Tool News

RENESAS TOOL NEWS on March 1, 2011: 110301/tn3

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 seven problems:

- With setting the values of array-type members of a structure or union in a loop (RXC#007)
- With calculating the address of an array-type variable by casting the variable to a void-type pointer (RXC#008)
- With using the "-optimize=branch" option (optimization at linking) (LNK-008)
- With performing linking (LNK-009)
- With Calling a Function By Using a Member-Function Pointer to Multi-Dimensional Array in a C++ Language Source Program (RXC#009)
- With using the "base=ram" compiler option together with a mathematical function (RXC#010)
- With calling a function within a function to which the #pragma inline_asm directive is applied (RXC#011)

---

1. Products Concerned

The following products are concerned in all the seven problems:
- The C/C++ compiler package for the RX family V.1.00 Release 00 and V.1.00 Release 01

2. Problem with Setting the Values of Array-Type Members of a Structure or Union in a Loop (RXC#007)

2.1 Description

If the values of array-type members of a structure or union are set in a loop by incrementing or decrementing the subscripts of the members, the values may be set in an incorrect area.

2.2 Conditions

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

(1) As a compiler option, optimize=2 or optimize=max is used.
(2) The variables of a structure or union are declared to be or defined as an array of type structure or union.
(3) Two arrays of type int or float are declared to be members of the structure or union in (2). They are hereafter called the array-type members.
(4) The types of both array-type members in (3) are equal to or less than 4 bytes in size, and the type of at least one member is 2 or 4 bytes in size.
(5) In the program exists a loop that is iterated by a loop variable.
(6) In the loop in (5), each of the two array-type members in (3) appears once or more times.
(7) The accesses to the two array-type members in (6) are both made by using linear expressions in which the subscript to the array of type structure or union is a constant, and the subscripts to the two array-type members are linear expressions containing the loop variable.

Here, the forms of the linear expressions are as follows:

```
  (A loop immutable variable or constant) * (A loop variable)
       + (A loop immutable variable or constant)
or
  (A loop immutable variable or constant) * (A loop variable)
       - (A loop immutable variable or constant)
```

The loop immutable variable or constant to which a loop variable is multiplied is hereafter called a linear factor, and the one added to or subtracted from a loop variable is a constant term.

(8) In the linear expressions of the subscripts to the two array-type members in (7), their linear factors are the same.
(9) The following objects are not qualified to be volatile:
   - The array of type structure or union in (2)
   - The array-type members of the structure or union in (3)
   - The loop variable in (5)
   - The loop variable in the linear expressions in (8)

Example:

```
--------------------------------------------------------------
// ccrx -cpu=rx600 -optimize=2
struct {
  signed short ss_a[10];    // Conditions (3) and (4)
  unsigned long uc_e[15];   // Conditions (3) and (4)
. . . . . . . . . . . . . . . . . . . . . . . . . .
} st_data[2];               // Condition (2)
```

```
void main(void){
 int i;
 for(i = 0; i < 10; i++){      // Condition (5)
     st_data[0].ss_a[i] = 0;   // Conditions (6)--(8)
                               Linear factor = 1,
                               Constant term = 0
     st_data[0].uc_e[i+5] = 1; // Conditions (6)--(8)
                               Linear factor = 1,
                               Constant term = 5
 }
}
----------------------------------------------------------------
```

2.3 Workarounds

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

(1) Enclose the functions containing any of the above conditions with #pragma optimize=0 and #pragma option, or #pragma optimize=1 and #pragma option.

(2) Use optimize=0 or optimize=1 as a compiler option.

(3) Qualify any of the following to be volatile:
- The array of type structure or union in Condition (2)
- Either of the array-type members in Condition (3)
- The loop variable in the loop in Condition (5)
- The loop variable in the linear expressions in Condition (8)

(4) Make the accesses to the two array-type members in different loop statements.

Example modified:

```
-----------------------------------------------------------
for(i = 0; i < 10; i++){
 st_data[0].ss_a[i] = 0;
}
for(i = 0; i < 10; i++){
 st_data[0].uc_e[i+5] = 1;
}
-----------------------------------------------------------
```

(5) Not to fulfill Condition (7), create a dummy static variable defined within the file for the subscript to the array of type structure or union. Then replace the subscript to either of the two array-type members with the dummy static variable.

Example modified:

```
-----------------------------------------------------------
struct {
 signed short ss_a[10];
 unsigned long uc_e[15];
. . . . . . . . . . . . . . . . . . . . . . . . . . .
} st_data[2];

static char dummy = 0;      // Dummy static variable
void main(void){
  int i;
  for(i = 0; i < 10; i++){
    st_data[dummy].ss_a[i] = 0; // Subscript to an array-type
                                   member replaced with dummy.
    st_data[0].uc_e[i+5] = 1;
  }
}
--------------------------------------------------------------
```

3. Problem with Calculating the Address of an Array-Type Variable by Casting the Variable to a Void-Type Pointer (RXC#008)

3.1 Description

In the address calculation of an array-type variable, incorrect address may be obtained if the variable is casted to a void-type pointer (void *) and then used.

3.2 Conditions

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

(1) An array-type global variable is declared.
(2) The variable in (1) is casted to a pointer type whose type size is different from that of the array-type variable in (1); then the variable is referenced.
(3) The casting in (2) is made from two or more cast expressions including the one to a void-type pointer (void *).
(4) To the expression in (3), two or more arithmetic operations are applied.
(5) Conditions (2), (3), and (4) are fulfilled within one expression.

Example:

```
--------------------------------------------------------------
unsigned char index[15];  // Condition (1)

unsigned long* p;
unsigned long column;

void main(){
  int i;
  column = 0x04;
  p = (unsigned long *)(void*)index + column - 1; // Conditions
```

```
                              (2)--(5)
    }
    ----------------------------------------------------------
```

3.3 Workarounds

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

(1) Declare an array-type local variable; not global.

(2) Assign the address of the array-type variable to a temp variable; then cast the variable to a pointer type.
Examples Original and Modified:

```
--------------------------------------------------
Original:
  p = (unsigned long *)(void*)index + column - 1;

Modified:
  unsigned char *temp = index;
  p = (unsigned long *)(void*)temp + column - 1;
---------------------------------------------------
```

(3) From the cast expressions, remove the cast expression to the void-type pointer (void *).
Examples Original and Modified:

```
---------------------------------------------------
Original:
p = (unsigned long *)(void*)index + column - 1;

Modified:
p = (unsigned long *)index + column - 1;
---------------------------------------------------
```

(4) Split the expression of arithmetic operations, which include the cast expressions, into two or more.
Examples Original and Modified:

```
--------------------------------------------------
Original:
p = (unsigned long *)(void*)index + column - 1;

Modified:
p = (unsigned long *)(void*)index + column;
p = p - 1;
--------------------------------------------------
```

4. Problem with Using the "-optimize=branch" Option (Optimization at Linking) (LNK-008)

4.1 Description

If a function is called immediately after if, else if, and else statements, the function call may not be made as a result of optimization by the -optimize=branch option.

4.2 Conditions

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

(1) A C/C++ source program is compiled by using the -cpu=rx600 option to generate the object file; then the object file is inputted into the linker.
(2) The -goptimize option is used
(3) The -optimize=0 option is not used to generate the object file in (1).
(4) The -optimize=branch option is effective at linking.
(5) In the source program in (1), a function is defined, the decision of the conditional branch in if, else if, and else statements is made, and return statements exist in any sections.
(6) A function call is made immediately after any of the return statements in (5)

Example:

```
--------------------------------------------------------------
//ccrx -cpu=rx600 -goptimize tp.c // Conditions (1)--(3)
//optlnk -start=P,B_1/0 -optimize=branch tp.obj // Condition (4)
//tp.c
unsigned char b,d;
void ok(void){}
void func(void)
{
  if ( !b && d ) {
     b=0;
     return;  // Condition (5)
   }
   ok() ;     // Condition (6)
}
--------------------------------------------------------------
```

4.3 Workarounds

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

(1) Do not use -goptimize at compilation.
(2) Use -optimize=0 at compilation.
(3) Use -nooptimize at linking.
(4) Use the disables optimization for the specified section option -section_forbid for the sections containing any of the C source involved at linking.

(5) After the function call in Condition (6), insert an expression
that is not removed by compilation.

5. Problem with Performing Linking (LNK-009)

5.1 Description

In an MCU of the RX family, the result of calculation of relocation
may incorrectly be evaluated when the relocation size is overflowing.
So, no error message is displayed at linking.

5.2 Conditions

This problem arises if the following conditions are all satisfied:

(1) The -cpu=rx600 option is used.

(2) Some instructions uses a addressing mode with displacement, and the
displacement contained in the address sections of instructions
is any one of the following:

disp:16[Rs].W
disp:16[Rs].UW
disp:16[Rs].L
disp:8[Rs].W
disp:8[Rs].UW
disp:8[Rs].L
disp:16[Rs]
disp:8[Rs]

(3) The displacements and the ranges of their values are as follows:

Displacement	Range of value	Size of instruction
disp:16[Rs].W	0x20000--0x3FFFF	-----
disp:16[Rs].UW	0x20000--0x3FFFF	-----
disp:16[Rs].L	0x40000--0xFFFFF	-----
disp:8[Rs].W	0x200--0x3FFFF	-----
disp:8[Rs].UW	0x200--0x3FFFF	-----
disp:8[Rs].L	0x400--0xFFFFF	-----
disp:16[Rs]	0x20000--0x3FFFF	Word
disp:16[Rs]	0x40000--0x3FFFF	Long
disp:8[Rs]	0x200--0x3FFFF	Word
disp:8[Rs]	0x400--0xFFFFF	Long

6. With Calling a Function By Using a Member-Function Pointer to Multi-Dimensional Array in a C++ Language Source Program (RXC#009)

6.1 Description

If a function is called by using a member-function pointer to multi-dimensional array, the C4099 internal error may arise.

6.2 Conditions

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

(1) A C++ language source file is compiled, or it is compiled by using the -lang=cpp option.

(2) A member-function pointer to multi-dimensional array is declared.

(3) A function is called by using the pointer in (2).

Example 1:

```
-------------------------------------------------------------
#include <stdio.h>
class TEST {
  public:
    void (TEST::*ActionFunc[2][2])(); // Condition (2)
    void print(){ printf("OK¥n"); }
  static void print2();
    void func(){
        this->ActionFunc[0][1] = print;
       (this->*ActionFunc[0][1])(); // Condition (3)
    }
};

TEST test;

void func1(){
    test.func();   // Member function is called.
}
-------------------------------------------------------------
```

Example 2:

```
-------------------------------------------------------------
#include <stdio.h>
class TEST {
  public:
    void print(){ printf("OK¥n"); }
  static void print2();
};

TEST test;

void func2(){
 void (TEST::*ActionFunc[2][2])(); // Condition (2)
 (test.*ActionFunc[0][1])(); // Condition (3)
 }
------------------------------------------------------------
```

6.3 Workarounds

To avoid this problem, use either of the following methods:

(1) Declare the member-function pointer in (2) to a single-dimensional array instead of a multi-dimensional array.

(2) Assign the member-function pointer to multi-dimensional array to a temporary member-function pointer; then call a function by using the temporary member-function pointer.

Example 1 modified by Workaround (1):

```
------------------------------------------------------------
#include <stdio.h>
class TEST {
 public:
   void (TEST::*ActionFunc[2][2])();
   void (TEST::**ActionFunc2[2])();  // Replaced with single-
                                     dimensional array
   void print(){ printf("OK¥n"); }
 static void print2();
   void func(){
      this->ActionFunc2[0][1] = print;  // Uses single-
                                        dimensional array
      (this->*ActionFunc2[0][1])();
   }
};

TEST test;

void func1(){
   test.func();
}
-------------------------------------------------------------------
```

Example 2 modified by Workaround (2):

```
-------------------------------------------------------------------
#include <stdio.h>
```

```
class TEST {
 public:
    void print(){ printf("OK¥n"); }
 static void print2();
};

TEST test;

void func2(){
 void (TEST::*fptr)() = test.ActionFunc[0][0];
 (test.*fptr)();    // Function call by using temporary
                       member-function pointer
}
-----------------------------------------------------------------
```

7. Problem with Using the "base=ram" Compiler Option together with a Mathematical Function (RXC#010)

7.1 Description

If the base=ram compiler option and a mathematical function are used together, the L2330(E) Relocation size overflow error may arise when the object file is linked with the library.
However, if the top of the RAM section (__RAM_TOP) is placed near the C section, the error message may not be displayed.

7.2 Conditions

This problem may arise if the following conditions are all satisfied:
(1) The -base=ram compiler option is used.
(2) Any of the following mathematical functions is used:
In the math.h or mathf.h file
- atan2f, ceilf, expf, floorf, fmodf, powf, tanhf, cosf, coshf, sinf, and sinhf (c89 standards)
- erfcf, expm1f, fmaf, lgammaf, and tgammaf (c99 standards)

In the complex.h file
- cacosf, cacoshf, cargf, casinf, casinhf, catanf, catanhf, ccosf, ccoshf, cexpf, clogf, clog10f, cpowf, csinf, csinhf, csqrtf, ctanf, and ctanhf (c99 standards)

In complex (EC++)
- arg, polar, cos, cosh, exp, log, log10, pow, sin, sinh, sqrt, tan, and tanh

(3) The relative distance between the top of the RAM section and the CONST section is greater than the displacement value used in the MOV instruction.

Example:

```
--------------------------------------------------------------
<x.c>
#include <mathf.h>
float x,y,z;
void main(void){
  z = powf(x, y);  // Mathematical function is used.
}
--------------------------------------------------------------
```

Execute the following three commands:

```
  ccrx -base=ram=r13 x.c
  lbgrx -head=runtime,mathf -base=ram=r13
  optlnk -start=B/0,C,P/ffe0000 x.obj -lib=stdlib.lib
```

Then the following error message is displayed:

```
 ** L2330 (E) Relocation size overflow : "_in_pows"-"P"-"00000083"
```

7.3 Workaround

Do not use -base=ram.

8. With calling a function within a function to which the #pragma inline_asm directive is applied (RXC#011)

8.1 Description

If in an assembly language inline function exists a variable or function that is not used anywhere except in the assembly language inline function to which the #pragma inline_asm is applied, the A4098 internal error may arise.

8.2 Conditions

This problem arises if the following conditions are all satisfied:

(1) In a C source file, the #pragma inline_asm directive is applied to an assembly language inline function, and the C source file is compiled to generate the object file by using the ccrx compile driver.

(2) In the assembly language inline function exists a variable or function that is not used anywhere except in the assembly language inline function.
In this Condition, any variable or function that has been declared to be extern, but not used anywhere is included.
Example:

```
--------------------------------------------------------------
#pragma inline_asm f1
static void f1(void);
extern void f2(void);   // Declared to be extern, but not used.
void main(void);
void main(void){
 f1();
}

static void f1(void){
  .GLB    _f2
  MOV.L   #1, R1
  MOV.L   #_f2 , R2   ; _f2 used only in inline_asm function.
  JMP     R2
}
------------------------------------------------------------
```

8.3 Workarounds

To avoid this problem, use either of the following methods:

(1) Add a dummy process that uses any variable or function involved in Condition (2).

(2) Generate the assembly source program by using the -output=src option, and then generate the object file.

9. Schedule of Fixing the Problems

The above seven problems have been resolved in the V.1.00 Release 02 product. For details of V.1.00 Release 02, see RENESAS TOOL NEWS Document No. 110301/tn4. This item of news is also accessible on and after March 14 at:

http://tool-support.renesas.com/eng/toolnews/110301/tn4.htm