ルネサス製コンパイラ
professional 版コンパイラ編

R20UT4026JJ0105
Rev.1.05
2021.3.15

目次

第 1 章　はじめに ... 3

1.1　ルネサス製コンパイラのライセンスの種類 ... 3

1.2　professional 版機能の評価方法 ... 4

第 2 章　MISRA-C:2004/2012 ルールによるソース・チェック機能 ... 5

2.1　MISRA-C:2004/2012 ルール ... 5

2.2　チェック可能なルール数 ... 5

2.3　指定方法 ... 5

2.4　C ソース例 ... 7

第 3 章　スタック破壊検出機能 ... 10

3.1　機能の概要 ... 10

3.2　生成コードのイメージ ... 11

3.3　使用方法 ... 12

3.4　C ソース例 ... 13

第 4 章　動的メモリ管理関数のセーフティ向上機能 ... 15

4.1　機能の概要 ... 15

4.2　生成コードのイメージ ... 16

4.3　使用方法 ... 17

4.4　C ソース例 ... 18

第 5 章　半精度浮動小数点 ... 21

5.1　機能の概要 ... 21

5.2　生成コードのイメージ ... 22

5.3　指定方法 ... 22

5.4　C ソース例 ... 23

第 6 章　制御レジスタ更新時の同期化機能 ... 24

6.1　機能の概要 ... 24

6.2　生成コードのイメージ ... 25

6.3 指定方法 ... 25
6.4 C ソース例 ... 27
6.5 補足事項 ... 29

第 7 章 不正な間接関数呼び出し検出機能 ... 30

7.1 機能の概要 ... 30
7.2 生成コードのイメージ ... 30
7.3 使用方法 ... 32
7.4 C ソース例 ... 34

第 8 章 改版履歴 ... 36

第1章　　はじめに

本書ではルネサス製コンパイラの professional 版を導入することを検討されているお客様向けに、professional 版に特化した機能の概要・使用方法・有効となる C ソース例等をご紹介します。

1.1 ルネサス製コンパイラのライセンスの種類

ルネサス製コンパイラ CC-RL／CC-RX／CC-RH には standard 版と professional 版のライセンスをご用意しています。

- **standard 版**

  C90/C99 規格に準拠した C 言語仕様をサポートします。CC-RX は C++言語もサポートします。

  また、強力な最適化機能と、組み込みプログラム記述に必要な基本機能を提供します。

- **professional 版**

  standard 版に加えて、以下のようなお客様のプログラムの性能や品質向上と開発期間の短縮に貢献する付加機能を提供します。また、今後も機能拡張を予定しています。

表 1-1 professional 版の機能一覧

付加機能	CC-RL	CC-RX	CC-RH
MISRA-C:2004/2012 ルールによるソース・チェック機能	○	○	○
スタック破壊検出機能	○	○	○
動的メモリ管理関数のセキュリティ強化	○	○	○
半精度浮動小数点	－	－	○
制御レジスタ更新時の同期化機能	－	－	○
不正な間接関数呼び出し検出機能	○	○	○

○：対応　　－：対応予定なし

professional 版の機能を使用するには、professional 版のコンパイラ ライセンスをご購入頂くか、以下のいずれかの方法でも professional 版の機能をご使用いただけます。

- **アップグレード（エディション）ライセンス**

standard 版ライセンスを保有されている場合、standard 版から professional 版へお得にアップグレードできる、アップグレード（エディション）ライセンスを販売しています。

ただし、本ライセンスはノードロック・ライセンス（permanent）のみ対応しています。フローティング・ライセンスや annual ライセンスには対応していません。

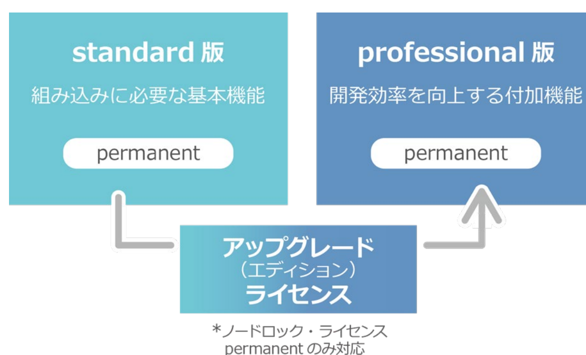


図 1-1 アップグレード（エディション）ライセンス

- **annual ライセンス**

有効期間を 1 年間としたライセンスです。professional 版 annual ライセンスにより、professional 版の機能を 1 年間ご使用可能です。従来の permanent ライセンスに対して低価格で導入いただけます。

また、annual ライセンスは開発メンバーの増減にフレキシブルに対応可能です。

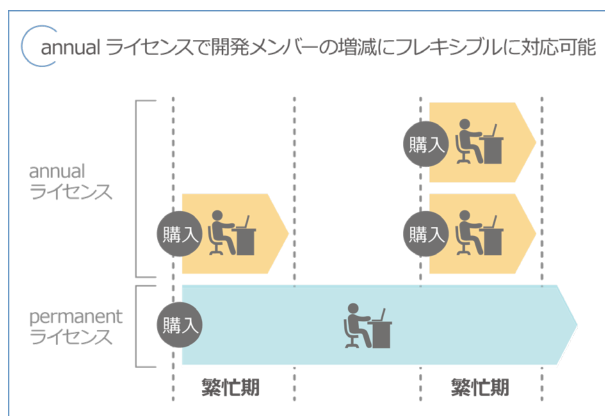


図 1-2 annual ライセンス

1.2 professional版機能の評価方法

professional 版の機能を評価・確認する場合、無償評価版をご使用ください。

初めて無償評価版をインストールした後、最初にビルドした日から 60 日間は professional 版の機能をご試用頂けます。61 日目以降は standard 版の機能に制限され、リンクサイズ（生成できるプログラムのサイズ）も制限されます。

また、本アプリケーションノートでは、professional 版の機能が有効となる C ソース例を記載しておりますので、ご活用ください。

第2章 MISRA-C:2004/2012ルールによるソース・チェック機能

コンパイラ起動時に MISRA-C ルール・チェックを行い、ルールから逸脱するソース記述がある場合にメッセージを出力します。この機能により、ユーザープログラムの品質向上が可能となります。

2.1 MISRA-C:2004/2012ルール

MISRA-C とは、MISRA (Motor Industry Software Reliability Association) が開発した C 言語のためのソフトウェア設計標準規格です。C 言語で記述する組み込みシステムで安全性・可搬性・信頼性を確保することを目的とした規格であり、2004 年に規定されたルールが MISRA-C:2004、2012 年に規定されたルールが MISRA-C:2012 です。

2.2 チェック可能なルール数

表 2-1 MISRA-C:2004 のルール数

ルール分類	**CC-RL** V1.10.00	**CC-RX** V3.03.00	**CC-RH** V2.03.00
必要ルール（121 件）	79	79	79
推奨ルール（20 件）	13	13	13
合計（141 件）	**92**	**92**	**92**

表 2-2 MISRA-C:2012 のルール数

ルール分類	**CC-RL** V1.10.00	**CC-RX** V3.03.00	**CC-RH** V2.03.00
必須ルール（16 件）	7	7	7
必要ルール（108 件）	90	90	90
推奨ルール（32 件）	27	27	27
合計（156 件）	**124**	**124**	**124**

なお、各リビジョンによって対応しているルール数は異なります。

2.3 指定方法

コンパイラ・オプションを指定することで、簡単に MISRA-C:2004/2012 ルールチェッカを起動できます。

また、オプションには引数を指定することも可能で、引数によってチェックするルール番号、無視するルール番号等の制御も可能です。

表 2-3 MISRA-C:2004/2012 ルールチェッカのオプション一覧

	オプション		
説明	**CC-RL**	**CC-RX**	**CC-RH**
MISRA-C:2004 ルールによるソース・チェックを行います。	-misra2004	-misra2004	-Xmisra2004
MISRA-C:2012 ルールによるソース・チェックを行います。	-misra2012	-misra2012	-Xmisra2012
ソース・チェックの対象外のファイルを指定します。	-ignore_files_misra	-ignore_files_misra	-Xignore_files_misra
言語拡張により部分抑止されるルールのソース・チェックを有効にします。	-check_language_extension	-check_language_extension	-Xcheck_language_extension
複数ファイルにまたがる MISRA-C:2012 ルールによるソース・チェックを行います。※	-misra_intermodule	-misra_intermodule	-misra_intermodule

※CC-RL V1.08、CC-RX V3.01.00、CC-RH V2.01.00 以降に指定が可能です。

統合開発環境 CS+、あるいは e^2 studio を使用している場合は、GUI 上の操作でオプション指定を制御することが可能です。

【CS+の場合】
[コンパイル・オプション] タブ -> [MISRA-C ルール検査] カテゴリ -> [MISRA-C 規格] プロパティで 2004 ルール/2012 ルールを選択し、[適用するルール]・[ルール・チェック対象外のファイル]・[拡張キーワードや拡張仕様をメッセージ出力する] ・[複数ファイルにまたがる検査を有効にする]プロパティ等で詳細な設定が可能です。

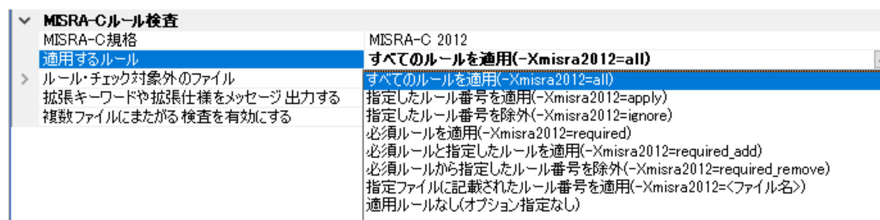


図 2-1 CS+の設定方法

【e^2 studio の場合】
[プロジェクト] -> [Renesas Tool Settings] でプロジェクトのプロパティダイアログを起動して[C/C++ ビルド] -> [設定] を選択、[Tool Settings] タブ内で[Compiler] -> [MISRA-C ルール検査] -> [MISRA-C ルールによるソースチェックを行う] で 2004 ルール/2012 ルールを選択し、[適用するルール]・[ルール・チェック対象外のファイル]・[C 言語規格から独自に拡張した言語仕様が有効な場合に一部抑止されるルールチェックを有効する]・[複数ファイルにまたがる検査を有効にする]等で詳細な設定が可能です。

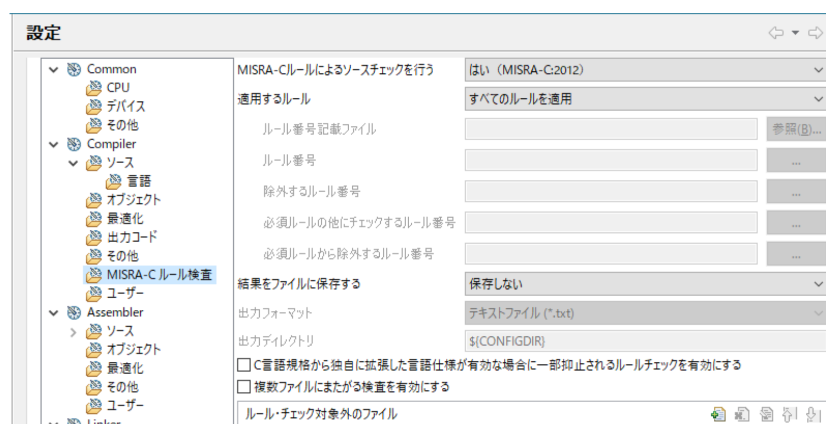


図 2-2 e^2 studio の設定方法

2.4 Cソース例

MISRA-C:2004/2012 ルールに違反する C ソース例と出力メッセージを説明します。

【例 1】MISRA-C: 2012 ルール番号 2.7 に違反する場合

MISRA-C 規格	ルール番号	分類	ガイドライン
MISRA-C:2012	2.7	推奨ルール	関数内に未使用のパラメータがあってはならない

```
1:   typedef signed int int32_t;
2:
3:   void func(int32_t a, int32_t b);
4:   void sub_func(int32_t a);
5:
6:   void func(int32_t a, int32_t b){
7:
8:     if (a != 0){
9:       sub_func(a);
10:    }
11:
12:    /* パラメータ変数 b を使用しない。*/
13:    return;
14:  }
```

6 行目で宣言されているパラメータ変数 b は、関数 func 内で使用していないため、以下のようなメッセージを標準エラー出力に出力します。CS+を使用している場合は出力ウィンドウ、e^2 studio を使用している場合はコンソールに出力します。

```
ファイル名.c(6):M0523086:Rule 2.7:There should be no unused parameters in functions
```

【例 2】MISRA-C:2004 ルール番号 9.2 と MISRA-C:2012 ルール番号 9.3 に違反する場合

MISRA-C 規格	ルール番号	分類	ガイドライン
MISRA-C:2004	9.2	必要ルール	配列及び構造体を 0 以外で初期化する場合は、構造を示し、それに合わせるために波括弧"{ }"を用いなければならない。
MISRA-C:2012	9.3	必要ルール	配列は部分的に初期化してはならない

```
1:   typedef int int32_t;
2:   #define ARRAY_SIZE 10
3:
4:   extern int32_t array_a[ARRAY_SIZE] = {1,2,3,4,5,6,7,8,9};
```

4 行目は配列の要素数 10 に対して 9 つの要素しか初期化されていないため以下のようなメッセージを標準エラー出力に出力します。CS+を使用している場合は出力ウィンドウ、e[2] studio を使用している場合はコンソールに出力します。

MISRA-C:2004 ルールの場合

ファイル名.c(4):M0523028:*Rule 9.2: Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.*

MISRA-C:2012 ルールの場合

ファイル名.c(4): M0523086:*Rule 9.3: Arrays shall not be partially initialized*

【例 3】MISRA-C:2004 ルール番号 10.1 と MISRA-C:2012 ルール番号 10.3 に違反する場合

MISRA-C 規格	ルール番号	分類	ガイドライン
MISRA-C:2004	10.1	必要ルール	次の条件に該当する場合、整数型の式の値を異なる潜在型に暗黙的に変換してはならない。 (a) 同じ符号属性をもつより大きな整数型への変換でない場合 (b) 式が複合式である場合 (c) 式が定数でなく、関数の実引数である場合 (d) 式が定数でなく、return 式である場合
MISRA-C:2012	10.3	必要ルール	式の値はより狭い実質的な型または異なる実質的な型分類でオブジェクトに代入してはならない

```
1:  typedef unsigned short uint16_t;
2:
3:  extern uint16_t b = sizeof(b);
```

3 行目は unsigend short 型の変数に、異なる型（sizeof 演算子の戻り値）を代入しているため、以下のようなメッセージを標準エラー出力に出力します。CS+を使用している場合は出力ウィンドウ、e[2] studio を使用している場合はコンソールに出力します。

MISRA-C:2004 ルールの場合

ファイル名.c(3):M0523028:*Rule 10.1: The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (a) it is not a conversion to a wider integer type of the same signedness, or (b) the expression is complex, or (c) the expression is not constant and is a function argument, or (d) the expression is not constant and is a return expression*

MISRA-C:2012 ルールの場合

ファイル名.c(3): M0523086:*Rule 10.3: The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category*

【例 4】MISRA-C:2012 ルール番号 12.1 と 20.7 に違反する場合

MISRA-C 規格	ルール番号	分類	ガイドライン
MISRA-C:2012	12.1	推奨ルール	式の中の演算子の優先順位は明白でなければならない
MISRA-C:2012	20.7	必要ルール	マクロパラメータの展開の結果生じる式は、括弧で囲まれなければならない

```
1:  typedef int int32_t;
2:
3:  #define FIELD_SIZE(x)   (x * 2)
4:  #define MAIN_SIZE        128
5:  #define HEADER_SIZE     16
6:
7:  extern int32_t text_areasize;
8:
9:  int32_t text_areasize = FIELD_SIZE(MAIN_SIZE - HEADER_SIZE);
```

9 行目はマクロ展開された際の式中の優先順位が明白でなく、またマクロの引数に括弧がないため、以下のようなメッセージを標準エラー出力に出力します。CS+を使用している場合は出力ウィンドウ、e^2 studio を使用している場合はコンソールに出力します。

なお、マクロ展開された式は、「128 -16 * 2 」となるため、「(128 -16) * 2 」を意図していた場合誤った結果になります。

```
ファイル名.c(9):M0523086:Rule 20.7: Expressions resulting from the expression of macro parameters shall be enclosed in parentheses
ファイル名.c(9):M0523086:Rule 12.1: The precedence of operators within expressions should be made explicit
```

第3章　スタック破壊検出機能

スタック領域の破壊有無を動的にチェックするコードをコンパイラが生成することにより、スタックのオーバーフローやセキュリティアタックの防止といった安全性を向上したプログラム開発が可能となります。

3.1 機能の概要

スタックは関数ごとに関数の入口（プロローグ処理）で確保され、その関数内で使用するローカル変数領域やレジスタを退避する領域から構成されます。スタック破壊検出機能を使用したスタックの場合、当該関数のスタックのローカル変数領域直前（上位番地に向かう方向）に 4 バイトの領域（CC-RL は 2 バイト）を確保し、指定値を埋め込みます。この値はユーザーが指定することもできますし、コンパイラに任意の値を指定させることもできます。本書ではこれを「監視領域」と呼びます。

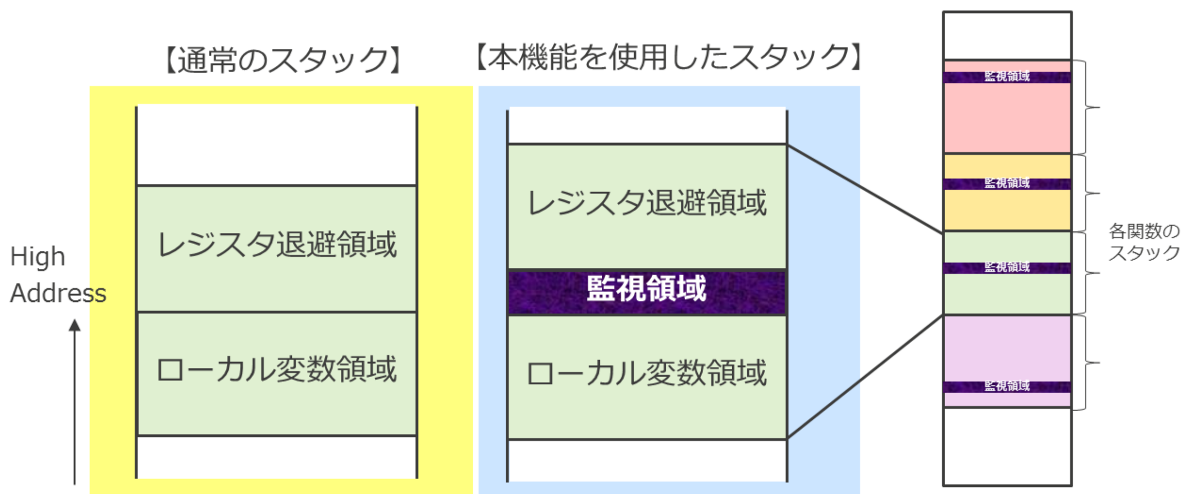


図 3-1 スタックのイメージ

当該関数を実行後、スタックを解放します。この際に、監視領域に埋め込んだ値が書き変わっていないかをチェックするコードを関数出口（エピローグ処理）に生成します。監視領域の値が書き変わっている場合、スタックのローカル変数領域でオーバーフロー等が発生し、監視領域や上位番地の領域（ここではレジスタ退避領域）を破壊している可能性があると判断できます。

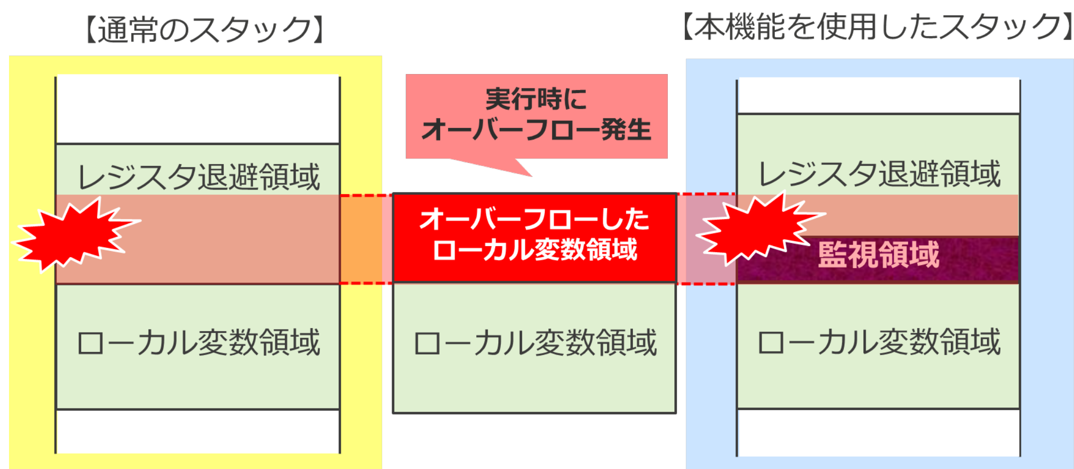


図 3-2 スタック破壊のイメージ

レジスタ退避領域には、この関数終了時に復帰する関数のアドレス情報も退避されていますので、復帰先アドレスが壊されてしまうと、全く意図しないアドレスに飛んでしまってプログラムが暴走してしまいます。

このようにスタック破壊が発生した場合に、本機能を使用している場合にはエラー関数に分岐します。エラー関数に分岐することで、スタック破壊が発生したことを動的に確認することが可能となり、プログラムの暴走を未然に防止することができます。

3.2 生成コードのイメージ

本機能を使用していない場合は、スタックを確保後に関数本体を実行し、関数本体を実行後にスタックを解放して関数を終了します。

本機能の有効時は、スタックを確保する際にスタック領域内に監視領域を埋め込み、関数本体を実行後に監視領域が書き換わっていないかをチェックします。書き換わっている場合はエラー関数"__stack_chk_fail"に分岐します。

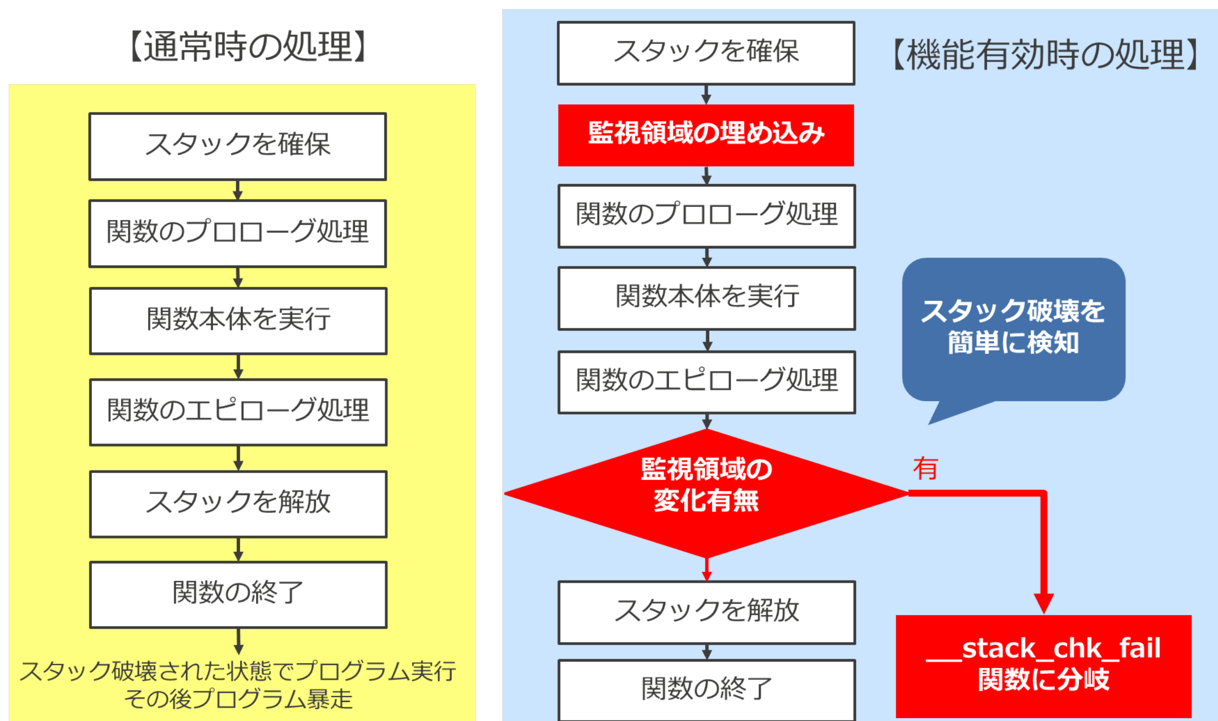


図 3-3 コンパイラの生成コード

本機能を使用していない場合、スタック破壊が発生すると、その後の処理でプログラムが暴走してしまう可能性があります。デバッグ時であれば、暴走の要因を調査することから始めなければなりません。

本機能を使用している場合、プログラムを実行時に"__stack_chk_fail"が呼び出されますので、スタック破壊が発生しても暴走前にプログラムを停止させることができますし、デバッグ時であれば、"__stack_chk_fail"が呼び出された関数を簡単に特定し、その関数の処理を早期に見直すこともできます。

なお、"__stack_chk_fail"関数はユーザーが定義する必要があります。スタックの破壊検出時に実行する処理を記述してください。

3.3 使用方法

コンパイラ・オプション、または拡張言語によりスタック破壊検出機能を有効にすることが可能です。

a. コンパイラ・オプションで指定

以下のコンパイラ･オプションを指定することで､特定の条件を持つ関数をスタック破壊検出対象としたり､全ての関数をスタック破壊検出対象とすることが可能です。また、引数に数値を指定することで、監視領域に引数で指定した値を埋め込みます。引数を省略した場合は、コンパイラが自動的に数値を指定し埋め込みます。

表 3-1 スタック破壊検出機能のオプション一覧

	オプション		
説明	**CC-RL**	**CC-RX**	**CC-RH**
8byte を超える構造体、共用体または配列をローカル変数に持つ関数をスタック破壊検出対象とします。	-stack_protector	-stack_protector	-Xstack_protector
全ての関数をスタック破壊検出対象とします。	-stack_protector_all	-stack_protector_all	-Xstack_protector_all

統合開発環境 CS+、あるいは e2 studio を使用している場合は、GUI 上の操作でオプション指定を制御することが可能です。

【CS+の場合】

[コンパイル・オプション] タブ -> [品質向上関連] カテゴリ -> [スタック破壊検出を行う] プロパティで ON/OFF を選択し､[スタック破壊検出用の埋め込み値] プロパティで監視領域に埋め込む値を指定可能です。

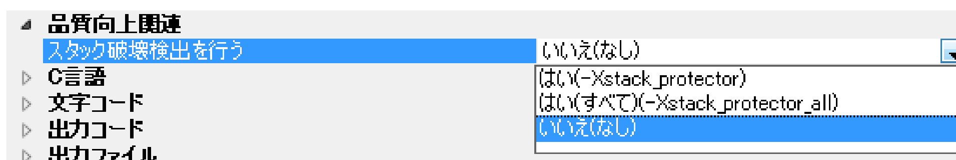


図 3-4 CS+の設定方法

【e2 studio の場合】

[プロジェクト] -> [Renesas Tool Settings] でプロジェクトのプロパティダイアログを起動して[C/C++ビルド] -> [Settings] を選択、[Tool Settings] タブ内で[Compiler] -> [その他] -> [スタック破壊検出を行う] で ON/OFF を選択し、[スタック破壊検出用の埋め込み値] で監視領域に埋め込む値を指定可能です。

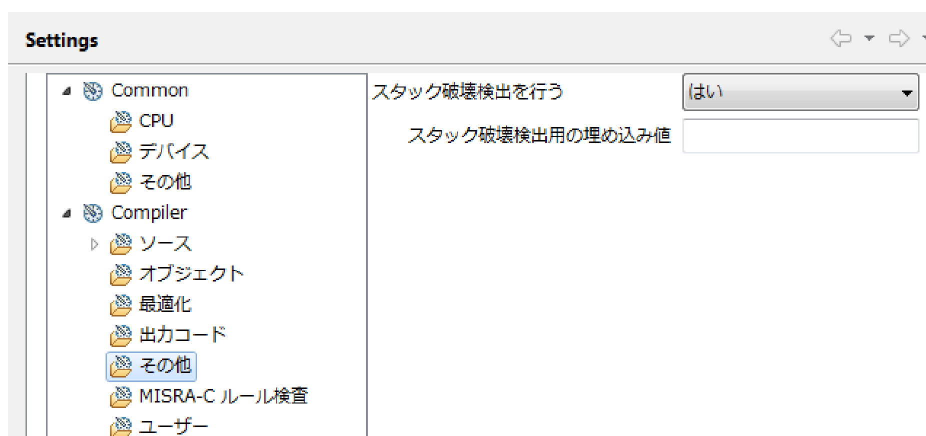


図 3-5 e2 studio の設定方法

b. 拡張言語で指定

以下の拡張言語を指定することで、特定の関数をスタック破壊検出対象とすることが可能です。

【指定形式】

```
#pragma stack_protector 関数名 (num=指定値)
```

*関数名*で指定した関数をスタック破壊検出対象として、監視領域に*指定値*を埋め込みます。また(num=*指定値*)を省略した場合はコンパイラが自動的に数値を指定し埋め込みます。

【指定形式】

```
#pragma no_stack_protector 関数名
```

*関数名*で指定した関数をスタック破壊検出対象としません。コンパイラ・オプションと拡張言語を両方指定した場合は拡張言語の指定を優先します。

3.4 C ソース例

本機能が有効となる例を C ソースで説明します。なお組込み関数の__halt();は CC-RX にはありません。

【例 1】領域計算を誤った場合

```
1:  #include <stdlib.h>
2:  #include <string.h>
3:
4:  typedef struct{
5:     char e_c[2];
6:     char line[8];
7:  } str_t;
8:
9:  #define STR_MAX 16
10: #define BUF_SIZE (sizeof(str_t*) * STR_MAX)
11:
12: #pragma stack_protector func
13: void func (str_t * str);
14:
15: void func (str_t * str){
16:    int i;
17:    char buf[BUF_SIZE];
18:
19:    for(i=0; i< BUF_SIZE; i+=sizeof(str_t)){
20:       memcpy(&buf[i], str, sizeof(str_t));
21:    }
22: }
23:
24: void __stack_chk_fail(void) {
25:    __halt();
26: }
```

10 行目で sizeof(str_t)にするつもりであったところを、sizeof(str_t*)と誤って記述していた場合、BUF_SIZE の値が構造体 str_t のサイズ(10)×16 ではなく、ポインタサイズ×16 になります。そのため、想定よりも小さい領域の確保となり、19 行目～21 行目までの for 文で確保した領域以上の領域に書き込みをしてしまいスタック破壊が起きます。

本機能を有効にしている場合には、"func"関数終了時にエラー関数"__stack_chk_fail"をコールしますので、スタック破壊を簡単に検出することができます。

【例 2】排他漏れの場合

```
1:   #define I_MAX (10)
2:   #define S_MAX (20)
3:
4:   int g_cnt_max;
5:   int s_buf[S_MAX];
6:
7:   void func(int a){
8:     int i;
9:     int buf[I_MAX];
10:
11:    if(I_MAX > a) {
12:      g_cnt_max = a;
13:    } else {
14:      g_cnt_max = I_MAX;
15:    }
16:
17:    /* <= intrpt_func()の割り込み発生 */
18:
19:    for (i= 0; i < g_cnt_max; i++){
20:      buf[i] = s_buf[g_cnt_max-i-1];
21:    }
22:  }
23:
24:  #pragma interrupt intpt_func
25:  void intrpt_func(){
26:    g_cnt_max = S_MAX;
27:  }
28:
28:  void __stack_chk_fail(void) {
29:    __halt();
30:  }
```

17 行目で、"intrpt_func"関数の割り込みが発生した場合、変数 g_cnt_max の値が書き換わり、19 行目から始まる while 文の処理でローカル変数 buf の領域以上の書き込みが発生し、スタック破壊が起きます。

本機能を有効にしている場合には、 "func"関数終了時にエラー関数"__stack_chk_fail"をコールしますので、スタック破壊を簡単に検出することができます。

第4章　動的メモリ管理関数のセーフティ向上機能

セーフティ機能を持った calloc／malloc／realloc 関数を使用してヒープ領域を確保することにより、ヒープ領域の二重解放やオーバーフローの防止といった安全性を向上したプログラム開発が可能です。

4.1 機能の概要

calloc／malloc／realloc 関数によりヒープ領域を確保する際、ヒープ領域の前後それぞれに 4 バイト（CC-RL は 2 バイト）を別に確保し、任意の値を埋め込みます。本書ではこれを「監視領域」と呼びます。

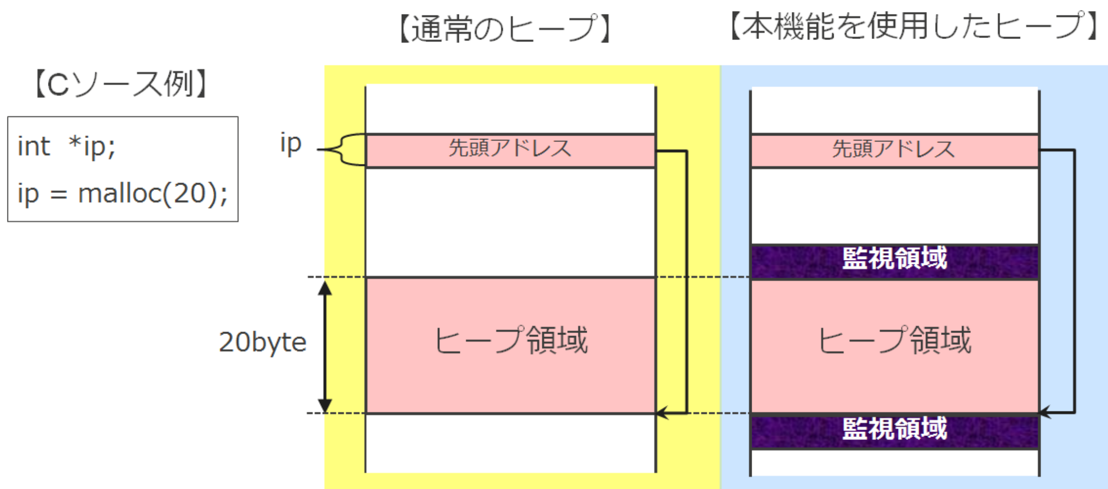


図 4-1 ヒープ領域のイメージ

ヒープ領域に対する操作後、free／realloc 関数によりヒープ領域を解放しますが、この際に監視領域に埋め込んだ値が変わっていないかをチェックします。値が変わっている場合は、ヒープ領域のオーバーフローが発生する不正なプログラムであると判断できますので、エラー処理に分岐します。

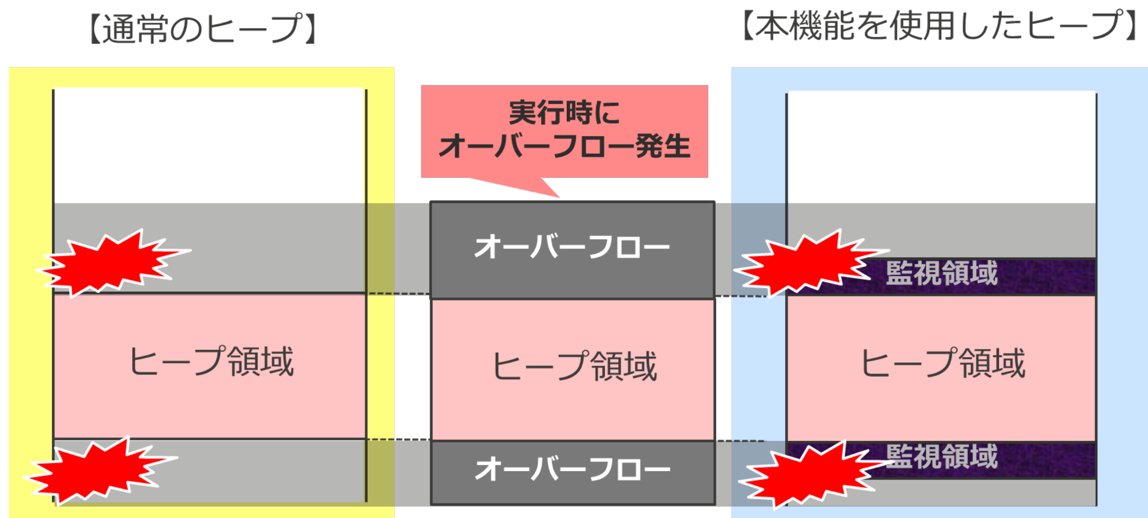


図 4-2 ヒープオーバーフローのイメージ

また、free／realloc 関数によりヒープ領域を解放する際に以下の操作を行った場合、同じくエラー処理に分岐します。

✓ calloc／malloc／realloc で確保した領域以外のポインタを free／realloc に渡す

✓ free で開放した後のポインタを再度 free／realloc に渡す

このように、ヒープ領域に対して不正な操作を行った場合にエラー関数に分岐しますので、ヒープ領域の不正操作が発生したことを動的に確認することが可能となり、プログラムの誤動作を早期に検出することができます。

4.2 生成コードのイメージ

本機能を使用していない場合は、ヒープ領域を確保してヒープ領域を操作し、操作後に free／calloc を使用してヒープ領域を解放します。

本機能の有効時は、確保したヒープ領域を解放する際、監視領域が書き換わっていないかをチェックします。書き換わっている場合はエラー関数"__heap_chk_fail"に分岐します。また、calloc／malloc／realloc で確保した領域以外のポインタや解放済みのポインタである場合にもエラー関数"__heap_chk_fail"に分岐します。

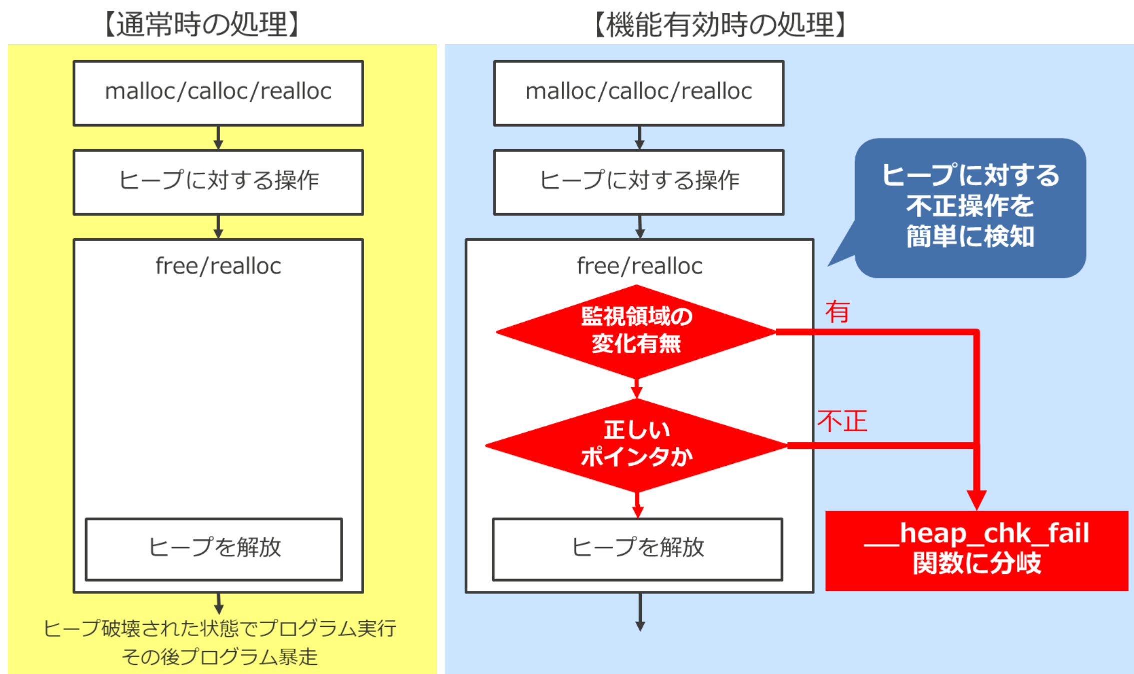


図 4-3 コンパイラの生成コード

本機能を使用していない場合、ヒープ領域に対する不正操作が発生すると、その後の処理でプログラムが暴走してしまう可能性があります。デバッグ時であれば、暴走の要因を調査することから始めなければなりません。

本機能を使用している場合、プログラムを実行時に"__heap_chk_fail"が呼び出されますので、不正なヒープ操作が発生しても暴走前にプログラムを停止させることができますし、デバッグ時であれば、"__heap_chk_fail"が呼び出された関数を簡単に特定し、その関数の処理を早期に見直すこともできます。

なお、"__heap_chk_fail"関数はユーザーが定義する必要があります。動的メモリ管理の異常時に実行する処理を記述してください。

4.3 使用方法

セーフティ機能を持った calloc／malloc／realloc 関数を含む標準ライブラリをリンクすることで使用します。本ライブラリのリンク方法はコンパイラごとに異なります。

表 4-1 セーフティ機能を持ったライブラリのリンク方法

CC-RL	リンカ・オプション”-library”で以下のライブラリをリンク CS+¥CC¥CC-RL¥Vx.xx.xx¥lib¥malloc_s.lib
CC-RX	ライブラリジェネレータ・オプション“-secure_malloc”を指定
CC-RH	リンカ・オプション”-library”で以下のライブラリをリンク CS+¥CC¥CC-RH¥Vx.xx.xx¥lib¥v850e3v5¥secure¥libmalloc.lib

統合開発環境 CS+、あるいは e^2 studio を使用している場合は、GUI 上の操作でオプション指定を制御することが可能です。

【CS+の場合】

CC-RL/CC-RH の場合は、[リンク・オプション]タブ -> [ライブラリ]カテゴリ -> [メモリの解放時にメモリ破壊を検出する]プロパティで ON/OFF を選択可能です。CC-RX の場合は、[ライブラリ・ジェネレート・オプション]タブ -> [オブジェクト]カテゴリ -> [メモリの解放時にメモリ破壊を検出する]プロパティで ON/OFF を選択可能です。

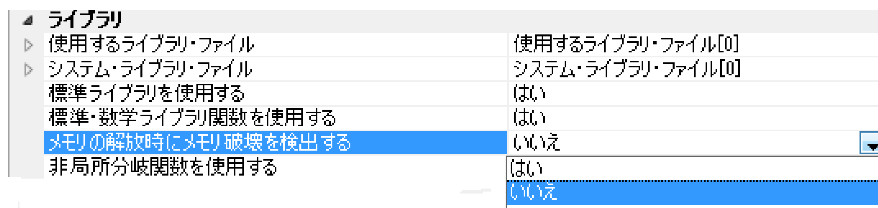


図 4-4 CS+の設定方法

【e^2 studio の場合】

CC-RL の場合は、[プロジェクト] -> [Renesas Tool Settings] でプロジェクトのプロパティダイアログを起動して[C/C++ ビルド] -> [Settings] を選択、[Tool Settings] タブ内で[Linker] -> [入力] -> [メモリの解放時にメモリ破壊を検出する] のチェックボックスで ON/OFF を選択可能です。CC-RX の場合は、[Tool Settings] タブ内で[Standard Library] -> [オブジェクト] -> [メモリの解放時にメモリ破壊を検出する] のチェックボックスで ON/OFF を選択可能です。

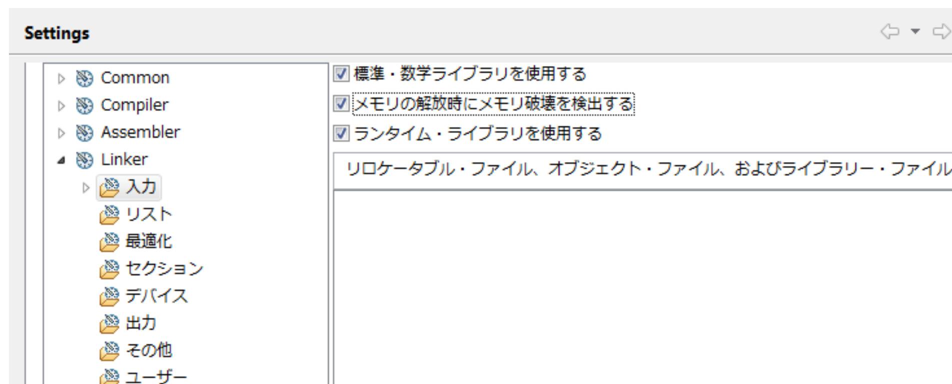


図 4-5 e^2 studio(CC-RL)の設定方法

4.4 Cソース例

本機能が有効となる例を C ソースで説明します。なお組込み関数の__halt();は CC-RX にはありません。

【例 1】コピー先の領域サイズがコピー元の領域サイズより小さい場合に、コピー先の領域サイズではなく
　　　　コピー元のサイズで memcpy を実行してバッファオーバーフローする場合

➢ func.c

```
1:  #include <stddef.h>
2:  #include <stdlib.h>
3:  #include <string.h>
4:
5:  typedef struct{
6:    char e_c[4];
7:    char line[28];
8:  } buf_t;
9:
10: void func(char *line){
11:   buf_t *bufa = NULL;
12:   bufa = (buf_t *)malloc(sizeof(buf_t));
13:
14:   memcpy(bufa, line, strlen(line));
15:
16:   free(bufa);
17:
18: }
19:
20: void __heap_chk_fail(void) {
21:   __halt();
22: }
```

➢ main.c

```
1:  extern void func(char *line);
2:
3:  char *line;
4:
5:  void main (void) {
6:    line = "ABCD1234567890qwertyuiopasdfghjklzxcvbnm";
7:    func(line);
8:  }
```

func.c の 14 行目はコピー先である bufa の領域サイズではなく、コピー元である line のサイズで memcpy を実行しています。このため、line のサイズが bufa の領域サイズより大きい場合、ヒープ領域の破壊が発生することになります。

一方、main.c の 7 行目で関数 func を呼び出す際、line のサイズ(40byte)が bufa の領域サイズ(32byte)より大きいため、ヒープ領域の破壊が発生するプログラムとなっています。

本機能を有効にしている場合は、bufa の領域を解放した際にエラー関数”__heap_chk_fail”をコールしますので、バッファオーバーフローを簡単に検出することができます。

【例 2】呼び出し先関数で誤って領域解放したことによりヒープ領域の二重解放をした場合

➢ func.c

```
1:  #include <stddef.h>
2:  #include <stdlib.h>
3:
4:  typedef struct{
5:  char e_c[4];
6:  char line[8];
7:  } struct_t;
8:
9:  extern int status;
10: void sub_func(struct_t *s);
11:
12: void func(int cond) {
13:   struct_t *strct = NULL;
14:   strct = (struct_t*)malloc(sizeof(struct_t));
15:
16:   if (cond == 0){
17:   } else {
18:     sub_func(strct);
19:   }
20:
21:   if(strct!=NULL){
22:     free(strct);
23:   }
24: }
25:
26: void sub_func(struct_t *s){
27:   if(status < 0){
28:     free(s);
29:   }
30: }
31:
32: void __heap_chk_fail(void) {
33:   __halt();
34: }
```

➢ main.c

```
1:  extern void func(int cond);
2:  int status;
3:
4:  void main (void) {
5:    status = -10;
6:    func(status);
7:  }
```

func.c の 27 行目の"status"が 0 未満である場合、14 行目で確保した領域を 28 行目で解放します。その後、22 行目でも同じ領域を解放しているため、解放済みのポインタを二重解放することになります。

一方、main.c の 5 行目で status に-10 を代入して func 関数に分岐しているため、二重解放が発生するプログ

ラムとなっています。

本機能を有効にしている場合は func.c の 22 行目を実行する際に”__heap_chk_fail”をコールしますので、二重解放を簡単に検出することができます。

第5章　　半精度浮動小数点

2 バイトの半精度浮動小数点型（half-precision floating-point type）をサポートすることにより、浮動小数点データが大量に存在するプログラムに対してデータサイズの削減が可能です。

なお、本機能は CC-RH に特化した機能です。

5.1 機能の概要

一般的な 4 バイト、8 バイトの浮動小数点型の他に、2 バイトの浮動小数点型をサポートしています。このデータ型を「半精度浮動小数点型」と呼び、__fp16 型として定義可能です。サイズ、整列条件とも 2 バイトで、データの内部表現は IEEE754-2008 の binary16 に準拠します。

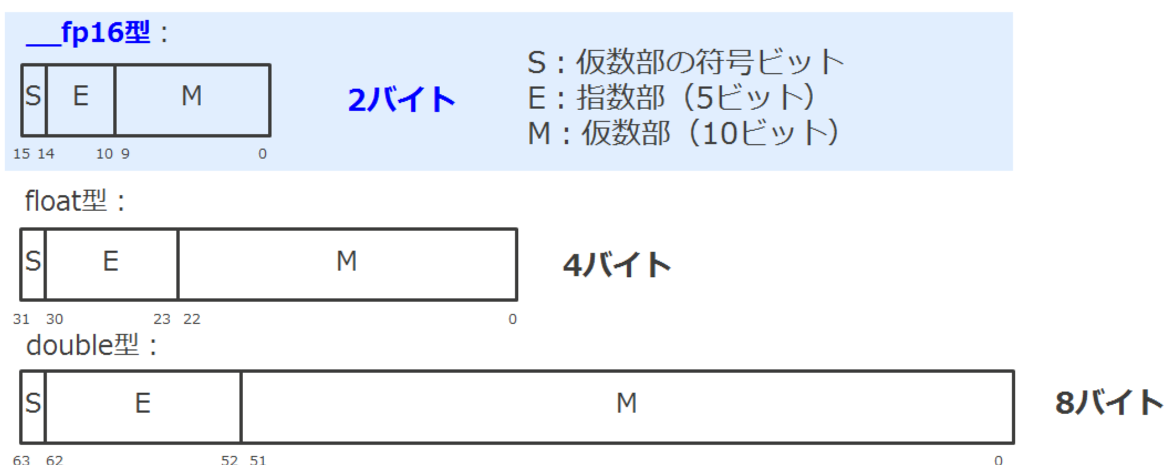


図 5-1 半精度浮動小数点 __fp16 型

以下の演算をコンパイラがサポートします。

- ✓ __fp16 型同士の代入
- ✓ __fp16 型から float 型への変換
- ✓ float 型から__fp16 型への変換

その他の演算は float 型へ変換してから行い、その結果は float 型に対する同演算と同じ型を持ちます。例えば、__fp16 型から double 型への変換は、一度 float 型に変換してから double 型に変換します。

5.2 生成コードのイメージ

float 型や double 型の浮動小数点数は、メモリから汎用レジスタにロードして演算等を行い、レジスタ値をメモリに格納します。

一方、半精度浮動小数点の場合は、メモリから汎用レジスタにロードし、FPU 命令の CVTF.HS（Floating-point Convert Half to Single）によって汎用レジスタ値を半精度から単精度に変換します。単精度浮動小数点として演算等を行い、FPU 命令の CVTF.SH（Floating-point Convert Single to Half）によって汎用レジスタ値を単精度から半精度に変換した後、メモリに格納します。

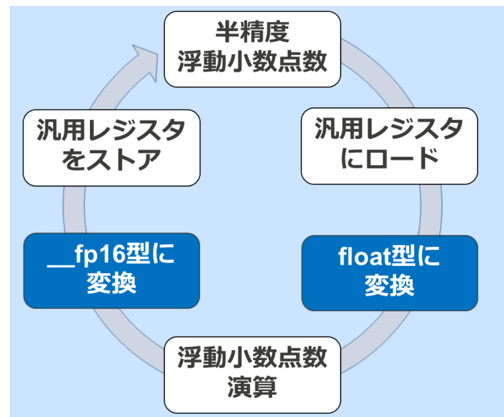


図 5-2 コンパイラの生成コード

5.3 指定方法

コンパイラ・オプション”-Xuse_fp16” を指定することで、半精度浮動小数点型を使用できるようになります。

統合開発環境 CS+を使用している場合は、GUI 上の操作でオプション指定を制御することが可能です。

【CS+の場合】

[コンパイル・オプション] タブ -> [出力コード] カテゴリ -> [半精度浮動小数点型を有効にする] プロパティで ON/OFF を選択可能です。

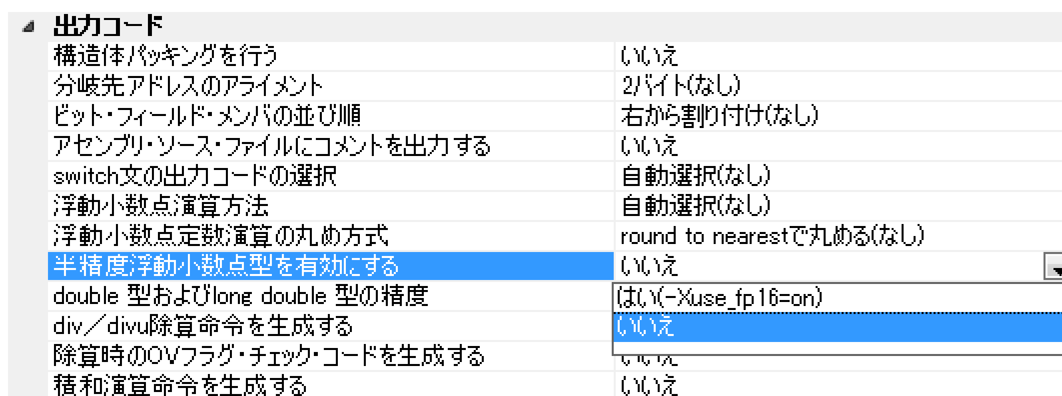


図 5-3 CS+の設定方法

5.4 Cソース例

【例】大量の浮動小数点型定数データを使用する場合

```
1:  const __fp16 coef[20] = {
2:     1.000000, 0.000000, 0.809017, 0.587785, 0.309017, 0.951057,
3:     -0.309017, 0.951057, -0.809017, 0.587785, -1.000000, 0.000000,
4:     -0.809017, -0.587785, -0.309017, -0.951057, 0.309017, -0.951057,
5:     0.809017, -0.587785
6:  };
7:
8:  void func(float *x, float *y, int r) {
9:     float xtmp = (*x);
10:    (*x) = coef[r] * (*x) - coef[r+1] * (*y);
11:    (*y) = coef[r] * (*y) + coef[r+1] * xtmp;
12: }
```

1～6 行目のように大量の浮動小数点型定数データを使用する場合に、半精度浮動小数点型を使用することで配列 coef のサイズを削減できます。単精度浮動小数点型で定義した場合は 80byte ですが、半精度浮動小数点型で定義した場合は 40byte となり、データサイズを半減できます。

第6章　制御レジスタ更新時の同期化機能

RH850 制御レジスタを連続して更新する場合のユーザー負荷を低減します。

なお、本機能は CC-RH に特化した機能です。

6.1 機能の概要

RH850 の制御レジスタは、ストア命令によって複数の制御レジスタを連続して更新する際、制御レジスタの更新順序がソース・ファイルの記述順と一致しない場合があります。そのため、更新する順序をソース・ファイルの記述順序と一致させるには同期化処理を手動で挿入する必要があります。同期化処理とは、先行命令の実行が完了するまで、次の命令を実行するのを待ち合わせる処理のことをいいます。

ただし、順序を保証するために必ず同期化処理が必要という訳ではありません。同じペリフェラルグループの制御レジスタを連続して更新する場合には、順序が保証されているため同期化処理は不要です。

そのため、ソース・ファイル上の全ての制御レジスタ更新処理に対して、その制御レジスタがどのペリフェラルグループに所属するのかハードウェアマニュアルを参照しながら目視で判断し、同期化処理を手動で挿入しなければなりません。

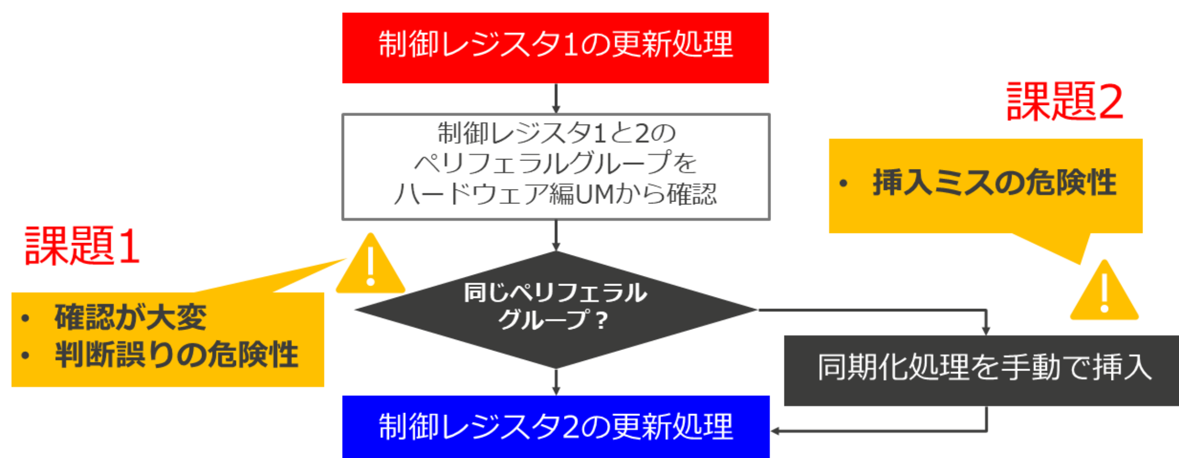


図 6-1 制御レジスタ更新処理に対する課題

CC-RH では以下のような機能により図 6-1 のそれぞれの改題を解決します。

これらを「制御レジスタ更新時の同期化機能」と呼びます。

- **同期化処理の挿入機能**
  ⇒ 課題 1 と課題 2 を同時に解決します。

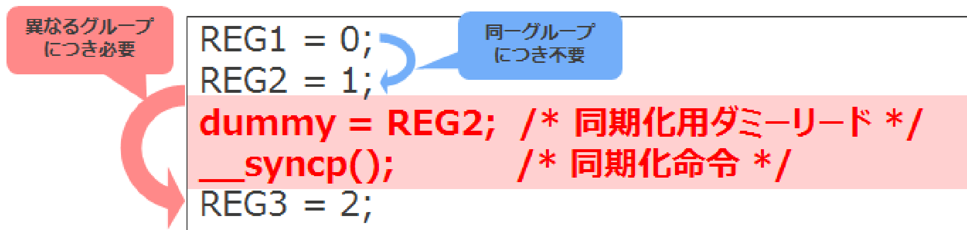
- **制御レジスタへの書き込み検出機能**
  ⇒ 課題 1 を解決します。

6.2 生成コードのイメージ

➢ **同期化処理の挿入機能**

制御レジスタへの書き込みに対して同期化処理をアセンブラソース・ファイルに対して自動で挿入します。このとき、同じペリフェラルグループ内の制御レジスタへの連続書き込みの場合は同期化処理を省略します。これにより、**ユーザーが同期化処理の必要有無を判断し、同期化処理を手動で挿入する作業が不要となります。**

【例】制御レジスタ REG1(CPU グループ) → REG2(CPU グループ) → REG3(0 グループ)の順番で連続して更新する場合、赤字の同期化処理をコンパイラが出力します。



➢ **制御レジスタへの書き込み検出機能**

制御レジスタへ書き込みしているソース・ファイル名と行番号、その制御レジスタが所属するペリフェラルグループ名とアドレスをインフォメーションメッセージで出力します。

```
src.c(9):M0536001:制御レジスタを更新します。(id=グループID, 制御レジスタのアドレス)
src.c(10):M0536001:制御レジスタを更新します。(id=グループID, 制御レジスタのアドレス)
src.c(11):M0536001 :制御レジスタを更新します。(id=グループID, 制御レジスタのアドレス)
```

同期化処理の挿入機能は、制御レジスタの更新順序がソース・ファイルの記述順と一致しなくても構わないケースでも同期化処理を挿入します。そのため、ユーザーの判断によって必要な個所のみに手動で同期化処理を挿入したい場合には制御レジスタへの書き込み検出機能を使用してください。**制御レジスタが所属するペリフェラルグループ名をハードウェアマニュアルから参照する負荷を省略できます。**

6.3 指定方法

a.と b.の 2 段階の手順によって制御レジスタ更新時の同期化機能を使用できるようになります。

a. 拡張言語でペリフェラルグループの範囲を指定

以下の拡張言語により、各ペリフェラルグループの先頭アドレス、終端アドレスを指定します。

【指定形式】

```
#pragma register_group 先頭アドレス, 終端アドレス, id="グループID"
```

グループ ID は、制御レジスタが所属するペリフェラルグループを指定するための識別子です。ハードウェア

マニュアルのレジスター覧等に記載されたペリフェラルグループとアドレス情報を参照し、各ペリフェラルグループに対してアドレス範囲を指定してください。同一グループでも連続していない場合もありますので、複数の#pragma register_group に同じ*グループ ID* を指定することができます。ハードウェアマニュアルに記載されているペリフェラルグループ名と一致している必要はありません。

なお、G4MH コアを搭載したマイコンの場合、この拡張言語を自動で生成する機能があります。詳細につきましては、「6.5 補足事項」をご参照ください。

b. コンパイラ・オプションを指定

コンパイラ・オプション" -store_reg" を指定することで、制御レジスタ更新時の同期化機能を使用できます。

【指定形式】

```
-store_reg=item
```

オプション	説明
-store_reg=*sync*	同期化処理の挿入機能を有効とします。 #pragma register_group で指定した範囲内の制御レジスタへの書き込みを検出し、書き込みの後に同期化処理を挿入します。同一グループへの書き込みが後に続くと判断できる場合は挿入しません。
-store_reg=*list*	制御レジスタへの書き込み検出機能を有効とします。 #pragma register_group で指定した範囲内の制御レジスタへの書き込みを検出し、ソース・ファイル上の記述位置を標準エラー出力に表示します。同一グループへの書き込みが後に続くと判断できる場合は表示しません。
-store_reg=*list_all*	制御レジスタへの書き込み検出機能を有効とします。 #pragma register_group で指定した範囲内の制御レジスタへの書き込みを検出し、ソース・ファイル上の記述位置を標準エラー出力に表示します。同一グループへの書き込みが後に続くかどうかにかかわらず全てを表示します。
-store_reg=*ignore*	#pragma register_group を無視します。

統合開発環境 CS+を使用している場合は、GUI 上の操作でオプション指定を制御することが可能です。

[コンパイル・オプション] タブ -> [出力コード] カテゴリ -> [制御レジスタ書き込みに対する処置モード] プロパティで選択可能です。

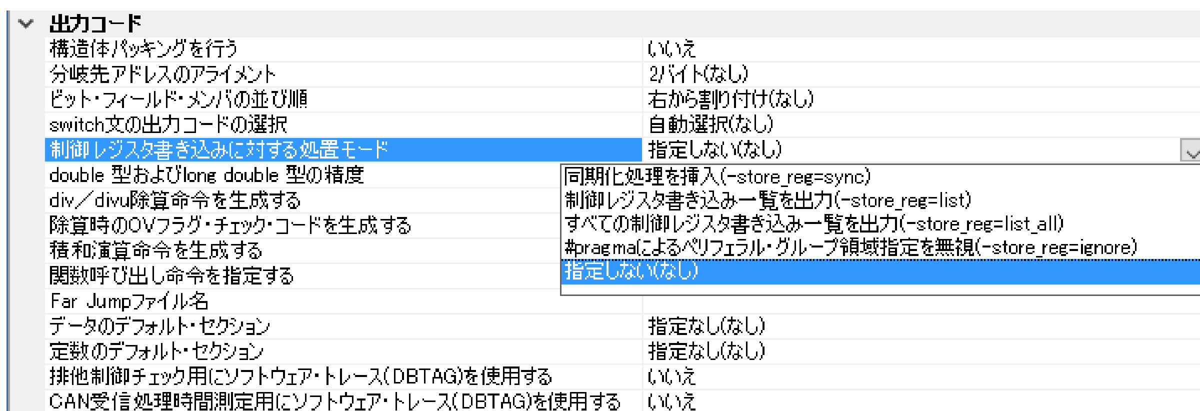


図 6-2 CS+の設定方法

6.4 Cソース例

【例】#pragma register_group で CPU グループと 0 グループの 2 つのペリフェラルグループを定義した場合

【pgroup.h】

```
#pragma register_group 0xfedf0000, 0xfedfffff, id="CPU"
#pragma register_group 0xfee00000, 0xfee0ffff, id="0"
```

【iodefine.h】

```
#define REG1 (*(volatile unsigned char*)0xfedf0000)    /* CPU グループの制御レジスタ */
#define REG2 (*(volatile unsigned char*)0xfedf0001)    /* CPU グループの制御レジスタ */
#define REG3 (*(volatile unsigned short*)0xfee00000)   /* 0 グループの制御レジスタ */
```

【src.c】

```
1:  #include “pgroup.h”
2:  #include “iodefine.h”
3:
4:  void func(void) {
5:      REG1 = 0;   // CPU グループの制御レジスタ更新
6:      REG2 = 1;   // CPU グループの制御レジスタ更新
7:      REG3 = 2;   // 0 グループの制御レジスタの更新
8:  }
```

5～7 行目で REG1 ⇒ REG2 ⇒ REG3 の順番で制御レジスタに値を書き込んでいます。

本機能を使用しない場合、この順番で制御レジスタを更新する必要がある場合は同期化処理の挿入要否を検討するため、REG1～REG3 が所属するペリフェラルグループについてハードウェアマニュアルを参照して特定しなければなりません。

その結果、5～6 行目は同一グループ、6～7 行目は異なるグループへの書き込みとなることから、REG2 ⇒ REG3 の間に同期化処理を挿入しなければなりません。また、関数 func の処理が終了した後、異なるグループに属する制御レジスタに値を書き込む可能性がある場合は、REG3 の更新後にも同期化処理を挿入しなければなりません。

➢ -store_reg=sync オプションを指定時

コンパイル時に以下のようなアセンブリ命令を生成します。

1:	_func:
2:	.stack _func = 0
3:	movhi 0x0000FEDF, r0, r2
4:	st.b r0, 0x00000000[r2]
5:	movhi 0x0000FEDF, r0, r2
6:	mov 0x00000001, r5
7:	st.b r5, 0x00000001[r2]
8:	**ld.bu 0x00000001[r2], r10**
9:	**syncp ;**
10:	movhi 0x0000FEE0, r0, r2
11:	mov 0x00000002, r5
12:	st.h r5, 0x00000000[r2]
13:	**ld.hu 0x00000002[r2], r10**
14:	**syncp ;**
15:	jmp [r31]

後続が同じグループへの書き込みのため同期化処理を挿入しない

REG1（CPU グループ）の更新

REG2（CPU グループ）の更新

後続が異なるグループへの書き込みのため同期化処理を挿入する

REG3（0 グループ）の更新

関数 func()からの復帰後の処理が不明のため同期化処理を挿入する

➢ **-store_reg=list オプションを指定時**

以下のようなユーザーが同期化処理の挿入要否を容易に検討できるメッセージを標準エラーに出力します。同一グループへの書き込みが後に続くと判断できる場合は表示しません。

5 行目は非表示

```
src.c(6):M0536001: 制御レジスタを更新します。(id=CPU, 0xfedf0001)
src.c(7):M0536001: 制御レジスタを更新します。(id=0, 0xfee00000)
```

➢ **-store_reg=list_all オプションを指定時**

以下のようなユーザーが同期化処理の挿入要否を容易に検討できるメッセージを標準エラーに出力します。同一グループへの書き込みが後に続くかどうかにかかわらず全てを表示します。

5 行目も表示

```
src.c(5):M0536001: 制御レジスタを更新します。(id=CPU, 0xfedf0000)
src.c(6):M0536001: 制御レジスタを更新します。(id=CPU, 0xfedf0001)
src.c(7):M0536001: 制御レジスタを更新します。(id=0, 0xfee00000)
```

6.5 補足事項

G4MH コアを搭載したマイコンの場合、CS+のプロジェクトで指定したマイコン向けに”iodefine_pgroup.h”を自動生成することが可能です。このファイルは当該マイコンのペリフェラルグループの範囲を CC-RH の拡張言語#pragma register_group で指定したヘッダ・ファイルです。

CS+から生成する方法は下記の通りです。

- CC-RH のプロパティの[I/O ヘッダ・ファイル生成オプション]タブ ⇒ [ビルド時に I/O ヘッダ・ファイルを更新する]で「はい(プロパティのみ)」を選択
- [ペリフェラル・グループ用 pragma 指令を出力する]で「はい(-pragma_peripheral_group=on)」を選択

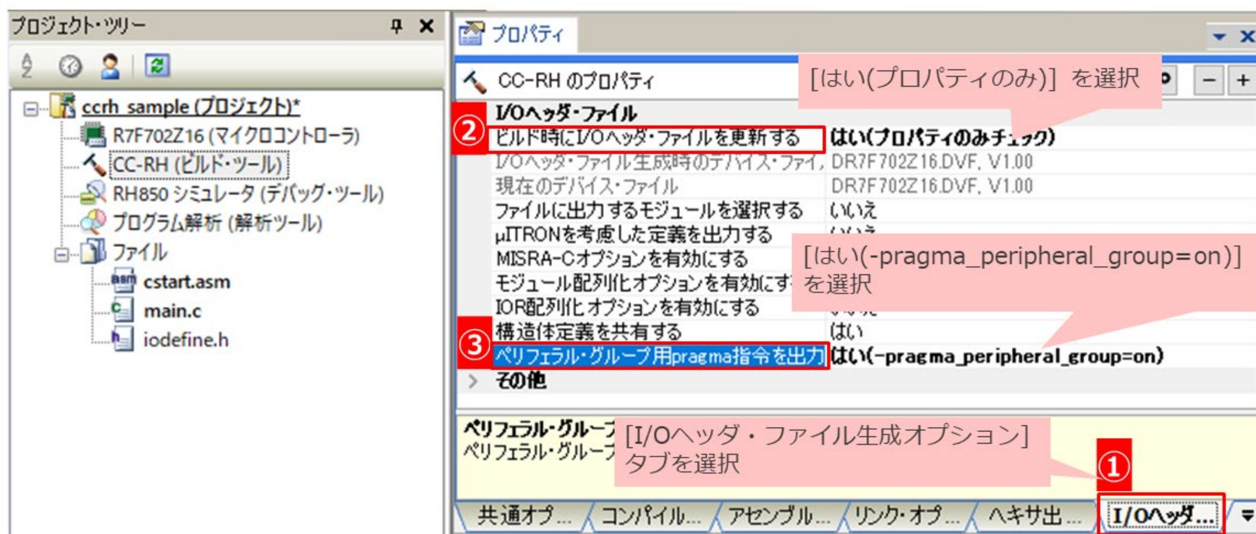


図 6-3 CS+のプロパティ設定

- [ファイル]メニュー ⇒ [プロジェクトを保存]を選択して一旦プロジェクトを保存
- プロジェクト・フォルダ直下に”iodefine_pgroup.h”が生成

このヘッダ・ファイルを C ソース・ファイルからインクルードすることで、当該 C ソース・ファイルに記述した制御レジスタ更新処理は同期化機能の対象となります。

第7章　　不正な間接関数呼び出し検出機能

信頼できないアドレスに対する間接関数呼び出しを未然に防止することで、お客様のプログラム品質を向上します。

7.1 機能の概要

間接関数呼び出しとは、関数のアドレスを実行時に取得して関数を呼び出す方法です。例えば、関数ポインタの隣に外部からの入力を蓄えるバッファがあり、このバッファにデータを書き込む場合を考えます。この書き込む処理にバッファ領域外を書き換えてしまう脆弱性があると、外部からの入力によって関数ポインタを書き換えることができてしまいます。書き換えられた関数ポインタを用いて間接関数呼び出しを行うと、ソフトウェアが暴走したり、最悪の場合はシステムを乗っ取られる可能性があります。これを未然に防止する機能が不正な間接関数呼び出し検出機能です。

下記の処理を全てコンパイラが自動で行います。

1. 間接呼び出しされる可能性のある関数を抽出して関数リストに登録
2. 間接関数呼び出しの直前で呼び出し先アドレスをチェックするコードを生成

7.2 生成コードのイメージ

間接関数呼び出しされる可能性のある関数をコンパイラがコンパイル時に C ソース・プログラムから自動で抽出します。リンク時にその情報を統合し、実行形式ファイル内に関数リスト（間接関数呼び出しされる可能性のある関数アドレスのリスト）を生成します。

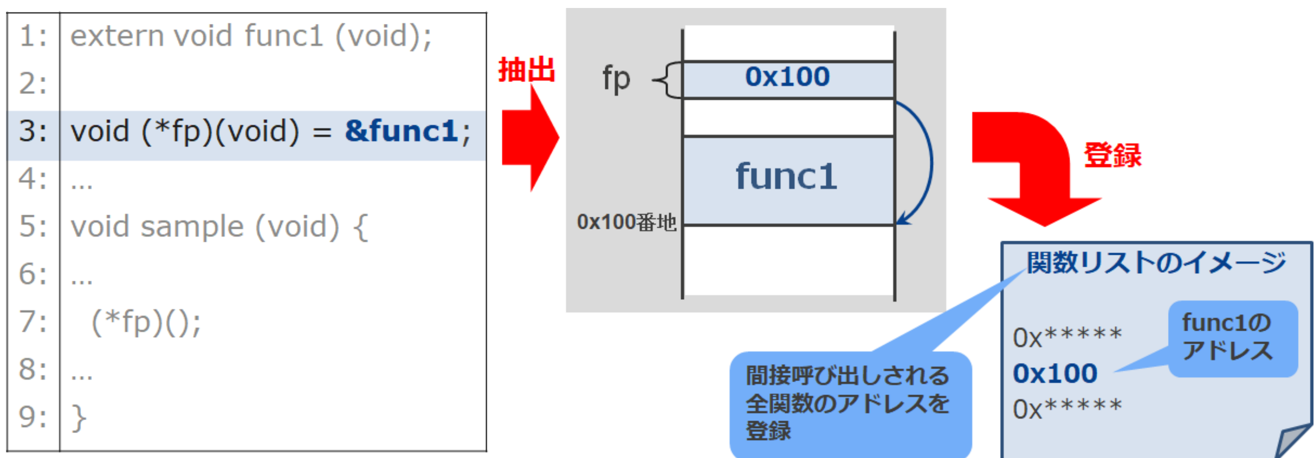


図 7-1　関数リストへの登録イメージ

また、間接関数呼び出しが行われる直前に、チェック関数__control_flow_integrity を呼び出します。

このチェック関数には、当該間接関数呼び出しによる分岐先アドレスが引数として渡されます。実行時に、チェック関数内で引数の分岐先アドレスが関数リストの中に含まれるかをチェックし、関数リストに含まれている場合は関数呼び出しの処理を正常に実行します。

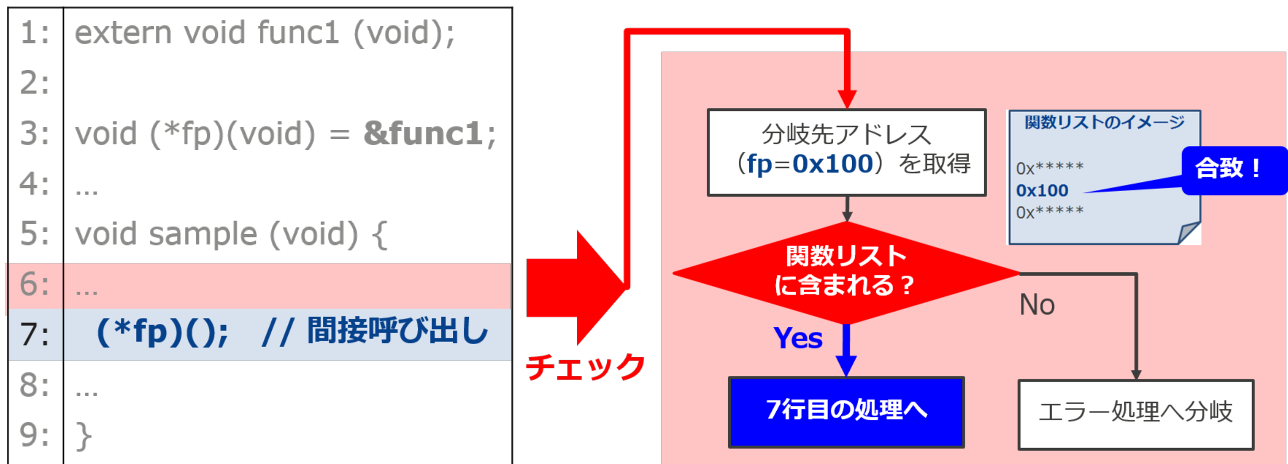


図 7-2 関数リストに含まれる関数が呼び出された場合

呼び出される関数が関数リストに含まれていない場合は、不正な間接関数呼び出しとみなして__control_flow_chk_fail 関数を呼び出してエラー処理に分岐します。

これにより、関数リストに登録されていない関数の間接呼び出しを防止し、結果としてプログラムの暴走や悪意のあるプログラム領域の書き換えを防止することができます。

なお、チェック関数__control_flow_integrity は標準ライブラリ内に含まれます。

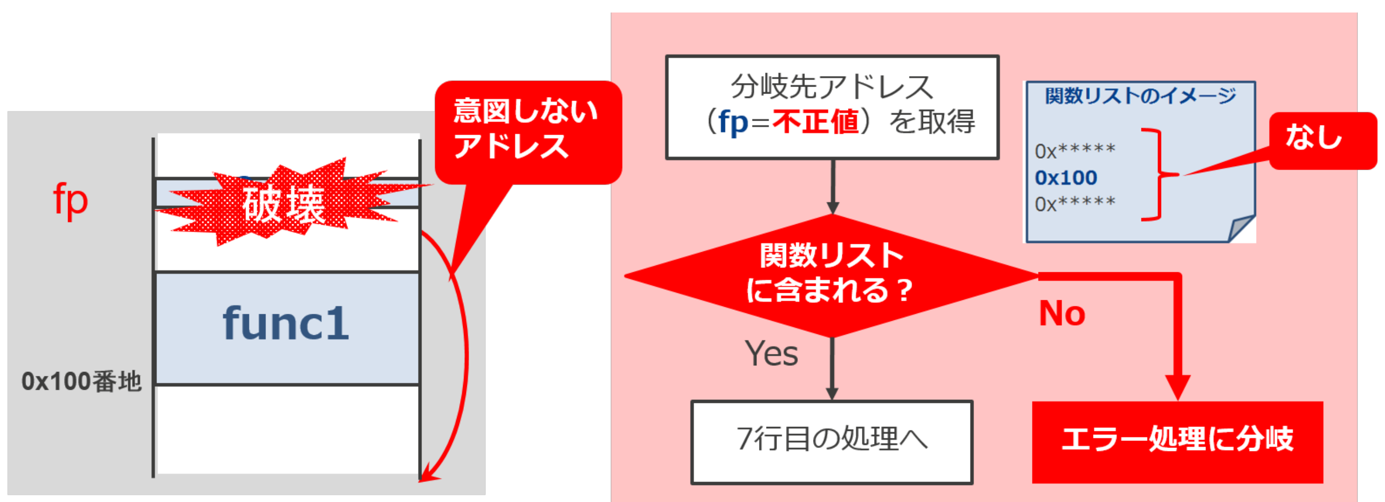


図 7-3 関数リストに含まれない関数が呼び出された場合

7.3 使用方法

本機能を有効にするには、下記のオプションを指定してください。

【コンパイル・オプション】

不正な間接関数呼び出しを検出するコードを生成します。

```
-control_flow_integrity
```

【リンク・オプション】

不正な間接関数呼び出し検出時に用いる関数リストを生成します。

```
-cfi
```

統合開発環境 CS+、あるいは e² studio を使用している場合は、GUI 上の操作でオプション指定を制御することが可能です。

【CS+の場合】

[コンパイル・オプション]タブ-> [品質向上関連]カテゴリ-> [不正な間接関数呼び出しを検出する]プロパティで ON/OFF を選択可能です。

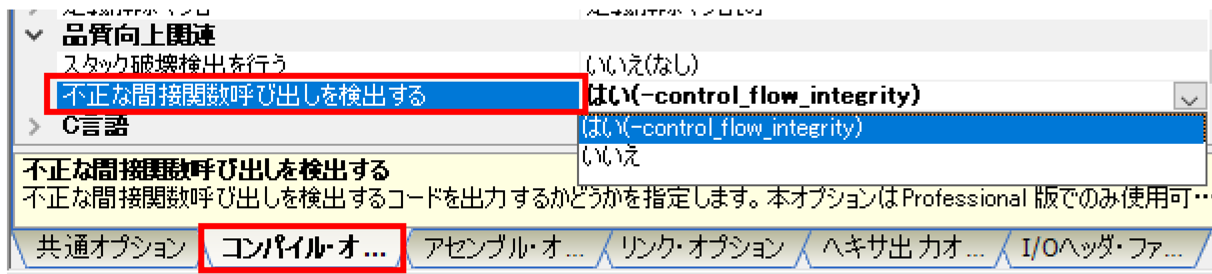


図　7-4　CS+の設定方法

リンク・オプション-cfi が OFF の場合にコンパイル・オプション-control_flow_integrity を有効にしようとすると以下の警告(W0293007)が表示されます。

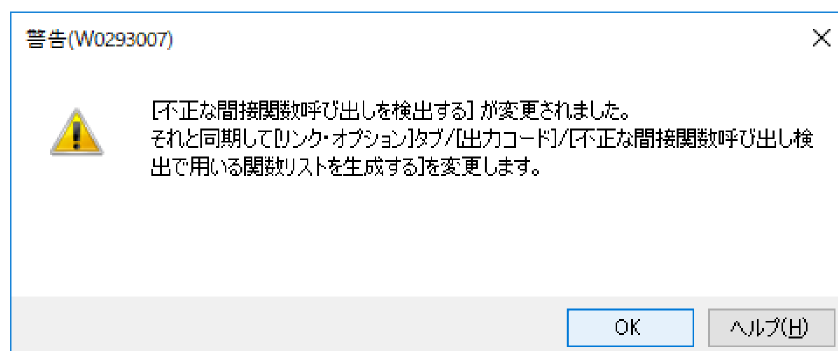


図 7-5 設定時の警告メッセージ

警告画面で[OK]を選択すると、[リンク・オプション]タブ-> [出力コード]カテゴリ-> [不正な間接関数呼び出し検出で用いる関数リストを生成する]プロパティが「はい」に変更されます。

これにより、不正な間接関数呼び出し検出時に用いる関数リストを生成します。

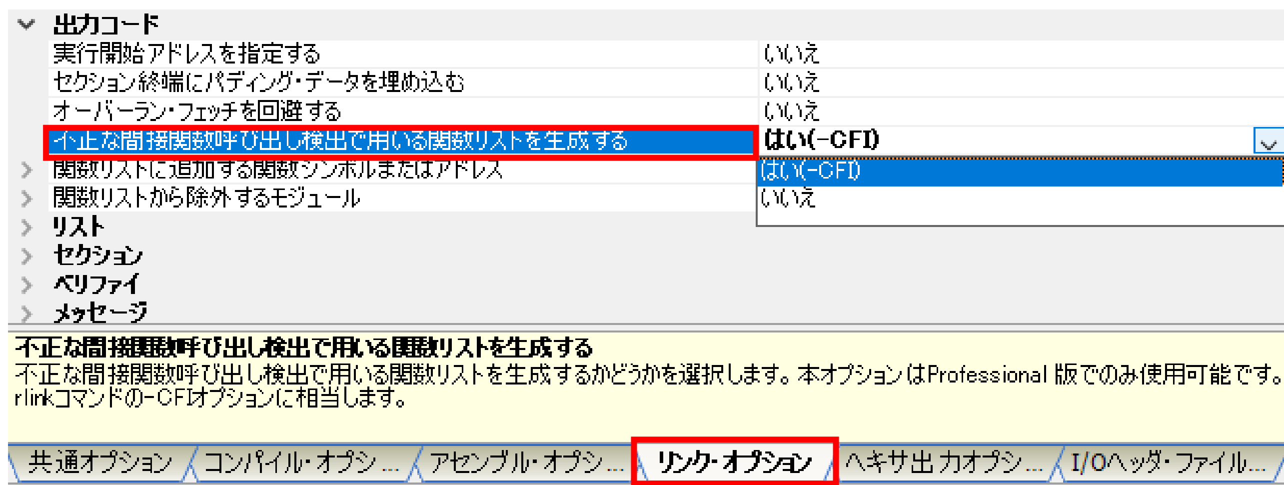


図 7-6 [リンク・オプション]タブ

【e² studio の場合】

メニューバーの[プロジェクト]から[プロパティ]を選択し、プロパティダイアログを起動します。[C/C++ ビルド] -> [設定] を選択、[ツール設定] タブ内で[Compiler] -> [その他] -> [不正な間接関数呼び出し検出コードを生成する] で ON/OFF を選択することができます。

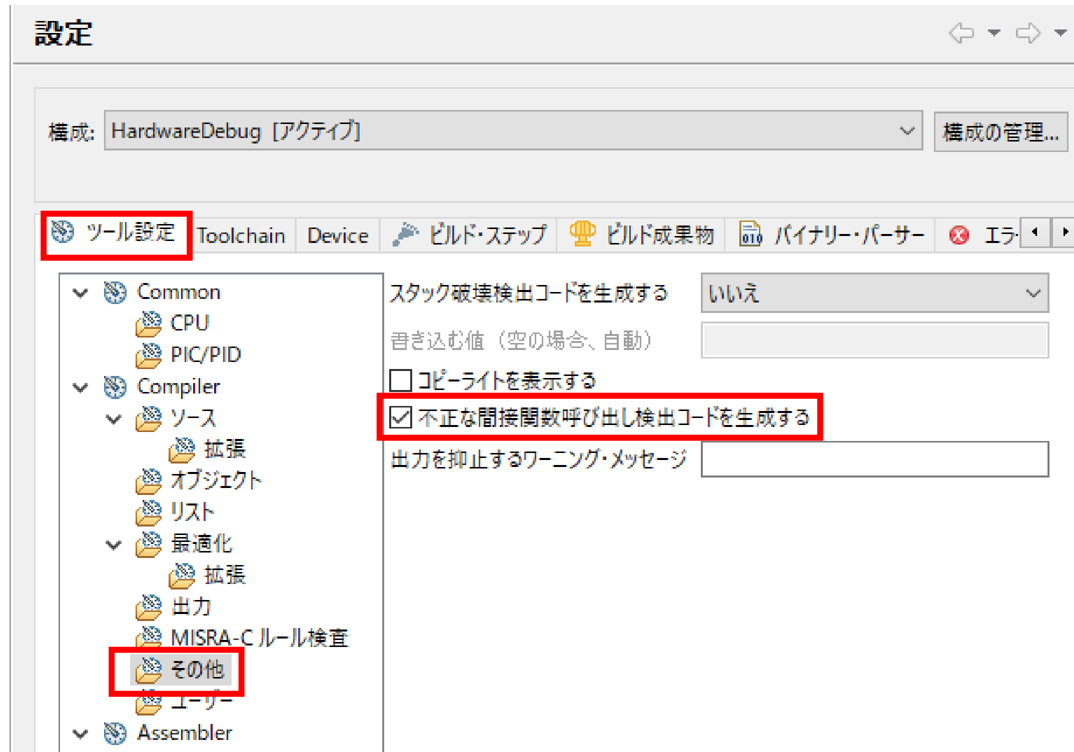


図 7-7 e² studio の設定方法

7.4 C ソース例

本機能が有効となる例を C ソースで説明します。

```
1:  #include <string.h>
2:  #define MAX 100
3:
4:  void __control_flow_chk_fail(void) //エラー関数の定義
5:  {
6:    __halt();
7:  }
8:
9:  void func2(void);
10: void func3(char* buf);
11:
12: char lbuf[MAX];
13: void func(int a, int b, int c, int d, void (*pf)(void)) {
14:   char buf[] = "buf";
15:   func3(buf);
16:   pf(); // 間接関数呼び出し
17: }
18:
19: void func2(void) {
20:   return;
21: }
22:
23: void func3(char* buf) {
24:   int i;
25:   for (i=0; i!=MAX; ++i) {
26:     buf[i] = 'a';
27:   }
28: }
29:
30: void main(void) {
31:   func(1,2,3,4, &func2); // スタック渡し
32: }
```

func3()でバッファオーバーフローを発生させ、図 7-8 のように、スタックフレーム上の引数 pf の値を書き換えています。その結果、16 行目で意図しない不正なアドレスに分岐してしまうことになります。

本機能を有効にすると、16 行目の直前で pf の値を引数としてチェック関数__control_flow_integrity を呼び出します。-cfi オプションを指定して生成する関数リストには pf と合致する値が登録されていないため、4 行目のエラー関数__control_flow_chk_fail が呼ばれます。__control_flow_chk_fail 関数はユーザーが定義し、不正な間接関数呼び出し検出時に実行する処理を記述してください。

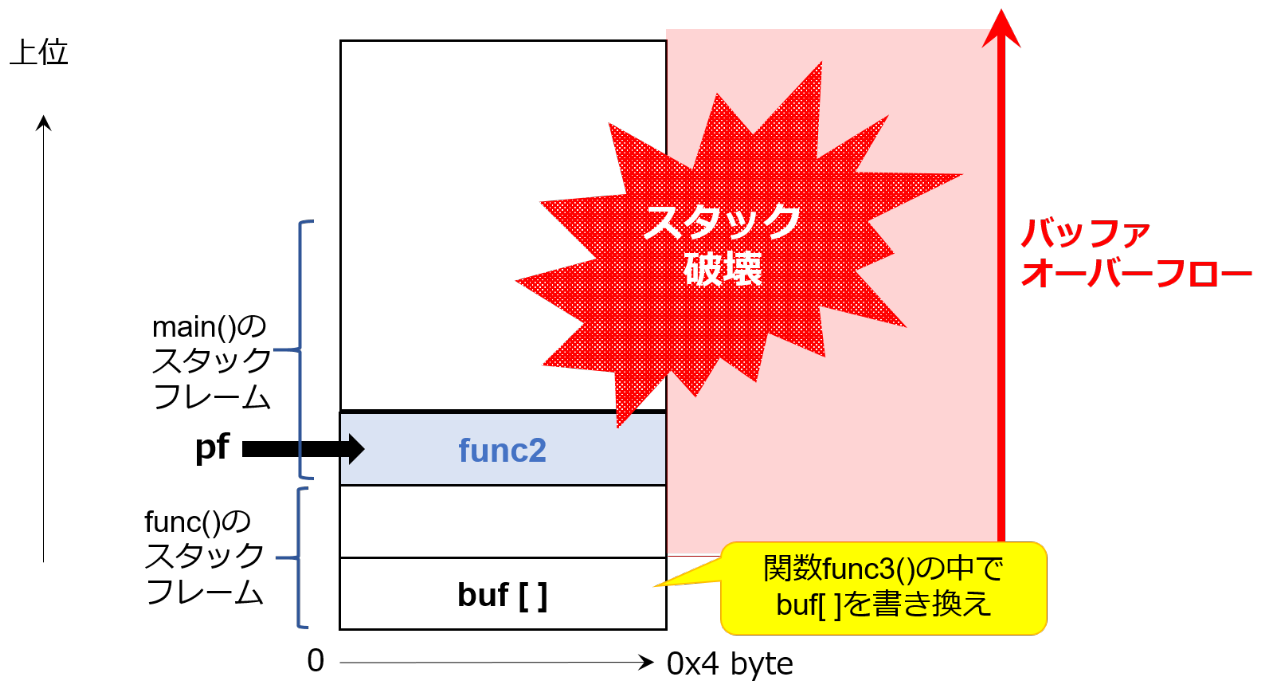


図 7-8 スタックイメージ

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。

第8章　改版履歴

Rev.	発行日	改定内容	
		ページ	ポイント
1.00	2017.4.18	-	初版発行
1.01	2017.8.9	3	「表 1-1」に制御レジスタ更新時の同期化機能を追加
		5	「表 2-2」を最新リビジョンのルール数に変更
		13-14	「3.4 C ソース例」の C ソース例を拡充してエラー処理”__stack_chk_fail()”を追記
		18-19	「4.4 C ソース例」の C ソース例を拡充してエラー処理”__heap_chk_fail()”を追記
		24-28	「第 6 章　制御レジスタ更新時の同期化機能」を追加
1.02	2018.4.3	3	「表 1-1」に不正な間接関数呼び出し検出機能を追加
		5	「表 2-2」を最新リビジョンのルール数に変更
		29-34	「第 7 章　不正な間接関数呼び出し検出機能」を追加
1.03	2019.3.1	5	「表 2-1」「表 2-2」を最新リビジョンに変更
		24	「図 6-1」を追加
		29	「6.5 補足事項」を追加
1.04	2020.3.12	4	「1.1」の「アップグレード（エディション）ライセンス」の名称と説明を変更
		5	「表 2-1」「表 2-2」を最新リビジョンに変更
1.05	2021.3.15	4	「図 1-1」「図 1-2」を追加
		5	「表 2-1」「表 2-2」を最新リビジョンに変更
		6	「表 2-3」に-misra_intermodule オプションの追加 「図 2-1」「図 2-2」を更新
		11	「図 3-3」に__stack_chk_fail 関数を追加
		16	「図 4-3」の処理フローを改訂し、__heap_chk_fail 関数を追加
		22	「図 5-2」を更新

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。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、お客様の責任において、お客様の機器・システムを設計ください。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. 当社製品または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. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 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。
6. 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。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 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。
9. 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。
10. 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。
11. 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。
12. 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。
13. 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。
14. 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.5.0-1　2020.10)

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。

© 2021 Renesas Electronics Corporation. All rights reserved.