マイクロコンピュータ技術情報

技術通知 78K4 シリーズ C コンパイラ CC78K4 使用制限事項の件	発行番号	ZBG－CD－05－0003号	1 /2
	発行日	2005年 1月 12日	
	発行部門	NEC エレクトロニクス株式会社 第四システム事業本部 汎用マイコンシステム事業部 開発ツールグループ 西林	

文書分類	○	使用制限事項		バージョン・アップ		ドキュメント誤記訂正（正誤表）		その他

関連資料		
	CC78K4 Ver.2.40 以上 操作編	資料番号：U16707JJ1V0UM00(第 1 版)
	CC78K4 Ver.2.30 以上 言語編	資料番号：U15556JJ1V0UM00（第 1 版）
	78K4 シリーズ C コンパイラ CC78K4 Ver.2.40 使用上の留意点	資料番号：SUD-DT-04-0118

CP(K),O

1. 対象製品

CC78K4　　Ver.2.40

2. 新たな制限事項

今回新たに制限事項No. 26を追加しました。詳細は、別紙1を参照してください。

・静的変数を浮動小数点定数で初期化すると、初期値が不正になる制限

3. 回避策

今回新たに追加した制限事項の回避策です。詳細は、別紙1を参照してください。

・該当する部分の定数を整数定数で初期化する。
あるいは浮動小数点定数を適切な整数型にキャストする。

4. 改善計画

今回新たに追加したNo. 26は、次バージョンで改善することを予定しています。

CC78K4 V2.41　　2005 年 4 月リリース予定

※ 本日程については予告なく変更する場合がございますので、改善品のリリース日程については、別途、弊社営業までお問い合わせください。

2. 制限事項一覧

制限事項の履歴とその詳細情報が含まれました使用制限事項一覧を別紙1に記載します。

3. 発行文書履歴

78K4 シリーズ C コンパイラ CC78K4 使用制限事項一覧 発行文書履歴

文書番号	発行日	記事
ZBG-CD-04-0054	2004 年 8 月 18 日	統合版初版
ZBG-CD-04-0074	2004 年 10 月 4 日	制限事項追加(No.23 ～ No.25)
ZBG-CD-05-0003	2005 年 1 月 12 日	制限事項追加(No.26)

以上

CC78K4 の使用制限事項一覧

1. 製品履歴

No.	仕様変更・追加／不具合事項	バージョン	
		V2.30	V2.40
1	文字列中に NULL 文字があると NULL 文字以降の文字列が無効になる制限	×	○
2	配列のオフセットを使用したコードが不正になる場合がある制限	×	○
3	フラッシュ領域中の関数のアドレスを参照するとコード不正となる場合がある制限	×	○
4	#if の定数式を正しく処理しない場合がある制限	×	○
5	多次元配列のアドレス参照後に同じ配列の要素を参照するとコード不正となる場合がある制限	×	○
6	ビット単位の AND/XOR/OR の複合代入演算と switch 文のコード不正となる場合がある制限	×	○
7	負の浮動小数点定数を符号なし整数型にキャストするとコード不正となる制限	×	○
8	浮動小数点数同士の論理和／論理積演算を行うとコード不正となる制限	×	○
9	関数パラメータが register 宣言した配列の場合にコード不正となる制限	×	○
10	#pragma section 使用時、複数ファイルで同名の変数を宣言すると、変数を正しいセクションに配置しない場合がある制限	×	○
11	浮動小数点定数と整数型との論理和／論理積演算を行うとコード不正となる制限	×	○
12	ラージ・モデルにおいて、サイズが 5 バイト以上の構造体／共用体を引数に持つ関数呼び出しで、コード不正となる場合がある制限	×	○
13	ビット・フィールドをメンバに持つ構造体配列に対して、配列要素の上位バイトにあるビット・フィールドへの代入演算で、配列の添字に変数を使用すると、コード不正となる場合がある制限	×	○
14	関数 scanf/sscanf で、10 進整数の変換指定子を使用し、文字列変換後の数値が 65536 以上または-65536 以下となる時に、不正な結果を返却する場合がある制限	×	○
15	ミディアム・モデルにおいて、関数 printf/sprintf/vprintf/vsprintf で、浮動小数点数の出力変換結果が不正となる場合がある制限	×	○
16	ブロック内で extern 宣言された外部変数の初期化がエラーにならず、また、アセンブラ・ソース中のディバグ情報が不正となる制限	×	×
17	ブロック内で extern 宣言された変数と同名の変数との結合が不正となる場合がある制限	×	×
18	関数プロトタイプ宣言または型修飾子 const, volatile を使用した宣言に typedef で定義された型（typedef 名）を使用するとエラーとなる場合がある制限	×	×
19	大きさが定義されていない多次元配列が不正動作となる場合がある制限	×	×
20	引数を持つ関数のアドレスを返却する関数において前記引数は参照できないが参照時エラーとならず不正なコードを出力する制限	×	×
21	signed 型のビット・フィールドを符号無しのビット・フィールドとして処理する制限	×	×
22	ラージ・モデルにおいて，1 関数内の auto 変数の総サイズが 65536 バイトを越えると，出力コードやディバグ情報が不正となる制限	×	×

×：該当する
○：該当しない
－：対象外

No.	仕様変更・追加／不具合事項	バージョン	
		V2.30	V2.40
23	signed char 型と定数値の演算がコード不正となる制限	×	×※
24	複合代入%=, /=の演算がコード不正となる制限	×	×※
25	自動変数の配列名を参照で W503 を出力する制限	×	×
26	静的変数を浮動小数点定数で初期化すると、初期値が不正になる制限	×	×※

×：該当する
〇：該当しない
－：対象外
※：チェックツールあり

2. 使用制限事項の詳細

No. 1 文字列中に NULL 文字があると NULL 文字以降の文字列が無効になる制限

【内　容】文字列中に NULL 文字があると、NULL 文字以降の文字列が無効になります。

（例）

```
const char str[] = "test¥0TEST";
```

NULL 文字以降の文字列に対し、領域が確保されません。

【回避策】次のように記述を変更してください。

（例）

```
const char str[] = {'t','e','s','t','¥0','T','E','S','T'};
```

【改善策】Ver.2.40 で修正済みです。

No. 2 配列のオフセットを使用したコードが不正になる場合がある制限

【内　容】スモール・モデルまたはミディアム・モデルにおいて、以下の条件(1)～(3)の少なくとも 1 つを満たす時に、不正コードとなります。

(1) オフセット付きの sreg/_sreg 変数と文字列との加減代入演算

(2) saddr 領域データと配列との条件式、または SFR シンボルと配列との条件式で、配列のオフセットが 256 以上

(3) norec 関数の引数_@NRARGx に配列を渡す時、配列のオフセットが 256 以上

（例）

```
#pragma sfr
_leaf void nrfunc(int, int, int, int *);
_sreg struct {
    int a;
    int b;
} st;
int a[200], i;
void func()
{
    st.b -= (int)"abc";                 /* (1) */
    if (CR00 == (int)&a[199]) i++;      /* (2) */
    nrfunc(1, 2, 3, &a[199]);           /* (3) */
}
```

【回避策】条件(1)～(3)のそれぞれの場合に、次のように回避してください。

(1) 加減算と代入演算とに分けてください。

（例）

```
st.b = st.b - (int)"abc";
```

(2) テンポラリ変数をもうけ、演算結果をテンポラリ変数に代入しながら演算してください。

（例）

```
int tmp1;
tmp1 = (int)&a[199];
if (CR00 == tmp1) i++;
```

(3) テンポラリ変数をもうけ、演算結果をテンポラリ変数に代入しながら演算してください。

（例）

```
int *tmp2;
tmp2 = &a[199];
nrfunc(1, 2, 3, tmp2);
```

【改善策】Ver.2.40 で修正済みです。

No. 3 フラッシュ領域中の関数のアドレスを参照するとコード不正となる場合がある制限

【内　容】フラッシュ領域にある関数のアドレスを参照すると、コード不正となる場合があります。

(例)

```
#pragma ext_table 0x2000
#pragma ext_func func1 1
#pragma ext_func func2 2

int func1(int, int);
int func2(int, int);
int (*func_b1)(int, int) = &func1;
_sreg int (*func_b2)(int, int) = &func2;
void func()
{
    func_b1 = func1;    /* 正常コード */
    func_b2 = func2;    /* 不正コード */
}
```

【回避策】ありません。

【改善策】Ver.2.40 で修正済みです。

No. 4 #if の定数式を正しく処理しない場合がある制限

【内　容】#if の定数式を正しく処理しない場合があります。

(例)

```
#define a
#if a
int i;
#endif

void func()
{
    i++;
}
```

【回避策】ありません。

【改善策】Ver.2.40 で修正済みです。

No. 5 多次元配列のアドレス参照後に同じ配列の要素を参照するとコード不正となる場合がある制限

【内　容】多次元配列のアドレス参照後に、同じ配列の要素を参照すると、コード不正となる場合があります。

(例)

```
unsigned char a[10][1];
unsigned char *p,c,uc1,uc2;
void func()
{
    p = a[uc1];
    c = a[uc1][uc2];    /* 不正コード */
}
```

【回避策】ありません。

【改善策】Ver.2.40 で修正済みです。

No. 6 ビット単位の AND/XOR/OR の複合代入演算と switch 文のコード不正となる場合がある制限

【内　容】ビット単位の AND/XOR/OR の複合代入演算で使用した変数を、直後の switch 文で参照すると、コード不正となる場合があります。

(例)

```
unsigned int ui, i;
void func(unsigned int p1, unsigned int p2)
{
    ui |= p2;
    switch (p2) {       /* 不正コード */
        case 0:
        case 1:
        case 2:
            i++;
    }
}
```

【回避策】ビット単位の AND/XOR/OR 演算と単純代入演算とに分けてください。

(例)

```
ui = ui | p2;
```

【改善策】Ver.2.40 で修正済みです。

No. 7 負の浮動小数点定数を符号なし整数型にキャストするとコード不正となる制限

【内　容】負の浮動小数点定数を符号なし整数型にキャストすると、コード不正となります。

(例)

```
unsigned long ans;
float f1 = -3.8f;
void func()
{
    int a;
    ans = f1;
    a = ((unsigned long)(-3.8f) != ans);
}
```

【回避策】符号付き整数型にキャストした後に、符号なし整数型にキャストしてください。

(例)

```
a = ((unsigned long)(long)(-3.8f) != ans);
```

【改善策】Ver.2.40 で修正済みです。

No. 8 浮動小数点数同士の論理和／論理積演算を行うとコード不正となる制限

【内　容】浮動小数点数同士の論理和／論理積演算を行うと、コード不正となります。

(例)

```
void func()
{
    int r1, r2;
    float f1 = 17, f2 = 16;
    r1 = f1 || f2;
    r2 = f1 && f2;
}
```

【回避策】ありません。

【改善策】Ver.2.40 で修正済みです。

No. 9 関数パラメータが register 宣言した配列の場合にコード不正となる制限

【内　容】関数パラメータが register 宣言した配列の場合に、コード不正となります。

(例)

```
void func(register char a[])
{
    register char *p;
    p = (char *)a;
}
```

【回避策】ポインタ型として記述してください。

(例)

```
void func(register char *a)
{
    register char *p;
    p = a;
}
```

【改善策】Ver.2.40 で修正済みです。

No. 10 #pragma section 使用時、複数ファイルで同名の変数を宣言すると、変数を正しいセクションに配置しない場合がある制限

【内　容】#pragma section 使用時、複数ファイルで同名の変数を宣言すると、変数を正しいセクションに配置しない場合があります。

(例)

```
--- test.c ---
#include "a.h"
#include "b.h"
#include "c.h"

--- a.h ---
#pragma section @@DATA DAT1
int i;
#pragma section @@DATA DAT2

--- b.h ---
#pragma section @@DATA DAT3
int j;
#pragma section @@DATA DAT4

--- c.h ---
#pragma section @@DATA DAT5
extern int i;     /* int i; でも同じ */
#pragma section @@DATA DAT6
```

【回避策】#pragma section 使用時は、複数ファイルで同名の変数を使用しないでください。

【改善策】Ver.2.40 で修正済みです。

No. 11 浮動小数点定数と整数型との論理和／論理積演算を行うとコード不正となる制限

【内　容】浮動小数点定数と整数型との論理和／論理積演算を行うと、コード不正となります。

（例）

```
void func()
{
        int rval;
        int cZero = 0;
        rval = 0.0F || cZero;          /* Windows 版でコード不正 */
        rval = cZero || 0.0F;          /* UNIX 版でコード不正 */
}
```

【回避策】論理和／論理積演算の演算数に、浮動小数点定数を記述しないでください。

【改善策】Ver.2.40 で修正済みです。

No. 12 ラージ・モデルにおいて、サイズが 5 バイト以上の構造体／共用体を引数に持つ関数呼び出しで、コード不正となる場合がある制限

【内　容】ラージ・モデルにおいて、サイズが 5 バイト以上の構造体／共用体を引数に持つ関数呼び出しで、コード不正となる場合があります。

（例）

```
struct {
        int a;
        int b;
        int c;
} st;
void (*fp)();

void func()
{
        (*fp)(st);
}
```

【回避策】ありません。

【改善策】Ver.2.40 で修正済みです。

No. 13 ビット・フィールドをメンバに持つ構造体配列に対して、配列要素の上位バイトにあるビット・フィールドへの代入演算で、配列の添字に変数を使用すると、コード不正となる場合がある制限

【内　容】ビット・フィールドをメンバに持つ構造体配列に対して、配列要素の上位バイトにあるビット・フィールドへの代入演算で、配列の添字に変数を使用すると、コード不正となる場合があります。

（例）

```
struct st {
    unsigned int bit0:1;
    unsigned int bit1:1;
    unsigned int bit2:1;
    unsigned int bit3:1;
    unsigned int bit4:1;
    unsigned int bit5:1;
    unsigned int bit6:1;
    unsigned int bit7:1;
    unsigned int bit8:1;
    unsigned int bit9:1;
    unsigned int bitA:1;
    unsigned int bitB:1;
    unsigned int bitC:1;
    unsigned int bitD:1;
    unsigned int bitE:1;
    unsigned int bitF:1;
} str[5];
unsigned char num;

void func()
{
    str[num].bit8 = 1;   /* 不正コード */
}
```

【回避策】テンポラリ変数をもうけ、演算結果をテンポラリ変数に代入しながら演算してください。

（例）

```
struct st *tmp;
tmp = &str[num];
tmp->bit8 = 1;
```

【改善策】Ver.2.40 で修正済みです。

No. 14 関数 scanf/sscanf で、10 進整数の変換指定子を使用し、文字列変換後の数値が 65536 以上または-65536 以下となる時に、不正な結果を返却する場合がある制限

【内　容】関数 scanf/sscanf で、10 進整数の変換指定子を使用し、文字列変換後の数値が 65536 以上または-65536 以下となる時に、不正な結果を返却する場合があります。
内部的に下位の桁から 1 文字ずつ数字に置き換える処理をしています。
(1) 1 文字取得して数値にする
(2) 入力文字列が終わりであれば、(7)にジャンプ
(3) 基数を乗算する（桁上げの計算）
(4) 次の 1 文字取得して数値にする
(5) 乗算した結果に加算する　　【この処理が該当】
(6) (2)にジャンプ
(7) 終了
上記の処理(5)の下位 2byte の加算の際に、65535 を超えた場合に不正な値になります。

(例)

```
long a;
char buf[16];

void func()
{
    strcpy(buf, "65536");
    sscanf(buf, "%ld", &a);
}
```

【回避策】次のいずれかの方法で回避してください。
(1) 入力文字列、変換指定子ともに、8 進/16 進表記に変更してください。

(例)

```
strcpy(buf, "0x10000");
sscanf(buf, "%lx", &a);
```

(2) 文字列から値を取得するのであれば、atol 関数を使用してください。

(例)

```
#include<stdlib.h>
    a = atol(buf);
```

【改善策】Ver.2.40 で修正済みです。

No. 15 ミディアム・モデルにおいて、関数 printf/sprintf/vprintf/vsprintf で、浮動小数点数の出力変換結果が不正となる場合がある制限

【内　容】ミディアム・モデルにおいて、関数 printf/sprintf/vprintf/vsprintf で、浮動小数点数の出力変換結果が不正となる場合があります。

(例)

```
#include <stdio.h>
void func()
{
    char res[20];
    sprintf(res, "%f", 1.2e-38);
}
```

【回避策】ありません。
【改善策】Ver.2.40 で修正済みです。

No. 16 ブロック内で extern 宣言された外部変数の初期化がエラーにならず、また、アセンブラ・ソース中のディバグ情報が不正となる制限

【内　容】ブロック内で extern 宣言された外部変数の初期化は，ANSI C 言語仕様に合致しないためエラーとすべき記述ですが，エラーとなりません。コンパイラは，初期値ありの外部変数が定義されたものと解釈してコードを出力します。
コンパイラが出力したオブジェクト中のディバグ情報は正常ですが，アセンブラ・ソース中のディバグ情報が不正になります。

（例）
```
int i ;
void f( void ) {
        extern int i = 2;
}
```

【回避策】ありません。

【改善策】制限事項とします。

No. 17 ブロック内で extern 宣言された変数と同名の変数との結合が不正となる場合がある制限

【内　容】ブロック内で extern 宣言された変数と同名の変数との結合が、以下の 4 つの条件のうち、いずれかに該当する場合に不正となります。

(1) ブロック内で extern 宣言された変数と、以降のブロック外で static 宣言された変数が同名である場合
この場合、エラーとならず、結合もしないため、この変数を参照すると不正なコードを出力します。

（例）
```
void  f ( void ) {
        extern  int  i ;
        i = 1 ;                 /* 不正コード出力 */
}
static  int  i ;
```

(2) ブロック内で extern 宣言された変数と、以降のブロック外で static 宣言されない変数が同名である場合
この場合、結合せずに、不正なコードを出力します。

（例）
```
void  f ( void ) {
        extern  int  i ;
        i = 1 ;                 /* 不正コード出力 */
}
int  i ;
```

(3) ブロック内で extern 宣言された変数と、以前のブロック外で extern 宣言されない変数が同名であり、さらに extern 宣言された変数があるブロックを囲んでいるブロック中で宣言されている自動変数が同名である場合
この場合、ブロック外の変数とブロック内で extern 宣言された変数は結合せず、不正なコードを出力します。

（例）
```
int  i  =  1 ;
void  f ( void ) {
        int  i ;
        {
                extern  int  i ;
                i = 1 ;     /* 不正コード出力 */
        }
}
```

(4) ブロック内で extern 宣言された変数と、他のブロック内で extern 宣言された変数が同名である場合
この場合、結合せず、不正なコードを出力します。

(例)

```
void  f1( void ) {
      extern  int  i ;
      i = 2 ;
}
void  f2( void ) {
      extern  int  i ;
      i = 3 ;
}
```

【回避策】 ありません。
【改善策】 制限事項とします。

No. 18 関数プロトタイプ宣言または型修飾子 const, volatile を使用した宣言に typedef で定義された型（typedef 名）を使用するとエラーとなる場合がある制限

【内　容】 関数プロトタイプ宣言または型修飾子 const、volatile を使用した宣言に、typedef で定義された型(typedef 名)を使用すると、typedef の展開が不正となり、エラーとなる場合があります。

(例1)

```
typedef  int  FTYPE( );
FTYPE  func ;
int  func ( void ) ;                  /* F713  Redefined  'func' */
```

(例2)

```
typedef  int  VTYPE[2] ;
typedef  int  *VPTYPE[3] ;
const  VTYPE  *a ;
const  int  ( *a )[2] ;               /* F713  Redefined  'a' */
volatile  VPTYPE  b[2] ;
volatile  int  *volatile  b[2][3] ;   /* F713  Redefined  'b' */
```

【回避策】 ありません。
【改善策】 制限事項とします。

No. 19 大きさが定義されていない多次元配列が不正動作となる場合がある制限

【内　容】 大きさが定義されていない多次元配列が、不正動作となる場合があります。

(例1)

```
char  c [ ] [3] = { { 1 }, 2, 3, 4, 5 } ;          /* 不正コード */
```

(例2)

```
char  c [ ] [2] [3] = { "ab", "cd", "ef" } ;   /* エラー(F756) */
```

【回避策】 多次元配列の大きさを定義してください。
【改善策】 制限事項とします。

No. 20 引数を持つ関数のアドレスを返却する関数において前記引数は参照できないが参照時エラーとならず不正なコードを出力する制限

【内　容】引数を持つ関数のアドレスを返却する関数において、前記引数は参照できないが、参照時エラーとならず、不正なコードを出力します。

(例1)

```
char  *c ;
int  *i ;
void  ( * f1 ( int    * ) ) ( char * ) ;
void  ( * f2 ( void     ) ) ( char * ) ;
void  ( * f3 ( int    * ) ) ( void    ) ;
void  main ( ) {
      ( * f1 ( i ) ) ( c ) ;          /* 正しい記述     (W510) */
      ( * f1 ( i ) ) ( i ) ;          /* 間違った記述   */
      ( * f2 (   ) ) ( c ) ;          /* 正しい記述     (W509) */
      ( * f2 (   ) ) (   ) ;          /* 間違った記述   (W509) */
      ( * f3 ( i ) ) (   ) ;          /* 正しい記述     (W509) */
      ( * f3 ( i ) ) ( i ) ;          /* 間違った記述   */
}
```

正しい記述に対して、ワーニング・メッセージ W509/W510 を出力してしまいます。逆に、間違った記述に対してエラー・メッセージを出力しません。ただし、出力コードは正常です。

(例2)

```
void  ( * f4 ( ) ) ( int p ) {
     p++ ;                            /* 間違った記述 */
}
```

エラーにすべき記述に対してエラーを出力せず、不正コードを生成します。

【回避策】ありません。

【改善策】制限事項とします。

No. 21 signed 型のビット・フィールドを符号無しのビット・フィールドとして処理する制限

【内　容】signed 型のビット・フィールドを、符号無しのビット・フィールドとして処理します。

【回避策】ありません。

【改善策】制限事項とします。

No. 22 ラージ・モデルにおいて，1 関数内の auto 変数の総サイズが 65536 バイトを越えると，出力コードやディバグ情報が不正となる制限

【内　容】ラージ・モデルにおいて、1 関数内の auto 変数の総サイズが 65536 バイトを越えると、出力コードやディバグ情報が不正となります。

(例) -ML 指定時

```
void func(long a, int b){
      int i;
      char tab1[35000];
      char tab2[35000];
      i = b;
}
```

【回避策】1 関数内の auto 変数の総サイズを 65535 バイト以下にしてください。

【改善策】制限事項とします。

No. 23 signed char 型と定数値の演算がコード不正となる制限

【内　容】-QC1 指定時に、条件(1)～(5)を少なくとも 1 つ満たすか、-QC1 または-QC2 指定時に、条件(6)を満たす場合に、不正なコードが出力される可能性があります。

(1) 右シフト演算子>>を使っている場合、左演算数が定数で、128～255 の定数を使い、右演算数が signed char 型である。

(2) 演算子/、%、<、<=、>、>=、/=、%=を使っている場合、一方の演算数が定数で、128～255 の定数を使い、もう一方の演算数が signed char 型である。

(3) 一方の演算数が定数で、128～255 の定数を使い、もう一方の演算数が signed char 型である二項演算の演算結果を 2 バイト幅以上の型に型変換している。

(4) 一方の演算数が定数で、128～255 の定数を使い、もう一方の演算数が signed char 型である二項演算の演算結果を、右演算数が signed char 型である右シフト演算子>>の左演算数に使っている。

(5) 一方の演算数が定数で、128～255 の定数を使い、もう一方の演算数が signed char 型である二項演算の演算結果を、一方の演算数が signed char 型である演算子/、%、<、<=、>、>=、/=、%=のもう一方の演算数に使っている。

(6) 二項演算で、一方の演算数が、演算子<<、>>、&、^、|を少なくとも 1 つ使った定数で、定数の値が-128～255 で、その演算結果が-128～-1 で、もう一方の演算数が 2 バイト幅以上である。

（例1）

```
signed char a;
if(a < 179/2){b++;}
```

（例2）

```
int i, j;
i = j & (-127 | 4);
```

【回避策】該当する部分の定数を signed int 型にキャストしてください。

（例1）

```
if(a < (signed int)179/2){b++;}
```

（例2）

```
i = j & ((signed int)-127 | 4);;
```

【改善策】Ver.2.41 で修正いたします。
また、本制限に該当するか否かをチェックするツールを用意しております。
ツールに関しましては、弊社営業または特約店にお問い合わせください。

No. 24 複合代入%=, /=の演算がコード不正となる制限

【内　容】次の条件を全て満たす場合に、符号なし除算、剰余の演算をするべきところで、符号あり演算のコードを出力してしまいます。

(1) 複合代入 %=, /= がある

(2) (1)の式の左辺が、unsigned short 型である

(3) (1)の式の右辺が、(signed) int 型である
定数の場合には、接尾語 u,U,l,L がついていなく、かつ、-32768～32767 の範囲内のときのみ該当します。

（例）

```
unsigned short a;
a /= 0x5555;
```

【回避策】複合代入"%=","/="ではなく、単純代入"="にしてください。

（例）

```
a = a / 0x5555;
```

【改善策】Ver.2.41 で修正いたします。
また、本制限に該当するか否かをチェックするツールを用意しております。
ツールに関しましては、弊社営業または特約店にお問い合わせください。

No. 25 自動変数の配列名を参照で W503 を出力する制限

【内　容】初期化なしの自動変数の配列に対して、式中で配列名を参照すると、W503 を出力してしまいます。

W503 Possible use of '変数名' before definition

注意:

「初期化」とは、int a[2]={0,0}; のような 宣言時の初期値のことです。a[0] = 0; a[1] = 0; のような代入式は含みません。

「配列名」とは、int a[2] の'a'自体のことです。a[0]、a[1]、&a[0]などは含みません。

（例）

```
void func(void)
{
    int a[2];
    int *b;

    a[0] = 0;
    a[1] = 0;
    b = a;                  /* この行で W503 が発生 */
}
```

【回避策】W503 が出力された場合には、該当箇所を確認していただき、文中で初期化されてした場合には、無視してください。

【改善策】制限事項とします。

No. 26 静的変数を浮動小数点定数で初期化すると初期値が不正になる制限

【内　容】下記以外の変数を浮動小数点定数で初期化すると、初期値が不正となります。

・long/unsigned long/浮動小数点型変数
・構造体/共用体のlong/unsigned long/浮動小数点型メンバ
・配列のlong/unsigned long/浮動小数点型要素

（例 1） signed char a1[3] = {1.0*100};

（例 2） #define VAR 1.0*100
signed char frame02 = VAR;

【回避策】以下のいずれかの方法で回避してください。

(1) 整数定数で初期化する。

上記（例 1）では以下のようにしてください。

```
signed char a1[3] = {100};
```

(2) 浮動小数点定数を適切な整数型にキャストする。

上記（例 1）では以下のようにしてください。

```
signed char a1[3] = {(signed char)(1.0*100)};
```

上記（例2）では以下のようにしてください。

```
#define VAR 1.0*100
signed char frame02 = (signed char)(VAR);
```

【改善策】V2.41 で修正いたします。
また、本制限に該当するか否かをチェックするツールを用意しております。
ツールに関しましては、弊社営業または特約店にお問い合わせください。

3. その他注意事項

特にありません。

以上