CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 ご使用上のお願い

CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 の使用上の注意事項をお知らせします。

注: 各注意事項の後ろの番号は、注意事項の識別番号です。

---

1. 未参照のシンボルがある場合の-smap オプションおよび-goptimize オプション使用に関する注意事項 (RXC#029)

1.1 該当製品およびバージョン

- CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 (型名 : PRX00CSP2-MWR)
  CC-RX コンパイラ V2.00.00 ～ V2.01.00

1.2 内容

外部変数アクセス最適化の-smap オプションおよびモジュール間最適化の-goptimize オプションを指定した場合に、外部変数へアクセスする際の参照アドレスが正しくない場合があります。

1.3 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) コンパイラの外部変数アクセス最適化の-smap オプションを指定している。
(2) モジュール間最適化の-goptimize オプションを指定している。
(3) リンカの最適化-optimize=symbol_delete オプションを指定している。 (注)
(4) ファイル内に未参照の外部変数の定義がある。

注: -goptimize オプションを指定した場合、リンカオプションの
-optimize=same_code, short_format, branch などを指定していなければ、
デフォルトで -optimize=symbol_delete は有効です。

発生例:

```
------------------------------------------------------------------------
int x, z;
static int a, b, c;

#pragma entry main
void main(void){
    x = a;
```

```
    z = c;
}

void func2(void){   // 発生条件(4)
    b++             // 最適化で未参照の func2 と func2 のみで
                    // 参照されている b は削除される
}
```

発生例のコンパイル結果:
-isa=rxv1, -output=abs, -goptimize, -optimize=2 および -smap で
コンパイルした例

```
_main:
            MOV.L   #_x,R14
            MOV.L   08H[R14],[R14]
            MOV.L   10H[R14],04H[R14]  ; 参照アドレスが正しくない
                                       ; MOV.L 0CH[R14],04H[R14]が
                                       ; 正しい
            MOV.L   #0,R1
            RTS

            .SECTION B,DATA
            .ORG    00000010H
_x:
            .BLKL   1
_z:
            .BLKL   1
__$a:
            .BLKL   1
__$c:
            .BLKL   1
            .END    _main
```

1.4 回避策

以下のいずれかの方法で回避してください。

(1) コンパイラの外部変数アクセス最適化の-smap オプションを外す。
    なお、-smap を-map に変更しても回避可能です。

(2) モジュール間最適化の-goptimize オプションを外す。

(3) リンカの最適化-optimize=symbol_delete オプションを外す。

(4) ソースファイルを修正し、未参照の外部変数を削除する。

2. 条件文におけるキーワード __evenaccess 指定変数の使用に関する注意事項(RXC#031)

2.1 該当製品およびバージョン

- CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 (型名 : PRX00CSP2-MWR)
  CC-RX コンパイラ V2.00.00 ～ V2.01.00

2.2 内容

if 文、条件演算子(? :)、または switch 文などの条件文を使用した際に、__evenaccess
指定されている変数に宣言サイズでアクセスしない場合があります。

2.3 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) コンパイル時に-optimize=1, -optimize=2 または -optimize=max のいずれかが指定されている。

(2) if 文、条件演算子(? :)、または switch 文などの条件文を記述している。

(3) 2 バイト、4 バイト、または 8 バイトの整数型に対して、同じサイズの変数（注 1）の参照が複数回ある。（注 2）

(4) (2)の条件文の条件によって、参照される(3)の変数が変わる。

(5) (3)の変数に__evenaccess 指定された変数がある。

(6) (3)のいずれかの変数参照の後からブロック末尾までの間に、他の volatile 変数の読み出し、または書き込みが無い。

注 1: 変数には以下を含みます。

- メンバ変数
- 同じ型で同じビット幅のビットフィールド

注 2: 変数の参照には定数アドレスによるメモリ参照も含みます。

発生例:

```
__evenaccess struct { unsigned short mem; } x0; // 発生条件(5)
unsigned short mem1;
unsigned char test(unsigned char x) {
unsigned short temp;
    if (x) {             // 発生条件(2)
        temp = x0.mem;   // 発生条件(3),(4)および(6)
    } else {
        temp = mem1;     // 発生条件(3),(4)および(6)
    }
    return (char)temp;
}
```

発生例に対する出力コード:
-isa=rxv1, -optimize=2 および -size 指定時の例

```
_test:
        CMP #00H, R1
        MOV.L #_x0, R14
        MOV.L #_mem1, R15
        BEQ L12
L11:    ; entry
        MOV.L R14, R15
L12:    ; entry
        MOVU.B [R15], R1  ; メモリから 2 バイトで読み出すべきところ、
                          ; 誤って 1 バイトで呼び出される
        RTS
```

2.4 回避策

以下のいずれかの方法で回避してください。

(1) -optimize=0 を指定する。

(2) 発生条件(3)の変数参照のいずれかの後ろから、発生条件(2)の分岐後の
合流前までに、volatile 修飾変数の読み出しを挿入する。

発生例に対する回避策(2)の適用例:

```
__evenaccess struct { unsigned short mem; } x0;
unsigned short mem1;
unsigned char test(unsigned char x) {
unsigned short temp;
    volatile char dummy;  // 回避策(2)
    if (x) {
        temp = x0.mem;
        dummy;            // 回避策(2)
    } else {
        temp = mem1;
    }
    return (char)temp;
}
```

3. const 変数へアクセスがある場合の-smap オプションおよび-goptimize オプション使用に関する注意事項 (RXC#032)

3.1 該当製品およびバージョン

- CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 (型名 : PRX00CSP2-MWR)
  CC-RX コンパイラ V2.00.00 ～ V2.01.00

3.2 内容

外部変数アクセス最適化の-smap オプションおよびモジュール間最適化の-goptimize
オプションを指定した場合に、const 変数へアクセスする際の参照アドレスが正しくない
場合があります。

3.3 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) -smap オプションを指定している。

(2) 同一 C/C++コンパイル単位内で、const 付きの静的な変数定義 (注 1) が複数あり、
それらに以下のいずれかを満たす変数と、いずれも満たさない変数が混在する。
  - volatile 付きである
  - 初期値がない

(3) (2)で定義した 2 つの変数を(2)と同一 C/C++コンパイル単位にある関数内で、
それぞれ読み出し、またはアドレス取得のいずれかをしている。

(4) (2)の 2 つの変数は、同じセクションに配置されている。 (注 2)

(5) 以下(5-1)、(5-2)または(5-3)のいずれかを満たす。

(5-1) (2)の const 付き変数定義が、関数外あるいは少なくとも 1 つの
関数内で、次の(a)(b)の順に並んでいない。
(a) 初期値付きでかつ volatile がない const 変数の定義
(b) (a)以外の const 変数の定義

(5-2) (2)の const 付き変数定義が関数外と関数内の両方で定義されており、

それらが次の(a)(b)の両方を満たす。

(a) 関数内で定義した変数に、初期値付きでかつ volatile がない const 変数がある。

(b) 関数外で定義した変数に、(a)以外の const 変数がある。

(5-3) (2)の const 付き変数定義が複数の関数内で定義されており、
それらが次の(a)(b)の両方を満たす。

(a) ある関数内で定義した変数に、初期値付きでかつ volatile がない const 変数がある。

(b) (a)の関数よりも先に定義した関数に、初期値付きでかつ volatile がない const 変数と、そうでない const 変数の両方がある。

注 1: 静的な変数定義には、関数外の変数定義および関数内の static 付きの変数定義を含みます。

注 2: 同じセクションに配置されるのは、次の(a)(b)の両方を満たしている場合です。

(a) #pragma section により異なる出力セクション名が指定されていない。

(b) コンパイル時に-nostuff または-nostuff=C オプションが有効、または 2 つの変数のアライメント数が同じ。

発生例 1:

コンパイルオプションに以下を指定した場合

```
ccrx -output=src -smap -cpu=rx600 file1.c
```

```
const volatile unsigned long var1 = 1; // 発生条件(2),(4)および(5-1)
const long var2 = 2;                   // 発生条件(2),(4)および(5-1)
void call_func1(unsigned long, const long *);
void func1(void) {
    const long *tmp = &var2;           // 発生条件(3)
    call_func1(var1, tmp);             // 発生条件(3)
}
```

発生例 1 に対する出力コード例:

```
        .SECTION    P,CODE
_func1:
        MOV.L #_var1, R14
        MOV.L [R14], R1
        ADD #04H, R14, R2 ; _var1 の 4 バイト先は_var2 のアドレス
                          ; ではなく、正しくない
        BRA _call_func1
        .SECTION    C,ROMDATA,ALIGN=4
_var2:
        .lword  00000002H
_var1:
        .lword  00000001H
```

発生例 2:

コンパイルオプションに以下を指定した場合

```
ccrx -output=src -smap -cpu=rx600 -nostuff=C file2.c
```

```
------------------------------------------------------------------------
const char  var3 = 3;                   // 発生条件(2),(4)
const unsigned short var4;              // 発生条件(2),(4)および(5-1)
const char  var5 = 5;                   // 発生条件(2),(4)および(5-1)
void call_func2(const char *, const char *);
void func2(void) {
    const char *tmp1 = &var5;           // 発生条件(3)
    const char *tmp2 = &var3;           // 発生条件(3)
    call_func2(tmp1, tmp2);
}
------------------------------------------------------------------------
```

発生例 2 に対する出力コード例:

```
------------------------------------------------------------------------
        .SECTION        P,CODE
_func2:
        MOV.L #_var3, R2
        ADD #04H, R2, R1   ; _var3 の 4 バイト先は_var5 のアドレス
                           ; ではなく、正しくない
        BRA _call_func2
        .SECTION        C,ROMDATA,ALIGN=4
_var3:
        .byte   03H
_var5:
        .byte   05H
_var4:
        .word   0000H
------------------------------------------------------------------------
```

発生例 3:

コンパイルオプションに以下を指定した場合

```
ccrx -output=src -smap -cpu=rx600 file3.c
```

```
------------------------------------------------------------------------
const long var6 = 6;                        // 発生条件(2),(4)
const volatile long var7 = 7;               // 発生条件(2),(4)および
                                            // (5-2)の(b)
void call_func3(const volatile long *, const long *);
void func3(void) {
    static const long var8 = 8;             // 発生条件(2),(4)および
                                            // (5-2)の(a)
    const volatile long *tmp1 = &var7;      // 発生条件(3)
    const long *tmp2 = &var8;               // 発生条件(3)
    call_func3(tmp1, tmp2);
}
------------------------------------------------------------------------
```

発生例 3 に対する出力コード例:

```
        .SECTION        P,CODE
_func3:
        MOV.L #_var7, R1
        ADD #04H, R1, R2  ; _var7の4バイト先は_var8のアドレス
                          ; ではなく、正しくない
        BRA _call_func3
        .SECTION        C,ROMDATA,ALIGN=4
_var6:
        .lword  00000006H
__$var8$1:
        .lword  00000008H
_var7:
        .lword  00000007H
```

発生例4:

コンパイルオプションに以下を指定した場合

```
ccrx -output=src -smap -cpu=rx600 file4.c
```

```
void call_func4(const volatile long *, const long *);
void func4(void) {
    static const long var9 = 9;            // 発生条件(2),(4)および
                                           // (5-3)の(b)
    static const volatile long var10 = 10; // 発生条件(2),(4)および
                                           // (5-3)の(b)
    call_func4(&var10, &var9);             // 発生条件(3)
}
    void func4a(void) {
    static const long var11 = 11;          // 発生条件(2),(4)および
                                           // (5-3)の(a)
 . . . . . .
}
```

発生例4に対する出力コード例:

```
        .SECTION        P,CODE
_func4:
        MOV.L #__$var9$1, R2
        ADD #04H, R2, R1  ; _var9の4バイト先は_var10のアドレス
                          ; ではなく、正しくない
        BRA _call_func4
        .SECTION        C,ROMDATA,ALIGN=4
__$var9$1:
        .lword  00000009H
__$var11$3:
        .lword  0000000BH
__$var10$2:
        .lword  0000000AH
```

3.4 回避策

以下のいずれかの方法で回避してください。

(1) 該当する C/C++コンパイル単位内のコンパイル時に、-smap オプションを無効にする。
なお、-smap を-map に変更しても回避可能です。

(2) 発生条件(2)の、const 付きの静的な変数定義の全てに対し、volatile の有無および初期値の有無が、ひとつの C/C++ソース内で互いに同じになるようにする。

(3) 発生条件(5-2)を満たさない場合に、発生条件(2)の const 付きの静的な変数定義が、関数外および全ての関数内のそれぞれで、次の(a)(b)の順序になるように定義順を並び替える。
(a) 初期値付きでかつ volatile がない const 変数の定義
(b) (a) 以外の const 変数の定義

(4) 発生条件(5-2)または(5-3)を満たす場合に、発生条件(5-2)あるいは(5-3)それぞれの、(a)または(b)のどちらか一方が存在しなくなるように該当する変数定義を移動、または削除する。

発生例 1 に対する回避策(2)の適用例:
当該 C/C++ソース中の全ての const 付き変数定義に volatile を付ける

```
const volatile unsigned long var1 = 1;
const volatile long var2 = 2;          // volatile を追加
void call_func1(unsigned long, const long *);
void func1(void) {
    const volatile long *tmp = &var2;  // var2 の型が変わったため
    call_func1(var1, tmp);
}
```

発生例 2 に対する回避策(3)の適用例:
const 付き変数定義の順序を並び替える

```
const char  var3 = 3;
const char  var5 = 5;
const unsigned short var4;              // 初期値がない定義の
                                        // 順序を変更
void call_func2(const char *, const char *);
void func2(void) {
    const char *tmp1 = &var5;
    const char * tmp2 = &var3;
    call_func2(tmp1, tmp2);
}
```

発生例 3 に対する回避策(4)の適用例:
関数外の定義を関数内に移動 (注)

```
const long var6 = 6;
                                        // 関数外の定義を移動
void call_func3(const volatile long *, const long *);
void func3(void) {
    static const long var8 = 8;
```

```
    static const volatile long var7 = 7; // 関数外の定義を
                                         // 関数内 static に移動
    const volatile long *tmp1 = &var7;
    const long *tmp2 = &var8;
    call_func3(tmp1, tmp2);
}
```

注: ソースの意味が変わるため、適用できない場合があります。

4. #pragma address を指定した、構造体、共用体および配列に関する注意事項 (RXC#033)

4.1 該当製品およびバージョン

- CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 (型名 : PRX00CSP2-MWR)
  CC-RX コンパイラ V2.01.00

4.2 内容

#pragma address を指定した、構造体、共用体または配列のいずれかへの
書き込み、または読み出しをした際、異なるアドレスに対して書き込み、
または読み出しが行われる場合があります。

4.3 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) #pragma address を使用している。
(2) (1)の #pragma address の対象は、構造体、共用体または配列のいずれかである。
(3) (2)の構造体、共用体または配列の内部に、当該構造体、共用体または配列
    の始点以外の箇所を先頭とする構造体または配列がある。
(4) (3)が構造体の場合は複数のメンバを持ち、かつオフセットが 0 でない
    メンバを参照している。
    (3)が配列の場合は複数の要素を持ち、かつ、オフセットが 0 でない要素
    を参照している。

発生例:

```
struct st1 {
    short a;
    short b;
};

struct st2 {
    int        c;
    struct st1 tbl2;         // 発生条件(3)
};

#pragma address A=0xFFFF0000 // 発生条件(1)
struct st2 A;                // 発生条件(2)

void func(void)
{
    A.a++;
```

```
    A.tbl2.b = 0;           // 発生条件(4)   A.tbl1.b に代入
                            // してしまう
}
```

4.4 回避策

次のいずれかの方法で回避してください。

(1) #pragma address を使用する代わりに、アドレスを定数で直接記載する。

(2) #pramga address を使用する代わりに、当該変数を別セクションに配置し、
リンカの-start オプションで配置アドレスを指定する。

5. 恒久対策

CubeSuite+版 RX ファミリ用 C/C++コンパイラパッケージ V2 CC-RX コンパイラの
次期バージョン (2014 年 7 月リリース予定)で改修する予定です。

株式会社 日立超LSIシステムズ