RENESAS ツールニュース

MAEC TOOL NEWS: MAECT-M3T-NC308WA-030201D

M3T-NC308WA, M3T-NC30WA ご使用上のお願い

Cコンパイラ(アセンブラ・統合化開発環境付き) M3T-NC308WA, M3T-NC30WAの使用上の注意事項を連絡します。

- 文字列リテラルの16進拡張表記に関する注意事項
- アドレスを引く演算に関する注意事項
- 反復処理の中の条件分岐に関する注意事項

---

1. 文字列リテラルの16進拡張表記に関する注意事項

1.1 該当製品

M32C/80, M16C/80, M16C/70シリーズ用

M3T-NC308WA V.5.00 Release 1

M16C/60, M16C/30, M16C/20, M16C/10シリーズ用

M3T-NC30WA V.5.00 Release 1 ～ V.5.00 Release 2

1.2 内容

文字列リテラル (二重引用符で囲まれた文字の列) 中に、16進拡張表記 (¥xまたは¥Xに続く16進数) で表現された文字がある場合、正しい文字に置換されない場合があります。

※ 文字定数(一重引用符で囲まれた文字)の16進拡張表記は正しく置換されます。

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) 文字列リテラルの中で、16進拡張表記が使用されている。

(2) (1)の16進拡張表記の中の16進数が、小文字の英字 a, b, c, d, e, f のいずれか1つ以上を含んでいる。

1.4 発生例

```
  const char s[] = "¥x5a¥x0d¥x0a";    /* 'Z', 制御文字CR, 制御文字LF */
------------------------------------------------------------------------
```

※ 5AH, 0DH, 0AHの3バイトになるべきところが、81H, 34H, 31Hになります。

1.5 回避策

以下のいずれかの方法で回避してください。
(1) 16進拡張表記に、大文字英字を使用する。
(2) 単純拡張表記を使用する。
(3) 8進拡張表記を使用する。
(4) 配列の初期化であれば、文字定数や16進整定数を使用する。

```
------------------------------------------------------------------------
  const char s1[] = "¥x5A¥x0D¥x0A";       /* 16進拡張表記(大文字) */
  const char s2[] = "Z¥r¥n";              /* 単純拡張表記 */
  const char s3[] = "¥132¥015¥012";       /* 8進拡張表記 */
  const char s4[] = { 'Z', 0x0d, 0x0a };  /* 文字定数と16進整定数 */
------------------------------------------------------------------------
```

1.6 恒久対策

本内容については、次期バージョンアップ時に改修する予定です。

ページの先頭へ

---

2. アドレスを引く演算に関する注意事項

2.1 該当製品

M32C/80, M16C/80, M16C/70シリーズ用 M3T-NC308WA V.5.00 Release 1

2.2 内容

変数のアドレスまたは関数のアドレスを変数から引く式を記述したC言語ソースをコンパイルすると、"System Error" が発生する場合があります。

2.3 発生条件

以下の条件をすべて満たす場合に発生します。
(1) 減算を含む式が存在する。
(2) 減算結果を、メモリ上の変数に格納している。(ポインタで間接参照される領域に格納する場合も含む)
(3) 被減数がメモリ上の変数である。(ポインタで間接参照される変数も含む)
(4) 減数は、以下のいずれかであるか、またはそれをキャストした値である。

- 関数外で宣言された変数のアドレス(配列名も含む)
- 関数内でstatic修飾付きで宣言された変数のアドレス(配列名も含む)
- 関数のアドレス
- 上記3つのいずれかを代入されたポインタ

2.4 発生例

```
------------------------------------------------------------------------
  extern int  iArr[10];          /* 発生条件(4) */
  extern long num;               /* 発生条件(2) */

  void func(long p)              /* 発生条件(3) */
  {
     num = p - (long)&iArr[0];   /* 発生条件(1)(2)(3)(4) */
  }
------------------------------------------------------------------------
```

2.5 回避策

関数内に一時変数を作成し、減数をその一時変数に格納してください。
また、格納後にダミーのasm関数を挿入してください。その一時変数を使って減算を行ってください。

```
------------------------------------------------------------------------
  void func(long p)
  {
     long tmp = (long)&iArr[0];  /* 一時変数に減数を格納 */
     asm();                      /* ダミーのasm関数 */
     num = p - tmp;              /* 一時変数を引く */
  }
------------------------------------------------------------------------
```

2.6 恒久対策

本内容については、次期バージョンアップ時に改修する予定です。

ページの先頭へ

---

3. 反復処理の中の条件分岐に関する注意事項

3.1 該当製品

M32C/80, M16C/80, M16C/70シリーズ用

M3T-NC308WA V.5.00 Release 1

M16C/60, M16C/30, M16C/20, M16C/10シリーズ用

M3T-NC30WA V.5.00 Release 1 ～ V.5.00 Release 2

3.2 内容

反復処理の中の条件分岐に関するコードが生成され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) オプション -O, -O[1-5], -OR, -OS のいずれか1つ以上を使用している。

(2) 反復して実行される処理を持つ関数がある。

(3) (2)の関数内で、char、unsigned char、unsigned int、unsigned long、またはポインタ型のいずれかであってstaticでない変数を宣言している。

(4) 反復処理の前に、(3)の変数に、定数(以下定数(a)と呼ぶ)を代入している。

(5) 反復処理の中で、演算子 "==" または "!=" を用いて、(3)の変数と定数(以下定数(b)と称する)を比較している。

(6) 定数(a)と定数(b)は最上位ビットの値が異なる。

(7) 定数(a)の最上位ビットが 1 のとき、反復処理の中で、(3)の変数に定数を減算している。
または、
定数(a)の最上位ビットが 0 のとき、反復処理の中で、(3)の変数に定数を加算している。

3.4 発生例

[例1]

```
------------------------------------------------------------------------
  int x, y;
  void func(void)
  {
     unsigned int i;                    /* 発生条件(3) */

     for (i = 0x8000U; i != 0x7fffU; i--) { /* 発生条件(2),(4),(5),(6),(7) */
        x += y;                         /* 定数(a) = 0x8000U */
     }                                  /* 定数(b) = 0x7fffU */
  }
```

[例2]

```
int func2(void)
{
   int near    *p = (int near *)0x7000;   /* 発生条件(3),(4) */
                                  /* 定数(a) = 0x7000 */
   int         sum = 0;

   while (p != (int near *)0x8000) {      /* 発生条件(2),(5),(6) */
                                  /* 定数(b) = 0x8000 */
      sum += *p;
      p++;                           /* 発生条件(7) */
   }
   return sum;
}
```

[例3]

```
void func3(void)
{
   char c = 0x80;                      /* 発生条件(3),(4) */
                                  /* 定数(a) = 0x80 */
TOP:                                 /* 発生条件(2) */
   c--;                             /* 発生条件(7) */
   if (c == 0x7f) {                    /* 発生条件(5),(6) */
                                  /* 定数(b) = 0x7f */
      a = 0;
      return;
   }
   b++;
   goto TOP;
}
```

3.5 回避策

以下のいずれかの方法で回避してください。

(1) コードが生成されない比較の個所で、"!=", "==" 演算子の代りに、"<=", ">=", "<", ">" のいずれかの演算子を使用してください。

(2) 他の演算子で置き換えができない場合（発生例(3)の場合など）は、比較と加算または比較と減算の間にダミーのasm関数を挿入してください。

[発生例1の回避例]

```
  int a, b;
  void func(void)
  {
     unsigned int i;

                                /* "!=" から ">" に置き換える */
     for (i = 0x8000U; i > 0x7fffU; i--) {
        a += b;
     }
  }
```

[発生例2の回避例]

```
  int func2(void)
  {
     int near    *p = (int near *)0x7000;
     int         sum = 0;

                                /* "!=" から "<" に置き換える */
     while (p < (int near *)0x8000) {
        sum += *p;
        p++;
     }
     return sum;
  }
```

[発生例3の回避例]

```
  void func3(void)
  {
     char c = 0x80;
  TOP:
     c--;
     asm();                     /* ここに asm(); を挿入する */
     if (c == 0x7f) {
        a = 0;
        return;
     }
     b++;
      goto TOP;
```

```
}
```

3.6 恒久対策

本内容については、次期バージョンアップ時に改修する予定です。
