Microcontroller Technical Information

CA850 V850 C Compiler Package Usage Restrictions	Document No.	ZBG-CD-10-0030	1/3
	Date issued	Sep 13, 2010	
	Issued by	MCU Tool Product Marketing Department MCU Software Division MCU Business Unit Renesas Electronics Corporation	
Related documents CA850 Ver. 3.20 C Language: U18513EJ1 (1st edition) CA850 Ver. 3.20 Assembly Language: U18514EJ1 (1st edition) CA850 Ver. 3.20 Operation: U18512EJ1 (1st edition) CA850 Ver. 3.20 Link Directives: U18515EJ1 (1st edition) PM+ Ver. 6.30 User's Manual: U18416EJ1 (1st edition) CA850 Ver. 2.50 Coding Technique: U16076EJ1 (1st edition)	Notification classification	√	Usage restriction
			Upgrade
			Document modification
			Other notification

1. Affected product

   CA850 (part number: CA703000)

2. New restrictions

   Restrictions No. 111 and No. 112 for the CA850 have been added.

   - No. 111 Restriction on incorrect compare operation optimization while casting
   - No. 112 Restriction on incorrect move optimization of assignment sentence

3. Workarounds

   The workarounds below are available for these restrictions. See attachments 1 for details.

   CA850:

   - No. 111 Insert '__asm("¥n");' before the line where incorrect comparison operation optimization occurs.

   - No. 112 Insert '__asm("¥n");' before the line which the checktool output.

4. Removed restrictions

The restrictions below have been removed. See attachments 1 and 2 for details.

CA850:

- No. 104 Incorrect number of loop executions
- No. 105 Incorrect string literals
- No. 106 Restriction involving format specifiers for the sscanf, fscanf, and scanf functions
- No. 107 Restriction involving the character string parameter for the atoi, atol, strtol, and strtoul functions
- No. 108 Restriction involving initialization of a structure that includes a bit field among its members
- No. 109 Restriction involving nested conditionally assembled pseudo instructions
- No. 110 Restriction involving assignment within switch and if statements

Other package tools:

- No. 9 Restriction on acquiring information on included C source files handled by PM+

5. Modification schedule

New restrictions described in this document will be corrected in CA850 V3.46, which will be released at the end of September 2010.

6. List of restrictions

A list of restrictions in the CA850, including the revision history and detailed information, is described on attachment 1, and a list of restrictions in other package tools is described on attachment 2.

7. Document revision history

V850 C Compiler Package CA850 - Usage Restrictions

Document Number	Issued on	Description
SBG-TT-0003-E	November 2, 2001	Newly created.
SBG-TT-0064-E	February 5, 2002	Addition of new restrictions (No. 68 to No. 71)
SBG-TT-0074-E	March 7, 2002	Addition of new restriction (No. 72)
SBG-TT-0154-E	July 12, 2002	Addition of new restrictions (No. 73 to No. 79)
SBG-TT-0218-E	October 10, 2002	Correction of restrictions (No. 2, No. 3, No. 14, No. 28, No. 43, No. 44, No. 60, No. 71, No. 73, No. 74, No. 75, No. 76, No. 77, No. 78, and No. 79)
SBG-DT-03-0027-E	January 31, 2003	Addition of new condition for restriction (No. 56) Addition of new restrictions (No. 80 to No. 82)
SBG-DT-03-0167-E	June 10, 2003	Addition of new restrictions (No. 83 to No. 87) Addition of new restrictions to other package tools (No. 1 and No. 2)
SBG-DT-03-0218-E	July 23, 2003	Correction of restrictions (No. 9, No. 10, No. 11, No. 19, No. 36, No. 37, No. 56, No. 80, No. 81, No. 83, No. 84, No. 85, No. 86, No. 87, and a part of No. 61)
SBG-DT-03-0254-E	September 26, 2003	Correction of erroneous descriptions • [Correction] in No. 9, No. 10, No. 11, No. 19, No. 36, and No. 37
SBG-DT-04-0005	January 9, 2004	Addition of new restrictions (No. 88 and No. 89)
SBG-DT-04-0119	March 24, 2004	Addition of new restriction (No. 90)
ZBG-CD-04-0009	May 28, 2004	Correction of restrictions (No. 1, No. 35, No. 59, No. 61, No. 88, No. 89, and No. 90) Addition of new restriction (No. 91)
ZBG-CD-04-0070	September 22, 2004	Addition of new restriction (No. 92)
ZBG-CD-05-0026	March 30, 2005	Addition of new restriction (No. 93) Addition of new restriction to other package tools (No. 3) Correction of restrictions (No. 47, No. 48, No. 91, No. 92) Correction of erroneous description (No. 13)
ZBG-CD-05-0062	June 29, 2005	Addition of new restrictions (No. 94 to No. 95) Correction of erroneous description (No. 7)
ZBG-CD-05-0077	September 1, 2005	Addition of new restrictions to other package tools (No. 4 to No. 6)
ZBG-CD-05-0112	December 5, 2005	Addition of new restriction (No. 97) Addition of new restriction to other package tools (No. 7)
ZBG-CD-06-0044	May 31, 2006	Addition of new restriction to other package tools (No. 8) Correction of restrictions (No. 94 and No. 97) Correction of restrictions to other package tools (No. 3 to No. 7)
ZBG-CD-07-0067	September 27, 2007	Addition of new restrictions (No. 98 to No. 103)
ZBG-CD-09-0025	May 21, 2009	Addition of new restrictions (No. 104 to No. 110) Addition of new restriction to other package tools (No. 9) Correction of restrictions (No. 93, No. 98 to No. 103) Correction of restriction to other package tools (No. 8)
ZBG-CD-10-0030	Sep 13, 2010	Addition of new restrictions (No. 111 and No. 112) Correction of restrictions (No. 104 to No. 110) Correction of restriction to other package tools (No. 9)

List of Usage Restrictions in CA850

1. Product History

No.	Restrictions and Changes/Additions to Specifications	Version									
		V2.41	2.50	2.6x	2.70	2.72	3.00	3.10	3.20	3.30	3.4x
1	Restriction on file name including space	×	×	×	○	○	○	○	○	○	○
2	Restriction on length of assembly source	×	○	○	○	○	○	○	○	○	○
3	Restriction on length of variable name	×	○	○	○	○	○	○	○	○	○
4	Restriction on precision during floating-point constant operation	×	×	×	×	×	×	×	×	×	×
5	Restriction on invalid processing of specific constant operation (Windows version only)	○	○	○	○	○	○	○	○	○	○
6	Restriction on structures used with conditional operator in function parameters	×	×	×	×	×	×	×	×	×	×
7	Restriction on indirect function calling	×	×	×	×	×	×	×	×	×	×
8	Restriction involving meaningless function definition	×	×	×	×	×	×	×	×	×	×
9	Restriction on number after preprocessor directive	×	×	○	○	○	○	○	○	○	○
10	Restriction on invalid identifier after preprocessor directive	×	×	○	○	○	○	○	○	○	○
11	Restriction on declaring `union` with `struct`	×	×	○	○	○	○	○	○	○	○
12	Restriction on extra `()` in function prototype	×	×	×	×	×	×	×	×	×	×
13	Restriction on using `extern` when defining structure	○	○	○	○	○	○	○	○	○	○
14	Restriction on initialization with character string	×	○	○	○	○	○	○	○	○	○
15	Restriction on `switch` statement at the end of infinite loop	○	○	○	○	○	○	○	○	○	○
16	Restriction on debugging information symbol	○	○	○	○	○	○	○	○	○	○
17	Restriction on successive assignment expressions	○	○	○	○	○	○	○	○	○	○
18	Restriction on register assignment of runtime library	○	○	○	○	○	○	○	○	○	○
19	Restriction on section file and global variables	×	×	○	○	○	○	○	○	○	○
20	Restriction on system call `ext_tsk`	○	○	○	○	○	○	○	○	○	○
21	Restriction on section allocation	×	×	×	×	×	×	×	×	×	×
22	Restriction on section file with variable defined in assembly source	×	×	×	×	×	×	×	×	×	×
23	Restriction on section file with tentative definition of external variables of same name in multiple files	×	×	×	×	×	×	×	×	×	×
24	Restriction on operation including pointer constant	○	○	○	○	○	○	○	○	○	○
25	Restriction on accessing bits of the peripheral I/O register	○	○	○	○	○	○	○	○	○	○
26	Restriction on `volatile` specification for nested structures and unions	○	○	○	○	○	○	○	○	○	○
27	Restriction on `NOT` operator used with constant 0	○	○	○	○	○	○	○	○	○	○
28	Restriction on assembler optimization in V850E mode	×	○	○	○	○	○	○	○	○	○
29	Restriction on specifying optimization option	×	×	×	×	×	×	×	×	×	×
30	Restriction on object file size during optimization	×	×	×	×	×	×	×	×	×	×

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

No.	Restrictions and Changes/Additions to Specifications	Version									
		V2.41	2.50	2.6x	2.70	2.72	3.00	3.10	3.20	3.30	3.4x
31	Restrictions on debugging during optimization	×	×	×	×	×	×	×	×	×	×
32	Restriction on using `_rcopy` function in loop	○	○	○	○	○	○	○	○	○	○
33	Restriction on address of structure member	×	×	×	×	×	×	×	×	×	×
34	Restriction on bit field	×	×	×	×	×	×	×	×	×	×
35	Restriction on `.option nomacro` pseudo instruction	×	×	×	○	○	○	○	○	○	○
36	Restriction on specifying only ~ (tilde) in macro	×	×	○	○	○	○	○	○	○	○
37	Restriction on ~ (tilde) when macro is nested	×	×	○	○	○	○	○	○	○	○
38	Restriction on invalid instruction `sal`	○	○	○	○	○	○	○	○	○	○
39	Restriction on file name for linker directive file	○	○	○	○	○	○	○	○	○	○
40	Restriction on referencing specific symbols and reserved symbols in C source	×	×	×	×	×	×	×	×	×	×
41	Restriction on specifying linker option `-r` (applies to UNIX version CA850 only)	○	○	○	○	○	○	○	○	○	○
42	Restriction on warning message related to EP-relative segment	○	○	○	○	○	○	○	○	○	○
43	Restriction on warning message related to segment	×	○	○	○	○	○	○	○	○	○
44	Restriction on object file name in archive file	×	○	○	○	○	○	○	○	○	○
45	Restriction on size of `rompsec` section displayed by `-m` option of romp850	○	○	○	○	○	○	○	○	○	○
46	Restriction on output file path specification option of performance checker	×	–	–	–	–	–	–	–	–	–
47	Restriction on output file path specification option of cross-reference tool	×	×	×	×	×	○	○	○	○	○
48	Restriction on output file path specification option of memory layout visualization tool	×	×	×	×	×	○	○	○	○	○
49	Restriction on 1-bit manipulation using &, `|`, or ~	○	○	○	○	○	○	○	○	○	○
50	Restriction on `unsigned short` in structure packing	○	○	○	○	○	○	○	○	○	○
51	Restriction on assignment of a union including indirect assignment	○	○	○	○	○	○	○	○	○	○
52	Restriction on creating a relinkable object	○	○	○	○	○	○	○	○	○	○
53	Restriction on `-Wi,-O4` specifications for `switch` statement	○	○	○	○	○	○	○	○	○	○
54	Restriction on global and static functions having the same name	○	○	○	○	○	○	○	○	○	○
55	Restriction on string literal in a function	○	○	○	○	○	○	○	○	○	○
56	Restriction on variable initialization	×	×	○	○	○	○	○	○	○	○
57	Restriction on declaring a pointer to a function that has a structure as its parameter	○	○	○	○	○	○	○	○	○	○
58	Restriction on label subtraction in `sld` or `sst` instruction displacement	○	○	○	○	○	○	○	○	○	○
59	Restriction on internal ROM checking by ROMization processor	×	×	×	○	○	○	○	○	○	○
60	Restriction on temporary file directory	×	○	○	○	○	○	○	○	○	○

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

No.	Restrictions and Changes/Additions to Specifications	Version									
		V2.41	2.50	2.6x	2.70	2.72	3.00	3.10	3.20	3.30	3.4x
61	Restriction on multibyte characters in a command file	×	×	○	○	○	○	○	○	○	○
62	Restriction on structure pointer for structure packing	○	○	○	○	○	○	○	○	○	○
63	Restriction on assignment to and reference of a structure	○	○	○	○	○	○	○	○	○	○
64	Restriction on optimization for two functions	○	○	○	○	○	○	○	○	○	○
65	Restriction on allocating `unsigned char` variable to `tidata_word` section	○	○	○	○	○	○	○	○	○	○
66	Restriction on constant expression in preprocessing	○	○	○	○	○	○	○	○	○	○
67	Restriction whereby code is moved due to assembler optimization	○	○	○	○	○	○	○	○	○	○
68	Restriction whereby the compiler stops during optimization	○	○	○	○	○	○	○	○	○	○
69	Restriction on project manager during building	○	○	○	○	○	○	○	○	○	○
70	Restriction on `static` function written in `#pragma section`	○	○	○	○	○	○	○	○	○	○
71	Restriction on pointer constant when accessing address with offset	×	○	○	○	○	○	○	○	○	○
72	Restriction on optimization for `if` statement	○	○	○	○	○	○	○	○	○	○
73	Restriction on optimization for `switch` statement	×	○	○	○	○	○	○	○	○	○
74	Restriction on initialization of bit field	×	○	○	○	○	○	○	○	○	○
75	Restriction on initialization of automatic array	×	○	○	○	○	○	○	○	○	○
76	Restriction on `ov` flag in machine-dependent optimization module	×	○	○	○	○	○	○	○	○	○
77	Restriction on shift in machine-dependent optimization module	×	○	○	○	○	○	○	○	○	○
78	Restriction on union containing integer and pointer	×	○	○	○	○	○	○	○	○	○
79	Restriction on compound assignment of values to bit field members when structured packing is specified	×	○	○	○	○	○	○	○	○	○
80	Restriction on optimization for union containing integer and pointer	×	×	○	○	○	○	○	○	○	○
81	Restriction on packed structure parameter	×	×*	○	○	○	○	○	○	○	○
82	Restriction on data swap include function	×	○	○	○	○	○	○	○	○	○
83	Restriction on inlining a function with structure as parameter	×	×*	○	○	○	○	○	○	○	○
84	Restriction on function pointer	×	×*	○	○	○	○	○	○	○	○
85	Restriction on included function `__sasf()`	–	×	○	○	○	○	○	○	○	○
86	Restriction on conversion from `float` to `short` or `unsigned short`	×	×*	○	○	○	○	○	○	○	○
87	Restriction on interrupt function	×	×*	○	○	○	○	○	○	○	○
88	Restriction on generating code for complex expressions	×	×*	×*	○	○	○	○	○	○	○
89	Restriction on parameter of inline function	×	×*	×*	○	○	○	○	○	○	○
90	Restriction on optimization for loop processing	×	×*	×*	○	○	○	○	○	○	○

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

No.	Restrictions and Changes/Additions to Specifications	Version									
		V2.41	2.50	2.6x	2.70	2.72	3.00	3.10	3.20	3.30	3.4x
91	Restriction that an undefined-symbol error occurs due to `if` and `goto` statements	–	–	–	×	○	○	○	○	○	○
92	Restriction on flash/external ROM re-link function during linking	–	–	–	×*	○	○	○	○	○	○
93	Restriction whereby error E2288 is output	–	×	×	×	×	×	×	○	○	○
94	Restriction on security ID and option byte	–	–	×	×	×	×	○	○	○	○
95	Restriction on floating point constants and integer type	×	×	×	×	×	×	×	×	×	×
96	Restriction on input conversion for I/O function in standard library	×	×	×	×	×	×	×	×	×	×
97	Restriction on assignment in machine-dependent optimization module	×	×*	×*	×*	×*	×*	○	○	○	○
98	Restriction whereby data in an automatic variable area is illegally deleted	×	×*	×*	×*	×*	×*	×*	○	○	○
99	Restriction on address specification with `-U` option of hx850	×	×	×	×	×	×	×	○	○	○
100	Restriction whereby `strcmp()` and `strncmp()` operations differ	×	×	×	×	×	×	×	○	○	○
101	Restriction on string literal with 65 or more characters	×	×	×	×	×	×	×	○	○	○
102	Restriction on optimization with assembler	–	×	×	×	×	×	×	○	○	○
103	Restriction whereby an assignment statement for an automatic variable whose address is acquired by a function is deleted illegally	×	×*	×*	×*	×*	×*	×*	○	○	○
104	Incorrect number of loop executions	×	×*	×*	×*	×*	×*	×*	×*	×*	○
105	Incorrect string literals	×	×*	×*	×*	×*	×*	×*	×*	×*	○
106	Restriction involving format specifiers for the `sscanf`, `fscanf`, and `scanf` functions	×	×	×	×	×	×	×	×	×	○
107	Restriction involving the character string parameter for the `atoi`, `atol`, `strtol`, and `strtoul` functions	×	×	×	×	×	×	×	×	×	○
108	Restriction involving initialization of a structure that includes a bit field among its members	×	×*	×*	×*	×*	×*	×*	×*	×*	○
109	Restriction involving nested conditionally assembled pseudo instructions	×	×	×	×	×	×	×	×	×	○
110	Restriction involving assignment within `switch` and `if` statements	×	×*	×*	×*	×*	×*	×*	×*	×*	○
111	Restriction on incorrect compare operation optimization while casting	○	○	○	×*	×*	×*	×*	×*	×*	×*
112	Restriction on incorrect move optimization of assignment sentence	×	×*	×*	×*	×*	×*	×*	×*	×*	×*

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

2. Details of Usage Restrictions

No. 1 Restriction on file name including space

[Description]

A file or directory name including a space must not be used.

[Workaround]

Do not use a name or directory name including a space.

[Correction]

This restriction has been removed in Ver. 2.70.

No. 2 Restriction on length of assembly source

[Description]

If one line of the assembly source output from a C source program handling a very long variable name exceeds 1,024 characters, an error occurs.

[Workaround]

Shorten the variable name to keep the length of each line to within 1,023 characters.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 3 Restriction on length of variable name

[Description]

If a variable name of 1,022 characters is specified with the `-g` option and the source file is compiled, an error occurs in the assembler because the assembler handles this variable name as 1,023 characters.

[Workaround]

Shorten the variable name.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 4 Restriction on precision during floating-point constant operation

[Description]

If a floating-point operation that might be inadvertently executed is specified for compilation that involves casting to an integer, the precision drops very slightly, and the value might become invalid as a result of casting to an integer. No problem occurs if floating point data is handled without casting it.

Example:

```
(long)(1.12 * 100);
```

[Workaround]

Change the above statement as follows:

```
float  f = 1.12;
(long)(f * 100);
```

Or,

```
float  f;
(long)(f = 1.12 * 100);
```

[Correction]

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

No. 5 Restriction on invalid processing of specific constant operation (Windows version only)

[Description]

If the expression `−2147483648/−1` or `−2147483648%-1` is generated as a result of optimization flow analysis, a Windows dialog box saying "The program has performed an invalid operation and will be shut down." is displayed during compilation.

Example:

```
void f(){
    int int_min = -2147483648;
    unsigned int ans1 = int_min/-1;
    unsigned int ans2 = int_min%-1;
}
```

[Workaround]

Change the above code as follows:

(1) Directly write the calculation result.

```
void f(){
     unsigned int ans1 = 2147483648;
     unsigned int ans2 = 0;
}
```

(2) Assign `−2147483648` to an external variable other than a `const` variable.

```
int int_min = -214783648;
void f(){
     unsigned int ans1 = int_min/-1;
     unsigned int ans2 = int_min%-1;
}
```

[Correction]

This restriction has been removed in Ver. 2.40.

No. 6 Restriction on structures used with conditional operator in function parameters

[Description]

The correct branch code is not generated if a structure used with a conditional operator in a parameter.

Example:

```
typedef struct {int i;}S;
S ss1, ss2;
int j;
void  func(){
     func_call((j>10)?ss1:ss2);
}
```

[Workaround]

Change the above statement to the following `if` statement:

```
if (j > 10){
     func_call(ss1);
}else {
     func_call(ss2);
}
```

[Correction]

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

No. 7 Restriction on indirect function calling

[Description]

If an indirect function call requires an offset, an internal compiler error occurs.

Message:

```
C2000: internal: gen_binary(): OP_CALL : left-child's operator is wrong
C5211: syntax error at line <num> in intermediate file
```

Example:

```
struct S {;
    int dummy;
    int func_body[0x100];
} sobj;
void f() {
    ((void(*)())sobj.func_body)();
}
```

[Workaround]

Separate the offset calculation from the call as follows:

```
void f(){
     void (*fp)() = (void(*)())&sobj.func_body;
     fp();
}
```

[Correction]

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

No. 8 Restriction involving meaningless function definition

[Description]

An error is not output for a meaningless function definition.

Example:

```
typedef int INTFN();
 INTFN f{return(0);}
```

[Workaround]

Avoid writing meaningless function definitions.

[Correction]

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

No. 9 Restriction on number after preprocessor directive

[Description]

An error is not output for a number specified after a preprocessor directive.

Example: `#if0`

[Workaround]

Avoid writing the code such as above.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 10 Restriction on invalid identifier after preprocessor directive

[Description]

An error is not output for an invalid identifier specified after a preprocessor directive.

Example: `#if $$$`

[Workaround]

Avoid writing the code such as above.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 11 Restriction on declaring `union` with `struct`

[Description]

An error is not output if a `union` is declared with `struct`.

Example:

```
union uu { char a; int b; };
struct uu uu;
```

[Workaround]

Avoid writing the code such as above.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 12 Restriction on extra ( ) in function prototype

[Description]

A syntax error is output for an extra ( ) in a function prototype.

Example:

```
typedef int Int;
void f1((Int));
```

[Workaround]

Modify the code as follows:

```
typedef int Int;
void f1(Int);
```

[Correction]

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

No. 13 Restriction on using `extern` when defining structure

[Description]

A syntax error is output when using `extern` to define a structure.

Example: `extern struct tag { int i; };`

[Workaround]

Remove `extern`.

```
struct tag { int i;};
```

[Correction]

This restriction has been removed in Ver. 2.20.

No. 14 Restriction on initialization with character string

[Description]

Because the code of a string literal is not output for a character string cast to a data type other than `char` or `unsigned char`, an internal error is output and the compiler is terminated.

Message:

```
C7308: undefined block number(xx) reference in opcode(xx)
```

Example:

```
struct S {
    long l;
} sobj = { (long)”string”};
```

[Workaround]

Change the code as follows:

```
char c[] = “string”;
struct S {
    long l;
} sobj = { (long)c};
```

[Correction]

This restriction has been removed in Ver. 2.50.

No. 15 Restriction on `switch` statement at the end of infinite loop

[Description]

If an infinite loop ends with a `switch` statement when `-Ot` or `-g` is specified (including when the statements following the `switch` statement are deleted as unnecessary processing), an internal error is output. This occurs only when a `switch` statement is used to branch from a `switch` statement to a jump table.

Message:

```
C6101: illegal intermediate file(operator)
```

[Workaround]

Do one of the following:

(1) Specify the `-Xcase=ifelse` option to avoid outputting the code to branch from a `switch` statement to a jump table.

(2) Specify the `-Xcase=binary` option to avoid outputting the code to branch from a `switch` statement to a jump table.

(3) Specify `_asm(“\n”)` at the end of the infinite loop.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 16 Restriction on debugging information symbol

[Description]

If optimization is not specified and the output of debugging information `-g` is specified, the linker outputs the message “undefined symbol” if an external or static variable is allocated to a register and the allocated area ends with either of the following:

- Multiplication by $2^n + 1$
- Multiplication by $2^n - 1$

[Workaround]

Do one of the following:

(1) Specify an optimization option (`-Os` or `-Ot`).

(2) Declare the external or static variable as `volatile`.

[Correction]

This restriction has been removed in Ver. 2.40.

No. 17 Restriction on successive assignment expressions

[Description]

When the following conditions are met, the required code is inadvertently deleted.

(1) There are insufficient temporary registers in the global optimization processing that occurs when the `-Os` or `-Ot` option is specified

(2) If (1) has occurred, temporary registers are saved to the stack and restored from the stack when required in the code generation phase after global optimization. If the C code below is detected at this time, the restore processing code is not generated.

(3) This bug occurs only when conditions (1) and (2) are both met, because, during the machine-dependent optimization phase after the code generation phase, necessary code is treated as an unnecessary instruction string and deleted.

Example:

```
<val> <op2>= (<val_1>) <op1> <imm_1>;
<val> <op2>= (<val_2>) <op1> <imm_2>;
          :
<val> <op2>= (<val_n-1>) <op1> <imm_n-1>;
<val> <op2>= (<val_n>) <op1> <imm_n>;
```

Explanation:

<*op1*>, <*op2*>: Operators

<*val*>: Variable

<*imm*>: Immediate value

<*val_n*>: Variable or variable + operator

<*n*>: 5 to 10 or more (depends on function scale and variable usage conditions)

op1: +, &, |, ^, -

"-" is only applicable in cases when the `add` instruction is used with sign inversion.

Left-right inversion of the *op1* operand might also occur.

op2: +, *, &, |, ^,

val: Variable identical to non-`volatile int`, `unsigned`, `long`, or `unsigned long`

Example:

```
x = a;
x |= b & 0x000000f0;
x |= c & 0x00000f00;
x |= d & 0x0000f000;
x |= e & 0x000f0000;
x |= f & 0x00f00000;  /* Code is not output */
x |= g & 0x0f000000;  /* Code is not output */
x |= h & 0xf0000000;  /* Code is not output */
```

[Workaround]

Do one of the following:

(1) Do not specify optimization options (`-Os` or `-Ot`) in source files containing the above C code.

(2) Modify the C code as follows:

```
x  = a              |
     b & 0x000000f0 |
     c & 0x00000f00 |
     d & 0x0000f000 |
     e & 0x000f0000 |
     f & 0x00f00000 |
     g & 0x0f000000 |
     h & 0xf0000000;
```

[Correction]

This restriction has been removed in Ver. 2.40.

No. 18 Restriction on register assignment of runtime library

[Description]

Because the multiplication/division operation is output by the runtime library, the register assignment is invalid, possibly causing the value of the `r6` or `r7` register to be corrupted.

This is true when the following five conditions are met:

(1) A V850 core device is specified.

(2) `-Os` or `-Ot` is specified.

(3) Either the multiplier in a multiplication expression or the divisor in a division expression is one of the following:

- An addition or subtraction expression involving a `signed char` or `unsigned char` variable and a constant that is in the range −16384 to 16383.
- A multiplication expression involving one of the following combinations where one operand is a variable and the other is a constant that is in the range for the specified data type:
  - `signed char` and `signed char`
  - `signed char` and `unsigned char`
  - `unsigned char` and `signed char`

(4) The constant in (3) is assigned to a register.

(5) The variable or constant of the multiplication or division expression in (3) is assigned to `r6` or `r7` when the operation is performed.

Example:

```
char s,t;
void func(int a, int b){
int c;
    c = ( s - 0xaa) * t;
    if ( b < 0xaa) c = 0;
    func2(c);
}
```

[Workaround]

Specify one of the following options:

- `-Wo,-preg0` (specifies that registers not be assigned to parameter registers)
- `-Wo,-XTm` (specifies that `mulh`/`divh` not be output)

[Correction]

This restriction has been removed in Ver. 2.40.

No. 19 Restriction on section file and global variables

[Description]

A warning message might not be output if a section mismatch occurs for a global variable in the section file and source file.

[Workaround]

Modify the section file so as not to cause inconsistency between sections.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 20 Restriction on system call `ext_tsk`

[Description]

When a task of a real-time OS is written by specifying `#pragma rtos_task` and the system call `ext_tsk` is used in the task, an error might occur if the task is compiled with the `-g` option specified, resulting in abnormal termination.

Example:

```
extern  int  func1(void);
#pragma rtos_task usrtsk
void usrtsk(int Param)
{
   int  i;
   i = func1();
   ext_tsk();
}
```

[Workaround]

Use a function pointer to indirectly call `ext_tsk`.

```
extern  int  func1(void);
#pragma rtos_task usrtsk
void usrtsk(int Param)
{
   int  i;
   void  (*ext_tsk_ptr)() = ext_tsk;
   i = func1();
   (*ext_tsk_ptr)();
}
```

[Correction]

This restriction has been removed in Ver. 2.30.

No. 21 Restriction on section allocation

[Description]

In the `tidata` section allocation specification by the `#pragma section` directive or in a `char` array of a structure or access to a `char` member specified by sf850 to be located in the `tidata` section, a linker error occurs if the displacement value of the `sst` or `sld` instruction used to access the array or member is exceeded.

[Workaround]

Do one of the following:

(1) Do not use the `char` member or `char` array of a structure that causes the linker error.

(2) Do not assign the `char` member or `char` array of a structure that causes the linker error to the `tidata` section.

[Correction]

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

No. 22　Restriction on section file with variable defined in assembly source

[Description]

In an application where a variable is defined in the assembly source and that variable is referenced in the C source, an error occurs during linking if the section file is generated by sf850.

[Workaround]

Delete the variable in the assembly source from the section file.

[Correction]

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

No. 23　Restriction on section file with tentative definition of external variables of same name in multiple files

[Description]

If a section file is generated by sf850 when the tentative definitions of external variables of the same name are in multiple files, symbols might be defined in duplicate during linking.

Message:

```
ld850: fatal error: symbol “_xxxx” multiply defined.
```

[Workaround]

If multiple tentative definitions of external variables of the same name exist, be sure to declare the variables using `extern` in the file that references them.

[Correction]

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

No. 24　Restriction on operation including pointer constant

[Description]

Invalid code is output as a result of an operation including a pointer constant and multiple expressions.

Example:

```
int i1;
int i2;
charc;
c = (char *)10 + il + i2 + 10;
```

[Workaround]

Enclose the expressions in parentheses and then cast the pointer.

Example:

```
c = (char *)(10 + il + i2 + 10);
```

[Correction]

This restriction has been removed in Ver. 2.40.

No. 25　Restriction on accessing bits of the peripheral I/O register

[Description]

The code generator (cgen) might output invalid code when a variable or expression that is assigned to a bit of a peripheral register is assigned to a register.

Example:

```
void func(unsigned char a)
{
   int i = 8;
   while(i){
        P0.0 = a;
        a >>= 1;
        i--;
   };
}
```

[Workaround]

Assign values instead of variables to a bit of the peripheral I/O register.

Example:

```
void func(unsigned char a)
{
   int i = 8;
   while(i){
        if(a & 0x1){
              P0.0 = 1;
        } else {
              P0.0 = 0;
        }
        a >>= 1;
        i--;
   };
}
```

[Correction]

This restriction has been removed in Ver. 2.40.

No. 26 Restriction on `volatile` specification for nested structures and unions

[Description]

If a structure (or union) member nested in a structure (or union) is accessed using a pointer declared as `volatile` and if it is written as the left operand of a compound assignment operator, `volatile` becomes invalid.

Example:

```
typedef union {
     struct {
          unsigned char a0;
          unsigned char a1;
          unsigned short a2;
     }a;
     unsigned long c;
}T;
#define s(*(volatile T *)(0x0100000)
s.a.a1 |= 0x1; {
```

[Workaround]

Change the compound assignment operator to a simple assignment operator.

Example:

```
    s.a.a1 = s.a.a1 | 0x1;
```

[Correction]

This restriction has been removed in Ver. 2.40.

No. 27 Restriction on `NOT` operator used with constant 0

[Description]

The `not` instruction is output by the `NOT` operator if it is used with the constant 0 when the optimization option `-Os` or `-Ot` is specified.

Example:

```
    a = 0;
    a ^= b;
```

[Workaround]

Modify the code so that the same value is assigned to the variable that would be when using the `NOT` operator with the constant 0.

```
    a = b;
```

[Correction]

This restriction has been removed in Ver. 2.40.

No. 28 Restriction on assembler optimization in V850E mode

[Description]

Although V850E is supported, optimization of the assembler is not executed in the V850E mode.

[Workaround]

There is no workaround.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 29 Restriction on specifying optimization option

[Description]

If the level of the optimization option specified during compilation is raised, the number of phases to be executed during compilation (such as the optimization and compilation) increases. If the `-Ot` option is specified, the size of the intermediate file created between these phases increases, sometimes causing a fatal error in some cases.

[Workaround]

Lower the optimization level by using `-Os`.

[Correction]

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

No. 30 Restriction on object file size during optimization

[Description]

When an optimization option is specified, the size of an object file including debugging information might significantly increase.

[Workaround]

Either lower the optimization level or use the `-g` option, which outputs the debugging information only to the file to be debugged.

[Correction]

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

No. 31 Restrictions on debugging during optimization

[Description]

The following restrictions apply when an optimization option is specified for debugging the source:

(1) When the value of a variable is referenced, a temporary value might be obtained in the middle of calculation instead of the correct value.

(2) If an element of an array, a member of a structure, or a user-defined pointer is assigned to a register, variables might be illegally displayed or modified in the **Variable** window of the debugger.

(3) If an element an automatic array or a member of a structure is not used, the area might be deleted. In this case, variables might be illegally displayed or modified in the **Variable** window of the debugger. The stack might be destroyed when a variable is modified.

[Workaround]

There is no workaround.

[Correction]

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

No. 32 Restriction on using `_rcopy` function in loop

[Description]

When the `_rcopy` function is used in a loop such as a `while` statement, an internal error occurs if the optimization option `-Os` or higher is specified.

Example 1:

```
while (_rcopy(&_S_romp, -1) < 0) {
            ;
}
```

Example 2:

```
for ( ; ; ) {
    ret = _rcopy (&_S_romp, -1);
    if (ret == 0) {
        break;
    }
}
```

[Workaround]

Create a function block that calls the `_rcopy` function and call this function in the loop.

```
void main(void)
{
     while (dummy()< 0){
         ;
     }
}
int dummy(){
     return rcopy(&_S_romp, -1);
}
```

[Correction]

This restriction has been removed in Ver. 2.30.

No. 33 Restriction on address of structure member

[Description]

If either of the conditions below is satisfied when a structure is packed, data access follows the data alignment of the device and the accessed address is masked. As a result, data will be missing or lost when accessing the address of the structure member.

Conditions:

(1) The device does not support misaligned access

(2) The device supports misaligned access but misaligned access is prohibited

Example:

```
struct test {
    char c;     /* offset 0 */
    int  i;     /* offset 1-4 */
} test;
int *ip, i;
void func(){
    i = *ip;    /* Accessed from a masked address */
}
void func2(){
    ip = &(test.i);
}
```

[Workaround]

There is no workaround.

[Correction]

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

No. 34 Restriction on bit field

[Description]

If the width of a bit field is less than the data type of a member when the bit field is accessed during structure packing, the bit field is read as having the width of the data type of that member. Consequently, an area outside the object (an area where there is no data) is also accessed. This access is usually executed correctly but it might be invalid if I/O is mapped.

Example:

```
struct S {
     int  x:21;
} sobj;  /* 3 bytes */
sobj.x = 1;
```

[Workaround]

There is no workaround.

[Correction]

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

No. 35 Restriction on *`.option nomacro`* pseudo instruction

[Description]

The `mov` instruction in the specified range of the *`.option nomacro`* pseudo instruction might result in a syntax error.

Example:

```
    .text
    .option nomacro
    mov 0x12345678, r10
```

[Workaround]

Remove the relevant `mov` instruction from the specified range of the `.option nomacro` pseudo instruction.

[Correction]

This restriction has been removed in Ver. 2.70.

No. 36 Restriction on specifying only ~ (tilde) in macro

[Description]

The assembler is terminated abnormally if only ~ is specified in a macro.

Example:

```
    .macro MACRO
    ~
    .endm
    MACRO
```

[Workaround]

Do not specify the above sort of code.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 37 Restriction on ~ (tilde) when macro is nested

[Description]

'~ (tilde)' might cause an error when a macro is nested.

Example:

```
    .macro inc.w        var, reg
           ld.w         var, reg
           add          1, reg
           st.w         reg, var
    .endm
    .macro aaa          rnum
           ld.w         [r~rnum], r30
           add          1, r30
           st.w         r30, [r~rnum]
    .endm
    .macro bbb          rnum
           inc.w        [r~rnum], r30
    .endm
    .set   pathreg, 25
    .text
    .globl f
  f:
    aaa    $pathreg     -- OK
    bbb    $pathreg     -- Error

    .macro MACRO
```

[Workaround]

Do not nest a macro.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 38 Restriction on invalid instruction `sal`

[Description]

If a `sal` instruction, which does not exist in the device, is specified, the assembler does not output an error but outputs invalid code. The compiler does not output this instruction.

Example: `sal    3, r10`

[Workaround]

There is no workaround. Do not specify the `sal` instruction in source code.

[Correction]

This restriction has been removed in Ver. 2.40.

No. 39 Restriction on file name for linker directive file

[Description]

In the file name specification of a mapping directive, a linker error occurs if a file name specified in braces begins with A, F, H, L, or V (or the lowercase equivalent), and the second character is a number.

Message:

```
ld850: syntax error: line num: string is illegal in file specification field.
```

[Workaround]

Use file names that do not satisfy the above conditions.

[Correction]

This restriction has been removed in Ver. 2.30.

No. 40 Restriction on referencing specific symbols and reserved symbols in C source

[Description]

Target-specific symbols such as `_gp_DATA` and reserved symbols such as `_stext` cannot be referenced in the C source.

[Workaround]

Do not use a target-specific symbol or reserved symbol in the C source.

[Correction]

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

No. 41 Restriction on specifying linker option `-r` (applies to UNIX version only)

[Description]

If the `-r` option is specified during linking in the UNIX version CA850 and the following conditions are satisfied, a core dump might occur:

(1) A TEXT section does not exist.

(2) Data is allocated to the `.sdata` or `.sbss` section.

(3) A section with an allocation attribute (?A is specified by a link directive) exists as well as an odd-sized section without an initial value ($NOBITS).

(4) A relocatable object is created using the `-r` option of the ld850, and linking is performed again.

[Workaround]

Do one of the following:

(1) Do not specify the `–r` option of the ld850.

(2) Specify an even number as the size of a section without an initial value.

Example:

Before correction:

```
#pragma section sidata begin
static unsigned char a;
#pragma section sidata end
static unsigned long b;
```

After correction:

```
#pragma section sidata begin
static unsigned char a;
static unsigned char dummy;      ← Add a dummy
#pragma section sidata end
static unsigned long b;
```

(3) Initialize the `char` variable.

Example:

Before correction:

```
#pragma section sidata begin
static unsigned char a;
#pragma section sidata end
static unsigned long b;
```

After correction:

```
#pragma section sidata begin
static unsigned char a = 0;      ← Initialize the variable
#pragma section sidata end
static unsigned long b;
```

[Correction]

This restriction has been removed in Ver. 2.30.

No. 42 Restriction on warning message related to EP-relative segment

[Description]

The ld850 outputs a warning message if the following conditions are satisfied:

(1) A V850E device is specified.

(2) The start address of the internal RAM in the device is not 0x03xxxxxx.

(3) An SEDATA or SIDATA segment exists.

Message:

```
ld850: warning: segment "SIDATA" (0x03xxxxxx-0x03xxxxxx) must be in
EP-relative-address-able range (0x0xxxxxxxc000-0x0xxxxxxx).
```

[Workaround]

Check the link map and confirm that SEDATA and SIDATA segments are allocated before and after the internal RAM address, and ignore the above message.

[Correction]

This restriction has been removed in Ver. 2.40.

No. 43 Restriction on warning message related to segment

[Description]

If a device with a 28-bit space, such as the V850E/MA1 or V850E/IA1, is specified and a section is allocated, and if the value of an address masked with 26 bits overlaps an allocation-prohibited address, the warning message below is output. Although a message is output, the created object does not contain errors.

Message:

```
ld850: warning: segment "XXXX" overflowed highest address of target machine.
```

Example:

If the internal ROM ranges from 0x00000000 to 0x0003ffff and external memory starts from 0x00100000, allocating the area of 0x00040000 to 0x000fffff is prohibited.

For the following areas, however, the above message is irrelevant because these areas can be allocated:

AREA1: 0x?4040000 to 0x?40fffff

AREA2: 0x?8040000 to 0x?80fffff

AREA3: 0x?c040000 to 0x?c0fffff

Specifically, addresses of AREA1, AREA2, and AREA3 immediately after the internal ROM and before the external memory are output unnecessarily.

[Workaround]

There is no workaround.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 44 Restriction on object file name in archive file

[Description]

If an object file name in the archive file has 15 characters or more, the following problems occur:

(1) Mapping cannot be performed as specified by the linker directive file.

(2) An invalid file name is displayed in the link map and error message.

[Workaround]

Specify an object file name within 15 characters when creating an archive file.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 45 Restriction on size of `rompsec` section displayed by `-m` option of romp850

[Description]

In the memory map displayed using the `-m` option of romp850, the total size of the `rompsec` section is incorrect. The displayed size is larger than the actual size because the size of the copy information section has been added.

[Workaround]

Subtract the size of the copy information section from the size of the `rompsec` section or output the dump information using dump850 to obtain the correct size.

[Correction]

This restriction has been removed in Ver. 2.30.

No. 46 Restriction on output file path specification option of performance checker

[Description]

If the folder to which the analysis result of a specified output file is to be output is specified by the path specification option `-o`, the result is not output to the specified folder but to the current folder (the executed folder). If the `-all` option is specified, the `-o` option outputs the analysis result to the specified folder.

[Workaround]

Execute analysis in the folder to which the result is to be output.

[Correction]

This issue does not apply to Ver. 2.50.

No. 47 Restriction on output file path specification option of cross-reference tool

[Description]

If the folder to which the analysis result of a specified output file is to be output is specified by the path specification option `-o`, the result is not output to the specified folder but to the current folder (the executed folder). If the `-all` option is specified, the `-o` option outputs the analysis result to the specified folder.

[Workaround]

Execute analysis in the folder to which the result is to be output.

[Correction]

This restriction has been removed in Ver. 3.00.

No. 48 Restriction on output file path specification option of memory layout visualization tool

[Description]

If the folder to which the analysis result of a specified output file is to be output is specified by the path specification option `-o`, the result is not output to the specified folder but to the current folder (the executed folder). If the `-all` option is specified, the `-o` option outputs the analysis result to the specified folder.

[Workaround]

Execute analysis in the folder to which the result is to be output.

[Correction]

This restriction has been removed in Ver. 3.00.

No. 49 Restriction on 1-bit manipulation using &, |, or ~

[Description]

When an optimization option `-Os` or `-Ot` is specified, if 1-bit manipulation using &, |, or ~ is performed on a `short int` variable assigned to a register under the following two conditions, invalid code will be output.

(1) The variable is non-`volatile` and automatic (not including static variables and parameters)

(2) A `signed int` or `signed long` variable is declared before the target variable is assigned to the register

Example:

```
void f(void){
   int i;
   short s = 0;
   i = g(&s);
   switch(i){
      case 100:
           s & = 0xfffe;
           break;
      case 200:
```

```
                s |= 1;
                break;
            case 300:
                s ^ = 1;
                break;
        }
        i = i / s;
    }
```

[Workaround]

Specify the `-Wo,-XTb` option. (This option limits the range affected by bit manipulation instructions)

[Correction]

This restriction has been removed in Ver. 2.40.

No. 50 Restriction on `unsigned short` in structure packing

[Description]

Invalid code will be output if the V850E is specified as the target device, structure packing is used, and an `unsigned short` member is assigned to an odd address. This bug does not apply to a `short` member.

Example:

```
        #pragma pack(1)
        struct s {
            unsigned char a;
                    unsigned short b;
        }*x;
        x->b = 512;
```

[Workaround]

Change the `unsigned short` member to a `short` member.

Note that the variables vary depending on the type.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 51 Restriction on assignment of a union including indirect assignment

[Description]

When optimizing the deletion of redundant assignment statements in a basic block, the assignment in (1) is illegally deleted when the assignment instructions are written in the following order:

(1) Assignment for a union that does not use the address operator &

(2) Indirect assignment for an undefined address

The following code might apply to undefined addresses:

- A pointer
- A variable other than a pointer that is used as a pointer by casting
- An absolute address

Remark Undefined address: Because a pointer is a variable, it is not assigned a value just by declaring it. Therefore, the address pointed to by a pointer is also undefined.

(3) Reference or assignment to a member of the union described in (1).

(4) Assignment to a member of the union described in (1)

Example:

```
typedef union{
     unsigned char a[2];
     unsigned short b;
} U;
U t;
unsigned short *p;
t.a[1] = 1;      /* The code on this line is illegally deleted */
*p = 0;
t.b >>= 4;
t.a[1] = 3;
```

[Workaround]

Specify the `-Wo,-Nce` option. (This option suppresses propagation of copying and optimization of deletion of common subexpressions.)

[Correction]

This restriction has been removed in Ver. 2.41.

No. 52 Restriction on creating a relinkable object

[Description]

When sections with the same name that should therefore be relocated exist in each of a number of input files during linking, if they are allocated to different sections using a linker directive file at the stage in which a relinkable object is created by specifying the `-r` option, an object file having invalid relocation information will be created. If this object file is linked, the instructions in the file will be illegally overwritten because the relocation information is wrong.

Example:

```
USRTEXT    : !LOAD ?RX {
     .USRTEXT1 = $PROGBITS ?AX .USRTEXT{a.o(libusr.a)};
     .USRTEXT2 = $PROGBITS ?AX .USRTEXT{b.o(libusr.a)};
     .USRTEXT3 = $PROGBITS ?AX .USRTEXT{c.o(libusr.a)};
     .USRTEXT4 = $PROGBITS ?AX .USRTEXT{d.o(libusr.a)};
};
```

[Workaround]

Specify the linker directive file so that the relocation information section is allocated to each input file separately. The section name of the relocation information is `.rela` + *section-name*, and the section attribute is `$RELA ?N`.

```
USRTEXT    : !LOAD ?RX {
     .USRTEXT1 = $PROGBITS ?AX .USRTEXT{a.o(libusr.a)};
     .USRTEXT2 = $PROGBITS ?AX .USRTEXT{b.o(libusr.a)};
     .USRTEXT3 = $PROGBITS ?AX .USRTEXT{c.o(libusr.a)};
     .USRTEXT4 = $PROGBITS ?AX .USRTEXT{d.o(libusr.a)};
};
     .rela.USRTEXT1 = $RELA ?N .rela.USRTEXT{a.o(libusr.a)};
     .rela.USRTEXT2 = $RELA ?N .rela.USRTEXT{b.o(libusr.a)};
     .rela.USRTEXT3 = $RELA ?N .rela.USRTEXT{c.o(libusr.a)};
     .rela.USRTEXT4 = $RELA ?N .rela.USRTEXT{d.o(libusr.a)};
```

[Correction]

This restriction has been removed in Ver. 2.41.

No. 53 Restriction on `-Wi,-O4` specification for `switch` statement

[Description]

If the `-Wi,-O4` option is specified, the V850E device is specified, and a `switch` statement is used as table jump code[Note], optimization is performed incorrectly and the code is illegally deleted.

Note In the CA850, table jump code is output when the following two conditions are satisfied for a `switch` statement:

- The number of `case` labels is 4 or more.
- The difference between the max. and min. values of the label is up to three times the number of `case` labels.

[Workaround]

Specify the `-Ol` option instead of the `-Wi,-O4` option.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 54 Restriction on global and static functions having the same name

[Description]

Even though global and static functions with the same name exist, a duplicate definition error is not output. Instead, an invalid error message is output.

(1) An internal error (`C7318: illegal label (xxxx)`) is output.

(2) The invalid debugging information symbol (`Gxxxx`) is judged as an undefined-symbol error during linking.

Example:

```
static void func(){}
void func(){}
```

[Workaround]

Do not define global and static functions with the same name.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 55 Restriction on string literal in a function

[Description]

When a string literal is defined in a function, an internal error occurs if its length exceeds 8,200 characters.

Message:

```
C2000: internal: string literal too long
```

Example:

```
void func(void)
{
    char *p = “abcde…”;      /* When exceeding 8200 characters */
}
```

[Workaround]

When defining a string literal, ensure that it does not exceed 8,200 characters.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 56 Restriction on variable initialization

[Description]

When either of the following conditions is satisfied, the error message below is output and compilation might stop.

(1) There are too many variable initializations.

(2) The code includes the debugging information output option `-g` and variable initializations.

A large amount of the following information is included:

- File names (including the names of include files)
- Function names
- Variable names
- Enumeration tag names and enumerator names
- Structure tag names, union tag names, and member names
- `typedef` names
- Label names

Message:

```
C7317: illegal block no (xxxxxxxx)
```

[Workaround]

Divide the file to reduce the number of initializations per compilation, or reduce the amount of the above information.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 57 Restriction on declaring a pointer to a function that has a structure as its parameter

[Description]

When debugging is specified (when the `-g` option is specified), if a pointer to a function that has a structure as its parameter is declared but the structure is not defined, the `.Gxxxx` symbol in the debugging information causes the following error to be output by the linker (id850).

Message:

```
ld850: fatal error: undefined symbol.
```

Example: `void  *(*x)(struct SSS *);`

[Workaround]

Define the structure used before the pointer to the function is declared.

Example:

```
struct SSS {
    unsigned char s;
    unsigned char t;
}sss_obj;
void *(*x)(struct SSS *);
```

[Correction]

This restriction has been removed in Ver. 2.41.

No. 58 Restriction on label subtraction in `sld` or `sst` instruction displacement

[Description]

When the subtraction of one label from another is specified for the displacement in the `sld` or `sst` instruction, and the expression value varies according to instruction unrolling by the assembler, the instruction might be invalid.

The CA850 does not create the code that causes this bug.

Example:

```
LAB1:
    sld.w  LAB2 – LAB1, r10
                 :
    mov     LAB2 – LAB1, r11          -- The instruction might be unrolled.
LAB2:
```

[Workaround]

Write the immediate value instead of a subtraction of one label from another.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 59 Restriction on internal ROM checking by ROMization processor

[Description]

When using the ROMization processor (romp850), the following warning message is output when it should not be, or not output when it should be, depending on whether a `.text` section exists and its allocated address. Note that the output object has no problem.

Message:

```
Warning: rompsec section overflowed highest address of target machine.
```

[Workaround]

Output the map by using the `-m` option of the romp850 and confirm that the `rompsec` section does not exceed the internal ROM. If does not exceed the ROM, ignore the above warning message.

[Correction]

This restriction has been removed in Ver. 2.70.

No. 60 Restriction on temporary file directory

[Description]

If a relative path is specified for the temporary file directory in the project manager, the command file (`pm*.cmd`) for the project manager might remain in the temporary file directory.

[Workaround]

Delete the command file after the project manager is exited.

Do not specify a relative path for the temporary file directory.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 61 Restriction on multibyte characters in a command file

[Description]

When a multibyte character is used in the command files for the following commands, the CA850 fails to identify delimiters in character strings, causing an erro:

(1) as850	(Corrected in Ver. 2.60.)
(2) ar850	(Corrected in Ver. 2.50.)
(3) dis850	(Corrected in Ver. 2.70.)
(4) dump850	(Corrected in Ver. 2.70.)
(5) hx850	(Corrected in Ver. 2.50.)
(6) romp850	(Corrected in Ver. 2.60.)

[Workaround]

Do not use multibyte characters in a command file.

When using the project manager, do not use multibyte characters for options, because they are specified by a command file.

[Correction]

This restriction has been removed in Ver. 2.70.

No. 62 Restriction on structure pointer for structure packing

[Description]

Invalid code might be output when structure packing (`-Xpack`) is used, and

- a structure pointer to be packed is declared as `register`

  or

- the optimization option `-Os` or `-Ot` is specified and the structure pointer to be packed is assigned to the register.

Example:

```
struct test {
    struct test *next ;
} ;

struct test list[20];
struct test *Head = &list[0] ;

func() {
    int i;
    struct test *pp ;

    pp = Head ;
    for ( i = 0 ; pp != 0 ; i++ ) {
        pp = pp->next ;/* The code on this line is invalid */
    }
}
```

[Workaround]

Delete the `register` declaration for the structure pointer and do not specify the optimization option.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 63 Restriction on assignment to and reference of a structure

[Description]

When the V850E is specified as the target device and the optimization options `-Os` and `-OI` are specified, optimization in which an `unsigned short` structure member is assigned a value or referenced might be executed incorrectly. As a result, the code might be illegally deleted.

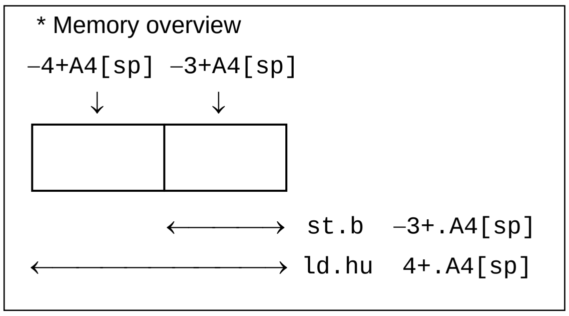
Example 1:

```
unsigned int   Testflg = 0;

struct test {
    unsigned char   b1:3;
    unsigned short  s1:2;
    unsigned short  s2:2;
```

```
          unsigned char b2:2;
          unsigned long l:2;
     } ;

     void func( struct test arg1 )
     {
          struct  test ausrt1 = arg1 ;

          ausrt1.b1 =  2;
          ausrt1.s1 =  2;
          ausrt1.b2 =  2; /* The code on this line is illegally deleted */
          ausrt1.s2 =  2;
          ausrt1.l  =  2;

          if (ausrt1.b2  != 2) {
             Testflg++ ;
          }
     }

     main() {
          struct test  arg;
          func(arg);
     }
```

The following assembler code will be output to the relevant section.

```
  #  :  ausrt1.b2 =  2;
            :
  st.br10, -3+.A4[sp]            ... (1)
  #  :  ausrt1.s2 =  2;
  ld.hu      -4+.A4[sp], r15     ... (2)
  mov        0x40, r16
  and        0xffffff9f, r15
  or         r16, r15
  st.h       r15, -4+.A4[sp]     ... (3)
```



When there is an `st` instruction that writes data to memory (1), if the `ld` instruction that reads data in memory (2) and a code that reads a part of the memory in (1) are successively executed, the memory usage for (2) is illegally checked. As a result, the code in (1) is illegally deleted.

Example 2:

```
     union test {
        char c[4];
        unsigned short us;
        unsigned int ui;
     };

     void func() {
        union test test;
        test.c[1] = test.c[0] = 2;/* The code on this line is illegally deleted */
        test.ui  = ((unsigned)test.us & (unsigned)0xffffe7ff) | 0x1000;
        func3(test);
    }
```

The code in (1) will be illegally deleted in the same manner as in Example 1.

```
#  :  test.c[1] = test.c[2] = 2;
mov     2, r10
st.b    r10, -6+.A4[sp]
st.b    r10, -7+.A4[sp]   … (1)
#  :  test.ui = ((unsigned)test.us & (unsigned)0xffffe7ff) | 0x10000;
ld.hu   -8+.A4[sp], r14
and     0xffffe7ff, r14
or      0x10000, r14
st.w    r14, -8+.A4[sp]
```

[Workaround]

Specify `-Wi,+stld_trans_opt=OFF` to suppress the optimization that causes this bug, or remove the optimization option `-Ol`.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 64 Restriction on optimization for two functions

[Description]

When the optimization option `-Os` or `-Ot` is specified, if the conditions below are satisfied, optimization for two functions might be executed incorrectly. As a result, the code might be illegally deleted.

(1) In the function called first (`f3`), a parameter of the called function (`f2`) is an address (pointer).

(2) In the function called first (`f3`), the value of the variable that assigns an address to the pointer is assigned before calling the function (`f2`).

(3) In the function called next (`f2`), a parameter of the called function (`f1`) is a value.

(4) In the function called next (`f2`), a line or part of a line of the code is moved immediately before the called function due to optimization (such as global optimization or loop optimization).

(5) In the function `f2`, the parameter of the function `f1` is not referenced after the function `f1` is called.

Example:

```
int f1(int i){
    i++;
    return i;
}

void f2(int* p, int i){
    int j = 0;                          … (4)
    if(i == 0){
        volatile int vi = f1(*p);       … (3)
          for(; j < 255; j++)
                 ;
    }
}
```

```
void f3(void){
    int i = 1;          /* The code on this line is illegally deleted */
    int* p = &i;                                … (2)
    f2(p, 0);                                   … (1)
}
```

(A) (1) to (3) correspond to the bug conditions (1) to (3).

(B) Because the code in (4) is moved to the address immediately before (3) by the global optimization, this applies to condition (4).
Because the code in (4) is only used in the `if` statement block, this code is moved to the address immediately before (3) in the `if` statement by optimization.

(C) Because there is no function reference in addresses after (3), this applies to condition (5).

[Workaround]

Specify `-Wp,-f- -Wo,-Ni` to suppress the optimization that causes this bug, or do not specify the optimization option.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 65 Restriction on allocating `unsigned char` variable to `tidata_word` section

[Description]

When the V850E is specified as the target device and an `unsigned char` variable is allocated to the `tidata_word` section, invalid code might be output, causing a linker error.

Example:

```
#pragma section tidata_word begin
char dummy[128];
struct test {
    unsigned char  a;
    unsigned short b;
    unsigned int   c;
} test_u;
#pragma section tidata_word end

unsigned char ua;

void func() {
    ua = test_u.a * ua; /* The code on this line is invalid */
}
```

[Workaround]

Allocate the relevant `unsigned char` variable to the `tidata_byte` section.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 66 Restriction on constant expression in preprocessor directive

[Description]

Processing becomes invalid if a constant expression that includes multiplication is specified for an `#if` preprocessor directive.

Example:

```
#define A 1
#define B 2

void main(){
#if (A * B) == 2
    func1();
#else
    func2(); /* Condition judgment is illegally performed and invalid code
will be output */
#endif
}
```

[Workaround]

There is no workaround.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 67 Restriction whereby code is moved due to assembler optimization

[Description]

When the conditions below are satisfied, the code for rewriting the stack pointer will be moved due to optimization by the assembler, and a malfunction might occur depending on the timing at which interrupt servicing occurs.

Conditions:

(1) The V850E is specified as the target device.
This condition applies to all devices including the V850E if the V850E common object file creation option `-cn` is specified.

(2) The optimization option `-Os` or `-Ot` is specified or the optimization option `-O` is specified as an assembler option.

(3) When the debugging information output option `-g` is not specified.

(4) The CPU bug prevention option (the compiler option -Xv850patch or the assembler option -p) is not specified.

Example:

• Assembler code

```
addi     4, sp, r15
ld.w     0x4[r15], r16
st.w     r16, 0x0[r14]
ld.w     0x8[r15], r17
add      8, sp                - - Stack pointer rewriting code
jmp      [lp]
```

• Assembler code after assembler optimization

```
addi     4, sp, r15
ld.w     0x4[r15], r16
add      8, sp               - - Code moves to this location    ... 1
st.w     r16, 0x0[r14]       - -                                ... 2
ld.w     0x8[r15], r17       - -                                ... 3
jmp      [lp]
```

If interrupt servicing occurs between 1 and 2 or 2 and 3, the subsequent operation will be invalid.

[Workaround]

Specify `+Oa-` as a compiler option or `+O` as an assembler option.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 68 Restriction whereby the compiler stops during optimization

[Description]

When the optimization option `-Os` or `-Ot` is specified, the compiler abnormally stops during optimization for a source file that includes a function that satisfies the following conditions:

(1) Only directly accesses to an address

(2) There is no parameter other than a structure.

(3) There is no return value.

Example:

```
#define ADDRESS ((unsigned char*)0x100000)
void func(void)
{
    *ADDRESS = ((*ADDRESS >> 4) & 0x0f) | ((*ADDRESS << 4) & 0xf0);
}
```

[Workaround]

Specify `-Wo,-Nj` to suppress the optimization that causes this bug, or do not specify the optimization option.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 69 Restriction on project manager during building

[Description]

The project manager freezes during building and commands will no longer be accepted.

[Workaround]

There is no workaround.

In such a case, shut down and restart the project manager.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 70 Restriction on `static` function written in `#pragma section`

[Description]

When a prototype of a `static` function is declared and defined in a `#pragma section`, the following error message will be output:

Message:

```
E2211: redeclaration of function name
```

Example:

```
#pragma section ... begin
static void func();

static void func(){              /* Overloading error */
      :
}
#pragma section ... end
```

[Workaround]

Declare the prototype and define a `static` function outside the `#pragma section`.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 71 Restriction on pointer constant when accessing address with offset

[Description]

During expression transformation through the compiler’s basic optimization, if the value of the pointer constant exceeds 24 bits when indirectly accessing an address with an offset, invalid code will be output.

Example:

```
void func(int adr)
{
    *((volatile unsigned char *)0x4002000 + adr) = 0xff;
}
```

Expression transformation:

(1) When accessing an address with an offset, the address will be calculated using “address (0x4002000) + offset (adr)”

(2) This expression is transformed to the access expression *`address`*`[`*`offset`*`]` through the basic optimization.

(3) The address (0x4002000) stored in an area is moved to another area so that the value will be used in calculation in (1) when the expression is transformed in (2).

During the processing in (3), the address is stored in an area smaller than the original area. As a result, the overflowed data is lost.

Output code example:

```
mov  255, r12
st.b r12, 8192[r11]
```

This value is invalid.

[Workaround]

Calculate the address in advance and modify the source code before assigning the value so that the address does not have an offset.

Example of preventive code:

```
void func(int adr)
       *((volatile unsigned char *)(0x4002000 + adr)) = 0xff;
}
```

[Correction]

This restriction has been removed in Ver. 2.50.

Note, however, that if the conditions under which this restriction applies occur in Ver. 2.41, the following error message will be output and compilation will stop.

Message:

```
E2179: compiler restriction: offset out of range [24bit]
```

No. 72 Restriction on optimization for `if` statement

[Description]

When optional optimization (`-Ol`) or the powerful optimization option `-Wi,-O4` is specified and, if a bit field is related to the conditional expression in an `if` statement and the branch condition in the conditional expression can be determined during compilation, invalid code might be output.

Example:

```
void func()
{
    struct st {
         int x:30;
    } st;

    int k = 1;
    int st.x = 80;

    k = (st.x == 50); if (k != 0) f(1);
    k = (st.x != 50); if (k != 1) f(2);
    k = (st.x != 70); if (k != 1) f(3); /* A code to call f(3) is output */
    k = (st.x != 70); if (k == 0) f(4);
}
```

In the above example, none of the conditions in the `if` statement are satisfied and therefore all the code to call `f(`*x*`)` can be removed by optimization. However, due to invalid optimization, the code to call `f(3)` is output.

[Workaround]

Specify `-Wi,+data_flow_opt=0` to suppress the optimization that causes this bug, or do not specify `-Ol` or `-Wi,-O4`.

[Correction]

This restriction has been removed in Ver. 2.41.

No. 73 Restriction on optimization for `switch` statement

[Description]

If the following conditions are satisfied, a variable to which a register should not be assigned by optimization is assigned a register. As a result, invalid code might be output.

(1) The optimization option `-Os` or `-Ot` is specified.

(2) The `switch` statement outputs a table branch code.

Table branch code will be output in either the following cases:

- `-Xcase=table` is specified.
- `-Xcase=ifelse` or `-Xcase=binary` is not specified, and the following conditions are satisfied:
    - The number of `case` labels is 4 or more
    - The difference between the maximum and minimum values of the label is up to three times the number of `case` labels

(3) The `switch` statement satisfies one of the following conditions:

- The values of the `case` labels are not consecutive. The order of the `case` labels does not matter.
- Multiple `case` labels branch to the same address.
- The `case` label processing continues without inserting a `break` statement.

Example:

When the optimization option `-Ot` is specified and compilation is executed **… Condition (1)**

```
void    g(void);
int     i;
void f(void){
    i = 0;
    do {
        switch(i){
            case 0:  ← The number of case labels is 4 and the difference between the max.
              i++;     and min. value of the case label is 3.  Therefore, table branch code
              break;   is output. … Condition (2)
            case 1:
              g();
              i++;   ← Case label processing continues without inserting break
           case 2:     … Condition (3)
              i++;
            case 3:
              g();
              i++;
              break;
            default:
              g();
              break;
        }
    } while(i < 4);
}
```

[Workaround]

Do one of the following:

(1) Remove the optimization option to reset to the default optimization.

(2) Specify `-Xcase=ifelse` (in CA850 Ver. 2.40 or later).

(3) Specify `-Xcase=binary` (in CA850 Ver. 2.40 or later).

[Correction]

This restriction has been removed in Ver. 2.50.

No. 74 Restriction on initialization of bit field

[Description]

Invalid code will be output if 32 is specified as the number of bits in a bit field and that field is then initialized.

When initialization is not executed, the following message will be output instead of invalid code.

```
W2172: constant out of range
```

Example:

```
int func(){
    struct {
    unsigned int x:32;  /* 32-bit bit field */
    } s1 = {0x12345678};/* Initialization */
}
```

[Workaround]

Specify a 32-bit data type instead of 32 bits.

Example:

```
int func(){
    struct {
    unsigned int x;
    } s1 = {0x12345678};
}
```

[Correction]

This restriction has been removed in Ver. 2.50.

No. 75 Restriction on initialization of automatic array

[Description]

If the following conditions are satisfied, invalid code will be output for initialization of an automatic array:

(1) The optimization option `-Os` or `-Ot` is specified.

(2) An automatic array is initialized.

(3) 32 or more initial values for the automatic array in (2) are omitted.

Example:

```
int a[37] = {1, 2, 3, 4, 5}; /* The 6th and subsequent values (32 values) are
omitted */
```

[Workaround]

Do one of the following:

(1) Remove the optimization option to reset to the default optimization.

(2) Specify 0 for omitted initializers to make the number of omitted values 31 or less.

Example:

```
int a[37] = { 1, 2, 3, 4, 5, 0 }; /* The 7th and subsequent values (31 values) are
omitted */
```

[Correction]

This restriction has been removed in Ver. 2.50.

No. 76 Restriction on `ov` flag in machine-dependent optimization module

[Description]

If the following conditions are satisfied, invalid code will be output in the machine-dependent optimization module:

(1) The optimization option `-Ol` or powerful optimization `-Wi,-O4` is specified.

(2) During data flow optimization, the following operation (`add`, `addi`, `sub`, `subr`, or `cmp`) exists in the machine-dependent optimization module.

- Addition of NaNs results in an underflow and the result is a negative value.
  Example: `0x7fffffff + 0x1 = 0x80000000`
- Addition of negative values results in an underflow and the result is a NaN.
  Example: `0x80000000 + 0x80000000 = 0x0`
- A negative value is subtracted from a non-negative value and the result is a negative value (same as comparison)
  Example: `0x7fffffff – 0x80000000 = 0xffffffff`

(3) A conditional branch or conditional assignment (<, <=, >, >=) references the `ov` flag in (2).

(4) The conditions in (2) and (3) are not detected in the global optimization module.

[Workaround]

Do one of the following:

(1) Specify `-Wi,+data_flow_opt=0` to suppress data flow optimization that causes this bug.

(2) Remove `-Ol` or `-Wi,-O4`.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 77 Restriction on shift in machine-dependent optimization module

[Description]

If the following conditions are satisfied, the equal sign comparison between the values shifted to the right arithmetically might be converted to a logical right shift.

(1) The optimization option `-Os` or `-Ot` is specified.

(2) Two values are shifted to the right arithmetically.

(3) The number of bits by which each value is shifted differs.

(4) After being shifted, the two values are not referenced.

(5) Both values are negative.

Example: When a = –2 and b = –4 (two values are negative).

```
int func(int a, int b) {
    a >>= 1;            /* Arithmetic right shift */
    b >>= 2;            /* Arithmetic right shift */
    if( a==b ) return 1; /* Equal sign comparison */
    else       return 0;
}
```

[Workaround]

Do one of the following:

(1) Specify `-Wi,+hole4_opt=0` to suppress the optimization that causes the bug.
This is an option to suppress peephole optimization of 4 instructions.

(2) Remove the optimization option.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 78 Restriction on union containing integer and pointer

[Description]

A compiler error will be output if an integer constant is assigned using a union containing an integer and pointer.

```
C5102: internal: internal error in 'IvarNode::IvarNode()'
```

This bug occurs when the following conditions are satisfied:

(1) The optimization option `-Os` or `-Ot` is specified.

(2) There is a union containing an `int(long)` or `unsigned int(long)` and a pointer.

(3) An integer constant is assigned to the integer in the union in (2).

(4) After assignment, an indirect reference, indirect assignment, or structure assignment is executed using the pointer in the union in (2).

(5) The integer constant in (3) can be optimized by propagation of copying in the basic block to the pointer access position in (4).

Example:

```
struct S {
    int a[10];
} s;
union {
    int          i;
```

```
        int*        p;
        struct S*   sp;
    } u;
    volatile int vi;
    void f(void){
        u.i = 0x100;       /* Indirect reference */
        vi = *u.p;
        u.i = 0x200;       /* Indirect assignment */
        *u.p = vi;
        u.i = 0x300;       /* Structure assignment */
        s = *u.sp;
    }
```

[Workaround]

Specify the address instead of the pointer.

Example:

```
    struct S {
        int a[10];
    } s;
        union {
        int         i;
        int*        p;
        struct S*   sp;
    } u;
    volatile int vi;
    void f(void){
        u.i = 0x100;       /* Indirect reference */
        vi = *(int*)0x100;
        u.i = 0x200;       /* Indirect assignment */
        *(int*)0x200 = vi;
        u.i = 0x300;       /* Structure assignment */
        s = *(struct S*)0x300;
    }
```

[Correction]

This restriction has been removed in Ver. 2.50.

No. 79 Restriction on compound assignment of values to bit field members when structure packing is specified

[Description]

When structure packing is specified, the result of a compound assignment of values to bit field members will be invalid if the following conditions are satisfied:

(1) The bit field members to be assigned a variable using compound assignment extend over a 32-bit boundary.

(2) The element of the structure array cannot be resolved.

Example: When compiled with packing value 1 (`-Xpack` = 1)

```
        struct S{
            short s1:7;
            short s2:7;
            short s3:7;
            short s4:7;
            short s5:7;    /* Extends over a 32-bit boundary */
```

```
    short s6:7;
}sobj[10];

int i = 1;                /* Variable */

void f1(){
    sobj[i].s1 = 1;
    sobj[i].s5 = 1;
    sobj[i].s6 = 1;
    sobj[i].s1 += 1;
    sobj[i].s5 += 1;      /* Compound assignment */
    sobj[i].s6 += 1;
}
```

Caution The following case is not affected by this bug because the element is resolved during compilation.

```
struct S2{
    short s1:7;
    short s2:7;
    short s3:7;
    short s4:7;
    short s5:7;
    short s6:7;
}sobj2[10];

const int i = 5;   /* const: Constant */

void f2(){
    sobj2[i].s1 = 1;
    sobj2[i].s5 = 1;
    sobj2[i].s6 = 1;

    sobj2[i].s1 += 1;
    sobj2[i].s5 += 1;
    sobj2[i].s6 += 1;
```

[Workaround]

Do one of the following:

(1) Change the compound assignment to a simple assignment.

Example:

```
sobj[i].s5 = sobj[i].s5 + 1;
```

(2) Specify the bit field members so they do not extend over a 32-bit boundary.

Example:

```
struct S{
    short s1:7;
    short s2:7;
    short s3:7;
    short s4:7;
    short dummy:4; /* Dummy member */
    short s5:7;    /* Does not extend over a 32-bit boundary */
    short s6:7;
}sobj[10];
```

[Correction]

This restriction has been removed in Ver. 2.50.

No. 80 Restriction on optimization for union containing integer and pointer

[Description]

If the conditions below are satisfied, an error occurs in the optimization module and the optimization module stops without outputting an error message. This issue do not apply if the optimization module ends normally without causing an error.

(1) Optimization is specified.
`-Og`, `-O`, `-Os`, or `-Ot` (in Ver. 2.50)
`-Os` or `-Ot` (in Ver. 2.41 or earlier)

(2) A union containing a pointer and 4-byte integer is specified.

(3) Optimization that propagates copying from the integer to the pointer in the basic block occurs for the union in (2).

(4) The element propagated as a result of optimization in (3) is not an integer constant.

(5) Indirect access by the pointer of the propagated address occurs or a structure is transferred.

Example:

If compilation is performed with optimization option `-Os` specified **… Condition (1)**

```
typedef union{
     unsigned char *p;          ← Union containing pointer and 4-byte integer … Condition (2)
     unsigned long a;
}TEST;

int e;

char
func(unsigned long data ){
     register TEST s;

   s.a = data;                  ← Variable data is copied to variable s.p … Condition (3)
                                ← Variable data is not an integer constant … Condition (4)
   if (*(s.p) > 0x00)    {      ← Variable s.p is referenced indirectly … Condition (5)
       if ( *(s.p) != 0xFF ){
             e++;
       }
   }
   return(1);
}
```

[Workaround]

Do one of the following:

(1) Lower the optimization level to:
`-Od` or `-Ob` (in Ver. 2.50)
Default optimization level (in Ver. 2.41 or earlier)

(2) Specify the `-Wo,-Nc` option to suppress propagation of copying in the basic block.

[Correction]

This restriction has been removed in Ver. 2.60.

No. 81 Restriction on packed structure parameter

[Description]

An invalid stack frame is generated if the conditions below are satisfied. The parameter register area must be allocated at the top of the stack, but it is allocated in the middle of the stack.

If a structure is specified as a return value, note that the parameter in the conditions below is expressed in the format that the return value of the structure is inserted in the 1st parameter, that is, 1st parameter = return value, 2nd parameter = 1st parameter of a function, 3rd parameter = 2nd parameter of a function.

(1) Structure packing is used. (The `-Xpack` option is used, or `#pragma pack` is written.)

(2) Multiple packed structures with 5 or more bytes are specified for the parameter of a function.

(3) The function in (2) cannot use a variable number of parameters.

(4) The packed structure specified for the 2nd or subsequent parameter is allocated extending over the 16th and 17th byte boundary in the parameter area of the stack frame.

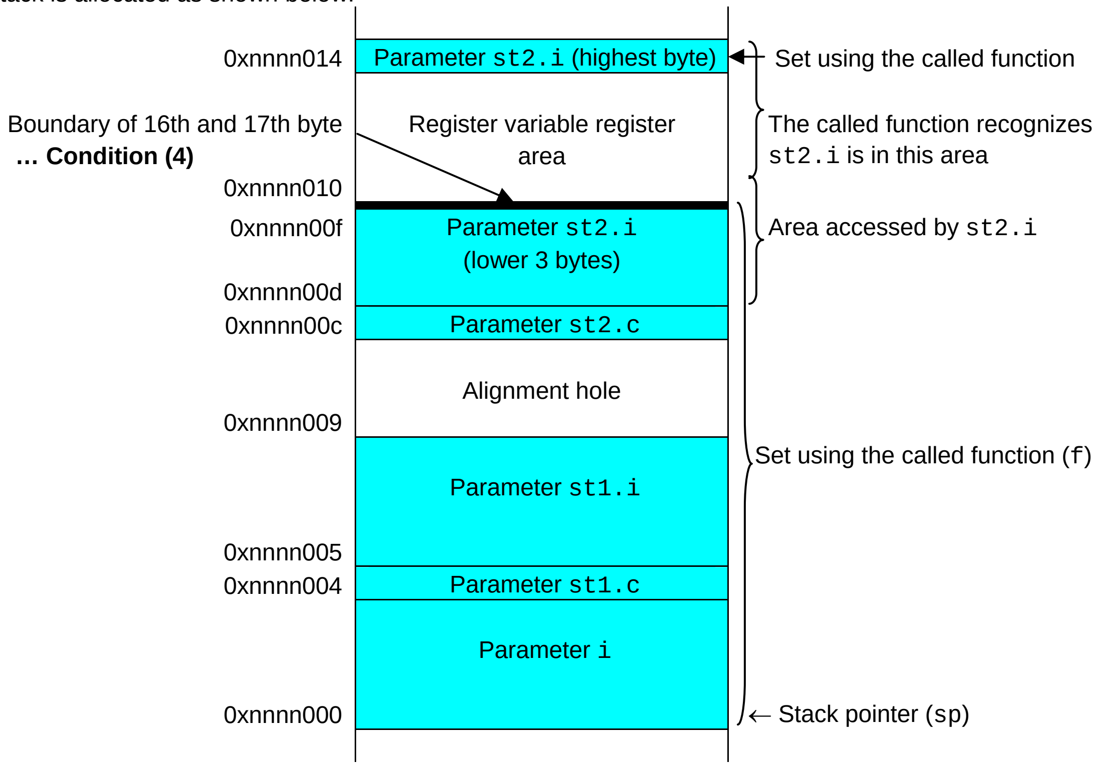
Consequently, the structure of condition (4) is allocated at an invalid location and invalid code will be output in response to the following accesses:

- Access as the structure
- Access of the structure member extending over the 16th and 17th byte boundary in the parameter area of the stack frame.

Example:

```
#pragma pack(1)                              ... Condition (1)
typedef struct {
    char c;
    int  i;
} ST;
void f( int i, ST st1, ST st2 ) {            ... Conditions (2) and (3)
     g(st2.c, st2.i);
}
```

The stack is allocated as shown below.


0xnnnn014
Parameter st2.i (highest byte)
Set using the called function
Boundary of 16th and 17th byte
... Condition (4)
Register variable register area
The called function recognizes st2.i is in this area
0xnnnn010
0xnnnn00f
Parameter st2.i (lower 3 bytes)
Area accessed by st2.i
0xnnnn00d
0xnnnn00c
Parameter st2.c
Alignment hole
0xnnnn009
Parameter st1.i
Set using the called function (f)
0xnnnn005
0xnnnn004
Parameter st1.c
Parameter i
0xnnnn000
← Stack pointer (sp)

[Workaround]

Do one of the following:

(1) Do not use packing of the relevant structure.

(2) Use the parameter as a pointer to the structure.

[Correction]

This restriction has been removed in Ver. 2.60.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 82 Restriction on data swap include function

[Description]

If the following conditions are satisfied, the code generator outputs the error message below and the compiler stops.

(1) The V850E is specified as the device used.

(2) Any of the include functions `__bsh`, `__bsw`, or `__hsw` is used.

(An include function that can be used only when the V850E is specified as the device is used.)

(3) A specific register is assigned to the parameter in (2).

(4) The result of (2) is assigned to multiple locations.

(5) One or more assigned locations in (4) are different registers from the one in (3).

(6) The register in (3) is used later.

Message:

```
C6202:internal: register overflow
```

Example 1:

```
int gi;
void func1() {
    int i1, i2;
    i1 = i2 = __bsw(gi);
    d_func( i1,i2 );
}
```

Example 2:

```
int gi;
void func2() {
    volatile int i1, i2;
    if( i1 = i2 = __bsw(gi) ){
         e_func( i1+i2 );
    } else {
         e_func( i1-i2 );
    }
}
```

[Workaround]

Specify `volatile` for a variable to which the return value of the relevant include function is assigned. At this time, do not use the relevant include function in the middle of an expression.

Example 1:

```
int gi;
void func1() {
    volatile int i1, i2; /* Specify volatile for the variable of the return
    value */
```

```
                i1 = i2 = __bsw(gi);
                d_func( i1,i2 );
            }
```

Example 2:

```
            void func2() {
                    volatile int i1, i2;
                    i1 = i2 = __bsw(gi);  /* Move the include function outside the
            expression */
                    if( i1 ){
                         e_func( i1+i2 );
                    } else {
                         e_func( i1-i2 );
                    }
                }
```

The following workarounds may be effective to avoid this bug because the register assignment conditions are changed:

(1) Change the optimization level

(2) Change the order of the expression and insert a variable.

[Correction]

This restriction has been removed in Ver. 2.50.

No. 83 Restriction on inlining a function with structure as parameter

[Description]

If the following conditions are satisfied, the code corresponding to the parameter is illegally deleted.

(1) The function to be inlined has a structure as a parameter.

(2) A member of the structure used as a parameter in (1) satisfies the following conditions:

<1> The member is an array, structure, or union, and is defined as the second or a later member.

<2> An element of the member in <1> is accessed.

<3> The member does not use the object of the structure in <1> or another member.

If structure packing is specified and the structure used as a parameter is not aligned to 4 bytes, this bug does not occur even if all these conditions are satisfied.

Example:

```
            struct S{
               int dummy1;
               int dummy2;
               int buf[1];                          /* Condition (2)-<1> */
            };
            struct S sobj;
            int num;
            static void sub( struct S sobj ){       /* Condition (1) */
               int i = 0;
               num = sobj.buf[i];                   /* Condition (2)-<2> */
            }                                       /* Condition (2)-<3> */
            func(){
               sub(sobj);
            }
```

Compilation result:

```
_func:
   add -.S5, sp
   ld.w      -4+.A5[sp], r11 ← The value is not passed as a parameter.
   st.w      r11, $_num
   #  19: }
   mov r0, r10
   add .S5, sp  --1
   jmp [lp]
```

[Workaround]

Specify the `-Wp,-a-` option to suppress deletion of unreferenced parameters in a function to be inlined.

[Correction]

This restriction has been removed in Ver. 2.60.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 84 Restriction on function pointer

[Description]

If the following conditions are satisfied, an invalid value might be assigned to a function pointer:

(1) A mask register is used. (The `-Xmask_reg` option is specified.)

(2) 254 or more variable or function names are included in a single compilation unit (including `extern` symbols).

(3) A function pointer is used.

(4) The address of a function is assigned to the function pointer in (3).

Example:

```
The -Xmask_reg option is specified        /* Condition (1) */
    …
(253 functions)                           /* Condition (2) */
    …
int func254() {return 254;}
    …
int (*funcp)();                           /* Condition (3) */
void main() {
    funcp = &func254;                     /* Condition (4) */
}
```

[Workaround]

Do one of the following:

(1) Remove the `-Xmask_reg` option.

(2) Keep the number of variable and function names to within 253 in a single compilation unit.

[Correction]

This restriction has been removed in Ver. 2.60.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 85 Restriction on included function `__sasf()`

[Description]

If the following conditions are satisfied, invalid code is output for `__sasf()`.

(1) The V850E is specified as the device.

(2) A mask register is used. (The `-Xmask_reg` option is specified.)

(3) The function `__sasf()` is used.

(4) The first parameter of `__sasf()` is an `unsigned short` or `unsigned char` variable allocated to a `tidata` or `tibss` section

(5) The second parameter of `__sasf()` is a conditional expression.

Example:

The V850E device and the `-Xmask_reg` option are is specified `/* Conditions (1), (2) */`

```
int gj, i1;
#pragma section tidata begin
unsigned short tus;
#pragma section tidata end
void func() {
    gj = __sasf( tus, i1 > 0 );    /* Conditions (3), (4), (5) */
}
```

Compilation result:

```
ld.w    $_i1, r13
cmp r0, r13                  ← Flag is set due to sasf
sld.h   %_tus, r14
and r21, r14                 ← Flag is cleared due to sasf
sasf    0xf, r14
st.w    r14, $_gj
```

[Workaround]

Make sure that the first parameter of `__sasf` is an automatic variable that is `volatile` and has a basic data type.

Example:

```
    int gj, i1;
    #pragma section tidata begin
    unsigned short tus;
    #pragma section tidata end
    void func() {
        volatile unsigned short a=tus; /* volatile automatic variable that
has a basic data type */
        gj = __sasf( a, i1 > 0 );
    }
```

[Correction]

This restriction has been removed in Ver. 2.60.

No. 86 Restriction on conversion from `float` to `short` or `unsigned short`

[Description]

Invalid code is output if the conditions in (1) or (2) are satisfied.

(1) <1> A `float` is converted to a `short` or `unsigned short`.

<2> The value after the type conversion is referenced from multiple locations.

<3> The first reference of <2> is for assignment.

<4> The second or a later reference of <2> is just for assignment to a register.

(2) <1> A `float` is converted to a `short` or `unsigned short`.

<2> The value after the type conversion is referenced from multiple locations.

<3> The first reference of <2> is for assignment.

<4> The second or a later reference of <2> is as the operand of an expression.

<5> The debugging information output option `-g` is specified.

Example:

```
float f = 70000.0;
short s;
short func() {
    return s = f;  /* Conditions (1)-<1>, (1)-<2>, (1)-<3>, and (1)-<4> */
}
```

Compilation result:

```
ld.w    $_f, r11
mov     r11, r6
jarl    ___trnc.sw, lp
mov     r6, r11
st.h    r11, $_s
mov     r11, r10      ← The return value remains an int.
```

[Workaround]

Specify the `-Wo,-Xer` option to suppress continuous use of the same register.

[Correction]

This restriction has been removed in Ver. 2.60.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 87 Restriction on interrupt function

[Description]

If the following conditions are satisfied, the `callt` instruction to call run-time execution of a function prologue/epilogue is illegally output to an interrupt function, and execution cannot be returned by `reti`:

(1) The V850E is specified as the device used.

(2) One of the following options is specified:

<1> `-Xpro_epi_runtime=on`

<2> An optimization option is specified.

- `-Os` (in Ver. 2.41 or earlier)
- An option other than `-Ot` is specified, or optimization is not specified (in Ver. 2.50).

(3) An interrupt function is used (for which the `__interrupt` or `__multi_interrupt` qualifier is specified).

(4) The total size of the parameter area, dynamic variable area, and register area for operation in a stack frame is smaller than 4 bytes or is 125 bytes or larger.
(The size can be confirmed by checking the value of `.R` *number* output at the end of the function in the assembly list.)

(5) The compiler uses any of `r25` to `r29` or `lp` (`r31`) in the function and does not use `r20` to `r24`.

Caution This bug does not occur when `-Xpro_epi_runtime=off` is specified.

Example:

Checking the total size of the parameter area, automatic variable area, and register area for operation in the assemble list

The `.R` symbol is defined after the function.

```
.set  .S20, 0xd4
.set  .F20, 0xd4
.set  .A20, 0x80
.set  .T20, 0x0
.set  .P20, 0x0
.set  .R20, 0xc0    ← This value
.set  .X20, 0xc0
```

[Workaround]

Specify the following option:

```
-Xpro_epi_runtime=off
```

[Correction]

This restriction has been removed in Ver. 2.60.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 88 Restriction on generating code for complex expressions

[Description]

Invalid code might be generated or a compiler error (`E3228: illegal operand (must be register)`) might be output when executing a complex expression that includes many operations (at least four operators are included) if the expression satisfies the conditions below.

This bug might occur even if the number of operators is less than 4, because expressions are combined internally by optimization and might then satisfy the conditions below.

This bug is more likely to occur when the number of temporary registers is insufficient upon code generation and the stack must be used instead. However, this does not mean that this bug always occurs, because the occurrence of this bug depends on the number of registers used and the amount of stack space used upon code generation.

This bug might occur under the conditions below. The conditions vary according to the CPU used (V850 core or V850E/V850ES core) and the version of the CA850 used.

Conditions:

(1) When using a V850E/V850ES core and CA850 Ver. 2.40 or 2.41, or when using a V850 core and CA850 Ver. 2.40, 2.41, 2.50, or 2.60

(a) The operation references a packed member of a structure, and the member is

(a-1) a 2-byte member aligned to an odd address: See Example 1.

(a-2) a 4-byte member aligned to an address that cannot be divided by 4: See Example 2.

(b) The operation is referenced from a structure pointer that uses structure packing, the packing value for the structure is 1, and a 2- or 4-byte member is referenced: See Example 1.

This bug might occur if (a-1), (a-2), or (b) is satisfied.

(2) When using a V850E/V850ES core and CA850 Ver. 2.50 or 2.60

- (a) The operation references a packed member of a structure, and:
  - (a-1) the member is a 2- or 4-byte member aligned to an odd address: See Example 1.
  - (a-2) the structure is allocated to the `tidata` section, and the member is a 2-byte member allocated to an odd address or a 4-byte member allocated to an address that cannot be divided by 4: See Examples 1 and 2.
- (b) The operation is referenced from a structure pointer that uses structure packing, the packing value for the structure is 1, and a 2- or 4-byte member is referenced: See Example 3.
- (c) The operation uses the result of the included function `__sasf()`.

This bug might occur if (a-1), (a-2), (b), or (c) is satisfied.

Caution The term structure includes bit fields. Bit field members are affected according to their data types, not bit widths. For example, `int a:8;` is a 4-byte data type, and `short b:8;` is a 2-byte data type.

Example 1:

```
/* Example of a member aligned to an odd address */
#pragma pack(1)
struct {
   char pad1;
      ...
     /*
      Up to here, the total size of members from the start is an odd number. */
   short mem; /* odd address */
};
```

Example 2:

```
/* Example of a member aligned to an address that cannot be divided by 4 (1) */
#pragma pack(1)
struct {
   char pad1;
      ...
     /*
      Up to here, the total size of members from the start is a (multiple of 4 +
1)
     */
   int mem; /* odd address */
};

/* Example of a member aligned to an address that cannot be divided by 4 (2) */
#pragma pack(1)
struct {
   short pad1;
   char  pad2;
      ...
     /*
     Up to here, the total size of members from the start is a (multiple of 4 + 3).
*/
   int mem;
};
```

Example 3:

```
/* Example when the packing value is 1 and a 2- or 4-byte member is referenced by
a structure pointer */

#pragma pack(1)
struct {
    char c;
    int  i;
} *pst1;

int i1, i2, i3, i4;

int func() {
    return((~i1) << ((~i2)<<(((i3)+(i4)) << (pst1->i))));
                                                  /* pst1->i is applicable. */
}
```

[Workaround]

Temporarily store the member and the result of the included function `__sasf()` in the expression that might cause this bug in a local variable declared as `volatile`, and use the local variable for the expression.

Example:

```
/* When bug applies to st1.i in the parameters of return */

#pragma pack(1)
struct {
    char c;
    int  i;
} st1;

int i1, i2, i3, i4, i5;

int func() {
     return((~i1) << ((~i2)<<(((i3)+(i4)) << (i5 + st1.i))));
}
```

```
/* Modification to prevent bug */

#pragma pack(1)
struct {
    char c;
    int  i;
} st1;

int i1, i2, i3, i4, i5;

int func() {
    volatile int  tmp = st1.i;
     return((~i1) << ((~i2)<<(((i3)+(i4)) << i5 + tmp)));
}
```

Temporarily store the value of `st1.i` in a variable declared as `volatile`, and use the variable where applies.

[Correction]

This restriction has been removed in Ver. 2.70.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 89 Restriction on parameter of inline function

[Description]

Invalid code is generated if an inline function satisfies the following conditions:

- The address of a parameter of the inline function is acquired, the address is cast to a pointer whose type differs from the parameter, and the parameter is referenced.
- The parameter above is not used at any other location.

The address resulting from typecasing becomes invalid where the inline function is expanded.

If `#pragma inline` is specified for a function in the program, this bug might apply to the function. In addition, depending on the combination of the optimization options `-Os` and `-Ot` and options related to inline expansion, the inline function might be expanded and this bug might occur.

Example of code that causes the bug:

```
#pragma inline subfunc

unsigned int s;

static void
subfunc(int i) {           /* this function is inlined */
    s = *((unsigned int*)&i);    /* cast */
}

void
func(int ii) {
    subfunc(ii);
}

void
main(void) {
    int t = 2;

    func(t);
    printf("%x\n", s);     /* references the parameter */
}
```

[Workaround]

Do one of the following:

(1) Suppress inline expansion.

(a) When using CA850 Ver. 2.40 or Ver. 2.41, do (a-1) or (a-2).

(a-1) Remove `#pragma inline` and specify an optimization option *other than* `-Os` or `-Ot`.

(a-2) Specify the options below if you do not remove `#pragma inline` or change the optimization options. Which option to specify varies depending on whether `#pragma inline` is specified and the combination of optimization options.

	`#pragma inline` Specified	`#pragma inline` Not Specified

When the optimization option other than `-Os` and `-Ot` is specified	Specify all the following options:
	`-Wp,-N0`
	`-Wp,-G0`
	`-Wp,-Sn`
		This bug does not apply.
		((a-1))

When the optimization option `-Ot` is specified		Specify all the following options:
		`-Wp,-N0`
		`-Wp,-G0`
		`-Wp,-Sn`

When the optimization option `-Os` is specified		`-Wp,-Sn`

How to specify options in PM and PM+:

In the **Compiler Options** dialog box, click the **Optimization2** tab and then input 0 in the **Code Threshold** text box for specifying the `-Wp,-N0` option, and 0 in the **Stack Threshold** field for specifying the `-Wp,-G0` option. To specify the `-Wp,-Sn` option, click the **Others** tab and input `-Wp,-Sn` in the **Any Option** text box.

(b) When using CA850 Ver. 2.50 or Ver. 2.60, do (b-1) or (b-2).

(b-1) Remove `#pragma inline` and specify an optimization option *other than* `-Ot`.

(b-2) Specify the options below if you do not remove `#pragma inline` or change the optimization options. Which option to specify varies depending on whether `#pragma inline` is specified and the combination of optimization options.

	`#pragma inline` Specified	`#pragma inline` Not Specified
When the optimization option other than `-Os` and `-Ot` is specified	`-Wp,-no_inline`	This bug does not apply. ((b-1))
When the optimization option `-Ot` is specified		`-Wp,-no_inline`
When the optimization option `-Os` is specified		This bug does not apply. ((b-1))

How to specify options in PM and PM+:

In the **Compiler Options** dialog box, click the **Detail of Optimization** tab and specify `No Expansion [-Wp,-no_inline]` in the **Control of Inline Expansion** drop-down list.

(2) Assign the parameter of the inline function to another variable before using the parameter.

(3) Specify the `-Wp,-a-` option to suppress deletion of unreferenced parameters.

When specifying settings in PM or PM+, open the **Compiler Options** dialog box, click the **Others** tab, and input `-Wp,-a-` in the **Any Option** text box.

[Correction]

This restriction has been removed in Ver. 2.70.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 90 Restriction on optimization for loop processing

[Description]

When a `for` loop, `while` loop, `do-while` loop, or `if-goto` loop is executed while the following conditions are satisfied, the loop is executed only once even if the loop count is set to two or more.

Conditions:

1. When using CA850 Ver. 2.40 or Ver. 2.41
   (a) The level of the specified optimization option is `-Os`/`-Ot` or higher.
   (b) An `unsigned int` variable is used as the loop variable.
   (c) The value of the loop variable is any of the following:

•	Loop initial value:	`0xfffe` or lower
	Loop end value:	`0xffff0002` or higher
	Loop increment:	`0xfffe` or lower
•	Loop initial value:	`0xffff0002` or higher
	Loop end value:	`0xfffe` or lower
	Loop decrement:	`0xfffe` or lower

   (d) <, ≤, >, or ≥ is used for loop variable comparison.
   (e) None of the following are performed:
   - Processing to assign a value to the loop variable inside the loop (excluding addition to or subtracting from the loop variable)
   - Branching from outside the loop to inside the loop
   - Branching from inside the loop to outside the loop

2. When using CA850 Ver. 2.50, Ver. 2.60, or Ver. 2.61
   (a) The specified optimization option is `-O`, `-Os`, or `-Ot`.
   (b) An `unsigned int` variable is used as the loop variable.
   (c) The value of the loop variable is any of the following:

•	Loop initial value:	`0x3ffffffe` or lower
	Loop end value:	`0xc0000002` or higher
	Loop increment:	`0x3ffffffe` or lower
•	Loop initial value:	`0xc0000002` or higher
	Loop end value:	`0x3ffffffe` or lower
	Loop decrement:	`0x3ffffffe` or lower

   (d) <, ≤, >, or ≥ is used for loop variable comparison.
   (e) None of the following are performed:
   - Processing to assign a value to the loop variable inside the loop (excluding addition to or subtracting from the loop variable)
   - Branching from outside the loop to inside the loop
   - Branching from inside the loop to outside the loop

Example:

```
void
main(void) {
    unsigned int  i;

    for(i=0; i<0xffffff00; i+=0x1111) {
                :
              (processing)
                   :
     }
}
```

[Workaround]

Do one of the following:

(1) Specify `-Wo,-No` to suppress the optimization of loop expansion not executed or executed once.

(2) Declare the `unsigned int` type loop variable as `volatile`.

(3) Lower the optimization level.

(4) Use `!=` for the loop comparison.

[Correction]

This restriction has been removed in Ver. 2.70.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 91 Restriction that an undefined-symbol error occurs due to `if` and `goto` statements

[Description]

If the following conditions are satisfied, an undefined-symbol error is output for the label not defined by the user during linking, as shown below. (The label name varies depending on the application used.)

```
ld850: CA850 error F4452: undefined symbol.
.G38  (referenced in ".\main.o")
```

Conditions:

(1) The optimization option level is `-Ob` (the default) or higher.

(2) The debugging information generation option `-g` is specified.

(3) An `if` statement includes a statement to assign two constants to the same variable, and a `goto` statement is specified without using `else` at the end of the `if` statement. There is no other code in the `if` statement.

(4) && and `||` are not used in the `if` statement condition.

(5) An `if` statement includes a statement to assign two constants to a variable, and the two constants satisfy the conditions below.

If “a” is a constant to which a value is assigned when the `if` statement conditions are satisfied, and “b” is a constant to which a value is assigned when the `if` statement conditions are not satisfied, and any of (a) to (d) is satisfied when these constants are signed integers, or any of (e) to (h) is satisfied when these constants are unsigned integers, the restriction applies. However, these conditions assume that the values of “a” and “b” are not floating-point numbers or pointers (although the indirect assignment of an integer pointer is regarded as a target for the conditions).

(a) $a = 0$ and $-32768 \le b \le 32767$

(b) $b - a = 2^n$, and $-32768 \le a \le 32767$

(c) $b = 0$ and $-32768 \le a \le 32767$

(d) $-32768 \le a - b \le 32767$ or $a - b = 2^n$, and $-32768 \le b \le 32767$

(e) $b = 0$ and $a \le 32767$

(f) $a - b \le 32767$ or $a - b = 2^n$, and $b \le 32767$

(g) $a = 0$ and $b \le 32767$

(h) $b - a \le 32767$ or $b - a = 2^n$, and $a \le 32767$

Example 1:

```
    int i;

      :
      :

    if(i){
        i = 0;                /* condition (5)-(a)*/
        goto label;           /* condition (3)*/
    }
    i = 1;                    /* condition (5)-(b)*/
label:
```

Even if a `goto` statement is not used explicitly, a `goto` statement might be output internally depending on the optimization status of the compiler. If the conditions are satisfied in such a case, the restriction applies.

Example 2:

```
if( num == 2 ){
    return 2;
}
if( num == 3 ){
    return 0;
}
return 0;
```

The above code internally outputs the code below, which satisfies the conditions, so the restriction applies.

Example 3:

```
unsigned char tmp;
    if( num == 2 ){
        tmp = 2;              /* condition (5)-(a)*/
        goto label;           /* condition 3 */
    }
    if( num == 3 ){
        tmp = 0;              /* condition (5)-(b) */
        goto label;           /* condition (3) */
    }
    tmp = 0;
label:
    return tmp;
```

[Workaround]

Specify the compiler option `-Wo,-Nx` to suppress `setf` optimization.

[Correction]

This issue has been corrected Ver. 2.72.

No. 92 Restriction on flash/external ROM re-link function during linking

[Description]

If the condition below is satisfied when flash/external ROM relinking is used in CA850 Ver. 2.70, an unresolved-relocation error below might occur or invalid code might be output without returning an error when linking a load module on the flash memory side.

```
ld850: CA850 error F4163: output section ".tibss.word" overflowed or
illegal label reference for symbol "symbol" in file "file.o"(value: 0xffff9048,
input section: .text, offset: 0x00000002, type: R_V850_REGWBYTE). "symbol" is
allocated in section ".tibss.word" (file: a.out).
```

Condition:

An external symbol in a load module on the boot side is referenced relative to `tp` (the text pointer) or `ep` (the element pointer) from a load module on the flash memory side.

When flash/external ROM relinking is used, a load module on the boot side and a load module on the flash memory side are created. When an external symbol in the load module on the boot side is referenced from the load module on the flash memory side, the offset is calculated based on the base pointer of the load module on the boot side and code is generated using the value. However, the value of `tp`, `gp`, or `ep`, which is used as the base pointer, becomes invalid due to this bug, so any offset values that were calculated using `tp` or `ep` also become invalid.

This bug does not apply to code that performs `gp`-relative reference because the offset values based on `gp` are calculated in a different way from the offset values based on `tp` and `ep`.

[Workaround]

There is no workaround.

[Correction]

This issue has been corrected in Ver. 2.72.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 93 Restriction whereby error E2288 is output

[Description]

If the conditions below are satisfied, error message E2288 is output and the CA850 is abnormally terminated.

```
E2288: return type mismatch xxxx (yyyy)
```

Conditions:

1. A function's data type does not match the data type of the value it returns.
   Error E2288 is output if code is specified in which the function's data type and the data type of the return value type do not match.
2. Structure packing is specified (`-Xpack`).
3. The function's data type is `struct` or `union`.

In PM+, the error message is displayed and the execution results in error, but the error count is incorrect.

Example:

```
struct S sobj;
struct S func1(){
     return &sobj;   /* Error E2288 */
}
```

[Workaround]

Correct the function's data type and the data type of the return value type.

[Correction]

This issue has been corrected in Ver. 3.20.

No. 94 Restriction on security ID and option byte

[Description]

If the following conditions are satisfied, the target values listed below become 0.

Conditions:

1. A device that has a security ID or option byte is specified.
2. The linker's `-B` option is specified.
3. The security ID or option byte is not defined in the assembler source file.

Target values:

1. The security ID specified using the `-Xsid` option.
2. Initial value of the security ID when `-Xsid` is omitted (in Ver. 3.00 only).

   The default value is 0xffffffffffffffffffff.
3. Initial value of the option byte.

   The default value is registered in the device file.

[Workaround]

Define the security ID and option byte in the assembler source file.

The security ID and option byte must be defined in the device file.

Example:

```
#---------------------------------------------------------------
#      SECURITY_ID
#---------------------------------------------------------------
.section      "SECURITY_ID"
.word         0xffffffff    --0-3 byte code,Address is 0x70-0x73
.word         0xffffffff    --4-7 byte code,Address is 0x74-0x77
.hword        0xffff        --8-9 byte code,Address is 0x78-0x79
#---------------------------------------------------------------
#      OPTION_BYTES
#---------------------------------------------------------------
.section      "OPTION_BYTES"
.hword        0x0000        --0-1 byte code,Address is 0x7a-0x7b
.hword        0x0000        --2-3 byte code,Address is 0x7c-0x7d
.hword        0x0000        --4-5 byte code,Address is 0x7e-0x7f
```

[Correction]

This issue has been corrected in Ver. 3.10.

No. 95 Restriction on floating point constants and integer type

[Description]

When a floating point value is converted to an `int` value, the range that can be expressed with integer values is the `signed int` or `signed long` type area, and, if the value is converted to an `unsigned int` or `unsigned long` while the range is exceeded, a compiler error might result.

```
E2519: invalid has occurred at compile time.
```

Example:

```
unsigned int ui = 2147483647.0;         /* OK */
unsigned int ui = 2147483648.0;         /* Error */
```

[Workaround]

Specify an integer.

Example:

```
unsigned int ui = 2147483648;
```

[Correction]

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

No. 96 Restriction on input conversion for I/O function in standard library

[Description]

When input conversion processing is performed for `printf`, `sprintf`, `vprintf`, or `vsprintf`, which are I/O functions in the standard library, the precision specified for the conversion specifier `g,G` is incremented.

Example:

```
printf("%.2g", 12.3456789);
/* The result should be 12, but 12.3 is output */
```

[Workaround]

There is no workaround.

[Correction]

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

No. 97 Restriction on assignment in machine-dependent optimization module

[Description]

When specifying optimization options[Note], if an `if` statement or `switch` statement that assigns the same value to three or more variables exists in a basic block, the value of the variable to which a value is assigned first might be undefined as a result of optimization.

Note One of the following sets of options is specified when using a CA850 earlier than Ver. 2.50:

Option for object size optimization + optional optimization (`-Os -Ol`)

Option for object size optimization + advanced optimization (-Os -Wi,-O4)

Option for object size optimization + optional optimization + advanced optimization (`-Os -Ol -Wi,-O4`)

Option for execution speed optimization + optional optimization (`-Ot -Ol`)

Option for execution speed optimization + advanced optimization (`-Ot -Wi,-O4`)

Option for execution speed optimization + optional optimization + advanced optimization (`-Ot -Ol -Wi,-O4`)

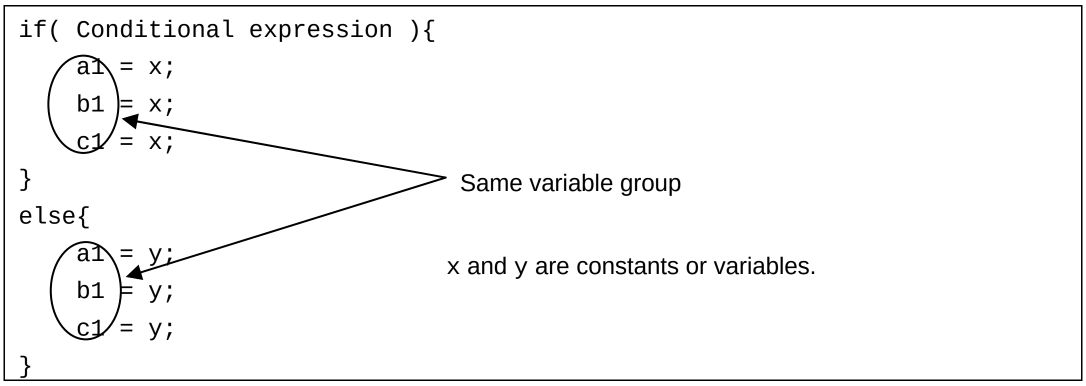
One of the following sets of options is specified when using CA850 Ver. 2.50 or later.

Option for object size optimization (`-Os`)

Option for execution speed optimization (`-Ot`)

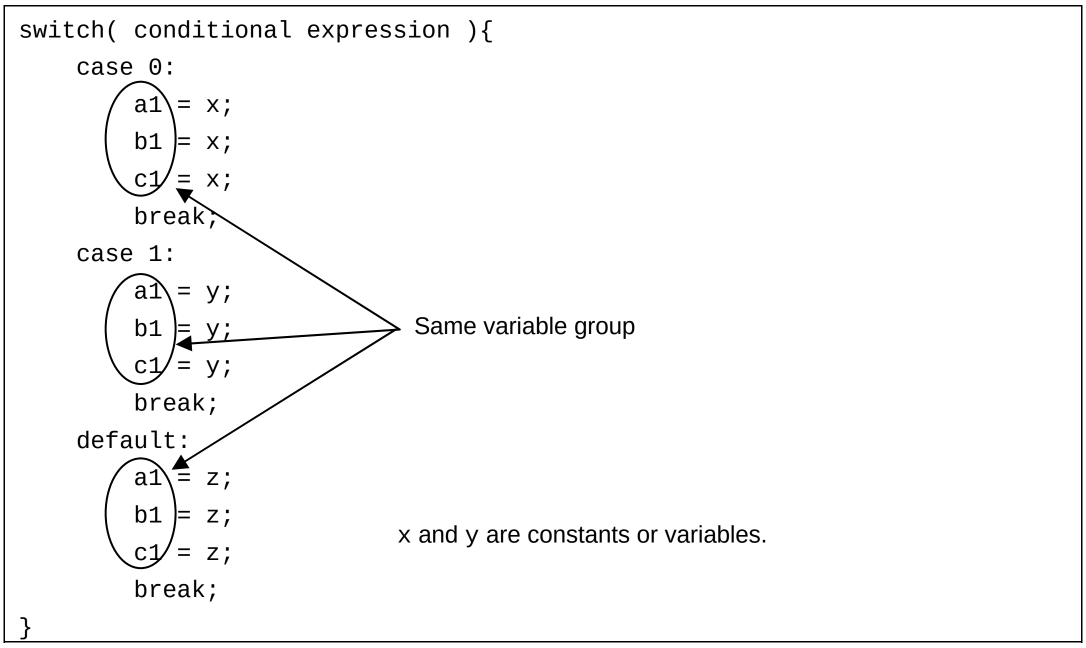
Option for Level 1 Advanced optimization + advanced optimization (`-O -Wi,-O4`)

Example 1:



```
if( Conditional expression ){
    a1 = x;
    b1 = x;
    c1 = x;
}
else{
    a1 = y;
    b1 = y;
    c1 = y;
}
```

Example 2:



```
switch( conditional expression ){
    case 0:
        a1 = x;
        b1 = x;
        c1 = x;
        break;
    case 1:
        a1 = y;
        b1 = y;
        c1 = y;
        break;
    default:
        a1 = z;
        b1 = z;
        c1 = z;
        break;
}
```

[Workaround]

Specify the following options to suppress the optimization that causes this bug:

```
-Wi,+cf_reg_trans_opt1=0
-Wi,+cf_reg_trans_opt2=0
-Wi,+cf_forward_reg_trans_opt=0
-Wi,+cf_reverse_reg_trans_opt=0
-Wi,+abs_ptn_opt=0
```

[Correction]

This issue has been corrected in Ver. 3.10.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 98 Restriction whereby data in an automatic variable area is illegally deleted

[Description]

If the following conditions are satisfied, some elements (of an array) or members (of a structure or union) in an automatic variable area are deleted and indirect referencing for that automatic variable might be performed incorrectly.

Conditions:

(1) The second or a later element of an automatic array or the second or a later member of an automatic structure or union is directly referenced or its address is acquired

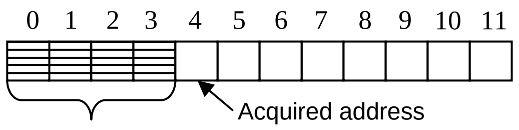
(2) That automatic variable is placed at the lowest address of the automatic variable area in that function.

In such a case, indirect referencing for an automatic variable placed at an address lower than the one referenced in (1) is performed incorrectly.

Example:

```
  char val;
  char *p;

  void func(){
    char str[12];

  /* Initialize the pointer by assigning the address of array str[4] to it */
    p = &str[4];
       .
       .
       .
   /* Indirectly references str[0] to str[3] by using p*/
    val = *(p–1);      ← This indirect reference is incorrect
   }
```

In this case, only the 8-byte area from `str[4]` to `str[11]` is allocated. Consequently, indirect referencing for `str[0]` to `str[3]` might be performed incorrectly.



Elements or members are deleted illegally.

[Workaround]

Assign the start address of the automatic variable to an external pointer that is modified with `volatile`.

Example:

```
  char val;
   char *p;
  char * volatile dmy;

  void func(){
    char str[12];

  /* Initialize the pointer by assigning the address of array str[4] to it */
    p = &str[4];
       .
       .
       .
  /* Assigns the start address of the array str to a dummy external pointer
modified with volatile*/
    dmy = &str[0];
   /* Indirectly references str[0] to str[3] by using p */
    val = *(p–1);      ← This indirect reference is correct
   }
```

[Correction]

This issue has been corrected in Ver. 3.20.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 99 Restriction on address specification with `-U` option of hx850

[Description]

Error F8651 occurs if the following conditions are satisfied:

Conditions:

(1) The option `-Ustart,size` or `-Unum,start,size` is specified.

(2) The hex conversion area is specified for the option in (1), and the specified area includes an address higher than the start address of a peripheral I/O register area.

```
F8651: specified address area(addr1 - addr2) overlaps I/O area (addr3 - addr4)
```

[Workaround]

There is no workaround.

[Correction]

This issue has been corrected in Ver. 3.20.

No. 100 Restriction whereby `strcmp()` and `strncmp()` operations differ

[Description]

When character strings are compared, the characters are interpreted as follows:

`strcmp()` function: Signed

`strncmp()`function: Unsigned

Consequently, the result of `strcmp()` and `strncmp()` operations differ. This restriction do not apply when the compared characters are ASCII codes (0x00 to 0x7f).

Example:

```
strcmp("aaa", "aa\x80");
  /* 0x80 is interpreted as a negative number (= a > 0x80) */
  /* -> An integer greater than 0 (225 in the above code) is returned.*/

strncmp("aaa", "aa\x80", 3);
  /* 0x80 is interpreted as a positive number (= a < 0x80) */
  /* -> An integer less than 0 (-31 in the above code) is returned. */
```

[Workaround]

There is no workaround.

[Correction]

This issue has been modified in Ver. 3.20 so that the `strcmp()` function will interpret character strings as unsigned.

No. 101 Restriction on string literal with 65 or more characters

[Description]

The following assembler error will be output if the second character of a string literal with 65 or more characters is `\\`:

```
E3249: illegal syntax
```

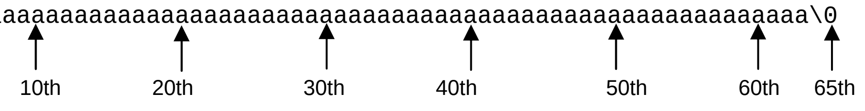
To count the number of characters included in character strings, `\0` is added at the end of the string literal and an escape character is handled as two characters. For example, escape characters such as `\\` and `\0` are handled as two characters. The following example causes an assembly error because it includes a string literal with 65 or more characters.

Example:

```
char a[]=”a\\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”;
  /* E3249 error */
```

Character string count:

```
 a\\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\0
```



[Workaround]

Do one of the following:

For the following character string,

”a\\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”;

1. Replace \\ with another character in the array declaration, and replace this character with \\ upon execution.

Example:

```
char a[]="axaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    void main() {
      a[1] = '\\';
```

2. Initialize the char array with characters.

Example:

```
char a[] = {   'a','\\','a','a','a','a','a','a','a',
                  'a','a','a','a','a','a','a','a','a','a',
                  'a','a','a','a','a','a','a','a','a','a',
                  'a','a','a','a','a','a','a','a','a','a',
                  'a','a','a','a','a','a','a','a','a','a',
                  'a','a','a','a','a','a','a','a','a','a',
```

3. Write the above character string in the assembler.

Example:

```
  .sdata
  .globl       _a, 62
_a:
  .str "a\\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\0"
```

[Correction]

This issue has been corrected in Ver. 3.20.

No. 102 Restriction on optimization with assembler

[Description]

As a result of specifying the pseudo instruction `.option volatile` or `.option nooptimize`, which is functionally equivalent to `.option volatile`, in assembler code, instructions that are placed in an area not subject to assembler optimization might be rearranged (optimized) as a result of expanding instructions.

When accessing a variable modified with `volatile` is specified in C, the pseudo instruction `.option volatile` and an instruction that accesses the variable are output. At that time, instructions newly generated through instruction expansion due to assembler optimization might be rearranged. Deletion and rearrangement of memory access instructions are not performed. With the C source files, the code is executed normally according to the ANSI-C `volatile`-type qualifier specifications.

Conditions:

If the following conditions are satisfied, the order of instructions might be changed and output.

(1) If the code is written in C

- (a) The optimization option `-Og`, `-O`, `-Os`, or `-Ot` is specified.
- (b) A variable modified with `volatile` is defined in another file.
- (c) There are variables that satisfy either of the following conditions:
  - ➢ `const` variables for which the location is not specified.
    (`-Xsconst[=num]` option, specification with a section file, nor `#pragma section` is specified)
  - ➢ There are variables for which the location is specified in any of the following ways:
    - • The directive `#pragma section` is used to specify `data` or `const` sections, based on the section type.
    - • The option `-Gnum` is used to allocate a variable larger than the size specified with `num` to a `.data` or `.bss` section.
    - • A section file is used to specify `data` or `const`, based on the section type, and the variable name is specified.
    - • The option `-Xsconst[=num]` is used to allocate a variable larger than the size specified with `num` to a `.const` section.
- (d) Accessing variables that satisfy conditions (b) and (c) is specified.

In this case, the restriction applies to the code of (d).

(2) If the code is written in assembly language

- (a) The optimization option `-O` is specified.
- (b) An assembly language instruction to be expanded is written in a range specified with the pseudo instruction `.option volatile` or `.option nooptimize`.
- (c) The labels in the instruction that satisfies condition (b) have not been defined so far (up to that line).

In this case, the restriction applies to the code of (d).

Example:

```
.bss
.globl    _Label1, 1
.lcomm    _Label1, 1, 1       ← _Label1 is defined in the same file

.option data  _Label2         ← _Label2 is defined in another file


.option volatile
st.b  r13, $_Label1
.option novolatile

.option volatile
st.b  r14, $_Label2
.option novolatile
```

In the above example, an instruction that references `_Label1` and `_Label2` is expanded into two instructions, as shown below. Among instructions that reference `_Label2`, instructions newly generated through instruction expansion might be rearranged.

Case where instructions are not rearranged:

```
.option volatile
movhi  0x0, gp, r1
st.b    r13, 0x0[r1]
.option novolatile


.option volatile
movhi  0x0, gp, r1
st.b    r14, 0x0[r1]
.option novolatile
```

Case where instructions are rearranged:

```
.option volatile
movhi  0x0, gp, r1
st.b    r13, 0x0[r1]
.option novolatile

movhi  0x0, gp, r1   ← This one is moved

.option volatile
st.b    r14, 0x0[r1]
.option novolatile
```

[Workaround]

Suppress the assembler optimization.

(1) For C source files

Specify the option `-Wa,+O`.

When using PM+, select the **Tool** menu, **Compiler Options**, the **Others** tab, and then input `-Wa,+O` in the **AnyOption** text box.

(2) For assembler source files

Do not specify the option `-O`. When using PM+, select the **Tool** menu, **Assembler Options**, and then clear the **Do Optimization** check box.

[Correction]

This issue has been corrected in Ver. 3.20.

No. 103 Restriction whereby an assignment statement for an automatic variable whose address is acquired by a function is deleted illegally

[Description]

If the following conditions are satisfied, an assignment statement for an automatic variable whose address is acquired by a function might be deleted illegally:

Conditions:

(1) The CA850 earlier than Ver. 2.50 is used:

(a) The optimization option `-Os` or `-Ot` is specified.

(b) One of the following is executed:

- The address of an automatic variable is assigned to an external 32-bit integer variable and then a function is called (example 1).
- The address of an automatic variable is assigned to an external variable whose address cannot be identified statically and then a function is called (example 2)

(2) The CA850 Ver. 2.50 or later is used:

(a) An optimization option of `-Og` or a higher level is specified.

(b) One of the following is executed:

- The address of an automatic variable is assigned to an external 32-bit integer variable and then a function is called (example 1).
- The address of an automatic variable is assigned to an external variable whose address cannot be identified statically and then a function is called (example 2)

Example 1:

```
int a;

void func1(void){
  int val;
  a = &val;
  val = 1;             ← Illegally deleted
  func2();
}
```

Example 2:

```
int* ary[10];

  void func1(int i){
  int val;
  ary[i] = &val;
  val = 1;             ← Illegally deleted
  func2();
}
```

[Workaround]

Do one of the following:

(1) Workaround using an option

(a) The CA850 earlier than Ver. 2.50 is used

Do not specify the optimization option `-Os` or `-Ot`.

(b) The CA850 Ver. 2.50 or later is used

Specify the optimization option `-Ob` or `-Od`.

(2) Workaround by modifying the source

Declare the automatic variable whose address is acquired as `volatile`.

Example:

```
    volatile int val;
```

[Correction]

This issue has been corrected in Ver. 3.20.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 104 Incorrect number of loop executions

[Description]

If all of the following conditions are satisfied, a loop might execute the incorrect number of times:

Conditions:

(1) `-Og`, `-O`, `-Os`, or `-Ot` is specified as the optimization option.

(2) The loop counter is not declared as `volatile`.

(3) The loop termination condition involves comparing the loop counter to a constant.

(4) A constant is added to or subtracted from the loop counter within the loop.

(5) The loop counter is used within the loop as described in (A) or (B).

(A) Conditions *a* to *e* are satisfied (example 1):

a. The loop counter is used to index an array.

b. The elements of the array in *a* are structures, unions, or arrays.

c. The size of the elements of the array in *a* is not a power of 2.

d. The size of the elements of the array in *a* is within 65,534 bytes.

e. The product of the constant used for the loop termination condition in (3) and the size of an element of the array in *a* is too large to store in a 32-bit integer.

(B) Conditions *f* to *i* are satisfied (example 2):

f. The loop counter is used as one multiplier in an expression with a constant.

g. The constant in *f* is not a power of 2.

h. The constant in *f* is in the range from –65,534 to 65,534.

i. The product of the constant used for the loop termination condition in (3) and the constant in *f* is too large to store in a 32-bit integer.

* Loop counter: This controls when a loop ends.

Example 1:

```
struct {
   int s1;
   int s2;
   int s3;            // The elements of the array "ary” are structures.
} ary[100];          // The size (12 bytes) is not a power of 2 and
                     // is within 65,534 bytes.

int i;                        // The loop counter i is not declared as volatile.
for ( i = 0; i < INT_MAX; i ++ ){ // The loop termination condition involves
                                  // comparing the loop counter to a constant.
                          // A constant is added to the loop counter i.

   ary[i].s1 = 1;         // The loop counter i is used to index the array.
  …
}
```

`INT_MAX*12` (which equals 0x5FFFFFFF4) is too large to store in a 32-bit integer, so this code might apply to this restriction.

Example 2:

```
volatile int vi;

int i = 0;               // The loop counter i is not declared as volatile.
while ( i < INT_MAX ){   // The loop termination condition involves
                         // comparing the loop counter to a constant.
  vi = i * 100;          // The loop counter i is used as one multiplier
                         // in an expression with a constant.
  …                      // The constant is in the range from -65,534 to 65,534.

  i++ ;                 // A constant is added to the loop counter i.
}
```

`INT_MAX*100` (which equals 0x31FFFFFF9C) is too large to store in a 32-bit integer, so this code might apply to this restriction.

[Workaround]

Do one of the following:

(1) Change the constant in the loop termination condition as follows:

- The product of this constant and the size of an element in an array indexed using the loop counter is not too large to store in a 32-bit integer.
- The product of this constant and a constant multiplied with the loop counter is not too large to store in a 32-bit integer.

(2) Declare the loop counter as `volatile`.

Modify with `volatile` the automatic variable whose address is acquired.

Before modification:

```
int i;
for ( i = 0; i < INT_MAX; i ++ ){

     ary[i].s1 = 1;
     …
}
```

After modification:

```
volatile int i;
for ( i = 0; i < INT_MAX; i ++ ){

      ary[i].s1 = 1;
      …
}
```

(3) Specify `-Od` or `-Ob` as the optimization option.

[Correction]

This issue has been corrected in Ver. 3.40.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 105 Incorrect string literals

[Description]

If the following conditions are satisfied, the contents of the string literal become incorrect:

Conditions:

(1) ASCII code 0x00 is used in a string literal.

(2) An ASCII code from 0x30 through 0x37 immediately follows 0x00 in (1).

Example: When the ASCII code following the ASCII code 0x00 is 0x37

```
char string1[] = "\x00\x37";
char string2[] = "\000\067";     // (37)16 = (67)8
char string3[] = "\x00" "7";     // (37)16 = ’7 ’
```

The correct output is 0x00, 0x37, 0x00, respectively, but 0x07 and 0x00 are output.

[Workaround]

Do one of the following:

(1) Initialize strings without using string literals.

```
char string4[] = {'\x00', '\x37', '\0'};
```

(2) Specify the ASCII code for a character other than 0x30 to 0x37 after 0x00, and then dynamically replace that character.

```
char string5[] = "\x00*";
string5[1] = '\x37';
```

[Correction]

This issue has been corrected in Ver. 3.40.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 106 Restriction involving format specifiers for the `sscanf`, `fscanf`, and `scanf` functions

[Description]

If conditions (1) to (3) are satisfied, the parameter for the format specifier described in (3) is overwritten:

Conditions:

(1) The `sscanf`, `fscanf`, or `scanf` function is used.

(2) There are less entered fields than format specifiers.

(3) The first extra format specifier character is `s`, `e`, `f`, `g`, `E`, `F`, `G`, or `[ ]` (examples 1 and 2).

If conditions (4) and (5) are satisfied, the parameter for the format specifier described in (5) is overwritten.

Conditions:

(4) The `sscanf`, `fscanf`, or `scanf` function is used.

(5) `[ ]` are used as format specifier characters, and the character pattern enclosed by them is not in the entered fields (example 3).

Example 1: When first extra format specifier is `f`

```
char ary1[5];
float  f1 = 2.0, f2 = 3.0;

sscanf ("aaaa", "%s %f %f", ary1, &f1, &f2);
                   ↑ First extra format specifier
```

The input field `aaaa` is stored in `ary1` as a character string. However, there is no input field for the format specifiers `%f` and `%f` after `%s`. In this case, the value of the parameter `f1` for the first extra format specifier is overwritten.

Expected values: `ary1 = "aaaa", f1 = 2.0, f2 = 3.0`

Output result: `ary1 = "aaaa", f1 = 0.0, f2 = 3.0`

Example 2: When first extra format specifier is s

```
int  data1, data2;
char ary2[5]="test";

sscanf ("1 2", "%d %d %s", &data1, &data2, ary2);
                     ↑ First extra format specifier
```

The input field `1` is stored in `data1` as a decimal integer. The input field 2 is stored in `data2` as a decimal integer. However, there is no input field for the format specifiers `%d %d` following after `%s`.
In this case, the value of the parameter `ary2` for the first extra format specifier is overwritten.

Expected values: `data1 = 1, data2 = 2, ary2 = "test"`

Output result: `data1 = 1, data2 = 2, ary2 = "\0"`

Example 3: When the character pattern enclosed by `[ ]` is not in the entered fields

```
char ary3[5] = "test";
char ary4[5] = "test";
char ary5[5] = "test";

sscanf ("aaaa bbbb cccc", "%s %[a] %s", ary3, ary4, ary5);
                            ↑ Format specifier that does not match the entered field
```

The input field `aaaa` is stored in `ary2` as a string literal. Next, the code searches for a in the input field `bbbb` and attempts to store the result in `ary3`, but the character is not found. In this case, the value of `ary3` is overwritten.

Expected values: `ary3 = "aaaa", ary4 = "test", ary5 = "test"`
Output result: `ary3 = "aaaa", ary4 = "\0", ary5 = "test"`

[Workaround]

There is no workaround.

[Correction]

This issue has been corrected in Ver. 3.40.

No. 107 Restriction involving the character string parameter for the `atoi`, `atol`, `strtol`, and `strtoul` functions

[Description]

If conditions (1) to (5) below are satisfied, the return value might be invalid. For the `strtol` and `strtoul` functions, the global variable `errno` is not set to `ERANGE`:

Conditions:

(1) The `atoi`, `atol`, `strtol`, or `strtoul` function is used.

(2) For the `atoi` or `atol` function, the character string parameter exceeds 32 bits when expressed as a decimal number.
For the `strtol` or `strtoul` function, the first character string parameter exceeds 32 bits when expressed using the base specified as the third parameter.

(3) When the portion of the character string parameter in (2) from the first character to a given character is converted to a number and compared to the absolute value of the lower 32 bits of the value that results when the portion of the same character string from the first character to the character following the given character above is converted to a number, the latter value is greater than the former.

(4) The comparison in (3) includes all characters in the string.

(5) For the `atoi`, `atol`, or `strtol` function, the lower 32 bits of the number to which the character string was converted in (2) are within the range from `LONG_MIN` to `LONG_MAX` after adding a sign.

Example 1: When the `strtoul` function is used

```
char *p;
unsigned long ul;

ul = strtoul("123456789", &p, 16);
```

The hexadecimal value 0x123456789 exceeds 32 bits, and the comparison in (3) includes all characters in the string.

Example: If the character string 12345678 is converted to hexadecimal (0x12345678) and the value is compared to the lower 32 bits of the value that results when the string including the next number (9) is converted to hexadecimal (0x123456789), the latter is greater than the former.
In other words, 0x12345678 < 0x23456789.

Expected values: `ul = ULONG_MAX    errno = ERANGE`
Output result: `ul = 0x23456789`

Example 2: When the `strtol` function is used

```
char *p;
signed long l;

l = strtol("-123456789", &p, 16);
```

The hexadecimal value 0x-123456789 exceeds 32 bits, and the comparison in (3) includes all characters in the string.

Expected values: `l = LONG_MIN    errno = ERANGE`
Output result: `l = DCBA9877(-0x23456789)`

Example 3: When the `atoi` function is used

```
signed int i;

i = atoi("5368709120");
```

The decimal value 5368709120 (which equals 0x140000000) exceeds 32 bits, and the comparison in (3) includes all characters in the string.

Example: If the character string "536870912" is converted to decimal (536870912, which equals 0x20000000) and the value is compared to the lower 32 bits of the value that results when the string including the next number ("5368709120") is converted to decimal (5368709129, which equals 0x140000000), the latter is greater than the former.
In other words, 0x20000000 < 0x40000000.

Expected values: `i = LONG_MAX`
Output result: `i = 1073741824(0x40000000)`

[Workaround]

There is no workaround.

[Correction]

This issue has been corrected in Ver. 3.40.

No. 108 Restriction involving initialization of a structure that includes a bit field among its members

[Description]

If conditions (1) and (2) are satisfied, the bit field is not correctly initialized (example 1):

Conditions:

(1) A structure is used that includes a bit field immediately followed by another structure (or union) as members.

(2) Both of the members included in the structure in 1 are initialized using initial values.

If conditions (3) to (5) are satisfied, the bit field might not be correctly initialized (example 2).

(3) An automatic array of structures is used.

(4) The structure in 3 includes a bit field and an element that is at least 126 bytes among its members.

(5) The initializer for this element is omitted, and the element is implicitly initialized to 0.

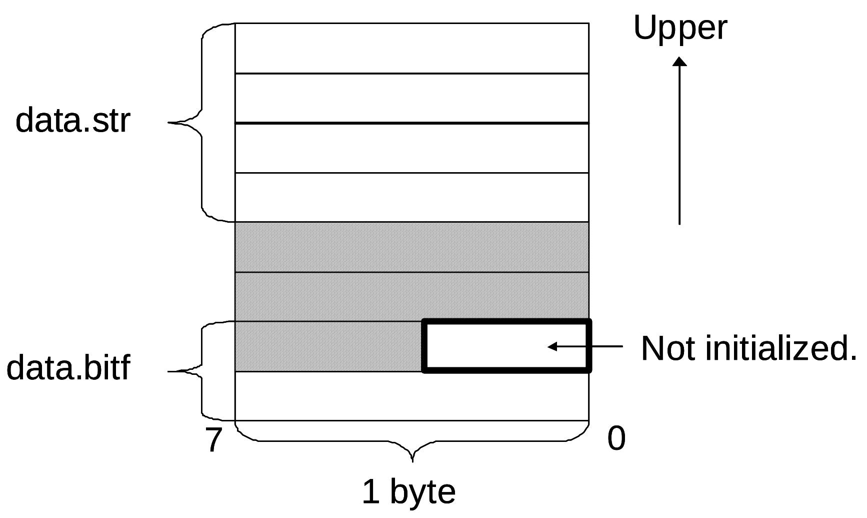
Example 1:

```
struct {
        int bitf : 12 ;       // A bit field is used.
        struct {              // A structure follows the bit field.
                int s ;
        } str ;
} data = { 0xFFF, { 2 } } ;  // Both members of the structure that includes the
                             // bit field are initialized.
```

The higher 4 bits of `data.bitf` are not initialized.

Expected values: `data.bitf  = 0xFFF`

Output result: `data.bitf  = 0xFF`

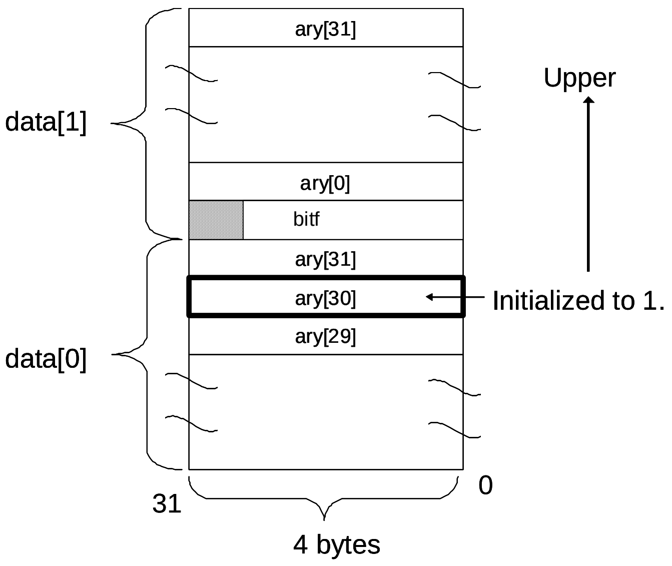


Example 2:

```
void func(void)
{
  struct {
     int bitf : 25;              // A bit field is used.
     int ary[32];                // The following element is at least 126 bytes.
  } data[2] = { { 1 }, { 1 } }; // The automatic array of structures data is used.
  …                             // ary is implicitly initialized to 0.
}
```

The offset of the code used to set `data[1].bitf` to 1 ends up too low. `data[0].ary[30]` is initialized to 1. `data[1].bitf` ends up undefined.

Expected values:	`data[0].ary[30] = 0`
	`data[1].bitf = 1`
Output result:	`data[0].ary[30] = 1`
	`data[1].bitf` is undefined.



[Workaround]

Do one of the following:

(1) Initialize the structure by assigning values to its members.

For example 1:

```
struct {
   int bitf : 12 ;
   struct {
        int s ;
   } str ;
} data ;                    // Not initialized
…
data.bitf = 0xFFF ;         // Changed to assignment
data.str.s = 2 ;
```

(2) Declare the bit field in a separate structure within the main structure.

For example 1:

```
struct {
   struct {
          int bitf : 12 ;
   } str2 ;                // Structure within the structure
   struct {
          int s ;
   } str ;
} data = { { 0xFFF }, { 2 } } ;
```

[Correction]

This issue has been corrected in Ver. 3.40.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 109 Restriction involving nested conditionally assembled pseudo instructions

[Description]

If the following conditions are satisfied, the error message F3510 is output:

Conditions:

(1) An `.elseif` or `.elseifn` pseudo instruction is used.

(2) The condition in (1) evaluates to true.

(3) There is an `.elseif` or `.else` pseudo instruction that corresponds to the pseudo instruction in (1).

(4) There is a pseudo instruction that is conditionally assembled nested within the block that corresponds to the pseudo instruction in (1).

Example:

```
.set   FLAG1, 0
.set   FLAG2, 1
.set   FLAG3, 1
.set   FLAG4, 0

.if FLAG1 == 1
  .set   TEMP, 1
.elseif FLAG2 == 1  -- Satisfies conditions (1) and (2)
   .if FLAG3 == 1   -- Satisfies condition (4)     ┐
     .set   TEMP, 2                                ├  Nested
   .endif                                          ┘
.elseif FLAG4 == 1  -- Satisfies condition (3)
  .set   TEMP, 3
.endif
```

[Workaround]

When nesting a conditionally assembled pseudo instruction, use either block 1 or 2, below.

(1) `.if` pseudo instruction block

```
.if FLAG1 == 1
  .set   TEMP, 1
.else
  .if FLAG2 == 1
    .if FLAG3 == 1      ┐
      .set   TEMP, 2    ├ Nested
    .endif              ┘
  .elseif FLAG4 == 1
    .set   TEMP, 3
  .endif
.endif
```

(2) `.elseif` pseudo instruction block that ends with `.endif`

```
.if FLAG1 == 1
  .set   TEMP, 1
.elseif FLAG4 == 1
  .set   TEMP, 3
.elseif FLAG2 == 1
   .if FLAG3 == 1       ┐
     .set   TEMP, 2     ├ Nested
 .endif                 ┘
.endif
```

[Correction]

This issue has been corrected in Ver. 3.40.

No. 110 Restriction involving assignment within `switch` and `if` statements

[Description]

If the following conditions are satisfied, the processing within `switch` or `if` statements might become incorrect as a result of optimization by the ca850:

Conditions:

(1) The C source code satisfies conditions (A) to (C) (example 1).

(A) For a version earlier than 2.50, the compiler optimization option `-Os` or `–Ot` is specified in addition to `-Ol`, but `–Wi,–O4` is not specified.
For Ver. 2.50 or later, the compiler optimization option `-Os` or `-Ot` is specified, but `–Wi,–O4` is not specified.

(B) Parallel processing is performed to assign a value to the same variable in a `switch` or `if` statement.

(C) After the assignment in (B), execution jumps back to the same position.

Example 1:

```
int func(int val) {
  int res;
  switch (val) {
  case 0xA: res = 0x1; break;
  case 0xB: res = 0x2; break;
  case 0xC: res = 0x3; break;
  case 0xD: res = 0x3; break;   <--- Parallel processing to assign a value to
  case 0xE: res = 0x3; break;        the same variable (res).
  default:  res = 0x3; break;
  }
  return res;                   <--- After a value is assigned to res, execution
}                                    jumps back to the same position.
```

(2) The assembly language output according to the C source code in (1) satisfies all of the following conditions (example 2):

(a) The `mov` or `ld` directive is used to transfer data to the same register at the end of multiple basic blocks (*Block A*).

(b) *Block A* combines everything into one block (*Block B*).

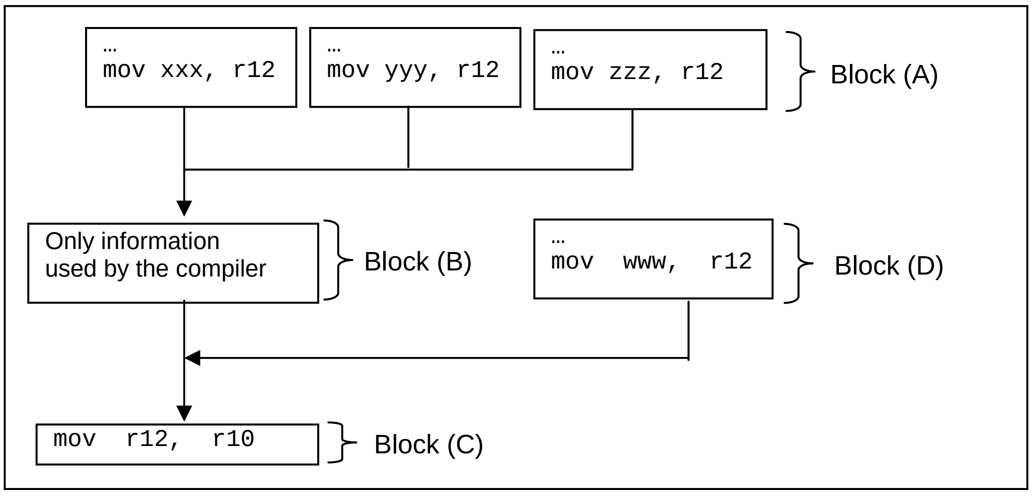
(c) There are no directives (or output code) in *Block B*, which contains information used by the compiler. At least one information item is included.

(d) In *Block C* (the block following *Block B*), the data is transferred to a register other than the one used in *Block A*.

(e) Among the blocks directly combined into *Block C*, there is a block (*Block D*) that performs the same transfer as that performed in *Block A*.

Remark A basic block is group of directives that are processed in order and ends with a directive that causes execution to jump.

Example 2:



[Workaround]

Do one of the following:

(1) Insert `__asm("\n");`. (The check tool outputs the position where this insertion is required.)

```
int func(int val) {
  int res;
  switch (val) {
  case 0xA: res = 0x1; break;
  case 0xB: res = 0x2; break;
  case 0xC: res = 0x3; break;
  case 0xD: res = 0x3; break;
  case 0xE: res = 0x3; break;
  default:  res = 0x3; break;
  }
  __asm("\n");   ←――――― Insert __asm("\n");.
  return res;
}
```

(2) Specify `-O` or lower as the optimization option.

(3) Specify the `-Wi,-O4` option.

[Correction]

This issue has been corrected in Ver. 3.40.

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

Contact an NEC Electronics sales representative or distributor for details.

No. 111 Restriction on incorrect compare operation optimization while casting

[Description]

No.111 occurs when the following conditions are satisfied:

Conditions:

(1) Compare operation(<, <=, >, >=, ==, !=) between integer operands.

(2) Either operand is casted as shown below:

- ・signed char ->unsigned char
- ・signed char ->unsigned short
- ・unsigned char ->signed char
- ・signed short ->unsigned short
- ・unsigned short ->signed short
- ・unsigned short ->signed char

(3) After casting the operand, its type is the same as the type of the other operand.

(4) Other operand is treated as constant by way of compiler optimization.

(5) Comparison used in (1) occurs between constant in (4) and before casting of (2). This will always be true or always be false).

Example :

```
short s; unsigned short us;
s = -1;
...
if (s == (short)us)
```

In this example, even if the result of comparison operation depends on its operands, CA850 incorrectly interprets the comparison result as always false. This results in eliminating both the if-sentence and if-true-block.

[Workaround]

Insert '__asm("¥n");' before the line where incorrect comparison operation optimization occurs.

Checktool will identify the line number. When a function that has incorrect comparison operation optimization is inlined, checktool will identify the function name and its line number.

```
short s; unsigned short us;
s = -1;
...
__asm("¥n");
if (s == (short)us)
```

[Correction]

This issue will be corrected in Ver. 3.46.

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

Contact an Renesas Electronics sales representative or distributor for details.

No. 112 Restriction on incorrect move optimization of assignment sentence

[Description]

No.112 occurs when the following conditions are satisfied, resulting in moving the statement assigning variable X in Basic Block 1 to the back of the sentence assigning variable X in Basic Block 2 incorrectly:

Conditions:

(1) Optimization level is –Og or higher (-Og, -O, -Os, -Ot, and associated code optimizations)

(2) Control flow of function consists of Basic Blocks like (a)～(e).

(a) Either non-volatile automatic variable X or non-volatile argument variable X is assigned in both Basic Block 1 and 2, then referenced in Basic Block 3.

(b) The value assigned to variable X in Basic Block 2 is either volatile variable expression or peripheral I/O register expression.

(c) The path from Basic Block 1 is the only one merging into Basic Block 2.

(d) After Basic Block 2, both path-merging-into Basic Block 3 and path-not exist.

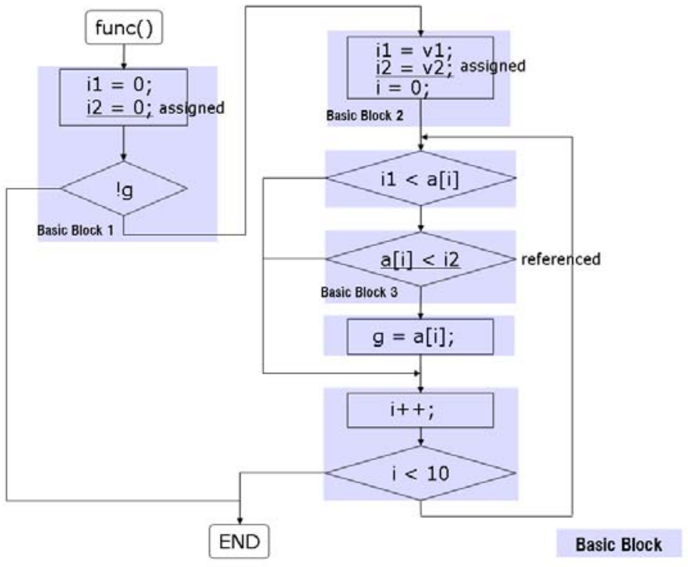
(e) No indirect access to variable X between Basic Block 1 and 3.

Note : A basic block is code that has one entry point,
one exit point, and no jump instructions contained within it.

Example :

```
int g, a[10];
volatile int v1, v2;
void func(void) {
  int i1 = 0, i2 = 0;
  if(!g){
    int i;
    i1 = v1;
    i2 = v2;
    for(i = 0; i < 10; i++)
      if(i1 < a[i] && a[i] < i2)
        g = a[i];
  }
}
```

Control flow of the example code is as follows.



[Workaround]

Insert '__asm("¥n");' before the line which the checktool output.

```
if(!g){
  int i;
  i1 = v1;
  i2 = v2;
 __asm("\n");
  for(i = 0; i < 10; i++)
    if(i1 < a[i] && a[i] < i2)
      g = a[i];
}
```

[Correction]

This issue will be corrected in Ver. 3.46.

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

Contact an Renesas Electronics sales representative or distributor for details.

List of Usage Restrictions in Other Package Tools

1. Product History

No.	Restrictions and Changes/Additions to Specifications	Version									
		V2.41	2.50	2.6x	2.70	2.72	3.00	3.10	3.20	3.30	3.4x
1	Restriction on replacement using regular expression in PM+		×	×	○	○	○	○	○	○	○
2	Restriction on square brackets in PM+		×	×	○	○	○	○	○	○	○
3	Restriction on linking `.prw` file with PM+		–	–	×	×	×	○	○	○	○
4	Restriction on system call display by stk850 (RX850 and RX850 Pro)		–	–	–	–	×	○	○	○	○
5	Restriction on specifying stack size of system calls with stk850 (RX850)		–	–	–	–	×	○	○	○	○
6	Restriction on handling of command file using PM+		×	×	×	×	×	○	○	○	○
7	Restriction on linking system calls of RX850 or RX850 Pro from library in PM+		–	–	–	–	×	○	○	○	○
8	Restriction on include files handled by PM+		–	–	×	×	×	×	○	○	○
9	Restriction on acquiring information on included C source files handled by PM+		–	–	–	–	–	–	×	×	○

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

2. Details of Usage Restrictions

No. 1 Restriction on replacement using regular expression in PM+

[Description]

With PM+, if replacement processing of a character string is performed using the regular expression ^, which indicates the line head, and $, which indicates the end of the line, the linefeed codes that exist before and after the relevant character string are included in the replacement target.

Example:

When **Regular Expression** is selected in the **Replace String** dialog box and replacement of `^xyz` with ABC is executed, the line feed is also replaced as shown below.

```
-----------------
abc
xyz
abc
-----------------
↓
-----------------
abcABC
abc
-----------------
```

[Workaround]

Do not use a regular expression for replacement.

[Correction]

This issue has been corrected in PM+ Ver. 5.20 (included with CA850 Ver. 2.70).

No. 2 Restriction on square brackets in PM+

[Description]

With PM+, if a square bracket is used in the file name, path name, or project group name, the workspace cannot be loaded and individual options cannot be set.

[Workaround]

Do not use square brackets for the file name, path name, or project group name.

[Correction]

This issue has been corrected in PM+ Ver. 5.20 (included with CA850 Ver. 2.70).

No. 3 Restriction on linking `.prw` file with PM+

[Description]

The `.prw` files are linked with PM+, but, if a workspace file whose name contains a space is double-clicked, the file might not be opened normally.

[Workaround]

Open the workspace file by using the PM+ **Open Workspace** menu.

[Correction]

This issue has been corrected in PM+ Ver. 6.10.

No. 4 Restriction on system call display by stk850 (RX850 and RX850 Pro)

[Description]

The stack size of system calls of the RX850 and RX850 Pro is unknown and displayed as ?.

The stack size of system calls of RX850 V3.20 and RX850 Pro V3.20 is 0, so this restriction does not affect the stack calculation result.

[Workaround]

Use the system calls with their stack size unknown, or set the additional margin for the stack to 0.

[Correction]

This issue has been corrected in stk850 Ver. 2.11.

No. 5 Restriction on specifying stack size of system calls with stk850 (RX850)

[Description]

In RX850 V3.20, the stack size of the following system calls is 0, but 12 may be added to the margin.

- `ter_tsk`
- `rel_wai`
- `get_blk`
- `pget_blk`
- `tget_blk`
- `rel_blk`

If restriction No. 4 applies, the stack size of the system call is unknown and calculated as 0, so No. 5 do not apply.

Restriction No. 5 applies when the size of the above system calls is not specified for the additional margin with the stk850 but the function is not displayed in the **Unknown Functions** field.

[Workaround]

Change the additional stack size margin for the relevant system call to 0.

[Correction]

This issue has been corrected in stk850 Ver. 2.11.

No. 6 Restriction on handling of command file using PM+

[Description]

If a source file is selected in the **Project** Window and then compiled while the **Use Command File** check box is selected in the **Compiler Options** dialog box or **Assembler Options** dialog box, the corresponding command file is deleted. As a result, either of the following error messages is output at the next build or rebuild:

```
F1303: cannot open file 'xxxx.cca'
F3503: can not open file xxxx.cas
```

[Workaround]

Re-generate the file required for building by selecting **Export Makefile** in the PM+ **Project** menu.

[Correction]

This issue has been corrected in CA850 Ver. 3.10 and PM+ Ver. 6.11.

No. 7 Restriction on linking system calls of RX850 or RX850 Pro from library in PM+

[Description]

If all the conditions below are satisfied, linking of system calls of the RX850 or RX850 Pro results in a linking error, and the following error message is output:

```
F4452: undefined symbol.
```

Conditions:

(1) The RX850 or RX850 Pro is used

(2) A library is specified in the **Linker Options** dialog box.

(3) System calls are used in the library specified in (2).

(4) The system calls of (3) are not used in a program registered as a source file

[Workaround]

Do the following:

(1) In the **Linker Options** dialog box, click the **Library** tab and specify the library of the RX850 or RX850 Pro in the **Library[-l]** text box.

Example:

When using RX850 Ver. 3.20 and the user library name is `libusr.a`:

```
usr;rx
```

(2) In the **Linker Options** dialog box, click the **Library** tab and specify the library path of the RX850 or RX850 Pro in the **Library Search Path[-L]** text box.

Example:

When using RX850 Pro Ver. 3.20:

```
C:\Program Files\NEC Electronics Tools\RX850 Pro\V3.20\lib850e\r32
```

(3) Clear the following check boxes in the **RX850 Pro Settings** dialog box. **[RX850 Pro only]**

- Link a Nucleus Library (U)
- Link a Interface Library (R)

[Correction]

This issue has been corrected in CA850 Ver. 3.10 and PM+ Ver. 6.11.

No. 8 Restriction on include files handled by PM+

[Description]

If a file name that includes a space is specified as an include file in a source file, the include file and all the following include files are handled as follows:

- Not registered to the **Include Files** folder in the **Project** window
- Not registered to a makefile
- Not made to be the target of the dependency during build

This issue also applies to include files specified in the portion to be false as a result of conditional compilation.

This issue only applies to the specification of include files in a source file, which are analyzed by PM+, but not to actual processing during compilation.

Example:

```
#include “C:\i n c\test.h”
```

[Workaround]

Specify file names that do not include spaces.

[Correction]

This issue has been corrected in CA850 Ver. 3.20 and PM+ Ver. 6.31.

No. 9 Restriction on acquiring information on included C source files handled by PM+

[Description]

If

1. The number of double quotation marks (") at the beginning of the source file (before a given #include statement) is odd, or
2. A < is specified at the beginning of the source file (before a given `#include` statement) and no > is specified afterwards:

Header files added by the `#include` statements might be treated as described below. However, these problems will not affect the processing when the project is rebuilt.

- Not added to the **Include Files** folder in the **Project** window
- Not added to a makefile
- Not treated as dependencies when building the project

Example 1:

```
/*xxx"""xxx*/
#include "usr1.h"  ← 3 double quotation marks before the #include statement
/*xxx"xxx*/
#include "usr2.h"  ← 6 double quotation marks before the #include statement
```

In the above code, this restriction applies to `usr1.h` and the header file included using `usr1.h`.

This restriction does not apply to `usr2.h`.

Example 2:

```
/*xxxx><xxx*/
#include <usr3.h> ← < is used before #include, but it is not followed by >.
#include <usr4.h> ← < is used before #include, and it is followed by >.
```

In the above code, this restriction applies to `usr3.h` and the header file included using `usr3.h`.

This restriction does not apply to `usr4.h`.

[Workaround]

Do one of the following:

1. If the number of double quotation marks (`"`) at the beginning of the source file (before a given `#include` statement) is odd, add another to make the number even.
2. If a < is specified at the beginning of the source file (before a given `#include` statement) and no > is specified afterwards, add one.

[Correction]

This issue will be corrected in the PM+ module included in CA850 Ver. 3.40.