# Renesas Compilers

Professional Editions

R20UT4026EJ0105
Rev.1.05
Mar.15.21

## Contents

# 1. Introduction

This application note is for those customers who are considering the professional edition of a Renesas compiler. It gives an overview, covers usage, and has useful examples of C source code that are specific to the professional edition.

## 1.1 Types of Licenses to Renesas Compilers

Renesas provides standard and professional editions of the licenses for its CC-RL, CC-RX, and CC-RH compilers.

- Standard edition
  The C-language specifications that comply with the ANSI standard are supported.
  The standard edition also provides powerful optimization functions and the basic functions that are required for writing embedded program code.

- Professional edition
  In addition to the features of the standard editions, this edition provides additional features which help to improve the quality of the customer's programs and shorten development periods.
  The features will be continuously expanded in the future.

**Table 1-1 Features of the Professional Editions**

| Features of the professional editions | CC-RL | CC-RX | CC-RH |
|---|---|---|---|
| Checking of source code against MISRA-C: 2004/2012 rules | √ | √ | √ |
| Detection of stack smashing | √ | √ | √ |
| Enhanced security for dynamic memory management functions | √ | √ | √ |
| Half-precision floating-point | — | — | √ |
| Synchronization features in the updating of control registers | — | — | √ |
| Detection of illicit indirect function calls | √ | √ | √ |

√: Supported. —: Support is not planned.

You can use the features of the professional edition by purchasing the compiler license for the professional edition or using either of the following.

- Upgrade (edition) license

  If you have a license for the standard edition, this license especially for upgrading from the standard edition to the professional edition is available for purchase. Note that this license only works with a node-locked license (permanent); it does not work with floating or annual licenses.

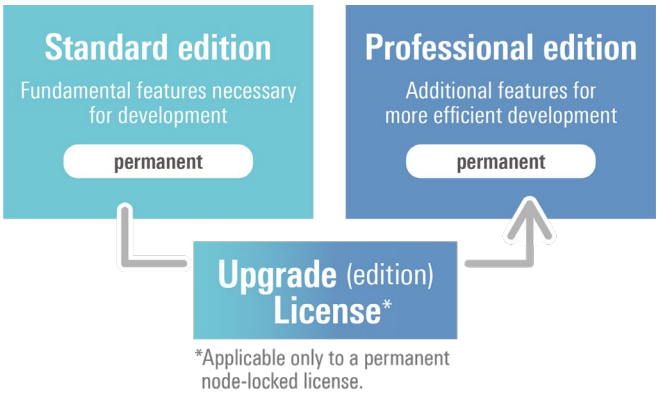



**Figure 1-1 Upgrade (edition) License**

- Annual license

  This license is valid for one year. You can use the features of the professional edition for one year with an annual license for the professional edition. The annual license can provide a thorough introduction to the professional edition at a low price relative to a permanent license.

  Annual licenses are useful in terms of flexibility in response to varying numbers of users in your team over time.

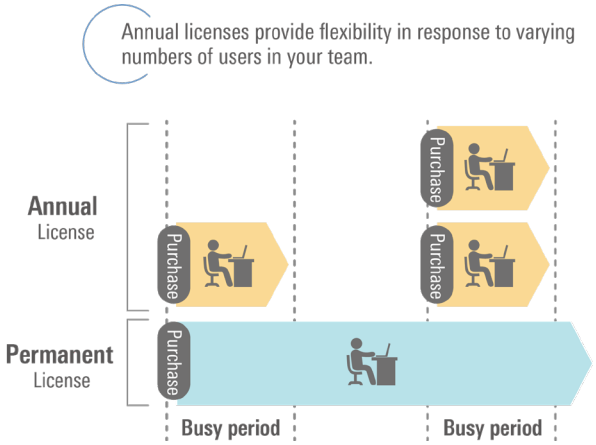



**Figure 1-2 Annual License**

## 1.2 Evaluating the Features of the Professional Edition

When you want to evaluate or confirm the features of the professional edition, use the evaluation version.

After you have installed the evaluation version for the first time, you can try its features for 60 days from the date of the first building. After the 61st day, the features become limited to those of the standard edition and the linkage size (the size of programs that can be generated) also becomes limited.

Make use of the examples of the C source code in this application note, since they demonstrate features of the professional edition.

# 2. Checking of Source Code against MISRA-C:2004/2012 Rules

When a compiler is started, it can check code against MISRA-C rules and output messages if the source code deviates from those rules. This feature can improve quality of the user program.

## 2.1 MISRA-C:2004/2012 Rules

MISRA-C is a set of software development guidelines for the C language developed by the Motor Industry Software Reliability Association (MISRA). The purpose is to maintain the safety, portability, and reliability of embedded systems programmed in the C language. MISRA-C:2004 and MISRA-C:2012 are the rules as standardized in 2004 and 2012, respectively.

## 2.2 Number of Rules to be Checked

**Table 2-1 Number of Rules in MISRA-C:2004**

| Classification of Rules | CC-RL V1.10.00 | CC-RX V3.03.00 | CC-RH V2.03.00 |
|---|---|---|---|
| Required rules (121) | 79 | 79 | 79 |
| Advisory rules (20) | 13 | 13 | 13 |
| Total number of rules (141) | **92** | **92** | **92** |

**Table 2-2 Number of Rules in MISRA-C:2012**

| Classification of Rules | CC-RL V1.10.00 | CC-RX V3.03.00 | CC-RH V2.03.00 |
|---|---|---|---|
| Mandatory rules (16) | 7 | 7 | 7 |
| Required rules (108) | 90 | 90 | 90 |
| Advisory rules (32) | 27 | 27 | 27 |
| Total number of rules (156) | **124** | **124** | **124** |

The numbers of supported rules depend on the revisions of the compilers.

## 2.3 Specifying Rules

You can easily start the rule checkers for MISRA-C:2004 and 2012 by specifying compiler options.

Parameters to control the rule numbers that are to be checked or ignored can also be specified for the options.

**Table 2-3 Options of Rule Checker for MISRA-C:2004 and 2012**

| | Option | | |
|---|---|---|---|
| **Description** | **CC-RL** | **CC-RX** | **CC-RH** |
| This option checks source code against the MISRA-C:2004 rules. | -misra2004 | -misra2004 | -Xmisra2004 |
| This option checks source code against the MISRA-C:2012 rules. | -misra2012 | -misra2012 | -Xmisra2012 |
| This option specifies files that will not be checked against the MISRA-C:2004 or MISRA-C:2012 rules. | -ignore_files_misra | -ignore_files_misra | -Xignore_files_misra |
| This option enables the source-code checking of the MISRA-C:2004 or MISRA-C:2012 rules, which are partially suppressed by the extended language specifications. | -check_language_extension | -check_language_extension | -Xcheck_language_extension |
| This option checks source code in multiple files against the MISRA-C:2012 rules (Note1). | -misra_intermodule | -misra_intermodule | -misra_intermodule |

Note1. This option is usable CC-RL V1.08.00, CC-RX V3.01.00, CC-RH V2.01.00 or later version.

When you are using CS+ or the e$^2$ studio as the integrated development environment, you can control the specification of options by operations in the GUI.

- For CS+
  Select rules from 2004 or 2012 by selecting the [Compile Options] tabbed page -> [MISRA-C Rule Check] category -> [MISRA-C specification] property. Detailed settings are enabled for [Apply rule], [Rule check exclusion file], [Output message of the enhanced key word and extended specifications] and [Enable checking that spans files]properties.

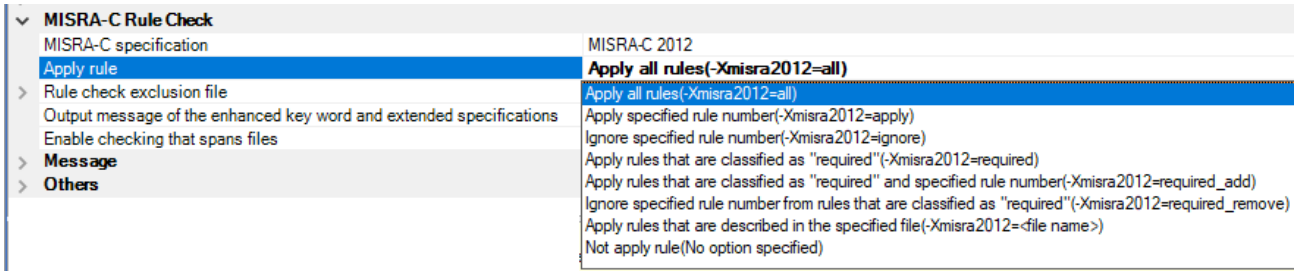



**Figure 2-1 Specifying Options in CS+**

- For the e$^2$ studio
  Activate the property dialog box of the project from [Project] -> [Renesas Tool Settings] and select [C/C++ build] -> [Settings]. Selecting 2004 rules or 2012 rules from [Compiler] -> [MISRA C Rule Check] -> [Check the sourse by MISRA-C] on the [Tool Settings] tabbed page enables detailed settings for [Apply rule], [Rule check exclusion file], [Output message of the enhanced key word and extended specifications], [Enable inter-module checking] and so on.

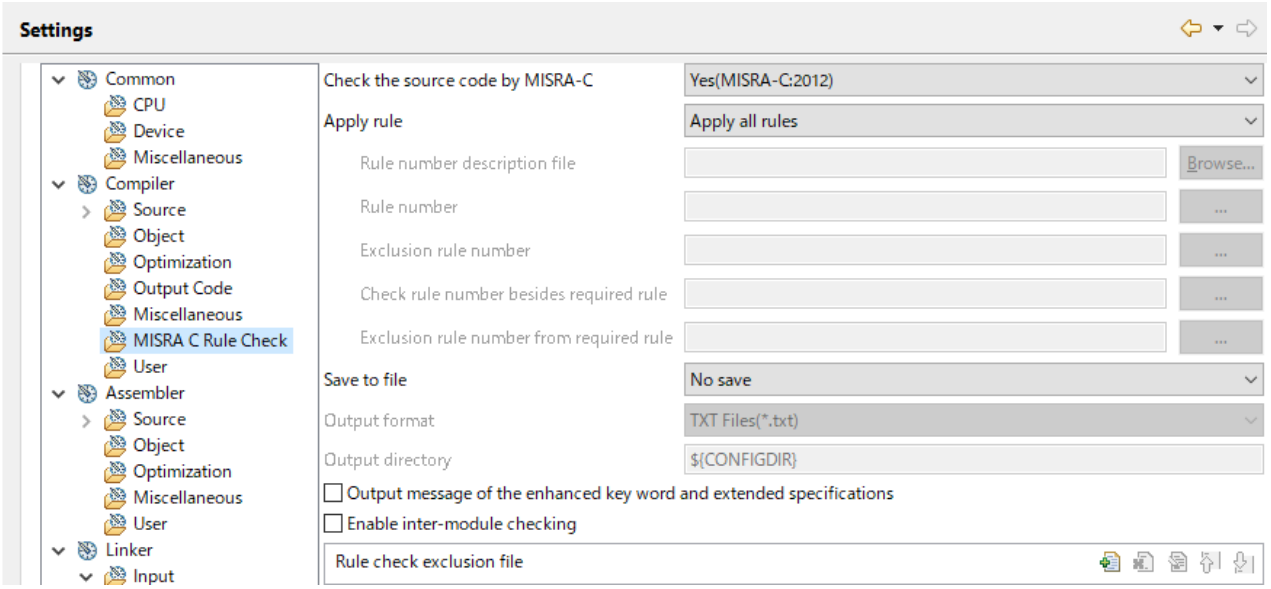



**Figure 2-2 Specifying Options in the e$^2$ studio**

## 2.4 Examples of C Source Code

This section describes examples of C source code which violates MISRA-C:2004/2012 rules and the corresponding output messages.

- Example 1: A violation of rule 2.7 of MISRA-C:2012

| MISRA-C Standards | Rule No. | Classification | Guideline |
|---|---|---|---|
| MISRA-C:2012 | 2.7 | Advisory rule | There should be no unused parameters in functions |

```
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:    /* Parameter variable b is not used.*/
13:    return;
14: }
```

Since parameter variable b declared in the sixth line is not used in the function *func*, the following message is displayed through the standard error output. The message is displayed in the output window in the case of CS+ and in the console in the case of the e$^2$ studio.

```
file name.c(6):M0523086:Rule 2.7:There should be no unused parameters in functions
```

- Example 2: Violations of rule 9.2 of MISRA-C:2004 and rule 9.3 of MISRA-C:2012

| MISRA-C Standards | Rule No. | Classification | Guideline |
|---|---|---|---|
| MISRA-C:2004 | 9.2 | Required rule | Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures. |
| MISRA-C:2012 | 9.3 | Required rule | Arrays shall not be partially initialized |

```
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};
```

In the fourth line, since only nine elements are initialized but the array has 10 elements, the following messages are displayed through the standard error output. The messages are displayed in the output window in the case of CS+ and in the console in the case of the e$^2$ studio.

- For the MISRA-C:2004 rule:

```
file name.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.
```

- For the MISRA-C:2012 rule:

```
file name.c(4): M0523086:Rule 9.3: Arrays shall not be partially initialized
```

- Example 3: Violations of rule 10.1 of MISRA-C:2004 and rule 10.3 of MISRA-C:2012

| MISRA-C Standards | Rule No. | Classification | Guideline |
|---|---|---|---|
| MISRA-C:2004 | 10.1 | Required rule | 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 | 10.3 | Required rule | The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category |

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

In the third line, since a different type (the value returned by the sizeof operator) is assigned to a variable of the unsigned short type, the following messages are displayed through the standard error output. The messages are displayed in the output window in the case of CS+ and in the console in the case of the e$^2$ studio.

- For the MISRA-C:2004 rule:

```
file name.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
```

- For the MISRA-C:2012 rule:

| |
|---|
| file name.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* |

- Example 4: Violations of rules 12.1 and 20.7 for MISRA-C:2012

| MISRA-C Standards | Rule No. | Classification | Guideline |
|---|---|---|---|
| MISRA-C:2012 | 12.1 | Advisory rule | The precedence of operators within expressions should be made explicit |
| MISRA-C:2012 | 20.7 | Required rule | Expressions resulting from the expression of macro parameters shall be enclosed in parentheses |

```
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);
```

In the ninth line, since the precedence in the macro-expanded expression is not explicit and the macro parameters are not enclosed in parentheses, the following messages are displayed through the standard error output. The messages are displayed in the output window in the case of CS+ and in the console in the case of the e$^2$ studio.

Note that the macro-expanded expression is calculated as ‘128 -16 * 2’, which causes an incorrect result if you had intended ‘(128 -16) * 2’.

| |
|---|
| file name.c(9):M0523086:*Rule 20.7: Expressions resulting from the expression of macro parameters shall be enclosed in parentheses*<br>file name.c(9):M0523086:*Rule 12.1: The precedence of operators within expressions should be made explicit* |

# 3. Detection of Stack Smashing

When the compiler generates the code for dynamically checking whether the stack area is smashed or not, it is possible to develop a program with improved safety features such as the prevention of stack overflows or security attacks.

## 3.1 Overview of the Feature

An area of stack is reserved for each function on entry to the function (prologue processing) and consists of the local variable area and register saving area used in that function. When the detection of stack smashing is applied to the stack, a 4-byte area (2-byte area for CC-RL) immediately before the local variable area of the stack (in the direction of increasing addresses) that is related to the function is acquired and a specified value is stored there. The user can specify the value or the compiler can specify an arbitrary value. This is referred to as the monitoring area in this document.

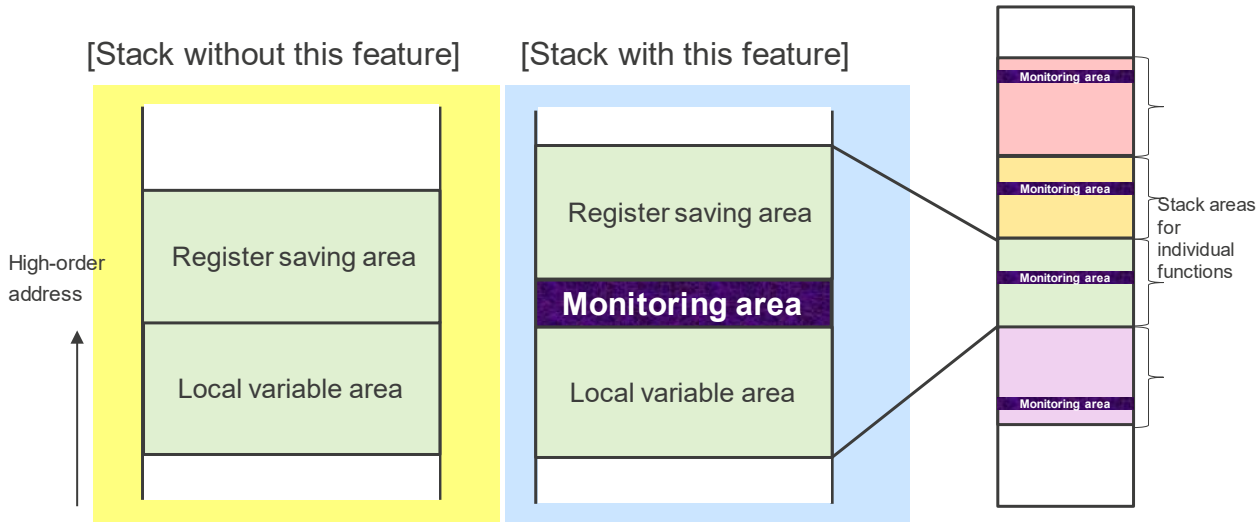



**Figure 3-1 Images of Stacks**

After the function is executed, the stack area it was using is released. Generate the code that checks whether the value stored in the monitoring area has not been overwritten at the exit from the function (epilogue processing). If that value has been overwritten, an overflow is considered to have occurred in the local variable area of the stack and the monitoring area or the areas at the higher addresses (register saving areas) might have been smashed.

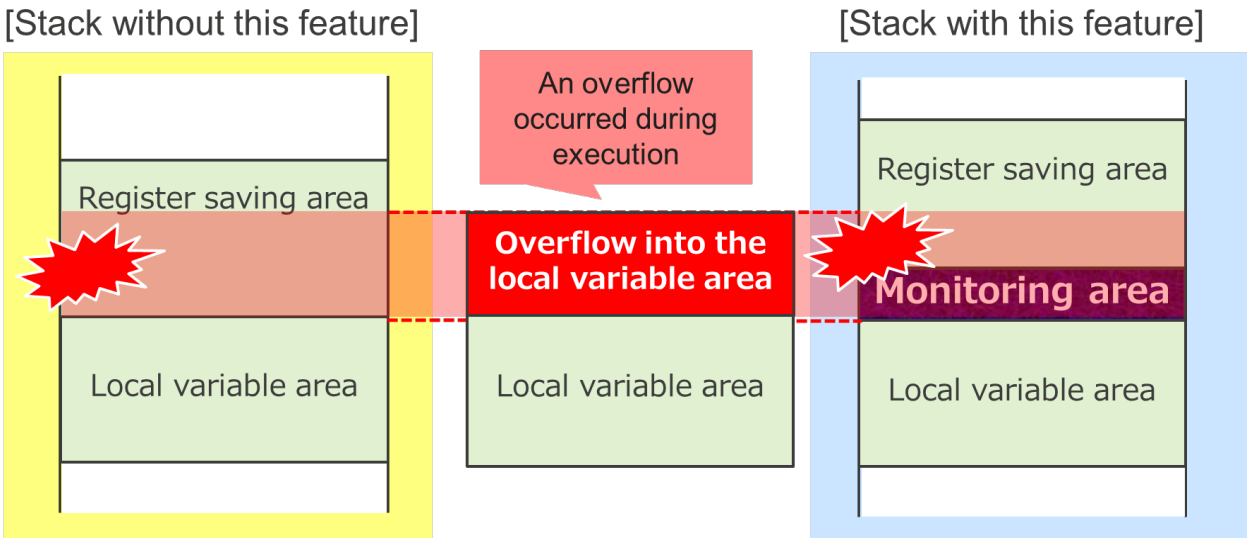



**Figure 3-2 Images of Stack Smashing**

Since information on the address for the return of execution from a function when the function ends is saved in the register saving area, if the address for return has been smashed, the program will jump to an unintended address and may enter runaway execution.

If stack smashing occurs, using the detection feature causes a branch to an error function, which enables dynamic checking for the generation of stack smashing and protects the program against entering runaway execution.

## 3.2 Overview of Generated Code

When this feature is not used, the function is executed after reserving the stack, the stack is released after the function has been executed, and exit from the function proceeds.

When this feature is enabled, a monitoring area is stored in the stack area when the stack area is reserved and a check that the monitoring area has not been overwritten is run after the function has been executed. If the monitoring area has been overwritten, the program branches to the error function "__stack_chk_fail".

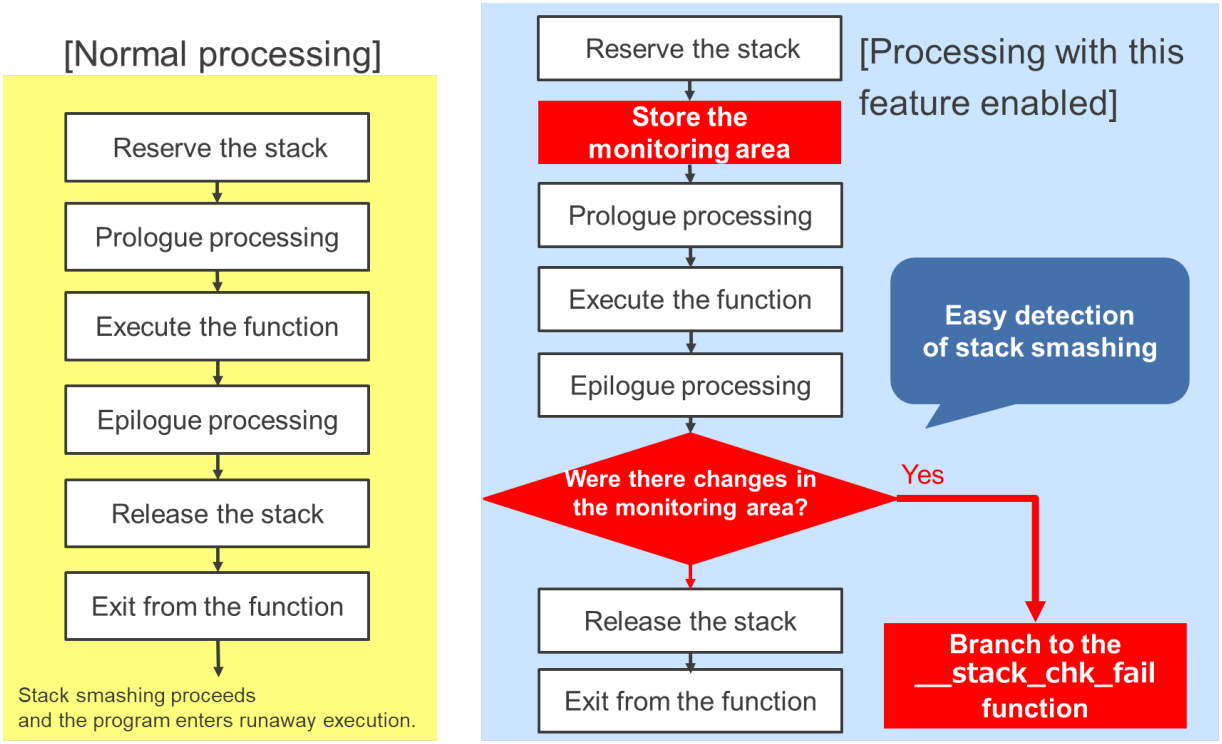



**Figure 3-3 Code Generated by a Compiler**

If stack smashing occurs while this feature is not in use, the program may enter runaway execution in the subsequent processing. Debugging must start with investigation of the reason for the runaway condition.

While this feature is in use, "__stack_chk_fail" is called during the execution of a program. Even if stack smashing occurs, you can stop the program entering runaway execution, by simply identifying the function that called "__stack_chk_fail" during debugging, and review the processing by the function at an early stage.

The __stack_chk_fail function needs to be defined by the user. Describe the processing to be executed upon detection of stack smashing.

## 3.3 How to Use This Feature

The detection of stack smashing can be activated and controlled by compiler options or extended language directives.

a. Specifying compiler options
When the compiler options below are specified, functions satisfying a specific condition can be set as targets for the detection of stack smashing or all functions can be made targets for the detection of stack smashing. When a numeric value is specified as a parameter, the value specified as the parameter is stored in the monitoring area. If this parameter is omitted, the compiler automatically specifies and stores a value.

**Table 3-1 Options for Detecting Stack Smashing**

| | Option | | |
|---|---|---|---|
| **Description** | **CC-RL** | **CC-RX** | **CC-RH** |
| This option generates a code for detection of stack smashing for only functions having a structure, union, or array that exceeds eight bytes as a local variable. | -stack_protector | -stack_protector | -Xstack_protector |
| This option generates a code for detection of stack smashing for all functions. | -stack_protector_all | -stack_protector_all | -Xstack_protector_all |

When you are using CS+ or the e$^2$ studio as the integrated development environment, you can control the specification of options by operations in the GUI.

- For CS+
Select [Yes] or [No] for the [Detect stack smashing] property from the [Quality Improvement] category on the [Compile Options] tabbed page. The value to be stored in the monitoring area can be specified as the [Value to be embedded for detecting stack smashing] property.

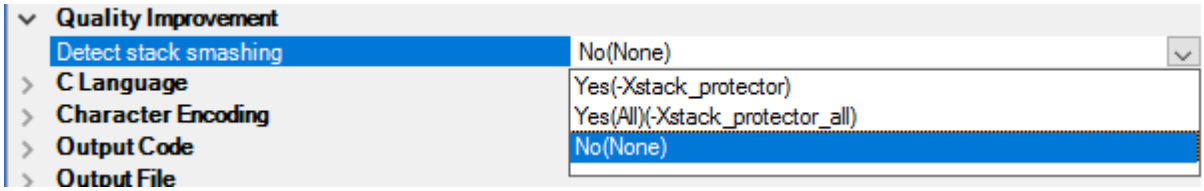



**Figure 3-4 Specifying Options in CS+**

- For the e$^2$ studio
  Activate the Property dialog box of the project from [Project] -> [Renesas Tool Settings] and select [C/C++ build] -> [Settings]. Select [Yes] or [No] for [Detect stack overflow] from [Compiler] -> [miscellaneous] on the [Tool Settings] tabbed page. The value to be stored in the monitoring area can be specified as the [Value to be embedded for detecting stack overflow] property.

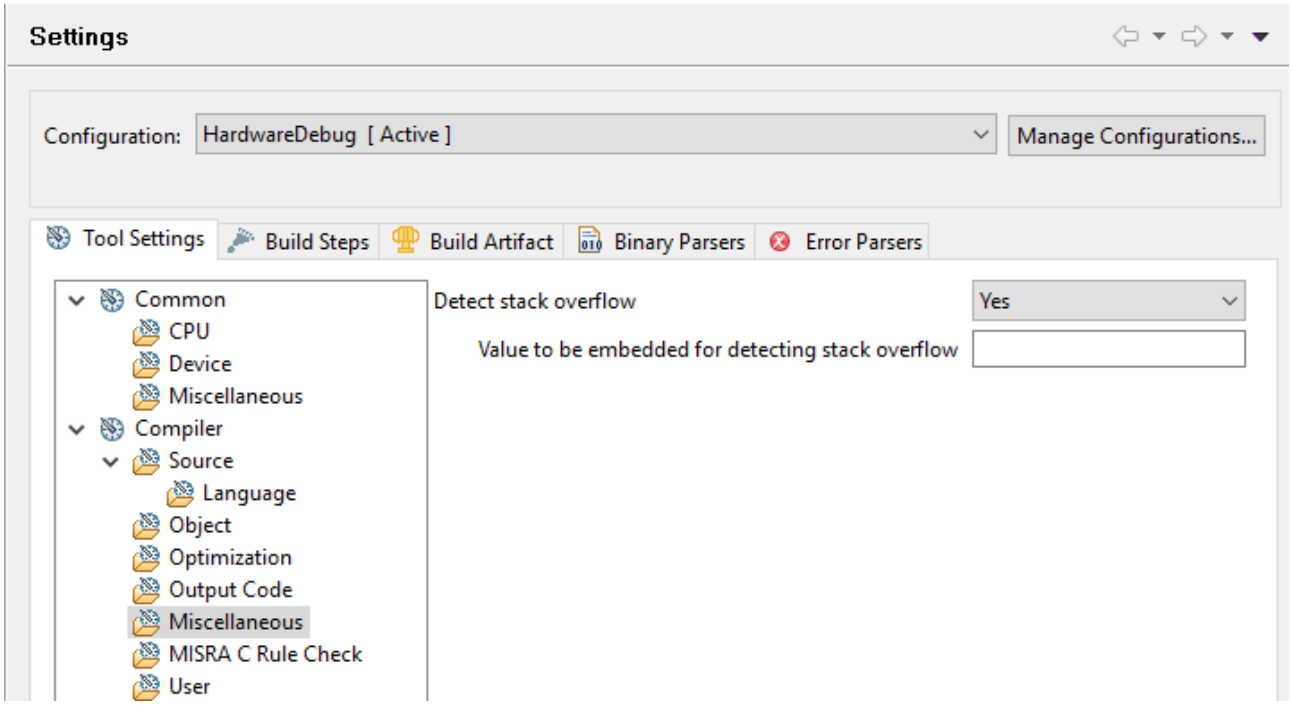



**Figure 3-5 Specifying Options in the e$^2$ studio**

b. Using extended language directives to specify detection
   When the extended language directive below is specified, specific functions can be made targets for the detection of stack smashing.

[Syntax]

| #pragma stack_protector *function name* (num=*Specified value*) |
|---|

The function specified by *function name* is the target for the detection of stack smashing and the *specified value* is stored in the monitoring area. When (num=*specified value*) is omitted, the compiler automatically specifies and stores a value.

[Syntax]

| #pragma no_stack_protector *function name* |
|---|

The function specified by *function name* is not a target for the detection of stack smashing. When both compiler options and extended language directives are specified, the directives are given priority.

## 3.4 Examples of C Source Code

The following gives examples of the C source code in which this feature is enabled. Note that the CC-RX compiler does not include the __halt(); intrinsic function.

- Example 1: Incorrect calculation of an area

```
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: }
```

In the 10th line, where sizeof(str_t*) has wrongly been written although it should have been sizeof(str_t), the value of BUF_SIZE is not the size of structure str_t (10) × 16 but the size of the pointer × 16. Thus less area than was assumed is secured, and more area than was secured is written by the for loop in the 19th to 21th lines and stack smashing occurs.

When the feature for detection is enabled, the error function “__stack_chk_fail” is called at the end of the function `func`, so detection of stack smashing is easy.

- Example 2: Failure to apply exclusive control

```
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:    /* <= Generation of an interrupt with intrpt_func() as the service routine.*/
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:
29: void __stack_chk_fail(void) {
30:    __halt();
31: }
```

If an interrupt with “intrpt_func” as its service routine occurs in the 17th line, the value of variable g_cnt_max is overwritten, processing of the for loop from line 19 leads to writing beyond the area for the local variable buf, and the stack is smashed.

If this feature is enabled, the error function “__stack_chk_fail” is called at the end of the function *func* and stack smashing has easily been detected.

# 4. Enhanced Security for Dynamic Memory Management Functions

Using the calloc, malloc, and realloc functions which have safety features for reserving memory in the heap enables the development of programs for which security is enhanced by preventing problems such as releasing memory twice or overflows in the heap.

## 4.1 Overview of the Feature

When part of the heap is reserved by the calloc, malloc, or realloc function, reserve the preceding and following four bytes (2 bytes for CC-RL) of the heap and store any value in those areas. This is referred to as the monitoring area in this document.

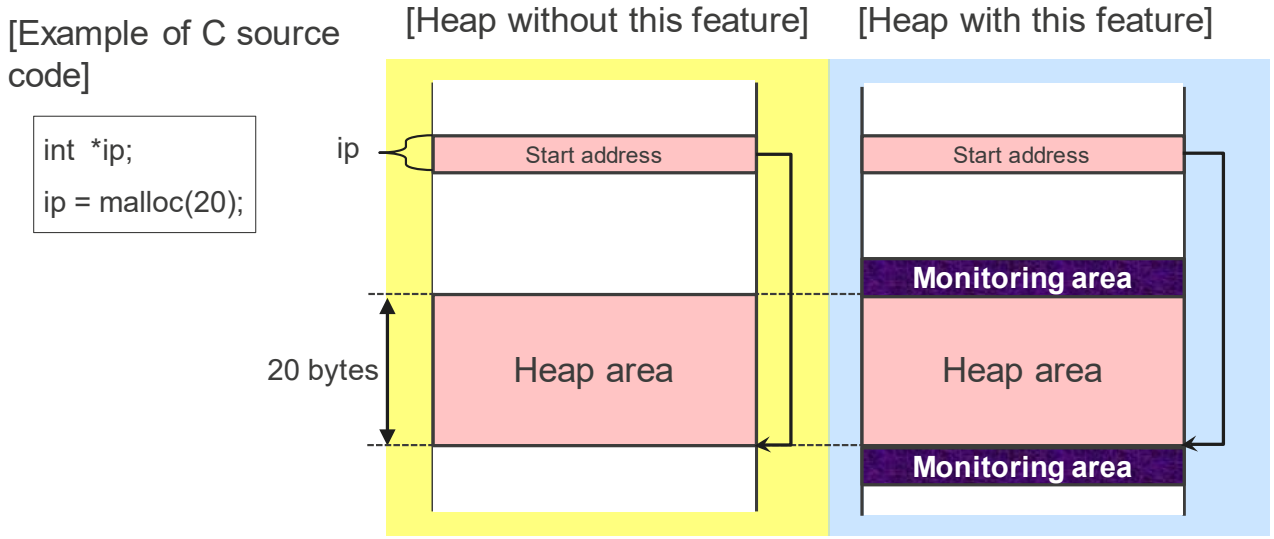



**Figure 4-1 Image of a Heap**

After an operation involving the heap, the given area is released by calling the free or realloc function. The value stored in the monitoring area is checked to see that it has not been overwritten. If the value has been overwritten, the program is regarded as incorrect because it causes an overflow in the heap and execution then branches to error processing.

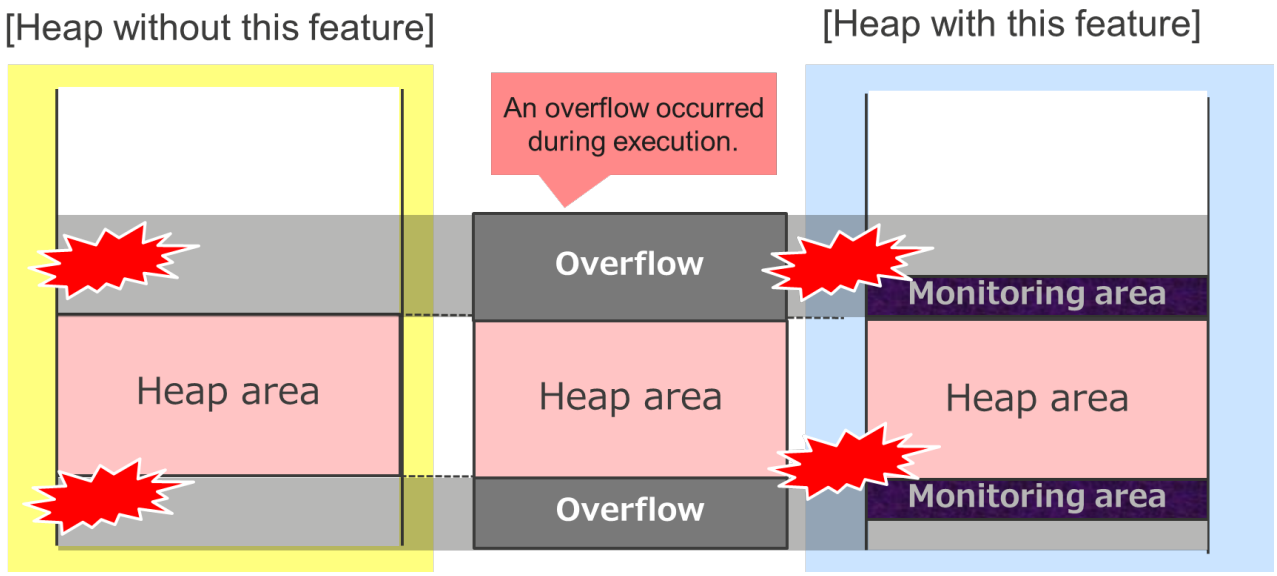



**Figure 4-2 Image of an Overflow Generated in the Heap**

If the following operations are performed when parts of the heap are released by the free or realloc function, execution will similarly branch to error processing.

- ✓ The pointer to an area other than that allocated by calloc, malloc, or realloc is passed to free or realloc.
- ✓ The pointer to an area released by free is passed again to free or realloc.

Performing an illicit operation on the heap causes execution to branch to an error function, which enables dynamic checking for the generation of erroneous operations in the heap, so that such malfunctions in programs are detectable at an early stage.

## 4.2 Overview of Generated Code

When this feature is not in use, reserve and proceed with operations in the area from the heap, and then use free or calloc to release the heap area.

When this feature is enabled, check that the monitoring area is not overwritten when the reserved heap area is released. If the monitoring area has been overwritten, the program branches to the error function "__heap_chk_fail". The program also branches to this function if the pointer is to an area other than one reserved by calloc, malloc, or realloc or the pointer is to an area that has already been released.

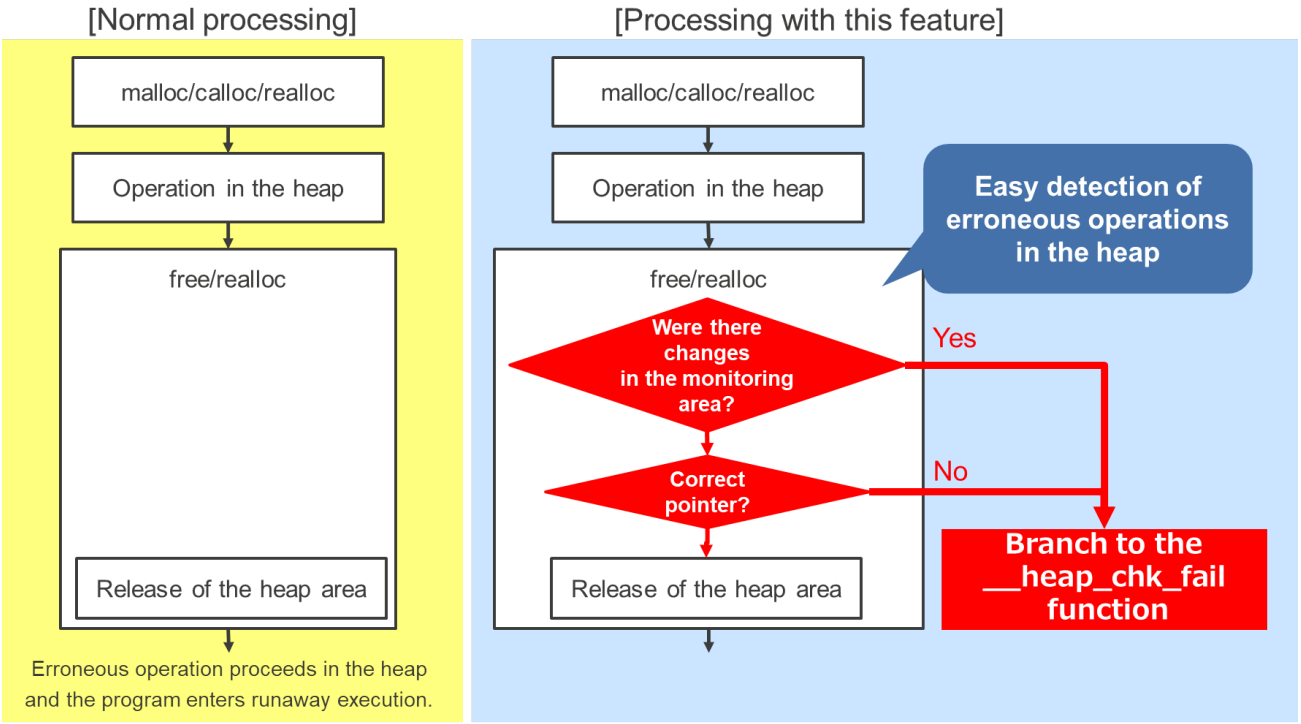



**Figure 4-3 Code Generated by a Compiler**

If erroneous operation occurs in the heap while this feature is not in use, the program may enter runaway execution in the subsequent processing. Debugging must start with investigation of the reason for the runaway condition.

While this feature is in use, "__heap_chk_fail" is called during the execution of a program. Even if erroneous operation occurs in the heap, you can stop the program entering runaway execution, by simply identifying the function that called "__heap_chk_fail" during debugging, and review the processing by the function at an early stage.

The __heap_chk_fail function needs to be defined by the user. Describe the processing to be executed when an error occurs in management of dynamic memory.

## 4.3 How to Use This Feature

This feature is used by linking to standard libraries that include versions of the calloc, malloc, and realloc functions with the safety feature included. The method of linking of this library differs from each compiler.

**Table 4-1 Libraries for Linking that Include the Safety Feature**

| | |
|---|---|
| **CC-RL** | Link the following library with the linker option “-library”.<br>CS+¥CC¥CC-RL¥Vx.xx.xx¥lib¥malloc_s.lib |
| **CC-RX** | Specify the library generator option “-secure_malloc”. |
| **CC-RH** | Link the following library with the linker option “-library”.<br>CS+¥CC¥CC-RH¥Vx.xx.xx¥lib¥v850e3v5¥secure¥libmalloc.lib |

When you are using CS+ or the e$^2$ studio as the integrated development environment, you can control specifying options by operating the GUI.

- For CS+
  For CC-RL and CC-RH, select [Yes] or [No] for the [Check memory smashing on releasing memory] property from the [Library] category on the [Link Options] tabbed page. For CC-RX, select [Yes] or [No] for the [Check memory smashing on releasing memory] property from the [Object] category on the [Library Generate Options] tabbed page.

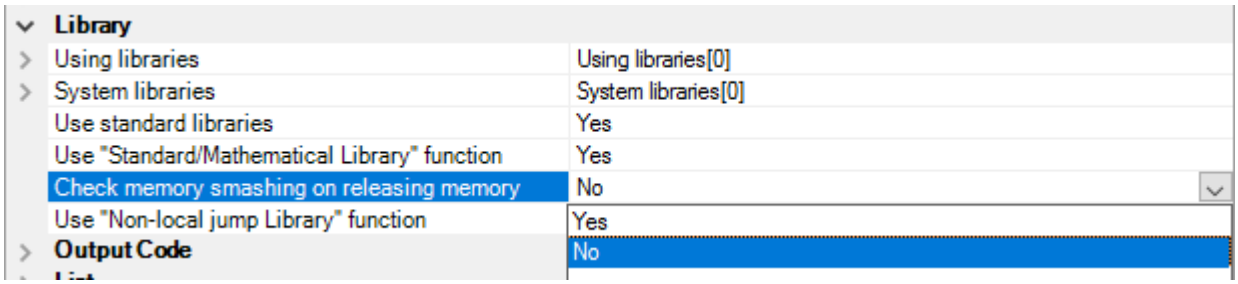



**Figure 4-4 Specifying Options in CS+**

- For the e$^2$ studio
  For CC-RL, activate the Property dialog box of the project from [Project] -> [Renesas Tool Settings] and select [C/C++ build] -> [Settings]. On the [Tool Settings] tabbed page, select or deselect the checkbox of [Check memory smashing on releasing memory] from [Linker] -> [Input]. For CC-RX, on the [Tool Settings] tabbed page, select or deselect the checkbox of [Check memory smashing on releasing memory] from [Standard Library] -> [Object].

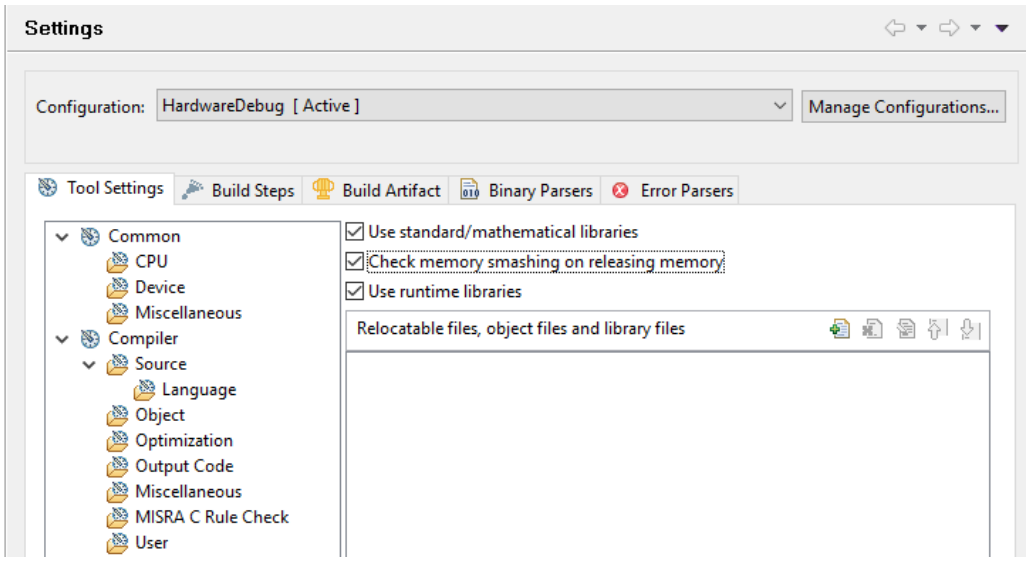



**Figure 4-5 Specifying Options in the e2 studio (CC-RL)**

## 4.4 Examples of C Source Code

The following gives examples of the C source code in which this feature is enabled. Note that the CC-RX compiler does not include the __halt(); intrinsic function.

- Example 1: The area at the destination for copying is smaller than the area taken up by the data at the source, memcpy is executed with the size parameter indicating the amount of data at the source, so a buffer overflows.

➢ 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:  }
```

In the 14th line of func.c, memcpy is executed with the size parameter indicating the length of line, which is the data at the source for copying, instead of the size of bufa, which is the destination for copying. If line takes up more space than is available in bufa, the bufa part of the heap will be smashed.

When func is called in the seventh line of main.c, since line takes up 40 bytes and the bufa type has 32 bytes, the program smashes the area for bufa in the heap.

When the feature for detection is enabled, the error-handling function “__heap_chk_fail” is called when the bufa area is released, so detection of the buffer overflow is easy.

- Example 2: Double release of an area in the heap due to erroneous release of the area by a called function

➢ 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:  }
```

When `status` in the 27th line in func.c is less than 0, the area reserved in the 14th line is released on the 28th line. Since the same area is later released on the 22nd line, a pointer to an area that has already been released will be released again.

Since -10 is assigned to `status` in the fifth line of main.c and the program then branches to the function `func`, an area in the heap is released twice.

When the feature for detection is enabled, the error-handling function "__heap_chk_fail" is called during execution of the 22nd line of func.c, so detection of the area in the heap being released twice is easy.

# 5. Half-precision Floating Point

Support for the 2-byte half-precision floating-point type can reduce the size of programs that contain large amounts of floating-point data.

Note that this feature is specific to CC-RH compiler.

## 5.1 Overview of the Feature

CC-RH supports 2-byte floating-point format (in addition to the typical 4- and 8-byte floating-point type formats). This data type is called half-precision floating-point, and can be defined as __fp16 type.

The size and the alignment condition are two bytes and the internal representation of data conforms to binary16 in the IEEE 754-2008 standard.

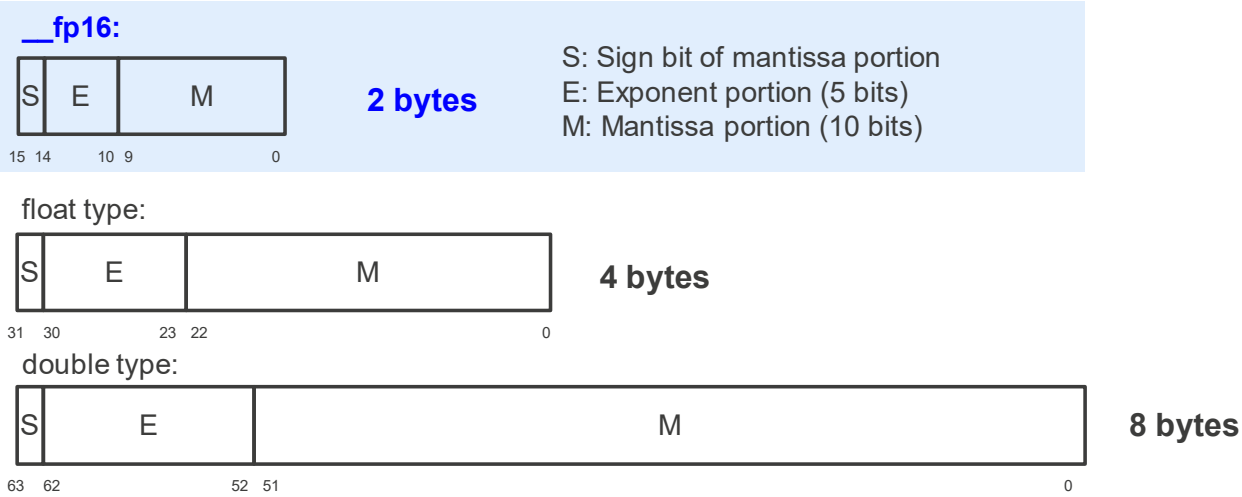



**Figure 5-1 __fp16 Type for Half-precision Floating Point**

The compiler supports the following operations.

- ✓ Assignment between __fp16 type values
- ✓ Type conversion from __fp16 to float
- ✓ Type conversion from float to __fp16

Other operations are to be performed after values have been converted into the float type, and the result will have the same type as that when the same operation is performed for variables of the float type. For example, in the case of type conversion from __fp16 to double, this proceeds after the value has been initially converted into the float type.

## 5.2 Overview of Generated Code

Float-type and double-type floating point values are loaded from memory to general-purpose registers to perform operations and the register values are then stored in memory.

Half-precision floating point values are loaded from memory to general-purpose registers and the values in the general-purpose registers are converted into single-precision floating point values by the FPU instruction CVTF.HS (Convert Floating-point Half to Single). Operations proceed with the single-precision floating point values and the resulting values in the general-purpose registers are stored in memory after they have been converted from single-precision floating point to half-precision floating point by the FPU instruction CVTF.SH (Convert Floating-point Single to Half).

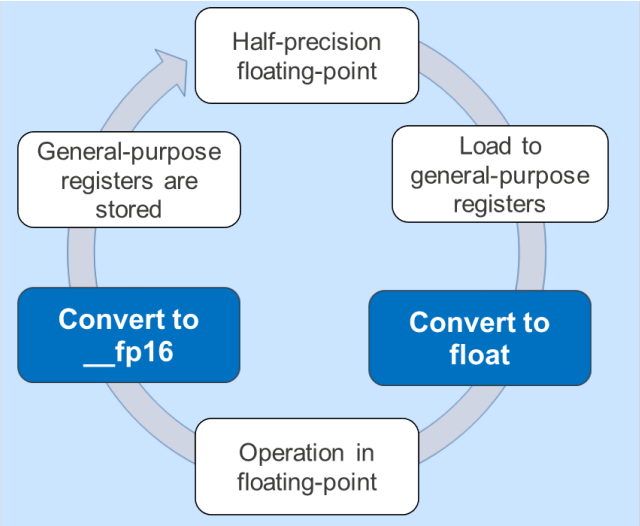



**Figure 5-2 Code Generated by a Compiler**

## 5.3 How to Specify the Half-precision Floating-point Type

Specifying the compiler option “-Xuse_fp16” allows use of the half-precision floating-point type.

When you are using CS+ as the integrated development environment, you can control the specification of options by operations in the GUI.

- For CS+
  Select [Yes] or [No] for the [Enable half precision floating-point type] property from the [Output Code] category on the [Compile Options] tabbed page.

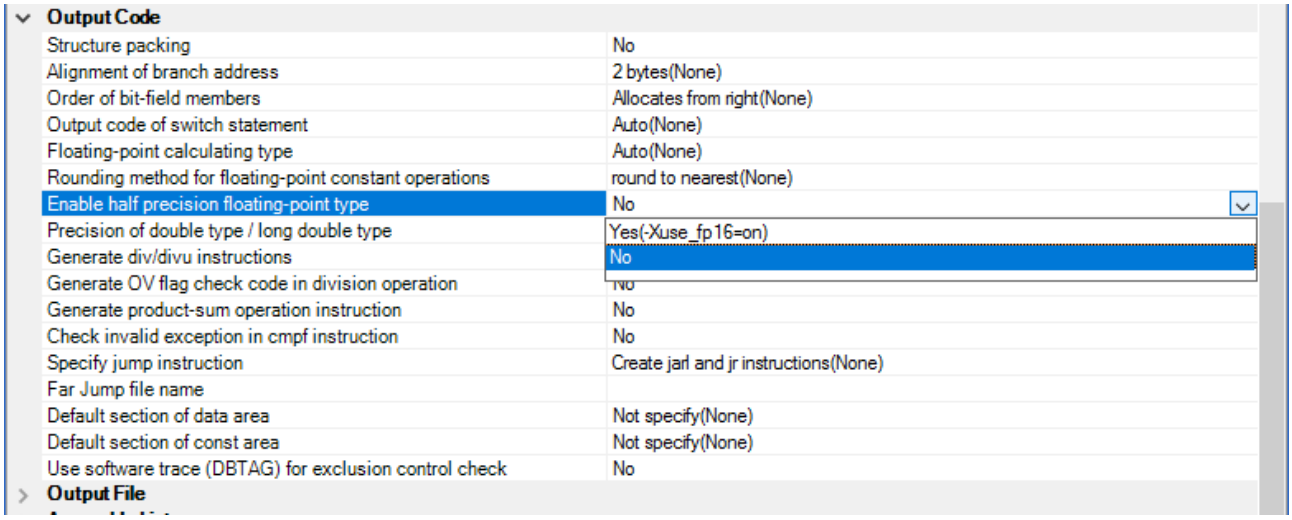



**Figure 5-3 Specifying Options in CS+**

## 5.4 Example of C Source Code

- Example: Use of a large number of floating-point constant values

```
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: }
```

When a large amount of floating-point constant data is used as shown in the first to sixth lines, using the half-precision floating-point type reduces the size of array coef. The size of the array is 40 bytes when it is defined as half-precision floating-point but 80 bytes when defined as single-precision floating-point; the amount of data is thus halved.

# 6. Synchronization Features in the Updating of Control Registers

These features reduce the load on the user when control registers of an RH850 are successively updated.

Note that these features are specific to the CC-RH compiler.

## 6.1 Overview of the Features

When control registers of an RH850 are successively updated by store instructions, the order of the control registers may not match that in which they were written in the source file. To make the order match, synchronization processing, which causes a wait until completion of the execution of a preceding instruction before proceeding with execution of the next instruction, must be manually inserted.

However, synchronization processing is not always required to guarantee the order. When control registers in the same peripheral group are successively updated, synchronization processing is not required because the order is guaranteed.

Thus, for a source file that includes processing for the updating of all control registers, the user is required to visually determine to which peripheral groups the control registers belong through reference to the hardware manual and to manually insert synchronization processing.

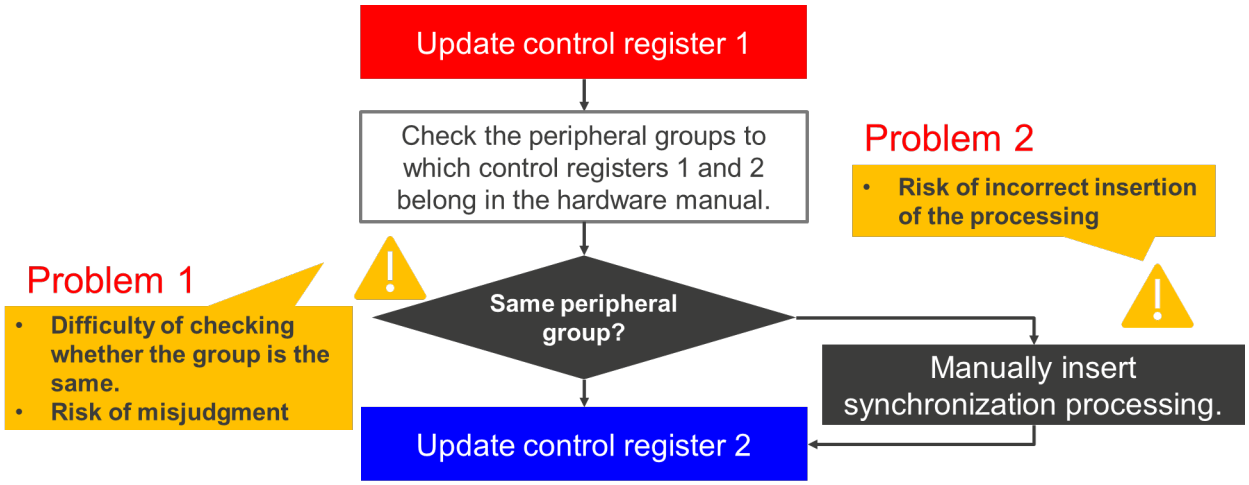



**Figure 6-1 Problems in the Updating of Control Registers**

In CC-RH, the following features solve the two problems indicated in Figure 6-1.

We refer to these as features for synchronization in the updating of control registers.

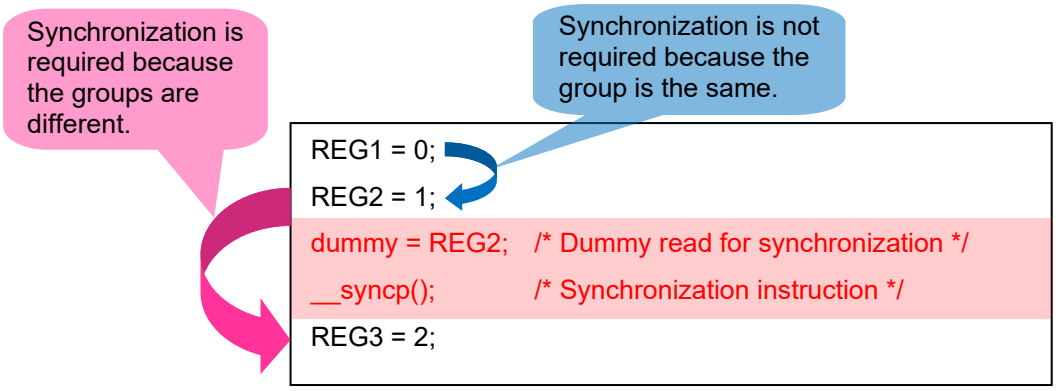
- Feature for the insertion of synchronization processing
  →Problems 1 and 2 are solved at the same time.
- Feature for the detection of writing to control registers
  →Problem 1 is solved.

## 6.2 Overview of Generated Code

➢ Feature for the insertion of synchronization processing

In the case of writing to control registers, the compiler automatically inserts synchronization processing in the assembler source file. If the control registers are in the same peripheral group, the synchronization processing is omitted. Thus **the user determines whether synchronization processing is required and need not manually insert code for synchronization processing.**

Example: When registers are successively updated in the order REG1 (CPU group), REG2 (CPU group), then REG3 (0 group), the compiler outputs code for synchronization processing as shown in red text below.




➢ Feature for the detection of writing to control registers

Information messages are output showing the names of source files that include writing to control registers, the line numbers, and the names of peripheral groups to which the control registers belong and the addresses of the registers.

```
src.c(9):M0536001:control register is written.(id=group ID, address of the control register)
src.c(10):M0536001:control register is written.(id= group ID, address of the control register)
src.c(11):M0536001:control register is written.(id= group ID, address of the control register)
```

For the feature for the insertion of synchronization processing, synchronization processing is inserted even in cases where the order of the control registers need not match that in which they were written in the source file. Accordingly, when synchronization processing is to be manually inserted only in the required locations on the basis of judgment by the user, use the feature for the detection of writing to control registers. **This can eliminate the load of referring to the hardware manual for the names of peripheral groups to which the registers belong.**

## 6.3 How to Use These Features

The feature for the insertion of synchronization processing during the updating of control registers becomes available through the following steps a and b.

a. Specifying the address ranges of peripheral groups with a language extension
   Specify the start address and end address of each peripheral group with the following language extension.

Format:

```
#pragma register_group start-address, end-address, id="group-ID"
```

The *group-ID* is an identifier for specifying the peripheral group to which the control registers belong. Refer to the peripheral group and address information that are described in the register list in the hardware manual and specify the address range for each peripheral group. In some cases, the address range may not be contiguous even for the same group and the same *group-ID* can be specified for more than one #pragma register_group directive.

The names of the peripheral groups need not match those given in the hardware manual.

If you are using an MCU which incorporates the G4MH core, the automatic generation of a file containing this language extension to define the ranges of peripheral groups is available. Refer to section 6.5, Supplementary Items, for details.

b. Specifying a compiler option
   The feature for the insertion of synchronization processing during the updating of control registers becomes available by specifying the -store_reg compiler option.

Format:

```
-store_reg=item
```

**Table 6-1 Options for Synchronization Features**

| Option | Description |
|---|---|
| -store_reg=*sync* | Enables the feature for the insertion of synchronization processing.<br>This option allows the compiler to detect writing to control registers in the ranges specified by #pragma register_group and insert synchronization processing after write instructions for these registers, except where the succeeding instructions will clearly be for writing to the same group, in which case the compiler does not insert synchronization processing. |
| -store_reg=*list* | Enables the feature for the detection of writing to control registers.<br>This option allows the compiler to detect writing to control registers in the ranges specified by #pragma register_group and display the line numbers in the source code of the write instructions in the standard error output, except where the succeeding instructions will clearly be for writing to the same group, in which case the compiler does not display the line number. |
| -store_reg=*list_all* | Enables the feature for the detection of writing to control registers.<br>This option allows the compiler to detect writing to control registers in the ranges specified by #pragma register_group and display the line numbers in the source code of the write instructions in the standard error output. The line numbers are displayed regardless of whether the succeeding instructions will clearly be for writing to the same group. |
| -store_reg=*ignore* | Any #pragma register_group directives are ignored. |

When you are using the CS+ integrated development environment, specifying these options can be controlled through the GUI.

The options can be selected in the [Handling mode of writing control register] property in the [Output Code] category on the [Compile Options] tabbed page.

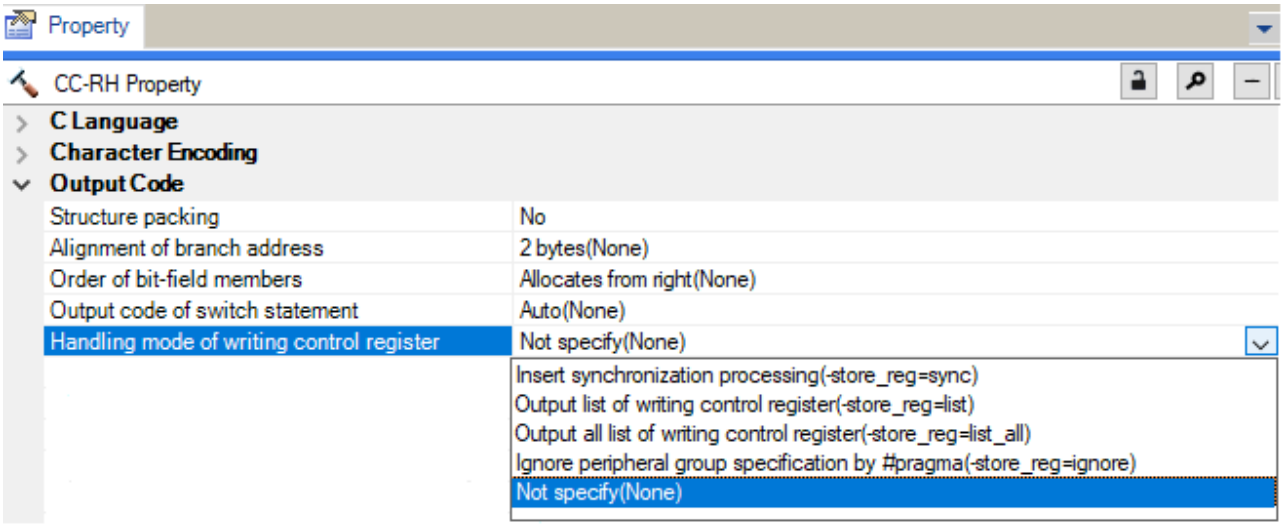



**Figure 6-2 Specifying Options in CS+**

## 6.4 Example of C Source Code

Example: When two peripheral groups (CPU and 0 groups) are defined with #pragma register_group

[pgroup.h]

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

[iodefine.h]

```
#define REG1 (*(volatile unsigned char*)0xfedf0000)     /* Control register of CPU group */
#define REG2 (*(volatile unsigned char*)0xfedf0001)     /* Control register of CPU group */
#define REG3 (*(volatile unsigned short*)0xfee00000)    /* Control register of 0 group */
```

[src.c]

```
1:  #include “pgroup.h”
2:  #include “iodefine.h”
3:
4:  void func(void) {
5:  REG1 = 0;   // Control register of CPU group is updated.
6:  REG2 = 1;   // Control register of CPU group is updated.
7:  REG3 = 2;   // Control register of 0 group is updated.
8:  }
```

In the fifth to seventh lines, values are written to control registers in the order REG1, REG2, and REG3.

When the control registers must be updated in this order without using this feature, the peripheral groups to which REG1 to REG3 belong must be specified through reference to the hardware manual since whether the insertion of synchronization processing is required must be considered.

Consequently, since writing is for the same group on the fifth to sixth lines and for different groups on the sixth to seventh lines, synchronization processing must be inserted between the assignments to REG2 and REG3. In addition, if a value may be written to a control register which belongs to a different group after processing of func has completed, synchronization processing must also be inserted after updating of REG3.

➢ Specifying the -store_reg=sync option

The compiler generates the assembly instructions shown below during compilation.

```
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]
```

**Synchronization processing is not inserted because this is followed by writing to the same group.**

REG1 (CPU group) is

REG2 (CPU group) is updated.

**Synchronization processing is inserted because this is followed by writing to a different group.**

REG3 (0 group) is updated.

**Synchronization processing is inserted because processing after return from func() is undefined.**

➢ Specifying the -store_reg=list option

The compiler displays the messages shown below in the standard error output so that the user can easily consider whether synchronization processing must be inserted or not, except where the succeeding instructions will clearly be for writing to the same group, in which case the compiler does not display the line number.

The fifth line is not indicated.

```
src.c(6):M0536001:M0536001:control register is written.(id=CPU, 0xfedf0001)
src.c(7):M0536001:M0536001:control register is written.(id=0, 0xfee00000)
```

➢ Specifying the -store_reg=list_all option

The compiler displays the messages shown below in the standard error output so that the user can easily consider whether synchronization processing must be inserted or not. The line numbers are displayed regardless of whether the succeeding instructions will clearly be for writing to the same group.

The fifth line is also indicated.

```
src.c(5):M0536001:M0536001:control register is written.(id=CPU, 0xfedf0000)
src.c(6):M0536001:M0536001:control register is written.(id=CPU, 0xfedf0001)
src.c(7):M0536001:M0536001:control register is written.(id=0, 0xfee00000)
```

## 6.5 Supplementary Items

If you are using an MCU which incorporates the G4MH core, you can set up the automatic generation of an “iodefine_pgroup.h” file for the MCU specified in the CS+ project. This header file contains #pragma register_group directives (a language extension of the CC-RH compiler) that specify the address ranges of the peripheral groups of the MCU.

The following procedure sets up automatic generation of the “iodefine_pgroup.h” file by CS+.

- From the [I/O Header File Generating Options] tabbed page of the [CC-RH Property] panel, select [Yes (Checking the property)] for [Update I/O header file on build].
- Select [Yes (-pragma_peripheral_group=on)] for [Output pragma directives for peripheral groups].

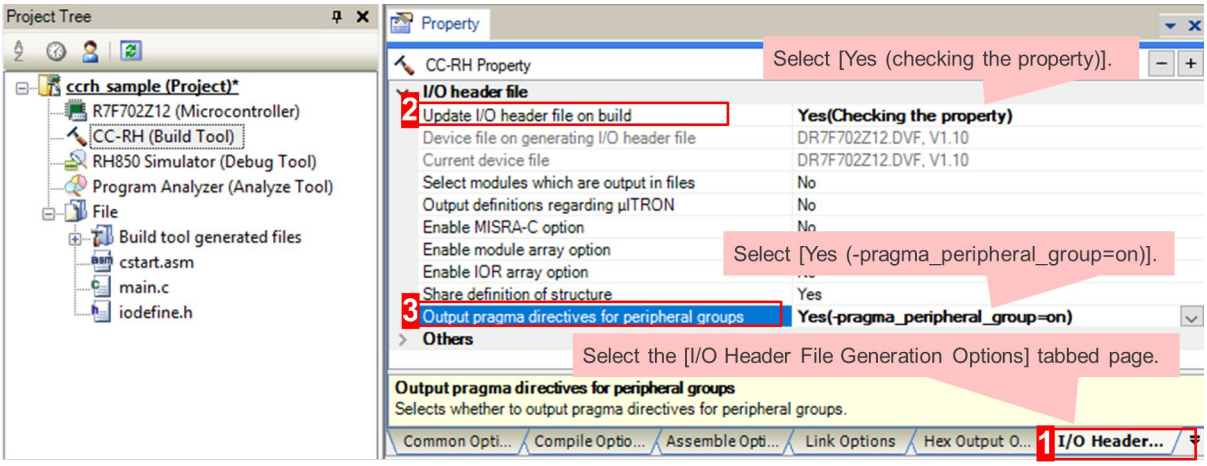



**Figure 6-3 Property Settings in CS+**

- Select [Save the project] from the [File] menu to save the project once.
- “iodefine_pgroup.h” is generated in the project folder.

Including this header file in the C source file places statements involving the updating of control registers within the scope of the synchronization feature when the C source file is compiled.

# 7. Detection of Illicit Indirect Function Calls

This feature improves the quality of user programs by preventing indirect function calls to non-trusted addresses.

## 7.1 Overview of the Feature

An indirect function call is a method of calling a function in which the address of the function to be called is acquired at runtime. Suppose a case where a buffer is located next to a function pointer area and data are to be written to the buffer. If the processing for writing to the buffer includes a vulnerability that allows modification of the data outside the buffer, the value of the function pointer can be modified through external input. When the modified function pointer is used in an indirect function call, software execution may go out of control or, in the worst-case scenario, the system may be taken over by a malicious attacker. The feature for detecting illicit function calls is provided to prevent this situation.

The compiler automatically executes the following processing.

1. Extracts from programs the functions that may be indirectly called and registers them in a list of such functions.
2. Generates code for checking the address of each function immediately before the function is indirectly called.

## 7.2 Overview of Generated Code

During compilation, the compiler automatically extracts from the C source programs the functions that may be indirectly called. The extracted information on the functions is collected in a list of safe function addresses (a list of the correct addresses of functions that may be indirectly called) during linkage.

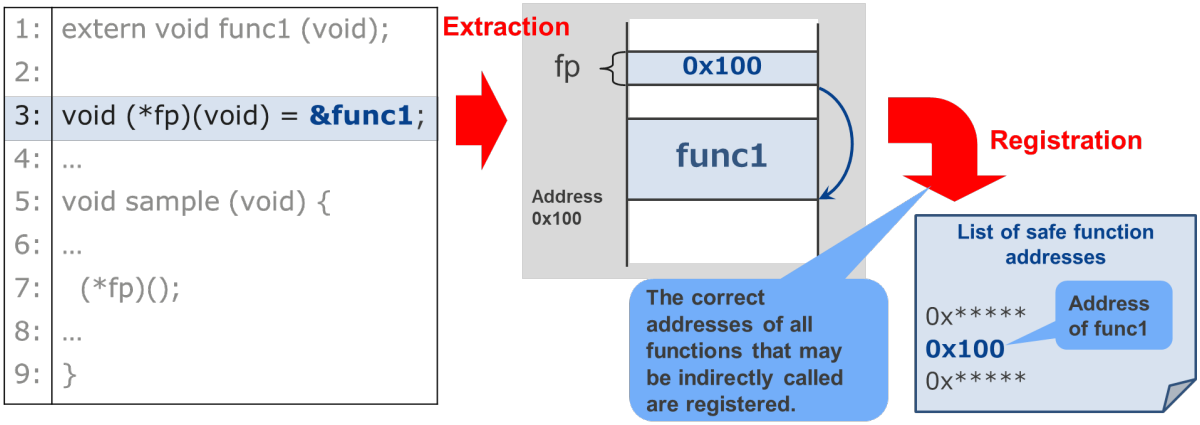



**Figure 7-1 Registration in the List of Safe Function Addresses**

The checking function “__control_flow_integrity” is called immediately before each indirect function call.

This function receives the destination address of the branch caused by the indirect function call as an argument. At runtime, the checking function searches the list of safe function addresses for the received destination address. If the address is found in the list, the function call is handled normally.

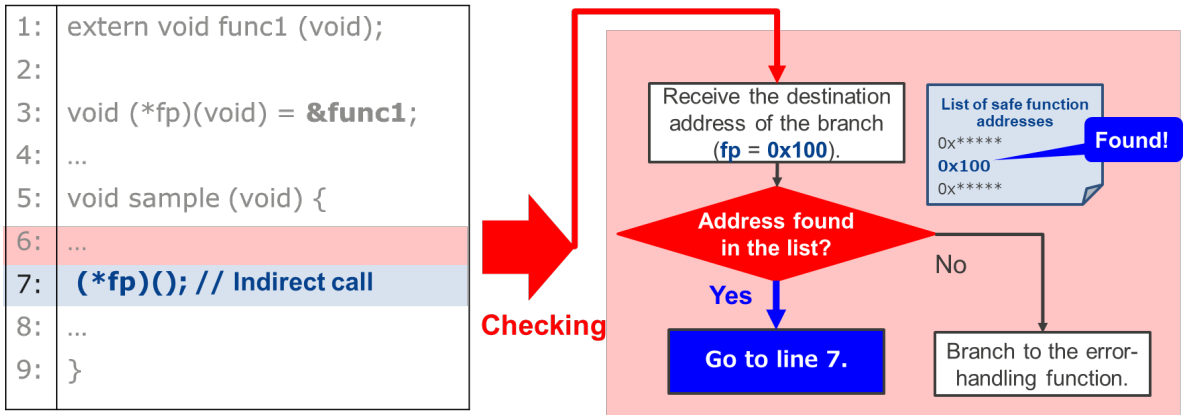



**Figure 7-2 Call of a Function at an Address Found in the List**

If the called function is not found in the list of safe function addresses, the indirect function call is judged to be illicit, and the “__control_flow_chk_fail” function is called to branch to the error-handling process.

In this way, indirect calls of functions that are not registered in the list of safe function addresses can be prevented, and the system can be protected against the program going out of control or its area being maliciously overwritten.

The checking function “__control_flow_integrity” is provided as part of the standard library.

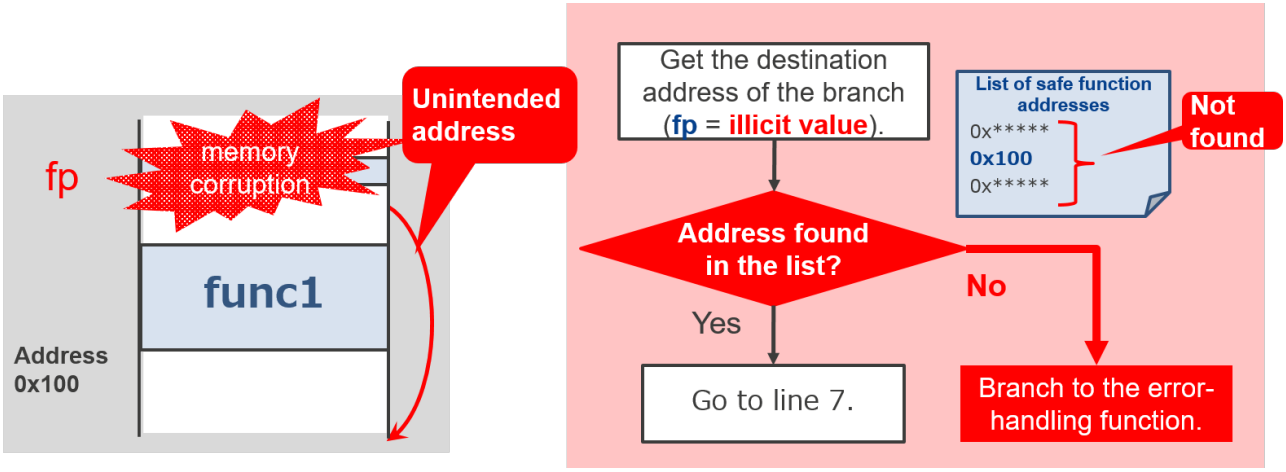



**Figure 7-3 Call of a Function at an Address Not in the List**

## 7.3 How to Use This Feature

Specify the following options to activate this feature.

**Compiler Option:**

The following option selects the generation of code for detecting illicit indirect function calls.

| |
|---|
| -control_flow_integrity |

**Linker Option:**

The following option selects generation of the list of safe function addresses to be used in detecting illicit indirect function calls.

| |
|---|
| -cfi |

If you are using CS+ or the e$^2$ studio as the integrated development environment, you can control the specification of these options through the GUI.

**For CS+:**

Select [Yes] or [No] for the [Detect invalid indirect function call] property under the [Quality Improvement] category on the [Compile Options] tabbed page.

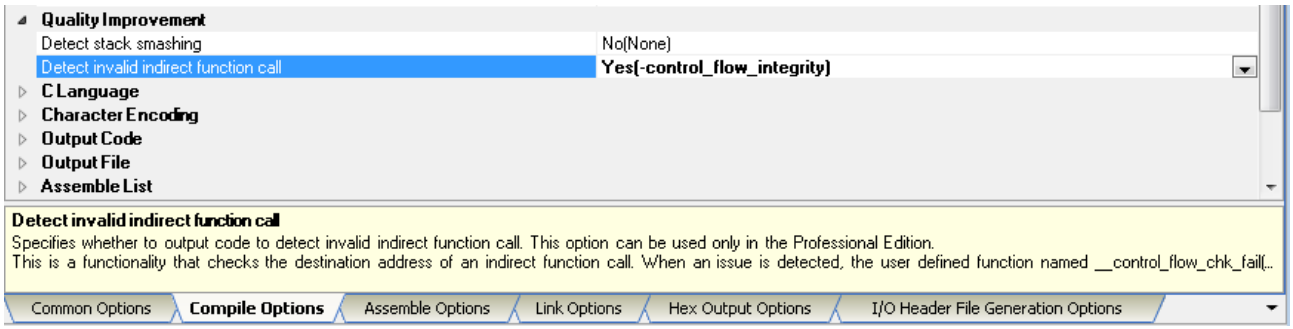



**Figure 7-4 Specifying the Compiler Option in CS+**

If the linker option “–cfi” is not specified before the compiler option “-control_flow_integrity”, the following warning (W0293007) will be displayed.

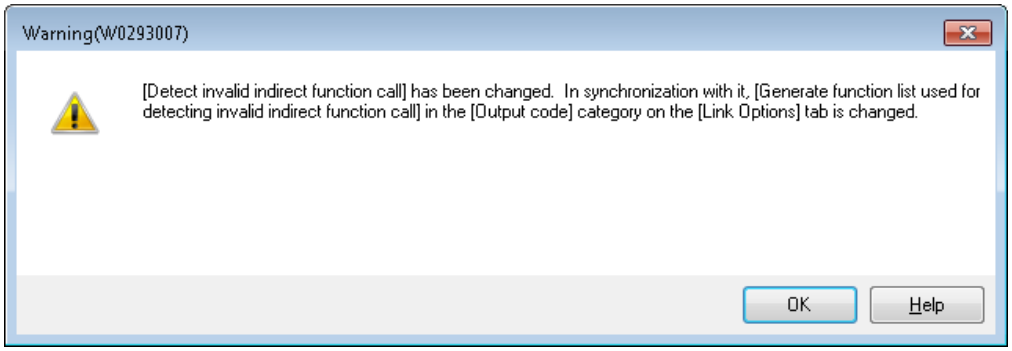



**Figure 7-5 Warning when Specifying the Option**

Clicking on [OK] in the warning message box will change the setting for the [Generate function list used for detecting invalid indirect function call] property under the [Output Code] category on the [Link Options] tabbed page to [Yes].

Specifying this option selects generation of the list of safe function addresses to be used in detecting indirect function calls.

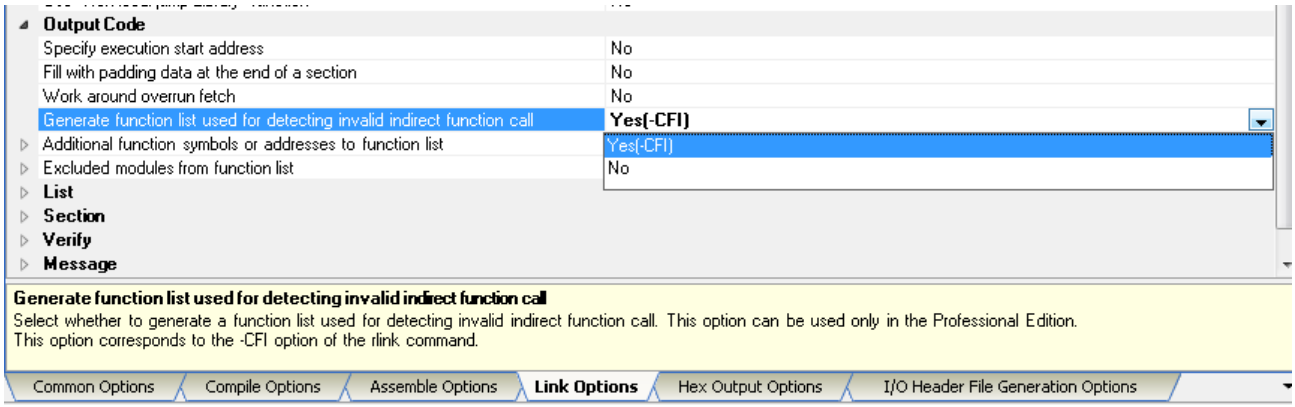



**Figure 7-6 [Link Options] Tabbed Page**

**For the e² studio:**

Open the Property dialog box by selecting [Properties] from the [Project] menu. Select [C/C++ Build] -> [Settings] and activate or deactivate [Generate an incorrect indirect function call detection code] for [Miscellaneous] under [Compiler] on the [Tool Settings] tabbed page.

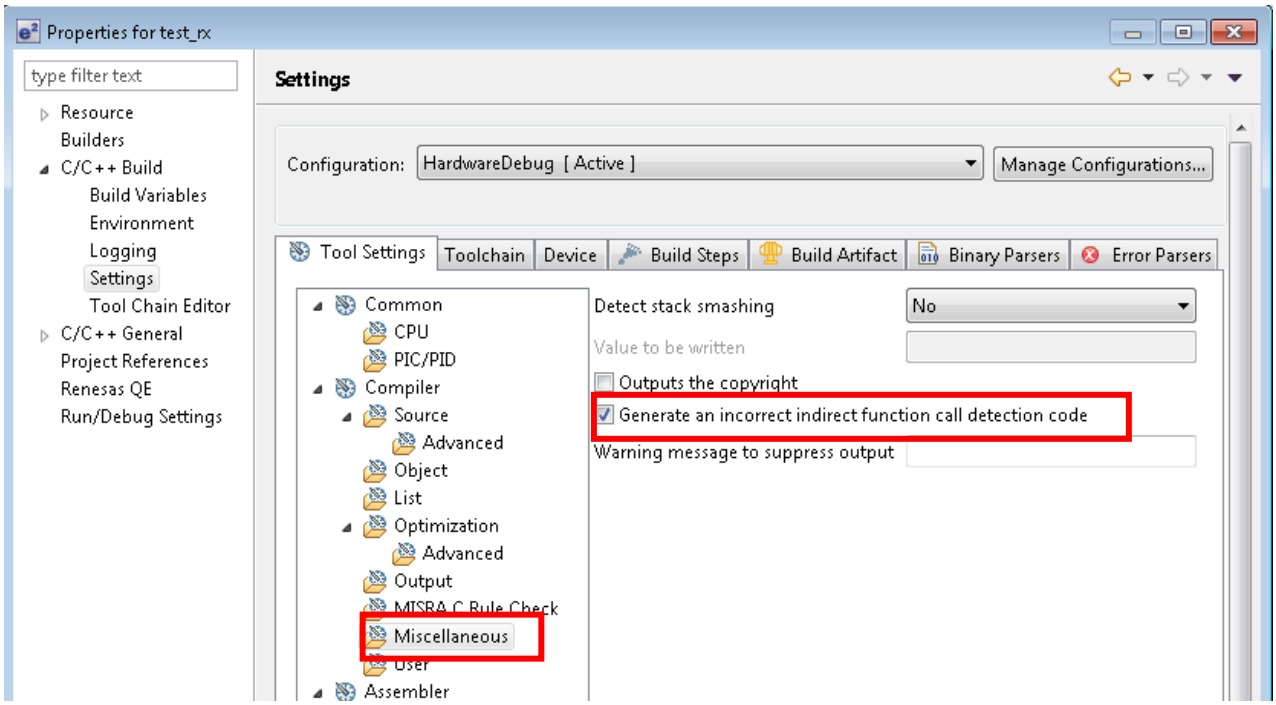



**Figure 7-7 Specifying the Option in the e² studio**

## 7.4 Example of C Source Code

The following gives an example of source code generated with this feature enabled.

```
1:  #include <string.h>
2:  #define MAX 100
3:
4:  void __control_flow_chk_fail(void) // Definition of the error-handling function
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(); // Indirect function call
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); // Passed through the stack
32: }
```

func3() causes the buffer to overflow and modifies the value of the parameter “pf” in the stack frame as shown in figure 7-8. As a result, execution branches to an illicit address other than that intended at line 16.

When this feature is enabled, the checking function “__control_flow_integrity” is called with the value of “pf” passed as an argument immediately before line 16. As the value of “pf” is not found in the list of safe function addresses generated through the setting of the option “-cfi”, the error-handling function “__control_flow_chk_fail” from line 4 is called. The user defines the “__control_flow_chk_fail” function; write the desired processing to be done when an illicit indirect function call is detected.

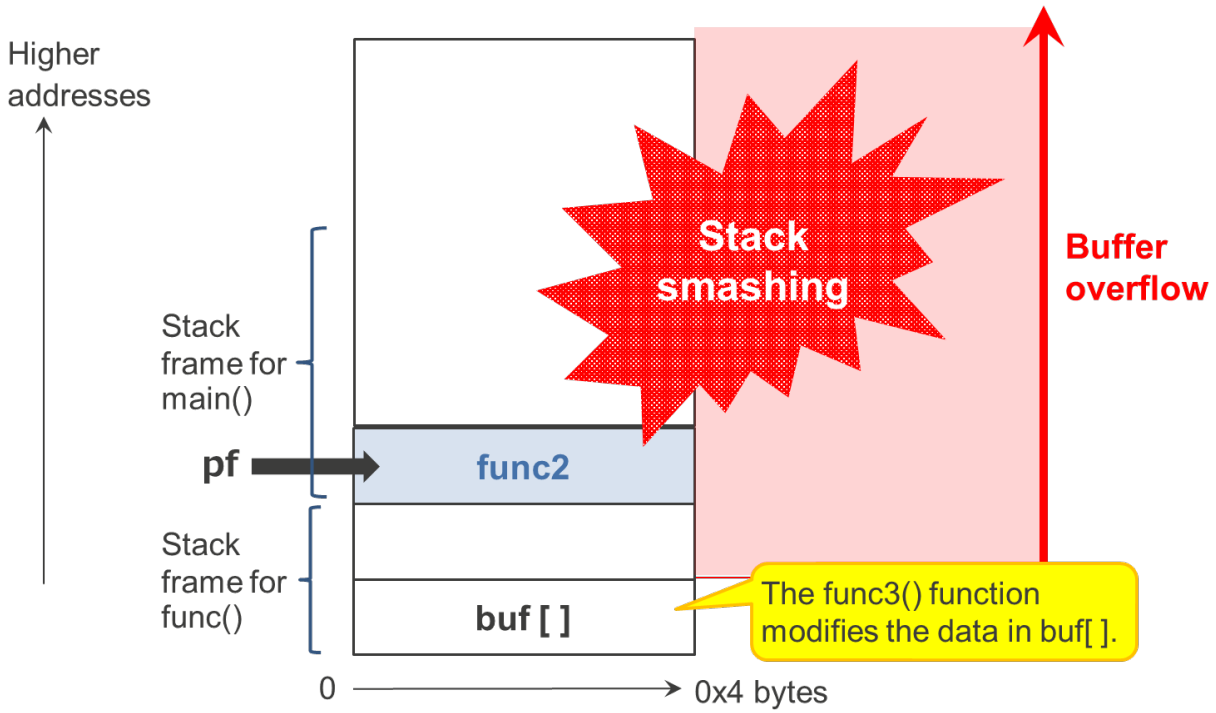



**Figure 7-8 Image of the Stack**

## Revision History

| Rev. | Date | Description | |
|---|---|---|---|
| | | Page | Summary |
| 1.00 | Jun 12, 2017 | — | First edition issued |
| 1.01 | Sep 12, 2017 | 2 | The title “Features of the Professional Editions” was added to table 1-1. |
| | | 4 | The number of rules in table 2-2 was modified according to the latest revision. |
| | | 14<br>15 | In section 3.4, Examples of C Source Code, examples of C source code were extended and error processing “__stack_chk_fail()” was added. |
| | | 20<br>22 | In section 4.4, Examples of C Source Code, examples of C source code were extended and error processing “__heap_chk_fail()” was added. |
| | | 28-34 | Section 6, Synchronization Features in the Updating of Control Registers, was newly added. |
| 1.02 | May 07, 2018 | 2 | The title “Features of the Professional Editions” was added to table 1-1. |
| | | 4 | The number of rules in table 2-2 was modified according to the latest revision. |
| | | 35-43 | Section 7, Detection of Illicit Indirect Function Calls, was newly added. |
| 1.03 | Mar. 01, 2019 | 5 | The revision numbers of the compilers in tables 2-1 and 2-2 were updated. The numbers of the required rules and the total number of the rules in table 2-2 were modified. |
| | | 29 | Figure 6-1 was added. |
| | | 37 | Section 6.5, Supplementary Items, was newly added. |
| 1.04 | Mar. 24, 2020 | 3 | In section 1.1, the name of “Upgrade license” was changed to “Upgrade (edition) license” and the related statement was modified. |
| | | 5 | The revision numbers of the compilers in tables 2-1 and 2-2 were updated. The numbers of the required rules, advisory rules, and the total number of rules in table 2-2 were modified. |
| 1.05 | Mar. 15, 2021 | 4 | Figure 1-1 and 1-2 were added. |
| | | 5 | The revision numbers of the compilers in tables 2-1 and 2-2 were updated. |
| | | 6 | The option “-misra_intermodule” was added to table 2-3.<br>Figure 2-1 and 2-2 were updated. |
| | | 12 | The __stack_chk_fail function was added to figure 3-3. |
| | | 18 | In figure 4-3, the processing flow was revised and the __heap_chk_fail function was added. |
| | | 25 | Figure 5-2 was updated. |

# Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

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

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

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

(Rev.5.0-1 October 2020)

## Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

## Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

## Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.