Microcontroller Technical Information

CC78K0 78K0 C Compiler Usage Restrictions	Document No.	ZMT-F35-11-0001		1/2
	Date issued	April 7, 2011		
	Issued by	MCU Tool Product Marketing Department MCU Software Division MCU Business Unit Renesas Electronics Corporation		
Related documents CC78K0 Ver. 3.70 Language: U17200EJ1V0UM00 CC78K0 Ver. 3.70 Operation: U17201EJ1V0UM00 78K0 C Compiler CC78K0 Ver. 4.10 Operating Precautions: ZUD-CD-10-0202	Notification classification	√	Usage restriction	
			Upgrade	
			Document modification	
			Other notification	

1. Affected products

CC78K0 V4.10 or earlier (For details, see *1. Product History* on the Attachment.)

2. New restrictions

The following restrictions have been added. See the attachment for details.

- No. 75 Macro expansion using the ## operator results in an error
- No. 76 Symbol information for an interrupt function is not output to the assembler source

3. Workarounds

The following workarounds are available for these restrictions. See the attachment for details.

- No. 75 Do either of the following:
  1. Do not use the ## operator.
  2. Place a function-like macro parameter immediately after the ## operator.
- No. 76 Define the interrupt function or output the object module file.

4. Modification schedule

No. 75 and No. 76 will be removed in the next revision.

5. List of restrictions

A list of restrictions in the CC78K0, including the revision history and detailed information, is described on the attachment.

6. Revision history

CC78K0 78K0 C Compiler Usage Restrictions Revision History

Document Number	Date Issued	Description
SBG-DT-03-0307	December 17, 2003	Newly created.
SBG-DT-04-0110	March 12, 2004	Addition of restrictions No. 17 to No. 19
ZBG-CD-04-0072	October 4, 2004	Addition of restrictions No. 20 to No. 24
ZBG-CD-05-0001	January 12, 2005	Addition of restriction No. 25
ZBG-CD-05-0054	August 31, 2005	Addition of conditions for restriction No. 21 Addition of restrictions No. 26 to No. 54
ZBG-CD-07-0038	June 28, 2007	Addition of restrictions No. 55 to No. 64
ZBG-CD-10-0024	August 16, 2010	Addition of restrictions No. 65 to No. 74
ZMT-F35-11-0001	April 7, 2011	Addition of restrictions No. 75 and No. 76

List of Restrictions in CC78K0

1. Product History

No.	Bugs and Changes/Addition to Specifications	Version				
		3.50	3.60	3.70	4.00	4.10
1	If a character string includes a null character, the character string following the null character is invalid.	○	○	○	○	○
2	Invalid code might be output if the address of a function allocated to the flash memory is referenced.	○	○	○	○	○
3	An `#if` constant expression might not be processed correctly.	○	○	○	○	○
4	If floating point values are read before integers by using `scanf` or `sscanf`, the integers cannot be input correctly.	○	○	○	○	○
5	Invalid code might be output if a structure pointer is assigned to a register and data is passed between members pointed to by the pointer.	○	○	○	○	○
6	Invalid code might be output if a negative floating point constant is type-cast to an `unsigned int`.	○	○	○	○	○
7	Invalid code might be output if a logical OR or AND operation is performed on floating point constants.	○	○	○	○	○
8	Invalid code might be output if an array declared as `register` is used as a function parameter.	○	○	○	○	○
9	If variables that have the same name are declared in multiple files while `#pragma section` is used, the variables might not be allocated to the correct section.	○	○	○	○	○
10	Invalid code might be output if a logical OR or AND operation is performed on a floating point constant and integer constant.	○	○	○	○	○
11	Initializing an external variable declared as `extern` in a block does not cause an error. In addition, the debugging information output to the assembler source is incorrect.	×	×	×	○	○
12	Linking a variable that has the same name to a variable declared as `extern` in a block might be invalid.	×	×	×	×	○
13	If a data type defined by using `typedef` (`typedef` *name*) is specified in a function prototype declaration or a declaration using a `const` or `volatile` modifier, the `typedef` expansion is invalid, and an error might result.	×	×	×	○	○
14	A multidimensional array that has an undefined size might not work properly.	×	×	×	×	○
15	If a function whose parameters cannot be referenced is referenced, no error occurs, but invalid code is output.	×	×	○	○	○
16	The `signed` type bit field is handled as an unsigned bit field.	×	×	×	×	×
17	Information of the startup routine and standard library of an older version of the CC78K0 might not be passed on to the new file.	×	○	○	○	○
18	Invalid code might be output as the result of a simple assignment operation in which the left and right operands are type-cast to a pointer for a structure, union, or array.	×*	○	○	○	○
19	Invalid code might be output as the result of the `memcpy` function if `#pragma inline` is specified.	×	○	○	○	○

×: Applicable, ○: Not applicable, –: Not relevant, *: Check tool available

No.	Bugs and Changes/Addition to Specifications	Version				
		3.50	3.60	3.70	4.00	4.10
20	Invalid code might be output as the result of an operation that uses a `signed char` and constant.	×	×*	○	○	○
21	Invalid code might be output as the result of an operation for a (`signed`) `int` and an `unsigned short`.	×	×*	○	○	○
22	If a conditional expression is followed by a simple assignment expression, debugging information is output to an invalid position.	×	×	○	○	○
23	The stack information of the `cprep2` function is invalid.	×	×	○	○	○
24	W0503 is output if the array name of an automatic variable is referenced.	×	×	×	×	○
25	If a static variable is initialized to a floating point constant, the initial value becomes invalid.	×	×*	○	○	○
26	Invalid code might be output if a function pointer is referenced using an asterisk (*) in code other than a function call.	×	×*	○	○	○
27	If the `-QR` option is specified, an `sreg` variable might be redundantly allocated to the `saddr` area used by the compiler.	×	×*	○	○	○
28	Invalid code might be output if the number of bits shifted by a shift operator (`<<`, `>>`, `<<=`, or `>>=`) is a constant that is 256 or larger.	×	×*	○	○	○
29	Invalid code might be output if the constant `0xffff` or `0xfffe` is added to or subtracted from a `long` or `unsigned long`.	×	×*	○	○	○
30	An error is output if a floating point number starting with `0e` or `0E` is specified.	×	×	○	○	○
31	Invalid code might be output as the result of an expression in which the result of performing a runtime library operation on a `long` is type-cast to a `signed int`, `unsigned int`, `signed short`, or `unsigned short`.	×	×*	○	○	○
32	Invalid code might be output for one side of a binary operation.	×	×*	○	○	○
33	If both operands of a logical operation (`&&` or `||`) are of the floating point type, and an expression that causes a side effect such as an increment/decrement operation or a function call is specified as the second operand, the comparison order becomes invalid.	×	×*	○	○	○
34	An error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators that have the same priority are specified in succession, is specified as an `#if` constant expression.	×	×	○	○	○
35	If one operand of a relational operation is a constant that cannot be expressed by `signed long`, the comparison result might be invalid.	×	×*	○	○	○
36	The operation result is not output as the `int` type but as the operand's type, depending on the types of the operands in logical negation operations, relational operations, or equality operations.	×	×*	○	○	○
37	An error is output if an increment or decrement expression of a floating point type is specified and the operand is a dereferenced pointer.	×	×	○	○	○

×: Applicable, ○: Not applicable, –: Not relevant, *: Check tool available

No.	Bugs and Changes/Addition to Specifications	Version				
		3.50	3.60	3.70	4.00	4.10
38	Invalid code might be output by an expression in which an expression of the format “address of `char` or `unsigned char` array + `unsigned char`” is type-cast to a pointer for an `int`, `unsigned int`, `short`, or `unsigned short`, and 0 is assigned to the value referenced by the pointer.	×	×*	○	○	○
39	Invalid code might be output if a `char` or `unsigned char` expression is specified in a `return` statement for a function that returns a pointer.	×	×*	○	○	○
40	An error might be output if the result of performing a runtime library operation on a `long` is type-cast to a `char` or `unsigned char` and a relational or equality operation is performed with a constant that can be expressed by `char` or `unsigned char`.	×	×	○	○	○
41	When initializing an array whose size is not defined, if elements in the initializer braces are enclosed inconsistently, an area of invalid size is allocated.	×	×	×	×	○
42	If a character string conversion function in the standard library is executed, error handling becomes invalid.	×	×	×	○	○
43	The operation becomes invalid when output conversion is specified by an I/O function in the standard library.	×	×	×	×	×
44	The incorrect size is output for the minimum value –32768 of an `int` or `short`.	×	×	×	×	×
45	If a function name or a function pointer is specified as the second and third operands of a conditional operation, an error is output, and then the function is called.	×	×	×	×	○
46	An error is output if an external pointer is initialized to a variable containing the operator `->`.	×	×	○	○	○
47	Error due to a mismatch between the parameter type and the identifier type in a function definition.	×	×	×	×	×
48	A parameter whose type is not declared in the identifier list in a function definition is not handled as an `int`, which causes an error.	×	×	×	×	×
49	The `#` operator cannot be used to expand a macro to a string literal.	×	×	×	×	×
50	If an `unsigned long` static variable is initialized to a floating point constant that is 0x80000000 or larger, the initial value becomes invalid.	×	×*	○	○	○
51	Invalid code might be output for an expression that includes a function call and a `struct` or `union`.	×	×*	○	○	○
52	Invalid code might be output for the assignment expression **`a = b`** *binary operator* **`c;`**.	×	×*	○	○	○
53	An error is output if an element of an array member of a structure at a constant address is referenced by using the dot operator (`.`).	×	×	○	○	○
54	An error is output for a function definition that has a certain pattern.	×	×	○	○	○
55	Invalid code might be output if a 1-byte static array element, which uses an `unsigned char` value returned by a function as an array subscript, is specified for the code of the assignment destination of a compound assignment, increment, or decrement operation.	×	×*	×*	○	○

×: Applicable, ○: Not applicable, –: Not relevant, *: Check tool available

No.	Bugs and Changes/Addition to Specifications	Version				
		3.50	3.60	3.70	4.00	4.10
56	A constant expression that includes multiple binary operators, which use the result of a binary operator causing an overflow, might be replaced with invalid values.	×	×*	×*	○	○
57	Invalid code might be output as the result of an operation that includes an increment or decrement operation.	×	×*	×*	○	○
58	The result of a `seizeof` operation for a function parameter that has an array might be invalid.	×	×*	×*	○	○
59	Invalid code might be output as the result of a compound assignment operation of a division or remainder operation if the `-qc` option is not specified.	×	×*	×*	○	○
60	Invalid code might be output when there is a bit field where the bit width assigned to a `saddr` area is from 2 bits to 7 bits, and the maximum constant value of the bit field is assigned to an expression, the assignment destination being re-evaluated with the same expression.	×	×*	×*	○	○
61	The error E0705 or invalid code might be output if a `norec` function that has `enum` parameters is called.	×	×*	×*	○	○
62	No error occurs if an identical function whose parameters include a different structure or union is declared multiple times.	×	×*	×*	○	○
63	Invalid code might be output if a pointer that points to a structure of 256 bytes or more is a `register` variable.	×	×*	×*	○	○
64	Work areas used by the compiler are corrupted if the static model option `-sm` and the expansion specification option `-zm2` are specified.	×	×	×	○	○
65	Invalid code might be output when referencing 1-byte data pointed to by a pointer to which `++` or `--` has been suffixed.	×	×*	×*	×*	○
66	Invalid code is output if the last element of an initializer list for a `char`, `signed char`, or `unsigned char` array is a character string and one or more constants or character constants are placed before the character string.	×	×*	×*	×*	○
67	Invalid code is output if referencing a pointer that has an offset obtained by subtracting one pointer from another.	×	×*	×*	×*	○
68	Invalid code is output if the `-qc` option has not been specified (sign extension is specified to be `int` type).	○	○	×*	×*	○
69	Invalid code is output as the result of the BCD operation function `adbcdw` or `sbbcdw`.	×	×	×	×	○
70	An error occurs if the `-ng` option is specified and a branch instruction is used in a function that includes the `asm` statement.	×	×	×	×	○
71	The line number is not output for a statement that immediately follows a nested `if` statement and is outside that statement's conditional block.	×	×	×	×	○
72	Invalid code is output when a value is assigned to a `long` variable in an interrupt function.	×	×*	×*	×*	○
73	Bank function calling code might not be output.	–	–	×*	×*	○
74	Invalid code might be output if using a 1-byte parameter or `auto` variable for a `norec` function.	×	×*	×*	×*	○

×: Applicable, ○: Not applicable, –: Not relevant, *: Check tool available

No.	Bugs and Changes/Addition to Specifications	Version				
		3.50	3.60	3.70	4.00	4.10
75	Macro expansion using the ## operator results in an error	×	×	×	×	×
76	Symbol information for an interrupt function is not output to the assembler source	×	×	×	×	×

×: Applicable, ○: Not applicable, –: Not relevant, *: Check tool available

2. Details of Usage Restrictions

No. 1 If a character string includes a null character, the character string following the null character is invalid.

Description:

If a character string includes a null character, the character string following the null character is invalid.

Example:

```
const char str[] = "test\0TEST";
```

No memory is allocated for the character string following the null character.

Workaround:

Change the code as follows:

```
const char str[] = {'t','e','s','t','\0','T','E','S','T'};.
```

Correction:

This issue has been corrected in V3.50.

No. 2 Invalid code might be output if the address of a function allocated to the flash memory is referenced.

Description:

Invalid code might be output if the address of a function allocated to the flash memory is referenced.

Example:

```
#pragma ext_table 0x2000
#pragma ext_func func1 1
#pragma ext_func func2 2

int func1(int, int);
int func2(int, int);
int (*func_b1)(int, int) = &func1;
__sreg int (*func_b2)(int, int) = &func2;

void func()
{
     func_b1 = func1;   /* normal code */
     func_b2 = func2;   /* invalid code */
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V3.50.

No. 3 An `#if` constant expression might not be processed correctly.

Description:

An `#if` constant expression might not be processed correctly.

Example:

```
#define a
#if a
int i;
#endif

void func()
{
     i++;
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V3.50.

No. 4 If floating point values are read before integers by using `scanf` or `sscanf`, the integers cannot be input correctly.

Description:

If floating point values are read before integers by using `scanf` or `sscanf`, the integers cannot be input correctly.

Example:

```
void func()
{
    int i;
    float f;

    sscanf("1.2 10", "%f%d", &f, &i);
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V3.50.

No. 5 Invalid code might be output if a structure pointer is assigned to a register and data is passed between members pointed to by the pointer.

Description:

Invalid code might be output if a structure pointer is assigned to a register and data is passed between members pointed to by the pointer.

Example:

```
struct tag {
    int a;
    int b;
};
void func()
{
    register struct tag *sp;
    sp->b = sp->a;
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V3.50.

No. 6 Invalid code might be output if a negative floating point constant is type-cast to an `unsigned int`.

Description:

Invalid code might be output if a negative floating point constant is type-cast to an `unsigned int`.

Example:

```
unsigned long ans;
float f1 = -3.8f;

void func()
{
    int a;
    ans = f1;

    a = ((unsigned long)(-3.8f) != ans);
}
```

Workaround:

Type-cast a constant to a `signed int` before type-casting the constant to an `unsigned int`.

Example:

```
a = ((unsigned long)(long)(-3.8f) != ans);
```

Correction:

This issue has been corrected in V3.50.

No. 7 Invalid code might be output if a logical OR or AND operation is performed on floating point constants.

Description:

Invalid code might be output if a logical OR or AND operation is performed on floating point constants.

Example:

```
void func()
{
    int r1, r2;
    float f1 = 17, f2 = 16;

    r1 = f1 || f2;
    r2 = f1 && f2;
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V3.50.

No. 8 Invalid code might be output if an array declared as `register` is used as a function parameter.

Description:

Invalid code might be output if an array declared as `register` is used as a function parameter.

Example:

```
void func(register char a[])
{
    register char *p;
    p = (char *)a;
}
```

Workaround:

Specify the parameter as a pointer.

Example:

```
void func(register char *a)
{
     register char *p;
     p = a;
}
```

Correction:

This issue has been corrected in V3.50.

No. 9 If variables that have the same name are declared in multiple files while `#pragma section` is used, the variables might not be allocated to the correct section.

Description:

If variables that have the same name are declared in multiple files while `#pragma section` is used, the variables might not be allocated to the correct section.

Example:

```
--- a.c ---
#include "a1.h"
#include "a2.h"
#include "a3.h"

--- a1.h ---
#pragma section @@DATA DAT1
int a;
#pragma section @@DATA DAT2

--- a2.h ---
#pragma section @@DATA DAT3
int b;
#pragma section @@DATA DAT4

--- a3.h ---
#pragma section @@DATA DAT5
extern int a;    /* same when int a;  */
#pragma section @@DATA DAT6
```

Workaround:

Do not declare variables that have the same name in multiple files if `#pragma section` is used.

Correction:

This issue has been corrected in V3.50.

No. 10 Invalid code might be output if a logical OR or AND operation is performed on a floating point constant and integer constant.

Description:

Invalid code might be output if a logical OR or AND operation is performed on a floating point constant and integer constant.

Example:

```
void func()
{
int rval;
int cZero = 0;

rval = 0.0F || cZero;  /* invalid code in Windows version */
rval = cZero || 0.0F;  /* invalid code in UNIX version */
}
```

Workaround:

Do not specify a floating point constant for the logical OR or AND operation.

Correction:

This issue has been corrected in V3.50.

No. 11 Initializing an external variable declared as `extern` in a block does not cause an error. In addition, the debugging information output to the assembler source is incorrect.

Description:

Because initializing an external variable declared as `extern` in a block is not compliant with the ANSI C specifications, it should cause an error, but the code does not. The compiler interprets the code as defining an external variable that has an initial value and outputs the code.

The compiler outputs the correct debugging information to the object file, but the debugging information output to the assembler source is incorrect.

Example:

```
int  i;
void  f(void) {
     extern int  i = 2;
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V4.00.

No. 12 Linking a variable that has the same name to a variable declared as `extern` in a block might be invalid.

Description:

Linking a variable that has the same name to a variable declared as `extern` in a block is invalid in any of the following cases:

(1) If a variable declared as `extern` in a block and a variable declared as `static` outside that block or the subsequent blocks have the same name

No error occurs and linking is not performed, so, if this variable is referenced, invalid code is output.

Example:

```
void  f(void) {
        extern int  i;
        i = 1;              /* Invalid code is output */
}
static int  i;
```

(2) If a variable declared as `extern` in a block and a variable not declared as `static` outside that block or the subsequent blocks have the same name

Linking is not performed and invalid code is output.

Example:

```
void  f(void) {
        extern int  i;
        i = 1;              /* Invalid code is output */
}
 int  i;
```

(3) If a variable declared as `extern` in a block and a variable not declared as `extern` outside the block before the `extern` variable have the same name, and an automatic variable that has the same name is declared in the block containing the `extern` variable

The variable outside the block and the variable declared as `extern` in the block are not linked, and invalid code is output.

Example:

```
int i = 1;
void  f(void) {
        int  i;
        {
               extern int  i;
               i = 1;       /* Invalid code is output */
        }
}
```

(4) If a variable declared as `extern` in a block and a variable declared as `extern` in another block have the same name

Linking is not performed, and invalid code is output.

Example:

```
void  f1(void) {
         extern int  i;
         i = 2;
}
void  f2(void){
         extern int  i;
         i = 3;
}
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V4.10.

No. 13 If a data type defined by using `typedef` (`typedef` *name*) is specified in a function prototype declaration or a declaration using a `const` or `volatile` modifier, the `typedef` expansion is invalid, and an error might result.

Description:

If a data type defined by `typedef` (`typedef` *name*) is specified in a function prototype declaration or a declaration using a `const` or `volatile` modifier, the `typedef` expansion is invalid, and an error might result.

Example 1:

```
typedef int  FTYPE();

FTYPE  func;
int  func(void);                /* E0713 Redefined 'func' */
```

Example 2:

```
typedef int  VTYPE[2];
typedef int  *VPTYPE[3];

const VTYPE  *a;
```

```
const int  (*a)[2];             /* E0713 Redefined 'a' */
volatile VPTYPE  b[2];
volatile int *volatile  b[2][3];    /* E0713 Redefined 'b' */
```

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V4.00.

No. 14 A multidimensional array that has an undefined size might not work properly.

Description:

A multidimensional array that has an undefined size might not work properly.

Example 1:

```
char  c[][3]={{1},2,3,4,5};     /* Invalid code */
```

Example 2:

```
char  c[][2][3]={"ab","cd","ef"};   /* Error (E0756) */
```

Workaround:

Define the size of the multidimensional array.

Correction:

This issue has been corrected in V4.10.

No. 15 If a function whose parameters cannot be referenced is referenced, no error occurs, but invalid code is output.

Description:

For a function returning the address of a function that has parameters, those parameters cannot be referenced. If referencing is attempted, no error occurs, but invalid code is output.

Example:

```
char *c;
int  *i;
void (*f1(int *))(char *);
void (*f2(void))(char *);
void (*f3(int *))(void);

void main() {
        (*f1(i))(c);        /* Correct code (W0510) */
        (*f1(i))(i);        /* Incorrect code */
        (*f2())(c);         /* Correct code (W0509) */
        (*f2())();          /* Incorrect code (W0509) */
        (*f3(i))();         /* Correct code (W0509) */
        (*f3(i))(i);        /* Incorrect code */
}
```

The warning W0509 or W0510 is output for correct code. Nothing is output for code that should produce a warning. However, the output code is normal.

```
void (*f4())(int p) {
       p++;                 /* Incorrect code */
}
```

No error is output for code that should cause an error and invalid code is generated.

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V3.70.

No. 16 The `signed` type bit field is handled as an unsigned bit field.

Description:

The `signed` type bit field is handled as an unsigned bit field.

Workaround:

There is no workaround.

Correction:

This will not be corrected, so regard it as a specification.

No. 17 Information of the startup routine and standard library of an older version of the CC78K0 might not be passed on to the new file.

Description:

If a project file created using an older version of the CC78K0 is loaded, information of the startup routine and standard library might not be passed on to the new file.

Workaround:

Set up the startup routine and standard library again.

Correction:

This issue has been corrected in V3.60.

No. 18 Invalid code might be output as the result of a simple assignment operation in which the left and right operands are type-cast to a pointer for a structure, union, or array.

Description:

Invalid code might be output if all the following conditions are satisfied:

(1) Simple assignment operation
(2) The left and right operands are both dereferenced data that uses a structure, union, array, or pointer.
(3) The left and right operands both reference a symbol address.
(4) The operands are 4-byte data (`long`, `float`, `double`, or `long double`).

Example:

```
typedef union {
unsigned long l;
} uni;
unsigned int a[10], b[10];

void func(void)
{
((uni*)(&b)) -> l = ((uni*)(&a)) -> l;
}
```

Workaround:

Modify the program so that the operation is processed using a temporary variable.

Example 1:

```
uni *tmp1, *tmp2;
tmp1 = (uni*)&a;
tmp2 = (uni*)&b;
tmp2->l = tmp1->l;
```

Example 2:

```
long tmp3;
tmp3 = ((uni*)(&a))->l;
((uni*)(&b))->l = tmp3;
```

Correction:

This issue has been corrected in V3.60.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 19 Invalid code might be output as the result of the `memcpy` function if `#pragma inline` is specified.

Description:

Invalid code might be output if all the following conditions are satisfied:

(1) `#pragma inline` is specified.

(2) The `memcpy` function is used.

(3) The third parameter of the `memcpy` function in (2) is not a constant and uses the HL register.

Example:

```
#pragma inline
int s[100];
void func(void)
{
int *t;
int u;
memcpy( s, t, u );
}
```

Workaround:

Do either of the following:

(1) Do not specify `#pragma inline` if the `memcpy` function is used.

(2) If `#pragma inline` is specified, specify a constant for the third parameter of the `memcpy` function.

Correction:

This issue has been corrected in V3.60.

No. 20 Invalid code might be output as the result of an operation that uses a `signed char` and constant.

Description:

Invalid code might be output if one of the conditions (1) to (5) is satisfied when the `-QC1` option is specified, or if condition *6* is satisfied when the `-QC1` or `-QC2` option is specified.

Conditions:

(1) The right-shift operator `>>` is used, the left operand is a constant from 128 to 255, and the right operand is a `signed char`.

(2) The operator `/`, `%`, `<`, `<=`, `>`, `>=`, `/=`, or `%=` is used, either the left or right operand is a constant from 128 to 255, and the other operand is a `signed char`.

(3) A binary operator operation is performed, in which either the left or right operand is a constant from 128 to 255 and the other operand is a `signed char`, and the obtained result is converted to a data type that is longer than 2 bytes.

(4) A binary operator operation is performed, in which either the left or right operand is a constant from 128 to 255 and the other operand is a `signed char`, the obtained result is used as the left operand for the right-shift operator `>>`, and the operator has a `signed char` as the right operand.

(5) A binary operator operation is performed, in which either the left or right operand is a constant from 128 to 255 and the other operand is a `signed char`, the obtained result is used as the operand on either side of the operator `/`, `%`, `<`, `<=`, `>`, `>=`, `/=`, or `%=`, and the operator has a `signed char` on the other side.

(6) A binary operator operation is performed, in which either the left or right operand is a constant that uses at least one of the operators `<<`, `>>`, `&`, `^`, or `|`, the constant is from –128 to 255, the obtained result is from –128 to –1, and the data type of the other operand is longer than 2 bytes.

Example 1:

```
signed char a;
if(a < 179/2){b++;}
```

Example 2:

```
int i, j;
i = j & (-127 | 4);
```

Workaround:

Type-cast the relevant constant to a `signed char`.

Example 1:

```
if(a < (signed int)179/2){b++;}
```

Example 2:

```
i = j & ((signed int)-127 | 4);;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 21 Invalid code might be output as the result of an operation for a (signed) int and an unsigned short.

Description:

If either condition below is satisfied, a signed operation is performed for an unsigned operation, which might cause an invalid result.

Condition 1: In a relational operation (<, >, <=, or >=), division operation (/), remainder operation (%), or compound assignment operation (%= or /=), the data type on one side is a (signed) int and that on the other side is an unsigned short.

Example 1:

```
unsigned short us1, us2;
void func1()
{
     us1 /= 0x5555;
     us2 %= 0x5555;
}
```

Example 2:

```
unsigned short us1, us2;
signed int si1;
void func2()
{
     us2 = us1 / si1;
}
```

Example 3:

```
int si, x;
unsigned short us;
void func3()
{
     if (si > us) x++;
}
```

Condition 2: In a compound assignment operation (%= or /=) or binary operation, the data type on one side is a (signed) int, that on the other side is an unsigned short, and the data type of the result is converted to a data type of 2 bytes or longer.

Example:

```
long l;
unsigned short us;
signed int si;
void func4()
{
     l = us + si;
}
```

Workaround:

Change the data type from `unsigned short` to `unsigned int`.

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 22 If a conditional expression is followed by a simple assignment expression, debugging information is output to an invalid position.

Description:

If a static 1-byte variable that cannot be assigned to the `saddr` area is referenced by a conditional expression and a constant is assigned to the same 1-byte variable in the simple assignment expression immediately after the conditional expression, debugging information is output to an invalid position. This does not affect the output code itself.

Example:

C Source Code	Corresponding Assembler Output

`if(TEST >= 10)`
 	
`{`
`  TEST = 0;`
 	
`}`
	
	`movw de,#_TEST`
	`mov a,[de]`
	`cmp a,#0AH ; 10`
	
	`bc $?L0003  ;(1)` ← A comparison instruction is output to the location for an assignment statement.
	`callt [@@clra0]`
	`mov [de],a`
	`br $?L0003`

Even if a breakpoint is specified at the position of `TEST = 0;` by using the debugger, the break occurs at (1) in the above program. That is, the break does not occur if the `if` statement is evaluated to be true, but occurs at the conditional expression.

Workaround:

Specify a breakpoint at a relevant location in the assembler code, by using the mixed display in the **Source** window of the debugger.

Correction:

This issue has been corrected in V3.70.

No. 23 The stack information of the `cprep2` function is invalid.

Description:

In the function information output by the compiler, usually the amount of the stack used by a run-time library function is added to the stack information of a function, including the run-time library functions.

If the program includes the `cprep2` function, however, 2 bytes too many are added.

Workaround:

There is no workaround.

If the program includes the `cprep2` function, subtract 2 bytes from the amount of the stack used.

Correction:

This issue has been corrected in V3.70.

No. 24 W0503 is output if the array name of an automatic variable is referenced.

Description:

W0503 is output if the array name of an uninitialized automatic variable is referenced.

```
W0503  Possible use of 'variable-name' before definition
```

Note:

Initialization refers to the assignment of a value to a variable when the variable is declared, not later.

Array name refers to, for example, the *a* in *int a[2]*. Indexes, operators, and other code are not included in array names.

Example:

```
void func(void)
{
   int a[2];
   int *b;

   a[0] = 0;
   a[1] = 0;
   b = a;            /* W0503 is output to this line */
}
```

Workaround:

If W0503 is output, check the location. You can ignore the message if the variable is initialized in the statement.

Correction:

This issue has been corrected in V4.10.

No. 25 If a static variable is initialized to a floating point constant, the initial value becomes invalid.

Description:

If a variable other than the following is initialized to a floating point constant, the initial value becomes invalid:

- `long`, `unsigned long`, or a floating point variable
- `long`, `unsigned long`, or a floating point member of a structure or union
- `long`, `unsigned long`, or a floating point element of an array

Example 1:
```
signed char a1[3] = {1.0*100};
```

Example 2:
```
#define VAR 1.0*100
signed char frame02 = VAR;
```

Workaround:

Do either of the following:

(1) Initialize the static variable to an integer constant.

Specify the following for the above example 1:

```
signed char a1[3] = {100};
```

(2) Type-cast the relevant floating point constant to an integer constant.

Specify the following for the above example 1:

```
signed char a1[3] = {(signed char)(1.0*100)};
```

Specify the following for the above example 2:

```
#define VAR 1.0*100
signed char frame02 = (signed char)(VAR);
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 26 Invalid code might be output if a function pointer is referenced using an asterisk (*) in code other than a function call.

Description:

Invalid code might be output if a function pointer is referenced using an asterisk (*) in code other than a function call.

Example:

```
void (*fp)();
int x;
void func()
{
   if (*fp)  x++;
}
```

Workaround:

Do not append an asterisk (*) to a function pointer.

Example:

```
if (*fp)  x++;
↓
if (fp)  x++;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 27 If the -QR option is specified, an sreg variable might be redundantly allocated to the saddr area used by the compiler.

Description:

If the -QR option is specified, an sreg variable might be redundantly allocated to the saddr area used by the compiler.

Example:

```
#pragma interrupt INTP0 inter
__boolean b1;
void func()
{
   b1 = 1;
}
void inter()
{   func();
}
```

--- Another file ---

```
__sreg char sc[152];
```

Workaround:

Do not specify the -QR option.

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 28 Invalid code might be output if the number of bits shifted by a shift operator (<<, >>, <<=, or >>=) is a constant that is 256 or larger.

Description:

Invalid code might be output if the number of bits shifted by a shift operator (<<, >>, <<=, or >>=) is a constant that is 256 or larger.

Example:

```
int i1, i2;
char c1, c2;
void func()
{
     i1 = i2 << 257;
     i2 <<= 257;
     c1 = c2 << 257;
     c2 <<= 257;
}
```

Workaround:

Make the number of shifted bits and data width the same.

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 29 Invalid code might be output if the constant `0xffff` or `0xfffe` is added to or subtracted from a `long` or `unsigned long`.

Description:

Invalid code might be output if the constant `0xffff` or `0xfffe` is added to or subtracted from a `long` or `unsigned long`.

Example:

```
unsigned long l1, l2;
void func()
{
l1 = l2 + 0xffff;
}
```

Workaround:

Use a temporary variable as follows:

```
long ltmp = 0xffff;
l1 = l2 + ltmp;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 30 An error is output if a floating point number starting with `0e` or `0E` is specified.

Description:

An error is output if a floating point number starting with `0e` or `0E` is specified.

Example:

```
float f1;
void func()
{
f1 = 0e0;
}
```

Workaround:

Specify `0` or `0.0` at the beginning of a floating point number.

Correction:

This issue has been corrected in V3.70.

No. 31 Invalid code might be output as the result of an expression in which the result of performing a runtime library operation on a `long` is type-cast to a `signed int`, `unsigned int`, `signed short`, or `unsigned short`.

Description:

Invalid code might be output if all the following conditions are satisfied:

(1) A normal model is specified.

(2) A 2-byte equality operation (`==` or `!=`) or a 2-byte unsigned relational operation (`<` or `>=`) in which the left operand is referenced by an automatic variable, or a 2-byte unsigned relational operation (`<=` or `>`) in which the right operand is referenced by a parameter, automatic variable, or pointer.

(3) The operand on the other side is a `long` on which a runtime library operation is performed that has been type-cast to a `signed int`, `unsigned int`, `signed short`, or `unsigned short`.

Example:

```
int *ip;
long l;
void func()
{
if (*ip == (int)(l * 5)) l = 0;
}
```

Workaround:

Do either of the following:

(1) Do not type-cast the operation result:

```
if (*ip == (l * 5)) l = 0;
```

(2) Use a temporary variable as follows:

```
int tmp;
tmp = l * 5;
if (*ip == tmp) l = 0;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 32 Invalid code might be output for one side of a binary operation.

Description:

If one of the following conditions is satisfied, the result of a logical or conditional operation that has been saved might not be restored normally, which results in invalid code being output.

Condition 1: All the following conditions are satisfied:

(1) The code includes a binary operation.

(2) One operand of the binary operation specified in (1) references the result of a logical operation (`&&` or `||`) or conditional operation (`?:`).

(3) The result of the other operand of the binary operation specified in (1) remains in a register.

(4) Only the `leaf` function is used in a normal model (for the conditional operations).

Example:

```
int func1(), func2();
int x, i;
void func()
{
if (func1() == (i && func2())) x++;
} /*  (3)   (1)     (2)    Corresponding to the above number */
```

Condition 2: All the following conditions are satisfied:

(1) The code includes a binary operation.

(2) One operand of the binary operation specified in (1) references the result of a logical operation (`&&` or `||`).

(3) The result of the other operand of the binary operation specified in (1) remains in `_@RTARGx` (parameter in the run-time library function).

Example:

```
float f1, f2;
long l;
int x;
void func()
{
if (++f1 == (l && f2)) x++;
} /*  (3)   (1)     (2)    Corresponding to the above number */
```

Condition 3: All the following conditions are satisfied:

(1) The code includes a binary operation.

(2) One operand of the binary operation specified in (1) references the result of a compound assignment.

(3) The result of the other operand of the binary operation specified in (1) remains in a register.

Example:

```
int ifunc(), i;
void func()
{
int *ip1, *ip2;
i = *ip1 == (*ip2 += ifunc());
} /*  (3)   (1)     (2)    Corresponding to the above number */
```

Workaround:

Use the corresponding workaround below.

For condition 1:

Use a temporary variable for at least one operand of the binary operation.

```
int tmp;
tmp = func1(); /* Assign to a temporary variable */
if (tmp == (i && func2())) x++;
```

For condition 2:

Use a temporary variable for at least one operand of the binary operation.

```
float tmp;
tmp = ++f1;  /* Assign to a temporary variable */
if (tmp == (l && f2)) x++;
```

For condition 3:

Use a temporary variable for at least one operand of the binary operation.

```
int tmp;
tmp = *ip1;               /* Assign to a temporary variable */
i = tmp == (*ip2 += ifunc());
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 33 If both operands of a logical operation (`&&` or `||`) are of the floating point type, and an expression that causes a side effect such as an increment/decrement operation or a function call is specified as the second operand, the comparison order becomes invalid.

Description:

If both operands of a logical operation (`&&` or `||`) are of the floating point type, and an expression that causes a side effect such as an increment/decrement operation or a function call is specified as the second operand, the comparison order becomes invalid.

Example:

```
int x;
float f1, f2;
void func()
{
  if (f1 || f2++) {
        x = 1;
  }
  else {
        x = 2;
  }
}
```

Remark Incorrect output: `f2++` is executed for `if(f1 || f2++)` regardless of whether `f1` is true or false.

Correct output: `f2++` is executed for `if(f1 || f2++)` only if `f1` is false.

Workaround:

Modify the source code as follows:

```
if (f1) {
        x = 1;
}
else if (f2++) {
        x = 1;
}
else {
        x = 2;
}
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 34 An error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators that have the same priority are specified in succession, is specified as an `#if` constant expression.

Description:

The E0501 error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators that have the same priority are specified in succession, is specified as an `#if` constant expression.

Example:

```
#if !~0
int i;
#endif
#if -1
int j;
#endif
```

Workaround:

Use parentheses as follows:

```
#if !~0      #if -1
   ↓           ↓
#if !(~0)   #if (-1)
```

Correction:

This issue has been corrected in V3.70.

No. 35 If one operand of a relational operation is a constant that cannot be expressed by `signed long`, the comparison result might be invalid.

Description:

Invalid code might be output if one operand of a relational operation is a constant that cannot be expressed by `signed long`, and either of the following conditions is satisfied:

Condition 1: The other operand can be handled as a constant, and expressed by `signed long`.

Example:

```
int x1,x2,x3,x4,x5,x6, i;
void func1()
{
      x1 = (i << 31) < 0x80000001;
      x2 = (i * 0) > 0x90000020;
      x3 = (i % 1) <= 0xa0000300;
      x4 = (i & 0) >= 0xb0004000;
      x5 = (i | 0xffff) < 0xc0050000;
      x6 = (i++, 300) > 0xffffffff;
}
```

Condition 2: One operand is a constant that is 0xffffff80 or larger, and the other operand is a `boolean`, `bit`, or `signed char`.

Example:

```
int x1, x2, x3, i;
char cfunc(), c1, c2;
void func2()
{
      x1 = c1 < 0xffffff80;
      x2 = cfunc() > 0xffffff91;
      x3 = (c1 + c2) <= 0xfffffffa2;
}
```

Workaround:

Workaround for condition 1:

Specify expressions that can be handled as constants by using constants.

```
x1 = (i << 31) < 0x80000001;     →  x1 = 0 < 0x80000001;
x2 = (i * 0) > 0x90000020;       →  x2 = 0 > 0x90000020;
x3 = (i % 1) <= 0xa0000300;      →  x3 = 0 <= 0xa0000300;
x4 = (i & 0) >= 0xb0004000;      →  x4 = 0 >= 0xb0004000;
x5 = (i | 0xffff) < 0xc0050000;  →  x5 = 0xffff < 0xc0050000;

x6 = (i++, 300) > 0xffffffff;    →  i++;
                                    x6 = 300 > 0xd0600000;
```

Workaround for condition 2:

Type-cast `boolean`, `bit`, or `signed char` expressions to `unsigned long`.

```
x1 = c1 < 0xffffff80;          →  x1 = (unsigned long)c1 < 0xffffff80;
x2 = cfunc() > 0xffffff91;     →  x2 = (unsigned long)cfunc() > 0xffffff91;
x3 = (c1 + c2) <= 0xfffffffa2; →  x3 = (unsigned long)(c1 + c2) <= 0xfffffffa2;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 36 The operation result is not output as the `int` type but as the operand's type, depending on the types of the operands in logical negation operations, relational operations, or equality operations.

Description:

Either of the following operations occurs:

(1) If the type of a logical negation operation, relational operation, or equality operation is any of the following, the result is not output as the `int` type but as the operand's type:

- `unsigned int`, `unsigned short`, pointer (2-byte pointer), array
- `unsigned char` with `-QC` specified

(2) If the right operand of the logical operator `&&` is the floating point constant `±0.0`, the result becomes the floating point type instead of the `int` type (unless the static model is used).

Example:

```
unsigned int ui1, ui2;
int xi1, xi2, xi3, i1;
void func()
{
     xi1 = !ui1 > i1;
     xi2 = (ui1 == ui2) > i1;
     xi3 = (ui1 && 0.0) > i1;
}
```

Workaround:

(1) Type-cast the result of a logical negation operation, relational operation, or equality operation to an `int`.

```
xi1 = !ui1 > i1;
xi2 = (ui1 == ui2) > i1;
    ↓
xi1 = (int)!ui1 > i1;
xi2 = (int)(ui1 == ui2) > i1;
```

(2) Do not use the floating point constant `±0.0` as the right operand of the logical operator `&&`.

```
xi3 = (ui1 && 0.0) > i1;
    ↓
xi3 = (ui1 && 0) > i1;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 37 An error is output if an increment or decrement expression of a floating point type is specified and the operand is a dereferenced pointer.

Description:

The E0105 error might be output if an increment or decrement expression of a floating point type is specified, while the result of a 4-byte data operation is retained in `_@RTARG0` and `_@RTARG4`, and the operand is a dereferenced pointer.

Example:

```
float *pf1;
int x;
void func()
{
    long l1, l2, l3, l4;
    x = (l1 & l2) < ((l3 - l4) + (*pf1)++);
}
```

Workaround:

Use a temporary variable as follows:

```
float tmp;
tmp = (*pf1)++;
x = (l1 & l2) < ((l3 - l4) + tmp);
```

Correction:

This issue has been corrected in V3.70.

No. 38 Invalid code might be output by an expression in which an expression of the format “address of `char` or `unsigned char` array + `unsigned char`” is type-cast to a pointer for an `int`, `unsigned int`, `short`, or `unsigned short`, and 0 is assigned to the value referenced by the pointer.

Description:

Invalid code might be output by an expression in which an expression of the format “address of `char` or `unsigned char` array + `unsigned char`” is type-cast to a pointer for an `int`, `unsigned int`, `short`, or `unsigned short`, and 0 is assigned to the value referenced by the pointer.

Example:

```
unsigned char table[10], idx;
void func()
{
unsigned char dummy;

    *((short *)(table + idx)) = 0x0;
}
```

Workaround:

Use a temporary variable as follows:

```
*((short *)(table + idx)) = 0x0;
        ↓
    short *tmp;
    tmp = (short *)(table + idx);
```

```
*tmp = 0x0;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 39 Invalid code might be output if a `char` or `unsigned char` expression is specified in a `return` statement for a function that returns a pointer.

Description:

Invalid code might be output if a `char` or `unsigned char` expression is specified in a `return` statement for a function that returns a pointer.

Example:

```
struct t {
    char c1;
    char c2;
    char c3;
} st = { 0x40, 0x01, 0x00 };
char *func()
{
    return st.c1;
}
```

Workaround:

Explicitly type-cast the `return` statement.

```
return st.c1;
       ↓
return (char *)st.c1;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 40 An error might be output if the result of an expression in which the result of performing a runtime library operation on a `long` is type-cast to a `char` or `unsigned char` and a relational or equality operation is performed with a constant that can be expressed by `char` or `unsigned char`.

Description:

The C0101 or C0104 error might be output if the result of an expression in which the result of performing a runtime library operation on a `long` is type-cast to `char` or `unsigned char` and a relational or equality operation is performed with a constant that can be expressed by `char` or `unsigned char`.

This problem occurs when an operand is an `unsigned char` (for a relational operation), or when an operand is a `char` or `unsigned char` and the constant is not 0 (for an equality operation).

Example:

```
long l1;
int x;
void func()
{
    if ((char)(l1 & 1) == 1) x++;
}
```

Workaround:

Use a temporary variable as follows:

```
char tmp;
tmp = (char)(l1 & 1);
if (tmp ==1) x++;
```

Correction:

This issue has been corrected in V3.70.

No. 41 When initializing an array whose size is not defined, if elements in the initializer braces are enclosed inconsistently, an area of invalid size is allocated.

Description:

When initializing an array whose size is not defined, if elements in the initializer braces are enclosed inconsistently, an area of invalid size is allocated

Example:

```
struct t {
    int a;
    int b;
} x[] = {1, 2, {3, 4}};
```

Workaround:

Do either of the following:

(1) Make sure the enclosing braces are consistent.

```
struct t {
    int a;
    int b;
} x[] = {{1, 2}, {3, 4}};
```

(2) Define the size of the array.

```
struct t {
    int a;
    int b;
} x[2] = {1, 2, {3, 4}};
```

Correction:

This issue has been corrected in V4.10.

No. 42 If a character string conversion function in the standard library is executed, error handling becomes invalid.

Description:

If a character string conversion function in the standard library is executed, error handling becomes invalid.

(1) If a character string cannot be converted by using the `strtod` function

If a character string that cannot be converted is specified, the position indicated by `p` is incorrect.

Example 1:

```
#include <stdlib.h>
double d;
char *p;
static char *string = "   XXX";
int x;
void func()
{
    d = strtod(string, &p);
    if (string == p) x++;
}
```

Remark Incorrect output: The position indicated by `p` is the top of `"XXX"`

Correct output: The position of `p` is the top of `"  XXX"`

(2) If a character string cannot be converted by using the `strtol` function, `errno` is not set.

If conversion cannot be performed as shown below, `errno` is not set to the `ERANGE` macro.

Example 2:

```
#include <stdlib.h>
#include <errno.h>
long l;
static char *string1 = "99999999999";
static char *string2 = "-99999999999";
int x;
void func()
{
     errno = 0;
     l = strtol(string1, NULL, 0);
     if (errno == ERANGE) x++;
     errno = 0;
     l = strtol(string2, NULL, 0);
     if (errno == ERANGE) x++;
}
```

(3) When the strtoul function is used and the character string to be converted starts with +, the conversion is not performed normally.
If a character string that cannot be converted is specified, errno is not set to the ERANGE macro.
Example 3:

```
#include <stdlib.h>
#include <errno.h>
unsigned long ul;
char *p;
static char *string = "99999999999";
int x;
void func()
{
     ul = strtoul("  +12", &p, 10);
     if (ul == 12L) x++;
     errno = 0;
     ul = strtoul(string, NULL, 0);
     if (errno == ERANGE) x++;
}
```

(4) When the strncpy function is used and the length of the copied character string is less than the number specified by the third parameter, null characters are not copied for the insufficient length.
Example 4:

```
#include <string.h>
char string1[] = "aaaaaaaaaa";
char string2[] = "bbbb\0bbbb";
void func()
{
     strncpy(string1, string2, 8);
}
```

Remark Incorrect output: "bbbb\0aaaaa"
Correct output: If the length is less than the specified number of characters, null characters are used as filler and "bbbb\0\0\0\0aa" is returned.

Workaround:

There is no workaround.

Correction:

This issue has been corrected in V4.00.

No. 43 The operation becomes invalid when output conversion is specified by an I/O function in the standard library.

Description:

When output conversion is specified by `printf`, `sprintf`, `vprintf`, or `vsprintf`, the result becomes invalid in the following cases:

(1) If the precision is specified as `.2` for the conversion specifier `d`, `i`, `o`, `u`, or `X`, the `0` flag is not ignored.

Example:

```
#include <stdio.h>
void func()
{
    printf("%04.2d\n", 77);
}
```

Remark Incorrect output: “0077”
Correct output: “ 77”

(2) For the conversion specifier `g` or `G`, the specified number of significant digits is increased by 1.

Example:

```
#include <stdio.h>
void func()
{
    printf("%.2g", 12.3456789);
}
```

Remark Incorrect output: “12.3”
Correct output: “12”

Workaround:

There is no workaround.

Correction:

This will not be corrected, so regard it as a specification.

No. 44 The incorrect size is output for the minimum value –32768 of an `int` or `short`.

Description:

The size of the minimum value –32768 of an `int` or `short` becomes 4.

Example:

```
int x;
void func()
{
     x = sizeof(-32768);
}
```

Remark Incorrect output: 4 is assigned to `x`.
Correct output: 2 is assigned to `x`.

Workaround:

Write the code as `–32767–1`.

Correction:

This will not be corrected, so regard it as a specification.

No. 45 If a function name or a function pointer is specified as the second and third operands of a conditional operation, an error is output, and then the function is called.

Description:

If a function name or a function pointer is specified as the second and third operands of a conditional operation, the E0307 error is output, and then the function is called.

Example:

```
void f1(), f2();
int x;
void func()
{
     (x ? f1 : f2)();
}
```

Workaround:

Specify an `if` statement instead of the conditional operator.

```
(x ? f1 : f2)();
     ↓
if (x) {
     f1();
}
else {
     f2();
}
```

Correction:

This issue has been corrected in V4.10.

No. 46 An error is output if an external pointer is initialized to a variable containing the operator ->.

Description:

The E0750 error is output if an external pointer is initialized to a variable containing the operator ->.

Example:

```
struct t {
    int i;
} b;
int *ip1 = &(&b)->i;
```

Workaround:

Specify the following:

```
int *ip1 = &(&b)->i;
     ↓
int *ip1 = &b.i;
```

Correction:

This issue has been corrected in V3.70.

No. 47 Error due to a mismatch between the parameter type and the identifier type in a function definition

Description:

Because argument promotion is not performed for the identifier type in a function definition, the parameter type and the identifier type in the function definition do not match, which causes the E0747 error.

Example:

```
int fn_char(int);
int fn_char(c)
char c;
{
     return 98;
}
```

Workaround:

Make the parameter type and the identifier type the same in the function definition.

Correction:

This will not be corrected, so regard it as a specification.

No. 48 A parameter whose type is not declared in the identifier list in a function definition is not handled as an `int`, which causes an error.

Description:

A parameter whose type is not declared in the identifier list in a function definition is not handled as an `int`, which causes the E0706 error.

Example:

```
void func(x1, x2, f, x3, lp, fp)
int (*fp)();
long *lp;
float f;
{
            ...
}
```

Workaround:

Declare the types of all parameters in a function definition.

Correction:

This will not be corrected, so regard it as a specification.

No. 49 The # operator cannot be used to expand a macro to a string literal.

Description:

Expansion is not executed correctly under either of the following conditions:

Condition 1: `'"'` cannot be expanded to a string literal by using the # operator, which causes a compile-time error.

Example:

```
#include <string.h>
#define str( a) (# a)
int x;
void func()
{
     if (strcmp(str('"'), "'\"'") == 0) x++;
}
```

Remark Incorrect output: A compile-time error occurs.

Correct output: `if (strcmp( ("'\"'") , "'\"'") == 0) x++;`

Condition 2: Nested macros that contain the # operator cannot be expanded to the correct string literals.

Example:

```
#define str(a) #a
#define xstr(a) str(a)
#define EXP 1
char *p;
void func()
{
     p = xstr(12EEXP);
}
```

Remark Incorrect output: `p = ("12E1");`
Correct output: `p =  ("12EEXP");`

Workaround:

There is no workaround.

Correction:

This will not be corrected, so regard it as a specification.

No. 50 If an `unsigned long` static variable is initialized to a floating point constant that is 0x80000000 or larger, the initial value becomes invalid.

Description:

If an `unsigned long` static variable is initialized to a floating point constant that is 0x80000000 or larger, the initial value becomes invalid.

Example:

```
unsigned long ul = 2200000000.0;
```

Workaround:

Do either of the following:

(1) Initialize the `unsigned long` static variable to an integer constant.

```
unsigned long ul = 2200000000;
```

(2) Type-cast the floating point constant to an appropriate integer type.

```
unsigned long ul = (unsigned long) 2200000000.0;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 51 Invalid code might be output for an expression that includes a function call and a `struct` or `union`.

Description:

Invalid code might be output for an expression that includes a function call and a `struct` or `union` if one of the following conditions is satisfied:

Condition 1: Either of the following conditions is satisfied:

(1) An expression exists that includes a function call and a `struct` or `union` assignment.

(2) A `struct` or `union` is used as a parameter to call a function by using a pointer.

Example:

```
struct t {
char c[16];
} st1, st2;
int x, i, idx;
int ifunc();
void (*fp[3])();
void func()
{
     x = ifunc() + (st1 = st2, i);
     fp[idx](st1);
}
```

Condition 2: Either of the following conditions is satisfied:

(1) Function calling by a function pointer occurs.

(2) The first parameter is a `struct` or `union` of three or four bytes and its value remains in a register.

The C0101 error is output when using an older function interface.

Example:

```
void (*fp)();
struct {
     char a;
     char b;
     char c;
     char d;
} st1, st2;
void func()
{
     fp(st1 = st2);
}
```

Workaround:

Do either of the following:

Workaround for condition 1:

(1) Do not specify a `struct` or `union` assignment in an expression that includes function calls.

```
x = ifunc() + (st1 = st2, i);
   ↓
st1 = st2;
x = ifunc() + i;
```

(2) To call a function, use the `struct` or `union` pointer, instead of the `struct` or `union` parameter.

```
fp[idx](st1);
   ↓
struct t *sp1;
sp1 = &st1;
fp[idx](sp1);
```

Workaround for condition 2: Do not specify a `struct` or `union` assignment as the first parameter of a function call.

```
fp(st1 = st2);
   ↓
st1 = st2;
fp(st1);
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 52 Invalid code might be output for the assignment expression `a = b` *binary operator* `c;`.

Description:

Invalid code might be output for the assignment expression `a = b` *binary operator* `c;` if all the following conditions are satisfied:

(1) The binary operator is `+`, `-`, `*`, `&`, `^`, `|`, or `<<`.

(2) `a` is an identifier, and `b` and `c` are identifiers or constants.

(3) Operand `a` is an `int`, `unsigned int`, `short`, or `unsigned short`.

(4) One of operands `b` and `c` is a `char` or `unsigned char`, and the other is a `long` or `unsigned long`.

Example:

```
char c1=0x12, c2=0x56;
int i1=0x34, i2=0x78;
void func()
{
 /*   (2)   (1)        Corresponding to the above numbers */
      i2 = c2 ^ 0x1ffff;

} /*      (3)    (4)    */
```

Workaround:

Type-cast the `long` or `unsigned long` operand to the type of the assignment destination.

```
i2 = c2 ^ 0x1ffff;
     ↓
i2 = c2 ^ (int)0x1ffff;
```

Correction:

This issue has been corrected in V3.70.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 53 An error is output if an element of an array member of a structure at a constant address is referenced by using the dot operator (`.`).

Description:

The C0101 error is output if an element of an array member of a structure at a constant address is referenced by using the dot operator (`.`).

Example:

```
struct st {
    char b[4];
} str;
char c;
void test(void)
{
      c = (*(struct st *)0x3E00).b[2];
}
```

Workaround:

Use the arrow operator (->).

```
c = (*(struct st *)0x3E00).b[2];
     ↓
c = ((struct st *)0x3E00)->b[2];
```

Correction:

This issue has been corrected in V3.70.

No. 54 An error is output for a function definition that has a certain pattern.

Description:

The E0705 error is output if the -QR option is specified in a normal model and a function that has four parameters of 1 byte, 1 byte, 2 bytes, and 2 bytes long, listed in that order from the first parameter, satisfies one of the following conditions:

(1) The function is noauto.

(2) All parameters are declared as register.

(3) The -QV option is specified

Example: When -QRV option is specified

```
void f(char p1, char p2, int p3, int p4)
{
                  ...
}
```

Workaround:

Change the declaration order for the parameters so that 1-byte, 1-byte, 2-byte, and 2-byte parameters are not listed in that order from the first parameter.

Correction:

This issue has been corrected in V3.70.

No. 55 Invalid code might be output if a 1-byte static array element, which uses an unsigned char value returned by a function as an array subscript, is specified for the code of the assignment destination of a compound assignment, increment, or decrement operation.

Description:

Invalid code might be output if a 1-byte static array element, which uses an unsigned char value returned by a function as an array subscript, is specified for the code of the assignment destination of a compound assignment, increment, or decrement operation.

Example 1:

------- C source----------

```
unsigned char a1[10];
unsigned char func1(viod);
void func()
{
        a1[func1()]++;
}
```

------- ASM source --------

```
_func:
                              ; push hl is not output
; line     5 :     ary1[func1()]++;
   call   !_func1
   movw hl,#_ary1
   mov  a,[hl+c]
   inc   a
   mov  [hl+c],a
; line     6 : }
                              ; pop hl is not output
   ret
```

Example 2:

------- C source ----------

```
signed char a2[10];
unsigned char func1(void);
signed char c1;
struct t1 {
   char m1;
} a3[10];
void func()
{
   a2[func1()] &= c1;          /* Invalid code */
   a3[func1()].m1 |= 0x55;     /* Invalid code */
}
```

-------- ASM source ----------

```
_func:
                        ; push hl is not output
   ; line     9 :     a2[func1()] &= c1;
      call      !_func1
      movw    hl,#_a2
      mov      a,[hl+c]
      and      a,!_c1
      mov      [hl+c],a
   ; line     10 :     a3[func1()].m1 |= 0x55;
      call        !_func1
      movw      hl,#_a3
      mov       a,[hl+c]
      or         a,#055H      ; 85
```

```
   mov        [hl+c],a
line    11 : }
                        ; pop hl is not output
   ret
```

Workaround:

Define a dummy local variable.

Example 1:

```
unsigned char a1[10];
unsigned char func1(void);
void func()
{
      unsigned char dummy;    /* Define a dummy variable */
    a1[func1()]++;
}
```

Example 2:

```
signed char a2[10];
unsigned char func1(void);
signed char c1;
struct t1 {
      char m1;
} a3[10];
void func()
{
   unsigned char dummy;       /* Define a dummy variable */
    a2[func1()] &= c1;
     a3[func1()].m1 |= 0x55;
}
```

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 56 A constant expression that includes multiple binary operators, which use the result of a binary operator causing an overflow, might be replaced with invalid values.

Description:

A constant expression that includes multiple binary operators, which use the result of a binary operator causing an overflow, might be replaced with invalid values.

Example:

```
long l1 = (10000 * 10000) / 10000;
long l2 = (30464 << 4) / 2;
long l3 = (30464 << 5) / 2;
long l4 = (65535U * 41200U) / 256L;
short s1 = (65535U * 41200U) / 100000;
void func()
{
   l1 = (10000 * 10000) / 10000;
```

```
    l2 = (30464 << 4) / 2;
    l3 = (30464 << 5) / 2;
    l4 = (65535U * 41200U) / 256L;
    s1 = (65535U * 41200U) / 100000;
}
```

Workaround:

Write constant expressions that do not include multiple binary operators.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 57 Invalid code might be output as the result of an operation that includes an increment or decrement operation.

Description:

(1) Invalid code might be output if an operand of a binary operation includes an increment or decrement operation for a bit field.

Example 1:

```
struct {
        int i : 9;
} *p;
int i;
int g(void);

void f(void)
{
        i = g() + p->i++;
}
```

(2) Invalid code might be output if an increment or decrement operator is suffixed to the left operand in a comma operation while the operation result remains in a register.

Example 2:

```
unsigned char c0, c1, c2, c3, c4;
void func()
{
        c0 = (c1 + c2) + (c3++, c4);
}
```

Workaround:

(1) Divide the expressions of an increment/decrement operation and a binary operation.

(2) Divide the expressions of a post-increment/decrement operation and a comma operation.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 58 The result of a `seizeof` operation for a function parameter that has an array might be invalid.

Description:

The result of a `seizeof` operation for a function parameter that has an array might be invalid, or the error E0529 might be output.

Example:

```
int x;
void func1(short a[ ])
{
   x = sizeof(a);            /* E0529: Sizeof returns zero */
}

void func2(short b[10])
{
   x = sizeof(b);
}
```

Workaround:

Change the function parameter type to a function pointer.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 59 Invalid code might be output as the result of a compound assignment operation of a division or remainder operation if the `-qc` option is not specified.

Description:

Invalid code might be output as the result of a compound assignment operation of a division or remainder operation when the `-qc` option is not specified, the left operand is a pointer to a `char` or `unsigned char` variable, and the right operand is a variable that can be expressed with 1 byte, stored in a stack, and whose offset is 256 or more.

Example:

```
char *p;

void f(unsigned char uc)
{
         char a[256];
         *p /= uc;
}
```

Workaround:

Specify the `-qc` option, or keep the number of array elements to 255 or less.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 60 Invalid code might be output if there is a bit field where the bit width assigned to an `saddr` area is from 2 bits to 7 bits, the maximum constant of the bit field is assigned to an expression, and the assignment destination is re-evaluated with the same expression.

Description:

Invalid code might be output if there is a bit field where the bit width assigned to an `saddr` area is from 2 bits to 7 bits, the maximum constant of the bit field is assigned to an expression, and the assignment destination is re-evaluated with the same expression.

Example:

```
__sreg struct {
        unsigned int b1 : 4;
        unsigned int b2 : 4;
} s;

void main(void)
{
        s.b2 = s.b1 = 0xf;
}
```

Workaround:

Divide the assignment expression.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 61 The error E0705 or invalid code might be output if a `norec` function that has `enum` parameters is called.

Description:

The error E0705 is output if the `-qr` option is not specified and a `norec` function that has two `enum` parameters is called. In addition, invalid code might be output if the `-qr` option is specified and a `norec` function that has two `enum` parameters is called.

Example:

```
enum E {
        A = 256
};

__leaf int func(enum E e1, enum E e2)
{
        return e1 == e2;
}

int main(void)
{
        func(A, A);
        return 0;
}
```

Workaround:

Change the `enum` to an `int` and the `enum` constant to a macro, as shown below.

Example:

```
#define A 256

__leaf int func(int e1, int e2)
{
        return e1 == e2;
}

int main(void)
{
        func(A, A);
        return 0;
}
```

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 62 No error occurs if an identical function whose parameters include a different structure or union is declared multiple times.

Description:

The error E0747 is not output if an identical function whose parameters include a different structure or union is declared multiple times.

Example:

```
struct st {
        int a;
} x;
struct st2 {
        char a;
} x2;

void func11(struct st a);
void func11(struct st2 a);
```

Workaround:

Do not declare the same function multiple times.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 63 Invalid code might be output if a pointer that points to a structure of 256 bytes or more is a `register` variable.

Description:

If a pointer that points to a structure of 256 bytes or more is assigned to a register, the register variable might be corrupted as the result of dereferencing its member.

Example:

```
struct st1 {
        char buf[250];
        struct st2 {
              char buf[6];
              int  i1;
              int  i2;
        } st2;
} st1;
int i1;
void func()
{
      register struct st1 *pst1 = &st1;
      i1 = pst1->st2.i1;
}
```

Workaround:

Do not declare `register` and disable the `-qv` option.

Alternatively, keep the structure size to 255 bytes or less.

Correction:

This issue has been corrected in V4.00.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 64 Work areas used by the compiler are corrupted if the static model option `-sm` and the expansion specification option `-zm2` are specified.

Description:

Work areas used by the compiler are corrupted if condition (a) or (b) is satisfied when the static model option `-sm` and the expansion specification option `-zm2` are specified.

This issue does not apply if the normal memory model is used.

(a) The function parameter includes any of the following:

- An address referenced using the `&` operator
- A structure
- A union

(b) The automatic variable is any of the following:

- An address referenced using the `&` operator
- A structure
- A union
- An array

Example for (b):

```
void func2(unsigned short *);
struct tag {
         unsigned char a[2];
};
void func1()
{
   unsigned char buf[4]; /* Array */
   struct tag st;        /* Structure */
   unsigned short ss;    /* Address referenced */

   func2(&ss);
   /* ... */
}
```

Workaround:

(1) For (a), use the `-zm1` option instead of the `-zm2` option.
Modifying the C source code does not resolve this issue.

(2) For (b), change the automatic variable to a local static variable.
Alternatively, use the `-zm2` option instead of the `-zm1` option.

```
void func2(unsigned short *);
struct tag {
         unsigned char a[2];
};
void func1()
{
      static unsigned char buf[4]; /* Array */
      static struct tag st;        /* Structure */
      static unsigned short ss;    /* Address referenced */

      func2(&ss);
      /* ... */
}
```

Correction:

This issue has been corrected in V4.00.

No. 65 Invalid code might be output when referencing 1-byte data pointed to by a pointer to which `++` or `--` has been suffixed.

Description:

Invalid code might be output if, immediately after referencing 1 byte to which a pointer points, the increment or decrement operator is suffixed to the pointer, and the memory to which the pointer points is referenced again.

Example:

```
void func()
{
      unsigned char tmp, *src, dst;
      *src = tmp + 0x80;
      dst += *src++;
}
```

Workaround:

Do either of the following:

(1) Divide the assignment and increment operations into different expressions:

```
dst += *src;
src++;
```

(2) Use temporary variables to divide the expressions:

```
tmp2 = tmp + 0x80;
*src = tmp2;
```

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 66 Invalid code is output if the last element of an initializer list for a `char`, `signed char`, or `unsigned char` array is a character string and one or more constants or character constants are placed before the character string.

Description:

If the last element of an initializer list for a `char`, `signed char`, or `unsigned char` array is a character string and one or more constants or character constants are placed before the character string, no error is output, but invalid code might be output.

Only string literals or constants can be used to initialize `char`, `signed char`, or `unsigned char` arrays.

Example:

```
[.c]
const char a1[ ] = {0x01, "abct"};
char *const TBL[3] = { a1 };
char *ptr1;
void func()
{
     ptr1 = TBL[0];
}

[.asm]
@@CNST          CSEG   UNITP
_a1:            DB     01H    ; 1
                DB     'ab'
_TBL:           DW     _a1    ; _TBL is an odd address
                DB     (4)

@@DATA          DSEG   UNITP
```

```
_ptr1:          DS      (2)

; line     1 : const char a1[] = { 0x01, "abc" };
; line     2 : char *const TBL[3] = { a1 };
; line     3 : char *ptr1;
; line     4 : void func()
; line     5 : {

@@CODEL         CSEG
_func:
; line     6 : ptr1 = TBL[0];
                movw    ax,!_TBL      ; Reference an odd address
                movw    !_ptr1,ax
```

Workaround:

Correctly specify the initial value.

```
[.c]
const char a1[ ] = { 0x01, 'a', 'b', 'c', '\0' };
char *const TBL[3] = { a1 };
char *const *ptr1;
void func()
{
     ptr1 = TBL[0];
}
```

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 67 Invalid code is output if referencing a pointer that has an offset obtained by subtracting one pointer from another.

Description:

Invalid code is output if all the following conditions are satisfied:

(1) A pointer to which an offset is added is referenced.

(2) The offset mentioned in (1) is the result of subtracting one pointer from another.

(3) A pointer mentioned in (2) has an offset.

Example:

```
[.c]
void main(void)
{
     char  *p1;
     char  *p2;
     char  *p3;

     *p1 = *(p2 + (p1 - (p3 + 2)));
}
```

Workaround:

Divide the expression.

```
[.c]
     void main(void)
{
     char  *p1;
     char  *p2;
     char  *p3;
     int tmp;                  /* Prepare a temporary variable */

     tmp = (p1 - (p3 + 2));    /* Assign the value to the temporary variable */
     *p1 = *(p2 + tmp);
}
```

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 68 Invalid code is output if the `-qc` option has not been specified (sign extension is specified to be `int` type).

Description:

Invalid code is output if all the following conditions are satisfied:

(1) The `-qc` option is not specified (sign extension is specified to be the `int` type)

(2) One of the following combinations of operands (regardless of whether they are right or left operands) is multiplied:
- An `sreg unsigned char` to which a constant from 0 to 255 is assigned and a constant from 0 to 255
- More than one `sreg unsigned char` to which a constant from 0 to 255 is assigned
- An `sreg unsigned char` to which a constant from 0 to 255 is assigned and an `sreg char` or `signed char` to which a constant from 0 to 127 is assigned

(3) The multiplication result is 256 or larger (which cannot be expressed as an `unsigned char`).

(4) The operation result is handled as an `int`.

In the following example, the value of `Temp1` should be `0x1fe` but it is output as `0xfe`.

Example:

```
[.c]

unsigned int   Temp1;
_sreg unsigned char  Byte1;
Temp1 = (Byte1 = 255) * 2;
```

Workaround:

Type-cast the variable to an `int` or `unsigned int`.

```
[.c]

unsigned int   Temp1;
unsigned char  Byte1;

Temp1 = (unsigned int) (Byte1 = 255) * 2;
```

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 69 Invalid code is output as the result of the BCD operation function `adbcdw` or `sbbcdw`.

Description:

Invalid code is output if one of the following conditions is satisfied while the BCD operation function `adbcdw` or `sbbcdw` is used:

(1) The return value of `adbcdw` or `sbbcdw` is assigned to an array or pointer.

(2) Another operation is executed before assigning the return value of `adbcdw` or `sbbcdw` to a temporary variable.

(3) A temporary variable to which the return value of `adbcdw` or `sbbcdw` has been assigned is used as is for other operations such as a conditional expression.

Example:

```
[.c]

void func()
{
      unsigned int   tmp1[3];
      unsigned int   tmp2, tmp3;
      unsigned int   a = 10, i = 0, *p;

      tmp1[i] = adbcdw(80, 50);                    /* (1) */
      tmp2 = adbcdw(80, 50) + (a + 1);             /* (2) */

      if ((tmp3 = adbcdw(80, 50) == *p ) …        /* (3) */
               ...
}
```

Workaround:

Prepare a function used only for calling `adbcdw` and `sbbcdw` and call the function. Use the same parameters and return values as those of `adbcdw` and `sbbcdw`.

Example:

```
[.c]

unsigned int adbcdw_new(unsigned int a, unsigned int b)
{
       return adbcdw(a, b);
```

```
}
void func()
{
    unsigned int   tmp1[3];
    unsigned int   tmp2, tmp3;
    unsigned int   a = 10, i = 0, *p;

    tmp1[i] = adbcdw_new(80, 50);                    /* (1) */
    tmp2 = adbcdw_new(80, 50) + (a + 1);             /* (2) */

    if ((tmp3 = adbcdw_new(80, 50) == *p ) …        /* (3) */
                ...
}
```

Correction:

This issue has been corrected in V4.10.

No. 70 An error occurs if the `-ng` option is specified and a branch instruction is used in a function that includes the `asm` statement.

Description:

If the `-ng` option is specified and a branch instruction is used after the `asm` statement in a function, an error might occur. This error might occur if both of the following conditions are satisfied:

(1) The `asm` statement is used in a function.

(2) A statement that causes processing to branch (such as `if`, `for`, or `while`) is used in the same function.

However, if the above conditions are satisfied, the error occurs during assembly and the object module file is not generated. If no error occurs, this restriction does not apply.

Example:

```
[.c]
unsigned int i;
void func()
{
     do {
         __asm("\t DB  (1000)");
         i++;
     } while ( i < 10) ;
}
```

Workaround:

Change to the `-g` option.

Correction:

This issue has been corrected in V4.10.

No. 71 The line number is not output for a statement that immediately follows a nested `if` statement and is outside that statement's conditional block.

Description:

If all of the conditions below are satisfied, the line number might not be output for a statement that immediately follows a nested `if` statement and is outside that statement's conditional block. However, the output code is correct. A break point cannot be specified for a statement for which the line number is not output.

(1) There are at least three levels of nested `if` statements.

(2) An `else` statement in a higher nested level has a larger line number than a nested `if` statement.

(3) At least one statement immediately follows an `if` statement in a higher nested level.

Example:

```
  [.c]
  int  f0, f1, f2, f3;
  int  g0, g1, g2;
  void func(void)
{
      if (f0){
          if (f1){      /* if statement in nested level 1 */
             g2 = 5;
          }
          else if (f2){/* if statement in nested level 2 */
              g2 = 4;
          }
          else if (f3){ /* if statement in nested level 3 (condition (1)) */
             g2 = 3;
          }
          else {
             g2 = 2;
          }
          g0 = 0x1234;  /* A statement immediately follows the if statement
                           in nested level 1 (condition (3)).*/
          g1 = 0x5678;
      }
      else {            /* This else statement has a larger line number than */
          g2 = 1;       /* the if statement in nested level 3 (condition (2)).*/
      }
}
```

Workaround:

Insert several blank lines before the statement that immediately follows the nested `if` statement and is outside that statement's conditional block.

For the above example, insert the lines before `g0 = 0x1234;`.

Correction:

This issue has been corrected in V4.10.

No. 72 Invalid code is output when a value is assigned to a `long` variable in an interrupt function.

Description:

When the assignment of a value to a `long` variable is specified in an interrupt function, the `-ql3` option or higher is specified, the compiler generates code that calls the runtime library function `@@dels03` or `@@hlls03`, the BC register is not used for other processing in the interrupt function, and no function is called from the interrupt function, the BC register contents are corrupted.

Example:

```
[.c]
unsigned long l;
unsigned int i;
__interrupt void func()
{
    l = (unsigned long)i;
}
```

Workaround:

Do either of the following:

(1) After the `return` statement at the end of the function (which must be added if not there), add code that increments the `long` variable.

```
[.c]
unsigned long l;
unsigned int i;
__interrupt void func()
{
          l = (unsigned long)i;
          return;
          l++;       /* Not executed */
}
```

(2) Create a dummy function that does not perform any processing, and call it from the interrupt function.

```
[.c]
unsigned long l;
unsigned int i;
void dummy { }
__interrupt void func()
{
          l = (unsigned long)i;
          dummy();
}
```

(3) Specify `-ql1` or `-ql2` as the `-ql` optimization option level.

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 73 Bank function calling code might not be output.

Description:

If one of the conditions below is satisfied, bank function calling code might not be output for functions allocated to the bank area by using the function information file.

In addition, `C0101: Internal error` might be output if condition *(3)* is satisfied. If this message is not output, there is no problem and the program code is not affected.

(1) A function is called by type-casting it to a function pointer.

(2) A function declared using a `typedef` type is called.

(3) The `-mf` option is specified, and a function is called using a function pointer.

Example:

```
[.c]
typedef void F(void);
typedef void (*FP)(void);
void func1(void);
F func2;
void func()
{
        ((FP)func1)();
        func2();
}
```

Workaround:

Do not call a bank function by type-casting it to a function pointer.

Do not use a `typedef` type to declare a bank function.

```
[.c]
void func1(void);
void func2(void);
void func()
{
       func1();
       func2();
}
```

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

For *(3)*, no check is performed.

No. 74 Invalid code might be output if using a 1-byte parameter or `auto` variable for a `norec` function.

Description:

Invalid code might be output if using a 1-byte parameter or `auto` variable for a `norec` function, and a `long` variable is dereferenced within the function.

Example:

```
[.c]
long buf[8]
norec void func(void)
{
    unsigned char c = 7;
    long *s = &buf[c-1], *d = &buf[c];
    *d = *s;
}
```

Workaround:

Change the `norec` function to a normal function, or change the 1-byte variable to a 2-byte variable.

Correction:

This issue has been corrected in V4.10.

A tool used to check whether this restriction applies is available.

For details, contact a Renesas Electronics distributor.

No. 75 Macro expansion using the `##` operator results in an error

Description:

The E0803 error might occur if the `##` operator is not followed by a function-like macro parameter, capital or small letter, or underscore (_).

In addition, using the `##` operator might cause errors other than E0803, such as E0711 or E0301.

Example 1:

```
[*.c]
#define m1(x) (x ## .c1 + 23)
#define m2(x) (x ## .c1 + 122)
struct t1 {
   unsigned char c1;
} st1;
unsigned char x1, x2;
void func1()
{
          x1 = m1(st1) + 100;     /* E0803 error (NG) */
          x2 = m2(st1) + 1;       /* No error   (OK) */
}
```

Example 2:

```
[*.c]
#define m3(x) (x ## 1)
unsigned char x3, uc1;
void func2()
{
          x3 = m3(uc);           /* E0711, E0301 error (NG) */
}
```

Workaround:

Do either of the following:

(1) Do not use the ## operator.

```
#define m1(x)   (x ## .c1 + 23)
#define m2(x)   (x ## .c1 + 122)
    ↓
#define m1(x)   ((x).c1 + 23)
#define m2(x)   ((x).c1 + 122)
```

(2) Place a function-like macro parameter immediately after the ## operator.

```
#define m3(x)   (x ## 1)
unsigned char x3, uc1;
void func2()
{
    x3 = m3(uc);
}
    ↓
#define m3(x, y) (x ## y)
unsigned char x3, uc1;
void func2()
{
    x3 = m3(uc, 1);
}
```

Correction:

This issue will be corrected in the next revision.

No. 76 Symbol information for an interrupt function is not output to the assembler source

Description:

The E3405 error occurs during linking if all the following conditions are satisfied:

(1) `#pragma interrupt` is used to specify the generation of a vector table for an interrupt function.

(2) The interrupt function is not defined in the same source.

(3) The `-no` option, assembler source module file output option (`-a` or `-sa`), and debugging information output option (`-g`) are enabled.

Example:

```
[*.c]
#pragma interrupt INTP0 inter
/*     Definition of this interrupt function is not subject to compilation
__interrupt void inter()
{
      ...
}
*/
```

Workaround:

Define the interrupt function or output the object module file.

Correction:

This issue will be corrected in the next revision.