R_CPU_Diag12(const uint32_t forceFail, int32_t *result)`	
Description	
1 Multiply accumulate (MAC) VMLA T2, VNMLA T1, VFMA T2, VFNMA T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Multiply subtract (MSB)** VMLS T2, VNMLS T1, VFMS T2, VFNMS T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **3 Square root** VSQRT (minus) T1, VSQRT (zero) T1, VSQRT (plus) T1 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_13.asm ファイル

Syntax	
void R_CPU_Diag13(const uint32_t forceFail, int32_t *result)	
Description	
1 Floating-point rounding 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 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 **2 Floating-point conversion** 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 の各命令を実行して local signatur、global signature の期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_14_1.asm ファイル

Syntax
`void R_CPU_Diag14_1(const uint32_t forceFail, int32_t *result)`

Description

1 VPOP T2 and VPUSH T2

以下のテストを実施します。

－Verify VPOP after VPUSH using single register

手順は次の通りです。

1. R4、R5 レジスタに値を設定し、S1、S0 レジスタへデータを書き込み
2. VPUSH 命令で S1 レジスタをスタックへ退避を実施
3. VPOP 命令で S0 レジスタへスタックから復帰を実施
4. R5 と R4 を介して S0 と S1 レジスタの期待値との一致を確認。

－Verify VPOP after VPUSH using multiple registers

手順は次の通りです。

1. S4～S7 と S0～S4 にデータを設定
2. VPUSH 命令で S4～S7 レジスタをスタックへ退避を実施
3. VPOP 命令で S0～S4 レジスタへスタックから復帰を実施
4. R4-R7 を介して S0 と S4、S1 と S5、S2 と S6、S3 と S7 の各レジスタで期待値との一致を確認

2 VLDR/VLDM T2 and VSTR/VSTM T2

以下のテストを実施します。

－Verify VLDR after VSTR using single register

手順は次の通りです。

1. S1、S0 レジスタへデータを書き込み
2. VSTR 命令で S1 レジスタをスタックへストアを実施
3. VLDR 命令で S0 レジスタへスタックからロードを実施
4. R4,R5 を介して S0、S1 レジスタの期待値との一致を確認

－Verify VLDM after VSTM using multiple registers

手順は次の通りです。

1.S4～S7 と S0～S4 にデータを設定
2.VSTM 命令で S4～S7 レジスタをスタックへストアを実施
3.VLDR 命令で S0～S4 レジスタへスタックからロードを実施
4.R4-R7 を介して S0 と S4、S1 と S5、S2 と S6、S3 と S7 の各レジスタで期待値との一致を確認

期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。

Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_14_2.asm ファイル

Syntax

```
void R_CPU_Diag14_2(const uint32_t forceFail, int32_t *result)
```

Description

3.VMOV

以下のテストを実施します。

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

手順は次の通りです。

1. S0, R4 にそれぞれデータを設定
2. “VMOV　S0, R4”を実施
3. R5 を介して S0 と R4 の各レジスタで期待値との一致を確認

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

手順は次の通りです。

1. S0(= R5), R4 にそれぞれデータを設定
2. “VMOV　R4, S0”を実施
3. R5 を介して S0 と R4 の各レジスタで期待値との一致を確認

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

手順は次の通りです。

1. S0, S1, R5, R4 にそれぞれデータを設定
2. “VMOV　S0, S1, R4, R5”を実施
3. R6 を介して S0 と R4、S1 と R5 の各レジスタで期待値との一致を確認

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

手順は次の通りです。

1. S0(=R6), S1(=R7)にそれぞれデータを設定
2. “VMOV　R4, R5, S0, S1”を実施
3. R6, R7 を介して S0 と R4、S1 と R5 の各レジスタで期待値との一致を確認

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

手順は次の通りです。

1. S0(=R6), R4 にそれぞれデータを設定(R4 には下記 2.の期待値：#9 の浮動小数点形式を設定)
2. “VMOV.F32　S0, #9”を実施
3. R5 を介して S0 と R4 の各レジスタで期待値との一致を確認

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

手順は次の通りです。

1. S0(=R6), S1(=R4)にそれぞれデータを設定
2. “VMOV.F32　S0, S1”を実施
3. R5, R4 を介して S0 と S1 の各レジスタで期待値との一致を確認

4 VMRS

以下のテストを実施します。

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

手順は次の通りです。

1. R4, R5(=FPSCR)にそれぞれデータを設定
   (**{FPSCR N, Z, C, V} = {1, 1, 1, 1}となる設定値**)
2. "VMRS　R4, FPSCR"を実行
3. R4, R5 を介して R5 と FPSCR の各レジスタで期待値との一致を確認

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

手順は次の通りです。

1. R4, R5(=FPSCR)にそれぞれデータを設定
   (**{FPSCR N, Z, C, V} = {0, 0, 0, 0}となる設定値**)
2. "VMRS　R4, FPSCR"を実行

3. R4, R5 を介して R5 と FPSCR の各レジスタで期待値との一致を確認

－VMRS (FPSCR to APSR with {FPSCR N, Z, C, V} = {1, 1, 1, 1})
手順は次の通りです。
1. R4(=APSR), R5(=FPSCR)にそれぞれデータを設定
({FPSCR N, Z, C, V} = {1, 1, 1, 1}となる設定値)
2. "VMRS　APSR_nzcv, FPSCR"を実行
3. R4, R5 を介して APSR と FPSCR の各レジスタで期待値との一致を確認
＊APSR と FPSCR の N, Z, C, V フラグの値を一致確認

－VMRS (FPSCR to APSR with {FPSCR N, Z, C, V} = {0, 0, 0, 0})
手順は次の通りです。
1. R4(=APSR), R5(=FPSCR)にそれぞれデータを設定
({FPSCR N, Z, C, V} = {0, 0, 0, 0}となる設定値)
2. "VMRS　APSR_nzcv, FPSCR"を実行
3. R4, R5 を介して APSR と FPSCR の各レジスタで期待値との一致を確認
＊APSR と FPSCR の N, Z, C, V フラグの値を一致確認

5 VMSR
以下のテストを実施します。
－VMSR (general-purpose register to FPSCR with {APSR N, Z, C, V} = {1, 1, 1, 1})
手順は次の通りです。
1. R5(= FPSCR), R4 にそれぞれデータを設定
({FPSCR N, Z, C, V} = {1, 1, 1, 1}となる設定値)
2. "VMSR　FPSCR, R4"を実行
3. R5 と R4 介して、R5 と R4 が一致することを確認
＊FPSCR の N, Z, C, V フラグの値を一致確認

－VMSR (general-purpose register to FPSCR with {FPSCR N, Z, C, V} = {0, 0, 0, 0})
手順は次の通りです。
1. R5(= FPSCR), R4 にそれぞれデータを設定
({FPSCR N, Z, C, V} = {0, 0, 0, 0}となる設定値)
2. "VMSR　FPSCR, R4"を実行
3. R5 と R4 を比較し、R5 と R4 が一致することを確認
＊FPSCR の N, Z, C, V フラグの値を一致確認

期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を" resultTemp"へ設定します。

Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ：N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_15_1.asm ファイル

Syntax
void R_CPU_Diag15_1(const uint32_t forceFail, int32_t *result)

Description

1. Detecting “0” fixed fault for FPU status and control registers : FPU ステータスおよび制御レジスタの「0」固定障害の検出
R7, R8 を使用して FPSCR レジスタの該当ビットへ”1”を書き込み後(0xf7c0009f)、読み出しを実行し期待値との一致を確認する。("0”固定になっていないことの確認)

2. Detecting “1” fixed fault for FPU status and control registers : FPU ステータスおよび制御レジスタの「1」固定障害の検出
R7, R8 を使用して FPSCR レジスタの該当ビットへ”0”を書き込み後(0x00000000)、読み出しを実行し期待値との一致を確認する。("1”固定になっていないことの確認)

3. Detecting “0” fixed fault for FPU data registers : 単精度レジスタの「0」固定障害の検出
R7, R8 を使用して単精度レジスタ(S0-S31)の各レジスタ毎に”0xffffffff”を書き込み後、読み出しを実行し期待値との一致を確認する。("0”固定になっていないことの確認)

4. Detecting “1” fixed fault for FPU data registers: 単精度レジスタの「1」固定障害の検出
R7, R8 を使用して単精度レジスタ(S0-S31)へ各レジスタ毎に”0x00000000”を書き込み後、読み出しを実行し期待値との一致を確認する。("0”固定になっていないことの確認)

期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。

Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_15_2.asm ファイル

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 : 任意の 2 ビット間の単精度レジスタの結合障害の検出 以下のテストを実施します。 **－Nearest neighbor coupling(Test pattern : 0x55555555)** **－Next nearest neighbor coupling(Test pattern : 0x33333333)** **手順は次の通りです。** 1. R7 に上記の各テストパターンを設定 2. R7、R8 を使用して単精度レジスタ(S0-S31)の各レジスタ毎にテストパターンを書き込み後、読み出しを実施 3. R7 と R8 の各レジスタと期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_15_3.asm ファイル

Syntax	
void R_CPU_Diag15_3(const uint32_t forceFail, int32_t *result)	
Description	
5. Detecting coupling fault for general purpose registers between any two bits : 任意の 2 ビット間の単精度レジスタの結合障害の検出 以下のテストを実施します。 **－4-fold neighbor coupling(Test pattern : 0x0f0f0f0f)** **－8-fold neighbor coupling(Test pattern : 0x00ff00ff)** **手順は次の通りです。** 1. R7 に上記の各テストパターンを設定 2. R7、R8 を使用して単精度レジスタ(S0-S31)の各レジスタ毎にテストパターンを書き込み後、読み出しを実施 3. R7 と R8 の各レジスタと期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
const uint32_t forceFail	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
int32_t *result	テスト結果の格納先ポインタ
Output Parameters	
int32_t *result	テスト結果(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	
5. Detecting coupling fault for general purpose registers between any two bits : 任意の 2 ビット間の単精度レジスタの結合障害の検出 以下のテストを実施します。 **－16-fold neighbor coupling(Test pattern : 0x0000ffff)** **手順は次の通りです。** 1. R7 に上記の各テストパターンを設定 2. R7、R8 を使用して単精度レジスタ(S0-S31)の各レジスタ毎にテストパターンを書き込み後、読み出しを実施 3. R7 と R8 の各レジスタと期待値との一致を確認する。 期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を” resultTemp”へ設定します。	
Input Parameters	
`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ： N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

■ r_cpu_diag_15_5.asm ファイル

Syntax

```
void R_CPU_Diag15_5(const uint32_t forceFail, int32_t *result)
```

Description

6. Detecting coupling fault for FPU data registers between any two registers : 任意の 2 つのレジスタ間の単精度レジスタの結合障害の検出

以下のテストを実施します。

－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

手順は次の通りです。

1. R4～R11 に各々テストパターンを設定し、R4 を S0 へ、R5 を S1 へ、…、R11 を S7 へ転送
2. S0 を S16 へ、S1 を S17 へ、…、S7 を S23 へ転送
3. S16～S23 を R12 を介して読み出し、転送元 R4～R11 と期待値との一致を確認

－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

手順は次の通りです。

1. R4～R11 に各々テストパターンを設定し、R4 を S9 へ、R5 を S10 へ、…、R11 を S8 へ転送
2. S9 を S24 へ、S10 を S25 へ、…、S8 を S31 へ転送
3. S24～S31 を R12 を介して読み出し、転送元 R4～R11 と期待値との一致を確認

期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を" resultTemp"へ設定します。

Input Parameters

`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ：N/A
`int32_t *result`	テスト結果の格納先ポインタ

Output Parameters

int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_15_6.asm ファイル

Syntax

```
void R_CPU_Diag15_6(const uint32_t forceFail, int32_t *result)
```

Description

6. Detecting coupling fault for FPU data registers between any two registers : 任意の 2 つのレジスタ間の単精度レジスタの結合障害の検出

以下のテストを実施します。

－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

手順は次の通りです。

1. R4～R11 に各々テストパターンを設定し、R4 を S18 へ、…、R9 を S23 へ、R10 を S16 へ、R11 を S17 へ転送
2. S18 を S0 へ、…、S23 を S5 へ、S16 を S6 へ、S17 を S7 へ転送
3. S0～S7 を R12 を介して読み出し、転送元 R4～R11 と期待値との一致を確認

－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

手順は次の通りです。

1. R4～R11 に各々テストパターンを設定し、R4 を S27 へ、…、R8 を S31 へ、R9 を S24 へ、R10 を S25 へ、R11 を S26 へ転送
2. S27 を S8 へ、…、S31 を S12 へ、S24 を S13 へ、S25 を S14 へ、S26 を S15 へ転送
3. S8～S15 を R12 を介して読み出し、転送元 R4～R11 と期待値との一致を確認

期待値と一致の場合は PASS(0x0001)、不一致の場合は FAIL(0x0000)を" resultTemp"へ設定します。

Input Parameters

`const uint32_t forceFail`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0　　：強制的にFAIL Others ：N/A
`int32_t *result`	テスト結果の格納先ポインタ

Output Parameters

int32_t *result	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
NONE	N/A

■ r_cpu_diag_16.asm ファイル

Syntax	
`void R_CPU_Diag16(const uint32_t forceFail, int32_t *result)`	
Description	
汎用レジスタ(R0-12、R14)に対して WALKPAT アルゴリズムによる CPU レジスタテスト処理を実施します。(アルゴリズムについては、**1.3.3(2)** WALKPAT 参照) テスト結果を" resultTemp"へ格納します。(0 : FAIL / 1 : PASS) 使用するテストパターンは以下のとおりです。 ◆テストパターン 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`	強制 FAIL オプション 0 に設定すると、関数は強制的に失敗します。 0 : 強制的にFAIL Others : N/A
`int32_t *result`	テスト結果の格納先ポインタ
Output Parameters	
`int32_t *result`	テスト結果(0 : FAIL / 1 : PASS)
Return Values	
`NONE`	N/A

1.2 ROM

この章では、CRC 演算器を使用した ROM /フラッシュメモリテストについて説明します。
(参照：*IEC 60730-1:2013 + A1 : 2015+A2:2020 Annex H –- H2.19.4.2 CRC – Double Word*)
CRC は、メモリの内容に基づいて単一ワードまたはチェックサムを生成する不具合/エラー制御方法です。

CRC チェックサムは、メッセージビットストリームのビット繰り上がりなし（減算ではなく XOR を使用）n 次の多項式の係数を表す、長さ n+1 の定義済み（short）ビットストリームによるバイナリ除算の剰余です。除算の前に、n 個のゼロがメッセージストリームに追加されます。CRC は、バイナリハードウェアへの実装が簡単で数学的にも分析しやすいため、よく使用されます。

ROM テストは、ROM 内容の CRC 値を予め生成して保存することで実現できます。ROM セルフテストでは、同じ CRC アルゴリズムを用いて新たに CRC 値を生成し、保存しておいた CRC 値と比較します。この手法は、すべての 1 ビットエラーと高い割合のマルチビットエラーを認識します。

他の CRC ジェネレータによって事前に生成された CRC 値と比較する場合、基本的な CRC アルゴリズムが同じであっても、計算結果が同一にならない要因がいくつかあるため注意が必要です。たとえば、データをアルゴリズムに供給する順序、使用されるルックアップテーブルで想定されるビット順序、あるいは実際の CRC 値のビットに必要な順序の組み合わせ等です。システムがビッグエンディアンとリトルエンディアンの両方に対応する場合も問題になります。また、一部のデバッガは ROM 上でのソフトウェアブレイクを実現するものがあり、その場合はデバッグ中に ROM の内容が書き換えられてしまう可能性があります。

参照用 CRC 値の計算方法は、使用するツールチェーンで異なります。詳しい手順は、2. 使用例の **2.2 ROM** を参照ください。

1.2.1 CRC32 アルゴリズム

RA MCU には、CRC32 アルゴリズムのサポートが可能な CRC（巡回冗長検査）演算器が内蔵されています。テストソフトウェアは、32 ビット CRC32 を生成するように CRC 演算器を設定します。

- 多項式 = 0x04C11DB7 ($x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$)
- 幅 = 32 bit
- 初期値 = 0xFFFFFFFF
- h’FFFFFFFF との XOR 演算結果が CRC に出力される

1.2.2 マルチチェックサム(Multi Checksum)

ROM テストでは、テスト対象の ROM 領域を図 1-1　Code FLASH block diagram on ROM test 図 1-1 のように 64K バイトに分割し、CRC を計算して特定の領域に格納します。

なお、 :本サンプルソフトでは コードフラッシュメモリ 1MB 製品のため、ビルド時に 0xFFFC0～0xFFFFF のアドレスに格納します。

また、セルフテストライブラリでは、64K バイトごとに処理を分割し、CRC 演算処理後、上記特定領域に保存された CRC 値との一致確認を行い、ROM テスト結果を判定します。

サンプルプロジェクトの「RA_SelfTests.c」を編集することで、分割処理の有効設定を変更することができます。（詳細は **2.2.2 マルチチェックサム対応設定**を参照ください。）

サンプルプロジェクトでは、チェックサム格納領域を除くコード FLASH 領域をテスト対象としています。

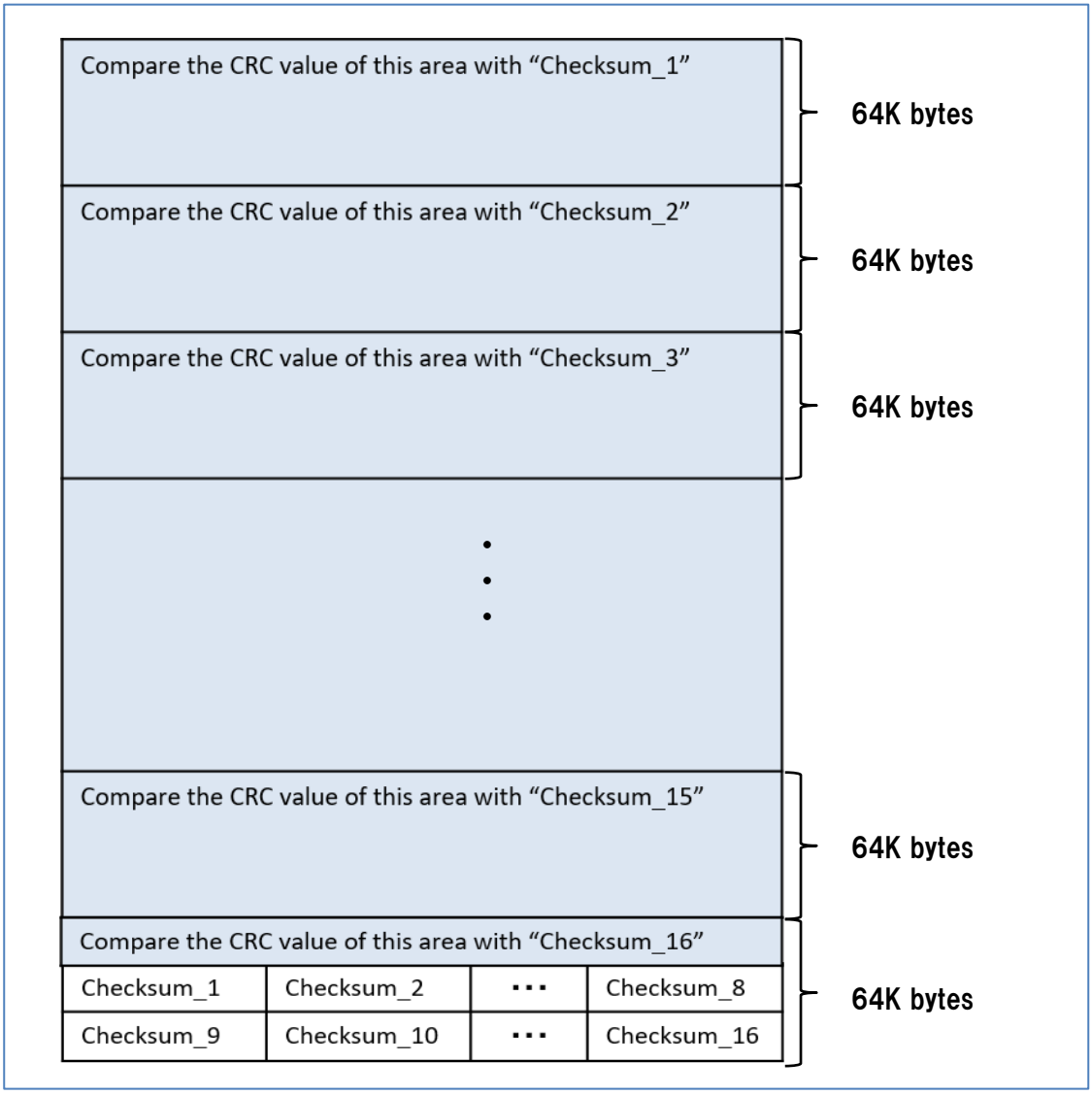

図 1-1　Code FLASH block diagram on ROM test

1.2.3　CRC ソフトウェア API

このセクションの関数は、CRC 値を計算し、ROM に格納されている値と比較してその正確性を検証するために使用されます。

すべてのソースは ANSI C で記述されます。renesas.h ヘッダファイルには、RA MCU のレジスタ定義が含まれます。

表 1-17 CRC ソフトウェア API ソースファイル

ファイル名	
crc.h	ROM テスト API 関数の定義
crc_verify.h	ROM テスト API 関数の定義
crc.c	ROM テスト実装部
CRC_Verify.c	ROM テスト実装部

■ CRC_Verify.c ファイル

Syntax	
bool_t CRC_Verify(const uint32_t ui32_NewCRCValue, const uint32_t ui32_AddrRefCRC)	
Description	
この関数は、参照 CRC が格納されているアドレスを提供することにより、新しい CRC 値を参照 CRC と比較します。	
Input Parameters	
const uint32_t ui32_NewCRCValue	計算された新しい CRC 値
const uint32_t ui32_AddrRefCRC	32 ビット参照 CRC 値が格納されるアドレス
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : True = テストパス、0 : False = テスト失敗

■ crc.c ファイル

Syntax	
void CRC_Init(void)	
Description	
CRC モジュールを初期化します。この関数は、他の CRC 関数を呼び出す前に呼び出す必要があります。	
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	
この関数は、単一の指定されたメモリ領域の CRC を計算します。	
Input Parameters	
const uint32_t* pui32_Data	テストするメモリの開始を指すポインタ
uint32_t ui32_Length	ロングワード単位のデータの長さ
Output Parameters	
NONE	N/A
Return Values	
Uint32_t	計算された CRC32 値

以下の関数は、メモリ領域を単純に開始アドレスと長さで指定できない場合に使用されます。それらは範囲/セクションにメモリ領域を追加する方法を提供します。これは、関数 CRC_Calculate が 1 回の関数呼び出しで時間がかかり過ぎる場合にも使用できます。

■ crc.c ファイル

Syntax	
void CRC_Start(void)	
Description	
データの受信を開始するためのモジュールを準備します。関数 CRC_AddRange を使用する前にこれを 1 回呼び出します。	
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	
複数のアドレス範囲で構成されるデータの CRC を計算する場合は、CRC_Calculate ではなくこの関数を使用します。最初に CRC_Start を呼び出し、次に必要なアドレス範囲ごとに CRC_AddRange を呼び出し、その後 CRC_Result を呼び出して CRC 値を取得します。	
Input Parameters	
const uint32_t* pui32_Data	テストするメモリ範囲の先頭を指すポインタ
uint32_t ui32_Length	ロングワード単位のデータの長さ
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
uint32_t CRC_Result(void)	
Description	
CRC_Start が呼び出され、CRC_AddRange 関数を使用して追加された、すべてのメモリ範囲に対する CRC 値を計算します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
uint32_t	計算された CRC32 の値

1.3 RAM

この章では、RAMテストと使用する2つのテストアルゴリズムついて説明します。

RAMテストの目的は、MCU内蔵SRAMからランダムな永続的な障害を検出することです。

RAMテストの主な機能は次のとおりです。

- スタックを含むメモリ全体のチェック。
- テストのブロックごとの実装
- 2つのテストアルゴリズムをサポート（Extend March-C-、WALKPAT）
- 2つのテストタイプ（破壊/非破壊テスト）をサポートします

1.3.1 RAM ブロックの定義(RAM Block Configuration)

RAMテストのターゲットは、RAM領域のRAMブロックです。

テスト対象のRAM領域とRAMブロックは、**表 1-20** で説明されているディレクティブによって構成されます。

図 1-2 は、RAM領域0がnブロックでどのように分割されるかを示しています。ディレクティブはイタリックで示されています。

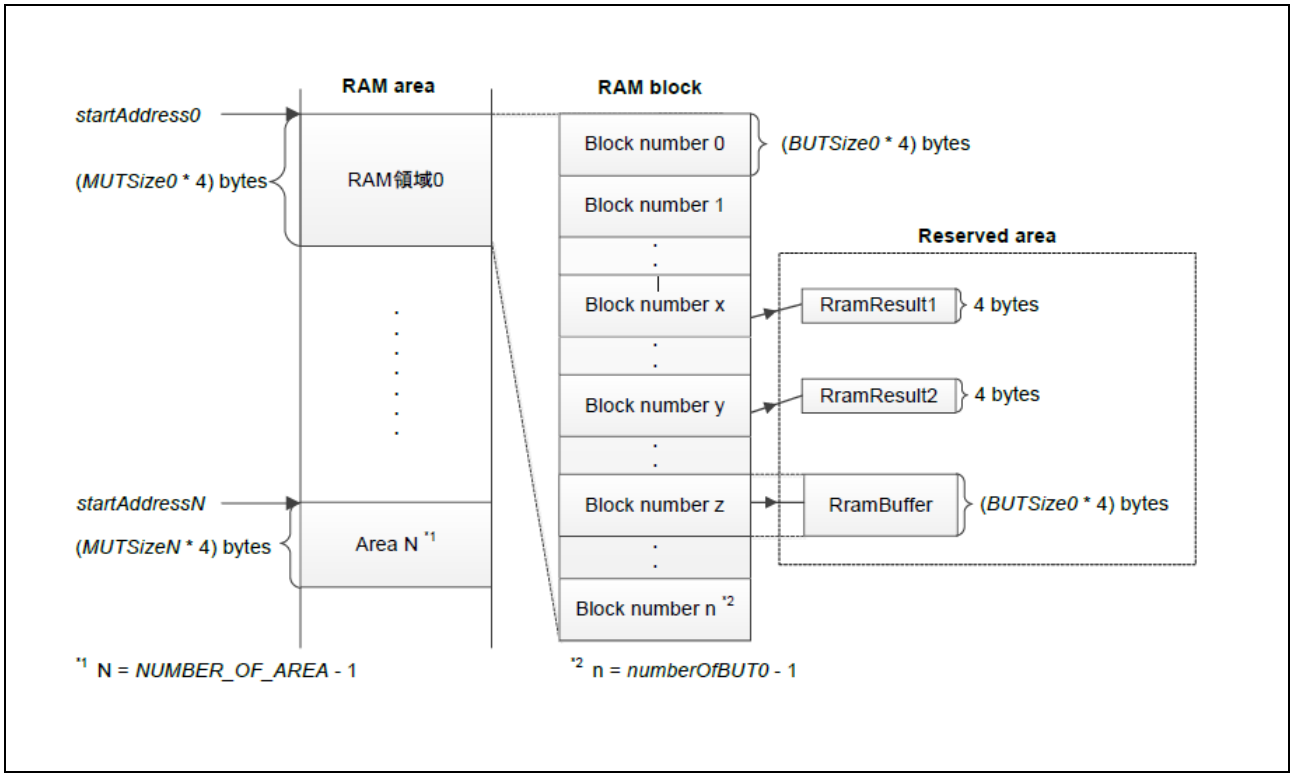

図 1-2　RAM Block Configuration (example)

1.3.2　予約領域について(Reserved Area)

RAM テストでは、ユーザーは次の予約領域を Secure 領域の RAM ブロックに割り当てる必要があります。

1. RAM テスト用バッファ (RramBuffer)

   非破壊検査では、テスト対象の RAM ブロックのデータ値が一時的にこのバッファに保存されます。

   ユーザーは、このバッファー用に特定の RAM ブロックを予約する必要があります。

2. テスト結果変数 (RramResult1）
3. テスト結果変数 (RramResult2)

テスト結果変数は、Secure 領域内に割り当てられ、2 つの異なる RAM ブロックに割り当てられます。テスト結果の 2 つのコピーを 2 つの異なるブロックに許可することにより、いずれかの変数を障害のあるブロックに格納できない場合でも、障害を検出できます。

予約領域は、このソフトウェアで事前定義されています。

具体的には「fsp.ld」、「RA_SelfTests.c」、「r_ram_diag_config.h」の各ファイルで予約領域の関連項目（データ保存バッファ、結果変数）を定義します。

本サンプルソフトにおける各定義箇所の該当部分を下記に記載します。

◆「fsp.ld」ファイル内の該当定義部分(青字)

```
__tz_RAM_S = ORIGIN(RAM);
  .ram_test_buffers :
  {
    . = ORIGIN(RAM);
    . = ALIGN(4);
    __RramBuffer_start = .;
    KEEP(*(RAM_TEST_BUFFER*))
    __RramBuffer_stop = .;
  } > RAM
```

◆「RA_Self Tests.c」ファイル内の該当定義部分(青字)

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

/*The periodic RAM (including Stack) tests requires a buffer.  Locate it in its own section after(higher address than) the stacks.*/
//-->chg : Moved RramBuffer[], RramResult1, RramResult2 to Secure erea.
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")));
//<--chg : Moved RramBuffer[], RramResult1, RramResult2 to Secure erea.
//<-- For RAM test of Class-C
```

◆「r_ram_diag_config.h」ファイル内の該当定義部分(青字)

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

ビルド後に生成される MAP ファイルで「予約領域」の位置を確認できます。

◆生成されたセキュア領域の MAP ファイル(「RA6M4_sec.map」)の該当箇所。

```
.ram_test_buffers
            0x20000000     0x300
            0x20000000             . = ORIGIN (RAM)
            0x20000000             . = ALIGN (0x4)
            0x20000000             __RramBuffer_start = .
 *(RAM_TEST_BUFFER*)
 RAM_TEST_BUFFER
            0x20000000     0x300 ./SelfTestLib/src/RA_SelfTests.o
            0x20000000             RramBuffer                    → RAM Buffer for temporarilly saved data : RamBuffer[]
            0x20000100             RAM_Test_dummy1
            0x200001fc             RramResult1                   → result variables : RramResult1
            0x20000200             RAM_Test_dummy2
            0x200002fc             RramResult2                   → result variables : RramResult2
            0x20000300             __RramBuffer_stop = .
```

(Note)　配置されるアドレスは、ご使用になる ld ファイルの定義内容により異なります。

1.3.3　RAM テストアルゴリズム

(1)　Extended March C-

「Extended March C-」は、RAM テストに使用される March-C のテストアルゴリズムの 1 つです。

アルゴリズムを以下の図 1-3 に示します。

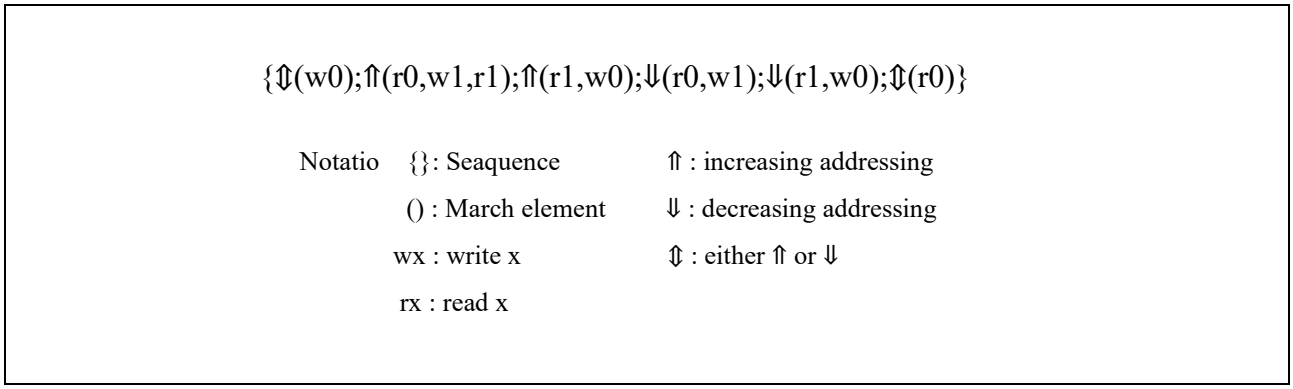

図 1-3 Extended March C- Algorithm

(2) WALKPAT

WALKPAT（Walking Pattern の略）は、RAM テストに使用されるテストアルゴリズムの 1 つです。アルゴリズムを以下の図 1-4 に示します。

```
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];
}
```

図 1-4 WALKPAT Algorithm

(3) アルゴリズムの特性

表 1-18 に、RAM テストで使用できる 2 つのテストアルゴリズムの特性を示します。

表 1-18 RAM テストアルゴリズムの特性 (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	✓
Complexity	11n	✓$2n^2$

n = the number of addressing cells of the memory(メモリのアドレス指定セルの数)

以下のアルゴリズムの説明は、1 ビットのワードメモリに関連していますが、m ビットのメモリにも適用できます。m ビットメモリは、次の方法で決定される回数だけ各アルゴリズムを繰り返すことで処理できます。

$$[\log 2\ m] + 1$$

このソフトウェアでは m = 32 ビットなので、アルゴリズムは 6 回繰り返され、次の 6 つの異なるパターンが適用されます。

#1: 00000000000000000000000000000000

#2: 00000000000000001111111111111111

#3: 00000000111111110000000011111111

#4: 00001111000011110000111100001111

#5: 00110011001100110011001100110011

#6: 01010101010101010101010101010101

1.3.4 RAM ソフトウェア API

RAM テストに関連するソフトウェア API ソースファイルは**表 1-19** の通りです。

RAM Test API を実行すると、RAM 領域の指定された 1 つの RAM ブロックがテストされます。

RAM 障害は、引数に出力された実行結果を確認することで検出できます。

コードをコンパイルする前に、テスト対象の RAM ブロックと予約領域を変更する必要があります（**1.3.2** を参照）。**表 1-20** に、構成のディレクティブを示します。ディレクティブは r_ram_diag_config.h にあります。

表 1-19 RAM ソフトウェア API ソースファイル

ファイル名	
r_ram_diag_config.h	RAM テストディレクティブの定義
r_ram_diag_config.inc	RAM テストの実行パターンの定義
r_ram_diag.c	RAM テスト API 関数の定義
r_ram_diag.h	RAM テスト API 関数の宣言
r_ram_marchc.asm	Extended March C-アルゴリズム関数の定義
r_ram_marchc.h	Extended March C-アルゴリズム関数の宣言
r_ram_walpat.asm	WALKPAT アルゴリズム関数の定義
r_ram_walpat.h	WALKPAT アルゴリズム関数の宣言

表 1-20 Directives for Software Configuration for RAM Test

ディレクティブ名	
NUMBER_OF_AREA	テスト対象のRAM領域の数（1～8）。 以下の場合を除き、1に設定してください。 — テスト中の複数のRAM領域が散発的な割り当てである — テスト中のRAMブロックが複数あり、各ブロックサイズが同じではない
startAddressN *1	テスト中のRAM領域への開始アドレス
MUTSizeN *1	テスト対象のRAM領域のサイズ（N）（ダブルワード）
numberOfBUTN *1	テスト中のRAMブロックの数。
BUTSizeN *1	テスト対象のRAMブロックのサイズ（N）（ダブルワード） BUTSizeN = MUTSizeN / numberOfBUTNによって計算
RAM_BUFFER_SIZE	テスト中のバッファ（RramBuffer）のサイズ（ダブルワード）

*1 : N = 0 ～ (NUMBER_OF_AREA – 1)

■ r_ram_diag.c ファイル

Syntax
`void R_RAM_Diag(uint32_t area, uint32_t index, uint32_t algorithm, uint32_t destructive)`

Description
この関数は RAM を検証します。 テスト結果は、結果変数(RramResult1, RramResult2)の戻り値で確認できます。 If Test result is PASS : RramResult1 = 1 and RramResult2 = 1 If Test result is FAIL : Other than above 次の順序で RAM テストを実行します。 1. 引数 area, index より RAM ブロックが有効なエリアかをチェックします。 2. マクロ関数(R_RAM_BLK_SADR, R_RAM_BLK_EADR)を使用して、テスト対象の RAM ブロックの開始アドレスと終了アドレスを算出します。 (sAdr、eAdr に算出した開始アドレス、終了アドレスを保存します。) 3. 引数 algorithm により該当するアルゴリズムの関数を呼び出します。 ・Extended March C-の場合(algorithm = RAM_ALG_MARCHC)：R_RAM_Diag_MarchC()関数 ・WALKPAT の場合(algorithm = RAM_ALG_WALPAT)：R_RAM_Diag_Walpat()関数 注意： 引数”destructive”によりデータの破壊テスト又は非破壊テストか選択されます。 （破壊テストの場合、テスト後に RAM ブロックは「0」にクリアされます。） 4. 呼び出された関数に戻ります。

Input Parameters	
`uint32_t area`	RAM 領域番号 ディレクティブ”NUMBER_OF_AREA”の値よりも小さくする必要があります。 値が無効な場合は 0（FAIL）を返します。
`uint32_t index`	“area”に設定された RAM エリアの RAM ブロックインデックス RAM ブロックインデックスは 0 から始まります。 ディレクティブ”numberOfBUTN”（**表 1-20** を参照)よりも小さくする必要があります。 値が無効な場合は 0（FAIL）を返します。
`uint32_t algorithm`	アルゴリズムを指定します。 0 (RAM_ALG_MARCHC) ： Extended March C- 1 (RAM_ALG_WALPAT) ： WALKPAT ※値が 0 以外の場合は”WALKPAT”
`uint32_t destructive`	メモリテストの種類を指定します 0 ：:**データ非破壊テスト** 1 ： **データ破壊テスト** ※無効な値が設定された場合、非破壊検査。 破壊テスト後、RAM ブロックは 0 にクリアされます。 注意： テストタイプに関係なく、バッファ付きのブロックの場合、RAM ブロックは常に 0 にクリアされます。

Output Parameters	
NONE	N/A

Return Values	
NONE	N/A

Syntax	
uint32_t R_RAM_Diag_GetVersion(void)	
Description	
この関数は、RAM テストソフトウェアのバージョン情報を返します。 バージョンは、”r_ram_diag.h”ファイルで定義	
Input Parameters	
NONE	N/A
Output Parameters	
uint32_t version	CPU テスト　ソフトウエアバージョン (0xXXXXYYYY → XXXX : Major, YYYY: Minor)
Return Values	
uint32_t	0xXXXXYYYY → XXXX : Major, YYYY: Minor

■ r_ram_marchc.asm ファイル

Syntax
`void R_RAM_Diag_MarchC(uint32_t start, uint32_t end, uint32_t destructive)`

Description

引数 start、end で指定された RAM ブロックに対して Extended March C-アルゴリズムによる RAM テスト処理を実施します。(アルゴリズムについては、**1.3.3(1)**参照)
非破壊テストの場合、指定された RamBuffer 領域へテスト領域の現在データを退避します。

テスト結果を以下へ格納します。

- RramResult1 (0 : FAIL / 1 : PASS)
- RramResult2 (0 : FAIL / 1 : PASS)

使用するテストパターンは以下のとおりです。(“r_ramdiag_config.inc”を参照)
◆テストパターン

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`	テスト対象ブロックの開始アドレス
`uint32_t end`	テスト対象ブロックの最終アドレス
`uint32_t destructive`	メモリテストの種類を指定します 0：**データ非破壊テスト** 1：**データ破壊テスト**
Output Parameters	
`RramResult1`	0 : FAIL / 1 : PASS
`RramResult2`	0 : FAIL / 1 : PASS
Return Values	
`NONE`	N/A

■ r_ram_walpat.asm ファイル

Syntax
`void R_RAM_Diag_walpat(uint32_t start, uint32_t end, uint32_t destructive)`

Description

引数 start、end で指定された RAM ブロックに対して WALKPAT アルゴリズムによる
RAM テスト処理を実施します。(アルゴリズムについては、**1.3.3(2)**参照)
非破壊テストの場合、指定された RamBuffer 領域へテスト領域の現在データを退避します。

テスト結果を以下へ格納します。

- RramResult1 (0 : FAIL / 1 : PASS)
- RramResult2 (0 : FAIL / 1 : PASS)

使用するテストパターンは以下のとおりです。(“r_ramdiag_config.inc”を参照)
◆テストパターン

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`	テスト対象ブロックの開始アドレス
`uint32_t end`	テスト対象ブロックの最終アドレス
`uint32_t destructive`	メモリテストの種類を指定します 0：**データ非破壊テスト** 1：**データ破壊テスト**
Output Parameters	
`RramResult1`	0 : FAIL / 1 : PASS
`RramResult2`	0 : FAIL / 1 : PASS
Return Values	
`NONE`	N/A

1.4　クロック

RA MCUは、クロック周波数精度測定回路（CAC）を備えています。CACは基準クロックで生成した時間内のターゲットクロックのパルスを数え、そのパルス数が許容範囲外の場合、割り込み要求を発生します。また、メインクロック発振器には、発振停止検出回路を備えています。

1.4.1　CAC によるメインクロック周波数の監視

メイン、SUB_CLOCK、HOCO、MOCO、LOCO、IWDTCLK、PCLKB のいずれか、または外部クロック CACREF 端子入力を基準クロックソースとして使用できます。

(a) 外部基準クロックを使用する場合

1. clock_monitor.h ファイルで、#define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK を定義します。
2. ターゲットクロックと基準クロックの周波数を Hz で提供してください。

(b) 内部クロックソースの 1 つを使用する場合

1. CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK が定義されていないことを確認します。
2. 参照クロックを必ず選択してください（ref_clock 入力パラメータを使用）。
3. ターゲットおよび基準クロックの周波数を Hz で提供してください。

メインクロックの周波数が実行時に構成された範囲から外れると、周波数エラー割り込みとオーバフロー割り込みの 2 種類の割り込みが生成されます。このモジュールのユーザは、これらの 2 種類の割り込みを有効にして処理する必要があります。割り込みのアクティブ化の例については、**2.4 章**を参照してください。許容周波数範囲は、以下を使用して調整できます。

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

内部のクロックを参照クロックに使用する場合、CAC 回路の参照クロック分周比（CACR2 レジスタの RCDS[1:0]）は、テスト関数内で 1/128 に固定されています。

ターゲットクロックの分周比（CACR1 レジスタの TCSS[1:0]）は、入力パラメータに基づき、テスト関数内で計算により 1/1, 1/4, 1/8, 1/32 から選択されます。ただし、どの分周比を選んでも、計算結果が 16 ビット幅の「CAC 上限/下限設定レジスタ」で設定可能な範囲内に収まらない場合はエラーとなります。

1.4.2　メインクロックの発振停止検出

RA MCU のメインクロック発振器には発振停止検出回路があります。メインクロックが停止すると、ノンマスカブル割り込み（NMI）が生成され、自動的に中速オンチップオシレータ（MOCO）に切り替わります。

ClockMonitor_Init 関数では、メインクロック発振器コントロールレジスタ（MOSCCR）のメインクロック発振器停止ビット（MOSTP）が 0（メインクロック発振器動作）の場合、以下のように発振停止検出を有効にし、NMI を許可します。

- 発振停止検出コントロールレジスタ（OSTDCR）
  - 発振停止検出機能有効ビット（OSTDE）：有効
  - 発振停止検出割込み許可ビット（OSTDIE）：許可
- ICU ノンマスカブル割り込みイネーブルレジスタ（NMIER）
  - 発振停止検出割込み許可ビット（OSTEN）：許可

発振停止で NMI が発生した場合、ユーザは NMI 割り込みを処理し、NMISR.OSTST ビット（発振停止検出割り込みステータスフラグ）をチェックする必要があります。

1.4.3 CLock ソフトウェア API

Clock テストに関連するソフトウェア API ソースファイルは**表 1-21** の通りです。

表 1-21 Clock ソースファイル

ファイル名	
clock_monitor.h	Clock テスト API 関数の宣言
clock_monitor.c	Clock テスト実装部

テストモジュールは、renesas.h **ヘッダファイル**を使用してペリフェラルレジスタにアクセスします。

■ clock_monitor.c ファイル

ClockMonitor_Init 関数には 2 つのバージョンがあります。

(a) 外部基準クロックを使用する場合の ClockMonitor_Init 関数
(CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK が定義されているとき)

Syntax	
void ClockMonitor_Init(clock_source_t target_clock, uint32_t MainClockFrequency, uint32_t ExternalRefClockFrequency, CLOCK_MONITOR_CACREF_PIN ePin, CLOCK_MONITOR_ERROR_CALL_BACK CallBack)	
Description	
1. CAC モジュールを使用して、CACREF 端子の入力を基準クロックとして、target_clock 入力パラメータで選択したターゲットクロックの監視を開始します。 2. SW は CACREF 端子を選択できます（詳しくは、**2.4 クロック**を参照ください）。システム構成に基づいて端子を選択するのはユーザの責任です。 3. 発振停止検出を有効にし、検出された場合に生成される NMI を構成します。	
Input Parameters	
clock_source_t target_clock	・CAC が監視するターゲットクロック。 ・クロックは、メインクロック、サブクロック、HOCO クロック、MOCO クロック、LOCO クロック、IWDTCLK クロック、および PCLKB クロックのいずれかです。
uint32_t MainClockFrequency	ターゲットクロックの周波数（単位：Hz） （パラメータは MainClockFrequency となっていますが、設定するのは target_clock で指定したターゲットクロックの周波数です。）
uint32_t ExternalRefClockFrequency	外部基準クロック（CACREF 入力端子）の周波数（単位：Hz）
CLOCK_MONITOR_CACREF_PIN ePin	CACREF に使用するピン
CLOCK_MONITOR_ERROR_CALL_BACK CallBack	ターゲットクロックが許容範囲外の場合、またはこの関数で入力パラメータから正しく CAC 回路を構成できなかった場合に呼び出される関数
Output Parameters	
NONE	N/A

Return Values	
NONE	N/A

(b) 基準クロックに内部クロックソースの 1 つを使用する場合の ClockMonitor_Init 関数 (CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK が定義されていない場合)

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. CAC モジュールを使用して、ref_clock 入力パラメータで選択した内部クロックを基準クロックとして、target_clock 入力パラメータで選択したターゲットクロックの監視を開始します。
2. 発振停止検出を有効にし、検出された場合に生成される NMI を構成します。

Input Parameters	
clock_source_t target_clock	・CAC が監視するターゲットクロック。 ・クロックは、メインクロック、サブクロック、HOCO クロック、MOCO クロック、LOCO クロック、IWDTCLK クロック、および PCLKB クロックのいずれかです。
clock_source_t ref_clock	・ターゲットクロック監視のために使用する基準クロック。 ・クロックはメインクロック、サブクロック、HOCO クロック、MOCO クロック、LOCO クロック、IWDTCLK クロック、または PCLKB クロック、のいずれかです。
uint32_t target_clock_frequency	ターゲットクロック周波数（単位：Hz）
uint32_t ref_clock_frequency	基準クロック周波数（単位：Hz）
CLOCK_MONITOR_ERROR_CALL_BACK CallBack	ターゲットクロックが許容範囲外の場合、またはこの関数で入力パラメータから正しく CAC 回路を構成できなかった場合に呼び出される関数
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
extern void cac_ferrf_isr(void)	
Description	
CAC 周波数エラー割り込みハンドラ。 ClockMonitor_Init 関数で登録されたコールバック関数を呼び出します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
extern void cac_ovff_isr(void)	
Description	
CAC オーバフローエラー割り込みハンドラ。 ClockMonitor_Init 関数で登録されたコールバック関数を呼び出します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
bool_t CAC_Err_Detect_Test(void)	
Description	
電源投入時に CAC 機能による周波数エラー検出とオーバフローエラー検出による割り込みが正常に動作していることを確認します。 一定時間内(ソフトウエアループによるカウント)に各割り込み発生が確認できた場合、”TRUE”を返します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : TRUE = PASS(各割り込み発生を確認)、0 : FALSE = FAIL(確認できず)

1.5　独立ウォッチドッグタイマ（IWDT）

ウォッチドッグタイマは、異常なプログラムの実行を検出するために使用されます。プログラムが期待どおりに実行されていない場合、ソフトウェアによるウォッチドッグタイマ更新が必要なタイミングで行われないため、エラーを検出します。

これには、RA MCU の独立ウォッチドッグタイマ（IWDT）モジュールが使用されます。ウィンドウ機能が含まれているため、指定した時間の直前ではなく、指定したウィンドウ内で更新を行う必要があります。エラーが検出された場合、内部リセットまたはノンマスカブル割り込み（NMI）を生成するように構成できます。

IWDT のすべての構成は、「オプション設定メモリ」内のオプション機能選択レジスタ 0（OFS0）で行います（構成の例については、**2.5 章**を参照）。オプション設定メモリとは、リセット後のマイコンの状態を選択するために利用可能な一連のレジスタのことで、コードフラッシュの領域に配置されます。

IWDT がリセットを引き起こしたかどうかを判断するために、リセット後に使用する関数が提供されています。

テストモジュールは、renesas.h ヘッダファイルを使用してペリフェラルレジスタにアクセスします。

1.5.1 IWDT ソフトウェア API

IWDT テストに関連するソフトウェア API ソースファイルは**表 1-22** の通りです。

表 1-22 独立ウォッチドッグタイマソースファイル

ファイル名	
iwdt.h	IWDT テスト API 関数の宣言
iwdt.c	IWDT テスト実装部

Syntax	
void IWDT_Init (void)	
Description	
独立ウォッチドッグタイマを初期化します。この関数を呼び出した後は、ウォッチドッグタイマエラーを防ぐために、IWDT_kick 関数を正しい時間に呼び出す必要があります。 注： 割り込みを生成するように構成されている場合、これはノンマスカブル割り込み（NMI）になります。これは NMISR.IWDTST フラグをチェックするユーザコードで処理する必要があります。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
void IWDT_Kick(void)	
Description	
ウォッチドッグタイマのカウントをリフレッシュします。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
bool_t IWDT_DidReset(void)	
Description	
IWDT がタイムアウトしたか、正しく更新されなかった場合は True を返します。これは、ウォッチドッグタイマがリセットを引き起こしたかどうか判断するために、リセット後に呼び出すことができます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	ウォッチドッグタイマがタイムアウトした場合は True、それ以外の場合は False

Syntax	
bool_t IWDT_Err_Detect_Test(void)	
Description	
電源投入時に IWDT 機能のカウンタアンダーフロー検出による割り込みが正常に動作していることを確認します。 一定時間内(ソフトウエアループによるカウント)に IWDT アンダーフローによる NMI 割り込み発生が確認できた場合、”TREU”を返します。 f_IWDT_ERROR_TEST を”1”に設定し、一定時間内に f_IWDT_ERROR_TEST が”0”になったことで判定します。 なお、NMI_Handler_callback()内で IWDT アンダーフロー／リフレッシュエラー割り込みステータスフラグが”1”の場合に f_IWDT_ERROR_TEST を”0”に設定する処理をユーザーで作成する必要があります。 詳細は、”**2.5 独立ウォッチドッグタイマ（IWDT）**”を参照ください。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	1 : TRUE = PASS(NMI 割り込み発生を確認)、0 : FALSE = FAIL(確認できず)

2. 使用例(Example Usage)

このセクションでは、アプリケーションソフトウェアにセルフテストライブラリを適用する方法に関する、いくつかの有用な提案をユーザに提供します。
セルフテストは次の 2 つのパターンに分けられます。

(a) 電源投入時のテスト

リセット後に一度実行されるテストです。これらはできるだけ早く実行する必要がありますが、特に起動時間が重要な場合は、すべてのテストを実行する前に初期化コードを実行して、たとえばより高速なメインクロックを選択できるようにすることもできます。

(b) 定期的なテスト

通常のプログラム操作を通じて定期的に実行されるテストです。このドキュメントでは、特定のテストを実行する頻度を判断することはできません。定期的なテストのスケジューリング方法は、アプリケーションの構造に応じてユーザが決定します。

以降のセクションでは、各テストの使用例を示します。

2.1 CPU

いずれかのCPUテストで障害が検出されると、CPU_Test_ErrorHandlerと呼ばれるユーザ指定の関数が呼び出されます。CPUのエラーは非常に深刻なので、この機能の目的は、ソフトウェアの実行に依存しない安全な状態にできるだけ早く到達することです。

2.1.1 電源投入時(Power-On)

CPUテストは、リセット後できるだけ早く実行する必要があります。

関数CPU_Test_ClassCを使用して、CPUテストを自動的に実行できます。

2.1.2 定期的(Periodic)

CPUを定期的にテストするには、電源投入テストと同様に、CPU_Test_ClassC関数を使用します。

定期的に呼び出すことでCPUテストを自動的に実行できます。

また、1回の関数呼び出しで実行されるテストをユーザは”r_cpu_diag_config.h”により選択できます。

2.1.3 CPUテストの事前準備

次にCPUテストの準備について説明します。
コードをコンパイルする前に、ディレクティブの設定によりCPUテストを構成します。
ディレクティブと各CPUテストの関係については、**表 1-15** を参照してください。
ディレクティブは、どのテストをコンパイルに含めるか、または除外するかを定義するために使用されます。
ディレクティブは、r_cpu_diag_config.hファイルに記載されています。
サンプルソフトは、すべてのCPUテストをビルドするように設定されています。
ディレクティブを "0"（テストから除外される）に設定すると、norm_null()という空の関数が実行されます。

例えば、CPUコアがCM33でFPUを使用していない場合、FPU関連のテストをCPU Testのコンパイルから除外することができます。(**表 1-15** の「BUILD_R_CPU_DIAG_11」から「BUILD_R_CPU_DIAG_15_6」までのディレクティブに「0」を設定)

次のページでは、CPUテストを構成するディレクティブの設定箇所を示します。

◆「r_cpu_diag_config.h」ファイル内の該当定義部分(青字)

以下の設定箇所で"1"を設定するとテスト実施対象、"0"を設定するとテスト実施対象外となります。

```
/**********************************************************************************
* 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 テストでは、テスト対象範囲の計算された CRC 値と、事前に保存されている参照 CRC 値を比較します。(32 ビット CRC32 多項式は「CRC-32」を使用します)

参照 CRC 値は、CRC 計算に含まれない ROM 領域に格納する必要があります。参照 CRC 値の計算方法は、開発環境によって異なります。

また、本サンプルソフトでは ROM テストの処理負荷軽減のため分割処理を行っており、Multi Checksum に対応しております。

RA MCU 内蔵の CRC モジュールは、CRC_Init 関数を呼び出して、使用する前に初期化する必要があります。分割して処理する場合は分割処理の初回のみ初期化してください。

2.2.1 事前の参照用 CRC 計算(Reference CRC Value Calculation in Advance)

GNU ツールには CRC の計算機能が付属しないため、以下に紹介する SRecord ツール（注）を使用して参照 CRC 値を計算します。ユーザは、このツールを利用して、予め参照用の CRC 値を ROM に書き込んでおき、セルフテストではこの値とテストで計算した値を比較します。

注：SRecord は、SourceForge のオープンソースプロジェクトです。詳細は下記を参照ください。

- SRecord Web Site (SRecord v1.64)
  http://srecord.sourceforge.net/

- CRC Checksum Generation with “SRecord” Tools for GNU and Eclips
  https://gcc-renesas.com/wiki/index.php?title=CRC_Checksum_Generation_with_%E2%80%98SRecord%E2%80%99_Tools_for_GNU_and_Eclipse

ダウンロードしたファイルを解凍すると、以下のプログラムが展開されます。

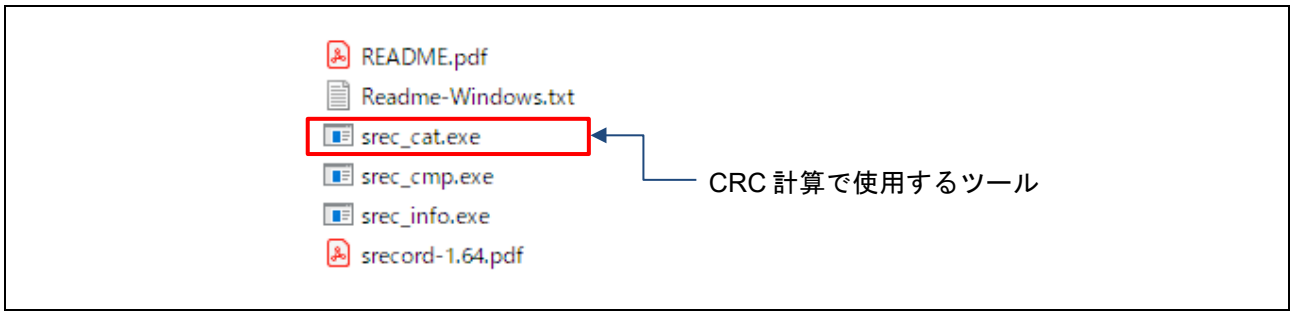

図 2-1 SRecord ツールの内容

プロジェクト及び SRecord ツールのフォルダ構成例を以下に示します。

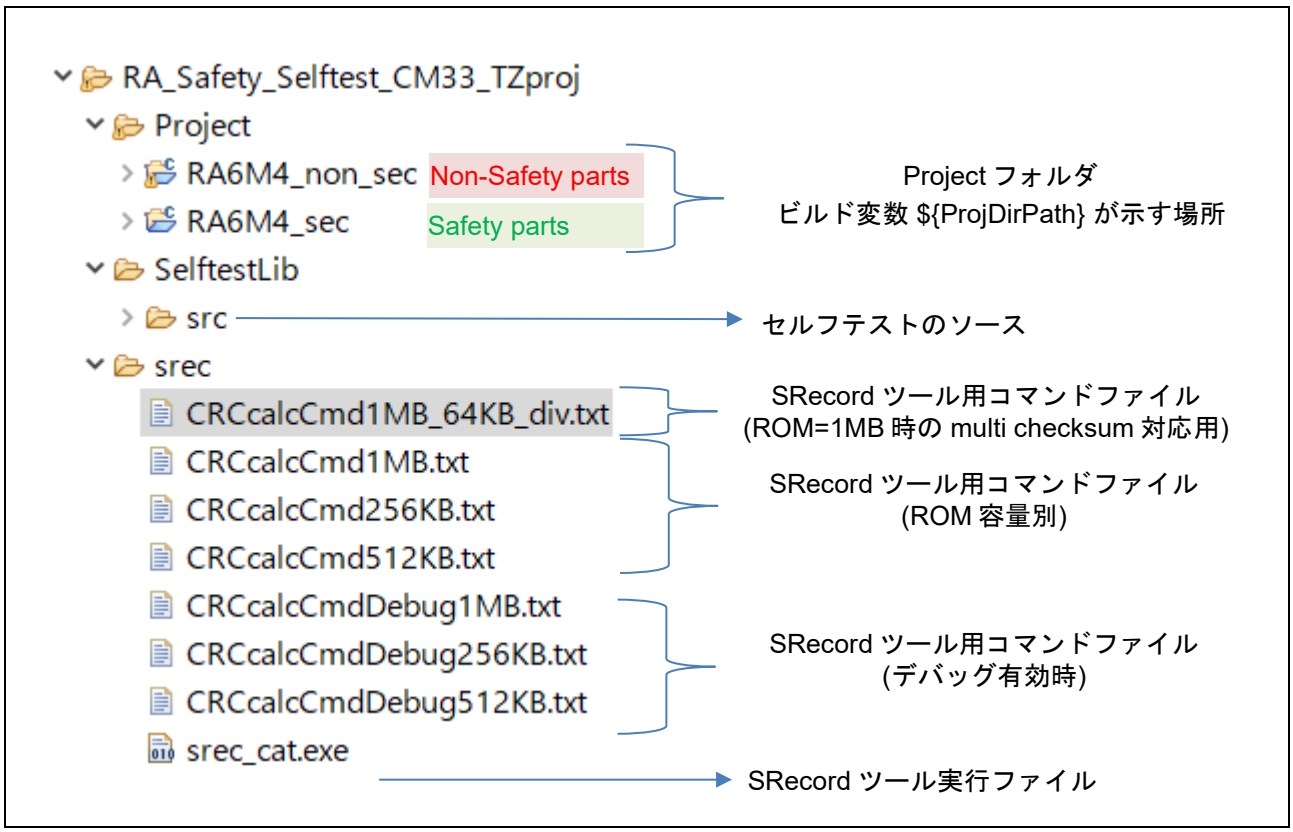

図 2-2　フォルダ構成例

TrustZone の Safety parts，Non-Safty parts を使用する場合、各プロジェクトのプロパティで設定が必要です。

◆”Safety parts”用プロジェクトでの設定

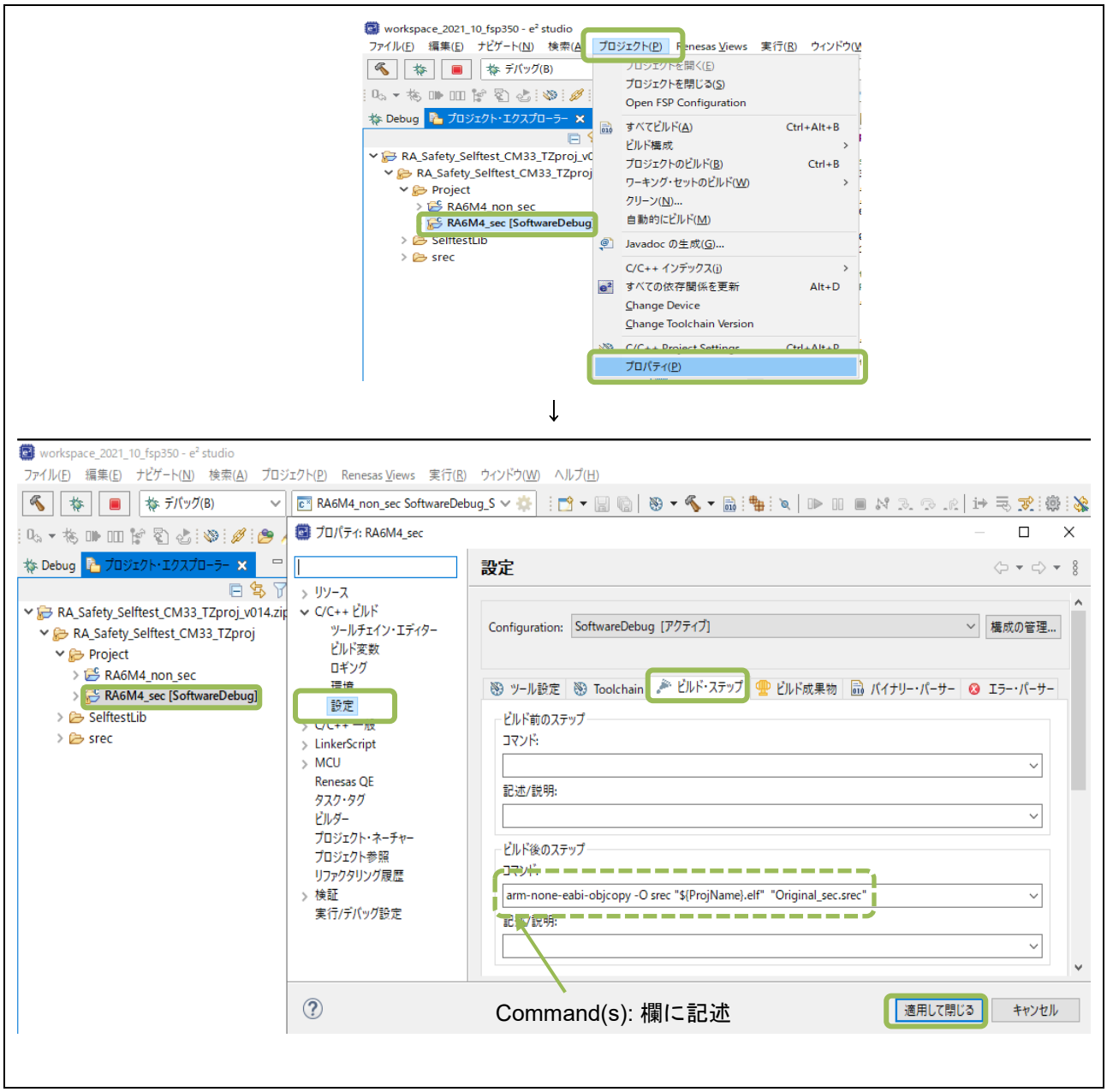

図 2-3 S レコードファイルの出力と SRecord ツールの起動(Safety Parts プロジェクトでの設定)

上図における「ビルドステップ(Build Steps)」タブの「ビルド後のステップ(Post-build steps)」で、以下のように記述します。＊()は e2studio 英語版時

■ コマンド（Command(s)）欄の記入例（改行せず 1 行に書きます）

```
arm-none-eabi-objcopy -O srec "${ProjName}.elf" "Original_sec.srec"
```

ここでは Safety parts で生成された*.elf より S レコードファイル” Original_sec.srec”の生成を行います。

◆”Non-Safety”用プロジェクトでの設定

e² studio の「プロジェクト」⇒「プロパティ」を開き「ビルド前のステップ」で、copy コマンドを使って Safety parts で作成した” Original_sec.srec”を Non-Safety 側のプロジェクトの該当フォルダへコピーします。

次に「ビルド後のステップ」で、objcopy コマンドを使って、Non-Safety parts で生成された*.elf ファイルから S レコードファイル” Original_non_sec.srec”を生成します。

更に” Original_non_sec.srec”と” Original_sec.srec”よりひとつの S レコードファイルに変換します。

変換後のファイル名を Original.srec とします。このファイルが、SRecord ツールの入力になります。

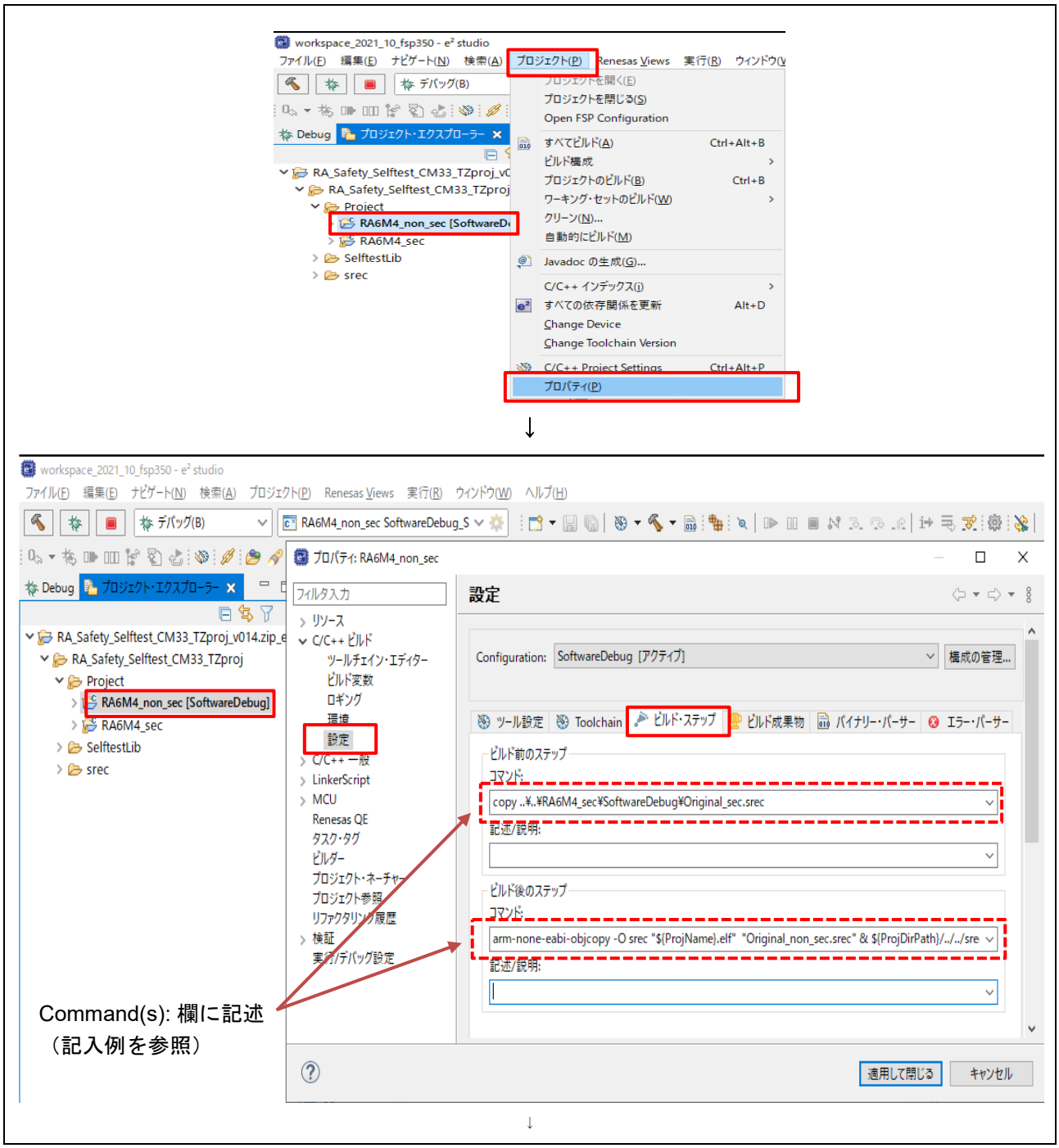

図 2-4 S レコードファイルの出力と SRecord ツールの起動(Non-Safety Parts プロジェクトでの設定)

図 2-4 における「ビルドステップ(Build Steps)」タブの「ビルド前のステップ(Pre-build steps)」及び「ビルド後のステップ(Post-build steps)」では、以下のように記述します。＊()は e2studio 英語版時

上図における「ビルドステップ Build Steps」タブの「ビルド前のステップ(Pre-build steps)」で、以下のように記述します。

■ 「ビルド前のステップ(Pre-build steps)」の Command(s):欄の記入例（改行せず 1 行に書きます）

```
copy ..\..\RA6M4_sec\SoftwareDebug\Original_sec.srec
```

copy コマンドを使って Safety parts で作成した” Original_sec.srec”を Non-Safety 側の該当フォルダへコピーします。

次に上図における「ビルドステップ(Build Steps)」タブの「ビルド後のステップ(Post-build steps)」で、以下のように記述します。

■「ビルド後のステップ(Post-build steps)」の Command(s):欄の記入例（改行せず 1 行に書きます）

[分割処理が有効時(DIV_AREA=1)]

```
arm-none-eabi-objcopy -O srec "${ProjName}.elf"  "Original_non_sec.srec" &
${ProjDirPath}/../../srec/srec_cat Original_non_sec.srec Original_sec.srec -o
Original.srec & ${ProjDirPath}/../../srec/srec_cat
@${ProjDirPath}/../../srec/CRCcalcCmd1MB_64KB_div.txt
```

[分割処理が無効時(DIV_AREA=0)]

```
arm-none-eabi-objcopy -O srec "${ProjName}.elf"  "Original_non_sec.srec" &
${ProjDirPath}/../../srec/srec_cat Original_non_sec.srec Original_sec.srec -o
Original.srec & ${ProjDirPath}/../../srec/srec_cat
@${ProjDirPath}/../../srec/CRCcalcCmd1MB.txt
```

上記 3 行目の"&"の前までが S レコードファイルの生成、3 行目の書式「**srec_cat @コマンドファイル**」が、srec_cat ツールの起動 になります。

コマンドファイルとして

「**CRCcalcCmd1MB_64KB_div.txt** 」(分割処理が有効時)

と

「**CRCcalcCmd1MB.txt**」(分割処理が無効時)

の記述例を以下に示します。

なお、分割処理の設定については”**2.2.2 マルチチェックサム対応設定**”を参照ください。

■ CRCcalcCmd1MB_64KB_div.txt ファイルの内容（例）

```
# CRC calculate
Original.srec                # Read srec file
-fill 0xFF 0x00000 0x100000  # 1MB ROM fill by 0xFF
#
-crop 0xF0000 0xFFFC0        # CRC calculate area (Test area 0xF0000 - 0xFFFC0  : 64KB-4) 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 0xE0000 0xF0000        # CRC calculate area (Test area 0xE0000 - 0xEFFFF : 64KB) for debug
-STM32-le  0x0FFFF8          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFF8.
-crop 0xFFFF8 0x100000       # Keep CRC area(0xFFFF8 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0xD0000 0xE0000        # CRC calculate area (Test area 0xD0000 - 0xDFFFF : 64KB) for debug
-STM32-le  0x0FFFF4          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFF4.
-crop 0xFFFF4 0x100000       # Keep CRC area(0xFFFF4 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0xC0000 0xD0000        # CRC calculate area (Test area 0xC0000 - 0xCFFFF : 64KB) for debug
-STM32-le  0x0FFFF0          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFF0.
-crop 0xFFFF0 0x100000       # Keep CRC area(0xFFFF0 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0xB0000 0xC0000        # CRC calculate area (Test area 0xB0000 - 0xBFFFF : 64KB) for debug
-STM32-le  0x0FFFEC          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFEC.
-crop 0xFFFEC 0x100000       # Keep CRC area(0xFFFEC - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0xA0000 0xB0000        # CRC calculate area (Test area 0xA0000 - 0xAFFFF : 64KB) for debug
-STM32-le  0x0FFFE8          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFE8.
-crop 0xFFFE8 0x100000       # Keep CRC area(0xFFFE8 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x90000 0xA0000        # CRC calculate area (Test area 0x90000 - 0x9FFFF : 64KB) for debug
-STM32-le  0x0FFFE4          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFE4.
-crop 0xFFFE4 0x100000       # Keep CRC area(0xFFFE4 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x80000 0x90000        # CRC calculate area (Test area 0x80000 - 0x8FFFF : 64KB) for debug
-STM32-le  0x0FFFE0          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFE0.
-crop 0xFFFE0 0x100000       # Keep CRC area(0xFFFE0 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x70000 0x80000        # CRC calculate area (Test area 0x70000 - 0x7FFFF : 64KB) for debug
-STM32-le  0x0FFFDC          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFDC.
-crop 0xFFFDC 0x100000       # Keep CRC area(0xFFFDC - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x60000 0x70000        # CRC calculate area (Test area 0x60000 - 0x6FFFF : 64KB) for debug
-STM32-le  0x0FFFD8          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFD8.
-crop 0xFFFD8 0x100000       # Keep CRC area(0xFFFD8 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x50000 0x60000        # CRC calculate area (Test area 0x50000 - 0x5FFFF : 64KB) for debug
-STM32-le  0x0FFFD4          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFD4.
-crop 0xFFFD4 0x100000       # Keep CRC area(0xFFFD4 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x40000 0x50000        # CRC calculate area (Test area 0x40000 - 0x4FFFF : 64KB) for debug
-STM32-le  0x0FFFD0          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFD0.
-crop 0xFFFD0 0x100000       # Keep CRC area(0xFFFD0 - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x30000 0x40000        # CRC calculate area (Test area 0x30000 - 0x3FFFF : 64KB) for debug
-STM32-le  0x0FFFCC          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFCC.
-crop 0xFFFCC 0x100000       # Keep CRC area(0xFFFCC - 0xFFFFF)
Original.srec                # Read srec file
#
-fill 0xFF 0x00000 0xF0000   # 0-0xF0000 ROM fill by 0xFF
-crop 0x20000 0x30000        # CRC calculate area (Test area 0x20000 - 0x2FFFF : 64KB) for debug
-STM32-le  0x0FFFC8          # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFC8.
-crop 0xFFFC8 0x100000       # Keep CRC area(0xFFFC8 - 0xFFFFF)
```

```
Original.srec                 # Read srec file
#
-fill 0xFF 0x00000 0xF0000    # 0-0xF0000 ROM fill by 0xFF
-crop 0x10000 0x20000         # CRC calculate area (Test area 0x10000 - 0x1FFFF : 64KB) for debug
-STM32-le  0x0FFFC4           # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFC4.
-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 0x10000         # CRC calculate area (Test area 0x0 - 0xFFFF : 64KB) for debug
-STM32-le  0x0FFFC0           # The algorithm used by the STM32 hardware unit is just a CRC32, and store CRC Value at 0xFFFC0.
-crop 0xFFFC0 0x100000        # Keep CRC area(0xFFFC0 - 0xFFFFF)
Original.srec                 # Read srec file
#
-fill 0xFF 0x000000 0x0FFFC0          # -fill 0xFF from 0x0 to 0xFFFC0
-Output addcrc.srec                   # Output of S-record file including CRC value
```

■ CRCcalcCmd1MB.txt ファイルの内容（例）

```
Original.srec                   # Read srec file
-fill 0xFF 0x00000 0x100000     # 1MB ROM fill by 0xFF
-crop 0x00000 0x0FFFFC          # CRC calculate area
-STM32-le  0x0FFFFC             # Calculate and output CRC value
-crop 0xFFFFC 0x100000          # Keep CRC area
Original.srec                   # Read srec file again
-fill 0xFF 0x000000 0x0FFFFC    # -fill 0xFF from 0x0 to 0xFFFFC
-Output addcrc.srec             # Output of S-record file including CRC value
```

デバイスにより ROM の容量が異なる場合は、アドレスの設定はデバイスに合わせて変更してください。また、デバッグを行う場合、デバッガによってはソフトウェアブレークのために ROM の内容を書き換えるものがあるので、その場合は演算の対象領域をデバッグ領域以外に設定する必要があります。

以上の操作で、プロジェクトフォルダ下のビルド構成フォルダ内に **addcrc.srec**（プログラムコードの後ろに CRC 演算結果を付加した S レコードファイル）ができるので、これをターゲットボードにダウンロードします。

プロジェクトツリーのトップで右クリックし、”デバッグ”　→　デバッグ”の構成 を選択します。

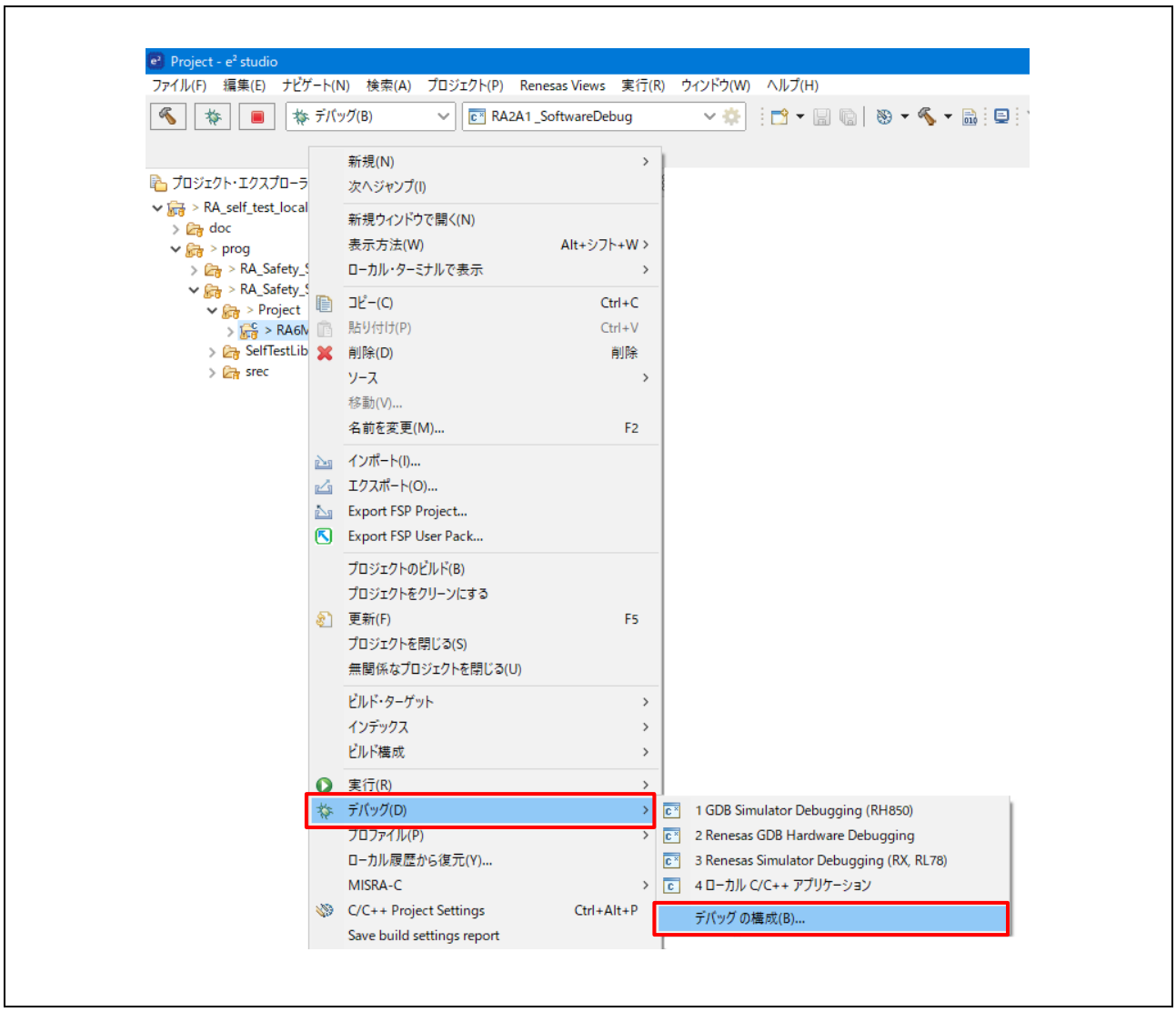

図 2-5 プロジェクトのデバッグ構成の選択

デバッグ構成のダイアログが表示されたら、Startup のタブを選び、使用するビルド構成を選択します。ELF ファイルからはシンボル情報だけを読み出し、addcrc.srec からは CRC 計算値を含むプログラムイメージを読み込むように設定します。

「デバッグ」ボタンを押下すると CRC 演算値がターゲットにダウンロードされます。

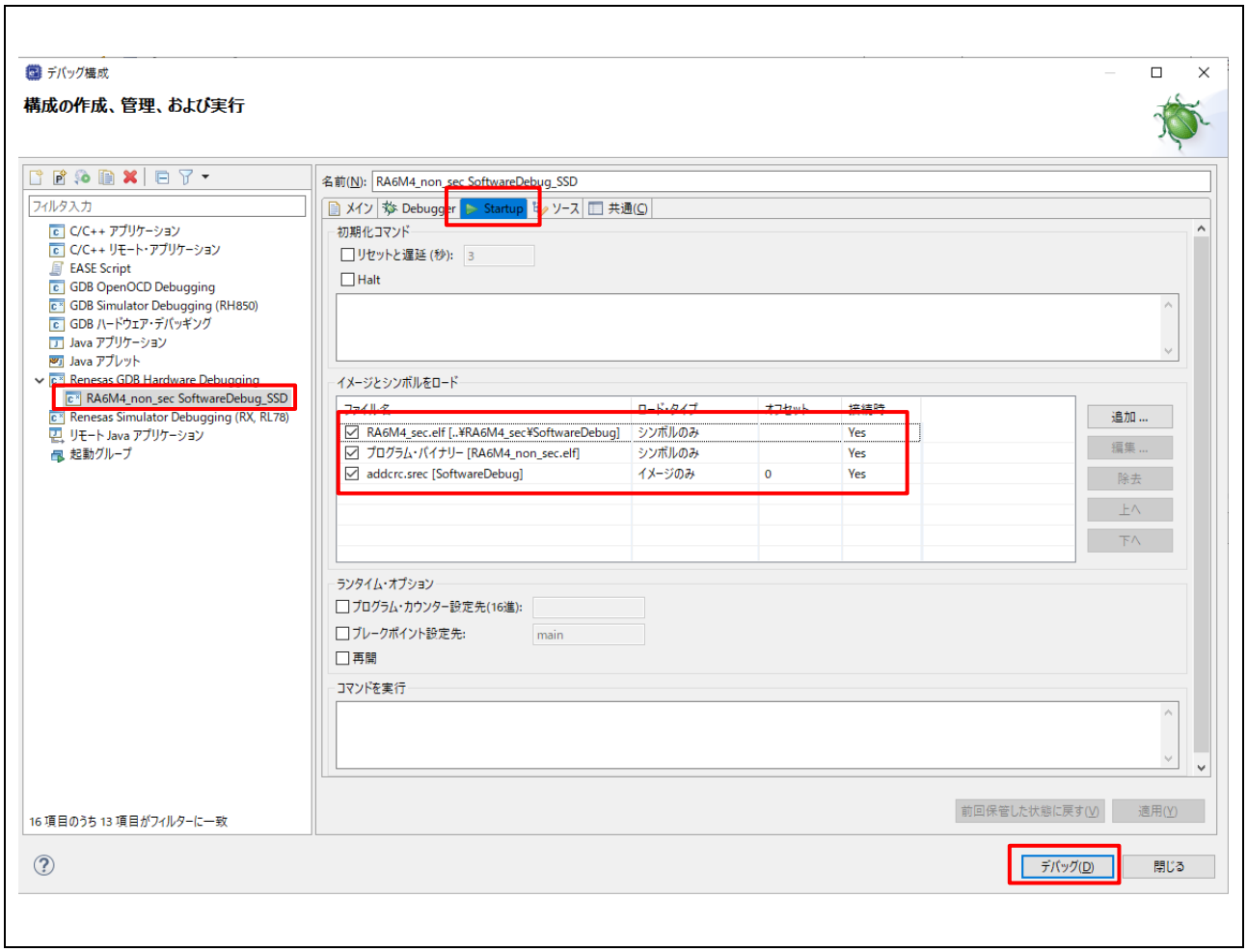

図 2-6　ロードイメージとシンボルの設定例

2.2.2　マルチチェックサム対応設定

1回のROMテストで全領域を行うには時間を要します。その対策として以下の設定で処理を分割することが可能です。

本サンプルソフト付属の”**RA_Self Tests.c**”を編集し、設定します。デフォルト設定は分割処理が有効です。

◆サンプルソフト(for RA6M4)の「RA_SelfTests.c」ファイル内の設定部分(青字)

分割処理の有効、無効を下記で設定します。

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

事前に計算されたCRC値の参照アドレスを下記で定義します。

```
/* The address where the 32bit reference CRC value will be stored.
   The linker must be configured to generate a CRC value and store it at this location. */
#define     DIV_AREA 1                      // 0:Not divide  1:Do divide
#if(DIV_AREA==1)
#define CRC_ADDRESS 0x000FFFC0  // Flash ROM 1MB  *The area from 0xFFFC0 to 0xFFFFF is stored Calurated CRC Value.
//#define CRC_ADDRESS 0x000BFFC0  // Flash ROM 768KB
//#define CRC_ADDRESS 0x0007FFC0  // Flash ROM 512KB
#else
#define CRC_ADDRESS 0x000FFFFC  // Flash ROM 1MB
//#define CRC_ADDRESS 0x000BFFFC  // Flash ROM 768KB
//#define CRC_ADDRESS 0x0007FFFC  // Flash ROM 512KB
#endif
```

上記設定により事前計算されたチェックサムを格納してください。

分割処理が有効(DIV_AREA=1)の場合　：　0xFFFC0～FFFFFの領域に格納してください。

分割処理が無効(DIV_AREA=0)の場合　：　0xFFFFC～FFFFFの領域に格納してください。

なお、格納方法については”2.2.1”を参照。

2.2.3 電源投入時(Power-On)

使用するすべての ROM メモリは、電源投入時にテストする必要があります。

この領域が 1 つの連続したブロックである場合、関数 CRC_Calculate を使用して、計算された CRC 値を計算して返すことができます。

使用する ROM が 1 つの連続したブロックにない場合は、次の手順を使用する必要があります。

1. CRC_Start を呼び出します。
2. CRC 計算に含めるメモリの各領域に対して CRC_AddRange を呼び出します。
3. CRC_Result を呼び出して、計算された CRC 値を取得します。

計算された CRC 値は、関数 CRC_Verify を使用して、ROM に格納されている参照 CRC 値と比較できます。

プロジェクトで使用されるすべての ROM 領域が CRC 計算に含まれるようにするのはユーザの責任です。

2.2.4 定期的(Periodic)

ROM が連続していても、CRC_AddRange メソッドを使用して ROM の定期的なテストを行うことをお勧めします。これにより、CRC 値をセクション単位で計算できるため、単一の関数呼び出しに時間がかかりすぎることはありません。電源投入テストで指定された手順に従い、各アドレス範囲が十分に小さいことを確認して、CRC_AddRange の呼び出しに時間がかかりすぎないようにします。

2.3 RAM

テストが必要な RAM の領域は、プロジェクトのメモリマップに応じて大きく変わる可能性があることを認識することが非常に重要です。

RAM をテストするときは、次の点に注意してください。

1. r_ram_diag.h を include してください。
2. r_ram_diag_config.h のディレクティブを必要に応じて変更してください（**表 1-20** を参照）
3. ECC および S キャッシュを無効にしてテストを実施してください。
4. R_RAM_Diag に必要なパラメーター（**1.3.4** を参照）を定義し、パラメータを渡し関数 R_RAM_Diag を呼び出してください。
5. 非破壊テストの場合、バッファ（RramBuffer）を割り当て、保護データが他のブロックに格納されるように設定してください。

2.3.1 電源投入時(Power-On)

電源投入時は、RAM テストを実施します。

最初に Extended March C-アルゴリズムを使用してテストを実施し、次に WALKPAT アルゴリズムを使用してテストを実施します。

破壊テストを選択することが可能です。

起動時間が非常に重要な場合は、テストする領域や使用するテストアルゴリズムを限定するなど微調整してください。

2.3.2 定期的(Periodic)

すべての定期的なテストは非破壊的でなければなりません。

定期的な RAM テストでは使用アルゴリズムを「Extended March C-」又は「WALKPAT」を選択してテストを実施します。(※サンプルプロジェクトでは、「WALKPAT」を選択)

また、テスト対象領域が広い場合、処理時間が長くなりますのででシステムに応じた RAM ブロックの分割が必要になります。

2.4　クロック

メインクロックの監視は、ClockMonitor_Init 関数の呼び出しで設定されます。次の #define によって決定されるように、外部または内部の基準クロックの使用の選択に応じて、この関数には 2 つのバージョンがあります。

```
#define CLOCK_MONITOR_USE_EXTERNAL_REFERENCE_CLOCK
```

参考例：

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

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

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

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

#endif
```

基準クロックに外部基準クロックを使用する場合、ユーザは使用する CACREF 端子を ClockMonitor_Init 関数の入力パラメータで指定することができます（上記の例では、eCLOCK_MONITOR_CACREF_A を指定）。RA MCU の各デバイスの端子と入力パラメータの関係を以下に示します。どの端子を使用するかは、ユーザがシステム構成に合わせて決定します。

表 2-1 CACREF 端子と入力パラメータ（CLOCK_MONITOR_CACREF_PIN ePin）

MCU	CACREF として設定可能な端子（ポート番号）	入力パラメータ ePin のシンボル
RA6M4	P204	eCLOCK_MONITOR_CACREF_A
	P402 (注)	eCLOCK_MONITOR_CACREF_B
	P500	eCLOCK_MONITOR_CACREF_C
	P600	eCLOCK_MONITOR_CACREF_D
	P611	eCLOCK_MONITOR_CACREF_E
	P708	eCLOCK_MONITOR_CACREF_F

注：P402 は、VBTICTLR (VBATT 入力コントロールレジスタ) の設定の影響を受けます。詳しくは、各 RA MCU のハードウェアユーザーズマニュアルの「I/O ポート」および「バッテリバックアップ機能」の章を参照ください。

ClockMonitor_Init 関数は、メインクロックが構成され、IWDT が有効になるとすぐに呼び出すことができます。IWDT を有効にする方法については、**2.5 章**を参照してください。

その後、クロック監視はハードウェア（CAC モジュール）によって実行されるため、定期的なテスト中にソフトウェアで行うべきことは特にありません。

CAC による割り込み生成を有効にするには、割り込みコントローラユニット（ICU）とネスト化ベクタ割り込みコントローラ（NVIC）の両方を構成する必要があります。

割り込みコントローラユニット（ICU）では、ICU イベントリンク設定レジスタ（IELSRn）に、CAC 周波数エラー割り込み、および CAC オーバフローに対応するイベント番号を設定します。

なお、e² studio で FSP（Flexible Software Package）を利用する場合、ICU の構成は、RA コンフィグレーションエディタの「Interrupts」タブで設定できます。

表 2-2 CAC 関連の IELSRn レジスタの設定

MCU	イベント名	IELSRn.IELS
RA6M4	CAC_FERRI	0x09E
	CAC_OVFI	0x0A0

ネスト化ベクタ割り込みコントローラ（NVIC）の設定は、clock_monitor.c ファイル内の CAC_Err_Detect_Test()関数で行っています。

ここで、NVIC_SetPriority()と NVIC_EnableIRQ()は FSP が提供する CMSIS 関数、*CAC_FREQUENCY_ERROR_IRQn* および *CAC_OVERFLOW_IRQn* は、FSP が生成した IRQ 番号です。

```
// CAC関連割込みのNVIC側設定
 /* CAC frequency error  ISR priority */
 NVIC_SetPriority(CAC_FREQUENCY_ERROR_IRQn,0);
 /* CAC frequency error  ISR enable */
 NVIC_EnableIRQ(CAC_FREQUENCY_ERROR_IRQn);
 /* CAC overflow  ISR priority */
 NVIC_SetPriority(CAC_OVERFLOW_IRQn,0);
 /* CAC overflow  ISR enable */
 NVIC_EnableIRQ(CAC_OVERFLOW_IRQn);
```

周波数エラー割り込み関連 NVIC 設定

オーバフローエラー割り込み関連 NVIC 設定

発振停止を検出すると、NMI 割り込みが発生します。本サンプルソフトでは次の例に示すように NMI 割り込みコールバック関数(NMI_Handler_callback)内で予め準備したエラー処理関数(" Clock_Stop_Detection()")を実行します。

```
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　独立ウォッチドッグタイマ（IWDT）

2.5.1　OFS0 レジスタの設定例（IWDT 関連）

独立ウォッチドッグタイマを構成するには、オプション設定メモリの OFS0 レジスタを設定する必要があります。例えば、オプション設定メモリを以下のように設定するとします。

表 2-3 OFS0 レジスタの設定例（IWDT 関連）

項目	OFS0 レジスタの設定値（例）
IWDT スタートモード（IWDTSTRT）	1： リセット後、IWDT は停止状態
IWDT タイムアウト期間選択（IWDTTOPS[1:0]）	10b：512 サイクル (0x01FF)
IWDT 専用クロック分周比（IWDTCKS[3:0]）	0010b：16 分周
IWDT ウィンドウ終了位置（IWDTRPES[1:0]）	00b：75%
IWDT ウィンドウ開始位置（IWDTRPSS[1:0]）	11b：100%
IWDT リセット割り込み要求（IWDTRSTIRQS）	0：ノンマスカブル割り込み要求、または割り込み要求を許可
IWDT 停止制御（IWDTSTPCTL）	1：スリープモード、スヌーズモード、またはソフトウェアスタンバイモードの状態にあるとき、カウント停止

e² studio で FSP (Flexible Software Package) を利用する場合、FSP の「BSP」タブのプロパティで、オプション設定メモリの設定ができます。

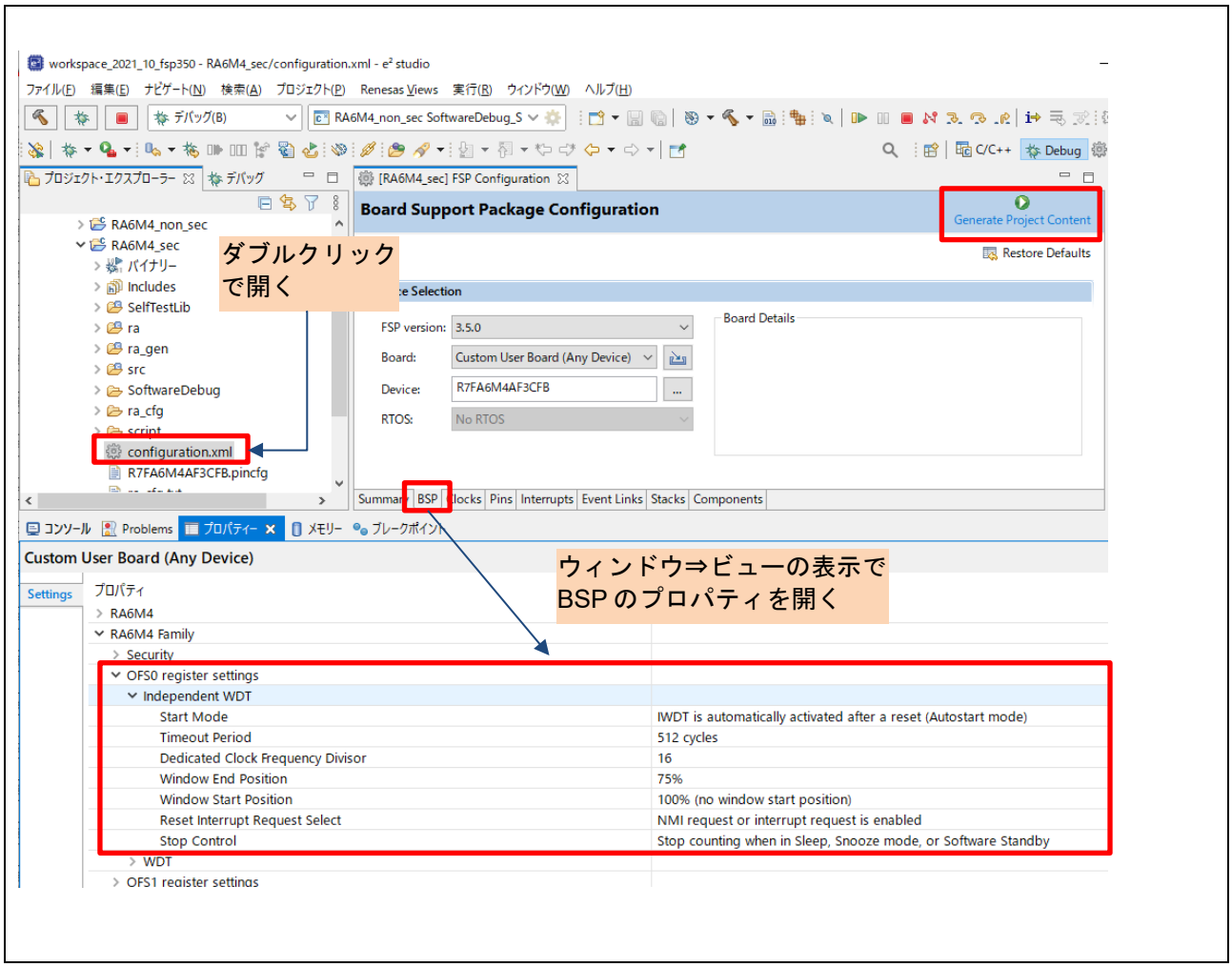

図 2-7. e² studio の FSP による OFS0 レジスタ設定例

「Generate Project Content」ボタンを押すと、プロパティでの設定内容が、下記ファイルの該当シンボルの定義に反映されます。

- 該当ファイル

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

- 該当シンボル部分（抜粋）

```
#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)
      :       :
```

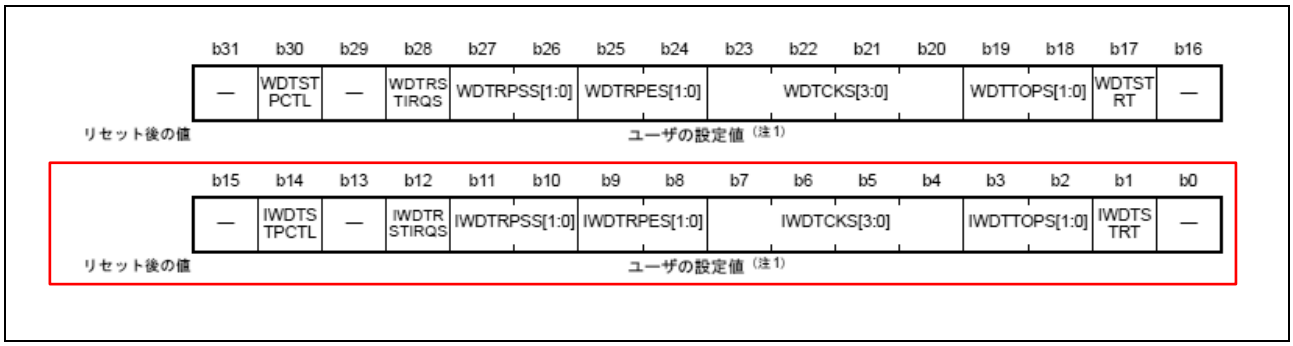

図 2-8. オプション機能選択レジスタ 0（OFS0）

IWDT の詳細につきましては、RA MCU のハードウェアユーザーズマニュアル「**25. 独立ウォッチドッグタイマ (IWDT)**」を参照ください。

独立ウォッチドッグタイマは、IWDT_Init を呼び出して、リセット後できるだけ早く初期化する必要があります。

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

この後、ウォッチドッグタイマは、ウォッチドッグタイマがタイムアウトしてリセットが実行されるのを防ぐために、定期的にリフレッシュする必要があります。ウィンドウ処理を使用する場合、リフレッシュは単に定期的であるだけでなく、指定されたウィンドウに一致するように時間を調整する必要があります。ウォッチドッグタイマの更新は以下で行います。

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

ウォッチドッグタイマがエラー検出時に NMI を生成するように設定されている場合、ユーザはその結果の割り込みを処理する必要があります。

エラー検出時にリセットを実行するようにウォッチドッグタイマが構成されている場合、リセット後に、コードは IWDT_DidReset を呼び出して IWDT がリセットを引き起こしたかどうかを確認する必要があります。

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

2.5.2　NMI 割込みコールバック関数の登録と記述例

P-ON 起動時に IWDT が正常に動作するかを API 関数：IWDT_Err_Detect_Test()内で確認します。

そのための事前準備として、ユーザは NMI 割り込みのコールバック関数(NMI_Handler_callback)内で IWDT アンダーフローによる割り込み要因だった場合に f_IWDT_ERROR_TEST を”0”に設定する処理を準備する必要があります。

ユーザーは、FSP(Flexible Software Package)が提供する BSP API 関数”R_BSP_GroupIrqWrite()”を使用してコールバックを登録することができます。
これを実施することにより、1 つ以上のグループ割り込みの通知を有効にすることができます。
NMI 割り込みが発生すると、NMI ハンドラーは割り込みの原因に対して登録されたコールバックがあるかどうかを確認し、登録されている場合は登録されたコールバック関数を呼び出します。

なお詳細は、下記の RA FSP (Flexible Software Package) のドキュメントを参照ください。

Renesas Flexible Software Package (FSP) v3.5.0 User’s Manual
の” 4.1.2 MCU Board Support Package(BSP)” – “◆ R_BSP_GroupIrqWrite()”を参照ください。

次に NMI 割り込みコールバック関数(NMI_Handler_callback)の登録及び記述例を記載します。

◎NMI 割り込みコールバック関数の登録

サンプルソフトの”RA_SelfTest.c”にあるコールバック関数登録時の記述例です。ユーザのシステムに合わせて登録を実施してください。

```
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);
}
```

◎NMI 割り込みコールバック関数(NMI_Handler_callback)の IWDT 割り込み要因発生の記述例(青字)

```
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(     IWDTSR_reg->IWDTSR_b.REFEF == 1 )
                            {
                                    Watchdog_Test_Failure();
                            }
                            else if( f_IWDT_ERROR_TEST == 0 )
                            {
                                    Watchdog_Test_Failure();
                            }
                    break;
                case BSP_GRP_IRQ_OSC_STOP_DETECT :
                    Clock_Stop_Detection();
                    break;
                     ・
                     ・
                     ・
                default:
                    break;
        }

        if( irq == BSP_GRP_IRQ_IWDT_ERROR )
        {
                f_IWDT_ERROR_TEST = 0;

                /*Clear flag*/
                IWDTSR_reg->IWDTSR_b.UNDFF = 0;

                __NOP(); __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
        }
        else
        {
                //              Error_Detected_Loop(ERROR_NMI_OTHER);
                Error_Detected_Loop(ERROR_NMI_OTHER);

                /*Should not return from an NMI*/
                while(1){;}
        }
}
```

ウェブサイトとサポート

RA MCU に関する情報や、ツール、ドキュメントのダウンロード、技術サポートなどは、下記の各ウェブサイトを通じて利用できます。

- RA 製品情報：　www.renesas.com/ra
- RA FSP (Flexible Software Package)：　www.renesas.com/FSP
- RA サポートフォーラム：　www.renesas.com/ra/forum
- Renesas サポート：　www.renesas.com/support

参考文書：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

改訂履歴

Rev.	発行日	説明	
		ページ	ポイント
1.00	2023.6.30	ー	初版

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策
   CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。
2. 電源投入時の処置
   電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. 電源オフ時における入力信号
   当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。
4. 未使用端子の処理
   未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。
5. クロックについて
   リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
6. 入力端子の印加波形
   入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。
7. リザーブアドレス（予約領域）のアクセス禁止
   リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
8. 製品間の相違について
   型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品、本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
6. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
10. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
12. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

(Rev.4.0-1　2017.11)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。