RENESAS Tool News

RENESAS TOOL NEWS on May 22, 2006: RSO-H8C-060522D

Notes on Using the C/C++ Compiler Packages for the H8SX, H8S, and H8 MCU Families

Please take note of the sixteen problems described below in using the C/C++ compiler packages V.4 through V.6 for the H8SX, H8S, and H8 MCU families.

1. Products and Versions Concerned

C/C++ compiler packages V.4 through V.6.01 Release 01

- V.4 (Windows, Solaris, and HP-UX edition)
- V.5 (Windows edition)
- V.6 (Windows, Solaris, and HP-UX edition)

2. Problems

2.1 On Referencing Variables No. 1 (H8C-0028)

Version Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
If you load the address of a variable into a register, its lowermost 1 or 2 bytes only are loaded. So an incorrect address will be referenced.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Any of the CPU options H8SXN, H8SXM, H8SXA, H8SXX, and AE5 is selected (for example, -cpu=h8sxn).

(2) The -optimize=1 optimizing option is selected.

(3) Either the conditions described in (3-1) or those

in (3-2) below are satisfied:

(3-1)
    (a) Two or more MOV.sz #const, ERn or SUB.sz ERn,ERn
        instructions are used.
    (b) The size sz in (a) is of word or long word.
    (c) The uppermost 1 or 2 bytes; or the lowermost 1 or 2 bytes
        of the two or more instructions in (a) are the same as each
        other in a register.
(3-2)
    (a) Between two MOV instructions is placed an instruction
        that uses the effective address (EA) of post-increment
        @ERn+, post-decrement @ERn-, pre-increment @+ERn, or
        pre-decrement @-ERn.
    (b) The register into which the effective address in (a) is
        loaded is the same as the register designated by the
        destination operands of the two MOV instructions.

Example of C Source Program:

```
----------------------------------------------------
 struct A {
 . . . . . .
    union {
        unsigned char c1;
           struct {
              unsigned char DS:1;
              unsigned char CS:1;
              unsigned char BS:1;
 . . . . . .
           }UN2;
     }UN1;
 };
 #define PTRA     (*(volatile struct
A*)0xFFF200)
 void f(){
    . . . . . .
    PTRA.UN1.UN2.DS = 0x00; //(A)
    . . . . . .
    PTRA.UN1.UN2.BS  = ...;
    . . . . . .
    PTRA.UN1.UN2.DS = 0x01; //(B)
 }
----------------------------------------------------
----
Code Generated:
----------------------------------------------------
----
 _f
 . . . . . .
 MOV.L      #H'00FFF211,ER0 ; Condition (3-1)
                      ; Address of code p-
>UN1.UN2.DS loaded
                      ; (Generated code from (A) in
Example)
 MOV.B      #1:8,R1L
 . . . . . .
 MOV.B      R3L,R0L
 EXTU.L     #2,ER0         ; ER0 is set to another
value
 MOV.L      ER0,ER4
 . . . . . .
 MOV.W       #H'F211:16,R0 ; Condition (3-1);
                     ; Lower 2 bytes of address
loaded
                     ; (Generated code from (B) in
Example)
 BSET.B     #7,@ER0
----------------------------------------------------
----
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) Use the #pragma option nooptimize directive in

order not to optimize every function in which this problem occurs.

(2) Define a dummy function in another file and make the function call to it immediately before the C statement from which an incorrect code is generated. (This method is effective only when the -goptimize Inter-module optimization information is selected.)

Example:

```
void f(){
. . . . . .
PTRA.UN2.DS = 0x00; // (A)
. . . . . .
PTRA.UN4.BS  = ...;
. . . . . .
dummy(); // dummy() defined in another file
PTRA.UN2.DS = 0x01; // (B)
}
```

(3) Use the -optimize=0 optimizing option (this performs no optimizations).

2.2 On Referencing Members of a Structure or Union Whose Size Is 4 Bytes or Less (H8C-0029)

Version Concerned:
V.6.01 Release 01

Description:
If members of a structure or union whose size is 4 bytes or less are referenced, incorrect results will be obtained, and if these members are used as operands of operations, no operations will be performed.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Any of the CPU options 2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXX, and AE5 is selected (for example, -cpu=2000n). Note that if the -legacy=v4 Compatibility of output object

code option is used, 2000N, 2000A, 2600N, and 2600A options are not involved.

(2) A structure or union whose size is 4 bytes or less is defined and declared.

(3) The structure or union in (2) includes a member whose size is different from that of the structure or union.

(4) The structure or union in (2) is not qualified to be volatile.

(5) The objects of the structure or union type in (2) are assigned to a register or passed as arguments to functions using the -structreg Allocating structure parameter or return value to register option.

Example of C Source Program:

```
---------------------------------------------------
 void main(){
   union tag_UNION {          // Conditions (2) and (4)
      signed int      m_si ;  // Condition (3)
      signed char     m_sc ;
      unsigned short  m_us ;
      long            m_sl ;
   } union_data ;
   union_data.m_si = 0;
   union_data.m_us = 0;
   . . . . . .
   union_data.m_si = union_data.m_us;
   printf(": %u ¥n",union_data.m_us);
   union_data.m_us -= 9952;          // (A)
   printf(": %u ¥n",union_data.m_us);// Condition (5)
 }
---------------------------------------------------
```

Code Generated:

```
---------------------------------------------------
 . . . . . .
 mov.l   #_printf:32,er3
 jsr     @er3

 ; Operation in (A) in Example not performed
 mov.l   #C_00000018:32,er0
 mov.l   er0,@sp
 mov.l   @(6:16,sp),er2
 mov.w   e2,@(4:16,sp)
 jsr     @er3
 mov.l   er2,er0
 mov.b   #h'0f:8,r1l
-----------------------------------------------------
```

Workarounds:
This problem can be circumvented either of the following ways:

(1) Qualify the structure or union involved to be volatile.

(2) Make the size of the structure or union involved greater than 4 bytes.

2.3 On Referencing Variables No. 2 (H8C-0030)

Versions Concerned:
V.6.01 Release 00 through V.6.01 Release 01

Description:
If any variable is referenced, an incorrect displacement of 2 bits wide is generated.

Conditions:
This problem occurs if the following conditions are all satisfied.

(1) Any of the CPU options H8SXN, H8SXM, H8SXA, H8SXX, and AE5 is selected.

(2) The -optimize=1 optimizing option is selected.

(3) The destination of transferring an operand of the MOV instruction or storing operational results is "@(symbol,ERn)" or "@(symbol+constant,ERn)".

Example:

```
-----------------------------------------------------
 typedef struct {
    char *c1;
    char *c2;
 }ST;

 ST st1[3];
 void func(long l1,long l2){
    ST st2[5];
    st1[l1].c2 = st2[l2].c2;
 }
```

Code Generated:

```
 _func:
   sub.w   #h'0028:16,r7
   shll.l  #3:5,er1
   add.l   sp,er1
   shll.l  #3:5,er0
   mov.l   @(4:16,er1),@(_st1+4:2,er0) ; Condition (3)
   ; "@(_st1+4:2,er0)" must be "@(_st1+4:32,er0)"
   add.w   #h'0028:16,r7
   rts
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) Use the #pragma option nooptimize directive in order not to optimize every function in which this problem occurs.

(2) Assign the variable to be referenced to a volatile-qualified variable; then use it for assignments or operations.

Example:

```
   typedef struct {

      char *c1;
      char *c2;
   }ST;

   ST st1[3];
   volatile char *dummy;

   void func(long l1,long l2){
      ST st2[5];
      dummy = st2[l2].c2;  // Variable is
assigned to volatile-
```

```
                        // qualified variable; then
    assignments
        st1[l1].c2 = dummy;  // or operations
    performed
      }
```

(3) Use the -optimize=0 optimizing option (this performs no optimizations).

2.4 On Assigning a Constant to a Member of a Bit Field (H8C-0031)

Versions Concerned:
V.6.00 Release 00 through 6.01 Release 01

Description:
If any constant is assigned to a member of a bit field, the constant is only ORed with the bits of the member without clearing them.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:
   Any of the CPU options H8SXN, H8SXM, H8SXA, and H8SXX is
   selected.
(B) When V.6.01 used:
   Any of the CPU options 2000N, 2000A, 2600N, 2600A, H8SXN,
   H8SXM, H8SXA, H8SXX, and AE5 is selected.
   Note that if the -legacy=v4 Compatibility of output object
   code option is used, 2000N, 2000A, 2600N, and 2600A options
   are not involved.

(2) A structure or union is declared which has bit field members of type unsigned long or signed

long.

(3) The bit sizes of the bit field members in (2) are within a range of 3 through 31.

(4) One of the bit field members in (2) extends over the 15th and 16th bit.

(5) To the bit field member in (4) is assigned a constant given by the value "-(2**(a bit size within the range in (3)) - 1)".
Here ** denotes a power.

Example:

```
---------------------------------------------------
struct st{        // Condition (2)
   long l1:14;

   long l2:3;    // Conditions (3) and (4)
}st1 = { 0, 2 };

void main(void){
   st1.l2 = -7;  // Condition (5)
}
---------------------------------------------------
```

Code Generated:

```
---------------------------------------------------
_main:
mov.l   @_st1:32,er1
or.l    #h'00008000:32,er1 ; st1.l2 not cleared to 0 and
ORed with -7
mov.l   er1,@_st1:32

---------------------------------------------------
```

Workaround:
To circumvent this problem, assign the constant involved to a volatile-qualified variable; then assign this variable to a bit field member.

Example:

```
struct st{
   long l1:14;
   long l2:3;
}st1 = { 0, 2 };
```

```
volatile long l;
void main(void){
   l = -7;     // Assign constant to volatile-qualified
   st1.l2 = l; // variable; then use it
}
```

2.5 On Using a Union Containing Two or More Members of the Same Structure or Union Type (H8C-0032)

Versions Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
In a union containing two or more members of the same structure or union type, these members are referenced using an incorrect offset value from the top of the union to them.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:
   Any of the CPU options H8SXN, H8SXM, H8SXA, and H8SXX is
   selected.
(B) When V.6.01 used:
   Any of the CPU options 2000N, 2000A, 2600N, 2600A, H8SXN,
   H8SXM, H8SXA, H8SXX, and AE5 is selected.
   Note that if the -legacy=v4 Compatibility of output object code
   option is used, 2000N, 2000A, 2600N, and 2600A options are not
   involved.

(2) A union type is used.

(3) The union in (2) contains a member of a structure or union type.

(4) The union in (2) contains another member of the same structure or union type as the member in (3).

(5) The offset values from the top of the union to the members in (3) and (4) are the same as each other.

(6) In the members in (3) and (4), the member declared secondly or later is defined or referenced.

Example of C Source Program:

```
---------------------------------------------------
typedef struct {
   unsigned int x;
} ST;
typedef union {       // Conditions (2) and (3)
   struct {
      unsigned char a;
      ST s2[1];
   } st1;           // Condition (4)
   struct {          // Condition (5)
      unsigned char c;
      ST s3;
   } st2;           // Condition (4)
} UN;
void func(){
   volatile int a=0;
   UN u;
   f(u.st2.s3.x);     // Condition (6)
}
---------------------------------------------------
```

Workaround:
To circumvent this problem, define a structure or union type with another name though the names of its members are the same as those of the members involved; then make the typedef declarations of the structure- or union-type members involved with different names from their originals.

Example:

```
      typedef struct {
         unsigned int x;
```

```
} ST;
typedef struct {
   unsigned int x;
} ST1;
typedef union {
   struct {
      unsigned char a;
      ST s2[1];
   } st1;
   struct {
      unsigned char c;
      ST1 s3;
   } st2;
} UN;
```

2.6 On Dividing the Product of an Expression and an Integer Constant by the Same Integer Constant (H8C-0033)

Versions Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
Dividing the product of an expression and an integer constant by the same integer constant brings an incorrect result.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:
Any of the CPU options H8SXN, H8SXM, H8SXA, and H8SXX is selected.

(B) When V.6.01 used:
Any of the CPU options 2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXX, and AE5 is selected.
Note that if the -legacy=v4 Compatibility of output object

code option is used, 2000N, 2000A, 2600N, and 2600A options
are not involved.

(2) An expression of type unsigned int is multiplied by any integer constant except 0.

(3) The product in (2) is divided by the integer constant in (2).

(4) The product in (2) exceeds the maximum value allowed to the type of the multiplication.

Example of C Source Program:

```
----------------------------------------------------
  unsigned long a=65536ul;
  unsigned long b;
  void func() {
     b=(65537ul*a)/65537ul;  // b=0
((65537*65536)/65537→0/65537=0)
                          // is correct, but it is replaced
                          // by incorrect b=a (=65536)
  }
----------------------------------------------------
```

Workaround:
To circumvent this problem, assign the product involved to a volatile- qualified variable; then divide this variable by the integer constant.

Example:

```
     void func() {
        volatile unsigned long c = 65537ul*a;
        b = c / 65537;
     }
```

2.7 On Iteration Statements Having an Volatile-Qualified Controlled Variable (H8C-0034)

Versions Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
In iteration statements having an volatile-qualified controlled variable, iterations are performed by an

incorrect number of times.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:
   Any of the CPU options H8SXN, H8SXM,
H8SXA, and H8SXX is
   selected.
(B) When V.6.01 used:
   Any of the CPU options 2000N, 2000A, 2600N,
2600A, H8SXN,
   H8SXM, H8SXA, H8SXX, and AE5 is selected.
   Note that if the -legacy=v4 Compatibility of
output object
   code option is used, 2000N, 2000A, 2600N,
and 2600A options
   are not involved.

(2) The -optimize=1 optimizing option is selected.

(3) An iteration statement exists in the program.

(4) The iteration statement in (3) has a controlled variable of type int.

(5) The controlled variable in (4) is qualified to be volatile. Or, the controlled variable in (4) is an external variable and is used together with the volatile=1 option.

(6) The reset expression of the controlled variable in (4) is an additive expression (addition or subtraction).

(7) In the iteration statement in (3) exists a function call. Or in the program exists a pointer-type variable pointing to the controlled variable in (4), and this pointer-type variable is reset in the iteration statement in (3).

Example of C Source Program:

```
-------------------------------------------------------
```

```
      extern void sub();
      volatile int i;  // Condition (5)
      void func() {
         i = 1;
         while (i) {  // Conditions (3) and (4)
            i--;     // Condition (6)
            sub();   // Condition (7)
                   // Controlled variable i may be reset at
source
                   // of function call, number of iterations
always 1
         }
         return;
      }
    ---------------------------------------------------
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) After the reset expression in Condition (6), place an expression that references the controlled variable in Condition (4).

(2) Use the #pragma option nooptimize directive in order not to optimize every function in which this problem occurs.

(3) Use the -optimize=0 optimizing option (this performs no optimizations).

2.8 On Using Comma Operators in the Controlling Expression of an Iteration Statement(H8C-0035)

Versions Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
If comma operators are used in the controlling expression of an iteration statement, the expressions including comma operators will be evaluated from right to left.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:
   Any of the CPU options H8SXN, H8SXM,
H8SXA, and H8SXX is
   selected.
(B) When V.6.01 used:
   Any of the CPU options 2000N, 2000A, 2600N,
2600A, H8SXN,
   H8SXM, H8SXA, H8SXX, and AE5 is selected.
   Note that if the -legacy=v4 Compatibility of
output object code
   option is used, 2000N, 2000A, 2600N, and
2600A options are
   not involved.

(2) The -optimize=1 optimizing option is selected.

(3) An iteration statement exists in the program.

(4) The iteration statement in (3) has a controlled variable of type int.

(5) The reset expression of the controlled variable in (4) is an additive expression (addition or subtraction).

(6) In the iteration statement in (3), the number of iterations is 1.

(7) In the controlling expression of the iteration statement in (3) exist two or more comma-delimited expressions.

Example of C Source Program:

```
    ---------------------------------------------------
    int A, B;
    void func() {
      int i;
      for (i=0; A++, B+=A, i<1; i++) { // Conditions
(3), (4), (5), (6)
                                  // and (7); B+=A evaluated
earlier
                                  // than A++
         B++;
    }
  }
  ---------------------------------------------------
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) Use no iteration statements.

(2) Use the #pragma option nooptimize directive in order not to optimize every function in which this problem occurs.

(3) Use the -optimize=0 optimizing option (this performs no optimizations).

2.9 On Making Assignments to Volatile-Qualified Members of a Structure (H8C-0036)

Versions Concerned:
V.4 through V.6.01 Release 01

Description:
When an assignment is made to a member of a structure, the compiler generates incorrect additional information for the optimizing linker. So if the optimizing option to save and restore registers is valid in the optimizing linkage editor, incorrect code for assignment will be generated.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.4 through V.6.00 Release 03 used:
   Any of the CPU options 300, 300HN, 300HA, 2000N, 2000A, 2600N
   and 2600A is selected.
(B) When V.6.01 used:
   Any of the CPU options 300, 300HN, 300HA, 2000N, 2000A, 2600N,
   and 2600A is selected.

Here, 2000N, 2000A, 2600N, or 2600A is used together with the
-legacy=v4 Compatibility of output object code option.

(2) The -goptimize Inter-module optimization information option option is selected.

(3) The -pack=1 Boundary alignment of structure, union, and class members option is selected.

(4) An assignment expression exists, and its right term is expressions with comma operator or linear expressions.

(5) In the assignment expression in (4), references are made to volatile-qualified members of any types except unsigned char and signed char.

(6) In the left term of the assignment expression in (4) exists a volatile-qualified member.

(7) In the optimizing linkage editor, the -optimize=register option to save and restore registers is valid.

Example of C Source Program:

```
    --------------------------------------------------
-
    long a;
    main(){
      struct {
        volatile long        vf1,vf2,vf3;
      } *st;
      . . . . . .
      st->vf1 = ((st->vf2 + st->vf3),(st->vf2 + st-
>vf3));
      // Conditions (3), (4), and (5)
    }
    --------------------------------------------------
-
```

Workarounds:
This problem can be circumvented either of the following ways:

(1) Do not use the -goptimize Inter-module

optimization information option option in the files in which this problem occurs.

(2) Invalidate the -optimize=register option that saves and restores registers in the optimizing linkage editor.

2.10 On the Overflow of an Array's Subscript of type unsigned short or unsigned int with Pointer Size Being 2 Bytes (H8C-0037)

Versions Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
When option -ptr16 for specifying pointer size or keyword __ptr16 for using a pointer size of 2 bytes is selected, an incorrect area is referenced if an array's subscript of type unsigned short or unsigned int overflows.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:
   Any of the CPU options H8SXN, H8SXM,
H8SXA, and H8SXX is
   selected.
(B) When V.6.01 used:
   Any of the CPU options 2000N, 2000A, 2600N,
2600A, H8SXN,
   H8SXM, H8SXA, H8SXX, and AE5 is selected.
   Note that if the -legacy=v4 Compatibility of
output object
   code option is used, 2000N, 2000A, 2600N,
and 2600A options
   are not involved.

(2) The -ptr16 option for specifying pointer size or the __ptr16 keyword for using a pointer size of 2 bytes is selected.

(3) A subscript to an array is an expression of type unsigned short or unsigned int that contains a constant, and this expression overflows.

Example of C Source Program:

```
    ----------------------------------------------------
    #include <stdio.h>
    unsigned int i=15;
    char
xcary[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
    void main(void){
       unsigned short soeji;

       soeji = (unsigned short)(i+0xFFFF); // Condition
(3)
       if(xcary[soeji] == 14)  printf("Hello¥n");
       // Or
       if(*(xcary+soeji) == 14)  printf("Hello¥n");
    }
    ----------------------------------------------------
```

Workarounds:
This problem can be circumvented either of the following ways:

(1) Use neither the -ptr16 option nor the __ptr16 keyword.

(2) Use no expression that overflows as the variable assigned to any subscript.

Example:

```
        #include <stdio.h>

        unsigned int i;
        char xcary[15];

        void main(void){
           unsigned short soeji;

           soeji = (unsigned short)(i-1);  //
Expression that does not
                                    overflow used
           if(xcary[soeji] == 14)  printf("Hello¥n");
        }
```

2.11 On Referencing Incorrect Constants If Loops of Iteration Statements Are Expanded (H8C-0038)

Versions Concerned:
V.6.01 Release 00 through V.6.01 Release 01

Description:
If the loop of an iteration statement is expanded, an incorrect constant is referenced.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) Any of the CPU options 300, 300HN, 2000N, and 2600N is selected. Here, 2000N or 2600N is used together with the -legacy=v4 Compatibility of output object code option.

(2) The -optimize=1 optimizing option is selected.

(3) The -speed=loop=[1|2] Optimization for speed option is valid.

(4) In an iteration statement exists a subtraction of a variable from a constant.

(5) The variable in (4) is incremented by 1 in the iteration statement in (4).

Example of C Source Program:

```
-------------------------------------------------------
void p027(){
  long a;
  long b=11;

  a=2147483640;
  while(a<2147483647){
     b+=sub27(2147483647-a);  // Condition (4)
     ++a;                     // Condition (5)
  }
}
-------------------------------------------------------
```

Code Generated:

```
-------------------------------------------------------
```

```
    jsr    @_sub27:16
    inc.l  #1,er5
    add.l  #h'0000ffff:32,er4 ; Immediate value must be
h'ffffffff
    cmp.l  #h'7fffffff:32,er5
    --------------------------------------------------
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) Qualify the controlled variable of the iteration statement to be volatile.

Example:

```
        void func(){
          volatile long a;
          long b=11;
```

(2) Use the #pragma option nooptimize directive in order not to optimize every function in which this problem occurs.

(3) Use the -optimize=0 optimizing option (this performs no optimizations).

2.12 On Accessing an Incorrect Element of an Array or a Structure or Union Type (H8C-0039)

Versions Concerned:
V.6.00 Release 00 through V.6.01 Release 01

Description:
If a structure or union declared to be extern is defined after the declaration of an array of the structure or union type, and if any element of the array except the top one is accessed, incorrect code that accesses the top element of the array is generated.

Conditions:
This problem occurs if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When V.6.00 used:

Any of the CPU options H8SXN, H8SXM,
H8SXA, H8SXX, and AE5
is selected.
(B) When V.6.01 used:
Any of the CPU options 2000N, 2000A, 2600N,
2600A, H8SXN,
H8SXM, H8SXA, H8SXX, and AE5 is selected.
Note that if the -legacy=v4 Compatibility of
output object
code option is used, 2000N, 2000A, 2600N,
and 2600A options
are not involved.

(2) An array of an incomplete structure type is declared.

(3) After the declaration of the array in (2), the incomplete structure in (2) is defined.

(4) Then an element of the array in (2) is accessed.

Example of C Source Program:

```
-------------------------------------------------------
extern struct TBL g[3]; // Condition (2)
struct TBL {          // Condition (3)
  int m;
};
struct TBL tbl;

void func()
{

   tbl.m = g[1].m;     // Condition (4)
   // g[0].m accessed in error
}
-------------------------------------------------------
```

Workaround:
To circumvent this problem, define the incomplete structure before declaring the array.

Example:

```
struct S {
  int m;
};
```

```
extern struct S a[10];
```

2.13 On Using an Expression Whose Evaluation Result Is TRUE or FALSE as a Return Value from a Function (H8C-0040)

Versions Concerned:
V.4 Release 00 through V.6.01 Release 01

Description:
If the expression whose evaluation result is TRUE or FALSE is used as a return value from a function, a value is returned without its type being converted to the one that matches with the return type.

Conditions:
This problem occurs if the following conditions are all satisfied:

(1) This problem occurs if the following conditions are all satisfied:

(A) When any of the versions 4 through 6.00 Release 03 used:
   Any of the CPU options 300, 300HN, 300HA, 2000N, 2000A, 2600N,
   and 2600A is selected.
(B) When V.6.01 used:
   Any of the CPU options 300, 300HN, 300HA, 2000N, 2000A, 2600N,
   and 2600A is selected.
   Here, 2000N, 2000A, 2600N, or 2600A is used together with the
   -legacy=v4 Compatibility of output object code option.

(2) The -optimize=0 optimizing option is selected (this performs no optimizations).

(3) The evaluation result of an equality expression is used as a return value from a function.

(4) The return value in (3) is of type long, float, or double; and can be of type pointer if any of the

CPU options 300HA, 2000A, and 2600A is selected.

Example of C Source Program:

```
    --------------------------------------------------
-------
      int wff1,wff2;

      float func(){          // Condition (4)
        return(wff1 == wff2); // Condition (3)
      }
    --------------------------------------------------
-------
```

Workaround:
To circumvent this problem, assign the evaluation result of the equality expression to a local variable and then use this variable as a return value.

Example:

```
     int wff1,wff2;

     float func(){
       int rtn; // Define local variable to which evaluation
result
                // of equality expression is assigned

       rtn = (wff1 == wff2);
       return(rtn); // Use value of local variable as return
value
     }
```

2.14 **On Referencing a Bit Field Member of a Union Type Where Values Are Assigned from the Lowermost Bit (H8C-0041)**

Versions Concerned:
V.6.01 Release 00 through V.6.01 Release 01

Description:
If an 8-bit or more available area exists in the upper part of a bit field member of a union type, and values are assigned from the lowermost bit to the upper of the bit field member, an incorrect area will be referenced.

Conditions:
This problem occurs if the following conditions are all satisfied:

(1) Any of the CPU options 2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXX, and AE5 is selected.
Note that if the -legacy=v4 Compatibility of output object code option is used, 2000N, 2000A, 2600N, and 2600A options are not involved.

(2) The -bit_order=right Bit field order specification option is , selected or the #pragma bit_order right directive is used.

(3) A union object is declared.

(4) A member of the union in (3) is a bit field.

(5) The object of the union type in (3) is declared to be an external variable, or an internal variable with a storage class of static.

(6) The bit size of the bit field member in (4) are as follows according to its types:

(6-1) If the -pack=1 Boundary alignment of structure, union, and
class members option or the #pragma pack 1 directive is used
for the union in (3):
- signed/unsigned short type: from 1 through 8 bits
- signed/unsigned int type: from 1 through 8 bits
- signed/unsigned long type: from 1 through 24 bits
(6-2) If the -pack=1 Boundary alignment of structure, union, and
class members option and the #pragma pack 1 directive are
not used for the union in (3):
- signed/unsigned long type: from 1 through 16 bits

Example of C Source Program:

```
    ---------------------------------------------------------
    #include <stdio.h>
    typedef union {         // Condition (3)
       unsigned short us:4;// Conditions (4) and (6)
     } UNI1;                // Condition (5)


    UNI1 uni1 = { 8 };

    void main(){
       if(uni1.us == 8)

          printf("OK¥n");

    }
    ---------------------------------------------------------
    Code Generated
    ---------------------------------------------------------
    SUBS    #4,sp
    MOV.B   @_uni1+2:32,r0l ; "@_uni1+2" must be
"@_uni1+1"
    AND.B   #h'0f:8,r0l
    ---------------------------------------------------------
```

Workaround:
To circumvent this problem, assign the bit field member to a non- static local variable of the same union type and then access it.

Example:

```
       #include <stdio.h>
       typedef union {

          unsigned short us:4;
       } UNI1;

       UNI1 uni1 = { 8 };


       void main(){
          UNI local_uni1=uni1;
          if(local_uni1.us == 8)
             printf("OK¥n");
```

```
    }
```

2.15 On the Division Operations Using a Divisor with the Expressible Minimum Constant Value (H8C-0042)

Versions Concerned:
V.4 through V.6.01 Release 01

Description:
When the dividend and divisor of a division operation are the same expressible minimum value, an incorrect result will be obtained.

Conditions:
This problem occurs if the following conditions are all satisfied:

(1) Either Condition (A) or Condition (B) below is satisfied.

(A) When any of the versions 4 through 6.00 Release 03 used:
 Any of the CPU options 300, 300HN, 300HA, 2000N, 2000A, 2600N,
 and 2600A is selected.
(B) When V.6.01 used:
 Any of the CPU options 300, 300HN, 300HA, 2000N, 2000A, 2600N,
 and 2600A is selected.
 Here, 2000N, 2000A, 2600N, or 2600A is used together with the
 -legacy=v4 Compatibility of output object code option.

(2) The -optimize=0 optimizing option is selected (this performs no optimizations).

(3) The Optimization for speed option (-speed or -speed=expression) is selected.

(4) A division operation is performed, where the dividend is of type signed short, signed int, or signed long.

(5) The dividend in (4) is a constant, and its value is the minimum expressible in the type of the

constant.

(6) The dividend and the divisor are given the same value.

Example of C Source Program:

```
----------------------------------------------------
short rtn;

void func(){
   short s;

   s = 0x8000;            // Condition (6)
   rtn = s/(short)0x8000; // Conditions (4), (5), and
(6)
   // Instead of 1, incorrect -1 is assigned to rtn
}
----------------------------------------------------
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) Use the -optimize=1 optimizing option.

(2) Use neither the -speed nor the -speed=expression option.

(3) Assign the constant of the divisor to a variable; then perform the division operation.

Circumvention of Example above:

```
     short rtn;

     void func(){
        short s;
        short tmp;

        s = 0x8000;
        tmp = (short)0x8000;  // Assign divisor to
tmp variable
        rtn = s / tmp;        // Perform operation
between variables
     }
```

2.16 On Referencing the Value of a Rewritten Variable (H8C-0043)

Versions Concerned:
V.4 through V.6.01 Release 01

Description:
Even if the value of a variable is rewritten in memory, memory does not load it into the register, resulting in an incorrect value being referenced.

Conditions:
This problem may occur if the following conditions are all satisfied:

(1) The -optimize=1 optimizing option is selected.

(2) Either Condition (a) or Condition (b) below is satisfied.

(a) The -eepmov Block transfer instruction option is selected
   to generate an eepmov instruction.
(b) Any of the include functions for generating block transfer
   instructions, eepmov, eepmovb, eepmovw, eepmovi, eepromb, ,
   eepromw, eepromb_exr, eepromw_exr, movmdb, movmdw, movmdl,
   and movsd, is used.

Example of C Source Program:

```
---------------------------------------------------
#include <stdio.h>
struct ST {
   char c[2];
};
struct ST  st1,st2,st3;
void sub1()
{
   struct ST  *pst;

   pst = &st3;
   st2 = *pst;
   *pst = st1;
```

```
  }
  void main()
  {
     st1.c[0] = 1;
     st2.c[0] = 2;
     st3.c[0] = 3;
     sub1();
     if ( st1.c[0]==1 && st2.c[0]==3 && st3.c[0]==1)
printf("OK¥n");
     . . . . . . . . . . . . . . . . . .
  }
  ---------------------------------------------------
  Code Generated:
  ---------------------------------------------------
  MOV.B      #3,R0H
  MOV.B      R0H,@_st3:32
  MOV.L      #_st3,ER6
  MOV.L      ER4,ER1
  MOV.L      ER6,ER2
  . . . . . . . . . . . . . . . . . . . . . .

  EEPMOV.B ; Condition (2), eepmov instruction
  . . . . . .
  if ( st1.c[0]==1 && st2.c[0]==3 && st3.c[0]==1)
printf("** OK **¥n");
  MOV.B      @ER5,R5L
  CMP.B      R0L,R5L
  BNE        L55:8
  MOV.B      @ER4,R4L
  CMP.B      R0H,R4L
  BNE        L55:8
  MOV.B      R0H,R1L ; Not read from memory but
copied
                     ; from register in error
  CMP.B      R0L,R1L
  ---------------------------------------------------
```

Workarounds:
This problem can be circumvented any of the following ways:

(1) Use the #pragma option nooptimize directive in order not to optimize every function in which this problem occurs.

(2) If the -eepmov Block transfer instruction option is selected, deselect it.

(3) If any of the include functions for generating block transfer instructions is used, discontinue using it.

(4) Define a dummy function in another file and make the function call to it immediately before the C statement from which an incorrect code is generated. Note that this function must not be expanded inline.

Example:
```
  void main(){
      st1.c[0] = 1;
      st2.c[0] = 2;
      st3.c[0] = 3;
      sub1();
      dummy();  // dummy() defined in another
file
      if ( st1.c[0]==1 && st2.c[0]==3 &&
st3.c[0]==1) printf("OK¥n");
      else
printf("NG¥n");
  }
```

(5) Use the -optimize=0 optimizing option (this performs no optimizations).

3. Schedule of Fixing the Problems

We plan to fix the above problems in the release of the next version, V.6.01 Release 02.