LAPIS SEMICONDUCTOR

FJXTCCU8

CCU8
ユーザーズマニュアル

プログラム開発支援ソフトウェア

2013 年 8 月発行

ご注意

本資料の一部または全部をラピスセミコンダクタの許可なく、転載・複写することを堅くお断りします。

本資料の記載内容は改良などのため予告なく変更することがあります。

本資料に記載されている内容は製品のご紹介資料です。ご使用にあたりましては、別途仕様書を必ずご請求のうえ、ご確認ください。

本資料に記載されております応用回路例やその定数などの情報につきましては、本製品の標準的な動作や使い方を説明するものです。したがいまして、量産設計をされる場合には、外部諸条件を考慮していただきますようお願いいたします。

本資料に記載されております情報は、正確を期すため慎重に作成したものですが、万が一、当該情報の誤り・誤植に起因する損害がお客様に生じた場合においても、ラピスセミコンダクタはその責任を負うものではありません。

本資料に記載されております技術情報は、製品の代表的動作および応用回路例などを示したものであり、ラピスセミコンダクタまたは他社の知的財産権その他のあらゆる権利について明示的にも黙示的にも、その実施または利用を許諾するものではありません。上記技術情報の使用に起因して紛争が発生した場合、ラピスセミコンダクタはその責任を負うものではありません。

本資料に掲載されております製品は、一般的な電子機器（AV 機器、OA 機器、通信機器、家電製品、アミューズメント機器など）への使用を意図しています。

本資料に掲載されております製品は、「耐放射線設計」はなされておりません。

ラピスセミコンダクタは常に品質・信頼性の向上に取り組んでおりますが、種々の要因で故障することもあり得ます。

ラピスセミコンダクタ製品が故障した際、その影響により人身事故、火災損害等が起こらないようご使用機器でのディレーティング、冗長設計、延焼防止、フェイルセーフ等の安全確保をお願いします。定格を超えたご使用や使用上の注意書が守られていない場合、いかなる責任もラピスセミコンダクタは負うものではありません。

極めて高度な信頼性が要求され、その製品の故障や誤動作が直接人命を脅かしあるいは人体に危害を及ぼすおそれのある機器・装置・システム（医療機器、輸送機器、航空宇宙機、原子力制御、燃料制御、各種安全装置など）へのご使用を意図して設計・製造されたものではありません。上記特定用途に使用された場合、いかなる責任もラピスセミコンダクタは負うものではありません。上記特定用途への使用を検討される際は、事前にローム営業窓口までご相談願います。

本資料に記載されております製品および技術のうち「外国為替及び外国貿易法」に該当する製品または技術を輸出する場合、または国外に提供する場合には、同法に基づく許可が必要です。

Windows は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。また、その他の製品名や社名などは、一般に商標または登録商標です。

ラピスセミコンダクタ株式会社

〒222-8575 神奈川県横浜市港北区新横浜二丁目 4 番 8 号

http://www.lapis-semi.com/jp/

目　次

1. 概要

C 言語は強力な汎用プログラミング言語で、効率的で小さく、ポータビリティのあるコードを生成できます。C 言語は、小さいので管理しやすく、豊富な演算子による柔軟性と新しい制御フローとデータ構造が利用できるという強力さを持っています。

CCU8 は、最適化 C コンパイラです。また、C 言語の基本的機能をサポートし、通常の C コンパイラにある機能を持っています。

CCU8 コンパイラは、1 つ以上の入力 C ソースファイルをコンパイルし、各入力ファイルに対してアセンブリファイルを出力します。入力ソースファイルは、標準の C ソースプログラムを含むテキストファイルです。出力ファイルは、再配置可能なアセンブリコードを含むテキストファイルです。

CCU8 の特徴は次のとおりです。

1. CCU8 のサポートする C 言語は、ANSI 標準に準拠して実装されています。アーキテクチャの制約から標準と異なる部分もあります。
2. さまざまなコマンドラインオプションがあります。
3. 割り込み処理ルーチンを記述する機能を用意しています。
4. アーキテクチャ依存の機能を利用できるようにプラグマをいくつか提供しています。
5. **long** 型、**float** 型および **double** 型をサポートするエミュレーションライブラリを提供しています。
6. ミックスドランゲージプログラミングを行うことができます。

2. 操作環境

2.1 ハードウェア

ハードウェア　　：　IBM-PC/ AT/ Pentium 互換機およびクローン機

オペレーティングシステム　：　Windows XP/Vista/7

2.2 システム構成

CCU8 では、次の情報を **CONFIG.SYS** ファイルに記述する必要があります。

files=30

必ずこの情報を CONFIG.SYS ファイルに追加してから CCU8 を呼び出してください。

2.3 環境変数

CCU8 は、2 つの環境変数 INCLU8 と TMP を使用します。

INCLU8 は、#include 前処理指令で指定されたインクルードファイルを検索するディレクトリを指定するのに使用できます。

INCLU8 環境変数は、DOS の SET コマンドで定義できます。SET コマンドは次の書式で記述します。

SET INCLU8=path

CCU8 は、コンパイル中にテンポラリファイルを使用します。これらのテンポラリファイルへのパスは、TMP 環境変数で指定することができます。**autoexec.bat** ファイルに次の一行を記述すると、CCU8 はコンパイル中に、指定された **path** にテンポラリファイルを作成します。

SET TMP=PATH

例 2.1

```
SET TMP=C:¥RAMDRIVE
```

CCU8 は、テンポラリファイルへのパスとして'C:¥RAMDRIVE'を使用します。

環境変数 TMP を指定していない場合、コンパイラはテンポラリファイルをカレントディレクトリに作成します。

2.4 インストール

CCU8 'C'コンパイラは、次の実行可能プログラムからなっています。

- CCU8
- CC1U8
- CC2U8

CCU8 はコンパイラのローダーで、**CC1U8** と **CC2U8** を実行します。この 3 つの実行可能プログラムは、同じディレクトリに存在する必要があります。

3. CCU8 の起動とコマンドラインオプション

3.1 CCU8 の起動

CCU8 は、次のようにコマンドラインを指定して起動します。

```
C:¥> CCU8 <CR>
C:¥> CCU8 @[path] response_file <CR>
C:¥> CCU8 /T string [options....] [path]file [file ....] <CR>
```

CCU8 は、CCU8 の実行可能プログラムと CC1U8/CC2U8 の実行可能プログラムとのバージョンの矛盾を検出するチェックを実行します。CCU8 の実行可能プログラムの製品バージョン番号が CC1U8 や CC2U8 の実行可能プログラムの製品バージョン番号と異なっている場合は、フェイタルエラーメッセージが表示されます。

`file` は、入力の C ソースファイルを指定し、".C"、".c"、".H"または".h"のいずれかの拡張子を持った名前です。それ以外の拡張子を発見すると、CCU8 はフェイタルエラーメッセージを表示してコンパイル処理は終了します。`file` にはパス名がついていてもかまいません。

CCU8 は、コマンドラインに指定されたファイルごとにアセンブリファイルを作成します。この出力ファイルには、U8 のアセンブリニーモニックが含まれます。

デフォルトでは、出力ファイルは入力ファイルと同じ名前で拡張子".asm"が付きます。デフォルトでは、出力ファイルは、カレントの作業ディレクトリに作成されます。

コマンドラインオプションには、次のものが指定できます。

/T	TYPE 擬似命令のオペランド文字列を指定する
/MS	Small メモリモデル(デフォルト)
/ML	Large メモリモデル
/near	near データアクセス(デフォルト)
/far	far データアクセス
/nofar	far データアクセスを制限する
/Ot	実行速度の最適化を行う
/Ol	ループの最適化を行う(デフォルト)
/Om	最大限の最適化を行う
/Og	広域的最適化を行う(デフォルト)
/Od	最適化を行わない
/Oa	別名チェックを行う
/Orp	関数のレジスタ退避・復帰を共通化する
/Orpn	関数のレジスタ退避・復帰を共通化しない
/LE	リストファイルを生成する
/Fa	アセンブリリストファイルを出力する
/CT	コールツリーリストファイルを出力する
/Zg	関数プロトタイプリストファイルを生成する
/LP	前処理の結果をファイルに出力する
/I	インクルードファイルのディレクトリを指定する
/PC	前処理の結果をコメント付きで出力する
/D	マクロを定義する
/U	マクロを未定義にする
/ST	スタックプローブルーチンを生成する
/SS	スタックサイズを変更する
/SD	デバッグ情報を生成する
/SL	識別子の最大長を変更する
/J	デフォルトの char 型を unsigned char 型とする
/PF	プラグマ引数の区切り文字としてコンマを使用する
/SYS	コンパイラのセグメント命名規則を変更する
/W	指定したレベルのワーニングを表示する
/Wc	指定したワーニングをエラーに変更する

/Wa	すべてのワーニングをエラーに変更する
/Za	拡張機能を無効にする
/NOWIN	ROM WINDOW 領域を使用しない
/Ff	高速エミュレーションライブラリを使用する
/KJ	文字列内の SHIFT-JIS 漢字をサポートする
/Lv	ローカル変数のレジスタ/スタック情報を出力する
/Zs	ローカル変数をスタック上に割り当てる
/Zp	構造体/共用体のパディングを削除する
/Zc	関数をファイル単位のセグメントに出力する
/V	バージョン情報を表示する
@	応答ファイルを指定する

コマンドラインオプションについては「3.2 コマンドラインオプション」で詳しく説明します。

起動時に次のように著作権についてのメッセージが表示されます。

```
CCU8 C Compiler, Ver.3.20
Copyright (C) 2008-2011 LAPIS Semiconductor Co., Ltd.
```

次のようにコマンドラインに入力します。

```
C:¥> CCU8 <CR>
```

次のように使用方法が表示されます。

```
CCU8 C Compiler, Ver.3.20
Copyright (C) 2008-2011 LAPIS Semiconductor Co., Ltd.

Usage:      CCU8 @[path]response_file
            CCU8 /T string [Options...] [path]filename...
            /T string Specify the operand string for TYPE instruction


                            -MEMORY MODEL-
/MS small model                         /near near data access specifier
/ML large model                         /far far data access specifier
/nofar restrict FAR access
                            -OPTIMIZATION-
/Ot optimize for speed                  /Ol enable loop optimizations
/Om enable maximum optimizations        /Og enable global optimizations
/Od disable optimizations               /Oa Enables alias checks
/Orp - enable optimization by register push/ pop routine
/Orpn - disable optimization by register push/ pop routine
                            -OUTPUT FILES-
```

```
/LE generate list file                  /Fa[filename] assembly listing file
/CT <filename> list calltree in a file  /Zg generate function prototypes

(Press <return> to continue)
                          -PREPROCESSOR-
/LP preprocessed output in a file       /I <directory> include file directory
/PC preprocessed output with comments   /D <identifier>[=[string]] define macro
/U <identifier> undefine macro
                              -STACK-
/ST generate stack probe routine        /SS <constant> change stack size
                              -DEBUG-
/SD generate debug information
                              -MISCELLANEOUS-
/J default char type is unsigned
/PF use comma as delimiter for pragma arguments
/SL<constant> change maximum identifier length
/SYS change compiler segment naming strategy
/W <constant> set warning level
/Wc <warning number> [,warning number,...]
       change the specified warning(s) to error(s)
/Wa change all warnings to errors
/NOWIN do not use ROM WINDOW area
/Ff use fast emulation library
/Za disable extensions
/KJ support SHIFT-JIS Kanji characters in strings
/Lv register/stack information for local variables
/Zs local variable on stack
/Zp Special packing for structure/union
/Zc do not generate separate segment per function
/V version information
@ <filename> response file
```

3.2 コマンドラインオプション

ここでは、コマンドラインに指定できるさまざまなオプションについて説明します。すべてのコマンドラインオプションでは、大文字/小文字を区別しています。/I、/Fa、/D、/U、/W オプションは、それぞれコマンドラインに 2 回以上指定することができます。/I、/Fa、/D、/U、/W 以外のオプションを 2 回以上指定すると、CCU8 はフェイタルエラーメッセージを表示します。また、/Fa、/D、/U、/W オプションは、コマンドラインのソースファイル間に指定することもできます。

3.2.1 マシンモデルオプション

ここでは、マシンモデルオプション/T について説明します。

3.2.1.1 文字列出力

構文: /T string

/T オプションにはどのような文字列でも指定できます。CCU8 は文字列をチェックしません。CCU8 は、指定された文字列を TYPE 擬似命令に使用したものをアセンブリファイルに出力します。このパラメータは、プリプロセッサオプション(/LP または/PC のいずれか)を指定した場合以外は、必ず指定しなければなりません。

例 3.1

```
C:¥> CCU8 /Tmu8 test.c  <CR>
```

上の例の/Tmu8 では、CCU8 が次のような TYPE 擬似命令を出力するよう指示しています。

```
type (mu8)
```

3.2.2 メモリモデルオプション

CCU8 では、次のメモリモデルをサポートしています。

1. Small メモリモデル

2. Large メモリモデル

メモリモデルは、対応するコマンドラインオプションで指定します。コマンドラインで指定できるメモリモデルのオプションは 1 つだけです。複数指定すると、CCU8 はフェイタルエラーメッセージを表示します。この節ではメモリモデルオプションについて詳しく説明します。

3.2.2.1 /MS オプション

構文 : /MS

/MS オプションは、Small メモリモデルでプログラムをコンパイルします。Small メモリモデルでは、コードに対して物理コードセグメントを 1 つ、データ(テーブルなど)に対して物理セグメントを 256 個使用します。このオプションは、デフォルトのメモリモデルオプションです。メモリモデルオプションをコマンドラインに指定していない場合、プログラムはこのモデルでコンパイルされます。

例 3.2

```
C:¥> CCU8 /Tmu8 /MS test.c <CR>
```

上の例での/MS コマンドラインオプションは、ソースファイル“test.c”を Small モデルメモリモデルでコンパイルするよう CCU8 に指示しています。

3.2.2.2 /ML オプション

構文 : /ML

/ML オプションは、Large メモリモデルでプログラムをコンパイルします。Large メモリモデルでは、コードに対して物理コードセグメントを 16 個、データ(テーブルなど)に対して物理セグメントを 256 個使用します。

例 3.3

```
C:¥> CCU8 /Tmu8 /ML test.c <CR>
```

上の例でのコマンドラインオプション/ML は、ソースファイル“test.c”を Large メモリモデルでコンパイルするよう CCU8 に指示しています。

例 3.4

```
C:¥> CCU8 /Tmu8 /MS /ML test.c <CR>
```

上の例のコマンドラインオプションには複数のメモリモデルオプションが指定されているため、CCU8 はフェイタルエラーを表示します。

3.2.2.3 /near オプション

構文：/near

/near オプションは、指定子が指定されていないすべてのデータ(テーブルなど)を near データとして扱うよう、CCU8 に指示します。このオプションは、デフォルトのデータアクセスオプションです。コマンドラインにデータアクセスオプションを指定していない場合も、プログラムはこのデータアクセスオプションでコンパイルされます。

例 3.5

```
C:¥> CCU8 /Tmu8 /near test.c <CR>
```

上の例のコマンドラインオプション/near は、ソースファイル“test.c”内で定義されているすべてのデータ(テーブルなど)を near データとして扱うよう CCU8 に指示しています。

3.2.2.4 /far オプション

構文：/far

/far オプションは、指定子が指定されていないすべてのデータ(テーブルなど)を far データとして扱うよう、CCU8 に指示します。

例 3.6

```
C:¥> CCU8 /Tmu8 /far test.c <CR>
```

上の例のコマンドラインオプション/far は、CCU8 にソースファイル“test.c”内で定義されているすべてのデータ(テーブルなど)を far データとして扱うよう指示しています。

例 3.7

```
C:¥> CCU8 /Tmu8 /near /far test.c <CR>
```

上のコマンドラインオプションでは、データアクセスオプションが複数指定されているため、CCU8 はフェイタルエラーを表示します。

3.2.3 最適化オプション

CCU8 で用意されている最適化の機能により、対象とする記憶容量や実行時間を削減することができます。これは、不必要な命令を削除したり、コードの並びを変えることによって行われます。

最適化を制御するオプションを次の表に示します。

表 3.1

オプション	最適化処理
/Od	最適化を無効にする
/Ol	ループの最適化を行う
/Og	広域的最適化を行う
/Oa	別名チェックを行う
/Om	最大限の最適化を行う
/Ot	実行速度の最適化を行う

次の最適化は、必ず実行されます。

1. 共通部分式の削除
2. 定数の畳み込み
3. 覗き穴最適化

この最適化は、入力プログラムの小さい一部分のみを調べて行われます。/Od オプションを指定してこれらの最適化を抑制することはできません。

次の最適化は、/Od オプションで抑制されなければ常に実行されます。

表 3.2
1. 冗長コードの削除
2. 冗長ブロックの削除
3. ジャンプの最適化
4. 代数恒等式を利用した最適化

他のオプションでは、これらの最適化を制御することはできません。

コマンドラインオプションを指定しない場合、デフォルトで以下の最適化が実行されます。

1. ループの最適化
2. 広域定数の畳み込み
3. 広域共通部分式の削除
4. 表 3.2 に示した最適化

3.2.3.1 /Od オプション

構文 : /Od

/Od オプションは、コンパイラに最適化を行わないよう指示します。ソースプログラムをデバッグ用にコンパイルする際に、このオプションが役に立ちます。なお、前述した最適化はおこなわれます。

このオプションでは、生成コードは大きくなり、実行時間も長くなります。

他の最適化オプションをこのオプションと同時に指定することはできません。指定すると、最適化オプションの不正な組み合わせを示すフェイタルエラーメッセージが表示されます。

例 3.8

```
C:¥> CCU8 /Od /Tmu8 test.c <CR>
```

上のコマンドラインでは、最適化されていない出力ファイル“test.asm”が作成されます。

例 3.9

```
C:¥> CCU8 /Od /Ol /Tmu8 test.c <CR>
```

上のコマンドラインでは、フェイタルエラーメッセージ“Illegal combination of optimization options”(最適化オプションの不正な組み合わせ)が表示されます。

3.2.3.2 /Og オプション

構文：/Og

/Og オプションを指定すると、CCU8 は広域的最適化のみを行います。CCU8 が行う広域的最適化には次のものがあります。

1. グローバル共通部分式の削除
2. 広域定数の畳み込み
3. コードの掘り下げ
4. コードの巻き上げ

/Og オプションによって、CCU8 は関数全体を調べて共通部分式の削除および定数の畳み込みを行います。

例 3.10

```
C:¥> CCU8 /Og /Tmu8 test.c <CR>
```

ループ最適化および別名チェックは、上のコマンドラインでは実行されません。ただし、表 3.2 で示した最適化は実行されます。

3.2.3.3 /Ol オプション

構文：/Ol

/Ol オプションを指定すると、CCU8 はループに関する最適化のみ実行します。

次のループ最適化を実行します。

1. ループ不変コードの移動
2. ループ変動コードの移動
3. ループの強さの軽減
4. 誘導変数の削除
5. ループの展開

例 3.11

```
C:¥> CCU8 /Ol /Tmu8 test.c <CR>
```

/Ol オプションによって、CCU8 はループの最適化を行います。広域的最適化および別名チェックは上のコマンドラインでは行われれません。ただし、表 3.2 で示したその他の最適化は行われます。

例 3.12

```
C:¥> CCU8 /Ol /Og /Tmu8 test.c <CR>
```

/Ol オプションによって、CCU8 はループの最適化を行います。また、/Og オプションで広域的最適化も行われます。別名チェックは上のコマンドラインでは行われません。表 3.2 で示したその他の最適化は行われます。

3.2.3.4 /Oa オプション

構文：/Oa

/Oa オプションによって、コンパイラは安全な最適化となるように別名チェックを行います。

別名とはプログラム内の同じメモリ位置を参照する名前(シンボリック参照)が複数あることです。/Oa オプションを指定すると、CCU8 は別名を検出して情報を保守します。最適化の際にこの情報が使用されます。

/Oa オプションを指定しなければ、出力サイズが小さくなるか、出力速度が速くなることがあります。ただし、/Oa オプションを使用して安全な出力を取得するようお勧めします。別名がプログラム内で使用されていない場合のみ、このオプションを無視してかまいません。

例 3.13

```
C:¥> CCU8 /Oa /Tmu8 test.c <CR>
```

/Oa オプションによって、CCU8 は別名チェックを実行します。ループの最適化および広域的最適化は上のコマンドラインでは行われません。表 3.2 で示したその他の最適化は行われます。

3.2.3.5 /Om オプション

構文 : /Om

/Om オプションによって、CCU8 は最大限の最適化を行います。/Om オプションを指定すると、CCU8 はそれ以上最適化が行えなくなるまで、すべての最適化を繰り返し行います。/Om を指定した場合には、/Og オプションおよび/Ol オプションは必要ありません。/Om を指定すれば広域的最適化およびループの最適化も行われるからです。

例 3.14

```
C:¥> CCU8 /Om  /Tmu8 test.c <CR>
```

/Om オプションによって、CCU8 はすべての最適化を繰り返し行います。別名チェックは行われません。

例 3.15

```
C:¥> CCU8 /Om /Og /Tmu8 test.c <CR>
```

/Om オプションによって、CCU8 はすべての最適化を繰り返し行います。/Om オプションで広域的最適化が実行されるので、上記のコマンドラインでの/Og オプションの指定は不要です。

3.2.3.6 /Ot オプション

構文 : /Ot

/Ot オプションによって、CCU8 は実行速度の最適化を行います。これによって CCU8 は広域的最適化、およびループの最適化も行います。デフォルトでは、別名チェックは行いません。実行速度の最適化を行うと出力コードのサイズが大きくなる場合があります。オプション/Om、/Ot および/Od の指定は、互いに排他的です。

例 3.16

```
C:¥> CCU8 /Ot /Tmu8 test.c <CR>
```

/Ot オプションによって、CCU8 は実行速度の最適化を実行します。

3.2.3.7 /Orp オプション

構文 : /Orp

/Orp オプションは、関数入口および出口でのレジスタ退避・復帰を共通化することをコンパイラに指示します。本オプションが指定されると、コンパイラは関数の入口で行うレジスタ退避

（プロローグ処理）および関数の出口で行うレジスタ復帰（エピローグ処理）を、それぞれサブルーチン呼び出しのコードとして生成します。

このオプションは、/Om オプション指定時にはデフォルトで指定されたものとみなします。/Od オプション指定時には、本オプションを指定することはできません（/Od と/Orp は相反するオプションであるためです）。/Od オプションと/Orp オプションを同時に指定した場合にはエラーを出力します。

/Orp オプションと後述の/Orpn オプションは同時に指定できません。これらのオプションを同時に指定した場合はエラーを出力します。

/Om、/Od オプション以外の場合は、本オプションを明示的に指定したときだけ有効となります。

注記:
/Orp オプションを指定した場合、コードサイズを削減することができますが、関数内で使用しないレジスタの退避・復帰も行うため、スタック消費量および関数の入口・出口の処理サイクル数は増加しますのでご注意ください（1 個の関数につき、スタック消費量は最大で 10 バイト増加、サイクル数は最大で 25 サイクル増加します）。

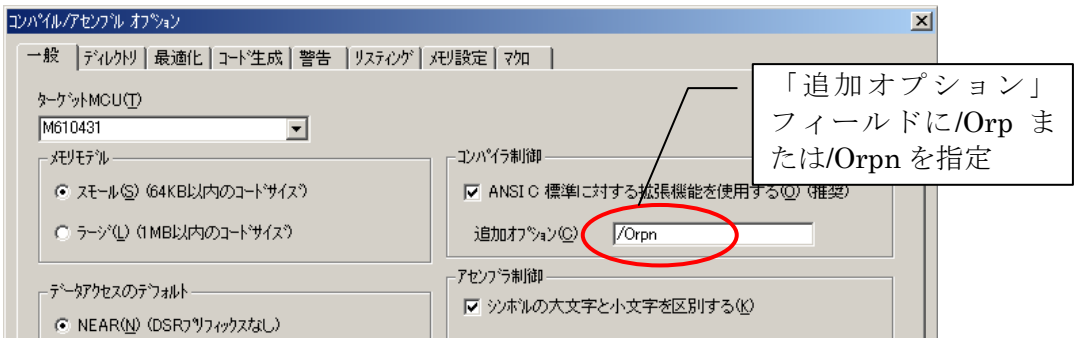
3.2.3.8 /Orpn オプション

構文：/Orpn

/Orpn オプションは、関数の入口・出口のレジスタ退避・復帰の処理を共通化しないことをコンパイラに指示します。本オプションが指定されたときは、関数内で使用するレジスタのみを退避・復帰するコードを生成します（CCU8 V3.31 以前と同様の出力となります）。

前述の/Orp オプションと/Orpn オプションは同時に指定できません。これらのオプションを同時に指定した場合はエラーを出力します。

注記:
/Orp オプションおよび/Orpn オプションを IDEU8 にて指定する場合は、「コンパイル/アセンブルオプション」ダイアログの「一般タブ」の「コンパイラ制御」の「追加オプション」フィールドに/Orp または/Orpn を指定してください。



3.2.3.9 最適化オプションのまとめ

/Oa オプションと他の最適化オプションを組み合わせて指定した時に行われる動作を次の表に示します。

表 3.3

最適化オプション		ループの最適化	広域的最適化
デフォルト	/Oa 無	実行される-安全でない (巻き上げ/掘り下げ以外)	実行される-安全でない (巻き上げ/掘り下げ以外)
	/Oa 有	実行される-安全 (巻き上げ/掘り下げ以外)	実行される-安全 (巻き上げ/掘り下げ以外)
/Od	/Oa 無	実行されない	実行されない
	/Oa 有	エラー	エラー
/Ol	/Oa 無	実行される-安全でない	実行されない
	/Oa 有	実行される-安全	実行されない
/Og	/Oa 無	実行されない	実行される-安全でない
	/Oa 有	実行されない	実行される-安全
/Om	/Oa 無	実行される-安全でない	実行される-安全でない
	/Oa 有	実行される-安全	実行される-安全
/Ot	/Oa 無	実行される-安全でない	実行される-安全でない
	/Oa 有	実行される-安全	実行される-安全

/Ol、/Og および/Oa の最適化オプションは組み合わせて指定できます。

最適化項目と最適化オプションの関係を以下にまとめます。

最適化項目 ＼ 最適化オプション		/Od	/Ol	/Og	/Om	/Ot	指定なし
局所最適化 定数伝搬 定数畳み込み 共通部分式削除 代数恒等式利用 代数的変換 コピー伝搬		●	●	●	●	●	●
覗き穴最適化 冗長命令削除 相対ジャンプ最適化 テールリカージョン		●	●	●	●	●	●
ループ最適化 ループ不変コード移動 ループ変動コード移動 誘導変数削除 ループの強さ軽減 ループ展開		-	●	-	●	●	●
広域最適化 定数伝搬 定数畳み込み 共通部分式削除		-	-	●	●	●	●
コード掘り下げ コード巻き上げ		-	-	●	●	●	-
実行速度最適化		-	-	-	-	●	-
その他最適化 冗長コード削除 冗長変数削除 代数変換 ジャンプ最適化		-	●	●	●	●	●
レジスタ退避・復帰共通化	/Orp *2	- *1	●	●	●	●	●
	指定なし	-	-	-	●	-	-
抑止機能 レジスタ退避・復帰共通化抑止	/Orpn *2	- *1	● *3	● *3	● *3	● *3	● *3
	指定なし	-	-	-	-	-	-
抑止機能 別名参照の最適化抑止	/Oa	- *1	●	●	●	●	●
	指定なし	● *4	-	-	-	-	-
抑止機能 レジスタ割り付け抑止	/Zs	● *4	●	●	●	●	●
	指定なし	● *4	-	-	-	-	-

*1：グレーで網掛けしている部分は組み合わせ不可（エラー）であることを示します。
*2：/Orp と/Orpn の同時指定は不可（エラー）。
*3：/Ol, /Og, /Ot, デフォルトの場合は、/Orpn は指定の有無に関わらず、レジスタ退避復帰共通化をしません。/Orpn が意味を持つ（共通化を抑止する）のは、/Om のときだけです。
*4：/Od の場合、/Oa、/Zs が指定されたものとみなして、別名参照の最適化抑止とレジスタ割り付け抑止を行います。

3.2.3.10 最適化オプションの組み合わせ

次の表は、最適化オプションの組み合わせをまとめたものです。

表 3.4

番号	組み合わせ	有効性	組み合わせが有効の場合に行われる最適化
1.	/Od /Og	無効	-
2.	/Od /Ol	無効	-
3.	/Od /Oa	無効	-
4.	/Od /Om	無効	-
5.	/Od /Ot	無効	-
6.	/Om /Ot	無効	-
7.	/Og /Ol	有効	ループの最適化 広域的最適化 その他の最適化
8.	/Og /Oa	有効	ループの最適化 別名チェック その他の最適化
9.	/Og /Om	有効	ループの最適化 広域的最適化 その他の最適化
10.	/Og /Ot	有効	ループの最適化 広域的最適化 実行速度の最適化 その他の最適化
11.	/Ol /Oa	有効	ループの最適化 別名チェック その他の最適化

番号	組み合わせ	有効性	組み合わせが有効の場合に実行される最適化
12.	/Ol /Om	有効	ループの最適化 広域的最適化 その他の最適化
13.	/Ol /Ot	有効	ループの最適化 広域的最適化 実行速度の最適化 その他の最適化
14.	/Oa /Om	有効	ループの最適化 広域的最適化 別名チェック その他の最適化
15.	/Oa /Ot	有効	ループの最適化 広域的最適化 実行速度の最適化 別名チェック その他の最適化
16.	/Og /Ol /Oa	有効	ループの最適化 広域的最適化 別名チェック その他の最適化
17.	/Og /Ol /Om	有効	ループの最適化 広域的最適化 その他の最適化
18.	/Ol /Oa /Om	有効	ループの最適化 広域的最適化 別名チェック その他の最適化

番号	組み合わせ	有効性	組み合わせが有効の場合に実行される最適化
19.	/Ol /Og /Oa /Om	有効	ループの最適化 広域的最適化 別名チェック その他の最適化
20.	/Og /Ol /Ot	有効	ループの最適化 広域的最適化 実行速度の最適化 その他の最適化
21.	/Ol /Oa /Ot	有効	ループの最適化 広域的最適化 実行速度の最適化 別名チェック その他の最適化
22.	/Ol /Og /Oa /Ot	有効	ループの最適化 広域的最適化 実行速度の最適化 別名チェック その他の最適化

3.2.4 出力ファイル

3.2.4.1 エラーリストオプション

構文：/LE

/LE オプションによって、CCU8 はエラーがあればエラーメッセージ付きでソースファイルのリストを生成します。ソースコードのすべてが行番号付きでリストされます。リストファイルの名前は、入力ファイルと同じ名前で拡張子“.LER”が付きます。

このオプションは、プリプロセッサオプション/LP や/PC と組み合わせて指定することはできません。指定するとフェイタルエラーが表示されます。

エラーメッセージやフェイタルエラーメッセージが生成されていなければ、各関数で使用されているスタックのサイズについての情報がリストファイルに出力されます。

例 3.17

```
C:¥> CCU8 /LE  /Tmu8 test.c <CR>
```

上のコマンドラインによって CCU8 は、リストファイル“test.ler”を生成します。出力したリストファイルは、行番号付きのソースが含まれ、エラーがあればエラーメッセージも含まれます。

3.2.4.2 コールツリーオプション

構文：/CT filename

このオプションによって、CCU8 は関数呼び出しのリストを生成します。コールツリーリストファイルには、左マージンで字下げして列記した関数名およびそれぞれの関数呼び出しが含まれます。

オプション/CT は filename を必ず指定しなければなりません。ファイル名を指定しなければ、CCU8 はエラーメッセージを表示します。CCU8 がデフォルトとする拡張子はありません。

このオプションは、プリプロセッサオプション/LP や/PC と組み合わせて指定することはできません。指定するとフェイタルエラーが表示されます。

例 3.18

```
C:¥> CCU8 /CT test.cal /Tmu8 test.c <CR>
```

上のコマンドラインの/CT オプションにより、CCU8 はコールツリーファイル“test.cal”を生成します。関数名およびそれぞれの関数呼び出しが字下げされて“test.cal”に出力されます。

例 3.19

```
C:¥> CCU8 /CT test.cal /Tmu8 t1.c t2.c <CR>
```

上のコマンドラインの/CT オプションにより、CCU8 はコールツリーファイル“test.cal”を生成します。ファイル t1.c および t2.c、両方のコールツリーリストが続けて“test.cal”に出力されます。しかし、関数情報を、ひとつのファイルから他のファイルに受け渡すことはできません。

例 3.20

```
C:¥> CCU8 /CT test.cal /LP test.c <CR>
```

上のコマンドラインでは、/CT オプションと/LP オプションは互いに排他的なので、CCU8 はフェイタルエラーを表示します。

3.2.4.3 アセンブリリストファイル

構文：/Fa[path]

/Fa オプションによって、CCU8 は指定された名前、パスまたはディレクトリでアセンブリリストファイルを生成します。パス付きまたはパス無しのファイル名を/Fa オプションで指定した場合、アセンブリリストはそのファイルに出力されます。指定したファイル名に拡張子がない場合には、出力ファイルに“.ASM”拡張子が付加されます。

ディレクトリのみが/Fa オプションで指定された場合、アセンブリリストはデフォルトのファイル名で指定のディレクトリに生成されます。

引数 path はオプションです。/Fa オプションで引数を指定しないと、アセンブリリストはデフォルトのファイル名でカレントディレクトリに生成されます。

/Fa オプションで指定したファイル名またはパスが正しくない時には、CCU8 はフェイタルエラーメッセージを表示します。

/Fa オプションでディレクトリを指定した場合、他の/Fa オプションの指定がされるまでのすべてのソースファイルにはその/Fa オプション指定がされているとみなされます。

/Fa オプションは 1 つのソースファイルに対してコマンドライン中に何回でも指定できます。ソースファイルの前に複数の/Fa オプションが指定された場合には、最後の/Fa オプションがそのソースファイルに対して指定されたものとみなされます。

/Fa オプションは、コマンドラインで複数のソースファイルの間に指定することもできます。

例 3.21

```
C:¥> CCU8 /Fa..¥ /Tmu8 test.c <CR>
```

上のコマンドラインでは、アセンブリリストファイル“test.asm”はカレントの作業ディレクトリの親ディレクトリに作成されます。

例 3.22

```
C:¥> CCU8 /Fad:¥asm¥ /Tmu8 test1.c test2.c<CR>
```

上のコマンドラインでは、アセンブリリスト出力ファイル“test1.asm”および“test2.asm”は、“d:¥asm”ディレクトリに作成されます。

例 3.23

```
C:¥> CCU8 /Faout1 /Faout2 /Tmu8 test.c <CR>
```

上のコマンドラインでは、アセンブリリストは“out2.asm”という名前で作成されます。

3.2.4.4 関数プロトタイプオプション

構文：/Zg

/Zg オプションによって、CCU8 は関数プロトタイプのリストを生成します。リストファイルの名前は、入力ファイルと同じ名前で“.PRO”の拡張子が付加されます。

このオプションは、プリプロセッサオプション/LP または/PC のいずれかと組み合わせて指定することはできません。指定するとフェイタルエラーが表示されます。

例 3.24

```
C:¥> CCU8 /Zg /Ot test.c <CR>
```

上のコマンドラインでは、CCU8 はリストファイル“test.pro”を生成します。出力されたリストファイルには、定義した関数の本体のプロトタイプが含まれています。

3.2.5 プリプロセッサオプション

3.2.5.1 /LP オプション

構文 : /LP

このオプションによって、CCU8 は各入力ファイルをプリプロセス(前処理)した結果のリストを出力します。このオプションを指定すると、CCU8 は、ソースファイルのテキストを操作するテキストプロセッサとして動作します。このオプションには次の機能があります。

1. マクロ展開
2. コメントの削除
3. ファイルの組み込み
4. 条件コンパイル
5. 行制御
6. エラー生成

これらはソースファイル内の前処理指令を処理することによって実行されます。

プリプロセスファイルの名前は、入力ファイルと同じ名前で拡張子".I"が付加されます。このオプションを指定すると、ソースファイルはコンパイルされません。

リストファイルオプション(/LE)、コールツリーオプション(/CT)および関数プロトタイプリスト(/Zg)は、/LP オプションと組み合わせて指定することはできません。この/LP オプションを指定すると、入力ファイルにはどのような拡張子でも付けることができます(拡張子を付けなくてもかまいません)。

例 3.25

```
C:¥> CCU8 /LP test.c <CR>
```

上の例の/LP は、コンパイラにプリプロセスファイル test.i を生成するよう指示します。コメントは削除されます。

例 3.26

```
C:¥> CCU8 /LP /LE test.c <CR>
```

上のコマンドラインでは、/LE と/LP オプションは互いに排他的なので、フェイタルエラーが生成されます。

3.2.5.2 /PC オプション

構文 : /PC

プリプロセッサは通常、処理中にソースファイル内のすべてのコメントを削除します。/PC オプションは、コンパイラにプリプロセッサ動作中でもコメントを残しておくよう指示します。CCU8 は、ソースファイル内に記述されているコメントを付けたままプリプロセスファイルを出力します。それ以外の機能はすべて/LP オプションと同様です。

プリプロセッサオプション/PC と/LP は互いに排他的です。2 つのオプションのいずれかだけをコマンドラインで指定できます。両方のオプションを同時に指定すると、CCU8 はフェイタルエラー“Duplicate preprocessor option”(プリプロセッサオプションの重複)を出力します。

プリプロセスファイルの名前は、入力ファイルと同じ名前で拡張子“.I”が付加されます。このオプションを指定すると、ソースファイルはコンパイルされません。

リストファイルオプション(/LE)、コールツリーオプション(/CT)および関数プロトタイプリスト(/Zg)は、/PC と組み合わせて指定することはできません。このオプションを指定すると、入力ファイルにはどのような拡張子でも付けることができます(拡張子を付けなくてもかまいません)。

例 3.27

```
C:¥> CCU8 /PC test10.inp <CR>
```

上の例の/PC は、コンパイラにプリプロセスファイル test10.i を作成するよう指示します。出力ファイル中にコメントは残っています。

例 3.28

```
C:¥> CCU8 /PC /LP key.c <CR>
```

オプション/LP と/PC は互いに排他的のため、上のコマンドラインではフェイタルエラーが表示されます。

3.2.5.3 /I オプション

構文 : /I <directory>

インクルードファイルを検索するディレクトリを/I オプションで指定できます。このオプションは、環境変数 INCLU8 の影響を一時的に無効にし、変更することができます。CCU8 は、まずこのオプションで指定されたディレクトリを検索してから、INCLU8 環境変数に指定されたディレクトリを検索します。

１つの/I オプションで指定できるのは１つのディレクトリ名だけです。複数のディレクトリ名を指定する場合は、/I オプションを繰り返して使用します。

例 3.29

```
C:¥> CCU8 /I ¥include /Tmu8 test.c <CR>
```

上のコマンドラインでは、CCU8 はインクルードファイルを環境変数 INCLU8 を使用して指定したディレクトリを検索する前にディレクトリ“¥include”から検索します。

例 3.30

```
C:¥> CCU8 /I include /I lib /LP test.c <CR>
```

上のコマンドラインでは、CCU8 はまずインクルードファイルをディレクトリ“include”で検索します。見つからなければ、ディレクトリ“lib”で検索します。それでも見つからない場合には、CCU8 は環境変数 INCLU8 で指定されたディレクトリを検索します。

3.2.5.4 /D オプション

構文：/D <identifier>[=[string]]

ここで、‘identifier’ はマクロで、‘string’ は置き換える文字列です。

引数の無いマクロを、/D オプションを使用してコマンドラインで定義できます。マクロ処理は、ソースファイルで指定されているときと同様に行われます。

/D オプションで指定された引数が識別子でない場合、CCU8 はフェイタルエラーメッセージを表示します。

例 3.31

```
C:¥> CCU8  /Tmu8 /DVALUE(a) test14.c <CR>
```

上のコマンドラインでは、‘VALUE(a)’ が識別子ではないので、CCU8 はフェイタルエラーメッセージを表示します。

/D とマクロの間に空白はあってもなくてもかまいません。/D で identifier だけを指定している時は、マクロの置き換え文字列は‘1’となります。

identifier と等号(=)との間に空白を入れることはできません。引数が等号(=)で終わっている場合は、マクロの置き換え文字列は空になります。

等号(=)と置き換え文字列との間に空白を入れることはできません。

/D オプションはコマンドラインの各ソースファイル名の前に指定できます。/D オプションで定義されたマクロは、コマンドラインで/D オプションの後で指定しているすべてのファイルを対象とみなします。

例 3.32

```
C:¥> CCU8  /Tmu8  /DVALUE1  test15.c  /DVALUE2=  test16.c <CR>
```

マクロ VALUE1 は 1 として定義され、‘test15.c’および‘test16.c’の両方を対象とします。 ただし、VALUE2 は置き換え文字列なしで定義されており、‘test16.c’のみを対象とします。

3.2.5.5 /U オプション

構文：/U <identifier>

前の方で定義したコマンドラインマクロを、コマンドラインで/U オプションを使用すれば未定義にできます。

identifier は、/U オプションと一緒に指定する必要があります。 identifier を指定しないと、CCU8 はフェイタルエラーメッセージを表示します。

/U と identifier の間に空白はあってもなくてもかまいません。

/U オプションはコマンドラインに何回でも指定できます。/U オプションは、最初のファイル名の前、または 2 つのソースファイル名の間に指定することができます。

/U オプションで指定した引数が識別子でない時には、CCU8 はフェイタルエラーメッセージを表示します。

指定した identifier を/D オプションで前の方で定義していない時には、CCU8 は/U オプションを無視します。

/U オプションは、事前定義マクロやソースファイルにユーザーが定義したマクロには影響しません。

例 3.33

```
C:¥> CCU8  /Tmu8  /DVALUE=1  test15.c test16.c /UVALUE  test17.c <CR>
```

マクロ‘VALUE’は 1 として定義され、‘test15.c’および‘test16.c’が処理対象とみされます。ただし、‘test16.c’の後では未定義にされているので、‘test17.c’は対象となりません。

3.2.6 スタック

3.2.6.1 スタックサイズオプション

構文 : /SS <constant>

/SS オプションは、プログラムのスタックサイズを設定します。CCU8 は、このオプションで指定したサイズを擬似命令 STACKSEG を使って出力します。このオプションによって、リンカ RLU8 がプログラムのスタック領域を確保できます。

このオプションを指定しないと、CCU8 はデフォルトのスタックサイズ 1024 バイトを指定します。

constant は 10 進定数でなければなりません。定数の有効範囲は 0 より大きく 65535 より小さい偶数値です。/SS と定数の間の空白はなくてもかまいません。

例 3.34

```
C:¥> CCU8 /SS 2048  /Tmu8 test.c <CR>
```

上のコマンドラインでは、CCU8 は STACKSEG 擬似命令でサイズ 2048 バイトのスタックサイズを設定します。

例 3.35

```
C:¥> CCU8 /SS0x0800   /Tmu8 test.c <CR>
```

/SS オプションに対するパラメータとして指定できるのは 10 進の定数だけなので、上のコマンドラインに対して CCU8 はフェイタルエラーを表示します。

3.2.6.2 スタックチェックオプション

構文 : /ST

/ST オプションを指定すると、CCU8 はアセンブリ出力にスタックプローブを追加します。

「スタックプローブ」とは、関数の入口で呼び出す短いルーチンで、関数が必要とするローカル変数を割り当てるのに十分な領域がプログラムのスタック内にあるかどうかを確かめます。スタックプローブルーチンは、スタック内に必要なサイズがないと判断した時には、C 関数 _stack_error へジャンプします。ユーザーは関数_stack_error を定義する必要があります。

このオプションを指定しないと、スタックプローブルーチンは呼び出されません。診断を行わないとスタックがオーバフローする可能性があります。

例 3.36

```
C:¥> CCU8 /ST /Tmu8 test.c <CR>
```

上のコマンドラインでは、“test.c”内の各関数のエントリコードでスタックプローブルーチンの呼び出しが生成されます。

3.2.7 デバッグオプション

3.2.7.1 /SD オプション

構文 : /SD

/SD オプションを指定すると、CCU8 は C ソースレベルデバッガに必要な情報を生成します。/SD オプションを指定せずにファイルをコンパイルすると、ソースレベルデバッガを使用したソースレベルのデバッグはできません。

デバッグ情報は文字列としてアセンブリ出力に埋め込まれます。

例 3.37

```
C:¥> CCU8 /SD /Tmu8 test.c <CR>
```

上のコマンドラインでは、デバッグ情報は文字列としてアセンブリ出力に埋め込まれます。

3.2.8 その他のオプション

3.2.8.1 /SL オプション

構文 : /SL <constant>

/SL オプションは識別子の最大長を設定します。 constant は、31 以上 254 以下の整数である必要があります。

このオプションを指定しないと、CCU8 は識別子の最大長を 31 とみなします。

例 3.38

```
C:¥> CCU8 /SL 40  /Tmu8 test.c <CR>
```

上の例では、CCU8 は識別子の最大長を 40 文字とします。“test.c”内で 40 文字を超えた識別子が見つかると、最初の 40 文字を識別子名とみなし、ワーニングメッセージを表示します。

例 3.39

```
C:¥> CCU8 /Tmu8 test.c <CR>
```

上の例では、CCU8 は 31 文字を識別子の最大長(デフォルトの最大識別子長)とします。

例 3.40

```
C:¥> CCU8 /SL 1023  /Tmu8 test.c <CR>
```

上のコマンドラインでは、 constant の値が 31 以上 254 以下の範囲を超えているのでフェイタルエラーが表示されます。

例 3.41

```
C:¥> CCU8 /SL /Tmu8 test.c <CR>
```

上のコマンドラインでは、 constant が/SL の後に指定されていないのでフェイタルエラーが表示されます。

3.2.8.2. /J オプション

構文 : /J

/J オプションは、CCU8 に **char** 型のデフォルトを **unsigned char** 型とするよう指示します。/J オプションがコマンドラインに指定されていると、CCU8 は **signed** 指定子のついていない **char** 型をすべて **unsigned char** 型として扱います。

例 3.42

```
char chr ;
```

デフォルトでは、CCU8 は変数 chr を **signed char** 型として扱います。/J オプションをコマンドラインに指定すると、CCU8 は変数 chr を **unsigned char** 型として扱います。

3.2.8.3 /PF オプション

構文 : /PF

プラグマの引数のデフォルト区切り文字は空白です。コマンドラインに/PF オプションを指定すれば、デフォルト(空白)を“,” (コンマ)に変更することができます。

/PF オプションを指定しない場合のプラグマの構文は次のとおりです。

```
#pragma pragma_keyword  [ argument1  argument2  ...]
```

/PF オプションをコマンドラインに指定した場合のプラグマの構文は次のとおりです。

```
#pragma pragma_keyword  [ argument1,  argument2,  ...]
```

例 3.43

```
#pragma interrupt  function_name, address
```

上のプラグラマの構文は、/PF をコマンドラインに指定すれば正しい構文です。指定していない場合には、CCU8 はワーニングメッセージを表示し、プラグマを無視します。

3.2.8.4 /SYS オプション

構文 : /SYS

/SYS オプションは、コンパイラにセグメントの命名規則を変更するよう指示します。このオプションはシステムファイルをコンパイルするときに使用されます。

例 3.44

```
C:¥> CCU8 /SYS /Tmu8 test.c <CR>
```

上の例では、CCU8 は“test.c”のコンパイルに別のセグメント命名規則を使用します。

3.2.8.5 /W オプション

構文 : /W <constant>

/W オプションは、指定したレベルのワーニングを表示するようコンパイラに指示します。constant は 0 以上 3 以下の 10 進数である必要があります。/W オプションは、どのコマンドラインオプションとでも同時に指定できます。

/W オプションに 0, 1, 2, 3 以外の引数を指定すると、フェイタルエラーメッセージが表示されます。

/W オプションに引数の指定がないと、フェイタルエラーメッセージが表示されます。

/W と引数の間に空白があってもなくてもかまいません。

/W オプションは、最初のソースファイル名の前、またはソースファイル名の間に指定できます。/W オプションが指定されていないと、デフォルトのレベルの 1 とみなされます。

例 3.45

```
C:¥> CCU8 /Tmu8 /W3  test.c <CR>
```

上の例では、CCU8 は“test.c”のコンパイル中に発生したレベル 3 までのワーニングを表示します。

例 3.46

```
C:¥> CCU8  /Tmu8  /W0  test20.c  /W3  test16.c <CR>
```

上の例では、CCU8 は“test20.c”のコンパイルのワーニングは表示せず、“test16.c”のコンパイル中に発生したレベル 3 までのワーニングを表示します。

3.2.8.6 /Wc オプション

構文：/Wc <warning number> [,warning number,…]

/Wc オプションは、ワーニングのレベルに関係なく、ワーニングをエラーとして出力するようにコンパイラに指示します。ワーニングは、コマンドラインにワーニング番号で指定します。/Wc オプションの後ろには、少なくとも 1 つワーニング番号を指定する必要があります。文字‘,’(コンマ)で区切って、複数のワーニング番号を指定できます。/Wc オプションは、どのコマンドラインオプションとでも一緒に指定できます。/Wc オプションを /Wa オプションと共に指定した場合、/Wc オプションは意味がなくなるため、無視されます。

ワーニング番号を指定しないと、フェイタルエラーメッセージが表示されます。

複数のワーニング番号の間に区切り文字‘,’がないと、フェイタルエラーメッセージが表示されます。

不正なワーニング番号を指定すると、フェイタルエラーメッセージが表示されます。

/Wc と引数の間には、空白はあってもなくてもかまいません。

/Wc オプションは、最初のソースファイル名の前か、ソースファイル名の間であれば、コマンドライン内の任意の位置に指定できます。

例 3.47

```
C:¥> CCU8 /Tmu8 /Wc W5017 test.c <CR>
```

上の例では、"test.c"のコンパイル中にワーニング W5017 が発生した場合、CCU8 はこのワーニングをエラーとして出力します。

例 3.48

```
C:¥> CCU8 /Tmu8 /W0 /Wc W5025, W5033,W6000 test16.c <CR>
```

上の例では、"test16.c"のコンパイル中にワーニング W5025、W5033、W6000 が発生した場合、CCU8 はこれらのワーニングをエラーとして出力します。"test16.c"のコンパイル中に他のワーニングが発生しても、/W0 が指定されているため、ワーニングは出力しません。

例 3.49

```
C:¥> CCU8 /Tmu8 /Wc W5025, W5033, W6000 /W2 test16.c <CR>
```

上の例では、"test16.c"のコンパイル中にワーニング W5025、W5033、W6000 が発生した場合、CCU8 はこれらのワーニングをエラーとして出力します。/W2 が指定されているため、"test16.c"のコンパイル中に発生したレベル 2 までの他のワーニングを出力します。

例 3.50

```
C:¥> CCU8 /Tmu8 /Wc W5025 /W2 test16.c /W1 /Wc W6000 test20.c<CR>
```

上の例では、CCU8 は"test16.c"のコンパイル中に、ワーニング W5025 が発生した場合はエラーとして出力し、レベル 2 までのワーニングが発生した場合はワーニングを出力します。また、"test20.c"のコンパイル中に、ワーニング W5025、W6000 が発生した場合はエラーとして出力し、レベル 1 までのワーニングが発生した場合はワーニングを出力します。

例 3.51

```
C:¥> CCU8 /Tmu8 /Wc W5025 /W2 test16.c /Wa /W1 /Wc W6000 test20.c<CR>
```

上の例では、CCU8 は"test16.c"のコンパイル中に、ワーニング W5025 が発生した場合はエラーとして出力し、レベル 2 までのワーニングが発生した場合はワーニングを出力します。また、"test20.c"のコンパイル中は、/Wa オプションと /W1 オプションが指定されているため、/Wc オプションを無視して、レベル 1 までのワーニングが発生した場合はエラーとして出力します。

注記:

ワーニングがエラーに変わりますが、その番号、メッセージ、および説明は、そのワーニング番号に対応するワーニングリストのものを参照します。

3.2.8.7 /Wa オプション

構文 : /Wa

/Wa オプションは、指定されたワーニングレベル(デフォルトはレベル 1)で発生したすべてのワーニングをエラーとして出力するようにコンパイラに指示します。このオプションは、/Wc オプションよりも優先されます。/Wa オプションは、どのコマンドラインオプションとでも一緒に指定できます。

/Wa オプションは、最初のソースファイル名の前か、ソースファイル名の間であれば、コマンドライン内の任意の位置に指定できます。

例 3.52

```
C:¥> CCU8 /Tmu8 /Wa /W3 /Wc W6000 test20.c<CR>
```

上の例では、 CCU8 は"test20.c"のコンパイル中に発生したレベル 3 までのすべてのワーニングをエラーとして出力します。

注記:

ワーニングがエラーに変わりますが、その番号、メッセージ、および説明は、そのワーニング番号に対応するワーニングリストのものを参照します。

3.2.8.8 /NOWIN オプション

構文 : /NOWIN

/NOWIN オプションは、ROM WINDOW 領域を使用しないようコンパイラに指示します。

例 3.53

```
C:¥> CCU8 /Tmu8 /NOWIN  /SD  /ML  test.c <CR>
```

上の例では、CCU8 は ROM WINDOW 領域を使用せずにファイル“test.c”をコンパイルします。

注記:

このオプションは、ROM WINDOW 領域を持たないマイクロコントローラ用のオプションです。現時点では ROM WINDOW 領域を持たないマイクロコントローラは存在しませんので、このオプションは指定しないでください。

3.2.8.9 /Ff オプション

構文 : /Ff

/Ff オプションは、高速エミュレーションライブラリを使用するようにコンパイラに指示します。このオプションは float タイプの実行時間を短縮するためにサポートされています。

例 3.54

```
C:¥> CCU8 /Tmu8 /Ff  /SD  /ML  test.c <CR>
```

上の例では、CCU8 は高速エミュレーションライブラリを使用してファイル“test.c”をコンパイルします。

3.2.8.10 /Za オプション

構文：/Za

/Za オプションは、ソースファイルのコンパイル中に ANSI C 標準に対する拡張をすべて適用しないようコンパイラに指示します。事前定義マクロ__STDC__については、CCU8 は 0 と展開します。

例 3.55

```
C:¥> CCU8 /Tmu8 /Za  /SD  /ML  test.c <CR>
```

上の例では、CCU8 は ANSI C 標準に従って必要なワーニングとエラーを表示します。

3.2.8.11 /KJ オプション

構文: /KJ

/KJ オプションは、文字列内で SHIFT-JIS 漢字(2 バイト)を使用できるようコンパイラに指示します。

例 3.56

```
C:¥> CCU8 /Tmu8 /KJ test.c <CR>
```

上の例では、ソースファイル“test.c”内の文字列で、SHIFT-JIS 漢字がサポートされます。

3.2.8.12 /Lv オプション

構文: /Lv

/Lv オプションは、ローカル変数に割り当てられているレジスタ/スタック情報を出力するようにコンパイラに指示します。この情報は、各関数の前に出力されます。このローカル変数の拡張情報を表示すると、アセンブリコードが分かりやすくなります。

例 3.57

```
C:¥>CCU8 /Tmu8 /Lv test.c <CR>
```

上記の例では、関数ごとに、実在する各ローカル変数に割り当てられているレジスタ/スタック情報が、出力ファイルに出力されます。実在するローカル変数とは、最適化実行後にも存在しているローカル変数のことです。

例 3.58

入力:

```
int g_i ;

void
fn ()
{
        int l_i1 ;
        register int l_i2 ;

        l_i1 = fn1 () ;
        l_i2 = fn1 () ;

        g_i = l_i1 + l_i2 ;
}
```

/Lv コマンドラインオプションを指定すると、コードが次のように生成されます。

出力:

```
_fn     :

;;***f*************************************
;;      register/stack information
;;*************************************
        _l_i1$0 set -2
;;      _l_i2$2 set er4
;;*************************************

        push    lr
        push    fp
        mov     fp,       sp
        add     sp,       #-02
        push    er4

;;      l_i1 = fn1 () ;
```

```
        bl      _fn1
        st      er0,        _l_i1$0[fp]

;;      l_i2 = fn1 () ;
        bl      _fn1
        mov     er4,        er0

;;      g_i = l_i1 + l_i2 ;
        l       er0,        _l_i1$0[fp]
        add     er0,        er4
        st      er0,        NEAR _g_i

;;}
        pop     er4
        mov     sp,         fp
        pop     fp
        pop     pc
```

3.2.8.13 /Zs オプション

構文: /Zs

/Zs オプションは、ローカル変数をレジスタに割り当てないようにコンパイラに指示します。代わりに、ローカル変数はスタックのメモリに割り当てられます。ただし、ローカル変数に register キーワードを指定した場合は /Zs コマンド行オプションよりも優先され、割り当て可能な空きレジスタがあればレジスタに割り当てられます。

例 3.59

```
C:¥>CCU8 /Tmu8 /Zs test.c <CR>
```

上記の例では、すべてのローカル変数が、スタックメモリに割り当てられます。ただし、register キーワードが指定されたローカル変数は、利用可能なレジスタがないときのみスタックメモリに割り当てられます。

例 3.60

入力:

```
int g_i ;

void
fn ()
{
        int l_i1 ;
        register int l_i2 ;

        l_i1 = fn1 () ;
        l_i2 = fn1 () ;

        g_i = l_i1 + l_i2 ;
}
```

/Zs コマンド行オプションを指定した場合、コードが次のように生成されます。

出力:

```
_fn     :


        push    lr
        push    fp
        mov     fp,       sp
        add     sp,       #-02
        push    er4

;;      l_i1 = fn1 () ;
        bl      _fn1
        st      er0,      -2[fp]

;;      l_i2 = fn1 () ;
        bl      _fn1
        mov     er4,      er0

;;      g_i = l_i1 + l_i2 ;
        l       er0,      -2[fp]
        add     er0,      er4
        st      er0,      NEAR _g_i

;;}
        pop     er4
        mov     sp,       fp
```

```
pop     fp
pop     pc
```

3.2.8.14 /Zp オプション

構文: /Zp

/Zp コマンドラインオプションは、構造体/共用体のメンバをメンバの型に基づいて配置するよう、コンパイラに指示します。配置は次のようになります。

メンバの型	型
Char、unsigned char	バイト境界
Int、unsigned int、short、unsigned short、long、unsigned long	ワード境界
Float、double	ワード境界
Struct	struct の全メンバが char 型である場合はバイト境界、それ以外の場合はワード境界
Union	char 型ではないメンバが含まれる場合はワード境界、それ以外の場合はバイト境界
Array	array 型が char の場合はバイト境界、それ以外の場合はワード境界
Pointer	ワード境界

Note：

構造体のすべてのメンバが char 型の場合は、構造体のサイズが奇数になることがあります。

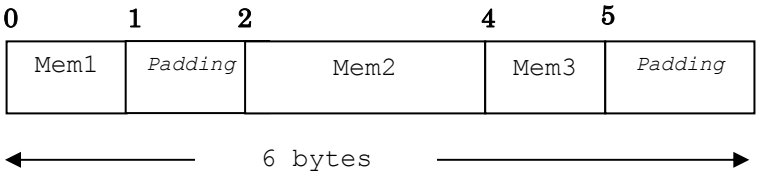
構造体のすべてのメンバが char 型ではない場合には、構造体のサイズは偶数になります。

/Zp オプションを指定した場合、構造体のコピーはバイト単位で行われることになるため、コードサイズは大きくなり、実行速度は遅くなります。

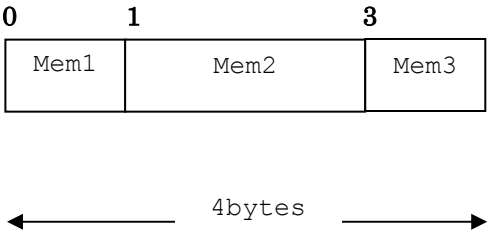
例 3.61

```
1.

struct ArrayMemberCharTag {

         unsigned char  mem1;
         unsigned char  mem2[2];
         unsigned char  mem3;

} ArrayMemberChar;
```

Zp なし



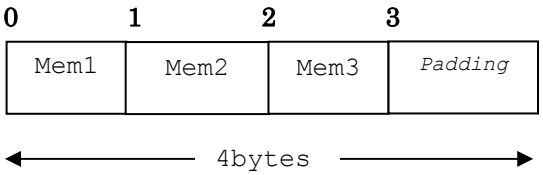
Zp あり



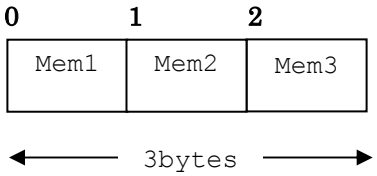
2.

```
struct MemberCharTag {

        unsigned char  mem1;
        unsigned char  mem2;
        unsigned char  mem3;

} MemberChar;
```

Zp なし



Zp あり

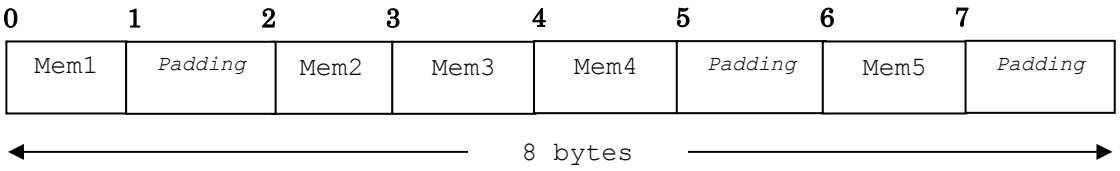


3.

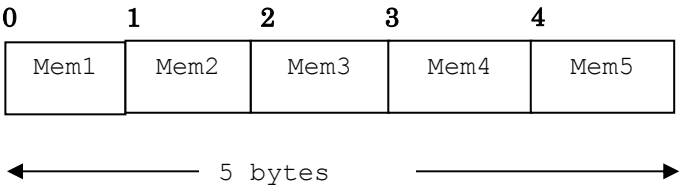
```
struct ArrayMemberCharTag {

        unsigned char  mem1;

        struct ArrayMemberCharTag2 {

                unsigned char  mem2;
                unsigned char  mem3;
                unsigned char  mem4;
```

```
            }ArrayMemberChar2;

            unsigned char  mem5;
} ArrayMemberChar;
```

Zp なし



Zp あり



3.2.8.15 /V オプション

構文: /V

/V オプションは、CCU8、CC1U8、および CC2U8 の実行可能プログラムのファイルバージョンを表示するようにコンパイラに指示します。このオプションは、コマンド行の 1 番目のオプションとして指定した場合だけ有効です。1 番目以外に指定した場合、このオプションは無視されます。

例 3.62

この例では、CCU8 のバージョン番号は 3.20 です。
CCU8.EXE ファイルのバージョンは 3.20.1 です。
CC1U8.EXE ファイルのバージョンは 3.20.1 です。

```
CC2U8.EXE ファイルのバージョンは 3.20.1 です。

C:¥>CCU8 /V  <CR>

    CCU8 C Compiler, Ver.3.20
    Copyright (C) 2008-2011 LAPIS Semiconductor Co., Ltd.

        CCU8.EXE    Ver.3.20.1
        CC1U8.EXE   Ver.3.20.1
        CC2U8.EXE   Ver.3.20.1
```

3.2.8.16 @オプション

構文: @ <filename>

@コマンドラインオプションは、指定された filename(レスポンスファイル)からコマンドラインオプションを受け取るようコンパイラに指示します。レスポンスファイルは、コマンドラインオプションが格納されているテキストファイルです。レスポンスファイルは、@マークを前に付けてコマンドラインの任意の位置に指定することができます。'@'と、ファイル名またはパスとの間には、空白は入れません。デフォルトの拡張子はありません。レスポンスファイルには、パスを含めることもできます。

CCU8 はレスポンスファイルの検索を、次のディレクトリ内で、次の順序で行います。

- カレントディレクトリ
- PATH 環境変数にリストされているディレクトリ
- CCU8 が置かれているディレクトリ

レスポンスファイルは、コマンドライン上で指定するとおりのコマンドラインオプションやファイル名を示す行からなります。これらの行には、コメントや MS-DOS の環境変数を含めることもできます。

たとえば、%env_var%は、指定された環境変数の内容に展開されます。環境変数 TMP が set TMP=C:¥tmp で設定されており、レスポンスファイルに /D TMP=%TMP%という行がある場合、CCU8 ではこの行が、/D TMP=C:¥TMP と展開されます。

各コマンドラインオプションを 1 行で指定し、行の最後にコメントを入れることができます。コメントはシャープ(#)、セミコロン(;)、または連続するスラッシュ 2 つで開始します。

これらの区切り文字と改行の間の文字は、空白 1 文字と同等に扱われます。

3.2.8.17 /nofar オプション

構文: /nofar

/nofar オプションは、物理セグメント#1 以上のデータメモリ空間を使用しないプログラムに対して使用可能です。/nofar オプションを指定すると、割り込み関数および SWI 関数の内部での DSR の保存/復帰コードの出力が抑制されます。

/nofar オプションを指定すると、FAR アクセスは禁止されます。__far 修飾子も禁止されます。

/nofar オプションと/far オプションを同時に指定することはできません。

/nofar オプション指定時の出力コードの例については、「5.1.2 割り込み関数内での DSR の使用の抑制」および「5.2.2 SWI 関数内での DSR の使用の抑制」を参照してください。

3.2.8.18 /Zc オプション

構文: /Zc

RLU8 Ver.1.50 から、参照されない関数・テーブルをリンクしない機能が追加されました。この機能への対応に伴い、CCU8 Ver.3.30 から、関数・テーブルが所属するセグメントをファイル単位ではなく、関数・テーブル単位で区切るよう、コンパイラに指示します。この機能によって、無駄な関数・テーブルのリンクがなくなり、コードサイズが小さくなるため、デフォルトの動作としています。

CCU8 Ver.3.21 以前のように、関数・テーブルが所属するセグメントを、関数・テーブル単位ではなく、ファイル単位で区切るよう、コンパイラに指示するには、/Zc オプションを指定します。

/Zc オプションを指定した場合と、指定しない場合の、コンパイラが出力するセグメント名の違いを以下に示します。

/Zc 指定の有無	関数の種類	デフォルトのセグメント名
/Zc 指定あり	通常の関数（スモールモデル）	$$NCOD*filename*
	通常の関数（ラージモデル）	$$FCOD*filename*
	割り込み関数	$$INTERRUPTCODE
	const 変数（near）	$$NTAB*filename*
	const 変数（far）	$$FTAB*filename*
/Zc 指定なし	通常の関数（スモールモデル）	$$*funcname*$*filename*
	通常の関数（ラージモデル）	$$*funcname*$*filename*
	割り込み関数	$$*funcname*$*filename*
	const 変数（near）	$$TAB*constname*$*filename*
	const 変数（far）	$$TAB*constname*$*filename*

funcname は関数名、*constname* は const 変数名、*filename* はファイル名を示します。

関数・割り込み関数・テーブル以外のセグメントに対しては、/Zc オプションの影響を受けません。

3.2.9 無効となるオプションの組み合わせ

コマンドラインオプションの無効な組み合わせは次のとおりです。

1. /LP および/PC
2. /LE およびプリプロセッサオプション(/LP もしくは /PC)
3. /CT およびプリプロセッサオプション(/LP もしくは /PC)

4. /Fa およびプリプロセッサオプション(/LP もしくは /PC)
5. /Zg およびプリプロセッサオプション(/LP もしくは /PC)
6. /Om および/Ot
7. /Od および他の最適化オプション(/Ol, /Og, /Oa, /Om および /Ot)
8. /nofer および/far

4. メモリモデル

ここでは、CCU8 がサポートするさまざまなメモリモデルとデータアクセス指定子、および追加のメモリモデル修飾子について説明します。

4.1 メモリモデル

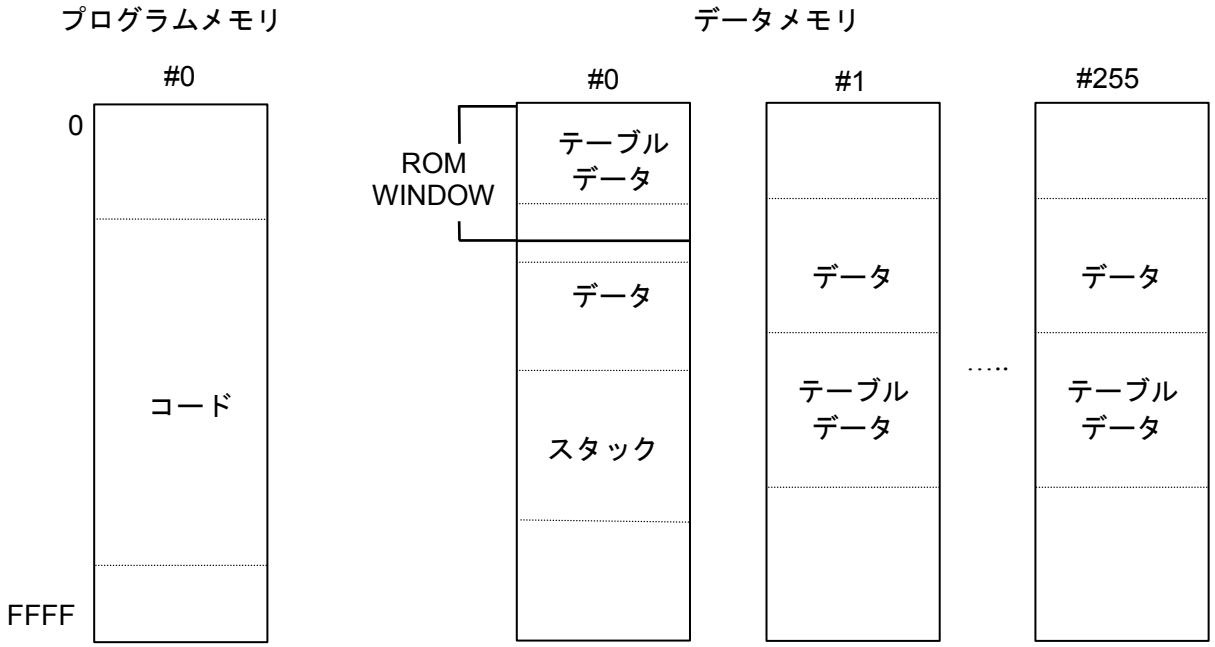
CCU8 は、次のメモリモデルオプションをサポートします。

1. Small メモリモデル
2. Large メモリモデル

メモリモデルに対応するコマンドラインオプションは次のとおりです。

1. /MS　Small メモリモデルに対応するオプション
2. /ML　Large メモリモデルに対応するオプション

4.1.1 Small メモリモデル

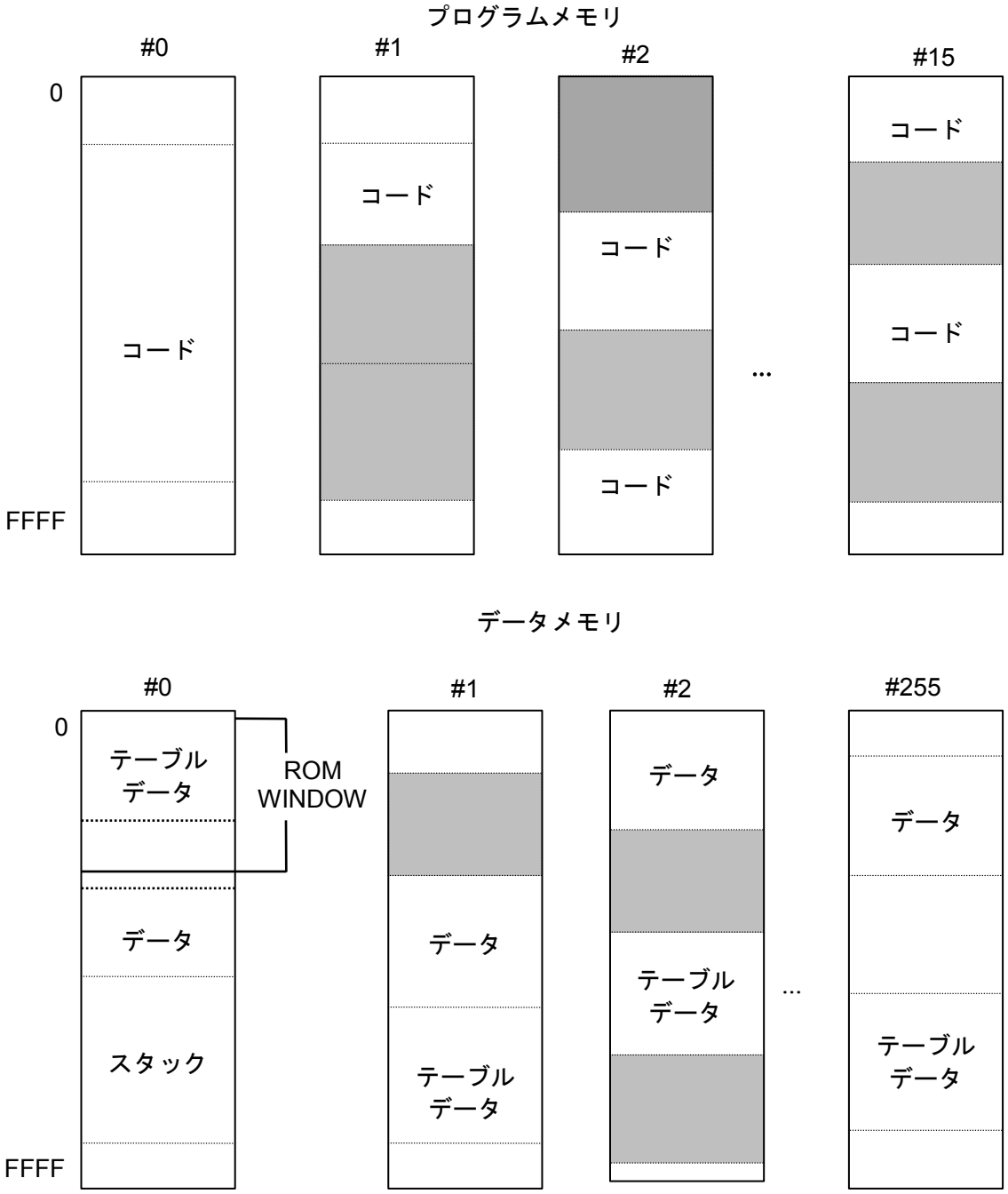


Small メモリモデルでのセグメントレジスタ使用方法

アクセス領域	セグメント レジスタ	変更	注記
コード	CSR	変更しない	
データ	DSR	far データの場合のみ変更	CCU8 は、DSR 設定のためのコードを付加する
テーブルデータ	DSR	far データの場合のみ変更	CCU8 は、DSR 設定のためのコードを付加する

Small メモリモデルは、ひとつのコードセグメントと複数のデータセグメントから構成されます。CCU8 は、コードをプログラムメモリの物理セグメント#0 に、データ(テーブルまたはその他)をデータメモリの物理セグメント#0 から#255 に割り当てます。ROM WINDOW 領域は、データメモリの物理セグメント#0 にのみ存在し、テーブルデータを格納するのにも使用されます。far データ(テーブルまたはその他)がない場合、物理セグメント#1 以上のデータメモリのアドレス空間は、空になります。

4.1.2 Large メモリモデル



Large メモリモデルでのセグメントレジスタ使用方法

アクセス領域	セグメントレジスタ	変更	注記
コード	CSR	変更する	ハードウェアは、PC/LR/ELR といっしょに CSR を自動的に保存、回復する
データ	DSR	far データの場合のみ変更	CCU8 は、DSR 設定のためのコードを付加する
テーブルデータ	DSR	far データの場合のみ変更	CCU8 は、DSR 設定のためのコードを付加する

Large メモリモデルは、複数のコードセグメントと複数のデータセグメントで構成されます。プログラムメモリの物理セグメント#0 から#15 にコードを、データメモリの物理セグメント#0 から#255 にデータ(テーブルまたはその他)を割り当てます。#1 以上の物理セグメントでは、プログラムメモリとデータメモリは、それぞれの物理セグメント(プログラムメモリまたはデータメモリの両方)で同じアドレス空間を共有します。影のついた部分のアドレス空間は、別の型(データかプログラム)のメモリに割り当てられているために利用できません。far データがない場合には、#1 以上の物理セグメントのデータメモリアドレス空間は空になります。

4.2 データアクセス

CCU8 は、アクセスするデータ(__near、__far)ごとに決まった固定サイズのポインタを使用します。

near データは、物理データセグメント#0 に置かれます。したがって near データのアクセスには、2 バイトポインタを使用します。この 2 バイトポインタを near ポインタと呼びます。

far データは、物理データセグメント#0 以上に置かれます。したがって far データのアクセスには、3 バイトポインタを使用します。この 3 バイトポインタを far ポインタと呼びます。

データアクセス指定子を指定せずに宣言したポインタの型は、/near または/far のコマンドラインオプション、あるいは NEAR または FAR プラグマによって決定されます。

/near コマンドラインオプションで特定のデータ項目を far データとして扱うには、__far 指定子を使用します。この指定子は、コマンドラインオプションによるデフォルトを無効にし、CCU8 は far データのアクセスのために DSR を設定するコードを挿入します。

逆に、/far コマンドラインオプションを無効にしてデータを near にする方法があります。__near 指定子が使用され、CCU8 は near データをアクセスするために DSR を設定するコードの挿入処理をスキップします。

__near 指定子および__far 指定子は関数に使用することはできません。

すべてのデータアクセス指定子は、互いに排他的です。

5. プラグマ

構文：

#pragma pragma_keyword arguments

#pragma 前処理指令は、アセンブリリストファイルにアーキテクチャ固有の命令を定義するよう CCU8 に指示します。コンパイラが認識できない命令をプラグマに定義すると、ワーニングメッセージが表示され、プラグマは無視されます。 pragma_keyword には、大文字/小文字の区別がありません。ここでは、CCU8 がサポートするプラグマについて説明します。

デフォルトでは、プラグマの引数を分離する区切り文字は空白です。デフォルトの区切り文字は、コマンドラインに/PF オプションを指定して“,” (コンマ)に変更できます。

5.1 INTERRUPT プラグマ

構文:

a. /PF オプションの指定がある場合:

```
#pragma INTERRUPT function_name, address, [category]
```

b. /PF オプションの指定が無い場合:

```
#pragma INTERRUPT function_name address [category]
```

interrupt プラグマは、C 言語で割り込み処理を行う関数を定義するのに使用します。このプラグマで指定した **function_name** の関数を、C ソースプログラム中に定義すると、その関数は割り込み処理ルーチンとして扱われます。このプラグマは、プラグマで指定する関数を定義する前に記述しなければなりません。関数定義より後にこのプラグマが記述されていると、ワーニングメッセージが表示され、プラグマは無視されます。interrupt プラグマで定義された関数は、そのモジュール内で static 宣言されなければなりません。static 宣言されなかった場合、コンパイラはワーニングを出力し、interrupt プラグマで定義された関数を static 関数として扱います。

このプラグマの **function_name** には、割り込み処理関数の名前を指定します。**function_name** の後に、割り込みベクタの **address** を続けて記述しなければなりません。値は、0x8 以上 0x7e 以下の偶数アドレスでなければなりません。**address** の後ろには、オプションで **category** が続きます。**category** の値は、1 か 2 でなければなりません。**category** の値が 1 の場合、多重割り込みの禁止を示します。この場合、この関数内に"**__EI**"組み込み関数が指定されているか、他の割り込み/SWI 関数への呼び出しがあると、コンパイラはエラーを表示します。**category** の値が 2 の場合、多重割り込みの許可を示します。category が指定されていない場合、**category** はデフォルトで 2 となり、多重割り込みの許可を意味します。

同じ割り込みベクタアドレスで別の関数名を使ってこのプラグマを複数回使用した場合、コンパイラはワーニングを表示して最初のプラグマを有効とします。ただし、同じ関数名を他の割り込みベクタアドレスに指定してもかまいません。

function_name は静的関数でなければなりません。静的関数でない場合、**CCU8** はワーニングメッセージを出力し、静的関数として処理します。

CCU8 は、割り込み処理関数で使用するレジスタをすべてこの関数の入口で待避し、関数の出口で対応するレジスタに復帰します。

CCU8 は次の場合にワーニングを表示します。

- 指定したシンボルが関数でない場合
- main 関数をこのプラグマで指定した場合
- このプラグマで指定した関数がコンパイルするファイル中に宣言されていない場合
- このプラグマで指定した関数に引数または戻り値がある場合
- このプラグマで指定した関数を **interrupt** プラグマ以外のプラグマがすでに指定している場合
- 関数定義の後でプラグマが指定している場合
- このプラグマで指定した関数を式に使用している場合
- 指定したアドレスが 0x8 以上 0x7e 以下の範囲外の場合
- 奇数アドレスを指定した場合
- category の値が 1 および 2 以外の場合
- ソースファイル中でその関数を呼び出している場合

次の指定は間違っている例です。

例 5.1

入力

```
int a ;
# pragma interrupt a 0x10
```

上の例では、‘a’は変数であり、関数ではありません。

例 5.2

入力

```
static void function ();
# pragma interrupt function 0x09
```

上の例では、奇数アドレスが指定されています。

例 5.3

入力

```
static int int10 (void) ;
# pragma interrupt int10 0x10
```

上の例では、int10 に戻り値があります。

5.1.1 レジスタの内容の保存

割り込みの処理後にプラグラムが正しく実行されるように、CCU8 は割り込み処理で使用される可能性のあるレジスタを保存します。

5.1.1.1 関数呼び出しのない、カテゴリ 1 の割り込み関数

例 5.4

入力

```
static void intr1_1a () ;
# pragma interrupt intr1_1a 0xA 1
int a, b, c ;
static void
intr1_1a ()
```

```
{

        a = b + c ;
}
```

上記の割り込み関数定義に対して生成されるコードは、次のとおりです。

出力

```
        rseg $$INTERRUPTCODE

_intr1_1a       :
        push    xr0

;;      a = b + c ;
        l       er0,    NEAR _b
        l       er2,    NEAR _c
        add     er0,    er2
        st      er0,    NEAR _a

;;}
        pop     xr0
        rti
```

5.1.1.2 関数呼び出しがある、カテゴリ 1 の割り込み関数

例 5.5

入力

```
static void intr () ;
# pragma interrupt intr 0xA 1
int a, b, c ;

static void
intr ()
{
        a = b + c ;
        fn1 () ;
}
```

出力

```
                rseg $$INTERRUPTCODE

_intr           :
                push    lr, ea
                push    xr0
                l       r0,         DSR
                push    r0

;;              a = b + c ;
                l       er0,        NEAR _b
                l       er2,        NEAR _c
                add     er0,        er2
                st      er0,        NEAR _a

;;              fn1 () ;
                bl      _fn1

;;}
                pop     r0
                st      r0,         DSR
                pop     xr0
                pop     ea, lr
                rti
```

5.1.1.3 関数呼び出しのない、カテゴリ 2 の割り込み関数

例 5.6

入力

```
static void intr1_2a () ;
/* # pragma interrupt intr1_2a 0xA
*/
# pragma interrupt intr1_2a 0xA 2
int a, b, c ;

static void
intr1_2a ()
{
        a = b + c ;
        __EI () ;
```

```
}
```

上記の関数定義に対して生成されるコードは、次のとおりです。

出力

```
_intr1_2a       :
        push    elr, epsw
        push    xr0

;;      a = b + c ;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;      __EI () ;
        ei

;;}
        pop     xr0
        pop     psw,pc
```

5.1.1.4 関数呼び出しのある、カテゴリ 2 の割り込み関数

例 5.7

入力

```
static void intr1_2b () ;
# pragma interrupt intr1_2b 0xA
/* or
# pragma interrupt intr1_2b 0xA 2
*/
int a, b, c ;

static void
intr1_2b ()
{
        a = b + c ;
        __EI () ;
```

```
        fn1 () ;
}
```

上記の割り込み関数定義に対して生成されるコードは、次のとおりです。

出力

```
_intr1_2b       :

        push    elr, epsw, lr, ea
        push    xr0
        l       r0,       DSR
        push    r0

;;      a = b + c ;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;      __EI () ;
        ei

;;      fn1 () ;
        bl      _fn1

;;}
        pop     r0
        st      r0,       DSR
        pop     xr0
        pop     ea, lr, psw, pc
```

5.1.2 割り込み関数内での DSR の使用の抑制

コマンドラインオプションの/nofar を指定すると、割り込み関数内での DSR の保存コードと戻りコードの出力が抑制されます。/nofar オプションは、物理セグメント#1 以上のデータメモリ空間を使用しない場合のみ指定可能です。

5.1.2.1 NOFAR を指定した関数呼び出しのある、カテゴリ 1 の割り込み関数

例 5.8

入力

```
static void InterruptFunction () ;
# pragma interrupt InterruptFunction 0xA 1
int a, b, c ;

static void
InterruptFunction ()
{
        a = b + c ;
        fn1 () ;
}
```

/nofar オプションを指定して上記の例をコンパイルすると、次のようなコードが生成されます。

出力

```
        rseg $$INTERRUPTCODE

_ InterruptFunction :
        push    lr, ea
        push    xr0

;;      a = b + c ;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;      fn1 () ;
        bl      _fn1

;;}
        pop     xr0
        pop     ea, lr
        rti
```

5.1.2.2 NOFAR を指定した関数呼び出しのある、カテゴリ 2 の割り込み関数

例 5.9

入力

```
static void InterruptFunction_2 () ;
# pragma interrupt InterruptFunction_2 0xA
/* or
# pragma interrupt InterruptFunction_2 0xA 2
*/
int a, b, c ;

static void
InterruptFunction_2 ()
{
        a = b + c ;
        __EI () ;

        fn1 () ;
}
```

コマンド行で/nofar オプションを指定すると、上記の割り込み関数定義に対して次のようなコードが生成されます。

出力

```
_ InterruptFunction_2     :

        push    elr, epsw, lr, ea
        push    xr0

;;      a = b + c ;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;      __EI () ;
        ei

;;      fn1 () ;
```

```
                bl      _fn1

        ;;}
                pop     xr0
                pop     ea, lr, psw, pc
```

5.2 SWI プラグマ

構文:

a. /PF オプションが指定されている場合:

```
#pragma SWI function_name, address, [category]
```

b. /PF オプションが指定されていない場合:

```
#pragma SWI function_name address [category]
```

swi プラグマは、C 言語で記述したソフトウェア割り込み処理関数を指定します。このプラグマに指定した **function_name** の関数を C 言語のソースプログラム内で定義すると、その関数はソフトウェア割り込み処理ルーチンとして扱われます。このプラグマは、指定した関数の定義より前に記述する必要があります。このプラグマを関数の定義の後に記述すると、ワーニングメッセージが表示され、無視されます。extern 関数をこのプラグマで指定してもかまいません。

このプラグマの **function_name** では、ソフトウェア割り込み処理関数の名前を指定します。**function_name** に続けて、割り込みベクタ **address** を記述しなければなりません。この値は、0x80 以上 0xfe 以下の範囲の偶数アドレスでなければなりません。**address** の後には、オプションで **category** が続きます。**category** の値は、1 か 2 でなければなりません。**category** の値が 1 の場合、多重割り込みの禁止を示します。この場合、この関数内に"**__EI**"組み込み関数が記述されているか、他の SWI 関数への呼び出しがあると、コンパイラはエラーを出力します。**category** の値が 2 の場合、多重割り込みの許可を示します。category が指定されなかった場合、**category** はデフォルトで 2 となり、多重割り込みの許可を意味します。

このプラグマは、同じ割り込みベクタアドレスに対して異なる関数名をこのプラグマで指定すると、コンパイラはワーニングを表示し、最初のプラグマを有効とします。ただし、異なる割り込みベクタアドレスで同じ関数名は使用できます。

CCU8 は、ソフトウェア割り込み処理関数で使用されているレジスタ R4～R15 をこの関数への入口でプッシュし、出口では対応するレジスタをポップします。ソフトウェア割り込み処理関数が戻り値と引数を持たない場合は R0～R3 も同様に保存されます。

次の場合に CCU8 はワーニングを表示します。

- 指定したシンボルが関数ではない場合
- main 関数をこのプラグマで指定した場合
- このプラグマで指定した関数がコンパイルするファイルに宣言されていない場合
- このプラグマで指定した関数がすでに **swi** 以外のプラグマに指定されている場合

- 関数定義の後にプラグマが指定されている場合
- 指定したアドレスが 0x80 以上 0xfe 以下の範囲内にない場合
- 奇数アドレスを指定した場合
- category の値が 1 および 2 以外の場合
- SWI プラグマで指定する前に関数を参照した場合
- SWI プラグマで指定されている関数が、関数へのポインタに割り当てられている場合

CCU8 は、次の場合にはエラーを表示します。

- '__EI'組み込み関数が、カテゴリ 1 の SWI 関数内で「呼び出された」場合
- カテゴリが 1 で、別の SWI 関数に対する呼び出しが行われた場合

以下に示すのは、エラーの場合の例です。

例 5.10

入力

```
int x;
# pragma SWI x 0x80
```

上記の例では、変数'x'は関数ではありません。

例 5.11

入力

```
static char function (int) ;
# pragma SWI function 0x09
```

上記の例では、奇数アドレスが指定されています。

例 5.12

入力

```
int a;
    void sfn1(void);
    void (*fn)() = sfn1;
    # pragma SWI sfn1 0x84 2       /* Warning */
    void sfn1()
```

```
{
    a += 10;
}
void func1()
{
    (*fn)();
}
```

上記の例では、関数 sfn1 はすでに参照されています。

5.2.1 レジスタの内容の保存

ソフトウェア割り込みの処理後にプログラムが正しく実行されるように、CCU8 は割り込み処理で使用される可能性のあるレジスタを保存します。

5.2.1.1 関数呼び出しのない、カテゴリ 1 のソフトウェア割り込み関数

例 5.13

入力

```
void function () ;
# pragma SWI function 0x80 1
int a, b, c ;
void
function()
{
    a = b + c ;
}
```

上記のソフトウェア割り込み関数定義に対して生成されるコードは、次のとおりです。

出力

```
            rseg $$INTERRUPTCODE

_function       :

;;{
            push    xr0

;;    a = b + c ;
```

```
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;}
        pop     xr0
        rti
```

5.2.1.2 関数呼び出しのある、カテゴリ 1 のソフトウェア割り込み関数

例 5.14

入力

```
void function () ;
# pragma SWI function 0x80 1
int a, b, c ;
void
function()
{
    a = b + c ;
    function2();
}
```

上記のソフトウェア割り込み関数定義に対して生成されるコードは、次のとおりです。

出力

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    lr, ea
        push    xr0
        push    r4
        l       r4, DSR
        push    r4

;;    a = b + c ;
```

```
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;    function2();
        bl      _function2

;;}
        pop     r4
        st      r4, DSR
        pop     r4
        pop     xr0
        pop     ea, lr
        rti
```

5.2.1.3 関数呼び出しのない、カテゴリ 2 のソフトウェア割り込み関数

例 5.15

入力

```
void function () ;

# pragma SWI function 0x80 2
int a, b, c ;
void
function()
{
    a = b + c ;

    __EI () ;
}
```

上記のソフトウェア割り込み関数定義に対して生成されるコードは、次のとおりです。

出力

```
        rseg $$INTERRUPTCODE

_function       :
```

```
;;{
         push    elr, epsw
         push    xr0

;;    a = b + c ;
         l       er0,      NEAR _b
         l       er2,      NEAR _c
         add     er0,      er2
         st      er0,      NEAR _a

;;    __EI () ;
         ei

;;}
         pop     xr0
         pop     psw, pc
```

5.2.1.4 関数呼び出しのある、カテゴリ 2 のソフトウェア割り込み関数

例 5.16

入力

```
void function () ;
void function2 () ;

# pragma SWI function 0x80 2
int a, b, c ;
void
function()
{
    a = b + c ;

    __EI () ;

    function2 () ;
}
```

上記のソフトウェア割り込み関数定義に対して生成されるコードは、次のとおりです。

出力

```
        rseg $$INTERRUPTCODE

_function       :

;;{
        push    elr, epsw, lr, ea
        push    xr0
        push    r4
        l       r4, DSR
        push    r4

;;    a = b + c ;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;    __EI () ;
        ei

;;    function2 () ;
        bl      _function2

;;}
        pop     r4
        st      r4, DSR
        pop     r4
        pop     xr0
        pop     ea, lr, psw, pc
```

5.2.2 SWI 関数内での DSR の使用の抑制

コマンドラインオプションの/nofar を指定すると、SWI 関数内での DSR の保存コードと戻りコードの出力が抑制されます。/nofar オプションは、物理セグメント#1 以上のデータメモリ空間を使用しない場合のみ指定可能です。

5.2.2.1 関数呼び出しがあり NOFAR オプションが指定されている、カテゴリ 1 のソフトウェア割り込み関数

例 5.17

入力

```
void function () ;
# pragma SWI function 0x80 1
int a, b, c ;
void
function()
{
    a = b + c ;
    function2();
}
```

次に示すのは、コマンド行で/nofar オプションを指定した場合に、上記のソフトウェア割り込み関数定義に対して生成されるコードです。

出力

```
            rseg $$INTERRUPTCODE

_function       :

;;{
            push    lr, ea
            push    xr0

;;    a = b + c ;
            l       er0,    NEAR _b
            l       er2,    NEAR _c
            add     er0,    er2
```

```
        st      er0,      NEAR _a

;;    function2();
        bl      _function2

;;}
        pop     xr0
        pop     ea, lr
        rti
```

5.2.2.2 関数呼び出しがあり NOFAR オプションが指定されている、カテゴリ 2 のソフトウェア割り込み関数

例 5.18

入力

```
void function () ;
void function2 () ;

# pragma SWI function 0x80 2
int a, b, c ;
void
function()
{
    a = b + c ;

    __EI () ;

    function2 () ;
}
```

次に示すのは、コマンド行で/nofar オプションを指定した場合に、上記のソフトウェア割り込み関数定義に対して生成されるコードです。

出力

```
        rseg $$INTERRUPTCODE

_function       :

;;{
```

```
            push    elr, epsw, lr, ea
            push    xr0

;;      a = b + c ;
            l       er0,      NEAR _b
            l       er2,      NEAR _c
            add     er0,      er2
            st      er0,      NEAR _a

;;      __EI () ;
            ei

;;      function2 () ;
            bl      _function2

;;}
            pop     xr0
            pop     ea, lr, psw, pc
```

5.3 INLINE プラグマ

構文:

a. /PF オプションが指定されている場合:

```
#pragma INLINE function_name [, function_name ...]
```

b. /PF オプションが指定されていない場合:

```
#pragma INLINE function_name [ function_name ...]
```

inline プラグマは、関数呼び出しではなくインライン展開することを指示します。

このプラグマは、関数の定義より前に記述しなければなりません。関数定義の後にこのプラグマを記述すると、CCU8 はワーニングメッセージを表示します。

複数個の関数名をこのプラグマで指定できます。プラグマに関数ではないものを指定すると、CCU8 はワーニングメッセージを表示します。このプラグマで指定した関数は、**static** 関数として扱われます。したがって、inline プラグマで指定した関数は、同じファイル内に定義する必要があります。

次の場合にはこのプラグマで指定した関数をインライン展開しません。

- 関数が可変個の引数を持っている場合
- inline プラグマの指定より前に関数が定義されている場合
- 関数内に switch 文（switch-jump テーブルを生成する（※1））がある場合

  ※1：switch-jump テーブルは、以下の条件をすべて満たす場合に生成されます。

  1. 制御式の型が(signed /unsigned)char/ short/int のいずれかである。
  2. case が 6 個以上ある。
  3. （（case の最大値 – case の最小値）/ （case の数））が 4 より小さい。

- 関数内に asm ブロックがある場合
- 関数が大きすぎる場合（ノード数が 99 個を超える場合（※2）、または関数内での関数呼び出しが 3 個を超える場合）

  ※2：C の記述では、1 つの演算を行う式を 1 行とした場合、30 行程度となります。

- 関数呼び出しの前に関数を定義していない場合
- インライン展開のレベルがインラインの深さ制限を超える場合
- INLINERECURSIONON プラグマが指定されていないにも関わらず、インライン関数を再帰的に呼び出している場合
- 対象の関数を、関数ポインタにより間接的に呼び出している場合
- 呼び出し側関数に、アクティブな最適化オプションとして Od が指定されている場合

すべてのインライン関数呼び出しを展開したら、その関数本体のコードを生成することはしません。インライン関数呼び出しが展開されないときには、CCU8 はワーニングメッセージを表示します。

ただし、次の場合はこのプラグマで指定した関数をインライン展開せず、ワーニングメッセージを表示しません。

- 呼び出し側関数に、アクティブな最適化オプションとして Od が指定されている場合。このオプションは、コマンド行または optimization プラグマで設定できます。
- inline プラグマで指定する前に関数が呼び出された場合
- 対象の関数を、関数ポインタにより間接的に呼び出している場合

次の場合に CCU8 はワーニングを表示します。

- 関数でないものを指定している場合
- main 関数をプラグマで指定している場合
- このプラグマで指定した関数を、すでに inline 以外のプラグマで指定している場合
- inline 関数の呼び出しがインライン展開されない場合
- 関数定義より後にプラグマで指定している場合

例 5.19

入力

```
int var ;
# pragma inline fn

int fn (int arg)
{
         return (arg*arg) ;
}

void fn1()
{
         var = fn (var) ;
}
```

出力

```
         type (mu8)
         model small, near
         $$NCOD5_5 segment code 2h #0h
         CFILE 0000H 00000011H "5_5.c"

         rseg $$NCOD5_5

_fn1     :
;;{
         push    lr

;;       var = fn (var) ;
         l       er0,      NEAR _var
         mov     er2,      er0
         bl      __imulu8sw
```

```
        st      er0,      NEAR _var

;;}
        pop     pc

        public _fn1
        _var comm data 02h #00h
        extrn code near : _main
        extrn code : __imulu8sw

        end
```

例 5.20

入力

```
# pragma inline fn
void fn ()
{
        fn () ;
}
fn1 ()
{
        fn () ;
}
```

“fn”が再帰的に呼び出されているので inline 関数“fn”は展開されません。

5.4 ABSOLUTE プラグマ

構文:

a. /PF オプションが指定されている場合:

```
#pragma ABSOLUTE name, [segment:]offset
```

b. /PF オプションが指定されていない場合:

```
#pragma ABSOLUTE name [segment:]offset
```

absolute プラグマは、グローバル変数または静的ローカル変数にアブソリュートアドレスを割り当てます。

物理セグメントアドレスが指定されていない場合は、0 とみなされます。near 変数に対して 0 以外の物理セグメントアドレスが指定されていると、CCU8 はワーニングメッセージを表示します。

物理セグメントアドレスには 0 から 0xff の間の値を指定することができ、オフセットには 0 から 0xffff の値を指定することができます。

absolute プラグマは、変数の宣言の前でも後でも指定できます。初期化済み変数をこのプラグマで使用することはできません。この場合は、変数の初期化より前にこのプラグマを記述しなければなりません。このプラグマを同じ変数に対して何回も使用すると、CCU8 はワーニングを表示し、最後のプラグマで指定したアドレスを割り当てます。extrn 変数をこのプラグマに指定すると、このプラグマは無視されます。

アブソリュートアドレスの有効範囲は次のとおりです。

- セグメントアドレス　　0x0 (near 変数用)
0x0 から 0xff (far 変数用)
- オフセットアドレス　　0x0 から 0xffff

次の場合に CCU8 はワーニングを表示します。

- このプラグマで指定したものがグローバル変数または静的ローカル変数でない場合
- 変数がすでに他のプラグマで指定されている場合
- このプラグマで指定した変数が同じファイル内で宣言されていない場合
- absolute プラグマで指定する変数に対して奇数アドレスを指定した場合
- 指定したアドレスがアブソリュートアドレスの範囲外の場合
- 変数の初期化の後にプラグマを指定した場合
- セグメント#1h 以上で指定された変数が __near 指定子で修飾されている場合

例 5.21

入力

```
int acc ;
# pragma absolute acc 0x40
```

出力

```
        public _acc

        dseg #00h at 040h
_acc :
```

```
        ds      02h
        extrn code near : _main
```

例 5.22

入力

```
long __far la ;
# pragma absolute la 0x2:0x1000
```

出力

```
        public _la

        dseg #02h at 01000h
_la :
        ds      04h
        extrn code near : _main
```

次の例は、誤った使用例です。

例 5.23

入力

```
# pragma absolute abs_data_var 100
void fn (void)
{
         int    abs_data_var    ;
}
```

上の例では、プラグマでローカル変数“abs_data_var”を指定しています。

5.5 ROMWIN プラグマ

構文:

a. /PF オプションが指定されている場合:

```
#pragma ROMWIN start_address, end_address
```

b. /PF オプションが指定されていない場合:

```
#pragma ROMWIN start_address end_address
```

romwin プラグマは、コンパイラに ROM WINDOW の境界について指示します。start_addr には 0x0 を、end_addr には次のいずれかを指定します。

ROM WINDOW 境界の有効な範囲は次のとおりです。

- start_address は、ゼロ以上でなければなりません。
- end_address は、start_address より大きくなければなりません。
- end_address に対して許される最大値は 0xffff です。

例 5.24

入力

```
#pragma ROMWIN 0 0x7fff
```

この例では、ROM WINDOW 境界は 0～0x7fff に設定されます。

出力

```
romwindow        0h, 07fffh
```

例 5.25

入力

```
#pragma ROMWIN 0x1000 0x9fff
```

この例では、ROM WINDOW 境界は 0x1000～0x9fff に設定されます。

出力

```
romwindow        1000h,    09fffh
```

以下に示すようなエラーの場合に対し、CCU8 はワーニングを表示します。

- end_address が start_address より小さい場合

  例 5.26

  ```
  #pragma ROMWIN 0x3000 0x2fff
  ```

- end_address の最大アドレスが 0xffff より大きい場合

  例 5.27

  ```
  #pragma ROMWIN 0 0x1ffff
  ```

5.6 NOROMWIN プラグマ

構文:

```
#pragma NOROMWIN
```

noromwin プラグマは、ROM WINDOW 領域を使用しないように指示します。**romwin** プラグマおよび **noromwin** プラグマは互いに排他的です。両方のプラグマを定義すると、最初に定義したプラグマが有効になります。CCU8 はワーニングを表示し、後のプラグマを無視します。

5.7 NVDATA プラグマ

構文:

a. /PF オプションが指定されている場合:

```
#pragma NVDATA variable [, variable ..]
```

b. /PF オプションが指定されていない場合:

```
#pragma NVDATA variable [ variable ..]
```

nvdata プラグマで変数を羅列して記述して、1 つ以上のグローバル変数もしくは **static** ローカル変数を NVRAM 領域に割り当てるよう、コンパイラに指示します。

ローカル変数はスタックに割り当てられるので、NVRAM 領域に割り当てることはできません。CCU8 は、このプラグマに指定されている変数の**'const'**修飾子を無視し、ワーニングメッセージを表示した後、この変数の型を NVDATA とします。変数にデータアクセス指定子(__near または__far)がない場合には、CCU8 はコマンドラインオプション/near または/far で指定したデフォルトを使用します。

次のような場合に CCU8 はワーニングを表示します。

- 指定したものがグローバル変数または **static** ローカル変数でない場合
- プラグマで指定した変数がコンパイルするファイルに宣言されていない場合
- **const** で修飾された変数が指定された場合
- 変数がすでに他のプラグマで指定されている場合
- このプラグマより前に変数が初期化されている場合

例 5.28

入力

```
int nvdata_var ;
# pragma nvdata nvdata_var
```

出力

```
          $$NNVDATA5_14 segment nvdata 2h #0h
CFILE 0000H 00000003H "5_14.c"
        public _nvdata_var
        extrn code near : _main

        rseg $$NNVDATA5_14
_nvdata_var :
        dw       00h
```

5.8 チェックスタックプラグマ

構文：

```
#pragma CHECKSTACK_ON

#pragma CHECKSTACK_OFF
```

checkstack_on プラグマは、このプラグマ以降に定義されている関数の入口にスタックプローブルーチンの呼び出しを追加するよう、コンパイラに指示します。

checkstack_off プラグマは、このプラグマ以降に定義されている関数の入口にスタックプローブルーチンの呼び出しを追加しないよう、コンパイラに指示します。

これら 2 つのプラグマは、コマンドラインの/ST オプションとは関係なく処理されます。

例 5.29

入力

```
# pragma CHECKSTACK_ON
void fn (void)
{
          fn1 (0) ;
          fn2 (1) ;
}
```

CCU8 は、関数“fn”に対して次のコードを生成します。

出力

```
_fn       :

          push    lr
          push    er0
          mov     r0,#02h
          mov     r1,#00h
          bl      __chstu8sw
          pop     er0

;;        fn1 (0) ;
          mov     er0,      #0
          bl      _fn1

;;        fn2 (1) ;
          mov     er0,      #1
          bl      _fn2

;;}
          pop     pc

          extrn code near : _fn2
          extrn code near : _fn1
          public _fn
          extrn code near : _main
          extrn code : __chstu8sw

          end
```

5.9 最適化制御プラグマ

5.9.1 OPTIMIZATION プラグマ

構文：

```
#pragma OPTIMIZATION [ARGUMENTS1 [ARGUMENT2 [ARGUMENT3 …. ]]]

#pragma OPTIMIZATION
```

注意：PF オプションを指定するときには、コンマ‘,’を使用して引数を区切ります。

optimization プラグマは、個別の関数に対して最適化オプションを設定します。optimization プラグマは最適化オプションよりも優先します。

これらのオプションにより、関数に対して実行される最適化のタイプが決まります。

引数として以下のオプションを指定できます。

1. Od ：最適化を無効にします。
2. Ol ：ループを最適化します。
3. Og ：グローバル最適化を実行します。
4. Om ：可能な限り最大限の最適化を行います（このオプションを指定した場合は、Orp も指定されたものとみなされます）。
5. Ot ：速度の最適化を行います。
6. default：コンパイラのデフォルトの最適化を行います。
7. Oa ：別名チェックを実行します。
8. Orp ：関数のレジスタ退避・復帰を共通化する。
9. Orpn ：関数のレジスタ退避・復帰を共通化しない。

注意： オプションはすべて大文字/小文字が区別されます。

プラグマは、次の optimization プラグマが検出されるまで、またはそれ以上 optimization プラグマが指定されていない場合には、ファイルの最後までが有効となります。

プラグマで引数として Orp のみ、または Orpn のみが指定されている場合は、コマンド行で指定されている最適化オプションが、以降の関数に対する有効な最適化オプションとして使用されます。ただし、コマンド行で/Od が指定されている場合で、かつプラグマの引数が Orp のみの場合は、CCU8 はワーニングを表示し、プラグマを無視します。

プラグマで引数が指定されていない場合は、コマンド行で指定されている最適化オプションが、以降の関数に対するアクティブな最適化オプションとして使用されます。

以下の場合にはプラグマは無視されます。

- プラグマが関数定義の内部に入力された場合。
- 指定されている引数が、上記の表で示されている有効な引数ではない場合。
- 複数の引数が指定されていて、以下の制限に引数が違反している場合。
  - Od を他の引数と一緒に指定することはできません。
  - ‘default’を Orp、Orpn 以外の他の引数と一緒に指定することはできません。
  - Om と Ot を一緒に指定することはできません。
  - Orp と Orpn を一緒に指定することはできません。

以下に、optimization プラグマの引数（オプション）と最適化項目の関係を示します。

最適化項目＼引数（オプション）		Od	Ol	Og	Om	Ot	default
局所最適化 定数伝搬 定数畳み込み 共通部分式削除 代数恒等式利用 代数的変換 コピー伝搬		●	●	●	●	●	●
覗き穴最適化 冗長命令削除 相対ジャンプ最適化 テールリカージョン		●	●	●	●	●	●
ループ最適化 ループ不変コード移動 ループ変動コード移動 誘導変数削除 ループの強さ軽減 ループ展開		-	●	-	●	●	●
広域最適化 定数伝搬 定数畳み込み 共通部分式削除		-	-	●	●	●	●
コード掘り下げ コード巻き上げ		-	-	●	●	●	-
実行速度最適化		-	-	-	-	●	-
その他最適化 冗長コード削除 冗長変数削除 代数変換 ジャンプ最適化		-	●	●	●	●	●
レジスタ退避復帰共通化	Orp *2	- *1	●	●	●	●	●
	指定なし	-	-	-	●	-	-
抑止機能 レジスタ退避・復帰共通化抑止	Orpn *2	- *1	● *3	●*3	●*3	●*3	●*3
	指定なし	-	-	-	-	-	-
抑止機能 別名参照の最適化抑止	Oa	- *1	●	●	●	●	●
	指定なし	● *4	-	-	-	-	-

*1：グレーで網掛けしている部分は組み合わせ不可（ワーニング）であることを示します。

*2：Orp と Orpn の同時指定は不可（ワーニング）。

*3：Ol, Og, Ot, default の場合は、Orpn は指定の有無に関わらず、レジスタ退避復帰共通化をしません。Orpn が意味を持つ（共通化を抑止する）のは、Om のときだけです。

*4：Od の場合、Oa が指定されたものとみなして、別名参照の最適化抑止を行います。

opt_on プラグマおよび opt_off プラグマにこのプラグマが与える影響は、コマンドラインオプションの場合と同じです。

例 5.30

入力

```
source.c
command line : ccu8 /Tu8 /Om source.c


 int a, b, c, d;

#pragma OPT_OFF          /* Invalid by the command line option /Om */

void Om_func()           /* Performs optimization by /Om */
{
         a = b + c;
         if (a != 0) {
                 d = b + c;     /* Deletes the common expressions */
         }
}


#pragma OPTIMIZATION default   /* Equivalent to the case where a
                                  default value is used for the
                                  command line option */

                               /* #pragma OPT_OFF is valid */
void Od_func()                 /* Suppresses optimization */
{
         a = b + c;
         if (a != 0) {
                 d = b + c;
         }
}

#pragma OPTIMIZATION Ol Og     /* Equivalent to the case where the
                                  command line options are /Ol /Og */

#pragma OPT_ON                 /* #pragma OPT_ON is valid */
```

```
void default_func()             /* Performs default optimization */
{
         a = b + c;
         if (a != 0) {
                 d = b + c;     /* Deletes the common expressions */
         }
}

#pragma OPTIMIZATION Od         /* Equivalent to the case where the
                                   command line option is /Od */

                                /* #pragma OPT_ON is invalid */
void Od_func_again()            /* Suppresses optimization */
{
         a = b + c;
         if (a != 0) {
                 d = b + c;
         }
}
```

インライン関数定義と呼び出し側関数が異なる場合に、最適化を指定してコンパイルを行うと、インライン関数定義と呼び出し側関数の両方がインライン展開可能な最適化でコンパイルされる場合にのみ、インライン展開が実行されます。インライン展開が実行されなかった関数に対しては、コンパイラは最後に関数の本体となるコードを生成します。

例 5.31

入力

```
Source2.c
command line : ccu8 /Tu8 /Om source2.c

int Om_val, Od_val;

#pragma inline Om_Ifunc Od_Ifunc
void Om_Ifunc()             /* Generates the entity when inline
                               expansion is not performed */
{
         Om_val = 0;
}

#pragma OPTIMIZATION Od     /* Suppresses optimization */
void Od_Ifunc()             /* Generates the entity since inline
                               expansion is not performed */
```

```
{
        Od_val = 0;
}

#pragma OPTIMIZATION/* Restarts optimization */
void Om_func()
{
        Od_Ifunc();/* Inline expansion is not performed */
        Om_Ifunc();/* <---Inline expansion is performed */
}

#pragma OPTIMIZATION Od    /* Suppresses optimization */
void Od_func()
{
        Od_Ifunc();/* Inline expansion is not performed */
        Om_Ifunc();/* Inline expansion is not performed */
}
```

出力:

```
_Od_Ifunc       :

;;{

;;      Od_val = 0;
        mov     er0,      #0
        st      er0,      NEAR _Od_val

;;}
        rt

_Om_func :

;;{

;;      Od_Ifunc();/* Inline expansion is not performed */
        bl      _Od_Ifunc

;;      Om_Ifunc();/* <--Inline expansion is performed */
        mov     er0,      #0
        st      er0,      NEAR _Om_val

;;}
        rt
```

```
_Od_func :

;;{
        push    lr

;;      Od_Ifunc();/* Inline expansion is not performed */
        bl      _Od_Ifunc

;;      Om_Ifunc();/* Inline expansion is not performed */
        bl      _Om_Ifunc

;;}
        pop     pc

_Om_Ifunc :             ;<---- Generates the entity also, since
                        ;      inline expansion has not been
                        ;      performed.
;;{

;;      Om_val = 0;
        mov     er0,    #0
        st      er0,    NEAR _Om_val

;;}
        rt
```

以下の場合には、CCU8 はワーニングを表示し、プラグマを無視します。

- 関数本体の中でプラグマが指定された場合。
- 引数として指定された句が有効な引数ではない場合。
- 他のオプションと一緒にオプション‘Od’が指定された場合
- Orp、Orpn 以外の他のオプションと一緒にオプション‘default’が指定された場合
- オプション‘Om’と‘Ot’が同時に指定された場合
- オプション‘Orp’と‘Orpn’が同時に指定された場合

例 5.32

入力

```
source1.c
command line : ccu8 /Tu8 /Om source1.c
int a, b, c, d;

void Om_func()
{
        a = b + c;
        if (a != 0) {
                d = b + c;  /* Deletes the common expressions */
        }
}

#pragma OPTIMIZATION Od      /* Suppresses subsequent optimization */
void Od_func()
{
        a = b + c;
        if (a != 0) {
                d = b + c;
        }
}
#pragma OPTIMIZATION  /* Resets optimization to the original
                                state */

void Om_func_again()
{
        a = b + c;
        if (a != 0) {
                d = b + c;  /* Deletes the common expressions */
        }
}
```

出力:

CCU8 によって生成されるコードの一部。

```
_Om_func :

;;{
;;      a = b + c;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a
```

```
;;       if (a != 0) {
         mov     er0,      er0
         beq     _$L1

;;               d = b + c;  /* Deletes the common expressions */
         st      er0,      NEAR _d

;;       }
_$L1 :

;;}
         rt


_Od_func :

;;{
;;       a = b + c;
         l       er0,      NEAR _b
         l       er2,      NEAR _c
         add     er0,      er2
         st      er0,      NEAR _a

;;       if (a != 0) {
         mov     er0,      er0
         beq     _$L4

;;               d = b + c;  <- Suppresses optimization
         l       er0,      NEAR _b
         add     er0,      er2
         st      er0,      NEAR _d

;;       }
_$L4 :

;;}
         rt

_Om_func_again   :

;;{
;;       a = b + c;
         l       er0,      NEAR _b
```

```
                l       er2,       NEAR _c
                add     er0,       er2
                st      er0,       NEAR _a

;;              if (a != 0) {
                mov     er0,       er0
                beq     _$L7

;;                      d = b + c;  /* Deletes the common expressions */
                st      er0,       NEAR _d

;;              }
_$L7 :

;;}
                rt
```

5.9.2 OPT_ON プラグマおよび OPT_OFF プラグマ

構文：

```
#pragma OPT_ON

#pragma OPT_OFF
```

このプラグマは、Ver.3 よりも前のバージョンとの互換性のために残されています。最適化を関数ごとに細かく設定するには、OPTIMIZATION プラグマを使用して下さい。

opt_on プラグマは、このプラグマ以降に定義されている関数の最適化を行うようコンパイラに指示します。コマンドラインで最適化オプションを指定しないくても行われるデフォルトの最適化が行われます。コマンドラインで/Od オプションを指定すると、このプラグマは無視されます。

opt_off プラグマは、このプラグマ以降に定義されている関数の最適化をしないようコンパイラに指示します。コマンドラインで最適化オプションを指定しなくても行われるデフォルトの最適化を行わないようにします。コマンドラインで/Om オプションを指定すると、このプラグマは無視されます。

5.10 ASM プラグマおよび ENDASM プラグマ

構文：

```
# pragma ASM

...                    /* assembly instruction block */

#pragma ENDASM
```

asm および endasm プラグマは、#asm および#endasm 指令によく似ています。#pragma asm と#pragma endasm の間に任意のテキストを指定できます。CCU8 はこのテキストブロックを処理しません。このブロックは、ソースファイルに記述したとおりにアセンブリリストファイルに出力されます。

次の場合には CCU8 はワーニングを表示します。

- 対応する **asm** プラグマ無しで **endasm** プラグマを指定した場合

次の場合には CCU8 はフェイタルエラーメッセージを表示します。

- 対応する **endasm** プラグマ無しで **asm** プラグマを指定した場合

次の例は、 “#pragma asm ~ # pragma endasm”の使用方法を示しています。

例 5.33

入力

```
fn ()
{
# pragma asm
        cplc                ;;      invert carry flag
        di                  ;;      disable interrupt
# pragma endasm
}
```

CCU8 は、関数“fn”に対して次の関数本体を生成します。

出力

```
_fn     :

;;# pragma asm
        cplc                ;;      invert carry flag
        di                  ;;      disable interrupt

;;}
        rt
```

次の例は誤った使用例です。

例 5.34

入力

```
fn ()
{
# pragma endasm
# pragma asm
        cplc                ;;      invert carry flag
        di                  ;;      disable interrupt
# pragma endasm
}
```

対応する **asm** プラグマが指定されていないので、 CCU8 は、最初の **endasm** プラグマに対してワーニングメッセージを表示します。

例 5.35

入力

```
fn ()
{
# pragma asm
        cplc                ;;     invert carry flag
        di                  ;;     disable interrupt
}
```

対応する **endasm** プラグマが指定されていないので、 CCU8 は、**asm** プラグマに対してフェイタルエラーメッセージを表示します。

5.11 INLINEDEPTH プラグマ

構文：

#pragma INLINEDEPTH constant

inlinedepth プラグマは、インライン関数呼び出しのときに、インライン展開できる深さのレベルの数を指定します。constant には、0 から 255 の間の値を指定します。**inlinedepth** プラグマは、ソースファイル中のどこにでも何度でも指定できます。**inlinedepth** プラグマを宣言すると、その次のソース行 1 行だけに効果が及びます。

inlinedepth プラグマを指定しないと、インライン展開の深さの制限はデフォルトの 8 に設定されます。インライン展開の深さを 0 と指定すると、インライン関数は展開されません。

次の場合には CCU8 はワーニングを表示します。

- constant の値が 0 から 255 の間にない場合
- constant に数値以外のものを指定した場合

例 5.36

入力

```
int      a, b ;
#pragma  inline fn1 fn2
#pragma  inlinedepth 3
```

```
void fn1 ()
{
        a ++ ;
}
void fn2 ()
{
        b ++ ;
        fn1 () ;
}
void fn ()
{
        fn2 () ;
}
```

CCU8 は、関数“fn”に対して次のコードを生成します。

出力

```
_fn     :

;;      fn2 () ;
        l       er0,    NEAR _b
        add     er0,    #1
        st      er0,    NEAR _b
        l       er0,    NEAR _a
        add     er0,    #1
        st      er0,    NEAR _a

;;}
        rt
```

5.12 INLINERECURSION プラグマ

構文：

```
#pragma INLINERECURSIONON

#pragma INLINERECURSIONOFF
```

inlinerecursion プラグマは、インライン関数の再帰呼び出しを展開するかどうかを指定します。インライン関数の再帰呼び出しは、直接呼び出しの場合と間接呼び出しの場合があります。

inlinerecursion プラグマは、ソースファイルのどこにでも何度でも指定できます。

inlinerecursion プラグマを宣言すると、それ以降のソース行に効果を及ぼします。

inlinerecursionon プラグマを指定すると、インライン関数の再帰呼び出しが展開されます。再帰呼び出しが展開されるレベルの数は、インライン展開の深さの指定によって決まります。再帰呼び出しの展開レベルを超えた時点でワーニングが出力され、以降は展開されません。

inlinerecursionoff プラグマを指定すると、インライン再帰関数の呼び出しは展開されません。

inlinerecursionon プラグマを指定しないと、インライン再帰関数の呼び出しはデフォルトでは展開されません。

例 5.37

```
int     a ;
#pragma Inline           fn
#pragma Inlinedepth      3
#pragma Inlinerecursionon
void fn ()
{
        a ++ ;
        fn () ;
}
main ()
{
        fn () ;
}
```

CCU8 は、上の例に対して次のコードを生成します。

```
_main   :
;;{

;;       fn () ;
```

```
        l       er0,      NEAR _a
        add     er0,      #1
        add     er0,      #1
        add     er0,      #1
        st      er0,      NEAR _a
        bl      _fn

;;}
_$$end_of_main :
        bal     $


_fn     :

;;}
        l       er0,      NEAR _a
        add     er0,      #1
        add     er0,      #1
        add     er0,      #1
        add     er0,      #1
        st      er0,      NEAR _a
        b       _fn
```

5.13 STACKSIZE プラグマ

構文 :

#pragma STACKSIZE constant

stacksize プラグマは、スタックサイズを設定します。constant には、スタックサイズをバイト単位で指定します。スタックサイズとして 0x0 以上 0xffff 以下の範囲の偶数の値を指定します。このプラグマと/SS コマンドラインオプションは同じ動作をします。

/SS オプションをコマンドラインで指定すると、このプラグマは無視されます。ワーニングメッセージは出力されません。このプラグマをソースファイル中に何回も指定すると、CCU8 はワーニングメッセージを表示し、最初のプラグマで指定したスタックサイズが採用されます。入力ソースファイルに main 関数の定義がある時にのみ、このプラグマは有効です。

次の場合に CCU8 はワーニングを表示します。

- ソースファイル中にこのプラグマが複数ある場合

次の例は、誤った使用例です。

例 5.38

入力

```
# pragma STACKSIZE 3001
```

stacksize プラグマでスタックサイズとして奇数を指定しているため、CCU8 は上記のプラグマに対してワーニングメッセージを出力します。

5.14 NEAR プラグマと FAR プラグマ

構文：

```
#pragma NEAR

#pragma FAR
```

NEAR プラグマおよび FAR プラグマを使用すると、ソースファイルの一部に対してデフォルトのデータ指定子を指定できます。

NEAR プラグマは、以降のデータ(テーブルなど)変数のデフォルトのデータ指定子として、_near を指定します。

FAR プラグマは、以降のデータ(テーブルなど)変数のデフォルトのデータ指定子として、_far を指定します。

これらのプラグマは、データ(テーブルなど)変数に対して明示的に指定された__near 指定子や__far 指定子は無効にしません。

プラグマとコマンドラインオプションが一致していない場合、NEAR および FAR プラグマは、/near および/far コマンドラインオプションよりも優先されます。

出力ファイルに表示されるデータモデルは、コマンドラインのデータモデル指定子(明示的に指定されている場合もされていない場合もあります)によって決まります。ただし、コンパイルでは、ソースファイル内で NEAR/FAR プラグマを使用することによって、データモデルが複数使用されることがあります。このような場合、コンパイラは次のように解釈します。

(a) ソースファイル内に#pragma NEAR/ FAR が指定されていない場合

コードのコンパイルは、コマンドラインで指定された'near'モードまたは'far'モードで行われます(デフォルトは'near')。この場合、ソース全体がこのデータモデルでコンパイルされます。

(b) ソースファイル内で#pragma NEAR/ FAR が指定されている場合

NEAR プラグマまたは FAR プラグマが検出されるまで、コードのコンパイルは、コマンドラインで指定された'near'モードまたは'far'モードで行われます(デフォルトは'near')。これ以降はプラグマの指定が優先され、コンパイルはプラグマで指定されたデータモデルで実行されます。

(c) ソースファイル内で#pragma NEAR/ FAR が複数指定されている場合

NEAR プラグマおよび FAR プラグマは、複数回使用できます。このような場合、新しいプラグマが検出されると、新しいプラグマ以降のコードのコンパイルは、別の NEAR/FAR プラグマが検出されるかファイルの終わりになるまで、新しく指定されたデータモデルで実行されます。

例 5.39

<code in part 1>
#pragma NEAR
  <code in part 2>
#pragma FAR
  <code in part 3>
#pragma NEAR
  <code in part 4>

上記の例では、/far コマンドラインオプションでファイルがコンパイルされている場合、次のような解釈が行われます。

part 1 では、コンパイラはすべてのデフォルトデータ変数を far データとして扱います。

part 2 では、コンパイラはすべてのデフォルトデータ変数を near データとして扱います。

part 3 では、コンパイラはすべてのデフォルトデータ変数を far データとして扱います。

part 4 では、コンパイラはすべてのデフォルトデータ変数を near データとして扱います。

例 5.40

```
# pragma NEAR
int a, b ;
void
func1 ()
{
        a = b ;
}
# pragma FAR
int d, e ;
void
func2 ()
{
        d = e ;
}
```

CCU8 は、上記の例に対して、次のコードを生成します。

```
_func1   :

;;       a = b ;
         l       er0,      NEAR _b
```

```
                st      er0,      NEAR _a

        ;;}
                rt


        _func2  :

        ;;      d = e ;
                l       er0,      FAR _e
                st      er0,      FAR _d

        ;;}
                rt
```

5.15 FASTFLOAT プラグマ

構文:

```
#pragma FASTFLOAT
```

fastfloat プラグマは、高速エミュレーションライブラリを使用するようにコンパイラに指示します。

このプラグマは、float 型の実行時間を短くするためにサポートされています。このプラグマは、コマンドラインオプション/Ff が指定されている場合、無視されます。このプラグマが複数回指定されている場合、CCU8 はワーニングメッセージを出力します。

5.16 SEGMENT プラグマ

SEGMENT プラグマは、関数や変数を配置するセグメントを変更するプラグマです。SEGMENT プラグマには、以下のプラグマが用意されています。

SEGCODE プラグマ：	関数を指定したセグメントに配置します。
SEGINTR プラグマ：	割り込み関数を指定したセグメントに配置します。
SEGINIT プラグマ：	初期化付き変数を指定したセグメントに配置します。
SEGNOINIT プラグマ：	初期化なしの変数を指定したセグメントに配置します。
SEGCONST プラグマ：	const 修飾された変数を指定したセグメントに配置します。
SEGNVDATA プラグマ：	不揮発性メモリに配置する変数を指定したセグメントに配置します。

5.16.1 SEGCODE プラグマ

SEGCODE プラグマを使用すると、通常の関数(非割り込み関数)のアセンブリ言語出力に対し、再配置可能なセグメント名またはアブソリュートアドレスを指定できます。

構文:

形式１：`#pragma SEGCODE "SegmentName"`

形式２：`#pragma SEGCODE [segment:]offset`

形式３：`#pragma SEGCODE`

形式 1 は、通常の関数(非割り込み関数)のアセンブリ言語出力に対して、再配置可能なセグメント名を指定します。

形式 2 は、通常の関数(非割り込み関数)のアセンブリ言語出力を割り当てるための開始アドレスを指定します。指定可能なアドレスの範囲は以下のとおりです。

Small メモリモデルの場合、0x0:0x0004～0x0:0xfffe

Large メモリモデルの場合、0x0:0x0004～0xf:0xfffe

形式 3 は、割り込み関数を除くすべての関数のアセンブリ言語出力をデフォルトのセグメント名に割り当てることを指定します。デフォルトのセグメント名は、以下に示すとおりです。

/Zc オプションが指定されていない場合：

デフォルトのセグメント名は$$*funcname*$*filename* となります。

/Zc オプションが指定されている場合：

Small メモリモデルの場合、デフォルトのセグメント名は$$NCOD*filename* となります。

Large メモリモデルの場合、デフォルトのセグメント名は$$FCOD*filename* となります。

SEGCODE プラグマによって影響を受ける関数を識別するためのルール:

- 関数 X には本体がなければなりません。
- 関数 X はインライン関数であってはなりません。

- 関数 X は SWI プラグマまたは INTERRUPT プラグマで定義されていてはなりません。

上記のルールは、SEGCODE プラグマの有効範囲内にある関数に適用されます。

SEGCODE プラグマの有効範囲:

SEGCODE プラグマの有効範囲は、任意の形式の他の SEGCODE プラグマによって終了します。たとえば、現在有効な SEGCODE プラグマが形式 1 (セグメント名)であるものとします。
この有効範囲は、次のいずれかの形式で終了します。

- セグメント名による SEGCODE (形式 1)
- アブソリュートアドレスによる SEGCODE (形式 2)
- デフォルトの SEGCODE (形式 3)

複数の SEGCODE プラグマで、同じセグメント名またはアブソリュートアドレスを指定することはできません。2 番目の SEGCODE プラグマは無視されます。

他のポイント

- SEGCODE プラグマは、関数の本体に対してのみ影響を与えます。
- SEGCODE プラグマは、関数のプロトタイプには適用されません。
- SEGCODE プラグマは、INLINE 関数に対しては影響を与えません。
    - 展開されていないコードはデフォルトのセグメントになります。
    - 展開されたコードは呼び出し側関数と同じになります。
- SEGCODE プラグマは、割り込み関数には影響を与えません。割り込み関数は、SWI プラグマまたは INTERRUPT プラグマで定義されます。

- プラグマのキーワード SEGCODE は、大文字/小文字が区別されません。
- SEGCODE プラグマを関数本体内で宣言した場合、SEGCODE プラグマはその次の関数からに対して有効となります。

例 5.41

入力

```
int a;
```

```
#pragma Segcode "Segname1" /* Segment name is set to "Segname1"
                              Segment is Relocatable */
void fn1(void)
{
        a += 10;
}

#pragma Segcode 0x1000     /* Segment name is default
                              Segment is allocated to 0x1000 */
void fn2(void)
{
        a += 20;
}

#pragma Segcode "Segname3" /* Segment name is set to "Segname3"
                              Segment is Relocatable */
void fn3(void)
{
        a += 30;
}

#pragma Segcode            /* Segment name is default
                              Segment is Relocatable */
void fn4(void)
{
        a += 40;
}

#pragma Segcode 0x2000     /* Segment name is default
                              Segment is allocated to 0x2000 */
void fn5(void)
{
        a += 50;
}

#pragma Segcode            /* Segment name is default
                              Segment is Relocatable */
void fn6(void)
{
        a += 60;
}
```

5.16.2 SEGINTR プラグマ

SEGINTR プラグマを使用すると、割り込み関数のアセンブリ言語出力に対し、再配置可能なセグメント名またはアブソリュートアドレスを指定できます。

構文:

```
形式１：#pragma SEGINTR "SegmentName"

形式２：#pragma SEGINTR Address

形式３：#pragma SEGINTR
```

形式 1 は、割り込み関数のアセンブリ言語出力に対して、再配置可能なセグメント名を指定します。

形式 2 は、割り込み関数のアセンブリ言語出力を割り当てるための開始アドレスを指定します。指定可能なアドレスの範囲は、0x0004～0xfffe です。

形式 3 は、割り込み関数のアセンブリ言語出力をデフォルトのセグメント名に割り当てることを指定します。デフォルトのセグメント名は、以下に示すとおりです。

/Zc オプションが指定されていない場合：

デフォルトのセグメント名は、$$*funcname*$*filename* となります。

/Zc オプションが指定されている場合：

デフォルトのセグメント名は、$$INTERRUPTCODE となります。

SEGINTR プラグマによって影響を受ける関数を識別するためのルール:

- 関数 X には本体がなければなりません。
- 関数 X は SWI プラグマまたは INTERRUPT プラグマで定義されていなければなりません。

上記のルールは、SEGINTR プラグマの有効範囲内にある関数に適用されます。

SEGINTR プラグマの有効範囲:

SEGINTR プラグマの有効範囲は、任意の形式の他の SEGINTR プラグマによって終了します。たとえば、現在有効な SEGINTR プラグマが形式 1 (セグメント名)であるものとします。
この有効範囲は、次のいずれかの形式で終了します。

- セグメント名による SEGINTR(形式 1)
- アブソリュートアドレスによる SEGINTR (形式 2)
- デフォルトの SEGINTR (形式 3)

複数の SEGINTR プラグマで、同じセグメント名またはアブソリュートアドレスを指定することはできません。2 番目の SEGINTR プラグマは無視されます。

他のポイント

- SEGINTR プラグマは、関数の本体に対してのみ影響を与えます。
- SEGINTR プラグマは、関数のプロトタイプには適用されません。
- SEGINTR プラグマは、関数には影響を与えません。関数は、SWI プラグマまたは INTERRUPT プラグマで定義されます。割り込み関数の本体に対してのみ影響を与えます。
- プラグマのキーワード SEGINTR は、大文字/小文字が区別されません。
- SEGINTR プラグマは、関数本体内で宣言できます。

例 5.42

入力

```
int a;
static void intr_fn1(void);
static void intr_fn2(void);
static void intr_fn3(void);
static void intr_fn4(void);
static void intr_fn5(void);
static void intr_fn6(void);

#pragma interrupt intr_fn1 0x8  1
#pragma interrupt intr_fn2 0xA  1
#pragma interrupt intr_fn3 0xC  1
#pragma interrupt intr_fn4 0xE  1
#pragma interrupt intr_fn5 0x10 1
#pragma interrupt intr_fn6 0x12 1
```

```
#pragma Segintr "Segname1" /* Segment name is set to "Segname1"
                              Segment is Relocatable */
static void intr_fn1(void)
{
        a += 10;
}

#pragma Segintr 0x1000     /* Segment name is default
                              Segment is allocated to 0x1000 */
static void intr_fn2(void)
{
        a += 20;
}

#pragma Segintr "Segname3" /* Segment name is set to "Segname3"
                              Segment is Relocatable */
static void intr_fn3(void)
{
        a += 30;
}

#pragma Segintr            /* Segment name is default
                              Segment is Relocatable */
static void intr_fn4(void)
{
        a += 40;
}

#pragma Segintr 0x2000     /* Segment name is default
                              Segment is allocated to 0x2000 */
static void intr_fn5(void)
{
        a += 50;
}

#pragma Segintr            /* Segment name is default
                              Segment is Relocatable */
static void intr_fn6(void)
{
        a += 60;
}
```

5.16.3 SEGINIT プラグマ

SEGINIT プラグマは、初期化されたグローバル変数、静的グローバル変数、および静的ローカル変数を、ユーザが指定したセグメントに配置するために使用します。

構文:

a. /PF オプションが指定されている場合:

```
形式１：#pragma Seginit   [__near/__far] , "Name"

形式２：#pragma Seginit   [__near/__far] , [segment:]offset

形式３：#pragma Seginit   [__near/__far]
```

b. /PF オプションが指定されていない場合:

```
形式１：#pragma Seginit   [__near/__far]  "Name"

形式２：#pragma Seginit   [__near/__far]  [segment:]offset

形式３：#pragma Seginit   [__near/__far]
```

形式１は、以降の影響を受けるすべての変数をプラグマで名前が指定されている再配置可能なデータセグメントにグループ化し、さらにその初期値に対する“Name”+”TAB”という名前のテーブルセグメントを作成するよう、コンパイラに指示します。コンパイラは$$init_info も生成します。これは、初期値で変数を初期化するためにスタートアップルーチンによって使用されます。

形式 2 は、プラグマの対象範囲内で影響を受けるすべての変数を、プラグマで指定されているアドレスから開始するアドレスに割り当てるよう、コンパイラに指示します。指定可能なアドレスの範囲は、プラグマのデータアクセスタイプに依存します。

__near の場合、指定可能なアドレスの範囲は 0x0:0x0～0x0:0xffff です。

__far の場合、指定可能なアドレスの範囲は 0x0:0x0～0xff:0xffff です。

形式 3 は、変数の割り当てをデフォルトの状態に戻します。#pragma Seginit __near はそれ以前に指定された Seginit __near プラグマを解除します。そして#pragma Seginit __far はそれ以前に指定された Seginit __far プラグマを解除します。

データアクセス指定子[{__near | __far}]はオプションです。プラグマでデータアクセス指定を省略すると、現在アクティブなデータアクセス指定子が、プラグマに対するデータアクセスとして使用されます。つまり、次のような優先順位になります。

1. それより前のソースファイルで指定されている、最も近い near プラグマまたは far プラグマ。
2. near/far プラグマが指定されていない場合は、コマンドラインオプションで示されているデータアクセス。
3. 上記のいずれも指定されていない場合は、デフォルトのデータアクセス指定子 near と見なされます。

例 5.43

```
コマンドラインオプション:/far

#pragma far
#pragma Seginit __near “Name”

Seginit プラグマに対するデータアクセスは__near
```

例 5.44

```
コマンドラインオプション:/near

#pragma far
#pragma Seginit “Name”

 Seginit プラグマに対するデータアクセスは__far
```

例 5.45

```
コマンドラインオプション:/far

#pragma Seginit “Name”

Seginit プラグマに対するデータアクセスは__far
```

プラグマによって影響を受ける変数を識別するためのルール:

Seginit セグメントに配置する変数を決定するには、以下のルールが適用されます。

I　変数が__near で修飾されている場合

Seginit プラグマに対して、near プラグマまたはコマンドラインによって暗黙的に near データアクセスが指定されている場合、または、プラグマ宣言内の__near キーワードに

よって明示的に near データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、初期化された変数である。
2. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
3. 変数が、absolute プラグマまたは nvdata プラグマで指定されていない。

以上の場合には、変数はアクティブな Seginit near セグメントにグループ化されます。

II 変数が__far で修飾されている場合

Seginit プラグマに対して、far プラグマまたはコマンドラインによって暗黙的に far データアクセスが指定されている場合、または、プラグマ宣言内の__far キーワードによって明示的に far データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、初期化された変数である。
2. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
3. 変数が、absolute プラグマまたは nvdata プラグマで指定されていない。

以上の場合には、変数はアクティブな Seginit far セグメントにグループ化されます。

プラグマの有効範囲

Seginit プラグマが指定されている場合、その影響は以下の箇所まで継続します。

1. 同じデータアクセス指定子で他の Seginit プラグマが指定されるまで。
2. 同じアクセス指定子で Segnvdata プラグマ(名前形式またはアドレス形式)が指定されるまで。
3. または他のプラグマ(Seginit または Segnvdata)が指定されない場合は、ファイルの最後まで。

例 5.46

```
#pragma Seginit __near “SegNear1”
#pragma Seginit __far “SegFar1”
int __near Var1 =1;                    /* Grouped with SegNear1 */
int __far  Var2 =2;                    /* Grouped With SegFar1 */

#pragma Seginit __near “SegNear2”      /* Scope of SegNear1 is over */
```

```
int __near Var3 =3;                    /* Grouped with SegNear2 */
int __far Var4 =4;                     /* Grouped With SegFar1 */

#pragma Seginit __far “SegFar2”        /* Scope of SegFar1 is over */
int __near Var5 =5;                    /* Grouped With SegNear2 */
int __far Var6 =6;                     /* Grouped With SegFar2 */

#pragma Seginit __near 0x0:0x100       /* Scope of SegNear2 is over */
int __near Va7 =7;                     /* grouped with Seginit __near
                                          0x0:0x100 */
int __far Var8 =8;                     /* Grouped With SegFar2 */

#pragma Segnvdata __far “Segnvdata1”   /* Scope of Seginit __far is
                                          over */
int __near Var9 =9;       /* Grouped with Seginit __near  0x0:0x102 */
int __far Var10 =10;      /* Grouped with Segnvdata1 Segment */

#pragma Seginit __near    /* Scope of Seginit __near 0x0:0x100  is
                             over */
int __near Var11 =11;     /* Grouped with the default segment for
                             near */
int __far Var12 =12;      /* Grouped with Segnvdata1 Segment */
```

注意：Seginit プラグマで指定できるセグメント名の最大長は、CCU8 が識別子として認識できる最大の長さから 3 を引いた値になります。これは、初期化用のテーブルとして生成されるセグメントの名前には Seginit プラグマで指定したセグメント名の後に“TAB”という 3 文字が付加されるためです。なお、識別子の長さはコマンドラインオプション/SL で設定できます。デフォルトは 31 文字です。

5.16.4 SEGNOINIT プラグマ

Segnoinit プラグマは、初期化されないグローバル変数、静的グローバル変数、および静的ローカル変数に対するセグメント名またはセグメント開始アドレスを指定するために使用します。

構文:

a. /PF オプションが指定されている場合:

形式１：`#pragma Segnoinit [__near/__far] , “Name”`

形式２：`#pragma Segnoinit [__near/__far] , [segment:]offset`

```
形式３：#pragma Segnoinit   [__near/__far]
```

b. /PF オプションが指定されていない場合:

```
形式１：#pragma Segnoinit   [__near/__far]  “Name”

形式２：#pragma Segnoinit   [__near/__far]  [segment:]offset

形式３：#pragma Segnoinit   [__near/__far]
```

形式 1 は、影響を受ける変数をプラグマで名前が指定されている再配置可能なデータセグメントに割り当てるよう、コンパイラに指示します。

形式 2 は、プラグマの対象範囲内で影響を受けるすべての変数を、プラグマで指定されているアドレスから開始するアドレスに割り当てるよう、コンパイラに指示します。

形式 3 は、変数の割り当てをデフォルトの状態に戻します。#pragma SEGNOINIT __near を指定すると、それ以前に指定された SEGNOINIT __near プラグマを解除します。#pragma SEGNOINIT __far を指定すると、それ以前に指定された SEGNOINIT __far プラグマを解除します。デフォルトの状態は次のとおりです。

初期化されないグローバル変数は、共用シンボルとして定義されます。初期化されない静的グローバル変数、および静的ローカル変数に対して使用されるセグメント名は、SMALL モデルの場合は$$NVAR*filename* に、Large モデルの場合は$$FVAR*filename* になります。

データアクセス指定子[{__near | __far}]はオプションです。プラグマでデータアクセス指定子を省略すると、現在アクティブなデータアクセス指定子が、プラグマに対するデータアクセスとして使用されます。つまり、次のような優先順位になります。

1. それより前のソースファイルで指定されている、最も近い near プラグマまたは far プラグマ。
2. near/far プラグマが指定されていない場合は、コマンドラインオプションで示されているデータアクセス。
3. 上記のいずれも指定されていない場合は、デフォルトのデータアクセス指定子 near と見なされます。

例 5.47

```
コマンドラインオプション:/far

#pragma far
#pragma Segnoinit __near "Name"
```

```
int __near var;

Segnoinit プラグマに対するデータアクセスは__near
```

例 5.48

```
コマンドラインオプション:/near

#pragma far
#pragma Segnoinit "Name"
int __far var;

Segnoinit プラグマに対するデータアクセスは__far
```

例 5.49

```
コマンドラインオプション:/far

#pragma Segnoinit "Name"
int __far var;

Segnoinit プラグマに対するデータアクセスは__far
```

例 5.50

```
コマンドラインオプション:デフォルト(指定なし)

#pragma Segnoinit "Name"
int __near var;

Segnoinit プラグマに対するデータアクセスは__near
```

プラグマによって影響を受ける変数を識別するためのルール:

I　変数が__near で修飾されている場合

Segnoinit プラグマに対して、near プラグマまたはコマンドラインによって暗黙的に near データアクセスが指定されている場合、または、プラグマ宣言内の__near キーワードによって明示的に near データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
2. 変数が初期化されていない。

3. 変数が const で修飾されていない。
4. 変数が、absolute プラグマまたは nvdata プラグマで指定されていない。

以上の場合には、変数はアクティブな Segnoinit near セグメントにグループ化されます。

II 変数が__far で修飾されている場合

Segnoinit プラグマに対して、far プラグマまたはコマンドラインによって暗黙的に far データアクセスが指定されている場合、または、プラグマ宣言内の__far キーワードによって明示的に far データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
2. 変数が初期化されていない。
3. 変数が const で修飾されていない。
4. 変数が、absolute プラグマまたは nvdata プラグマで指定されていない。

以上の場合には、変数はアクティブな Segnoinit far セグメントにグループ化されます。

プラグマの有効範囲

Segnoinit プラグマが指定されている場合、その影響は以下の箇所まで継続します。

1. 同じデータアクセス指定子で他の Segnoinit プラグマが指定されるまで。
2. 同じアクセス指定子で Segnvdata プラグマ(名前形式またはアドレス形式)が指定されるまで。
3. 同じデータアクセスタイプの Segnoinit プラグマまたは Segnvdata プラグマがない場合は、現在のソースコードファイルの最後まで。

例 5.51

入力

```
#pragma SEGNOINIT __near "SegNear1"
#pragma SEGNOINIT __far "SegFar1"
int __near Var1 ;                    /* Grouped with SegNear1 */
int __far  Var2 ;                    /* Grouped With SegFar1 */

#pragma SEGNOINIT __near "SegNear2"/* Scope of SegNear1 is over */
int __near Var3 ;                  /* Grouped with SegNear2 */
int __far Var4 ;                   /* Grouped With SegFar1 */
```

```
#pragma SEGNOINIT __far "SegFar2"   /* Scope of SegFar1 is over */
int __near Var5 ;                   /* Grouped With SegNear2 */
int __far Var6 ;                    /* Grouped With SegFar2 */

#pragma SEGNOINIT __near 0x0:0x100 /* Scope of SegNear2 is over */
int __near Var7 ;                   /* Assigned to address 0x0:0x100*/
int __far Var8 ;                    /* Grouped With SegFar2 */

#pragma Segnvdata __far "Segnvdata1" /* Scope of SEGNOINIT __far is
                                        Over */
int __near Var9 ;                    /* Assigned to address 0x0:0x102
int __far Var10 ;                    /* Grouped with Segnvdata1
                                        Segment */

#pragma SEGNOINIT __near             /* Scope of SEGNOINIT __near
                                        0x0:0x100 is over */
int __near Var11 ;                   /* Assigned to default segment*/
int __far Var12 ;                    /* Grouped with Segnvdata1
                                        Segment */
```

出力

```
rseg SegNear1
_Var1 :
        ds 02h
        rseg SegFar1
_Var2 :
        ds 02h
_Var4 :
        ds 02h
        rseg SegNear2
_Var3 :
        ds 02h
_Var5 :
        ds 02h

        rseg SegFar2
_Var6 :
        ds 02h
_Var8 :
        ds 02h
        dseg #00h at 0100h
```

```
_Var7 :
         ds      02h
         dseg #00h at 0102h
_Var9 :
         ds      02h
rseg Segnvdata1
_Var10 :
         dw      00h
_Var12 :
         dw      00h
_Var11 comm data 02h #00h
```

5.16.5 SEGCONST プラグマ

Segconst プラグマは、ユーザー指定セグメント内で const で修飾されているグローバル変数、静的グローバル変数、または静的ローカル変数をグループ化するために使用されます。

構文:

形式１：`#pragma Segconst [ __near/__far] , “Name”`

形式２：`#pragma Segconst [ __near/__far] , [segment:]offset`

形式３：`#pragma Segconst [ __near/__far]`

b. /PF オプションが指定されていない場合:

形式１：`#pragma Segconst [ __near/__far] “Name”`

形式２：`#pragma Segconst [ __near/__far] [segment:]offset`

形式３：`#pragma Segconst [ __near/__far]`

形式 1 は、プラグマの対象範囲内で影響を受けるすべての変数を、プラグマで名前が指定されている再配置可能なテーブルセグメントにグループ化するよう、コンパイラに指示します。

形式 2 は、プラグマの対象範囲内で影響を受けるすべての変数を、プラグマで指定されているアドレスから開始するアドレスに割り当てるよう、コンパイラに指示します。指定可能なアドレスの範囲は、プラグマのデータアクセスタイプに依存します。

__near の場合、指定可能なアドレスの範囲は 0x0:0x0004～0x0:0xffff です。

__far の場合、指定可能なアドレスの範囲は 0x0:0x0004～0xff:0xffff です。

第 3 の形式は、変数の割り当てをデフォルトの状態に戻します。#pragma Segconst __near を指定すると、それ以前の Segconst __near プラグマが解除されます。#pragma Segconst __far を指定すると、それ以前の Segconst __far プラグマが解除されます。

データアクセス指定子[{__near | __far}]はオプションです。プラグマでデータアクセス指定を省略すると、現在アクティブなデータアクセス指定子が、プラグマに対するデータアクセスとして使用されます。つまり、次のような優先順位になります。

1. それより前のソースファイルで指定されている、最も近い near プラグマまたは far プラグマ。
2. near/far プラグマが指定されていない場合は、コマンドラインオプションで示されているデータアクセス。
3. 上記のいずれも指定されていない場合は、デフォルトのデータアクセス指定子 near と見なされます。

例 5.52

```
コマンドラインオプション:/far

#pragma far
#pragma Segconst __near “Name”

Segconst プラグマに対するデータアクセスは__near
```

例 5.53

```
コマンドラインオプション:/near

#pragma far
#pragma Segconst “Name”

Segconst プラグマに対するデータアクセスは__far
```

例 5.54

```
コマンドラインオプション:/far

#pragma  Segconst “Name”

Segconst プラグマに対するデータアクセスは__far
```

プラグマによって影響を受ける変数を識別するためのルール:

Segconst セグメントに入れる変数を決定するためには、以下のルールが適用されます。

I　　変数が__near で修飾されている場合

Segconst プラグマに対して、near プラグマまたはコマンドラインによって暗黙的に near データアクセスが指定されている場合、または、プラグマ宣言内の__near キーワードによって明示的に near データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
2. 変数が const で修飾されている。
3. 変数が、absolute プラグマで指定されていない。

以上の場合には、変数はアクティブな Segconst near セグメントにグループ化されます。

II　　変数が__far で修飾されている場合

Segconst プラグマに対して、far プラグマまたはコマンドラインによって暗黙的に far データアクセスが指定されている場合、または、プラグマ宣言内の__far キーワードによって明示的に far データアクセスが指定された場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
2. 変数が const で修飾されている。
3. 変数が、absolute プラグマで指定されていない。

以上の場合には、変数はアクティブな Segconst far セグメントにグループ化されます。

プラグマの有効範囲

Segconst プラグマが指定されている場合、その影響は以下の箇所まで継続します。

1. 同じデータアクセス指定子で他の Segconst プラグマが指定されるまで。

2. 他の Segconst プラグマが指定されない場合は、ファイルの最後まで。

例 5.55

入力

```
#pragma Segconstt  __near “SegNear”
#pragma Segconst  __far “SegFar”

cosnt int __near  nearvar1 =1 ;  /* Will be grouped with SegNear */
const int __far  farvar1 =1 ;    /* Will be grouped with SegFar */

#pragma Segconst  __near         /* Scope of SegNear is over */

const int __near nearvar2 =2 ;   /* Will be grouped with the default
                                    segment for near */
const int __far  farvar2 =2 ;    /* Will be grouped with SefFar */

#pragma Segconst  __far          /* Scope of SegFar is over */

const int __near  nearvar3 =3 ;  /* Will be grouped with the default
                                    segment for near */
const int _near  farvar3 =3 ;    /* will be grouped with the default
                                    segment for far */
```

5.16.6 SEGNVDATA プラグマ

Segnvdata プラグマは、不揮発性メモリに割り当てられているセグメント名またはセグメント開始アドレスに、変数のグループを割り当てるために使用されます。このプラグマは、グローバル変数、静的グローバル変数、または静的ローカル変数のグループを割り当てます。

構文:

a. /PF オプションが指定されている場合:

形式１：`#pragma Segnvdata  [__near/__far] , “Name”`

形式２：`#pragma Segnvdata  [__near/__far] , [segment:]offset`

形式３：`#pragma Segnvdata  [__near/__far]`

b. /PF オプションが指定されていない場合:

形式１：`#pragma Segnvdata  [__near/__far]  “Name”`

形式２：#pragma Segnvdata　[__near/__far]　[segment:]offset

形式３：#pragma Segnvdata　[__near/__far]

形式 1 は、プラグマの対象範囲内で影響を受ける変数を、プラグマで名前が指定されている再配置可能なデータセグメントに割り当てるよう、コンパイラに指示します。

形式 2 は、プラグマの対象範囲内で影響を受ける変数を、プラグマで指定されているアドレスから開始するアドレスに割り当てるよう、コンパイラに指示します。指定可能なアドレスの範囲は、プラグマのデータアクセスタイプに依存します。

__near の場合、指定可能なアドレスの範囲は 0x0:0x0～0x0:0xffff です。

__far の場合、指定可能なアドレスの範囲は 0x0:0x0～0xff:0xffff です。

形式 3 は、変数の割り当てをデフォルトの状態に戻します。#pragma SEGNVDATA __near を指定すると、それ以前に指定された SEGNVDATA __near プラグマを解除します。#pragma SEGNVDATA __far を指定すると、それ以前に指定された SEGNVDATA __far プラグマを解除します。デフォルトの状態は次のとおりです。

初期化されないグローバル変数は、共用シンボルとして定義されます。初期化されない静的グローバル変数、および静的ローカル変数に対して使用されるセグメント名は、SMALL モデルの場合は$$NVAR*filename* に、Large モデルの場合は$$FVAR*filename* になります。

初期化付きの変数は、__near データの場合は$$NINITVAR に、__far データの場合は$$FINITVAR*filename* になります。

データアクセス指定子[{__near | __far}]はオプションです。プラグマでデータアクセス指定を省略すると、現在アクティブなデータアクセス指定子が、プラグマに対するデータアクセスとして使用されます。つまり、次のような優先順位になります。

1. それより前のソースファイルで指定されている、最も近い near プラグマまたは far プラグマ。
2. または、near/far プラグマが指定されていない場合は、コマンドラインオプションで示されているデータアクセス。
3. 上記のいずれも指定されていない場合は、デフォルトのデータアクセス指定子 near と見なされます。

プラグマによって影響を受ける変数を識別するためのルール:

I　変数が__near で修飾されている場合

Segnvdata プラグマに対して、near プラグマまたはコマンドラインによって暗黙的に near データアクセスが指定されている場合、または、プラグマ宣言内の__near キーワードによって明示的に near データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
2. 変数が const で修飾されていない。
3. 変数が、nvdata プラグマまたは absolute プラグマで指定されていない。

以上の場合には、変数はアクティブな Segnvdata near セグメントにグループ化されます。

II 変数が__far で修飾されている場合

Segnvdata プラグマに対して、far プラグマまたはコマンドラインによって暗黙的に far データアクセスが指定されている場合、または、プラグマ宣言内の__far キーワードによる明示的な far データアクセスが指定されている場合で、かつ、変数が以下の条件を満たしている場合

1. 変数が、グローバル変数、静的グローバル変数、または静的ローカル変数である。
2. 変数が const で修飾されていない。
3. 変数が、nvdata プラグマまたは absolute プラグマで指定されていない。

以上の場合には、変数はアクティブな Segnvdata far セグメントにグループ化されます。

プラグマの有効範囲

Segnvdata プラグマが指定されている場合、その影響は以下の箇所まで継続します。

- 同じデータアクセス指定子で他の segnvdata プラグマが指定されるまで。
- データアクセス指定子が同じ segnoinit プラグマ(名前形式またはアドレス形式)が指定されるまで。
- データアクセス指定子が同じ seginit プラグマ(名前形式またはアドレス形式)が指定されるまで。
- データアクセスタイプが同じ segnvdata プラグマ、segnoinit プラグマ、または seginit プラグマがない場合は、現在のソースコードファイルの終了まで。

例 5.56

入力

```
#pragma SEGNVDATA __near "SegNear1"
#pragma SEGNVDATA __far "SegFar1"
int __near Var1  = 1 ;              /* Grouped with SegNear1 */
int __far  Var2  =  1 ;             /* Grouped With SegFar1 */

#pragma SEGINIT  __near "SegNear2" /* Scope of SegNear1 is over */
                                    /* but scope of  SegFar1 is still
                                       continue*/
int __near Var3  = 1 ;              /* Grouped with SegNear2 */
int __far Var4 ;                    /* Grouped With SegFar1 */

#pragma SEGNVDATA __far "SegFar2"  /*Scope of SegFar1 is over */
int __near Var5 = 1 ;               /* Grouped With SegNear2; */
int __far Var6 ;                    /* Grouped With SegFar2 */
```

上記の例に対しては、次のようなコードが生成されます。

出力

```
          rseg    SegNear1
_Var1 :
          dw      01h

          rseg    SegFar1
_Var2 :
          dw      01h

          rseg   SegFar1
_Var4 :
          dw      00h

          rseg SegFar2
_Var6 :
          dw      00h

          rseg $$init_info
          dw $$INITTAB_1
          dw $$INITVAR_1
          dw 4
          db seg $$INITTAB_1
          db seg $$INITVAR_1
          ……..
```

```
                ..................
                rseg   SegNear2TAB
        $$INITTAB_1 :
                dw      01h
                dw      01h

                rseg SegNear2
        $$INITVAR_1 :
        _Var3 :
                ds      02h
        _Var5 :
                ds      02h
```

5.17 SEGDEF プラグマ

構文:

a. /PF オプションが指定されている場合:

```
#pragma SEGDEF "segment_name", "segment_type"
```

b. /PF オプションが指定されていない場合:

```
#pragma SEGDEF "segment_name" "segment_type"
```

segdef プラグマは、セグメント定義の疑似命令がアセンブリ言語ファイルに対する出力であることを、コンパイラに対して指示します。__segbase_n、__segbase_f、および__segsize の各関数で参照されているセグメントシンボルが同じコンパイル単位で定義されていないときには、segdef プラグマを使用します。したがって、同じ名前のセグメントシンボルが同じコンパイル単位内で定義されている場合には、コンパイラはこのプラグマを無視します。

同じ名前のセグメントシンボルに対するセグメントタイプが異なる場合、CCU8 はワーニングメッセージを表示してプラグマを無視します。それ以外の場合には、ワーニングは表示されず、プラグマは無視されます。

segdef プラグマで指定するセグメント名は、デフォルトのセグメント、コンパイラによって自動的に生成されるセグメント、または segcode、segintr、seginit、segnoinit、segconst、segnvdata の各プラグマを使用してユーザーが定義するセグメントです。

セグメントタイプとしては、CODE、DATA、TABLE、または NVDATA を指定できます。セグメントタイプは、大文字でも小文字でも指定できます。他のセグメントタイプを指定すると、CCU8 はそのプラグマを無視し、プラグマに対するワーニングメッセージを表示します。

segdef プラグマによって定義されているセグメントシンボルが同じコンパイル単位内で参照されていない場合でも、CCU8 は、ワーニングメッセージを表示せずに、セグメント定義の疑似命令を出力します。

segdef プラグマによって生成されるセグメント定義命令の物理セグメント属性は ANY です。これは、プラグマはセグメントシンボルを参照するためだけに必要であるためです。segdef プラグマを使用する 2 つの例を次に示します。

例 5.57

入力

```
#pragma SEGDEF "SEGRAM" "DATA"
```

出力

```
SEGRAM  segment data any
```

例 5.58

入力

```
--------module1.c--------------
#pragma Segconst  "SEGMENT1"      /* Segment definition */
const int siGVar;
--------module2.c--------------
#pragma SEGDEF "SEGMENT1" "table" /* Indicates that Segment with name
                                     "SEGMENT1" has been defined in
                                     other compilation unit */
void fn(void)
{
  unsigned int uisize = __segsize("SEGMENT1"); /* Segment symbol is
}                                                 referenced */
```

segdef プラグマに対する疑似セグメント定義は、1 つのコンパイル単位 module2.c で生成され、別のコンパイル単位 Module1.c で定義されています。

出力

```
--------module1.asm--------------
  SEGMENT1 segment table 2h #0h   /* Actual segment definition */
  ..............
  ..............
  rseg SEGMENT1
_siGVar :
  dw  00h
--------module2.asm--------------
  ..............
```

```
.............
SEGMENT1  segment table any     /* Segment pseudo definition is
                                   generated for SEGDEF pragma */
.............
.............
l er0,  $$S1
.............
.............
    rseg $$NTABModule2
$$S1 :
  DW  SIZE  SEGMENT1
.............
.............
```

6. 出力ファイル

出力ファイルとデフォルトの拡張子を次の表に示します。

表 6.1	
出力ファイル	デフォルト 拡張子
*アセンブリファイル	.ASM
ソース/エラーリスト	.LER
**コールツリーリスト	-
プリプロセスファイル	.I
関数プロトタイプファイル	.PRO

* コマンドラインの/Fa オプションを使用して、アセンブリファイル名の拡張子を変更することができます。

**コールツリーリストファイルにはデフォルトの拡張子がありません。

出力ファイルを得るためのコマンドラインオプションを次の表に示します。

表 6.2	
出力ファイル	コマンドライン オプション
*アセンブリファイル	/Fa
ソース/エラーリスト	/LE
コールツリーリスト	/CT
プリプロセスファイル	/LP または /PC
関数プロトタイプファイル	/Zg

* /Fa オプションをコマンドラインに指定しないと、CCU8 がデフォルトのアセンブリファイル名でアセンブリファイルを生成します。

6.1 アセンブリファイル

CCU8 が生成する出力ファイルは、U8 コアのアセンブリニーモニックを含むアセンブリファイルです。

ここでは、コンパイラが出力コードを生成する際の規約について説明します。

6.1.1 コメントセクション

出力アセンブリファイルは、コメントセクションで始まります。コメントセクションは次の情報で構成されます。

1. コンパイルオプション
2. バージョン番号
3. ファイル名

6.1.1.1 コンパイルオプション

コマンドラインでファイル名と一緒に指定されたコンパイルオプションが、その順番で記述されています。

例 6.1

コマンドライン

```
C:¥>CCU8 /Tmu8 /MS /SS 10000 test.c
```

上のコマンドラインに対して、コンパイルオプションはコメントセクションに次のように出力されます。

出力

```
;; Compile Options : /Tmu8 /MS /SS 10000
```

6.1.1.2 バージョン番号

ソースファイルをコンパイルしたコンパイラのバージョンがコメントセクションに出力されます。

例 6.2

```
;; Version Number  : Ver.1.00
```

6.1.1.3 ファイル名

ユーザーがコマンドラインに指定したソースファイル名がコメントセクションに出力されます。

例 6.3

コマンドライン

```
C:¥>CCU8 /Tmu8 /MS ..¥source¥test.c
```

上のコマンドラインに対して次のようにコメントセクションにソースファイル名が出力されます。

出力

```
;; File Name       : ..¥source¥test.c
```

6.1.2 アセンブラ初期化擬似命令セクション

このセクションは、CCU8 が出力する RASU8 に必要な擬似命令で構成されます。

6.1.2.1 TYPE 擬似命令

TYPE 擬似命令が出力の始めに生成されます。/T オプションで指定した文字列が、この擬似命令と一緒に出力されます。

例 6.4

コマンドライン

```
C:¥>CCU8 /Tmu8 test.c <CR>
```

上のコマンドラインに対して、次の擬似命令が“test.asm”に出力されます。

出力

```
type (mu8)
```

6.1.2.2 MODEL 擬似命令

ハードウェアのメモリモデルとデフォルトのデータメモリアクセスをアセンブリリストファイル内で指定するのに MODEL 擬似命令を使用します。ハードウェアメモリモデルによって次のいずれかが出力されます。

small	small メモリモデル
large	large メモリモデル

デフォルトのデータメモリアクセスによって次のいずれかが出力されます。

near	near データアクセス指定子
far	far データアクセス指定子

例 6.5

コマンドライン

```
C:¥>CCU8  /MS  /Tmu8 test.c <CR>
```

上のコマンドラインにより、“test.asm”に次の擬似命令が出力されます。

出力

```
model small, near
```

例 6.6

コマンドライン

```
C:¥>CCU8  /MS /far /Tmu8 test.c <CR>
```

上のコマンドラインにより、“test.asm”に次の擬似命令が出力されます。

出力

```
model small, far
```

6.1.2.3 NOFAR 命令

MODEL 疑似命令のすぐ後に、NOFAR 疑似命令が生成されます。この疑似命令は、コマンド行で/nofar オプションが指定されていると生成されます。

例 6.7

コマンドライン

```
C:¥>CCU8  /Tmu8 /nofar test.c <CR>
```

上記のコマンド行に対しては、次のような疑似命令が test.asm に出力されます。

出力

```
nofar
```

6.1.2.4 セグメント定義擬似命令

アセンブリ出力ファイルのこのセクションは、使用されるすべての再配置可能なセグメントの定義で構成されます。

RLU8 Ver.1.60 から、参照されない関数・テーブルをリンクしない機能が追加されました。この機能への対応に伴い、CCU8 Ver.3.30 から、/Zc オプションの有無によって生成されるセグメント名が変更されます。

セグメント名	説明
$$*funcname*$*filename*	関数および割り込み関数（スモールモデル/ラージモデル）のセグメント定義を指定する。 (/Zc オプションを指定していない時のみ出力)
$$NCOD*filename*	関数（スモールモデル）のセグメント定義を指定する。 (/Zc オプション指定時のみ出力)
$$FCOD*filenam*	関数（ラージモデル）のセグメント定義を指定する。 (/Zc オプション指定時のみ出力)
$$INTERRUPTCODE	割り込み関数のセグメント定義を指定する。 (/Zc オプション指定時のみ出力)
$$TAB*constname*$*filename*	const で修飾されている near/far 変数のセグメント定義を指定する。 (/Zc オプションを指定していない時のみ出力)
$$NTAB*filename*	const で修飾されている near 変数のセグメント定義を指定する。 (/Zc オプション指定時のみ出力)
$$FTAB*filename*	const で修飾されている far 変数のセグメント定義を指定する。 (/Zc オプション指定時のみ出力)
$$NVAR*filename*	未初期化の near スタティック変数(グローバル/ローカル)のセグメント定義を指定する。
$$FVAR*filename*	未初期化の far スタティック変数(グローバル/ローカル)のセグメント定義を指定する。
$$NINITVAR	const 修飾子が付いていない、初期化対象の near 変数のセグメント定義を指定する。
$$NINITTAB	const 修飾子が付いていない、初期化対象の near 変数の初期値を指定する。
$$FINITVAR*filename*	const 修飾子が付いていない、初期化対象の far 変数のセグメント定義を指定する。
$$FINITTAB*filename*	const 修飾子が付いていない、初期化対象の far 変数の初期値を指定する。

$$init_info	const 修飾子が付いていない far 変数の初期化情報テーブルを指定する。
$$NNVDATA*filename*	nvdata プラグマで指定された near 変数のセグメント定義を指定する。
$$FNVDATA*filename*	nvdata プラグマで指定された far 変数のセグメント定義を指定する。
$$content_of_init	absolute プラグマで指定された、const 修飾子なしの初期化対象変数のセグメント定義を指定する。

それぞれのセグメント定義には、セグメントの名前とそのセグメントに対応するプロパティが含まれています。

例 6.8

```
$$NCOD file segment code 2h #0h
```

上のセグメント定義では、セグメント'$$NCOD*file*'を境界 2 の 0 番目の物理コードセグメントに配置することを示しています。

6.1.3 手続きセクション

このセクションは、ソースファイルに定義されているすべての関数に対して生成されるアセンブリ命令とアセンブリ指令で構成されます。

このセクションの内容は、さらに次のように分類できます。

1. 再配置可能セグメント定義
2. 関数名ラベル
3. C ソースレベルのデバッグ情報
   - CGLOBAL 指令
   - CSGLOBAL 指令
   - CARGUMENT 指令
   - CLOCAL 指令
   - CSLOCAL 指令
   - CLABEL 指令
   - CSTRUCTTAG 指令
   - CSTRUCTMEM 指令
   - CLINE 指令

- CBLOCK 指令
- CBLOCKEND 指令
- CFILE 指令
- CUNIONTAG 指令
- CUNIONMEM 指令
- CENUMTAG 指令
- CENUMMEM 指令
- CTYPEDEF 指令
- CFUNCTION 指令
- CFUNCTIONEND 指令
- C ソース行

4. 各ステートメントに対するアセンブリ命令

6.1.3.1 再配置可能なセグメント定義

関数は、その型によって決められたセグメントに置かれます。特定のセグメントに関数を置くには、'rseg'擬似命令を使用します。たとえば、関数を'NCOD*file*'セグメントに配置するように指定するには、次のような出力が必要です。

```
rseg NCOD file
```

例 6.9

入力

```
/* um608.c*/
void fn ()
{
}
```

出力

```
rseg $$NCODum608
```

ソースファイルに記述された順に、すべての関数がアセンブリファイルに出力されます。出力しようとする関数が直前の関数と同じセグメントに配置される時は、'rseg'指令は出力されません。

6.1.3.2 関数名ラベル

関数は、コロン(:)が後ろについた関数名で始まります。このラベルは、その後のアセンブリ命令群がこの関数コードの一部であることを示しています。関数名には、‘_’が前に付加されます。

例 6.10

入力

```
int func ()
{
}
```

上の例の関数‘func’に対して次のように関数名ラベルが出力されます。

出力

```
_func :
```

6.1.3.3 C ソースレベルデバッグ情報

6.1.3.3.1 CGLOBAL 指令

この指令は、グローバル変数定義の各々にに対して出力されます。

構文

```
CGLOBAL usg_typ attrib size “global_name” hierarchy
```

6.1.3.3.2 CSGLOBAL 指令

この指令は、静的グローバル変数定義の各々に対して出力されます。

構文

```
CSGLOBAL attrib size “static_global_name” hierarchy
```

6.1.3.3.3 CARGUMENT 指令

この指令は、引数変数定義の各々に対して出力されます。

構文

```
CARGUMENT attrib size offset “argument_name” hierarchy
```

6.1.3.3.4 CLOCAL 指令

この指令は、ローカル変数定義の各々に対して出力されます。

構文

```
CLOCAL attrib size offset block_id “local_name” hierarchy
```

6.1.3.3.5 CSLOCAL 指令

この指令は、静的ローカル変数定義の各々に対して出力されます。

構文

```
CSLOCAL attrib size alias_no block_id “static_local_name” hierarchy
```

6.1.3.3.6 CLABEL 指令

この指令は、ラベル定義の各々に対して出力されます。

構文

```
CLABEL label_no "label_name"
```

6.1.3.3.7 CSTRUCTTAG 指令

この指令は、構造体変数定義の各々に対して出力されます。

構文

```
CSTRUCTTAG fn_id block_id su_id tot_mem tot_size “tag_name”
```

6.1.3.3.8 CSTRUCTMEM 指令

この指令は、構造体メンバ変数定義の各々に対して出力されます。

構文

```
CSTRUCTMEM attrib size offset “member_name” hierarchy
```

6.1.3.3.9 CLINE 指令

CLINE 指令は、アセンブリ命令が生成される C 言語の実行文の各々に対して出力されます。

構文

```
CLINE line_atr line_no start_column end_column
```

CLINE 指令には、ソースファイル内の C 言語の文の行属性、行番号、開始カラム番号と終了カラム番号が付加されます。C ソースコード文の最初の行に対しては行属性フィールドは 0 になります。CCU8 はこのフィールドを使用して、ソースコード行の 1 行が最適化によって複数の部分に分けられてできる複数の行を区別します。

6.1.3.3.10 CBLOCK 指令、CBLOCKEND 指令

CBLOCK 指令は、ソースファイル内の‘{’の各々に対して出力されます。CBLOCK には関数番号、ブロック番号、および行番号も出力されます。同様に、ソースファイル内の‘}’の各々に対して CBLOCKEND 指令が出力され、あわせて関数番号、ブロック番号、および対応する行番号(CBLOCK 指令で指定)も出力されます。

構文

```
CBLOCK fn_id block_id line_no
CBLOCKEND fn_id block_id line_no
```

6.1.3.3.11 CFILE 指令

インクルードファイルの出力とソースファイルの出力を区別するために、CFILE 指令が出力されます。CFILE 指令にはファイル番号が付加されます。インクルードファイルを検出するたびに、そのインクルードファイルに対応するファイル番号と合わせて出力されます。

構文

```
CFILE file_id total_line “filename”
```

6.1.3.3.12 CUNIONTAG 指令

この指令は、共用体変数定義の各々に対して出力されます。

構文

```
CUNIONTAG fn_id block_id un_id tot_mem tot_size “tag_name”
```

6.1.3.3.13 CUNIONMEM 指令

この指令は、共用体メンバ変数定義の各々に対して出力されます

構文

```
CUNIONMEM attrib size “member_name” hierarchy
```

6.1.3.3.14 CENUMTAG 指令

この指令は、列挙型変数定義の各々に対して出力されます。

構文

```
CENUMTAG fn_id block_id enum_id tot_mem “tag_name”
```

6.1.3.3.15 CENUMMEM 指令

この指令は、列挙型メンバ変数定義の各々に対して出力されます。

構文

```
CENUMMEM value “member_name”
```

6.1.3.3.16 CTYPEDEF 指令

この指令は、typedef 変数定義の各々に対して出力されます。

構文

```
CTYPEDEF fn_id block_id attrib “type_name” hierarchy
```

6.1.3.3.17 CFUNCTION 指令

関数名ラベルの各々の前には、CFUNCTION 指令が付加されます。各々の CFUNCTION 指令には対応する関数番号があり、指令と一緒に出力されます。

構文

```
CFUNCTION fn_id
CFUNCTIONEND fn_id
```

6.1.3.3.18 C ソース行

アセンブリ命令群が出力される C の行の各々に対して、コメントとして対応する C 言語の文が出力されます。

例 6.11

入力

```
a = fn ();
```

上の C の文に対して、C ソース行が次のようにアセンブリファイルに出力されます。

出力

```
;; a = fn () ;
```

6.1.3.4 アセンブリ命令

1 行もしくは、複数のアセンブリ命令が、C の 1 文に対して生成されます。

例 6.12

入力

```
int b, c ;
void
fn ()
{
          b = fun1 () ;
          c += b ;
}
```

出力

```
…
;;       b = fun1 () ;
         bl      _fun1
         st      er0,      NEAR _b

;;       c += b ;
         l       er0,      NEAR _c
         l       er2,      NEAR _b
         add     er0,      er2
         st      er0,      NEAR _c
…
```

6.1.4 シンボル宣言セクション

このセクションは、ソースファイルで指定されているさまざまな型の変数のシンボル宣言で構成されます。

次の 3 種類のシンボル宣言があります。

1. comm
2. public
3. extrn

プラグマで指定されていない初期化されていないグローバルデータ変数は、‘comm’擬似命令を使って出力されます。

例 6.13

入力

```
long a;
```

出力

```
_a comm data 04h #00h
```

上の例では、‘a’は 4 バイトのサイズで 0 番目のデータセグメント内のロケーションに割り当てられます。

初期化済みグローバルデータ変数は、public 擬似命令で出力されます。

例 6.14

入力

```
int a = 7 ;
```

出力

```
public _a
```

上の例では、変数‘a’は public として出力されます。

関数呼び出しはあるが、関数本体がファイル中に定義されていない関数は、**‘extrn’**として出力されます。同様に、ソースに**‘extern’**として宣言されている変数もまた、‘extrn’として出力されます。

例 6.15

入力

```
extern int a ;
main ()
{
          a = 1 ;
          fn () ;
}
```

出力

```
.....
extrn code near : _fn
.....
extrn data near : _a
```

上の例では、関数‘fn’ の本体が定義されていないため、‘extrn’として出力されます。また、‘a’は**‘extern’**として宣言されています。そのため、記憶域が割り当てられず、‘extrn’として出力されます。

メモリ初期化擬似命令 DW と DB、およびメモリ割り当て擬似命令 DS は、初期化されるグローバルデータ変数の出力に使用します。CCU8 は、静的もしくは静的でない、集合体(配列、構造体、または共用体)の初期化されるグローバルデータ変数に対しても出力に同様に使用します。

例 6.16

入力

```
long var = 10 ;
const int cint = 20 ;
```

出力

```
                rseg $$NINITTAB
                dw      0ah
                dw      00h

                rseg $$NTABum615
_cint :
                dw      014h

                rseg $$NINITVAR
_var :
                ds      04h
```

上の例では、DS 擬似命令を使用して‘$$NINITVAR’データセグメントに 4 バイトが割り当てられます。初期値は、DW 擬似命令を使用して‘$$NINITTAB’に出力されます。同様に、**const** 変数 ‘cint’に対しては、DW 擬似命令を使用してメモリを割り当て初期化します。

グローバルデータおよび**静的**変数に対しては、RAM セグメントにこれらの変数のメモリを割り当て、ROM セグメントにその初期値を定義します。スタートアップコードは、“main”関数を呼び出す前にこの初期値を ROM セグメントから RAM セグメントにコピーします。

アセンブリファイルの出力例を次に示します。

例 6.17

入力

```
int      a, b ;
int      c = 10 ;
void fn ( void )
{
         b = fn1 () ;
         a = b * c ;
         return ;
}
```

出力

```
;; Compile Options : No command line options
;; Version Number  : <version>
;; File Name       : usr616.int

         type (mu8)
         model small, near
         $$NINITVAR segment data 2h #0h
         $$NINITTAB segment table 2h any
         $$NCODusr616 segment code 2h #0h
CFILE 0000H 00000009H "usr616.c"

         rseg $$NCODusr616

_fn      :
;;{
         push    lr

;;       b = fn1 () ;
         bl      _fn1
         st      er0,    NEAR _b

;;       a = b * c ;
         l       er2,    NEAR _c
         bl      __imulu8sw
         st      er0,    NEAR _a

;;}
```

```
        pop     pc

        extrn code near : _fn1
        public _c
        public _fn
        _a comm data 02h #00h
        _b comm data 02h #00h
        extrn code near : _main

        rseg $$NINITTAB
        dw      0ah

        rseg $$NINITVAR
_c :
        ds      02h
        extrn code : __imulu8sw

        end
```

6.2 エラーリスト

開発中のプログラムのデバッグでは、ソースリストが役に立ちます。このリストは、開発を終了したプログラムの構造をドキュメント化する際にも役に立ちます。

ソースリストには、ソースファイル内に記述された各関数の番号付きのソースコード行に診断メッセージが生成されて挿入されています。コンパイル中のエラー、またはワーニングメッセージが、次の例で示すように、エラーが発生した行の後に表示されます。

例 6.18

入力

```
int  a ;
int b ;

void fn ()
{
        output_fn () ;

        if ( a == b [1] )
                return a ;
}
```

上のプログラム“um617.c”を/LE/Tmu8 オプションでコンパイルすると、次のリストファイルが生成されます。

出力

```
                                                    Page :          1
                                                    Date : 09-22-2000
                                                    Time :  14:32:03
CCU8 C Compiler Ver.1.00, Source List
Source File : um617.c

Line #  Source Line

   1 int  a ;
   2 int b ;
   3
   4 void fn ()
   5 {
   6      output_fn () ;
   7      if ( a == b [1] )
***** um617.c(7) : Error : E5003 : Subscript on non array
   8         return a ;
   9 }
***** um617.c(8) : Error : E5038 : Void function returning value
  10


Error(s)   :  2
Warning(s) :  0
```

ソースファイルをエラーやフェイタルエラー無しでコンパイルできた場合は、エラーなしのリストファイルが生成されます。次の例は、エラーの無い全ソースリストです。

例 6.19

入力

```
int a ;
int b ;

void
begin ( x, y)
int x ;
int y ;
{
     function () ;
     end ( x, y) ;
     return ;
}
```

```
int
end ( x, y)
int x ;
int y ;
{
     int z ;
     z = function1 () ;
     function2 () ;
     z += x + y ;
     return (z) ;
}
```

出力:

```
                                                    Page :          1
                                                    Date : 02-09-2001
                                                    Time :  11:51:17
CCU8 C Compiler Ver.1.00, Source List
Source File : um618.c


Line #  Source Line

    1 int a ;
    2 int b ;
    3
    4 void
    5 begin ( x, y)
    6 int x ;
    7 int y ;
    8 {
    9     function () ;
    10    end ( x, y) ;
    11    return ;
    12 }
    13
    14 int
    15 end ( x, y)
    16 int x ;
    17 int y ;
    18 {
    19    int z ;
    20    z = function1 () ;
    21    function2 () ;
    22    z += x + y ;
    23    return (z) ;
    24 }
    25


Error(s)   :  0
Warning(s) :  0

                                          Page :          2
                                          Date : 02-09-2001
                                          Time :   11:51:17
```

```
CCU8 C Compiler Ver.1.00, Source List
Source File : um618.c
```

STACK INFORMATION

FUNCTION	LOCALS	CONTEXT	OTHERS	TOTAL
_begin	0	6	0	6
_end	2	8	0	10

6.3 コールツリーリスト

コールツリーリストファイルは、左マージンで手続き名を字下げしたリストです。呼び出しは、各レベルごとに空白 3 文字分の字下げをして表示します。

パスがすでに出力されている場合は、省略記号(...)が表示されます。再帰呼び出しにはアスタリスク(*)が付けられます。未定義の手続きを呼び出している場合には、疑問符(?)が付けられます。

例 6.20

入力

```
void
fn ()
{
}

void
fn1 ()
{
          fn () ;
          fn1 () ;
          fn2 () ;
}
```

上のソースファイル“um620.c”に対して、CCU8 は次のようなコールツリーリストを生成します。

```
CCU8 C Compiler, Ver.1.00, Calltree Listing

Source File : um620.c

fn
fn1
| fn...
| fn1*
| fn2?
```

上の例では、関数“fn”が前にリストされているので、2 番目の“fn”には省略記号が付いています。関数“fn1”は再帰的に呼び出されているので、関数“fn1”の後にアスタリスクが付いています。関数“fn2”の定義が関数呼び出しの前にないので、関数“fn2”の後に疑問符が付けられます。

複数のソースファイルがコンパイルに指定されている時には、それぞれのソースファイルのコールツリーリストが 1 つのコールツリーファイルに出力されます。しかし、1 つのソースファイルのコールツリーの情報が他のソースファイルのコールツリーの情報に持ち越されることはありません。

例 6.21

入力

```
/* um621a.c */
void fn ()
{
        fn () ;
}

/* um621b.c */
void fn1 ()
{
        fn () ;
}
```

上のコードでは、関数“fn”がソースファイル“um621a.c”に定義されており、“um621b.c”の中の関数“fn1”が関数“fn”を呼び出しています。

出力

```
CCU8 C Compiler, Ver.1.00, Calltree Listing

Source File : um621a.c

fn
| fn*

Source File : um621b.c

fn1
| fn?
```

関数“fn1”のコールツリーリストには、関数“fn”がソースファイル“um621b.c”で定義されていないので、疑問符が“fn”の後に付いています。

6.4 関数プロトタイプリスト

CCU8 は、/Zg オプションをコマンドラインに指定すると、ソースファイルの名前に拡張子 ".pro"が付いたプロトタイプリストファイルを作成します。関数プロトタイプファイルは、ソースファイルで定義している関数のプロトタイプで構成されています。

例 6.22

入力

```
/* um622.c */
int fn1 (long a, char *cptr)
{
         fn3 () ;
}

static void fn2 (int *iptr, char c)
{
}
```

CCU8 は、上のファイルを/Zg オプションでコンパイルすると、"um622.pro"を作成します。この関数プロトタイプファイルには、次の関数プロトタイプが含まれています。

```
extern  int fn1(long a,char *cptr);
static  void fn2(int *iptr,char c);
```

ソースファイルに定義された関数は、**static** として扱われない限り、関数プロトタイプファイル内で **extern** 関数として宣言されます。

例 6.23

入力

```
/* um623.c */
void fn1 (int *iptr, char c)
{
}
static void fn2 (int *iptr, char c)
{
}
extern void fn3 (int *iptr, char c)
{
```

```
}
```

CCU8 は、上のファイルを/Zg オプションでコンパイルすると、 “um623.pro” ファイルに次のプロトタイプを出力します。

```
extern  void fn1(int *iptr,char c);
static  void fn2(int *iptr,char c);
extern  void fn3(int *iptr,char c);
```

引数無しの関数は、関数プロトタイプファイルには **void** を引数として取る関数として宣言されます。

例 6.24

入力

```
/* um624.c */
void fn1 ()
{
}
```

CCU8 は、上のファイルを/Zg オプションでコンパイルすると、“um624.pro”ファイルに次のプロトタイプを出力します。

```
extern  void fn1(void);
```

戻り値の型の指定が無い関数は、関数プロトタイプファイルには **int** を戻す関数として宣言されます。

例 6.25

入力

```
/* um625.c */
fn1 ()
{
}
```

CCU8 は、上のファイルを/Zg オプションでコンパイルすると、“um625.pro”ファイルに次のプロトタイプを出力します。

```
extern  int fn1(void);
```

7. 最適化

CCU8 は、プログラムに必要な記憶領域の削減や実行時間の短縮のため、さまざまな最適化を行います。最適化は、不要な命令を削除したり、コードを編成し直すことによって行われます。

CCU8 は、次のような最適化を行います。

1. 少なくて早い命令を使用するようにコードの一部分を変更または移動します。
2. 冗長なコード、または未使用のコードを削除します。

CCU8 は、デフォルトではすべての最適化を行います。最適化オプション、/Od、/Ol、/Oa、/Og、/Ot および/Om を使用すれば、最適化の処理をきめ細かく制御できます。

7.1 広域的最適化

広域的最適化は、異なるコードの基本ブロックにまたがって行われます(基本ブロックとは、最初のステートメントから最後のステートメントへ順に制御が移ってゆく、連続した実行文のかたまりのことです)。

広域的最適化に含まれる最適化には次のものがあります。

1. 定数の伝搬
2. 定数の畳み込み
3. 共通部分式の削除
4. コードの掘り下げ
5. コードの巻き上げ

/Og オプションを使用して、この最適化を有効にしたり無効にすることができます。

7.1.1 定数の伝搬

式で使用される変数は、定数値に置き換えられます。

例 7.1

```
int a, b, x, y, m, n ;
const_propagate ()
{
       a = 45 ;
       if ( b < x)
       {
              m = a  ;      /* changed to m = 45 */
       }
       y = a + m ;          /* changed to y = 45 + m */
}
```

上記の関数 const_propagate に対し、CCU8 は次のようなアセンブリコードを生成します。

```
_const_propagate :

;;       a = 45 ;
         mov     er0,      #45
         st      er0,      NEAR _a

;;       if ( b < x)
         l       er0,      NEAR _b
         l       er2,      NEAR _x
         cmp     er0,      er2
         bges    _$L1

;;               m = a  ;         /* changed to m = 65 */
         mov     er0,      #45
         st      er0,      NEAR _m

;;       }
_$L1 :

;;       y = a + m ;       /* changed to y = 45 + m */
         l       er0,      NEAR _m
         add     er0,      #45
         st      er0,      NEAR _y

;;}
         rt
```

7.1.2 定数の畳み込み

結果の定数式はコンパイル時に計算され、計算された結果が式で使用されます。

例 7.2

```
int a, b, x, y, m, n ;
const_folding ()
{
       a = 45 ;
       if ( b < x)
       {
              m = a + 20 ;  /* changed to m = 65 */
       }
       y = a + m ;          /* changed to y = 45 + m */
}
```

上記の関数 const_folding に対し、CCU8 は次のようなアセンブリコードを生成します。

```
_const_folding :

;;       a = 45 ;
         mov     er0,      #45
         st      er0,      NEAR _a

;;       if ( b < x)
         l       er0,      NEAR _b
         l       er2,      NEAR _x
         cmp     er0,      er2
         bges    _$L1

;;               m = a + 20 ;     /* changed to m = 65 */
         mov     r0,#041h
         mov     r1,#00h
         st      er0,      NEAR _m

;;       }
_$L1 :

;;       y = a + m ;       /* changed to y = 45 + m */
         l       er0,      NEAR _m
         add     er0,      #45
         st      er0,      NEAR _y

;;}
```

```
        rt
```

7.1.3 共通部分式の削除

何回も繰り返される部分式を削除します。削除される部分式のところには、一度求めた結果を一時的に保存した値に置き換えられます。

例 7.3

```
int a, b, x, y, m, n ;
common_sub_exp ()
{
    x = a + b ;           /* a + b is also assigned to a temporary */
    if (a < b)
        m = a + b + y ;   /* a + b is replaced by the temporary */
    else
        n = (a + b) >> 4 ; /* a + b is replaced by the temporary */
}
```

上の関数‘common_sub_exp’に対して、CCU8 が生成するアセンブリコードを次に示します。

```
_common_sub_exp         :

        push    er4

;;      x = a + b ;/* a + b is also assigned to a temporary */
        l       er0,    NEAR _a
        l       er2,    NEAR _b
        add     er0,    er2
        st      er0,    NEAR _x
        mov     er4,    er0

;;      if (a < b)
        l       er0,    NEAR _a
        cmp     er0,    er2
        bges    _$L1

;;      m = a + b + y ;   /* a + b is replaced by the temporary */
        mov     er0,    er4
        l       er2,    NEAR _y
        add     er0,    er2
        st      er0,    NEAR _m

;;      else
        pop     er4
        rt
```

```
_$L1 :

;;       n = (a + b) >> 4 ; /* a + b is replaced by the temporary */
         mov     er0,      er4
         srlc    r0,       #04h
         sra     r1,       #04h
         st      er0,      NEAR _n

;;}
         pop     er4
         rt
```

7.1.4 コードの掘り下げ

別々の制御の流れで同じ文のシーケンスを実行した後合流している時には、共通するシーケンス部分を合流点まで掘り下げ(下に移動し)ます。不必要な重複した文は削除されます。

例 7.4

```
int a, b, e, x, y, z, m, n ;
sink ()
{
     if ( a == b )
     {
            func () ;
            m = e + 25 ;  /* two statements are sinked */
            return (e) ;
     }
     x = y + z ;
     m = e + 25 ;         /* two statements are removed */
     return (e) ;
}
```

上の関数‘sink’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_sink :
;;{
         push    lr

;;if ( a == b )
         l       er0,      NEAR _a
         l       er2,      NEAR _b
         cmp     er0,      er2
         bne     _$L1
```

```
;;func () ;
         bl      _func

;;}
_$L0 :
         l       er0,      NEAR _e
         add     er0,      #25
         st      er0,      NEAR _m
         l       er0,      NEAR _e

;;return (e) ;
         pop     pc

;;}
_$L1 :

;;x = y + z ;
         l       er0,      NEAR _y
         l       er2,      NEAR _z
         add     er0,      er2
         st      er0,      NEAR _x

;;return (e) ;
         bal     _$L0
```

7.1.5 コードの巻き上げ

これはコードの掘り下げと良く似ていますが、コードの移動方向が逆になります。異なる制御の流れで共通したところから分岐した後同じ文のシーケンスを実行している場合は、同じシーケンス部分を分岐点まで巻き上げ(上に移動し)ます。不必要な重複した文は削除されます。

例 7.5

```
int a, b, x, y, z, m ;
hoist ()
{
     if ( a == b )
     {
          m = x + y ;   /* statement hoisted */
          x = z ;
     }
     else
     {
```

```
            m = x + y ;   /* statement removed */
            fn1 () ;
        }
    }
```

上の関数‘hoist’ に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_hoist   :

;;{
         l       er0,      NEAR _x
         l       er2,      NEAR _y
         add     er0,      er2
         st      er0,      NEAR _m

;;       if ( a == b )
         l       er0,      NEAR _a
         l       er2,      NEAR _b
         cmp     er0,      er2
         bne     _$L1

;;               x = z ;
         l       er0,      NEAR _z
         st      er0,      NEAR _x

;;       else
         rt
_$L1 :

;;               fn1 () ;
         b       _fn1
```

7.2 ループの最適化

ループの最適化は、ループ内の文に対して実行されます。

ループの最適化に含まれる最適化には次のものがあります。

1. ループ不変コードの移動
2. ループ変動コードの移動
3. 誘導変数の削除
4. 強さの軽減
5. ループの展開

/Ol オプションを使用してこの最適化を有効にしたり、無効にしたりすることができます。

7.2.1 ループ不変コードの移動

毎回ループを実行しても値が変わらない式を、不変式と呼びます。このような式を検出するとループの外側に移動させ、一度だけしか評価しないようにします。

例 7.6

```
unsigned int x, m, n, o, p, r, i, y [10] ;
loop_invar ()
{
    do
    {
        p = n / o ;       /* moved outside the loop */
        x = m * r + i ;   /* sub-expression m * r is moved outside the
                             loop */
        y [i] += x ;
        i ++ ;
    } while ( x < i ) ;
}
```

上の関数‘loop_invar’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_loop_invar   :

          push    lr
          push    er8
          push    er4

;;{
          l       er0,      NEAR _n
          l       er2,      NEAR _o
          bl      __uidivu8sw
          st      er0,      NEAR _p
          l       er0,      NEAR _m
          l       er2,      NEAR _r
          bl      __uimulu8sw
          mov     er8,      er0

;;        do
_$L3 :

;;                x = m * r + i ;  /* sub-expression m * r is moved
                                      outside the loop */
          mov     er0,      er8
          l       er2,      NEAR _i
          add     er0,      er2
          st      er0,      NEAR _x

;;                y [i] += x ;
          add     er2,      er2
          l       er4,      NEAR_y[er2]
          add     er4,      er0
          st      er4,      NEAR_y[er2]

;;                i ++ ;
          l       er0,      NEAR _i
          add     er0,      #1
          st      er0,      NEAR _i

;;        } while ( x < i ) ;
          l       er0,      NEAR _x
          l       er2,      NEAR _i
          cmp     er0,      er2
          blt     _$L3
```

```
;;}
        pop     er4
        pop     er8
        pop     pc
```

7.2.2 ループ変動コードの移動

ループを実行するたびに一定の値だけ変わる式を、変動式と呼びます。このような式を検出するとループの外側に移動させ、その最終値(ループ出口での値)を一度だけ評価します。

例 7.7

```
int i, a ;
loop_variant_code_motion ()
{
   for ( i = 1 ; i < 11 ; i ++ )
        a += i ;
}
```

上のループは次のものに置き換えられます。

```
a += 55 ;
i = 11 ;
```

上の関数 loop_variant_code_motion に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_loop_variant_code_motion:

;;{
        l       er0,      NEAR _a
        add     er0,      #55
        st      er0,      NEAR _a
        mov     er0,      #11
        st      er0,      NEAR _i

;;}
        rt
```

7.2.3 誘導変数の削除

誘導変数とは、ループ内の他の変数または定数の関数によって値が変更される変数です。2 つ以上の誘導変数が存在すると、他の変数の線形関数である変数は削除され、その削除される変数が使用されているところのすべてがその変数の関数に置き換えられます。必要に応じて、削除された変数はその最終値で初期化されます。

例 7.8

```
char a [10] ;
int i, j ;
induction_var_elim ()
{
    for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
    {     /* i, j are induction variables */
          a [ i ] = j + 3 ;
    }
}
```

上のループは次のように変換されます。

```
j = 10 ;
for ( i = 0 ; i < 10 ; i ++)
{
      a [ i ] = i + 3 ;      /* j is replaced by i */
}
```

上の関数‘induction_var_elim’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_induction_var_elim            :

;;       for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
         mov     er0,      #0
         st      er0,      NEAR _i
         mov     er0,      #10
         st      er0,      NEAR _j

_$L3 :

;;               a [ i ] = j + 3 ;
         l       r0,       NEAR _i
         add     r0,       #03h
         l       er2,      NEAR _i
         st      r0,       _a[er2]

;;       for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
```

```
            move    er0,      er2
            add     er0,      #1
            st      er0,      NEAR _i

;;          for ( i = 0 , j = 0 ; i < 10 ; i ++ , j ++ )
            cmp     r0,       #0ah
            cmpc    r1,       #00h
            blts    _$L3

;;}
            rt
```

7.2.4 強さの軽減

操作が多くなるループ内の式は、操作が少なくなるよう変更されます。

例 7.9

```
int a [10] ;
int i ;
strength_reduction ()
{
     for ( i = 0 ; i < 10 ; i ++ )
     {
          a [i] = 0 ;     /* for accessing ith element of the 'a',
                             multiplication by 2 */
                          /* is necessary, because 'a' is an array of
                             'int' */
     }
}
```

上のループは次のように変換されます。

```
for ( i = 0, i1 = 0, temp = 0;  i < 10 ; temp += 2, i1 ++, i+= 2)
{
     * ( a + temp ) = 0 ; /* variable i1 is varies similar to that of
                             loop */
     /* control variable i. So strength reduction is
        performed on the variable i1. multiplication
        inside the loop is removed by introducing
        temporary variable (incremented by 2
        instead  of 1 because 'a' is an array of int).*/
}
```

上の関数‘strength_reduction’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_strength_reduction :

;;{
         push    er4

;;    for (i = 0, i1=0; i<10 ; i1++, i+=2)
         mov     er0,      #0
         st      er0,      NEAR _i
         st      er0,      NEAR _i1
         mov     er4,      er0
_$L3 :

;;        a[i1] = 0;       /* for accessing ith element of the
                                    'a', multiplication by 2 */
         mov     er2,      #0
         st      er2,      NEAR _a[er4]

;;    for (i = 0, i1=0; i<10 ; i1++, i+=2)
         l       er0,      NEAR _i1
         add     er0,      #1
         st      er0,      NEAR _i1
         add     er4,      #2
         l       er0,      NEAR _i
         add     er0,      #2
         st      er0,      NEAR _i

;;    for (i = 0, i1=0; i<10 ; i1++, i+=2)
         cmp     r0, #0ah
         cmpc    r1, #00h
         blts    _$L3

;;}
         pop     er4
         rt
```

7.2.5 ループの展開

一定の回数だけ実行されるループの本体を、可能なら繰り返し回数分だけ繰り返して展開します。ループ制御文は削除されます。

例 7.10

```
loop_unroll ()
{
    int i ;
    for ( i = 0 ; i < 2 ; i ++)
         function () ;
}
```

上のループは次のように変換されます。

```
function () ;
function () ;
```

上の関数‘loop_unroll’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_loop_unroll    :
;;{
        push    lr

;;{
        bl      _function
        bl      _function

;;}
        pop     pc
```

7.3 その他の最適化

他に実行される最適化には次のものがあります。

1. 冗長コードの削除
2. 冗長変数の削除
3. 代数による変換
4. ジャンプの最適化
5. const 変数の即値への置き換え

7.3.1 冗長コードの削除

決して実行されることのないコード部分を、「冗長」(dead)コードと呼びます。これらは、入力ソースプログラムを調べている際、もしくは定数伝搬などのような最適化前処理の際に冗長として検出できる文のかたまりです。

例 7.11

```
int a, p, q, r ;
dead_code ()
{
    a = 10 ;
    r = p + q ;
    if ( a < 10)    /* if statement removed */
           fn1 () ; /* statement removed */
}
```

上の関数‘dead_code’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_dead_code      :

;;      a = 10 ;
        mov     er0,    #10
        st      er0,    NEAR _a

;;      r = p + q ;
        l       er0,    NEAR _p
        l       er2,    NEAR _q
        add     er0,    er2
        st      er0,    NEAR _r

;;}
```

```
                rt
```

7.3.2 冗長な変数の削除

変数は式によって値が代入されます。変数の中には、その値がプログラムで使用されないことがあります。このような変数を「冗長な変数」(dead variable)と呼びます。冗長な変数を検出すると削除します。また冗長な変数には、代入された値が使用される前に、新たに値が代入されるような変数も含みます。不必要な代入が削除されます。

例 7.12

```
int x, m, n, r, p, q ;
dead_var ()
{
     int l ;
     x = m + n ;     /* statement removed */
     r = p * q ;
     x = r >> 2 ;
     l = x + r ;     /* statement is removed variable l is a dead
                        variable */
}
```

上の関数‘dead_var’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_dead_var        :
;;{
        push     lr

;;      r = p * q ;
        l        er0,    NEAR _p
        l        er2,    NEAR _q
        bl       __imulu8sw
        st       er0,    NEAR _r

;;      x = r >> 2 ;
        srlc     r0,     #02h
        sra      r1,     #02h
        st       er0,    NEAR _x
;;}
        pop      pc
```

7.3.3 代数による変換

レジスタを最適に使用するために交換則と結合則に従って、式を変更します。

例 7.13

```
int a, x, b, c ;
alg_transfer ()
{
    a = x + ( b - c ) ;
}
```

上の文は次のように変更されます。

```
a = ( b - c ) + x ;
```

上の関数‘alg_transfer’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_alg_transfer     :

;;       a = x + ( b - c ) ;
         l       er0,      NEAR _b
         l       er2,      NEAR _c
         sub     r0,       r2
         subc    r1,       r3
         l       er2,      NEAR _x
         add     er0,      er2
         st      er0,      NEAR _a
;;}
         rt
```

7.3.4 ジャンプの最適化

ジャンプ命令の使用が最小限になるようにコードブロックを再配置します。ジャンプ命令へジャンプするジャンプ命令は、ジャンプの実行回数を減らすように変更されます。

例 7.14

```
LABEL2 :
   goto LABEL1;     /* LABEL1 is replaced by LABEL2 */
LABEL1 :
   goto LABEL2;
```

7.3.5 const 変数の即値への置き換え

この最適化は、const で修飾されている変数を即値に置き換えることで実行されます。この最適化は、/Od オプション以外の最適化が指定されていると有効になります。

例 7.15

```
const char c = 10 ;          /* Handled using the immediate value */
static const char sc = 20 ;  /* Handled using the immediate value */

char x , y ;
void Const_Repl_Fun ()
{
         x = c ;    /* The value of ‘c’ is directly moved to x */
         y = sc ;   /* The value of ‘sc’ is directly moved to y */
}
```

上記の関数 Const_Repl_Fun に対して CCU8 が生成するアセンブリコードを次に示します。

```
_Const_Repl_Fun  :

;;{

;;    x = c ;
         mov     r0, #0ah
         st      r0, NEAR _x

;;    y = sc ;
         mov     r0, #014h
         st      r0, NEAR _y

;;}
         rt
```

7.4 覗き穴最適化

覗き穴最適化は、出力されたアセンブリ言語命令に対して行われます。

この最適化に含まれる最適化には次のものがあります。

1. 冗長な転送命令の削除
2. 相対ジャンプの最適化

7.4.1 冗長な転送命令の削除

生成されたアセンブリ命令に対して、レジスタとの不要な転送をしていないかどうかを調べます。

例 7.16

```
l     r0,   #20h
st    r0,   _one
l     r0,   #20h    ; this instruction is removed
st    r0,   _two
```

7.4.2 相対ジャンプの最適化

ジャンプ先が許可範囲を超えている相対ジャンプ命令は、条件付ジャンプ命令と無条件ジャンプ命令の組み合わせに置き換えます。条件付きジャンプ命令と無条件ジャンプ命令が連続している時は、その組み合わせを 1 つの条件付きジャンプ命令に置き換えます。

例 7.17

```
   bne  _$L2
   bal  _$L1
_$L2 :
        :
        :
_$L1 :
```

上の命令は次のように置き換えられます。

```
   beq  _$L1
        :
        :
_$L1 :
```

7.4.3 テールリカージョン最適化

生成されたアセンブリ命令でリターン命令の前の関数呼び出しがスキャンされ、絶対ジャンプに変換されます。

例 7.18

```
bl      _fn1
rt
```

上の命令は次のように置き換えられます。

```
b       _fn1
```

7.5 局所最適化

基本ブロック内で行われる最適化には次のものがあります。

1. 定数の伝搬
2. 定数の畳み込み
3. 共通部分式の削除
4. 代数恒等式の利用
5. 代数的な変換
6. コピー伝搬

基本ブロック内で行われるこれらの最適化は、最適化オプションのどれにも影響されません。常に最適化が行われます。

7.5.1 定数の伝搬

式で使用される変数を解析し、定数にできるものは変更します。

例 7.19

```
int c, d ;
local_constant_prop ()
{
     c = 30 ;
     d = c ;     /* instead of c, 30 is assigned to d */
}
```

上の関数‘local_constant_prop’に対して、CCU8 が生成するアセンブリコードを次に示します。

```
_local_constant_prop      :

;;      c = 30 ;
        mov      er0,      #30
        st       er0,      NEAR _c

;;      d = c ;  /* instead of c, 30 is assigned to d */
        st       er0,     NEAR _d

;;}
        rt
```

7.5.2 定数の畳み込み

結果の式は、定数が分析されて、可能であればコンパイル時に計算されます。

例 7.20

```
int  d ;
local_constant_folding ()
{
         d = 30 + 20 ;     /* the resultant value 50 computed at */
                           /* compile time  is assigned to ‘d’ */
}
```

上記の関数 local_constant_prop に対し、CCU8 は次のようなアセンブリコードを生成します。

```
_local_constant_folding    :

;;{

;;       d = 30 + 20 ;            /* the resultant value 50 computed
                                     at */
         mov     er0,      #50    /* compile time is assigned */
         st      er0,      NEAR _d

;;}
         rt
```

7.5.3 共通部分式の削除

部分式の繰り返しがあるコードを、部分式を一度評価するだけですませるように変更します。

例 7.21

```
unsigned int a, b, c, d, x, y ;
local_cse ()
{
     a = b + c * d ;        /* c *d is evaluated and assigned to a
                               temporary */
     x = c * d / y ;        /* value of c * stored in the temporary is
                               used not evaluated again */
}
```

上の関数‘local_cse’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_local_cse      :

         push    lr
         push    er4

;;       a = b + c * d ;   /* c *d is evaluated and assigned to a
                               temporary */
         l       er0,      NEAR _c
         l       er2,      NEAR _d
         bl      __uimulu8sw
         mov     er2,      er0
         l       er4,      NEAR _b
         add     er2,      er4
         st      er2,      NEAR _a

;;       x = c * d / y ;   /* value of c * stored in the temporary is
                               used not evaluated again */
         l       er2,      NEAR _y
         bl      __uidivu8sw
         st      er0,      NEAR _x

;;}
         pop     er4
         pop     pc
```

7.5.4 代数恒等式の利用

基本演算 ID (0 と 1)を使用する代数法則に準拠する式を変更し、不必要な演算を除去します。この処理では、以下の式だけが考慮されます。

a+0
a-0
a*0
a*1
a/1

a は整数型の変数です。

例 7.22

```
int a, b, c, d ;
alg_identities ()
{
    a = b + 0 ;    /* addition is eliminated */
    c = d * 1 ;    /* multiplication is eliminated */
}
```

上の関数‘alg_identities’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_alg_identities  :

;;      a = b + 0 ;        /* addition is eliminated */
        l         er0,     NEAR _b
        st        er0,     NEAR _a

;;      c = d * 1 ;        /* multiplication is eliminated */
        l         er0,     NEAR _d
        st        er0,     NEAR _c

;;}
        rt
```

7.5.5 代数的な変換

交換法則と結合法則を使用して、レジスタの使用が最適化されるよう式を変更します。

例 7.23

```
int a, x, b, c ;
alg_transfer ()
{
        a = x + ( b - c ) ;
}
```

上記のステートメントは、次のように変換されます。

```
a = ( b - c ) + x ;
```

上の関数‘alg_transfer’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_alg_transfer    :

;;       a = x + ( b - c ) ;
         l       er0,      NEAR _b
         l       er2,      NEAR _c
         sub     r0, r2
         subc    r1, r3
         l       er2,      NEAR _x
         add     er0,      er2
         st      er0,      NEAR _a

;;}
         rt
```

7.5.6 コピー伝搬

変数 x と y に対する代入 x <- y があるとすると、この最適化では、途中のステートメントによって x または y の値が変更されない限り、以降の x の使用を y の使用に置き換えます。この最適化は、直線的なコードの中でのみ行われます。この最適化の結果により、冗長変数の削除のような他の最適化を実行できる可能性があります。

この最適化は、基本型(char、short、int、および long の符号ありバージョンまたは符号なしバージョン)およびこれらの基本型に対するポインタに対してのみ実行されます。

2 つの変数のサイズが同じでも符号が異なる場合は、この最適化は実行されません。

例 7.24

```
int g, j, k ;
void
copy_propagation(int a)
{
         k = a ;
         g = j + k ;
}
```

上記のプログラムは、次のように変換されます。

```
int g, j, k ;
void
copy_propagation(int a)
{
          k = a ;
          g = j + a ;      /* 'k' is replaced with 'a' */
}
```

上の関数‘alg_transfer’に対して、CCU8 が生成するアセンブリコードを以下に示します。

```
_copy_propagation :

;;{

;;       k = a ;
         st      er0,      NEAR _k

;;       g = j + k ;
         l       er2,      NEAR _j
         add     er0,      er2
         st      er0,      NEAR _g

;;}
         rt
```

7.6 最適化における別名参照の効果

「別名」とは、すでに他の名前で参照しているメモリの同じロケーションを参照するのに使用するもう一つの名前です。

ロケーションは複数の変数で参照できるため、変数に対して最適化を行うのは安全ではありません。デフォルトでは、CCU8 は別名のチェックは行いません。次にあげる前提を守らないと、CCU8 がデフォルトで行う最適化によって安全でないコードを生成することがあります。

1. 変数を直接使用する場合には、その変数を参照するのにポインタを使用しない。
2. ポインタを使用して変数を参照する場合には、その変数を直接参照しない。
3. ポインタを使用してメモリのロケーションの内容を変更する場合には、同じロケーションをアクセスするのに他のポインタを使用しない。

「参照」とは、代入式の右辺または左辺で変数を使用すること、または関数呼び出しの引数として変数を使用することです。

コマンドラインオプション/Oa を指定すると、CCU8 は最適化の実行中に別名をチェックします。これによって正しいコードを得ることができますが、最適化範囲が狭くなります。

例 7.25

```
int a, b, c, x, y, *ptr ;
alias_check ()
{
     a = b + c ;
     if (x < a)
     {
          * ptr = 56 ;
          y = b + c ;     /* By default, alias are ignored, so   */
                          /* b + c, evaluated earlier is used    */
                          /* if /Oa option is specified b + c is */
                          /* evaluated again                     */
     }
}
```

上のコード部分では、デフォルトで共通部分式の削除の最適化が行われます。部分式‘b + c’は一度だけ評価され、2 回目の評価は行われずに一時的に格納された値が使用されます。

ポインタ‘ptr’が変数‘b’ または変数‘c’を指していなければ、最適化の結果は正しくなりますが、ポインタ‘ptr’が変数‘b’ または変数‘c’を指していると、共通部分式を削除してしまうと正しくない値を ‘y’に代入してしまいます。

上のコード部分を/Oa オプションを使用してコンパイルすると、部分式‘b + c’の評価は最適化されません。したがって、正しい値が‘y’に代入されます。

上の関数‘alias_check’に対して、デフォルトのコマンドラインオプション(すべての最適化を行う)で CCU8 が生成するアセンブリコードを以下に示します(/Oa オプション無し)。

```
_alias_check      :
         push    bp
         push    er4

;;       a = b + c ;
         l       er0,      NEAR _b
         l       er2,      NEAR _c
         add     er0,      er2
         st      er0,      NEAR _a
         mov     er4,      er0

;;       if (x < a)
         l       er2,      NEAR _x
         cmp     er2,      er0
         bges    _$L1

;;               * ptr = 56 ;
         l       bp,       NEAR _ptr
         mov     er0,      #56
         st      er0,      [bp]

;;               y = b + c ;  /* By default, alias are ignored, so  */
         st      er4,      NEAR _y

;;       }
_$L1 :

;;}
         pop     er4
         pop     bp
         rt
```

上の関数‘alias_check’に対して、/Oa オプションをコマンドラインで指定したとき(別名チェックを行う)に CCU8 が生成するアセンブリコードを以下に示します。

```
_alias_check      :
        push    bp

;;      a = b + c ;
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _a

;;      if (x < a)
        l       er2,      NEAR _x
        cmp     er2,      er0
        bges    _$L1

;;              * ptr = 56 ;
        l       bp,       NEAR _ptr
        mov     er0,      #56
        st      er0,      [bp]

;;              y = b + c ;  /* By default, alias are ignored, so  */
        l       er0,      NEAR _b
        l       er2,      NEAR _c
        add     er0,      er2
        st      er0,      NEAR _y

;;      }
_$L1 :

;;}
        pop     bp
        rt
```

8. コンパイラ出力の改善

8.1 最適化の制御

CCU8 には、プログラムの実行速度を改善する最適化オプションがいくつかあります。また、ソースプログラム内で局所的にループおよびグローバルな最適化を制御するプラグマもあります。

8.1.1 デフォルトの最適化

デフォルトでは、CCU8 はすべての最適化を行います。最適化をまったく行わないようにするには、/Od オプションを指定しなければなりません。

8.1.2 別名チェックの緩和

デフォルトの最適化では、CCU8 は安全ではない最適化を行ってしまいます。コマンドラインオプション/Oa を指定すれば安全な最適化になります。しかし、/Oa オプションを指定すると、出力のサイズが大きくなり、実行速度も遅くなります。

複数の別名が同じロケーションを参照していても、それが直接的にも間接的にも使用されていない時には、ユーザーは/Oa オプションを省略しても安全です。また、別名がプログラム内で使用されていたとしても、1 つの関数内で同じロケーションを複数の名前で参照していなければ、やはり/Oa を省略しても安全です。

8.1.3 局所的な最適化の制御

コマンド行から制御できるすべての最適化は、optimization プラグマを使用して制御することもできます。このプラグマを使用すると、ソースファイル内の個別の関数に対する最適化を有効または無効にできます。このプラグマの引数としては Od、Om、Ot、Og、Ol、Oa、および default があり、それぞれの引数によって、それ以降の関数に対して適用する最適化を指定することができます。

詳細については、5.9 節を参照してください。

8.1.4 最大限の最適化

コマンドラインオプション/Om によって、コンパイラは最大限の最適化を行います。デフォルトでは、すべての最適化は一度しか行われません。しかし、/Om オプションを指定すると、CCU8 はそれ以上の最適化を行えなくなるまで一連の最適化を繰り返し実行します。

/Om オプションに/Oa オプションも指定すると、安全で、かつ、最大限の最適化が行われた出力結果を得ることができます。

8.1.5 実行速度の最適化

コマンドラインオプション/Ot によって、コンパイラは実行速度の最適化を行います。

/Ot オプションに/Oa オプションも指定すると、安全で、かつ、実行速度の最適化が行われた出力結果を得ることができます。

8.1.5.1 乗算に対する実行速度の最適化

乗算に対する実行速度の最適化は、/Ot オプションを指定した場合にのみ適用されます。

乗算 A*B で、A が符号ありまたは符号なしの変数、B が $2^n \pm m$ または 2^n という値の正の整数定数である場合、実行速度の最適化には以下の条件が適用されます。

被乗数 A が char または unsigned char の変数の場合は、乗数である正整数 B をバイナリ表現に変換し、1 または[-1]を示すビットの総数が 3 以下のときには、実行速度の最適化が実行されます。

例 8.1

char 型の乗算

入力

```
char A, Res;

void main()
{
        Res = A * 20;     /* 20 ---> 10100(2)  */
}
```

/Ot オプションを指定して例 8.1 をコンパイルすると、次のようなアセンブリコードが生成されます。

出力

```
;;  Res = A * 20;          /* 20 ---> 10100(2)  */
l       r0,     NEAR _A
mov     r1,     r0
sll     r0,     #02h
add     r0,     r1
sll     r0,     #02h
st      r0,     NEAR _Res
```

/Ot オプションを指定しないと、上記の例に対して生成されるアセンブリコードは次のようになります。

出力

```
;;  Res = A * 20;          /* 20 ---> 10100(2)  */
l       r0,     NEAR _A
mov     r2,     #014h
mul     er0,    r2
st      r0,     NEAR _Res
```

被乗数 A が int、unsigned int、short、または unsigned short の場合は、乗数である正の整数 B の値に関係なく、実行速度の最適化が実行されます。

例 8.2

int 型の乗算

入力

```
int A, Res;

void main()
{
         Res = A * 73;         /* 73 ---> 1001001(2)  */
}
```

/Ot オプションを指定して上記の例をコンパイルすると、次のようなアセンブリコードが生成されます。

出力

```
;;    Res = A * 73;         /* 73 ---> 1001001(2)  */
l         er0,    NEAR _A
mov       er2,    er0
sllc      r1,     #03h
sll       r0,     #03h
add       er0,    er2
sllc      r1,     #03h
sll       r0,     #03h
add       er0,    er2
st        er0,    NEAR _Res
```

/Ot オプションを指定しないで上記の例をコンパイルすると、次のようなアセンブリコードが生成されます。

出力

```
;;    Res = A * 73;         /* 73 ---> 1001001(2)  */
l         er0,    NEAR _A
mov       r2,     #049h
mov       r3,     #00h
bl        __imulu8sw
st        er0,    NEAR _Res
```

被乗数 A が long 変数の場合、long 型の乗算に対しては実行速度の最適化は行われません。long 型の乗算を実行するには、__lmulu8sw エミュレーションルーチンが使用されます。

8.1.5.2 除算に対する実行速度の最適化

次の条件を満たす除算に対しては、実行速度の最適化が行われます。

除算 A/B において、A が符号付き変数で、B が 2^n という値の正の整数である場合。

A の符号を調べることで、除算はシフト演算に変換されます。

例 8.3

符号付きの char 型変数に関する除算

入力

```
signed char Res , A ;
void CharDivision ()
{
        Res = A / 8 ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;      Res = A / 8 ;
        l       r0,NEAR _A
        bps     _$M1
        add     r0,#07h
_$M1 :
        sra     r0,#03h
        st      r0,NEAR _Res
```

例 8.4

符号付きの int 型変数に関する除算

入力

```
signed int Res , A ;

void IntDivision ()
{
        Res = A /128  ;
```

```
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;       Res = A /128 ;
         l       er0,      NEAR _A
         bps     _$M1
         add     r0,#07fh
         addc    r1,#00h
_$M1 :
         srlc    r0,#07h
         sra     r0,#07h
         st      er0,      NEAR _Res
```

例 8.5

符号付きの long 型変数に関する除算

入力

```
signed long Res , A ;
void LongDivision ()
{
        Res = A / 512  ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;       Res = A / 512;
         l       er0,      NEAR _A
         l       er2,      NEAR _A+02h
         bps     _$M1
         add     r0,#0ffh
         addc    r1,#01h
         addc    r2,#00h
         addc    r3,#00h
_$M1:
         mov     r0,r1
         mov     r1,r2
         mov     r2,r3
         extbw   er2
```

```
        srlc    r0,#01h
        srlc    r1,#01h
        sra     r2,#01h
        st      er0,      NEAR _Res
        st      er2,      NEAR _Res+02h
```

8.1.5.3 シフト演算に対する実行速度の最適化

シフト演算に対する実行速度の最適化は、/Ot オプションが指定されている場合に行われます。この最適化は、符号付き/なしの char 型, short 型, int 型, long 型変数を含む右シフトおよび左シフト演算子に対し、適用されます。

例 8.6

符号付き char 型変数を含む左シフト演算

入力

```
signed char Res , A , B ;
void CharShift ()
{
        Res = A << B  ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;    Res = A << B ;
        l       r0,NEAR _A
        l       r1,NEAR _B
        cmp     r1,#07h
        bgt     _$M1
        sll     r0,r1
        b       _$M2
_$M1 :
        mov     r0,#00h
_$M2 :
        st      r0,NEAR _Res
```

例 8.7

符号付き int 型変数を含む右シフト演算

入力

```
signed int Res , A , B ;
void IntShift ()
{
        Res = A >> B  ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;    Res = A >> B ;
        l       er0,      NEAR _A
        l       er2,      NEAR _B
_$M2 :
        cmp     r2,       #07h
        cmpc    r3,       #00h
        ble     _$M1
        srlc    r0,       #07h
        sra     r1,       #07h
        add     er2,      #-7
        bne     _$M2
_$M1 :
        srlc    r0,       r2
        sra     r1,       r2
        st      er0,      NEAR_Res
```

例 8.8

符号なし long 型変数を含む左シフト演算

入力

```
unsigned long Res , A , B ;
void ULongShift ()
{
        Res = A << B  ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;    Res = A << B ;
         l       er0,      NEAR _A
         l       er2,      NEAR _A+02h
         l       er4,      NEAR _B
_$M2 :
         cmp     r4,       #07h
         cmpc    r5,       #00h
         ble     _$M1
         sllc    r3,       #07h
         sllc    r2,       #07h
         sllc    r1,       #07h
         sll     r0,       #07h
         add     er4,      #-7
         bne     _$M2
_$M1 :
         sllc    r3,       r4
         sllc    r2,       r4
         sllc    r1,       r4
         sll     r0,       r4
         st      er0,      NEAR _Res
         st      er2,      NEAR_Res+02h
```

ビットフィールドメンバ、およびシフト値に対するビットごとの AND 演算を含むシフト演算の場合には、生成されるコードでの最適化は、以下で説明する default オプションで実行されます。

シフト値が、3 ビット以下のビットフィールドオペランドである場合、最適化はデフォルトで行われます。

例 8.9

3 ビット以下のビットフィールドメンバを含む左シフト演算

入力

```
struct  sTag
{
         unsigned int  B : 3 ;
} sObj ;

unsigned int Res , A  ;

void BitFieldShift ()
{
         Res = A << sObj.B  ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;    Res = A << sObj.B  ;
         l       er0,      NEAR _sObj
         and     r0,       #07h
         l       er2,      NEAR _A
         sllc    r3,       r0
         sll     r2,       r0
         st      er2,      NEAR _Res
```

値が 7 以下のシフト値に対してビットごとの AND 演算を実行する場合には、最適化はデフォルトで行われます。

例 8.10

値が 7 以下の右オペランドに対するビットごとの AND 演算を含む左シフト演算

入力

```
unsigned int Res , A  , B ;

void BitwiseANDShift ()
{
         Res = A << ( B & 7 ) ;
}
```

上記の関数に対して生成されるアセンブリコードは、次のようになります。

出力

```
;;    Res = A << ( B & 7 ) ;
         l       er0,      NEAR _B
         and     r0,       #07h
         l       er2,      NEAR _A
         sllc    r3,       r0
         sll     r2,       r0
         st      er2,      NEAR_Res
```

8.2 スタックプローブの削除

プログラムの実行は、スタックプローブとして知られているスタックチェックルーチンの呼び出しを削除することで速度を上げることができます。スタックプローブは、プログラムが必要なローカル変数を割り当てるのに十分な領域があることを検査します。

スタックプローブを削除することによるマイナス面は、スタックのオーバーフローが検出されなくなることです。しかし、プログラムが利用できるスタック領域を超えないことがわかっている場合には、この方法は役に立ちます。

デフォルトでは、スタックプローブルーチンは呼び出されません。コマンドラインオプション /ST によって、CCU8 は、それぞれの関数の始めにスタックプローブルーチンを呼び出すようにします。

スタックチェックは、#pragma CHECKSTACK_ON または#pragma CHECKSTACK_OFF のいずれかを使用して局所的に制御できます。#pragma CHECKSTACK_OFF の後に記述されている関数に対してはスタックチェックが行われなくなります。また、#pragma CHECKSTACK_ON の後に記述されている関数に対しては行われるようになります。

8.3 変数の割り当ての制御

CCU8 のプラグマを使用して、変数の割り当てを制御することができます。これにより、CCU8 はさまざまなアドレッシングモードを使用して、アセンブリ出力を改善することができます。

#pragma ABSOLUTE を使用すると、変数をデータメモリの任意の位置に割り当てることができます。NVDATA、ROMWIN などの他のプラグマを使用すると、変数をメモリのどの部分に割り当てるかを指定することができます。

#pragma SEGNOINT を使用すると、初期化されていないグローバル変数、静的グローバル変数、および静的ローカル変数に対して、セグメント名またはセグメント開始アドレスを指定できます

#pragma SEGNVDATA を使用すると、グローバル変数、静的グローバル変数、および静的ローカル変数に対して、セグメント名またはセグメント開始アドレスを指定できます。変数は、不揮発性メモリ領域に配置されます。

#pragma SEGINT を使用すると、初期化されたグローバル変数、静的グローバル変数、および静的ローカル変数に対して、セグメント名またはセグメント開始アドレスを指定できます。

#pragma SEGCONST を使用すると、グローバル変数、静的グローバル変数、および静的ローカル変数宣言 const に対して、セグメント名またはセグメント開始アドレスを指定できます。変数は、リードオンリーメモリ領域に配置されます。

segnoinit、segnvdata、seginit の各プラグマで指定されるセグメントに配置される変数は、absolute プラグマまたは nvdata プラグマで指定することはできません。

segconst プラグマで指定されるセグメントに配置される変数は、absolute プラグマで指定することはできません。

8.4 ミックスドランゲージプログラミング

ここでは、CCU8 を使ってコンパイルされる C プログラムおよび関数で U8 アセンブリ言語ルーチンを使用する方法を説明します。特に、C 言語プログラムからアセンブリ言語ルーチンを呼び出す方法、およびアセンブリ言語ルーチンから C 言語関数を呼び出す方法について説明します。

8.4.1 レジスタの内容の保持

CCU8 はレジスタの内容を、次のように保持します。

通常の関数およびソフトウェア割り込み関数は、レジスタ R0～R3 の保存を行わずに自由に使用できます。共用あるいはソフトウェア割り込み関数がレジスタ R4～R15 (BP と FP を含む)を使用する場合は、関数のプロローグ処理でレジスタを保存し、関数のエピローグ処理でそのレジスタを復元します。関数呼び出しの間に、レジスタ R4～R15 の内容は保持され、レジスタ R0～R3 の内容は破壊されます。ソフトウェア割り込み関数に戻り値と引数がない場合は、レジスタ R0～R3 の内容も保持されます。

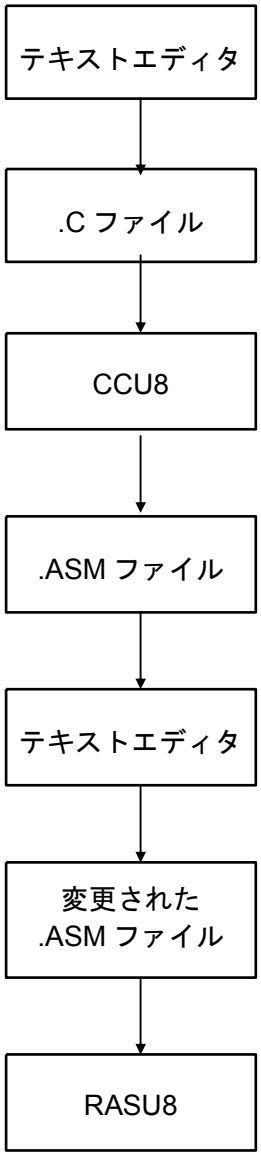
割り込み関数は、関数で使用されるすべてのレジスタを関数のプロローグ処理で保存し、関数のエピローグ処理で対応するレジスタを復元します。したがって、複数の言語が混在するプログラミングでは、前記のプロトコルに従う必要があります。

8.4.2 アセンブリ言語と C プログラムの結合

アセンブリ言語ルーチンと C 言語プログラムを結合する方法はいくつかあります。

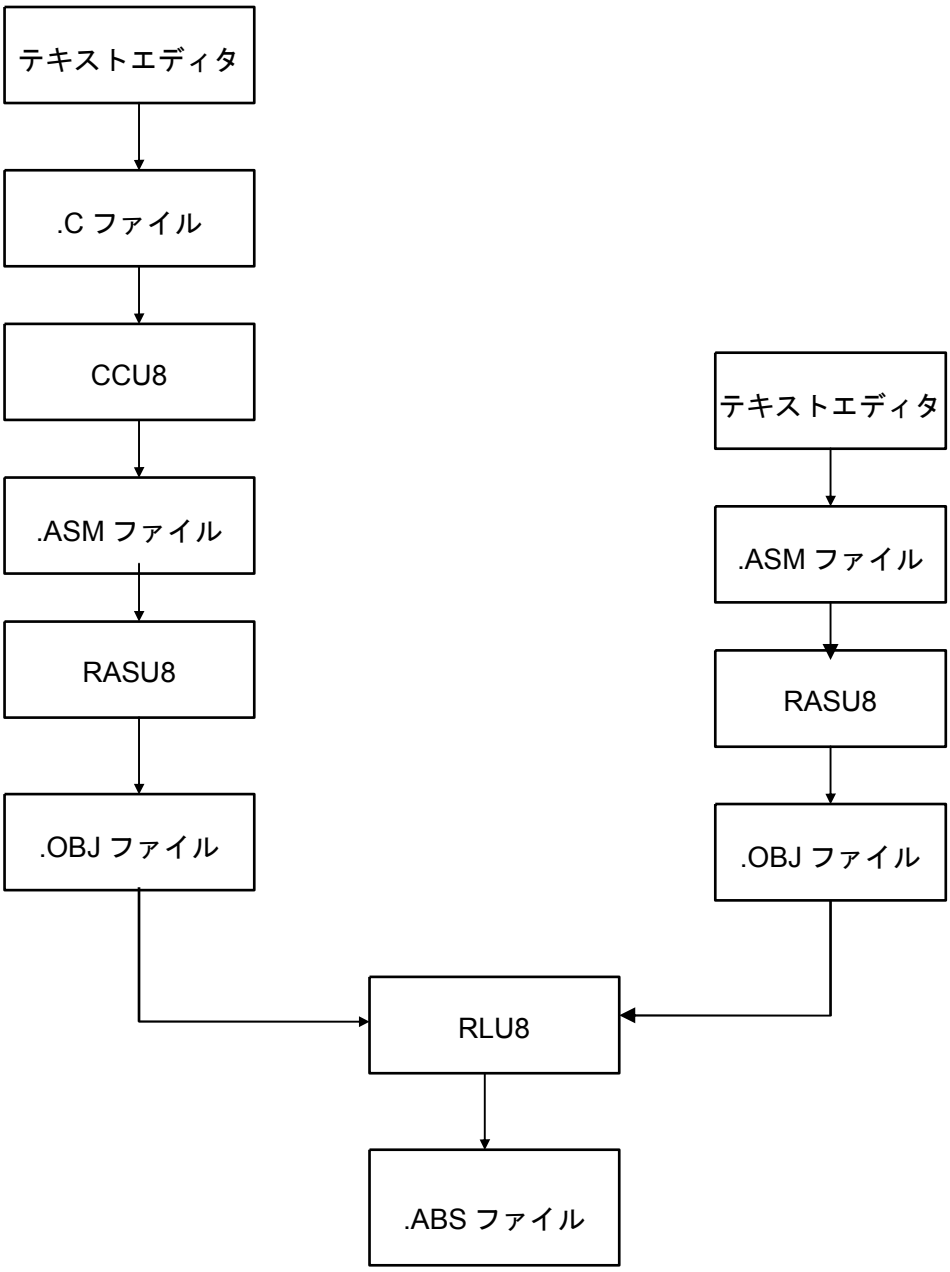
方法 1

この方法では、プログラマは C プログラムを記述し、CCU8 を使用して C プログラムをコンパイルします。CCU8 が生成する出力は、CCU8 ニーモニックのアセンブリ言語ファイルです。プログラマは、テキストエディタを使用してファイルを編集したり、必要なアセンブリ言語ルーチンを追加できます。できあがったファイルを RASU8 を使用してアセンブルし、RLU8 を使用してリンクしてアブソリュートファイルを生成します。



方法 2

この方法では、プログラマは C プログラムを記述し、CCU8 を使用して C プログラムをコンパイルします。コンパイラは、.ASM ファイルを出力として生成します。プログラマは、C プログラムと組み合わせるためのアセンブリルーチンを含むアセンブリ言語ファイルを作成します。2 つのアセンブリプログラムファイルをそれぞれ RASU8 を使用してアセンブルします。結果として、2 つの.OBJ ファイルができます。この 2 つの.OBJ ファイルをリンカ RLU8 を使用してリンクします。



方法 3

#asm および #endasm を使用する

この方法では、プログラマは**前処理指令#asm** および**#endasm** を使用してソースファイルに直接アセンブリ命令を記述します。手続き、または手続きの一部を、アセンブリ言語で記述し、2 つの前処理指令**#asm** および**#endasm** で挟みます。CCU8 は、2 つの前処理指令の間に挟まれたものをそのまま、出力ファイルに置きます。ローカル変数はレジスタに割り当てられるので、asm ブロック内のローカル変数へのアクセスは、意図された結果が得られないかもしれません。したがって、asm ブロック間の受け渡しデータは、グローバル変数経由でなければなりません。

方法 4

#pragma asm および#pragma endasm を使用する

この方法では、プログラマはプラグマ**#pragma asm** およびプラグマ**#pragma endasm** を使用してソースファイルに直接アセンブリ命令を記述します。プラグマに挟まれているテキストの処理は、前処理指令**#asm** および前処理指令**#endasm** の場合と同じです。

方法 5

__asm キーワードを使用する

構文:

__asm (string)

この方法では、プログラマは**__asm** キーワードを使用してソースファイルに直接アセンブリ命令を記述します。手続きまたは手続きの一部を、**__asm** キーワードの文字列引数として指定できます。CCU8 は、このキーワードの引数をそのまま、出力ファイルに置きます。

__asm キーワードの戻り値は使用できません。

次の場合に、CCU8 はエラーメッセージを表示します。

- 指定された引数が文字列でない場合
- 複数の引数が指定された場合
- **__asm** キーワードの戻り値を使用している場合

次の例は誤っている例です。

例 8.11

入力

```
void fn ()
{
        __asm (“DI¥n”, “EI¥n” ) ;
}
```

上のプログラムでは、**__asm** キーワードに複数の引数を指定しているので、CCU8 はエラーメッセージを出力します。

例 8.12

入力

```
void fn ()
{
        return __asm (“DI¥n”, “EI¥n” ) ;
}
```

上のプログラムでは、**__asm** キーワードの戻り値が使用されているので、CCU8 はエラーメッセージを出力します。

8.4.3 CCU8 の呼び出し規約

CCU8 では一定の規約にしたがって、C 言語の関数に値を渡したり、C 言語の関数呼び出しから値を受け取ります。アセンブリ言語ルーチンも、この規約に従わなければなりません。CCU8 では、任意の関数に引数を渡すのに、それを R0、R1、R2 および R3 レジスタに割り当てます。レジスタに割り当てられる引数の型には、char 型、unsigned char 型、short 型、 unsigned short 型、int 型、 unsigned int 型、long 型、 unsigned long 型、float 型および pointer 型があります。それ以外の型のものやレジスタが足りなくてレジスタに割り当てられない時には、引数をスタックに割り当てます。

ソフトウェア割り込み関数の動作は、通常の関数呼び出しと同じです。

引数のレジスタ割り当ての規則を以下に示します。

- 左から右の順で引数をレジスタ R0 から R3 に割り当てる。
- 1 バイトの引数は、Rn(n は 0、1、2 または 3)に割り当てる。
- 2 バイトの引数は、下位バイトを Rn に、上位バイトを Rn+1 に割り当てる(n は 0 または 2)。
- 4 バイトの引数は、最下位バイトを R0 に、次の下位バイトを R1 に、3 番目のバイトを R2 に、最上位バイトを R3 に割り当てる。

- near ポインタ引数は、オフセットの下位バイトを Rn に、オフセットの上位バイトを Rn+1 に割り当てる(n は 0 または 2)。
- far ポインタ引数は、オフセットの下位バイトを R0 に、オフセットの上位バイトを R1 に、セグメントを R2 に割り当てる。
- 引数に割り当てられた R0、R1、R2 および R3 レジスタは、必要に応じてプロローグ処理で R8、R9、R10 および R11 レジスタにコピーして保存します。したがって、R8、R9、R10 および R11 レジスタは、プロローグ処理でスタックにプッシュされ、関数のコードの中で使用されればエピローグ処理でスタックからポップされます。

例 8.13

入力

```
char g_var;
void function(char c)
{
        g_var = c;
}
```

出力

```
_function       :

;;{

;;   g_var = c;
        st      r0,NEAR _g_var

;;}
        rt
```

例 8.14

入力

```
# pragma SWI function 0x80
int c;
void
function(int a, int b)
{
        c = a + b ;
}
```

出力

```
_function       :
;;{
        push    elr, epsw
;;   c = a + b ;
        add     er0,      er2
        st      er0,      NEAR _c
;;}
        pop     psw, pc
```

引数のスタック割り当ての規則を以下に示します。

- 各引数の値は右から左へスタックにプッシュする。
- 引数が式の場合、CCU8 はスタックにプッシュする前に式の値を計算する。式の評価は左から右へ行うが、引数はスタックに右から左へプッシュされる。
- スタックにプッシュされるバイト数は、引数のサイズと同じです。
- 関数が制御を戻したら、呼び出し側でスタックから引数を削除する必要がある。引数としてプッシュしたバイト数を SP に加えればよい。

例 8.15

入力

```
int var1 ;
char __far * var2 ;
char var3 ;

void
fn1 ( int a, char __far * b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

出力

```
_fn1    :
        push    fp
        mov     fp,     sp
        push    er10
        mov     r10,    r2

;;      var1 = a ;
        st      er0,    NEAR _var1

;;      var2 = b ;
        l       er0,    2[fp]
        l       r2,     4[fp]
        st      er0,    NEAR _var2
        st      r2,     NEAR _var2+02h

;;      var3 = c ;
        st      r10,    NEAR _var3

;;}
        pop     er10
        mov     sp,     fp
        pop     fp
        rt
```

例 8.16

入力

```
char var1 ;
long var2 ;
int var3 ;

void
fn2 ( char a, long b, int c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

出力

```
_fn2    :
        push    fp
        mov     fp,     sp
        push    er10
        mov     er10,   er2

;;      var1 = a ;
        st      r0,     NEAR _var1

;;      var2 = b ;
        l       er0,    2[fp]
        l       er2,    4[fp]
        st      er0,    NEAR _var2
        st      er2,    NEAR _var2+02h

;;      var3 = c ;
        st      er10,   NEAR _var3

;;}
        pop     er10
        mov     sp,     fp
        pop     fp
        rt
```

例 8.17

入力

```
char __far * var1 ;
int var2 ;
char var3 ;

void
fn3 ( char __far * a, int b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

出力

```
__fn3   :
        push    fp
        mov     fp,     sp

;;      var1 = a ;
        st      er0,    NEAR _var1
        st      r2,     NEAR _var1+02h

;;      var2 = b ;
        l       er0,    2[fp]
        st      er0,    NEAR _var2

;;      var3 = c ;
        st      r3,     NEAR _var3

;;}
        mov     sp,     fp
        pop     fp
        rt
```

例 8.18

入力

```
char var1 ;
int var2 ;
char var3 ;

void
fn4 ( char a, int b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

出力

```
__fn4   :
        push    fp
        mov     fp,     sp

;;      var1 = a ;
        st      r0,     NEAR _var1

;;      var2 = b ;
        st      er2,    NEAR _var2

;;      var3 = c ;
        l       r0,     2[fp]
        st      r0,     NEAR _var3

;;}
        mov     sp,     fp
        pop     fp
        rt
```

例 8.19

入力

```
double var1 ;
int var2 ;
char var3 ;

void
fn5 ( double a, int b, char c )
{
        var1 = a ;
        var2 = b ;
        var3 = c ;
}
```

出力

```
_fn5    :
        push    fp
        mov     fp,     sp
        push    xr4
        push    er8
        mov     er8,    er0

;;      var1 = a ;
        lea     2[fp]
        l       qr0,    [ea]
        lea     OFFSET _var1
        st      qr0,    [ea]

;;      var2 = b ;
        st      er8,    NEAR _var2

;;      var3 = c ;
        st      r2,     NEAR _var3

;;}
        pop     er8
        pop     xr4
        mov     sp,     fp
        pop     fp
        rt
```

例 8.20

入力

```
# pragma SWI function 0x80
void
function(long a, long b, int c)
{
         function2(a, b, c);
}
```

出力

```
_function       :

;;{
         push    elr, epsw, lr, ea
         push    fp
         mov     fp,       sp
         push    xr8
         l       r8,       DSR
         push    r8
         mov     er8,      er0
         mov     er10,     er2

;;    function2(a, b, c);
         l       er0,      14[fp]
         push    er0
         l       er0,      10[fp]
         l       er2,      12[fp]
         push    xr0
         mov     er0,      er8
         mov     er2,      er10
         bl      _function2
         add     sp,       #6

;;}
         pop     r8
         st      r8,       DSR
         pop     xr8
         mov     sp,       fp
         pop     fp
         pop     ea, lr, psw, pc
```

8.4.4 戻り値

C 言語プログラムに値を戻したり、C 言語関数から戻り値を受け取るには、アセンブリ言語ルーチンは CCU8 の戻り値規約に従わなければなりません。

関数の戻り値が 4 バイト以下の場合には、CCU8 は関数の戻り値を R0、R1、R2、R3 レジスタ(使用するレジスタ数は戻り値のサイズによります) に割り当てます。戻り値が構造体か共用体か double 型の場合には、CCU8 は戻り値が代入される変数のアドレスを最初の引数に割り当てます(つまり、関数が__noreg 修飾子で修飾されていない場合、ER0 にアドレスが入ります)。したがって、戻り値は呼び出された関数が書き換えます。

ソフトウェア割り込み関数の動作は、通常の関数呼び出しと同じです。

戻り値がレジスタに割り当てられている場合の規則を以下に示します。

- 戻り値は、R0 から R3 のレジスタに割り当てられる。
- 1 バイトの戻り値は、R0 レジスタに割り当てられる。
- 2 バイトの戻り値は、下位バイトを R0 に、上位バイトを R1 に割り当てる。
- 4 バイトの戻り値は、最下位バイトを R0 に、次の下位バイトを R1 に、3 番目のバイトを R2 に、最上位バイトを R3 に割り当てる。
- near ポインタ型の戻り値は、下位オフセットのバイトを R1 に、上位オフセットのバイトを R1 に割り当てる。
- far ポインタ型の戻り値では、低位オフセットのバイトを R0 に、高位オフセットのバイトを R1 に、セグメントを R2 に割り当てる。

例 8.21

入力

```
int add_int (int a, int b)
{
         return ( a + b ) ;
}
long add_long (long a, long b)
{
         return ( a + b ) ;
}
double add_double ( double a, double b)
{
         return ( a + b ) ;
}
```

出力

```
_add_int :

;;       return ( a + b ) ;
         add     er0,      er2

;;}
         rt


_add_long        :
         push    fp
         mov     fp,       sp
         push    xr4

;;       return ( a + b ) ;
         l       er4,      2[fp]
         l       er6,      4[fp]
         add     er0,      er4
         addc    r2,       r6
         addc    r3,       r7

;;}
         pop     xr4
         mov     sp,       fp
         pop     fp
         rt

_add_double      :

         push    lr
         push    fp
         mov     fp,       sp
         push    xr4
         push    er8
         mov     er8,      er0

;;       return ( a + b ) ;
         lea     12[fp]
         l       qr0,      [ea]
         push    qr0
         lea     4[fp]
```

```
            l       qr0,      [ea]
            push    qr0
            bl      __daddu8sw
            add     sp,       #8
            pop     qr0
            lea     [er8]
            st      qr0,      [ea]

;;}
            pop     er8
            pop     xr4
            mov     sp,       fp
            pop     fp
            pop     pc
```

例 8.22

入力

```
# pragma SWI function 0x80 1

int a;

int function()
{
    return a ;
}
```

出力

```
_function       :

;;{

;;    return a ;
            l       er0,      NEAR _a

;;}
            rti
```

8.4.5 アセンブリ言語の割り込み処理ルーチン

CCU8 では、割り込み処理ルーチンを C 言語で記述することができます。割り込み処理ルーチンは CODE メモリの物理セグメント 0 に置かなければなりません。対応する割り込みベクタを、ルーチンの開始アドレスで初期化しなければなりません。割り込み処理ルーチンの最後のステートメントは rti 命令でなければなりません。

8.4.6 C 言語変数の参照

アセンブリルーチンは、C ソースプログラムで使用するグローバル変数を参照できます。初期化されたグローバル変数は、アセンブリルーチンで、“EXTRN”として宣言すれば参照できます。このような変数は、アセンブリで“PUBLIC”として宣言しないでください。初期化されていないグローバル変数は、“PUBLIC”または“EXTRN”または“COMM”の擬似命令を使用して宣言すれば参照できます。C プログラムで“**extern**”として宣言されたグローバル変数は、“PUBLIC”または“COMM”として宣言すれば、アセンブリルーチンで参照できます。

8.5 ‘__noreg’による関数の修飾

関数を**__noreg** で修飾すると、引数にスタックを使用するようにコンパイラに指示できます。

関数のプロトタイプまたは定義で__noreg 関数指定子が指定されていると、CCU8 は、引数をレジスタではなくスタックで渡します。デフォルトでは、__noreg 修飾子は、main 関数および可変引数関数に適用できます。

Swi 関数の動作は、通常の関数呼び出しと同じです。

例 8.23

入力

```
int __noreg add ( int a, int b );
int var1, var2 ;
int __noreg add ( int a, int b )
{
        int     l_ret     ;
        l_ret = a + b ;
        return ( l_ret ) ;
}
fn ()
{
        var1 = add ( var1, var2 ) ;
}
```

出力

```
_add    :
;;{
        push    fp
        mov     fp,       sp

;;      return ( l_ret ) ;
        l       er0,      2[fp]
        l       er2,      4[fp]
        add     er0,      er2

;;}
        mov     sp,       fp
```

```
        pop     fp
        rt


_fn     :
;;{
        push    lr

;;      var1 = add ( var1, var2 ) ;
        l       er0,      NEAR _var2
        push    er0
        l       er0,      NEAR _var1
        push    er0
        bl      _add
        add     sp,       #04h
        st      er0,      NEAR _var1

;;}
        pop     pc
```

例 8.24

入力

```
# pragma SWI add 0x80
int __noreg add ( int a, int b );
int __noreg add ( int a, int b )
{
        int l_ret   ;
        l_ret = a + b ;
        return ( l_ret ) ;
}
```

出力

```
_add    :

;;{
        push    elr, epsw
        push    fp
        mov     fp, sp

;;    return ( l_ret ) ;
        l       er0,      6[fp]
        l       er2,      8[fp]
```

```
            add     er0,      er2

;;}
            mov     sp, fp
            pop     fp
            pop     psw, pc
```

8.6 組み込み関数

8.6.1 __EI() および __DI()

CCU8 は、組み込み関数 **__EI()**および**__DI()**をサポートしています。組み込み関数を呼び出すと、組み込み関数の本体がアセンブリリストファイルにインライン展開されます。

これらの関数名は、予約済みのキーワードです。組み込み関数がソースファイル内で定義されている場合、CCU8 はエラーメッセージを出力します。

これらの組み込み関数に引数が指定された場合、CCU8 はエラーメッセージを出力します。

これらの組み込み関数のプロトタイプは、次のとおりです。

void __EI(void);

void __DI(void);

組み込み関数"**__EI()**"はアセンブリコード **EI** に展開され、組み込み関数"**__DI()**"はアセンブリコード **DI** に展開されます。

例 8.25

入力:

```
static void intr () ;
# pragma interrupt intr 0xA
/* or
# pragma interrupt intr 0xA 2
*/

static void
intr ()
{
        __EI () ;
}
```

上記のプログラムで定義されている”intr”関数に対して生成されるコードは、次のとおりです。

出力

```
_intr    :
         push    elr, epsw

;;       __EI () ;
         ei

;;}
         pop     psw, pc
```

例 8.26

入力

```
         int __EI (int arg1) ;
```

上記の例では、組み込み関数“**__EI**”のプロトタイプが再定義されているため、CCU8 はエラーを出力します。

8.6.2 __segbase_n()

この組み込み関数は、セグメントの開始アドレスを取得するために使用されます。関数は __near ポインタ型を返します。

この組み込み関数のプロトタイプは次のとおりです。

void __near * __segbase_n("segment_name");

組み込み関数__segbase_n に対するパラメータとしては、有効なセグメント名を指定します。関数は、パラメータとして指定されたセグメントの開始アドレスを返します。

__segbase_n()関数に渡すセグメント名は、デフォルトのセグメントの名前、コンパイラによって自動的に生成される名前、または segcode、segintr、seginit、segnoinit、segconst、segnvdata の各プラグマを使用してユーザーが定義した名前です。

__segbase_n 関数は、ローカルとグローバルの両方の有効範囲で使用できます。

ローカル有効範囲では、関数はアセンブラの OFFSET オペランドを使用するアセンブラ命令に展開されます。

次に示す例は、__segbase_n をローカルな有効範囲で使用した場合です。

例 8.27

入力

```
#pragma segnoinit __near "SEGNAME"
int var;
void Function()
{
         int __near * np = __segbase_n("SEGNAME") ;
}
```

上記の例に対して、CCU8 は次のような出力を生成します。

出力

```
mov r0, #BYTE1 OFFSET SEGNAME     ;; Lower byte of near pointer is
                                  ;; accessed
mov r1, #BYTE2 OFFSET SEGNAME     ;; Higher byte of near pointer is
                                  ;; accessed
```

```
.....................
.....................
          rseg SEGNAME
_var :
          ds 02h
```

グローバルな有効範囲では、関数はグローバル変数を初期化するために使用され、この関数によって返される値を格納するためのテーブルセグメントが生成されます。

例 8.28

入力

```
#pragma segnoinit __near "SEGRAM1"
int var;
unsigned int __near * nbase = __segbase_n( "SEGRAM1" ) ;
```

上記の例に対して、CCU8 は次のような出力を生成します。

出力

```
rseg $$NINITTAB            ;; Table segment is generated
dw  OFFSET  SEGRAM1        ;; OFFSET portion of SEGRAM1 of type
                           ;; address is stored

  rseg $$NINITVAR
_nbase :
  ds  02h                  ;; Memory equivalent to size of pointer
                           ;; variable is reserved

rseg SEGRAM1

var :
  ds 02h                   ;; Memory equivalent to size of integer
                           ;; variable is reserved
```

8.6.3 __segbase_f()

この組み込み関数は、セグメントの開始アドレスを取得するために使用されます。関数は__far ポインタ型を返します。

この組み込み関数のプロトタイプは次のとおりです。

void __far * __segbase_f("segment_name");

組み込み関数__segbase_f に対するパラメータとしては、有効なセグメント名を指定します。関数は、パラメータとして指定されたセグメントの開始アドレスを返します。

__segbase_f()関数に渡すセグメント名は、デフォルトのセグメントの名前、コンパイラによって自動的に生成される名前、または segcode、segintr、seginit、segnoinit、segconst、segnvdata の各プラグマを使用してユーザーが定義した名前です。

__segbase_f 関数は、ローカルとグローバルの両方の有効範囲で使用できます。

ローカル有効範囲では、関数はアセンブラの SEG オペランドと OFFSET オペランドを使用するアセンブラ命令に展開されます。

例 8.29

入力

```
#pragma segnoinit __far "SEGNAME1"
int siVar;
void fn(void)
{
         int __far * fp = __segbase_f("SEGNAME1") ;
}
```

上記の例に対して、CCU8 は次のような出力を生成します。

出力

```
mov r0, #BYTE1 OFFSET SEGNAME1    ;; Lower byte of near pointer is
                                  ;; accessed
mov r1, #BYTE2 OFFSET SEGNAME1    ;; Higher byte of near pointer is
                                  ;; accessed
mov r2, #SEG SEGNAME1             ;; Physical segment address portion
                                  ;; of SEGNAME1 is accessed
...........
...........
...........
  rseg SEGNAME1
_siVar :
  ds 02h
```

グローバルな有効範囲では、関数はグローバル変数を初期化するために使用され、この関数によって返される値を格納するためのテーブルセグメントが生成されます。

例 8.30

入力

```
#pragma segnoinit __far "SEGNAME1"
int var;
unsigned int __far  * fbase = __segbase_f( "SEGNAME1" ) ;
```

上記の例に対して、CCU8 は次のような出力を生成します。

出力

```
  ..............
  ..............
  public _fbase
  ..............
  ..............

rseg $$NINITTAB
  dw  OFFSET  SEGNAME1     ;; OFFSET portion of SEGRAM1 of type
                           ;; address is stored
  db  SEG SEGNAME1         ;; Physical segment address portion of
                           ;; SEGNAME1 is accessed

rseg $$NINITVAR
_fbase :
  ds  03h                  ;; Memory for far pointer variable is
                           ;; reserved

rseg SEGNAME1
var :
  ds 02h                   ;; Memory for integer type variable is
                           ;; reserved
```

8.6.4 __segsize()

この組み込み関数は、セグメントのサイズを取得するために使用されます。関数が返すサイズは unsigned int 型です。

この組み込み関数のプロトタイプは次のとおりです。

unsigned int __segsize("segment_name");

組み込み関数__segsize に対するパラメータとしては、有効なセグメント名を指定します。関数は、パラメータで指定されたセグメントのサイズを返します。関数は、SIZE オペランドを使用するアセンブラ命令に展開されます。

この関数は、ローカルとグローバルの両方の有効範囲で使用できます。

次に示す例は、この関数をローカルな有効範囲で使用した場合です。

例 8.31

入力

```
#pragma segnoinit "SEGRAM"
int sivar1;
void fn(void)
{
         unsigned int uisize = __segsize("SEGRAM") ;
}
```

上記の例に対して/near オプションを指定すると、CCU8 は次のような出力を生成します。

出力

```
.................
.................
l er0,  $$S0      ;; Size of segment is loaded in er0 register
.................
.................
rseg $$NTABsize1  ;; Table segment for near data access type
$$S0 :
DW  SIZE  SEGRAM  ;; Size of segment is stored
.................
.................
  rseg SEGRAM
_sivar1 :
  ds 02h
.................
.................
```

上記の例に対して/far オプションを指定すると、CCU8 は次のような出力を生成します。

出力

```
.................
.................
l er0,  $$S0      ;; Size of segment is loaded in er0 register
.................
```

```
.................
rseg $$FTABsize1   ;; Table segment for far data access type
$$S0 :
DW  SIZE  SEGRAM
.................
.................
  rseg SEGRAM
_sivar1 :
  ds 02h
```

次に示す例は、この関数をグローバルな有効範囲で使用した場合です。

例 8.32

入力

```
#pragma segdef "SEGRAM1" "DATA"
unsigned int size = __segsize( "SEGRAM1" ) ;
```

上記の例に対し、CCU8 は次のような出力を生成します。

出力

```
.................
.................

SEGRAM1 segment data any      ;; Code generated for Segdef pragma
                              ;; Refer Segdef pragma specification for
.................             ;; more details
.................
public _size
.................
.................

rseg $$NINITTAB
  dw  SIZE  SEGRAM1

rseg $$NINITVAR
_size :
  ds  02h
.................
.................
```

8.7 スタートアップルーチン

スタートアップルーチン“$$start_up”は、スタックと SFR 初期化を含むアセンブリ言語ルーチンです。スタートアップルーチンからメイン関数への制御の渡しは、ジャンプ命令によって行われます。

```
b    _main
```

スタートアップルーチンは、スタートアップアセンブリソースファイルの形で別に提供されます。ユーザーは、このファイルを変更して初期化処理を追加します。スタートアップオブジェクトファイルは、RLU8 を起動する際に直接指定します。

9. エミュレーションライブラリ

U8 のアーキテクチャでは **long**、**float** および **double** のデータ型をサポートしていませんが、CCU8 ではこれらのデータ型をサポートします。これらのデータ型は、float と long のエミュレーションルーチンを使用してサポートしています。このルーチンは、3 つのライブラリ longu8.lib、floatu8.lib、doubleu8.lib で提供されています。**long**、**float** および **double** のデータ型を使用した算術演算は、すべて、これらのルーチンを使用して行われます。CCU8 は、算術演算を行うために、対応するルーチンの呼び出し命令を出力します。これらのルーチンは、Small メモリモデルおよび Large メモリモデルの両方のメモリモデルに提供しています。

エミュレーションライブラリには、以下のファイルがあります。

ライブラリファイル名	内容
LONGU8.LIB	整数演算用エミュレーションライブラリ
DOUBLEU8.LIB	倍精度浮動小数点演算用エミュレーションライブラリ
FLOATU8.LIB	単精度浮動小数点演算用エミュレーションライブラリ （高速エミュレーションライブラリ）

エミュレーションライブラリに用意しているルーチンと、そのスタック消費量を以下の表に示します。

LONGU8.LIB

ルーチン	機能	スタック消費量	
		スモール（*xx*=sw）	ラージ（*xx*=lw）
__cdivu8*xx*	符号つき char 型（8 ビット）同士の除算	4	6
__cmodu8*xx*	符号つき char 型（8 ビット）同士の剰余算	4	6
__cmulu8*xx*	符号つき char 型（8 ビット）同士の乗算	2	2
__idivu8*xx*	符号つき int 型(16 ビット)同士の除算	14	18
__imodu8*xx*	符号つき int 型（16 ビット）同士の剰余算	14	18
__uidivu8*xx*	符号なし int 型（16 ビット）同士の除算	10	12
__uimodu8xx	符号なし int 型（16 ビット）同士の剰余算	10	12
__imulu8*xx*	符号つき/符号なし int 型（16 ビット）同士の乗算	6	6
__ldivu8*xx*	符号つき long 型（32 ビット）同士の除算	20	24
__lmodu8*xx*	符号つき long 型（32 ビット）同士の剰余算	20	24
__uldivu8*xx*	符号なし long 型（32 ビット）同士の除算	18	20
__ulmodu8*xx*	符号なし long 型（32 ビット）同士の剰余算	18	20
__lmulu8*xx*	符号つき/符号なし long 型（32 ビット）同士の乗算	22	24
__indru8*xx*	関数の間接コール（ラージモデルのみ）	-	0
__chstu8*xx*	スタックオーバーフローのチェック	4	4
__regpushu8*xx*	関数入口でのレジスタ退避	12	12
__regpopu8*xx*	関数出口でのレジスタ復帰	0	0

DOUBLEU8.LIB

ルーチン	機能	スタック消費量	
		スモール（*xx*=sw）	ラージ（*xx*=lw）
__faddu8*xx*	float(32 ビット)型同士の加算	48	52
__fsubu8*xx*	float(32 ビット)型同士の減算	48	52
__fmulu8*xx*	float(32 ビット)型同士の乗算	48	52
__fdivu8*xx*	float(32 ビット)型同士の除算	66	72
__fcmpu8*xx*	float(32 ビット)型同士の比較	46	50
__fildu8*xx*	符号つき long 型から float 型への変換	36	40
__fuldu8*xx*	符号なし long 型から float 型への変換	36	40
__ftodu8*xx*	float 型から double 型への変換	36	40
__ftolu8*xx*	float 型から long 型への変換	36	40
__flnotu8*xx*	float(32 ビット)型の論理否定	6	6
__fnegu8*xx*	float(32 ビット)型の符号反転	4	4
__daddu8*xx*	double(64 ビット)型同士の加算（演算は double 型で行います)	48	52
__dsubu8*xx*	double(64 ビット)型同士の減算（演算は double 型で行います)	48	52
__dmulu8*xx*	double(64 ビット)型同士の乗算（演算は double 型で行います)	48	52
__ddivu8*xx*	double(64 ビット)型同士の除算（演算は double 型で行います)	66	72
__dcmpu8*xx*	double(64 ビット)型同士の比較（比較は double 型で行います)	46	50
__dildu8*xx*	符号つき long 型から double 型への変換	36	40
__duldu8*xx*	符号なし long 型から double 型への変換	36	40
__dtofu8*xx*	double 型から float 型への変換	36	40
__dtolu8*xx*	double 型から long 型への変換	36	40
__dlnotu8*xx*	double(64 ビット)型の論理否定	6	6
__dnegu8*xx*	double(64 ビット)型の符号反転	4	4

FLOATU8.LIB

ルーチン	機能	スタック消費量	
		スモール（*xx*=sw）	ラージ（*xx*=lw）
__faddu8*xx*	float(32 ビット)型同士の加算	40	44
__fsubu8*xx*	float(32 ビット)型同士の減算	40	44
__fmulu8*xx*	float(32 ビット)型同士の乗算	40	44
__fdivu8*xx*	float(32 ビット)型同士の除算	48	52
__fcmpu8*xx*	float(32 ビット)型同士の比較	38	42
__fildu8*xx*	符号つき long 型から float 型への変換	32	36
__fuldu8*xx*	符号なし long 型から float 型への変換	32	36
__ftodu8*xx*	float 型から double 型への変換	32	36
__ftolu8*xx*	float 型から long 型への変換	32	36
__flnotu8*xx*	float(32 ビット)型の論理否定	6	6
__fnegu8*xx*	float(32 ビット)型の符号反転	4	4
__daddu8*xx*	double(64 ビット)型同士の加算（演算は float 型で行います）	40	44
__dsubu8*xx*	double(64 ビット)型同士の減算（演算は float 型で行います）	40	44
__dmulu8*xx*	double(64 ビット)型同士の乗算（演算は float 型で行います）	40	44
__ddivu8*xx*	double(64 ビット)型同士の除算（演算は float 型で行います）	48	52
__dcmpu8*xx*	double(64 ビット)型同士の比較（比較は float 型で行います）	38	42
__dildu8*xx*	符号つき long 型から double 型への変換	32	36
__duldu8*xx*	符号なし long 型から double 型への変換	32	36
__dtofu8*xx*	double 型から float 型への変換	32	36
__dtolu8*xx*	double 型から long 型への変換	32	36
__dlnotu8*xx*	double(64 ビット)型の論理否定	6	6
__dnegu8*xx*	double(64 ビット)型の符号反転	4	4

10. コンパイラ出力のアセンブルおよびリンク

CCU8 は、アセンブリファイルを出力として作成します。オブジェクトファイルの作成は、コンパイラからの出力をリロケータブルアセンブラ RASU8 を使用してアセンブルして行われます。アセンブラを呼び出すには、次のコマンドラインを使用してください。

```
C:> RASU8 FILE <CR>
```

ここで FILE には、コンパイラ CCU8 が作成した出力ファイルの名前を指定します。複数のファイルをコンパイルしたときには、出力ファイルを別々にアセンブルします。

C 言語では、大文字と小文字は区別されるため、CCU8 は大文字/小文字を区別したコードを生成します。デフォルトで、RASU8 は大文字/小文字の区別をします。

アセンブラは、オブジェクトファイルを出力として生成します。C ソースレベルデバッガを使用して C 言語プログラムをデバッグするときは、次のように/CC オプションを使用してコンパイラ出力をアセンブルする必要があります。

```
C:> RASU8 FILE /CC <CR>
```

このオプションは、必要なデバッグ情報を付加したオブジェクトファイルを作成することをアセンブラに通知します。このオプションは、 CCU8 で/SD オプションを使用してファイルをコンパイルした場合のみ、RASU8 に対して指定しなければなりません。

RASU8 が作成したオブジェクトファイルは、オブジェクトリンカ RLU8 を使用してリンクできます。リンカは、出力としてアブソリュートオブジェクトファイルを生成します。

オブジェクトプログラムをリンクするには、次のコマンドラインを使用します。

```
C:> RLU8 FILE1 FILE2,.....,/CC <CR>
```

FILE1 FILE2,...は、RLU8 への入力オブジェクトファイルの名前です。/CC オプションは、入力は CCU8 がコンパイルし、RASU8 を使用してアセンブルしたファイルであることを RLU8 に通知します。それによって、RLU8 は適切な手順でスタックに空間を確保し、スタックポインタを初期化します。

CCU8 が作成した出力アセンブリファイルは、ライブラリ doubleu8.lib、floatu8.lib および longu8.lib に用意されているルーチンを利用します。RLU8 は、これらの 3 つのライブラリファイルを検索して、外部参照を解決します。/CC オプションを指定すると、環境変数 LIBU8 に指定された標準ディレクトリでこれらのライブラリファイルを検索します。環境変数は、DOS プロンプトで次のコマンドラインによって設定できます。

```
C:> SET LIBU8=directory <CR>
```

directory には、RLU8 がライブラリファイルの検索に使用する標準ディレクトリの名前を指定します。

デバッグ情報付きのアブソリュートファイルを作成するには、次のように/SD オプションを使用してオブジェクトファイルをリンクする必要があります。

```
C:> RLU8 FILE1 FILE2,.....,/CC /SD <CR>
```

このオプションは、必要なデバッグ情報付きのアブソリュートファイルを作成するようリンカに通知します。

11. 終了コード

CCU8 は、終了時にオペレーティングシステムに制御を渡しますが、オペレーティングシステムに制御を渡すときに、CCU8 は終了コードと呼ばれる数値を返します。終了コードと対応する終了状態を次の表に示します。

表 11.1

終了コード	終了状態
0	正常終了
1	コンパイル中にワーニング発生
2	コンパイル中にエラー発生
3	フェイタルエラーによる強制終了

終了コード 0 (正常終了)は、ワーニングまたはエラー無しでファイルの最後までコンパイル処理が行われたことを示します。

終了コード 1 (ワーニング)は、コンパイル処理はファイルの最後まで行われ、コンパイル中にワーニングメッセージが出力されたことを示します。エラーは検出されていません。出力ファイルは作成されています。

終了コード 2 (エラー)は、コンパイル処理はおそらくファイルの最後まで行われ、コンパイル中にエラーメッセージが生成されたことを示します。ワーニングも発生しているかもしれません。この場合、出力ファイルは作成されません。

終了コード 3 (フェイタル)は、フェイタルエラーによってコンパイルが異常終了したことを示します。この場合、出力ファイルは作成されません。

12. エラーメッセージおよびワーニングメッセージ

コンパイラが出力するエラーメッセージには、次の 3 種類があります。

1. フェイタルエラーメッセージ
2. エラーメッセージ
3. ワーニングメッセージ

それぞれの種類のメッセージを、番号順にエラーの簡単な説明と合わせて以下に示します。すべてのメッセージには、エラーが発生したファイル名と行番号が表示されます。

12.1 フェイタルエラーメッセージ

フェイタルエラーメッセージは、コンパイラがプログラムをそれ以上処理することができないような重大な問題が発生したことを示します。フェイタルエラーメッセージを表示した後、実行は即座に終了します。 CCU8 では次のフェイタルエラーメッセージが生成されます。

12.1.1 コマンドライン

F0000 Source file not given

コンパイルするソースファイルがコマンドラインに指定されていません。

F0001 Invalid filename, ‘.C’ or ‘.H’ extension expected

ソースファイルのファイル名に、.C または.H または.c または.h 以外の拡張子が付いています。

F0002 Invalid command line option '*option*'
コマンドラインに不正な'*option*'が指定されています。

F0003 Directory not specified with /I option
インクルードディレクトリ名が/I オプションで指定されていません。

F0004 Filename not specified with /CT option
コールツリーファイル名が/CT オプションで指定されていません。

F0005 Type is not specified with /T option
DCL ファイル名が/T オプションで指定されていません。

F0006 Constant not specified with /SS option
スタックサイズ定数が/SS オプションで指定されていません。

F0007 Constant not specified with /SL option
識別子の最大長が/SL オプションで指定されていません。

F0008 Macro is not specified with /D option.
マクロ名が/D オプションで指定されていません。

F0009 Invalid constant for /SS option
不正な定数が/SS オプションで指定されています。

F0010 Invalid stack size.
/SS オプションで指定する定数は、0 以上 65535 以下でなければなりません。

F0011 Stack size should be even.
/SS オプションで指定するスタックサイズは、偶数でなければなりません。

F0012 Invalid constant for /SL option
不正な定数または定数でないものが/SL オプションで指定されています。

F0013 Invalid identifier length
/SL オプションで指定した定数は、31 以上 254 以下の範囲外です。

F0014 Duplicate command line option '*option*'
option は、コマンドラインに複数指定されています。

F0015 Duplicate preprocessor option

プリプロセッサオプション/LP および/PC が両方ともコマンドラインに指定されています。

F0016 Duplicate memory model option

メモリモデルオプション/MS および/ML が両方ともコマンドラインに指定されています。

F0017 Duplicate data access specifier option

データアクセス指定子オプション/near および/far が両方ともコマンドラインに指定されています。

F0018 /CT and preprocessor options are mutually exclusive

/CT オプションが/LP オプションまたは/PC オプションと組み合わせて指定されています。

F0019 /LE and preprocessor options are mutually exclusive

/LE オプションが/LP オプションまたは/PC オプションと組み合わせて指定されています。

F0020 Illegal combination of optimization options.

最適化オプションが誤って使用されています。

F0021 Type is not specified

必須オプションの/T が指定されていません。

F0022 Insufficient memory

コンパイラのメモリ不足です。

F0023 Error in accessing the input file

コンパイル中にコンパイラが入力ファイルにアクセスできません。

F0024 Invalid identifier for /D option

不正な識別子が/D オプションで指定されています。

F0025 Invalid identifier for /U option

不正な識別子が/U オプションで指定されています。

F0026 Macro is not specified with /U option

マクロ名が/U オプションで指定されていません。

F0027 Invalid constant for /W option

不正な定数が/W オプションで指定されています。

F0028 Invalid warning level

/W オプションで指定した定数は、0 以上 3 以下でなければなりません。

F0029 Warning level is not specified with /W option

/W オプションにワーニングレベルが指定されていません。

F0030 Invalid warning number for /Wc

/Wc オプションに不正なワーニング番号が指定されています。

F0031 Warning number is not specified with /Wc option

/Wc オプションにワーニング番号が指定されていません。

F0032 /Zg and preprocessor options are mutually exclusive

/Zg オプションが、/LP オプションまたは /PC オプションと組み合わせて指定されています。

F0033 '*identifier*' option cannot be specified after source file name

/I、/Fa、/D、/U、/W 以外のコマンドラインオプションが、ソースファイル名の後に指定されています。

F0034 Unable to open response file 'filename'

指定した filename のファイルがオープンできません。

F0035 Response file not found 'filename'

指定した filename のファイルが存在しないか、見つかりません。

F0040 CC1U8 is not in the path

CC1U8 の実行可能プログラムが、パス内に存在しません。

F0041 CC2U8 is not in the path

CC2U8 の実行可能プログラムが、パス内に存在しません。

F0042 Unable to remove file ‘filename’

ファイル‘filename’が削除できません。

F0043 Unable to read input file

ファイル‘filename’が読み取れません。

F0044 Unexpected end of file

ファイルが不正な位置で終わっています。

F0045 /Fa and preprocessor options are mutually exclusive.

/Fa オプションが、/LP オプションまたは/PC オプションと共に指定されました。

F0046 Incompatible version of ‘CC1U8.EXE or CC2U8.EXE’

CC1U8.EXE または CC2U8.EXE の製品バージョンが、CCU8.EXE と異なっていました。

F0047 Illegal combination of /nofar and /far

コマンド行で/nofar オプションと/far オプションの両方が指定されました。

F0050 Unable to open input file ‘*filename*’

指定された‘*filename*’が存在しないか、オープンできないか、見つかりませんでした。

F0051 Unable to open output file

コンパイラが出力ファイルをオープンできませんでした。次のいずれかが原因と考えられます。

* 領域不足のため、ファイルがオープンできない。
* ‘filename’と同じ名前の読み取り専用ファイルがすでに存在する。
* /Fa オプションで指定された出力ファイルのパスまたはディレクトリが存在しない。

F0052 Unable to open list file

コンパイラがリストファイルをオープンできませんでした。次のいずれかが原因と考えられます。

* 領域不足のため、ファイルがオープンできない。
* ‘filename’ と同じ名前の読み取り専用ファイルがすでに存在する。

F0053 Unable to open calltree file

コンパイラがコールツリーファイルをオープンできませんでした。次のいずれかが原因と考えられます。

* 領域不足のため、ファイルがオープンできない。
* ‘filename’ と同じ名前の読み取り専用ファイルがすでに存在する。

12.1.2 一般

F1000 File close error

コンパイラは入力ファイルまたは出力ファイルをクローズできません。

F1001 Internal stack overflow

ソースプログラムの処理中に、コンパイラ内の内部スタックオーバーフローが発生しました。

F1002 Internal compiler error

CCU8 の内部動作に障害が発生しました。

F1003 Insufficient memory

コンパイラがメモリを使い切ってしまいました。

F1004 Too many errors

ソースプログラム内のエラーの数が、コンパイラの最大限度数を超えました。

F1005 Floating point overflow

浮動小数点演算でオーバーフローが発生しました。

F1006 Divide overflow

整数型の定数の除算、剰余算でオーバーフローが発生しました。

F1007 Unable to read input file

コンパイル処理中に、コンパイラが入力ファイルの読み込みまたはアクセスができませんでした。

12.1.3 プリプロセッサ

F2000 Bad preprocessor directive '*string*'

#の後に指定した *string* は、正しい前処理指令ではありません。

F2001 Incomplete assembly block

#asm 前処理指令が対応する#endasm で終了していないか、#pragma asm プラグマが対応する#pragma endasm プラグマで終了していません。

F2002 Unexpected end of file

ファイルの終わりを予期しないところで検出しました。

F2003 Line number exceeds maximum value

指定したソースファイルが大きすぎます。

F2004 Too many nested '#ifxxxx's

前処理指令#ifxxxx のネストの最大を超えています。

F2005 Unable to open include file 'filename'

指定した#include 'filename'が存在していないか、オープンできないか、または見つかりません。

F2006 Integer constant expression expected

#if および#elif 前処理指令の両方には定数式を指定しなければなりません。

F2007 Path exceeds maximum limit

前処理指令#include に指定したファイルパスが最大長を超えている可能性があります。

F2008 '#if[n]def' expected an identifier

#ifdef または#ifndef 前処理指令には識別子を指定しなければなりません。

F2009 '#endif' expected

#if、#ifdef または#ifndef 前処理指令が#endif 前処理指令で終了する前にファイルの終了を検出しました。

F2010 Parameter buffer overflow

マクロのパラメータの文字数が最大長を超えています。

F2011 Macro buffer overflow

マクロ定義中の置き換えトークン文字列が最大長を超えています。

F2012 Too many nested include files

#include ファイルのネストが越えています。再帰が発生している可能性があります。

F2013 Internal buffer overflow

1 つの識別子に対するマクロ展開がコンパイラの最大値を超えています。

12.1.4 字句

F3000 String too long

メモリ不足のため全文字定数を保持できません。

12.1.5 構文および意味

F4000 Struct/Union nesting too deep

構造体/共用体のネストレベルの数がコンパイラの制限値を超えています。

F4001 Parser stack overflow

ソースプログラムの処理中にコンパイラの解析部スタックがオーバーフローしました。

F4002 Too many nesting levels

制御文(ループ/スイッチ/if)のネストレベルの数がコンパイラの制限値を超えています。

F4003 Automatic allocation exceeds 32k

ローカル(スタック)変数のヒープサイズがコンパイラの制限値を超えています。

F4004 Unexpected ‘token’

この token は正しく使われていません。

F4005 Operand stack overflow

ソースプログラムの処理中にコンパイラのオペランドスタックがオーバーフローしました。

12.2 エラーメッセージ

12.2.1 プリプロセッサ

E2000 #error：‘*string*’

コンパイラが#error 前処理指令を発見し、指定メッセージ *string* を表示しています。

E2001 ‘##’ cannot occur at the beginning of a macro definition

##演算子は前後に 2 つのトークンを必要とするので、マクロ定義は##演算子で始められません。

E2002 Parameter expected after ‘#’

#演算子の後に続くトークンは仮引数でなければなりません。

E2003 Formal parameter missing after ‘#’

#演算子の後に続くトークンは仮引数でなければなりません。

E2004 Reuse of formal parameter ‘identifier’

指定した識別子がマクロ定義の仮引数リスト内で 2 回使用されています。

E2005 Invalid line number in ‘#line’ directive

#line 前処理指令が不正な行番号を発見しました。

E2006 Unexpected in formal list ‘token’

指定した token がマクロ定義の仮引数リスト内で間違って使用されています。

E2007 Missing terminator ‘*character*’

#include 前処理指令のファイル名は‘>’または括弧‘””で終わっていなければなりません。

E2008 Unexpected end of line

マクロ定義内で予期せず行の終わりを検出しました。

E2009 ‘## cannot occur at the end of a macro definition

##演算子は前後に 2 つのトークンを必要とするので、マクロ定義は##演算子で終了できません。

E2010 ‘#define’ syntax

#define 前処理指令の構文が正しくありません。

E2011 ‘defined (*identifier*)’ expected

defined 演算子が間違って使用されています。

E2012 ‘#include’ expected a file name, found ‘no token’

#include 前処理指令で必要なファイル名指定がされていません。

E2013 Double quotes or angle brackets expected after ‘#include’

#include 前処理指令ではファイル名をかぎ括弧(<>)で囲んでいる、または二重引用符(“”)で囲んでいる必要があります。

E2014 ‘#line’ syntax

#line 前処理指令の構文が正しくありません。

E2015 ‘#line’ expected a string as a file name

#line 前処理指令で必要なファイル名指定がされていません。

E2016 Expected preprocessor command, found ‘*character*’

指定された *character* は頭に番号記号(#)が付いていますが、前処理指令の最初の文字ではありません。

E2017 ‘#undef’ expects an identifier

マクロ名は#undef 前処理指令で指定できません。

12.2.2 字句

E3000 Empty Character constant

誤った文字定数‘’が使用されています。

E3001 Too many characters in constant

複数の文字またはエスケープシーケンスを含んだ文字定数が使用されています。

E3002 Constant too big

整定数の範囲を超えています。

E3003 Hex constant must have atleast one hex digit
文字 0x の後に 16 進の値がありません。

E3004 Unmatched close comment '*/'
コンパイラは、コメント開始文字/*より前にコメント終了文字*/を発見しました。

E3005 Illegal escape sequence
¥の後の文字は有効なエスケープシーケンスではありません。

E3006 Bad octal number 'token'
8 進定数の列挙中に 8 または 9 が発見されました。

E3007 Invalid character 'character'
不正な文字'character'が発見されました。

E3008 Exponent value expected
浮動小数点数値で e または E の指定の後に指数が指定されていません。

E3009 Newline in string
文字列定数内に予期しない行の終わりがあります。

E3010 Newline in character literal
文字定数内に改行文字があります。

E3011 Invalid KANJI character
文字列内で、不正な漢字が検出されました。

E3012 Bad suffix on number
数値内で不正なサフィックスが検出されました。

12.2.3 構文および意味

E4000 More than one storage class specifier
複数の記憶クラス指定子が 1 つの宣言文で使用されています。

E4001 Unknown size struct/union
未定義の構造体または共用体のサイズを取得しようとしました。

E4002 Illegal combination of type specifiers

型指定子の誤った組み合わせが 1 つの宣言文に使用されています。

E4003 Function cannot return array

関数の戻り値が配列です。

E4004 'void' on variable

void はポインタ変数と関数の宣言のみに使用できます。また、関数への仮引数の時も同じです。

E4005 Redefinition of formal parameter '*identifier*'

指定された'*identifier*'が関数の仮引数リストで 2 回使用されています。

E4006 Nonaddress expression

項目の初期化に使用された式が左辺式にできませんし、定数にもなりません。

E4007 Redefinition of variable '*identifier*'

指定した'*identifier*'が、複数定義されています。

E4008 '*identifier*' not in parameter list

仮引数リストにない仮引数が宣言されています。

E4009 Syntax error：'*token*'

指定した *token* が構文エラーの原因です。

E4010 Unexpected '*token*'

予期しないところで *token* を検出しました。

E4011 Function cannot return function

関数の戻り値が関数です。

E4012 Array element type cannot be function

関数の配列は許可されていません。ただし、関数へのポインタの配列は許可されています。

E4013 Redefinition of struct/union/enum tag '*identifier*'

指定された'*identifier*'はすでに他の構造体または共用体または列挙タグに使用されています。

E4014 Missing subscript

多次元配列の定義で第 1 次元以外の次元の添え字の値がありません。

E4015 Bit-field must be of type int or char

ビットフィールドには int 型または char 型以外を指定できません。

E4016 Bit-field cannot have a modified type

構造体内のビットフィールドはポインタまたは配列または関数として宣言できません。

E4017 Named bit-field cannot have size '0'

構造体内の名前が付けられたビットフィールドのサイズが 0 です。名前が付けられていないビットフィールドのみサイズを 0 にできます。

E4018 Bit-field size out of range

ビットフィールド宣言に指定されたビット数が、整数ビットフィールドでは 0 以上 16 以下の範囲、または文字ビットフィールドでは 0 以上 8 以下の範囲にありません。

E4019 Struct/Union member redefinition '*identifier*'

'*identifier*' が同じ構造体、または同じ共用体の複数のメンバに対して使用されています。

E4020 Unexpected constant

指定された定数が間違って使用されています。

E4021 Expected formal parameter list, not a type list

関数本体が関数宣言文の後に開始されています。関数宣言文には仮引数リストではなく型リストのみが記述できます。

E4022 Struct/Union too large

構造体変数または共用体変数の大きさがコンパイラの限界 64K を超えています。

E4023 Value out of range for enum constant

列挙定数の値が int 型に指定できる値の範囲外です。

E4024 Cannot use address of automatic variables as static initializer

静的変数を自動変数のアドレスで初期化しようとしました。グローバル変数または静的ローカル変数または外部変数のアドレスのみが、静的ローカル変数およびグローバル変数を初期化するのに使用できます。

E4025 Function cannot be a struct/union member

構造体または共用体のメンバは関数として宣言できません。

E4026 '*identifier*' uses unknown struct/union/enum

未定義の構造体または共用体を使用して、'*identifier*'が構造体変数または共用体変数として宣言されています。

E4027 Static function '*identifier*' has no body

関数が静的関数またはインライン関数として宣言され、定義されていないのに呼び出されました。

E4028 Negative subscript

配列のサイズを定義している値が負の数です。

E4029 Integral constant expression expected

整定数式を指定してください。

E4030 '*identifier*' already has a body

すでに関数本体が定義されている関数'*identifier*'の本体を定義しようとしました。

E4031 Nonconstant initializer

初期化で定数でないオフセットが使用されています。

E4032 Undefined struct/union tag

未定義の構造体または共用体のタグを使用して、識別子が構造体変数または共用体変数として宣言されました。

E4033 Left of '*identifier*' has undefined struct/union

'*identifier*'または->'*identifier*'の左オペランドが、本体が定義されていない構造体または共用体の名前またはポインタです。

E4034 Illegal initialization

初期化式は誤っています。

E4035 Function cannot be initialized

関数を初期化しようとしました。

E4036 Too many initializers

初期化の数が初期化するオブジェクト数を越えました。

E4037 Array initialization needs curly braces

配列集合体型を初期化するには大括弧({})が必要です。

E4038 Struct/Union initialization needs curly braces

構造体または共用体などのような集合体型を初期化するには、初期化は大括弧({})で囲まれていなければなりません。

E4039 Same type qualifier is used more than once

1 つの宣言中に、同じ型修飾子が、直接かまたは複数の typedef による宣言で、同じ指定子リストまたは修飾子リストに複数記述されています。

E4040 '*identifier*' typedef cannot be used for function definition

typedef が、関数定義に記述されています。

E4041 Invalid subscript

配列サイズに定義されている値がゼロです。

E4042 '*qualifier*' can qualify data only

データ型でないオブジェクトが__near または__far で修飾されています。

E4043 '*qualifier*' can qualify functions only

関数型でないオブジェクトが__noreg で修飾されています。

E4044 Segment lost during conversion

far ポインタを near ポインタに変換しようとしました。

E4045 More than one '*qualifier*' qualifier specified

修飾子__near、__far、または__noreg のいずれかが複数指定されました。

E4046 Illegal combination of __near and __far

変数が__near および__far の両方で修飾されています。

E4047 Illegal combination of __near and __huge

変数が__near および__huge の両方で修飾されています。

E4048 Illegal combination of __far and __huge

変数が__far および__huge の両方で修飾されています。

E4049 Huge data can not be referred by far qualified pointer

huge データは near 修飾のポインタのみで参照できます。

E4050 Huge data should be declared with pointer

huge データはポインタのみで宣言してください。

E4051 Array element type cannot be huge

huge ポインタの配列は許されていません。

E4052 Bit-field must have type ‘int’, ‘signed int’ or ‘unsigned int’

ビットフィールドが、 int 型、符号付き int 型、符号なし int 型、以外の型で指定されました。このエラーは、/Za オプションを指定したときにのみ表示されます。

E4053 Typename expected

宣言が宣言修飾子なしに指定されました。このエラーは、/Za オプションを指定したときのみ表示されます。

E4054 ‘*identifier*’：cannot initialize extern variable within block scope

外部変数がブロック内で初期化されました。このエラーは、/Za オプションを指定したときのみ表示されます。

E4055 File must contain atleast one external linkage

外部リンクがないソースファイルが指定されました。各ファイルには、最低 1 つのグローバルオブジェクト(データ変数もしくは関数のいずれか)がなければなりません。このエラーは、/Za オプションを指定したときのみ表示されます。

E4056 __EI not allowed in function ‘identifier’

多重割り込みを禁止するように、関数‘identifier’が指定されました。

E4057 Interrupt / SWI function ‘identifier1’ is not allowed in category 1 function ‘identifier2’

関数‘identifier2’が、複数割り込みを禁止するカテゴリ 1 として指定されています。このため、割り込み/SWI 関数‘identifier1’は許可されませんでした。

E4059 __far or __huge not allowed with /nofar

修飾子__farまたは__hugeのいずれかが、ソースファイルで指定されました。このエラーは、/nofarオプションが指定された場合にのみ表示されます。

E4060 Segment name 'segment name' specified in built-in function is invalid

引数として組み込み関数に渡された文字列が、有効なセグメント名ではありません。このエラーは、文字列に 1 文字以上が含まれ、それがセグメント名として許可されない場合に表示されます。

E4061 Segment count exceeds 65535 limit

CCU8 によって生成された、1 コンパイル単位内のセグメントの数が、65535 を超えています。

E4062 Undefined segment name 'segment name'

組み込み関数で使用されているセグメント名が、同じコンパイル単位内で定義されていません。組み込み関数で使用されるセグメント名は、segment/segdef プラグマによって定義されているもの、またはコンパイラによって自動的に生成されたものでなければなりません。

E4063 segbase_f not allowed with /nofar

__segbase_f 関数は__far ポインタを返しますので、/nofar オプションが指定された場合は有効ではありません。

12.2.4 式

E5000 Expression does not evaluate to a function

オペランドが関数のように使用されていますが、関数ではありません。

E5001 '*identifier*' is not a function

関数として宣言されていない'*identifier*'に対して関数本体を定義しようとしました。

E5002 '*identifier*' undefined

指定された *identifier* が使用前に定義されていません。

E5003 Subscript on non array

配列ではない変数に対して添え字が使用されました。

E5004 '*operator*'：illegal for struct/union

構造体型および共用体型の値には指定された'*operator*'を使用できません。

E5005 Left of .'*identifier*' must have struct/union type

'.'演算子の左オペランドは構造体型または共用体型でなければなりません。

E5006 '*identifier*' is not struct/union member

'.'または'->'演算子の右側の識別子は指定された構造体型または共用体型のメンバではありません。

E5007 '*operator*' needs lvalue

指定した'*operator*'には左辺式オペランドがありません。

E5008 Lval specifies 'const' object

'const'で修飾された識別子は変更できません。const で指定したオペランドの代入または修正は誤っています。

E5009 '&' on register variable

レジスタ変数に対する'&'は誤っています。

E5010 Left of ->'*identifier*' must have struct/union pointer

'->'演算子の左オペランドは構造体または共用体のポインタでなければなりません。

E5011 Illegal indirection
間接演算子(*)がポインタでない値に適用されました。

E5012 ‘~’ : bad operand
演算子‘~’に対するオペランドは誤っています。

E5013 ‘!’ : bad operand
演算子‘!’に対するオペランドは誤っています。

E5014 ‘unary plus’ : bad operand
単項プラスに対するオペランドは誤っています。

E5015 ‘unary minus’ : bad operand
単項マイナスに対するオペランドは誤っています。

E5016 ‘*operator*’ : bad left operand
指定された演算子に対する左オペランドは誤っています。

E5017 ‘*operator*’ : bad right operand
指定された演算子に対する右オペランドは誤っています。

E5018 Pointer ‘+’ non integral value
整数でない値をポインタに加算しようとしました。

E5019 ‘+’ : 2 pointers
2 つのポインタを加算しようとしました。

E5020 Pointer ‘-’ non integral value
整数でない値をポインタから減算しようとしました。

E5021 ‘=’ : left operand must be lvalue
‘=‘の左オペランドは左辺式としなければなりません。

E5022 ‘&’ on bit-field
ビットフィールドのアドレスを取得しようとしました。

E5023 ‘*identifier*’ unknown size
‘*identifier*’オブジェクトのサイズが不明です。

E5024 Struct/Union comparison is illegal

2 つの構造体または共用体は比較できません。構造体または共用体の個々のメンバは比較できます。

E5025 Non-integral index

非整数式が配列の添え字に使用されています。

E5026 '*operator*': incompatible types

操作に一致しないオペランドを使用した式(たとえば、ポインタと整数でないオペランドを使用した式など)を検出しました。

E5027 Illegal index, indirection not allowed

添え字がポインタにならない式に適用されています。

E5028 Cast to function type is illegal

オブジェクトが関数型にキャストされました。

E5029 Cast to array type is illegal

オブジェクトが配列型にキャストされました。

E5030 Illegal cast

キャスト操作に使用された型は有効な型ではありません。

E5031 Unknown size

オブジェクトのサイズが不明です。

E5032 Subscript too large

添え字の値が 65535 を超えました。

E5033 Size exceeds limit

定義されたオブジェクトのサイズが 65535 を超えました。

E5034 '*identifier*' size exceeds limit

'*identifier*'オブジェクトのサイズが 65535 を超えました。

E5035 Too few actual parameters

関数に渡される実引数が仮引数で指定した数より少ないです。

E5036 Too many actual parameters

関数に渡される実引数が仮引数で指定した数より多いです。

E5037 Void function returning value

void キーワードで戻り値を返さないよう定義している関数が値を返している。

E5038 Illegal sizeof operand

sizeof 演算子に対するオペランドとしてビットフィールドが指定されました。

E5039 '*identifier*'：has bad storage class

指定した記憶クラスは、このコンテキストでは使用できません。たとえば、自動記憶クラスの指定子は、外部レベルでの変数の宣言には使用できません。

E5040 Parameter has bad storage class

指定された記憶クラスは、このコンテキストでは使用できません。

12.2.5 制御文

E6000 Illegal break
do、for、while または switch 文内でのみ break 文が使用できます。

E6001 Illegal continue
do、for、while 文内でのみ continue 文が使用できます。

E6002 Label '*identifier*' defined more than once
ラベル'*identifier*'が 1 つの関数内で複数定義されています。

E6003 Case '*constant*' already given
指定した case 値は switch 文内ですでに使用されています。

E6004 More than one 'default'
switch 文に複数の'default'キーワードが指定されています。

E6005 Label not defined '*identifier*'
goto 文で使用されたラベル'*identifier*'は関数内に定義されていません。

E6006 'case' without switch
'case'キーワードは switch 文内でのみ使用できます。

E6007 'default' without switch
'default'キーワードは switch 文内でのみ使用できます。

E6008 Switch expression is not integral
switch 式が整数ではありません。

E6009 Controlling expression has type 'void'
制御文の条件式が'void'です。

12.3 ワーニングメッセージ

/W コマンドラインオプションでワーニングレベルを表す定数を指定することで、ワーニングのチェックを特定のレベルに制限できます。/W0 コマンドラインオプションで、ワーニングチェックを無効にすることもできます。/W の後ろに指定できる定数と意味を次に示します。

表 12.1	
レベル	説明
0	ワーニングメッセージを表示しません。
1	重大なワーニングメッセージを表示します。
2	データ損失を引き起こすワーニングメッセージを表示します。
3	情報量の多いワーニングメッセージを表示します。

/W オプションを省略すると、ワーニングレベルはデフォルトで 1 になります。

ワーニングメッセージおよびその意味を以下に示します。ワーニング番号の後の括弧で囲まれた数は、ワーニングレベルを表します。

12.3.1 プリプロセッサ

W2000(1) ‘#undef’ ignored for predefined macro ‘identifier’

定義済みマクロ‘identifier’を undef しようとしました。

W2001(1) Not enough arguments for macro ‘identifier’

識別子のマクロ定義で指定した仮引数の数より少ない数の実引数を、指定したマクロ‘identifier’に指定されています。

W2002(1) ‘#define’ ignored for predefined macro ‘macroname’

事前定義マクロ‘macroname’をマクロとしてインストールしようとしました。

W2003(1) Close bracket expected

マクロ定義またはマクロ呼び出しの中に‘)’がありません。

W2004(1) Unexpected characters following directive ‘directive’

余分な文字が前処理指令の処理後に発見されました。

W2005(1) Redefinition of macro '*identifier*'

指定された'*identifier*'を再定義しています。

W2006(1) Comma separator missing

マクロ定義中の仮引数リストはコンマで区切らなければなりません。

W2007(1) Argument expected before '*character*'

マクロ呼び出しには引数が必要です。

W2008(1) Extra arguments ignored for macro '*macroname*'

識別子のマクロ定義に指定された仮引数の数より多い数の実引数を、指定したマクロ'*macroname*'で指定しています。

12.3.2 字句

W3000(1) Identifier truncated to '*identifier*'

識別子の最大長は、/SL オプションに指定された値によって異なります。/SL オプションが指定されていない場合には識別子の最大長は 31 文字になります。識別子は許された最大長で切り捨てられ、残りの文字は無視されます。

W3001(1) String too long, truncated

文字列の長さが 1023 文字を超えています。

W3002(3) Possible nested comment

コメントの開始'/*'がコメント内に見つかりました。コメントはネストできません。

12.3.3 構文と意味

W4000(1) Auto/Register ignored for global variables

グローバル変数を自動記憶クラスまたはレジスタ記憶クラスとして宣言しようとしました。

W4001(1) Formal parameters ignored

関数が引数無しと宣言されていますが、関数定義に仮引数宣言が含まれているか、関数呼び出しに引数が指定されています。

W4002(1) ‘const’ ignored on argument

関数の仮引数はスタックに割り当てられているので、‘const’は仮引数では無視されます。/Za オプションが指定されていると、ワーニングメッセージは表示されません。

W4003(1) Second parameter list is longer than first

関数が 2 回以上宣言されていて、最初の宣言での引数型リストより、2 回目の宣言での方が長く宣言されています。

W4004(1) First parameter list is longer than second

関数が 2 回以上宣言されていて、2 回目の宣言での引数型リストより、最初の宣言での方が長く宣言されています。

W4005(1) ‘const’ ignored for struct/union member ‘identifier’

構造体および共用体では、’const’で修飾された変数は使用できません。

W4006(1) Function was declared with formal parameter list

関数が引数を取るように宣言されていますが、関数定義に仮引数宣言がないか。あるいは、関数呼び出しに引数が指定されていません。

W4007(1) ‘*identifier*’：array bound overflows

配列に対する初期化が多すぎます。余分な初期化は無視されます。

W4008(1) Parameter *number* declaration different

プロトタイプの引数宣言の型が仮引数の宣言と異なります。

W4009(1) Declared subscripts for arrays different

操作対象の 2 つのオペラントか配列であり、それらの配列の添え字の宣言が異なっています。

W4010(1) Function was declared with variable arguments

プロトタイプと関数の実定義との間で引数が一致しません。

W4011(1) Function was not declared with variable arguments

プロトタイプと関数の実定義との間で引数が一致しません。

W4012(1) ‘const’ ignored on local variable ‘*identifier*’

すべての const で修飾された変数は、データメモリに割り当てられます。しかし、ローカル変数はスタックに割り当てられるので、ローカル変数を修飾した‘const’は無視されます。

W4013(1) No declaration specifiers ; ‘int’ assumed

変数が宣言指定子の指定なしに宣言されました。型指定子‘int’が変数に対して仮定されます。

W4014(1) Sign information ignored for bit field

ビットフィールドのメンバが符号付きで宣言されました。

W4015(1) memory attribute on cast ignored

キャスト式でのメモリ修飾子がポインタでないオブジェクトを修飾しています。

W4016(1) __near ignored on struct/union member ‘*identifier*’

__near 修飾された変数は構造体または共用体では使用できません。

W4017(1) __far ignored on struct/union member ‘*identifier*’

__far 修飾された変数は構造体または共用体では使用できません。

W4018(1) __far ignored on local variable ‘*identifier*’

ローカル変数はスタックに割り当てられているので、__far で修飾できません。

W4019(1) __far ignored on argument variable ‘*identifier*’

引数はスタックに割り当てられているので、__far で修飾できません。

W4020(1) Indirection to different types

式で使用されているポインタが異なったメモリを指しています(ポインタサイズの不一致)。

W4021(1) __noreg ignored for ‘main’

関数‘main’が __noreg で修飾されています。

W4022(1) Missing return value for function ‘*function name*’

関数は値を返すよう宣言されていますが、戻り値なしで返されました。

W4023(1) ‘*function name*’：no return value

関数‘*function name*’は値を返すよう宣言されていますが、return 文のないパスがあります。

W4024(3) Tag ‘*tag name*’ used prior to definition

関数プロトタイプ内で未定義の構造体または共用体を参照しようとしました。

W4025(1) Segment lost during conversion

far ポインタを near ポインタに変換しようとしました。

W4026(1) Line splice character encountered in comment line

行のコメント処理中に、行継続文字が検出されました。

W4027(2) Cast operation may lead to odd boundary access

文字型(符号付きまたは符号なし)へのポインタ型を、ワード境界アクセスの必要な他の型へのポインタに対してキャストしようとしました。

12.3.4 式

W5000(1) '*identifier*' function used as an argument

引数として関数を渡そうとしました。

W5001(1) Function used as an argument

関数の仮引数が関数として宣言されていますが、許されていません。仮引数は関数へのポインタに変換されます。

W5002(1) '*operator*'：different levels of indirection

式に異なるレベルの間接参照がある。

W5003(1) Atleast one void operand

型 void の式がオペランドとして使用されました。

W5004(1) '&' on array ignored

アドレス演算子(&)を配列に適用しようとしました。

W5005(1) Constant too large, converted to 'int'

case 文で指定した定数が最大整数値を超えました。

W5006(1) Division by zero

除算演算(/)の 2 番目のオペランドがゼロになりました。1 に変換されます。

W5007(1) Mod by zero

剰余演算(%)の 2 番目のオペランドがゼロになりました。1 に変換されます。

W5008(1) ‘*operator*’：indirection to different types

異なる型の値にアクセスするための式に間接演算子(*)が使用されています。

W5009(1) Function Parameter lists differed

仮引数の型が関数宣言(プロトタイプ)の対応する型と一致していません。

W5010(1) Far pointer truncated to ‘int’

far ポインタが‘int’型の変数に代入されています。セグメントアドレスは失われます。

W5011(1) Far pointer converted to ‘long’

Large コードまたは Large データメモリモデルでは、ポインタには long 変数が割り当てられます。

W5012(1) Near pointer converted to ‘long’

near ポインタが‘long’型の変数に代入されています。セグメントアドレスはゼロになります。

W5013(1) Parameter mismatch, actual parameter converted

実引数宣言での型が仮引数宣言と異なります。適切な変換が行われます。

W5014(1) Cast to different memory

コードメモリオブジェクトをデータメモリオブジェクトにキャストしています。またはその反対にデータメモリオブジェクトをコードメモリオブジェクトにキャストしています。

W5015(3) ‘*operator*’：signed / unsigned mismatch

関係演算子もしくは等価演算子の 2 つのオペランド間での符号情報に不一致があります。両オペランドは、対応する符号なしのものに変換されます。

W5016(3) Switch expression evaluates to 0/1

ブール値となる式が switch 式として使用されています。

W5017(1) Local variable ‘*identifier*’ used before initialization

参照する値が事前に割り当てられていないローカル変数を参照しました。

W5018(2) Unary minus operator applied to unsigned type

単項マイナス演算子のオペランドが符号なしの型です。この演算の結果は符号なしのままです。

W5019(2) ‘*operator*’：integer constant overflow

演算子(+、-、*)によって行われる演算が結果の整数定数に割り当てられたサイズをオーバーフロー、あるいはアンダーフローしました。

W5020(2) Float constant overflow

算術演算子と、桁数の少ない浮動小数点型への変換を含む演算で、浮動小数点型の定数に割り当てられたサイズからのオーバフローまたはアンダフローが発生しました。

W5021(2) ‘*operator*’：truncation of constant value

大きな値の整数定数を小さいサイズのロケーションに代入しようとしました。

W5022(2) ‘type conversion’：possible loss of data

浮動小数点型が整数型に変換されました。データが失われる可能性があります。

W5023(2) Conversion between different integral types

代入式のオペランドが異なった整数型で、左オペランドが右オペランドより小さいサイズです。

W5024(2) Conversion between different floating point types

代入式のオペランドが異なった float 型で、左オペランドが右オペランドより小さいサイズです。

W5025(3) ‘*identifier*’：unreferenced formal parameter

宣言された仮引数が関数本体では参照されていません。

W5026(3) Expression is useful only for its side effects

プログラムの実行には式はなんの影響も与えませんが、副次的な作用だけには有効です。

W5027(3) Meaningless use of an expression

式はなんの役にも立ちません。

W5028(3) Assignment within conditional expression

指定した制御式には、代入演算子が含まれます。等価演算子の代わりに、代入演算子が間違って使用されたようです。

W5029(3) ‘*identifier*’：unreferenced local variable

ローカル変数が関数の内で参照されていません。

W5030(3) ‘*function*’：unreferenced static function

静的関数が指定されたコンパイル単位内で参照されていません。

W5031(1) ‘*identifier*’：call to function with no prototype

関数呼び出しが、プロトタイプなしに行われています。

W5032(3) ‘*identifier*’：unreferenced static variable

関数内に定義された静的ローカル変数が関数内部で参照されていないか、またはファイル内に定義された静的グローバル変数がファイル内で参照されていません。

W5033(2) ‘*operator*’：value is out of range

値の型のビットサイズより大きい定数で値をシフトしようとしました。

W5034(3) Relational operation always results in *1/0*

符号なしの値とゼロを比較しています。この比較の結果は、常に TRUE か、常に FALSE です。

W5035(3) Expression within ‘sizeof’ is not evaluated

‘sizeof’演算子内の式は評価されません。

W5036(1) Sizeof returns 0

sizeof 演算子内の式が 0 と評価されました。

W5037(1) ‘*identifier*’：attempt to return address of auto variable

自動変数のアドレスを返そうとしています。

W5038(1) Table data cannot be allocated to physical segment #0

ROMWINDOW が指定されていないため、Const 修飾された変数を物理セグメント #0 に割り当てることができません。

12.3.5 制御

W6000(3) Switch statement contains no default label

default ブロックのない switch 文を検出しました。

W6001(3) Unreachable code

表示している行には制御フローが届きません。

W6002(3) ‘*identifier*’：unreferenced label

ラベルは関数内で参照されていません。

12.3.6 プラグマ

W8000(1) Unknown pragma ‘*token*’

無効なキーワードが前処理指令#pragma で指定されています。

W8001(1) ‘main’ cannot be specified in ‘*pragma keyword*’ pragma.

main 関数が‘*pragma keyword*’プラグマで指定されています。

W8002(1) ‘*pragma keyword*’ pragma variables should be global or static local

指定された変数はグローバル変数でも静的ローカル変数でもありません。

W8003(1) Vector address out of range for pragma ‘*pragma keyword*’

Interrupt プラグマまたは Swi プラグマに指定されたベクタアドレスが範囲外です。

ベクタアドレスの有効な範囲は次のとおりです。

Interrupt - 0x8 ~ 0x7e

Swi - 0x80 ~ 0xfe

W8004(1) Expected even vector address, for pragma ‘*pragma keyword*’

Interrupt プラグマまたは Swi プラグマで奇数ベクタアドレスが指定されています。

W8005(1) More than one function for the same vector address

Interrupt プラグマまたは Swi プラグマで同じベクタアドレスに 2 つの異なる関数が指定されています。

W8006(1) Pragma argument delimiter ‘,’ expected

/PF オプションがコマンドラインに指定されているので、プラグマ引数の区切り文字には‘,’(コンマ)を使用してください。

W8007(1) Pragma must appear before function definition

プラグマで指定された関数はプラグマで記述する前に本体を定義してはいけません。このワーニングメッセージは、 Interrupt プラグマまたは Swi プラグマ前処理指令で指定している関数がプラグマより前に定義されているとき表示されます。

W8008(1) Interrupt function has parameter/return value

Interrupt プラグマまたは Swi プラグマで指定している関数にパラメータや戻り値が指定されています。

W8009(1) '*pragma keyword*' address exceeds range

Absolute プラグマで指定されたアドレスは有効範囲外です。

アドレスの有効な範囲は次のとおりです。

Absolute(データ) - 0x0 ~ 0xffff

W8010(1) Pragma must appear before variable initialization.

プラグマで指定された変数はすでに初期化されています。

W8011(1) Duplicate pragma '*pragma keyword*'

プラグマ *pragma keyword* が複数指定されています。このワーニングメッセージはソースファイルの中で Stacksize プラグマを複数指定したときに表示されます。

W8012(1) Specified stack size out of range

stacksize プラグマで指定した定数は範囲外です。stacksize の有効な範囲は 0x0 以上 0xfffe 以下の偶数値です。

W8013(1) Expected even number as stack size

stacksize プラグマで指定したサイズが偶数ではありません。

W8014(1) More than one pragma specified for '*symbol_name*'

'*symbol_name*'が複数のプラグマで指定されています。

W8015(1) '*pragma keyword*' pragma expects function name

指定されたシンボルは関数ではありません。Interrupt プラグマまたは Swi プラグマには、関数名の指定が必要です。

W8016(1) Pragma keyword expected, found no token

#pragma の後にトークンがありません。

W8017(1) Unexpected characters following pragma '*pragma keyword*'

有効なプラグマ'*pragma keyword*'の後に予期しない文字が見つかりました。

W8018(1) Function cannot be specified in pragma '*pragma keyword*'

プラグマ'*pragma keyword*'の変数に関数が指定されています。

W8019(1) Enum constants are not allowed in pragma.

列挙定数がプラグマ前処理指令で指定されています。

W8020(1) 'Absolute' address leads to odd boundary access

このワーニングメッセージは次の理由で表示されます。

* 初期化変数に奇数アドレスが指定されました。

W8021(1) Invalid 'Absolute' address for the variable 'token'

変数'token'に対して Absolute プラグマで指定されたアブソリュートアドレスが 0xffff を超えています。

W8022(1) Interrupt function '*function name*' used in expression

Interrupt プラグマで指定された関数'*function name*'が式に使用されています。このプラグマで指定した関数は C プログラムで直接的または間接的に呼び出せません。

W8023(1) Constant expected, found no token

定数が#pragma 前処理指令に必要ですが、トークンがありません。

W8024(1) Constant expected, found '*token*'

定数が#pragma 前処理指令に必要ですが、'*token*'を検出しました。

W8025(1) Pragma syntax error

指定された#pragma の構文を CCU8 が認識できません。

W8026(1) Variable '*token*' specified in pragma not declared

プラグマで指定した変数がファイルで宣言されていません。プラグマで指定する変数は、すべてファイル内で宣言する必要があります。

W8027(1) Identifier expected for pragma, found no token

識別子が#pragma 前処理指令に必要ですが、トークンがありません。

W8028(1) Identifier expected for pragma, found '*token*'

識別子が#pragma 前処理指令で必要ですが、'*token*'を検出しました。

W8029(1) Unexpected 'Endasm' pragma ignored

'Endasm'プラグマが対応する Asm プラグマ無しに指定されました。

W8030(1) Identifier or constant expected for pragma, found ','

識別子または定数が#pragma 前処理指令に必要ですが、','が発見されました。

W8031(1) Segment number exceeds range

指定されたセグメント番号が 0 以上 255 以下の範囲にありません。

W8032(1) Segment should be 0 for ‘near’ variables

‘near’変数にゼロでないセグメントが指定されています。

W8033(1) Identifier expected for pragma, but found ‘,’

識別子が#pragma 前処理指令で必要ですが、‘,’ (コンマ)が発見されました。

W8034(1) Constant expected for pragma, found ‘,’

定数が#pragma 前処理指令で必要ですが、‘,’ (コンマ)が発見されました。

W8035(1) ‘*function name*’ specified in ‘Inline’ pragma is not expanded.

以下にあげる理由のいずれかのため、‘inline’プラグマで指定された関数‘*function name*’を展開しません。

* ジャンプ、ラベルまたはループが存在
* 関数が展開するのに大きすぎる
* 関数の引数の数が可変
* 関数本体が ASM ブロックを含む
* 関数定義がプラグマ宣言より前にある
* インライン展開レベルがインラインの深さ制限を超えている
* 再帰フラグがオフなのにインライン関数が再帰呼び出しされている

W8036(1) Inline depth out of range

Inlinedepth プラグマで指定されたインラインの深さが範囲外です。Inlinedepth の有効な範囲は 0 以上 255 以下です。

W8037(1) Invalid Romwin range

Romwin プラグマで指定されたアドレスが範囲外です。

アドレスの有効な範囲は次のとおりです。

start_address は 0 以上です。

end_address は start_address より大きい値です。

end_address の最大値は 0xffff です。

W8038(1) Interrupt function ‘*function name*’ declared without static modifier

Interrupt プラグマで指定された関数‘*function name*’が static 修飾子の指定なしに宣言されています。

W8039(1) Illegal combination of pragma ‘Romwin and Noromwin’

‘Romwin’プラグマおよび‘Noromwin’プラグマは互いに排他的です。

W8040(1) Absolute pragma variable ‘identifier’ in Romwin area is not const qualified

Absolute プラグマ変数は Romwin 領域内にあるのに、const で修飾されていません。

W8041(1) Absolute pragma variable ‘identifier’ outside Romwin area is const qualified

Absolute プラグマ変数は Romwin 領域外にあるのに、const で修飾されています。

W8042(1) ‘const’ variables cannot be specified in ‘pragma_keyword’ pragma

‘const’で修飾された変数が、’pragma_keyword’プラグマ内で指定されました。’const’で修飾された変数を指定できるのは、Absolute プラグマ内だけです。

W8043(1) Mismatch between command line option /NOWIN and pragma ROMWIN

ROMWIN プラグマは、コマンドラインオプションの/NOWIN と矛盾します。

W8044(1) Invalid category for Interrupt pragma

Interrupt プラグマで指定されたカテゴリの値が、有効ではありません。カテゴリの有効な値は 1 と 2 です。

W8045(1) Invalid stack size

‘#pragma stack size’で指定されたスタックサイズが0でした。

W8046(3) Segment name ‘Name of the segment’ is already specified in ‘Name of the Pragma’ pragma. Segment will be allocated to physical segment '0'

現在のコードセグメント(segcodeまたはsegintr)プラグマで指定されているセグメント名が、他のコードセグメントプラグマですでに指定されています。このワーニングは、名前が同じ2つのコードセグメントの物理セグメント属性が異なる場合に表示されます。このワーニングは、コマンドラインオプション/ML /W3が設定されている場合にのみ表示されます。__near指定子で指定されている「セグメント名」が__farで指定されているセグメント名と同じ場合には、ワーニングメッセージが表示された後で、論理セグメントが物理セグメント#0に割り当てられます。

W8047(1) Segment name ‘Name specified in the segment’ is already specified in

‘Name of the pragma’ pragma. Pragma Ignored

現在のセグメントプラグマで指定されているセグメント名が、他のセグメントプラグマですでに指定されています。
注: 次の場合には、このワーニングは表示されません。
1. segcodeプラグマとsegintrプラグマで同じ名前が指定されている場合
2. seginitプラグマとsegnoinitプラグマで同じ名前が指定されている場合

W8048(3) "Segment name 'Name of the segment' is already allocated to '__near/__far' 'Name of the pragma' segment. Segment will be allocated to physical segment '0'

このプラグマで指定されているものと同じ名前が、これより前のプラグマで、異なるデータアクセスタイプで指定されています。また、seginitプラグマとsegnoinitプラグマでは同じ名前を指定できますが、データアクセス指定子が異なっていると、このワーニングが表示されます。

W8049(1) Segment 'Name of the segment' is already allocated to 'CODE/DATA/TABLE' segment by 'Name of the pragma ' pragma. Pragma Ignored"

seginitセグメントでは、テーブルセグメントの名前は、プラグマで指定された名前の後に'TAB'を付加することで生成されます。

例 12.1

```
# pragma Seginit __near "SegA"
```

この場合のテーブルセグメントの名前はSegATABです。

seginitプラグマによって生成されたテーブルセグメントの名前とプラグマで指定されている名前が競合する場合は、このワーニングが表示されて、後のプラグマは無視されます。

注: seginit のテーブルセグメント名と segconst プラグマで指定されている名前が競合する場合には、このワーニングは表示されません。

W8050(1) Segment 'Segment name' is already allocated to 'Physical Segment [ANY /#0] '

segment in physical segment '[__near/__far]' by 'Name of the pragma' pragma. Segment will be allocated to physical segment 0.

segconst のテーブルセグメント名と seginit のテーブルセグメント名は同じでもかまいませんが、segconst の物理セグメント属性が near の場合は、このワーニングが表示されて、セグメントは物理セグメント 0 に割り当てられます。

W8051(1) Address out of the range '*Range*'.

プラグマで指定されているアドレスが、プラグマに有効な範囲内ではありません。

プラグマの範囲は、セグメントプラグマおよび使用されいてるデータアクセス指定子によって決まります。

W8052(1) Address should be outside ROMWIN range '*Rom Window*'. Pragma Ignored",

プラグマのアドレスは、romwinプラグマによって定義されるROM WINDOWの外側でなければなりません。

この制限は、seginit、segnoinit、segnvdataの各プラグマのアドレス指定形式に適用さ

れます。

W8053(3) ROM Window Address not defined

segconstプラグマのアドレス指定形式の場合、プラグマの定義の前にROM WINDOWが定義されていないと、このワーニングメッセージが表示されます。

注: このワーニングは、ワーニングレベルが3に設定されている場合にのみ表示されます。

W8054(1) Segment name ‘Name Specified’ specified in the pragma is an assembler reserved word. Pragma Ignored

セグメントプラグマで指定されている名前がアセンブラの予約語である場合は、プラグマは無視され、このワーニングが表示されます。

W8055(1) Invalid category for Swi pragma

swiプラグマで指定されているカテゴリの値が有効ではありません。カテゴリの有効な値は1と2です。

W8056(1) Swi function ‘function name’ already referenced

関数‘function name’は、swiプラグマで指定する前に参照することはできません。

W8057(1) Swi function ‘function name’ assigned to a pointer to function

関数‘function name’は、関数に対するポインタに割り当てられたSwi関数です。

W8058(1) Segment name expected, found comma

プラグマ引数のデフォルトの区切り文字は「空白文字」です。この動作は、コマンドラインオプション/PFを指定することで変更できます。/PFを指定すると、プラグマに対する引数の区切り文字はコンマになります。

セグメントプラグマの名前指定形式で区切り文字として「空白文字」の代わりにコンマが検出され、/PFオプションが指定されてないと、このワーニングが表示されます。

W8059(1) Segment address expected, found comma

プラグマ引数のデフォルトの区切り文字は「空白文字」です。この動作は、コマンドラインオプション/PFを指定することで変更できます。/PFを指定すると、プラグマに対する引数の区切り文字はコンマになります。

セグメントプラグマのアドレス指定形式で区切り文字として「空白文字」の代わりにコンマが検出され、/PFオプションが指定されてないと、このワーニングが表示されます。

W8060(1) Seginit segment name exceeds the limit ‘ limit ’, the name will be truncated to ‘Number of characters‘ characters",

seginitのセグメント名の場合、テーブルセグメントの名前の長さが有効な長さと見なされます。テーブルセグメント名を作成するには、セグメント名の後にさらに‘TAB’の3文字が付加されるので、seginitプラグマで指定できる名前の最大長は、‘Maximum Length -3’になります。Maximum Lengthは、/SLコマンドラインオプションで定義される値です。デフォルトは31です。

W8061(1) Variable 'Variable Name' is already allocated to default segment

変数'Variable Name'が、最初にデフォルトセグメントに配置された後、ユーザー定義のセグメントで有効になっています。変数は、デフォルトセグメントに残ります。変数が最初に記述されている箇所では初期化され、2番目の記述箇所では初期化されません。

W8062 (1) Variable 'Variable Name' is already allocated to ‘Segment’ segment with '__near/__far' specifier

変数'Variable Name'は、最初にユーザー定義のセグメントSegment1に配置された後、他のユーザー定義セグメントSegment2で有効になっています。変数は、最初に配置されたユーザー定義セグメントSegment1に残ります。変数は、記述されている両方の箇所で初期化されないか、または2番目の記述箇所でのみ初期化されません。

W8063 (1) Variable 'Variable Name' is reallocated to default segment

変数'Variable Name'は、ユーザー定義のセグメント名に最初に配置された後、デフォルトセグメントで有効になっています。変数は、デフォルトセグメントに再配置されます。変数は、最初の記述箇所では初期化されず、2番目の記述箇所で初期化されます。

W8064 (1) Variable 'Variable Name' is reallocated to 'Segment1' segment with '__near/far' specifier

変数'Variable Name'は、最初にユーザー定義のセグメントSegment1に配置された後、他のセグメントSegment2で有効になっています。変数は、後のユーザー定義セグメントSegment2に再配置されます。変数は、最初の記述箇所では初期化されず、2番目の記述箇所で初期化されます。

W8065(1) Variable 'Variable Name' is already allocated to the segment starting at the address 'Segment Address'

変数'Variable Name'は、アドレス'Segment Address'から開始するユーザー定義セグメントに最初に配置された後、他のユーザー定義セグメントSegment2で有効になっています。変数は、最初に配置されたセグメントSegment1に残ります。変数は、記述されている両方の箇所で初期化されないか、または2番目の記述箇所でのみ初期化さ

れません。

W8066(1) Variable 'Variable Name' is reallocated to the segment starting at the address 'Segment Address'

変数'Variable Name'は、ユーザー定義のアドレス'Segment Address'から開始するセグメントに最初に配置された後、他のユーザー定義セグメント Segment で有効になっています。変数は、後のユーザー定義セグメント Segment に再配置されます。変数は、最初の記述箇所では初期化されず、2 番目の記述箇所で初期化されます。

W8067(1) Address should be within the ROMWIN range '*romwin range*'.Pragma ignored.

segconstプラグマで指定されるアドレスは、それより前のソースファイルでROMWINプラグマを使用して指定されているROM WINDOWの内部に存在しなければなりません。このワーニングは、物理セグメント0のアドレスに対して表示されます。

W8068(1) Absolute address for 'Variable Name' crosses physical segment boundary. 'Variable Name' allocated to address 'Segment number ：0x0000'

特定の変数を割り当てるためのアドレスが、物理セグメント境界をまたいでいます。変数は、同じ物理セグメントの0番目のオフセットに割り当てられます。

W8069(1) Duplicate absolute 'Data/code' segment definition. Pragma Ignored

複数のデータプラグマまたは複数のコードプラグマで、同じアドレスが指定されています。

W8070(1) Argument '*Argument*' specified in optimization pragma is invalid. Pragma ignored

最適化プラグマで指定できる引数は、Od、Om、Ot、Og、Ol、Oa、またはdefaultのいずれかです。これら7つ以外のオプションを指定すると、このワーニングが表示されます。オプションでは大文字/小文字が区別されることに注意してください。

W8071(1) Invalid segment type 'Segment Type' in SEGDEF pragma. Pragma Ignored

segdefプラグマで指定されているセグメントタイプが、DATA、CODE、NVDATA、TABLEのいずれでもありません。

W8072(1) Argument 'Od' cannot be specified with any other arguments. Pragma ignored.

optimizationプラグマで、Odオプションと同時に他の最適化オプションが指定されると、このワーニングが表示されます。

W8073(1) Argument 'default' cannot be specified with any other arguments. Pragma ignored

defaultオプションと同時に他の最適化オプションが指定されると、このワーニングが表示されます。

W8074(1) Optimization control pragma cannot be specified within a function body. Pragma Ignored

optimizationプラグマが関数本体の内部で指定されると、このワーニングが表示されます。プラグマは無視されます。

W8075(1) Arguments 'Om' and 'Ot' cannot be specified concurrently. Pragma Ignored.

optimizationプラグマでOm引数とOt引数の両方を同時に指定すると、このワーニングが表示されます。ワーニングが表示された後、プラグマの定義は無視されます。

W8076(1) Segment type redefinition for 'Segment Name' in 'Pragma Name' pragma. Pragma Ignored.

segdefプラグマで指定されているセグメント名が、異なるセグメントタイプを持つ他のセグメントの名前と一致します。他のセグメントは、デフォルトセグメントまたはいずれかのセグメントプラグマでユーザーが指定したセグメントです。

W8077(1) Segment type expected, found comma

segdefプラグマの定義で空白文字の代わりにコンマが区切り文字として使用されていますが、/PFオプションが指定されていません。