RAファミリ
RA MCUのためのIEC60730/60335セルフテスト･ライブラリ (CM4_CM23)

R01AN5540JJ0100
Rev.1.00
2020.12.18

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

洗濯機、食器洗い機、乾燥機、冷蔵庫、冷凍庫、クッカー/ストーブなどの器具は、クラス B の分類に分類される傾向があります。

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 B 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（下記の注を参照）。

これらのルーチンは IEC60730 準拠を基本として開発されましたが、ルネサス MCU のセルフテスト用の任意のシステムに実装できます。

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。エンドユーザが、これらのルーチンをシステム全体の設計に統合する方法には柔軟性がありますが、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。

エラー処理ルーチンの定義は、割り込みハンドラルーチンだけでなく、ユーザにも要求されることに注意してください。ソフトウェアルーチンでカバーされるエラーは非常に重大であり（プログラムカウンタの障害など）、ソフトウェアエラー処理だけでなく、ハードウェア安全メカニズム（たとえば、独立ウォッチドッグタイマ IWDT）の利用がなければ正しいソフトウェア機能は保証できません。

注：このドキュメントは、European Norm EN60335-1：2002 / A1：2004 Annex R に基づいており、規格 IEC 60730-1（EN60730-1：2000）がいくつかの点で使用されています。上記規格の Annex R には、定義、情報、および該当する段落について IEC 60730-1 にジャンプする単一のシートが含まれています。

ターゲット

- **デバイス**：
  - ルネサス RA ファミリ MCU（Arm® Cortex®-M4、Arm® Cortex®-M23）※シリーズとグループは、次ページ参照
- **開発環境**（下記のいずれか）：
  - GNU-GCC ARM Embedded 9.2.1.20191025 / e² studio 7.8.0
  - IAR / EWARM Version 8.50.1

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 1. RA ファミリ MCU セルフテスト機能リスト

		CPU コア	Arm® Cortex®-M4		Arm® Cortex®-M23	
		シリーズ	RA6	RA4	RA2	
		グループ	RA6M1	RA4M1	RA2A1	RA2L1
テスト機能	CPU		○	○	○(注)	○(注)
	ROM		○	○	○	○
	RAM		○	○	○	○
	クロック		○	○	○	○
	独立ウォッチドッグタイマ (IWDT)		○	○	○	○
	電圧		○	○	○	○
	ADC	ADC12	○	―	―	○
		ADC14	―	○	―	―
		ADC16	―	―	○	―
	温度		○	○	○	○
	Port Output Enable (POE)		○	○	○	○
	GPIO		○	○	○	○

注：FPU テストを除く

目次

1. テスト ... 5
1.1 CPU ... 5
1.1.1 CPU ソフトウェア API ... 7
1.2 ROM ... 16
1.2.1 CRC32 アルゴリズム ... 16
1.2.2 CRC ソフトウェア API ... 16
1.3 RAM ... 20
1.3.1 RAM テストアルゴリズム ... 20
1.3.2 RAM ソフトウェア API ... 22
1.4 クロック ... 32
1.4.1 CAC によるメインクロック周波数の監視 ... 32
1.4.2 メインクロックの発振停止検出 ... 32
1.5 独立ウォッチドッグタイマ (IWDT) ... 36
1.6 電圧 ... 38
1.7 ADC ... 39
1.8 温度 ... 44
1.9 Port Output Enable（POE） ... 47
1.10 GPIO ... 53

2. 使用例 ... 54
2.1 CPU ... 54
2.1.1 電源投入時 ... 54
2.1.2 定期的 ... 54
2.2 ROM ... 54
2.2.1 事前の参照用 CRC 計算 ... 55
2.2.2 電源投入時 ... 60
2.2.3 定期的 ... 60
2.3 RAM ... 60
2.3.1 電源投入時 ... 60
2.3.2 定期的 ... 60
2.4 クロック ... 61
2.5 独立ウォッチドッグタイマ（IWDT） ... 63
2.6 電圧 ... 66
2.7 ADC ... 66
2.7.1 電源投入時 ... 66
2.7.2 定期的 ... 66
2.8 温度 ... 67
2.8.1 電源投入時 ... 67
2.8.2 定期的 ... 67

2.9　Port Output Enable（POE）..........67
2.9.1　I/O ポートの設定..........68
2.9.2　割り込みの設定..........68
2.10　GPIO..........70

3.　ベンチマーク..........71
3.1　RA4M1 測定結果..........71
3.1.1　CPU..........72
3.1.2　ROM..........72
3.1.3　RAM..........72

ウェブサイトとサポート..........75

改訂履歴..........76

1. テスト

1.1 CPU

この章では、CPU テストルーチンについて説明します。(参照：*IEC 60730: 1999 + A1 ：2003 Annex H-Table H.11.12.1 CPU*)

このソフトウェアは、次の CPU レジスタをテストします。

表 1.1 テストされるレジスタ一覧表

<table>
<tr><th colspan="2">CPU コア</th><th colspan="2">Arm® Cortex®-M4</th><th colspan="2">Arm® Cortex®-M23</th></tr>
<tr><th colspan="2">グループ</th><th>RA6M1</th><th>RA4M1</th><th>RA2A1</th><th>RA2L1</th></tr>
<tr><td rowspan="5">テスト対象レジスタ</td><td>汎用レジスタ</td><td colspan="2">R0-R12</td><td colspan="2">R0-R12</td></tr>
<tr><td>制御レジスタ</td><td colspan="2">MSP,
PSP,
LR,
PSR(ASPR, IPSR,
EPSR), PRIMASK,
CONTROL,
FAULTMASK,
BASEPRI</td><td colspan="2">MSP,
PSP,
LR,
PSR(APSR, IPSR,
EPSR), PRIMASK,
CONTROL</td></tr>
<tr><td>プログラムカウンタ</td><td colspan="2">PC</td><td colspan="2">PC</td></tr>
<tr><td>FPU 拡張レジスタ
(S0～S31)</td><td colspan="2">S0-S31</td><td colspan="2">該当なし</td></tr>
<tr><td>FPU 制御レジスタ
(注)</td><td colspan="2">CPACR, FPCCR,
FPCAR,
FPSCR, FPDSCR</td><td colspan="2">該当なし</td></tr>
</table>

注：レジスタ名が同じでも、デバイスによってビットフィールドの構成が異なるものがあります。

ソースファイル cpu_test.c は、C 言語を使用した CPU テストの実装を提供し、アセンブリ言語関数（CPU_Test_Control）に依存してレジスタにアクセスします。汎用レジスタのカップリングテストバージョンを使用するには、ファイル cpu_test_coupling.c も必要です。結合テストはアセンブリ言語関数に依存します。

- TestGPRsCouplingStart_A
- TestGPRsCouplingR0_A
- TestGPRsCouplingR1_R3_A
- TestGPRsCouplingR4_R6_A
- TestGPRsCouplingR7_R9_A
- TestGPRsCouplingR10_R12_A

- TestGPRsCouplingStart_B
- TestGPRsCouplingR0_B
- TestGPRsCouplingR1_R3_B
- TestGPRsCouplingR4_R6_B
- TestGPRsCouplingR7_R9_B
- TestGPRsCouplingR10_R12_B
- TestGPRsCouplingEnd

あるいは、CPU_Test_General_Low、CPU_Test_General_High アセンブリ言語関数を使用して、汎用レジスタをテストします。

ソースファイル cpu_test.c は、FPU_Control アセンブリ言語関数にも依存して FPU 制御レジスタをアクセスします。FPU 拡張レジスタの結合テストバージョンを使用する場合は fpu_test_coupling.c ファイルも必要です。

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

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

あるいは、FPU_Exten アセンブリ言語関数を使用して FPU 拡張レジスタをテストします。

cpu_test.h ヘッダファイルは、CPU テストへのインタフェースを提供します。

本テストは各レジスタの基本的な読み書きをテストしています。API 関数には、テストの結果を示す戻り値はありません。代わりにこれらのテストのユーザは、次の宣言でエラー処理関数を作成する必要があります。

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

エラーが検出された場合、CPU テストはこの関数にジャンプします。この関数は 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。

1.1.1 CPU ソフトウェア API

表 1.2 CPU ソフトウェア API ソースファイル

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

RENESAS

■ cpu_test.c ファイル

Syntax	
void CPU_Test_All(void)	
Description	
以下に詳述するすべてのテストを次の順序で実行します。 1. カップリング汎用レジスタテストを使用する場合（下記、注 1 を参照）： 　　CPU_Test_GPRsCouplingPartA 　　CPU_Test_GPRsCouplingPartB 2. カップリング汎用レジスタテストを使用しない場合: 　　CPU_Test_General_Low 　　CPU_Test_General_High 3. CPU_Test_Control 4. CPU_Test_PC 5. カップリング FPU 拡張レジスタテストを使用する場合（下記、注 2 を参照）： 　　FPU_Test_FPUCouplingPartA 　　FPU_Test_FPUCouplingPartB 6. カップリング FPU 拡張レジスタテストを使用しない場合 　　FPU_Exten 7. FPU_Control プロセッサが特権モードであることを確認するのは、呼び出し元の関数の責任です。この関数が非特権モードで呼び出された場合、一部のレジスタビットは非特権モードでアクセスできないため、テストは失敗します。さらに CPU_Test_Control 関数はスタックポインタレジスタ（MSP と PSP）をテストするため、CPU_Test_All 関数を実行して復元する前に、スタックポインタの監視を無効（MSPMPUCTL. ENABLE =0、PSPMPUCTL.ENABLE= 0）にし、関数リターン後に再び戻す必要があります。 また、このテストの間、割り込みが発生しないようにするのは呼び出し元関数の責任です。 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。 詳細については、個々のテストを参照してください。 注 1：コード内の #define USE_TEST_GPRS_COUPLING は、汎用レジスタのテストに使用される関数を選択するために使用されます。 注 2：コード内の #define USE_TEST_FPU_COUPLING は、FPU 拡張レジスタのテストに使用される関数を選択するために使用されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
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。 独自のセクションに配置された関数を呼び出すことで PC が動作していることをテストします。実行のために関数が呼び出される際、PC レジスタビットが必要になるように、離れた場所に関数を配置することができます。 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

■ CPU_Test_General_Low.asm ファイル

Syntax	
void CPU_Test_General_Low(void)	
Description	
汎用レジスタ R0、R1、R2、R3、R4、R5、R6 および R7 をテストします。レジスタはペアでテストされます。 レジスタの各ペアについて、 1. 両方のレジスタに h'55555555 を書き込みます。 2. 両方のレジスタを読み、一致していることを確認します。 3. 両方のレジスタに h'AAAAAAAA を書き込みます。 4. 両方のレジスタを読み、一致していることを確認します。 このテストの間、例外を無効にするのは呼び出し元関数の責任です。 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

■ CPU_Test_General_High.asm ファイル

Syntax	
void CPU_Test_General_High(void)	
Description	
汎用レジスタ R8、R9、R10、R11 および R12 をテストします。レジスタはペアでテストされます。 レジスタの各ペアについて、 1. 両方のレジスタに h'55555555 を書き込みます。 2. 両方のレジスタを読み、一致していることを確認します。 3. 両方のレジスタに h'AAAAAAAA を書き込みます。 4. 両方のレジスタを読み、一致していることを確認します。 このテストの間、例外を無効にするのは呼び出し元関数の責任です。 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

■ CPU_Test_Control.asm ファイル

Syntax	
`void CPU_Test_Control(void)`	
Description	
この関数は、制御レジスタ（デバイスにより異なるため「表 1.1 テストされるレジスタ」を参照）をテストします。 このテストでは、レジスタ R1～R4 が機能していることを前提としています。 通常、各レジスタのテスト手順は次のとおりです。 各レジスタについて、 1. `h'55555555` を書き込みます。 2. 読み返して、値が `h'55555555` であることを確認します。 3. `h'AAAAAAAA` を書き込みます。 4. 読み返して、値が `h'AAAAAAAA` であることを確認します。 ただし、レジスタ内の特定のビットの制限により、この手順が許可されない場合があることに注意してください。これらのケースでは他のテスト値が選択されています。 プロセッサが特権モードであることを確認するのは、呼び出し元の関数の責任です。非特権モードでこの関数が呼び出されると、非特権モードでは一部のレジスタビットにアクセスできないため、テストは失敗します。 このテストの間、例外を無効にするのは呼び出し元関数の責任です。 注：このテストでは例外を無効にする必要があるため、FAULTMASK および PRIMASK はテストされません。したがって、これらは FAULTMASK および PRIMASK を変更するテスト中にアクティブ化されません。 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

■ cpu_test_coupling.c ファイル

Syntax	
void CPU_Test_GPRsCouplingPartA(void)	
Description	
汎用レジスタ R0 から R12 をテストします。レジスタ間の結合障害が検出されます。 汎用レジスタテストはパート A とパート B で構成され、この関数はパート A です。 このテストの間、割り込みが発生しないようにするのは呼び出し元関数の責任です。 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

Syntax	
void CPU_Test_GPRsCouplingPartB(void)	
Description	
汎用レジスタ R0 から R12 をテストします。レジスタ間の結合障害が検出されます。 汎用レジスタテストはパート A とパート B で構成され、この関数はパート B です。 このテストの間、割り込みが発生しないようにするのは呼び出し元関数の責任です。 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

■ fpu_test_coupling.c ファイル

Syntax	
`void FPU_Test_FPUCouplingPartA(void)`	
Description	
FPU 拡張レジスタ S0～S31 をテストします。レジスタ間の結合障害が検出されます。 FPU 拡張レジスタテストはパート A とパート B で構成され、この関数はパート A です。 このテストの間、割り込みが発生しないようにするのは呼び出し元関数の責任です。 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

Syntax	
`void FPU_Test_FPUCouplingPartB(void)`	
Description	
FPU 拡張レジスタ S0～S31 をテストします。レジスタ間の結合障害が検出されます。 FPU 拡張レジスタテストはパート A とパート B で構成され、この関数はパート B です。 このテストの間、割り込みが発生しないようにするのは呼び出し元関数の責任です。 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

■ fpu_exten.asm ファイル

Syntax	
`void FPU_Exten(void)`	
Description	
FPU 拡張レジスタ S0～S31 をテストします。レジスタはペアでテストされます。 レジスタの各ペアについて、 1. 両方のレジスタに `h'55555555` を書き込みます。 2. 両方のレジスタを読み、それらが一致することを確認します。 3. 両方のレジスタに `h'AAAAAAAA` を書き込みます。 4. 両方のレジスタを読み、それらが一致することを確認します。 このテストの間、例外を無効にするのは呼び出し元関数の責任です。 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

■ fpu_control.asm ファイル

Syntax	
void FPU_Control(void)	
Description	
FPU 制御レジスタ（デバイスにより異なるため「表 1.1 テストされるレジスタ」を参照）をテストします。このテストは、レジスタ R1～R10 が機能していることを前提としています。 通常、各レジスタのテスト手順は次のとおりです。 各レジスタについて、 1. h'55555555 を書き込みます。 2. 読み取り、値が h'55555555 であることを確認します。 3. h'AAAAAAAA を書き込みます。 4. 読み返して、値が h'AAAAAAAA であることを確認します。 ただし、レジスタ内の特定のビットの制限により、この手順が許可されない場合があることに注意してください。これらのケースでは、他のテスト値が選択されています。 プロセッサが特権モードであることを確認するのは、呼び出し元の関数の責任です。非特権モードでこの関数が呼び出されると、非特権モードでは一部のレジスタビットにアクセスできないため、テストは失敗します。 このテストの間、例外を無効にするのは呼び出し元関数の責任です。 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

1.2 ROM

この章では、CRC ルーチンを使用した ROM /フラッシュメモリテストについて説明します。(参照：*IEC 60730：1999 + A1：2003 Annex H – H2.19.4.1 CRC– Single 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 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 1.3 CRC ソフトウェア API ソースファイル

ファイル名
crc.h crc_verify.h
crc.c CRC_Verify.c

■ 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`	True＝テストパス、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

March テストは、RAM をテストする効果的な方法としてよく知られている一連のテストです。

March テストは、March エレメントの有限シーケンスで構成されます。March エレメントは、次のセルに進む前に、メモリアレイ内のすべてのセルに適用される有限の操作シーケンスです。

一般に、アルゴリズムを構成する March エレメントが多いほど、その障害カバー率は向上しますが、実行時間が長くなります。

アルゴリズム自体は破壊的ですが (現在の RAM 値は保持されない)、提供されているテスト関数は非破壊的なオプションを提供するため、メモリの内容を保持できます。これは、実際のアルゴリズムを実行する前にメモリを提供されたバッファにコピーし、テストの最後にバッファからメモリを復元することによって実現されます。API には RAM テスト領域だけでなく、バッファも自動的にテストするオプションが含まれています。

テスト中の RAM 領域は、テスト中は他の用途に使用できません。そのためスタックに使用される RAM のテストが特に困難になります。この問題を解決するために、API にはスタックのテストに使用できる関数が含まれています。

次のセクションでは、特定の March テストについて説明します。続いて、ソフトウェア API の仕様を示します。

1.3.1 RAM テストアルゴリズム

(1) March C

March C アルゴリズム（van de Goor 1991）は、合計で 10 の演算がある 6 つのマーチエレメントで構成され、次の故障を検出します:

1. 縮退故障（SAF：Stuck-At Faults）
    - セルまたはラインの論理値は常に 0 または 1 です
2. 遷移故障（TF：Transition Faults）
    - 0→1 または 1→0 の遷移ができないセルまたはライン
3. カップリング故障（CF：Coupling Faults）
    - 1 つのセルへの書き込み操作により、2 番目のセルの内容が変更される
4. デコーダ故障（AF：Address decoder Faults）
    - アドレスデコーダに影響を与える障害
    - 特定のアドレスで、セルにアクセスできない
    - 特定のセルにアクセスできない
    - 特定のアドレスで、複数のセルに同時にアクセスされる
    - 特定のセルに、複数のアドレスからアクセスできる

6 つの March エレメントがあります。

1. アレイにすべて 0 を書き込む
2. 最下位アドレスから開始して、0 をリード、1 をライト、アレイをビットごとにインクリメント
3. 最下位アドレスから開始して、1 をリード、0 をライト、アレイをビットごとにインクリメント
4. 最上位アドレスから始めて、0 をリード、1 をライト、アレイをビットごとにデクリメント
5. 最上位アドレスから始めて、1 をリード、0 をライト、アレイをビットごとにデクリメント
6. アレイからすべての 0 をリード

(2) March X

注：このアルゴリズムは RA MCU 用には実装されていないため、下記の March X WOM バージョンの参考情報として提示します。

March X アルゴリズムは、合計で 6 つの演算がある 4 つの March エレメントで構成されます。次の故障を検出します。

1. 縮退故障（SAF）
2. 遷移故障（TF）
3. 反転カップリング故障（CFin：inversion Coupling Faults）
4. アドレスデコーダ故障（AF）

4 つの March エレメントがあります。

1. アレイにすべて 0 をライト
2. 最下位アドレスから開始して、0 をリード、1 をライト、アレイをビットごとにインクリメント
3. 最上位アドレスから開始して、1 をリード、0 をライト、アレイをビットごとにデクリメント
4. アレイからすべて 0 をリード

(3) March X WOM (Word-Oriented Memory version)

March X Word-Oriented Memory（WOM）アルゴリズムは、2 段階で標準 March X アルゴリズムから作成されました。まず、標準の March X は、シングルビットデータパターンの使用から、メモリアクセス幅に等しいデータパターンの使用に変換されます。この段階でのテストは、主にアドレスデコーダ障害を含むワード間障害を検出しています。2 番目の段階は、2 つの March エレメントを追加することです。1 つ目は H/L ビットが交互になるデータパターンを使用し、2 つ目はその逆を使用します。これらのエレメントの追加は、ワード内カップリングフォールトを検出することです。

6 つの March エレメントがあります。

1. アレイにすべて 0 をライト
2. 最下位アドレスから開始して、0 をリード、1 をライト、アレイをワード単位でインクリメント
3. 最上位アドレスから開始して、1 をリード、0 をライト、ワードごとにデクリメント
4. 最下位アドレスから開始して、0 をリード、h'AA をライト、アレイをワード単位でインクリメント
5. 最上位のアドレスから始めて、h'AA をリード、h'55 をライト、ワードごとにデクリメント
6. アレイからすべての h'55 をリード

1.3.2 RAM ソフトウェア API

RAM テストの 2 つの実装が利用可能です。

(a) 標準実装
(b) HW 実装：
　このバージョンでは、データ演算回路（DOC：Data Operation Circuit）と、オプションでデータトランスファコントローラ（DTC：Data Transfer Controller）を使用してテストの実行を支援します。

両方の実装は同じコア API を共有しますが、HW 実装にはいくつかの追加機能があります。詳細は、本章の (3) 節をご覧ください。

(1) March C API

このテストは、8、16、または 32 ビットの RAM アクセスを使用するように構成できます。

これは、ヘッダファイルの `#defining RAMTEST_MARCH_C_ACCESS_SIZE` を次のいずれかにすることで実現されます。

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

単一の関数呼び出しでテストできる RAM の最大サイズを制限するとテストが高速化し、スタックとコードのサイズが小さくなる場合があります。これは、テスト領域に含まれる「word」の数を保持するために使用される変数のサイズを制限することによって行われます。「word」サイズは選択したアクセス幅です。

これは、ヘッダファイルの `#defining RAMTEST_MARCH_C_MAX_WORDS` を次のいずれかにすることで実現されます:

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

表 1.4 March C API ソースファイル

実装	ヘッダファイル	ソースファイル
標準実装	ramtest_march_c.h	ramtest_march_c.c
HW 実装	ramtest_march_c.h	ramtest_march_c_HW.c
	ramtest_march_HW.h	ramtest_march_HW.c

標準実装と HW 実装のどちらか一方のファイルセットを、コンパイラのビルド対象で選択してください。

ソースは ANSI C で記述され、`renesas.h` 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。

注：API は関数呼び出しで 1 つのワードだけをテストします。しかし、ワード間でテストする結合故障は、関数を使用して 1 ワードより大きいデータ範囲をテストすることが重要です。

■ ramtest_march_c.c ファイル

Syntax	

`bool_t RamTest_March_C( const uint32_t ui32_StartAddr,`
`const uint32_t ui32_EndAddr,`
`void* const p_RAMSafe)`	

Description	
March C (Goor 1991) アルゴリズムを用いた RAM メモリテスト	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word アドレス。これは、選択したメモリアクセス幅に合わせる必要があります。
const uint32_t ui32_EndAddr	テストする RAM の最後の word アドレス。これは、選択したメモリアクセス幅に合わせて、ui32_StartAddr 以上の値にする必要があります。
void* const p_RAMSafe	破壊的なメモリテストの場合は、NULL に設定します。 非破壊メモリテストの場合は、テスト領域の内容をコピーするのに十分な大きさで、選択したメモリアクセス幅に合わせたバッファの先頭を設定します。
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

Syntax	

`bool_t RamTest_March_C_Extra( const uint32_t ui32_StartAddr,`
`const uint32_t ui32_EndAddr,`
`void* const p_RAMSafe)`	

Description	
March C（Goor 1991）アルゴリズムを用いた非破壊 RAM メモリテスト。 この関数は、RamTest_March_C 関数とは異なり、使用前に「RAMSafe」バッファをテストします。 「RAMSafe」バッファのテストが失敗すると、テストは中止され、関数は False を返します。	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word アドレス。これは、選択したメモリアクセス幅に合わせる必要があります。
const uint32_t ui32_EndAddr	テストする RAM の最後の word アドレス。これは、選択したメモリアクセス幅に合わせて、ui32_StartAddr 以上の値にする必要があります。
void* const p_RAMSafe	テスト領域の内容をコピーするのに十分な大きさで、選択したメモリアクセス幅に揃えられたバッファの先頭に設定します。
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

(2) March X WOM API

このテストは、8、16、または 32 ビットの RAM アクセス用に構成できます。

これは、ヘッダファイル内の #defining RAMTEST_MARCH_X_WOM_ACCESS_SIZE に次のいずれかを選ぶことで実現されます:

- RAMTEST_MARCH_X_WOM_ACCESS_SIZE_8BIT
- RAMTEST_MARCH_X_WOM_ACCESS_SIZE_16BIT
- RAMTEST_MARCH_X_WOM_ACCESS_SIZE_32BIT

テストの実行時間を短縮するために、1 回の関数呼び出しでテストできる RAM の最大サイズを制限することを選択できます。これは、テスト領域に含まれる「word」の数を保持するために使用される変数のサイズを制限することによって行われます。「word」サイズは、選択したアクセス幅と同じです。

これは、ヘッダファイルの #defining RAMTEST_MARCH_X_WOM_MAX_WORDS を次のいずれかにすることによって実現されます:

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

表 1.5 March X WOM API ソースファイル

実装	ヘッダファイル	ソースファイル
標準実装	ramtest_march_x_wom.h	ramtest_march_x_wom.c
HW 実装	ramtest_march_x_wom.h	ramtest_march_x_wom_HW.c
	ramtest_march_HW.h	ramtest_march_HW.c

標準実装と HW 実装のどちらか一方のファイルセットを、コンパイラのビルド対象で選択してください。

ソースは ANSI C で記述され、renesas.h 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。

注：API は関数呼び出しで 1 つのワードだけをテストします。しかし、ワード間でテストする結合故障は、関数を使用して 1 ワードより大きいデータ範囲をテストすることが重要です。

■ ramtest_march_x_wom.c ファイル

Syntax	
`bool_t RamTest_March_X_WOM( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
WOM 用に変換された March X アルゴリズムに基づく RAM メモリテスト	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word アドレス。これは、選択したメモリアクセス幅に合わせる必要があります。
const uint32_t ui32_EndAddr	テストする RAM の最後の word アドレス。これは、選択したメモリアクセス幅に合わせ、ui32_StartAddr 以上の値にする必要があります。
void* const p_RAMSafe	破壊的なメモリテストの場合は、NULL に設定します。
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

Syntax	
`bool_t RamTest_March_X_WOM_Extra( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe)`	
Description	
WOM 用に変換された March X アルゴリズムに基づく非破壊 RAM メモリテスト。 この関数は、RamTest_March_X_WOM 関数とは異なり、使用前に「RAMSafe」バッファをテストします。 「RAMSafe」バッファのテストが失敗すると、テストは中止され、関数は False を返します。	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word アドレス。これは、選択したメモリアクセス幅に合わせる必要があります。
const uint32_t ui32_EndAddr	テストする RAM の最後の word アドレス。これは、選択したメモリアクセス幅に合わせ、ui32_StartAddr 以上の値にする必要があります。
void* const p_RAMSafe	テスト領域の内容をコピーするのに十分な大きさで、選択したメモリアクセス幅に揃えられたバッファの先頭に設定します。
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

(3) March C と March X WOM ー HW 実装の固有 API

March C および March X WOM テストの HW 実装は、DOC（データ演算回路）とオプションで DTC（データトランスファコントローラ）を使用して、テストの実行を支援します。DTC はテスト中の RAM を初期化するために使用され、DOC は RAM から読み戻された値を期待値と比較するために使用されます。

RAM テストの間、他の何も DOC または DTC にアクセスしないことを確認するのはユーザの責任です。

表 1.6 HW 実装ファイル

テスト	ヘッダファイル	ソースファイル
(1) March C	ramtest_march_c.h	ramtest_march_c_HW.c
(2) March X WOM	ramtest_march_x_wom.h	ramtest_march_x_wom_HW.c
HW 共通(初期化)	ramtest_march_HW.h	ramtest_march_HW.c

DTC オプションの使用に関しては、次の #define により制御されます。

表 1.7 DTC オプションの定義

定義するファイル	#define	定義時の意味
ramtest_march_HW.h	RAMTEST_USE_DTC	DTC を初期化し、使用します

■ ramtest_march_HW.c（初期化）

Syntax	
`void RamTest_March_HW_Init(void)`	
Description	
RAM テストの「HW」実装で使用されるハードウェア（DOC およびオプションで DTC）を初期化します。DTC は、RAMTEST_USE_DTC が定義されている場合にのみ使用されます。 HW 実装を使用する他の RAM テスト関数を使用する前に、この関数を呼び出します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
void	N/A

Syntax	
`bool_t RamTest_March_HW_PreTest(void)`	
Description	
これは、ハードウェア（DOC および DTC）が使用前に正しく機能しているかどうかを確認するために使用できます。DOC および（`RAMTEST_USE_DTC` が定義されている場合）DTC のクイック機能テストが実行されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストフェイル

Syntax	
`bool_t RamTest_March_HW_Is_Init(void)`	
Description	
`RamTest_March_HW_Init` が呼び出されたかどうかを確認します。 これは特定の RAM テストで使用され、HW を使用する前に HW が初期化されていることを確認します。 通常、ユーザはこの関数を使用する必要はありません。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

■ ramtest_march_c_HW.c

Syntax	

`bool_t RamTest_March_C_DTC( const uint32_t ui32_StartAddr,`
`                            const uint32_t ui32_EndAddr,`
`                            void* const p_RAMSafe)`	

Description	
［**HW 実装版**］March C (Goor 1991) アルゴリズムを用いた RAM メモリテスト	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word アドレス。これは、選択したメモリアクセス幅に合わせる必要があります。
const uint32_t ui32_EndAddr	テストする RAM の最後の word アドレス。これは、選択したメモリアクセス幅に合わせて、ui32_StartAddr 以上の値にする必要があります。
void* const p_RAMSafe	破壊的なメモリテストの場合は、NULL に設定します。 非破壊メモリテストの場合は、テスト領域の内容をコピーするのに十分な大きさで、選択したメモリアクセス幅に合わせたバッファの先頭を設定します。
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

■ ramtest_march_x_won_HW.c

Syntax	

`bool_t RamTest_March_X_WOM_DTC( const uint32_t ui32_StartAddr,`
`                                const uint32_t ui32_EndAddr,`
`                                void* const p_RAMSafe)`	

Description	
［**HW 実装版**］WOM 用に変換された March X アルゴリズムに基づく RAM メモリテスト	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word アドレス。これは、選択したメモリアクセス幅に合わせる必要があります。
const uint32_t ui32_EndAddr	テストする RAM の最後の word アドレス。これは、選択したメモリアクセス幅に合わせ、ui32_StartAddr 以上の値にする必要があります。
void* const p_RAMSafe	破壊的なメモリテストの場合は、NULL に設定します。
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストまたはパラメータチェックのフェイル

(4) RAM テストスタック API

この API を使用すると、スタックを含む RAM 領域で RAM テストを実行できます。RAM テストを実行する関数にはスタックが必要なので、これらの関数はスタックを提供された新しい RAM 領域に再配置して、元のスタック領域をテストできるようにします。テスト領域にあるスタック（メインまたはプロセス）に応じて、または両方のスタックに応じて、呼び出すことができる 3 つの関数が提供されます。

プロセッサが特権モードであることを確認するのは、呼び出し元関数の責任です。この関数が非特権モードで呼び出されると、一部のレジスタビットが非特権モードでアクセスできないため、テストは失敗します。

注：スタックテスト関数は、関数ポインタ渡しで前述の March RAM テストの 1 つを利用します。使用前に初期化を必要とするテストの場合、これらの関数のいずれかを呼び出してテストする前に、初期化が完了していることを確認するのは、ユーザの責任です。

表 1.8 RAM テストスタック API ソースファイル

ファイル名
ramtest_stack.h
ramtest_stack.c
StartBothTestAssembly.asm StartMainTestAssembly.asm StartProcTestAssembly.asm

■ ramtest_stack.c ファイル

Syntax	
`bool_t RamTest_Stack_Main( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe, const uint32_t ui32_NewMSP, const TEST_FUNC fpTest_Func)`	
Description	
メインスタックを含む（プロセススタックは含まない）領域の RAM テスト	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word のアドレス。これは fpTest_Func の要件に適応しなければなりません。
const uint32_t ui32_EndAddr	テストする RAM の最後の word のアドレス。これは fpTest_Func の要件に適応しなければなりません。
void* const p_RAMSafe	テスト RAM 領域と同じサイズのバッファの先頭に設定します。これは、fpTest_Func の要件に適応しなければなりません。
const uint32_t ui32_NewUSP	再配置先のメインスタックの新しいスタックポインタ値
const TEST_FUNC fpTest_Func	使用する実際のメモリテストへの TEST_FUNC タイプの関数ポインタ。 Typedef bool_t(*TEST_FUNC)(uint32_t, uint32_t, void*); 例：RamTest_March_X_WOM
Output Parameters	
NONE	N/A
Return Values	
bool_t	True＝テストパス、False＝テストまたはパラメータチェックのフェイル

Syntax	
`bool_t RamTest_Stack_Proc( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe, const uint32_t ui32_NewPSP, const TEST_FUNC fpTest_Func)`	
Description	
プロセススタックを含む（メインスタックは含まない）領域の RAM テスト	
Input Parameters	
`const uint32_t ui32_StartAddr`	テストする RAM の最初の word のアドレス。これは `fpTest_Func` の要件に適応しなければなりません。
`const uint32_t ui32_EndAddr`	テストする RAM の最後の word のアドレス。これは `fpTest_Func` の要件に適応しなければなりません。
`void* const p_RAMSafe`	テスト RAM 領域と同じサイズのバッファの先頭に設定します。これは、`fpTest_Func` の要件に適応しなければなりません。
`const uint32_t ui32_NewPSP`	再配置先のプロセススタックの新しいスタックポインタ値
`const fpTest_Func`	使用する実際のメモリテストへの TEST_FUNC タイプの関数ポインタ。 `Typedef bool_t(*TEST_FUNC)( uint32_t, uint32_t, void*);` 例：`RamTest_March_X_WOM`
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True＝テストパス、False＝テストまたはパラメータチェックのフェイル

Syntax	
`bool_t RamTest_Stacks( const uint32_t ui32_StartAddr, const uint32_t ui32_EndAddr, void* const p_RAMSafe, const uint32_t ui32_NewPSP, const uint32_t ui32_NewMSP, const TEST_FUNC fpTest_Func)`	
Description	
メインスタックとプロセススタックの両方のスタックを含む領域の RAM テスト	
Input Parameters	
const uint32_t ui32_StartAddr	テストする RAM の最初の word のアドレス。これは fpTest_Func の要件に適応しなければなりません。
const uint32_t ui32_EndAddr	テストする RAM の最後の word のアドレス。これは fpTest_Func の要件に適応しなければなりません。
void* const p_RAMSafe	テスト RAM 領域と同じサイズのバッファの先頭に設定します。これは、fpTest_Func の要件に適応しなければなりません。
const uint32_t ui32_NewPSP	再配置先のプロセススタックの新しいスタックポインタ値
const uint32_t ui32_NewMSP	再配置先のメインスタックの新しいスタックポインタ値
const TEST_FUNC fpTest_Func	使用する実際のメモリテストへの TEST_FUNC タイプの関数ポインタ。 Typedef bool_t(*TEST_FUNC)(const uint32_t, const uint32_t, void* const); 例：RamTest_March_X_WOM
Output Parameters	
NONE	N/A
Return Values	
bool_t	True＝テストパス、False＝テストまたはパラメータチェックのフェイル

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ーフロー割り込みの 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。

表 1.9 Clock ソースファイル

ファイル名
clock_monitor.h
clock_monitor.c

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. 使用例の 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、または 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 入力に使用する端子
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、または 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 オーバー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

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。

表 1.10 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.c

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

1.6 電圧

RA MCU には低電圧検出 (LVD) モジュールがあります。これは、指定された電圧を下回る電源電圧 (VCC) を検出するために使用できます。提供されているサンプルコードは、電圧検出回路 1 を使用して、VCC が指定されたレベルを下回ったときにノンマスカブル割り込み (NMI) を生成する方法を示しています。ハードウェアはリセットを生成することもできますが、この動作はサンプルコードではサポートされていません。

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。

表 1.11 Voltage ソースファイル

ファイル名
voltage.h
voltage.c

Syntax	
void VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL eVoltage)	
Description	
電圧監視を初期化して開始します。VCC が指定された電圧を下回ると、NMI が生成されます。 注：ノンマスカブル割り込み (NMI) は、NMISR.LVDST 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。 注：電圧しきい値 eVoltage は、公称 Vcc 電圧よりも低い値に設定する必要があります。	
Input Parameters	
VOLTAGE_MONITOR_LEVEL eVoltage	指定された低電圧レベル。詳細については、voltage.h 内の列挙型 VOLTAGE_MONITOR_LEVEL を参照
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

1.7 ADC

ADC にはテストに使用できる診断モードがあります。診断モードは、ADC が変換に通常使用されるたびにテストが実行されるように構成できます。診断基準電圧（期待される結果）は、ゼロ、ハーフスケール、フルスケール（RA2A1 は、ゼロ、＋フルスケール、－フルスケール）の間で自動的に回ります。

各 RA MCU に内蔵された ADC (A/D コンバータ) と診断基準電圧を以下に示します。

表 1.12 各 RA MCU 内蔵の ADC

グループ	RA6M1	RA4M1	RA2L1	RA2A1
内蔵 ADC	ADC12	ADC14	ADC12	ADC16
ユニット数	2	1	1	1
診断基準電圧	ゼロ／ハーフスケール／フルスケール			ゼロ／ ±フルスケール

診断 SW は、診断基準電圧に対する AD 変換を提供します。

なお、RA6M1 の ADC12 は 2 ユニットあります（ユニット 0 とユニット 1）。ユニット 1 のテストを行う場合は「_u1」の付いた関数を使用してください。

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。

表 1.13 ADC ソースファイル

ファイル名
test_adc.h
test_adc.c

Syntax	
`void Test_ADC_Init(void)`	
Description	
ADC モジュール（ユニット 0）を初期化します。これは他の ADC 関数を使用する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	
`void Test_ADC_Init_u1(void)`	
Description	
ADC モジュールユニット 1 を初期化します。これは他の ADC 関数を使用する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	
`bool_t Test_ADC_Wait(void)`	
Description	
この関数は、ADC モジュール（ユニット 0）によって AD 変換が行われている間待機します。このテストは ADC 構成を保存しないため、実行時の定期的なテストとしてではなくパワーオンテストとして適しています。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストフェイル

Syntax	
`bool_t Test_ADC_Wait_u1(void)`	
Description	
この関数は、ADC モジュールユニット 1 によって AD 変換が行われている間待機します。このテストは ADC 構成を保存しないため、実行時の定期的なテストとしてではなくパワーオンテストとして適しています。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
bool_t	True = テストパス、False = テストフェイル

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2"><code>void Test_ADC_Start(const ADC_ERROR_CALL_BACK Callback)</code></td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">ADC が使用されるたびに診断テストが実行されるように、ADC モジュール（ユニット 0）をセットアップします。診断基準電圧（「表 1.12 各 RA MCU 内蔵の ADC」を参照）は、それぞれのデバイスに応じて自動的に回転します。
ユーザコードは定期的に、またはすべての AD 変換完了後に <code>Test_ADC_CheckResult</code> 関数を呼び出して、診断結果を確認する必要があります。</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td><code>const ADC_ERROR_CALL_BACK Callback</code></td><td>エラーが検出された場合に呼び出す関数。
注：この関数はパラメータ <code>bCallErrorHandler</code> が True に設定されて <code>Test_ADC_CheckResult</code> が呼び出された場合にのみ呼び出されます。</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td colspan="2"><code>void Test_ADC_Start_u1(const ADC_ERROR_CALL_BACK Callback)</code></td></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td colspan="2">ADC が使用されるたびに診断テストが実行されるように、ADC モジュールユニット 1 をセットアップします。診断基準電圧（「表 1.12 各 RA MCU 内蔵の ADC」を参照）は、それぞれのデバイスに応じて自動的に回転します。
ユーザコードは定期的に、またはすべての AD 変換完了後に <code>Test_ADC_CheckResult_u1</code> 関数を呼び出して、診断結果を確認する必要があります。</td></tr>
<tr><th colspan="2">Input Parameters</th></tr>
<tr><td><code>const ADC_ERROR_CALL_BACK Callback</code></td><td>エラーが検出された場合に呼び出す関数。
注：この関数はパラメータ <code>bCallErrorHandler</code> が True に設定されて <code>Test_ADC_CheckResult_u1</code> が呼び出された場合にのみ呼び出されます。</td></tr>
<tr><th colspan="2">Output Parameters</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
<tr><th colspan="2">Return Values</th></tr>
<tr><td>NONE</td><td>N/A</td></tr>
</table>

Syntax	
`bool_t Test_ADC_CheckResult(const bool_t bCallErrorHandler)`	
Description	
ADC モジュール（ユニット 0）の診断結果が期待どおりであることを確認します。 これは、`Test_ADC_Start` の後に呼び出し、次に定期的に、または ADC 変換が完了するたびに呼び出す必要があります。 注：実際の結果は、期待される結果の特定の許容範囲内であることが許されています。詳細については、`test_adc.c` の `ADC_TOLERANCE` を参照してください。	
Input Parameters	
`const bool_t bCallErrorHandler`	`Test_ADC_Start` 関数に提供されるエラーコールバック関数を呼び出すには True を設定し、そうでない場合は False を設定します。
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = テストパス、False = テストフェイル

Syntax	
`bool_t Test_ADC_CheckResult_u1(const bool_t bCallErrorHandler)`	
Description	
ADC モジュールユニット 1 の ADC 診断結果が期待どおりであることを確認します。 これは、`Test_ADC_Start_u1` の後に呼び出し、次に定期的に、または ADC 変換が完了するたびに呼び出す必要があります。 注：実際の結果は、期待される結果の特定の許容範囲内であることが許されています。詳細については、`test_adc.c` の `ADC_TOLERANCE` を参照してください。	
Input Parameters	
`const bool_t bCallErrorHandler`	`Test_ADC_Start_u1` 関数に提供されるエラーコールバック関数を呼び出すには True を設定し、そうでない場合は False を設定します。
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = テストパス、False = テストフェイル

1.8 温度

RA MCU には、MCU 温度を監視できる温度センサモジュールがあります。ADC モジュールユニット 1 も、温度センサと組み合わせて必要です。

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。

表 1.14 温度ソースファイル

ファイル名
temperature.h
temperature.c

Syntax	

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

Description	
温度センサを初期化し、ADC モジュールを有効にします。許容温度範囲を ADC 出力値で指定します。この関数を呼び出した後、Temperature_Start 関数を定期的に呼び出して、温度センサ出力で ADC 変換を実行し、残りの関数を使用して結果を確認する必要があります。 センサの温度勾配は負であるため、温度の上昇は ADC 変換値の減少を意味し、温度の低下は ADC 変換値の増加を意味します。詳細については、RA MCU のユーザーズマニュアル・ハードウェア編の「温度センサ」および「A/D コンバータ」の章を参照してください。	
Input Parameters	
uint16_t Temperature_ADC_Value_Min	温度センサを読み取るときに ADC が出力する最小値を指定します。
uint16_t Temperature_ADC_Value_Max	温度センサを読み取るときに ADC が出力する最大値を指定します。
TEMPERATURE_ERROR_CALL_BACK Error_callback	この関数は、温度（ADC 値）が指定された許容範囲外の場合、関数 Temperature_CheckResult によって呼び出されます。
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
void Temperature_Start(void)	
Description	
ADC 変換を開始して温度を読み取ります。これは ADC モジュールを使用し、現在の設定を破壊します。この動作に問題がないことを確認するのはユーザの責任です。 この関数に続いて、関数 Temperature_Read_Wait または Temperature_CheckResult を使用します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
void Temperature_Wait_Finish(void)	
Description	
この関数は、Temperature_Start によって開始された温度変換が完了するまでブロックされます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
uint16_t Temperature_Read_Wait (void)	
Description	
この関数は、Temperature_Start によって開始された温度変換が完了するまでブロックし、ADC 値を返します。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
uint16_t	ADC 出力値

Syntax	
`bool_t Temperature_CheckResult(bool_t bCallErrorHandler)`	
Description	
この関数は、`Temperature_Start` によって開始された温度変換が完了するまでブロックし、ADC 出力値が `Temperature_Init` で指定された範囲内にあるかどうかをチェックします。	
Input Parameters	
`bool_t bCallErrorHandler`	True に設定すると温度が指定された制限を下回った場合に `Temperature_Init` に登録されたコールバックが呼び出されます。それ以外の場合は False に設定します。
Output Parameters	
`NONE`	N/A
Return Values	
`bool_t`	True = 結果は指定された制限内です、False = 結果は指定された制限を超えています。

1.9 Port Output Enable（POE）

GPT 用ポートアウトプットイネーブル（POEG）を使用すると、以下のいずれかの条件で汎用 PWM タイマ（GPT）出力端子を出力無効状態にできます。

- GTETRGn 端子の入力レベル検出
- GPT からの出力禁止要求
- コンパレータ割り込み要求検出
- クロック発生回路の発振停止検出
- レジスタ設定

本ソフトウェアは、上記の条件のうち、GTETRGn 入力端子の立ち上がりエッジが検出されたとき、またはクロック発生回路の発振停止が検出されたときに、GPT チャネル 0 の出力端子（GTIOC0A 端子と GTIOC0B 端子）をハイインピーダンス状態に設定する方法を提供します。

POEG の GTETRGn 入力端子の本数（グループ数）は、デバイスにより異なります。

表 1.15 各 MCU の GTETRGn 端子の記号 n（グループ）

MCU	GTETRGn 端子
RA2A1、RA2L1、RA4M1	n = A、B
RA6M1	n = A、B、C、D

POEG のグループはユーザが選択します。ユーザは、システムに合わせて GTETRGn として使用する I/O ポートは `POE.h` ファイルで設定してください。また、`GPT_Init` 関数と `POE_Init` 関数の入力パラメータ「group」には、選択した POEG グループを指定してください。

さらに、POEG によって生成された割り込みの処理を有効にする必要があります。GTETRGn 入力端子の設定と POE 割り込み有効化の詳細については、2.9 章を参照してください。

本ソフトウェアによる POEG による汎用 PWM タイマ（GPT）の出力禁止制御の対象となるのは、GPT のチャネル 0（GTIOC0A 端子と GTIOC0B 端子）のみです。

表 1.16 本ソフトウェアで出力禁止動作の対象としている GPT チャネル

MCU	GPT チャネル	
RA2A1、RA2L1、RA4M1	GPT320	汎用 PWM タイマ 0（32 ビット）
RA6M1	GPT32EH0	汎用 PWM タイマ 0（32 ビット拡張・高分解能）

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。

表 1.17 Port Output Enable ソースファイル

ファイル名
POE.h GPT.h
POE.c GPT.c

■ GPT.c ファイル

Syntax	
`void GPT_Init(POE_group_t group)`	
Description	
この関数は、GPT チャネル 0 の出力禁止制御を、入力パラメータ「group」で指定された POEG グループと関連付けて構成します。 注：PWM 波形出力にかかわる設定は、ユーザソフトウェアで行ってください。（例：GTIOC0A 端子と GTIOC0B 端子の I/O ポートへの割り付け、汎用 PWM タイマ I/O コントロールレジスタ (GTIOR) による GTIOCA 端子/GTIOCB 端子出力禁止値の選択など。）	
Input Parameters	
POE_group_t group	GPT チャネル 0 に関連付ける POEG グループ
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

■ POE.c ファイル

Syntax	
`void POE_Init(POE_CALL_BACK Callback, POE_group_t group)`	
Description	
この関数は、入力パラメータ「group」で指定された POEG グループを構成します。 POEG は、次の場合に GPT チャネル 0 の GTIOC0A 端子、および GTIOC0B 端子をハイインピーダンス状態にします。 1.　GTETRGn 入力端子の立ち上がりエッジが検出された場合（→同時に割り込みが発生） 2.　発振停止が検出された場合（→同時に NMI が発生）	
Input Parameters	
POE_CALL_BACK Callback	GTETRGn 入力ピンの立ち上がりエッジが検出された場合に呼び出される関数
POE_group_t group	本関数により構成される POEG グループ
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void POE_ClearFlags_ga(void)`	
Description	
この関数は、POEG グループ A 設定レジスタ（POEGGA）の PIDF（ポート入力検出フラグ）、IOCF（GPT または ACMPHS からの出力禁止要求検出フラグ）、OSTPF（発振停止検出フラグ）、および SSF（ソフトウェア停止フラグ）をクリアします。 これにより、出力端子がハイインピーダンス状態から解放されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void POE_ClearFlags_gb(void)`	
Description	
この関数は、POEG グループ B 設定レジスタ（POEGGB）の PIDF（ポート入力検出フラグ）、IOCF（GPT または ACMPHS からの出力禁止要求検出フラグ）、OSTPF（発振停止検出フラグ）、および SSF（ソフトウェア停止フラグ）をクリアします。 これにより、出力端子がハイインピーダンス状態から解放されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void POE_ClearFlags_gc(void)`	
Description	
この関数は、POEG グループ C 設定レジスタ（POEGGC）の PIDF（ポート入力検出フラグ）、IOCF（GPT または ACMPHS からの出力禁止要求検出フラグ）、OSTPF（発振停止検出フラグ）、および SSF（ソフトウェア停止フラグ）をクリアします。 これにより、出力端子がハイインピーダンス状態から解放されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
`void POE_ClearFlags_gd(void)`	
Description	
この関数は、POEG グループ D 設定レジスタ（POEGGD）の PIDF（ポート入力検出フラグ）、IOCF（GPT または ACMPHS からの出力禁止要求検出フラグ）、OSTPF（発振停止検出フラグ）、および SSF（ソフトウェア停止フラグ）をクリアします。 これにより、出力端子がハイインピーダンス状態から解放されます。	
Input Parameters	
NONE	N/A
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

Syntax	
extern void poeg_group_event0_isr(void)	
Description	
POEG グループ A の出力禁止要求の割り込みハンドラ。POE_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 poeg_group_event1_isr(void)	
Description	
POEG グループ B の出力禁止要求の割り込みハンドラ。POE_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 poeg_group_event2_isr(void)	
Description	
POEG グループ C の出力禁止要求の割り込みハンドラ。POE_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 poeg_group_event3_isr(void)`	
Description	
POEG グループ D の出力禁止要求の割り込みハンドラ。`POE_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

1.10 GPIO

GPIO リードバックレベル検出機能は、ポートが出力モード時に、端子のデジタル出力レベルを読み出す機能です。端子部の不良診断を行うことができます。

GPIO リードバックレベル検出機能では、以下のような手順で端子をチェックします。

1. ポートコントロールレジスタ 1（PCNTR1）の PDRn ビットを 1 にして出力ポートに設定する
2. ポートコントロールレジスタ 3（PCNTR3）の POSRn ビットに 1 をセットすることで High を出力、PORRn ビットに 1 をセットすることで Low を出力する
3. ポートコントロールレジスタ 2（PCNTR2）の PIDR ビットで各端子の状態を読む

テスト対象のポートは、gpio_config.h ヘッダファイルで指定します。

表 1.18 GPIO ソースファイル

ファイル名
gpio.h gpio_config.h
gpio.c

Syntax	
void GPIO_Start(const GPIO_ERROR_CALL_BACK Callback)	
Description	
この関数は GPIO リードバックレベル検出機能を利用して、ポートが出力モード時に端子のデジタル出力レベルを読み出し、端子部の不良診断を行います。 テスト対象のポートは、gpio_config.h ヘッダファイルで指定します。	
Input Parameters	
const GPIO_ERROR_CALL_BACK Callback	端子部の不良を検出した（ポートの読み出し値が期待値と異なる）場合に呼び出される関数
Output Parameters	
NONE	N/A
Return Values	
NONE	N/A

2. 使用例

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。

以下のセクションでは、各テストタイプ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

すべての 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。

注：この関数は、デバイスを非特権モードにする前に呼び出す必要があります。

関数 `CPU_Test_All` を使用して、すべての CPU テストを自動的に実行できます。

2.1.2 定期的

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_All` 関数を使用して、すべての CPU テストを自動的に実行できます。または 1 回の関数呼び出しで実行されるテストの量を減らすため、ユーザは CPU 定期テストがスケジュールされるたびに、個々の CPU テスト関数を順番に呼び出すことも選択できます。

2.2 ROM

ROM は、その内容の CRC 値を計算し（CRC32）、予め保存しておいた参照 CRC 値（CRC 計算に含まれていない ROM 内の特定の場所に追加する必要がある）と比較することでテストします。参照 CRC 値の計算方法は、使用する開発環境によって異なります。

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。また、結果が一致するためには、テスト対象となる ROM セクションが、事前の CRC 値計算と ROM テストの両方に同じように含まれていることを確認します。

2.2.1 事前の参照用 CRC 計算

(1) GNU/e² studio の場合

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。

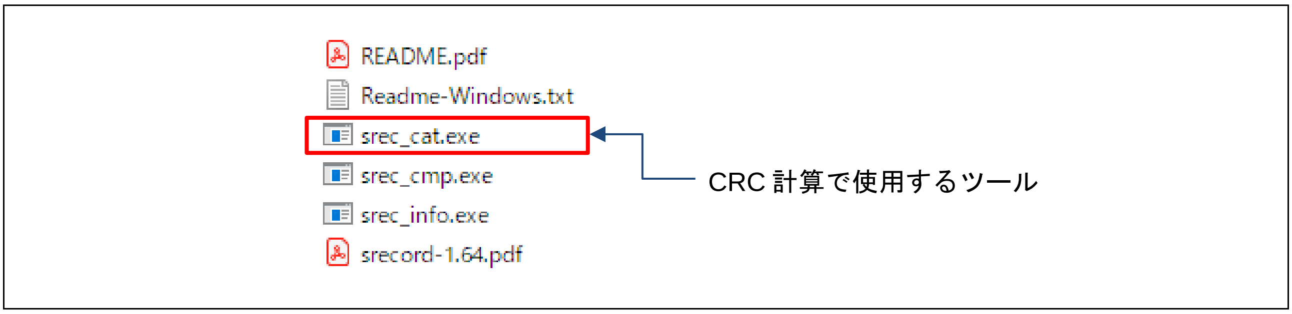

図 2.1 SRecord ツールの内容

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。

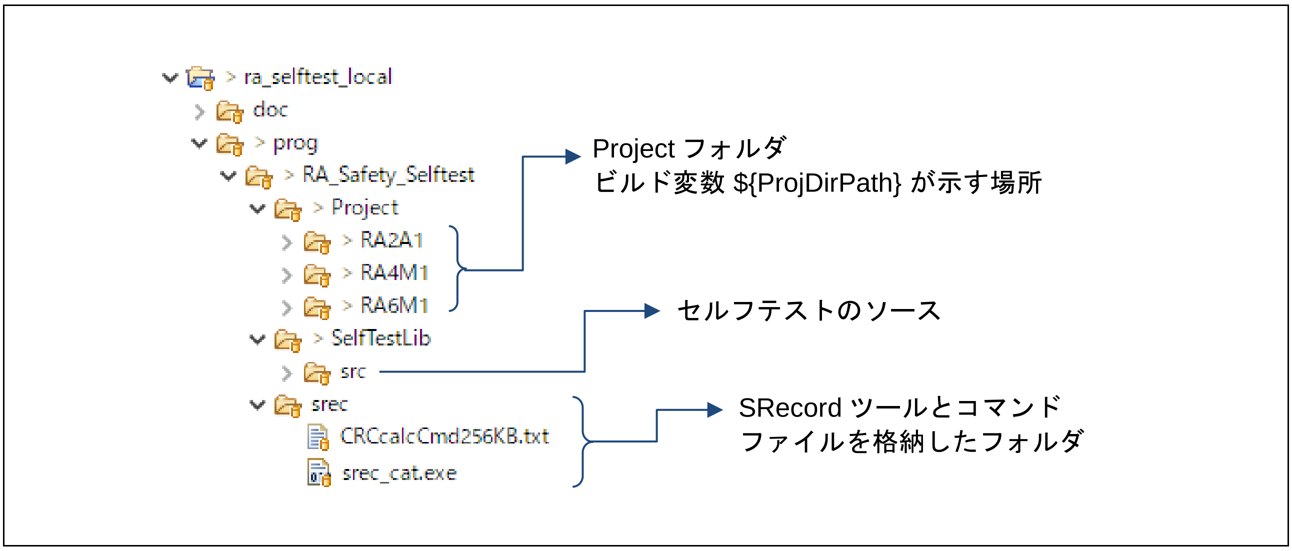

図 2.2 フォルダ構成例

e² studio の「プロジェクト」⇒「プロパティ」を開き、ビルド後のステップで、objcopy コマンドを使って、.elf ファイルから S レコードファイルを生成します。ここでは、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。

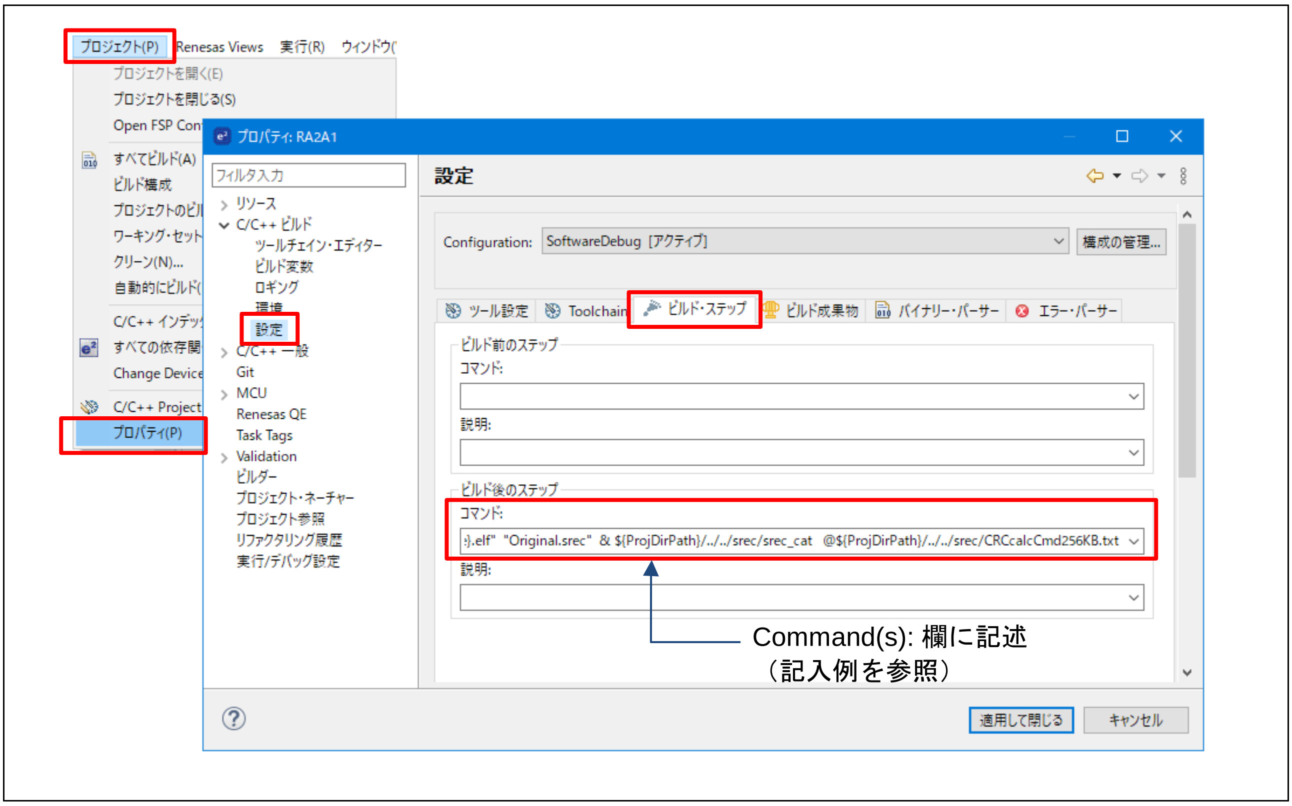

図 2.3 S レコードファイルの出力と SRecord ツールの起動

上図における「Build Steps」タブの「Post-build steps」で、以下のように記述します。

■ 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.srec" &
${ProjDirPath}/../../srec/srec_cat @${ProjDirPath}/../../srec/CRCcalcCmd256KB.txt
```

1 行目の"&"の前までが S レコードファイルの生成、2 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「**CRCcalcCmd256KB.txt**」の記述例を以下に示します。

■ CRCcalcCmd256KB.txt ファイルの内容（例）

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

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。

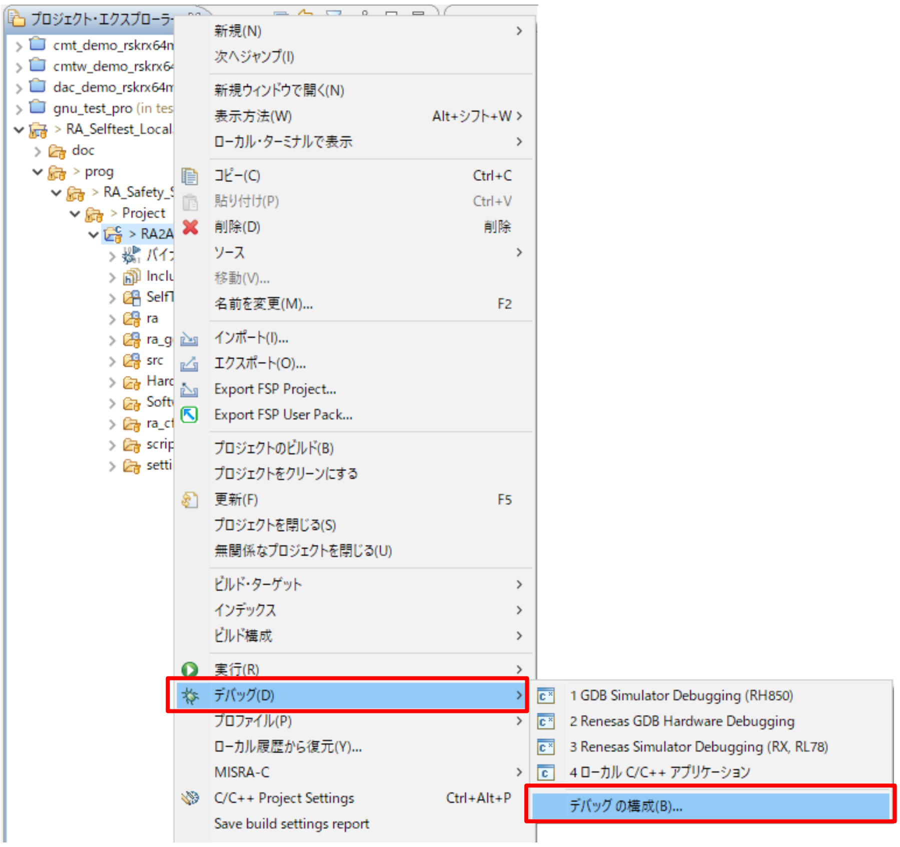

図 2.4 プロジェクトのデバッグ構成の選択

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。

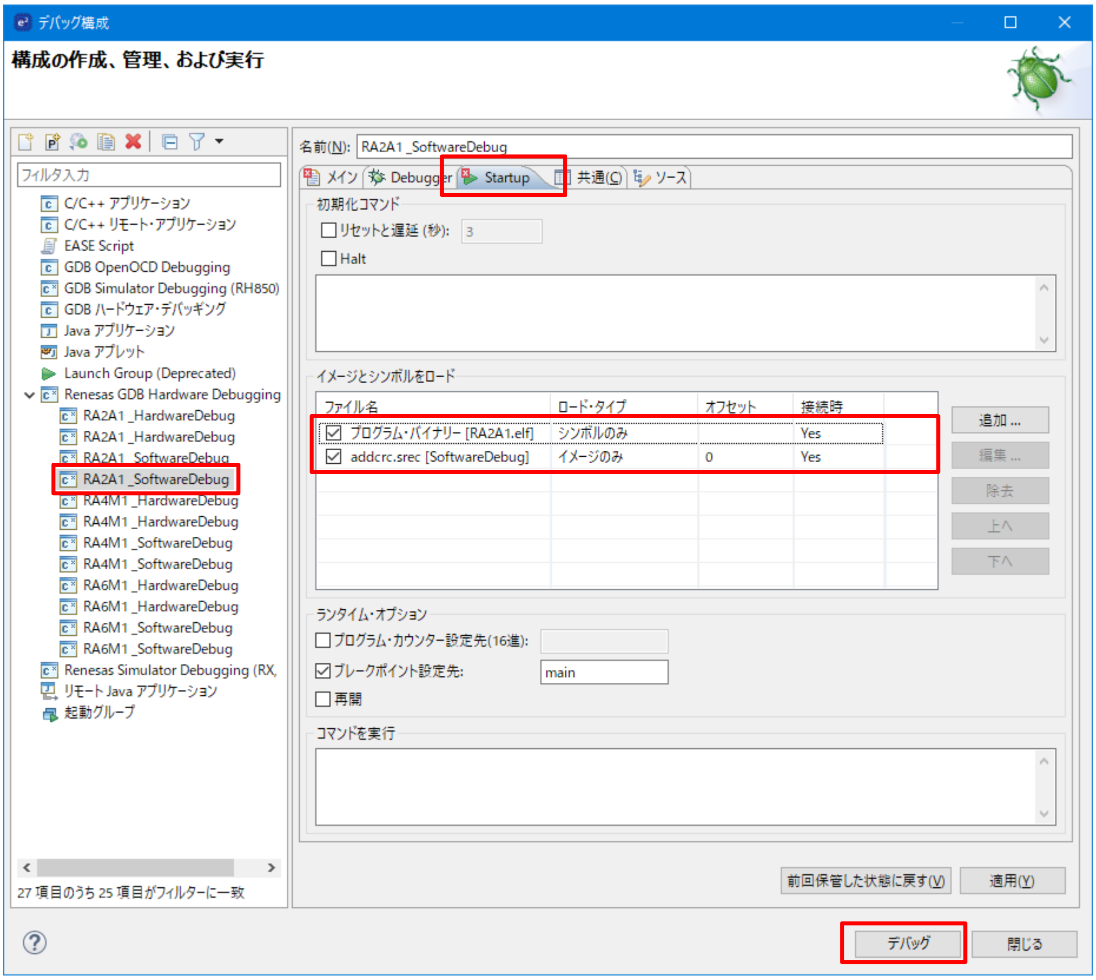

図 2.5 ロードイメージとシンボルの設定例

(2) IAR/EWARM の場合

IAR の EWARM には CRC 値の計算機能があり、ユーザが指定した場所にあるビルド済みファイルに CRC を追加することができます。Project → Options を選択し、Linker オプションで Checksum のタブを選び、CRC 生成に関する設定を行います。

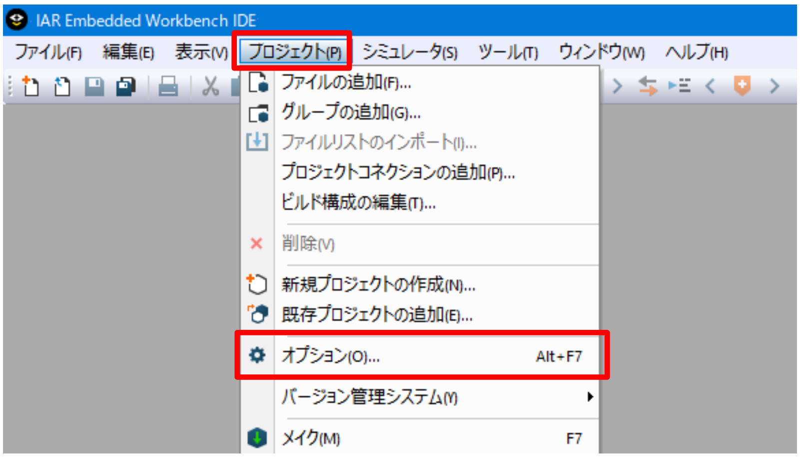

図 2.6 プロジェクト→オプション...の選択

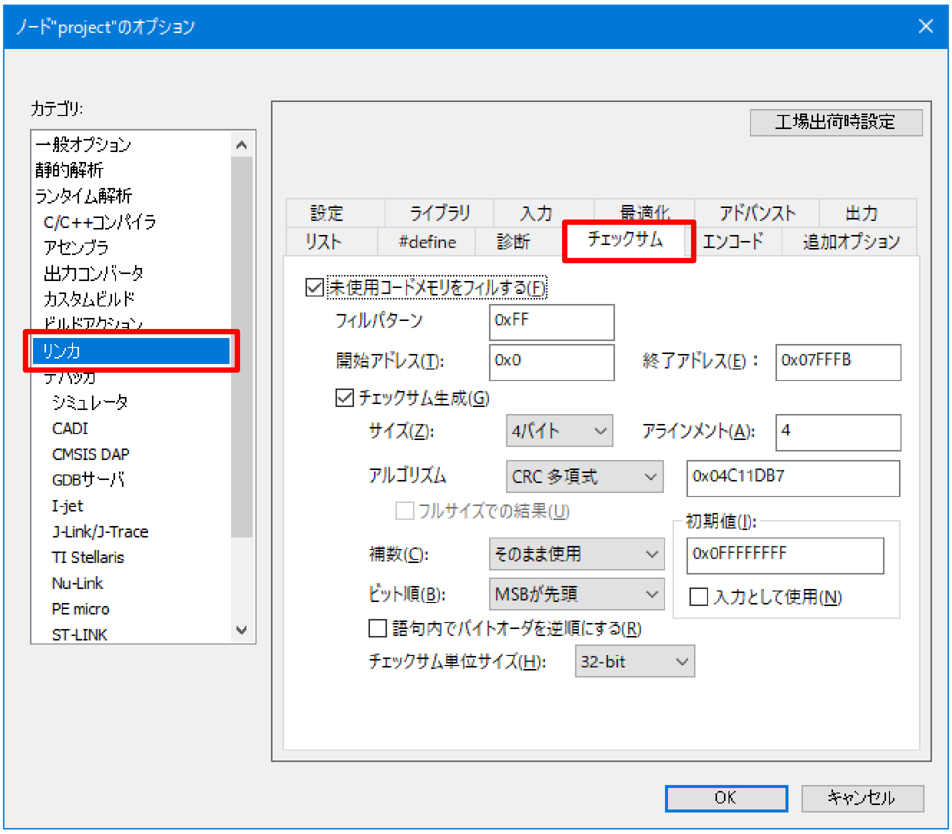

図 2.7 参照用 CRC の追加例（パラメータは MCU に合わせて設定してください）

2.2.2 電源投入時

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。

2.2.3 定期的

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 テストの「HW」バージョンを使用している場合（DOC および DTC が使用されている場合）、テストを実行する前に関数 RamTest_March_HW_Init を呼び出す必要があります。

RAM をテストするときは、次の点に注意してください。

1. テスト中の RAM は、現在のスタックを含め、他の処理には使用しないでください。
2. 非破壊テストでは、テスト対象のメモリ領域を完全にコピーおよび復元できる RAM バッファが必要です。
3. スタックにはメインとプロセスの 2 つがあります。スタックテストでは、2 つのスタックのうちのいずれか、または両方をテストできます。スタックのテストには、対象のスタック領域を再配置できる RAM バッファが必要です。

2.3.1 電源投入時

電源投入時に、スタック以外の RAM で完全な破壊テストを実行できます。スタックは非破壊テストでテストする必要があります。ただし、起動時間が非常に重要な場合は、これを微調整して、電源投入 RAM テストの前に使用されたスタックの領域のみが低速の非破壊テストを使用して実行され、残りのスタックが 破壊テストされます。

2.3.2 定期的

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。定期的なテストは割り込みハンドラから呼び出されるため、デバイスは特権モードであると想定されます。

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
RA2A1	P302	eCLOCK_MONITOR_CACREF_A
	P407	eCLOCK_MONITOR_CACREF_B
RA2L1、RA4M1	P204	eCLOCK_MONITOR_CACREF_A
	P400	eCLOCK_MONITOR_CACREF_B
RA6M1	P402	eCLOCK_MONITOR_CACREF_A
	P600	eCLOCK_MONITOR_CACREF_B
	P708	eCLOCK_MONITOR_CACREF_C

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 オーバーフローに対応するイベント番号を設定します（RA2L1 を除く）。

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。

また、IAR/EWARM の場合は、IAR/EWARM 用の「RA Smart Configurator」を使用することで、同様に ICU の構成を「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（RA6M1、RA4M1、RA2A1）

MCU	イベント名	IELSRn.IELS
RA6M1	CAC_FERRI	0x87
	CAC_OVFI	0x89
RA4M1	CAC_FERRI	0x47
	CAC_OVFI	0x49
RA2A1	CAC_FERRI	0x35
	CAC_OVFI	0x37

表 2.3 CAC 関連の IELSRn レジスタの設定（RA2L1）

MCU	イベント名	IELSRn	IELS[4:0]
RA2L1	CAC_FERRI	group1 (IELSR1/9/17/25)	0x0B
		group5 (IELSR5/13/21/29)	
	CAC_OVFI	group3 (IELSR3/11/19/27)	0x08
		group7 (IELSR7/15/23/31)	

ネスト化ベクタ割り込みコントローラ（NVIC）の設定は、RA_SelfTests.c ファイル内の test_main()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);
```

発振停止を検出すると、NMI 割り込みが発生します。次の例に示すように、ユーザコードはこの NMI 割り込みを処理し、NMISR.OSTST フラグを確認する必要があります。

```
if(1 == R_ICU->NMISR_b.OSTST)
{
   Clock_Stop_Detection();
   /* Clear OSTST bit by writing 1 to NMICLR.OSTCLR bit */
   R_ICU->NMICLR_b.OSTCLR = 1;
}
```

次に、OSTDCR.OSTDF ステータスビットを読み取って、メインクロックのステータスを判別できます。

2.5 独立ウォッチドッグタイマ（IWDT）

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 2.4 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 は無効
IWDT タイムアウト期間選択（IWDTTOPS[1:0]）	11b：2048 サイクル
IWDT 専用クロック分周比（IWDTCKS[3:0]）	1111b：128 分周
IWDT ウィンドウ終了位置（IWDTRPES[1:0]）	11b：0%（ウィンドウの終了位置設定なし）
IWDT ウィンドウ開始位置（IWDTRPSS[1:0]）	11b：100%（ウィンドウの開始位置設定なし）
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

e^2 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。

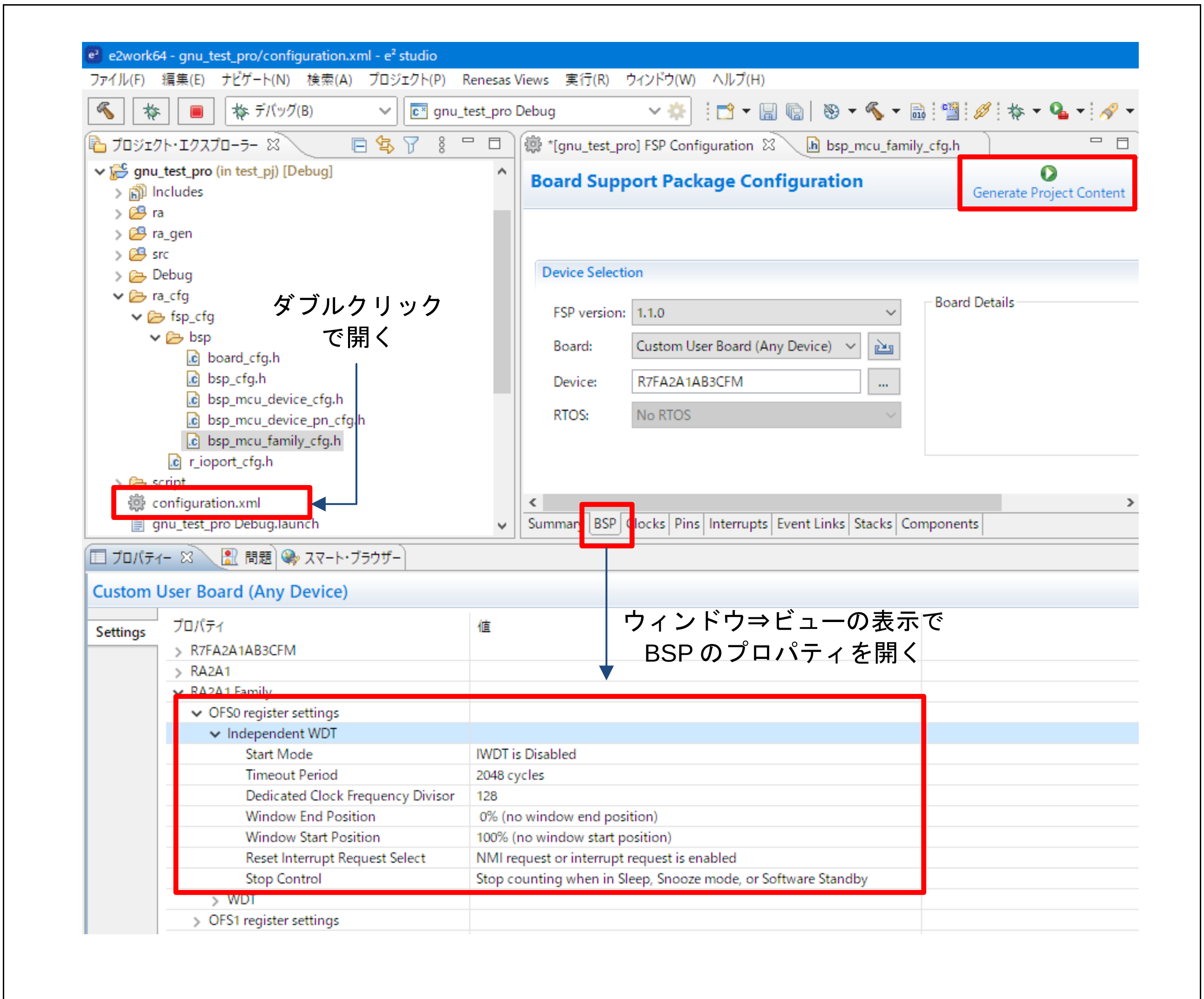

図 2.8 e^2 studio の FSP による OFS0 レジスタ設定例

IAR/EWARM の場合は、IAR/EWARM 用の「RA Smart Configurator (SC)」を使用することで、同様に SC 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。

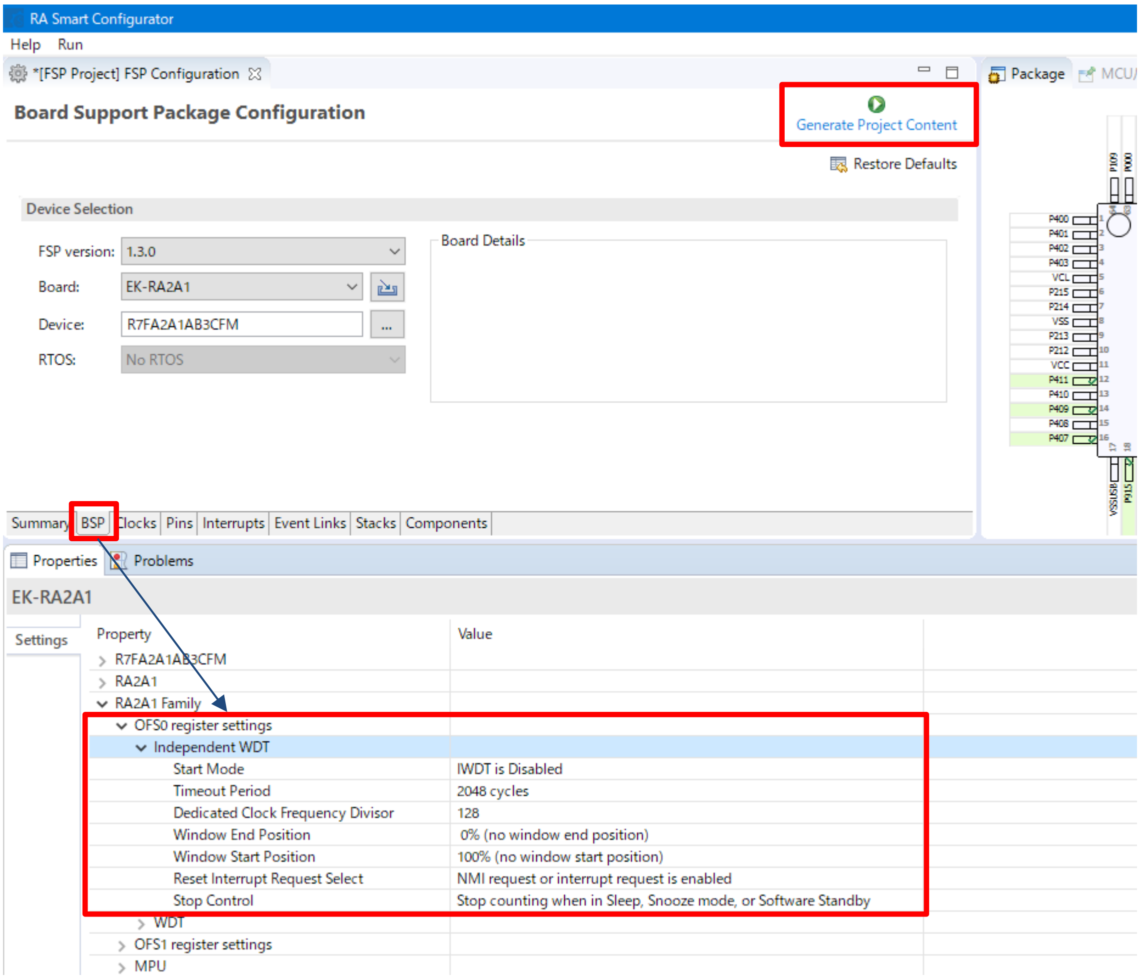

図 2.9 IAR/EWARM 用の RA Smart Configurator (SC) による OFS0 レジスタ設定例

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

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	WDTSTPCTL	—	WDTRSTIRQS	WDTRPSS[1:0]		WDTRPES[1:0]		WDTCKS[3:0]				WDTTOPS[1:0]		WDTSTRT	—
リセット後の値	ユーザの設定値 (注1)															

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	IWDTSTPCTL	—	IWDTRSTIRQS	IWDTRPSS[1:0]		IWDTRPES[1:0]		IWDTCKS[3:0]				IWDTTOPS[1:0]		IWDTSTRT	—
リセット後の値	ユーザの設定値 (注1)															

図 2.10 オプション機能選択レジスタ 0（OFS0）

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*/
   }
}
```

2.6 電圧

低電圧検出（LVD）モジュールは、VoltageMonitor_Init 関数を呼び出して主電源電圧を監視するように構成されています。これは、パワーオンリセット後、できるだけ早くセットアップする必要があります。

`VoltageMonitor_init` 関数を呼び出す前と NMI ルーチンの両方で LVD1SR.DET ビットを 0 に設定することに注意してください。詳細については、各 RA MCU のユーザーズマニュアル・ハードウェア編の「低電圧検出（LVD）モジュール」の章を参照してください。

電圧閾値 eVoltage を Vcc 公称値よりも低く設定することに注意してください。次の例では電圧が **2.99 V** を下回った場合に NMI を生成するように電圧モニタを設定します。

```
VoltageMonitor_Init(VOLTAGE_MONITOR_LEVEL_2_99);
```

低電圧状態が検出されると、ユーザが処理する必要がある NMI 割り込みが生成されます。

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

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

2.7 ADC

ADC モジュールには、さまざまな基準電圧をテストできる診断モードが組み込まれています。許容される不正確さを説明するために、期待される結果は、以下を使用して定義された許容範囲内に収まることが許されています:

```
#define ADC_TOLERANCE 8    // ADC16の場合の値は16
```

この値は ADC の定格である最大絶対精度として設定されます。校正済みのシステムでは、この許容誤差を厳しくすることができます。

ADC テストモジュールは、`Test_ADC_Init` を呼び出して初期化する必要があります。

RA6M1 の ADC12 は 2 ユニット（ユニット 0 とユニット 1）あるため、ユニット 1 のテストを行う場合は「_u1」の付いた関数を使用してください。

2.7.1 電源投入時

電源投入時に、`Test_ADC_Wait` 関数を使用して ADC モジュールをテストできます。この関数は AD 変換が実行されるまで待機します。この関数の戻り値で結果を確認する必要があります。

2.7.2 定期的

定期的なテストは、`Test_ADC_Start` の 1 回の呼び出しで開始する必要があります。その後、ADC モジュールは、使用されるたびにリファレンス変換を実行します。基準電圧は 0 V、VREF/2、VREF（RA2A1 の場合は、0V、+VREF、−VREF）の間で回ります。

これらの参照変換の結果は、`Test_ADC_CheckResult` 呼び出しを使用して定期的にチェックする必要があります。

2.8 温度

MCU の温度をテストするときは、ADC モジュールが使用されることを覚えておくことが重要です。したがって、ユーザのコードでも ADC を使用してアナログピンを監視する場合は、ADC モジュールのリソース共有を慎重に検討することが重要です。

Temperature_Init を呼び出して使用する前に、温度センサを初期化する必要があります。この関数には、ADC 出力で表される許容温度範囲を渡す必要があります。これらの値を実験で計算または検索する方法の詳細については、各 RA MCU のユーザーズマニュアル・ハードウェア編を参照してください。

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

2.8.1 電源投入時

電源投入時に温度テストを行う場合、手順は定期テストで説明したものと同じになります。

2.8.2 定期的

温度センサにより ADC モジュールが定期的に使用されます。温度を読み取るには、ユーザは次の関数を呼び出します。

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

次の呼び出しを使用して、Temperature_Init 関数により提供された許容範囲に対して結果を確認できます。

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

定期的なテストが SW アプリケーションを長時間ブロックすることを回避するために、定期的なテストがスケジュールされるたびに、以前のスケジュールされたテストで開始された温度テストの結果を実際にチェックし、新しい変換を開始するように構成できます。

ユーザのコードは、Temperature_Is_Finished または Temperature_Wait_Finish 関数を使用して、アプリケーションが ADC を使用してアナログピンを読み取ることができる時期を判断できます。

2.9 Port Output Enable（POE）

GPT 用ポートアウトプットイネーブル（POEG）は、汎用 PWM タイマ（GPT）の出力端子を出力禁止状態にする機能です。出力禁止の条件と、各 MCU で使用可能な POEG グループについては、1.9 章をご覧ください。

例えば POEG グループ A を使用する場合の初期化と起動は、次のように POE_Init 関数で行います。

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

POEG グループはユーザが選択します。ユーザは POE_Init 関数の説明を注意深く調べ、RA MCU のユーザーズマニュアル・ハードウェア編を参照して、POE のサンプル構成がユーザのシステムの要件を満たしているかどうかを判断する必要があります。

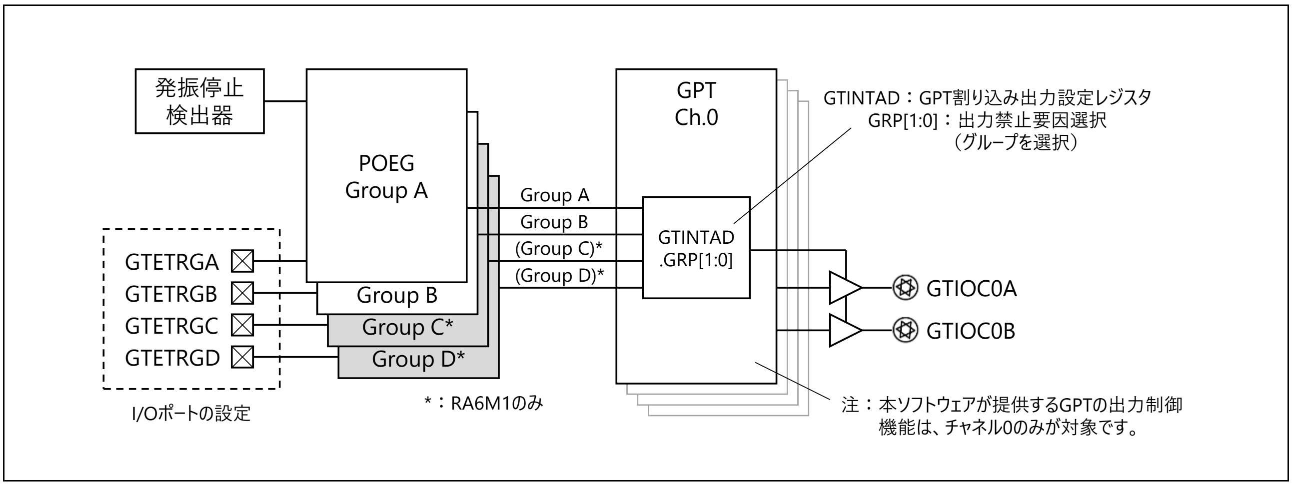

図 2.11 POEG と GPT（汎用 PWM タイマ）の関係概略

2.9.1 I/O ポートの設定

ユーザのシステムで使用されるピンによっては、renesas.h ヘッダファイルを目的の動作に合わせて調整する必要がある場合があります。また、端子を汎用 PWM タイマ（GPT）の GTETRGn として使用する端子に設定します。

POEG は、I/O ポートの制御レジスタで、PmnPFS.PMR、および PmnPFS.PSEL[4:0] ビットの設定によって端子が周辺機能 GPT 対応端子として指定された場合にのみ出力禁止制御を行います。端子が汎用入出力端子に設定されている場合、POEG は出力禁止制御を行いません。

- PmnPFS.PMR = 1　：端子を内蔵周辺機能の入出力ポートとして使用する
- PmnPFS.PSEL[4:0] = 00010b　：GPT 対応端子に指定

表 2.5 I/O ポートの設定（GTETRGn として指定できる I/O ポート）

MCU	GTETRGn	GTETRGn として指定できる I/O ポート (※)
RA6M1	GTETRGA	P100、P105、P401
	GTETRGB	P101、P104
	GTETRGC	P213、P503
	GTETRGD	P212、P504
RA4M1	GTETRGA	P100、P105、P213、P401
	GTETRGB	P101、P104、P212
RA2A1	GTETRGA	P400、P213
	GTETRGB	P109、P212
RA2L1	GTETRGA	P100、P105、P213、P401、P503
	GTETRGB	P101、P104、P212、P504

※：MCU のパッケージによっては対象の I/O ポートが存在しない場合があります。また、システムで他の用途に使用されていないことを確認してください。

2.9.2 割り込みの設定

POE による割り込みを有効にするには、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）の両方のモジュールの設定が必要です。

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 POE グループイベントに対応するイベント番号を設定します（RA2L1 を除く）。

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。

また、IAR/EWARM の場合は、IAR/EWARM 用の「RA Smart Configurator (SC)」を使用することで、同様に ICU の構成を RA SC の「Interrupts」タブで設定できます。

表 2.6 POEG 関連の IELSRn レジスタの設定（RA2A1、RA4M1、RA6M1）

MCU	イベント名	IELSRn.IELS
RA6M1	POEG_GROUP0	0x09A
	POEG_GROUP1	0x09B
	POEG_GROUP2	0x09C
	POEG_GROUP3	0x09D
RA4M1	POEG_GROUP0	0x055
	POEG_GROUP1	0x056
RA2A1	POEG_GROUP0	0x041
	POEG_GROUP1	0x042

表 2.7 POEG 関連の IELSRn レジスタの設定（RA2L1）

MCU	イベント名	IELSRn	IELS[4:0]
RA2L1	POEG_GROUP0	group2 (IELSR2/10/18/26)	0x0B
		group6 (IELSR6/14/22/30)	
	POEG_GROUP1	group3 (IELSR3/11/19/27)	0x0B
		group7 (IELSR7/15/23/31)	

ネスト化ベクタ割り込みコントローラ（NVIC）の設定は、RA_SelfTests.c ファイル内の test_main()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、*POEG0_EVENT_IRQn*, *POEG1_EVENT_IRQn*, *POEG2_EVENT_IRQn* および *POEG3_EVENT_IRQn* は FSP が生成した IRQ 番号です。

```
// POEグループ関連割込みのNVIC側設定例

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

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

2.10 GPIO

各デバイスでテスト対象とするポートの指定は、gpio_config.h ヘッダファイル内の g_GPIO_Test_Port 構造体で（ポート m，ビット n のように）定義します。

ポートの指定例：

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

3. ベンチマーク

3.1 RA4M1 測定結果

ベンチマーク測定環境を以下に示します。

表 3.1 測定環境

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

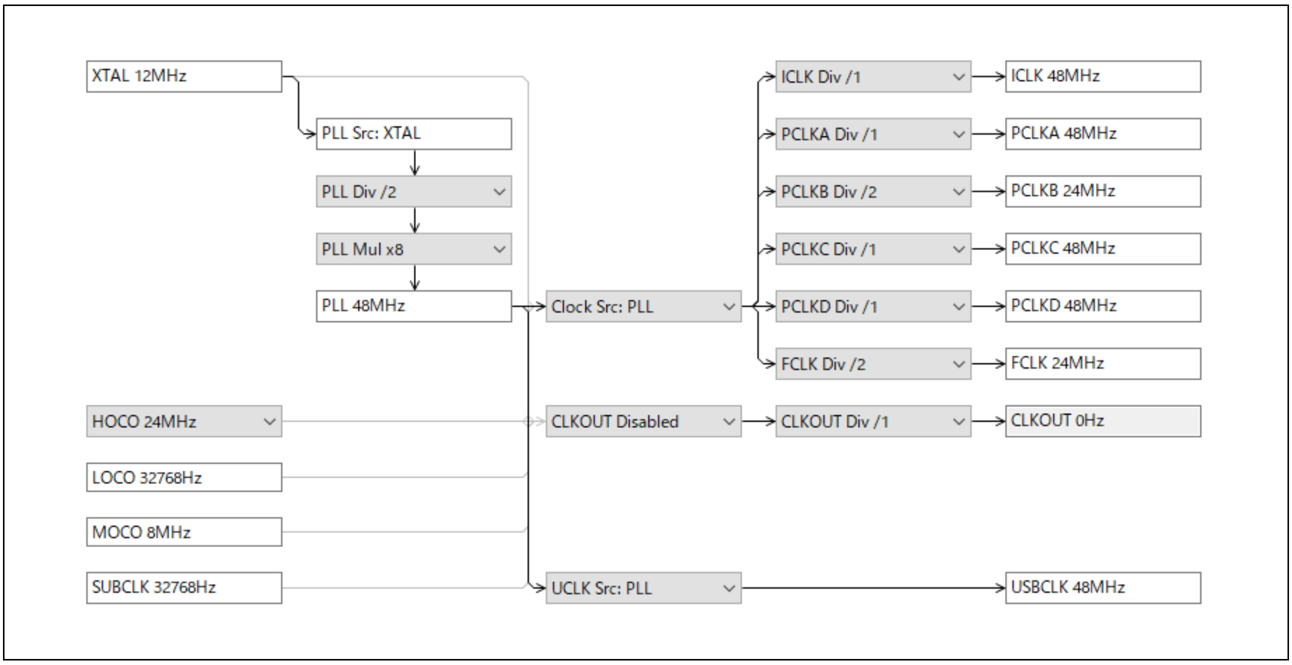

図 3.1 クロック構成

表 3.2 IAR/EWARM ビルドオプション

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

3.1.1 CPU

表 3.3 CPU テスト結果［ICLK：48MHz］

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

3.1.2 ROM

表 3.4 CRC32 テスト結果［ICLK：48MHz］

Measurement		Result	Unit
ROM Usage		114	bytes
RAM Usage		-	bytes
Stack Usage		-	bytes

`CRC_Init`
`CRC_Calculate`
(256KB ROM overall)
`CRC_Verify`
	Clock Cycle Count	783360	-

	Time Measured	16.32	ms

3.1.3 RAM

8 ビットおよび 32 ビットのアクセス幅で行ったテストの測定結果を以下に示します。

(1) March C

表 3.5 March C テスト結果（8-bit access, 32-bit word limit）［ICLK：48MHz］

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

表 3.6 March C テスト結果（32-bit access, 32-bit word limit）［ICLK：48MHz］

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

(2) March X WOM

表 3.7 March X WOM テスト結果（8-bit access, 32-bit word limit）［ICLK：48MHz］

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

表 3.8 March X WOM テスト結果（32-bit Access, 32-bit Word limit）［ICLK：48MHz］

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

(3) スタックテスト

表 3.9 スタックテスト結果［ICLK：48MHz］

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

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

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。

改訂履歴

Rev.	発行日	説明	
		ページ	ポイント
1.00	2020.12.18	―	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

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。

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/

© 2020 Renesas Electronics Corporation. All rights reserved.