RENESAS Tool News

RENESAS TOOL NEWS on February 01, 2016: 160201/tn1

Notes on Using the C Compilers for the RL78 and 78K0R Families of MCUs (CA78K0R) and for the 78K0R Family of MCUs (CC78K0R)

When using the C Compilers for the RL78 and 78K0R Families of MCUs (CA78K0R) and for the 78K0R Family of MCUs (CC78K0R), take note of the problems described in this note regarding the following points.

1. Binary Operators of Identifiers which are Qualified as volatile
2. Pre-Incrementing, Pre-Decrementing, and Post-Decrementing Array Elements
3. Sequential Access when Array Elements and Structure or Union Members are Cast from the near to the far Attribute
4. Problem with the strcmp and strncmp Functions Returning Incorrect Values
5. Problem with the strtoul Function Returning Incorrect Values

---

1. Binary Operators of Identifiers which are Qualified as volatile

1.1 Applicable product and versions

CA78K0R V1.70 to V1.71
(included in the CS+ integrated development environment)
CA78K0R V1.20 to V1.70
(included in the CubeSuite+ integrated development environment)
CA78K0R V1.00 to V1.10
(included in the CubeSuite integrated development environment)
CC78K0R V1.00 to V2.13
(included in the PM+ integrated development environment)

1.2 Descriptions

If the result of a binary operation on an identifier qualified as volatile is substituted to another identifier, it may produce an error or incorrect assembler code.

1.3 Conditions

This problem may arise if the following conditions are all met.
(1) The assignment expression is any of the following.

```
a = b  c;
a *= b  c;
a += b  c;
a -= b  c;
a &= b  c;
a ^= b  c;
a |= b  c;
```

(2) The binary operator of (1) is any of *, +, -, <<, &, ^, or |.

(3) The operand a to the left of an assignment operator under (1) is an identifier of the char, signed char, or unsigned char type.

(4) The operands b and c under (1) are identifiers qualified as volatile and of a different type* from the left operand a of (1).

*: short, unsigned short, int, or unsigned int type

Example:

```
------------------------------------------------------------------
volatile unsigned short vus1, vus2;  /* Condition(4) */
                                     /* Identifiers qualified as */
                                     /* volatile */
unsigned char uc1;  /* Condition(3), identifier of unsigned char type */
void func(void)
{
   uc1 = (vus2 - vus1);            /* Conditions (1) and (2) */
                          /*Substitution with binary operator "-" */
}
------------------------------------------------------------------
```

Example of assembler code output for the above example of the conditions:

```
------------------------------------------------------------------
; line     5 :     uc1 = (vus2 - vus1);
movw    ax,!_vus2   ; Loads lower-order byte of vus2 to x register.
movw    ax,!_vus1   ; Loads lower-order byte of vus1 to x register, and
                    ; overwrites the register's value.
mov     a,x
sub     a,x
mov     !_uc1,a
------------------------------------------------------------------
```

1.4 Workaround

To avoid this problem, do any of the following:

(1) Eliminate the volatile qualification of operand b and c in condition (1).

Example of workaround 1:

```
---------------------------------------------------------------
unsigned short vus1, vus2;   /* Eliminate the volatile qualification. */
unsigned char uc1;
void func(void)
{
   uc1 = (vus2 - vus1);
}
---------------------------------------------------------------
```

(2) Cast "b c" to the same type as a.

Example of workaround 2:

```
---------------------------------------------------------------
volatile unsigned short vus1, vus2;
unsigned char uc1;
void func(void)
{
   uc1 = (unsigned char)(vus2 - vus1);   /* Cast to the type of uc1. */
}
---------------------------------------------------------------
```

2. Pre-Incrementing, Pre-Decrementing, and Post-Decrementing Array Elements

2.1 Applicable product and versions

CA78K0R V1.70 to V1.71
 (included in the CS+ integrated development environment)
CA78K0R V1.20 to V1.70
 (included in the CubeSuite+ integrated development environment)
CA78K0R V1.00 to V1.10
 (included in the CubeSuite integrated development environment)*
CC78K0R V2.00 to V2.13
 (included in the PM+ integrated development environment)*
 *: V1.00 to V1.01 of CA78K0R and V1.00 to V2.12 of CC78K0R are not within the scope of the note with respect to post-decrementing.

2.2 Description

Assembler code produced in response to pre-incrementing, pre-decrementing, or post-decrementing of array elements might be incorrect.

2.3 Conditions

This problem may arise if either set of conditions 1 or 2 below is met.

Conditions 1

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

(1) Automatic variables or formal arguments are not used in the

function.
(2) An expression includes two or more indirect references.
We refer to the two indirect references as "indirect reference A" and "indirect reference B".
(3) Indirect reference A is to an array element whose index is not a constant (e.g.: ary[idx]) or an indirect reference which uses a pointer (e.g.: *ptr).
(4) Indirect reference B is to an array element, the index is not a constant, the size of the elements is 1 byte, and the array was qualified as far.
(5) The array element in indirect reference B is pre-incremented or pre-decremented.

Conditions 2
This problem may arise if the following conditions are all met.
(1) Reference to an array element
(2) The size of the elements of the array is 1 or 2 bytes.
(3) The index of the array is of the unsigned short or unsigned int type.
(4) The index of the array is post-decremented from the value 0.

Example of conditions 1:

```
----------------------------------------------------------------
unsigned char __far fuca1[2];
unsigned char ucaa1[2][2];
unsigned char x1;
unsigned short us1, us2;
void func(void)
{
 /* Condition (1) Automatic variables and formal arguments were not */
 /* used. */
 /* Condition (2) Indirect reference A: ucaa1[us1], */
 /* indirect reference B: fuca1[us2] */
 /* Condition (3) An array element with a non-constant index: */
 /*  ucaa1[us1] */
 /* Condition (4) An array element with a non-constant index for a */
 /* far array with 1-byte elements: fuca1[2] */
 /* Condition (5) Pre-incrementing: ++fuca1[us2] */
   x1 = ucaa1[us1][++fuca1[us2]];
}
----------------------------------------------------------------
```

Example of assembler code output for the above example of the conditions:

```
----------------------------------------------------------------
```

```
; line    13 :     x1 = ucaa1[us1][++fuca1[us2]];
movw    ax,!_us1
addw    ax,ax
addw    ax,#loww (_ucaa1)
movw    de,ax
movw    ax,!_us2
addw    ax,#loww (_fuca1)
movw    hl,ax
mov     ES,#highw (_fuca1)
inc     ES:[hl+0]
mov     ES,_@SEGHL  ; Reference to _@SEGHL, which has not been set
mov     a,ES:[hl]
shrw    ax,8
addw    ax,de
movw    de,ax
mov     a,[de]
mov     !_x1,a
-----------------------------------------------------------------
```

Example of conditions 2:

```
-----------------------------------------------------------------
unsigned char uca1[10], uc1;
unsigned short us1;
void func(void)
{
 /* Condition (1) Reference to an array element: uc1 = uca1[us1--] */
 /* Condition (2) The size of the elements of the array is 1 byte: */
 /* uca1[10] */
 /* Condition (3) The array index is of the unsigned short type: us1 */
 /* Condition (4) The array index is post-decremented from 0: us1-- */
   uc1 = uca1[us1--];
}
-----------------------------------------------------------------
```

Example of assembler code output for the above example of the conditions:

```
-----------------------------------------------------------------
; line    10 :     uc1 = uca1[us1--];
decw    !_us1           ; When us1 is 0
movw    bc,!_us1        ; The value of register bc becomes 0xffff
mov     a,_uca1+1[bc]   ; Substitution by uca1[1+0xffff]
mov     !_uc1,a
-----------------------------------------------------------------
```

2.4 Workaround

The workarounds depend on the applicable conditions.

In case of conditions 1
To avoid this problem, do any of the following.
(1) Change the far array to a near array.
(2) Set temporary variables, and operate with substitution of the results of pre-incrementing and pre-decrementing to temporary variables.

Example a workaround for conditions 1

```
------------------------------------------------------------
unsigned char __far fuca1[2];
unsigned char ucaa1[2][2];
unsigned char x1;
unsigned short us1, us2;
void func(void)
{
   unsigned char temp;      /* Temporary variable */
   temp = ++fuca1[us2];     /* Substitute the result of */
                            /* pre-incrementing. */
   x1 = ucaa1[us1][temp];
}
------------------------------------------------------------
```

In the case of conditions 2:
Separate the array reference from decrementing of the index.

Example of a workaround for conditions 2

```
------------------------------------------------------------
unsigned char uca1[10], uc1;
unsigned short us1;
void func(void)
{
   uc1 = uca1[us1];         /* Reference to an array element */
   us1--;                   /* Decrementing */
}
------------------------------------------------------------
```

3. Sequential Access when Array Elements and Structure or Union Members are Cast from the near to the far Attribute

3.1 Applicable product and versions

CA78K0R V1.70 to V1.71
(included in the CS+ integrated development environment)
CA78K0R V1.20 to V1.70
(included in the CubeSuite+ integrated development environment)

CA78K0R V1.00 to V1.10
(included in the CubeSuite integrated development environment)
CC78K0R V2.00 to V2.13
(included in the PM+ integrated development environment)

3.2 Description

Sequential access cast as far to an array element or structure or union member declared with the near attribute might lead to incorrect assembler code.

3.3 Conditions

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

(1) Automatic variables or formal arguments are used in the function.
(2) Access to an array element or structure or union member statically declared with the near attribute
(3) After the access in (2), the address is cast to a far pointer to a 1-byte array element or structure or union member at "address in (2) +1" in the assembler code, and the value is loaded by an indirect reference (not including cases where a function call or an assembler statement follows the access in (2)).
(4) After the access in (3), access to the array element or structure or union member at "address in (3) + offset" proceeds (not including cases where a function call or an assembler statement follows the access in (3)).
The offset in the assembler code has a value from -2 to +255.
(5) The array elements, structure members, or union members in (2), (3), and (4) share the same identifier.

Example

```
------------------------------------------------------------------
unsigned char __near const uca1[7] = { 1, 2, 3, 4, 5, 6, 7 };
unsigned char __near uc1, uc2, uc3;

#define data0   (*(unsigned char __far *)&uca1[0])
#define data1   (*(unsigned char __far *)&uca1[1])
#define data2   (*(unsigned char __far *)&uca1[2])
#define data3   (*(unsigned char __far *)&uca1[3])
#define data4   (*(unsigned char __far *)&uca1[4])
#define data5   (*(unsigned char __far *)&uca1[5])
#define data6   (*(unsigned char __far *)&uca1[6])

void func(void)
{
   unsigned char dummy;  /* Condition (1) */
```

```
    uc1 = data2;  /* Conditions (2) & (5) */
                  /* Access to uca1[2] with an indirect reference */
                  /* to a far address */
    uc2 = data3;  /* Conditions (3) & (5) */
                  /* Access to uca1[2+1] with an indirect reference */
                  /* to a far address */
    uc3 = data4;  /* Conditions (4) & (5) */
                  /* Access to uca1[3+1] with an indirect reference */
                  /* to a far address */
}
```

Example of assembler code output for the above example of the conditions:

```
; line 14 :     uc1 = data2;
movw    de,#mirlw (_uca1+2)
mov     a,[de]
mov     !_uc1,a
; line    15 :     uc2 = data3;
mov     a,[de+1]
mov     !_uc2,a
; line    16 :     uc3 = data4;
mov     a,[de+1]    ; Access is to uca1[3] instead of uca1[4].
mov     !_uc3,a
```

3.4 Workarounds

To avoid this problem, do any of the following:

(1) Change the access in condition (3) to have a cast to a near pointer.

Example 1:

```
#define data3    (*(unsigned char __near *)&uca1[3])
```

---

(2) Eliminate the cast from the access in condition (3).

Example 2:

```
#define data3    (uca1[3])
```

---

4. Problem with the strcmp and strncmp Functions Returning Incorrect Values

4.1 Applicable product and versions

CA78K0R V1.70 to V1.71
(included in the CS+ integrated development environment)
CA78K0R V1.20 to V1.70
(included in the CubeSuite+ integrated development environment)
CA78K0R V1.00 to V1.10
(included in the CubeSuite integrated development environment)
CC78K0R V1.00 to V2.13
(included in the PM+ integrated development environment)

4.2 Description

Comparison of arguments by the strcmp and strncmp functions may produce incorrect return values.

4.3 Conditions

This problem arises if the following conditions are all met.

(1) Arguments are compared by either of the following functions.
- strcmp(s1, s2)
- strncmp(s1, s2, n)

(2) The first different pair of characters met in byte-by-byte comparison of the character codes pointed to in s1 and s2 satisfies either of the following sub-conditions.

(a) The character code currently pointed to in the character string s1 is 0x80 or greater, and the difference between this and the character code pointed to in the character string s2 is 0x80 or greater.

(b) The character code currently pointed to in the character string s2 is 0x80 or greater, and the difference between this and the character code pointed to in the character string s1 is greater than 0x80.

Example

```
------------------------------------------------------------
#include
int x1, x2, x3;
void func(void)
{
   x1 = strcmp("¥xc0", "¥x3e"); /* Conditions (1) & (2) */
                          /* The value of x1 becomes negative. */
   x2 = strcmp("¥xc0", "¥x40"); /* Conditions (1) & (2) */
                          /* The value of x2 becomes negative. */
   x3 = strcmp("¥x40", "¥xc2"); /* Conditions (1) & (3) */
                          /* The value of x3 becomes positive. */
}
```

4.4 Workaround

There is currently no way to prevent this problem.

5. Problem with the strtoul Function Returning Incorrect Values

5.1 Applicable product and versions

CA78K0R V1.70 to V1.71
(included in the CS+ integrated development environment)
CA78K0R V1.20 to V1.70
(included in the CubeSuite+ integrated development environment)
CA78K0R V1.00 to V1.10
(included in the CubeSuite integrated development environment)
CC78K0R V1.00 to V2.13
(included in the PM+ integrated development environment)

5.2 Description

Conversion of character strings to integers by the strtoul function may produce an incorrect return value or errno.

5.3 Conditions

This problem arises if the following conditions are both met:

(1) The following function is used to convert a character string to an integer.
- strtoul(nptr, endptr, base)

(2) nptr, the character string to be subject to conversion, includes a - (minus) sign.

Example

```
---------------------------------------------------------------
#include
char *endptr;
unsigned long ans1, ans2;
void func(void)
{
  /* Conditions (1) & (2) The value of ans1 becomes 0 instead of */
  /*  4294966062. */
   ans1 = strtoul("-1234", &endptr, 10);
  /* Conditions (1) & (2) The value of ans2 becomes 0 instead */
  /* of ULONG_MAX and the value of errno becomes 0 instead of ERANGE.*/
   ans2 = strtoul("-4294967300", &endptr, 10);
}
---------------------------------------------------------------
```

5.4 Workaround

There is currently no way to prevent this problem.

6. Schedule for Fixing the Problem

The five problems described above will be fixed in the next version.
