マイクロコントローラ技術情報

技術通知 78K0R C コンパイラ CC78K0R 使用制限事項の件	発行番号	ZMT－F35－11－0002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	CC78K0R Ver.2.00　言語編	資料番号：U18548JJ1V0(第 1 版)
	CC78K0R Ver.2.00　操作編	資料番号：U18549JJ1V0(第 1 版)
	78K0R　Cコンパイラ　CC78K0R Ver.2.13　使用上の留意点	資料番号：ZUD-CD-10-0100

1. 対象製品

CC78K0R　　（対象バージョンは別紙1の製品履歴を参照してください。）

2. 新たな制限事項

今回新たに制限事項 No.30-34 を追加しました。詳細は、別紙1を参照してください。

- No.30　-qjオプション指定に関するC0101エラーとなる制限
- No.31　far領域のアドレスを long/unsigned long 型にキャストしたときの制限
- No.32　##演算子を使ったマクロ展開でエラーとなる制限
- No.33　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
- No.34　ESレジスタの設定コードに関する制限

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。

- No.30　以下のいずれかで回避してください。
  - ・ジャンプ最適化オプション(-qj)を無効にしてください。
  - ・実行しない処理自体を削除する、あるいは#if 0 で囲んでコンパイル対象としないでください。
- No.31　アドレスが初期値の時は、long/unsigned long 型にキャストしないでください。
- No.32　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.33　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。
- No.34　間接参照を除く浮動小数点型変数のインクリメント／デクリメント演算直後にダミーのポインタ間接参照式を記述してください。

4. 改善計画

今回新たに追加した No. 30-34 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。

6. 発行文書履歴

78K0R　C コンパイラ CC78K0R 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-06-0088	2006 年 12 月 21 日	初版
ZBG-CD-07-0012	2007 年 1 月 29 日	No.20 を追加
ZBG-CD-07-0082	2007 年 11 月 29 日	No.21-22 を追加
ZBG-CD-08-0040	2008 年 9 月 18 日	No.23-26 を追加
ZBG-CD-09-0026	2009 年 5 月 20 日	No.27-28 を追加
ZBG-CD-10-0014	2010 年 3 月 30 日	No.29 を追加
ZMT-F35-11-0002	2011 年 4 月 7 日	No.30-34 を追加

以上

CC78K0R 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						
		1.00	1.10	1.20	2.00	2.10	2.12	2.13
1	norec 関数を使用できない制限	×	－	－	－	－	－	－
2	リアルタイム OS 用拡張機能を使用できない制限	×	○	○	○	○	○	○
3	メモリ・モデルのコンパクト・モデルを使用できない制限	×	－	－	－	－	－	－
4	有効な関数名長が 249 文字の関数が、248 文字となる制限	×	○	○	○	○	○	○
5	アセンブラ・ソースの関数情報が不正となる制限	×	○	○	○	○	○	○
6	関数定義の返り値で、型指定子の右に型修飾子を記述し、型指定子が構造体型、共用体型、列挙体型、typedef 名の時に、型指定子が int 型となる制限	×	○	○	○	○	○	○
7	割り込み関数(#pragma vect/#pragma interrupt)の拡張機能を記述したＣソースで、関数定義が１つも無い場合、アセンブル時にエラーが発生する制限	×	○	○	○	○	○	○
8	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	×	×	×	○	○	○	○
9	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	×	×	×	×	○	○	○
10	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	×	×	×	×	○	○	○
11	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	×	×	×	×	○	○	○
12	配列アドレスのポインタへの代入で、ポインタが指す配列の構造と、アドレスを参照した配列の構造が同じである記述に対し、ワーニング W0411を出力する制限	×	○	○	○	○	○	○
13	ミディアム・モデルまたはラージ・モデルでアセンブラ・ソース出力時に、分岐命令最適化で C0101エラーとなる場合がある制限	×	○	○	○	○	○	○
14	directmap変数をショート・ダイレクト・アドレシング可能な領域とSFRアドレシング可能な領域にまたがって配置すると、コード不正となる場合がある制限	×	○	○	○	○	○	○
15	typedef名を使った名前なし配列型パラメータ宣言を記述すると、E0301エラーとなる制限	×	○	○	○	○	○	○
16	無条件分岐命令がコード不正となる制限	×	○	○	○	○	○	○
17	volatile修飾型を型変換すると、volatile修飾のコードを出力しない場合がある制限	×	×	○	○	○	○	○
18	-qjオプション未指定時のデバッガ上の表示が不正となる場合がある制限	×	○	○	○	○	○	○

×：該当する
○：該当しない
－：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						
		1.00	1.10	1.20	2.00	2.10	2.12	2.13
19	far配置のchar配列型(signed,unsignedも含む)のdirectmap変数の配置アドレス指定に、文字列定数を記述してもエラーとならない制限	×	×	×	○	○	○	○
20	関数内にtypedef宣言、構造体型宣言、共用体型宣言、enum型宣言 のいずれかがある場合、コード不正になる制限	× ※	× ※	○	○	○	○	○
21	-raまたは-rcオプション指定時の間接参照でコード不正になる制限	×	×	× ※	○	○	○	○
22	配列またはポインタのES設定コードが出力されない場合がある制限	×	×	× ※	○	○	○	○
23	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	×	×	× ※	× ※	○	○	○
24	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	×	×	× ※	× ※	○	○	○
25	-qcオプションが未指定（int型拡張をする）の時、コード不正となる制限	×	×	× ※	× ※	○	○	○
26	BCD演算関数”adbcdw”, ”sbbcdw”のコード不正となる制限	×	×	×	×	○	○	○
27	-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	×	×	×	×	×	○	○
28	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	―	×	×	×	×	○	○
29	オペランドがword[BC]の命令（mov, movw）におけるアクセスの制限	×	×	× ※	× ※	× ※	× ※	○
30	-qjオプション指定に関するC0101エラーとなる制限	×	×	×	×	×	×	×
31	far領域のアドレスを long/unsigned long 型にキャストしたときの制限	―	―	―	―	× ※	× ※	× ※
32	##演算子を使ったマクロ展開でエラーとなる制限	×	×	×	×	×	×	×
33	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	×	×	×	×	×	×	×
34	ESレジスタの設定コードに関する制限	×	×	× ※	× ※	× ※	× ※	× ※

×：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 norec 関数を使用できない制限

【内　容】norec 関数を使用できません。
W0332 を出力し、norec 関数を通常関数とみなします。
norec の代わりに _leaf を記述した場合も同様です。

【回避策】ありません。

【改善策】今後、この制限を仕様とします。

No. 2 リアルタイム OS 用拡張機能を使用できない制限

【内　容】以下のリアルタイム OS 用拡張機能を使用できません。
(a) RTOS 用割り込みハンドラ(#pragma rtos_interrupt …)
(b) RTOS 用割り込みハンドラ修飾子(_rtos_interrupt)
(c) RTOS 用タスク関数(#pragma rtos_task)

【回避策】ありません。

【改善策】V1.10 で修正しました。

No. 3 メモリ・モデルのコンパクト・モデルを使用できない制限

【内　容】メモリ・モデルのコンパクト・モデルを使用できません。
コンパクト・モデル(-mc オプション)を指定できません。
(a)PM+をご利用の場合
「コンパイラオプションの設定」→「メモリ・モデル」タブの「コンパクト」を選択できません。
(b)コマンドライン版をご利用の場合
CC78K0R のオプションに「-mc」を指定できません。

【回避策】スモール・モデル(-ms オプション)を使用し、スモール・モデルでは配置できない変数を_far 型修飾子を指定して配置してください。

【改善策】今後、この制限を仕様とします。

No. 4 有効な関数名長が 249 文字の関数が、248 文字となる制限

【内　容】アセンブラ・ソース出力時、出力したアセンブラ・ソース中で、ミディアム・モデル、ラージ・モデル、または_far 型修飾子を付加した、有効な関数名長が 249 文字の関数呼び出しが、当該関数名から 1 文字少ない 248 文字の関数呼び出しとなり、不正コードとなります。
現象としては、以下の 2 通りです。
(a) 当該関数名から 1 文字少ない 248 文字の関数が存在しない場合
リンク時にエラーが発生します。
(b) 当該関数名から 1 文字少ない 248 文字の関数名が別途存在する場合
248 文字の関数を呼び出すコードとなります。エラーは発生しません。

【例】

```
void
sub_func0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789(void)
{
                                        :
}
```

```
void main(void)
{
 sub_func0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456
 789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_
 0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123456789_0123
 456789();
 }
```

【回避策】関数名を 248 文字以下にしてください。
【改善策】V1.10 で修正しました。

No. 5 アセンブラ・ソースの関数情報が不正となる制限
【内　容】アセンブラ・ソースの関数情報(定義関数の返り値、引数)の表示内容が、ラージ・モデル(-ml オプション)指定時の構造体を返す関数定義の箇所のみ不正となります。

【例】

[test.c]

```
struct st {
        long l[2];
} gst;
struct st stfunc() {
        return gst;
}
void main() {
        gst = stfunc();
}
```

[test.asm]

```
          :
        movw    bc,#loww (L0003)   ☆コードは正しい
        mov     e,#highw (L0003)   ☆コードは正しい
        ret
       :
; *** Code Information ***
;
; $FILE D:¥work¥test.c
; $FUNC stfunc(4)
;       bc=(void)    ★ 表示不正。 bc,de が正しい。
;       CODE SIZE= 24 bytes, CLOCK_SIZE= 21 clocks, STACK_SIZE= 0 bytes
;
; $FUNC main(7)
;       void=(void)
;       CODE SIZE= 32 bytes, CLOCK_SIZE= 31 clocks, STACK_SIZE= 4 bytes
;
; $CALL stfunc(8)
;       bc,de=(void)  ☆呼び出し側は正しい
```

【回避策】ありません。
【改善策】V1.10 で修正しました。

No. 6 関数定義の返り値で、型指定子の右に型修飾子を記述し、型指定子が構造体型、共用体、列挙体型、typedef 名の時に、型指定子が int 型となる制限

【内　容】関数定義の返り値で、型指定子の右に型修飾子を記述し、型指定子が構造体型、共用体、列挙体型、typedef 名の時に、型指定子が int 型となります。
定義した関数名のデバッグ情報が不正となります。
同一ファイル中で、適合するプロトタイプ宣言を記述すると、E0747 エラーとなります。
また、別ファイルで、適合するプロトタイプ宣言を記述すると、リンク時に E3403 エラーとなります。

【例】

```
struct t1 {
      char a[4];
      int b;
};
struct t1 _far *stp1;
struct t1 _far st1;

/* プロトタイプ宣言なし */
struct t1 _far *func1()
{
     return stp1; /* W0411 発生 */
}

/* プロトタイプ宣言あり */
struct t1 _far func2();
struct t1 _far func2() /* E0747 発生 */
{
     return st1; /* E0402 発生 */
}
```

【コード不正の例】

```
typedef unsigned long DWORD;
unsigned long ldata1 = 0x12348765;
unsigned long ldata2;
DWORD _far func1()
{
        return ldata1;
}
void func()
{
     ldata2 = func1();
}
```

関数 func1()の返り値を unsigned long 型ではなく int 型とみなすので、変数 ldata2 の値は 0x12348765 ではなく 0xffff8765 となります。

【回避策1】型指定子と型修飾子を含む typedef 名を使うことにより回避できます。

```
typedef struct t1 _far STR1;
STR1 *func1()
{
        return stp1;
}

STR1 func2();
STR1 func2()
{
     return st1;
}
```

【回避策 2】 型修飾子の記述位置を変えることにより回避できます。

```
/* プロトタイプ宣言なし */
_far struct t1 *func1()
{
    return stp1;
}

/* プロトタイプ宣言あり */
_far struct t1 func2();
_far struct t1 func2()
{
    return st1;
}
```

【改善策】V1.10 で修正しました。

No. 7 割り込み関数(#pragma vect/#pragma interrupt)の拡張機能を記述した C ソースで、関数定義が 1 つも無い場合、アセンブル時にエラーが発生する制限

【内 容】割り込み関数(#pragma vect/#pragma interrupt)の拡張機能を記述した C ソースで、関数定義が 1 つも無く、 デバッグ情報出力時(-g)かつ、アセンブラ・ソース・モジュール・ファイルに、出力指定時(-a/-sa)に、対応するベクタテーブルと C ソースコメントが出力されません。アセンブル時にはエラーが発生します。
オブジェクト・モジュール・ファイルには正しく出力されます。
【例】

[vect.c]

```
#pragma interrupt INTP0 func
/* func()は別モジュールで定義 */
int dmy;
```

[vect_stub.c]

```
_interrupt void func(void) {}
void main(void) {}
```

[アセンブル時のエラーメッセージ]

```
vect.asm(42) : RA78K0R error E2404: Public symbol is undefined '_@vect08'
```

【回避策】以下のいずれかの方法で回避できます。
・#pragma vect(interrupt)を記述したソースには、関数を 1 つ以上定義してください。
・アセンブラ・ソース・モジュール・ファイル出力指定(-a/-sa)を指定しないでください。

[回避策の例]

```
#pragma interrupt INTP0 func
/* func()は別モジュールで定義 */
void dmy_func() { }
```

【改善策】V1.10 で修正しました。

No. 8 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】V2.00 で修正しました。

No. 9 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】V2.10 で修正しました。

No. 10 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】V2.10 で修正しました。

No. 11　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】V2.10 で修正しました。

No. 12 配列アドレスのポインタへの代入で、ポインタが指す配列の構造と、アドレスを参照した配列の構造が同じである記述に対し、ワーニング W0411を出力する制限

【内　容】配列アドレスのポインタへの代入で、ポインタが指す配列の構造と、アドレスを参照した配列の構造が同じである記述に対し、ワーニング W0411 を出力します。

【例】

```
char a1[2];
char a2[2][3];
char (*p1)[2];
char (*p2)[2][3];
char (*p3)[3];
void func()
{
      p1 = &a1;        /* 正しい記述だがW0411が出る */
      p2 = &a2;        /* 正しい記述だがW0411が出る */
      p3 = &a2[1];     /* 正しい記述だがW0411が出る */
}
```

【回避策】以下のように配列のアドレスを参照しない方法で、W0411 の出力を回避できます。

```
char a1[2];
char a2[2][3];
char *p4;
char (*p5)[3];
void func()
{
      p4 = a1;
      p5 = a2;
}
```

※不具合例の 3 番目を回避する方法はありません。

【改善策】V1.10 で修正しました。

No. 13 ミディアム・モデルまたはラージ・モデルでアセンブラ・ソース出力時に、分岐命令最適化で C0101 エラーとなる場合がある制限

【内　容】ミディアム・モデルまたはラージ・モデルで、アセンブラ・ソース出力時に、br !!addr20 → br $!addr20 の分岐命令最適化で C0101 エラーとなる場合があります。
C0101 エラーが発生しなければ、CC78K0R が出力したオブジェクト・モジュール・ファイルとアセンブラ・ソース・ファイルは正常です。

【回避策】CC78K0R からアセンブラ・ソース・ファイルを出力しないでください。
【改善策】V1.10 で修正しました。

No. 14 directmap変数をショート・ダイレクト・アドレシング可能な領域とSFRアドレシング可能な領域にまたがって配置すると、コード不正となる場合がある制限

【内　容】directmap 変数をショート・ダイレクト・アドレシング可能な領域と SFR アドレシング可能な領域にまたがって配置すると、コード不正となる場合があり、アセンブルすると、Operand out of range (saddr) や Illegal operand などのエラーが発生します。

【例】

```
 #define ADDR      0xfff1f
 _directmap struct t1 {
     char c1;
     char c2;
  } dst1 = { ADDR };
_directmap int di1 = ADDR;
void func()
{
      ++dst1.c2;
      di1 += 2;
}
```

```
[.asm]
; line      13 :       ++dst1.c2;
       inc        _dst1+1              ; コード不正
; line      14 :       di1 += 2;
       add        _di1,#02H
       addc       _di1+1,#00H          ; コード不正
```

【回避策】directmap 変数をショート・ダイレクト・アドレシング可能な領域と SFR アドレシング可能な領域にまたがって配置しないでください。

【改善策】V1.10 で修正しました。

No. 15 typedef 名を使った名前なし配列型パラメータ宣言を記述すると、E0301 エラーとなる制限

【内　容】typedef 名を使った名前なし配列型パラメータ宣言を記述すると、E0301 エラーとなります。

【例】

```
typedef int TYP;
void func(TYP [4]); /* E0301: Syntax error */
```

【回避策】名前付き配列型パラメータ宣言を記述してください。

```
typedef int TYP;
void func(TYP a[4]);
```

【改善策】V1.10 で修正しました。

No. 16 無条件分岐命令がコード不正となる制限

【内　容】以下の無条件分岐命令がコード不正となります。出力アセンブラ・ソースは正常です。

(1) callt/割り込み関数中の BR !addr16 命令
分岐先が BR 命令の次のアドレスから −128 ～ +127 バイトを超えた場合

(2) far 関数中の BR !!addr20 命令
分岐先が BR 命令の次のアドレスから −32768 ～ +32767 バイトを超えた場合

【回避策】CC78K0R が出力したアセンブラ・ソースを RA78K0R に通してオブジェクト・モジュール・ファイルを作成してください。

【改善策】V1.10 で修正しました。

No. 17 volatile 修飾型を型変換すると、volatile 修飾のコードを出力しない場合がある制限

【内　容】volatile 修飾型を型変換すると、volatile 修飾のコードを出力しない場合があります。
【例】のように、adrs[id] を SFR のアドレスとして扱う場合、意図したコードになりません。
この場合、Word アクセスになりません。

【例】

```
volatile unsigned short *adrs[ ] = {
        0xff40,
        0xff42,
        0xff44,
};
int id;
unsigned short s1;
void func()
{
        s1 = *adrs[id] & 0x8000;
}
```

【回避策】演算数を volatile 修飾型で明示的にキャストしてください。

```
s1 = *(volatile unsigned int *)adrs[id] & 0x8000;
```

と記述する。

【改善策】V1.20 で修正しました。

No. 18 -qj オプション未指定時のデバッガ上の表示が不正となる場合がある制限

【内　容】-qj オプション未指定時、以下の (1) または (2) を満たすと、デバッガ上の表示が不正となります。

(1) ネストした内側の if-else 文の 1 番目の副文(真の部分)を通って if-else 文を抜ける時に、実行しない 2 番目の副文(偽の部分)を通る。
ただし、else 文が空文の時は発生しない。

(2) ネストした内側の else なしの if 文の副文(真の部分)を通らずに if 文を抜ける時に、実行しない副文(真の部分)を通る。

【例 1】

```
int a=1, b=1, c;
void func1()
{
    if (a) {
        if (b) {
            c = 1;
        }
        else {
            c = 2;
        }
    }
    else {
        c = 3;
    }
}
```

【例 2】

```
int a=1, b=0, c=0;
void func2()
{
    if (a) {
        if (b) {
            c = 1;
        }
    }
    else {
        c = 2;
    }
}
```

【回避策】 -qj オプションを指定してください。

【改善策】V1.10 で修正しました。

No. 19 far 配置の char 配列型(signed,unsigned も含む)の directmap 変数の配置アドレス指定に、文字列定数を記述してもエラーとならない制限

【内　容】far 配置の char 配列型(signed,unsigned も含む)の directmap 変数の配置アドレス指定に、文字列定数を記述してもエラーにならず、0 番地に配置します。
ラージ・モデルであれば、_far 修飾子がない場合でも該当します。

【例】

```
_directmap _far char cary[10] = {"string"};
char c;
void func(void) {
      c = cary[5];
}
```

【回避策】 配置アドレスを整数で指定してください。
【改善策】 V2.00 で修正しました。

No. 20 関数内に typedef 宣言、構造体型宣言、共用体型宣言、enum 型宣言 のいずれかがある場合、コード不正になる制限

【内　容】関数内に typedef 宣言、構造体型宣言、共用体型宣言、enum 型宣言 のいずれかがある場合、HL レジスタを破壊するコードを生成します。

【コード不正イメージ】
フレームを生成しない(HL レジスタの退避/復帰が不要な)関数に対して、不要なフレームポインタ設定コードが生成されます。

```
_func1:
    push    ax
    push    bc
    movw    hl,sp   ★ 破壊コード
```

【該当例】
下記 func1～func5 のような記述が不具合に該当します。
割り込み関数(__interrupt)の指定が無い場合も、制限事項に該当します。
（例1）

```
__interrupt void func1() {
        typedef int T1;
}

__interrupt void func2() {
        struct S1 {
            int i;
        };
}

__interrupt void func3() {
        union U1 {
            int i;
        };
}

__interrupt void func4() {
        enum E1 {
            E11
        };
}
```

func1～4 は、制限事項内容に該当します。

（例2）
```
__interrupt void func5() {
        static union U2 {
                int i;
        } u2;
}
```
関数内static変数 u2 は フレーム内部に作成されないため、該当します。

【非該当条件】

以下の1), 2), 3)のいずれかの条件に該当した場合には不正なコードは生成しません。

1) __flashf関数、RTOS割り込み関数、RTOSタスク関数の場合
(V1.00 は RTOS割り込み関数、RTOSタスク関数をサポートしていません)

2) -qr オプションを指定していない場合(デフォルトは-qrオプション指定はオフになります)
以下のいずれかのとき該当しません。
2-1) 割り込み関数の場合は、自動変数がある、または関数呼び出しがある。
2-2) 通常関数の場合は、引数または自動変数がある。

3) -qr オプションを指定した場合(デフォルトは-qrオプション指定はオフになります)
以下のいずれかのとき該当しません。
3-1) 割り込み関数の場合は、関数呼び出しがある。
3-2) 割り込み関数、通常関数共に、引数または自動変数がある場合で、内部的に
その引数または自動変数がHLレジスタに割り付けられた場合(*)

(*) -qrオプションを指定した場合には、引数または自動変数がHLレジスタ以外の汎用レジスタまたは内部メモリに割り付けられることがあります。
その場合、チェックツールを使用して確認することができます。アセンブラソースで直接確認していただくことも可能です。

【非該当例】
```
__interrupt void func1() {
        typedef int T1;
        func6();
}
```
割り込み関数中に他の関数呼び出しが存在するため、該当しません。

```
void func2() {
        struct S1 {
                int i;
        } s1;
}
```
自動変数s1がフレーム内部に作成されるため、該当しません。

```
int gi;
void func3() {
        union U1 {
                int i;
        };
        gi = TOE0L.3;
}
```
関数内にHLレジスタを使用するコードがあれば、該当しません。

【回避策】 typedef 宣言、構造体型宣言、共用体型宣言、enum 型宣言 を関数外に記述してください。
割り込み関数(__interrupt)の指定が無い場合も同じ対処を行ってください。

（例1）

```
typedef int T1;                          /* typedef宣言を関数外に記述 */
__interrupt void func1() {
}


struct S1 {                               /* 構造体型宣言を関数外に記述 */
        int i;
};
  __interrupt void func2() {
}


  union U1 {                              /* 共用体型宣言を関数外に記述 */
          int i;
 };
  __interrupt void func3() {
}


enum E1 {                                 /* enum型宣言を関数外に記述 */
      E11
};
__interrupt void func4() {
}
```

（例2）

```
union U2 {                               /* 共用体型宣言を関数外に記述 */
        int i;
};
__interrupt void func5() {
        static union U2 u2;
}
```

【改善策】 V1.20 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、弊社営業または特約店にお問い合わせください。

No. 21 -ra または-rc オプション指定時の間接参照でコード不正になる制限

【内　容】以下の条件をすべて満たすときにコード不正になります。

（1）-ra または-rc オプションを指定

（2）任意のオブジェクトのポインタのポインタでオブジェクトが far 配置
（または far 修飾子を外して、-ml 指定）

（3）オブジェクトのポインタに代入後、同一式で再利用

（例1）

[.c]

```
_far char **p;
_far char *cp;
void func(void)
{
        cp = ++(*p);
}
```

“++(*p)”のさす先(アドレス)が不正になります。
変数 cp へ代入する値が正しくありません。

[.asm]

```
 ; line          5 :        cp  =  ++(*p);
   movw      de, !_p
   mov       a, [de+2]
   mov       _@SEGAX, a
   movw      ax, [de]
   incw      ax
   movw      [de], ax
                                  ; xch     a, x  が出ていない
   movw      !_cp, ax
   mov       a, _@SEGAX
   mov       !_cp+2,a
```

（例 2）

[.c]

```
void func(void)
{
        _far char **tAu;
        *tAu[0]++ = 1;
}
```

“*tAu[0]++”のさす先(アドレス)が不正のため、不正なアドレスに値を代入します。

【回避策】 以下のいずれかで回避してください。
（1） 代入と、前置／後置インクリメント、デクリメントを分けて記述する
（2） -ra または-rc オプションを無効にしてください。

（例 1 の回避策）
[.c]
```
_far char **p;
_far char *cp;
void func(void)
{
        ++(*p);
        cp = (*p);
}
```

（例 2 の回避策）
[.c]
```
void func(void)
{
        _far char **tAu;
        *tAu[0] = 1;
        tAu[0]++;
}
```

【改善策】 V2.00 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、弊社営業または特約店にお問い合わせください。

No. 22 配列またはポインタの ES 設定コードが出力されない場合がある制限

【内　容】条件 1、条件 2、条件 3 のいずれかをすべて満たす場合に、配列またはポインタの ES 設定コードが出力されない場合があります。

[条件 1]

（1） 二項演算の左辺が int または unsigned int になる式
(変数は対象外、ポインタや配列を使用した式も対象外)

（2） 二項演算の右辺 unsigned char の far 配列(静的にアドレスが決められない)、
または unsigned char の far ポインタ

（3） 二項演算子が+, −, &, |, ^のいずれか
(+, &, | , ^ の場合は左辺と右辺を入れ替えても該当)

（4） スタック変数と HL レジスタに割りついた変数が無い
(HL レジスタがスタックのフレームポインタ、またはレジスタ変数として使用の場合は非該当)

(条件 1 の例)

[.c]

```
unsigned char _far ar1[5];
unsigned int idx1, x1, x2;
void func(void)
{
        x1= (x2 << 1)  + ar1[idx1] ;
}
```

[.asm]

```
; line        5 :          x1 = (x2 << 1)  + ar1[idx1];
 movw      ax, !_idx1
 addw       ax, #loww (_ar1)
 movw      hl, ax
 movw      ax, !_x2
 addw       ax, ax
 xch        a, x
                                          ; mov    ES,#highw (_ar1) が出ていない
 and        a, ES:[hl]
 xch        a, x
 addc       a, #00H                       ; 0
 movw      !_x1, ax
```

[条件 2]
（1） 関係演算の一方のオペランドが静的に値を特定できる式であり、値と関係演算子が以下の組合わせである

演算子	演算型	値※
>, <=	signed char signed short signed int signed long	−1
	unsigned char	0x7f
	unsigned int unsigned short	0x7fff
	unsigned long	0x7fffffff
>=, <	signed char signed short signed int signed long	0
	unsigned char	0x80
	unsigned int unsigned short	0x8000
	unsigned long	0x80000000

※整数値が右項の場合。左項になると演算子が反転する。

（2） 関係演算の他方のオペランドが整数型の far 配列(静的にアドレスが決められない)または far ポインタ
（3） スタック変数と HL レジスタに割りついた変数が無い
（HL レジスタがスタックのフレームポインタ、またはレジスタ変数として使用の場合は非該当）

[条件 3]
（1） 等値演算の一方のオペランドが静的に値を 0 と特定できる式
（2） 等値演算の他方のオペランドがビットフィールドメンバを含む構造体型、共用型の far 配列(静的にアドレスが決められない)または far ポインタ
（3） スタック変数と HL レジスタに割りついた変数が無い
（HL レジスタがスタックのフレームポインタ、またはレジスタ変数として使用の場合は非該当）

（条件 2 と 3 の例）
[.c]
```
struct st {
                int b0:1;
                int b1:2;
} _far *pbf;
int _far *p;
unsigned char uc;
unsigned int   x1;
void func(void)
{
      x1= (x1 + 1) – ((x1 + 1) – (*p < 0)) ;                    /* 条件 2 */
      x1= (x1 + 1) – ((x1 + 1) – (pbf->b0 == (uc = 0))) ;       /* 条件 3 */
}
------------------------
```

【回避策】 関数内にダミーの自動変数を定義してください。

（条件1の例）

```
[ .c ]
unsigned char _far ar1[5];
unsigned int idx1, x1, x2;
void func(void)
{
        unsigned char dummy;                /* ダミーの自動変数を定義 */
        x1= (x2 << 1) + ar1[idx1];
}
--------------------------
```

【改善策】 V2.00 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、弊社営業または特約店にお問い合わせください。

No. 23 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/unsigned char 型の配列の初期値定義は、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, "abc" };
char *const TBL[3] = { a1 };
char *const *ptr1;
void func()
{
        ptr1 = TBL;
        **ptr1 = 0x12
}
-------
[.asm]
@@CNST          CSEG    MIRRORP
_a1:    DB      01H     ; 1
        DB      'ab'
TBL:    DW      low (_a1)           ; _TBL が奇数アドレス
        DB      (4)

@@DATA          DSEG    BASEP
_ptr1:  DS      (2)

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

```
@@CODEL      CSEG
_func:
; line      6 :    ptr1 = TBL;
    movw          ax,#loww (_TBL)    ; _TBL が奇数アドレス
    movw          !_ptr1,ax
; line      7 :    **ptr1 = 0x12;
    movw          de,ax
    movw          ax, [de]           ; 奇数アドレスを参照
    movw          de,ax
    mov           [de+0], #012H
```

【回避策】 初期値を正しく記述してください。

```
[ .c ]
const char a1[ ] = { 0x01, 'a', 'b', 'c', '¥0' };
char *const TBL[3] = { a1 };
char *const *ptr1;
void func()
{
        ptr1 = TBL;
        **ptr1 = 0x12;
}
```

【改善策】 V2.10 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、弊社営業または特約店にお問い合わせください。

No. 24 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になります。
(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】 V2.10 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、弊社営業または特約店にお問い合わせください。

No. 25 -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 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 型変数」と「0～255 の定数」

・「0～255 の定数を代入した unsigned char 型変数」同士

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

(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;
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】 V2.10 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、弊社営業または特約店にお問い合わせください。

No. 26 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】 V2.10 で修正しました。

No. 27 -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】 V2.12 で修正しました。

No. 28 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】 V2.12 で修正しました。

No. 29 オペランドがword[BC]の命令(mov, movw)におけるアクセスの制限

【内　容】 以下の条件(1)あるいは(2)のいずれかの場合、出力コードが不正となります。

オペランド"word[BC]"で word+BC のアドレスが 10000H を超えると意図しないアドレスをアクセスします。

(1) 「ポインタ−定数」の形で減算を使用して間接参照する場合

ただし、以下の条件は除きます。

(a) ラージ･モデルを使用し、ポインタの指す先を near 領域指定していない場合

(b) ポインタの指す先を far 領域指定して使用している場合

(c) ポインタの指すアクセス先サイズが 1 バイトで、減算する定数が 1 の場合

(d) ポインタの指すアクセス先サイズが 1 バイトで、減算する定数が 2、最適化条件としてスピード優先最適化[注]を行っていない場合

(e) ポインタの指すアクセス先サイズが 2 バイトで、減算する定数が 1、最適化条件としてスピード優先最適化[注]を行っていない場合

注：−qx1 を指定している場合、または−q オプションで"l"を指定していない場合

(2) 配列の要素を参照する際に、添字の式中の変数が負の値となる場合、もしくは「集成体*1を示すアドレス+オフセット(負の値となる変数を含む式)」の形で間接参照する場合

*1 構造体、配列です。

【条件(1)の例】

```
-------*.c-------------
unsigned char x[5], *ucp1;
unsigned short y[5], *usp1;
signed short si1;
void func1()
{
        /*********ポインタのアクセス先のサイズが 1 バイトの場合**********/
 x[0] = *(ucp1 - 1);       /* 問題なし */
 x[1] = *(ucp1 - 2);       /* スピード優先最適化指定時は問題発生、未指定時は問題なし */
 x[2] = *(ucp1 - 3);       /* 問題発生 */
 x[3] = ucp1[-4];          /* 問題発生 *(ucp1 −4)と同じ意味となる記述 */
 x[4] = *(ucp1 + si1 - 5); /* 問題発生 */
        /*********ポインタのアクセス先のサイズが 2 バイトの場合**********/
 y[0] = *(usp1 - 1);       /* スピード優先最適化指定時は問題発生、未指定時は問題なし */
 y[1] = *(usp1 - 2);       /* 問題発生 */
 y[2] = *(usp1 - 3);       /* 問題発生 */
 y[3] = usp1[-4];          /* 問題発生 *(usp1 −4)と同じ意味となる記述 */
 y[4] = *(usp1 + si1 - 5); /* 問題発生 */
 }
 --------*.asm---------(上記のアセンブル･ソースの抜粋)
 ; line    25 :   x[2] = *(ucp1 - 3);    /* NG */
 $DGL   0,9
        movw  bc,!_ucp1       ;[INF] 3, 1
        mov   a,65533[bc]     ;[INF] 3, 1 ← 不正なアドレスをアクセスする可能性がある
        mov   !_x+2,a         ;[INF] 3, 1
             :
---------------------
```

【条件(2)の例】

```
 unsigned char x[4], *ucp1, uca1[10];
 signed short sidx;
 signed char cidx;
 void func2()
 {
   x[0] = uca1[cidx];          /* cidx が負の値で問題発生、正の値(0 含む)では問題なし */
   x[1] = *(&uca1[3] + cidx); /* cidx が負の値で問題発生、正の値(0 含む)では問題なし */
   x[2] = uca1[sidx];          /* sidx が負の値で問題発生、正の値(0 含む)では問題なし */
   x[3] = *(&uca1[4] + sidx); /* sidx が負の値で問題発生、正の値(0 含む)では問題なし */
 }
```

--------*.asm---------(上記のアセンブル・ソースの抜粋)

```
; line    36 :      x[0] = uca1[cidx];            /* cidx が負なら NG */
 $DGL  0,20
       mov     a,!_cidx       ;[INF] 3, 1
       sarw    ax,8           ;[INF] 2, 1
       movw    bc,ax          ;[INF] 1, 1
       mov     a,_uca1[bc]    ;[INF] 3, 1  ← 不正なアドレスをアクセスする可能性がある
       mov     !_x,a          ;[INF] 3, 1
--------------------
```

【回避策】

・条件(1)の回避策

「ポインタ-定数」の形で減算を使用して間接参照する場合は「ポインタ-定数」の結果を一度別のポインタに代入してからアクセスしてください。

例 1:

<回避前>	<回避後>

unsigned char x, *ucp1;
void func1()
{
x = *(ucp1 - 3);
 	
}
	unsigned char x, *ucp1, *ucp2;
	void func1()
	{
	ucp2 = ucp1 - 3;
	x = *ucp2;
	}

例 2:

<回避前>	<回避後>

unsigned char x, *ucp1;
void func2()
{
x = ucp1[-4];
 	
}
	unsigned char x, *ucp1, *ucp2;
	void func2()
	{
	ucp2 = ucp1 - 4;
	x = *ucp2;
	}

・条件(2)の回避策

配列の添字が符号付きの式で配列の要素を参照する際に添字の式中の変数が負の値となる場合、もしくは「集成体を示すアドレス+オフセット(負の値となる変数)」の形で間接参照する場合、上記の式の演算結果をポインタに代入してからアクセスしてください。

例 1:

<回避前>	<回避後>

unsigned char x, uca1[10];
signed char cidx;
void func1()
{
x = uca1[cidx];
 	
}
	unsigned char x, uca1[10], *ucp1;
	signed char cidx;
	void func1()
	{
	ucp1 = uca1 + cidx;
	x = *ucp1;
	}

例 2:

<回避前>	<回避後>

unsigned char x, uca1[10];
signed char cidx;
void func2()
{
x = *(&uca1[3] + cidx);
 	
}
	unsigned char x, uca1[10], *ucp1;
	signed char cidx;
	void func2()
	{
	ucp1 = &uca1[3] + cidx;
	x = *ucp1;
	}

【改善策】 V2.13 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. 30 -qjオプション指定に関するC0101エラーとなる制限
【内　容】 以下の条件をすべて満たすときに、内部エラー（C0101）になります。
(1) ジャンプ最適化オプション（-qj）を有効にする。
(2) if(0)、あるいは if(1)の else 節など、実行しない処理中で以下のいずれかを記述する。
- 配列、構造体、共用体の初期化処理
- テーブルジャンプを伴う switch 文

```
[*.c]
struct t1 {
      unsigned char uc1;
};
char c;
void func1()
{
    struct t1 st0 = {0x07};
    if(0) {
         struct t1 st0 = {0x08};     /* 実行しない処理 */
         c++;
    }
 }
```

【回避策】 以下のいずれかで回避してください。
・ ジャンプ最適化オプション(-qj)を無効にしてください。
・ 実行しない処理自体を削除する、あるいは#if 0 で囲んでコンパイル対象としないでください。
【改善策】 V2.30 で修正いたします。

No. 31 far領域のアドレスを long/unsigned long 型にキャストしたときの制限

【内　容】far 領域のアドレスを long/unsigned long 型にキャストした値を初期値とすると、アドレスの最上位 1 バイトの値が 0FH 固定となります。

```
[*.c]
__far int const fi1 = 5;
__far unsigned long const ula1[] = { &fi1 };
__far unsigned long const ula2[] = { (unsigned long)&fi1 };

[*.asm]
@@CNSTL  CSEG  PAGE64KP
_fi1:    DW    05H    ; 5
_ula1:   DG    _fi1
_ula2:   DW    loww (_fi1)
         DW    0FH    ; 15
```

【回避策】アドレスが初期値の時は、long/unsigned long 型にキャストしないでください。

【改善策】V2.30 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、販売店または特約店にお問い合わせください。

No. 32 ##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】V2.30 で修正いたします。

No. 33 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】V2.30 で修正いたします。

No. 34 ESレジスタの設定コードに関する制限

【内　容】 間接参照を除く浮動小数点型変数のインクリメント／デクリメント演算後に far 変数を参照すると、ES レジスタの設定コードが出ない場合があります。

```
[*.c]
__far char c1;
float f1;
void func()
{
        ++f1;
        c1 = 5;
}
```

【回避策】 間接参照を除く浮動小数点型変数のインクリメント／デクリメント演算直後にダミーのポインタ間接参照式を記述してください。

```
[*.c]
__far char c1;
float f1;
void func()
{
        char *cp1;          /* ダミー変数*/
        ++f1;
        *cp1;               /* ダミーの間接参照式 */
        c1 = 5;
}
```

【改善策】 V2.30 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、販売店または特約店にお問い合わせください。

以上