RENESAS Tool News

RENESAS TOOL NEWS on June 1, 2014: 140601/tn2

Note on Using C/C++ Compiler and IDE for RX Family V1 for CubeSuite+ and C/C++ Compiler package for RX Family for High-performance Embedded Workshop

When using the C/C++ Compiler and IDE for RX Family V1 for CubeSuite+ and the C/C++ Compiler package for RX Family for High-performance Embedded Workshop, take note of the following problems:

- With the members of structures qualified as volatile or evenaccess (RXC#026)
- With issuing system calls to the real-time OS (RXC#027)
- With function calls where the subsequent processing is the same (RXC#028)
- With using the & operator (RXC#030)

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 the Members of Structures Qualified as Volatile or Evenaccess (RXC#026)

1.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V1 for CubeSuite+
  CC-RX Compiler V1.02.00 through V1.02.01
- C/C++ Compiler package for RX Family for High-performance Embedded Workshop V.1.00 Release 00 through V.1.02 Release 01

1.2 Description

In some cases, as a return value, using a pointer to a member of a structure or passing a member by reference where that member is qualified as volatile or __evenaccess will cause either of the qualifiers to become invalid.

1.3 Conditions

This problem arises if the following conditions are all met:

(1) The return value of the function is either a pointer to
the structure (see Note 1) or the structure is passed by
reference (see Note 2).

(2) Either of the qualifiers below has been added within the structure
described in (1) above.
(a) volatile
(b) __evenaccess

(3) A member of the structure is referred to by using the "." operator
or "->" operator in the function call in (1) above.

(4) Optimization in the form of in-line expansion is applied to
the function described in (1) above.

Note 1: This applies to both the C and C++ languages.
Note 2: This only applies to the C++ language.

The problem arises when the effect of the qualifier described
in condition (2) on the member of the structure referred to
in condition (3) is lost.
The result is that the optimization performed on the assumption of
the qualifier being valid becomes incorrect, leading to the generation
of incorrect code.

Example:

```
------------------------------------------------------------------------
struct ST { unsigned char PRT; };
volatile struct ST *st2f()     /* Conditions (1), and (2) */
{
   return (volatile struct ST*)0x2000;
}
unsigned char func_st2(void)
{
   st2f()->PRT = 0x10;     /* Conditions (3), and (4) */
   st2f()->PRT = 0x11;     /* Conditions (3), and (4) */
   st2f()->PRT |= 0x22;    /* Conditions (3), and (4) */ /* (NG-1) */
   return st2f()->PRT;     /* Conditions (3), and (4) */ /* (NG-2) */
}
------------------------------------------------------------------------
```

Output code for the example:
When the inline=100 option is set for the compiler:

```
------------------------------------------------------------------------
MOV.L       #00002000H,R5
MOV.B       #10H,[R5]
```

```
MOV.B      #11H,[R5]
MOV.B      #33H,[R5]     ; (NG-1) Not being read
MOV.L      #00000033H,R1  ; (NG-2) Not being read
------------------------------------------------------------------------
```

1.4 Workarounds

To avoid this problem, do any of the following:

(1) If the return value of the function indicates the address of the structure, use a pointer to the structure when referring to the member.

(2) When defining the structure, qualify the member as volatile.

(3) Use either method below to prevent in-line expansion of the corresponding function:

(a) Specify #pragma noinline for the corresponding function. If #pragma inline was specified for the corresponding function, remove the #pragma inline directive at the same time.

(b) Specify -inline=0 at compilation.

2. Problem with Issuing System Calls to the Real-Time OS (RXC#027)

2.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V1 for CubeSuite+
  CC-RX Compiler V1.02.00 through V1.02.01
- C/C++ Compiler package for RX Family for High-performance Embedded Workshop V.1.00 Release 00 through V.1.02 Release 01

2.2 Description

When a system call to the real-time OS (see Note below) is issued as the last statement of a function, in some cases the RTS instruction will not be generated at the end of the object code for the function after return from the system call.

Note: The affected products are listed below:

- RI600V4 V1.03.00 or earlier
- RI600PX V1.02.00 or earlier
- RI600/4 V.1.01 Release 01 or earlier
- RI600/PX V1.00 Release 01 or earlier

2.3 Conditions

This problem arises if the following conditions are all met:

(1) Either the optimize=2 or optimize=max option is used.

(2) There are two or more exits from the function due to any of the control statements listed below (see Note):

- if statement
- switch statement

- for statement
- while statement

(3) Another function is called immediately before each the exit from the function.
(4) A #pragma oscall directive is specified for at least one function call described in (3) above.
(5) None of the #pragmas listed below are specified for the function described in (2) above:
- #pragma interrupt
- #pragma task
- #pragma taskexception
- #pragma almhandler
- #pragma cychandler

(6) No parameters and automatic variables of the function in (2) have the volatile qualifier.
(7) There is no reference to the address of the parameters and automatic variables of the function in (2).
(8) The function described in (2) does not have a call to setjmp().

Note:
When a control statement in the format shown below is immediately before the exit of the function listed below, there are two routes to the exit from the function. Thus, the compiler assumes there are two different exits from the function.

Example where the compiler assumes there are two exits under condition (2):

```
----------------------------------------
void func(void)
{
 . . . . . .
      /* Exit from the function */
   if (conditional expression) {
 . . . . . .
      /* Exit from the function */
   }
   return;
}
----------------------------------------
```

Example 1:
Case that produces erroneous code

```
-------------------------------------------------------------------------
#pragma oscall /s=0 5 tsk()
void tsk(void);
void sub(void);
```

```
int a;
void func()
{
   int b[1] = {0};
   if (a) {
      sub();    /* Conditions (2), and (3) */
   }else{
      tsk();    /* Conditions (2), (3), and (4) */
   }
}
void dummy(void){a=10;}
```

Results of compiling Example 1 when -optimize=2, -size, and -output=src are specified.

```
_func:
      SUB        #04H,R0
L10:
      MOV.L      #L14,R5
      MOV.L      #_a,R4
      MOV.L      [R5],[R0]
      MOV.L      [R4],R5
      CMP        #00H,R5
      BEQ        L12
L11:
      ADD        #04H,R0
      BRA        _sub
L12:
      .ASSERT     'tsk' >> ..file@.mrc
      INT        #05H
                 ; The RTS instruction that should be here is not.
_dummy:
      MOV.L      #_a,R4
      MOV.L      #0000000AH,[R4]
      RTS
```

Example 2:
Case where an error is returned at the time of assembling

```
#pragma oscall /s=0 5 tsk()
void tsk(void);
void sub(void);
int a;
```

```
void func()
{
   int b[1] = {0};
   if (a) {
      tsk();  /* Conditions (2), and (3) */
   }else{
      sub();  /* Conditions (2), (3), and (4) */
   }
}
```

When example 2 above applies, the assembler error below will be generated on assembly or in compilation with the -output=obj specification so that an object file is generated.

```
 A2111 (E) Symbol is undefined
```

Results of compiling Example 2 when -optimize=2, -size, and -output=src are specified.

```
func:
      .ASSERT    'tsk' >> ..file@.mrc
      INT        #05H
      BRA        L13  ; Specifies a non-existent label (L13)
L12:
      ADD        #04H,R0
      BRA        _sub
```

2.4 Workarounds

To avoid this problem, do either of the following:

(1) Use either optimzie=0 or optimize=1.

(2) Insert code which works as a dummy instruction immediately before any of the exits from the applicable function.

Example: The built-in function nop()

Example of applying workaround (2) above to Example 1 of the problem:

```
#include <machine.h>    /* Addition to enable the  */
                    /* built-in function nop() */
#pragma oscall /s=0 5 tsk()
void tsk(void);
void sub(void);
int a;
void func()
```

```
{
   int b[1] = {0};
   if (a){
         sub();
         /* Problem is avoidable even a call of nop() here. */
   }else{
         tsk();
         /* Problem is avoidable even a call of nop() here. */
   }
         nop();   /* Add call of nop() here. */
}
void dummy(void){a=10;}
```

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

```
#include <machine.h>    /* Addition to enable the  */
                  /* built-in function nop() */
#pragma oscall /s=0 5 tsk()
void tsk(void);
void sub(void);
int a;
void func()
{
   int b[1] = {0};
   if (a) {
         tsk();
         /* Problem is avoidable even a call of nop() here. */
   }else {
         sub();
         /* Problem is avoidable even a call of nop() here. */
   }
   nop();   /* Add call of nop() here. */
}
```

---

3. Problem with Function Calls Where the Subsequent Processing Is the Same (RXC#028)

3.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V1 for CubeSuite+
  CC-RX Compiler V1.02.00 through V1.02.01
- C/C++ Compiler package for RX Family for High-performance Embedded Workshop V.1.00 Release 00 through V.1.02 Release 01

3.2 Description

If multiple function calls share the same subsequent processing, in some cases where erroneous code is generated since the destination for calling the function when optimization is applied at the time of linkage will be incorrect.

3.3 Conditions

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

(1) All of the following options are specified in compilation:
   (a) -goptimize
   (b) -optimize=2 (including the case where -optimize is not specified) or -optimize=max
   (c) -speed
   (d) -branch=32

(2) Either of the cases below applied to the specification of options at linkage:
   (a) A -optimize option other than -optimize=symbol_delete is specified.
   (b) Neither -optimize nor -nooptimize is specified.

(3) There are two or more calls to the same function (see Note below) in a function.

(4) The function calls indicated in (3) above have the same processing in all routes from immediately after the function call to the exit from the function.

Note:
This includes calls of routines at the time of execution.
When the operations below are compiled and conditions for option specifications (1) and (2) above apply, the operation is replaced by the call of the routine at execution:

1. Single-precision floating-point-type operation when -nofpu or RX200 is selected.
2. Double-precision floating-point-type operation
3. 64-bit-integer type operation other than addition and subtraction
4. 32-bit integer-type and single-precision floating-point-type division and modular arithmetic when -nouse_div_inst is selected.

When the problem arises, the information used for optimization at linkage becomes incorrect.
As a result, the optimization processing will lead to the generation of erroneous code at the address of the branch destination of the function call to which condition (3) above applies.

Example:
Assuming that -goptimize, -optimize=2, -speed, and -branch=32 were specified for the option at the time of compilation and

neither -nooptimize nor -optimize linkage option were specified.

```
int a;
void func(int b)
{
   if (b) {
      a = sub(1);   // Conditions (3), and (4)
   }else {
      a = sub(2);   // Conditions (3), and (4)
   }
}
```

In the above case, the address for the call to sub(1) leads to a branch to itself after linkage so that erroneous code for an endless loop is generated.

3.4 Workarounds

To avoid this problem, do any of the following:

(1) Do not specify the -goptimize option at compilation.

(2) Specify any of the options below at compilation:

   (a) -optimize=0 or -opttimize=1

   (b) -size

   (c) -branch=24 or -branch=16

(3) Specify the -nooptimize option at linkage.

(4) Add a dummy instruction to one of the routes in condition (4).

Example 1 showing workaround (4):
Inserting code to update the value of a dummy global variable.

```
int dummy;    // Dummy global variable
int a;
void func(int b)
{
   if (b) {
      a = sub(1);
      dummy = 0;   // Assign a value to the dummy global variable.
   } else {
      a = sub(2);
   }
}
```

Example 2 showing workaround (4):
Example where the nop() built-in function is inserted.

```
#include <machine.h>  // Required for using nop()
int a;
void func(int b)
{
   if (b) {
      a = sub(1);
      nop();    // Insert the nop() built-in function as a dummy.
   } else {
      a = sub(2);
   }
}
```

---

4. Problem with Using the & Operator (RXC#030)

4.1 Product and Versions Concerned

- C/C++ Compiler and IDE for RX Family V1 for CubeSuite+
  CC-RX Compiler V1.02.00 through V1.02.01
- C/C++ Compiler package for RX Family for High-performance Embedded Workshop V.1.00 Release 00 through V.1.02 Release 01

4.2 Description

In some cases, an incorrect result will be obtained due to the bitwise logical AND operation (& operation) using the & operator for a variable and a constant value.

4.3 Conditions

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

(1) -optimize=2 (including the case where -optimize is not specified) or -optimize=max is specified at compilation.
(2) Assignment to a 1- or 2-byte-type variable is performed.
(3) The value assigned in (2) above is the result of an & operation.
(4) Neither operand of the & operation in (3) is of an 8-byte type.
(5) One of the operands of the & operation in (3) is a constant value.
(6) Among the bits of the constant value in (5), one has a 0 that has an effect on the variable produced by the assignment in (2).
Note 1 in the example below describes an example where this is applicable.
(7) After the above assignment, the variables produced by the assignment is expanded to a larger type.
(8) The variable which did not have a constant value in the & operation described in (3) might take on a value that is not of the type that should have been produced by the assignment in (2).
Note 2 in the example below describes a case where this is not applicable.

Example:

```
------------------------------------------------------------------------
unsigned long yyy;
void func(unsigned long xxx) {          // Condition (8) (see Note 2)
   unsigned char aaa = xxx & 0x000000fe;  // Conditions (2), (3),
                                 // (4), (5),
                                 // and (6) (see Note 1)

   yyy = aaa;                        // Condition (7)
}
------------------------------------------------------------------------
```

In the above example, code in which the 24 higher-order bits are not masked by the & operator is generated.
Thus, when the value of XXX is in the range from 0x00000100 to 0xffffffff at the time of execution of the corresponding code, the resulting code is different from that which was expected.

Note 1:

An example where description in (6) above applies is given below:
When the variable produced by the assignment is one byte.

```
   ---------------------------------------------------------------
   unsigned char  a1 = xxx & 0x012345fe; // Only 1 bit is 0 in
                                 // the lower-order byte.
   unsigned char  a2 = xxx & 0x000000fb; // Only 1 bit is 0 in
                                 // the lower-order byte.
   ---------------------------------------------------------------
```

When the variable produced by the assignment is two bytes.

```
   ---------------------------------------------------------------
   unsigned short a3 = xxx & 0x0123fffe; // Only 1 bit is 0 in
                                 // the 2 lower-order bytes.
   ---------------------------------------------------------------
```

Note 2:

An example where the description in (8) above does not apply is given below:
The problems in execution that meet the conditions appear in the code of the function func in the example.
However, an incorrect result will not be obtained, but only when the call is made from the foo function.

```
   ---------------------------------------------------------------
   unsigned long yyy;
   void func(unsigned long xxx) {          // Condition (8)
      unsigned char aaa = xxx & 0x000000fe; // Conditions (2), (3),
                                 // (4), (5), and (6)
      yyy = aaa;                        // Condition (7)
```

```
}
. . . . . .
unsigned char zzz;
void foo(void)
{
   func(zzz);  // Since the value of zzz will not surpass
               // the range of aaa, an incorrect result will not
               // be obtained, but only when this call is made.
}
```

4.4 Workarounds

To avoid this problem, do any of the following:
(1) Change the variable to which condition (2) applies to a 4-byte type.
(2) Qualify the variable to which condition (2) applies as volatile.
(3) Specify either the -optimize=0 or -optimize=1 option at compilation.

Example of workaround (1):

```
unsigned long yyy;
void func(unsigned long xxx) {
   unsigned long aaa = xxx & 0x000000fe; // Change to a 4-byte type.
   yyy = aaa;
}
```

---

5. Schedule for Fixing the Problems

We do not plan to modify the C/C++ Compiler and IDE for RX Family V1 for CubeSuite+ and the C/C++ compiler package for RX family for High-performance Embedded Workshop to change this behavior.
Please apply any of the above workarounds to avoid problems.

Please consult your local Renesas Electronics marketing office or distributor if any of the technical information is not clear.

Note that the four points described in this note have already been rectified in the C/C++ Compiler and IDE for RX Family V2 for CubeSuite+. Thus, we would like customers who are using the C/C++ Compiler and IDE for RX Family V1 for CubeSuite+ to consider upgrading their versions to V2 (see Note).

Note that the C/C++ Compiler package for RX Family for High-performance Embedded Workshop is only available as V1 but there is no V2.
Thus, we would like customers using the C/C++ Compiler package for RX Family for High-performance Embedded Workshop to consider

changing to the following products and using the V2 compiler (see NOTE):
- C/C++ Compiler and IDE for RX Family V2 for CubeSuite+
- RX Family C/C++ Compiler Package V2 (without IDE)

Note: Upgrading versions has a fee.
