To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

HITACHI SEMICONDUCTOR TECHNICAL UPDATE

Classification of Production	Development Environment			No	TN-CSX-045A/E	Rev	1
THEME	H8S,H8/300 Series C/C++ compiler Bug Report		Classification of Information	1. Spec change 2. Supplement of Documents (3.) Limitation of Use 4. Change of Mask 5. Change of Production Line			
PRODUCT NAME	PS008CAS5-MWR PS008CAS4-MWR PS008CAS4-SLR PS008CAS4-H7R PS008CAS3-MWR PS008CAS3-SLR PS008CAS3-H7R	Lot No. All	Reference Documents	H8S, H8/300 Series C/C++ Compiler Assembler Optimizing Linkage Editor ADE-702-247 Rev. 1.0 H8S,H8/300 Series C/C++ Compiler User's Manual ADE-702-059D Rev.5.0		Effective Date Eternity	

H8S,H8/300 series C/C++ compiler Version 1.0 to 4.0.03 have some bugs.

Refer to the attached document, PS008CAS5-021007E, for details.

A user who has the following product should be informed.

H8S,H8/300 series C/C++ compiler Package (for Windows) Ver.5.0, 5.0.01, 5.0.02

Host computer (media)	H8S,H8/300 series C/C++ compiler Package Product name
Windows (CD)	PS008CAS5-MWR

H8S,H8/300 series C/C++ compiler Package(for Windows) Ver.4.0, 4.0r1, 4.0A, 4.0Ar1, 4.0Ar2
H8S,H8/300 series C/C++ compiler Package(for SPARC) Ver.4.0, 4.0A, 4.0Ar1, 4.0B, 4.0.05, 4.0.06
H8S,H8/300 series C/C++ compiler Package(for HP9000) Ver4.0, 4.0A, 4.0Ar1, 4.0B, 4.0Br1, 4.0.05, 4.0.06

Host computer (media)	H8S,H8/300 series C/C++ compiler Package Product name
Windows (CD)	PS008CAS4-MWR
SPARC (CD)	PS008CAS4-SLR
HP9000 (CD)	PS008CAS4-H7R

H8S,H8/300 series C/C++ compiler Package(for Windows) Ver.3.0, 3.0A, 3.0B, 3.0Br1, 3.0C, 3.0Cr1.
H8S,H8/300 series C/C++ compiler Package(for SPARC, HP9000) Ver.3.0, 3.0B, 3.0C

Host computer (media)	H8S,H8/300 series C/C++ compiler Package Product name
Windows (CD)	PS008CAS3-MWR
SPARC (CD)	PS008CAS3-SLR
HP9000 (CD)	PS008CAS3-H7R

Attached:
"Notes on bugs of H8S,H8/300 Series C/C++ compiler" (PS008CAS4-021007E) , 5 pages

Notes on bugs of H8S,H8/300 Series C/C++ compiler

The H8S,H8/300 Series C/C++ compiler has some bugs shown below. These bugs will be fixed in the version 4.0.04 compiler. This version of the compiler will be shipped in the H8S,H8/300 Series C/C++ compiler Package Version 5.0.03 for Windows and Version 4.0.07 for UNIX.

1. Incorrect code related to a unary minus operator

[The versions concerned] The version 1.0 to 4.0.03 of the H8S,H8/300 Series C/C++ compiler

[The symptom of the bug]

The evaluation of a unary minus operator might be wrong if the unary minus operator is applied after a type conversion from a char, short or int type variable, including an implicit type conversion.

[The example of the bug #1]

[The source program]

```
char c:
short s;
int i;
long l;
void sub()
{
    s = –c;              /* when c=–128, s=–128 (s=128 is correct) */
    i = –c;              /* when c=–128, i=–128 (i=128 is correct) */
    l = – (long)c;       /* when c=–128, s=–128 (s=128 is correct) */
    l = – (long)s;       /* when s=–32768, l=–32768 (l=32768is correct) */
    l = –(long)i;        /* when i=–32768, l=–32768 (l=32768is correct) */
}
```

[The incorrect code] s=–c in the above source program is incorrectly compiled as follows.

```
MOV.B   @_c,R0L
NEG.B   R0L
EXTS.W  R0
MOV.W   R0,@_s
```

[The correct code] s=–c in the above source program should be compiled as follows

```
MOV.B   @_c,R0L
EXTS.W  R0
NEG.W   R0
MOV.W   R0,@_s
```

[The condition in which the bug occurs]

The bug occurs if both of the following conditions are satisfied.

(1) A unary minus operator is applied after one of the following type conversions is done

- A char type variable is converted into the short, int, or long type
- A short type variable is converted into the long type
- An int type variable is converted into the long type

(2) Each of the above variables has the value shown below

- The char type variable has –128.
- The short type variable has –32768.
- The int type variable has –32768.

[How to avoid the bug]

Avoid the bug through one of the following methods.

(1) Apply a type conversion before a unary minus operator.

[The example of code change] s=–c; in the example of the bug #1 is changed as follows.

```
s = c;
s = –s;
```

(2) Invalidate the optimization by specifying the –optimize=0 option.

Other examples in which the bug occurs are shown below

[The example of the bug #2]

```
char  c;
int  i1,i2;

i1 = –c + i2;
```

[The example of the bug #3]

```
char c;
int  func();

int func()
{
      return( –c );
}
```

2. Incorrect generation of an AND instruction

[The versions concerned] The version 1.0 to 4.0.03 of the H8S,H8/300 Series C/C++ compiler

[The symptom of the bug]

An AND instruction might be incorrectly generated depending on the environment of the compiler (see the [Remark] below) if a bit-wise AND operator is applied after a shift operator or if the same compound assignment operators of &=, |= or ^= is applied to the same variable in succession.

[The example of the bug #1]

[The source program]

```
unsigned int X;
sub( unsigned int Y )
{
      X = (Y >> 14) & 0x2 ;
}
```

[The incorrect code]

```
MOV.W    @_Y,R0
ROTL.W   #H'2,R0
AND.L    #H'20002,ER0   ; Incorrect code
MOV.W    R0,@_X
```

[The correct code]

```
MOV.W    @_Y,R0
ROTL.W   #H'2,R0
AND.W    #H'2,ER0       ; Correct code
MOV.W    R0,@_X
```

[The example of the bug #2]

[The source program]

```
sub( int Y )
{
      Y &= 0x3;
      Y &= 0x2;
      return Y ;
}
```

[The incorrect code]

```
AND.L    #H'20002,ER0   ; Incorrect code
```

[The correct code] s=-c in the above source program should be compiled as follows

```
AND.W    #H'2,R0        ; Correct code
```

[The condition in which the bug occurs]

[Case #1]

The bug occurs if both of the following conditions are satisfied.

(1) A bit-wise AND operator is applied after a constant shift operator is applied to a variable.

(2) The type of the variable and the direction of the shift operator have the following relationship.

The << is applied to a signed char/short/int type variable.

The >> is applied to an unsigned char/short/int type variable.

[Case #2]

The bug occurs if both of the following conditions are satisfied.

(1) A variable is on the left hand side and a constant is on the left hand side of a bit-wise compound assignment operator of &=, |= or ^=.

(2) Consecutively, a compound assignment expression of the same operator is specified. The same variable is on the left hand side and a constant is on the left hand side of the same bit-wise compound assignment operator.

[Remark]

The occurrence of the bug depends on the contents of the host computer’s memory area freed by the compiler.

Even with the same source program and the same options are specified, the bug may or may not occur depending on the environment (or host operating system) of the compiler.

[How to avoid the bug]

[Case #1]

Avoid the bug through one of the following methods.

(1) Put the result of the shift operation into a volatile variable, and apply the bit-wise AND operator to the volatile variable.

[The example of code change]

```
volatile unsigned int dummy;
dummy = Y >> 14;
X = dummy & 0x2;
```

(2) Invalidate the optimization by specifying the –optimize=0 option.

[Case #2]

Avoid the bug through one of the following methods.

(1) Combine the two consecutive bit-wise compound assignments into one.

[The example of code change]

```
Y &= 0x2;
```

(2) Invalidate the optimization by specifying the –optimize=0 option.

3. Inactivation of the loop invariant code motion

[The versions concerned] The version 3.0 to 4.0.03 of the H8S,H8/300 Series C/C++ compiler

[The symptom of the bug]

The optimization of the loop invariant code motion may not work depending on the environment of the compiler (see the [Remark] below).

[The example of the bug #1]

[The source program]

```
for (  ...  ;  ...  ;  ...  ){
      for (  ...  ;  ...  ;  ...  )
        X = 0;
 }
```

A loop like this is compiled as follows.

[The code that the optimization does not work]

```
        …
        BRA     L1              ; (A)
L2:
    …
    BRA         L3
L4:
    …
    SUB.W       E0,E0           ; (X)
    MOV.W       E0,@_X
L3:
    …
        Bcc     L4
L1:
        …
        Bcc     L2              ; (B)
```

If the register E0 at (X) above is not used at any other point between (A) and (B), the code of (X) can be moved to the loop entrance as shown below.

[The code the optimization has worked]

```
        …
        SUB.W   E0,E0           ;←---------+ code motion
        BRA     L1              ; (A)      |
L2:                                        |
    …                                      |
    BRA         L3                         |
L4:                                        |
    …                                      |
                                ; (X)------+
    MOV.W       E0,@_X
```

```
L3:
   …
        Bcc     L4
L1:
        …
        Bcc     L2              ; (B)
```

This optimization, or code motion, makes the loop execution faster. But the bug may prevent the code motion from working even though the conditions for the code motion to work is satisfied.

[The condition in which the bug occurs]

The loop invariant code motion does not work if all of the following conditions are satisfied.

(1) A loop (e.g. for , while and so on) exits.
(2) A loop invariant code such as SUB.W E0,E0 above exists in the loop.
(3) Some code is deleted from the loop through another optimization.

[Remark]

The occurrence of the bug depends on the contents of the host computer's memory area freed by the compiler. Even with the same source program and the same options are specified, the bug may or may not occur depending on the environment (or host operating system) of the compiler.

[How to avoid the bug]

Avoid the bug through the following method.

(1) Dividing the loop into two may activate the loop invariant code motion.

4. Incorrect branch code

[The versions concerned] The version 1.0 to 4.0.03 of the H8S,H8/300 Series C/C++ compiler

[The symptom of the bug]

A conditional branch after a comparison may be incorrectly taken or incorrectly not taken if one operand of the comparison has caused an overflow in its operation (Case #1). Furthermore, the same symptom occurs if that overflow is the result of an optimization to perform an operation in the smaller size (e.g. int type operation is performed in char type operation) triggered by a type-conversion (Case #2).

[The example of the bug – Case #1]

[The source program]

```
int i1=1;
int i2=–32767;
    if ( (i1 – i2) < 0)
        printf("OK\n");
    else
        printf("NG\n");
```

The result of (i1 – i2) should be –32768 and "OK" should be printed. But the bug prints "NG".

[The incorrect code]

```
        MOV.W   @_i1,R0
        MOV.W   @_i2,R1
        SUB.W   R1,R0
        BGE     Ln
```

[The correct code]

```
        MOV.W   @_i1,R0
        MOV.W   @_i2,R1
        SUB.W   R1,R0
        MOV.W   R0,R0           ; This instruction is mistakenly deleted in the incorrect code above
        BGE     Ln
```

The V (overflow) flag of the CCR affects the BGE instruction. The BGE instruction above is used assuming that MOV.W R0,R0 above always clears the V flag. But SUB.W R1,R0 above can change the V flag. Deleting MOV.W R0,R0 does not guarantee that the V flag is always cleared when the BGE instruction is executed.

[The example of the bug – Case #2]

[The source program]

```
int c1=1;
int c2=–127;
    if ( (char) (c1 – c2) < 0)
        printf("OK\n");
    else
        printf("NG\n");
```

The result of (c1 – c2) should be –128 and "OK" should be printed. But the bug prints "NG". According to the

ANSI standard, (c1 – c2) should be an int-type operation and then no overflow occurs. But the optimization to perform the operation in the smaller size (i.e., char) can cause an overflow.

[The condition in which the bug occurs]

The bug occurs if all of the following conditions are satisfied.

(1) An expression is compared to a constant.

If (<expression> OP <constant>)

(2) For the OP, or operation, in (1) the <constant> has the following value, respectively.

OP	<constant>
<	0 or 1
>	0 or 1
<=	0
>=	0

(3) The <expression> in (1) has one of the following skeleton.

```
(char) ([unsigned]char {+|–} [unsigned]char)      /* including the case where an operand is 1 or 2 */
(int  {+|–} int)                                  /* including the case where an operand is 1 or 2 */
(int ) ([unsigned] int  {+|–} [unsigned] int)     /* including the case where an operand is 1 or 2 */
(long )([unsigned] int  {+|–} [unsigned] int)     /* including the case where an operand is 1 or 2 */
( long  {+|–} long)                               /* including the case where an operand is 1 or 2 */

(char) (–char)                                    /* the value given is –128 */
(–int)                                            /* the value given is –32768 */
(–long)                                           /* the value given 0x80000000 */

(char++)                                          /* the value given is 127 */
(int++)                                           /* the value given is 32767 */
(long++)                                          /* the value given is 0x7FFFFFFF */

(char– –)                                         /* the value given is –128 */
(int– –)                                          /* the value given is –32768 */
(long– –)                                         /* the value given is 0x80000000 */
```

[How to avoid the bug]

Avoid the bug by executing an expression before comparison.

[The example of code change – Case #1]

```
int i1=1;
int i2=–32767;
int i3;
i3 = i1 – i2;
if ( i3 < 0 )
    printf("OK\n");
else
    printf("NG\n");
```

[The example of code change – Case #2]

```
char c1=1;
char c2=–127;
char c3;
c3=(char)(c1 – c2);
if ( c3 < 0 )
    printf("OK\n");
else
    printf("NG\n");
```