Microcontroller Technical Information

CC78K0S 78K0S C Compiler Usage Restrictions	Document No.	ZBG-CD-07-0047		1/5
	Date issued	August 30, 2007		
	Issued by	Development Tool Solution Group Multipurpose Microcomputer Systems Division Microcomputer Operations Unit NEC Electronics Corporation		
Related documents CC78K0S Ver.1.50 or Later - Operation: U16654EJ1V0UM00 (1st edition) CC78K0S Ver.1.30 or Later - Language: U14872EJ1V0UM00 (1st edition) 78K0S C Compiler CC78K0S V1.50 Operating Precautions: SUD-DT-03-0388-E	Notification classification	√	Usage restriction	
			Upgrade	
			Document modification	
			Other notification	

1. Affected products

   CC78K0S V1.41/V1.50

2. New restrictions

   The following restrictions (No. 37 to No. 71) have been added. See the attachment for details.

   - No. 37 An invalid code is output if a function pointer with an asterisk (*) is referred to in a code other than a function call.
   - No. 38 When the -QR option is specified, an *sreg* variable may be redundantly allocated to the *saddr* area that is used by the compiler.
   - No. 39 An invalid code may be output if the number of bits shifted by a shift operator (<<, >>, <<=, or >>=) is a constant that is 256 or larger.
   - No. 40 An invalid code may be output if the constant 0xffff or 0xfffe is added to or subtracted from a *long* or *unsigned long* type variable.
   - No. 41 An error is output if a floating point number starting with 0e or 0E is described.
   - No. 42 An invalid code may be output as the operation result for one side of a binary operation.
   - No. 43 When 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 described as the second operand, the compare order becomes invalid.
   - No. 44 An error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators with the same priority are described in succession, is described as a *#if* constant expression.
   - No. 45 If one operand of a relational operation is a constant that cannot be expressed by the *signed long* type, the compare result may be invalid.
   - No. 46 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.
   - No. 47 An error is output when an increment/decrement expression of the floating point type is described and the operand is an indirect reference expression using a pointer.

- No. 48 An invalid code is output if a *char/unsigned char* type expression is described as a *return* statement for a function that returns a pointer.
- No. 49 An error may be output if the operation result of a *long* type run-time library is cast to a *char* or *unsigned char* type and a relational or equality operation is performed with a constant that can be expressed by the *char* or *unsigned char* type.
- No. 50 When initializing an array whose size is not defined when elements in the initializer braces are enclosed inconsistently, the size of the secured area becomes invalid.
- No. 51 When a character string conversion function in the standard library is executed, the error handling operation becomes invalid.
- No. 52 The operation becomes invalid when output conversion processing is performed for an I/O function in the standard library.
- No. 53 The size of the minimum value –32768of the *int/short* type becomes 4.
- No. 54 An error is output, if a function name or a function pointer is described as the second and third operands of a conditional operation, and then the function is called.
- No. 55 An error is output if an external pointer variable is initialized to a variable containing the operator “->”.
- No. 56 An error is output because the parameter type and the type of the identifier in a function definition do not match.
- No. 57 In an identifier list in a function definition, a parameter that is not declared is not handled as the *int* type, and an error results.
- No. 58 The # operator cannot be expanded correctly.
- No. 59 The initial value becomes invalid if an unsigned *long* type static variable is initialized to a floating point constant that is 0x80000000 or larger.
- No. 60 An invalid code may be output for an expression that includes a function call and a structure/union.
- No. 61 An invalid code is output for the assignment expression “a = b binary operator c;”.
- No. 62 An error is output if an array member element of a constant address structure is referred to using a dot operator (.).
- No. 63 An error is output for a function definition that has a certain pattern.
- No. 64 An integral constant expression that includes two or more binary operators, which use the result of a binary operator causing an overflow, may be replaced with invalid values.
- No. 65 An invalid code may be output as a result of an operation an including increment or decrement operation.
- No. 66 The result of a *sizeof* operation for a function parameter with an array type may be invalid.
- No. 67 An invalid code may 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.
- No. 68 An error F705 or invalid code may be output if a *norec* function which contains *enum* type parameters is called.
- No. 69 No error occurs even if an identical function whose parameters include a different structure or union type is declared multiple times.

- No. 70 An invalid code may be output when a pointer that points to a structure of 256 bytes or more is a register variable.
- No. 71 Work areas used by the compiler are corrupted when static model option -sm and expansion specification option -zm2 are specified.

3. Workarounds

The following workarounds are available for this restriction. See the attachment for details.

- No. 37 Do not append an asterisk (*) to a function pointer.
- No. 38 Do not specify the -QR option.
- No. 39 Match the number of shifted bits and the data width.
- No. 40 Insert a temporary variable as follows.
- No. 41 Describe 0 or 0.0 at the beginning of a floating point number.
- No. 42 Insert a temporary variable for at least one operand of the binary operation.
- No. 43 Modify the source code.
- No. 44 Use parentheses.
- No. 45 Implement either of the following workarounds.
  (1) Describe an expression that can be handled as constants, using constants.
  (2) Cast a boolean, bit, or *signed char* type expression to the *unsigned long* type.
- No. 46 Implement either of the following workarounds.
  (1) Cast the result of a logical negation operation, relational operation, or equality operation with the *int* type.
  (2) Do not use the floating point constant "±0.0" as the right operand of the logical operator &&.
- No. 47 Insert a temporary variable.
- No. 48 Explicitly cast the return statement.
- No. 49 Insert a temporary variable.
- No. 50 Implement either of the following workarounds.
  (1) Unify the brace enclosing method.
  (2) Define the size of the array.
- No. 51 There is no workaround.
- No. 52 There is no workaround.
- No. 53 Describe as "–32767–1".
- No. 54 Describe an *if* statement instead of the conditional operator.
- No. 55 Modify the source code as follows.

```
int *ip1 = &b.i;
```

- No. 56 Match the parameter type and the type of the identifier in the function definition.
- No. 57 Declare all parameters in a function definition.
- No. 58 There is no workaround.
- No. 59 Implement either of the following workarounds.
  (1) Initialize the *unsigned long* type static variable to an integer constant.
  (2) Cast the floating point constant to an appropriate integer type.

- No. 60 Implement either of the following workarounds.
  (1) Do not describe a structure/union assignment in an expression that contains function calls.
  (2) When calling a function using a function pointer, use the structure/union pointer, instead of the structure/union argument.
- No. 61 Cast the *long* or *unsigned long* type operand as the type of the assignment destination.
- No. 62 Use the arrow operator (->).
- No. 63 Change the order of declaration for the arguments so that 1-byte, 1-byte, 2-byte, and 2-byte width arguments are not listed in that order from the first argument.
- No. 64 Write constant expressions that do not include two or more binary operators.
- No. 65 Describe the expressions of an increment/decrement operation and a binary operation separately.
- No. 66 Modify the function parameter type to a function pointer.
- No. 67 Divide the assignment expression.
- No. 68 Modify an *enum* type to *int* type, and an enumeration constant to a macro.
- No. 69 Do not describe declaration of the same function multiple times.
- No. 70 Do not describe register declaration; in addition, disable the -qv option.
- No. 71 Use the -zm1 option, instead of the -zm2 option, or change the auto variable to an in-function static variable.

4. Modification schedule

Restrictions No. 37 to No. 49, No. 55, and No. 59 to No. 71 will be corrected in CC78K0S V2.00, which is planned for release in September 2007.

* For the detailed release schedule of modified products, contact an NEC Electronics sales representative.

5. List of restrictions

A list of restrictions in the CC78K0S, including the revision history and detailed information, is described on the attachment.

6. Document revision history

CC78K0S 78K0S C Compiler Usage Restrictions

Document Number	Date Issued	Description
SBG-DT-04-0007	January 23, 2004	Newly created.
SBG-DT-04-0112	March 12, 2004	Addition of restrictions No. 27 and No. 28
ZBG-CD-04-0073	October 4, 2004	Addition of restrictions No. 29 to No. 35
ZBG-CD-05-0002	January 12, 2005	Addition of restriction No. 36
ZBG-CD-07-0047	August 30, 2007	Addition of a new condition for restriction No. 30 Addition of new restrictions No. 37 to No. 71

List of Restrictions in CC78K0S

1. Product History

No.	Bugs and Changes/Addition to Specifications	Version		
		V1.30	V1.41	V1.50
1	If a character string includes a NULL character, the character string following the NULL character is invalid.	×	○	○
2	An invalid code may be output if top or bottom one-byte data is shifted as a result of addition/subtraction between constant 1 or 255 and 2-byte data.	×	○	○
3	The operation result is invalid when a member indicated by the pointer for a structure or union is the pointer for the array, and a *sizeof* operation is performed on the pointer for the array.	×	○	○
4	An invalid code may be output if a postfix increment/decrement operation expression is described in a *return* statement when the -sm option is specified	×	○	○
5	An invalid code may be output when a series of identifiers without type names are described in an argument for a function prototype declaration	×	○	○
6	An invalid code may be output as a result of outputting an unnecessary *push* instruction when a shift operation of a *long/unsigned long* type is executed	×	○	○
7	When only the bit field is described as a union member, the union size becomes 0	×	○	○
8	An invalid code may be output when a macro including a comma is defined	×	○	○
9	When a pointer array is initialized by using one character string, the area size of the array is invalid.	×	○	○
10	The constant expression of *#if* may not be executed correctly.	×	○	○
11	If values are read in order of floating-point type to integer type in scanf/sscanf, the value of the integer type cannot be input correctly.	×	○	○
12	An invalid code is output if the -ql option is not specified and the *sreg/__sreg* variable is incremented immediately after it is used in a conditional expression.	×	○	○
13	An invalid code is output if a structure pointer is assigned to a register and data is passed between members pointed to by the pointer.	×	○	○
14	An invalid code may be output if the shift count in a right-shift operation is larger than the number of bits of the shifted variable or is a negative value when using the normal model and the -ql4 option is specified.	×	○	○
15	An invalid code may be output if an external variable is multiplied by 2 when the -ql option is not specified or the -qq option is specified.	×	○	○
16	An invalid code is output if an operation is performed between array elements cast to the *int* type when using the normal model.	×	○	○
17	An invalid code is output if a negative floating-point constant is cast to an unsigned integer type.	×	○	○
18	An invalid code is output if a logical OR or logical AND operation is performed between floating-point constants.	×	○	○
19	If variables with the same name are declared in multiple files while using *#pragma section*, the variables may not be allocated to the correct section.	×	○	○

×: Applicable, ○: Not applicable, –: Not relevant

No.	Bugs and Changes/Addition to Specifications	Version		
		V1.30	V1.41	V1.50
20	An invalid code is output if a logical OR or logical AND operation is performed between a floating-point constant and integer-type constant.	×	○	○
21	The initialization of an external variable declared with *extern* within a block does not result in an error. In addition, the debugging information in the assembler source is incorrect.	×	×	×
22	Binding a variable with the same name to a variable declared with *extern* in the block is sometimes invalid.	×	×	×
23	If a type defined by *typedef* (*typedef* name) is used in a function prototype declaration or a declaration using a *const* or *volatile* type modifier, the *typedef* expansion is invalid, and an error results.	×	×	×
24	Sometimes a multidimensional array with an undefined size does not operate properly.	×	×	×
25	In a function returning the address of a function with arguments, those arguments cannot be referred to. There is no error when referred to, but invalid code is output.	×	×	×
26	The *signed* type bit field is handled as an unsigned bit field.	×	×	×
27	An invalid code is output as a result of a simple assignment operation in which the left-side and right-side operands are cast to the pointer for a structure, union, or array.	×	×	×*
28	An invalid code may be output as a result of the *memcpy* function when *#pragma inline* is specified.	×	×	×
29	An invalid code is output as a result of an operation that includes *signed char* type operator and constant values.	×	×	×*
30	An invalid code may be output as a result of an operation for a (*signed*) *int* type and an *unsigned short* type.	×	×	×*
31	Debug information is output to an invalid position when a conditional expression is followed by a simple assignment expression.	×	×	×
32	An invalid code is output after the STOP and HALT functions.	×	×	×
33	Characters are garbled in the Compiler Options dialog box.	–	–	×
34	Stack information of the *cprep2* function is invalid.	×	×	×
35	W503 is output when the array name of an automatic variable is referred to.	×	×	×
36	The initial value becomes invalid if a static variable is initialized with the floating constant.	×	×	×*
37	An invalid code is output if a function pointer with an asterisk (*) is referred to in a code other than a function call.	×	×	×*
38	When the -QR option is specified, an *sreg* variable may be redundantly allocated to the *saddr* area that is used by the compiler.	×	×	×*
39	An invalid code may be output if the number of bits shifted by a shift operator (<<, >>, <<=, or >>=) is a constant that is 256 or larger.	×	×	×*
40	An invalid code may be output if the constant 0xffff or 0xfffe is added to or subtracted from a *long* or *unsigned long* type variable.	×	×	×*
41	An error is output if a floating point number starting with 0e or 0E is described.	×	×	×

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

No.	Bugs and Changes/Addition to Specifications	Version		
		V1.30	V1.41	V1.50
42	An invalid code may be output as the operation result for one side of a binary operation.	×	×	×*
43	When 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 described as the second operand, the compare order becomes invalid.	×	×	×*
44	An error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators with the same priority are described in succession, is described as a *#if* constant expression.	×	×	×
45	If one operand of a relational operation is a constant that cannot be expressed by the *signed long* type, the compare result may be invalid.	×	×	×*
46	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.	×	×	×*
47	An error is output when an increment/decrement expression of the floating point type is described and the operand is an indirect reference expression using a pointer.	×	×	×
48	An invalid code is output if a *char*/*unsigned char* type expression is described as a *return* statement for a function that returns a pointer.	×	×	×*
49	An error may be output if the operation result of a *long* type run-time library is cast to a *char* or *unsigned char* type and a relational or equality operation is performed with a constant that can be expressed by the *char* or *unsigned char* type.	×	×	×
50	When initializing an array whose size is not defined when elements in the initializer braces are enclosed inconsistently, the size of the secured area becomes invalid.	×	×	×
51	When a character string conversion function in the standard library is executed, the error handling operation becomes invalid.	×	×	×
52	The operation becomes invalid when output conversion processing is performed for an I/O function in the standard library.	×	×	×
53	The size of the minimum value –32768 of the *int/short* type becomes 4.	×	×	×
54	An error is output, if a function name or a function pointer is described as the second and third operands of a conditional operation, and then the function is called.	×	×	×
55	An error is output if an external pointer variable is initialized to a variable containing the operator "->".	×	×	×
56	An error is output because the parameter type and the type of the identifier in a function definition do not match.	×	×	×
57	In an identifier list in a function definition, a parameter that is not declared is not handled as the *int* type, and an error results.	×	×	×
58	The # operator cannot be expanded correctly.	×	×	×
59	The initial value becomes invalid if an *unsigned long* type static variable is initialized to a floating point constant that is 0x80000000 or larger.	×	×	×*

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

No.	Bugs and Changes/Addition to Specifications	Version		
		V1.30	V1.41	V1.50
60	An invalid code may be output for an expression that includes a function call and a structure/union.	×	×	×*
61	An invalid code is output for the assignment expression “a = b binary operator c;”.	×	×	×*
62	An error is output if an array member element of a constant address structure is referred to using a dot operator (.).	×	×	×
63	An error is output for a function definition that has a certain pattern.	×	×	×
64	An integral constant expression that includes two or more binary operators, which use the result of a binary operator causing an overflow, may be replaced with invalid values.	×	×	×*
65	An invalid code may be output as a result of an operation an including increment or decrement operation.	×	×	×*
66	The result of a *sizeof* operation for a function parameter with an array type may be invalid.	×	×	×*
67	An invalid code may 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.	×	×	×*
68	An error F705 or invalid code may be output if a *norec* function which contains *enum* type parameters is called.	×	×	×*
69	No error occurs even if an identical function whose parameters include a different structure or union type is declared multiple times.	×	×	×*
70	An invalid code may be output when a pointer that points to a structure of 256 bytes or more is a register variable.	×	×	×*
71	Work areas used by the compiler are corrupted when static model option -sm and expansion specification option -zm2 are specified.	×	×	×

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

2. Details of Bugs and Additions to Specifications

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 area is secured for the character string following the NULL character.

[Workaround]

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

[Correction]

This issue has been corrected in V1.41.

No. 2 An invalid code may be output if top or bottom one-byte data is shifted as a result of addition/subtraction between constant 255 or 1 and 2-byte data.

[Description]

An invalid code may be output if top or bottom one-byte data is shifted as a result of addition/subtraction between constant 255 or 1 and 2-byte data.

Example:

```
int r;
void func()
{
    int cnt;
    int +=255;
    r = cnt;
}
```

[Workaround]

Specify the -qc option and divide the additive assignment operation into an additive operation and assignment operation.

Example:

```
cnt = cnt + 255;
```

[Correction]

This issue has been corrected in V1.41.

No. 3 The operation result is invalid when a member indicated by the pointer for a structure or union is the pointer for the array, and a *sizeof* operation is performed on the pointer for the array.

[Description]

The operation result is invalid when a member indicated by the pointer for a structure or union is the pointer for the array, and a *sizeof* operation is performed on the pointer for the array.

Example:

```
struct t {
    char (*a)[5];
    struct t *b;
} st;

void func()
{
    int x;
    x = sizeof(*(st.b->a));
}
```

[Workaround]

Describe the type name.

Example:

```
x = sizeof(char [5]);
```

[Correction]

This issue has been corrected in V1.41.

No. 4 An invalid code may be output if a postfix increment/decrement operation expression is described in a return statement when the -sm option is specified.

[Description]

An invalid code may be output if a postfix increment/decrement operation expression is described in a return statement when the -sm option is specified.

Example:

```
int func()
{
    int i = 10;
    return(i++);
}
```

[Workaround]

Describe the result of the increment/decrement operation minus 1 in the return statement.

Example:

```
    i++;
    return (i - 1);
```

[Correction]

This issue has been corrected in V1.41.

No. 5 An invalid code may be output when a series of identifiers without type names are described in an argument for a function prototype declaration.

[Description]

An invalid code may be output when a series of identifiers without type names are described in an argument for a function prototype declaration.

Example:

```
long AA;
int func(AA);
void main()
{
    func(AA);
}
```

[Workaround]

Describe the type name.

[Correction]

This issue has been corrected in V1.41.

No. 6 An invalid code may be output as a result of outputting an unnecessary *push* instruction when a shift operation of a *long/unsigned long* type is executed.

[Description]

An invalid code may be output as a result of outputting an unnecessary *push* instruction when a shift operation of a long/unsigned long type is executed.

Example:

```
long y, z;
long a, b, c, d, e, f;
void func()
{
   long x;
   x = ((a << 1) & (b << 1)) + ((c << 1) | (d << 1)) ^ ((e << 1) | (f << 1));
   y += z;
}
```

[Workaround]

Insert a temporary variable, and execute the operation by assigning the operation result to the temporary variable.

Example:

```
long tmp;
tmp = ((a << 1) & (b << 1)) + ((c << 1) | (d << 1));
x = tmp ^ ((e << 1) | (f << 1));
```

[Correction]

This issue has been corrected in V1.41.

No. 7 When only the bit field is described as a union member, the union size becomes 0.

[Description]

When only the bit field is described as a union member, the union size becomes 0.

Example:

```
union uni {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:1;
} x;
```

[Workaround]

Insert a type other than a bit field as a dummy.

```
union uni {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:1;
    unsigned int dummy;
} x;
```

[Correction]

This issue has been corrected in V1.41.

No. 8 An invalid code may be output when a macro including a comma is defined.

[Description]

An invalid code may be output when a macro including a comma is defined.

Example:

```
#define MARK  ',' /* correct value is 0x2C, but is recognized as 0xAC */
```

[Workaround]

Describe the definition as shown below.

```
#define MARK  (',')
```

[Correction]

This issue has been corrected in V1.41.

No. 9 When a pointer array is initialized by using one character string, the area size of the array is invalid.

[Description]

When a pointer array is initialized by using one character string, the area size of the array is invalid.

Example:

```
char *string[ ] = {"123"};
```

[Workaround]

Describe the element of the array.

```
char *string[1] = {"123"};
```

[Correction]

This issue has been corrected in V1.41.

No. 10 The constant expression of *#if* may not be executed correctly.

[Description]

The constant expression of *#if* may not be executed correctly.

Example:

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

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

[Workaround]

There is no workaround.

[Correction]

This issue has been corrected in V1.41.

No. 11 If values are read in order of floating-point type to integer type in scanf/sscanf, the value of the integer type cannot be input correctly.

[Description]

If values are read in order of floating-point type to integer type in scanf/sscanf, the value of the integer type 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 V1.41.

No. 12 An invalid code is output if the -ql option is not specified and the *sreg/__sreg* variable is incremented immediately after it is used in a conditional expression.

[Description]

An invalid code is output if the -ql option is not specified and the sreg/__sreg variable is incremented immediately after it is used in a conditional expression.

Example:

```
sreg char sc = 1;
void func()
{
    if (sc) sc++;
}
```

[Workaround]

Specify the -ql option.

[Correction]

This issue has been corrected in V1.41.

No. 13 An invalid code is output if a structure pointer is assigned to a register and data is passed between members pointed to by the pointer.

[Description]

An invalid code is 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]

Do not assign a structure pointer to a register.

[Correction]

This issue has been corrected in V1.41.

No. 14 An invalid code may be output if the shift count in a right-shift operation is larger than the number of bits of the shifted variable or is a negative value when using the normal model and the -ql4 option is specified.

[Description]

An invalid code may be output if the shift count in a right-shift operation is larger than the number of bits of the shifted variable or is a negative value when using the normal model and the -ql4 option is specified.

Example:

```
void func()
{
    int i, a;
    long lvals;
    i = -1 >> 16;
    lvals = -1;
    if (i == (lvals >> 16)) a = 1; else a = 0;
    i = -1 >> 17;
    lvals = -1;
    if (i == (lvals >> 17)) a = 1; else a = 0;
}
```

[Workaround]

Set the optimization level of the -ql option to -ql3 or lower, or avoid setting the right-shift count to the number of bits of the shifted variable or larger. In addition, do not set the right-shift count to a negative value because the shift count will be converted into an *unsigned* type.

[Correction]

This issue has been corrected in V1.41.

No. 15 An invalid code may be output if an external variable is multiplied by 2 when the -ql option is not specified or the -qq option is specified.

[Description]

An invalid code may be output if an external variable is multiplied by 2 when the -ql option is not specified or the -qq option is specified.

Example:

```
void func()
{
    int  x, i;
    for ( i = 1; i < 2; i ++ ) {
           x = i * 2 ;
    }
}
```

[Workaround]

Remove the -qq option and specify the -ql option.

[Correction]

This issue has been corrected in V1.41.

No. 16 An invalid code is output if an operation is performed between array elements cast to the *int* type when using the normal model.

[Description]

An invalid code is output if an operation is performed between array elements cast to the int type when using the normal model.

Example:

```
void func()
{
    short sh1[2] = { 3,  58 };
    int j = 1;
    int d;
    d = ((int)sh1[j] + (int)sh1[j]);
}
```

[Workaround]

Do not perform an operation between array elements cast to the *int* type.

[Correction]

This issue has been corrected in V1.41.

No. 17 An invalid code is output if a negative floating-point constant is cast to an unsigned integer type.

[Description]

An invalid code is output if a negative floating-point constant is cast to an unsigned integer type.

Example:

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

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

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

[Workaround]

Cast a constant to a signed integer type before casting the constant to the unsigned integer type.

Example:

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

[Correction]

This issue has been corrected in V1.41.

No. 18 An invalid code is output if a logical OR or logical AND operation is performed between floating-point constants.

[Description]

An invalid code is output if a logical OR or logical AND operation is performed between 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 V1.41.

No. 19 If variables with the same name are declared in multiple files while using *#pragma section*, the variable may not be allocated to the correct section.

[Description]

If variables with the same name are declared in multiple files while using *#pragma section*, the variable may 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 use variables with the same name in multiple files when using *#pragma section*.

[Correction]

This issue has been corrected in V1.41.

No. 20 An invalid code is output if a logical OR or logical AND operation is performed between a floating-point constant and integer-type constant.

[Description]

An invalid code is output if a logical OR or logical AND operation is performed between a floating-point constant and integer-type 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 describe a floating-point constant for the logical OR or logical AND operation.

[Correction]

This issue has been corrected in V1.41.

No. 21 The initialization of an external variable declared with *extern* within a block does not result in an error. In addition, the debugging information in the assembler source is incorrect.

[Description]

Since it is not compliant with the ANSI C language specifications, the initialization of an external variable declared with *extern* within a block should produce an error, but the description does not result in an error. The object defined as an external variable with initial value is interpreted and the code is output by the compiler.

The debugging information in the object output by the compiler is correct, but the debugging information in the assembler source is incorrect.

Example:

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

[Workaround]

There is no workaround.

[Correction]

This issue will be corrected in V2.00.

No. 22 Binding a variable with the same name to a variable declared with *extern* in the block is sometimes invalid.

[Description]

Binding a variable with the same name to a variable declared with *extern* in the block is invalid in either of the following cases.

(1) When a variable declared with *extern* in a block and a variable declared with *static* after outside the block have the same name

Since no error occurs and there is no binding, an invalid code is output when this variable is referred to.

Example:

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

(2) When a variable declared with *extern* in a block and a variable not declared with *static* outside the block after a variable declared with extern have the same name

There is no binding, and an invalid code is output.

Example:

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

(3) When a variable declared with *extern* in a block and a variable not declared with *extern* outside the block before a variable declared with *extern* have the same name, and an automatic variable declared in a block containing the block with the variable declared with *extern* has the same name

The variable outside the block and the variable declared with *extern* in the block are not bound, and an invalid code is output.

Example:

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

(4) A variable declared with *extern* in a block and a variable declared with *extern* in another block have the same name

There is no binding, and an 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]

Regard this issue as a usage restriction.

No. 23 If a type defined by *typedef* (*typedef* name) is used in a function prototype declaration or a declaration using a *const* or *volatile* type modifier, the *typedef* expansion is invalid, and an error results.

[Description]

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

Example 1:

```
typedef int  FTYPE();

FTYPE  func;
int  func(void);                /* F713 Redefined 'func' */
```

Example 2:

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

const VTYPE  *a;
const int  (*a)[2];             /* F713 Redefined 'a' */
volatile VPTYPE  b[2];
volatile int *volatile  b[2][3];    /* F713 Redefined 'b' */
```

[Workaround]

There is no workaround.

[Correction]

This issue will be corrected in V2.00.

No. 24 Sometimes a multidimensional array with an undefined size does not operate properly.

[Description]

Sometimes a multidimensional array with an undefined size does not operate properly.

Example 1:

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

Example 2:

```
char  c[][2][3]={"ab","cd","ef"};   /* Error (F756) */
```

[Workaround]

Define the size of the multidimensional array.

[Correction]

Regard this issue as a usage restriction.

No. 25 In a function returning the address of a function with arguments, those arguments cannot be referred to. There is no error when referred to, but invalid code is output.

[Description]

In a function returning the address of a function with arguments, those arguments cannot be referred to. There is no error when referred to, 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 description (W510) */
        (*f1(i))(i);        /* Incorrect description */
        (*f2())(c);         /* Correct description (W509) */
        (*f2())();          /* Incorrect description (W509) */
        (*f3(i))();         /* Correct description (W509) */
        (*f3(i))(i);        /* Incorrect description */
}
```

W509 or W510 is output for a correct description. Nothing is output for a description that should produce a warning. However, the output code is normal.

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

An error is not output for a description that should cause an error. An invalid code is output.

[Workaround]

There is no workaround.

[Correction]

This issue will be corrected in V2.00.

No. 26 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]

Regard this issue as a usage restriction.

No. 27 An invalid code is output as a result of a simple assignment operation in which the left-side and right-side operands are cast to the pointer for a structure, union, or array.

[Description]

An invalid code is output when the following four conditions are satisfied.

(1) Simple assignment operation

(2) The left-side and right-side operands are both indirection reference data that uses a structure, union, array, or pointer

(3) The left-side and right-side operands both reference the address of a symbol.

(4) 4-byte data (*long*, *float*, *double*, or *long double* type) is the target of the operation.

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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 28 An invalid code may be output as a result of the *memcpy* function when *#pragma inline* is specified.

[Description]

An invalid code may be output when the following three conditions are satisfied.

(1) *#pragma inline* is specified.

(2) A *memcpy* function is used.

(3) The third argument of the *memcpy* function in (2) is not a constant (when the third argument uses the HL register)

Example:

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

[Workaround]

Implement any of the following workarounds.

(1) Do not specify *#pragma inline* when the *memcpy* function is used.

(2) If *#pragma inline* is specified, describe a constant for the third argument of the *memcpy* function.

[Correction]

This issue will be corrected in V2.00.

No. 29 An invalid code is output as a result of an operation that includes *signed char* type operator and constant values.

[Description]

An invalid code may be output if at least 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) If a right-shift operator “>>” is used, a left operand is a constant ranging from 128 to 255, and a *signed char* type operand is used on the right side.

(2) If a right-shift operator “/”, “%”, “<”, “<=”, “>”, “>=”, “/=”, or “%=” is used, either left or right operand is a constant ranging from 128 to 255, and a *signed char* type operand is used on the other side.

(3) If a binary operator operation is performed, in which either a left or right operand is a constant ranging from 128 to 255 and a *signed char* type operand is used on the opposite side, and obtained the result is converted into a type that is longer than 2 bytes.

(4) If a binary operator operation is performed, in which either a left or right operand is a constant ranging from 128 to 255 and a *signed char* type operand is used on the opposite side, and the obtained result is used as a left operand, for the right-shift operator “>>” with a *signed char* type operand as the right operand.

(5) If a binary operator operation is performed, in which either a left or right operand is a constant ranging from 128 to 255 and a *signed char* type operand is used on the opposite side, and the obtained result is used as an operand on a side, for the operator "/", "%", "<", "<=", ">", ">=", "/=", or "%=" whose opposite-side operand is a *signed char* type operand.

(6) If a binary operator operation is performed, in which either a left or right operand is a constant that uses at least one of the operators "<<", ">>", "&", "^", or "|", and the constant ranges from −128 to 255, the obtained result ranges from −128 to −1, and the type of the opposite-side 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]

Cast the relevant constant as a signed *int* type.

Example 1:

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

Example 2:

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

[Correction]

This restriction will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 30 An invalid code may be output as a result of an operation for a (*signed*) *int* type and an *unsigned short* type.

[Description]

When either of the following conditions is satisfied, a signed operation is performed for an unsigned operation, which may cause an invalid operation result.

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

Example 1 for condition 1:

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

Example 2 for condition 1:

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

Example 3 for condition 1:

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

Condition 2: In a compound assignment operation (%= or /=) or binary operation, one side is the (*signed*) *int* type, and the other side is the *unsigned short* type, and the type of the operation result is converted into the type with a width of 2 bytes or longer.

Example for condition 2:

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

[Workaround]

Modify an *unsigned short* type to an *unsigned int* type.

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 31 Debug information is output to an invalid position when a conditional expression is followed by a simple assignment expression.

[Description]

When a static 1-byte variable that cannot be assigned to the *saddr* area is referred to 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, debug information is output to an invalid position. This does not affect the output code itself.

Example:

C Source Description	Corresponding Output Assembler

if(TEST >= 10)
 	
{
  TEST = 0;
 	
}
	
	movw de,#_TEST
	mov a,[de]
	cmp a,#0AH ; 10
	
	bc $?L0003 ;(1)
	callt [@@clra0]
	mov [de],a
	br $?L0003

A compare instruction is output to the location for an assignment statement.

A break occurs at (1) in the above program even if the breakpoint has been set to the position of TEST = 0; using the debugger. That is, the break does not occur when a conditional expression of an *if* statement results in True, but occurs at the conditional expression.

[Workaround]

Set a breakpoint to a relevant location in the assembler code, in the mixed display in the Source window of the debugger.

[Correction]

This issue will be corrected in V2.00.

No. 32 An invalid code is output after the STOP and HALT functions

[Description]

When an object (*.rel) is output from the CC78K0S while the STOP and HALT functions of the CPU control instruction have been described, the following unnecessary code will be output next to the above functions.

- In the case of a STOP function

  ROR A,1
- In the case of a HALT function

  ROL A,1

[Workaround]

Specify the -s or -sa option, output the assembler source file (*.asm), then assemble the files in the RA78K0S to create an object.

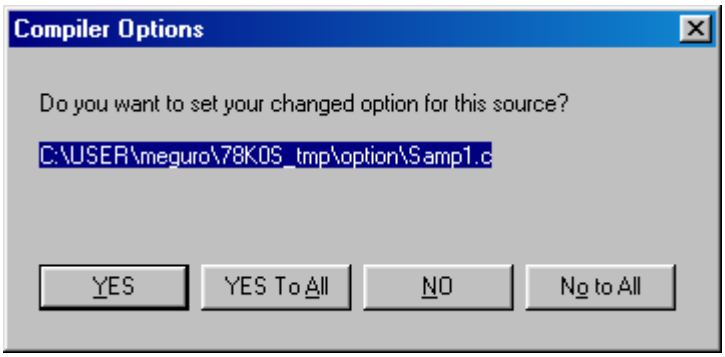
[Correction]

This issue will be corrected in V2.00.

No. 33 Characters are garbled in the Compiler Options dialog box.

[Description]

The characters in the following message dialog box are garbled.



This dialog box is output if a compiler option that is applied to overall files is set while setting file-specific options in the Compiler Options dialog box of PM plus.

[Workaround]

Set compiler options that are applied to overall files then set file-specific options.

[Correction]

This issue will be corrected in V2.00.

No. 34 Stack information of the *cprep2* function is invalid.

[Description]

In function information output by the compiler, usually the amount of stacks consumed by a run-time library is added to the stack information of a function that includes the run-time library.

If the program includes the *cprep2* function, however, another 2 bytes are excessively added.

[Workaround]

There is no workaround.

If the program includes the *cprep2* function, subtract 2 bytes from the amount of stacks consumed.

[Correction]

This issue will be corrected in V2.00.

No. 35 W503 is output when the array name of an automatic variable is referred to.

[Description]

W503 is output when the array name of an automatic variable without initialization is referred to.

W503 Possible use of '*variable-name*' before definition

Note:

"Initialization" means a declaration such as int a[2]={0,0};. It does not include assignment expressions such as a[0] = 0; a[1] = 0;.

An "array name" is 'a' in int a[2]. It does not include a[0], a[1], nor &a[0].

Example:

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

   a[0] = 0;
   a[1] = 0;
   b = a;            /* W503 is output to this line */
}
```

[Workaround]

When W503 is output, check the relevant location. Ignore the message if initialization has been performed in the statement.

[Correction]

Regard this issue as a usage restriction.

No. 36 The initial value becomes invalid if a static variable is initialized with the floating constant.

[Description]

The initial value becomes invalid if a variable other than the ones below is initialized with the floating constant.

- *long/unsigned long/floating* type variable
- *long/unsigned long/floating* type member of structure/union
- *long/unsigned long/floating* type element of array

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

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

[Workaround]

Implement either of the following workarounds.

(1) Initialize the static variable with the integer constant.

Describe as follows for the above example 1.

```
signed char a1[3] = {1};
```

(2) Cast the relevant floating constant as an integer constant type.

Describe as follows for the above example 1.

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

Describe as follows for the above example 2.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 37 An invalid code is output if a function pointer with an asterisk (*) is referred to in a code other than a function call.

[Description]

An invalid code is output if a function pointer with an asterisk (*) is referred to in a 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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 38 When the -QR option is specified, an *sreg* variable may be redundantly allocated to the *saddr* area that is used by the compiler.

[Description]

When the -QR option is specified, an *sreg* variable may be redundantly allocated to the *saddr* area that is 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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 39 An invalid code may be output if the number of bits shifted by a shift operator (<<, >>, <<=, or >>=) is a constant that is 256 or larger.

[Description]

An invalid code may 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]

Match the number of shifted bits and the data width.

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 40 An invalid code may be output if the constant 0xffff or 0xfffe is added to or subtracted from a *long* or *unsigned long* type variable.

[Description]

An invalid code may be output if the constant 0xffff or 0xfffe is added to or subtracted from a *long* or *unsigned long* type variable.

Example:

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

[Workaround]

Insert a temporary variable as follows.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 41 An error is output if a floating point number starting with 0e or 0E is described.

[Description]

An error F312 is output if a floating point number starting with 0e or 0E is described.

Example:

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

[Workaround]

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

[Correction]

This issue will be corrected in V2.00.

No. 42 An invalid code may be output as the operation result for one side of a binary operation.

[Description]

When any of the following three conditions is satisfied, the result of a logical or conditional operation that has been saved may not be restored normally, which results in an invalid code being output.

Condition 1: When the following four conditions are satisfied.

(1) The code includes a binary operation.

(2) One operand of the binary operation that is described in (1) refers to the operation result of a logical operation (&& or ||) or conditional operation (?:).

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

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

Example for condition 1:

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

Condition 2: When the following three conditions are satisfied.

(1) The code includes a binary operation.

(2) One operand of the binary operation that is described in (1) refers to the operation result of a logical operation (&& or ||).

(3) The operation result of the other operand of the binary operation that is described in (1) remains in _@RTARGx (argument in the run-time library).

Example for condition 2:

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

Condition 3: When the following three conditions are satisfied.

(1) The code includes a binary operation.

(2) One operand of the binary operation that is described in (1) refers to the operation result of a compound assignment operation.

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

Example for condition 3:

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

[Workaround]

Implement the corresponding workaround out of the following.

Workaround for condition 1:

Insert 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++;
```

Workaround for condition 2:

Insert 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++;
```

Workaround for condition 3:

Insert 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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 43 When 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 described as the second operand, the compare order becomes invalid.

[Description]

When 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 described as the second operand, the compare order becomes invalid.

Example:

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

Remark Illegal operation: f2++ is executed for “if(f1 || f2++)” regardless of whether f1 is true or false
Normal operation: f2++ is executed for “if(f1 || f2++)” only when 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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 44 An error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators with the same priority are described in succession, is described as a *#if* constant expression.

[Description]

The F501 error is output if an expression starting with a unary plus or unary minus operator, or an expression in which operators with the same priority are described in succession, is described as a *#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 will be corrected in V2.00.

No. 45 If one operand of a relational operation is a constant that cannot be expressed by the *signed long* type, the compare result may be invalid.

[Description]

An invalid code may be output if one operand of a relational operation is a constant that cannot be expressed by the *signed long* type, and either of the following conditions is satisfied.

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

Example for condition 1:

```
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 of the boolean, bit, or *signed char* type.

Example for condition 2:

```
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:

Describe an expression that can be handled as constants, 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:

Cast a boolean, bit, or *signed char* type expression to the *unsigned long* type.

```
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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 46 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) When the type of a logical negation operation, relational operation, or equality operation is any of the following, the operation 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) When the right operand of the logical operator && is the floating point constant "±0.0", the operation result does not become the *int* type, but becomes the floating point type instead (excluding cases where 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) Cast the result of a logical negation operation, relational operation, or equality operation with the *int* type.

```
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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 47 An error is output when an increment/decrement expression of the floating point type is described and the operand is an indirect reference expression using a pointer.

[Description]

The F105 error may be output when an increment/decrement expression of the floating point type is described, while the operation result of 4-byte data is held in _@RTARG0 and _@RTARG4, and the operand is an indirect reference expression using a pointer.

Example:

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

[Workaround]

Insert a temporary variable as follows.

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

[Correction]

This issue will be corrected in V2.00.

No. 48 An invalid code is output if a *char*/*unsigned char* type expression is described as a *return* statement for a function that returns a pointer.

[Description]

An invalid code is output if a *char/unsigned char* type expression is described as 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 cast the return statement.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 49 An error may be output if the operation result of a *long* type run-time library is cast to a *char* or *unsigned char* type and a relational or equality operation is performed with a constant that can be expressed by the *char* or *unsigned char* type.

[Description]

The F101 or F104 error may be output if the operation result of a *long* type run-time library is cast to a *char* or *unsigned char* type and a relational or equality operation is performed with a constant that can be expressed by the *char* or *unsigned char* type.

This bug occurs when an operand is an *unsigned char* type (for a relational operation) or a *char* or *unsigned char* type and the constant is not 0 (for an equality operation).

Example:

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

[Workaround]

Insert a temporary variable as follows.

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

[Correction]

This issue will be corrected in V2.00.

No. 50 When initializing an array whose size is not defined when elements in the initializer braces are enclosed inconsistently, the size of the secured area becomes invalid.

[Description]

When initializing an array whose size is not defined when elements in the initializer braces are enclosed inconsistently, the size of the secured area becomes invalid.

Example:

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

[Workaround]

Implement either of the following workarounds.

(1) Unify the brace enclosing method.

```
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]

Regard this issue as a usage restriction.

No. 51 When a character string conversion function in the standard library is executed, the error handling operation becomes invalid.

[Description]

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

(1) When conversion cannot be performed using the *strtod* function

If a character string that cannot be converted is specified, the position of *p* becomes invalid.

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 Illegal operation: The position of *p* is the top of "XXX"
Normal operation: The position of *p* is the top of " XXX"

(2) When conversion cannot be performed 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 copy source character string is less than the number specified by the third argument, 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 Illegal operation: “bbbb\0aaaaa”

Normal operation: 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 will be corrected in V2.00.

No. 52 The operation becomes invalid when output conversion processing is performed for an I/O function in the standard library.

[Description]

When output conversion processing is performed for *printf*, *sprintf*, *vprintf*, or *vsprintf*, the operation becomes invalid under the following conditions.

(1) When a 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 Illegal operation: “0077”
Normal operation: “ 77”

(2) For the conversion specifier "g,G", the result is “specified precision + 1”.
In the following case, "12" is not output as the conversion result.

Example:

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

Remark Illegal operation: “12.3”
Normal operation: “12”

[Workaround]

There is no workaround.

[Correction]

Regard this issue as a usage restriction.

No. 53 The size of the minimum value –32768 of the *int*/*short* type becomes 4.

[Description]

The size of the minimum value –32768 of the *int/short* type becomes 4.

Example:

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

Remark Illegal operation: The value of x becomes 4.
Normal operation: The value of x becomes 2.

[Workaround]

Describe as “–32767–1”.

[Correction]

Regard this issue as a usage restriction.

No. 54 An error is output, if a function name or a function pointer is described as the second and third operands of a conditional operation, and then the function is called.

[Description]

The F307 error is output, if a function name or a function pointer is described as the second and third operands of a conditional operation, and then the function is called.

Example:

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

[Workaround]

Describe an *if* statement instead of the conditional operator.

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

[Correction]

Regard this issue as a usage restriction.

No. 55 An error is output if an external pointer variable is initialized to a variable containing the operator "->".

[Description]

The F750 error is output if an external pointer variable is initialized to a variable containing the operator "->".

Example:

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

[Workaround]

Describe as follows.

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

[Correction]

This issue will be corrected in V2.00.

No. 56 An error is output because the parameter type and the type of the identifier in a function definition do not match.

[Description]

Because argument promotion for the type of an identifier in a function definition is not performed, the parameter type and the type of the identifier in the function definition do not match, thus causing the F747 error.

Example:

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

[Workaround]

Match the parameter type and the type of the identifier in the function definition.

[Correction]

Regard this issue as a usage restriction.

No. 57 In an identifier list in a function definition, a parameter that is not declared is not handled as the *int* type, and an error results.

[Description]

In an identifier list in a function definition, a parameter that is not declared is not handled as the *int* type, thus causing the F706 error.

Example:

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

[Workaround]

Declare all parameters in a function definition.

[Correction]

Regard this issue as a usage restriction.

No. 58 The # operator cannot be expanded correctly.

[Description]

Expansion is not performed correctly under either of the following conditions.

Condition 1: [""] cannot be expanded correctly with a # operator, and an error results at compilation.

Example for condition 1:

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

Remark Illegal operation: An error is output at compilation.
Normal operation: if (strcmp(("'\"") , "'\"") == 0) x++;

Condition 2: Macros that contain a # operator and a nested structure cannot be expanded correctly.

Example for condition 2:

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

Remark Illegal operation: “p = ("12E1");”
Normal operation: “p = ("12EEXP");”

[Workaround]

There is no workaround.

[Correction]

Regard this issue as a usage restriction.

No. 59 The initial value becomes invalid if an *unsigned long* type static variable is initialized to a floating point constant that is 0x80000000 or larger.

[Description]

The initial value becomes invalid if an *unsigned long* type static variable is initialized to a floating point constant that is 0x80000000 or larger.

Example:

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

[Workaround]

Implement either of the following workarounds.

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

unsigned long ul = 2200000000;

(2) Cast the floating point constant to an appropriate integer type.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 60 An invalid code may be output for an expression that includes a function call and a structure/union.

[Description]

An invalid code may be output for an expression that includes a function call and a structure/union if one of the following conditions is satisfied.

Condition 1: Either of the following two conditions is satisfied.

(1) An expression exists that contains a function call and a structure/union assignment

(2) A structure/union is used as an argument in function calling by a function pointer

Example for condition 1:

```
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 two conditions is satisfied.

(1) Function calling by a function pointer occurs

(2) The first argument is a structure/union of three or four bytes and its value remains in a register

The F101 error is output when using an older function interface.

Example for condition 2:

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

```
{
     fp(st1 = st2);
}
```

[Workaround]

Implement either of the following workarounds.

Workaround for condition 1:

(1) Do not describe a structure/union assignment in an expression that contains function calls.

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

(2) When calling a function using a function pointer, use the structure/union pointer, instead of the structure/union argument.

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

Workaround for condition 2: Do not describe a structure/union assignment as the first argument of a function call.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 61 An invalid code is output for the assignment expression “a = b binary operator c;”.

[Description]

An invalid code is output for the assignment expression “a = b binary operator c;” when the following four conditions are satisfied.

(1) The binary operator in question is any of +, -, *, &, ^, |, or <<.

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

(3) Operand a is of the *int*, *unsigned int*, *short*, or *unsigned short* type.

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

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]

Cast the *long* or *unsigned long* type operand as the type of the assignment destination.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 62 An error is output if an array member element of a constant address structure is referred to by using a dot operator (.).

[Description]

A F101 error is output if an array member element of a constant address structure is referred to by using a 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 will be corrected in V2.00.

No. 63 An error is output for a function definition that has a certain pattern.

[Description]

An F705 error is output when the -QR option is specified in a normal model and a function that has four arguments of 1-byte, 1-byte, 2-byte, and 2-byte width, listed in that order from the first argument, satisfies either of the following conditions.

(1) The relevant function is the *noauto* function.

(2) Register declaration is applied to all arguments

(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 order of declaration for the arguments so that 1-byte, 1-byte, 2-byte, and 2-byte width arguments are not listed in that order from the first argument.

[Correction]

This issue will be corrected in V2.00.

No. 64 An integral constant expression that includes two or more binary operators, which use the result of a binary operator causing an overflow, may be replaced with invalid values.

[Description]

An integral constant expression that includes two or more binary operators, which use the result of a binary operator causing an overflow, may 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 two or more binary operators.

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 65 An invalid code may be output as a result of an operation an including increment or decrement operation.

[Description]

(1) An invalid code may 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) An invalid code may be output if a postfix increment/decrement operand is written as 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) Describe the expressions of an increment/decrement operation and a binary operation separately.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 66 The result of a *sizeof* operation for a function parameter with an array type may be invalid.

[Description]

The result of a *seizeof* operation for a function parameter with an array type may be invalid, or an error F529 may be output.

Example:

```
int x;
void func1(short a[ ])
{
x = sizeof(a);           /* F529: Sizeof returns zero */
}
```

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

[Workaround]

Modify the function parameter type to a function pointer.

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 67 An invalid code may 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.

[Description]

An invalid code may 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.

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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 68 An error F705 or invalid code may be output if a *norec* function which contains *enum* type parameters is called.

[Description]

An error F705 is output when the *-qr* option is not been specified and if a *norec* function which contains two *enum* type parameters is called. In addition, an invalid code may be output when the *-qr* option is specified and if a *norec* function which contains two *enum* type 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]

Modify an *enum* type to *int* type, and an enumeration 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 will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 69 No error occurs even if an identical function whose parameters include a different structure or union type is declared multiple times.

[Description]

Error F747 is not output even if an identical function whose parameters include a different structure or union type 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 describe declaration of the same function multiple times.

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 70 An invalid code may be output when 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 value may be corrupted as a result of indirect reference to 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 describe register declaration; in addition, disable the -qv option.

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

[Correction]

This issue will be corrected in V2.00.

A tool used to check whether this restriction applies or not is available.

Contact an NEC Electronics sales representative or distributor for details.

No. 71 Work areas used by the compiler are corrupted when static model option -sm and expansion specification option -zm2 are specified.

[Description]

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

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

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

- Address is referred to with an “&” operator
- Structure
- Union

(b) Any of the following is the auto variable.

- Address is referred to with an “&” operator
- Structure
- Union
- 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 referred to */

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

[Workaround]

(1) In case of (a), use the -zm1 option, instead of the -zm2 option.
Modifying C source description does not resolve this issue.

(2) In case of (b), change the auto variable to an in-function 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 referred to */

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

[Correction]

This issue will be corrected in V2.00.

3. Cautions

None.