RENESAS Tool News

RENESAS TOOL NEWS on May 16, 2011: 110516/tn2

Notes on Using the C/C++ Compiler Package for the RX Family of MCUs

When you use the C/C++ compiler package for the RX family of MCUs, take note of the following problems:

- With the return value of a function that is of a 1- or 2-byte integral type with #pragma option issued (RXC#012)
- With handling local variables of type union by using string-handling functions (RXC#013)
- With using the value of an array-type member of an array-type structure or union for dynamic initialization (RXC#014)
- With using the pointer to an array-type member of a structure or union (RXC#015)

---

1. Product and Versions Concerned

The following versions of the product are concerned with all the problems:

The C/C++ compiler package for the RX family
V.1.00 Release 00 and V.1.00 Release 02

2. Problem with the Return Value of a Function That Is of a 1- or 2-Byte Integral Type with #pragma Option Issued (RXC#012)

2.1 Description

When preprocessor directive #pragma option is issued, a function that must return a value of a 1- or 2-byte integral type may returns a value that cannot be represented with 1 or 2 bytes.

2.2 Conditions

This problem arises if the following conditions are all satisfied:

(1) A function is defined which returns a value of a 1- or 2-byte integral type.

(2) In a compilation unit where the function in (1) exists, #pragma option is issued, and an optimizing option (called type A for convenience) is selected by #pragma option.
Here, a compilation unit means a source file and the include files included in it.
(3) Any of the options that are effective for the source code written before the #pragma option line are converted to the type-A optimizing options.
(4) For the function in (1), optimize=2 or optimize=max is selected on the command line, or optimize=2 is selected by #pragma option.

Example sample1.c:

```
------------------------------------------------------------
int b;
unsigned short func_ushort(unsigned short a)   /* Condition (1) */
{
   unsigned short tmp;
   tmp = a + b + 1;
   return tmp;
}
void fun(void)
{
  . . . . . .
}
#pragma option optimize=1     /* Type-A option in Condition (2) */
 void dummy(void) { }
------------------------------------------------------------
```

Example of command line:

```
------------------------------------------------------------
ccrx -output=src -optimize=2 sample1.c
         /* Condition (4) */
         /* Effective for func_ushort in Example sample1.c */
------------------------------------------------------------
```

2.3 Workarounds

To avoid this problem, use any of the following ways:
(1) Assign the return value of the function to a volatile-qualified variable of type integral.
Then use this variable.

Example:

```
------------------------------------------------------------
int b;
unsigned short func_ushort(unsigned short a)
{
```

```
    volatile unsigned short tmp;
     /* Assign the return value to a volatile-qualified variable */
    tmp = a + b + 1;
    return tmp;
}
#pragma option optimize=1
void dummy(void) { }
```

---

(2) Move the function in Condition (1) to another file in which #pragma option is not issued.
(3) For the function in Condition (1), make optimize=0 or optimize=1 effective on the command line or by using #pragma option.

3. Problem with Handling Local Variables of Type Union by Using String-Handling Functions (RXC#013)

3.1 Description

When a local variable of type union is declared, and if a member of the union is read and/or written by using a string-handling function, a read and a write may be made in incorrect order.

3.2 Conditions

This problem may arise if the following conditions are all satisfied:
(1) Compile option optimize=2 or optimize=max is effective.
(By default, optimize=2 is effective.)
(2) Compile option library=intrinsic is effective.
(By default, library=intrinsic is effective.)
(3) A local variable of type union is declared.
(4) A read from or write to the memory area of a member of the union in (3) is made by using any of the string-handling functions memcpy, strcpy, and strncpy.
(5) Each function used in (4) satisfies the following:
- In memcpy, the third argument (size of data transferred in bytes) is a constant equal to or greater than 1.
- In strncpy, the third argument (size of data transferred in bytes) is a constant equal to or greater than 1, and the second argument (source of copy) is a string literal whose number of bytes including the terminating NULL character is equal to or greater than that of the third argument.

(6) Either of the following is satisfied:
(a) Compile option "speed" is effective.
(b) Compile option "size" is effective, and each size of data

transferred described in (5) is 1, 2, or 4 bytes.
(By default, "size" is effective.)

(7) A read from or write to another member of the same union is made whose memory area overlaps that of the member described in (4).

(8) The address of the member described in (7) is not referred.

Example in memcpy:

```
#include <string.h>
union U {
   char a[4];
   long b;
} u1, u2;
void main(char *p)
{
   union U u0;         /* Condition (3) */
   u1.b = u0.b;
   p+=4;
   memcpy(u0.a, p, 4);  /* Conditions (4), (5), and (6) */
   u2.b = u0.b;        /* Conditions (7) and (8) */
}
```

If this problem arises, the order of handling processes of conditions (4) and (7) is interchanged.

Example of code generated:

```
_main:
   SUB     #04H,R0
   MOV.L   [R1],[R0]
   MOV.L   [R0],R4     ; Value of u0.b (here called A) in Condition (7)
                          of above example is saved on R4.
   MOV.L   #_u1,R5
   MOV.L   R4,[R5]
   MOV.L   #_u2,R5
   MOV.L   04H[R1],[R0] ; memcpy in Condition (4)
   MOV.L   R4,[R5]     ; Value read in Condition (7)
                       ; Not the value read by memcpy, but value A
                       ; used.
   RTSD    #04H
```

3.3 Workarounds

To avoid this problem, use any of the following ways:

(1) Within the function that has called the string-handling function

involved, reference the address of the member that has been read or write in Condition (7)
Example modified:
Add the following line anywhere within the function:
&u0.b;
(2) Qualify the local variable in Condition (3) to be volatile.
(3) Use -library=function in compilation.
(4) Use -optimize=0 or -optimize=1 in compilation.

4. Problem with Using the Value of an Array-Type Member of an Array-Type Structure or Union for Dynamic Initialization (RXC#014)

4.1 Description

In dynamic initialization, if the initializing expression (the right term) reads the value of an array-type member of an array-type structure or union whose subscript is an expression containing a variable, the read may be made from an incorrect address.

4.2 Conditions

This problem arises if the following conditions are all satisfied:
(1) An array-type structure or union has array-type members.
(2) In an array-type structure or union and its array-type members in (1), the two arrays involved are different from each other in the number of elements.
(3) Dynamic initialization is performed. The expression contains a variable n and an operational expression that reads out the value of an array-type member (an element of the array) in (1) in any of the following methods:
(a) Referencing the array-type member that has subscript n
(b) Indirect referencing to the address of the array name of the array-type member plus the address of variable n
(c) Indirect referencing to the address of the array name of the array-type member minus the address of variable n
(4) If variable n is 0, the address of the array-type member read out in (3) is the same as the beginning address of the array-type structure or union.
(5) The array-type member in (3) is directly referenced by using a member reference operator (. or ->), not using any variable of type pointer.
(6) Variable n is not 0 during compilation of code.

When the value of the array-type member is read in the operational

expression in Condition (3), the address of the array-type member cannot correctly be calculated, the read may be made from an incorrect address if the above conditions are satisfied.

Example 1. Condition (3)-(a) satisfied:

```
union {
   short a[3];          /* Conditions (1) and (2) */
} s1[2];               /* Conditions (1) and (2) */
int func01(int n)
{
   int ret = s1[0].a[n];  /* Conditions (3)-(a), (4), and (5) */
   return ret;
}
```

Example 2. Condition (3)-(b) satisfied:

```
struct {
   short a[3];          /* Conditions (1) and (2) */
} s2[2];               /* Conditions (1) and (2) */
int ans02;
void func02(int n)
{
   int ret = *(s2->a + n);  /* Conditions (3)-(b), (4), and (5) */
   ans02 = ret + 1;
}
```

Example 3. Condition (3)-(c) satisfied:

```
struct {
   char  a,b;
   short c[3];          /* Conditions (1) and (2) */
} s3[2];               /* Conditions (1) and (2) */
int ans03;
void func03(int n)
{
   int ret = *(s3->c - 1 + n); /* Conditions (3)-(c), (4), and (5) */
   ans03 = ret + 1;
}
```

4.3 Workaround

To avoid this problem, use any of the following ways:

(1) Do not perform dynamic initialization

Example 1 modified:

```
union {
   short a[3];
} s1[2];
int func01(int n)
{
   int ret;
   ret = s1[0].a[n];     /* Not initialization but assignment */
   return ret;
}
```

(2) In an array-type structure, move the line of an array-type member so that Condition (4) is not satisfied.

Example 2 modified:

```
struct {
   int dummy;          /* Moves line of array-type member */
   short a[3];
} s2[2];
int ans02;
void func02(int n)
{
   int ret = *(s2->a + n);
   ans02 = ret + 1;
}
```

(3) Substitute a variable for the constant in the operational expression if Condition (3)-(c) is satisfied.

Example 3 modified:

```
struct {
   char  a,b;
   short c[3];
} s3[2];
int ans03;
void func03(int n)
{
   int x = -1;
   int ret = *(s3->c + x + n);  /* Variable substituted
```

```
                                    for constant */
    ans03 = ret + 1;
  }
  ------------------------------------------------------------
```

5. Problem with Using the Pointer to an Array-Type Member of a Structure or Union (RXC#015)

5.1 Description

When a structure or union (here called A) having array-type members is declared, and each element of the array as members is a structure (here called B), an incorrect address may be referenced if the array as members of the structure or union A is referenced and then a member of the structure B is referenced by using the pointer to an array-type member of the structure or union A.

5.2 Conditions

This problem arises if the following conditions are all satisfied:

(1) A structure or union (here called A) is declared, which has array-type members.

(2) The address of an array-type member of A is obtained by using any of the following (a) and (b):
   (a) The member name
   (b) The member name or the reference to the array applied an address operator (&)

(3) Any of the following expressions is used:
   (a) An offset is added to either of the expressions in (2).
   (b) Either of the expressions in (2) is used to the array which is referred by the subscript.
   (c) Either of the expressions in (2) is cast to the pointer to the structure B, and then an offset is added to it.
   (d) Either of the expressions in (2) is cast to the pointer to the structure B, and then the array is referenced.

(4) The array-type member in (2) is referenced by a member operator (. or ->) operating on any of the expressions in (3).

(5) A write or read is made by using the reference in (4).

Example 1. Conditions (2)-(a) and (3)-(a) satisfied:

```
------------------------------------------------------------
struct BS{ long a; };
struct ST{
  long a;
  struct BS st[3];
```

```
}at[1] = {1, {2, 3, 4}};          /* Condition (1) */
long x;
func(){
  x = (((struct BS*)at->st) + 1)->a;  /* Conditions (2)-(a),
                                (3)-(a), (4), and (5) */
}
```

Because an incorrect address is referenced, 2 is assigned to x where 3 would be.

Example 2. Conditions (2)-(b) and (3)-(b) satisfied:

```
struct BS{ int a; };
struct ST{
  int a;
  struct BS st[4];
}at = {1,{2, 3, 4, 5}};          /* Condition (1) */
int x;
func(){
  x = ((struct BS*)&at.st[0])[1].a;   /* Conditions (2)-(b),
                                (3)-(b), (4), and (5) */
}
```

Because an incorrect address is referenced, 4 is assigned to x where 3 would be.

Example 3. Conditions (2)-(b) and (3)-(c) satisfied:

```
struct BS{ long a,b; };
struct ST{
   long a[2],b,c,d;
}at = {{1, 2}, 3, 4, 5};        /* Condition (1) */
func(){
   ((struct BS*)&at.a + 1)->b = 9; /* Conditions (2)-(b),
                               (3)-(c), (4), and (5) */
}
```

Because a write is made to an incorrect area, 9 is written to the area where 3 has been stored though 9 would be written to the area where 4 has been stored.

5.3 Workaround

Assign the result of referencing an address to another variable of type pointer; then use this variable.

Example 1 modified:

```
---------------------------------------------------------------
struct BS{ long a; };
struct ST{
   long a;
   struct BS st[3];
}at[1] = {1, {2, 3, 4}};

long x;
func(){
   struct BS *ptr;
   ptr = (struct BS*)at->st + 1;
               /* Portion of address calculation
                     assigned to pointer; then use it */

   x = ptr->a;
}
---------------------------------------------------------------
```

Example 2 modified:

```
---------------------------------------------------------------
struct BS{ int a; };
struct ST{
   int a;
   struct BS st[4];
}at = {1,{2, 3, 4, 5}};
int x;
func(){
   struct BS *ptr = &at.st[0];
               /* Portion of address calculation
                     assigned to pointer; then use it */
   x = ptr[1].a;
}
---------------------------------------------------------------
```

Example 3 modified:

```
---------------------------------------------------------------
struct BS{ long a,b; };
struct ST{
   long a[2],b,c,d;
}at = {{1, 2}, 3, 4, 5};
func(){
   struct BS *ptr;
   ptr = (struct BS*)&at.a + 1;
               /* Portion of address calculation
```

```
                    assigned to pointer; then use it */

   ptr->b = 9;
}
------------------------------------------------------------
```

6. Schedule of Fixing the Problems

The above problems have been fixed in the V.1.01 Release 00 product, which were published on May 16, 2011.

For details of V.1.01 Release 00, see:
http://tool-support.renesas.com/eng/toolnews/110516/tn3.htm
This Web page will be opened on May 20.

[Disclaimer]
The past news contents have been based on information at the time of publication. Now changed or invalid information may be included. The URLs in the Tool News also may be subject to change or become invalid without prior notice.

© 2010-2016 Renesas Electronics Corporation. All rights reserved.