マイクロコントローラ技術情報

技術通知 78K0 C コンパイラ CC78K0 使用制限事項の件	発行番号	ZMT－F35－11－0001号	1/2
	発行日	2011年 4月 7日	
	発行部門	ルネサス エレクトロニクス株式会社 MCU事業本部 ソフトウェア統括部 MCUツール技術部	

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		
	CC78K0 Ver.3.70 言語編	資料番号：U17200JJ1V0(第 1 版)
	CC78K0 Ver.3.70 操作編	資料番号：U17201JJ1V0(第 1 版)
	78K0 C コンパイラ CC78K0 Ver.4.10 使用上の留意点	資料番号：ZUD-CD-10-0202

1. 対象製品

CC78K0　　V4.10 以前

※詳細は「別紙 1 CC78K0 の使用制限事項一覧」を参照ください

2. 新たな制限事項

今回新たに制限事項 No.75～No.76 を追加しました。詳細は、別紙1を参照してください。

・No.75　##演算子を使ったマクロ展開でエラーとなる制限
・No.76　アセンブラ・ソースに割り込み関数のシンボル情報が出ない制限

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。

・No.75　以下のいずれかで回避してください。
　・## 演算子の直前のトークンと直後のトークンを連結して、別のトークンを生成しない時は、
　## 演算子を使わないでください。
　・## 演算子の直前のトークンと直後のトークンを連結して、別のトークンを生成する時は、
　## 演算子の直後に関数形式マクロのパラメータを置いてください。
・No.76　割り込み関数を定義する、あるいはオブジェクト・モジュール・ファイルを出力してください。

4. 改善計画

今回新たに追加した No.75～No.76 は、次版で修正いたします。

5. 制限事項一覧

制限事項の履歴とその詳細情報を使用制限事項一覧として別紙1に記載します。

6. 発行文書履歴

78K0　C コンパイラ CC78K0 使用制限事項一覧　発行文書履歴

文書番号	発行日	記事
SBG-DT-03-0307	2003.12.17	初版
SBG-DT-04-0110	2004.3.12	制限事項追加（No.17、No.18、No.19）
ZBG-CD-04-0072	2004.10.4	制限事項追加（No.20 ～ No.24）
ZBG-CD-05-0001	2005.1.12	制限事項追加（No.25）
ZBG-CD-05-0054	2005.8.31	・制限事項 No.21 の制限の条件を追加 ・制限事項追加（No.26 ～ No.54）
ZBG-CD-07-0038	2007.6.28	制限事項追加（No.55 ～ No.64）
ZBG-CD-10-0024	2010.8.16	制限事項追加（No.65 ～ No.74）
ZMT-F35-11-0001	2011.4.7	制限事項追加（No.75 ～ No.76）

以上

CC78K0 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				
		3.50	3.60	3.70	4.00	4.10
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	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る制限	○	○	○	○	○
3	#if の定数式を正しく処理しない場合がある制限	○	○	○	○	○
4	scanf/sscanf で浮動小数点型→整数型の順に値を読むと整数型の値を正しく入力できない制限	○	○	○	○	○
5	構造体ポインタをレジスタに割り当てポインタが指すメンバ間でデータを受け渡すとコード不正となる制限	○	○	○	○	○
6	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	○	○	○	○	○
7	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	○	○	○	○	○
8	関数のパラメータが register 宣言した配列の場合にコード不正となる制限	○	○	○	○	○
9	#pragma section 使用時に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	○	○	○	○	○
10	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	○	○	○	○	○
11	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・ソース中のデバッグ情報が不正となる制限	×	×	×	○	○
12	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	×	×	×	×	○
13	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制限	×	×	×	○	○
14	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	×	×	×	×	○
15	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	×	×	○	○	○
16	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	×	×	×	×	×
17	古いプロジェクトのスタートアップ・ルーチン、標準ライブラリの情報が引き継がない制限	×	○	○	○	○
18	左右演算数を構造体／共用体／配列へのポインタにキャストした単純代入演算式がコード不正となる制限	× ※	○	○	○	○
19	#pragma inline で memcpy 関数がコード不正となる制限	×	○	○	○	○
20	signed char 型と定数値の演算がコード不正となる制限	×	× ※	○	○	○

×：該当する
○：該当しない
ー：対象外
※：チェックツールあり

No.	仕様変更・追加／不具合事項	バージョン				
		3.50	3.60	3.70	4.00	4.10
21	(signed) int 型と unsigned short 型の演算結果がコード不正となる制限	×	× ※	○	○	○
22	条件式、単純代入演算式連続のデバッグ情報位置が不正となる制限	×	×	○	○	○
23	cprep2 関数のスタック情報が不正となる制限	×	×	○	○	○
24	自動変数の配列名を参照で W0503 を出力する制限	×	×	×	×	○
25	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となる制限	×	× ※	○	○	○
26	関数呼び出し以外で関数ポインタを"*"付きで参照するとコード不正となる制限	×	× ※	○	○	○
27	-QR 指定時、コンパイラが使用する saddr 領域に sreg 変数が重複して割り当たる制限	×	× ※	○	○	○
28	シフト演算(<<,>>,<<=,>>=)のシフト数が 256 以上の定数の時にコード不正となる制限	×	× ※	○	○	○
29	long/unsigned long型の変数に定数0xffffまたは0xfffeを加算/減算するとコード不正となる制限	×	× ※	○	○	○
30	0eまたは0Eから始まる浮動小数点数を記述すると、エラーとなる制限	×	×	○	○	○
31	long型ランタイムライブラリの演算結果をint/unsigned int/short/unsigned short型にキャストした式で、コード不正となる制限	×	× ※	○	○	○
32	二項演算の両方の演算数のうち、一方の演算結果がコード不正となる制限	×	× ※	○	○	○
33	論理演算(&&、||)の両方の演算数が浮動小数点型の時、第二演算数にインクリメント/デクリメント演算あるいは関数呼び出しで、比較順序が不正となる制限	×	× ※	○	○	○
34	#ifの定数式に、単項+,-演算子から始まる式、または同じ優先度の演算子が連続した式を記述するとエラーとなる制限	×	×	○	○	○
35	関係演算の一方の演算数がsigned long型で表現できない定数の時、比較結果が不正となる制限	×	× ※	○	○	○
36	論理否定演算、関係演算、等価演算の演算数の型によって、演算結果がint型とならず、演算数の型となる制限	×	× ※	○	○	○
37	浮動小数点型のインクリメント/デクリメント式を記述し、演算数がポインタによる間接参照式の時に、エラーとなる制限	×	×	○	○	○
38	“char/unsigned char型配列のアドレス＋unsigned char型の式”をint/unsigned int/short/unsigned shortへのポインタにキャストし、そのポインタが指す内容に定数0を代入する式で、コード不正となる制限	×	× ※	○	○	○
39	ポインタを返す関数のreturn文に、char/unsigned char型の式を記述するとコード不正となる制限	×	× ※	○	○	○

×：該当する
○：該当しない
ー：対象外
※：チェックツールあり

No.	仕様変更・追加／不具合事項	バージョン				
		3.50	3.60	3.70	4.00	4.10
40	long型ランタイムライブラリの演算結果をchar/unsigned char型にキャストし、char/unsigned char型で表現可能な定数と関係/等価演算すると、エラーとなる制限	×	×	○	○	○
41	大きさが定義されていない配列の初期化で、初期化子の中括弧の囲みが不統一な場合、確保される領域のサイズが不正となる制限	×	×	×	×	○
42	文字列変換関数の標準ライブラリのエラー処理動作が不正となる制限	×	×	×	○	○
43	入出力関数の標準ライブラリの出力変換で、動作が不正となる制限	×	×	×	×	×
44	int/short型の最小値-32768のサイズが4となる制限	×	×	×	×	×
45	条件演算の第2,3項に関数名または関数ポインタを記述して関数を呼び出すと、エラーとなる制限	×	×	×	×	○
46	外部ポインタ変数を演算子->を含む変数で初期化するとエラーとなる制限	×	×	○	○	○
47	関数定義の識別子の型に対して、仮引数型と関数定義の識別子の型とが合わずに、エラーとなる制限	×	×	×	×	×
48	関数定義の識別子並びで、宣言していない仮引数をint型とせずエラーとなる制限	×	×	×	×	×
49	#演算子が正しく展開できない制限	×	×	×	×	×
50	unsigned long型の静的変数を、0x80000000以上の浮動小数点定数で初期化すると、初期値が不正となる制限	×	× ※	○	○	○
51	関数呼び出しと構造体/共用体を含む式で、コード不正となる制限	×	× ※	○	○	○
52	「a = b 二項演算子 c;」の代入式で、コード不正となる制限	×	× ※	○	○	○
53	ドット演算子[.]を使って、定数番地構造体の配列メンバの要素を参照するとエラーとなる制限	×	×	○	○	○
54	特定のパターンの引数を持つ関数定義で、エラーとなる制限	×	×	○	○	○
55	複合代入／インクリメント／デクリメント演算の代入先にunsigned char型を返す関数の返り値を配列添字に使った要素が1バイトの静的配列の要素を記述すると、コード不正となる場合がある制限	×	× ※	× ※	○	○
56	桁あふれする二項演算の結果を使う、2 つ以上の二項演算子を含む定数式を不正な値に置き換える場合がある制限	×	× ※	× ※	○	○
57	インクリメント/デクリメント演算を含んだ演算で、コード不正となる場合がある制限	×	× ※	× ※	○	○
58	配列型を持つ関数パラメータのsizeof演算結果が不正となる制限	×	× ※	× ※	○	○
59	-qcオプション未指定時に、除算・剰余算の複合代入演算がコード不正となる場合がある制限	×	× ※	× ※	○	○
60	saddr領域に割り当てたビット幅が2ビット以上7ビット以下のビット・フィールドに、ビット・フィールドの最大定数値を代入後、同じ式で代入先を再評価するとコード不正となる制限	×	× ※	× ※	○	○

×：該当する
○：該当しない
ー：対象外
※：チェックツールあり

No.	仕様変更・追加／不具合事項	バージョン				
		3.50	3.60	3.70	4.00	4.10
61	enum型のパラメータを持つnorec関数を呼ぶと、エラーE0705を出力、あるいはコード不正となる場合がある制限	×	× ※	× ※	○	○
62	異なる構造体/共用体型をパラメータに持つ同一関数の宣言が、エラーとならない制限	×	× ※	× ※	○	○
63	サイズが256byte以上の構造体を指すポインタがレジスタ変数の時に、コード不正となる場合がある制限	×	× ※	× ※	○	○
64	スタティック・モデル・オプション”-sm”および拡張仕様オプション”-zm2”指定時に、コンパイラが使用するワークエリアを壊す制限	×	×	×	○	○
65	後置++/--ポインタが指す 1 バイトデータ参照時にコード不正となる制限	×	× ※	× ※	× ※	○
66	char/signed char/unsigned char 型配列の初期化子並びの最後が文字列で、文字列の前に 1 個以上の定数または文字定数が並ぶ時にエラーとならず、コード不正となる制限	×	× ※	× ※	× ※	○
67	ポインタ同士の減算結果をオフセットとしたポインタ参照時に、コード不正となる制限	×	× ※	× ※	× ※	○
68	-qcオプションが未指定（int型拡張する）の時、コード不正となる制限	○	○	× ※	× ※	○
69	BCD演算関数”adbcdw”, ”sbbcdw”のコード不正となる制限	×	×	×	×	○
70	-ngオプションを指定し、ASM文を含む関数において、分岐命令でエラーとなる制限	×	×	×	×	○
71	ネストしたif文を抜けた直後の文に対して、行番号情報が出ない制限	×	×	×	×	○
72	割り込み関数内のlong型変数への代入でコード不正となる場合がある制限	×	× ※	× ※	× ※	○
73	バンク関数呼び出しコードが出ない場合がある制限	－	－	× ※	× ※	○
74	norec関数にて1バイトの引数またはauto変数使用時に、コード不正となる場合がある制限	×	× ※	× ※	× ※	○
75	##演算子を使ったマクロ展開でエラーとなる制限	×	×	×	×	×
76	アセンブラ・ソースに割り込み関数のシンボル情報が出ない制限	×	×	×	×	×

×：該当する
○：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】V3.50 で修正済みです。

No. 2 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】V3.50 で修正済みです。

No. 3 #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】V3.50 で修正済みです。

No. 4 scanf/sscanf で浮動小数点型→整数型の順に値を読むと整数型の値を正しく入力できない制限

【内　容】scanf/sscanf で浮動小数点型→整数型の順に値を読むと、整数型の値を正しく入力できません。

(例)

```
void func()
{
    int i;
    float f;

    sscanf("1.2 10", "%f%d", &f, &i);
}
```

【回避策】ありません。

【改善策】V3.50 で修正済みです。

No. 5 構造体ポインタをレジスタに割り当てポインタが指すメンバ間でデータを受け渡すとコード不正となる制限

【内　容】構造体ポインタをレジスタに割り当て、ポインタが指すメンバ間でデータを受け渡すと、コード不正となります。

(例)

```
struct tag {
    int a;
    int b;
};
void func()
{
    register struct tag *sp;
    sp->b = sp->a;
}
```

【回避策】構造体ポインタをレジスタに割り当てないようにしてください。

【改善策】V3.50 で修正済みです。

No. 6 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】V3.50 で修正済みです。

No. 7 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】V3.50 で修正済みです。

No. 8 関数のパラメータが 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】V3.50 で修正済みです。

No. 9 #pragma section 使用時に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)

```
--- a.c ---
#include "a1.h"
#include "a2.h"
#include "a3.h"

--- a1.h ---
#pragma section @@DATA DAT1
int a;
#pragma section @@DATA DAT2

--- a2.h ---
#pragma section @@DATA DAT3
int b;
#pragma section @@DATA DAT4

--- a3.h ---
#pragma section @@DATA DAT5
extern int a;   /* int a; 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】V3.50 で修正済みです。

No. 10 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】V3.50 で修正済みです。

No. 11 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・ソース中のデバッ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バッグ情報は正常ですが，アセンブラ・ソース中のデバッグ情報が不正となります。

（例）

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

【回避策】ありません。

【改善策】V4.00 で修正済みです。

No. 12 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】V4.10 で修正しました。

No. 13 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 ) ;                  /* E0713 Redefined 'func' */
```
(例2)
```
typedef  int  VTYPE[2] ;
typedef  int  *VPTYPE[3] ;

const  VTYPE  *a ;
const  int  ( *a )[2] ;               /* E0713 Redefined 'a' */
volatile  VPTYPE  b[2] ;
volatile  int  *volatile  b[2][3] ;   /* E0713 Redefined 'b' */
```

【回避策】ありません。
【改善策】V4.00 で修正済みです。

No. 14 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(E0756) */
```

【回避策】多次元配列の大きさを定義してください。
【改善策】V4.10 で修正しました。

No. 15 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      (W0510) */
      ( * f1 ( i ) ) ( i ) ;          /* 間違った記述   */
      ( * f2 (   ) ) ( c ) ;          /* 正しい記述      (W0509) */
      ( * f2 (   ) ) (   ) ;          /* 間違った記述   (W0509) */
      ( * f3 ( i ) ) (   ) ;          /* 正しい記述      (W0509) */
      ( * 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 W0509/W0510 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】V3.70 で修正済みです。

No. 16 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. 17 古いプロジェクトのスタートアップ・ルーチン、標準ライブラリの情報を引き継がない制限

【内　容】古いバージョンのプロジェクト・ファイルを読み込んだ時、スタートアップ・ルーチン、標準ライブラリの情報を引き継がない場合があります。

【回避策】スタートアップ・ルーチン、標準ライブラリを再設定してください。

【改善策】V3.60 で修正済みです。

No. 18 左右演算数を構造体／共用体／配列へのポインタにキャストした単純代入演算式がコード不正となる制限

【内　容】次の 4 つの条件をすべて満たす場合に、不正なコードを出力します。

(1) 単純代入演算式

(2) 左右演算数が、どちらも構造体、共用体、配列、ポインタを使用した間接参照データ

(3) 左右演算数が、どちらもシンボルのアドレスを参照

(4) 演算対象のデータが共に 4byte データ(long/float/double/long double 型)

(例)

```
typedef union {
    unsigned long l;
} uni;
unsigned int a[10], b[10];

void func(void)
{
    ((uni*)(&b)) -> l = ((uni*)(&a)) -> l;
}
```

【回避策】テンポラリ変数を介して処理するように変更してください。

(例1)

```
uni *tmp1, *tmp2;
tmp1 = (uni*)&a;
tmp2 = (uni*)&b;
tmp2->l = tmp1->l;
```

(例2)

```
long tmp3;
tmp3 = ((uni*)(&a))->l;
((uni*)(&b))->l = tmp3;
```

【改善策】V3.60 で修正済みです。

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. 19 #pragma inline で memcpy 関数がコード不正となる制限

【内　容】次の条件をすべて満たす時、不正なコードを出力する場合があります。

(1) #pragma inline を指定している

(2) memcpy 関数が存在する

(3) (2)の memcpy 関数の第 3 引数が定数以外

(第 3 引数が HL レジスタを使用するコードの場合)

(例)

```
#pragma inline
int s[100];
void func(void)
{
    int *t;
    int u;
    memcpy( s, t, u );
}
```

【回避策】次のいずれかの方法で回避してください。

(1) memcpy 関数を使用する場合は、#pragma inline を指定しない。

(2) #pragma inline を指定する場合は、memcpy 関数の第 3 引数に定数を記述する。

【改善策】V3.60 で修正済みです。

No. 20 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】V3.70 で修正済みです。
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. 21 (signed) int 型と unsigned short 型の演算結果がコード不正となる制限

【内　容】次のいずれかの条件を満たす場合に、符号なし演算をするべきところで、符号あり演算をして演算結果が不正となる場合があります。

(条件1)関係演算(<,>,<=,>=)、除算(/)、剰余算(%)、複合代入演算(%=,/=)において、一方が(signed) int 型、もう一方が unsigned short 型である。

(条件1の例1)

```
unsigned short us1, us2;
void func1()
{
    us1 /= 0x5555;
    us2 %= 0x5555;
}
```

（条件1の例 2）

```
unsigned short us1, us2;
signed int si1;
void func2()
{
    us2 = us1 / si1;
}
```

（条件1の例 3）

```
int si, x;
unsigned short us;
void func3()
{
    if (si > us) x++;
}
```

（条件2）複合代入演算(%=,/=)または二項演算において、一方が(signed) int 型、もう一方が unsigned short 型の時に、演算結果を 2 バイト幅を超える型に型変換している。

（条件2の例1）

```
long l;
unsigned short us;
signed int si;
void func4()
{
    l = us + si;
}
```

【回避策】unsigned short を、unsigned int に変更してください。

【改善策】V3.70 で修正済みです。
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. 22 条件式、単純代入演算式連続のデバッグ情報位置が不正となる制限

【内　容】saddr 領域に割り当たらない静的な 1 バイト変数を条件式で参照し、直後の単純代入演算式で同じ 1 バイト変数に定数を代入した場合に、デバッグ情報の出力位置が不正となります。
デバッグ情報だけで、出力コードには影響ありません。

(例)

C ソース記述	対応する出力アセンブラ

if(TEST >= 10)
 	
{
　TEST = 0;
 	
}
	
	　movw de,#_TEST
	　mov a,[de]
	　cmp a,#0AH ; 10
	
	　bc $?L0003　;(1)
	　callt [@@clra0]
	　mov [de],a
	　br $?L0003

← 代入文に比較の命令が対応してしまっています。

ディバッガで「TEST = 0;」にブレークポイントを設定しても上記の(1)でブレークしてしまいます。つまり、if 文の条件式が真の場合ではなく、条件式の部分でブレークしてしまいます。

【回避策】該当箇所については、ディバッガのソースウインドウでの混合表示にて、アセンブラコードにブレークポイントを設定してください。

【改善策】V3.70 で修正済みです。

No. 23 cprep2 関数のスタック情報が不正となる制限

【内　容】コンパイラが出力する関数情報では、ランタイム・ライブラリを含む関数のスタック情報にランタイム・ライブラリのスタック消費量が加算されています。
しかし、cprep2 関数がある場合には、2byte 余計に加算されてしまいます。

【回避策】ありません。cprep2 関数がある場合には、スタック消費量より 2byte 減算して計算してください。

【改善策】V3.70 で修正済みです。

No. 24 自動変数の配列名を参照で W0503 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、W0503 を出力してしまいます。

W0503 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 W0503 が発生 */
}
```

【回避策】W0503 が出力された場合には、該当箇所を確認していただき、文中で初期化していた場合には、無視してください。

【改善策】V4.10 で修正しました。

No. 25 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となる制限

【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};

（例 2） #define VAR 1.0
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)};
```

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】V3.70 で修正済みです。
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. 26 関数呼び出し以外で関数ポインタを"*"付きで参照するとコード不正となる制限

【内　容】関数呼び出し以外で関数ポインタを"*"付きで参照するとコード不正となります。

（例）

```
void (*fp)();
int x;
void func()
{
    if (*fp)   x++;
}
```

【回避策】関数ポインタに*を付けないでください。

```
if (*fp)   x++;
      ↓
if (fp)    x++;
```

【改善策】V3.70 で修正済みです。
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. 27 -QR 指定時、コンパイラが使用する saddr 領域に sreg 変数が重複して割り当たる制限

【内　容】-QR 指定時、コンパイラが使用する saddr 領域に sreg 変数が重複して割り当たる場合があります。

（例）

```
#pragma interrupt INTP0 inter
_boolean b1;
void func()
{
    b1 = 1;
}
void inter()
{
    func();
}

--- 別ファイル ---
_sreg char sc[152];
```

【回避策】-QR を指定しないでください。

【改善策】V3.70 で修正済みです。
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. 28 シフト演算(<<,>>,<<=,>>=)のシフト数が 256 以上の定数の時にコード不正となる制限

【内　容】シフト演算(<<,>>,<<=,>>=)のシフト数が 256 以上の定数の時にコード不正となる場合があります。

（例）

```
int i1, i2;
char c1, c2;
void func()
{
    i1 = i2 << 257;
    i2 <<= 257;
    c1 = c2 << 257;
    c2 <<= 257;
}
```

【回避策】シフト数をデータ幅と同じにしてください。

【改善策】V3.70 で修正済みです。
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. 29 long/unsigned long 型の変数に定数 0xffff または 0xfffe を加算/減算するとコード不正となる制限

【内　容】long/unsigned long 型の変数に定数 0xffff または 0xfffe を加算/減算するとコード不正となる場合があります。

（例）

```
unsigned long l1, l2;
void func()
{
    l1 = l2 + 0xffff;
}
```

【回避策】以下の記述のようにテンポラリ変数を介してください。

```
long ltmp = 0xffff;
l1 = l2 + ltmp;
```

【改善策】V3.70 で修正済みです。
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. 30 0e または 0E から始まる浮動小数点数を記述すると、エラーとなる制限

【内　容】0e または 0E から始まる浮動小数点数を記述すると、E0312 エラーとなります。

（例）

```
float f1;
void func()
{
    f1 = 0e0;
}
```

【回避策】0 または 0.0 と記述してください。
【改善策】V3.70 で修正済みです。

No. 31 long 型ランタイムライブラリの演算結果を int/unsigned int/short/unsigned short 型にキャストした式で、コード不正となる制限

【内　容】次の条件をすべて満たす時に、コード不正となる場合があります。

(1) ノーマルモデル指定時

(2) 2バイトの等価演算(==,!=)、2バイトの符号なし関係演算(<, >=)で、左演算数が"引数/自動変数/ポインタの参照先"、または 2 バイトの符号なし関係演算(<=,>)で、右演算数が"引数/自動変数/ポインタの参照先"

(3) もう一方の演算数が、long 型ランタイムライブラリの演算結果を signed int/unsigned int /signed short/unsigned short 型にキャストした式

(例)

```
int *ip;
long l;
void func()
{
   if (*ip == (int)(l * 5)) l = 0;
}
```

【回避策】次のいずれかの方法で回避してください。

(1) キャストしないでください。

```
if (*ip == (l * 5)) l = 0;
```

(2) テンポラリ変数を介してください。

```
int tmp;
tmp = l * 5;
if (*ip == tmp) l = 0;
```

【改善策】V3.70 で修正済みです。
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. 32 二項演算の両方の演算数のうち、一方の演算結果がコード不正となる制限

【内　容】次のいずれかの条件を満たす時に、論理演算、条件演算のコード実行中に退避した演算結果を正常に復帰できずに、コード不正となる場合があります。

(条件1) 次の条件をすべて満たす場合

(1) 二項演算がある

(2) (1)の一方の演算数が論理演算(&&, ||)、条件演算(? :)の演算結果を参照

(3) (1)のもう一方の演算数の演算結果がレジスタに残っている

(4) 条件演算の場合は、ノーマルモデルで leaf 関数のみ

(条件1の例)

```
int func1(), func2();
int x, i;
void func()
{
   if (func1() == (i && func2())) x++;
} /*  (3)   (1)      (2)                上記番号に対応 */
```

（条件2） 次の条件をすべて満たす場合
(1) 二項演算がある
(2) (1)の一方の演算数が論理演算(&&, ||)の演算結果を参照
(3) (1)のもう一方の演算数の演算結果が_@RTARGx（ランタイムライブラリの引数）に残っている

（条件2の例）

```
float f1, f2;
long l;
int x;
void func()
{
    if (++f1 == (l && f2)) x++;
} /*  (3)   (1)  (2)              上記番号に対応 */
```

（条件3）次の条件をすべて満たす場合
(1) 二項演算がある
(2) (1)の一方の演算数が複合代入演算の演算結果を参照
(3) (1)のもう一方の演算数の演算結果がレジスタに残っている

（条件3例）

```
int ifunc(), i;
void func()
{
    int *ip1, *ip2;
    i = *ip1 == (*ip2 += ifunc());
} /*    (3)  (1)      (2)            上記番号に対応 */
```

【回避策】以下のいずれかの方法で回避してください。
（条件1の回避策）二項演算の演算数の内、少なくとも1つをテンポラリ変数を介して演算してください。

```
int tmp;
tmp = func1();                  /* テンポラリ変数に代入 */
if (tmp == (i && func2())) x++;
```

（条件2の回避策）二項演算の演算数の内、少なくとも1つをテンポラリ変数を介して演算してください。

```
float tmp;
tmp = ++f1;                    /* テンポラリ変数に代入 */
if (tmp == (l && f2)) x++;
```

（条件3の回避策）二項演算の演算数の内、少なくとも1つをテンポラリ変数を介して演算してください。

```
int tmp;
tmp = *ip1;                     /* テンポラリ変数に代入 */
i = tmp == (*ip2 += ifunc());
```

【改善策】V3.70 で修正済みです。
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. 33 論理演算(&&、||)の両方の演算数が浮動小数点型の時、第二演算数にインクリメント/デクリメント演算あるいは関数呼び出しで、比較順序が不正となる制限

【内　容】論理演算(&&、||)の両方の演算数が浮動小数点型の時、第二演算数にインクリメント/デクリメント演算あるいは関数呼び出しのような副作用のある式を記述すると、比較順序が不正となります。

(例)

```
int x;
float f1, f2;
void func()
{
    if (f1 || f2++) {
        x = 1;
    }
    else {
        x = 2;
    }
}
```

※(不正な動作) "if(f1 || f2++)"で、f1の真、偽に関わらずf2++を実行します
(正しい動作) "if(f1 || f2++)"で、f1が偽の時のみf2++を実行します

【回避策】以下のように記述を変更してください。

```
if (f1) {
    x = 1;
}
else if (f2++) {
    x = 1;
}
else {
    x = 2;
}
```

【改善策】V3.70 で修正済みです。
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. 34 #if の定数式に、単項+,−演算子から始まる式、または同じ優先度の演算子が連続した式を記述するとエラーとなる制限

【内　容】 #if の定数式に、単項+,−演算子から始まる式、または同じ優先度の演算子が連続した式を記述すると E0501 エラーとなります。

(例)

```
#if !~0
int i;
#endif
#if -1
int j;
#endif
```

【回避策】括弧を付けてください。

```
#if !~0          #if -1
   ↓               ↓
#if !(~0)        #if (-1)
```

【改善策】V3.70 で修正済みです。

No. 35 関係演算の一方の演算数が signed long 型で表現できない定数の時、比較結果が不正となる制限

【内　容】 関係演算の一方の演算数が signed long 型で表現できない定数の時、次のいずれかの条件を満たす時にコード不正となる場合があります。

（条件1） もう一方の演算数が定数とみなすことのできる式で、signed long 型で表現できる値である

（条件1の例）

```
int x1,x2,x3,x4,x5,x6, i;
void func1()
{
    x1 = (i << 31) < 0x80000001;
    x2 = (i * 0) > 0x90000020;
    x3 = (i % 1) <= 0xa0000300;
    x4 = (i & 0) >= 0xb0004000;
    x5 = (i | 0xffff) < 0xc0050000;
    x6 = (i++, 300) > 0xffffffff;
}
```

（条件2） 一方の演算数が0xffffff80以上の定数で、もう一方の演算数がboolean型,bit型もしくはsigned char型である

（条件2の例）

```
int x1, x2, x3, i;
char cfunc(), c1, c2;
void func2()
{
    x1 = c1 < 0xffffff80;
    x2 = cfunc() > 0xffffff91;
    x3 = (c1 + c2) <= 0xffffffa2;
}
```

【回避策】

（条件1の回避策）

定数とみなすことのできる式を、定数で記述してください。

```
x1 = (i << 31) < 0x80000001;   →   x1 = 0 < 0x80000001;
x2 = (i * 0) > 0x90000020;     →   x2 = 0 > 0x90000020;
x3 = (i % 1) <= 0xa0000300;    →   x3 = 0 <= 0xa0000300;
x4 = (i & 0) >= 0xb0004000;    →   x4 = 0 >= 0xb0004000;
x5 = (i | 0xffff) < 0xc0050000; →  x5 = 0xffff < 0xc0050000;

x6 = (i++, 300) > 0xffffffff;  →   i++;
                                   x6 = 300 > 0xd0600000;
```

（条件2の回避）

boolean型, bit型もしくはsigned char型の式を、unsigned long型にキャストしてください。

```
x1 = c1 < 0xffffff80;          →  x1 = (unsigned long)c1 < 0xffffff80;
x2 = cfunc() > 0xffffff91;     →  x2 = (unsigned long)cfunc() > 0xffffff91;
x3 = (c1 + c2) <= 0xfffffffa2; →  x3 = (unsigned long)(c1 + c2) <= 0xfffffffa2;
```

【改善策】V3.70 で修正済みです。
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. 36 論理否定演算、関係演算、等価演算の演算数の型によって、演算結果が int 型とならず、演算数の型となる制限

【内　容】以下のいずれかで不正となります。

(1)論理否定演算、関係演算、等価演算の演算数の型が以下の時、演算結果がint型とならず、演算数の型となる。

・unsigned int、unsigned short、ポインタ(*1)、配列
・-QC 指定時の unsigned char

(*1) 2 バイト・サイズのポインタのみ。

(2)論理演算&&の右演算数が浮動小数点定数±0.0 の時、演算結果が int 型とならず、浮動小数点型となる。(スタティック・モデルを除く)

（例）

```
unsigned int ui1, ui2;
int xi1, xi2, xi3, i1;
void func()
{
    xi1 = !ui1 > i1;
    xi2 = (ui1 == ui2) > i1;
    xi3 = (ui1 && 0.0) > i1;
}
```

【回避策】(1) 論理否定演算、関係演算、等価演算の結果を int 型でキャストしてください。

```
xi1 = !ui1 > i1;
xi2 = (ui1 == ui2) > i1;
        ↓
xi1 = (int)!ui1 > i1;
xi2 = (int)(ui1 == ui2) > i1;
```

(2) 論理演算＆＆の右演算数に浮動小数点定数±0.0 を使わないでください。

```
xi3 = (ui1 && 0.0) > i1;
        ↓
xi3 = (ui1 && 0) > i1;
```

【改善策】V3.70 で修正済みです。
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. 37　浮動小数点型のインクリメント/デクリメント式を記述し、演算数がポインタによる間接参照式の時に、エラーとなる制限

【内　容】　4 バイト・データの演算結果が_@RTARG0 と_@RTARG4 に残っている状態で、浮動小数点型のインクリメント/デクリメント式を記述し、演算数がポインタによる間接参照式の時に、E0105 エラーとなる場合があります。

（例）

```
float *pf1;
int x;
void func()
{
        long l1, l2, l3, l4;
        x = (l1 & l2) < ((l3 - l4) + (*pf1)++);
}
```

【回避策】以下の記述のようにテンポラリ変数を介してください。

```
float tmp;
tmp = (*pf1)++;
x = (l1 & l2) < ((l3 - l4) + tmp);
```

【改善策】V3.70 で修正済みです。

No. 38 “char/unsigned char 型配列のアドレス＋unsigned char 型の式”を int/unsigned int/short/ unsigned short へのポインタにキャストし、そのポインタが指す内容に定数 0 を代入する式で、コード不正となる制限

【内　容】“char/unsigned char 型配列のアドレス＋unsigned char 型の式”を int/unsigned int/short /unsigned short へのポインタにキャストし、そのポインタが指す内容に定数 0 を代入する式でコード不正となる場合があります。

（例）

```
unsigned char table[10], idx;
void func()
{
    unsigned char dummy;
    *((short *)(table + idx)) = 0x0;
}
```

【回避策】以下の記述のようにテンポラリ変数を介してください。

```
*((short *)(table + idx)) = 0x0;
        ↓
short *tmp;
tmp = (short *)(table + idx);
*tmp = 0x0;
```

【改善策】V3.70 で修正済みです。
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. 39 ポインタを返す関数のreturn文に、char/unsigned char型の式を記述するとコード不正となる制限

【内　容】ポインタを返す関数のreturn文に、char/unsigned char型の式を記述するとコード不正となります。

（例）

```
struct t {
    char c1;
    char c2;
    char c3;
} st = { 0x40, 0x01, 0x00 };
char *func()
{
    return st.c1;
}
```

【回避策】return 式を明示的にキャストしてください。

```
return st.c1;
    ↓
return (char *)st.c1;
```

【改善策】V3.70 で修正済みです。
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. 40 long型ランタイムライブラリの演算結果をchar/unsigned char型にキャストし、char/unsigned char型で表現可能な定数と関係/等価演算すると、エラーとなる制限

【内　容】long型ランタイムライブラリの演算結果をchar/unsigned char型にキャストし、char/ unsigned char型で表現可能な定数と関係/等価演算後、C0101,C0104エラーとなる場合があります。
関係演算の場合は演算数がunsigned char型の時に、等価演算の場合は演算数がchar/unsigned char型で定数が0以外の時に発生します。

（例）

```
long l1;
int x;
void func()
{
    if ((char)(l1 & 1) == 1) x++;
}
```

【回避策】以下の記述のようにテンポラリ変数を介してください。

```
char tmp;
tmp = (char)(l1 & 1);
if (tmp == 1)   x++;
```

【改善策】V3.70 で修正済みです。

No. 41 大きさが定義されていない配列の初期化で、初期化子の中括弧の囲みが不統一な場合、確保される領域のサイズが不正となる制限

【内　容】大きさが定義されていない配列の初期化で、初期化子の中括弧の囲みが不統一な場合、確保される領域のサイズが不正となります。

（例）

```
struct t {
    int a;
    int b;
} x[] = {1, 2, {3, 4}};
```

【回避策】次のいずれかの方法で回避してください

(1) 中括弧の囲み方を統一させる。

```
struct t {
    int a;
    int b;
} x[] = {{1, 2}, {3, 4}};
```

(2) 配列の大きさを定義する。

```
struct t {
    int a;
    int b;
} x[2] = {1, 2, {3, 4}};
```

【改善策】V4.10 で修正しました。

No. 42 文字列変換関数の標準ライブラリのエラー処理動作が不正となる制限

【内　容】文字列変換関数の標準ライブラリのエラー処理の動作が不正となります。

(1) 関数strtodで変換できない場合の動作が不正となります。
以下のように変換できない文字列が指定された場合、pの位置が不正となります。
(例 1)

```
#include <stdlib.h>
double d;
char *p;
static char *string = "     XXX";
int x;
void func()
{
        d = strtod(string, &p);
        if (string == p) x++;
}
```

※(不正な動作) pの位置が"XXX"の先頭になります
(正しい動作) pの位置が"　XXX"の先頭になります

(2) 関数strtolで変換できない場合、errnoが設定されません。
以下のように変換できなかった場合、errnoにマクロERANGEを設定しません。
(例 2)

```
#include <stdlib.h>
#include <errno.h>
long l;
static char *string1 = "99999999999";
static char *string2 = "-99999999999";
int x;
void func()
{
        errno = 0;
        l = strtol(string1, NULL, 0);
        if (errno == ERANGE) x++;
        errno = 0;
        l = strtol(string2, NULL, 0);
        if (errno == ERANGE) x++;
}
```

(3) 関数strtoulで変換対象列の先頭が"+"の時、正しく変換できません。
また変換できない文字列の場合、errnoにマクロERANGEを設定しません。

(例 3)

```
#include <stdlib.h>
#include <errno.h>
unsigned long ul;
char *p;
static char *string = "99999999999";
int x;
void func()
{
        ul = strtoul("   +12", &p, 10);
        if (ul == 12L) x++;
        errno = 0;
        ul = strtoul(string, NULL, 0);
        if (errno == ERANGE) x++;
}
```

(4) 関数strncpyでコピー元の文字列の長さが第三引数の指定文字数に満たない時、コピー先の指定文字数に満たない部分にナル文字を複写しません。

(例 4)

```
#include <string.h>
char string1[] = "aaaaaaaaaa";
char string2[] = "bbbb¥0bbbb";
void func()
{
        strncpy(string1, string2, 8);
}
```

※(不正な動作) “bbbb¥0aaaaa”となります
(正しい動作) 指定文字数に満たない部分はナル文字を書き込み“bbbb¥0¥0¥0¥0aa”となります

【回避策】ありません。
【改善策】V4.00 で修正済みです。

No. 43 入出力関数の標準ライブラリの出力変換で、動作が不正となる制限
【内 容】関数printf,sprintf,vprintf,vsprintfの出力変換で、以下のような場合に動作が不正となります。
(1) 変換指定子"d, i, o, u, xおよびX"に対して、".2"のように精度を指定した場合に、0フラグを無視しません。
(例)

```
#include <stdio.h>
void func()
{
        printf("%04.2d¥n", 77);
}
```

※(不正な動作) “0077”となります
(正しい動作) “ 77”となります

(2) 変換指定子"g,G"に対して、指定した精度+1を精度とします。
（例）

```
#include <stdio.h>
void func()
{
    printf("%.2g", 12.3456789);
}
```

※（不正な動作） "12.3"となります
（正しい動作） "12"となります

【回避策】ありません。
【改善策】制限事項とします。

No. 44 int/short型の最小値-32768のサイズが4となる制限
【内 容】int/short型の最小値-32768のサイズが4となります。
（例）

```
int x;
void func()
{
    x = sizeof(-32768);
}
```

※(不正な動作) xの値が4となります
（正しい動作) xの値が2となります

【回避策】(-32767-1)と記述してください。
【改善策】制限事項とします。

No. 45 条件演算の第2,3項に関数名または関数ポインタを記述して関数を呼び出すと、エラーとなる制限
【内 容】条件演算の第2,3項に関数名または関数ポインタを記述して関数を呼び出すと、E0307エラーとなります。

（例）

```
void f1(), f2();
int x;
void func()
{
    (x ? f1 : f2)();
}
```

【回避策】条件演算子ではなくif文にしてください。

```
(x ? f1 : f2)();
        ↓
if (x) {
        f1();
}
else {
        f2();
}
```

【改善策】V4.10 で修正しました。

No. 46 外部ポインタ変数を演算子->を含む変数で初期化するとエラーとなる制限

【内　容】外部ポインタ変数を演算子->を含む変数で初期化するとE0750エラーとなります。

（例）

```
struct t {
        int i;
} b;
int *ip1 = &(&b)->i;
```

【回避策】以下の記述で回避できます。

```
int *ip1 = &(&b)->i;
          ↓
int *ip1 = &b.i;
```

【改善策】V3.70 で修正済みです。

No. 47 関数定義の識別子の型に対して、仮引数型と関数定義の識別子の型と合わずに、エラーとなる制限

【内　容】関数定義の識別子の型に対して実引数拡張をしていないため、仮引数型と関数定義の識別子の型と適合せずに、E0747エラーとなります。

（例）

```
int fn_char(int);
int fn_char(c)
char c;
{
        return 98;
}
```

【回避策】仮引数型と関数定義の識別子の型を合わせてください。

【改善策】制限事項とします。

No. 48 関数定義の識別子並びで、宣言していない仮引数をint型とせずエラーとなる制限
【内　容】関数定義の識別子並びで、宣言していない仮引数をint型とせずE0706エラーとなります。

（例）

```
void func(x1, x2, f, x3, lp, fp)
int (*fp)();
long *lp;
float f;
{
                    :
}
```

【回避策】関数定義の仮引数はすべて宣言してください。
【改善策】制限事項とします。

No. 49 #演算子が正しく展開できない制限
【内　容】以下のいずれかの条件で正しく展開できません。
（条件1）#演算子で「'"'」を正しく展開できずに、コンパイル時にエラーとなります。

（条件1の例）

```
#include <string.h>
#define str( a) (# a)
int x;
void func()
{
    if (strcmp(str('"'), "'¥"'") == 0) x++;
}
```

※(不正な動作) コンパイル時にエラーとなります
（正しい動作） if (strcmp(("'¥"'") , "'¥"'") == 0) x++; となります

（条件2）#演算子とネストを含むマクロを正しく展開できません。

（条件2の例）

```
#define str(a) #a
#define xstr(a) str(a)
#define EXP 1
char *p;
void func()
{
    p = xstr(12EEXP);
}
```

※(不正な動作) “p = ("12E1");”となります
（正しい動作） “p = ("12EEXP") ;”となります

【回避策】ありません。
【改善策】制限事項とします。

No. 50 unsigned long型の静的変数を、0x80000000以上の浮動小数点定数で初期化すると、初期値が不正となる制限

【内　容】unsigned long型の静的変数を、0x80000000以上の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。

（例）

```
unsigned long ul = 2200000000.0;
```

【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。

```
unsigned long ul = 2200000000;
```

(2) 浮動小数点定数を適切な整数型にキャストする。

```
unsigned long ul = (unsigned long) 2200000000.0;
```

【改善策】V3.70 で修正済みです。
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. 51 関数呼び出しと構造体/共用体を含む式で、コード不正となる制限

【内　容】次のいずれかの条件を満たす時、構造体／共用体のコードが不正となる場合があります。

（条件1）次のどちらかの条件を満たす場合

(1) 関数呼び出しと構造体/共用体代入を含む式
(2) 関数ポインタによる関数呼び出しで引数が構造体/共用体

（条件1の例）

```
struct t {
    char c[16];
} st1, st2;
int x, i, idx;
int ifunc();
void (*fp[3])();
void func()
{
    x = ifunc() + (st1 = st2, i);
    fp[idx](st1);
}
```

（条件2）次の条件をすべて満たす場合

(1) 関数ポインタによる関数呼び出し

(2) 第一引数が 3,4 バイトの構造体/共用体で値がレジスタに残っている

旧仕様の関数インタフェースでは、C0101 エラーとなります。

（条件2の例）

```
void (*fp)();
struct {
    char a;
    char b;
    char c;
    char d;
} st1, st2;
void func()
{
     fp(st1 = st2);
}
```

【回避策】以下のいずれかの方法で回避してください。

（条件1の回避策）以下の方法で回避してください。

(1)関数呼び出しを含む式に、構造体/共用体代入を記述しない。

```
x = ifunc() + (st1 = st2, i);
                ↓
st1 = st2;
x = ifunc() + i;
```

(2)関数ポインタによる関数呼び出し時は、構造体/共用体引数ではなく構造体/共用体ポインタを使う。

```
fp[idx](st1);
        ↓
struct t *sp1;
sp1 = &st1;
fp[idx](sp1);
```

（条件2の回避策）関数呼び出しの第一引数に構造体/共用体代入を記述しない。

```
fp(st1 = st2);
      ↓
st1 = st2;
fp(st1);
```

【改善策】V3.70 で修正済みです。

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. 52 「a = b 二項演算子 c;」の代入式で、コード不正となる制限

【内　容】「a = b 二項演算子 c;」の代入式で、以下の条件をすべて満たす時にコード不正となります。

(1) 二項演算子が ＋、－、＊、&、^、｜、＜＜ のどれかである
(2) a が識別子、b と c が識別子または定数である
(3) 演算数 a が int/unsigned int/short/unsigned short 型である
(4) 演算数bとcの一方がchar/unsigned char型、もう一方がlong/unsigned long型である

(例)

```
char c1=0x12, c2=0x56;
int i1=0x34, i2=0x78;
void func()
{
 /*    (2)    (1)          上記の番号に対応 */
       i2 = c2 ^ 0x1ffff;
} /*        (3)     (4)     */
```

【回避策】long/unsigned long型演算数を代入先の型にキャストしてください。

```
i2 = c2 ^ 0x1ffff;
        ↓
i2 = c2 ^ (int)0x1ffff;
```

【改善策】V3.70 で修正済みです。
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. 53 ドット演算子[.]を使って、定数番地構造体の配列メンバの要素を参照するとエラーとなる制限

【内　容】ドット演算子[.]を使って、定数番地構造体の配列メンバの要素を参照すると C0101 エラーとなります。

(例)

```
struct st {
    char b[4];
} str;
char c;
void test(void)
{
        c = (*(struct st *)0x3E00).b[2];
}
```

【回避策】矢印演算子[->]を使用してください。

```
c = (*(struct st *)0x3E00).b[2];
        ↓
c = ((struct st *)0x3E00)->b[2];
```

【改善策】V3.70 で修正済みです。

No. 54 特定のパターンの引数を持つ関数定義で、エラーとなる制限

【内　容】ノーマルモデルで-QRオプション指定時に、第一引数から順に1byte幅、1byte幅、2byte幅、2byte幅の４つの引数を持つ関数が、次のいずれかの条件を満たす場合に、E0705エラーとなります。

(1) 関数がnoauto関数である
(2) 全ての引数をレジスタ宣言している
(3) -QVオプションを指定している

（例）-QRVオプション指定時

```
void f(char p1, char p2, int p3, int p4)
 {
                              :
 }
```

【回避策】第一引数から 1byte 幅、1byte 幅、2byte 幅、2byte 幅の並びにならないように、引数の宣言順序を変えてください。

【改善策】V3.70 で修正済みです。

No. 55 複合代入／インクリメント／デクリメント演算の代入先に unsigned char 型を返す関数の返り値を配列添字に使った要素が1バイトの静的配列の要素を記述すると、コード不正となる場合がある制限

【内　容】複合代入／インクリメント／デクリメント演算の代入先に unsigned char 型を返す関数の返り値を配列添字に使った要素が1バイトの静的配列の要素を記述すると、コード不正となる場合があります。

（例1）

-------C ソース----------

```
unsigned char a1[10];
unsigned char func1(viod);
 void func()
{
        a1[func1()]++;
}
```

-------ASM ソース--------

```
_func:
                                          ; push  hl  が出ていない
; line        5 :        ary1[func1()]++;
      call     !_func1
      movw  hl,#_ary1
      mov    a,[hl+c]
       inc     a
      mov    [hl+c],a
; line        6 : }
                                          ; pop  hl  が出ていない
      ret
```

（例 2）

```
-------Cソース----------
signed char a2[10];
unsigned char func1(void);
signed char c1;
struct t1 {
    char m1;
} a3[10];
void func()
{
    a2[func1()] &= c1;              /* コード不正 */
    a3[func1()].m1 |= 0x55;         /* コード不正 */
}
--------ASMソース----------
_func:
                                    ; push hl が出ていない
; line      9 :       a2[func1()] &= c1;
    call        !_func1
    movw        hl,#_a2
    mov         a,[hl+c]
    and         a,!_c1
    mov         [hl+c],a
; line      10 :      a3[func1()].m1 |= 0x55;
    call        !_func1
    movw        hl,#_a3
    mov         a,[hl+c]
    or          a,#055H         ; 85
    mov         [hl+c],a
line      11 : }
                                    ; pop hl が出ていない
    ret
```

【回避策】ダミーのローカル変数を定義してください

（例1）

```
unsigned char a1[10];
unsigned char func1(void);
void func()
{
    unsigned char dummy;            /* ダミー変数を定義 */
    a1[func1()]++;
}
```

（例2）

```
signed char a2[10];
unsigned char func1(void);
signed char c1;
struct t1 {
    char m1;
} a3[10];
void func()
{
    unsigned char dummy;            /* ダミー変数を定義 */
    a2[func1()] &= c1;
    a3[func1()].m1 |= 0x55;
}
```

【改善策】V4.00 で修正済みです。
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. 56 桁あふれする二項演算の結果を使う、2つ以上の二項演算子を含む定数式を不正な値に置き換える場合がある制限

【内　容】桁あふれする二項演算の結果を使う、2つ以上の二項演算子を含む定数式を不正な値に置き換える場合があります。

（例）

```
long l1 = (10000 * 10000) / 10000;
long l2 = (30464 << 4) / 2;
long l3 = (30464 << 5) / 2;
long l4 = (65535U * 41200U) / 256L;
short s1 = (65535U * 41200U) / 100000;
void func()
{
    l1 = (10000 * 10000) / 10000;
    l2 = (30464 << 4) / 2;
    l3 = (30464 << 5) / 2;
    l4 = (65535U * 41200U) / 256L;
    s1 = (65535U * 41200U) / 100000;
}
```

【回避策】2 つ以上の二項演算子を含まない定数式を記述してください。
【改善策】V4.00 で修正済みです。
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. 57 インクリメント/デクリメント演算を含んだ演算で、コード不正となる場合がある制限

【内　容】(1) 二項演算の演算数がビット・フィールドのインクリメント/デクリメント演算を含んでいると、コード不正となる可能性があります。

（例1）

```
struct {
        int i : 9;
} *p;
int i;
int g(void);

void f(void)
{
        i = g() + p->i++;
}
```

(2) 演算結果がレジスタに残ってい時に、コンマ演算の左演算数に後置インクリメント/デクリメント演算を記述すると、コードと不正となる可能性があります。

（例2）

```
unsigned char c0, c1, c2, c3, c4;
void func()
{
        c0 = (c1 + c2) + (c3++, c4);
}
```

【回避策】(1) インクリメント/デクリメント演算と、二項演算の式を分けてください。
(2) 後置インクリメント/デクリメント演算と、コンマ演算の式を分けてください。

【改善策】V4.00 で修正済みです。
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. 58 配列型を持つ関数パラメータのsizeof演算結果が不正となる制限

【内　容】配列型を持つ関数パラメータのsizeof演算結果が不正、あるいはエラーE0529となります。

（例）

```
int x;
void func1(short a[ ])
{
        x = sizeof(a);                  /* E0529: Sizeof returns zero */
}

void func2(short b[10])
{
        x = sizeof(b);
}
```

【回避策】関数パラメータの型をポインタに変えてください。

【改善策】V4.00 で修正済みです。
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. 59 -qcオプション未指定時に、除算・剰余算の複合代入演算がコード不正となる場合がある制限

【内　容】 -qcオプション未指定時、左演算数がchar/unsigned char型へのポインタ、右演算数がオフセット256以上のスタックにある1byteで表現できる型の変数の場合に、除算・剰余算の複合代入演算結果が不正となる可能性があります。

（例）

```
char *p;

void f(unsigned char uc)
{
        char a[256];
        *p /= uc;
}
```

【回避策】 -qc オプションを指定してください。
あるいは配列の要素数を 255 以下にしてください。

【改善策】 V4.00 で修正済みです。
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. 60 saddr領域に割り当てたビット幅が2ビット以上7ビット以下のビット・フィールドに、ビット・フィールドの最大定数値を代入後、同じ式で代入先を再評価するとコード不正となる制限

【内　容】 saddr領域に割り当てたビット幅が2ビット以上7ビット以下のビット・フィールドに、ビット・フィールドの最大定数値を代入後、同じ式で代入先を再評価するとコード不正となります。

（例）

```
_sreg struct {
        unsigned int b1 : 4;
        unsigned int b2 : 4;
} s;

void main(void)
{
        s.b2 = s.b1 = 0xf;
}
```

【回避策】 代入式を分けてください。

【改善策】 V4.00 で修正済みです。
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. 61 enum型のパラメータを持つnorec関数を呼ぶと、エラーE0705を出力、あるいはコード不正となる場合がある制限

【内　容】 enum型のパラメータを2つ持つnorec関数を呼ぶと、-qrオプション未指定時はエラーE0705を出力します。
また、enum型のパラメータを2つ以上持つnorec関数を呼ぶと、-qrオプション指定時はコード不正となります。

（例）

```
enum E {
        A = 256
};

_leaf int func(enum E e1, enum E e2)
{
    return e1 == e2;
}

int main(void)
{
    func(A, A);
    return 0;
}
```

【回避策】 以下のように enum 型を int 型、列挙定数をマクロに変えてください。

（例）

```
#define A 256

_leaf int func(int e1, int e2)
{
    return e1 == e2;
}

int main(void)
{
    func(A, A);
    return 0;
}
```

【改善策】 V4.00 で修正済みです。
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. 62 異なる構造体/共用体型をパラメータに持つ同一関数の宣言が、エラーとならない制限

【内　容】 異なる構造体/共用体型をパラメータに持つ同一関数の宣言をしても、エラーE0747が出力されません。

（例）

```
struct st {
        int a;
} x;
struct st2 {
        char a;
} x2;

void func11(struct st a);
void func11(struct st2 a);
```

【回避策】 同一関数の宣言を複数個記述しないようにしてください。

【改善策】 V4.00 で修正済みです。
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. 63 サイズが256byte以上の構造体を指すポインタがレジスタ変数の時に、コード不正となる場合がある制限

【内　容】 サイズが256byte以上の構造体を指すポインタがレジスタに割り当たった時、そのメンバへの間接参照でレジスタ変数の値が破壊される場合があります。

（例）

```
struct st1 {
        char buf[250];
        struct st2 {
                char buf[6];
                int   i1;
                int   i2;
        } st2;
} st1;
int i1;
void func()
{
        register struct st1 *pst1 = &st1;
        i1 = pst1->st2.i1;
}
```

【回避策】 register 宣言を行わず、さらに-qv オプションを無効にしてください。
あるいは、構造体のサイズを 255byte 以下にしてください。

【改善策】 V4.00 で修正済みです。
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. 64 スタティック・モデル・オプション"-sm"および拡張仕様オプション"-zm2"指定時に、コンパイラが使用するワークエリアを壊す制限

【内　容】 スタティック・モデル・オプション"-sm"および拡張仕様オプション"-zm2"指定時に、以下の(a)または(b)のいずれかの条件を満たす時に、コンパイラが使用するワークエリアを壊します。
メモリ・モデルがノーマルの場合は該当しません。

(a) 関数パラメータが以下のいずれか1つに該当している
・&演算子でアドレス参照している
・構造体
・共用体

(b) オート変数が以下のいずれか1つに該当している
・&演算子でアドレス参照している
・構造体
・共用体
・配列

((b)の例)

```
void func2(unsigned short *);
struct tag {
        unsigned char a[2];
};
void func1()
{
        unsigned char buf[4]; /* 配列 */
        struct tag st;        /* 構造体 */
        unsigned short ss;    /* アドレス参照あり */

        func2(&ss);
        /* ... */
}
```

【回避策】 (1) (a)の場合は-zm2 オプションを-zm1 オプションに変えてください。
C ソースの記述を変更して回避することはできません。

(2) (b)の場合はオート変数を関数内 static 変数に変更してください。
または-zm2 オプションを-zm1 オプションに変更してください。

```
void func2(unsigned short *);
struct tag {
        unsigned char a[2];
};
void func1()
{
        static unsigned char buf[4]; /* 配列 */
        static struct tag st;        /* 構造体 */
        static unsigned short ss;    /* アドレス参照あり */

        func2(&ss);
        /* ... */
}
```

【改善策】 V4.00 で修正済みです。

No. 65 後置++/--ポインタが指す1バイトデータ参照時にコード不正となる制限

【内　容】 ポインタが指す1バイトの参照直後に、後置インクリメント/デクリメントする同ポインタが指す先を参照すると、コード不正となる場合があります。

【例】

```
void func()
{
    unsigned char tmp, *src, dst;
    *src = tmp + 0x80;
    dst += *src++;
}
```

【回避策】 以下のいずれかの方法で回避してください。

(1) 代入＆後置インクリメントを、別々の式に分ける。

```
dst += *src;
src++;
```

(2) テンポラリ変数を用意して式を分割する。

```
tmp2 = tmp + 0x80;
*src = tmp2;
```

【改善策】V4.10 で修正しました。

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、販売店または特約店にお問い合わせください。

No. 66 char/signed char/unsigned char 型配列の初期化子並びの最後が文字列で、文字列の前に 1 個以上の定数または文字定数が並ぶ時にエラーとならず、コード不正となる制限

【内　容】 char/signed char/unsigned char 型配列の初期化子並びの最後が文字列で、文字列の前に 1 個以上の定数または文字定数が並ぶ時にエラーとならず、コード不正となる場合があります。

char/signed char/unsigned char 型は、文字列リテラルもしくは定数値による初期化しか許されません。

【例】

[.c]

```
const char a1[ ] = {0x01, “abct”};
char *const TBL[3] = { a1 };
char *ptr1;
void func()
{
    ptr1 = TBL[0];
}
```

[.asm]

```
@@CNST   CSEG     UNITP
_a1:     DB       01H        ; 1
         DB       'ab'
_TBL:    DW       _a1        ; _TBL が奇数アドレス
         DB       (4)

@@DATA   DSEG     UNITP
_ptr1:   DS       (2)

; line    1 : const char a1[] = { 0x01, ″abc″ };
; line    2 : char *const TBL[3] = { a1 };
; line    3 : char *ptr1;
; line    4 : void func()
; line    5 : {

@@CODE   CSEG
_func:
; line    6 :   ptr1 = TBL[0];
        movw    ax,!_TBL        ; 奇数アドレスを参照
        movw    !_ptr1,ax
```

【回避策】 初期値を正しく記述してください。

```
const char a1[] = { 0x01, 'a', 'b', 'c', '¥0' };
char *const TBL[3] = { a1 };
char *ptr1;
void func()
{
        ptr1 = TBL[0];
}
```

【改善策】V4.10 で修正しました。

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、販売店または特約店にお問い合わせください。

No. 67 ポインタ同士の減算結果をオフセットとしたポインタ参照時に、コード不正となる制限

【内　容】以下の条件をすべて満たす時にコード不正となります。

(1)「ポインタ＋オフセット」の参照

(2) (1)のオフセットが、ポインタ同士の減算

(3) (2)のポインタ同士の減算において、ポインタがオフセット付き

【例】

[.c]

```
void main(void)
{
      char   *p1;
      char   *p2;
      char   *p3;
      *p1 = *(p2 + (p1 – (p3 + 2)));
}
```

【回避策】 式を分割してください。

[.c]

```
void main(void)
{
      char   *p1;
      char   *p2;
      char   *p3;
      int    tmp;                    /* テンポラリ変数を用意する */
      tmp = (p1 - (p3 + 2));         /* テンポラリ変数に代入する */
      *p1 = *(p2 + tmp);
}
```

【改善策】 V4.10 で修正しました。

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、販売店または特約店にお問い合わせください。

No. 68 -qcオプションが未指定(int型拡張する)の時、コード不正となる制限

【内　容】以下の条件をすべて満たす時にコード不正となります。

(1) -qc 未指定(int 型拡張する)

(2) 次のいずれかの組合わせの乗算(演算子の左右を入れ換えても発生する)

・「0～255 の定数を代入した unsigned char 型 sreg 変数」と「0～255 の定数」

・「0～255 の定数を代入した unsigned char 型 sreg 変数」同士

・「0～255 の定数を代入した unsigned char 型 sreg 変数」と「0～127 の定数を代入した char/signed char 型 sreg 変数」

(3) 乗算結果が 256 以上(unsigned char 型で表現できない値)

(4) 演算結果を int 型としてあつかう

以下の例では、正しくは Temp1 が 0x1fe になるはずが、0xfe になる。

【例】

[.c]

```
unsigned int     Temp1;
_sreg unsigned char   Byte1;
Temp1 = (Byte1 = 255) * 2;
```

【回避策】 変数を int/unsigned int 型にキャストしてください。

[.c]

```
unsigned int     Temp1;
unsigned char    Byte1;

Temp1 = (unsigned int) (Byte1 = 255) * 2;
```

【改善策】 V4.10 で修正しました。

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、販売店または特約店にお問い合わせください。

No. 69 BCD演算関数"adbcdw", "sbbcdw"のコード不正となる制限

【内　容】BCD 演算関数"adbcdw"および"sbbcdw"を使用すると、コード不正となる場合があります。

以下のいずれかの条件でコード不正になります。

(1) 代入先が配列またはポインタで、アドレス計算コードが発生する。

(2) テンポラリ変数を代入する前に、他の演算でレジスタを使用したコードが発生する。

(3) 代入したテンポラリ変数を、そのまま条件式など他の演算に使用する。

【例】

[.c]

```
void func()
{
    unsigned int     tmp1[3];
    unsigned int     tmp2, tmp3;
    unsigned int     a = 10, i = 0, *p;

    tmp1[i] = adbcdw(80, 50);                  /* (1) */
    tmp2 = adbcdw(80, 50) + (a + 1);           /* (2) */
    if ((tmp3 = adbcdw(80, 50) == *p ) ・・・    /* (3) */
        :
}
```

【回避策】 関数"adbcdw", "sbbcdw"を呼び出すだけの関数を用意し、その関数を呼び出してください。
引数、返り値は関数 adbcdw,sbbcdw と同じにしてください。

【例】
[.c]

```
unsigned int adbcdw_new(unsigned int a, unsigned int b)
{
        return adbcdw(a, b);
}
void func()
{
    unsigned int    tmp1[3];
    unsigned int    tmp2, tmp3;
    unsigned int    a = 10, i = 0, *p;

    tmp1[i] = adbcdw_new(80, 50);               /* (1) */
    tmp2 = adbcdw_new(80, 50) + (a + 1);        /* (2) */
    if ((tmp3 = adbcdw_new(80, 50) == *p ) ...  /* (3) */
        :
}
```

【改善策】 V4.10 で修正しました。

No. 70 -ngオプションを指定し、ASM文を含む関数において、分岐命令でエラーとなる制限
【内　容】 -ngオプションを指定し、ASM 文を含む関数において、ASM 文より後ろにある分岐命令でエラーとなる場合があります。
以下のすべての条件を満たすときエラーとなる場合があります。
(1) 関数内に ASM 文がある。
(2) 同関数内に分岐命令を出力する文(if 文, for 文, while 文など)がある。

ただし、この時、アセンブル時にエラーとなりますので、オブジェクト・モジュール・ファイルは生成されません。 エラーとならなければ、本制限に該当しません。

【例】
[.c]

```
unsigned int i;
void func()
{
    do {
        _asm("¥t DB   (1000)");
        i++;
    } while ( i < 10) ;
}
```

【回避策】 -gオプションに変更してください。
【改善策】 V4.10 で修正しました。

No. 71 ネストしたif文を抜けた直後の文に対して、行番号情報が出ない制限

【内　容】 以下のすべての条件を満たすときに、ネストした if 文を抜けた直後の文に対して、行番号情報が出ない場合があります。 ただし、出力コードは正しいです。
行番号情報が出なかった行にはブレーク・ポイントの設定が出来ません。

(1) 2 つ以上の if 文が入れ子になっているネストレベル 3 以上の if 文がある。
(2) 上のネストレベルにある else が if 文の行番号より大きい。
(3) 直後に文が続く、上のネストレベルにある if 文が少なくとも1つある。

【例】

[.c]

```
int   f0, f1, f2, f3;
int   g0, g1, g2;
void func(void)
{
        if (f0) {
            if (f1) {           /* ネストレベル 1 の if 文 */
                g2 = 5;
            }
            else if (f2) {      /* ネストレベル 2 の if 文 */
                g2 = 4;
            }
            else if (f3) {      /* ネストレベル 3 の if 文(条件(1)) */
                g2 = 3;
            }
            else {
                g2 = 2;
            }
            g0 = 0x1234;        /* ネストレベル 1 の if 文の直後に文あり(条件(3)) */
            g1 = 0x5678;
        }
        else {                  /* ここの else がネストレベル 3 の if 文の */
            g2 = 1;             /* 行番号より大きい(条件(2))              */
        }
}
```

【回避策】 ネストした if 文を抜けた直後の文の前に、空行を数個挿入してください。
上記例では「g0 = 0x1234;」の前に、空行を数個挿入してください。

【改善策】 V4.10 で修正しました。

No. 72 割り込み関数内のlong型変数への代入でコード不正となる場合がある制限

【内　容】 割り込み関数内で long 型変数への代入を記述し、-ql3 オプション以上を指定し、ランタイム・ライブラリ関数@@dels03 または@@hlls03 を呼び出すコードをコンパイラが生成した際に、割り込み関数内の他の処理で BC レジスタを使わず、割り込み関数内から関数を呼ばない時に、BC レジスタの中身を破壊します。

【例】

```
[ .c ]
unsigned long l;
unsigned int i;
_interrupt void func()
{
    l = (unsigned long)i;
}
```

【回避策】 以下のいずれかで回避してください。

(1) 関数末尾の return 文(なければ追加)の後に、long 型変数をインクリメントする処理を追加してください。

```
[ .c ]
unsigned long l;
unsigned int i;
_interrupt void func()
{
        l = (unsigned long)i;
        return;
        l++;            /* 実行されない */
}
```

(2) 空の処理のダミー関数を用意し、割り込み関数から呼び出してください。

```
[ .c ]
unsigned long l;
unsigned int i;
void dummy { }
_interrupt void func()
{
        l = (unsigned long)i;
        dummy();
}
```

(3) 最適化オプション-ql のレベルを、-ql1 または-ql2 にしてください。

【改善策】 V4.10 で修正しました。
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、販売店または特約店にお問い合わせください。

No. 73 バンク関数呼び出しコードが出ない場合がある制限

【内　容】以下の条件のいずれかを満たす時に、関数情報ファイルでバンク領域に配置するように指定した関数に対して、バンク関数呼び出しコードが出ない場合があります。
また、(3)に対しては「C0101：Internal error」が出る場合があります。
出なければ問題はなく、プログラム・コードには影響はありません。

(1) 関数ポインタにキャストして関数を呼び出す。
(2) typedef 名を使って宣言した関数を呼び出す。
(3) -mf オプションを指定し、関数ポインタを使って関数を呼び出す。

【例】

```
[ .c ]
typedef void F(void);
typedef void (*FP)(void);
void func1(void);
F func2;
void func()
{
        ((FP)func1)();
        func2();
}
```

【回避策】 関数ポインタにキャストしてバンク関数を呼び出さないようにしてください。
typedef 名を使ってバンク関数を宣言しないようにしてください。

```
[ .c ]
void func1(void);
void func2(void);
void func()
{
        func1();
        func2();
}
```

【改善策】 V4.10 で修正しました。
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、販売店または特約店にお問い合わせください。
ただし、条件(3)に関してはチェックを行いません。

No. 74 norec関数にて1バイトの引数またはauto変数使用時に、コード不正となる場合がある制限

【内　容】norec 関数にて1バイトの引数または auto 変数の使用時に、norec 関数内で long 型変数に対して間接参照すると、コード不正となる場合があります。

【例】

```
[ .c ]
long buf[8];
norec void func(void)
{
    unsigned char c = 7;
    long *s = &buf[c-1], *d = &buf[c];
    *d = *s;
}
```

【回避策】 norec 関数でなく通常の関数にするか、1バイト変数を2バイト変数に変えてください。

【改善策】 V4.10 で修正しました。
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、販売店または特約店にお問い合わせください。

No. 75 ##演算子を使ったマクロ展開でエラーとなる制限

【内　容】## 演算子の直後が、関数形式マクロのパラメータでなく、大小英字でも下線 _ でもない時、E0803 エラーが出る場合があります。
演算子の直前のトークンと直後のトークンを連結して、別のトークンを生成する時は、E0711,E0301 などの E0803 以外のエラーが出る場合があります。

【例1】

```
[*.c]
#define m1(x) (x ## .c1 + 23)
#define m2(x) (x ## .c1 + 122)
struct t1 {
        unsigned char c1;
} st1;
unsigned char x1, x2;
void func1()
{
   x1 = m1(st1) + 100;     /* E0803 エラー (NG) */
   x2 = m2(st1) + 1;       /* ノーエラー   (OK) */
}
```

【例2】

```
[*.c]
#define m3(x) (x ## 1)
unsigned char x3, uc1;
void func2()
{
   x3 = m3(uc);            /* E0711, E0301 エラー (NG) */
}
```

【回避策】## 演算子の直前のトークンと直後のトークンを連結して、別のトークンを生成しない時は、
演算子を使わないでください。

```
#define m1(x)    (x ## .c1 + 23)
#define m2(x)    (x ## .c1 + 122)
 ↓
#define m1(x)    ((x).c1 + 23)
#define m2(x)    ((x).c1 + 122)
```

演算子の直前のトークンと直後のトークンを連結して、別のトークンを生成する時は、
演算子の直後に関数形式マクロのパラメータを置いてください。

```
#define m3(x)    (x ## 1)
unsigned char x3, uc1;
void func2()
{
 x3 = m3(uc);
}
 ↓
#define m3(x, y) (x ## y)
unsigned char x3, uc1;
void func2()
{
 x3 = m3(uc, 1);
}
```

【改善策】 次版で修正いたします。

No. 76 アセンブラ・ソースに割り込み関数のシンボル情報が出ない制限

【内　容】以下の条件をすべて満たすときに、リンク時に E3405 エラーになります。
(1) #pragma interrupt による割り込み関数のベクタテーブルの生成指定がある。
(2) 同一ソース内に割り込み関数の定義がない。
(3) -no, アセンブラ・ソース・モジュール・ファイル出力(-a or -sa)、
デバッグ情報出力(-g)オプションを有効にする

```
[*.c]
#pragam interrupt INTP0 inter
/*                              割り込み関数の定義がコンパイル対象ではない
__interrupt void inter()
{
            :
}
*/
```

【回避策】 割り込み関数を定義する、あるいはオブジェクト・モジュール・ファイルを出力してください。
【改善策】 次版で修正いたします。

以上