RENESAS ツールニュース

RENESAS TOOL NEWS 2011年05月16日：110516/tn2

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 4件を連絡します。

- #pragma option使用時の、1または2バイトの整数型の関数戻り値に関する 注意事項 (RXC#012)
- 共用体型のローカル変数を文字列操作関数で操作する場合の注意事項 (RXC#013)
- 配列型構造体または共用体の配列型メンバから読み出した値を動的初期化に 用いる場合の注意事項 (RXC#014)
- 構造体または共用体の配列型メンバを指すポインタを使用して配列型メンバの 配列を参照し、さらに配列の要素である別の構造体のメンバを参照する場合の 注意事項 (RXC#015)

---

1. 該当製品

4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 ～ V.1.00 Release 02

2. #pragma option使用時の、1または2バイトの整数型の関数戻り値に関する

注意事項 (RXC#012)

2.1 内容

#pragma optionを使用している場合、戻り値として1または2バイトの
整数型を返す関数が、1または2バイトで表現できない値を返す場合があります。

2.2 発生条件

以下の条件をすべて満たす場合に発生します。
(1) 1または2バイトの整数型を戻り値として返す関数がある。
(2) (1)の関数を含むコンパイル単位内に#pragma optionがあり、
その#pragma optionで最適化オプションAを指定している。
コンパイル単位とは、ソースファイルと、それにインクルードされる
ファイルを意味します。
(3) #pragma optionの前のソースコードに対して有効なオプション群のうち

最適化オプションAに変化するオプションがある。

(4) (1)の関数に対して、コマンドラインによってoptimize=2または optimize=maxが指定されているか、または#pragma optionによって optimize=2が指定されている。

発生例 sample1.c :

```
------------------------------------------------------------
int b;
unsigned short func_ushort(unsigned short a)   /* 発生条件(1) */
{
   unsigned short tmp;
   tmp = a + b + 1;
   return tmp;
}
void fun(void)
{
   . . . . . .
}
#pragma option optimize=1     /* 発生条件(2) のオプションA */
 void dummy(void) { }
------------------------------------------------------------
```

コマンドライン例:

```
------------------------------------------------------------
ccrx -output=src -optimize=2 sample1.c
/* 発生条件(4) */
/* 発生例で示した関数func_ushortに対して有効 */
------------------------------------------------------------
```

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) 関数の戻り値を、volatile修飾した整数型の変数に一旦代入して使用する。

発生例の回避例 :

```
------------------------------------------------------------
int b;
unsigned short func_ushort(unsigned short a)
{
   volatile unsigned short tmp;  /* volatile修飾した変数に代入 */
   tmp = a + b + 1;
   return tmp;
}
#pragma option optimize=1
void dummy(void) { }
------------------------------------------------------------
```

(2) 発生条件(1)の関数を#pragma optionがない他のファイルに移動する。

(3) コマンドラインもしくは#pragma optionを用いて、
発生条件(1)の関数に対してoptimize=0かoptimize=1が有効になるようにする。

3. 共用体型のローカル変数を文字列操作関数で操作する場合の注意事項 (RXC#013)

3.1 内容

共用体型のローカル変数があり、その共用体のメンバを文字列操作関数で読み書きすると、読み書きの順序を誤る場合があります。

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) コンパイルオプションoptimize=2または =maxが有効である。
注： デフォルトではoptimize=2が有効になっています。
(2) コンパイルオプションlibrary=intrinsicが有効である。
注： デフォルトではlibrary=intrinsicが有効になっています。
(3) 共用体型のローカル変数がある。
(4) 文字列操作関数memcpy, strcpyおよび strncpy のいずれかを使用した、(3)の共用体メンバのメモリ領域の読み出しまたは書き込みがある。
(5) (4)の関数は、以下を満たしている。
- memcpyの場合、第3引数 (転送サイズ) が1以上の定数である。
- strncpyの場合、第3引数 (転送サイズ) が1以上の定数で、かつ第2引数 (コピー元) が第3引数のバイト数以上のバイト数 (NULL文字を含む) の文字列リテラルである。
- strcpyの場合、第2引数 (コピー元) が1バイト以上 (NULL文字を含む) の文字列リテラルである。
(6) 以下のいずれかを満たしている。
(a) コンパイルオプションspeedが有効である。
(b) コンパイルオプションsizeが有効であり、(5)の転送サイズが1,2または4バイトのいずれかである。
注： sizeはデフォルトでは有効になっています。
(7) (4)と同じ共用体内の領域が重なっている別のメンバに対し、読み出しまたは書き込みを行っている。
(8) (7)のメンバのアドレスが取得されていない。

memcpyの場合の発生例：

```
#include <string.h>
union U {
    char a[4];
    long b;
} u1, u2;
void main(char *p)
{
```

```
    union U u0;          /* 条件(3) */
    u1.b = u0.b;
    p+=4;
    memcpy(u0.a, p, 4);  /* 条件(4)(5)および(6) */
    u2.b = u0.b;         /* 条件(7)(8) */
}
```

問題が発生すると、条件(4)と(7)の処理順序が入れ替わります。

生成コード例:

```
_main:
   SUB      #04H,R0
   MOV.L    [R0],R4     ; 発生例の発生条件(7)にあたるu0.b の値を
                            R4に格納. . . A
   MOV.L    #_u1,R5
   MOV.L    R4,[R5]
   MOV.L    #_u2,R5
   MOV.L    04H[R1],[R0] ; 条件(4)のmemcpy
   MOV.L    R4,[R5]      ; 発生条件(7)の読み出し。
                         ; memcpy関数で読み出した値ではなくAの値を使う
   RTSD     #04H
```

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) 該当の文字列操作関数を使用している関数内で、発生条件(7)で
読み出し、または書き出したメンバのアドレスを参照する。
発生例の場合の回避例：
以下を関数内の任意の場所に追加する。
&u0.b;

(2) 発生条件(3)の変数をvolatile修飾する。

(3) コンパイル時に-library=functionを使用する。

(4) コンパイル時に-optimize=0 または =1を使用する。

4. 配列型構造体または共用体の配列型メンバから読み出した値を動的初期化に用いる場合の注意事項 (RXC#014)

4.1 内容

動的初期化に含まれる初期化式 (右辺) が、変数を含む式を添え字とする
配列型構造体または共用体の配列型メンバの読み出しをする場合、間違った
アドレスから読み出すことがあります。

4.2 発生条件

以下の条件をすべて満たす場合に発生します。

(1) 配列型メンバを持つ配列型構造体または共用体がある。

(2) (1)の配列型メンバと配列型構造体または共用体とで、互いの要素の数が異なる。

(3) 変数nを用いた動的な初期化式があり、その式は(1)の配列型メンバの要素を以下のいずれかの方法で読み出す演算式を含む。

 (a) 添え字をnとしたメンバ配列参照

 (b) 配列型メンバ名のアドレスに、変数nを加算したアドレスで間接参照

 (c) 配列型メンバ名のアドレスに定数を加減算して、変数nを加算したアドレスでポインタ間接参照

(4) 変数nが0の場合、(3)で読み出される配列型メンバの要素のアドレスが、配列型構造体または共用体の先頭アドレスと同じである。

(5) (3)の配列型メンバは、配列型構造体または共用体からメンバ演算子 (. または ->) を用い、ポインタ型変数等を経由せずに直接指し示されている。

(6) コード実行時、変数nが0ではない。

発生条件を満たすと、条件(3)の演算式で配列型メンバを読み出すとき、
配列要素のアドレスが正しく計算されず、間違ったアドレスから
読み出します。

発生例1 (発生条件(3)(a)に該当する場合) :

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

発生例2 (発生条件(3)(b)に該当する場合) :

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

```
}
-------------------------------------------------------------
```

発生例3 (発生条件(3)(c)に該当する場合) :

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

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) 動的初期化を用いない。

発生例1の回避例 :

```
   -------------------------------------------------------------
   union {
      short a[3];
   } s1[2];
   int func01(int n)
   {
      int ret;
      ret = s1[0].a[n];     /* 初期化式ではなく代入式にする */
      return ret;
   }
   -------------------------------------------------------------
```

(2) 配列型構造体の場合、発生条件(4)を満たさないように配列型メンバの位置を変更する。

発生例2の回避例 :

```
   -------------------------------------------------------------
   struct {
      int dummy;            /* メンバの位置を変更する */
      short a[3];
   } s2[2];
   int ans02;
   void func02(int n)
   {
      int ret = *(s2->a + n);
```

```
    ans02 = ret + 1;
}
-------------------------------------------------------------
```

(3) 発生条件(3)(c)に該当する場合、演算式中の定数を変数に置き換える。
発生例3の回避例：

```
-------------------------------------------------------------
struct {
    char  a,b;
    short c[3];
} s3[2];
int ans03;
void func03(int n)
{
    int x = -1;
    int ret = *(s3->c + x + n);  /* 演算式中の定数式を変数に
                                    置き換える */
    ans03 = ret + 1;
}
-------------------------------------------------------------
```

5. 構造体または共用体の配列型メンバを指すポインタを使用して配列型メンバの配列を参照し、さらに配列の要素である別の構造体のメンバを参照する場合の注意事項 (RXC#015)

5.1 内容

構造体または共用体(A)に配列型メンバがあり、その配列型メンバの配列の各要素が構造体(B)のとき、構造体または共用体(A)の配列型メンバのアドレスをポインタとして使用して(または配列型メンバを指すポインタを使用して)配列型メンバを配列参照し、さらに構造体(B)のメンバを参照すると誤ったアドレスを参照する場合があります。

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) 配列型メンバを持つ構造体(A)または共用体(A)がある。
(2) Aのメンバのアドレスを、以下のいずれかを用いて取得している。
  (a) メンバ名
  (b) メンバ名または配列参照にアドレス演算子(&)を適用
(3) 以下のいずれかの式がある。
  (a) (2)の式にオフセットを加算している。
  (b) (2)の式を配列参照に使用している。
  (c) (2)の式を構造体Bへのポインタにキャストし、オフセットを加算して

いる。
(d) (2)の式を構造体Bへのポインタにキャストし、配列参照している。
(4) (3)の式 に適用するメンバ演算子 (. または ->) によって、メンバ参照
している。
(5) (4)の式を用いて、書き込みまたは読み出しを行っている。

発生例1 (発生条件(2)(a)および (3)(a)に該当する場合) :

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

xには正しくは3が入りますが、参照アドレスを誤るため2が入ります。

発生例2 (発生条件(2)(b)および (3)(b)に該当する場合) :

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

xには正しくは3が入りますが、参照アドレスを誤るため4が入ります。

発生例3 (発生条件(2)(b)および (3)(c)に該当する場合) :

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

4が格納されている領域を正しくは9に変更しますが、書き込み領域を誤るため、

3が格納されている領域が9に書き変わります。

5.3 回避策

アドレス参照の結果を、別のポインタ型の変数に代入して使用してください。

発生例1の回避例：

```
-------------------------------------------------------------
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;
            /* アドレス計算の部分をポインタに代入して、それを使用 */
   x = ptr->a;
}
-------------------------------------------------------------
```

発生例2の回避例：

```
-------------------------------------------------------------
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];
            /* アドレス計算の部分をポインタに代入して、それを使用 */
   x = ptr[1].a;
}
-------------------------------------------------------------
```

発生例3の回避例：

```
-------------------------------------------------------------
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;
```

```
            /* アドレス計算の部分をポインタに代入して、それを使用 */
    ptr->b = 9;
}
```

---

6. 恒久対策

2011年5月16日にリリースしたV.1.01 Release 00で改修しました。
V.1.01 Release 00の詳細は、RENESAS TOOL NEWS 資料番号110516/tn3を
参照ください。以下のURLでも参照できます。(5月20日から公開予定)
https://www.renesas.com/search/keyword-search.html#genre=document&q=110516tn3
