RENESAS Tool News

RENESAS TOOL NEWS on June 1, 2014: 140601/tn1

Note on Using C/C++ Compiler and IDE for RX Family V2 for CubeSuite+ and RX Family C/C++ Compiler Package V2 (without IDE)

When using the C/C++ Compiler and IDE for RX Family V2 for CubeSuite+ and the RX Family C/C++ Compiler Package V2 (without IDE), take note of the following problems:

- With using the -smap and -goptimize options when there is no reference to a symbol (RXC#029)
- With specifying the __evenaccess keyword for a variable which is used in a conditional statement (RXC#031)
- With using the -smap and -goptimize options when there is access to a const variable (RXC#032)
- With specifying #pragma address for structures, unions, and arrays (RXC#033)

Note:
The numbers at the end of the above items are from a consecutive index of problems in the compiler packages for the RX family of MCUs.

---

1. Problem with Using the -smap and -goptimize Options When There Is No Reference to a Symbol (RXC#029)

1.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V2 for CubeSuite+
  CC-RX Compiler V2.00.00 through V2.01.00
- RX Family C/C++ Compiler Package V2 (without IDE)
  CC-RX Compiler V2.01.00

1.2 Description

When the -smap option for optimizing external variable access and -goptimize option for optimization between modules are specified, in some cases the reference address used for access to external variables will not be correct.

1.3 Conditions

This problem arises if the following conditions are all met:

(1) -smap compiler option to optimize access to external variables is specified.

(2) -goptimize option for optimization between modules is specified.

(3) -optimize=symbol_delete option for optimizing linkage is specified
(see Note below).

(4) An external variable is defined but not referred to in a file.

Note: When the -goptimize option is specified, -optimize=symbol_delete is effective by default unless -optimize=same_code, short_format, and branch are specified.

Example:

```
-----------------------------------------------------------------------
int x, z;
static int a, b, c;

#pragma entry main
void main(void){
   x = a;
   z = c;
}

void func2(void){   // Condition (4)
   b++           // There is no reference to func2 and the only
                 // reference to b is from func2, so both are
                 // deleted in optimization.
}
-----------------------------------------------------------------------
```

Results of compilation:

Example of compilation with -isa=rxv1, -output=abs, -goptimize, -optimize=2, and -smap

```
-----------------------------------------------------------------------
_main:
        MOV.L   #_x,R14
        MOV.L   08H[R14],[R14]
        MOV.L   10H[R14],04H[R14]  ; Incorrect address reference
                                   ; Correct address reference would
                                   ; be MOV.L 0CH[R14],04H[R14]
        MOV.L   #0,R1
        RTS

        .SECTION B,DATA
```

```
        .ORG    00000010H
_x:
        .BLKL   1
_z:
        .BLKL   1
__$a:
        .BLKL   1
__$c:
        .BLKL   1
        .END    _main
```

1.4 Workarounds

To avoid this problem, do any of the following:

(1) Exclude the -smap compiler option which optimizes access to external variables.

Note that this problem can be avoided also by changing -smap to -map.

(2) Exclude the -goptimize option which is used for optimization between modules.

(3) Exclude the -optimize=symbol_delete option which is used for optimizing linkage.

(4) Correct the source file and delete external variables to which there is no reference.

2. Problem with Specifying the __evenaccess Keyword for a Variable Which is Used in a Conditional Statement (RXC#031)

2.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V2 for CubeSuite+
  CC-RX Compiler V2.00.00 through V2.01.00
- RX Family C/C++ Compiler Package V2 (without IDE)
  CC-RX Compiler V2.01.00

2.2 Description

When a conditional statement such as an if statement, conditional operator (? :), or switch statement is used, in some cases access to a variable for which __evenaccess is specified will not proceed with the declared size.

2.3 Conditions

This problem may arise if the following conditions are all met:

(1) -optimize=1, -optimize=2, or -optimize=max is specified at compilation.

(2) The code includes a conditional statement such as an if statement, conditional operator (? :), or switch statement.

(3) There are two or more references by variables with the same

size (see Note 1) to the 2-byte, 4-byte, or 8-byte integer type (see Note 2).

(4) Variables referred to in (3) change in accord with the conditions of the conditional statement described in (2) above.

(5) The __evenaccess qualifier is specified for some among the variables in (3) above.

(6) There is no reading or writing of other volatile variables after reference to any of the variables in (3) above until the end of the block.

Notes:

1. Variables include the items below:
   - Member variables
   - Bit fields with the same type and the same bit width
2. Reference to a variable includes reference to memory at a constant address.

Example:

```
---------------------------------------------------------------------
__evenaccess struct { unsigned short mem; } x0; // Condition (5)
unsigned short mem1;
unsigned char test(unsigned char x) {
unsigned short temp;
   if (x) {            // Condition (2)
      temp = x0.mem;   // Conditions (3), (4), and (6)
   } else {
      temp = mem1;     // Conditions (3), (4), and (6)
   }
   return (char)temp;
}
---------------------------------------------------------------------
```

Output code for the example:

Example where -isa=rxv1, -optimize=2, and -size are specified.

```
---------------------------------------------------------------------
_test:
      CMP #00H, R1
      MOV.L #_x0, R14
      MOV.L #_mem1, R15
      BEQ L12
L11:   ; entry
      MOV.L R14, R15
L12:   ; entry
      MOVU.B [R15], R1  ; Memory is erroneously read in a 1-byte unit
                        ; instead of a 2-byte unit.
      RTS
```

2.4 Workarounds

To avoid this problem, do either of the following:
(1) Specify -optimize=0
(2) Insert reading of the variable qualified as volatile after any variable reference that satisfies condition (3) and before convergence following the branch under condition (2).

Example of applying workaround (2) above to the example of the problem:

```
---------------------------------------------------------------------
__evenaccess struct { unsigned short mem; } x0;
unsigned short mem1;
unsigned char test(unsigned char x) {
unsigned short temp;
   volatile char dummy;  // Workaround (2)
   if (x) {
      temp = x0.mem;
      dummy;          // Workaround (2)
   } else {
      temp = mem1;
   }
   return (char)temp;
}
---------------------------------------------------------------------
```

3. Problem with Using the -smap and -goptimize Options When There Is Access to a Const Variable (RXC#032)

3.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V2 for CubeSuite+
  CC-RX Compiler V2.00.00 through V2.01.00
- RX Family C/C++ Compiler Package V2 (without IDE)
  CC-RX Compiler V2.01.00

3.2 Description

The address used for reference in access to a const variable is incorrect in some cases when the -smap option for optimizing access to external variables and the -goptimize option used for optimization between modules are specified.

3.3 Conditions

This problem may arise if the following conditions are all met:
(1) -smap option is specified.
(2) Two or more variables defined as static (see Note 1) are also

qualified as const in the same C/C++ compilation unit.
Among these, at least one meets one of the conditions below and at least one does not.
- Qualified as volatile
- No initial value

(3) Of the two types of variable defined in (2) above, at least one of each is either read or has its address acquired in a function in the same C/C++ compilation unit.

(4) Both of the variables in (2) above are in the same section (see Note 2).

(5) Any condition among (5-1), (5-2), and (5-3) below is met.

(5-1) The const-qualified variable definitions in (2) above are in an order other than (a) then (b) below outside the function or within at least one function:
(a) Definition of a const variable with an initial value but not as volatile
(b) Definition of the const variable other than (a) above

(5-2) The const-qualified variables in (2) are defined both outside and inside the function, and the definitions meet both conditions (a) and (b) below:
(a) Variable definition in the function includes const and an initial value but not volatile.
(b) Variable definition outside the function includes const other than (a) above.

(5-3) The const-qualified variables in (2) are defined in two or more functions, and the definitions meet both conditions (a) and (b) below:
(a) Variable defined in a function has const and an initial value but does not have volatile.
(b) Functions that were defined before the function in (a) have both the const variable with an initial value but not with volatile and the const variable that has other combinations of the initial value and volatile.

Notes:

1. Variables defined as static include variable definitions both inside and outside functions.
2. The two variables in (2) above are only considered to be allocated to the same section when both conditions (a) and (b) below are met:
   (a) Different output section names are not specified by #pragma section.
   (b) -nostuff or -nostuff=C option is effective, or the alignment number of the two variables is the same at compilation.

Example 1:
When compilation includes options as shown below:
ccrx -output=src -smap -cpu=rx600 file1.c
<file1.c>

```
------------------------------------------------------------------------
const volatile unsigned long var1 = 1; // Conditions (2), (4), and (5-1)
const long var2 = 2;                   // Conditions (2), (4), and (5-1)
void call_func1(unsigned long, const long *);
void func1(void) {
   const long *tmp = &var2;          // Condition (3)
   call_func1(var1, tmp);            // Condition (3)
}
------------------------------------------------------------------------
```

Example of output code for Example 1 above

```
------------------------------------------------------------------------
      .SECTION    P,CODE
_func1:
      MOV.L #_var1, R14
      MOV.L [R14], R1
      ADD #04H, R14, R2 ; This instruction is incorrect since
                        ; the point four bytes after the address
                        ; of _var1 is not the address of _var2.
      BRA _call_func1
      .SECTION    C,ROMDATA,ALIGN=4
_var2:
      .lword  00000002H
_var1:
      .lword  00000001H
------------------------------------------------------------------------
```

Example 2:
When compilation includes options as shown below:
ccrx -output=src -smap -cpu=rx600 -nostuff=C file2.c
<file2.c>

```
------------------------------------------------------------------------
const char  var3 = 3;                  // Conditions (2), and (4)
const unsigned short var4;             // Conditions (2), (4), and (5-1)
const char  var5 = 5;                  // Conditions (2), (4), and (5-1)
void call_func2(const char *, const char *);
void func2(void) {
   const char *tmp1 = &var5;          // Condition (3)
   const char *tmp2 = &var3;          // Condition (3)
   call_func2(tmp1, tmp2);
```

```
}
```

Example of output code for Example 2 above

```
      .SECTION       P,CODE
_func2:
      MOV.L #_var3, R2
      ADD #04H, R2, R1  ; This instruction is incorrect since
                        ; the point four bytes after the address
                        ; of _var3 is not the address of _var5.
      BRA _call_func2
      .SECTION       C,ROMDATA,ALIGN=4
_var3:
      .byte   03H
_var5:
      .byte   05H
_var4:
      .word   0000H
```

Example 3:
When compilation includes options as shown below:
ccrx -output=src -smap -cpu=rx600 file3.c
<file3.c>

```
const long var6 = 6;                    // Conditions (2), and (4)
const volatile long var7 = 7;           // Conditions(2), (4),
                                        // and (5-2)(b)
void call_func3(const volatile long *, const long *);
void func3(void) {
   static const long var8 = 8;          // Conditions(2), (4),
                                        // and (5-2)(a)
   const volatile long *tmp1 = &var7;   // Condition (3)
   const long *tmp2 = &var8;            // Condition (3)
   call_func3(tmp1, tmp2);
}
```

Example of output code for Example 3 above

```
      .SECTION       P,CODE
_func3:
      MOV.L #_var7, R1
      ADD #04H, R1, R2  ; This instruction is incorrect since
```

```
                    ; the point four bytes after the address
                    ; of _var7 is not the address of _var8.
      BRA _call_func3
      .SECTION      C,ROMDATA,ALIGN=4
_var6:
      .lword  00000006H
__$var8$1:
      .lword  00000008H
_var7:
      .lword  00000007H
```

Example 4:
When compilation includes options as shown below:
ccrx -output=src -smap -cpu=rx600 file4.c
<file4.c>

```
void call_func4(const volatile long *, const long *);
void func4(void) {
   static const long var9 = 9;          // Conditions(2), (4),
                                        // and (5-3)(b)
   static const volatile long var10 = 10; // Conditions(2), (4),
                                        // and (5-3)(b)
   call_func4(&var10, &var9);           // Condition(3)
}
   void func4a(void) {
   static const long var11 = 11;        // Condition(2), (4),
                                        // and (5-3)(a)
. . . . . .
}
```

Example of output code for Example 4 above

```
      .SECTION      P,CODE
_func4:
      MOV.L #__$var9$1, R2
      ADD #04H, R2, R1  ; This instruction is incorrect since
                    ; the point four bytes after the address
                    ; of _var9 is not the address of _var10.
      BRA _call_func4
      .SECTION      C,ROMDATA,ALIGN=4
__$var9$1:
      .lword  00000009H
__$var11$3:
```

```
        .lword  0000000BH
__$var10$2:
        .lword  0000000AH
-----------------------------------------------------------------------
```

3.4 Workarounds

To avoid this problem, do any of the following:

(1) Disable the -smap option for compilation of the corresponding C/C++ compilation unit.
Note that changing -smap to -map will also suffice.

(2) For all const-qualified static variable definitions that meet condition (2), make consistent settings in terms of the presence or absence of the volatile qualifier and initial values within a single body of C/C++ source code.

(3) When condition (5-2) is not met, reorder const-qualified static variable definitions that meet condition (2) to be in the order of (a) then (b) below both outside and inside functions.
(a) Definition of a const variable with an initial value but not with volatile.
(b) Definition of the const variable that differs from (a) above.

(4) When either of conditions (5-2) and (5-3) is met, either move or delete the corresponding part of the variable definition so that it no longer meets either (a) or (b).

Example of applying workaround (2) above to Example 1 of the problem:
Add volatile to all const-qualified variable definitions
in the corresponding C/C++ source.

```
-----------------------------------------------------------------------
const volatile unsigned long var1 = 1;
const volatile long var2 = 2;         // Add volatile
void call_func1(unsigned long, const long *);
void func1(void) {
   const volatile long *tmp = &var2;  // Due to change in the type
                                      // of var2.
   call_func1(var1, tmp);
}
-----------------------------------------------------------------------
```

Example of applying workaround (3) above to Example 2 of the problem:
Reorder the definitions of the const-qualified variables.

```
-----------------------------------------------------------------------
const char  var3 = 3;
const char  var5 = 5;
const unsigned short var4;            // Change the order of
                                      // the definitions without
                                      // initial values.
```

```
void call_func2(const char *, const char *);
void func2(void) {
   const char *tmp1 = &var5;
   const char * tmp2 = &var3;
   call_func2(tmp1, tmp2);
}
```

Example of applying workaround (4) above to Example 3 of the problem:
Move the definition from outside the function to inside the function
(see Note).

```
const long var6 = 6;
                              // Move the definition outside
                              // the function.
void call_func3(const volatile long *, const long *);
void func3(void) {
   static const long var8 = 8;
   static const volatile long var7 = 7; // Move the definition
                              // outside the function to
                              // static inside the function.
   const volatile long *tmp1 = &var7;
   const long *tmp2 = &var8;
   call_func3(tmp1, tmp2);
}
```

Note: This example of a workaround will not be applicable in some
cases due to changing the meaning of the source code.

4. Problem with Specifying #pragma Address for Structures, Unions, and Arrays (RXC#033)

4.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V2 for CubeSuite+
  CC-RX Compiler V2.01.00
- RX Family C/C++ Compiler Package V2 (without IDE)
  CC-RX Compiler V2.01.00

4.2 Description

In some cases, the address used in writing to or reading from
a structure, union, or array for which #pragma address is specified is incorrect.

4.3 Conditions

This problem may arise if the following conditions are all met:

(1) #pragma address is used.
(2) The #pragma address directive in (1) applies to a structure, union, or array.
(3) The structure, union, or array in (2) has a structure or array that
starts at a different point than its own starting point.
(4) The structure starting at a different point in case (3) above has
multiple members and reference is made to a member whose offset
is other than 0.
The array starting at a different point in case (3) above has
multiple elements and reference is made to an element whose offset
is other than 0.

Example:

```
---------------------------------------------------------------------
struct st1 {
   short a;
   short b;
};

struct st2 {
   int       c;
   struct st1 tbl2;        // Condition (3)
};

#pragma address A=0xFFFF0000 // Condition (1)
struct st2 A;              // Condition (2)

void func(void)
{
   A.a++;
   A.tbl2.b = 0;         // Condition (4)   The assignment is
                    //              erroneously to A.tbl1.b.
}
---------------------------------------------------------------------
```

4.4 Workarounds

To avoid this problem, do either of the following:
(1) Instead of using #pragma address, write the address directly as a constant.
(2) Instead of using #pragma address, arrange the corresponding
variable in another section and specify the address of the section with the -start option of the linker.

5. Schedule for Fixing the Problems

All the above problems will be fixed in the next version of
the C/C++ Compiler and IDE for RX Family V2 for CubeSuite+
(scheduled for release in July of 2014) and

the next version of the RX Family C/C++ Compiler Package V2 (without IDE) (scheduled for release in July of 2014).
