RENESAS ツールニュース

RENESAS TOOL NEWS 2007年03月01日：070301/tn4

H8SX, H8S およびH8ファミリ用C/C++コンパイラパッケージ V.4 ～ V.6 ご使用上のお願い

H8SX, H8S およびH8ファミリ用C/C++コンパイラパッケージ V.4 ～ V.6 の使用上の 注意事項12件を連絡します。

1. 該当製品

V.4.0 ～ V.6.01 Release 02
製品型名
V.4
Windows版: PS008CAS4-MWR
Solaris版: PS008CAS4-SLR
HP-UX版: PS008CAS4-H7R
V.5
Windows版: PS008CAS5-MWR
V.6
Windows版: R0C40008XSW06R
Solaris版: R0C40008XSS06R
HP-UX版: R0C40008XSH06R

2. 内容

2.1 同じ部分式を使用する際の注意事項(H8C-0057)

該当バージョン:
V.4.0 ～ V.4.0.09
V.5.0 ～ V.5.0.06
V.6.00 Release 00 ～ V.6.00 Release 03
V.6.01 Release 00 ～ V.6.01 Release 02
問題:
関数内の制御式の中に同じ部分式が複数あったとき、分岐先が正しくない場合があります。
発生条件:
以下の条件をすべて満たす場合、発生することがあります。
(1) CPUオプションH8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを

使用している。(例: コマンドラインでは-cpu=H8SXN)
(2) 最適化オプションを使用している(コマンドラインでは使用していない、
または-optimize=1を使用している)。
(3) 関数内の以下の選択文または繰り返し文の制御式に、同じ部分式を
2回以上用いている。
部分式は、単一の制御式か複数の制御式に書いているかは問いません。
(a) if文
(b) for文
(c) while文
(d) do文

発生例:

```
----------------------------------------------------------------------
long a,b;
long sub(void)
{
   long rc;

   rc= -1;
   if ((a>10) && (b>0)){                    // 発生条件(3)
      rc = 1;
   }
   else {
      if (b>0){                             // 発生条件(3)
         rc = 0;
      }
   }
   return (rc);
}
----------------------------------------------------------------------
```

回避策:

以下のいずれかの方法で回避してください。
(1) 最適化オプションを使用しない(コマンドラインでは-optimize=0を
使用)。
(2) 該当する関数に対して、#pragma option nooptimize拡張機能を使用する

2.2 #pragma inline_asmおよび#pragma interrupt使用時の注意事項(H8C-0058)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03
V.6.01 Release 00 ～ V.6.01 Release 02

問題:

#pragma interruptを適用した関数の中で#pragma inline_asmを適用した関数
を呼び出したとき、レジスタの退避および回復コードが出力されない場合が
あります。

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オプション2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは -cpu=2000N)

(2) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。

(3) オブジェクト形式オプションで、アセンブリプログラム(コマンドラインでは-code=asmcode)を使用している。

(4) #pragma interruptまたは__interruptを適用した関数がある。

(5) (4)の関数は、以下の拡張機能を適用していない。

(a) #pragma regsaveまたは__regsave

(b) #pragma asm

(6) #pragma inline_asmを適用した関数がある。

(7) (4)の関数内で、(6)の関数が呼び出されている。

(8) (6)の関数は戻り値がない、または戻り値を(4)の関数で使用していない。

発生例:

```
----------------------------------------------------------------------
#pragma inline_asm(sub)                    // 発生条件(6)
#pragma interrupt(func)                    // 発生条件(4)
void sub(void)                             // 発生条件(7)
{
   MOV.W   #1,R0                           // 発生条件(9)
}
void func(void)
{
   sub();                                  // 発生条件(5)および(8)
}
----------------------------------------------------------------------
```

回避策:

以下のいずれかの方法で回避してください。

(1) #pragma interruptを適用した関数に対して、#pragma regsave または __regsaveを適用する。

(2) #pragma inline_asmを、__asm および #pragma inlineに変更する。

(3) 関数のインライン展開を行わない関数を作成し、#pragma interruptを適用した関数から呼び出す。

2.3 memcpyのインライン展開時の注意事項(H8C-0059)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03

V.6.01 Release 00 ～ V.6.01 Release 02

問題:

memcpy関数をインライン展開したとき、データ転送が指定した回数より少ない

場合があります。
発生条件:
以下の条件をすべて満たす場合、発生することがあります。
(1) CPUオプションH8SXA, H8SXXおよびAE5のいずれかを使用している。
(例: コマンドラインでは-cpu=H8SXA)
(2) memcpy/strcpyのインライン展開オプション(コマンドラインでは
-library=intrinsic)を使用している。
(3) ソースプログラム中でmemcpy関数を使用し、その第3引数の値に0x60001
から0x60005の値を設定している。
発生例:

```
---------------------------------------------------------------------
#include

char source[0x60001];
char destination[0x60001];

void test(void){
   memcpy(destination, source, 0x60001);              // 発生条件(3)
}
---------------------------------------------------------------------
```

回避策:
以下のいずれかの方法で回避してください。
(1) memcpy関数の第3引数をvolatile修飾の付いたsize_t型変数に代入し、
その変数を第3引数に使用する。
例 :

```
    ---------------------------------------------------------------
    #include

    char source[0x60001];
    char *destination;

    void test(void){
       volatile size_t transfer_size = 0x60001;
       memcpy(destination, source, transfer_size);
    }
    ---------------------------------------------------------------
```

(2) memcpy/strcpyのインライン展開オプション(-library=intrinsic)を
使用しない。

2.4 255文字を超える識別子使用時の注意事項(H8C-0060)

該当バージョン:
V.4.0～V.4.0.09
V.5.0～V.5.0.06

V.6.00 Release 00 ～ V.6.00 Release 03
V.6.01 Release 00 ～ V.6.01 Release 02

問題:

シンボル名、セクション名またはファイル名の長さが255文字以上になったとき、間違ったオブジェクトを出力する場合があります。

発生条件:

以下の条件をすべて満たす場合、発生することがあります。

(1) シンボル名、セクション名またはファイル名の長さが255文字以上である。

(2) モジュール間最適化オプション(コマンドラインでは-goptimize)を使用している。

回避策:

以下のいずれかの方法で回避してください。

(1) シンボル名、セクション名およびファイル名のそれぞれの長さを254文字以下にする。

(2) モジュール間最適化オプション(-goptimize)を使用しない。

2.5 配列の添え字計算でオーバーフローが発生するときの注意事項(H8C-0061)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03
V.6.01 Release 00 ～ V.6.01 Release 02

問題:

配列の添え字計算時にオーバーフローが発生するとき、正しくないアドレスをアクセスする場合があります。

発生条件:

以下の条件をすべて満たす場合、発生することがあります。

(1) CPUオプション2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは-cpu=2000N)

(2) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。

(3) 配列型変数を定義および宣言している。

(4) (3)の配列の添え字は定数との加減算を行い、計算後に型変換を行っている。

(5) 以下のいずれかの条件を満たしている。

(a) (4)の型変換は、unsigned long型への拡張である。

(b) 以下の条件をすべて満たしている。

・CPUオプション2000N, 2600N, H8SXNおよびH8SXMのいずれかを使用している。

・(4)の型変換は、unsigned int型またはunsigned short型への拡張である。

(6) (4)の加減算の結果が、オーバーフローしている。

発生例:

```
---------------------------------------------------------------------
#include

unsigned int a = 10;
unsigned int array[100];                    // 発生条件(2)

void main(void){
   unsigned int i;

   for (i=0; i<100; i++){
      array[i] = 0;
   }

   array[4] = 1;
   array[0] = array[(unsigned long)(a + 65530u)];// 発生条件(3)～(5)

   if (array[0] == array[4]){
      printf("correct¥n");
   }
}
---------------------------------------------------------------------
```

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(4)での加減算式を、volatile修飾した変数に代入し、その変数を使用する。

(2) 添え字の加減算の定数を、volatile修飾した変数に代入して使用する。

(3) 発生条件(4)の加減算式を、オーバーフローしないように修正する。

2.6 const修飾したメンバの参照時の注意事項(H8C-0062)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03

V.6.01 Release 00 ～ V.6.01 Release 02

問題:

const修飾した構造体型変数または共用体型変数を繰り返し文の中で宣言したとき、それらのメンバの参照が正しくない場合があります。

発生条件:

以下の条件をすべて満たす場合、発生することがあります。

(1) 最適化オプションを使用している(コマンドラインでは使用していない、または-optimize=1を使用している)。

(2) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。

(3) 関数内に繰り返し文がある。

(4) (3)の繰り返し文の中に、構造体型変数または共用体型変数への代入が

ある。
(5) 以下のいずれかの条件を満たしている。
(a) (4)の代入先の構造体型変数または共用体型変数を、const修飾している。
(b) (4)の代入先の構造体型変数または共用体型変数のメンバを、const修飾している。

発生例:

```
typedef struct {
   char m1;
   char m2;
} S;

S s;

long func(void){

   long i;
   long val = 0;

   for (i=0; i<2; i++){                    // 発生条件(3)
      const S t = s;                      // 発生条件(4)および(5)

      val += t.m1;
      val += t.m2;
   }

   return val;
}
```

回避策:

以下のいずれかの方法で回避してください。
(1) 最適化オプションを使用しない(コマンドラインでは-optimize=0を使用)。
(2) 構造体型変数、共用体型変数またはそれらのメンバをconst修飾しない。
(3) 構造体型変数または共用体型変数への代入を繰り返し文の前で行う。

2.7 volatile修飾した変数と定数との加算における注意事項(H8C-0063)

該当バージョン:

V.6.01 Release 00 ～ V.6.01 Release 02

問題:

volatile修飾した変数の加算を行ったとき、アクセス回数が記述した回数と異なる場合があります。

発生条件:

以下の条件をすべて満たす場合、発生することがあります。

(1) CPUオプション2000N, 2000A, 2600Nおよび2600Aのいずれかを使用している。
(例: コマンドラインでは-cpu=2000N)

(2) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。

(3) 関数内に加算式を代入する式がある。

(4) (3)の代入式の左辺にある変数をvariable修飾している。

(5) (4)の変数は、unsigned long型またはsigned long型のいずれかである。

(6) (3)の代入式の右辺の加算式は以下のいずれかである。
(a) 変数 + 定数
(b) 定数 + 変数

(7) (6)の変数と(4)の変数は、同じ変数である。

(8) (6)の定数は、3, 5, 6または8である。

発生例:

```
---------------------------------------------------------------------
volatile unsigned long a;              // 発生条件(4)

void main(void){
   a = a + 3;                          // 発生条件(3)および(5)から(8)
}
---------------------------------------------------------------------
```

回避策:

以下の方法で回避してください。

(1) 加算に用いた定数を外部変数に代入し、その変数を加算式に用いる。
例 :

```
---------------------------------------------------------------------
volatile unsigned long a;
unsigned long b = 3;

void main(void){
   a = a + b;
}
---------------------------------------------------------------------
```

2.8 3バイトの構造体型変数を使用するときの注意事項(H8C-0064)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03
V.6.01 Release 00 ～ V.6.01 Release 02

問題:

3バイトの構造体型変数を含む4バイトの構造体型変数において、3バイトの構造体型変数の転送を行うとき、それ以外の1バイトの領域のデータを誤って書き換える場合があります。

発生条件:
以下の条件をすべて満たす場合、発生することがあります。
(1) CPUオプション2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは -cpu=2000N)
(2) 最適化オプションを使用している(コマンドラインでは使用していない、または-optimize=1を使用している)。
(3) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。
(4) ソースプログラムの中で、3バイトの構造体型変数をメンバとして持つ4バイトの構造体型変数を定義および宣言している。
(5) (4)の構造体型変数を、volatile修飾していない。
(6) (4)の3バイトのメンバを転送している。

発生例:

```
------------------------------------------------------------------------
typedef struct {
    char a[3];
} ST3;

typedef struct {                            // 発生条件(4)
    ST3  st3;
    char x;
} ST;

ST3 stg;                                    // 発生条件(5)

void sub(ST);

void main(void){
    ST st;
    st.x = 10;
    st.st3 = stg;                           // 発生条件(6)
    sub(st);
}
------------------------------------------------------------------------
```

回避策:
以下のいずれかの方法で回避してください。
(1) 最適化オプションを使用しない(コマンドラインでは-optimize=0を使用)。
(2) 3バイトの構造体型変数を含む4バイトの構造体型変数をvolatile修飾する。

2.9 ビット毎の論理積演算をするときの注意事項(H8C-0065)

該当バージョン:
V.6.00 Release 00 ～ V.6.00 Release 03,
V.6.01 Release 00 ～ V.6.01 Release 02
問題:
変数と定数とのビット毎の論理積演算の結果を判定するとき、正しい判定結果を得ることができない場合があります。
発生条件:
以下の条件をすべて満たす場合、発生することがあります。
(1) CPUオプション2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは -cpu=2000N)
(2) 最適化オプションを使用している(コマンドラインでは使用していない、または-optimize=1を使用している)。
(3) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。
(4) 以下のいずれかの条件を満たしている。
(a) 以下の条件をすべて満たしている。
・変数と定数の論理積演算を行っている。
・論理積演算で使用する変数は、unsigned long型、またはsigned long型のいずれかである。
・論理積演算で使用する定数は、0xFFFF以下または下位2バイトが0x0000のいずれかである。
・論理積演算の結果と0とを比較している。
・上記の演算結果は条件式のみで使用される。
・論理積演算で使用する変数は、以下のすべての条件を満たしている。
- evenaccessキーワードが付加されていない。
- volatile修飾していない。
- スタック領域に割り付けられた仮引数である。
(b) インクリメントまたはデクリメントを行っているポインタ型変数がある。
(c) 連続した領域を参照している配列型などの変数がある。
(d) 仮引数がスタック領域に割り付けられている。
発生例:

```
---------------------------------------------------------------------
void func(long dummy1, long dummy2, signed long data1)
{
   if ((data1 & 0x00008000) == 0){              // 発生条件(4)-(a)
      ans1 = 10;
   }else{
      ans1 = 20;
   }
}
```

回避策:

以下のいずれかの方法で回避してください。

(1) 最適化オプションを使用しない(コマンドラインでは-optimize=0を使用)。
または、発生条件を満たす関数に対して#pragma option nooptimizeを適用する。

(2) 発生条件(4)-(a)を満たすときは、論理積演算で使用する変数をvolatile修飾およびevenaccessキーワードを付加した別の変数に代入し、その変数を用いて論理積演算を行う。

2.10 共用体型変数を初期化するときの注意事項(H8C-0066)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03,
V.6.01 Release 00 ～ V.6.01 Release 02

問題:

3バイトの共用体型変数の一番始めのメンバが3バイトより小さく、その共用体型変数に初期化指定子を付加しているとき、同じ共用体の別メンバへ代入するコードが出力されます。

発生条件:

以下の条件をすべて満たす場合、発生します。

(1) CPUオプションH8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは-cpu=H8SXN)

(2) ソースプログラムの中において、初期化指定子がある共用体型変数を宣言している。

(3) (2)の共用体型変数のサイズが3バイトである。

(4) (2)の共用体変数の一番始めのメンバが、1バイトまたは2バイトの変数である。

発生例:

```
---------------------------------------------------------------------
typedef union {                                 // 発生条件(3)
   char a;                                     // 発生条件(4)
   char b[3];
} UNI;

void sub(UNI);

void func(void){
   volatile UNI uni = {1};                          // 発生条件(2)

   sub(uni);
}
---------------------------------------------------------------------
```

回避策:

以下のいずれかの方法で回避してください。

(1) 共用体型変数の宣言と初期化を同時に行わない。

```
typedef union {
   char a;
   char b[3];
} UNI;

void sub(UNI);

void func(void){
   volatile UNI uni;
   uni.a = 1;

   sub(uni);
}
```

(2) 該当する共用体型変数のアドレスを、ポインタ型変数に代入する式を追加する。

```
typedef union {
   char a;
   char b[3];
} UNI;

void sub(UNI);

void func(void){
   volatile UNI uni = {1};
   volatile UNI *p;
   p = &uni;

   sub(uni);
}
```

2.11 12ビット幅のビットフィールドを使用するときの注意事項(H8C-0067)

該当バージョン:

V.6.01 Release 00 ～ V.6.01 Release 02

問題:

12ビットのビットフィールドとして定義された構造体型変数のメンバに値を代入したとき、同じ記憶域単位に定義された他のビットフィールドメンバの

値が書き換えられます。

発生条件:

以下の条件をすべて満たす場合、発生します。

(1) CPUオプションH8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは-cpu=H8SXN)

(2) スピード優先最適化オプションの四則演算、比較、代入式の高速化サブオプション(コマンドラインでは-speed=expression)を使用している。またはスピード優先最適化オプション(コマンドラインでは-speed)を使用している。

(3) 構造体型変数を定義および宣言している。

(4) (3)の構造体型変数に、12ビットのビットフィールドのメンバが存在する。

(5) (4)のビットフィールドメンバは、以下のいずれかの型で定義されている。

(a) signed int型

(b) unsigned int型

(c) signed short型

(d) unsigned short型

(6) (4)のメンバの前に、(4)のメンバと同じ型の1ビットのビットフィールドが存在する。

(7) (4)のメンバへの代入文がある。

(8) (7)の代入の後で、同じ記憶域単位の別のビットフィールドメンバの参照が行われている。

発生例:

```
---------------------------------------------------------------------
#include

typedef struct {                        // 発生条件(3)
   int broken_data1:1;                    // 発生条件(5)および(6)
   int target_data:12;                   // 発生条件(4)および(5)
   int broken_data2:3;
} ST;

void main(void){
   ST st;                             // 発生条件(3)
   st.broken_data1 = -1;
   st.broken_data2 = -1;
   st.target_data  =  0;                 // 発生条件(7)

   if (st.broken_data1 == -1              // 発生条件(8)
      && st.broken_data2 == -1){
      printf("correct¥n");
   }
```

```
}
```

回避策:

以下のいずれかの方法で回避してください。

(1) スピード優先最適化オプションの四則演算、比較、代入式の高速化サブオプション(-speed=expression)を使用しない。

(2) 12ビットのビットフィールドのメンバのオフセットを1ビット以外にする。

```
#include

typedef struct {
    int target_data:12;             // メンバの順序を入れ替える
    int broken_data1:1;              // メンバの順序を入れ替える
    int broken_data2:3;
} ST;

void main(void){
    ST st;
    st.broken_data1 = -1;
    st.broken_data2 = -1;
    st.target_data  =  0;

    if (st.broken_data1 == -1
        && st.broken_data2 == -1){
        printf("correct¥n");
    }
}
```

2.12 埋め込みアセンブル機能を使用するときの注意事項(H8C-0068)

該当バージョン:

V.6.00 Release 00 ～ V.6.00 Release 03,
V.6.01 Release 00 ～ V.6.01 Release 02

問題:

埋め込みアセンブル機能を用いたとき、次の問題が発生する場合があります。

・ディスプレースメント付きアドレッシングモードで、定数値が書き換わる。

・スタックに割りついた変数のアクセスで、間違った変数をアクセスする。

発生条件:

以下のいずれかの条件を満たす場合、発生することがあります。

(1) 以下の条件をすべて満たしている。

(a) CPUオプション2000A, 2600A, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは-cpu=2000A)

(b) CPUオプションのアドレス空間サイズとして1M, 16Mまたは256Mを使用

している。(コマンドラインでは:20,:24または:28を使用している)
(c) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。
(d) 埋め込みアセンブル機能 __asmを使用している。
(e) (d)の複文の中に以下のアドレッシングモードまたは命令のいずれかを使用している。
・ MOVA命令
・ ディスプレースメント付きレジスタ間接
・ ディスプレースメント付きインデックスレジスタ間接
(f) (e)のディスプレースメント値に、0x10000以上の定数値を記述している。
(2) 以下の条件をすべて満たしている。
(a) V.6.01 Release 02を使用している。
(b) CPUオプション2000N, 2000A, 2600N, 2600A, H8SXN, H8SXM, H8SXA, H8SXXおよびAE5のいずれかを使用している。(例: コマンドラインでは-cpu=2000N)
(c) 出力オブジェクト互換オプション(コマンドラインでは-legacy=v4)を使用していない。
(d) 埋め込みアセンブル機能 __asmを使用している。
(e) スタックに割りついた局所変数または引数が存在する。
(f) (d)の中でディスプレースメント付きレジスタ間接の命令を用いて、(e)の変数または引数をアクセスしている。
(g) (e)の変数または引数は、スタックポインタからのオフセットが0以外である。

発生例:

```
----------------------------------------------------------------------
void func(void){
  __asm{                          // 発生条件(1)-(d)
    mov.l @(0x0010000, er0), er1      // 発生条件(1)-(e)および(f)
  }
}
----------------------------------------------------------------------
```

回避策:

(1) 発生条件(1)を満たしている場合、以下の方法で回避してください。
発生条件(1)-(e)の命令を使用しない。
例 :

```
    ----------------------------------------------------------------
    void func(void){
      __asm{
        mov.l er0, er4
        add.l #0x00010000:32, er4
        mov.l @er4, er1
      }
```

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

(2) 発生条件(2)を満たしている場合、以下のいずれかの方法で回避してください。

(a) 埋め込みアセンブル機能__asmを#pragma asmに変更する。

(b) 埋め込みアセンブル機能で記述している部分を、#pragma inline_asmを適用した関数にして呼び出す。

3. 恒久対策

本内容は、V.6.01 Release 03で改修する予定です。
