RENESAS ツールニュース

RENESAS TOOL NEWS 2011年03月01日：110301/tn3

RXファミリ用C/C++コンパイラパッケージ ご使用上のお願い

RXファミリ用C/C++コンパイラパッケージの使用上の注意事項 7件を連絡します。

- ループ内で構造体および共用体の配列型メンバの値を設定する際の注意事項 (RXC#007)
- Void型ポインタへの配列型変数のキャストを使用したアドレス計算に関する注意事項 (RXC#008)
- -optimize=branchオプション (リンク時の最適化) に関する注意事項 (LNK-008)
- リンク時の注意事項 (LNK-009)
- C++言語ソースで多次元配列へのメンバ関数ポインタで関数呼び出しをする場合の注意事項 (RXC#009)
- base=ramコンパイラオプションと数学関数を使用する場合の注意事項 (RXC#010)
- #pragma inline_asmを適用した関数内で関数呼び出しする場合の注意事項 (RXC#011)

---

1. 該当製品

7件すべての問題に以下の製品が該当します。
RXファミリ用C/C++コンパイラパッケージ V.1.00 Release 00 および
V.1.00 Release 01

2. ループ内で構造体および共用体の配列型メンバの値を設定する際の注意事項 (RXC#007)

2.1 内容

ループ内で、構造体および共用体の配列型メンバの値を、そのメンバの添え字を更新して設定している場合、誤った領域に値を設定することがあります。

2.2 発生条件

以下の条件をすべて満たす場合に、発生することがあります。
(1) optimize=2 または optimize=maxをコンパイラオプションを使用している。
(2) 構造体または共用体の変数が、構造体または共用体型の配列として宣言または定義されている。
(3) 2つの整数型または浮動小数点型配列が(2)の構造体または共用体メンバとして宣言されている。
(4) (3)の配列メンバはいずれも型サイズが4バイト以下であり、少なくとも一方は2バイトまたは4バイトである。
(5) ループ帰納変数によって繰り返し制御されたループが存在する。
(6) (5)のループ内に、(3)の2つの配列メンバがそれぞれ1回以上出現する。

(7) (6)の配列メンバへのアクセスは、いずれも構造体配列または共用体配列の添え字が定数で
かつ配列メンバの添え字がループ帰納変数の1次式である。(注)

注: 1次式は以下の形式
(ループ不変変数または定数) * (ループ帰納変数) + (ループ不変変数または定数)
もしくは
(ループ不変変数または定数) * (ループ帰納変数) - (ループ不変変数または定数)

以降、ループ変数が乗じられているループ不変変数または定数は
1次係数、ループ変数に加算または減算しているループ不変変数または定数は定数項と呼ぶ。

(8) (7)の2つの配列メンバのそれぞれの添え字の1次式において、1次係数が同じである。
(9) 以下はいずれもvolatile修飾されていない。
- (2)の構造体配列または共用体配列
- (3)の配列メンバ
- 発生条件(5)のループのループ制御変数
- 発生条件(8)の一次式のループ帰納変数

発生例：

```
-----------------------------------------------------------------
// ccrx -cpu=rx600 -optimize=2
struct {
  signed short ss_a[10];    // 発生条件(3)および(4)
  unsigned long uc_e[15];   // 発生条件(3)および(4)
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
} st_data[2];               // 発生条件(2)

void main(void){
  int i;
  for(i = 0; i < 10; i++){        // 発生条件(5)
     st_data[0].ss_a[i] = 0;     // 発生条件(6)、(7)および(8)
                                 1次係数＝1、定数項＝0
     st_data[0].uc_e[i+5] = 1; // 発生条件(6)、(7)および(8)
                               1次係数＝1、定数項＝5
  }
}
-----------------------------------------------------------------
```

2.3 回避策

以下のいずれかの方法で回避してください。

(1) #pragma optimize=0 と#pragma option、または#pragma optimize=1 と
#pragma optionで発生条件のいくつかを含む関数をはさんで記述する。
(2) optimize=0 または optimize=1をコンパイラオプションに使用する。
(3) 以下のいずれかをvolatile修飾する。
- 発生条件(2)の構造体配列または共用体配列
- 発生条件(3)の配列メンバのいずれか一方
- 発生条件(5)のループのループ制御変数
- 発生条件(8)の一次式のループ帰納変数
(4) 発生条件に該当する2つの配列メンバへのアクセスをそれぞれ別の
ループ文に分ける。
発生例の場合：

```
------------------------------------------------------------
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) 発生条件(7)を満たさないように、構造体配列または共用体配列の添え字用にダミーの
ファイル内static変数を用意して、発生条件に該当する2つの配列
メンバのいずれか一方の添え字を置き換える。
発生例の場合：

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

static char dummy = 0;      // ダミーのstatic変数
void main(void){
  int i;
  for(i = 0; i < 10; i++){
     st_data[dummy].ss_a[i] = 0; // 配列メンバの添え字を置き換え
     st_data[0].uc_e[i+5] = 1;
  }
}
------------------------------------------------------------
```

3. Void型ポインタへの配列型変数のキャストを使用したアドレス計算に関する注意事項 (RXC#008)

3.1 内容

配列型変数の要素のアドレス計算で、void型ポインタ (void *) にその変数を

キャストして使用した場合、間違ったアドレスになることがあります。

3.2 発生条件

以下の条件をすべて満たす場合に、発生することがあります。
(1) 配列型の大域変数を宣言している。
(2) (1)の変数を、配列型のサイズと異なる型のポインタ型へキャストし、
参照している。
(3) (2)のキャストは、複数のキャスト式から成り、void型ポインタ (void *)
へのキャスト式を含んでいる。
(4) (3)の式に対して複数回の演算をしている。
(5) (2)(3)(4)を1式で行っている。

発生例：

```
----------------------------------------------------------------------
unsigned char index[15];  // 発生条件(1)

unsigned long* p;
unsigned long column;

void main(){
  int i;
  column = 0x04;
  p = (unsigned long *)(void*)index + column - 1; // 発生条件(2)、(3)、
                                        (4)および(5)
}
----------------------------------------------------------------------
```

3.3 回避策

以下のいずれかの方法で回避してください。
(1) 配列型の局所変数で宣言する。
(2) 配列のアドレスを、temp変数に代入してからポインタ型へキャストする。
発生例の場合：

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

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

(3) キャスト内のvoid型ポインタ (void *) へのキャスト式を記述しない。
発生例の場合：

```
----------------------------------------------------------
変更前：
```

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

変更後：

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

---

(4) キャストを含む演算式を複数式に分ける。

発生例の場合：

変更前：

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

変更後：

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

---

4. -optimize=branchオプション (リンク時の最適化) に関する注意事項 (LNK-008)

4.1 内容

if, else if, else文直後に関数呼び出し式がある場合、-optimize=branchの
最適化により関数が呼び出されないことがあります。

4.2 発生条件

以下の条件をすべて満たす場合に、発生することがあります。

(1) -cpu=rx600オプションを使用して、C/C++ソースプログラムから生成した
　オブジェクトファイルをリンカに入力している。
(2) (1) のオブジェクトファイルを生成時に -goptimizeを使用している。
(3) (1)のオブジェクトファイルを生成時に -optimize=0を使用していない。
(4) リンク時に -optimize=branch を有効にしている。
(5) (1)のソースプログラム内に関数定義があり、if, else if, else文の条件分岐
　判定があり、1つ以上の節でreturn文がある。
(6) (5)を満たす文の直後に関数コールがある。

発生例：

```
//ccrx -cpu=rx600 -goptimize tp.c // 発生条件(1)、(2)および(3)
//optlnk -start=P,B_1/0 -optimize=branch tp.obj // 発生条件(4)
//tp.c
unsigned char b,d;
void ok(void){}
void func(void)
{
  if ( !b && d ) {
    b=0;
```

```
    return;  // 発生条件(5)
  }
  ok() ;    // 発生条件(6)
}
-------------------------------------------------------------
```

4.3 回避策

以下のいずれかの方法で回避してください。

(1) コンパイル時に-goptimizeを使用しない。
(2) コンパイル時に-optimize=0を使用する。
(3) リンク時に-nooptimizeを使用する。
(4) リンク時に該当するコードを含むセクションに対して-section_forbid
　最適化部分抑止オプション を指定する。
(5) 発生条件(6)を満たす関数コール以降にコンパイル時に削除されない1式を
　追加する。

5. リンク時の注意事項 (LNK-009)

5.1 内容

RXマイコンのリロケーション演算結果のチェックでリロケーションサイズを超えたことを正しく判定できずに、リンク時にエラーを出力しないことがあります。

5.2 発生条件

以下の条件をすべて満たす場合に発生します。

(1) -cpu=rx600オプションを使用している。
(2) ディスプレースメント付きアドレッシングモードを指定した命令があり、
　そのディスプレースメントは以下のいずれかである。
　　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) ディスプレースメント値が以下の範囲である。
　　disp:16[Rs].W のとき、0x20000～0x3FFFF
　　disp:16[Rs].UW のとき、0x20000～0x3FFFF
　　disp:16[Rs].L のとき、0x40000～0xFFFFF
　　disp:8[Rs].W のとき、0x200 ～0x3FFFF
　　disp:8[Rs].UW のとき、0x200 ～0x3FFFF
　　disp:8[Rs].L のとき、0x400 ～0xFFFFF
　　disp:16[Rs]かつ命令サイズがワードのとき 0x20000～0x3FFFF
　　disp:16[Rs]かつ命令サイズがロングのとき 0x40000～0x3FFFF
　　disp:8[Rs]かつ命令サイズがワードのとき 0x200～0x3FFFF

disp:8[Rs]かつ命令サイズがロングのとき　0x400～0xFFFFF

6. C++言語ソースで多次元配列へのメンバ関数ポインタで関数呼び出しをする場合の注意事項(RXC#009)

6.1 内容

C++言語ソースで多次元配列へのメンバ関数ポインタで関数呼び出しをすると、C4099内部エラーが発生する場合があります。

6.2 発生条件

以下の条件をすべて満たす場合に発生することがあります。

(1) C++言語ソースをコンパイルしている。もしくは、-lang=cpp オプションを使用してコンパイルしている。

(2) 多次元配列へのメンバ関数ポインタを宣言している。

(3) (2)のポインタを使用して関数を呼び出している。

発生例1：

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

TEST test;

void func1(){
   test.func();   // メンバ関数呼び出し
}
-------------------------------------------------------------
```

発生例2:

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

```
TEST test;

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

6.3 回避策

以下のいずれかの方法で回避してください。

(1) 発生条件(2)のメンバ関数ポインタを一次元配列への宣言にする。

(2) 多次元配列へのメンバ関数ポインタを、一時的なメンバ関数ポインタに代入して、それを使用して関数を呼び出す。

回避策1による発生例1の回避：

```
-------------------------------------------------------------------
#include <stdio.h>
class TEST {
 public:
   void (TEST::*ActionFunc[2][2])();
   void (TEST::**ActionFunc2[2])();  // 一次元配列にする
   void print(){ printf("OK¥n"); }
 static void print2();
   void func(){
      this->ActionFunc2[0][1] = print;  // 一次元配列使用に変更
      (this->*ActionFunc2[0][1])();
   }
};

TEST test;

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

回避策2による発生例2の回避:

```
-------------------------------------------------------------------
#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)();    // 一時的に用意したメンバ関数ポインタによる関数
                    呼び出し
}
```

---

7. base=ramコンパイラオプションと数学関数を使用する場合の注意事項 (RXC#010)

7.1 内容

base=ramを使用し、数学関数を使用した場合、ライブラリとオブジェクトファイルのリンク時に
L2330(E) Relocation size overflow エラーが発生することがあります。
ただし、RAMセクション先頭 (__RAM_TOP) とCセクションが近くに配置されている場合、エラーが出力されない場合があります。

7.2 発生条件

以下のすべてを満たす場合、発生することがあります。
(1) -base=ramを使用している。
(2) 以下のいずれかの数学関数を使用している。
math.hまたはmathf.h :
c89規格: atan2f, ceilf, expf, floorf, fmodf, powf, tanhf,
cosf, coshf, sinfおよび sinhf
c99規格: erfcf, expm1f, fmaf, lgammafおよび tgammaf
complex.h :
c99規格: cacosf, cacoshf, cargf, casinf, casinhf, catanf,
catanhf, ccosf, ccoshf, cexpf, clogf, clog10f, cpowf,
csinf, csinhf, csqrtf, ctanfおよび ctanhf
complex (EC++) : arg, polar, cos, cosh, exp, log, log10, pow, sin,
sinh, sqrt, tanおよび tanh

(3) RAMセクション先頭とCONSTセクションとの相対値が、MOV命令のディスプレースメント値の範囲外になった場合

発生例:

```
<x.c>
#include <mathf.h>
float x,y,z;
void main(void){
 z = powf(x, y);  // 数学関数を使用している
}
```

以下のコマンドを実行する
```
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
```
以下のエラーが出力される。
```
** L2330 (E) Relocation size overflow : "_in_pows"-"P"-"00000083"
```

7.3 回避策

-base=ramを使わないでください。

8. #pragma inline_asmを適用した関数内で関数呼び出しする場合の注意事項 (RXC#011)

8.1 内容

変数または関数が、#pragma inline_asm が適用されたアセンブラ
埋め込みインライン関数内のみにあって、そのアセンブラ埋め込み
インライン関数内以外には使用されていない場合、A4098内部エラー
が発生することがあります。

8.2 発生条件

以下の条件をすべて満たす場合に発生します。
(1) ccrx コンパイルドライバで、#pragma inline_asm を使用したCソースを
コンパイルして、オブジェクトを生成している。
(2) アセンブラ埋め込みインライン関数にはその関数以外では使用されて
いない変数または関数がある。それには、extern宣言されているが、
まだ使用されていない変数または関数を含む。

発生例:

```
-------------------------------------------------------------
#pragma inline_asm f1
static void f1(void);
extern void f2(void);   // extern宣言のみで、使用されていない
void main(void);

void main(void){
 f1();
}

static void f1(void){
  .GLB    _f2
  MOV.L   #1, R1
  MOV.L   #_f2 , R2   ; _f2 はinline_asm 以外で使用されていない
  JMP     R2
}
-------------------------------------------------------------
```

8.3 回避策

以下のいずれかの方法で回避してください。

(1) 発生条件(2)に該当する変数または関数を使用するダミーの処理を追加する。

(2) -output=srcオプションでアセンブリソースに出力してからオブジェクトを生成する。

9. 恒久対策

7件の問題はすべてV.1.00 Release 02で改修済みです。

V.1.00 Release 02の詳細は、RENESAS TOOL NEWS 資料番号110301/tn4を参照ください。以下のURLでも参照できます。(3月4日から公開予定)

https://www.renesas.com/search/keyword-search.html#genre=document&q=110301tn4
