注意：この日本語版文書は参考資料としてご利用ください。
最新情報は必ずオリジナルの英語版をご参照願います。

PIC® MCU 向け MPLAB® XC8 C コンパイラ ユーザガイド

開発ツールをご使用のお客様へ

重要:
どのような文書でも内容は時間が経つにつれ古くなります。本書も例外ではありません。Microchip 社のツールとマニュアルは、お客様のニーズを満たすために常に改良を重ねており、実際のダイアログやツールの内容が本書の説明とは異なる場合があります。最新の PDF 文書については、Microchip 社のウェブサイト(www.microchip.com/)をご覧ください。

文書は各ページのフッターに表記している「DS」番号で識別します。DS 番号のフォーマットは DS<文書番号><リビジョン>です。<文書番号>は 8 桁の番号、<リビジョン>はアルファベットの大文字です。

各ツールの最新文書は、onlinedocs.microchip.com/で見つける事ができます。

目次

開発ツールをご使用のお客様へ ... 1

1. 序章 ... 5
   - 1.1. 本書の表記規則 ... 5
   - 1.2. 推奨参考資料 ... 6

2. コンパイラの概要 ... 7
   - 2.1. 対応デバイス ... 7
   - 2.2. C 言語規格 ... 7
   - 2.3. ホストとライセンス ... 8
   - 2.4. 用語の定義 ... 8
   - 2.5. 互換開発ツール ... 8

3. 使い方および FAQ ... 9
   - 3.1. コンパイラのインストールと有効化 ... 9
   - 3.2. コンパイラの起動 ... 10
   - 3.3. ソースコードの書き方 ... 11
   - 3.4. アプリケーションを設計意図通りの動作にする ... 19
   - 3.5. コンパイル プロセスを理解する ... 23
   - 3.6. 動作に問題のあるコードの修正 ... 28

4. コマンドライン ドライバ ... 32
   - 4.1. コンパイラの起動 ... 32
   - 4.2. コンパイル シーケンス ... 34
   - 4.3. ランタイム ファイル ... 37
   - 4.4. コンパイラ出力 ... 38
   - 4.5. コンパイラのメッセージ ... 40
   - 4.6. オプションの説明 ... 42
   - 4.7. MPLAB X IDE 統合開発環境 ... 67

5. C 言語の機能 ... 79
   - 5.1. C 規格への準拠 ... 79
   - 5.2. デバイスに固有の機能 ... 81
   - 5.3. サポートするデータ型と変数 ... 91
   - 5.4. メモリの割り当てとアクセス ... 106
   - 5.5. 演算子と命令文 ... 113
   - 5.6. レジスタの使用 ... 115
   - 5.7. 関数 ... 116
   - 5.8. 割り込み ... 121
   - 5.9. メイン、スタートアップ、リセット ... 129
   - 5.10. ライブラリ ... 131
   - 5.11. C 言語とアセンブリ言語の併用 ... 133
   - 5.12. 最適化 ... 140
   - 5.13. 前処理 ... 142
   - 5.14. プログラムのリンク ... 150

6. マクロアセンブラ ... 158

6.1. MPLAB XC8 のアセンブリ言語 158
6.2. アセンブリレベルの最適化 185
6.3. アセンブリ リストファイル 186

7. リンカ 194
7.1. 動作 194
7.2. psect と再配置 200
7.3. マップファイル. 200

8. ユーティリティ 205
8.1. アーカイバ/ライブラリアン 205
8.2. Hexmate. 206

9. ライブラリ関数 224
9.1. ライブラリ サンプルコード 224
9.2. <xc.h>デバイス固有関数 227

10. エラーおよび警告メッセージ 244
10.1. メッセージ 0～499 244
10.2. メッセージ 500～999 286
10.3. メッセージ 1000～1499 313
10.4. メッセージ 1500～1999 343
10.5. メッセージ 2000～2499 345

11. 処理系定義の振る舞い 356
11.1. 概要 356
11.2. コンパイル 356
11.3. 環境 356
11.4. 識別子. 357
11.5. 文字 357
11.6. 整数. 358
11.7. 浮動小数点数 359
11.8. 配列とポインタ 360
11.9. ヒント. 360
11.10. 構造体、共用体、列挙体、ビットフィールド 361
11.11. 修飾子 361
11.12. 前処理ディレクティブ 361
11.13. ライブラリ関数 362
11.14. アーキテクチャ 365

12. 改訂履歴 366

Microchip 社ウェブサイト 368

顧客変更通知サービス 368

カスタマサポート 368

Microchip 社のデバイスコード保護機能 368

法律上の注意点 368

商標 369
品質管理システム 369
各国の営業所とサービス 370

1. 序章

1.1 本書の表記規則

本書には以下の表記規則を適用しています。

表 1-1. 本書の表記規則

表記	適用	例
Arial、MS ゴシックフォント		
二重かぎカッコ: 『』	参考資料	『MPLAB® IDE ユーザガイド』
太字のテキスト	テキストの強調	...は**唯一の**コンパイラです...
角カッコ: [ ]	ウィンドウ名	[Output]ウィンドウ
	ダイアログ名	[Settings]ダイアログ
	メニューの選択肢	[Enable Programmer]を選択
かぎカッコ: 「 」	ウィンドウまたはダイアログのフィールド名	「Save project before build」
右山カッコ(>)で区切った下線付き斜体テキスト	メニュー項目の選択	*File>Save*
角カッコ([ ])で囲んだ太字のテキスト	ダイアログのボタン	**[OK]**をクリックする
	タブ	**[Power]**タブをクリックする
N'Rnnnn	Verilog 形式の数値(N は総桁数、R は基数、n は各桁の値)	4'b0010、2'hF1
山カッコ(< >)で囲んだテキスト	キーボードのキー	<Enter>、<F1>を押す
`Courier New` フォント		
標準書体の `Courier New`	サンプル ソースコード	`#define START`
	ファイル名	`autoexec.bat`
	ファイルパス	`c:\mcc18\h`
	キーワード	`_asm、_endasm、static`
	コマンドライン オプション	`-Opa+、-Opa-`
	ビット値	`0、1`
	定数	`0xFF、'A'`
斜体 *`Courier New`*	変数の引数	*`file.o`* (*`file`* は有効な任意のファイル名)
角カッコ: [ ]	オプションの引数	`mcc18 [options] file [options]`
中カッコとパイプ文字:{ | }	引数のどれかを選択する場合(OR 選択)	`errorlevel {0|1}`
省略記号: ...	繰り返されるテキスト	`var_name [, var_name...]`

	ユーザが定義するコード	`void main (void)`
		`{ ...`
		`}`

1.2 推奨参考資料

本書には、PIC ターゲット デバイス向けに ISO/IEC 9899:1999 標準(C99)プログラミング言語を使ってビルドを行う場合の MPLAB XC8 C コンパイラの使い方と機能を記載しています。参考資料として、Microchip 社が提供する以下の文書を推奨します。

MPLAB® XC8 C コンパイラ レガシー ユーザガイド
このバージョンのユーザガイドは、従来の xc8 コマンドライン ドライバ(ISO/IEC 9899:1990 標準(C90)プログラミング言語)を使うレガシー プロジェクト向けに書かれています。本コンパイラをレガシーモードで使う場合、仕様と出力が異なるフロントエンドを使います。

AVR® MCU 向け MPLAB® XC8 C コンパイラ ユーザガイド
8 ビット AVR デバイスをターゲットとするプロジェクト向けに書かれた XC8 C コンパイラユーザガイドです。

PIC® MCU 向け MPLAB® XC8 C コンパイラ リリースノート
MPLAB XC8 C コンパイラの使用に関する最新情報は、MPLAB® XC8 C コンパイラ リリースノート(HTML)に記載されているます。この文書は、本コンパイラのインストール フォルダ内の「docs」サブフォルダに保存されています。リリースノートには、本書に記載できなかった最新情報と既知の問題を記載しています。

Microchip Unified Standard Library リファレンス ガイド
このガイド文書には C 言語標準ライブラリによって定義された型、マクロ、関数に関する情報とサンプルコードが記載されています。この文書は全ての MPLAB XC C コンパイラに同梱されています。

開発ツールのリリースノート
各種開発ツールの最新情報については、MPLAB X IDE インストール フォルダ内の「docs」サブフォルダに保存されている各ツールの Readme ファイルを参照してください。

2. 本コンパイラの概要

MPLAB XC8 C コンパイラは、単独使用が可能で最適化機能を備えた C 言語向け ISO C99 クロスコンパイラです。

本コンパイラは全ての Microchip 社製 8 ビット PIC®および AVR®マイクロコントローラをサポートしますが、本書では PIC デバイス専用に xc8-cc ドライバの使用法を説明します。さらに C99 規格準拠でビルドされたプログラム向けのドライバ使用法も記載しています。本コンパイラを Microchip 社製 AVR デバイス向けに使う場合、『MPLAB® XC8 C Compiler User's Guide for AVR® MCU 』(DS50002750)を参照してください。C90 規格またはレガシーのコンパイラ ドライバ(xc8)を使う場合、『MPLAB® XC8 C Legacy Compiler User's Guide 』(DS50002053)を参照してください。

Note: 本書で説明する MPLAB XC8 の機能は、読者が Microchip 社製 PIC デバイスを使い、プログラムを C99 C 言語規格向けにビルドする事を前提としています。Microchip 社製 AVR デバイスまたは C90 規格向けに本コンパイラを使う場合、これらの機能は本書の説明とは異なる場合があります。

C99 規格向けにコンパイルする場合、本コンパイラは clang コンパイラ フロントエンドを使います。C90 プロジェクト向けにビルドする場合は、cpp/p1 フロントエンドを使います。

2.1 対応デバイス

本書では、Microchip 社の全ての 8 ビット PIC デバイス (ベースライン、拡張ベースライン、ミッドレンジ、拡張ミッドレンジ、PIC18 コアを含む) 向けに MPLAB XC8 C コンパイラがサポートする機能について説明します。デバイスの INI ファイル (本コンパイラのインストール フォルダ内の pic/dat/ini フォルダに格納) 内の ARCH フィールドで、コードをビルドする際に本コンパイラが使用するコア アーキテクチャを確認してください。各デバイスコアの特徴は以下の通りです。

- **ベースライン コア** は 12 ビット幅の命令セットを使います。このコアを採用したデバイスは PIC10、PIC12、PIC16 から選べます。ベースライン デバイスの ARCH フィールドの値は PIC12 です。
- **拡張ベースライン コア** も 12 ビット幅の命令セットを使いますが、命令が追加されています。一部の拡張ベースライン デバイスは割り込みをサポートし、割り込み用の命令が追加されています。拡張ベースラインコアを採用したデバイスは PIC12 と PIC16 から選べます。拡張ベースライン デバイスの ARCH フィールドの値は PIC12E または PIC12IE (割り込みサポート品)です。
- **ミッドレンジ コア** は 14 ビット幅の命令セットを使います。この命令セットにはベースライン コアよりも豊富な命令が含まれます。データメモリ バンクおよびプログラムメモリ ページも拡大されています。ミッドレンジ コアを採用したデバイスは PIC12、PIC14、PIC16 から選べます。ミッドレンジ デバイスの ARCH フィールドの値は PIC14 です。
- **拡張ミッドレンジ コア** も 14 ビット幅の命令セットを使いますが、命令と機能が追加されています。拡張ミッドレンジ コアを採用したデバイスは PIC12 と PIC16 から選べます。拡張ミッドレンジ デバイスの ARCH フィールドの値は PIC14E または PIC14EX です。
- **PIC18 コア** は 16 ビット幅の命令セットを使い、追加の命令と拡張されたレジスタセットを備えています。PIC18 コアを備えたデバイスの製品番号は PIC18 で始まります。一部の PIC18 デバイスは、拡張されたデータメモリとベクタ割り込みコントローラ モジュールを備えています。この割り込みコントローラは、固定アドレスの 2 優先度ベクタの代わりに、1 つまたは複数の割り込みベクタテーブルをサポートします。PIC18 デバイスの ARCH フィールドの値は PIC18 または PIC18XV (拡張データメモリ/ベクタ割り込みコントローラ モジュール内蔵品)です。

本コンパイラはターゲット デバイスが備える命令セット、アドレス指定モード、メモリ、レジスタをフルに活用します。

本コンパイラがサポートするデバイスの完全な一覧を表示する方法は、4.6.2.8 「print-devices」を参照してください。更新されたデバイスファミリ パック(DFP)をダウンロードすると、新しいデバイスのサポートが追加される場合があります。

2.2 C 言語規格

本コンパイラは、特に明記しない限り ISO/IEC 9899:1990 規格(C90 規格と呼ぶ) および ISO/IEC 9899:1999 規格(C99 規格と呼ぶ) に準拠し、単独使用が可能です。本コンパイラには、8 ビット PIC 組み込み制御アプリケーション向けにカスタマイズされた言語拡張も含まれます。

2.3 ホストとライセンス

MPLAB XC8 C コンパイラは、各種のオペレーティング システムに対応します。ご使用のコンパイラ バージョンがどのオペレーティング システムに対応するかは、コンパイラのリリースノートを参照してください。

本コンパイラはライセンスがなくても使えますが、ライセンスを購入する事でより高度な最適化が可能となります。ライセンスは後からいつでも購入できます。ライセンスがなくても基本的なコンパイラ動作、サポートするデバイス、利用可能メモリはライセンス付きのコンパイラと同じです。

2.4 用語の定義

本書では「コンパイラ」という用語を、MPLAB XC8 C コンパイラを構成する各種アプリケーションの全てまたは一部を指す意味で使います。ある動作をどのアプリケーションが実行したのか識別する事が重要ではない場合、「コンパイラが実行した」と表現します。

同様に「コンパイラ」という用語は、しばしばコマンドライン ドライバを指す意味でも使います(MPLAB XC8 C コンパイラ パッケージ向けドライバの固有名称は `xc8-cc` です)。ドライバとそのオプションは、4.6 「オプションの説明」に記載しています。「コンパイラ オプション」は通常「コマンドライン ドライバ オプション」を意味します。

同様に、「コンパイル」は、ソースコードから実行可能バイナリイメージを生成するための全てまたは一部の手順を意味します。

2.5 互換開発ツール

本コンパイラは、以下を含む各種 Microchip 社製ツールと連携します。

- MPLAB X IDE (www.microchip.com/mplab/mplab-x-ide): 全ての 8 ビット PIC および AVR デバイスをサポートする統合開発環境
- AVR®および SAM デバイス向け Microchip Studio (www.microchip.com/mplab/microchip-studio): 全ての 8 ビット AVR デバイスをサポート
- MPLAB X シミュレータ
- コマンドライン MDB シミュレータ(『Microchip Debugger (MDB) User's Guide』(DS52102)参照)
- 全ての Microchip 社製デバッグツールとプログラマ(www.microchip.com/mplab/development-boards-and-tools)
- 8 ビット PIC デバイスをサポートするデモボードとスタータキット

ツールがお客様のデバイスをサポートするかどうかは、各ツールのマニュアルで確認してください。

3. 使い方および FAQ

以下には Microchip 社製 8 ビット PIC デバイス向けにプロジェクトをビルドする際によく生じる疑問や問題に対処するためのヘルプと参照情報を記載しています。各セクションの冒頭にあるリンクをクリックする事で、興味のある項目を素早く見つける事ができます。一部の項目には、複数のセクションからリンクしています。

本章の内容

- コンパイラのインストールと有効化
- コンパイラの起動
- ソースコードの書き方
- アプリケーションを設計意図通りの動作にする
- コンパイル プロセスを理解する
- 動作に問題のあるコードの修正

3.1 コンパイラのインストールと有効化

以下では、本コンパイラのインストールまたは有効化の際に生じる疑問に答えます。

- コンパイラはどのようにインストールして有効にするのですか
- コンパイラの有効化に成功したかどうかはどのようにして確認できますか
- 同じコンパイラの複数のバージョンをインストールできますか

3.1.1 コンパイラはどのようにインストールして有効にするのですか

XC コンパイラのインストーラは、コンパイラのインストールとライセンスの有効化を同時に行います。『MPLAB® XC C コンパイラのインストールとライセンス認証』(DS52059)は、ウェブページ(https://www.microchip.com/compilers) 内の**[Documentation]** タブから入手できます。この文書には、シングルユーザ ライセンスとネットワーク ライセンスに関する詳細と、コンパイラを評価用に有効にする方法が記載されています。

3.1.2 コンパイラの有効化に成功したかどうかはどのようにして確認できますか

コンパイラが正しくインストールまたは有効化されていないと疑われる場合、コンパイラの問題と IDE の問題を分離するために、MPLAB X IDE を使わずにコンパイラの動作を確認するのが最善の方法です。

本コンパイラに同梱の `xclm` アプリケーションを使ってコンパイラのステータスを調べる事ができます。このアプリケーションは、DOS プロンプトに以下を入力する事で呼び出せます(適切なコンパイラパスを指定する必要があります)。

```
"C:\Program Files (x86)\Microchip\xc8\v2.00\bin\xclm" -status
```

これにより、コンピュータにインストールされたライセンスが表示され、コンパイラが正常に有効化されたかどうかを確認できます。コンパイラ ライセンスのステータスは、MPLAB X IDE 内のメニュー(*Tools > LicensesLicense > Status*) により確認する事もできます。

3.1.3 同じコンパイラの複数のバージョンをインストールできますか

はい、できます。本コンパイラとインストール プロセスは、同一コンパイラの複数のバージョンをインストールできるように設計されています。複数のバージョンをインストールした場合、MPLAB X IDE 内のオプションを変更する事で容易にそれらのバージョンを切り換える事ができます(参照: 3.2.4 「ビルドに使うコンパイラはどのように選択するのですか」)。

本コンパイラは、バージョンを表す名前が付いたディレクトリにインストールされます。これはインストーラによって指定される既定値ディレクトリです。

例: MPLAB XC8 コンパイラのバージョン 1.44 と 1.45 は、別々のディレクトリに格納されます。

```
C:\Program Files (x86)\Microchip\xc8\v1.44\
C:\Program Files (x86)\Microchip\xc8\v1.45\
```

3.2 コンパイラの起動

以下では、コマンドラインまたは MPLAB X IDE 内からコンパイラを実行する方法について説明します。また、オプションを指定してコンパイラを意図した通りに動作させる方法と、ビルドプロセスそのものについても説明します。

- MPLAB IDE 内からコンパイルする方法を教えてください
- コマンドラインからコンパイルする方法を教えてください
- make ユーティリティを使ってコンパイルする方法を教えてください
- ビルドに使うコンパイラはどのように選択できますか
- ライブラリのビルド方法を教えてください
- 利用可能なコンパイラ オプションとそれらの機能について知りたいのですが、何を見ればよいですか
- MPLAB X IDE のデバッグビルドでは何が違うのですか

以下の項目も参照してください。

- デバッガを使うには、コンパイルする際に何をする必要がありますか
- プロジェクト内でライブラリ ファイルを使う方法を教えてください
- 関数を独自のセクションに配置する方法を教えてください
- コンパイラはどの最適化レベルを適用しますか

3.2.1 MPLAB IDE 内からコンパイルする方法を教えてください

MPLAB X 統合開発環境内でプロジェクトをセットアップするためのガイドとして、MPLAB X IDE ユーザガイドとオンライン ヘルプが利用できます。

また、弊社ウェブサイトのコンパイラ ページ内の[Documentation]タブから『組み込み技術者のための MPLAB® XC8 ユーザガイド 』(DS50002400)をダウンロードできます。本コンパイラの「docs」フォルダに保存されている『MPLAB® XC8 入門ガイド』(DS50002173)も参照してください。

3.2.2 コマンドラインからコンパイルする方法を教えてください

コンパイラ ドライバの名前は全ての 8 ビットデバイスに対して `xc8-cc` です(例: Windows の場合、ドライバのファイル名は `xc8-cc.exe` です)。このアプリケーションは、コンパイルの全ての手順で使われます。このファイルは、コンパイラ ディストリビューションの「bin」ディレクトリに格納されています。個々のコンパイラ アプリケーション (アセンブラ、リンカ等)を別々に呼び出して実行しないでください。プロジェクトが複数のソースファイルで構成されている場合でも、1 つの命令でコンパイルとリンクを実行できます。

ドライバコマンドのフォーマットは 4.1 「コンパイラの起動」に記載しています。複数のコンパイラをインストールした場合のドライバの選択方法については、3.2.4 「ビルドに使うコンパイラはどのように選択するのですか」を参照してください。 ドライバに対するコマンドライン オプションの詳細は 4.6 「オプションの説明」を参照してください。ドライバに渡す事ができるファイルの一覧と説明は 4.1.3 「入力ファイルのタイプ」に記載しています。

3.2.3 make ユーティリティを使ってコンパイルする方法を教えてください

make ユーティリティ(`make` 等)を使うと、コンパイルは 2 つのステップで処理されます。この場合、最初のステップで中間ファイルが生成され、次のステップで最終的なコンパイルとリンクが実行されて 1 つのバイナリ出力が生成されます(4.2.3 「マルチステップの C コンパイル」参照)。

MPLAB XC8 C コンパイラは、従来のコンパイラ(XC16 および XC32 を含む)とは異なる中間生成ファイル フォーマットを使うユニークな技術(OCG と呼ぶ、4.6.6.9 「local オプション」参照)を採用しています。XC8 は、C ソースファイル向けの中間生成ファイルとして、オブジェクト ファイルの代わりに p-code ファイル(拡張子は `.p1`)を使います。

3.2.4 ビルドに使うコンパイラはどのように選択できますか

本コンパイラは、同時に複数のバージョンをインストールできるように設計されています。MPLAB X IDE 内でプロジェクトを作成した場合、プロジェクト プロパティ内の設定を変更するだけで、異なるコンパイラをプロジェクトのビルド用に選択できます。

実際に使うコンパイラは **[Project Properties]** ダイアログで選択します。このダイアログ内で**[Configuration]** カテゴリを選択(**Conf: [default]**、**default** はプロジェクト コンフィグレーションの名前)すると、インストール済みの MPLAB XC8 コンパイラのリストが右端の**[Compiler Toolchain]** に表示されます。そこから必要なコンパイラを選択してください。

その後に**[XC8 Global options]**、**[MPLAB XC8 Compiler]**、**[MPLAB XC8 Linker]** カテゴリを選択すると、そのコンパイラ向けの制御項目が表示されます。これらの右側にオプションのウィンドウ枠が表示されます。各カテゴリは複数のウィンドウ枠を持ち、これらはウィンドウ枠の上端近くにあるプルダウン メニューから選択できます。

3.2.5 ライブラリのビルド方法を教えてください

ライブラリアン `xc8-ar` を使って p-code (`.p1`)モジュールまたはオブジェクト(`.o`)モジュールからライブラリをビルドします。ライブラリにはタイプの異なるこれらのモジュールを一緒に含める事ができますが、ライブラリに含める C ソースコードは必ず p-code モジュールへとプリビルドしておく必要があります。アセンブリ ソースコードにのみオブジェクト モジュールを使います。p-code ファイルのビルドについては 4.2.3 「マルチステップの C コンパイル」を参照してください。ライブラリアン オプションについては 8.1 「アーカイバ/ライブラリアン」を参照してください。

例:

```
xc8-cc -mcpu=16f877a -c lcd.c utils.c io.c
xc8-ar -r myLib.a lcd.p1 utils.p1 io.p1
```

上記は `myLib.a` という名前のライブラリ ファイルを生成します。

MPLAB X IDE では、ライブラリ ファイルを最終的な出力としてビルドするライブラリ プロジェクトを作成できます。お客様のソースコードのライブラリを作成するために新規プロジェクトを作成する場合、[Microchip Embedded] カテゴリからライブラリ プロジェクトを選択します。

MPLAB X IDE 内でお客様のライブラリコードを C 言語レベルで構築する場合、ライブラリ ソースファイルとそれらを使用するプロジェクトの間の相対パスは、ライブラリ ビルドコマンドの実行位置とビルド時にソースファイルが格納されていた位置の間の相対パスと一致する必要があります。

3.2.6 利用可能なコンパイラ オプションとそれらの機能について知りたいのですが、何を見ればよいですか

コマンドラインで`--help` オプションを使うと、全てのコンパイラ オプションを含むリストが表示されます。

4.6 「オプションの説明」にも全てのオプションを記載しています。

3.2.7 MPLAB X IDE のデバッグビルドでは何が違うのですか

MPLAB X は、製品向けのビルドとデバッグ向けのビルド用に別々のボタンとメニューを備えています。

デバッグビルドを実行する場合、MPLAB ICD 4 等のデバッグツールを使ってプロジェクトをデバッグできるよう、MPLAB X はコンフィグレーション ビットも設定します。

MPLAB X IDE では、デバッグビルドを実行する場合にのみ、デバッガツール向けにメモリが予約されます。3.4.4 「デバッガを使うには、コンパイルする際に何をする必要がありますか」を参照してください。

デバッグビルドを実行する場合、MPLAB X IDE はプリプロセッサ マクロ `__DEBUG` を 1 に設定します。このマクロは、製品向けビルドでは定義されません。このマクロに対して`#ifdef` ディレクティブ等を使う事で、ソース内のコードを条件付きでコンパイルできます(5.13.1 「プリプロセッサ ディレクティブ」参照)。これにより、開発デバッグ中のプログラムは製品用とは異なるものとなります。

3.3 ソースコードの書き方

以下では、ソースコードを書く際に生じる問題について以下の項目に分けて説明します。

- C 言語の仕様
- デバイスに固有の機能
- メモリ割り当て
- 変数
- 関数
- 割り込み
- アセンブリコード

3.3.1 C 言語の仕様

ここでは、質問がよく寄せられる C 言語そのものに直接関係するソースコード上の問題について説明します。

- 式はどのタイミングでキャストすべきですか
- 明示的な型変換によって式の結果が変化してしまう事はありますか
- 英数字以外の文字をプログラムに入力する方法はありますか
- 別のソースファイル内で定義されている変数を使う方法を教えてください
- 別のソースファイル内で定義されている関数を使う方法を教えてください

3.3.1.1 式はどのタイミングでキャストすべきですか

式はキャスト演算子(丸カッコで囲んだ型、例: (int))を使ってキャストできます。いかなる場合も、型の変換は本当に必要な時にのみ慎重に行う必要があります。

説明のために以下の例を使います。

```
unsigned long l;
unsigned int i;
i = l;
```

この例では、long 型を int 型に代入するため、l の値は切り詰められます。コンパイラは代入演算子の右辺式の型(long)を左辺値の型(int) に変換します。これを「明示的な型変換」と呼びます。通常、コンパイラは、切り詰めによってデータの損失が生じる可能性がある事を示す警告を出力します。

long から int への変換を意図している場合、上の例では int 型へのキャストは不要であり、使うべきではありません。コンパイラは両辺のオペランドの型を認識し、適切な変換を実行します。キャストを使った場合、後でコードを変更する際に誤りを招く可能性があります。

例えば、

```
i = (int)l;
```

というコードを使っても結果は同じです。しかし、後で i を long に変更する際は、忘れずにキャストも変更するか削除する必要があります。これを忘れると l の値は代入時に切り詰められてしまい、正しい結果は得られません。非常に重要な事として、キャストが使われているとコンパイラは切り詰めに関する警告を生成しないので注意が必要です。

キャストは、コンパイラによって使われる型と実際に必要な型が異なる場合にのみ使うべきです。例として、除算結果を浮動小数点型の変数に代入する場合について考えます。

```
int i, j;
float fl;
fl = i/j;
```

この場合、整数除算の実行後に、丸め処理された整数結果が float 型に変換されます。i の値が 7、j の値が 2 であった場合、除算結果として 3 が得られ、これが float 型(3.0) に変換されて fl に代入されます。除算を float 型で実行する必要がある場合、以下のキャストが必要です。

```
fl = (float)i/j;
```

i または j のどちらかをキャストするだけで、コンパイラは浮動小数点除算としてコード化します。この場合、fl に代入される結果は 3.5 です。

明示的なキャストを使うと、使わなかった場合に生成されていた警告が生成されなくなる可能性があります。これも多くの問題の要因となります。なぜなら、コンパイラが生成する警告の数が多いほどコード内の潜在的なバグは見つけやすくなるからです。

3.3.1.2 明示的な型変換によって式の結果が変化してしまう事はありますか

はい、あります。コンパイラは常に整数拡張を使い、これを無効にする方法はありません(5.5.1 「整数拡張」参照)。加えて、2 項演算子のオペランドの型は、C 言語規格に従って共通の型に変換されます。オペランドの型が変更されると最終的な式の値が変化する可能性があるため、バイナリ演算子を処理する際に適用される C 言語規格の型変換規則を理解する事が非常に重要です。オペランドの型は、キャストによって手動で変更できます(3.3.1.1 「式はどのタイミングでキャストすべきですか」参照)。

3.3.1.3 英数字以外の文字をプログラムに入力する方法はありますか

C 言語規格(およびそれに準拠する MPLAB XC8 C コンパイラ) は、ソースプログラム内の文字および文字列リテラル内での拡張文字コードセットの利用をサポートしません。エスケープ シーケンスを使ってこれらの文字を入力する方法については、5.3.7 「定数の型と書式」を参照してください。

3.3.1.4 別のソースファイル内で定義されている変数を使う方法を教えてください

他のソースファイルで定義されている変数が `static` とも `auto` とも指定されていない場合、現在のファイルにその変数の宣言(定義ではない)を追加する事で、その変数にアクセスできます。この宣言には、キーワード `extern` と、その変数の定義内で指定されている型と変数名を使います。

```
extern int systemStatus; // declare systemStatus for use here
```

この記憶域クラス指定子は C 言語規格の一部であり、詳細は C 言語の参考書等を参照してください。

ファイル内のどこで宣言するかによって、その変数のスコープが決まります。関数の中に `extern` 宣言を書いた場合、変数のスコープはその関数内に制限されます。関数の外に書いた場合、ファイル内の全ての関数からその変数にアクセスできます。

多くの場合、宣言はヘッダファイル内に書き、そのヘッダファイルを C ソースコードにインクルード(`#include`)します(5.13.1 「プリプロセッサ ディレクティブ」参照)。

3.3.1.5 別のソースファイル内で定義されている関数を使う方法を教えてください

他のソースファイルで定義されている関数が `static` ではない場合、現在のファイルにその関数の宣言(定義ではない)を追加する事で、その関数を呼び出す事が可能となります。この宣言にはキーワード `extern` と関数プロトタイプを使います。宣言ではパラメータ名を省略できます。パラメータ名を宣言に含める場合、定義と一致させる必要があります。例:

```
extern int readStatus(int);  // declare readStatus for use here
```

この記憶域クラス指定子は C 言語規格の一部であり、詳細は C 言語の参考書等を参照してください。

多くの場合、宣言はヘッダファイル内に書き、そのヘッダファイルを C ソースコードにインクルード(`#include`)します(5.13.1 「プリプロセッサ ディレクティブ」参照)。

3.3.2 デバイスに固有の機能

以下では、Microchip 社製 PIC デバイスに特有の機能を設定または制御するためのコードについて説明します。

- コンフィグレーション ビットの設定方法を教えてください
- PIC デバイスの ID ロケーションの使い方を教えてください
- ミッドレンジ デバイスでは、リセットの原因はどのように特定できますか
- SFR にはどのようにアクセスできますか
- 関数を独自のセクションに配置する方法を教えてください
- SFR とその中のビットを表す名前はどのようにして見つけるのですか

以下の項目も参照してください。

- デバッガを使うには、コンパイルする際に何をする必要がありますか

3.3.2.1 コンフィグレーション ビットの設定方法を教えてください

コンフィグレーション ビットは、コード内でマクロまたは `config` プラグマを使って設定します。MPLAB X IDE を使って適切なプラグマを生成できますが、そのコードをプロジェクトに含まれるソースファイルにコピーする必要があります。これらのビットの設定方法は 5.2.5 「コンフィグレーション ビットへのアクセス」を参照してください。

3.3.2.2 PIC デバイスの ID ロケーションの使い方を教えてください

`Config` プラグマを使って ID ロケーション値をプログラミングできます(5.2.6 「ID ロケーション」参照)。

3.3.2.3 ミッドレンジ デバイスでは、リセットの原因はどのように特定できますか

リセットの原因は、STATUS レジスタの TO ビットと PD ビットによって特定できます。しかし、これらのビットは `main()`の前に実行される実行時起動コードによってすぐに上書きされてしまいます(5.9.2 「実行時起動コード」参照)。

C コードからアクセス可能な位置に STATUS レジスタの内容を保存しておく事で、実行再開後にリセットの原因を特定できます(5.9.2.4 「ステータス レジスタの保持」参照)。

3.3.2.4 SFR にはどのようにアクセスできますか

メモリマップ上の SFR に割り当てられた変数を定義したヘッダファイル(5.2.3 「デバイス ヘッダファイル」参照)がコンパイラと一緒に配布されます。これらは C 変数であるため、他の C 変数と同様に扱えます(同じ構文を使ってアクセスできます)。

構造体により、SFR 内の個々のビットに直接アクセスする事もできます(構造体内のビットフィールド メンバーにより、レジスタ内の個々の要素にアクセスできます)。SFR 内の各ビットに割り当てられた 1 ビット変数も個々に定義されていますが、これらはレガシー プロジェクト内でのみ使います(5.2.7 「C コードから SFR を使う」参照)。

通常、これらの変数にはデバイス データシートで指定されている名前が割り当てられます。これらの名前が認識されない場合、3.3.2.5 「SFR とその中のビットを表す名前はどのようにして見つけるのですか」を参照してください。

3.3.2.5 SFR とその中のビットを表す名前はどのようにして見つけるのですか

特殊機能レジスタ(SFR)とその中のビットには、そのレジスタ専用に割り当てられた変数を使ってアクセスします。しかし、これらの変数の名前がターゲット デバイスのデータシートで指定されている名前と一致しない場合があります。

<xc.h>ヘッダファイルを調べる事で、そのデバイスに対応するデバイス固有ヘッダファイルを見つける事ができます。このファイルは、これらの特殊な変数へのアクセスを可能にする変数を定義します。以前のコンパイル時に生成された前処理済みファイルを調べる事でも、これらの変数名を見つける事ができます。対応するソースファイルが<xc.h>をインクルードしていた場合、ターゲット デバイスの全ての SFR 変数とビットの定義が前処理済みファイル内で示されます。

MPLAB X IDE 内で default コンフィグレーションを使ってコンパイルした場合、前処理済みファイルはプロジェクトの build/default/production ディレクトリ(通常ビルド時)または build/default/debug ディレクトリ(デバッグビルド時) に保存されます。コマンドラインでコンパイルを行った場合、通常、前処理済みファイルはソースファイルのフォルダ内に保存されます。これらのファイルの拡張子は「.i」です。

3.3.3 メモリ割り当て

以下は、ソースコードがメモリ割り当てに及ぼす影響についての質問です。

- 変数を特定のアドレスに配置する方法を教えてください
- 変数を独自のセクションに配置する方法を教えてください
- 変数を特定のアドレス領域に配置する方法を教えてください
- 関数を特定のアドレスに配置する方法を教えてください
- 変数をプログラムメモリ内に配置する方法を教えてください
- 関数を固有のセクションに配置する方法を教えてください
- 関数を特定のアドレス領域に配置する方法を教えてください
- 特定のメモリ領域をコンパイラが使わないようにする方法を教えてください

以下も参考にしてください。

- 予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか
- 高書き込み耐性フラッシュを使ってコードではなくデータを保存する方法を教えてください

3.3.3.1 変数を特定のアドレスに配置する方法を教えてください

__at(*address*)コンストラクタを使って変数を絶対アドレス指定する方法が最も容易です(5.4.4 「絶対アドレス変数」参照)が、固有のセクションに変数を配置する事も可能です(5.14.3 「割り当てられたセクションの変更とリンク」参照)。絶対アドレス変数は、生成されたコード内で、その変数のシンボルよりもユーザが指定したアドレスを優先的に使います。

3.3.3.2 変数を固有のセクションに配置する方法を教えてください

__section()指定子を使って変数を新しいセクション(psect) 内に配置します。次に-Wl ドライバ オプションを使って、そのセクションを必要なアドレスにリンクします。これらの手順を示すサンプルコードは、5.14.3 「割り当てられたセクションの変更とリンク」に記載しています。

3.3.3.3 変数を特定のアドレス領域に配置する方法を教えてください

変数を固有のセクション(psect) 内に配置し、メモリ領域を定義してそこに新しいセクションを配置する必要があります。
`__section()`指定子を使って変数を新しいセクション内に配置します。`-Wl` ドライバ オプションを使ってメモリ領域を定義し、そこに新しいセクションを配置します。これらの手順を示すサンプルコードは、5.14.3 「割り当てられたセクションの変更とリンク」に記載しています。

3.3.3.4 関数を特定のアドレスに配置する方法を教えてください

`__at(address)`コンストラクタを使って関数を絶対アドレス指定する方法が最も容易です(5.7.3 「既定値の関数メモリ割り当てを変更する」参照)が、関数を固有のセクション内に配置する事も可能です(5.14.3 「割り当てられたセクションの変更とリンク」参照)。指定されたアドレスは、生成されたコード内で、その関数のシンボルよりも優先的に使われます。

3.3.3.5 変数をプログラムメモリ内に配置する方法を教えてください

`const` 修飾子付きの変数は、読み出し専用となります。これらのオブジェクトには書き込めないため、通常はプログラムメモリ内に配置されます(貴重なデータ RAM を節約するため)。ただし、`const` 修飾子付きパラメータと一部の `const` 修飾子付き `auto` オブジェクトは例外です。詳細は 5.1.2.4 「 const auto オブジェクト」と 5.4.3 「プログラム空間内のオブジェクト」を参照してください。`const` 修飾付き変数は、絶対アドレス指定により特定アドレスに配置する事もできます(5.4.4.2 「プログラムメモリ内の絶対アドレス オブジェクト」参照)。

3.3.3.6 関数を固有のセクションに配置する方法を教えてください

`__section()`指定子を使って関数を新しいセクション(psect) 内に配置します。次に`-Wl` ドライバ オプションを使って、そのセクションを必要なアドレスにリンクします。これらの手順を示すサンプルコードは、5.14.3 「割り当てられたセクションの変更とリンク」に記載しています。

3.3.3.7 関数を特定のアドレス領域に配置する方法を教えてください

メモリの特別な領域内に 1 つまたは複数の関数を配置する事で、それらの関数のコードを保護できます。これを行うには、関数を固有のセクション(psect) 内に配置し、メモリ領域を定義してそこに新しいセクションを配置する必要があります。
`__section()`指定子を使って関数を新しいセクション内に配置します。`-Wl` ドライバ オプションを使ってメモリ領域を定義し、そこに新しいセクションを配置します。これらの手順を示すサンプルコードは、5.14.3 「割り当てられたセクションの変更とリンク」に記載しています。

3.3.3.8 特定のメモリ領域をコンパイラが使わないようにする方法を教えてください

ビルド時にメモリを予約できます。`-mreserve` オプションを使う事で、ビルド時にデータメモリとプログラムメモリのアドレス領域を調整する事ができます(4.6.1.18 「メモリ予約オプション」参照)。既定値では、内蔵メモリの全領域を使う事ができます。しかし、これらのオプションを指定する事で、メモリの一部の領域を予約する事ができます。

3.3.4 変数

以下では、プログラム内での変数の定義、使い方、型に関連する質問に答えます。

- 期待通りの浮動小数点結果が得られません。なぜですか
- 変数の個々のビットにはどのようにアクセスできますか

以下も参考にしてください。

- 割り込みとメインライン コードの間でデータを共有する方法を教えてください
- 変数を特定のアドレスに配置する方法を教えてください
- 変数をプログラムメモリ内に配置する方法を教えてください
- PIC18 デバイスで、変数をプログラムメモリ内に配置する方法を教えてください
- 変数をローテートする方法を教えてください
- RAM バンクの使用/割り当て方法を教えてください
- 拡張ミッドレンジ PIC デバイスでのリニアメモリの使用法を教えてください。
- 変数と関数の位置を調べる方法を教えてください

3.3.4.1 期待通りの浮動小数点結果が得られません。なぜですか

C90 規格向けにビルドする場合にのみ、`float` 型と `double` 型の両方で浮動小数点型のサイズを調整できます(4.6.14.2 「short-float オプション」と 4.6.14.1 「short-double オプション」参照)。

浮動小数点型変数は有限数のビットを使って値を表現するため、代入された値の近似値を保持する事しかできません(5.3.4 「浮動小数点データ型」参照)。浮動小数点型変数は、厳密に表現可能な一連の離散実数値の中の 1 つを保持する事しかできません。これに含まれない値を代入すると、最も近い値に丸め処理されます。浮動小数点型変数の仮数部のビット数を増やすと厳密に表現可能な値の数が増え、丸め処理による平均誤差は減少します。

浮動小数点型の算術演算を実行すると、必ず丸め処理が発生します。これによって結果が正しくないかのように見える場合もあります。

3.3.4.2 変数の個々のビットにはどのようにアクセスできますか

これにはいくつかの方法があります。最も簡単で移植性の高い方法は、以下に示すように整数値を定義し、マクロを使ってこの値を読み出し、マスク値と論理演算を使って変数内の特定のビットをセットまたはクリアします。

```
#define  testbit(var, bit)   ((var) & (1 <<(bit)))
#define  setbit(var, bit)    ((var) |= (1 << (bit)))
#define  clrbit(var, bit)    ((var) &= ~(1 << (bit)))
```

1 行目は、整数 var 内の bit (ビット番号) に対応するビットがセットされているかどうか確認します。2 行目は、var 内の対応するビットをセットします。3 行目は、var 内の対応するビットをクリアします。別の方法として、整数型変数とビットフィールドを格納した構造体(5.3.5.2 「構造体内のビットフィールド」参照)を共用体(union)として定義する事もできます。

例:

```
union both {
  unsigned char byte;
  struct {
    unsigned b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
  } bitv;
} var;
```

これにより、var.byte を使ってバイトの全体にアクセスする事も、var.bitv.b0～var.bitv.b7 を使ってバイト内の個々のビットにアクセスする事もできます。

本コンパイラは bit 変数(5.3.2.1 「ビットデータの型と変数」)と構造体内のビットフィールドをサポートします。

3.3.5 関数

以下では、関数に関連する質問に答えます。

- 関数の最適なサイズは何で決まりますか
- 未使用の関数が削除されてしまう事を防ぐ方法を教えてください
- 関数をインライン展開する方法を教えてください

以下の項目も参照してください。

- 関数のサイズを調べる方法を教えてください
- 関数を特定のアドレスに配置する方法を教えてください
- 各関数で使われているリソースを調べる方法を教えてください
- 変数と関数の位置を調べる方法を教えてください
- C 言語で割り込みを使う方法を教えてください

3.3.5.1 関数の最適なサイズは何で決まりますか

一般的に言うと、関数のソースコードは小さくまとめるべきです。そうする事でコードが読みやすくなり、デバッグも容易です。少数のタスクだけを実行する関数は、コードを書く事もデバッグする事も非常に容易です。また、関数を小さくする事で、コーディング エラーの原因になりかねない副作用も抑える事ができます。しかし、組み込みプログラミングの領域では、小さな関数とそれらの呼び出しが多数存在するとメモリとスタックが不足する場合があるため、妥協が強いられる事が多くあります。

PIC10/12/16 デバイスでは、関数コードの保存と実行用に使われるプログラムメモリは複数のページで構成されます。1 ページよりもサイズが大きなアセンブリコードを生成する C 関数を書く事はできますが、可能であれば、そのような関数はより小さなルーチンに分割する事を推奨します(5.7.2 「実行可能コードのメモリ割り当て」参照)。他のページ内への呼び出しおよびジャンプ シーケンスを実行すると、同一ページ内でそれらを実行するよりもアセンブリコードが大幅に長くなります。関数がページ境界をまたぐほど大きい場合、ループ(または関数内でのジャンプを要求するその他のコード コンストラクタ) は、繰り返しのたびに長くなったジャンプのコードを使う可能性があります。

PIC18 の場合、内部メモリのページングの影響は他の PIC に比べて小さくなります。PIC18 の命令セットは、ページングの影響を受けずに任意位置への呼び出しとジャンプを可能にします。しかし、関数をできるだけ小さくするよう心掛ける事は必要です。

割り込み関数は、メモリページのサイズを超えないように書く必要があります。割り込み関数を複数ページに分割する事はできません。

どのデバイスでも、関数を小さくする事で、リンカはより容易に(エラーが生じる事なく)関数をメモリに割り当てる事ができます。

3.3.5.2 未使用の関数が削除されてしまう事を防ぐ方法を教えてください

C 関数のシンボルが手書きのアセンブリコード内で参照される場合、その関数は決して削除されません。たとえその関数が呼び出されなくても、あるいは C コード内でアドレスが割り当てられなかったとしても、削除されません。

この参照を含めるには、アセンブリ ソースファイルを手書きし、そのファイルをプロジェクトに追加する必要があります。このファイルでは、シンボルを参照する事だけが必要です。

例:

```
GLOBAL _myFunc
```

myFunc は C コード内での関数名です(アセンブリコード内では、名前の先頭にアンダースコアを付けます、5.11.3.3 「等価アセンブリ シンボル」参照)。以上により、最適化による関数の削除を防ぐ事ができます。

3.3.5.3 関数をインライン展開する方法を教えてください

inline 指定子(5.7.1.2 「inline 指定子」参照)または#pragma inline を使う事により、可能であれば関数をインライン展開するようコンパイラに示唆する事ができます。これは示唆に過ぎず、コンパイラはこれを無視する場合があります。この指定子/プラグマと intrinsic プラグマ[1]を混同しないよう注意が必要です(5.13.3.4 「#pragma intrinsic ディレクティブ」参照)。intrinsic プラグマは、対応するソースコードを持たない関数用であり、それらの関数はコンパイル中にコード ジェネレータによって特別に展開されます。

3.3.6 割り込み

以下では、割り込みおよび割り込みサービスルーチンに関する質問に答えます。

- C 言語で割り込みを使う方法を教えてください

以下も参考にしてください。

- 割り込みルーチンを高速にする方法を教えてください
- 割り込みとメインライン コードの間でデータを共有する方法を教えてください

3.3.6.1 C 言語で割り込みを使う方法を教えてください

ターゲット デバイスがどの割り込みハードウェアを備えているか知る必要があります。大部分のベースライン PIC デバイスは割り込み機能を全く備えていません。割り込み機能を備えたベースライン デバイスと、ミッドレンジ デバイスは単一の割り込みベクタを使います。PIC18 は、2 つの分離した割り込みベクタ領域を実装し、シンプルな優先度スキームを適用します。一部の PIC18 デバイスは、ベクタ割り込みコントローラを使って複数の割り込み関数を呼び出します。

C ソースコードでは、割り込みサービスルーチンとして機能するように関数を書く事ができます(5.8.1 「割り込みサービスルーチンの書き方」参照)。それらの関数は、関数の本文コードを実行する前と後でプログラム コンテクストを退避および復元し、各種のリターン命令を使います(5.8.4 「コンテクスト スイッチ」参照)。

割り込み関数の中のコードは何でも実行できますが、リアルタイム性能を向上させるには 3.5.7 「割り込みルーチンを高速にする方法を教えてください」を参照してください。

割り込み条件が発生する前に、周辺モジュールを正しく設定し、割り込みを有効にしておく必要があります(5.8.5 「割り込みの有効化」参照)。PIC18 の場合、適切な SFR に書き込む事によって割り込み要因の優先度を指定しておく必要があります。

3.3.7 アセンブリコード

以下では、C プロジェクトの一部としてアセンブリコードを書く際に生じる疑問に答えます。

- アセンブリコードと C コードを組み合わせる方法を教えてください
- アセンブリ ソースファイルには命令以外に何が必要ですか
- アセンブリコードから C オブジェクトにアクセスする方法を教えてください

[1] この指定子の元の名前は in-line でしたが、混乱を防ぐために変更されました。

- アセンブリコード内から SFR にアクセスする方法を教えてください
- アセンブリコードを書く際に考慮すべき事柄は何ですか

3.3.7.1 アセンブリコードと C コードを組み合わせる方法を教えてください

理想的には、全ての手書きアセンブリコードは、呼び出し可能な分離したルーチンとして書くべきです。そうする事で、コンパイラが生成するアセンブリコードと手書きアセンブリコード間での影響をある程度防ぐ事ができます。手書きコードは、分離したアセンブリ モジュールに収めた上で、プロジェクトに追加できます(5.11.1 「アセンブリ言語モジュールとの連携」参照)。

必要に応じて、アセンブリコードを C コード内に追加できます(5.11.2 「インライン アセンブリ」参照)。追加するアセンブリコードは nop、sleep、clrwdt 等のみに制限するのが理想的です。これらの命令の全てをインライン化するマクロが既に用意されています(9.2.19 「NOP マクロ」参照)。レジスタの内容やデバイスステートを変更するような複雑なインライン アセンブリは、十分に注意を払わないと期待以外の動作の原因となります。コンパイラによって使われるレジスタに関しては、5.6 「レジスタの使用」を参照してください。

3.3.7.2 アセンブリ ソースファイルには命令以外に何が必要ですか

アセンブリコードには、命令そのもの以外にアセンブラ ディレクティブが必要です。全てのディレクティブの動作は、6.1.9 「アセンブラ ディレクティブ」に記載しています。よく使われるディレクティブを以下に挙げます。

全てのアセンブリコードは 1 つの psect (セクション)内に格納する必要があります。そうする事で、リンカはアセンブリコードをひとまとめに扱ってメモリに配置できます。psect の全般的な情報は 5.14.1 「コンパイラが生成するpsect」 に記載しています。psect の作成および指定にはPSECT ディレクティブ(6.1.9.36 「PSECT ディレクティブ」参照)を使います。

GLOBAL ディレクティブ(6.1.9.21 「GLOBAL ディレクティブ」参照)もよく使います。これは、シンボルを複数のソースファイルにまたがってアクセスできるようにするためのディレクティブです。

3.3.7.3 アセンブリコードから C オブジェクトにアクセスする方法を教えてください

大部分の C オブジェクトにはアセンブリコードからアクセスできます。C ソース内のシンボルと、そのソースから生成されるアセンブリコード内のシンボルは、互いに対応付けられます。手書きアセンブリコードは、コンパイラが生成したアセンブリコード内の等価シンボルにアクセスする必要があります。詳細は 5.11.3.3 「等価アセンブリ シンボル」を参照してください。

GLOBAL アセンブラ ディレクティブを使う事で、そのシンボルが別のどこかで定義済みである事をアセンブラに知らせます(6.1.9.21 「GLOBAL ディレクティブ」参照)。型の情報はありませんが、これは C 言語の宣言と等価です。シンボルがアセンブリコードと同じモジュール内で定義されている場合、このディレクティブは不要であり、使うべきではありません。

アセンブリコードからアクセスされる全ての C 変数は、volatile で修飾されているかのように扱われます。 C コード内では volatile 修飾子を指定する事を推奨します。そうする事で、そのオブジェクトが外部コードからアクセス可能である事を明確に示せます。

3.3.7.4 アセンブリコード内から SFR にアクセスする方法を教えてください

アセンブリコードから SFR にアクセスするための最も安全な方法は、対応する SFR と同じアドレスを持つシンボルをアセンブリコード内で定義する事です。ユーザがこれらのシンボルを定義しなくて済むよう、本コンパイラと一緒にヘッダファイルが提供されます(詳細は 5.11.3.4 「アセンブリコードからのレジスタへのアクセス」参照)。

SFR にアクセスする C コードを書いても、そのコードのコンパイルによって生成されたシンボルにアクセスできるという保証はありません。

3.3.7.5 アセンブリコードを書く際に考慮すべき事柄は何ですか

アセンブリコードを手書きする場合、デバイスの特定機能の管理とアセンブリ命令およびオペランドのフォーマットは、ユーザの責任で行う必要があります。ユーザには以下の対応が求められます。

- RAM 変数にアクセスする場合、その位置を読み書きする前に変数のバンクを選択する必要があります。これには 1 つまたは複数のアセンブリ命令が必要です。実際のコードは、使用するデバイスと変数の格納位置に応じて異なります。オブジェクトが共通メモリ(PIC18 ではアクセスバンクと呼ぶ) 内に配置されている場合、またはフルアドレスを指定する命令(PIC18 の movff 命令等)を使う場合、バンクの選択は不要です。ターゲット デバイスのデータシートで、メモリの構成と、バンク選択を制御する命令とレジスタを確認してください。バンクを正しく選択しないと、期待した動作とならない可能性があります。BANKSEL 疑似命令を使うと、この手順を簡略化できます(6.1.1.2 「バンクとページの選択」参照)。
- バンクオフセットのみが命令のファイルレジスタ オペランドとして使われるよう、アクセス先の RAM 変数のアドレスを適切にマスクする必要があります。フルアドレスを指定する命令(PIC18 の movff 命令等)では、

このマスクは不要です。各命令が要求するアドレス オペランドについては、デバイスのデータシートで確認してください。アドレスを正しくマスクしないと、フィックスアップ エラー(3.6.8 「フィックスアップ オーバーフロー エラーへの対処方法を教えてください」参照)または期待以外の動作が発生する可能性があります。BANKMASK マクロによりアドレスを切り詰める事ができます(5.11.3.4 「アセンブリコードからのレジスタへのアクセス」参照)。

- いずれかのルーチンに対する呼び出しまたはジャンプを実行する前に、適切な命令を使ってそのルーチンのプログラムメモリ ページを選択しておく必要があります。これには PAGESEL 疑似命令(6.1.1.2 「バンクとページの選択」参照)が使える他、必要に応じて自動的にページ選択命令を追加する `fcall` または `ljmp` 疑似命令(PIC18 では不要、6.1.1.7 「ロングジャンプと呼び出し」参照) が使えます。
- ストレージとして使う RAM は予約しておく必要があります。C コード内で定義された変数にのみアクセスする場合、これはコンパイラによって予約されます。アセンブリコード内でのみ使われる全ての変数に対しては、適切なディレクティブ(DS (6.1.9.13 「DS ディレクティブ」参照)または DABS (6.1.9.8 「DABS ディレクティブ」参照)等)を使ってメモリを予約する必要があります。多くの場合、アセンブリコード内ではなく C コード内でオブジェクトを定義する方が容易です。
- 手書きアセンブリコードは全て psect 内に配置する必要があります。これを行うためのディレクティブについては6.1.9.36 「PSECT ディレクティブ」参照してください。psect については5.14.1 「コンパイラが生成するpsect」を参照してください。ユーザが定義するpsect にはフラグ (オプション) の指定が必要な場合があります。`delta`、`space`、`reloc`、`class` フラグには特に注意が必要です(6.1.9.36.4 「delta フラグ」、6.1.9.36.18 「space フラグ」、6.1.9.36.3 「class フラグ」参照)。これらのフラグを正しく設定しないと、ほぼ確実にコンパイラエラーまたは期待以外の動作が発生します。psect がクラスを指定し、その psect がそのクラスによって定義されたメモリレンジ内の任意位置に配置可能である場合(7.1.1 「A: リンカクラスの定義」参照)、psect をリンクするためのオプションを指定する必要はありません。これ以外の場合、リンカ オプションを使って psect をリンクする必要があります。psect の通常のリンク方法に関しては 7.1.17 「P: psect の配置」参照してください。リンカを直接実行する事なくこのオプションを指定する方法については 4.6.11.8 「Wl: リンカに対するオプションの指定」を参照してください。C コード内のインライン アセンブリコードは、コンパイラが生成したアセンブリコードと同じ psect 内に置かれます。これを別の psect へ移動してはいけません。
- アセンブリコード内でレジスタに書き込む場合、そのレジスタはコンパイラが生成したコードによって既に使われていない事が必要です。アセンブリコードを別のモジュール内で書く場合、既定値によりコンパイラは全てのレジスタがそれらのルーチンによって使われると想定するため、これはそれほど大きな問題になりません(5.6 「レジスタの使用」参照)。しかしインライン アセンブリの場合、コンパイラはそのような想定をしません(5.11.2 「インライン アセンブリ」参照)。従って、周囲のコンパイラ生成コードで既に使われているリソースをアセンブリコード内で使う(変更する)場合、それら全てのリソースを慎重に退避および復元する必要があります。

3.4 アプリケーションを設計意図通りの動作にする

以下ではプログラミング テクニック、アプリケーション、事例を紹介します。アプリケーションで特定のタスクを実行する際に生じる疑問にも答えます。

- 出力ポートでグリッチが発生する原因は何ですか
- psect を手動でリンクする場合、どこにリンクできますか
- ブートローダとダウンロード可能アプリケーションのリンク方法を教えてください
- デバッガを使うには、コンパイルする際に何をする必要がありますか
- リセットの直後にコードを実行させる方法を教えてください
- 割り込みとメインライン コードの間でデータを共有する方法を教えてください
- コードの悪用を防ぐ方法を教えてください
- Printf を使って周辺モジュールにテキストを出力する方法を教えてください
- コード内でオシレータを設定する方法を教えてください
- PIC18 で、変数を外部プログラムメモリ内に配置する方法を教えてください
- コードに遅延を実装する方法を教えてください
- 変数をローテートする方法を教えてください
- 起動時に変数がクリアされるのを防ぐ方法を教えてください
- 高書き込み耐性フラッシュを使ってコードではなくデータを保存する方法を教えてください

3.4.1 出力ポートでグリッチが発生する原因は何ですか

ほとんどの場合、グリッチは通常変数を使ってポートの一部のビットまたはポートそのものにアクセスする事によって発生します。これらの変数には `volatile` 修飾子を使う必要があります。

ポートに割り当てられた変数の保持値(つまりポートに実際に書き込まれる値) は、直接電気信号に変換されます。これらの変数が保持する値は、コードが変更を意図した時にのみ、1 回の遷移で現在の値から新しい値へ変化する事が不可欠です(5.3.8.2 「volatile 型修飾子」参照)。コンパイラは、1 動作で volatile 変数への書き込みを実行させようとします。

3.4.2 psect を手動でリンクする場合、どこにリンクできますか

コンパイラによって生成された psect (セクション) に対するリンカ オプションは変更しない事を推奨します。これらを変更する必要がある場合、または配置を指定する必要があるユーザ定義 psect が存在する場合、メモリ内のこれらの配置先はデバイスまたはコンパイラによって制限される場合があります。

psect を適切なコンパイラ リンカクラスにリンクする事を推奨します(5.14.2 「既定値リンカクラス」参照)。そうする事で、これらのメモリ領域の定義には全ての制約が考慮されます。必要に応じて、ユーザ独自のリンカクラスを定義する事もできます(3.3.3.3 「変数を特定のアドレス領域に配置する方法を教えてください」参照)。ユーザ独自の psect と類似する内容を保持するコンパイラ生成 psect に適用されるメモリ配置の制約については、5.14.1 「コンパイラが生成するpsect」を参照してください。

大部分の制約は、メモリ境界(データバンクまたはプログラムメモリ ページ)をまたぐ psect に関係します。代表的な制約として、実行可能コードを保持する全ての psect は、デバイスページ境界をまたいではいけません。変数を保持するコンパイラ生成 psect も、それらが生成されたデータバンクの中でリンクされる必要があります。従って、psect の最大サイズはこれらの境界によって制限されます。

3.4.3 ブートローダとダウンロード可能アプリケーションのリンク方法を教えてください

方法の細部はターゲット デバイスとプロジェクトの要件に応じて異なりますが、ブートローダを使うアプリケーションをコンパイルする際の一般的なアプローチでは、ブートローダとアプリケーションに別々のプログラムメモリ空間を割り当てます(それぞれに専用のメモリを持たせます)。そうする事で、アプリケーションの動作とブートローダの動作が互いに影響し合う事を防げます。これには、ブートローダまたはアプリケーションのどちらかをメモリ内でオフセットする必要があります。このために、リセットと割り込みのアドレスをアドレス 0 からオフセットし、全てのプログラムコードも同量だけオフセットします。

PIC18 の場合、通常はアプリケーション コードをオフセットし、ブートローダはオフセットせずにリンクします。従って、ブートローダはリセットおよび割り込みコードの位置に実装します。ブートローダのリセットおよび割り込みコードには内容がほとんどなく、アプリケーションが定義する本当の(そしてオフセット済みの)リセットおよび割り込みコードへ制御をリダイレクトします。

ミッドレンジ デバイスで割り込みを使う場合、通常これは不可能です。リセットに関連するコードを除くブートローダの全ての内容をオフセットする事を検討してください。これは常にブートローダによって定義される必要があります。アプリケーション コードは、割り込み位置でリンクされるコードを定義できます。ブートローダは、ブートローダによって定義されたリセットコードの上書きを試みる全てのアプリケーション コードを再配置する必要があります。

ベクタ割り込み(VIC)モジュールを使わないデバイスでは、-mcodeoffset オプション(4.6.1.3 「codeoffset オプション」参照) により、オフセット量を指定してプログラムコード (リセットとベクタを含む)を移動させる事ができます。加えて、このオプションは、アドレス 0 (リセットベクタ)からオフセット アドレスの間のプログラムメモリ領域がプログラムによって使われないように制限します。これらの予約された領域に何も残されていない事を常にマップファイル(7.3 「マップファイル」参照)で確認してください。

VIC モジュールを備えたデバイスでは (たとえレガシーモードで動作する場合でも)、IVTBASE レジスタを調整する事でハードウェア ベクタ位置を移動させる必要があります。

コマンドラインまたは MPLAB X IDE 内で HEX ファイルをプロジェクト ファイルとして追加する事により、ブートローダ向け HEX ファイルの内容をアプリケーションのコードにマージできます。これにより、1 つのイメージ内でブートローダとアプリケーション コードを含む 1 つの HEX ファイルが得られます。HEX ファイルは HEXMATE アプリケーション(8.2 「Hexmate」参照) によりマージされます。このアプリケーションからオーバーラップに関する警告が出力されないか確認してください。この警告が出力された場合、メモリがブートローダとダウンロード可能アプリケーションの両方によって使われている可能性があります。

3.4.4 デバッガを使うには、コンパイルする際に何をする必要がありますか

MPLAB XC8 コンパイラでビルドしたコードは、MPLAB ICD4 や PICkit 4 等のデバッガを使ってデバッグできます。これらのデバッガは、デバイスのデータメモリとプログラムメモリの一部をデバッグ用に使う可能性があります。従って、プログラムコードがこれらのデバッグ用リソースを使わないようにする事が重要です。

MPLAB X IDE では、デバッグビルドを実行する際に適切なメモリが予約されます。通常の Build Project または Clean and Build を実行する場合は、全てのデバイスメモリが利用できます。コマンドラインでビルドする場合、デバッガを使う事をコンパイラに知らせる必要があります。これには、コンパイラの-mdebugger オプションを使います(4.6.1.6 「debugger オプション」参照)。

MPLAB X IDE の docs/ ReservedResources ディレクトリ内に保存されている ReservedResourcesByDeviceFamilyAndTool.html ファイルにより、各デバイスが各種デバッガツール向けに使用する(-mdebugger オプションによって予約される)メモリとその他のリソースを確認できます。

デバイスメモリがデバッガによって使われる場合、プログラム向けに利用できるメモリ量は減少します。このため、デバッガを選択した時にコードまたはデータがデバイスに収まりきらなくなる可能性があります。デバッガツールをプログラマとしてのみ使う場合、ツールはデバイスメモリを要求しないため、-mdebugger オプションは不要です。

コンパイラが予約したリソースがデバッガのメモリ要求を満たしているかどうかは、以下により確認できます。

デバッガを選択せずにコンパイルした場合に生成されたマップファイルと、デバッガを選択してコンパイルした場合に生成されたマップファイルを保存します。それらのマップファイル内のリンカ オプションを比較し、-A オプション(7.1.1 「A: リンカクラスの定義」参照) の違いを見つけます。例えば、デバッガを選択しなかった場合に CODE クラス向けに定義されたメモリが-A オプションによって以下の通りに指定されたとします。

```
-ACODE=00h-0FFh,0100h-07FFh,0800h-0FFFhx3
```

ICD3 をデバッグとして選択した場合、このオプションは以下の通りに変更されます。

```
-ACODE=00h-0FFh,0100h-07FFh,0800h-0FFFhx2,01800h-01EFFh
```

これは、1F00～1FFF のメモリ領域がコンパイラによって削除された(従って、プログラムはこの領域を使えなくなった) 事を示しています。3.5.16 「予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか」も参照してください。

3.4.5 リセットの直後にコードを実行させる方法を教えてください

リセットベクタにリンクされる特別な「powerup」アセンブリコードを容易に追加できるようにするため、専用のフック が用意されています(5.9.3 「電源投入時コード」参照)。このコードが実行された後に実行時起動コードが実行され、その後にmain( )関数が実行されます(5.9 「メイン、スタートアップ、リセット」参照)。

3.4.6 割り込みとメインライン コードの間でデータを共有する方法を教えてください

割り込みとメインライン コードの両方からアクセスされる変数は、プログラムによって破壊されたり誤読されたりしやすくなります。volatile 修飾子(5.3.8.2 「volatile 型修飾子」参照)を使う事で、その変数に対して最適化を実行しないようコンパイラに指示できます。これにより、この問題をある程度解消できます。

もう 1 つの問題は、コンパイラ/デバイスがデータにアトミックにアクセスできるかどうかに関係します。8 ビット PIC デバイスでは、これが問題になる事はほとんどありません。アトミックなアクセスとは、1 命令で変数の全体にアクセスする事を意味します。このようなアクセスに割り込む事はできません。変数がアトミック アクセスできているかどうかは、コンパイラが生成したリストファイルのアセンブリコードを調べる事で分かります(6.3 「アセンブリ リストファイル」参照)。1 命令で変数にアクセスしていれば、それはアトミックです。変数へのアクセス方法はステートメントに応じて様々です。通常、これらの問題を完全に防ぐ最善の方法は、メインライン コードで変数にアクセスする前に割り込みを無効にし、アクセス後に割り込みを再度有効にする事です(5.8.5 「割り込みの有効化」参照)。

3.4.7 コードの悪用を防ぐ方法を教えてください

フラッシュ プログラムメモリを備えた多くのデバイスでは、このメモリの全域または一部の領域を書き込み保護できます。書き込み保護を有効にするには、デバイス コンフィグレーション ビットを正しく設定する必要があります(5.2.5 「コンフィグレーション ビットへのアクセス」とデバイス データシート参照)。

また、プログラムメモリの未使用領域を未プログラム状態のまま残さずに値を書き込んでおく事で、第三者が未使用メモリ領域にコードを書き込む事を防げます。未使用領域には、何らかの命令に対応する値を書き込んでも、全てのビットをセットしても構いません。そうする事で、それらの値は変更できなくなります(作り込みの方法として、これらのデータへのアクセス結果次第でオリジナルとは動作が異なる実行となるようにします)。

コンパイラの HEXMATE ユーティリティ(8.2 「Hexmate」参照)を使うと、未使用位置に書き込む事ができます。この機能は、コマンドライン ドライバ オプションを使って要求できます(4.6.11.10 「fill オプション」参照)。HEXMATE は HEX ファイル専用であるため、この機能は HEX/COF ファイル出力を生成する場合にのみ利用できます(これは既定値動作です)。

ライブラリファイルまたは中間生成 p-code ファイルを他者に提供する必要がある場合、-mshroud オプションを使ってファイルを難読化する事により、オリジナルのソースコードを読めなくできます(4.6.1.21 「shroud オプション」参照)。

3.4.8 Printf を使って周辺モジュールにテキストを出力する方法を教えてください

printf()関数は以下の 2 つの機能を有します。

1. ユーザが指定した書式文字列と変換指定子に基づいてテキストを書式化する
2. この書式化されたテキストを出力先(またはストリーム) へ送信(出力)する

詳細は 5.10.2 「スマート IO コード」と『Microchip Unified Standard Library リファレンスガイド』を参照してください。

printf()関数は全ての書式化を実行した後に、ヘルパー関数 putch()を呼び出して書式化済みテキストの各バイトを送信します。putch()関数をカスタマイズする事で、printf()を使ってデータを任意の周辺モジュールまたは任意の場所へ送信できます。printf の出力先として LCD、SPI モジュール、USART 等が選択できます。

putch()関数のスタブは、コンパイラの「pic/sources」フォルダ内に保存されています。これをユーザのプロジェクトにコピーし、渡された 1 バイトパラメータを必要な出力先へ送信するように変更します。printf()を使う前に、送信先の周辺モジュールを通常の方法で初期化しておく必要があります。ミッドレンジ PIC デバイスでの USART 向け putch()の例を以下に示します。

```
void putch(char data) {
  while( ! TXIF) // check buffer
    continue;     // wait till ready
  TXREG = data;   // send data
}
```

グローバル変数を使って出力先を選択する事により、printf()を使ってデータを各種の出力先へ送信できます。putch()関数は、この変数の値に基づいて、いずれかの出力先へデータバイトを送信します。

3.4.9 コード内でオシレータを設定する方法を教えてください

全ての PIC デバイスは各種のオシレータモードを備えており、プロジェクト内でデバイスのコンフィグレーションビットに書き込む事によりモードを選択する必要があります。オシレータモードについては、デバイスのデータシートを参照してください。コンフィグレーション ビットの設定方法については、5.2.5 「コンフィグレーション ビットへのアクセス」を参照してください。

一部のデバイスは OSCCON レジスタを備えており、これを使って実行時の属性(クロック源、内部クロック周波数等)を制御できます。このレジスタには、C ソースコード内で通常の方法により書き込む事ができます。

一部のデバイスでは、OSCTUNE レジスタを使って実行時に内部オシレータの動作を調整できます。その他のデバイスでは、デバイスに書き込み済みの値を使って、内部オシレータの校正を実行できます。コンパイラが生成する実行時起動コード(5.9.2 「実行時起動コード」参照) は、既定値によりオシレータの校正を実行します。この機能は、オプション(4.6.1.15 「osccal オプション」参照) により無効にできます。

コンパイラの一部のビルトイン遅延関数を使う場合、_XTAL_FREQ マクロを使って、コンパイラにシステム周波数を知らせる必要があります。このマクロはデバイスの動作周波数に全く影響を及ぼしません(9.2.14「delay_ms ビルトイン関数」参照)。

3.4.10 PIC18 デバイスで、変数をプログラムメモリ内に配置する方法を教えてください

読み出し専用変数をプログラムメモリ内に配置するだけでよいのであれば、それらの変数を const として修飾します(5.4.3 「プログラム空間内のオブジェクト」参照)。変数を外部プログラムメモリ内に配置する場合は、__far 修飾子(5.3.9.3 「far 型修飾子」参照)を使い、-mram オプションでメモリを指定します(4.6.1.17 「ram オプション」参照)。コンパイラは、__far 修飾子付き変数の変更を許可します。これらの変数へのアクセスには、内部データメモリ内の変数にアクセスするよりも長い時間を要する事に注意が必要です。外部メモリへのアクセスモードは、オプション(4.6.1.9 「emi オプション」参照)で指定できます。

3.4.11 コードに遅延を実装する方法を教えてください

正確な遅延が必要な場合または遅延期間中に他のタスクを実行する必要がある場合、最善の方法はタイマを使って割り込みを生成する事です。

コード内でこれらが問題なく可能である場合、コンパイラのビルトイン delay 疑似関数(_delay()、__delay_ms()、__delay_us())が使えます(9.2.9 「_delay ビルトイン関数」参照)。これらの全ては、インラインアセンブリ命令へと展開されるか、指定されたサイクル数または時間を費やす命令の(ネストされた)ループへと展開されます。引数(遅延サイクル数または時間) は定数で指定する必要があり(変数および関数呼び出しは不可)、遅延の長さは 50,463,240 サイクル未満である事が必要です。__delay_ms()または__delay_us()を使う場合、プリプロセッサ マクロ_XTAL_FREQでデバイスのクロック周波数を正しく設定しておく必要があります。

これらのコードシーケンスは、nop 命令および/またはループを形成する命令のみを使います。これらの疑似関数のウォッチドッグ バージョン(例: _delaywdt()) は、clrwdt 命令も使えます。

3.4.12 変数をローテートする方法を教えてください

C 言語にはローテートのための演算子はありませんが、シフトおよびビット単位 OR 演算子を使ってローテートできます。PIC デバイスはローテート命令を備えるため、コンパイラはシフトと OR を使ってローテーションを表現する事を試み、そのように表現したローテート命令を出力コード内で使います(5.5.2 「ローテート」参照)。

3.4.13 起動時に変数がクリアされるのを防ぐ方法を教えてください

__persistent 修飾子(5.3.9.5 「persistent 型修飾子」参照)を使う事で、その変数を実行時起動コードによってクリアされない別の psect 内に配置します。

3.4.14 高書き込み耐性フラッシュを使ってコードではなくデータを保存する方法を教えてください

プログラムメモリ空間に高書き込み耐性フラッシュメモリを実装しているデバイスでは、コンパイラがフラッシュメモリ領域を実行可能コード向けに使わないように予約する必要があります(3.3.3.8 「特定のメモリ領域をコンパイラが使わないようにする方法を教えてください」参照)。デバイスがこのメモリを実装しているかどうかは、デバイスデータシートに記載されています。

3.5 コンパイル プロセスを理解する

以下では、ビルドプロセス中にコンパイラが何を行ったか(どのように出力コードをエンコードしたか、オブジェクトをどこに配置したか等)を調べる方法について説明します。また、コンパイラがサポートする機能についても解説します。

- コンパイラのライセンスの有無によって何が異なるのですか
- コードを小さくする方法を教えてください
- RAM の使用量を削減する方法を教えてください
- コードを高速にする方法を教えてください
- プログラミング時間を短縮する方法を教えてください
- コンパイラは全ての内容をどのようにメモリに配置するのですか
- 割り込みルーチンを高速にする方法を教えてください
- C 変数はどこまで大きくできますか
- デバイスの RAM バンクの使用/割り当て方法を教えてください
- 拡張ミッドレンジ PIC デバイスでリニアメモリを使う方法を教えてください
- コンパイラはどのデバイスをサポートしますか
- コンパイラが生成したコードを詳しく調べる方法を教えてください
- 関数のサイズを調べる方法を教えてください
- 各関数で使われているリソースを調べる方法を教えてください
- 変数と関数の位置を調べる方法を教えてください
- 予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか
- メモリの残量を調べる方法を教えてください
- プロジェクト内でライブラリ ファイルを使う方法を教えてください
- コンパイラはどの最適化レベルを適用しますか
- デバッガを選択するとメモリが不足します。どうしてですか
- コンパイラが関数に対してどのスタックモデルを割り当てたのか調べる方法を教えてください
- コンフィグレーション ビットまたは ID ロケーションに書き込まれた値を調べる方法を教えてください
- プロジェクトのチェックサムが変更されないようにする方法を教えてください

以下も参考にしてください。

- 警告/エラーメッセージの意味を調べる方法を教えてください
- MPLAB X IDE のデバッグビルドでは何が違うのですか
- 未使用の関数が削除されてしまう事を防ぐ方法を教えてください
- ライブラリのビルド方法を教えてください

3.5.1 コンパイラのライセンスの有無によって何が異なるのですか

ライセンスがあると、より高度なコードの最適化が可能となり、コンパイラが生成するコードをより小さく、より高速にできます。ライセンスがなくても、最適化を除く全ての機能(対応ターゲット デバイス、デバイスメモリの使用等) が使えます。

3.5.2 コードを小さくする方法を教えてください

これには各種の方法がありますが、その結果はプロジェクトごとに異なります。コンパイラが生成したアセンブリコードに対して以下の対策が適用可能かどうかをアセンブリ リストファイル(6.3 「アセンブリ リストファイル」参照)で確認します。

できるだけ小さなサイズのデータ型を使う事で、それらにアクセスするコードを小さくします(これにより RAM の使用量も減少します)。本コンパイラでは、1 ビット型と非標準の 24 ビット整数型が利用できます。可能な限りマルチビットのビットフィールドを使わないようにします。これらにアクセスするコードは非常に長くなる可能性があります。5.3 「サポートするデータ型と変数」には全てのデータ型とサイズを記載しています。

サイズの異なる 2 種類の浮動小数点型が存在しますが、可能な限り浮動小数点型を使わずに固定小数点算術演算コードを書くよう努めます。

できるだけ符号なし型を使います。特に、異なる型とサイズが混在する式では、これを心掛けます。演算子が作用するオペランドのサイズは、可能な限り同じに揃えるよう常に心掛けます。

ループまたは条件付きコードを使う場合、以下のように強い(命令との結びつきによる)停止条件を使います。

```
for(i=0; i!=10; i++)
```

上記は以下よりも推奨されます。

```
for(i=0; i<10; i++)
```

通常、「等しいかどうか」(== または !=)を判別する方が、「より小さいかどうか」(<)を比較するよりも効率的に実装できます。

場合によっては、0 に向かってデクリメントするループカウンタを使った方が、同じ繰り返し数で 0 からカウントアップするよりも効率的です。これは、特にループインデックスが 1 バイト幅型である場合に顕著となります。その場合、上記は以下のように書き換える事ができます。

```
for(i=10; i!=0; i--)
```

関数パラメータの指定順序を、最初にバイトサイズのパラメータを指定するように変更する事で、コードを少し小さくできる場合があります。最初のパラメータがバイトサイズである場合、パラメータ転送にはレジスタが使われます。

例:

```
char calc(char mode, int value);
```

上記を以下に変更

```
char calc(int value, char mode);
```

全ての最適化機能を有効にします(4.6.6 「最適化を制御するためのオプション」参照)。コンパイラが何を最適化してくれるのか理解する事で、コンパイラによる最適化を上手に活用し、コンパイラが最適化してくれる C コードをわざわざ手作業で最適化するといった時間の無駄を省けます(5.12 「最適化」および 6.2 「アセンブリレベルの最適化」参照)。例えば、4 の乗算演算を 2 ビットのシフト演算に変更する必要はありません。この種の最適化はコンパイラが検出してくれます。

3.5.3 RAM の使用量を削減する方法を教えてください

できるだけ小さなサイズのデータ型を使います。これにより、それらにアクセスするためのコードのサイズも小さくなります。本コンパイラでは 1 ビット型と非標準の 24 ビット整数型(__int24 と__uint24) が利用できます。5.3 「サポートするデータ型と変数」には全てのデータ型とサイズを記載しています。これには、サイズの異なる 2 種類の浮動小数点型が含まれます。

global または static 変数の代わりに auto 変数を使うよう心掛けます。同時にアクティブにならない auto 変数同士はメモリの割り当てを共有できるため RAM の使用量を削減できる可能性があります。auto 変数のメモリ割り当てはコンパイルド スタックで行います(5.4.2.2 「自動記憶期間オブジェクト」参照)。

関数との間で大きなオブジェクトを授受するよりも、それらのオブジェクトを参照するポインタを授受した方が効率的です。これは特に大きな構造体を渡す場合に当てはまりますが、long 型変数を渡す場合にも RAM の節約効果が得られる可能性があります。

プログラム全体を通して変更する必要のないオブジェクトは、const 修飾子を使ってプログラムメモリ内に配置できます(5.3.8.1 「const 型修飾子」と 5.4.2 「データメモリ内のオブジェクト」参照)。これにより貴重な RAM を節約できますが、実行は遅くなります。

全ての最適化機能を有効にします(4.6.6 「最適化を制御するためのオプション」参照)。コンパイラが何を最適化してくれるのか理解する事で、コンパイラによる最適化を上手に活用し、コンパイラが最適化してくれる C コードをわざわざ手作業で最適化するといった時間の無駄を省けます(5.12 「最適化」参照)。この種の最適化はコンパイラが検出してくれます。

3.5.4 コードを高速にする方法を教えてください

実行速度はコードのサイズに大きく依存します。従って多くの場合、コードサイズの削減を図る事で、実行時間を短縮できます(2.6.2 「コードを小さくする方法を教えてください」と 2.6.7 「割り込みルーチンを高速にする方法を教えてください」参照)。しかし、シーケンスによっては、コードサイズが増えても高速化できる場合があります。

レベル O3 の最適化を選択した場合、コンパイラはプログラムの実行時間の削減を優先して最適化を行います(プログラムのサイズは増加する可能性があります)。これに対し、レベル Os の最適化は純粋にコードサイズを削減するよう設計されています(4.6.6 「最適化を制御するためのオプション」参照)。

一部のライブラリ乗算ルーチンは、オペランドの 1 つの値が小さいほど高速になります(詳細は 5.2.9 「乗算」を参照してください)。

一般的に、実行速度の向上に最も大きく寄与するのがプロジェクトで使うアルゴリズムです。プログラム内で高速化が必要なセクションを見極める必要があります。配列を線形探索しているループがあれば、別の探索方法(ハッシュテーブル、関数等) に変更します。結果を再計算している箇所があれば、それらをキャッシュできないか検討します。

3.5.5 プログラミング時間を短縮する方法を教えてください

リンカは、プログラムメモリの両端からセクションを配置できます。低アドレス端から高アドレス側に向かって配置されるセクションもあれば、高アドレス端から低アドレス側に向かって配置されるセクションもあります。これはコードの動作に影響を及ぼさず、リンクも容易ですが、デバイスメモリ空間の全体に広がる HEX ファイルが生成される可能性があります。この HEX ファイルをデバイスにプログラミングするには長い時間を要する可能性があります。

このような場合にプログラミング時間を短縮するには、デバイスのプログラムメモリの全てを使わないようにリンカに指示します。-mreserve オプションを使う事で、プログラムメモリの上位部分を予約します(4.6.1.18 「メモリ予約オプション」参照)。

3.5.6 コンパイラは全ての内容をどのようにメモリに配置するのですか

ほとんどの場合、アセンブリ命令とコードおよびデータの両方に関係するディレクティブは、複数のセクション (psect と呼ぶ) にグループ分けされて、デバイスのメモリ種別ごとに異なるコンテナに配置されます。このプロセスの概要は 5.14.1 「コンパイラが生成するpsect」に記載しています。例外として、絶対アドレス指定された変数(5.4.4 「絶対アドレス変数」参照)は、定義時に指定されたアドレスに配置され、psect 内には配置されません。

3.5.7 割り込みルーチンを高速にする方法を教えてください

割り込みコードのコードサイズの削減方法に関しては、3.5.2 「割り込みルーチンを高速にする方法を教えてください」を参考にしてください。 多くの場合、コードは小さいほど高速です。

コンパイラは、ISR 内に書かれたコードに加えて、コンテクストを切り換えるためのコードを生成します。これは、割り込みの発生直後とリターン直後に実行され、割り込み処理時間にはこれらの実行時間も含まれます(5.8.4 「コンテクスト スイッチ」参照)。通常このコードは、ISR 内で使うレジスタだけを保存するよう最適化されます。従って、ISR 内で使われるレジスタの数が少ないほど、コンテクスト スイッチコードの実行時間も減少します。コードが複雑になるとレジスタの使用が増加するため、複雑な命令文や複雑なコードを含む関数への呼び出しは避けるべきです。割り込みコード内でコンパイラがどのレジスタを使っているかは、アセンブリ リストファイルで調べる事ができます(6.3 「アセンブリ リストファイル」参照)。

ミッドレンジ デバイスはコンパイラによって使われるレジスタを数個しか備えておらず、コンテクスト スイッチコードはほとんど存在しません。一部のデバイスは、コンテクストを自動的にシャドーレジスタに保存するため、コンパイラが生成するコンテクスト スイッチコードはさらに削減されるか完全に除去されます(5.6 「レジスタの使用」参照)。

ISR 内ではフラグをセットするだけでリターンし、メインライン コード内でフラグをチェックする事によって割り込みを処理するよう配慮します。このようにして複雑な割り込み処理コードを ISR の外に出す事で、ISR によるレジスタの使用を防ぐ事ができます。割り込みコードとメインライン コードで共有する変数には必ず volatile 修飾子(5.3.8.2 「volatile 型修飾子」参照)を付けます(3.4.6 「割り込みとメインライン コードの間でデータを共有する方法を教えてください」参照)。

3.5.8 C 変数はどこまで大きくできますか

これは個々の C オブジェクト(配列、構造体等) のサイズに関する質問です。全ての変数の総サイズの事ではありません。

この質問に答えるには、その変数がどのメモリ空間に配置されるのか知る必要があります。const 修飾付きの静的記憶期間オブジェクトはプログラムメモリ内に配置され、その他のオブジェクトはデータメモリ内に配置されます。プログラムメモリ内のオブジェクトのサイズについては、5.4.3.1 「オブジェクト サイズの制限」を参照してください。データメモリ内のオブジェクトは、大まかに auto 変数と非 auto 変数に分類されます。これらのサイズについては、5.4.2.1.2 「オブジェクト サイズの制限」と 5.4.2.2.1 「オブジェクト サイズの制限」を参照してください。

3.5.9 デバイスの RAM バンクの使用/割り当て方法を教えてください

本コンパイラは、ターゲット デバイス内で利用可能な全ての RAM バンクを自動的に使います。既定値のメモリ割り当てを変更する場合にのみ、ユーザによる操作が必要です。その場合、特殊な bank( )修飾子(5.3.9.1 「bank 型修飾子」参照)とこれらの修飾子の解釈方法を指定するオプション(4.6.1.1 「addrqual オプション」参照)を使って、手動で変数を割り当てます。

デバイス上の全てのメモリがセクション(psect) に格納されたデータおよびコードとして使える保証はありません。

3.5.10 拡張ミッドレンジ PIC デバイスでリニアメモリを使う方法を教えてください

リニアアドレス指定モードは、拡張ミッドレンジ デバイスのバンク型データメモリに 1 つの連続したリニアなブロックとしてアクセスするための手段です(5.4.1 「アドレス空間」参照)。使用するデバイスがこのメモリを実装しているかどうかと、動作の詳細はデータシートに記載されています。

リニアメモリの使用は完全に自動的です。1 つのデータバンクよりも大きなオブジェクトも通常の方法で定義でき、それらにはリニアドレス指定モードを使ってアクセスします(5.4.2.1.2 「オブジェクト サイズの制限」参照)。メモリ内の特定位置で絶対アドレス指定のオブジェクトを定義する場合、通常のバンク型アドレス指定の他に、必要に応じてリニアドレス指定も使えます(5.4.4.1 「データメモリ内の絶対アドレス オブジェクト」参照)。

3.5.11 コンパイラはどのデバイスをサポートしますか

通常、コンパイラ リリースのたびに新しいデバイスのサポートが追加されます。コンパイラがどのデバイスをサポートしているかは、以下を含む各種の方法で調べる事ができます(5.2.1 「サポートするデバイス」も参照)。

- 本コンパイラの docs フォルダ内にある HTML ファイル(pic_chipinfo.html と pic18_chipinfo.html)をウェブブラウザで開く
- コマンドラインで-mprint-devices オプション(4.6.2.8 「print-devices」参照)を指定してコンパイラドライバを実行する事によりディスプレイに全てのデバイスを含むリストを表示する

3.5.12 コンパイラが生成したコードを詳しく調べる方法を教えてください

アセンブリ リストファイル(6.3 「アセンブリ リストファイル」参照)は、プログラムのほぼ完全なアセンブリ出力を示します。これには、プログラムにリンクされたライブラリ ルーチンと実行時起動コードが含まれます(5.9.2 「実行時起動コード」参照)。コマンドラインを使う場合、オプション-Wa,-a を指定すると、アセンブリ リストファイルが生成されます(4.6.10 「アセンブラ オプション」参照)。アセンブリ リストファイルの拡張子は「.lst」です。

このリストファイルはアセンブリ命令、一部のアセンブリ ディレクティブ、プログラムに関する情報(コールグラフ、ポインタ参照グラフ、各関数の情報等)を示します。アセンブリ オプティマイザが有効である場合、一部のアセンブリ ディレクティブ(中間アセンブリ ファイル内で生成されるディレクティブ) はリストファイルに表示されません。コンパイラが生成する全てのアセンブリ ディレクティブを見る必要がある場合、アセンブリ オプティマイザを一時的に無効にします(4.6.6 「最適化を制御するためのオプション」参照)。

3.5.13 関数のサイズを調べる方法を教えてください

関数のサイズを含む情報はマップファイルで提供されます。このファイルの最後の方にあるヘッダ「MODULE INFORMATION」を見てください。この情報については、7.3.2.8 「モジュール情報」で説明します。

3.5.14 各関数で使われているリソースを調べる方法を教えてください

アセンブリ リストファイルには、ライブラリ関数を含む各 C 関数の情報が含まれています(6.3 「アセンブリ リストファイル」参照)。この情報は各関数が使うレジスタ、各関数が呼び出す関数(これはコールグラフ(6.3.6 「コールグラフ」参照) にも示される)、各関数が要求するデータメモリのバイト数を示します。関数が使う auto 変数、パラメータや一時変数は、他の関数で使われる同様の変数とで共通するメモリアドレスとオーバーラップして使われる可能性があります。これは、これら変数がコンパイラによって確保されたコンパイルド スタックエリアに確保されるためです(5.4.2.2 「自動記憶期間オブジェクト」参照)。

3.5.15 変数と関数の位置を調べる方法を教えてください

変数と関数の配置先は、アセンブリ リストファイル(6.3 「アセンブリ リストファイル」参照)またはマップファイル(7.3 「マップファイル」参照)で調べる事ができます。マップファイルには静的記憶期間オブジェクトに関係するシンボルのみが示されます。アセンブリ リストファイルには、そのリストファイル内に示されるコードの全てのシンボル(自動記憶期間オブジェクトを含む) が示されます。各アセンブリ モジュールは独自のリストファイルを持ちます。

C 言語上のシンボルとアセンブリコード内のシンボルは互いに対応付けされており、これらのシンボルはマップファイルとリストファイルの両方で示されます(5.11.3.3 「等価アセンブリ シンボル」参照)。変数に対応するシンボルには、その変数の最下位バイトのアドレスが割り当てられます。関数に対しては、その関数向けに生成された最初の命令のアドレスが割り当てられます。

3.5.16 予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか

メモリ予約オプション(2.4.3.6 「関数を独自のセクションに配置する方法を教えてください」参照)は、リンカによって使われるクラスに割り当てるアドレスのレンジを調整します。大部分の変数と関数は、これらのクラスのアドレスレンジ内にリンクされたセクション(5.14.1 「コンパイラが生成するpsect」参照) 内に配置されるため、メモリ予約オプションの影響を受けます。

一部のセクションは、アドレスレンジ内の任意位置にリンクされるのではなく明示的に特定アドレスに配置されます。例えば、リセット時に実行されるコードを保持するセクションは常にアドレス 0 (8 ビットデバイス向けに定義されているリセット位置) にリンクされます。このようなセクションは、`-mrom` オプションによってメモリアドレス 0 が予約されていても影響を受けません。リセットと割り込みに関連するコードを保持するセクションは、`-mcodeoffset` オプション(4.6.1.3 「codeoffset オプション」参照)を使ってシフトさせる事ができます。

アセンブリ リストファイル(6.3 「アセンブリ リストファイル」参照)を調べる事で、オブジェクトとコードを保持しているセクションの名前を確認できます。マップファイル(7.3 「マップファイル」参照) 内のリンカ オプションを調べる事で、psect が明示的にリンクされているのかそれともクラス内の任意位置にリンクされているのかを判別できます。リンカ オプション`-p` (7.1.17 「P: psect を配置する」)と`-A` (7.1.1 「A: リンカクラスの定義」)も参照してください。

3.5.17 メモリの残量を調べる方法を教えてください

コンパイル後にコンパイラが出力するメモリ使用量サマリと MPLAB X IDE 内のメモリ表示はどちらもメモリの使用量と残量を示しますが、そのメモリが連続した 1 ブロックなのか、それとも複数の小さなチャンクに分割されているのかは分かりません。小さなブロックに分割された空きメモリは、大きなサイズのオブジェクト用には使えません。このため、オブジェクトに対して空きメモリの総量が明らかに十分であっても、メモリ不足エラーが発生する場合があります(3.6.6 「「Can't find space...」エラーへの対処方法を教えてください」 参照)。

マップファイルには、各リンカクラス内に残されている利用可能メモリが明確に示されます(7.3.2.5 「未使用アドレスレンジ」参照)。また、メモリにバンクまたはページ境界が存在する場合、そのクラス内で最大の連続したブロックも示されます。

3.5.18 プロジェクト内でライブラリ ファイルを使う方法を教えてください

コンパイル時に、コンパイラは自動的に全ての適用可能標準ライブラリをビルドプロセスにインクルードします。ユーザがこれらのファイルを制御する必要はありません。ユーザ独自のライブラリ ファイルをビルドする方法については、3.2.5 「ライブラリのビルド方法を教えてください」を参照してください。

ユーザが独自にビルドした 1 つまたは複数のライブラリ ファイルを使う場合、コマンドラインで、コンパイルするファイルのリストにそれらのファイルを含める必要があります。ライブラリ ファイルは、ソースファイル向けファイルリスト内の任意位置で指定できます。しかし複数のライブラリ ファイルが存在する場合、それらはコマンドラインで指定された順番に検索されます。

例:

```
xc8-cc -mcpu=16f1937 main.c int.c lcd.a
```

MPLAB X IDE を使ってプロジェクトをビルドする場合、プロジェクト内で表示される「`Libraries`」フォルダに 1 つまたは複数のライブラリ ファイルを追加します。それらのファイルは、追加された順番に検索されます。IDE は、

ビルド シーケンス中の適切な時点で、それらのファイルをコンパイラに渡します。

3.5.19 コンパイラはどの最適化レベルを適用しますか

最適化は C 言語レベルとアセンブリ レベルの両方のコンパイル時に適用されます。これらの最適化については、5.12 「最適化」と 6.2 「アセンブリレベルの最適化」で説明します。最適化を制御するためのオプションについては、4.6.6 「最適化を制御するためのオプション」で説明します。

3.5.20 デバッガを選択するとメモリが不足します。どうしてですか

ハードウェア ツール デバッガ(MPLAB REAL ICE または ICD 3 等)を使う場合、オンボード デバッグ エグゼクティブを格納するためのメモリリソースが PIC デバイス上で必要となります。MPLAB X IDE プロジェクト内でデバッガを選択してデバッグビルドを実行する場合、デバイス上のメモリの一部がデバッグ用に使われます(プログラム用に使えなくなります)。このため、プログラムがメモリの大部分を使い果たしている場合、「can't find space」メモリエラーが発生する可能性があります。

3.5.21 コンパイラが関数に対してどのスタックモデルを割り当てたのか調べる方法を教えてください

アセンブリ リストファイル内の関数情報セクション(6.3.3 「関数情報」参照)を調べます。各関数の情報の最終行に、その関数に対して使われるスタックモデル(再入可能または再入不可)が示されます。

3.5.22 コンフィグレーション ビットまたは ID ロケーションに書き込まれた値を調べる方法を教えてください

それらの値を調べるには、アセンブリ リストファイルで`#pragma config` ディレクティブの出力を確認します。下の例は、書き込まれた設定値が `0x1F` である事を示しています。このファイルには、その値が何を意味するのかも示されます。

```
;Config register CONFIG2H @ 0x300003
;       Watchdog Timer Enable bit
; ...
;       WDTPS = 0xF, unprogrammed default
psect   config
org 3145731
db 31
```

3.5.23 プロジェクトのチェックサムが変更されないようにする方法を教えてください

ユーザが MPLAB X IDE または Hexmate (8.2 「Hexmate」参照)を使ってビルドしたプロジェクトのチェックサムは、コンパイラが出力したファイルから計算されます。チェックサムを計算するアルゴリズムは、出力ファイルが少しでも変更されると異なるチェックサム値が生成されるように設計されています。チェックサムはプロジェクトのソースコードから計算されるのではありません。ビルドするたびにチェックサムが変更されないようにするには、コンパイラの出力が変化しないようにする必要があります。

以下の場合、ビルド時のコンパイラ出力は変化します(従ってプロジェクトのチェックサムも変化します)。

- コンパイラ バージョンが異なる
- コンパイラ オプションが異なる
- プロジェクトのソースコード、ヘッダファイル、ライブラリコードが異なる
- ソースファイルまたはライブラリがコンパイルされる順序が異なる
- `__DATE__`および`__TIME__`等、ビルドした時の時刻に応じて出力が変化するマクロがソースコードで使われている
- ソースファイルの保存場所が事なる(`__FILE__`等、ソースファイルの保存場所に応じて出力が変化するマクロがソースコードで使われている場合)

ツール(Hexmate、MPLAB X IDE 等)ごとにチェックサム アルゴリズムが異なる場合、コンパイラ出力が同じであってもツールが異なるとチェックサムの計算結果が異なります。そのような状況は稀にしかありませんが、コンパイラと IDE のリリースノートでツールのアルゴリズムが変更されていないか確認してください。

3.6 動作に問題のあるコードの修正

以下では、プロジェクトがコンパイラエラーによってビルドできない、あるいはビルドできても期待通りに動作しないといった問題について説明します。

- 警告/エラーメッセージの意味を調べる方法を教えてください
- プログラム内でエラーまたは警告の原因となっているコードを見つける方法を教えてください
- 不要な警告が生成されないようにする方法を教えてください
- LED を点滅させる事すらできません。どうしてですか
- ハードウェア スタックのオーバーフローを防ぐ方法を教えてください
- 「Can't find space...」エラーへの対処方法を教えてください
- 「Can't generate code...」エラーへの対処方法を教えてください
- フィックスアップ オーバーフロー エラーへの対処方法を教えてください
- 割り込み使用時の変数の破損と異常動作の原因は何ですか

3.6.1 警告/エラーメッセージの意味を調べる方法を教えてください

10. 「エラーおよび警告メッセージ」には、各警告/エラー メッセージの説明と、そのようなエラー/警告の原因となり得るコードの例を記載しています。

3.6.2 プログラム内でエラーまたは警告の原因となっているコードを見つける方法を教えてください

ソースコードに関連する構文エラーが発生した場合、通常はコンパイラからのメッセージにコード行が示されます(4.5.1 「メッセージの概要」参照)。MPLAB IDE 内でコンパイルしている場合、メッセージをダブルクリックすると、エディタは問題のある行へジャンプします。しかし、問題のコードがいつも簡単に見つかるとは限りません。

場合によっては、注目すべき行の次の行でエラーが報告される場合もあります。これは、C 言語の命令文はソースファイル内で複数行にまたがる事があるからです。コンパイラが次の命令文のスキャンを始めるまでエラーの存在を検出できないといった状況が起こり得ます。

例:

```
input = PORTB   // oops - forgot the semicolon
if(input>6)
// ...
```

代入文でセミコロンが抜けている事は、次の行(`if( )`文)で報告されます。

エラーがコード ジェネレータではなくアセンブラに起因する場合もあります。C ソースファイルから生成されたアセンブリコード内に問題がある場合、コンパイラはその問題箇所に対応する C ソースファイル内の行を示そうとします。ソースがアセンブリ モジュールであれば、エラーメッセージはエラーの原因となったソース アセンブリコード内の行を直接示す事ができます。どちらの場合も、エラーメッセージ内の情報は、C コードではなくアセンブリコード内の何らかの問題に関係しているという事に注意が必要です。

最後に、C ソースコード内のどの特定行にも関係しないエラーもあります。コンパイラ オプション内のエラーまたはリンカエラーがこれに該当します。プログラム内で定義されている変数の数が多すぎる場合、エラーを引き起こす特定の行は存在しません。これはプログラム全体の問題だからです。コードがエラーの直接的原因でなくても、直近に処理されたファイルとソースの名前および行番号が表示される場合があるという事に注意が必要です。

エラーまたは警告を生成したアプリケーションを特定するには、10. 「エラーおよび警告メッセージ」を参照してください。各メッセージの冒頭に表示されるカッコ内の右側に、そのメッセージを生成したアプリケーションの名前が表示されます。エラーを生成したアプリケーションを知る事で、問題の解析が容易になります。コンパイラ アプリケーションの名前は 4.2 「コンパイル シーケンス」に記載しています。コンパイラが生成したアセンブリコードは、アセンブリ リストファイル(6.3 「アセンブリ リストファイル」参照)で見る事ができます。リンカがオブジェクトをどこに配置しようとしたかは、マップファイル(7.3 「マップファイル」参照)を見ると分かります。

3.6.3 不要な警告が生成されないようにする方法を教えてください

警告は異常動作を生じる可能性がある状況を示します。多くの場合、警告が報告されてもコードは有効です。コードがエラーの潜在的原因とならい事を確認した後に、各種の方法で警告が出力されないようにする事ができます。

- 警告レベルしきい値を調整する事で、特定の重要度を持つ警告のみ出力させる事ができます(4.5.3.1 「メッセージの無効化」参照)。
- 指定した ID を持つ全ての警告を無効にできます。
- 場合によっては、プラグマを使ってソースコードの特定行に関する警告を無効にする事もできます(5.13.3.11 「#pragma warning ディレクティブ」参照)。

3.6.4 LED を点滅させる事すらできません。どうしてですか

TRIS レジスタを設定してポートに値を書き込むだけの単純なプログラムでも、動作を妨げる各種の要因が存在します。

- デバイスのコンフィグレーション レジスタが正しく設定されている事を確認します(5.2.5 「コンフィグレーション ビットへのアクセス」参照)。また、これらのレジスタ内の全てのビットが既定値状態のままではなく明示的に指定されている事も確認します。全ての設定機能は、デバイスのデータシートに記載されています。例えば、オシレータソースを指定するコンフィグレーション ビットの設定が正しくない場合、デバイスクロックが動作すらしない事もあります。
- 内部オシレータを使っている場合、コンフィグレーション ビットに加えて、オシレータの周波数とモードを設定するために初期化が必要な SFR があるかもしれません(5.2.7 「C コードからSFR を使う」とデバイス データシート参照)。
- デバイスがリセットしないようにするため、コンフィグレーション ビットでウォッチドッグ タイマを OFF にするか、コード内でウォッチドッグ タイマをクリアします。デバイスがリセットしている場合、プログラム内の LED を点滅させるコード行まで到達できません。テストプログラムが動作するようになるまで、デバイスリセットの原因となり得る他の全ての機能を OFF にします。
- ポートビットに対応するデバイスピンは、しばしば他の周辺モジュールと多重化されています。例えば、1 本のピンをポート内の 1 つのビットに接続する事も、アナログ入力として使う事も、コンパレータ出力として使う事もできます。LED に接続したピンが使用するポートに内部で接続されていなければ、LED は期待通りに動作しません。デバイス データシート内のポート機能テーブルには各ピンの全ての用途が記載されているため、ピンを共有している周辺モジュールを特定するために役立ちます。
- 「読み出し-変更-書き込み」問題が生じない事を確認します。デバイスが独立した「ラッチ」レジスタを備えていない場合(一部のミッドレンジ PIC デバイスがこれに該当)、この問題が発生する可能性があります。特にポート出力が大きな負荷(LED 等)を駆動する場合、この問題は発生しやすくなります。1 つのビットをセットすると別の LED が消灯する等、異常な動作が生じる可能性があります。これを防ぐには、一時変数を使って独自のラッチを作成し、ポートを直接読み書きするのではなくラッチ変数を変更し、ラッチの変更が完了した後に、ラッチ全体をポートへコピーします。そうする事で、ポートを変更するためにそのポートを読み出さなくて済みます。詳細はデバイスのマニュアルを参照してください。

3.6.5 ハードウェア スタックのオーバーフローを防ぐ方法を教えてください

8 ビット PIC デバイスは、関数(および割り込み関数) の戻りアドレス向けにのみ使われるハードウェア スタックを備えています(5.2.4 「スタック」参照)。関数呼び出しおよび割り込みのネストが深すぎる場合、スタックはオーバーフロー (ロールオーバーして以前のエントリを上書き)します。その後(時には呼び出しシーケンス中にかなり時間が経ってから)、破損した戻りアドレスにコードが達した時点で問題が発生します。

コンパイラはスタックの深さを追跡し、必要に応じて、ハードウェア スタックを必要としない呼び出し方法に切り換えます(PIC10/12/16 デバイスのみ)。ユーザは、スタックが明らかにオーバーフローした場合のコンパイラの動作をある程度制御できます(4.6.1.23 「stackcall オプション」参照)。

コールグラフは、コンパイラが検出した呼び出しの階層構造と深さを示します。コールグラフはアセンブリ リスト ファイル内に示されます。コールグラフについては 6.3.6 「コールグラフ」参照してください。

コンパイラはプログラムの実行時の挙動を監視する事はできないため、最悪ケースを想定した場合にオーバーフローが発生する可能性がある事をユーザに知らせる事しかできません。プログラム全体が C 言語で書かれている場合、コンパイラがオーバーフローを見逃す事はありません。しかし、コンパイラはスタックを使うアセンブリコードを考慮しないため、注意が必要です。

3.6.6 「Can’t find space...」エラーへの対処方法を教えてください

このメッセージには各種のバリエーションが存在しますが、いずれもデータまたは命令のブロックを配置するための空間が不足している事を示します。十分なメモリが残されているように見えても、ページやバンク等によってメモリが分割されているためにこのメッセージが出力される場合があります。これらの各メッセージの詳細は 10. 「エラーおよび警告メッセージ」を参照してください。

3.6.7 「Can’t generate code...」エラーへの対処方法を教えてください

このメッセージは、コンパイラがどのようにしても C 表現をコンパイルする事ができなかった場合に出力されます 10. 「エラーおよび警告メッセージ」を参照してください。通常、このメッセージはコード内の問題を示しません。コードをコンパイルできない原因は、コンパイラに問題がある場合もあれば、コードに問題がある場合もあります(例: コードが利用可能な量を超えるレジスタまたはリソースを要求した場合等)。ベースライン デバイスでは、この問題が生じる可能性は高くなります。いずれの場合も、通常は表現をシンプルにしたり長い命令文を複数の短い命令文に分割する事により、コンパイルが可能となります。表現が複雑である場合、追加の変数を使って中間結果を保持する事が必要になるかもしれません。

3.6.8 フィックスアップ オーバーフロー エラーへの対処方法を教えてください

シンボルに割り当てられたアドレスがアセンブリ命令のアドレス フィールドに対して大きすぎる場合、フィックスアップ (シンボルによる参照を実際のアドレスに置き換えるリンカ動作)がオーバーフローする可能性があります。大部分の 8 ビット PIC アセンブリ命令は、ファイルアドレス(現在選択中のバンク内のオフセット)を指定します。これらの命令でマスクなしの完全なアドレスが指定された場合、リンカは大きなアドレス値を命令向けにエンコードできないため、フィックスアップ オーバーフロー エラーが発生します。例えば、ミッドレンジ デバイスの命令は 0〜0x7F のファイルアドレスしか許容しません。しかし、RAM のデータバンクを 4 個備えたデバイスの場合、変数のアドレスレンジは 0〜0x1FF となります。

例えば、 アドレス 0x1D0 に配置される変数のシンボルがこれらの命令で指定された場合、シンボルがその最終値で置き換えられた時に、この値は命令のアドレス フィールドに収まらなくなります。

多くのジャンプおよび呼び出し命令も、デスティネーション オペランドとして切り詰められたアドレスを使います。PIC18 では `call` および `goto` 命令にフルアドレスが使えますが、分岐および相対呼び出し命令には使えません。これらの命令に対するデスティネーション ラベルがマスクされない場合、フィックスアップ エラーが発生する可能性があります。

フィックスアップ処理は、命令だけでなくアセンブラ ディレクティブのオペランドに対しても適用されます。従って、ディレクティブに対するオペランドがオーバーフローした場合もフィックスアップ エラーが発生する可能性があります。

例えばシンボルがリンカによって値 0x238 に確定した場合:

```
DB error
```

上記のディレクティブは 1 バイト値を期待するため、フィックスアップ オーバーフロー エラーが生成されます。

ほとんどの場合、フィックスアップ エラーは手書きのアセンブリコードによって発生します。アセンブリコードを書く際は、デスティネーションのバンクまたはページを選択するための命令を追加し、その命令で使われるアドレスをマスクする必要があります(2.4.7.5 「アセンブリコードを書く際に考慮すべき事柄は何ですか」参照)。

場合によっては、C コードから生成されたアセンブリコードによってフィックスアップ オーバーフロー メッセージが生成される事もあります。通常これは、レンジを超えたジャンプに関連します。C コードのスイッチ命令文が長くなりすぎると、このメッセージが生成される可能性があります。コンパイラが生成した psect のリンク方法が変更された場合にも、フィックスアップ オーバーフローが発生する可能性があります。なぜなら、新しい psect の位置がコンパイラによる想定と矛盾する可能性があるからです。

重要な事として、これはアセンブリ命令での問題であり、問題のある命令を見つけない限りコンパイルを続ける事はできません。問題のある命令を追跡する詳細な方法は、10. 「エラーおよび警告メッセージ」内の対応するエラー番号を参照してください。

3.6.9 割り込み使用時の変数の破損と異常動作の原因は何ですか

通常これは、割り込みとメインライン コードの両方によって使われる変数が存在する事によって発生します。コンパイラが変数へのアクセスを最適化する場合、またはアクセスが割り込みルーチンによって割り込まれる場合、変数が破損する恐れがあります。詳細は 3.4.6 「割り込みとメインライン コードの間でデータを共有する方法を教えてください」を参照してください。

4. コマンドライン ドライバ

MPLAB XC8 C コンパイラのコマンドライン ドライバ(xc8-cc) により、C コードの生成/アセンブリ/リンクを含む全てのコンパイル手順を実行できます。このコマンドライン ドライバを使ってコンパイラを起動する事を推奨します。そうする事で、複雑な内部アプリケーションを気にする事なく全てのコンパイル手順を一貫したインターフェイスを介して容易に実行できます。IDE からコンパイルする場合でも、IDE は最終的に xc8-cc を呼び出します。

レガシー プロジェクトをビルドする場合または旧式のコマンドライン ドライバとコマンドライン オプションを使う場合は、xc8 ドライバ アプリケーションが使えます。その使用方法は『MPLAB® XC8 C コンパイラ ユーザガイド』に記載されています。この文書には C90 に則したコンパイル方法も記載されています。

以下では、コンパイル中にドライバが実行する手順、ドライバによって処理および生成可能なファイル、コンパイラの動作を制御するためのコマンドライン オプションについて説明します。

4.1 コンパイラの起動

以下では、コマンドラインで xc8-cc を起動する方法と、コンパイラに渡す事ができる入力ファイルについて説明します。

4.1.1 ドライバのコマンドライン フォーマット

xc8-cc ドライバを使って C ソースおよびアセンブリ ソースファイルに加えてリンク オブジェクト ファイルとライブラリ アーカイブをコンパイル/アセンブルする事で、最終的なプログラム イメージを生成します。

本ドライバ向けの基本的コマンド フォーマットを以下に示します。

```
xc8-cc [options] files
```

C ソースファイル hello.c をコンパイル/リンクする場合のコマンド例を以下に示します。

```
xc8-cc -mcpu=16F877A -O2 -o hello.elf hello.c
```

本書では、コンパイラ アプリケーションがコンソールの検索パス内に保存されていると想定しています。検索パスを指定するための環境変数については、4.1.2 「ドライバの環境変数」を参照してください。検索パスを設定しない場合、ドライバを呼び出すには完全なパスとドライバ名を指定する必要があります。

通常はファイル名の前で *options* (先頭に「-」を付けて識別)を指定しますが、必須ではありません。

コマンドライン オプションでは大文字と小文字が区別されます。オプションの詳細は 4.6 「オプションの説明」に記載しています。xc8-cc 向けコマンドライン オプションの多くは全ての MPLAB XC コンパイラで共通しているため、デバイスおよびコンパイラ間の移植が容易です。

files では、複数のファイル(C ソースおよびアセンブラ ソースファイル、再配置可能オブジェクト ファイル、アーカイブ ファイル)を一緒に並べて指定できます。これらのファイルの指定順序はプログラムの動作に影響を及ぼしませんが、コードまたはデータの配置には影響を及ぼす可能性があります。アーカイブ ファイルは指定された順序で検索されるため、それらのファイルの指定順はプログラムへリンクされるモジュールに影響を及ぼす可能性があります。

また、一部の出力ファイルのベース名は-o オプションを使って明示的に指定しない限り、xc8-cc コマンドラインで最初に指定された C ソースファイルのベース名に基づきます。

make システムを使ってコードをビルドする場合、特殊な中間生成 p-code ファイル フォーマットを熟知している必要があります(4.2.3 「マルチステップの C コンパイル」参照)。プロジェクト内でアセンブリ ソース モジュールが使われない限り、MPLAB XC8 C コンパイラでオブジェクト ファイルを使う事はほとんどありません。

4.1.1.1 長いコマンドライン

xc8-cc ドライバには、ドライバ オプションと引数を格納したコマンドライン ファイルを渡す事ができます。これを利用する事で、オペレーティング システムによるコマンドラインの長さに関する制限を回避できます。

コマンドライン ファイルは、引数を格納したファイルの名前の直前に(スペースを挟まずに)シンボル「@」を付ける事により指定します。コンパイラ ドライバから呼び出される大部分の内部アプリケーションに対しても、これと同じ方法で引数を渡す事ができます。

コマンドライン ファイル内では、各引数を 1 つまたは複数のスペースで区切る必要があります。また、バックスラッシュ-リターンを使って複数行に展開できます。コマンドライン ファイルには空白行を格納できます(空白行は無視されます)。

コマンドライン ファイルの例を以下に示します(ファイル名は xyz.xc8)。これはテキスト エディタを使って書かれ、プロジェクトのコンパイルに必要なオプションとファイル名を含みます。

```
-mcpu=16F877A -Wl,-Map=proj.map -Wa,-a \
-O2 main.c isr.c
```

このファイルを保存しておけば、以下のコマンドを使ってコンパイラを呼び出せます。

```
xc8-cc @xyz.xc8
```

コマンドライン ファイルは make ファイルおよびユーティリティの代わりとして容易に使う事ができ、コンパイラ オプションとソースファイル名を保存しておく事ができます。MPLAB X IDE でコマンドライン ファイルを使う事もできます。このファイルの名前は、[Project Poperties] 内の *XC8 Linker > Additional options > Use response file to link* フィールドで指定します。

4.1.2 ドライバの環境変数

本コンパイラは、実行のための環境変数を定義も要求もしません。

PATH 環境変数を設定しておく事で、完全なコンパイラ パスを指定する事なくコンパイラドライバを呼び出せます。

コンパイラのインストール時に、適切なダイアログ内で**[Add xc8 to the path environment variable]**チェックボックスを選択する事により、この環境変数を自動的に更新できます。

PATH 変数によって指定されたパスは、コンパイラ ドライバ向けにのみ使われます。ドライバが起動した後は、ドライバ自身が内部コンパイラ アプリケーション (アセンブラ、リンカ等)へのアクセスを管理します。

IDE を使う場合、通常は PATH 変数を定義しなくてもプロジェクトのプロパティ内でコンパイラを選択できます。

4.1.3 入力ファイルのタイプ

xc8-cc ドライバは、単純にファイルのタイプ(拡張子) に基づいてソースファイル、中間生成ファイル、ライブラリファイルを識別します。表4-1 に示すファイル拡張子が認識されます(大文字と小文字は区別)。

表 4-1. 入力ファイルのタイプ

拡張子	ファイルのタイプ
.c	C ソースファイル
.i	前処理済み C ソースファイル
.p1	p-code 中間生成ファイル
.s	アセンブリ ソースファイル
.S	前処理が必要なアセンブリ ソースファイル
.as または .asm	アセンブリ ソースファイル
.o	再配置可能オブジェクト ファイル
.a	アーカイブ (ライブラリ) ファイル
.hex	Intel HEX 形式ファイル

本コンパイラにはソースファイルのベース名に関する制約はありませんが、ホスト オペレーティング システムによる制約(大文字/小文字の区別、名前の長さ、その他) には注意が必要です。

アセンブリ ソースファイルと C ソースファイルには、たとえそれらが異なるフォルダに格納されていても、同じベース名を使ってはいけません。例えば、プロジェクトが init.c という名前の C ソースファイルを含む場合、init.s という名前のアセンブリ ソースファイルをプロジェクトに追加してはいけません。また、IDE プロジェクトと同じベース名をそのプロジェクト内のソースファイルに使ってはいけません。

MPLAB XC8 では、アセンブリ ソースファイルに対して拡張子「.as」または「.asm」が使えますが、他の XC コンパイラではこれらの拡張子は使えません。アセンブリ ソースファイルには拡張子「.s」を使う事を推奨します。アセンブルの前に前処理が必要なアセンブリ ソースファイルには拡張子「.S」を使うか、拡張子「.s 」を使ってオプション「-x」を指定する事ができます。

「ソースファイル」と「モジュール」は同じ意味で使われる事が多いですが、本書ではコンパイル シーケンス中の異なる段階でのソースコードを指す意味で使い分けます。

本書では、プログラムの全部または一部が書かれたファイルを「ソースファイル」と呼びます。これらのファイルは C コード、プリプロセッサ ディレクティブ、命令を格納します。ソースファイルは、コンパイラ ドライバによって内部でプリプロセッサに渡されます。

本書では、与えられたソースファイルに対するプリプロセッサの出力を「モジュール」と呼びます。モジュールは、#include プリプロセッサ ディレクティブによって指定された全てのヘッダファイルがインクルードされ、処理後にその他のプリプロセッサ ディレクティブ (デバッグ向けの一部のコマンドを除く)が除去された後にプリプロセッサから出力されます。従って、モジュールは 1 つのソースファイルと複数のヘッダファイルを 1 つにまとめた物であり、プリプロセッサからその他のコンパイラ アプリケーションに渡されます。モジュールは、「変換ユニット (translation unit)」とも呼ばれます。

これらの用語はアセンブリ ソースファイルに対しても適用されます。アセンブリ ソースファイルを前処理して他の.inc ファイルをインクルードする事により、アセンブリ モジュールが生成されます。

4.2 コンパイル シーケンス

プロジェクトをコンパイルするために、ドライバは各種の内部アプリケーションを呼び出します。以下では、これらの内部アプリケーションを紹介します。また、特に複数のソースファイルを含むプロジェクトにおいて、内部アプリケーションがどのようにビルドプロセスに関与するのか説明します。make システムを使ってプロジェクトをビルドする場合、これらの情報が特に役立ちます。

4.2.1 コンパイラ アプリケーション

主な内部コンパイラ アプリケーションとファイルを図 4-1 に示します。

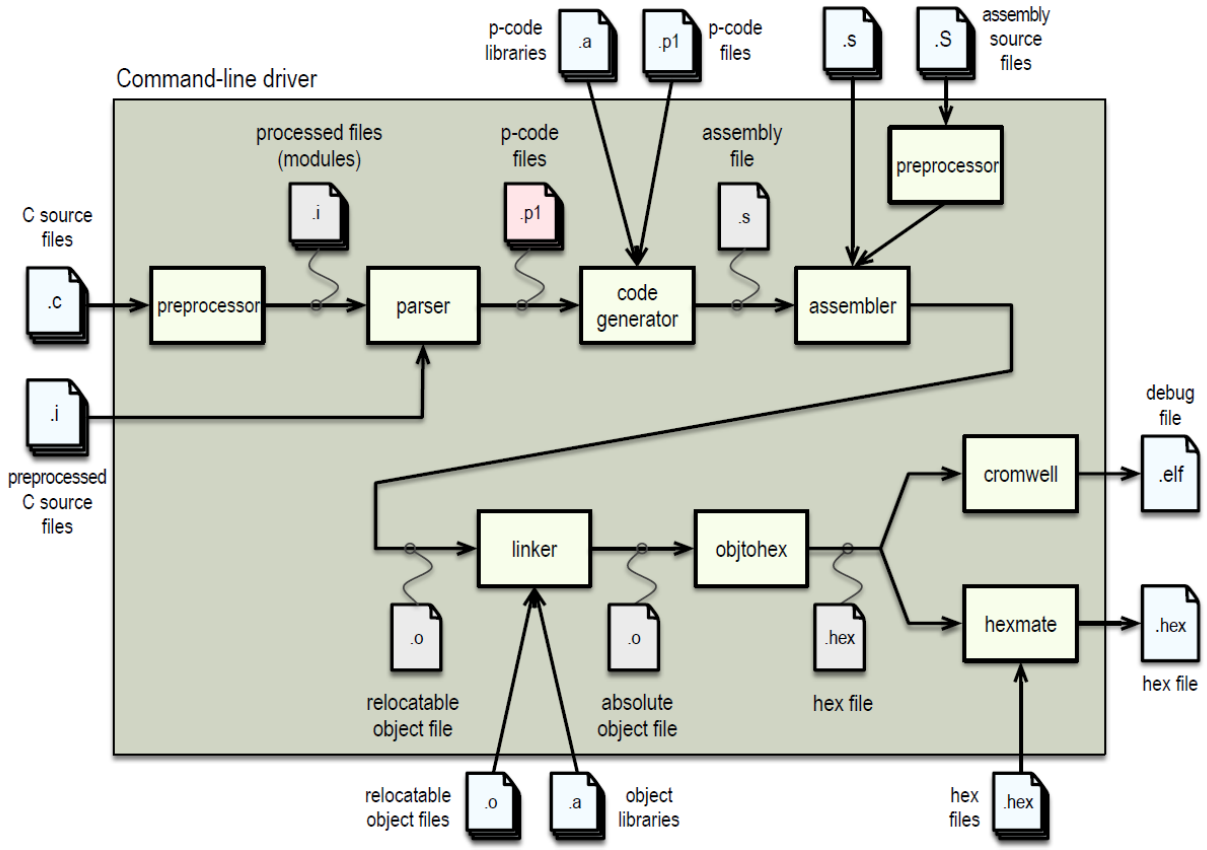
大きな緑のボックスがコンパイラを表します。コンパイラは、コマンドライン ドライバ xc8-cc により制御されます。C ソースファイル(図の左端)がコンパイラに渡され、出力ファイル(図の右端にある HEX ファイルと ELF デバッグファイル)が生成されます。その過程において、コンパイラの内部で多数のアプリケーションが使われ、各種の一時ファイルが生成されます。コンパイラの内部動作を理解する事は必ずしも必要ではありませんが、ツールを使いこなす助けとなります。

ドライバは、必要な時に必要なコンパイラ アプリケーションを呼び出します。これらのアプリケーションはコンパイラの「bin」フォルダに格納されており、図 4-1 ではドライバ(緑のボックス) の中の小さな黄のボックスとして示されています。

C99 規格向けにコンパイルする場合、clang フロントエンド アプリケーションがプリプロセッサとパーサー アプリケーションの両方の役目を果たします。

この図には、コンパイル シーケンス中に各アプリケーションによって生成される一時ファイルも示されています。C ソースコードの中間生成ファイルは赤のボックスで示されています。これらの一時ファイルの一部は、コンパイルが終了した後も削除されずに残されます。ドライバ オプションを使って特定のアプリケーションの実行後にコンパイルを一時停止させる事により、そのアプリケーションの出力をファイルに残して調べる事ができます。

図 4-1. コンパイラの内部アプリケーションとファイル



内部アプリケーションの直接実行は、hexmate およびアーカイバ(xc8-ar) に対してのみ推奨します。コマンドライン オプションについては 8. 「ユーティリティ」を参照してください。

4.2.2 シングルステップの C コンパイル

xc8-cc ドライバを使うと、1 つのコマンドだけで 1 つまたは複数の C ソースファイルを完全にコンパイルできます(リンク処理を含む)。

4.2.2.1 1 つの C ファイルをコンパイルする方法

1 つの C ソースファイルを含むプログラムをコンパイルおよびリンクする方法を示すため、2 つの値を加算する単純な C プログラムとして以下のコードを使います。このコードをテキストエディタにコピーし、テキストファイル(ex1.c)として保存します。

```
#include <xc.h>

unsigned int
add(unsigned int a, unsigned int b)
{
  return a + b;
}

int
main(void)
{
  unsigned int x, y, z;
  x = 2;
  y = 5;
  z = add(x, y);

  return 0;
}
```

分かりやすくするため、このコードはデバイス コンフィグレーション ビットを指定せず、加算以外は何もしません。

端末でプロンプトに以下のコマンドを入力する事で、プログラムをコンパイルします。この例では、以下を想定します。
- 端末の作業ディレクトリは上記のソースファイルを格納したフォルダへ移動済みである事
- コンパイラは既定値のフォルダにインストール済みであり、そのディレクトリはホストの検索パスに含まれている事

```
xc8-cc -mcpu=16F877A ex1.c
```

このコマンドは、ソースファイル(ex1.c)を PIC16F877A デバイス向けにコンパイルし、出力を ex1.elf に書き込みます。このファイルは、開発環境によって使われます。

ドライバは、直近のビルドコマンドの後にソースファイルが変更されたかどうかに関係なくソースファイルをコンパイルします。インクリメンタル ビルドを行うには、開発環境と make ユーティリティを使う必要があります(4.2.3 「マルチステップの C コンパイル」)。

何も指定しない場合、HEX ファイルと ELF ファイルが最終出力として生成されます。

コンパイルの完了後も中間生成ファイルは残されますが、-save-temps オプション(4.6.5.5 「save-temps オプション」参照)を指定しない限りその他の一時ファイルの大部分は削除されます。このオプションを指定すると、実行時起動ファイルを除く全ての生成ファイルが保存されます。IDE を使ってビルドする場合、一部の生成ファイルはプロジェクト ソースファイルとは異なるフォルダに保存される可能性があります(4.6.2.3 「O: 出力ファイルの指定」参照)。

4.2.2.2 複数の C ファイルをコンパイルする

以下では、複数の C ソースファイルを含むプロジェクトを 1 度にコンパイル/リンクする方法について説明します。以下のコードを「add.c」という名前のテキストファイルにコピーします。

```
/* add.c */
#include <xc.h>

unsigned int
add(unsigned int a, unsigned int b)
{
  return a + b;
}
```

以下のコードを「ex1.c」という名前の別のテキストファイルにコピーします。

```
/* ex1.c */
#include <xc.h>

unsigned int add(unsigned int a, unsigned int b);

int
main(void) {
  unsigned int x, y, z;
  x = 2;
  y = 5;
  z = add(x, y);

  return 0;
}
```

分かりやすくするため、このコードはデバイス コンフィグレーション ビットを指定せず、加算以外は何もしません。以下をコマンドラインに入力する事で、両方のファイルをコンパイルします。

```
xc8-cc -mcpu=16F877A ex1.c add.c
```

この命令は ex1.c と add.c を一度にコンパイルします。コンパイルされたモジュールは関連するコンパイラ ライブラリとリンクされ、実行ファイル ex1.elf が生成されます。

4.2.3 マルチステップの C コンパイル

マルチステップのコンパイル手順を使って、1 つまたは複数の C ソースファイルを含むプロジェクトをビルドできます。make ユーティリティではこの手順が使えます。このユーティリティは直近のビルド以後にどのソースファイルが変更されたか認識し、必要なファイルだけをコンパイルする事でコンパイル時間を短縮します。インクリメンタル ビルドは統合開発環境によっても実行されます。

通常 make ユーティリティはコンパイラを複数回呼び出します。すなわち、コンパイルの第 1 ステップでは各ソースファイルに対して 1 回ずつ呼び出して中間生成ファイルを出力し、第 2 ステップではそれらの中間生成ファイルをリンクして最終出力を生成するために 1 回呼び出します。

直近のビルド以後に 1 つのファイルだけが変更された場合、未変更のソースファイルから中間生成ファイルを生成する必要はありません。

以下の例では、make ユーティリティを使ってファイル ex1.c と add.c をコンパイルします。make ユーティリティを使ってこれらのファイルをコンパイルする場合のコマンドラインの例を以下に示します。

```
xc8-cc -mcpu=16F877A -c ex1.c
xc8-cc -mcpu=16F877A -c add.c
```

```
xc8-cc -mcpu=16F877A ex1.p1 add.p1
```

第 1 ステップ(最初の 2 つのコマンド)で使われている-c オプションは、指定されたファイルを中間生成ファイル フォーマットにコンパイルしますが、リンクはしません。第 2 ステップ(3 つ目のコマンド) により、中間生成ファイルがリンクされて最終的な出力(ex1.elf) が生成されます。コンパイルの第 2 ステップを実行する前に、プロジェクトを構成する全てのファイルが生成済みである事が必要です。

Note: C ソース向け中間生成ファイルとして常に p-code ファイル(拡張子は.p1)を使う必要があります。

上記の例では、コマンドライン ドライバ xc8-cc を使って最終的なリンク手順を実行しています。リンカ アプリケーション(hlink)を直接呼び出す事はできますが、コマンドが複雑であるため推奨しません。リンカ アプリケーションを直接実行する際は、上記のドライバ オプションではなくリンカ オプションを指定する必要があります。

ユーザ独自のライブラリ アーカイブ ファイルを生成する事を目的として中間生成ファイルを生成する場合もあります。

関連リンク

7.「リンカ」

4.2.4 アセンブリ ソースのコンパイル

C プロジェクトに含まれるアセンブリ ソースファイルは、C ソースファイルと同様の方法でコンパイルします。コンパイラ ドライバは、アセンブリ ソースファイルを認識して適切な内部コンパイラ アプリケーションに渡します。このため、1 つのビルドコマンド内で C ソースファイルとアセンブリ ソースファイルを一緒に指定する事ができます。

```
xc8-cc -mcpu=16F877A proj.c spi.s
```

アセンブラに渡す前に前処理が必要な(すなわち C プリプロセッサ ディレクティブを含んだ)アセンブリ ファイルには、拡張子.S (大文字)を付ける必要があります(例: spi.S)。

-S オプションを指定する事で、コンパイラを使って C ソースコードからアセンブリ ファイルを生成できます。そのアセンブリ出力をベースにしてユーザ独自のアセンブリ ルーチンを作成し、コマンドドライバを使ってそれらをコンパイルできます。

MPLAB XC8 は、C ソースファイルをビルドする前にアセンブリ ソースファイルをビルドします。このため、アセンブリコードに含まれる情報を後続のコード ジェネレータに渡す事ができます(5.11.3 「アセンブリコードと C コードの間の相互作用」参照)。

アセンブリ ソースファイルからは、中間生成ファイルとしてオブジェクト ファイル(拡張子は.o) が出力されます。-c オプション(4.6.2.1 「C: 中間生成ファイルへのコンパイル」参照)を指定すると、アセンブリ ソースファイルのビルドが終了してオブジェクト ファイルが生成された時点でコンパイルが停止します。

4.3 ランタイム ファイル

コンパイラは、コマンドラインで指定された C およびアセンブリ ソースファイルとユーザ定義ライブラリに加えて、コンパイラが生成したソースファイルとプリコンパイル済みライブラリファイル(以下の内容を含む)をプロジェクトにリンクさせる事ができます。

- C 標準ライブラリ ルーチン
- 暗黙的に呼び出される算術ライブラリ ルーチン
- 実行時起動(RTS)コード

4.3.1 ライブラリ ファイル

C 標準ライブラリは、標準化された関数のセット(文字列、算術、入出力ルーチン等)を含みます。これらの関数の使用方法と動作については 9.1 「ライブラリ サンプルコード」で説明します。ユーザ独自ライブラリの作成および使用方法の詳細は 5.10 「ライブラリ」を参照してください。

これらのライブラリは、異なるオプションの組み合わせで複数回ビルドされます。アプリケーションをコンパイル/リンクするためにコンパイラ ドライバを呼び出すと、ドライバは同じオプションを使ってビルドされた適切なターゲット ライブラリを選択します。通常、標準ライブラリの検索パスを指定する必要はなく、手動でライブラリ ファイルをプロジェクトにインクルードする必要もありません。

4.3.1.1 標準ライブラリのパスと名前

コンパイラは、ユーザが選択した C 言語規格に基づいて、標準ライブラリを `pic/lib/c99` または `pic/lib/c90` ディレクトリ内で検索します。

標準ライブラリの名前は *`family-type-options`*`.a` です。

- *`family`* : PIC18 デバイスに対して `pic18`、その他の全ての 8 ビット PIC デバイスに対して `pic`
- *`type`*: ライブラリ機能のタイプを表す(標準ライブラリ機能に対して `stdlib`、固有機能に対して `intlib` を指定)
- *`options`*: 各種のコンパイラ オプションまたはモードに対応するためにライブラリのバリアントを示す名前をハイフンで区切って指定(例: `htc` = MPLAB XC8 で使う C 言語の既定値設定、`d32` = 32 ビット double 型、`sp` = コードサイズの最適化)。

例えば 24 ビット double 型を使うベースラインおよびミッドレンジ デバイス向け標準ライブラリの場合、`pic-stdlib-d24-sz.a` と指定します。

4.3.2 起動と初期化

実行時起動コードは、C プログラム内の `main()`関数の実行前に必要な初期化タスクを実行します。このコードが実行するタスクについては5.9 「メイン、スタートアップ、リセット」を参照してください。

コンパイラは、選択されたターゲット デバイスと各種コンパイラ オプションに基づいて適切な実行時起動コードを選択します。

4.3.2.1 実行時起動コードの生成

従来方式のコンパイルではプリコンパイル済みの実行時起動コードをリンクする必要がありましたが、MPLAB XC8 C コンパイラはユーザ プログラムに必要な実行時起動コードを自動的に決定し、ユーザがプロジェクトをビルドするたびにこのコードを生成します。

ドライバの既定値動作では起動モジュールは保存されますが、オプション`-mno-keep-startup` (4.6.1.13 「keep-startup オプション」参照)を使ってこのモジュールを削除させる事ができます。MPLAB X IDE を使ってプロジェクトをビルドする場合、**[Project Properties]** 内で指定しない限り、このファイルは削除されます。

削除されなかった起動モジュールは、`startup.s` という名前で現在の作業ディレクトリに保存されます。IDE を使ってコンパイルを実行する場合、保存先は IDE によって指定されます。**「default」**コンフィグレーションが選択されている場合、MPLAB X IDE はこのファイルをプロジェクト フォルダ内の `dist/default/production` (製品向けビルドの場合)または `dist/default/debug` (デバッグビルドの場合) に保存します。

実行時起動コードの生成は自動的に処理されるためユーザからの介入は不要ですが、必要に応じてオプション`-Wl,--no-data-init` を指定する事で一部の動作を制御できます。このオプションの使い方は、4.6.12 「リンカオプション」に記載しています。

実行時起動コードは `main()`の前に実行されますが、リセット直後に特殊な初期化が必要な場合、5.9.3 「電源投入時コード」に記載した機能を使う必要があります。

4.4 コンパイラ出力

コンパイラは、コンパイル中に各種のファイルを生成します。これらの多くはコンパイル完了後に削除される一時ファイルまたは中間生成ファイルです。しかし、一部のファイルは削除されずにデバイスのプログラミングまたはデバッグ用に使われます。また、オプションを指定してコンパイルの中間処理を一時停止させる事で中間生成ファイルを削除せずに残して、それらの内容を調べる事ができます。

Note: 本書では、「プロジェクト名」は IDE 内で作成されたプロジェクトの名前またはコマンドライン上で指定された最初の C ソースファイルのベース名(拡張子を除いたファイル名)を指します。

4.4.1 出力ファイル

一般的な出力ファイルの型と拡張子(大文字/小文字を区別)を表 4-2 に示します。

表 4-2. 出力ファイル

拡張子	ファイル型	生成方法
.elf	Dwarf デバッグ情報を含む ELF (Executable and Linkable Format)ファイル	-o オプション
.cof	COFF (Common Object File Format)ファイル	-gcoff オプション
.p1	p-code ファイル(中間生成ファイル)	-c
.s	アセンブリ ファイル	-S オプション
.i	前処理済み C ファイル	-E、-o オプション

xc8-cc の既定値動作は ELF と Intel HEX 出力を生成します。-o オプション(4.6.2.3 「O: 出力ファイルの指定」参照) によって変更されない限り、これらのファイルのベース名はプロジェクト名と同じです。

-gcoff オプションを指定する事で COFF ファイルを生成できます(4.6.5.2 「G: デバッグ情報生成オプション」参照)。

ELF/DWARF ファイルは、プロジェクトに関するデバッグ情報をデバッガに提供します。このフォーマットにより、他のフォーマット(COFF 等)に比べてより正確なデバッグが可能となります。COFF を使った場合、プロジェクトの動作に関する一部の情報がデバッガで利用できない可能性があります。開発環境は、通常 ELF ファイルを生成するようコンパイラに要求します。

多くの出力ファイルは、元のソースファイルと同じベース名を持ちます。例えば-c オプションを使った場合、input.c という名前のソースファイルから input.p1 が生成されます。

4.4.2 分析ファイル

コンパイラは 2 つの重要なファイルを生成します(アセンブラがアセンブリ リストファイルを生成し、リンカがマップファイルを生成します)。これらは、表 4-3 に示すオプションを指定する事により生成できます。

表 4-3. 分析ファイル

拡張子	ファイル型	生成方法
file.lst	アセンブリ リストファイル	-Wa,-a=file.lst ドライバ オプション
file.map	マップファイル	-Wl,-Map=file.map ドライバ オプション

アセンブリ リストファイルは、元のソースコードと生成されたアセンブリコードの間の対応を示します。このファイルからは、C ソースがどのようにエンコードされたか、アセンブリ ソースコードがどのように最適化されたか、といった情報が得られます。アセンブリ リストファイルは、オブジェクトにアクセスするコンパイラ生成コードがアトミックかどうかを確認する際に不可欠です。また、このファイルは、全てのオブジェクトとコードが配置されている領域を示します。

リストファイルを生成するためのアセンブラ オプションは-a です。このオプションは、ドライバ オプション (例: -Wa,-a=*file*.lst) を使ってドライバからアセンブラに渡す事ができます。

プログラム全体(ライブラリ コードを含む) から 1 つのリストファイルが生成されます。

マップファイルは、メモリ内のオブジェクトの配置に関する情報を提供します。このファイルは、ユーザ定義リンカオプションが正しく処理されたかどうかを確認する場合や、オブジェクトと関数の正確な位置を調べる場合に役立ちます。

マップファイルを生成するためのリンカオプションは-Map *file* です。このオプションは、ドライバ オプション (例: -Wl,-Map=*file*.map) を使ってドライバからリンカに渡す事ができます。

プロジェクトのビルド時に 1 つのマップファイルが生成されます(リンカが実行されて完了した場合)。

© 2022 Microchip Technology Inc. and its subsidiaries

4.5 コンパイラのメッセージ

全てのコンパイラ アプリケーションは、コンパイル処理中にテキスト メッセージによって状況を報告します。

以下では、メッセージのタイプと、それらが生成された時のコンパイラの挙動について説明します。

アドバイザリ メッセージ	コンパイラが遭遇した状況、またはコンパイラが実行しようとしている動作に関する情報を提供します。情報は「参考として」示され、通常は対処する必要はありません。このタイプのメッセージが出力された後もコンパイルは継続されます。
警告 メッセージ	コンパイルは可能であるものの、実行時に問題が生じる恐れのあるソースコードまたはその他の状況を知らせます。警告の原因となったコードまたは状況は調査する必要があるものの、そのモジュールのコンパイルは続行され、残りのモジュールもコンパイルされます。
エラー メッセージ	不正な(またはコンパイルできない)ソースコードを示します。そのモジュール内の残りのソースコードに対するコンパイルは試みられますが、最初に見つかったエラー要因によってさらなるエラーが報告される可能性があります。プロジェクト内の他のモジュールのコンパイルは実行されますが、プロジェクトはリンクされません。
致命的エラー メッセージ	コンパイルをこれ以上進める事ができない状況を示します。このメッセージが出力されるとコンパイルは直ちに停止します。

警告およびエラーメッセージの一覧は以下の関連リンクに記載しています。

関連リンク

10.「エラーおよび警告メッセージ」

4.5.1 メッセージの概要

大部分のアプリケーションは集中型メッセージ システムを使ってコードとコンパイル処理の妥当性を報告します。

Note: C99 プロジェクト向けのコンパイル時に使われる clang フロントエンドはこのメッセージ システムをバイパスするため、ここで説明する機能によって制御されません。このフロントエンドは C ソースコードの構文エラー等のメッセージを管理します。その他の全てのアプリケーションは、ここで説明するメッセージ システムを使います。

各メッセージには一意の番号が割り当てられています。メッセージ システムは、メッセージ発行元のアプリケーションが要求するメッセージ番号に基づいて、対応するメッセージ タイプと文字列を各種 MDF (Message Description File) の 1 つから特定します。MDF は、コンパイラのインストール ディレクトリ内の `pic/dat` に保存されています。一部のアプリケーションはこの情報のコピーを内蔵しており、コマンドラインで MDF が指定されなかった場合または MDF が見つからなかった場合に、それらの情報を使います。

メッセージの重要度に関するしきい値を設定する事で、ユーザが重要と考えるメッセージだけを表示させる事ができます。加えて、特定番号のメッセージを無効にする事もできます。場合によっては、プラグマを使って特定コード行内でのみ特定メッセージ番号を無効にする事も可能です。これらの方法は 4.5.3.1 「メッセージの無効化」に記載しています。

メッセージがトリガされた時に、コンパイラが生成するメッセージ文字列に加えて、各種の情報(メッセージ番号、メッセージをトリガしたコードの行番号、問題があるコードを格納しているファイルの名前、メッセージを発行したアプリケーション等)を表示させる事ができます。

エラーメッセージが出力されるたびに内部カウンタがインクリメントします。エラーメッセージの回数が特定数に達すると、そのモジュールのコンパイル処理は停止します。コンパイルを停止するエラー回数は、`-fmax-errors` オプション(4.6.4.1 「max-errors オプション」)を使って調整できます。このカウンタは、内部コンパイラ アプリケーションごとにリセットされます。例えば最大エラー回数を 5 に設定した場合、各アプリケーション (パーサー、コード ジェネレータ、リンカ、ドライバ等)でそれぞれ最大 5 回のエラーが許容されます。

MDF 内の情報はテキストエディタを使って編集できますが、これは推奨しません。メッセージ生成動作は、以下で説明するオプションまたはプラグマを使って変更する事を推奨します。

© 2022 Microchip Technology Inc. and its subsidiaries

4.5.2 メッセージのフォーマット

既定値により、メッセージは人間が読める形式で出力されます。内部アプリケーションごとに処理するファイルのフォーマットが異なるため、メッセージの書式もアプリケーションごとに異なります。

問題の箇所を C ソースコード内の行番号で示す事ができるアプリケーション(パーサー等)もあれば、モジュール名とレコード番号までしか示せないアプリケーション(リンカ等)もあります。ソースコードではなくドライバ オプションの問題に関するメッセージが出力される場合もあります。

clang フロントエンドが生成するメッセージのフォーマットを変更する事はできませんが、その他のコンパイラ アプリケーションからのメッセージは以下で説明する変更が可能です。

コンパイラは環境変数を使います。それらの設定値は、全てのコンパイラ アプリケーションによって生成される全てのメッセージに対するテンプレートとして使われます。これらの環境変数の名前と効果を表 4-4 に示します。

表 4-4. メッセージの環境変数

変数	効果
HTC_MSG_FORMAT	全てのアドバイザリ メッセージ
HTC_WARN_FORMAT	全ての警告メッセージ
HTC_ERR_FORMAT	全てのエラーおよび致命的エラー メッセージ

これらの環境変数の値は、メッセージ フォーマット用のテンプレートとして使われる文字列です。文字列内で printf に似たプレースホルダを配置する事で、メッセージのフォーマットをカスタマイズできます。各プレースホルダに対応する情報を表 4-5 に示します。

表 4-5. メッセージのプレースホルダ

プレースホルダ	置換される情報
%a	アプリケーション名
%c	列番号
%f	ファイル名
%l	行番号
%n	メッセージ番号
%s	メッセージ文字列(MDF から)

DOS バッチファイル内でこれらのオプションを使う場合、プレースホルダを指定するために 2 個の「%」文字が必要です(DOS では 1 個の「%」は引数として解釈され、コンパイラに渡されません)。

例:

```
SET HTC_ERR_FORMAT="file %%f: line %%l"
```

4.5.3 メッセージ挙動の変更

コンパイラが生成するメッセージの一部の属性は変更可能です。変更によりメッセージ全体が無効になる場合もあります。生成される警告メッセージの数を制御する事でデバッグがやりやすくなる場合もあります。

4.5.3.1 メッセージの無効化

番号付きの各警告メッセージには、重要度を示す値が割り当てられています。この値は MDF 内で-9〜9 に設定されています。この値が大きいほど警告の重要度は高くなります。

-mwarn ドライバ オプション(4.6.4.2 「warn オプション」参照)を使って警告レベルのしきい値を調整する事により、重要度の低い警告メッセージを無効にできます。重要度が現在のしきい値より低い全ての警告は表示されません。

既定値のしきい値は 0 です(重要度が 0 以上の警告のみ表示されます)。このオプションは全てのコンパイラ アプリケーション (従って全てのコンパイル処段) に適用されます。

C99 向けにビルドする場合、clang フロントエンドが使われます。このアプリケーションは番号付きメッセージ システムを使いません。パーサーからの警告メッセージには、そのメッセージの生成がどのオプションによって有効にされているのかが示されるため、そのメッセージを無効にしたい場合に役立ちます。例えば、以下の警告が発行されたとします。

```
init.c:8:11: warning: implicit conversion changes signedness:'int' to 'unsigned int' [-Wsign-conversion]
```

この警告はオプション-Xparser -Wno-sign-conversion を使って無効にできます(警告メッセージに示される角カッコ内のオプション(上の例では[-Wsign- conversion]) に「no-」を付加したオプションを指定する事で、そのメッセージを無効にできます)。-Xparser ドライバ オプションは、オプション引数をパーサー(clang)アプリケーションに渡します。

全てのアプリケーションからの全ての警告は、-w オプションを使って無効にできます。

Note: エラーまたは警告メッセージを無効にしても、メッセージをトリガした問題そのものがなくなるわけではありません。警告メッセージを無効にする際は、常に十分な注意が必要です。

4.5.3.2 メッセージタイプの変更

一部のメッセージのタイプを変更する事もできます。例えば、通常は警告として発行されるメッセージをエラー メッセージに変更する事ができます。これは、パーサーまたはコード ジェネレータによって生成されるメッセージに対してのみ可能です。このプラグマの詳細は 5.13.3.11 「#pragma warning ディレクティブ」を参照してください。

4.6 オプションの説明

コンパイル処理の大部分は、コマンドライン ドライバ(xc8-cc) にオプションを渡す事により制御できます。

多くのオプションは GCC スタイルに従いますが、本コンパイラは GCC に基づいていないため、以下に記載していないその他の GCC オプションは使えません。旧式の xc8 ドライバ オプション(『MPLAB® XC8 C Compiler User's Guide』(DS50002053)参照) の多くは、等価な GCC スタイル オプションが利用可能でなければ xc8-cc でも使えますが、使うと警告が出力される可能性があります。

全てのオプションで大文字と小文字が区別され、オプション名の前に付加された 1 つまたは 2 つのダッシュ文字によりオプションとして認識されます(例: -c または --version)。

--help オプションを指定すると、コマンドラインで使えるオプションの簡単な説明が表示されます。

開発環境内でコンパイルする場合、プロジェクト プロパティ内の選択に基づくオプションがコンパイラへ渡されます。コマンドラインでコンパイルを実行する場合、プロジェクトに必要なオプションが本コンパイラの既定値オプションとは異なる場合があります。これらの既定値オプションがプロジェクトに対して適切であるかどうかは、ユーザが確認する必要があります。

4.6.1 PIC デバイスに固有のオプション

MPLAB XC8 コンパイラを使って Microchip 社の 8 ビット PIC デバイスをコンパイルする場合に使えるオプションを表 4-6 に示します。

表 4-6. PIC デバイス固有オプション

オプション	制御内容
-maddrqual=*action*	記憶域(メモリアドレス)修飾子に対するコンパイラの応答
-mchecksum=*specs*	チェックサムまたはハッシュの生成と配置
-mcodeoffset=*offset*	リセットおよび割り込みベクタに適用するオフセット
-m[no-]config	既定値のコンフィグレーション ビット値を使ってデバイスをプログラミングするかどうか
-mcpu=*device*	コードをビルドする際のターゲット デバイス
-mdebugger=*type*	コード実行時に使うデバッガ
-m[no-]default-config-bits	-m[no-]config 参照
-mdfp=*path*	使用するデバイスファミリ パック
-m[no-]download	最終的な HEX ファイルのコンディショニング方法

.....続き

オプション	制御内容
-m[no-]download-hex	-m[no-]download 参照
-memi=*mode*	使用する外部メモリ インターフェイス
-merrata=*type*	コンパイラが適用するエラッタ回避策
-mheap=*size*	ヒープの最大許容サイズ
-m[no-]ivt=*address*	起動時に選択する割り込みベクタテーブル
-m[no-]keep-startup	コンパイル後に実行時起動ソースを削除するかどうか
-mmaxichip	仮想的なフルメモリ デバイスの指定
-m[no-]osccal	オシレータを校正するかどうか
-m[no-]oscval=*value*	オシレータの校正値
-mram=*ranges*	プログラム用に利用可能なデータメモリ領域
-mreserve=*ranges*	予約するメモリ領域
-m[no-]resetbits	デバイス ステータスビットを保存するかどうか
-mrom=*ranges*	プログラム用に利用可能なプログラムメモリ領域
-m[no-]save-resetbits	-m[no-]resetbits 参照
-mshroud	出力ファイルでソースコードを難読化するかどうか
-mstack=*model*[:*size*]	既定値で使用するデータスタック
-m[no-]stackcall	ルックアップ テーブルによる関数呼び出しが可能かどうか
-msummary=*types*	表示するメモリ使用量サマリ情報の内容
-mundefints=*action*	未実装割り込みに対するコンパイラの対応
-m[no-]use-ivt	-m[no-]ivt 参照

4.6.1.1 addrqual オプション

-maddrqual=*action* オプションは、C ソースコード内の非標準(PIC 固有)メモリ修飾子に対するコンパイラの対応を、 表 4-7 の通りに指定します。

表 4-7. メモリ修飾子に対するコンパイラの対応

action	対応
require	非標準メモリ修飾子を尊重します。それらに適合できない場合、エラーが生成されます。
request	可能な限り、非標準メモリ修飾子を尊重します。それらに従う事ができなくてもエラーは生成されません。
ignore	非標準メモリ修飾子は無視され、それらの修飾子が使われていないかのようにコードがコンパイルされます。
reject	非標準メモリ修飾子が見つかると直ちにエラーが生成されます。

__near 修飾子は、このオプションの影響を受けます。PIC18 デバイスでは、このオプションは__far 修飾子に影響を及ぼします。その他の 8 ビットデバイスでは、このオプションは__bank(*x*)修飾子に影響を及ぼします。既定値では、これらの修飾子は無視されます(エラーなしで許容されますが、修飾子の効果はありません)。このオプションにより、これらの修飾子に対するコンパイラの対応を選択できます。

例えば、-maddrqual=request オプションを指定した場合、コンパイラは全ての非標準修飾子を尊重するよう試み、それができない場合はユーザに通知する事なく無視します。

4.6.1.2 checksum オプション

`-mchecksum=specs` オプションは、指定されたアドレス領域に対してハッシュ値(例: チェックサムまたは CRC)を計算し、その結果を指定されたデスティネーション アドレスで HEX ファイルに保存します。このオプションのフォーマットは以下の通りです。

```
-mchecksum=start-end@destination[,specifications]
```

start、*end* 、*destination* 属性は既定値により 16 進定数です。入力レンジを定義するアドレス(start - end)は、通常アルゴリズム幅の整数倍で指定します。そうしなかった場合、足りない入力ワード位置は値が 0 のバイトによって埋められます。destination は、ハッシュ結果の保存位置を指定します。このアドレスは、ハッシュが計算されるアドレス領域の外である事が必要です。

specifications により、以下の引数をコンマで区切ったリストとして指定する事ができます。

表 4-8. checksum オプションの引数

引数	概要
width=*n*	結果の幅を 10 進数で指定します(オプション)。大部分のアルゴリズムでは 1～4 のバイト幅で結果を計算できます。SHA アルゴリズムの場合、この値はビット幅を表します。正の値で width を指定した場合、結果はビッグエンディアンのバイト順で保存されます。負の値で width を指定した場合、結果はリトルエンディアンのバイト順で保存されます。幅を指定しなかった場合、2 バイト幅の結果がリトルエンディアンのバイト順で保存されます。Fletcher アルゴリズムの場合、固定幅であるため width 引数を指定する必要はありませんが、この引数を使って結果のエンディアンを変更する事ができます。
offset=*nnnn*	ハッシュ計算で使われる初期 16 進値またはオフセットを指定します。SHA アルゴリズムに対してこの引数は不要です。
algorithm=*n*	Hexmate に実装されているハッシュ アルゴリズムの 1 つを 10 進値で選択します。選択可能なアルゴリズムを表 4-9 に示します。 この引数を指定しなかった場合、既定値アルゴリズム (8 ビットチェックサムの加算、表 4-9 内のアルゴリズム 1) が使われます。
polynomial=*nn*	CRC アルゴリズムを使う場合に多項式の値を選択します。
code=*nn*.*Base*	結果内の各バイトを追跡する 16 進コードを選択します。これにより、ハッシュ結果の各バイトを命令に埋め込む事ができます。例えば code=34 は、結果の各バイトをミッドレンジ デバイスの retlw 命令に埋め込みます。code=0000 は、ハッシュの各バイトに 2 個の 0x00 バイトを追加します。code=nn の後に.*Base* を任意に追加できます。*Base* は、後続のコードシーケンスが追加される前に出力されるハッシュのバイト数です。 例えば、code=1122.2 を指定した場合、ハッシュ結果の各 2 バイトの後に 0x11 のバイトと 0x22 のバイトが出力されます。
revword=*n*	ワード幅を 10 進値で指定します。非 0 の値を指定した場合、ハッシュ値を計算する時に各ワード内のバイトは逆順に読み出されます。各ワードは、HEX ファイル内でアドレスに対して整列します。現時点では、この幅は 0 または 2 である事が必要です。0 を指定した場合、バイトを逆順に読み出す機能は無効になります。これは、revword サブオプションを何も指定しなかった場合と同じです。PIC 内蔵ハードウェア CRC モジュールが利用できる場合、Hexmate からの CRC と PIC の Scanner モジュールからの結果を一致させるため、このサブオプションを使う必要があります。
skip=*n*.*Bytes*	ワード幅を 10 進値 *n* で指定します。非 0 の値を指定した場合、各ワード内の *Bytes* で指定した上位側バイトはハッシュ値の計算中にスキップされます。各ワードは、HEX ファイル内でアドレスに対して整列します。 現時点では、この幅は 0 (スキップ機能は無効、skip サブオプションを何も指定しないのと同じ)または 2 以上の値である事が必要です。この値の後に.*Bytes* を任意に追加できます。*Bytes* は、各ワード内でスキップするバイトの数を表します。例えば skip=4.2 を指定した場合、各 4 バイトワード内の最上位 2 バイトがスキップされます。

revword および skip 機能は、処理中のシーケンス内のデータ位置に対して作用するのではなく、HEX ファイルアドレスに整列したワードに対してのみ作用します。言い換えると、ワードの整列は checksum オプションで指定された start および end アドレスの影響を受けません。

`--fill` オプション(4.6.11.10 「fill オプション」参照)を一緒に指定しない場合、`checksum` オプションで指定されたアドレスレンジ内の未使用位置には自動的に以下の値が書き込まれます。

0xFFF: ベースライン デバイス、0x3FFF: ミッドレンジ デバイス、0xFFFF: PIC18 デバイス

これにより、計算に未知の値が使われる事を防いで、結果の精度を確保する事ができます。

例:

```
-mchecksum=800-fff@20,width=1,algorithm=2
```

このオプションにより、アドレスレンジ 0x800～0xfff から計算された 1 バイトのチェックサムがアドレス 0x20 に保存されます。この場合、16 ビット加算アルゴリズムが使われます。表 4-9 に、利用可能なアルゴリズムを示します。これらの詳細は 8.2.3 「ハッシュ値の計算」に記載しています。

表 4-9. チェックサム アルゴリズムの選択

選択値	アルゴリズムの概要
-5	バイト順を反転した巡回冗長検査(CRC)結果
-4	初期値から 32 ビット値を減算
-3	初期値から 24 ビット値を減算
-2	初期値から 16 ビット値を減算
-1	初期値から 8 ビット値を減算
1	初期値に 8 ビット値を加算
2	初期値に 16 ビット値を加算
3	初期値に 24 ビット値を加算
4	初期値に 32 ビット値を加算
5	巡回冗長検査(CRC)
7	Fletcher16 のチェックサム(8 ビット処理、16 ビットの結果)
8	Fletcher32 のチェックサム(16 ビット処理、32 ビットの結果)
10	SHA-2 (現在は SHA256 のみサポート)

ハッシュ計算は、Hexmate アプリケーションによって実行されます。このドライバ オプションの情報は、Hexmate アプリケーションの実行時に渡されます。

4.6.1.3 codeoffset オプション

`-mcodeoffset=`*`offset`* オプションは、メモリ内のリセットおよび割り込みベクタ位置のオフセットを指定します。これにより、このオフセット アドレスまでのメモリ領域がコードまたはデータによって使われる事を防ぐ事ができます。このオプションは、ブートローダを書く場合に通常必要となります。

オフセット アドレスの値は 16 進定数であると見なされます。16 進数表記を表す先頭の `0x` または末尾の h は使えますが、使わなくても構いません。

このオプションは、ベクタ割り込みをサポートするデバイスの割り込みベクタテーブル位置に影響を及ぼしません。それらのデバイスがレガシーモードで動作する場合も、低優先度および高優先度割り込みベクタ位置に影響を及ぼしません。これら 2 つのタスクのどちらを実行する場合も、IVTBASE レジスタを調整します。

例えば、`-mcodeoffset=600` を指定した場合、ミッドレンジ PIC デバイスのリセットベクタはアドレス 0 から 0x600 に移動し、割り込みベクタはアドレス 4 から 0x604 に移動します。PIC18 デバイスの場合、リセットベクタは 0x608 に移動し、割り込みベクタは 0x618 に移動します。この場合、アドレス 0～0x5FF の領域にはコードもデータも配置されません。

PIC デバイスでは、リセットおよび割り込みベクタ位置は固定されているため、実際のベクタ位置からオフセットしたリセットおよび割り込みルーチンまで制御をリダイレクトできるコードをユーザの責任で提供する必要があります。

4.6.1.4 config オプション

-mconfig オプションを指定すると、コード内で config プラグマによって設定されていないコンフィグレーション ビットは、コンパイラによって既定値に設定されます。-mdefault-config-bits は、このオプションと等価です。

既定値動作は未指定のビットを設定しません。これは、-mno-config または(-mno-default-config-bits)オプションを使って明示的に指定できます。

4.6.1.5 cpu オプション

ビルド時に-mcpu=*device* オプションを使ってターゲット デバイスを指定する必要があります。これは、必ず指定する必要がある唯一のオプションです。

例えば-mcpu=18f6722 は PIC18F6722 を選択します。このオプションで選択できるデバイスの一覧は、-mprint-devices オプション(4.6.2.8 「print-devices」参照)を使って表示できます。

4.6.1.6 debugger オプション

-mdebugger=*type* オプションは、デバッガとして使用する開発ツールを選択します。xc8-cc は、表 4-10 に示すデバッガをサポートします。適切なデバッガを選択しないと、ランタイムエラーが発生する可能性があります。

表 4-10. 選択可能なデバッガ

タイプ	選択されるデバッガ
none	デバッガを使わない(既定値)
icd2	MPLAB® ICD 2
icd3	MPLAB ICD 3
icd4	MPLAB ICD 4
pickit2	PICkit™ 2
pickit3	PICkit 3
pickit4	PICkit 4
realice	MPLAB REAL ICE™インサーキット エミュレータ
snap	MPLAB Snap

例:

```
xc8-cc -mcpu=16F877AA -mdebugger=icd4 main.c
```

上記のオプション指定により、MPLAB ICD4 向けデバッグ エグゼクティブによって使われるリソースがデバイス内部で確保されます(そのリソースがソースコードに割り当てられる事を防ぎます)。

このオプションを使ってデバッガが設定されるとプリプロセッサ マクロが定義され、デバッガの選択に基づくソースコードの変更が可能となります(5.13.2 「定義済みマクロ」参照)。

開発ツールのデバッグ機能を使わない場合(例: MPLAB ICD 4 をプログラマとしてのみ使う場合)、メモリリソースはツールによって使われないため、debugger オプションは none に設定できます。

4.6.1.7 dfp オプション

-mdfp=*path* オプションを指定すると、ターゲット デバイス(-mcpu オプションで指定) 向けのデバイスサポートはデバイスファミリ パック(DFP) の内容から取得されます。*path* は、DFP の「xc8」サブ ディレクトリへのパスです。

このオプションを指定しなかった場合、xc8-cc ドライバはコンパイラと一緒に配布されたデバイス固有ファイルを可能な限り使います。

Microchip 社の開発環境は、自動的にこのオプションを使って、どのデバイス固有情報を使うのかコンパイラに指示します。コンパイラ向けの追加の DFP を入手した場合、このオプションをコマンドラインで使います。

DFP はデバイス固有ヘッダファイル、コンフィグレーション ビットデータ、ライブラリ等の情報を含みます。DFP を使う事で、コンパイラを更新しなくても新しいデバイスの機能が使えるようになります。DFP は実行コードを含まず、既存ツールまたは標準ライブラリ関数に対するバグ対策も改良も提供しません。

このオプションを指定した場合、プリプロセッサはインクルード ファイルを最初に*<DFP>*/xc8/pic/include/proc および*<DFP>*/xc8/pic/include ディレクトリで検索し、見つからなければ標準の検索ディレクトリを検索します。

4.6.1.8 download オプション

-mdownload オプションは、Intel HEX 形式ファイルをブートローダ向けにコンディショニングします。このオプションの効果は-mdownload-hex オプションと等価です。

このオプションを指定すると、Intel HEX 形式ファイル内のデータレコードは 16 バイト長にパディングされ、16 バイト境界に整列されます。

既定値の動作は HEX ファイルを変更しません。これは-mno-download(-mno-download-hex)オプションを使って明示的に指定できます。

4.6.1.9 emi オプション

-memi=*mode* オプションにより、一部の PIC18 デバイスで利用可能な外部メモリ インターフェイスのモードを選択できます。

このインターフェイスは表 4-11 に示すモードで動作できます。

表 4-11. 外部メモリ インターフェイスのモード

モード	動作
wordwrite	16 ビット ワード書き込みモード(既定値)
byteselect	16 ビット バイト選択モード
bytewrite	8 ビット バイト書き込みモード

例えば-memi=bytewrite オプションは、8 ビットバイト書き込みモードを選択します。

モードの選択は、外部データ インターフェイスへの書き込み時に生成されるコードに影響を及ぼします。16 ビット ワード書き込みモードでは、ダミーの読み書きを追加する事で常に偶数バイトが書き込まれるようにできます。16 ビットバイト選択モードまたは 8 ビットバイト書き込みモードではダミーの読み書きは生成されず、より効率的なコードが得られます。

このオプションを指定してもデバイスは選択されたモードの初期化コードを生成しないので注意が必要です。コード内で必要な設定は、デバイスのデータシートに記載されています。

4.6.1.10 errata オプション

-merrata=*type* オプションを使うと、文書化されたシリコンエラッタ問題に対するソフトウェア回避策を指定できます。エラッタの既定値セットが各デバイスに適用されますが、このオプションと表 4-12 に示す引数を使ってこの既定値セットを調整できます。

表 4-12. エラッタ回避策

タイプ	番号	回避する問題
4000	0	4000h アドレス境界をまたぐプログラムメモリへ アクセス/ジャンプ
fastints	1	高速割り込みシャドーレジスタの破損
lfsr	2	LFSR 命令の不具合対策
minus40	3	-40℃でのプログラムメモリ読み出し
reset	4	goto 命令がリセットベクタ位置に存在できない

.....続き

タイプ	番号	回避する問題
bsr15	5	BSR が値 15 を保持する時のフラグ問題
daw	6	DAW 命令の不正結果
eedatard	7	RD セット後の即値命令における EEDAT の読み出し
eeadr	8	EEADR ロード直後に RD ビットがセットされない
ee_lvd	9	EEPROM への書き込み前に LVD が安定する必要がある
fl_lvd	10	フラッシュへの書き込み前に LVD が安定する必要がある
tblwtint	11	tblwt 命令の前の割り込みレジスタのクリア
fw4000	12	Flash write exe must act on opposite side of 4000h boundary
resetram	13	非同期の場合に RAM 内容が破損する可能性がある、 書き込みアクセス中にリセットが発生する
fetch	14	命令フェッチの破損 – 要求された位置で FFFFh (nop)に適用
clocksw	15	外部オシレータクロック源への切り換え時のコードの破損 – HFINTOSC 高消費電力モードへの最初の切り換えに適用
branch	16	bra または brw 命令中の割り込みからの復元時に PC が無効になる可能性がある — 分岐命令を避ける
brknop2	17	ハードウェア ブレークポイントが bra 命令の影響を受ける可能性がある — 後続の位置への分岐を避ける
nvmreg	18	リセット後にプログラムはプログラム フラッシュメモリではなくデータフラッシュにアクセスする — NVMCON レジスタを調整する
bsr63	19	BSR が 63 を保持している時に movff 命令の実行に失敗する

現時点では、回避策は主に PIC18 デバイス向けに使われますが、clocksw および branch エラッタは一部の拡張ミッドレンジ デバイスにのみ適用されます。

ソフトウェア回避策を無効にするには、以下のオプションを指定します。

```
-merrata=none
```

例えば、アドレス 4000h をまたぐジャンプのエラッタのみ無効にし、その他の既定値回避策を全て有効にする場合、オプションを以下の通りに指定します。

```
-merrata=default,-4000
```

プリプロセッサ マクロ_ERRATA_TYPES (5.13.2 「定義済みマクロ」参照) の値により、適用されるエラッタが示されます。表 4-12 に示した各エラッタは、このマクロの値の 1 つのビット位置に対応します(表内の最上段のエラッタは最下位ビットに対応)。_ERRATA_TYPES マクロ内でセットされているビット位置に対応するエラッタが適用されます。ヘッダファイル<errata.h>は、各エラッタ値の定義(例: ERRATA_4000、ERRATA_FETCH 等)を格納しています。このヘッダとコンパイラが定義した_ERRATA_TYPES マクロを使う事で、どのエラッタが有効なのか調べる事ができます。

4.6.1.11 heap オプション

-mheap=[*size*|auto]オプションは、ヒープ用に予約するメモリのサイズを指定します。

ヒープは、標準の動的メモリ割り当て関数 malloc()、calloc()、realloc()、free()で使われます。その動作については5.4.7.1 「PIC デバイスの動的メモリ割り当て」を参照してください。

コンパイラは、このオプションの引数 *size* で指定されたバイト数(10 進値で指定) のメモリをヒープ用に予約します。実行時に、ヒープは最大でこのサイズまでメモリを使えます。このメモリは無条件に(たとえプログラム内で動的メモリ割り当て関数が使われていなくても)予約されます。

この引数には auto を指定する事もできます。その場合、コンパイラは他の静的メモリ割り当てとスタックの割り当てを行った後に、ヒープ用に使える最大メモリサイズを計算します。コンパイラは、プログラム内に動的メモリ割り当て関数への呼び出しが存在する場合にのみ、計算したサイズのメモリを予約します。-mheap オプションが指定されなかった場合、コンパイラは「auto」設定(-mheap=auto と等価)を使います。「auto」設定の場合、コンパイラがプログラム内で動的メモリ割り当て関数を検出してメモリをヒープ用に予約すると、アドバイザリ メッセージが出力されます。

再入可能モデルを使ってコンパイルされた関数向けのソフトウェア スタックのサイズも動的であるため、それ用にメモリの予約が必要です。これには-mstack オプション(4.6.1.22 「stack オプション」参照)を使います。-mheap オプションと-mstack オプションの両方で引数 size に対して auto を指定した場合、コンパイラは静的メモリ割り当てを行った後に、残りの未使用メモリをヒープとスタックに均等に割り当てます。

4.6.1.12 ivt オプション

割り込みベクタテーブルを備えた PIC18 デバイスでは、-mivt=*address* オプションにより、プログラムの実行開始時に使う割り込みベクタテーブルを選択します。

引数 *address* で指定されたアドレスは、起動時に IVTBASE レジスタに書き込まれます。例えば-mivt=0x200 と指定した場合、ベースアドレスが 200h の割り込みベクタテーブルが選択されます。指定されたアドレスのテーブルは、ソースコード内でベクタによって実装される必要があります。これには、ユーザが書く割り込みルーチン定義内で base 引数を使います(5.8.1 「割り込みサービスルーチンの書き方」参照)。既定値の動作では、ベクタテーブルはアドレス 0x8 に配置されたままです。この既定値動作は、オプション-mno-ivt を使って明示的に指定できます。

4.6.1.13 keep-startup オプション

-mkeep-startup オプションは、コンパイル後に実行時起動コードが削除される事を防ぎます。既定値では、このファイルは削除されません。-mno-keep-startup オプションを指定すると、このファイルは各ビルド後に削除されます。

4.6.1.14 maxichip オプション

-mmaxichip オプションを指定すると、コンパイラは選択されたデバイスと同じコアおよび周辺モジュールを持つ仮想的デバイス向けに、そのデバイスファミリで使える最大限のメモリリソースを使ってビルドします。ターゲットデバイスにプログラムが収まりきらない場合、このオプションを使う事で、どの程度コードとデータを削減すればデバイスに書き込む事ができるのか検討する事ができます。

選択されたデバイスでプログラムメモリ、データメモリ、EEPROM の容量が不足した場合、通常コンパイラは処理を停止します。このオプションを使うと、PIC18 およびミッドレンジ デバイスのプログラムメモリは、アドレス 0 から外部メモリのボトムアドレスまたは PC レジスタが許容する最大アドレス(どちらか低い方)まで最大化されます。ベースライン デバイスのプログラムメモリは、アドレス 0 からコンフィグレーション ワードまでのアドレス位置まで最大化されます。

データメモリのバンク数は BSR レジスタ(PIC18)、STATUS レジスタ内の RP ビット(ミッドレンジ デバイス)、FSR レジスタ内のバンク選択ビット(ベースライン デバイス)で定義された選択可能バンクの最大数まで拡張されます。追加された各バンク内の RAM 容量は、物理的に実装されたバンク内の最大連続メモリ領域のサイズと同じです。

デバイスが EEPROM を備える場合、そのサイズは EEADR または NVMADR レジスタ内のビットの数によって決まるサイズまで適切に最大化されます。

このオプションを使う場合にデバイスで利用できるメモリのサイズと構成はマップファイル(7.3 「マップファイル」)で確認できます。

Note: -mmaxichip オプションを使った場合、実際のデバイス向けにビルドは行われません。生成されたコードは、シミュレータで実行する事もデバイスにロードする事もできません。このオプションを使っても、大きなコードがデバイスに書き込めるようになるわけではありません。

4.6.1.15 osccal オプション

一部の PIC10/12/16 デバイスでは、-mosccal オプションを使ってオシレータを校正できます。

このオプションを使うと、デバイスに書き込み済みの校正定数を使ってオシレータを校正するためのコードが生成されます。-mno-osccal を指定した場合、この初期化を実行するコードは、実行時起動コードから省略されます。

4.6.1.16 oscval オプション

一部の PIC10/12/16 デバイスでは、`-moscval=value` オプションを使ってオシレータの校正値(16 進数)を指定する事ができます。通常この校正値はデバイスに書き込み済みですが、このオプションを使うと別の値を指定する事ができます。

校正値はプログラムメモリのトップに保存され、`movlw` 命令内へカプセル化されます。実行時起動コードは、この命令を実行して WREG を校正レジスタ内に保存します。`-moscval=55` オプションを指定すると、起動時に値 0x55 がオシレータ校正レジスタに書き込まれます(5.2.11 「オシレータ校正定数」参照)。`-mno-oscval` オプションを指定した場合、たとえ引数を指定しても、デバイスに現在保存されている値が校正レジスタに書き込まれます。これは、`-moscval` オプションを何も指定しなかった場合と同じです。

4.6.1.17 ram オプション

`-mram=ranges` オプションは、ターゲット デバイス向けに割り当てられたデータメモリを調整するために使います。このオプションを指定しない場合、デバイスが内蔵する RAM の全てが利用可能となります。従ってこのオプションは、特殊なメモリ要件が存在する場合にのみ使います。ターゲット デバイス上に存在しないメモリ領域を指定した場合、たとえコンパイルに成功しても実行時に異常動作が発生する可能性があります。

例えば下記のオプションは、デバイス内に既に存在するメモリに対して追加のメモリレンジを指定します。

```
-mram=default,+100-1ff
```

これにより、100h～1ffh のレンジが内蔵メモリに追加されます。外部レンジのみを使用し、内蔵メモリの全てを無視する場合、下記のオプションを使います。

```
-mram=0-ff
```

このオプションは、対応する chipinfo ファイル内で内蔵メモリとして既に定義されているメモリレンジを予約するためにも使えます。この場合、下記の通り、レンジの前にマイナス文字「-」を付けます。

```
-mram=default,-100-103
```

この場合、定義された内蔵メモリの全てが使われますが、100h～103h のアドレス領域は RAM オブジェクトの割り当て用に使われません。

このオプションは、リンカクラス(7.1.1 「A: リンカクラスの定義」参照) によって使われるメモリレンジを調整します。このオプションの影響を受けるクラスを使わない psect に格納された全てのオブジェクトは、このオプションによって指定された有効メモリのレンジ外にリンクされる可能性があります。

PIC18 デバイスでは、このオプションを使って `far` オブジェクト向けに RAM を指定する事もできます。これらのオブジェクトは PIC18 の拡張メモリ内に保存されます。内蔵プログラムメモリより上のアドレスを持つ追加のメモリをこのオプションで指定した場合、そのメモリは外部 RAM として実装された拡張メモリであると見なされます。

例えば下記のオプションは、外部 RAM がアドレス 0x20000～0x20fff の拡張メモリ空間に実装されている事を示します。

```
-mram=default,+20000-20fff
```

4.6.1.18 reserve オプション

`-mreserve=ranges` オプションを使うと、プログラムによって通常使われるメモリを予約できます。このオプションの一般的書式は以下の通りです。

```
-mreserve=space@start:end
```

space には、`ram` (データメモリ空間を指定)または `rom` (プログラムメモリ空間を指定)のどちらかを指定できます。*start* と *end* は、除外するアドレスレンジを指定します。例えば`-mreserve=ram@0x100:0x101` と指定した場合、データメモリからアドレス 100h～101h (2 バイト) が予約されます。

このオプションは`-mram` および`-mrom` オプションと同様のタスクを実行しますが、このオプションを使ってプログラム向けに利用可能なメモリを追加する事はできません。

4.6.1.19 resetbits オプション

PIC10/12/16 デバイスで`-mresetbits` オプションを使うと、実行時起動コードによって保存されたステータス レジスタの内容にアクセス可能となります(5.9.2.4 「ステータス レジスタの保持」参照)。このオプションの効果は`-m[no-]save-resetbits` オプションと等価です。

このオプションが有効である場合、保存されたレジスタにプログラム内でアクセスできます。コンパイラは、保存された STATUS レジスタシンボルへの参照を検出すると、自動的にこのオプションを有効にします。

4.6.1.20 rom オプション

`-mrom=`*`ranges`* オプションは、ターゲット デバイス向けに割り当てられた既定値プログラムメモリを変更するために使います。このオプションを指定しない場合、デバイスが内蔵するプログラムメモリの全てが利用可能となります。従ってこのオプションは、特殊なメモリ要件が存在する場合にのみ使います。ターゲット デバイス上に存在しないメモリ領域を指定した場合、たとえコンパイルに成功しても実行時に異常動作が発生する可能性があります。

例えば、下記のオプションは、デバイス内に既に存在するメモリに対して追加のメモリレンジを指定します。

```
-mrom=default,+100-2ff
```

これにより、100h～2ffh のレンジが内蔵メモリに追加されます。外部レンジのみを使用し、内蔵メモリの全てを無視する場合、下記のオプションを使います。

```
-mrom=100-2ff
```

このオプションは、デバイス コンフィグレーション ファイル内で内蔵メモリとして既に定義されているメモリレンジを予約するためにも使えます。この場合、下記の通り、レンジの前にマイナス文字「-」を付けます。

```
-mrom=default,-100-1ff
```

この場合、定義された内蔵メモリの全てが使われますが、100h～1ffh のアドレス領域は ROM オブジェクトの割り当て用に使われません。

このオプションは、リンカクラス(7.1.1 「A: リンカクラスの定義」参照)によって使われるメモリレンジを調整します。このオプションの影響を受けるクラスを使わない psect に格納された全てのコードまたはオブジェクトは、このオプションによって指定された有効メモリのレンジ外にリンクされる可能性があります。

一部の psect (特に `const` 修飾子付きのデータを保持する psect) は、しきい値アドレスより上のアドレスにリンクするする必要があるので注意が必要です。このオプションを使ってしきい値アドレスより上のメモリレンジを除外した場合、それらの psect は配置できなくなる可能性があります。

4.6.1.21 shroud オプション

`-mshroud` オプションは、中間生成ファイルまたはライブラリ ファイルを条件付きソースコードからビルドして配布する場合に使います。

このオプションを使うと、配布されたファイルから元のソースコードを再構成できなくするために、通常これらのファイルに含まれている C コメントとライン番号および変数名が削除または難読化されます。

4.6.1.22 stack オプション

`-mstack=`*`model`*`[:`*`size`*`|auto]`オプションは、プログラムが使うスタックモデルを選択します。これにより、スタック指定子を使わない関数のスタックベース変数に割り当てられるメモリ位置が決まります。加えて、このオプションは、スタック向けに予約するメモリのサイズを指定します。

関数は、スタック指定子(5.7.1.3 「reentrant / nonreentrant 指定子」参照)を使って、その関数のスタックベース変数向けに使うスタックを指定できます。関数がこれらの指定子を使わない場合、`-mstack` オプションは、コンパイラが適用するスタックのタイプを制御します。

データスタックにはコンパイルド スタックとソフトウェア スタックがあります(5.2.4.2 「データスタック」参照)。このオプションで使えるスタックモデルを表 4-13 に示します。

表 4-13. stack のサブオプション

モデル	スタックベース変数の既定値メモリ割り当て
compiled または nonreentrant	スタック指定子を使わない全ての関数向けにコンパイルド スタックを使います。これらの関数は再入不可となります。
software または reentrant	スタック指定子を使わない全ての関数向けにソフトウェア スタックを使います。これらの関数は再入可能となります。
hybrid	スタック指定子を使わず再入呼び出しされない関数にのみコンパイルド スタックを使い、スタック指定子を使わないその他の全ての関数にソフトウェア スタックを使います。関数は、要求された場合にのみ再入可能になります。

-mstack オプションを使わないと、スタック指定子を使わない全ての関数はコンパイルド スタックを使い、再入不可になります。

ベースラインおよびミッドレンジ デバイスはコンパイルド スタックのみサポートするため、これらのデバイスで software(reentrant)または hybrid モデルを選択しても効果はありません。加えて、全てのデバイスの全ての割り込み関数はコンパイルド スタックを使う必要がありますが、割り込み関数から呼び出される関数にはソフトウェア スタックが使えます。

ハイブリッド モデル(-mstack=hybrid)を指定した場合、プログラム内で再入呼び出しさる関数にはソフトウェア スタックが使われ、そうではない関数にはコンパイルド スタックが使われます。このモデルは、要求された場合にのみ再入を許可し、プログラム内の大部分の関数には効率の高いコンパイルド スタックを使います。

Note: software(reentrant)モデルの使用には注意が必要です。コンパイル時にソフトウェア スタックの最大ルーチンサイズを正確に知る事はできないため、コンパイラはオーバーフローを予測できません。オーバーフローが発生するとオブジェクトまたはレジスタが破損する可能性があります。全ての関数に対してソフトウェア スタックを指定すると、スタックサイズが大幅に増加する可能性があります。通常、ソフトウェア スタック上のオブジェクトにアクセスするコードは、コンパイルド スタック上のオブジェクトにアクセスするコードよりも大きくなります。

このオプションは、ソフトウェア スタック向けにコンパイラが予約するメモリの最大サイズを指定する事もできます。指定されたサイズは、ソフトウェア スタックに対してのみ適用されます。コンパイルド スタックのサイズは固定されておりコンパイル時に既知であるため、このオプションの影響を受けません。

メインライン コードと各割り込み関数に対して個別にソフトウェア スタックが生成されますが、これはプログラムレベルで透過的に行われます。コンパイラは、各スタックに対するメモリの割り当てを自動的に管理します。プログラムが割り込み関数を何も定義していない場合、利用可能な全てのメモリがメインライン コード向けのソフトウェア スタック用に使える場合があります。

スタックの各領域向けにコンパイラが予約を試みる最大空間サイズは、reentrant スタックタイプに続けて予約する各メモリ空間のバイト数(10 進数)をコロンで区切ったリストとして入力する事により指定できます。最初の値はメインライン コードに対応し、後続の値は最低優先度割り込みから最高優先度割り込みまで順番に対応します (PIC18 には 2 つの割り込み優先度がありますが、その他の割り込みをサポートスルデバイスには 1 つしかありません)。以下に例を示します。

```
-mstack=reentrant:200:0:40
```

PIC18 プロジェクトで上記のオプションを指定すると、メインライン スタック向けに 200 バイト、低優先度割り込みスタックに 0 バイト、高優先度割り込みスタックに 40 バイトが予約されます。実行時に、スタックはデータを破損する事なく最大でこのサイズまでメモリを使えます。このメモリは無条件に(たとえプログラム内に再入可能関数が存在していなくても)予約されます。

全てのサイズ引数を auto として指定する事もできます。その場合、コンパイラは他の静的メモリ割り当てとスタックおよびヒープの割り当てを行った後に、このスタックに使える最大メモリサイズを計算します。コンパイラが再入可能スタックを使う関数の存在を検出した場合、この計算されたサイズのメモリだけが予約されます。-mstack オプションが指定されなかった場合、コンパイラはプログラムによって使われる可能性がある全てのスタックに対して「auto」設定を適用します。「auto」を指定した場合、コンパイラによって再入可能関数が検出されてメモリが予約されるとアドバイザリ メッセージが出力されます。PIC18 デバイス向けの例を以下に示します。

```
-mstack=reentrant:auto:30:50
```

上記を指定した場合、低優先度割り込みスタックは最大で 30 バイトまで、高優先度割り込みスタックは最大で 50 バイトまで、オーバーフローせずに使えます。メインライン コードには、その他の静的メモリ割り当てとスタックおよびヒープの割り当て後に残されたメモリが使えます。割り込みを使わない場合、このオプションを使って割り込みスタックサイズを明示的に 0 に設定する事を推奨します。

このオプションを使って最大スタックサイズを指定する場合、各サイズに対して数値または auto のどちらかを指定する必要があります(size フィールドを空白のままにしてはいけません)。このオプションで指定されたサイズがスタックメモリ向けに利用可能なメモリサイズを超えていた場合、警告が出力され、利用可能なメモリサイズだけがスタック用に使われます。

動的メモリ割り当て関数によって使われるヒープのサイズも動的であるため、それ用にメモリの予約が必要です。これには-mheap オプション(4.6.1.11 「heap オプション」参照)を使います。-mheap オプションと-mstack オプションの両方で引数 size に対して auto を指定した場合、コンパイラは静的メモリ割り当てを行った後に、残りの未使用メモリをヒープとスタックに均等に割り当てます。

4.6.1.23 stackcall オプション

-mstackcall オプションにより、ルックアップ テーブル方式の関数呼び出しをコンパイラに許可します。

ハードウェア関数のリターンアドレス スタック(5.2.4.1 「関数戻りアドレス用スタック」参照) がフルになった後にさらに関数のネスティングが発生すると、関数の戻り値が破損する可能性があります。このオプションを有効にした場合、スタックがフルになるとコンパイラはルックアップ テーブル方式の関数呼び出しに切り換えます(5.7.7 「関数の呼び出し」参照)。

4.6.1.24 summary オプション

-msummary=*type* オプションは、ビルド完了時に表示されるサマリ内の情報のタイプを選択します。既定値の場合(または mem タイプを選択した場合)、全てのメモリ空間の総メモリ使用量を示すメモリ使用量サマリが表示されます。

psect タイプを指定する事で psect サマリを表示できます。このサマリは、リンカによってグループ化された後の各 psect とそれらのメモリレンジを表示します。表 4-14 に、選択可能なサマリタイプを示します。既定値のサマリ出力は、mem タイプを選択した場合と同じです。

生成された HEX ファイルの SHA ハッシュを表示する事もできます。これらを使う事で、HEX ファイルが前回のビルドから変更されたかどうかを容易に調べる事ができます。

表 4-14. サマリのタイプ

タイプ	内容
psect	psect 名とそれらのリンク先アドレスのサマリを表示します。
mem	使用メモリの簡潔なサマリを表示します(既定値)。
class	各メモリ空間内の全てのクラスを示すサマリを表示します。
hex	アドレスと最終的な出力ファイルを構成する HEX ファイルのサマリを表示します。
file	サマリを画面に表示すると共にファイルにも保存します。
sha1	HEX ファイルの SHA1 ハッシュを表示します。
sha256	HEX ファイルの SHA256 ハッシュを表示します。
xml	サマリ情報を画面に表示し、メインメモリ空間の使用情報を XML ファイルに保存します。
xmlfull	サマリ情報を画面に表示し、全てのメモリ空間の使用情報を XML ファイルに保存します。 このサマリタイプを指定すると、デバイス上のメモリ空間に関する情報(空間の名前、アドレス指定可能単位、サイズ、使用量、空き容量)が XML ファイルに保存されます。

4.6.1.25 undefints オプション

-mundefints=*action* オプションを使うと、未初期化割り込みベクタ(未定義のレガシー低優先度および高優先度ベクタを含む) と割り込みベクタテーブル内のエントリに対するコンパイラの対応を制御できます。

未初期化ベクタが検出されると、コンパイラは警告を生成します。この警告は、対応として ignore を指定する事により無効にできます。

指定可能な全ての対応を表 4-15 に示します。これらはベクタ割り込みコントローラ'(VIC)モジュールを使用するデバイス向けに使える他、VIC を備えないデバイスまたは VIC を備えていてもベクタテーブルを無効にしてレガシーモードで動作するデバイス向けにも使えます。

例えば、割り込み関数へリンクされていないベクタ位置によって実行されるソフトウェア ブレークポイントを使う場合、-mundefints:swbp オプションを指定します。

VIC を使うプロジェクトの場合、既定値の対応は、ベクタテーブルの直後に配置されたリセット命令のアドレスを各未割り当てベクタ位置に書き込みます。その他の全てのデバイスでは、それらのベクタ位置は書き込まれず、他の目的に使えます。

リセット命令またはソフトウェア ブレークポイント命令を実装していないターゲット デバイスに対して reset または swbp を指定した場合、それ自身へジャンプ可能な命令が代わりに書き込まれます。

表 4-15. undefints のサブオプション

対応	VIC を使うデバイス	その他のデバイス
ignore	何も対応しません。ベクタ位置はプログラムコード用に使えます。	何も対応しません。ベクタ位置はプログラムコード用に使えます(既定値)。
reset	各未割り当てベクタ位置にリセット命令のアドレスを書き込みます(既定値)。	各未割り当てベクタ位置にリセット命令を書き込みます。
swbp	各未割り当てベクタ位置にソフトウェア ブレークポイント命令のアドレスを書き込みます。	各未割り当てベクタ位置にソフトウェア ブレークポイント命令を書き込みます。

既定値割り込み要因に対応する未割り当てベクタ位置には、既定の割り込み関数を割り当てる事もできます(5.8.1 「割り込みサービスルーチンの書き方」参照)。

ターゲット デバイスが割り込みをサポートしない場合、-mundefints オプションは無視されます。

4.6.2 出力の種類を制御するためのオプション

表 4-16 に示すオプションは、コンパイラが生成する出力の種類を制御します。表の後で各オプションについて説明します。

表 4-16. コンパイラ出力の種類を制御するオプション

オプション	生成されるファイル
-c	中間生成ファイル
-E	前処理済みファイル
-o *file*	指定された名前を持つ出力ファイル
-S	アセンブリ ファイル
-v	Verbose コンパイル
-xassembler-with-cpp	全てのソースファイルの前処理後の出力
--help	ヘルプ情報のみ
-mprint-devices	デバイス情報のみ
--version	コンパイラ バージョン情報

4.6.2.1 C: 中間生成ファイルへのコンパイル

-c オプションを指定すると、コマンドラインで指定された各ソースファイルから中間生成ファイルが生成されます。

C ソースファイルの場合、コンパイルは構文解析の後に停止し、p-code ファイル(拡張子は.p1)が保存されます。アセンブリ ソースファイルの場合、アセンブラの実行後にコンパイルが終了し、再配置可能オブジェクト ファイル(拡張子は.o)が保存されます。

このオプションを使う事で、make ユーティリティを使ったマルチステップ ビルドが容易に行えます。

4.6.2.2 E: 前処理のみ

-E オプションを指定すると、前処理済み C ソースファイル (モジュールまたは変換ユニットとも呼ぶ) が生成されます。

このオプションを指定した場合、ビルドシーケンスは前処理後に終了し、対応するソースファイルと同じベース名を持つファイル(拡張子は.i)が保存されます。

前処理済みソースファイルを調べる事で、プリプロセッサ マクロが期待通りに展開されたかどうか確認できます。このオプションを使う事で、別のヘッダファイルを必要としない C ソースファイルを生成する事もできます。ソースファイルを仕事仲間やデバイスの技術サポート部門に送信する場合、このオプションを指定すると様々なフォルダに保存されている多数のヘッダファイルを別に送信しなくて済むため便利です。

4.6.2.3 O: 出力ファイルの指定

-o オプションは、出力ファイルのベース名とディレクトリを指定します。

例えば-o main.elf と指定した場合、生成された出力は main.elf という名前のファイルに保存されます。ファイル名と一緒に既存のディレクトリを指定する事もできます。例えば-o build/main.elf と指定した場合、出力ファイル main.elf は「build」フォルダに保存されます。

このオプションを使って出力ファイルのタイプ (フォーマット)を変更する事はできません。

このオプションでは、指定されたファイルのベース名のみが意味を持ちます。このオプションを使って出力ファイルの拡張子を変更する事はできません。

4.6.2.4 S: アセンブリへのコンパイル

-S オプションを指定すると、コマンドラインで指定された各ソースファイルからアセンブリ ファイルが生成されます。

このオプションを指定した場合、コンパイル シーケンスは早期に終了し、対応するソースファイルと同じベース名を持つアセンブリ ファイル(拡張子は.s)が保存されます。

例:

```
xc8-cc -mcpu=16F877A -S test.c io.c
```

上記のオプションを指定した場合、指定された 2 つのソースファイルに対応する 2 つのアセンブリ ファイル (test.s と io.s) が生成されます。

アセンブラの最適化機能が有効である場合、出力ファイルはアセンブラによって最適化されます。無効である場合、コード ジェネレータの出力がそのままファイルに出力されます。最適化済みのアセンブリ ファイルでは、多数のアセンブラ ディレクティブが削除される場合があります。

このオプションを使うと、アセンブリ リストファイルに表示される行番号およびオペコード情報に気を散らされずにコンパイラが出力したアセンブリ コードを確認できるので便利です。コンパイラが出力したアセンブリ ファイルは、ユーザ独自のアセンブリコードのベースとして使う事もできます。

4.6.2.5 V: Verbose コンパイル

-v オプションは、Verbose コンパイルを指定します。

このオプションを指定すると、内部コンパイラ アプリケーションの実行時に、それらの名前とパスおよびコマンドラインから渡された引数が表示されます。

このオプションを使うと、指定したドライバ オプションが期待通りに渡されたかどうかを確認できます。また、どの内部アプリケーションが警告またはエラーを出力しているのかも確認できます。

4.6.2.6 X: ソース言語オプションの指定

-x*language* オプションに続けてソースファイルを指定する事で、そのソースファイルの言語を(ファイルの拡張子に関係なく)指定できます。

本コンパイラが許容する言語を表 4-17 に示します。

表 4-17. 言語オプション

言語	概要
assembler	アセンブリ ソースコード

.....続き	
言語	**概要**
assembler-with-cpp	前処理が必要なアセンブリ ソースコード

例:

```
xc8-cc -mcpu=18f4520 -c -xassembler-with-cpp init.s
```

このオプションは、アセンブリ ソースファイル init.s に対して(拡張子が「.S 」ではなく「.s」であるのにも関わらず) プリプロセッサを実行するようコンパイラに指示します。

4.6.2.7 help

--help オプションを指定すると、ドライバは xc8-cc コンパイラ オプションに関する情報を表示した後に停止します。

4.6.2.8 print-devices

-mprint-devices オプションを指定すると、コンパイラがサポートするデバイス(-mcpu オプションで指定可能なデバイス) のリストが表示されます。このオプションは、コンパイラがリリースされた時に正式にサポートされていたデバイスのみを表示します。デバイスファミリ パック(DFP) によって使用可能となる追加のデバイスは、このリストに表示されません。

コンパイラはデバイスリストを表示した後に停止します。

4.6.2.9 version

--version オプションを指定すると、コンパイラはバージョン情報を表示した後に停止します。

4.6.3 C 言語の方言を制御するためのオプション

表 4-18 に示すオプションは、コンパイラで使う C 言語の方言の種類を定義します。各オプションについては、表の後で説明します。

表 4-18. 方言の制御オプション

オプション	制御内容
-ansi	C 言語規格
-f[no-]signed-char -f[no-]unsigned-char	一般的な char 型の符号指定
-mext=*extension*	有効な言語拡張
-std=*standard*	C 言語規格

4.6.3.1 ansi オプション

-ansi オプションは-std=c90 と等価であり、使用する C 規格を制御します。

4.6.3.2 signed-char オプション

-fsigned-char オプションは一般的な char 型オブジェクトを符号付きにします。

既定値では、一般的な char 型は unsigned char と等価です。

-fsigned-char (または-fno-unsigned-char オプション) は、一般的な char 型を符号付きにします。

一般的な char 型の型指定は、コンパイラに頼らずに char 型オブジェクトの宣言時に明示的に指定する事を推奨します。

4.6.3.3 ext オプション

-mext=*extension* オプションは、コンパイル中に使われる言語拡張を制御します。選択可能な拡張を表 4-19 に示します。

表 4-19. 選択可能な C 言語拡張オプション

extension	C 言語の仕様
xc8	ネイティブ XC8 拡張(既定値)
cci	全ての MPLAB XC コンパイラで使える共通 C インターフェイス(CCI : Common C Interface)

cci 拡張を有効にする事で、全てのソースコードとコンパイラ オプションが CCI に準拠しているかどうか確認するようコンパイラに要求します。このインターフェイスを使ってコードをコンパイルする事により、全ての MPLAB XC コンパイラ間での移植がより容易となります。CCI に準拠しないコードまたはオプションは、コンパイラからの警告により示されます。

4.6.3.4 std オプション

-std=*standard* オプションは C 規格を指定します。コンパイラは、ソースコードがその規格に準拠していると見なします。表 4-20 に選択可能な規格を示します。

MPLAB XC8 は、2 つの規格に対して異なるコンパイラ フロントエンドを使うため、規格を切り換えるとコンパイラの挙動が変化する場合があります。

表 4-20. 選択可能な C 言語規格

standard	サポート
c89 または c90	ISO C90 (ANSI)プログラム(全てのデバイス向けに p1 フロントエンドを使用)
c99	ISO C99 プログラム(PIC18 および拡張ミッドレンジ デバイス向けに clang フロントエンドを使用)

4.6.3.5 unsigned-char オプション

-funsigned-char オプションは一般的な char 型オブジェクトを符号なしにします。

既定値では、一般的な char 型は unsigned char と等価です。

一般的な char 型の型指定は、コンパイラに頼らずに char 型オブジェクトの宣言時に明示的に指定する事を推奨します。

4.6.4 警告とエラーを制御するためのオプション

警告とは、本質的には誤りではないが疑わしい構文を報告したり、誤りがあったかもしれない事を示唆したりするための分析メッセージです。

表 4-21 に示すオプションは、コンパイラが生成するメッセージを制御します。表の後で各オプションについて説明します。

表 4-21. 警告とエラーのオプション

オプション	制御内容
-fmax-errors=*n*	エラーが n 回発生した後にコンパイルを停止します。
-mwarn=*level*	このレベル以上の警告のみ出力します。
-w	全ての警告メッセージを抑制します。
-Wpedantic	規格外の言語拡張を許容します。

4.6.4.1 max-errors オプション

-fmax-errors=*n* オプションは、各コンパイラ アプリケーション(clang フロントエンドを除く) およびドライバが出力するエラーの最大回数を設定します。エラーがこの回数を超えると、それらの実行は停止します。

既定値では、各アプリケーションはエラーメッセージを 20 回表示した後に停止します。例えば-fmax-errors=10 を指定した場合、各アプリケーションは 10 回のエラーの後に停止します。

xc8-ccが使用するメッセージ システムの詳細は、4.5 「コンパイラのメッセージ」を参照してください。

4.6.4.2 warn オプション

`-mwarn=level` オプションは、警告レベルのしきい値を設定します。各警告には-9～9 のレベルが割り当てられています。警告レベルは、曖昧な型変換と構造体に対してコンパイラがどの程度こだわるのかを示します。各警告にはレベルが設定されており、レベルが高いほど警告メッセージの重要度が増します。警告メッセージのレベルが設定されたしきい値を超えると、その警告は表示されます。既定値の警告レベルしきい値は 0 であり、通常の全ての警告が表示されます。

一部の警告メッセージは実行中にエラーが発生する可能性のあるコードを示すため、このオプションを設定する際は注意が必要です。

このオプションは、clang フロントエンドからの警告を制御しません。

本コンパイラのメッセージ システムの詳細は、4.5 「コンパイラ メッセージ」を参照してください。

4.6.4.3 W: 全ての警告の無効化

`-w` オプションは、全ての警告メッセージの表示を無効にします。一部の警告メッセージは実行中にエラーが発生する可能性のあるコードを示すため、このオプションを設定する際は注意が必要です。

4.6.4.4 pedantic オプション

`-Wpedantic` オプションは、C89/90 準拠プログラムのビルド時に、全ての特殊な(非標準の)キーワードが ANSI C に厳格に準拠する事を要求します。

このオプションを指定した場合、C 規格に厳格に準拠するために、非標準キーワードは先頭に 2 個のアンダースコアを含む必要があります(例: `__persistent`)。

4.6.5 デバッグのためのオプション

表 4-22 に示すオプションは、コンパイラが生成するデバッグ出力を制御します。表の後で各オプションについて説明します。

表 4-22. デバッグ オプション

オプション	制御内容
`-f[no-]instrument-functions`	一式の関数プロファイリング情報を生成
`-gformat`	生成するデバッグ情報のタイプ
`-mchp-stack-usage`	スタック使用情報と警告を生成
`-mcodecov`	一式のコードカバレッジ情報を生成
`-save-temps`	コンパイル後も中間生成ファイルを保持するかどうか

4.6.5.1 instrument-functions オプション

`-finstrument-functions` オプションは、出力に分析コードを埋め込む事で、適切なハードウェアによる関数プロファイリングを可能にします。詳細は 5.2.13 「関数プロファイリング」を参照してください。

4.6.5.2 G: デバッグ情報生成オプション

`-g`*`format`* オプションは、ハードウェア ツール向けにデバッグ用の追加情報を生成するようコンパイラに指示します。

サポートするフォーマットを表 4-23 に示します。

表 4-23. サポートするデバッグファイル フォーマット

format	デバッグファイル フォーマット
`-gcoff`	COFF
`-gdwarf-3`	ELF/DWARF リリース 3
`-ginhx32`	拡張されたリニアアドレス レコードを持つ Intel HEX ファイル(64KB を超えるアドレスの使用が可能)
`-ginhx032`	上位アドレスを 0 に初期化する INHX32 ファイル

既定値では、本コンパイラは DWARF リリース 3 ファイルを生成します。

このオプションは、最適化を有効にしても使う事ができるため、最適化済みのコードをデバッグできます。しかし、最適化されたコードでは、ソースコードに存在しない変数が使われていたり、制御フローが予想外に変更されていたり等、驚くような結果が生じる場合があります。

4.6.5.3 スタック ガイダンス オプション

`-mchp-stack-usage` オプションを指定すると、プログラムが解析され、プログラムによって使われる全てのスタックの推定最大深さが報告されます。このオプションは、PRO ライセンスのコンパイラでのみ有効にできます。

本コンパイラが生成するスタック ガイダンス レポートの詳細は、5.2.15 「スタック ガイダンス」を参照してください。

4.6.5.4 codecov オプション

`-mcodecov=`*`suboptions`* オプションは、プログラムの出力に分析コードを埋め込みます。これにより、ソースコード中の実行範囲及び未実行範囲を解析できます。詳細は 5.2.14 「コードカバレッジ」を参照してください。

`suboptions` は必ず指定する必要がありますが、現在サポートしているサブオプションは `ram` のみです。

4.6.5.5 save-temps オプション

`-save-temps` オプションは、コンパイル終了後に一時ファイルを保持するようコンパイラに指示します。

中間生成ファイルは現在作業中のディレクトリに保存され、対応するソースファイルに基づく名前が付けられます。例えば`-save-temps` を指定して `foo.c` をコンパイルすると、`foo.i` および `foo.s` と `foo.o` オブジェクト ファイルが生成されます。

`-save-temps=cwd` オプションは`-save-temps` と等価です。

4.6.6 最適化を制御するためのオプション

コンパイラの最適化を制御するためのオプションを表 4-24 に示します。表の後で各オプションについて説明します。この表には、オプションごとにコンパイラ ライセンスが必要かどうかも示しています。利用可能な最適化の種類については、5.12 「最適化」を参照してください。

これらのオプションの多くは複数の最適化機能の組み合わせを指定し、特定の最適化機能を無効にする場合もあります。このため、全てのオプションを選択してもコードサイズは最小とはなりません。

表 4-24. 一般的な最適化オプション

オプション	ライセンス	最適化方法
`-O0`	不要	最小限の最適化(既定値)
`-O` `-O1`	不要	最適化レベル 1
`-O2`	不要	最適化レベル 2
`-O3`	必要	最適化レベル 3
`-Og`	不要	デバッグ用(最適化は控えめ)
`-Os`	必要	サイズ重視の最適化
`-fasmfile`	不要	アセンブリ ソースファイルの最適化
`-fcacheconst`	必要	const 修飾されたオブジェクトを可能であれば RAM 内の未使用領域に配置
`-flocal`	不要	ローカル最適化
`--nofallback`	不要	選択された最適化レベル以外でのコンパイル検知(ライセンス制約から指定外の最適化を防止する)

4.6.6.1 O0: 最適化レベル 0

`-O0` オプションは、基本的な最適化のみ適用します。これは既定値の最適化レベルです(`-O` オプションを指定しなかった場合、この最適化レベルが適用されます)。

この最適化レベルを指定した場合、コンパイラはコンパイルに要する時間とリソースを削減する事と、デバッグで予測通りの結果が得られる事を目標とします。

4.6.6.2 O1: 最適化レベル 1

-O1 または-O オプションは、レベル 1 の最適化を要求します。

レベル 1 の最適化は、コードサイズと実行時間の削減を目標としますが、デバッグの容易さも適度に保たれます。

この最適化レベルは、ライセンスなしのコンパイラでも利用できます。

4.6.6.3 O2: 最適化レベル 2

-O2 オプションは、レベル 2 の最適化を要求します。

このレベルでは、コンパイラはサイズと速度のトレードオフが生じない最適化処理のほとんど全てを実行します。この最適化レベルは、ライセンスなしのコンパイラでも利用できます。

4.6.6.4 O3: 最適化レベル 3

-O3 オプションは、レベル 3 の最適化を要求します。

このオプションは、実行時間を削減する最適化処理の全てを要求します(サイズの増加は許容)。この最適化レベルは、ライセンスありのコンパイラでのみ利用できます。

4.6.6.5 Og: デバッグ向け最適化オプション

-Og オプションは、デバッグを大きく阻害するような最適化を無効にします。これにより、高速なコンパイルとデバッグの容易さ維持しつつ適度なレベルの最適化が可能です。

4.6.6.6 Os: サイズ重視の最適化オプション

-Os オプションは、サイズを重視した最適化を要求します。

このオプションは、コードサイズが増加しない全ての最適化処理を要求します。

このオプションを指定すると、コードサイズの削減を目的として設計された追加の最適化(手続きの抽象化等)も実行されますが、それらの最適化によって実行速度は低下する可能性があります。

この最適化レベルは、ライセンスありのコンパイラでのみ利用できます。

4.6.6.7 asmfile オプション

-fasmfile オプションは、手書きアセンブリ ソースファイルに対するアセンブラ最適化を有効にします。この最適化は、コード ジェネレータによって C ソースから生成された中間生成アセンブリ出力に対して効果を有しません。このオプションはいつでも選択できますが、実際に実行される最適化はコンパイラ ライセンスの有無に応じて異なります。既定値では、手書きアセンブリ ソースコードを最適化しません。

4.6.6.8 cacheconst オプション

-fcacheconst=[on|off|auto]オプションは、const 修飾されたオブジェクトを未使用データメモリ内でキャッシュ可能にする最適化を有効にし、それらのオブジェクトにはプログラムメモリからではなくデータメモリからアクセスします。この機能は、ライセンスありのコンパイラでのみ利用できます。

多くの場合、プログラムメモリ内のオブジェクトにアクセスするコードは、データメモリ内のオブジェクトにアクセスするコードよりも長くて低速になります。しかし、デバイス内のデータメモリはプログラムメモリに比べて小量であり、データメモリは書き込み可能オブジェクト向けに使える唯一のメモリです。この最適化オプションは、プログラムがデータメモリを使う事を許可し、本来プログラムメモリ内に配置される読み出し専用オブジェクト向けにデータメモリの全ての未使用領域を自動的に使います。

このオプションの引数を off として指定した場合、最適化は完全に無効になり、通常プログラムメモリ内に配置されるオブジェクトはデータメモリ内でキャッシュされません。

この最適化は、最適化レベル 3 を使う場合に既定値により有効となる他、最適化レベル 3 または s で引数 on を指定した場合に有効となり、コンパイラは、通常の方法でオブジェクトをデータメモリに配置します。この手順の後に未使用データメモリ位置が存在する場合、通常はプログラムメモリに保存されるオブジェクトの中から適切な物をこの未使用データメモリ位置に保存するよう試みます。コンパイラは、可能な限り多くのオブジェクトを、各種の要因によって優先順位付けされたデータメモリへとキャッシュします。コンパイラは、データメモリ内のこれらのオブジェクトにアクセスするコードを生成します。通常、これらのコードはプログラムメモリ内のオブジェクトにアクセスするコードよりも高速です。

この最適化を auto モードで有効にした場合(最適化レベル s では既定値で auto は有効、レベル 3 または s で引数 auto を指定した場合にも有効)、コンパイラは引数 on が指定された場合と同じ予備手順を実行しますが、コードサイズが増加しないと判断した場合にのみ、各オブジェクトを実際にキャッシュします。データメモリ内のオブジェク

トにアクセスするコードは、プログラムメモリ内のオブジェクトにアクセスするコードよりも常に小さくなりますが、データメモリ内のオブジェクトには、実行時起動コードによって代入値をコピーする必要があります(5.9.2.1 「オブジェクトの初期化」参照)。ほとんどの場合、データメモリ内のオブジェクトを初期化するための起動コードサイズの増加量は、オブジェクトの各アクセスを実行するコードサイズの削減量に比べるとわずかです。しかし、そうならない場合、このオプションで auto を指定しておくと、コンパイラはオブジェクトをデータメモリへリダイレクトしません。

-fcacheconst オプションを指定しなかった場合のコンパラの挙動:

最適化レベル 0、1、2 を選択した場合、この最適化は無効です(-fcacheconst=off を指定した場合と同じ)。
最適化レベル 3 を選択した場合、この最適化は有効です(-fcacheconst=on を指定したのと同じ)。
最適化レベル s を選択した場合、この最適化は auto モードに設定されます(-fcacheconst=auto を指定したのと同じ)。
最適化レベル 2 以下を使う場合、この最適化を有効にする事はできません。-fcacheconst オプションを指定しても無視されます。

プログラムはソフトウェア スタック(再入可能スタックモデルを使うようコンパイルされる全ての関数向け)またはヒープ(動的メモリ割り当て関数向け)を使うプログラムは、通常これらのコンストラクタ向けに可能な限り多くのデータメモリを要求します。このため、これらのコンストラクタが使われる場合、最適化レベルの選択に関係なく、この最適化は既定値により無効になります。これらのケースでは、-fcacheconst オプションを指定する事により、この最適化を強制的に適用できます(最適化レベルが許容する場合)。ローカル最適化(-flocal オプション)が有効にされると、この最適化は無効になります。この場合、-fcacheconst オプションを使ってこの最適化を有効にする事はできません。

絶対アドレス指定オブジェクト(__at()を使って定義)、セクション内のオブジェクト(__section()を使って定義)、volatile として指定されたオブジェクトは決してキャッシュされません。しかし、その他のオブジェクトの内のどのオブジェクトがこの最適化によってキャッシュされ、どこに配置されるかを勝手に憶測してはいけません。プログラムの開発が進むにつれ、キャッシュされるオブジェクトとそれらの配置は、ビルドするたびに異なる可能性があります。アセンブリコードから C コードで定義されたプログラムメモリ内オブジェクトにアクセスする場合、この最適化を無効にする必要があります。

この最適化により、プログラムはデータメモリの全てを使おうとするため、純粋に const 修飾されたオブジェクトのキャッシュ用に使われるデータメモリの量を示す情報がマップ内で提供されます。重要な注意点として、この最適化を有効にした場合、データメモリは完全に(またはほぼ完全に)使われているように見えるかもしれませんが、プログラム内で追加のオブジェクトを定義できる可能性は残されています。次のビルド時に、コンパイラはこれらのオブジェクトを通常の方法でデータメモリ内に配置しようと試み、それに成功した場合、キャッシュされる const 修飾オブジェクトは減少します。

4.6.6.9 local オプション

-flocal オプションは、プログラムに対して最適化が適用される範囲を制限します。

このオプションは、ライブラリまたは個々のプログラム モジュールに対して OCG (Omniscient Code Generation)最適化を適用しますが、これらの最適化のスコープはライブラリまたはモジュール内のコードのみに制限されます。通常、あるモジュール内の最適化は他のモジュールまたはライブラリ内のコードの影響を受けます。しかし、そのような状況を避けたい場合もあります。このオプションを有効にしてコンパイルしたソースの出力は通常よりも大きくなりますが、このオプションを有効していない他のプロジェクト コードが変更されても、再ビルド時にほとんど変化しません。配布されて様々なアプリケーションで使われる検証済みコードでは、ビルドのたびに出力が変化したのでは不都合な場合があります。

ローカル最適化を使うビルドコマンドで指定された全てのコードは 1 つのグループを構成します。別々のビルドコマンドを使ってソースコードをビルドする事により、必要な数のグループを作成できます。ローカル最適化を使わずにビルドされた全てのコードは、既定値(制約なし) のグループに含まれます。本コンパイラと一緒に提供される標準ライブラリは、ローカル最適化を無効にしてビルドされており、常にこの既定値グループに含まれます。

ローカル最適化を有効にすると、多くの最適化のスコープは制限されますが、必ずしも最適化そのものが無効になるわけではありません。最適化は各グループ内で実行され、別のグループに含まれるコードに依存する最適化は行われません。例えば、他のグループに含まれる common コードシーケンスの抽象化は行われませんが、同じグループに含まれるシーケンスの抽象化は行われます。グループに含めるソースコード モジュールは極少数に制限できるため(これらは通常の方法で 1 つのライブラリへとビルド可能)、プロジェクトの大半を完全に最適化できます。

既定値では、このオプションは無効です。ミッドレンジおよび PIC18 デバイスでは、ビルド時にこのオプションを有効にできます。非対応のデバイスでこの最適化を選択すると、エラーメッセージが出力されます。

ローカル最適化を使ってコードをビルドした場合、そのグループ内で定義されている全ての変数は、near で修飾されない限りバンク型メモリに配置されます。バンク選択命令は、ほとんどの場合出力されると考えるべきです。関数呼び出しの前後のページ選択命令は常に出力され、変数伝播は無効になり、浮動小数点型のサイズは float および double 型の両方で 32 ビットに固定され(これはプログラム全体に適用される)、ポインタのサイズはそれらの定義

に基づいて固定可能です(5.3.6.2 「ポインタ ターゲット修飾子」参照)。一部のアセンブリ最適化(手続きの抽象化、ル ーチンのインライン化、psect のマージ、ピープホール最適化等)も制限されます。

4.6.6.10 nofallback オプション

--nofallback オプションを使うと、-O オプションによる指定よりも低レベルの最適化が気付かずに実行されてしまうのを防ぐ事ができます。

例えば、ライセンスなしのコンパイラに対してレベル s の最適化を実行するよう要求した場合、--nofallback オプションを指定しないと、低レベルの最適化が実行されます。このオプションを指定した場合、指定されたレベルの最適化が実行できない時にコンパイラはエラーを出力し、コンパイルを停止します。従って、このオプションを指定しておけば、誤ってライセンスなしのコンパイラでビルドしてしまう事を防止できます。

4.6.7 プリプロセッサを制御するためのオプション

プリプロセッサを制御するためのオプションを表 4-25 に示します。表の後で各オプションについて説明します。

表 4-25. プリプロセッサ オプション

オプション	制御内容
-D*macro* -D*macro*=*text*	プリプロセッサ マクロの定義
-M	依存情報の生成
-mcmacros	C プログラムに関連する定義済みマクロ
-MD	依存情報の生成
-MF	依存性情報の書き込み先
-MM	依存情報の生成
-MMD	依存情報の生成
-U*macro*	プリプロセッサ マクロの未定義化
-Wp,*option*	プリプロセッサに渡すオプション
-Xpreprocessor *option*	プリプロセッサに渡すオプション

4.6.7.1 D: マクロの定義

-D*macro* オプションにより、プリプロセッサ マクロを定義できます。さらに、-D*macro*=*text* の形式で指定する事により、ユーザ定義置換文字列をマクロと一緒に定義する事もできます。オプション名とマクロ名の間に 1 つのスペース文字を挿入しても構いません。

例として、マクロ名の後で置換テキストを指定しない場合、-D*macro* オプションは *macro* で指定された名前を持つプリプロセッサ マクロを定義し、その置換テキストを 1 として指定します。これは、コンパイルされる各モジュールの先頭に#define *macro* 1 を配置するのと等価です。

-D*macro*=*text* は、*macro* で指定された名前と *text* で指定された置換テキストを持つプリプロセッサ マクロを定義します。これは、コンパイルされる各モジュールの先頭に#define *macro text* を配置するのと等価です。

このオプションをどちらの形式で指定も、生成される識別子(マクロ名) の定義は#ifdef または#ifndef ディレクティブによって確認できます。

例えば、-DMY_MACRO (または-D MY_MACRO)を指定して以下のコードをビルドした場合、

```
#ifdef MY_MACRO
int input = MY_MACRO;
#endif
```

int 変数 input の定義がコンパイルされ、この変数には値 1 が代入されます。

-DMY_MACRO=0x100 を指定して上記のサンプルコードをコンパイルした場合、変数定義は最終的に以下の通りにコンパイルされます。

```
int input = 0x100;
```

置換テキストの使われ方については、5.13.1.1 「プリプロセッサの算術演算」を参照してください。

マクロを C 文字列リテラルとして定義する場合、オペレーティング システム内の解釈の問題を回避する必要があります。C 文字列 "hello world" (ダブルクオートを含む)を Windows 環境内で渡す場合、"-DMY_STRING=\\\"hello world\\\"" を使います(置換テキストはスペースを含むため、オプション全体をダブルクオートで囲む必要があります)。Linux または Mac OS X 環境では、-DMY_STRING=\"hello\ world\" を使います。

コマンドライン上の全ての-D オプションは、どの-U オプションよりも前に処理されます。

4.6.7.2 M: make 規則の生成

-M オプションは、make に適した規則(各オブジェクト ファイルの依存情報を記述)を出力するよう、プリプロセッサに指示します。

プリプロセッサは、各ソースファイルに対して 1 つの make 規則を出力します。この規則のターゲットはそのソースファイル向けのオブジェクト ファイル名であり、そのソースファイルの依存情報はそのソースファイルがインクルードする全てのヘッダファイルです。この規則は 1 行で書く事もできますが、長い場合はバックスラッシュ+改行シーケンスを使って複数行に書く事もできます。

依存情報は、拡張子.d が付いたファイルに出力され、前処理後にコンパイルは停止します。

4.6.7.3 cmacros オプション

-mcmacros オプションは、プロジェクトをビルドする際にコマンドライン ドライバが C プログラムに関係するプリプロセッサ マクロを定義する事を要求します。これは、ドライバの既定値動作です。

このオプションを-mno-cmacros と指定すると、C 関連のプリプロセッサ マクロは定義されなくなります。この場合、デバイスに関連するマクロ(例: 16F1937、_ROMSIZE 等) は定義されますが、C プログラムに関連するマクロ(例: __XC8、__OPTIM_FLAGS、__SIZEOF_CHAR__等)は定義されません。-mno-cmacros オプションを指定する場合、注意が必要です。プロジェクトの動作がこれらのマクロの存在またはこれらのマクロの値に依存しない事を確認する必要があります。

4.6.7.4 MD: 依存情報をファイルに書き込むためのオプション

-MD オプションは、依存情報をファイルに書き込みます。

このオプションは-M に似ていますが、依存情報がファイルに書き込まれた後もコンパイルは継続されるという点で異なります。依存情報が書き込まれたファイルには、対応するソースファイル名の拡張子を.d に変更した名前が付けられます。

4.6.7.5 MF: 依存情報ファイルを指定するためのオプション

-MF *file* オプションは、-M または-MM オプションによる依存情報の書き込み先ファイルを指定します。-MF オプションを指定しないと、プリプロセッサは前処理出力と同じ出力先に依存情報を出力します。

-MF をドライバ オプション-MD または-MMD と一緒に使った場合、既定値の依存情報出力ファイルは上書きされます。

4.6.7.6 MM: Quated ヘッダ向けに make 規則を生成するためのオプション

-MM オプションは-M と同様のタスクを実行しますが、出力にシステムヘッダは含まれません。

4.6.7.7 MMD: ユーザヘッダ向け make 規則を生成するためのオプション

-MMD オプションは、-MD と同様のタスクを実行しますが、出力にはユーザ ヘッダファイルのみが含まれます。

4.6.7.8 U: マクロの未定義化

-U*macro* オプションは、マクロ macro を未定義化します。

-D を使って定義されたマクロは、このオプションによって未定義化されます。全ての-U オプションは、全ての-D オプションの後に評価されます。

4.6.7.9 Wp:プリプロセッサに直接オプションを渡すためのオプション

-Wp,*option* オプションを指定すると、*option* がプリプロセッサに渡されて、プリプロセッサ オプションとして解釈されます。option は、コンマで区切る事で複数のオプションを指定できます。

4.6.7.10 Xpreprocessor オプション

-Xpreprocessor *option* オプションを指定すると、*option* がプリプロセッサに渡されて、プリプロセッサ オプションとして解釈されます。これにより、コンパイラが認識しないシステム固有のプリプロセッサ オプションを指定できます。

4.6.8 パーサー向けのオプション

パーサーの動作を制御するためのオプションを表 4-26 に示します。表の後で各オプションについて説明します。

表 4-26. パーサー オプション

オプション	制御内容
-Xparser *option*	パーサーに渡すオプション

4.6.8.1 Xparser オプション

-Xparser *option* オプションは、その引数をパーサーに直接渡します。例えば-Xparser -v は、パーサーを Verbose モードで実行します。-Xp1 および-Xclang オプションは、このオプションの代替として使えます。

4.6.9 アセンブラ向けのオプション

アセンブラの動作を制御するためのオプションを表 4-27 に示します。表の後で各オプションについて説明します。

表 4-27. アセンブリ オプション

オプション	制御内容
-Wa,*option*	アセンブラに渡すオプション
-Xassembler *option*	アセンブラに渡すオプション

4.6.9.1 Wa: アセンブラに直接オプションを渡すためのオプション

-Wa,*option* オプションは、*option* 引数をアセンブラに直接渡します。*option* は、コンマで区切る事で複数のオプションを指定できます。例えば-Wa,-a は、-a オプションをアセンブラに渡します(アセンブリ リストファイルの生成を要求)。

4.6.9.2 Xassembler オプション

-Xassembler *option* オプションを指定すると、*option* がアセンブラに渡されて、アセンブラ オプションとして解釈されます。このオプションにより、コンパイラが認識しないか-Wa オプションでは解析できないシステム固有のアセンブラ オプションを指定できます。

4.6.10 アセンブラ オプション

よく使われるアセンブラ オプションを表 4-28 に示します。

表 4-28. アセンブラ オプション

オプション	制御内容
-Wa,-a	アセンブリ リストファイルの生成

4.6.11 リンクのためのオプション

リンカの動作を制御するためのオプションを表 4-29 に示します。表の後で各オプションについて説明します。

-c、-S、-E のいずれかを指定した場合、リンカは実行されません。

表 4-29. リンクオプション

オプション	制御内容
-l*library*	スキャンするライブラリ ファイルを指定します。
-mc90lib	C90 準拠ライブラリをリンクするかどうか指定します。
-m[no-]clink	C プログラムに関連するリンカオプションを生成および使用します。
-mserial=*options*	出力にシリアル番号を挿入します。
-nodefaultlibs	ライブラリコードをプロジェクトとリンクするかどうか指定します。
-nostartfiles	実行時起動モジュールをリンクするかどうか指定します。

.....続き

オプション	制御内容
-nostdlib	ライブラリコードと起動コードをプロジェクトとリンクするかどうか指定します。
-Wl,*option*	リンカに渡すオプションを指定します。
-Xlinker *option*	リンカに渡すシステム固有オプションを指定します。
--fill=*options*	未使用メモリに値を書き込みます。

4.6.11.1 L: ライブラリ ファイルを指定するためのオプション

-l*library* オプションは指定されたファイル(拡張子は.a)を検索し、リンク時に未解決のシンボル用にこのライブラリ アーカイブをスキャンします。

標準のシステム ディレクトリに加えて、-L で指定されたディレクトリが検索されます。

-l オプションを使った場合(例: -lmylib)とコマンドラインでファイル名を指定した場合(例: mylib.a) の唯一の違いは、コンパイラは-l で指定されたライブラリを複数のディレクトリ(-L オプションで指定) 内で検索するという事です。

4.6.11.2 c90lib オプション

-mc90lib オプションは、C90 言語規格と互換のライブラリセットでリンクします。既定値では、C99 互換ライブラリセットがリンクされます。

このオプションは、プロジェクト内のソースコード向けに使われる言語仕様を変更しません。言語仕様を変更するには-std オプションを使います(4.6.3.4 「std オプション」参照)。

4.6.11.3 clink オプション

-mclink オプションは、プロジェクトをビルドする際に、C プログラムに関連するリンカオプションをコマンドライン ドライバが生成するよう要求します。これは、ドライバの既定値動作です。

このオプションを-mno-clink の形式で指定する事により、C プログラム向けのリンカオプションを手動で指定できます。この場合もリンカクラス オプションは定義されますが、psect をクラスに配置する全てのオプション(例: -p リンカオプション) は定義されません。-mno-clink オプションを指定する場合、注意が必要です。プロジェクトが正しく動作するよう、適切なリンカオプションを手動で指定する必要があります。

4.6.11.4 serial オプション

-mserial=*options* オプションを使うと、16 進コードをプログラムメモリ内の特定アドレスに保存できます。例えば、このオプションを使ってプログラムメモリ内にシリアル番号を配置する事ができます。

保存するデータのバイト幅は、オプション内の 16 進コード パラメータのバイト幅によって決まります。プログラム メモリ アドレス 1000h に 1 バイトの値 0 を保存する場合、-mserial=00@1000 と指定します。同じ値を 4 バイト値として保存する場合、-mserial=00000000@1000 と指定します。

このオプションは、Hexmate の-serial オプションと機能的に同じです。このオプションを使った制御の詳細は 8.2.2.21 「serial」を参照してください。

ドライバは値が保存された位置でラベルも定義します。この値は C コードから_serial0 として参照できます。このシンボルにアクセスするには、例えば以下の通りに宣言しておく必要があります。

```
extern const int _serial0;
```

4.6.11.5 nodefaultlibs オプション

-nodefaultlibs オプションは、標準システム ライブラリがプロジェクトにリンクされる事を防ぎます。ユーザが指定したライブラリだけをリンカに渡します。

4.6.11.6 nostartfiles オプション

-nostartfiles オプションは、実行時起動モジュールがプロジェクトにリンクされる事を防ぎます。

4.6.11.7 nostdlib オプション

-nostdlib オプションは、標準のシステム起動ファイルおよびライブラリがプロジェクトにリンクされる事を防ぎます。リンカには起動ファイルは渡されず、ユーザが指定したライブラリだけが渡されます。

4.6.11.8 Wl: リンカに直接オプションを渡すためのオプション

`-Wl,`*`option`* オプションを指定すると、*`option`* はリンカ アプリケーションに渡されて、リンカオプションとして解釈されます。*`option`* は、コンマで区切る事で複数のオプションを指定できます。このため、例えば `-pcodeStart,codeEnd` のように複数の psect 名をコンマで区切るリンカオプションを渡す事はできません。その場合、`-Xlinker` オプションが使えます。

4.6.11.9 xlinker オプション

`-Xlinker` *`option`* オプションを指定すると、*`option`* はリンカに渡されて、リンカ オプションとして解釈されます。これにより、コンパイラが認識しないシステム固有のリンカオプションを渡す事ができます。

例: `-Xlinker -presetVec=0h,intCode=04h`

4.6.11.10 fill オプション

`--fill=`*`options`* オプションを使うと、各種の方法で未使用メモリに指定した値を書き込む事ができます。

このオプションは、Hexmate の `-fill` オプションと機能的に同じです。このオプションを使った制御の詳細は 8.2.2.12 「fill」を参照してください。

4.6.12 リンカオプション

よく使われるリンカオプションを表 4-30 に示します。

表 4-30. リンカオプション

オプション	制御内容
`-Wl,--[no-]data-init`	実行時起動コード内で C オブジェクトをクリアおよび初期化します。
`-Wl,-Map=`*`mapfile`*	リンカ マップファイルを生成します。

4.6.13 ディレクトリを検索するためのオプション

ディレクトリ検索動作を制御するためのオプションを表 4-31 に示します。表の後で各オプションについて説明します。

表 4-31. ディレクトリ検索オプション

オプション	制御内容
`-I`*`dir`*	プリプロセッサ インクルード ファイルの検索ディレクトリを指定します。
`-L`*`dir`*	ライブラリの検索ディレクトリを指定します。
`-nostdinc`	ヘッダの検索ディレクトリを指定します。

4.6.13.1 I: インクルード ファイルの検索パスを指定するためのオプション

`-I`*`dir`* オプションは、ヘッダファイルの検索ディレクトリのリストの先頭に、*`dir`* で指定されたディレクトリを追加します。オプション名とディレクトリ名の間に 1 つのスペース文字を挿入しても構いません。

このオプションは、相対パスでも絶対パスでも指定できます。また、このオプションを複数回指定する事で追加の検索ディレクトリを指定できます。その場合、ディレクトリは指定された順番に(左から右へ) 検索されます。標準のシステム ディレクトリは、このオプションで指定されたディレクトリの後で検索されます。

Windows OS の場合、ディレクトリ区切り記号のバックスラッシュ文字は意図せぬエスケープ シーケンスを形成してしまう可能性があります。ディレクトリ区切り文字で終わるインクルード ファイルパスをダブルクオートで囲んで指定する場合、`-I "E:\"`ではなく`-I "E:\\"` として指定する必要があります(エスケープシーケンス「\"」として解釈される事を防ぐため)。MPLAB X IDE はプロジェクト プロパティ内でユーザが指定したインクルード ファイルパスをダブルクオートで囲むので注意が必要です。また、検索パスは出力ディレクトリに対して相対的である事にも注意が必要です(プロジェクト ディレクトリに対して相対的なのではありません)。

4.6.13.2 L: ライブラリの検索パスを指定するためのオプション

`-L`*`dir`* オプションを使うと、`-l` オプションで指定されたライブラリ ファイルの検索パスに追加するディレクトリを指定できます。コンパイラは自動的に標準のライブラリ パスを検索するため、このオプションだけでユーザ独自のライブラリにリンクできます。

4.6.13.3 nostdinc オプション

-nostdinc オプションは、ヘッダファイル向けに標準システム ディレクトリがプリプロセッサによって検索される事を防ぎます。-I オプションによって指定されたディレクトリ(および、適切であれば現在作業中のディレクトリ)だけが検索されます。

4.6.14 コード生成規則のオプション

表 4-32 に示すオプションは、コード生成時に適用されるデバイスに非依存の規則を制御します。各オプションについては、表の後で説明します。

表 4-32. コード生成規則オプション

オプション	制御内容
-f[no-]short-double	double 型のサイズ
-f[no-]short-float	float 型のサイズ

4.6.14.1 short-double オプション

-fshort-double オプションは、double 型のサイズを制御します。

C99 規格向けにビルドする場合、全ての浮動小数点型は IEEE754 32 ビット フォーマットである事が必要です。C90 規格向けにビルドする場合、このオプションを使って double 型オブジェクト向けに 24 ビット浮動小数点フォーマットを明示的に要求できます。このオプションを-fno-short-double と指定する事で、double 型を完全な 32 ビット IEEE754 フォーマットに変更できます。このオプションの選択は、プログラムの全てのモジュールに対して一貫している事が必要です。

4.6.14.2 short-float オプション

-fshort-float オプションは、float 型のサイズを制御します。

C99 規格向けにビルドする場合、全ての浮動小数点型は IEEE754 32 ビット フォーマットである事が必要です。C90 規格向けにビルドする場合、このオプションを使って float 型オブジェクト向けに IEEE754 フォーマットの 24 ビット形態を明示的に要求できます。このオプションを-fno-short-float と指定する事で、float 型を完全な 32 ビット IEEE754 フォーマットに変更できます。このオプションの選択は、プログラムの全てのモジュールに対して一貫している事が必要です。

4.7 MPLAB X IDE 統合開発環境

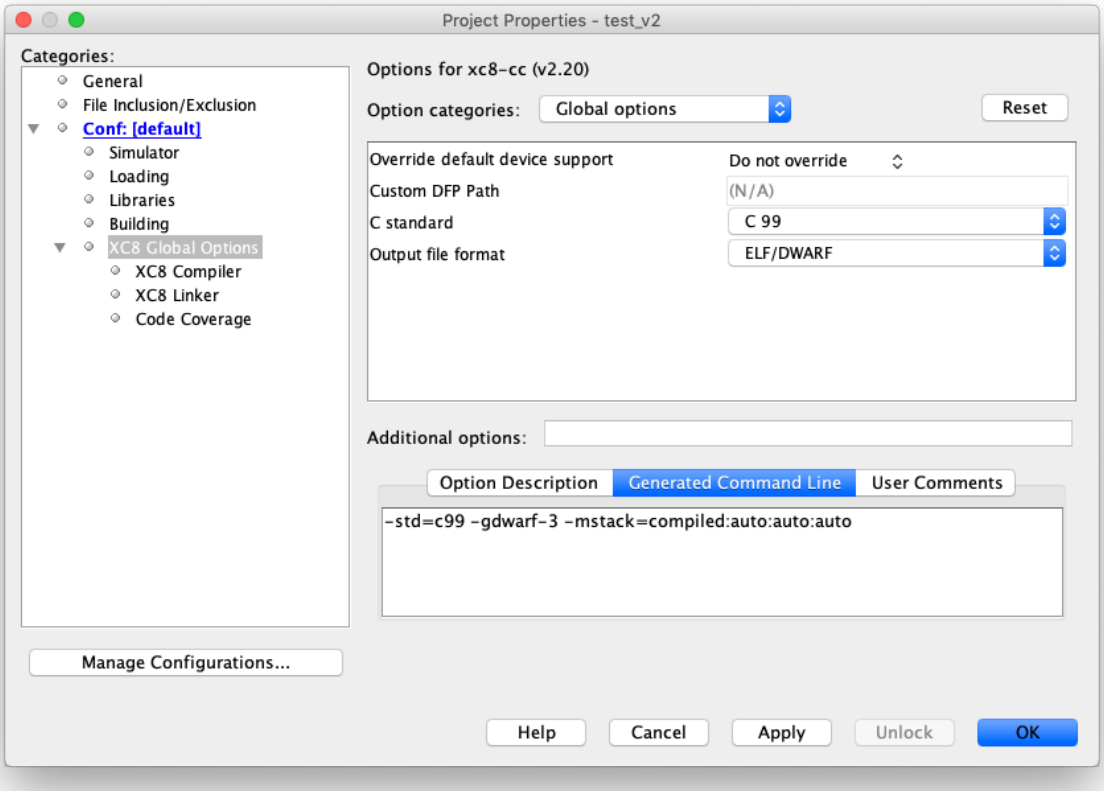
MPLAB X IDE から 8 ビット言語ツールを統合および制御する事により 8 ビット PIC MCU ファミリデバイス向けのアプリケーション コードを GUI 環境で開発できます。

MPLAB X IDE をインストールし、MPLAB XC8 C コンパイラ向けにプロジェクトを作成および設定する方法は、『MPLAB® X IDE ユーザガイド』を参照してください。

4.7.1 MPLAB X IDE オプションと等価のオプション

以下では MPLAB X IDE の**[Project Properties]** による制御と MPLAB XC8 コマンド ドライバ オプションの間の対応について説明します。各オプションの詳細は、本書内の対応するセクションを参照してください。MPLAB X IDE 内でいずれかのオプションをクリックすると、ダイアログの下部にある**[Option Description]** フィールドにオンラインヘルプとサンプルコードが表示されます。

4.7.1.1 XC8 Global Options - Global options



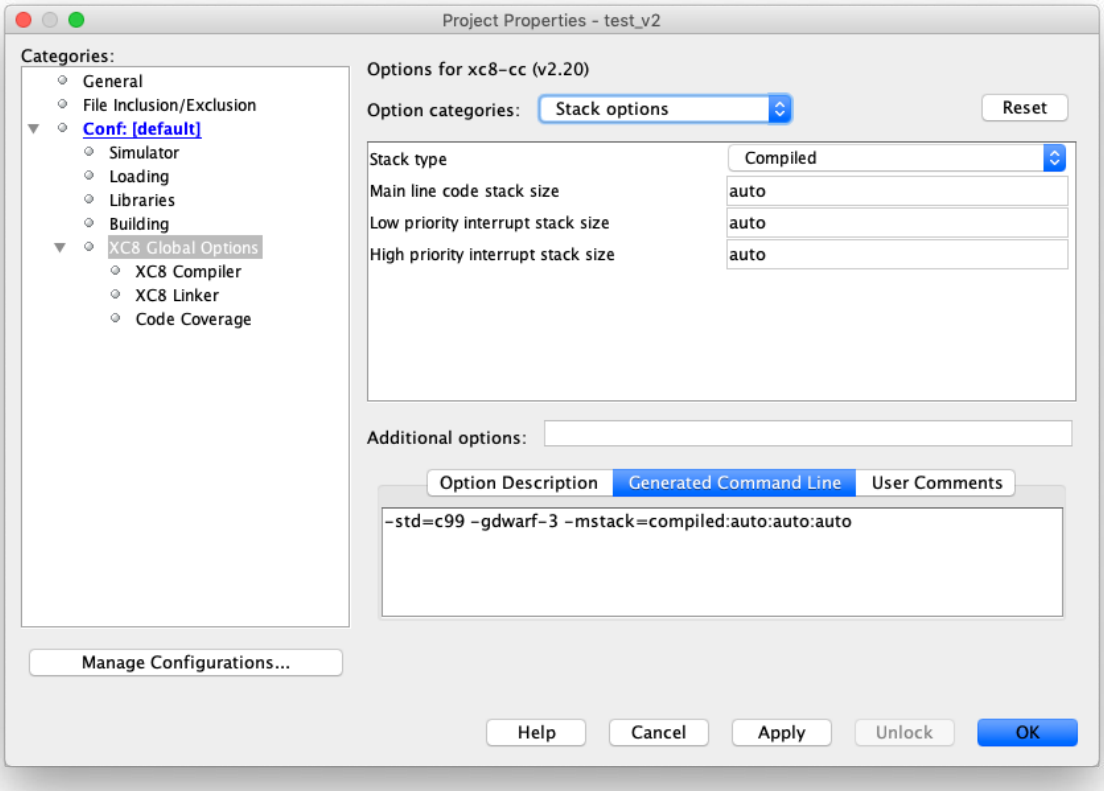
Override default device support これを選択する事で、デバイスファミリ パック(DFP)の管理方法を指定できます。**[Do not override]**を選択した場合、MPLAB X IDE は選択可能な DFP のリストを提供します。ユーザが自分でダウンロードした DFP を使う場合、**[User specified location]**を指定した後に**[Customer DFP path]** フィールドに DFP へのパスを入力します。**[Compiler location]** を選択する事で、コンパイラに同梱された DFP を使う事もできます。

Custom DFP path **[Override default device support]** オプションで**[User specified location]**を選択した場合、このフィールドに DFP へのパスを入力します。

C standard 使用する言語規格を選択します。4.6.3.4 「std オプション」を参照してください。

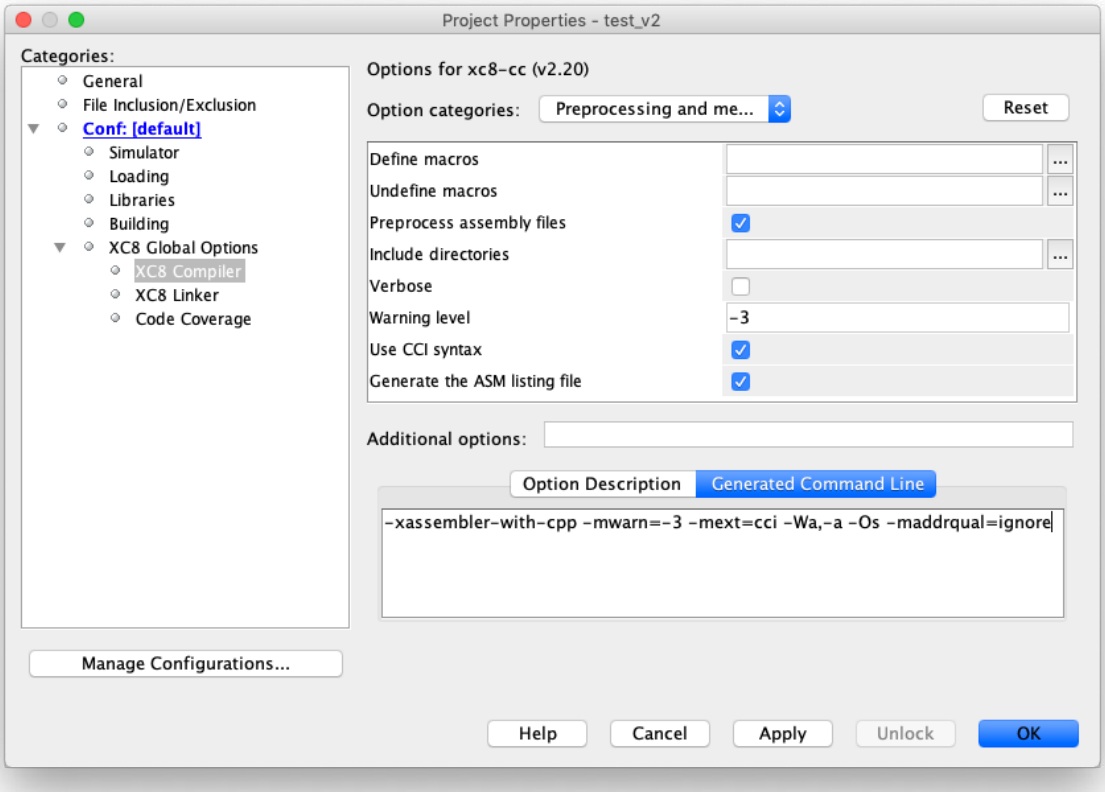
Output file format これは出力ファイルのタイプを選択します。4.6.5.2 「G: デバッグ情報生成オプション」を参照してください。

4.7.1.2 XC8 Global Options - Stack options



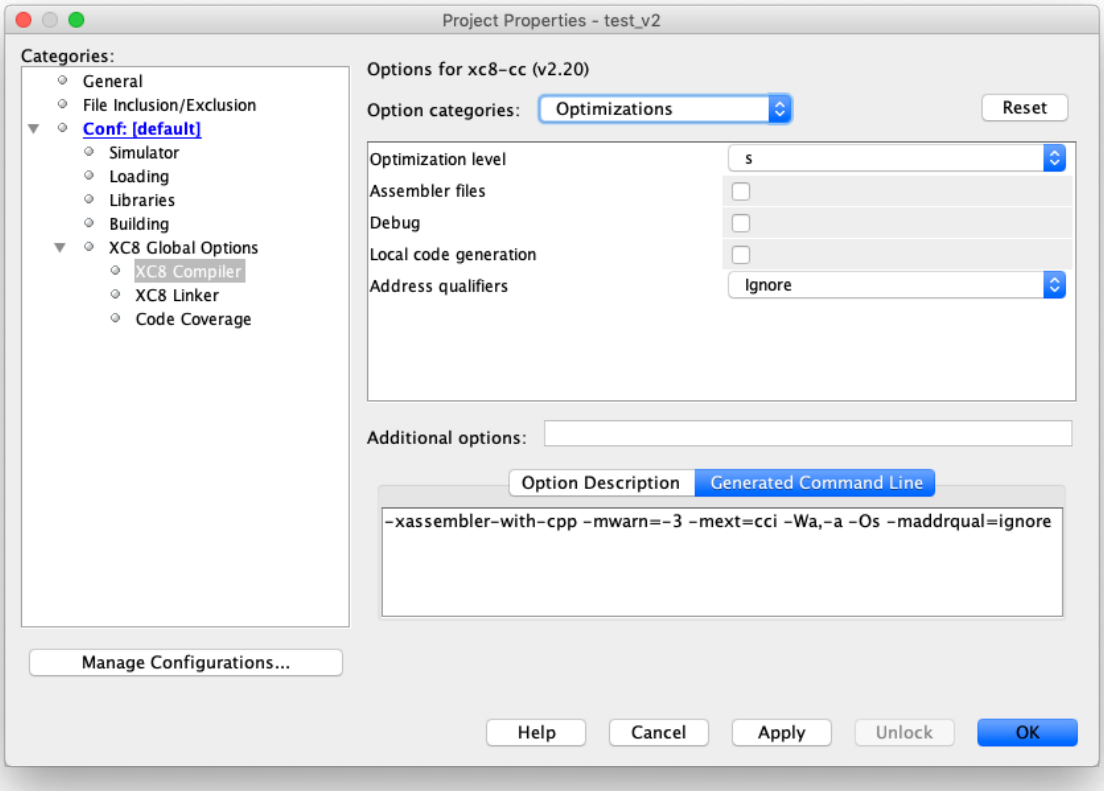
Stack options このダイアログ内の全てのフィールドは、使用するデータスタックの選択とその設定に関連します。4.6.1.22 「stack オプション」を参照してください。

4.7.1.3 XC8 Compiler - Preprocessing and messaging オプション



Define macros	このフィールドでは、プリプロセッサ マクロを定義できます。4.6.7.1 「D: マクロの定義」を参照してください。
Undefine macros	このフィールドでは、プリプロセッサマクロを未定義化できます。4.6.7.8 「U: マクロの未定義化」を参照してください。
Preprocess assembly files	このチェックボックスは、アセンブリ ソースファイルを前処理するために使います。4.6.9.2 「Xassembler オプション」を参照してください。
Include directories	このフィールドでは、ヘッダファイルの検索ディレクトリを指定できます。4.6.13.1 「I: インクルード ファイルの検索パスを指定するためのオプション」を参照してください。
Verbose	このチェックボックスは、ビルド コマンドラインを表示します。4.6.2.5 「V: Verbose コンパイル」を参照してください。
Use CCI syntax	このチェックボックスは、CCI 言語拡張の適用を要求します。4.6.3.3 「ext オプション」を参照してください。
Generate the ASM listing file	このチェックボックスは、アセンブリ リストファイルを生成します。4.6.10 「アセンブラ オプション」を参照してください。

4.7.1.4 XC8 Compiler - Optimizations オプション



Optimization level	プログラムの最適化レベルを選択します。4.6.6 「最適化を制御するためのオプション」を参照してください。
Assembly files	アセンブリ ソースファイルの最適化を有効にします。4.6.6.7 「asmfile オプション」を参照してください。
Debug	コードのデバッグを困難にするような過剰な最適化を抑制します。4.6.6.5 「Og: デバッグ向け最適化オプション」を参照してください。
Local code generation	一部の最適化の適用範囲を制限します。4.6.6.9 「local オプション」を参照してください。
Address qualifiers	C ソースコード内の非標準メモリ識別子に対するコンパイラの応答を制御します。4.6.1.1 「addrqual オプション」を参照してください。

4.7.1.5 XC8 Linker - Runtime オプション

図 4-2. PIC18 デバイス向けリンカ ランタイム オプション

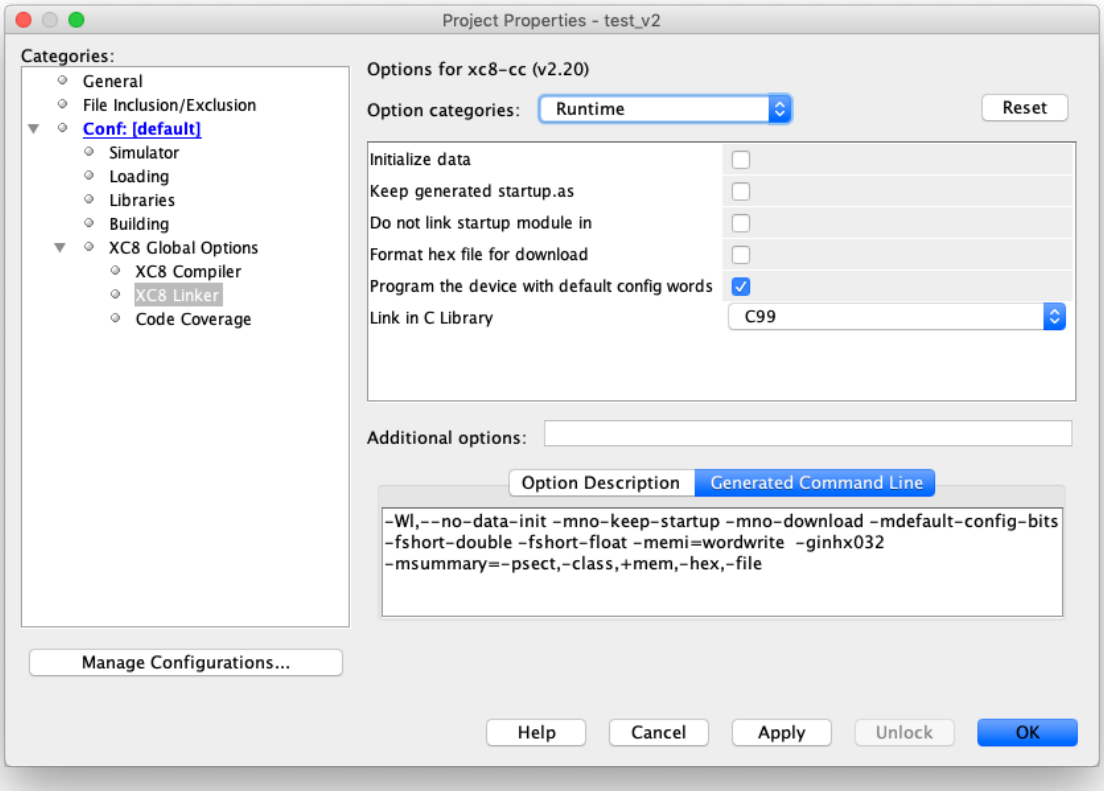
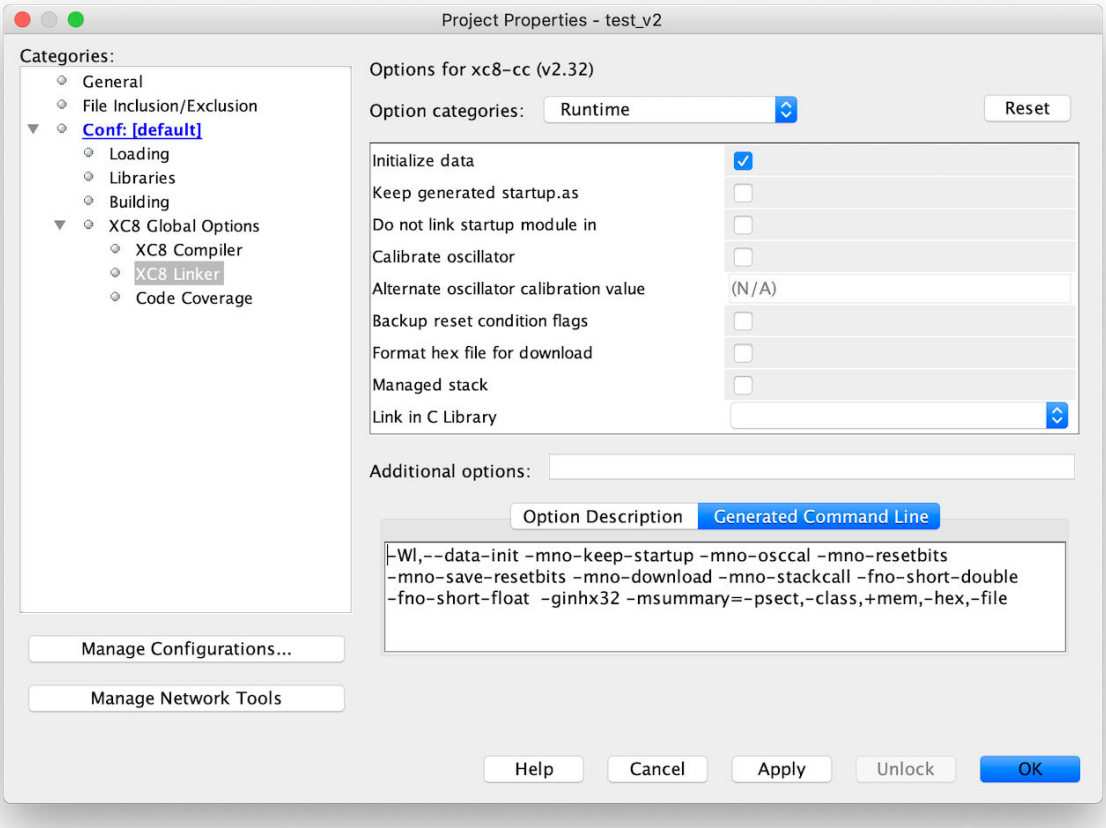


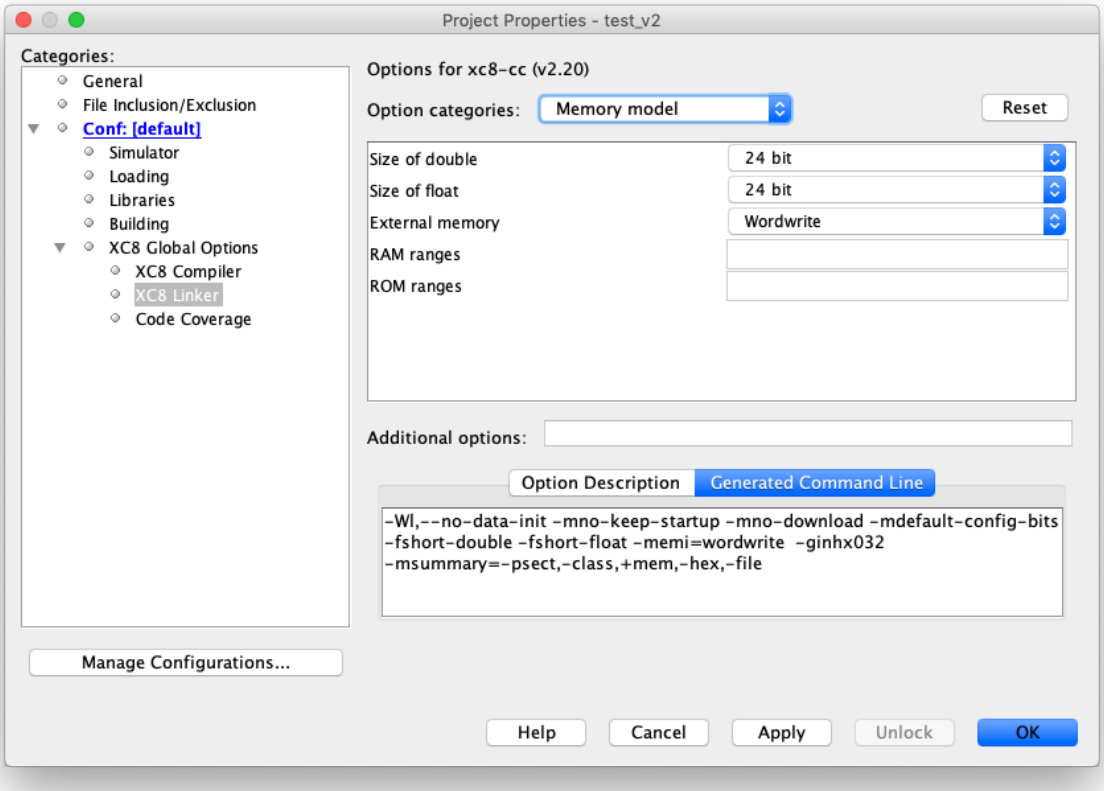
図 4-3. ミッドレンジおよびベースライン デバイス向けリンカ ランタイム オプション



Initialize data	初期化済みC オブジェクトに開始値を代入するかどうか指定します。4.6.12 「リンカオプション」を参照してください。
Keep generated startup.as	コンパイル後に実行時起動モジュールを保持するかどうか指定します。4.6.1.13 「keep-startup オプション」を参照してください。
Do not link startup module in	実行時起動モジュールをプログラムにリンクするかどうか指定します。4.6.11.6 「nostartfiles オプション」を参照してください。
Calibrate oscillator	オシレータ校正定数をプログラムにリンクされた実行時起動モジュールによって適用するかどうかを指定します。一部のデバイスはこの校正機能を備えていません。4.6.1.15 「osccal オプション」を参照してください。
Alternate oscillator calibration value	デバイスに書き込み済みのオシレータ校正定数とは異なる校正定数を指定できます。一部のデバイスはこの校正機能を備えていません。4.6.1.16 「oscval オプション」を参照してください。
Backup reset condition flags	実行時起動コードがステータス レジスタの状態を保存するかどうか指定します。 保存する事で、リセットの原因を後で調べる事ができます。このオプションは、PIC18 デバイスでは利用できません。4.6.1.19 「resetbits オプション」を参照してください。
Format hex file for download	最終的な HEX ファイルの特殊なフォーマットを制御します。4.6.1.8 「download オプション」を参照してください。
Managed stack	ルックアップ テーブル方式の関数呼び出しを採用するかどうか指定します。4.6.1.23 「stackcall オプション」を参照してください。

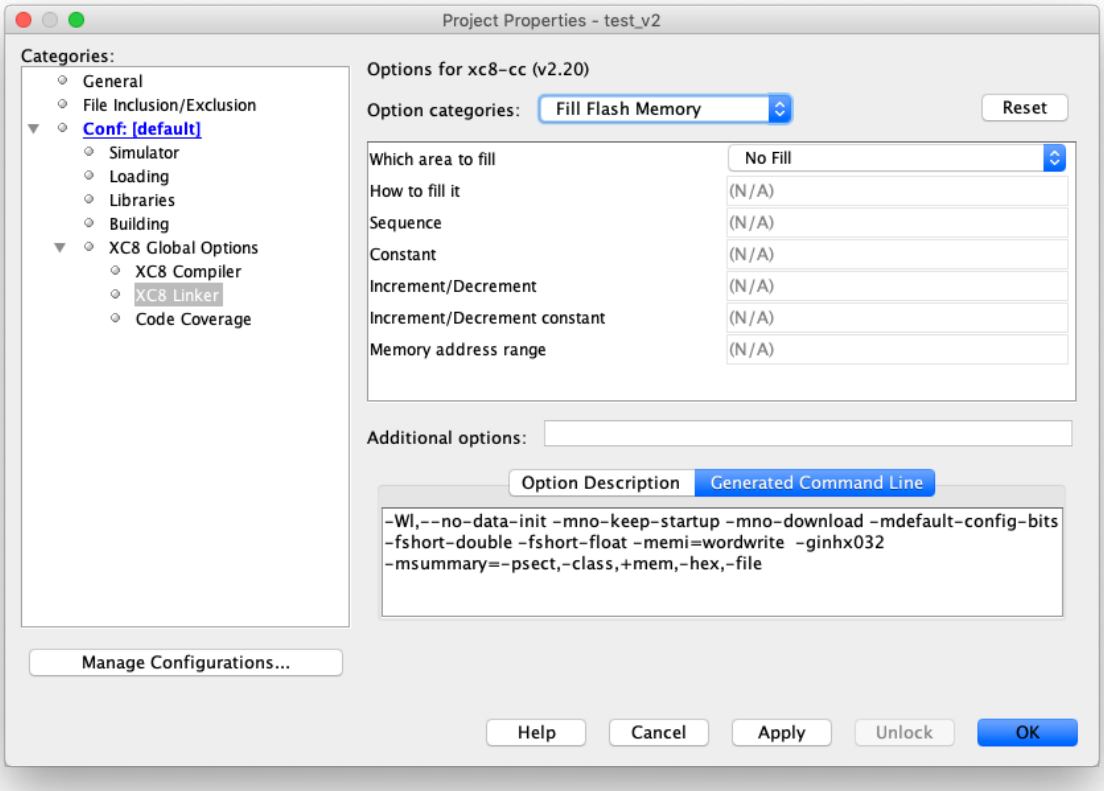
Program the device with default config words	未指定のコンフィグレーション ビットを既定値に設定するかどうか指定します。4.6.1.4 「config オプション」を参照してください。
Link in C library	プログラムにリンクするライブラリのインクルードと言語規格を指定します。これは `-mc90lib` オプション(4.6.11.2 「c90lib オプション」参照)と `- nodefaultlibs` オプション(4.6.11.5 「nodefaultlibs オプション」参照)を使います。

4.7.1.6 XC8 Linker - Memory model options



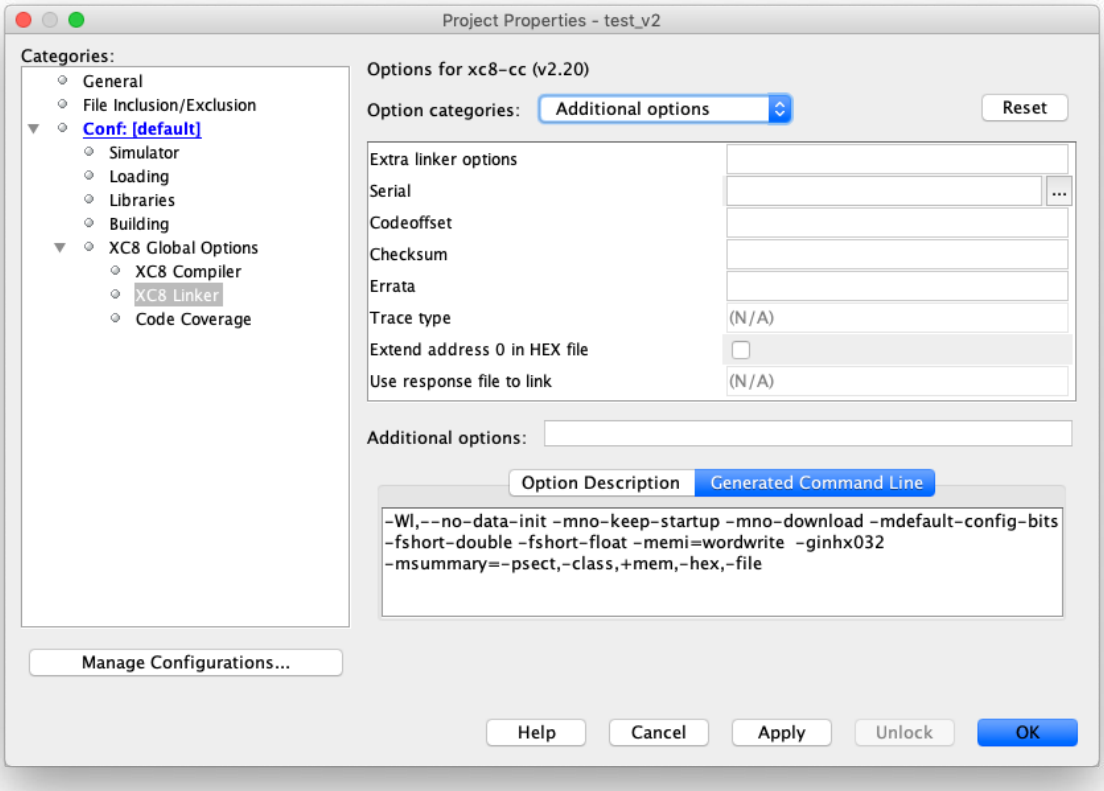
Size of double	double 型のサイズを選択します。4.6.14.1 「short-double オプション」を参照してください。
Size of floats	float 型のサイズを選択します。4.6.14.2 「short-float オプション」を参照してください。
External memory	外部メモリへのアクセス方法を選択します。4.6.1.9 「emi オプション」を参照してください。
RAM ranges	プログラムが使うデータメモリを調整します。4.6.1.17 「ram オプション」を参照してください。
ROM ranges	プログラムが使うプログラムメモリを調整します。4.6.1.20 「rom オプション」を参照してください。

4.7.1.7 XC8 Linker - Fill flash memory options



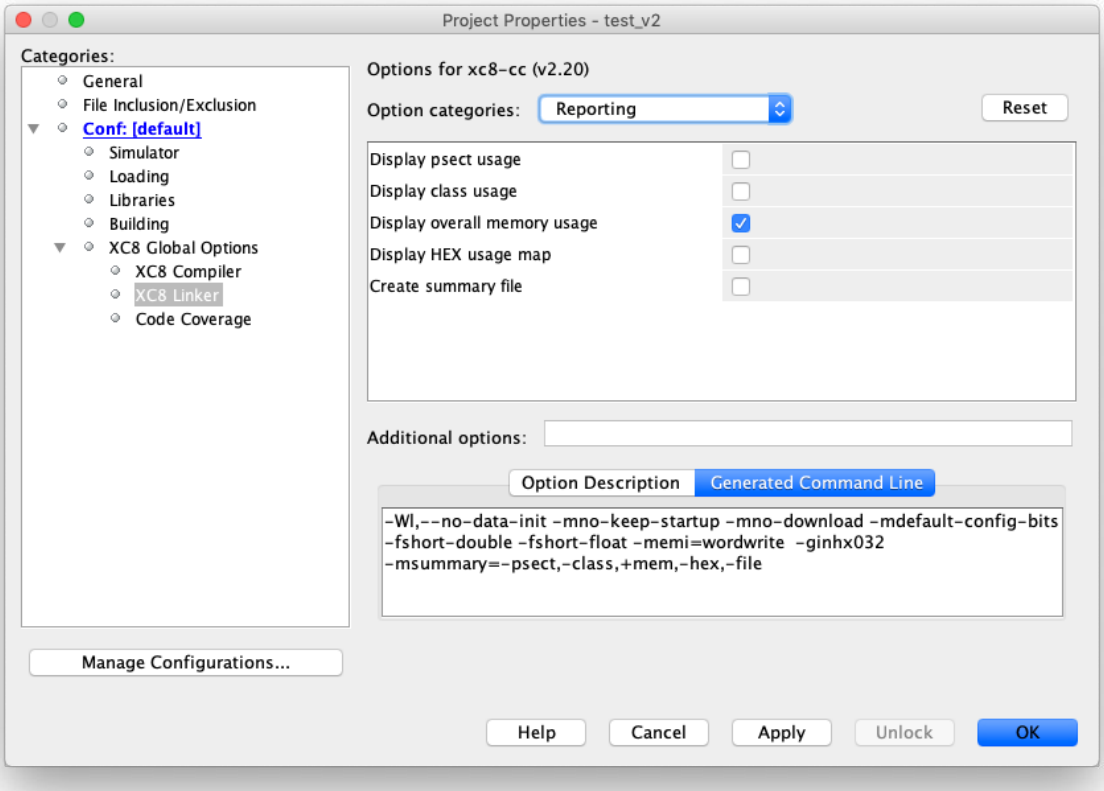
Fill options このダイアログ内の全てのフィールドは、未使用メモリ領域への値の書き込みに関連します。4.6.11.10 「fill オプション」を参照してください。

4.7.1.8 XC8 Linker - Additional options



Extra linker options	IDE から制御できない追加のリンカ関連オプションを指定できます。4.6.11.8 「Wl: オプションをリンカに渡すためのオプション」を参照してください。
Serial	16 進コードを HEX ファイル内に配置できます。4.6.11.4 「serial オプション」を参照してください。
Codeoffset	メモリ内でプログラム イメージ全体をオフセットさせる事ができます。これらの名前が認識されない場合、4.6.1.3 「codeoffset オプション」を参照してください。
Checksum	ハッシュ値を計算して HEX ファイルに挿入できます。 4.6.1.2 「checksum オプション」を参照してください。
Errata	コンパイラが適用するエラッタ回避策を調整できます。 4.6.1.10 「errata オプション」を参照してください。
Trace type	この選択は未実装です。ネイティブ トレースがサポートされます。
Extend address in HEX file	拡張されたリニアアドレスで HEX ファイルを生成します。4.6.5.2 「G: デバッグ情報生成オプション」を参照してください。
Use response file to link	このオプションを有効にすると、リンク処理中にコンパイラは[Project Properties] 内の他のリンク設定よりもコマンドライン オプション ファイルを優先して使います。4.1.1.1 「長いコマンドライン」を参照してください。このオプションは、Windows で MPLAB X IDE を実行する場合にのみ有効です。

4.7.1.9 XC8 Linker - Reporting



Display psect usage ビルド後のサマリに psect (セクション) 情報を含める事ができます。4.6.1.24 「summary オプション」を参照してください。

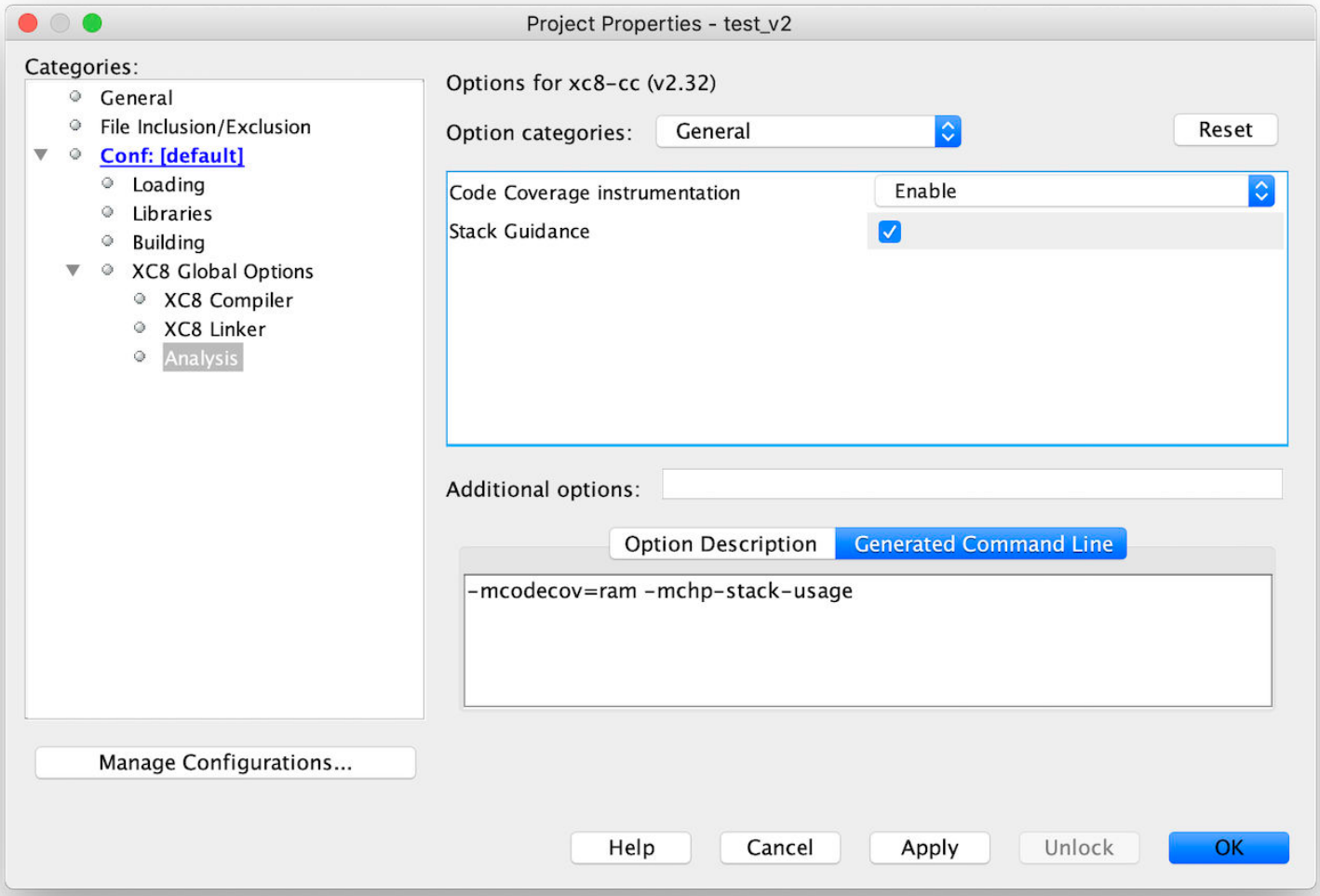
Display class usage ビルド後のサマリにリンカクラス (メモリ領域) 情報を含める事ができます。4.6.1.24 「summary オプション」を参照してください。

Display overall memory summary ビルド後のサマリに一般的なメモリ使用率情報を含める事ができます。4.6.1.24 「summary オプション」を参照してください。

Display HEX usage map ビルド後のサマリに含まれるデバイスメモリの使用率情報をグラフに表示できます。4.6.1.24 「summary オプション」を参照してください。

Create summary file ビルド後のサマリをファイルに保存できます。4.6.1.24 「summary オプション」を参照してください。

4.7.1.10 Analysis - General



Code Coverage instrumentation

これを有効にすると、コードの実行を記録するアセンブラ シーケンスがプログラム出力に実装されます。この記録を調べる事で、プロジェクトのソースコードが実際にどの程度実行されたのかを容易に解析できます。4.6.5.4 「codecov オプション」と 5.2.14 「コードカバレッジ」を参照してください。

Stack Guidance

ライセンス付き PRO コンパイラのスタック ガイダンス機能を有効にします。
この機能は、プログラムが使用する全てのスタックの最大深さを推定し、レポートを出力します。4.6.5.3 「スタック ガイダンス オプション」と 5.2.15 「スタック ガイダンス」を参照してください。

5. C 言語の機能

MPLAB XC8 コンパイラは、8 ビット PIC デバイス向け ROM ベース アプリケーションの開発を容易にする各種の特殊な機能と C 言語に対する拡張をサポートします。以下では、8 ビット PIC デバイスに固有の特殊な言語機能について説明します。

5.1 C 規格への準拠

本コンパイラは ISO/IEC 9899:1990 規約(C90 規約と呼ぶ) および ISO/IEC 9899:1999 規約(C99 規約と呼ぶ) に準拠し、単独使用が可能です。言語規格は`-std` オプションを使って選択できます(4.6.3.4 「std オプション」参照)。

本処理系はオペレーティング システムについて何も仮定せず、ストリーム、ファイル、スレッドに対するサポートを提供しません。以下では、規格から外れたコンパイラ機能について説明します。

5.1.1 CCI (Common C Interface)

本コンパイラは、Microchip 社製 XC コンパイラに共通の CCI (Common C Interface) 規格に準拠し、C ソースコードが CCI に準拠するかどうかを検証する事ができます。

CCI は C 規格の改良版であり、MPLAB XC コンパイラ ファミリ全体を通して処理系定義の動作と規格外拡張を標準化する事を目的とします。

CCI 規格への準拠は、`-mext=cci` オプション(4.6.3.3 「ext オプション」参照) により有効にできます。

5.1.2 C90 規約からの逸脱

MPLAB XC8 C コンパイラに実装されている C 言語は、以下の通りに C90 規約から逸脱する可能性があります。

5.1.2.1 再入可能な関数

MPLAB XC8 C コンパイラは、一部のデバイスで関数の再入をサポートしません。

これは、それらのデバイスがデータスタックをハードウェアで実装していないためです。本コンパイラが各デバイス ファミリ向けに使うスタックモデルの詳細は 5.2.4 「スタック」を参照してください。

拡張ミッドレンジおよび PIC18 デバイスでは、ソフトウェア スタックをデータ用に使う事で、関数を再入可能にできます。これらのデバイスでは、再帰的な関数も使えます。ベースラインおよびその他のミッドレンジ デバイスはソフトウェア スタックをサポートしないため、関数を再入可能にはできません。

デバイスが再入可能関数(ソフトウェア スタック)をサポートしない場合、または関数にコンパイルド スタックを使う必要がある場合、本コンパイラは関数を複製する事により、メインライン コードと割り込みコードから呼び出される関数を再入可能であるかのようにコード化する事ができます。詳細は 5.8.7 「関数の複製」を参照してください。

5.1.2.2 ポインタ型に対する sizeof 演算子

C 言語の `sizeof` 演算子は、ポインタ**型**またはポインタを格納した構造体または配列**型**に対して使われると問題が生じる可能性があります。しかしこの演算子はポインタ、構造体、配列**識別子**に対して使う事ができます。これは、本コンパイラによって使われるポインタのサイズが動的であるためです。

例:

```
char * cp;
size_t size;
size = sizeof(char *);
size = sizeof(cp);
```

最初の `size = sizeof(char *);` では、選択されたターゲット デバイスでポインタ用に使える最大サイズが代入されます。次の `size = sizeof(cp);`では、ポインタ変数 cp の実際のサイズが代入されます。

ポインタ変数オペランドを使う `sizeof` 演算子は、配列の宣言内でエレメント数を表す式として使う事はできません。例えば、以下の配列のサイズは予測不可能です。

```
unsigned buffer[sizeof(cp) * 10];
```

5.1.2.3 空白の関数パラメータ リスト

C90 規約に準拠するプロジェクトをビルドする場合、関数が空白のパラメータ リストを使って宣言されると、本コンパイラは「その関数はパラメータを何も持たない」と想定します。

この C90 規約によると、空白のパラメータリストで**宣言**された関数(下記例) は、波カッコ内で `void` が指定されたのと同様に、「関数は引数を何も受け取らない」と想定されます。

```
int range()
{ ...}
```

宣言内での同様のプロトタイプの例を以下に示します。

```
extern int range();
```

これは、「関数パラメータ情報は提供されない」という事を指定します。C90 規約に準拠する場合、MPLAB XC8 C コンパイラは「その関数はパラメータを一切持たない」と解釈します。

C99 規約に準拠してコンパイルする場合、上記の宣言は、「関数のパラメータに関して情報が何も指定されていない」と正しく解釈されます。

5.1.2.4 const として指定された auto オブジェクト

ベースラインまたはミッドレンジ デバイスの場合、`const` として指定された全ての `auto` オブジェクトは、`static` も指定されたかのように扱われます。これはオブジェクトのスコープに影響を及ぼしませんが、オブジェクトの値はプログラムメモリに保存されるという事と、変数が定義された時にその変数に代入される初期化子は定数式でなければならないという事を意味します。

PIC18 を使ったプロジェクトの場合、`const auto` オブジェクトはデータメモリに保存されるため、この制限は受けません。

5.1.3 C99 規約からの逸脱

MPLAB XC8 C コンパイラに実装されている C 言語は、以下の通りに C99 規約から逸脱します。

5.1.3.1 ライブラリ サポート

C99 準拠 C ライブラリは本コンパイラに同梱されますが、一部の機能が含まれていません。既知の問題に関する固有の情報は、本コンパイラのリリースノートを参照してください。

5.1.3.2 インライン化された関数

関数のインライン化では、以下の通りに C 規約から逸脱します。

規格では、`inline` 関数はそれが定義された変換ユニット内でのみインライン化されるべきであるとされます。MPLAB XC8 では、インライン化は任意のモジュール内で発生可能です。

規格では、`inline` 関数と同じ変換ユニット内で `extern` 宣言が提供されない場合、その定義は**インライン化された定義**であるとされます。これは、その定義はその関数の外部定義を提供せず、外部定義の代替であるかのように振る舞うという事も意味します。この処理系により、外部定義は `inline` 関数によって暗黙的に提供されます。

`inline` 定義に加えて外部定義が提供される場合にインライン化された定義の変換ユニット内でコンパイラがどちらの定義を使うべきかは、規格によって指定されていません。MPLAB XC8 では、`inline` 定義と外部定義の両方に遭遇すると、関数再定義エラーが出力されます。

5.1.3.3 有効な型に対するエイリアス

本コンパイラは、エイリアスされた型のチェックを行わず、エイリアスされた型の結果としてエラーが生じる可能性のある最適化も実行しません。

5.1.3.4 restrict ポインタ型修飾子

本コンパイラは、プログラム内の `restrict` ポインタ型修飾子を許容しますが、無視します。

5.1.3.5 可変長配列

配列のサイズはコンパイル時に既知である事が必要です。従って、配列の次元指定は定数式である事が必要です。関数プロトタイプは、配列で[*] 構文を使って可変長配列型を指定する事はできません。

5.1.3.6 柔軟な配列メンバー

本コンパイラは、構造体内の最後のメンバーとして不完全型配列を受け付けません。構造体の全ての配列メンバーは、エレメントの数を指定する必要があります。

5.1.3.7 複素数のサポート

本コンパイラは複素数型をサポートしません。_Complex と_Imaginary および関連する型が使われると警告が出力され、それらは無視されます。<complex.h> ヘッダもサポートしません。

5.1.3.8 拡張識別子

現時点では、C 識別子に拡張文字(カナ漢字等)は使えません。

5.1.4 処理系定義の振る舞い

ISO C 規格の一部の機能は処理系定義の振る舞いをします。これは、一部の C コードの厳密な挙動はコンパイラによって異なる可能性があるという事を意味します。本コンパイラの厳密な挙動については本書全体を通して詳細に説明しており、11. 「処理系定義の振る舞い」に全てをまとめて記載しています。

5.2 デバイスに固有の機能

MPLAB XC8 コンパイラは、8 ビット PIC アーキテクチャと命令セットに直接関係する各種機能を備えています。これらの機能について以下で詳細に説明します。

5.2.1 サポートするデバイス

MPLAB XC8 C コンパイラは全ての 8 ビット PIC および AVR デバイスをサポートします。PIC デバイスをプログラミングする場合は本書を参照してください。AVR デバイスをプログラミングする場合は『MPLAB XC8 C Compiler User's Guide for AVR® MCU』を参照してください。

新しい PIC デバイスは頻繁にリリースされます。使用中のコンパイラが特定のデバイスをサポートしているかどうかは、以下の方法で確認できます。

- コマンドラインからオプション-mprint-devices (4.6.2.8 「print-devices」参照)を指定してコンパイラを実行すると、サポートされる全てのデバイスのリストが表示されます。
- ウェブブラウザから pic_chipinfo.html ファイルを開くとサポートされる全てのベースラインおよびミッドレンジ デバイスのリストが表示され、pic18_chipinfo.html ファイルを開くとサポートされる全ての PIC18 デバイスのリストが表示されます。これらのファイルは、本コンパイラのインストール ディレクトリの下の docs ディレクトリに保存されています。

5.2.2 命令セットのサポート

本コンパイラは PIC10/12/16 デバイス向けの全ての命令セットと、標準(レガシー) の PIC18 命令セットをサポートします。一部の PIC18 で利用可能な拡張命令モードは、現在サポートしていません。コンフィグレーション ビット(通常は XINST)でこの命令セットを有効にすると、本コンパイラはエラーを出力します。

5.2.3 デバイス ヘッダファイル

ユーザが書いた各 C ソースファイルには、通常 1 つのヘッダファイルをインクルードします。<xc.h>ファイルは、プロジェクトのビルド時にデバイスまたはアーキテクチャに固有のその他のヘッダファイルをインクルードする汎用ヘッダファイルです。

このファイルをインクルードする事により、専用の変数を介する SFR へのアクセスが可能となる他、特殊なメモリアクセスまたは特殊な命令シーケンス(CLRWDT( )等)を可能とするマクロが利用できるようになります。

C ソースファイルにデバイス固有ヘッダファイルを直接インクルードしてはいけません。そうするとコードの移植性が低下する上に、これらのヘッダファイルはコードのコンパイルに必要な全ての定義を格納していない場合があります。

本コンパイラに同梱されるヘッダファイルは、そのコンパイラ バージョン専用です。将来の新しいコンパイラ バージョンには、変更されたヘッダファイルが同梱されます。プロジェクトに以前のバージョンで利用していたヘッダファイルをコピーしてインクルードしてはいけません。そのようなプロジェクトは、コンパイラの新しいバージョンと非互換となる可能性があります。

アセンブリ インクルード ファイル(.inc) については 5.11.3.4 「アセンブリコードからのレジスタへのアクセス」を参照してください。

5.2.4 スタック

8 ビット PIC 上で実行されるプログラムは、スタックを 2 通りの目的で使います。すなわち、1 つのスタックを関数の戻りアドレスを保存するために使い、1 つまたは複数のスタックをデータ割り当て用に使います。

5.2.4.1 関数戻りアドレス用スタック

8 ビット PIC デバイスは、関数の戻りアドレスを保存するためにハードウェア スタックを使います。このスタックを直接操作する事はできません。このスタックの深さはデバイスのデータシートと chipinfo ファイルに記載されています。

関数のネスト深さが最大ハードウェア スタック深さを超えると、プログラムエラーが発生します。割り込み関数と暗黙的に呼び出されるライブラリ関数も、このスタックを使うという事に注意が必要です。

本コンパイラでは、一部のデバイス向けに-mstackcall オプションを使ってスタックの使用を管理できます(4.6.1.23 「stackcall オプション」参照)。このオプションにより、スタックとは別の関数呼び出し方法が有効になり、スタックのみを使うよりも深い関数のネストが可能となります。

コード ジェネレータは、アセンブラ リストファイル内にコールグラフを生成します(6.3.6 「コールグラフ」参照)。コールグラフは、各関数呼び出しでのスタックレベルを示し、スタック深さの指標として使えます。コード ジェネレータは、最大スタック深さを超えた場合に警告を生成する事もできます。

これらの警告とコールグラフにより、スタックの使用状況を知る事ができます。最適化または割り込みの使用により、プログラムのスタック深さはコンパイラが特定したレベルから増加または減少する可能性があります。

5.2.4.2 データスタック

本コンパイラは 2 種類のデータスタック (コンパイルド スタックとソフトウェア スタック)を実装可能です。どちらのスタックも、自動記憶期間を持つスタックベース変数(auto 変数、パラメータ変数、一時変数等)を保存します。

プログラムは、これらのスタックのどちらかだけ使う事も、両方を使う事もできます。コンパイラ オプション、指定子、関数の呼び出し方法によって、使用するスタックが決まります。コンパイラが関数のスタックベース オブジェクトをどのように割り当てるかは、5.4.2.2 「自動記憶期間オブジェクト」 を参照してください。

スタックが使うメモリは、「stack」という名前のセクションによって予約されます。

5.2.4.2.1 コンパイルド スタックの動作

コンパイルド スタックは、自動記憶期間オブジェクト向けに指定される 1 つまたは複数のメモリ領域です。コンパイルド スタック内のオブジェクトが割り当てられた空間には、コンパイラによって割り当てられたシンボルを介してアクセス可能な静的アドレスが割り当てられます。これがスタックベース オブジェクトにアクセスする最も効率の良い方法です(スタック ポインタを使わないため)。

スタックベース オブジェクトをコンパイルド スタック内に割り当てる関数は再入可能ではありません。なぜなら、その関数のオブジェクトの各インスタンスは同一シンボル (アドレス)を介してアクセスされるからです(5.11.3 「アセンブリコードと C コードの間の相互作用」参照)。コンパイルド スタック内のメモリは、通常のソフトウェア スタックと同様に再使用可能です。2 つの関数が決して同時にアクティブにならない場合、それらの関数のスタックベース オブジェクトはメモリ内でオーバーラップしても構いません(データは破損しません)。コンパイラは、どの関数が同時にアクティブになるか見極める事ができ、可能な場合に自動的にメモリを再使用します。コンパイラは、割り込み関数(およびそこから呼び出される関数)がそれぞれに専用のメモリを必要とする事を考慮します(5.8.7 「関数の複製」参照)。

コンパイルド スタックのサイズはコンパイル時に特定可能であるため、コンパイラは利用可能な空間を確認できます。

5.2.4.2.2 ソフトウェア スタックの動作

ソフトウェア スタックは、自動記憶期間オブジェクト向けに使われる動的割り当てメモリであり、スタックポインタを介して間接的にアクセスされます。ソフトウェア スタック上のオブジェクトへのアクセスは低速になる可能性がありますが、ソフトウェア スタックを使う関数は再入可能となります。ソフトウェア スタックは、拡張ミッドレンジおよび PIC18 デバイスでのみ利用できます。

関数が呼び出されると、ソフトウェア スタックはメモリのチャンクを関数のスタックベース オブジェクト向けに割り当て、メモリ内のスタックのサイズは増加します。関数が終了すると使われていたメモリは解放され、他の関数向けに利用可能となります。従ってソフトウェア スタックのサイズは、プログラムの実行中に動的に変化します。オブジェクトがスタックに割り当てられると、スタックは高アドレスへ向かって伸びて行きます。関数から戻るとスタックのサイズは減少し、そのスタックベース オブジェクトは使われなくなります。

レジスタ (スタックポインタ) は、スタックの先頭位置(Top of Stack) のアドレスを保持するために恒久的に割り当てられます。MPLAB XC8 コンパイラは、ソフトウェア スタック内で次に利用可能な位置を保持するために、FSR1 レジスタをスタックポインタとして使います。このレジスタの値は変数がスタックに追加(プッシュ)されると増加し、関数から戻って変数がスタックから削除(ポップ)されると減少します。フレームポインタを保持するためのレジスタは割り当てられません。スタックへアクセスするには、スタックポインタに対するオフセットを使う必要があります。

ソフトウェア スタックを使う関数がプログラム内に存在する場合、プログラムの実行中は(たとえ実行中の関数がソフトウェア スタックを使っていなくても) FSR1 レジスタをスタックポインタとして保持する必要があります。

このレジスタは汎用的に使えなくなるため、生成されるコードの効率が低下するか、「Can't generate code」エラーが生成される可能性があります。

コンパイル時にスタックの最大サイズを正確に知る事はできないため、通常コンパイラは可能な限り大きな空間をスタック向けに予約します。スタックは常に単一のメモリレンジに割り当てられます。このメモリレンジはバンク境界をまたぐ事ができます。このメモリレンジは、必要に応じて複数の領域(メインライン コード用に 1 つと各割り込みコード用に 1 つずつ) に分割される場合があります。各領域の最大サイズは、-mstack オプション(4.6.1.22 「stack オプション」)を使って指定できます。スタックポインタは割り込みが発生した時にリロードされ、その割り込みコードによって使われるスタック領域にアクセスします。割り込みコードが完了した時に、スタックポインタは割り込みコンテクスト スイッチコードによって復元されます。

コンパイラはスタック全体として予約されたメモリのオーバーフローを検出する事はできません。また、実行時にスタック オーバーフローに関するチェックも行われません。ソフトウェア スタックがオーバーフローするとデータは破損し、コードの実行に失敗する可能性があります。

5.2.5 コンフィグレーション ビットへのアクセス

config プラグマにより、アセンブリ ソースファイル内でコンフィグレーション ビット(またはヒューズ)と ID ロケーション レジスタを指定する事ができます。コンフィグレーション ビット(またはヒューズ)は、基本的なデバイスの動作(オシレータモード、ウォッチドッグ タイマ、プログラミング モード、コード保護等)を指定します。プログラムを正常に実行するには、これらのビットを正しく設定する必要があります。

このプラグマの形態は以下の通りです。

```
#pragma config setting = value
#pragma config register = literal_value
```

setting : コンフィグレーション設定記述子(例: WDT)、*value* : 状態を指定するテキスト(例: OFF)または数値

setting または *value* トークンまたは *setting = value* 式をダブルまたはシングル クォーテーションで囲む事により、プリプロセッサが実行するマクロ置換からそれらを保護できます。

例:

```
#pragma config "WDT = ON"      // turn on watchdog timer
#pragma config WDTPS = 0x1A    // specify the timer postscale value
```

純粋に数値のように見える value トークンであっても、実際には値のテキスト記述である場合があります。
例:

```
config WDTPS = 32
```

32 は、一部のデバイス向けのテキスト記述であり、ウォッチドッグ タイマのポストスケールが 1:32 である事を示します(数値の 32 を示すのではありません)。このような場合、コンフィグレーション レジスタの対応するビットには、数値として 32 が書き込まれるのではない可能性があります。アセンブラは最初に *value* が定義済みの文字列を表しているかどうか確認し、そうではない場合にのみ *value* は数値定数であると見なし、他の数値定数オペランドと同様の制約を *value* に適用します。

コンフィグレーション マクロ内で使われる OFF および ON トークンが 0 および 1 と等価であると見なしてはいけません。多くの場合、それは誤りです。

register フィールドはコンフィグレーション レジスタまたは ID ロケーション レジスタの名前であり、必ず *value* で数値定数を指定する必要があります。
例:

```
#pragma config CONFIG1L = 0x8F
```

利用可能な *setting*、*register*、*value* フィールドの設定は、デバイスの chipinfo ファイル(例: pic_chipinfo.html 、pic18_chipinfo.html) に記載されています。このファイルはコンパイラ インストール ディレクトリ内の docs ディレクトリに保存されています。ターゲット デバイスへのリンクをクリックすると、このプラグマ向けの適切な設定と値が表示されます。詳細はデバイスのデータシートを参照してください。

各コンフィグレーションの設定用に config プラグマを 1 つずつ使う事も、複数の設定をコンマで区切って 1 つのプラグマで指定する事もできます。このプラグマは、デバイスを完全に設定し終えるまで必要に応じて何度でも使えます。

© 2022 Microchip Technology Inc. and its subsidiaries

以下に、1 つのコンフィグレーション レジスタ内の個々の設定を指定して書き込む方法と、レジスタ全体として書き込む方法を示します。

```
// PIC18F67K22
// VREG Sleep Enable bit :Enabled
// LF-INTOSC Low-power Enable bit :LF-INTOSC in High-power mode during Sleep
// SOSC Power Selection and mode Configuration bits :High Power SOSC circuit selected
// Extended Instruction Set :Enabled
#pragma config RETEN = ON, INTOSCSEL = HIGH, SOSCSEL = HIGH, XINST = ON

// Alternatively
#pragma config CONFIG1L = 0x5D

// IDLOC @ 0x200000
#pragma config IDLOC0 = 0x15
```

config プラグマは実行可能コードを生成しません。上記のどちらの方法で設定する場合も、このプラグマは関数定義の外に配置するのが理想的です。

不適正なプログラム動作を防ぐため、コンフィグレーション ワード内の全てのビットに書き込む必要があります。どのビットも既定値/未プログラム状態のままにしてはいけません。コンフィグレーション ビットの既定値状態は全てが論理 High であるとは限りません。既定値状態が論理 Low のビットも存在します。詳細はデバイスのデータシートを参照してください。

MPLAB X IDE を使うと、内部ツールを利用して必要なプラグマを生成し、それらをソースコードにコピー&ペーストできます。MPLAB X IDE の「Configuration Bits」ウィンドウの詳細は『MPLAB® X IDE User's Guide 』を参照してください。

5.2.6 ID ロケーション

8 ビット PIC デバイスは、アドレス指定可能メモリ領域の外に、ID 番号等のプログラム情報が保存できるメモリ領域として ID ロケーションを備えています。ID ロケーションへのデータの配置には config プラグマと専用のレジスタ名を使います。

例:

```
#pragma config IDLOCx = value
```

x : ID ロケーションの番号(位置)、 *value*: ID ロケーションに配置するニブルまたはバイト

value は 10 進数または 16 進数(接頭辞 0x)で指定する必要があります。2 進数(接頭辞 0b)で指定してはいけません。*value* がターゲット デバイス上の各 ID ロケーションの最大許容値より大きい場合、値は切り詰められて警告が出力されます。各 ID ロケーションのサイズはデバイスごとに異なります。詳細はデバイスのデータシートを参照してください。

```
#pragma config IDLOC0 = 1
#pragma config IDLOC1 = 4
```

上記は、最初の 2 つの ID ロケーションに値 1 と 4 を書き込みます。レジスタ名と値のペアをコンマで区切る事により、1 つのプラグマを使って複数の ID ロケーションに値を書き込む事ができます。

```
#pragma config IDLOC0 = 1, IDLOC1 = 4
```

config プラグマは実行可能コードを生成しません。このプラグマは関数定義の外に配置するのが理想的です。

5.2.7 C コードから SFR を使う

特殊機能レジスタ(SFR) はメモリにマッピングされたレジスタであり、それらのレジスタのアドレスに配置された絶対アドレス C 構造体変数によりアクセスします。これらの構造体には通常の方法でアクセスできるため、特別な構文を使わずに SFR にアクセスできます。

SFR は MCU と周辺モジュールの動作を制御します。読み出し専用のレジスタもあれば、書き込み専用のレジスタもあります。常にデバイスのデータシートでこれらのレジスタに関する情報を確認してください。

SFR 構造体はヘッダファイル<xc.h> 内で定義済みです。このヘッダファイルをソースファイルにインクルードする事により、SFR へのアクセスが可能になります(5.2.3 「デバイス ヘッダファイル」参照)。構造体はレジスタ全体に対してマッピングされ、構造体内のビットフィールドにより個々の SFR ビットへのアクセスが可能です。通常、構造体には、対応するレジスタの名前(デバイス データシーに記載) の後に「bits」を付加した名前が与えられます(3.3.2.5 「SFR とその中のビットを表す名前はどのようにして見つけるのですか」参照)。

例えば下のコードは、汎用ヘッダファイルをインクルードし、PORTA レジスタ全体をクリアした後に、ビットフィールド定義を使って PORTA の bit 2 をセットします。

```
#include <xc.h>
int main(void)
{
    PORTA = 0x00;
    PORTAbits.RA2 = 1;
}
```

一部の SFR では、C コードまたはインライン アセンブリからアクセスする際に注意が必要です。コンパイラは一部のレジスタを使って計算の中間結果を保持するため、それらのレジスタに直接書き込むと期待以外の動作が発生する可能性があります。コンパイラによって使われるレジスタのリストは 5.6 「レジスタの使用」に記載しています。

5.2.7.1 PIC18 のレジスタに固有の問題

PIC18 が使う一部の SFR は、マルチバイト値を形成するようグループ化されています(例: TMRxH および TMRxL レジスタの組み合わせにより 16 ビットタイマ カウント値を形成)。デバイスと動作モードによっては、これらのレジスタをハードウェア要件に従った方法で読み出す必要があります。例えば、TMRxL レジスタを TMRxH レジスタの前に読み出さないと有効な 16 ビット値は得られません。

これらのレジスタにマッピングされたマルチバイト変数を読み出す事は推奨しません(これらのバイトの読み出し順は保証されないため)。デバイス データシートで指定されている順序で、SFR の個々のバイトに直接アクセスする事を推奨します。この方法により、移植性が大きく向上します。

以下のコードは、2 つのタイマレジスタの値を符号なし C 変数 count にコピーします。

```
count = TMR0L;
count += TMR0H << 8;
```

タイマレジスタの読み書きを実行するためのマクロも利用できます。9.1 「ライブラリ サンプルコード」に記載したマクロ READTIMER*x* および WRITETIMER*x* を参照してください。これらは正しいバイト順での読み書きを保証します。

5.2.8 ビット命令

MPLAB XC8 C コンパイラは、ビット値ではない整数値に対してもビット命令を使おうと試みます。例えば、整数型値の中の 1 ビットを変更するためにビット処理演算子とマスクを使う場合、本コンパイラはビット命令を使って同じ機能を実行できないか判定するためにマスク値を確認します。

```
unsigned int foo;
foo |= 0x40;
```

上記のコードから、下記の命令が生成されます。

```
bsf _foo,6
```

整数型値の中の個々のビットをセットまたはクリアするために以下のマクロが使えます。

```
#define bitset(var, bitno)    ((var) |= 1UL << (bitno))
#define bitclr(var, bitno)    ((var) &= ~(1UL << (bitno)))
```

上記と同じ演算を foo に対して実行するには、下記の bitset()マクロが使えます。

```
bitset(foo, 6);
```

5.2.9 乗算

PIC18 命令セットには各種の 8x8 ビット ハードウェア乗算命令が含まれており、本コンパイラは多くの状況でこれらの命令を使います。PIC18 以外のターゲット デバイスは、乗算用に常にライブラリコードを使います。

本コンパイラは、8x8 ビット整数乗算を以下の 3 通りの方法で実装できます。

ハードウェア乗算命令(HMI) これらのアセンブリ命令は PIC18 でのみ利用でき、最も効率的な乗算方法を提供します。

ビット単位反復 (8loop) 専用の乗算命令が使えない場合、この処理系は最小のコードを生成します。この方法では、1 つのループがオペランド内のビットパターンを循環し、ビット単位で結果を構成します。この処理系の演算速度はオペランド値に応じて変化しますが、一般的に最も遅い乗算方法です。

ループアンロールしたビット単位処理 (8seq) この処理系は、上記のビット単位反復(8loop)と同じ命令のシーケンスを実行しますが、ループはアンロール(展開)されます。8loop 方式に比べると、生成されるコードは大きいですが実行はより高速です。

8 ビットよりも大きなオペランドの乗算は、以下の 2 通りの方法で実行できます。

ビット単位反復 (xloop) これは前記の 8 ビット乗算と同じアルゴリズムを使いますが、ループをオペランドの全ての(x)ビットに対して実行します。8 ビット乗算と同様に、この処理系は最小のコードを生成しますが、通常は最も低速な乗算方法です。

バイト単位分解 (bytdec) この方式は、乗算を複数の 8 ビット乗算に分解して合計します。各 8 ビット乗算には上記の 8 ビット乗算法のいずれかが使えます。この方法は、ハードウェア乗算命令が使える PIC18 で効果的に使えます。他のデバイスでは、演算は高速になりますがコードが非現実的なサイズになる場合があります。

浮動小数点オペランドの乗算も同様の方法で実行します(ビット単位反復(xloop)またはバイト単位分解(bytdec)を使って仮数部を乗算)。

表 5-1 と表 5-2 に、整数および浮動小数点オペランドの乗算用に本コンパイラが選択する乗算法の一覧を示します。選択する乗算法はオペランドのサイズ、最適化のタイプ、ターゲット デバイスに応じて異なります。

表 5-1 に、速度の最適化(4.6.6 「最適化を制御するためのオプション」参照)を有効にした場合に選択される乗算法を示します。

表 5-1. 速度の最適化が有効な場合の乗算法

デバイス	8 ビット	16 ビット	24 ビット	32 ビット	24 ビット FP	32 ビット FP
PIC18	HMI	bytdec+HMI	bytdec+HMI	bytdec+HMI	bytdec+HMI	bytdec+HMI
拡張ミッドレンジ	8seq	bytdec+8seq	bytdec+8seq	bytdec+8seq	bytdec+8seq	bytdec+8seq
ミッドレンジ ベースライン	8seq	16loop	24loop	32loop	24fploop	32fploop

表 5-2 に、サイズの最適化を有効にした場合または C レベルの最適化を何も有効にしなかった場合に選択される乗算法を示します。

表 5-2. 最適化が無効またはサイズの最適化が有効な場合の乗算法

デバイス	8 ビット	16 ビット	24 ビット	32 ビット	24 ビット FP	32 ビット FP
PIC18	HMI	bytdec+HMI	24loop	32loop	24fploop	32fploop
拡張ミッドレンジ	8loop	bytdec+8loop	24loop	32loop	24fploop	32fploop
ミッドレンジ ベースライン	8loop	16loop	24loop	32loop	24fploop	32fploop

乗算ルーチンのソースコード(実装アルゴリズム解説あり)がコンパイラ インストール ディレクトリ内の `pic/c99/sources` ディレクトリに保存されています。それらのファイル名は `Umulx.c` (`x` はオペランドのビット数)です。

ターゲット デバイスと最適化設定からビット単位反復法が選択された場合、C コード内の乗算オペランド順を調整する事で演算速度が向上する可能性があります。取りうる数値範囲が確定している場合、乗算の左のオペランドの方を小さな値にします。

例:

```
x = 10;
y = 200;
result = x * y; // first multiply
result = y * x; // second multiply
```

`x * y` でも `y * x` でも変数 `result` に代入される値は同じですが、`x * y` の方が高速です。

5.2.10 ベースライン PIC MCU に固有の命令

ベースライン デバイスでは、メモリにマッピングされていない OPTION および TRIS SFR が使えます。

これらのレジスタの定義には特殊な修飾子`__control` を使います。この修飾子により、そのレジスタは書き込み専用であり、通常のアドレス空間の外に位置し、特殊な命令を使ってアクセスする必要がある事を示します。

C コード内でこれらの SFR が書き込まれる場合、本コンパイラは適切な命令を使って値を保存します。例えば TRIS レジスタを下記のコードで設定する場合、

```
TRIS = 0xFF;
```

コンパイラは下記の通りにエンコードします。

```
movlw 0ffh
TRIS
```

複数の出力ポートを備えたベースライン PIC デバイスでは、オブジェクト `TRISA`、`TRISB`、`TRISC` 向けに `__control` 定義が上記と同じ方法で使えます。

`__control` 修飾子を使う全てのレジスタはフルバイトとしてアクセスする必要があります。これらのレジスタ内の特定のビットにアクセスする必要がある場合、レジスタの内容を一時変数に複製した後に、その一時変数にアクセスする必要があります。

5.2.11 オシレータ校正定数

一部のベースラインおよびミッドレンジ デバイスのプログラムメモリには、オシレータ校正定数が書き込み済みです。この定数を読み出して OSCCAL レジスタに書き込む事で、内部 RC オシレータを校正できます。

一部のベースライン PIC デバイス(PIC10F509 等)では、校正定数は `movlw` 命令としてプログラムメモリの最後尾に保存されます。リセット時にプログラム カウンタはこの命令を指し示し、プログラム カウンタが 0x0000 (そのデバイスの実質的なリセットベクタ位置)にロールオーバーする前にこの命令が実行されます。そのようなデバイスでは、リセット後に W レジスタの内容を OSCCAL レジスタに書き込むコードが既定値の実行時起動コード(5.9.2 「実行時起動コード」参照) によってを自動的にインクルードされます。その他のコードは不要です。

その他のデバイス(PIC12F629 等)でも、オシレータ校正定数はプログラムメモリの最後尾に保存されますが、`retlw` 命令として保存されます。コンパイラの起動コードは、この値を取得して設定を実行するためのコードを自動的に生成します。この値は、`__osccal_val()`を呼び出す事により実行時に読み出す事もできます(そのプロトタイプは `<xc.h>`内で提供されます)。

例:

```
unsigned char calVal;
calVal = __osccal_val();
```

起動時の校正値のロードは、`-mno-osccal` オプション(4.6.1.15 「osccal オプション」参照) により無効にできます。

Note: 校正定数の保存位置はコード保護されないため、デバイスを再プログラミングすると内容は失われます。このため、デバイスを消去する前に校正定数を保存しておき、新しいプログラムとデータを書き込む際に、この定数を元の位置に再度書き込む必要があります。
インサーキット エミュレータ(ICE)を使う場合、校正用 `retlw` 命令によって使われる位置には書き込めず、後続の `__osccal_val()`への呼び出しは機能しません。ICE でこの関数を呼び出すコードをテストするには、`retlw` 命令を適切な位置に書き込む必要があります。実際のデバイスをプログラミングする際は、この命令を忘れずに削除する必要があります。そうしないと、校正値が破損してしまいます。

レガシー プロジェクトでは、マクロ`_READ_OSCCAL_DATA()` が使えます。このマクロは`__osccal_val()`関数へ置き換わります。

5.2.12 MPLAB REAL ICE インサーキット エミュレータのサポート

Microchip 社の MPLAB REAL ICE インサーキット エミュレータを使う場合、本コンパイラはログおよびトレース機能(Instrumented Trace)をサポートします。Instrumented Trace 機能の詳細は、エミュレータのマニュアルを参照してください。

一部のデバイスは Instrumented Trace 機能をサポートしません。また、現時点では、本コンパイラはネイティブ トレースのみサポートします。

ログおよびトレース マクロの呼び出しは、ソースコードに手書きで追加するか、MPLAB X IDE エディタ内で適切な位置を右クリックする事により挿入できます(エミュレータのマニュアル参照)。これらのマクロをアセンブリコード内で使ってはいけません。これらのマクロを使う全てのモジュールには`<xc.h>` ヘッダをインクルードする必要があります。これらのマクロの形態は以下の通りです。

```
__TRACE(id);
__LOG(id, expression);
```

MPLAB X IDE は、コンパイル時に *id* に適切な値を自動的に代入します。必要に応じて、この値を手動で指定する事もできます。TRACE の *id* は 0x40～0x7F の定数、LOG の *id* は 0x0～0x7F の定数である事が必要です。各マクロには、識別のために一意の番号を与える必要があります。TRACE マクロと LOG マクロの両方に同じ番号を使う事ができます。

TRACE マクロは、C ソースコード内のトレースする必要がある位置に挿入します。挿入されたマクロは、実行された時にデバッガと IDE への情報送信をトリガします。これにより、コード実行がその位置に達した事が記録されます。

LOG マクロの *expression* は任意の整数または 32 ビット不動小数点式であり、その値はプログラム位置と一緒に記録されます。通常、*expression* は単純に変数名であり、その変数の内容が記録されます。

TRACE および LOG マクロを追加するとコードのサイズは増加します(これらのマクロはデバイスにダウンロードされるプログラム イメージに影響を及ぼすため)。

以下に、ソースコードに追加されたこれらのマクロの例を示します。

```
#include <xc.h>
inpStatus = readUser();
if(inpStatus == 0) {
    __TRACE(id);
    recovery();
}
__LOG(id, expression);
```

5.2.13 関数プロファイリング

関数プロファイリング機能を提供するため、本コンパイラは MPLAB REAL ICE インサーキット エミュレータ向けに関数登録コードを生成できます。プロファイリング結果を得るには、電源監視ボードと MPLAB X IDE に加えて MPLAB XC8 C コンパイラ向けコード プロファイリングをサポートする電源監視プラグインが必要です。

`-finstrument-functions` オプション(4.6.5.1 「 instrument-functions オプション」参照) はこの機能を有効にし、アセンブリコードを各関数のプロローグとエピローグに挿入します。このコードは、関数の実行開始時と終了時に実行時情報をデバッガへ送信します。この情報と電源監視ボードからの計測値により、各関数が消費する電力を調べる事ができます。この機能は透過的ですが、プロファイリングを有効にする際は以下に注意する必要があります。

- プロファイリング コードにより、プログラムのサイズは増加し、実行速度は低下します。
- 使われるハードウェア スタックのレベルが 1 つ増加します。

© 2022 Microchip Technology Inc. and its subsidiaries

• RAM メモリの使用量が増加します。

• プロファイリングされる関数はインライン化されません。

ハードウェア スタックの制約等によって関数のプロファイリングができない場合、関数が `inline` として指定されていれば、本コンパイラはその関数をインライン化する場合があります。関数のインライン化の詳細は 5.7.1.2 「inline 指定子」を参照してください。

5.2.14 コードカバレッジ

特別なライセンス(SW006026-COV)を購入すると、本コンパイラのコードカバレッジ機能を使ってプロジェクトのソースコードのどの範囲が実行されたのかを解析できます。

この機能は、全ての拡張ミッドレンジ デバイスと PIC18 デバイスで利用できます。

この機能を有効にすると、プロジェクトのプログラム イメージに小さなアセンブリ シーケンスが実装されます。プログラム イメージが実行されると、これらのシーケンスによってデバイス RAM 内の予約域にコード実行の有無が記録されます。デバイスに保存された記録を後で解析する事により、プロジェクトのソースコードのどの部分が実行されたのかを特定できます。コンパイラが提供するライブラリコードの実行は記録されません。

関数プロファイリング機能は関数の実行状況を示しますが、コードカバレッジ機能はもっと小さなコードブロックの実行状況を示します。この機能の基本的な目的は、プログラムの全ての部分が実行されたか(従ってテストされたか)どうかを確認する事です。この機能は、コードが正しく実行されたどうかを検証しません。また、関数プロファイリング機能とは異なり、コードの実行時間も示しません。

コードカバレッジ機能を有効にすると、本コンパイラは外部ツール(`xc-ccov`)を呼び出して、プロジェクトを解析するための最も効率的な方法を決定します。このツールは、シーケンスの開始位置に置かれた単一のエントリポイントとシーケンスの終了位置に置かれた単一のエンドポイントを持つ 1 つまたは複数の命令からなるシーケンスを基本ブロックと見なします。ただしブロックの全てが解析される必要はなく、ツールはより完全に解析可能なブロックの最小セットを決定します。

本コンパイラでコードカバレッジ機能を有効にするには`-mcodecov` オプションを使います。

プロジェクトのビルド用に指定された全てのコンパイラ オプションは、コードカバレッジ機能の使用時も効力を有します。なぜなら、それらは最終的に解析されるプログラム イメージに影響を与えるからです。最終製品の状態を正確に解析するには、ビルド オプションを最終リリース向けに使用するオプションと同じにする必要があります。

コードカバレッジ機能を有効にした場合、解析される基本ブロックあたり 1 ビットの RAM が割り当てられるため、プロジェクトが要求するデータメモリ量は増加します。

解析された各基本ブロックには 1 つのアセンブリ命令が挿入されます。コードカバレッジ ビットの格納位置とターゲット デバイスによっては、バンクを選択するための命令が必要になる場合があります。コードカバレッジ機能を有効にすると、一部の最適化は無効になる(その結果、ビルドされたプロジェクトのサイズは増加する)事に注意が必要です。

コードカバレッジ データを生成するには、解析されたプロジェクト コードが実行される必要があり、追加されたアセンブリ シーケンスによってその実行速度はわずかに低下します。プログラムソースの全ての部分が実行されて記録されるように入力とスティミュラスを実行中のプログラムに提供する必要があります。

コードカバレッジデータは MPLAB X IDE 内で解析できます。コンパイラが生成した ELF ファイル内の情報により、電源監視プラグインはコードカバレッジ結果が格納されているデバイスメモリ位置を特定して読み出し、それらを分かりやすく表示する事ができます。新しいコードカバレッジ機能の使用法は、
www.microchip.com/Developmenttools/ProductDetails/SW006026-COV を参照してください。

5.2.15 スタック ガイダンス

PRO コンパイラ ライセンスが有効である場合、コンパイラのスタック ガイダンス機能を使ってプログラムが使うスタックの深さを推定する事ができます。

実行時にスタック オーバーフローによってプログラムエラーが発生すると、特にプログラムが複雑で割り込みが使われている場合、その原因を追跡するのは困難となります。本コンパイラのスタック ガイダンス機能は、プログラムのコールグラフを生成して解析する事により各関数のスタック使用状況を示すレポートを生成します。このレポートにより、プログラムが使うスタックの深さを推測できます。開発中にプログラムのスタック使用状況を監視する事で、スタック オーバーフローが発生する危険性を軽減できます。

この機能は、コマンドライン オプション`-mchp-stack-usage` により有効にできます。

有効にすると、スタック ガイダンス機能は自動的に実行されます。コンパイラをコマンドラインから実行する場合、ビルドに成功した後にレポートがコンソール画面に直接表示されます。MPLAB X IDE 内でビルドを実行する場合、同じレポートが**[Output]** ウィンドウ内のビルド画面に表示されます。

より詳細なスタック使用情報をマップファイルに保存する事もできます。これは、コマンドライン オプション `-Wl,-Map=`*`mapfile`* (または MPLAB X IDE Project Properties 内の対応する設定)を使って要求できます。

5.2.15.1 スタック ガイダンス情報

スタック ガイダンス機能は、MPLAB XC8 C コンパイラでコンパイルされたプログラムによって使われる以下のスタック使用状況を推定します。

- ハードウェア スタック - 関数の戻りアドレス用
- ソフトウェア データスタック - 再入可能モデルを使ってコンパイルされた全ての関数用

ハードウェア スタックの説明は 5.2.4.1 「関数戻りアドレス用スタック」に記載しています。このスタックのサイズは、デバイスによって固定されています。関数のネストが深すぎる場合、スタックがオーバーフローして最終的にプログラムエラーが発生します。割り込みもこのスタックを使います。一部のデバイスでは、`-mstackcall` オプションを使って異なる関数呼び出し方式を採用する事で、スタックの使用量を削減できます。

本コンパイラは、関数によって定義されたデータ オブジェクト用にソフトウェア スタックまたはコンパイルド スタックのどちらかを使うように関数をエンコードします(5.2.4.2 「データスタック」参照)。1 つの関数は常にどちらかのスタックタイプだけを使います。しかし、1 つのプログラムにコンパイルド スタックを使う関数とソフトウェア スタックを使う関数が混在しても構いません。通常、プログラムは 1 つのスタックモデル(`-mstack` オプション) に従い、全ての関数が同じタイプのスタックを使うようにコンパイルされます。

コンパイルド スタックは、スタックベース オブジェクト向けの静的なデータメモリ割り当てです。このスタックのサイズはコンパイル時に正確に特定でき、実行時に変化しないため、このスタックが過大となる場合はコンパイラによってメモリエラーが出力されます。スタック ガイダンス機能は、コンパイルド スタックが使われるという事を示すのみであり、コンパイルド スタックに関して他に何の情報も提供しません。

これに対しソフトスタックは実行時に発生する動的なメモリ割り当てであり、そのサイズを予測する事はコンパイルド スタックよりも困難です。これには、スタック ガイダンス機能が提供する情報が役立ちます。スタック ガイダンス機能は、ソフトウェア スタックが実行時にオーバーフローする危険性がある事をユーザに警告します。

ビルドに成功した後に表示されるスタック使用量情報の例を以下に示します。

```
================================ STACK USAGE GUIDANCE ================================

The program uses a compiled stack

model.The call-graph beginning at

main(),
   uses an estimated 17 of 31 hardware stack locations and a compiled stack

However, the following cautions exist:

1) There is 1 "high-priority" interrupt service routine detected:
   isr() call-graph uses an estimated 1 hardware stack location and a compiled stack
You must add stack allowances for those functions.

2) Current optimization level might use a level of hardware
stack.You must add a hardware stack allowance for this feature.

======================================================================================
```

この情報は以下の内容を示します。

- プログラムが使うスタックモデル
- ハードウェア スタックの推定使用量
- ソフトウェア スタックの推定使用量

さらに、各種の警告メッセージが表示される場合があります。これらは、前述の推定によって報告されたスタックオーバーフローの可能性を明確に示すと共に、プログラムの総スタック使用量を求めるために考慮に入れる必要がある追加のスタック使用量を示します。各種要因によりこの追加のスタックがいつ使われるか知る事はできないため、早期のサマリにはこれらの情報は含まれません。

以下の警告が表示される場合があります。

Recursive functions	プログラム内に再帰的関数呼び出しが検出された事を示します。再帰的呼び出しの反復回数は予測できないため、ハードウェア スタックのレベル数とソフトウェア スタックの総サイズは特定できません。従ってスタックガイダンスは不可能です。この警告には検出された再帰的関数の名前が示されます。
External functions	外部関数への呼び出しが検出された事を示します。外部関数とは、C プログラム内で定義されていない関数の事です。例えば、アセンブリコードは外部関数と見なされます。外部関数の場合、それらが消費するハードウェア スタックのレベル数も、それらがソフトウェア スタックに配置するデータも知る事はできません。このため外部関数が呼び出される場合、ガイダンスは不可能です。この警告には呼び出された外部関数の名前が示されます。
Deviations from the current stack model	本コンパイラは、`-mstack` オプション(全ての関数向けのスタックタイプを指定)によって指定されたスタックモデルを使ってプログラムをビルドしますが、一部の関数向けに異なるスタックタイプが使われる場合があります(スタック指定子を使って指定)。この警告には、モデルによって指定されたのとは異なるスタックタイプを使う関数のリストを示します。
Main-line code exceeds hardware stack size	この警告は、メインライン コード コールグラフの関数呼び出し深さがターゲット デバイスで利用可能なハードウェア スタックのレベル数を超えた事を示します。一部のデバイスでは、`-mstackcall` オプション(4.6.1.23 「stackcall オプション」参照)を使って、そのように多数のスタックレベルを必要としない別の関数呼び出し方式をコンパイラに適用させる事ができます。この方式を使うと、コンパイラが出力するスタック ガイダンス レポート内に示されるスタック使用量が減少します。
Interrupts might exceed hardware stack size	メインラインおよび割り込みコードに関連するコールグラフの深さは既知であっても、割り込みはいつでもトリガ可能であるため、コンパイラはプログラムの関数コールグラフの総深さを高い信頼性で特定する事はできません。この警告は割り込み関数が存在する事を示し、プログラムの総ハードウェア スタック使用量を検討するにはそれらの割り込み関数のスタック使用量を考慮に入れる必要がある事をユーザに警告します。
Call graph exceeds software stack size	スタック ガイダンス機能が、「示されたコールグラフ (メインラインまたは割り込み)が割り当てられたよりも多くのソフトウェア データスタックを使った(または使う可能性がある)」と、推定した事を示します。`-mstack` オプションを使って、各コールグラフに割り当てるサイズを変更する事ができます(4.6.1.22 「stack オプション」参照)。
Interrupts might exceed software stack size	メインラインおよび割り込みコールグラフによって使われるソフトウェア スタックのサイズが既知であっても、割り込みはいつでもトリガ可能であるため、コンパイラはプログラムの総スタック使用量を高い信頼性で特定する事はできません。この警告は割り込み関数が存在する事を示し、プログラムの総ソフトウェア スタック使用量を検討するにはそれらの割り込み関数のスタック使用量を考慮に入れる必要がある事をユーザに警告します。
Optimization level might use hardware stack	一部の最適化によって呼び出しが追加される場合があり、そうするとハードウェア スタックの使用量は報告された量よりも増加します。この警告は、現在の最適化設定によってハードウェア スタックの使用量が増加する可能性がある事を示し、プログラムのハードウェア スタック使用量を検討する際にこの事を考慮に入れる必要がある事をユーザに警告します。
Debugger might use hardware stack	一部のデバッガが使用するデバッグ エグゼクティブは、デバイスのハードウェア スタック レベルを消費します。この警告には、デバッガによって要求されるレベルが示されます。これは、プログラムのハードウェア スタック使用量を検討する際に考慮に入れる必要があります。

5.3 サポートするデータ型と変数

C 言語で使われる値の保存サイズ、書式、レンジは、その値に割り当てられたデータ型によって決まります。これらの値を保存する変数とオブジェクトは、C 言語標準に沿って定義されます。

5.3.1 識別子

識別子は C オブジェクトおよび関数を表現するために使われ、厳密な規則に従う必要があります。

C 識別子は文字と数字の組み合わせであり、アンダースコア「_」は文字として数えられます。識別子の頭文字に数字を使う事はできません。アンダースコアは識別子の頭文字に使えますが、そのような識別子はコンパイラ用に予約されているため、ユーザ プログラム内の C ソースコードによって定義すべきではありません。この制限は、アセンブリコード領域の識別子には適用されません。

識別子では大文字と小文字を区別するため、main と Main は同じではありません。

識別子では、最大 255 文字が有意です。2 つの識別子が 255 文字目よりも後でのみ異なる場合、コンパイラはそれらを同じシンボルであると見なします。

5.3.2 整数データ型

MPLAB XC8 コンパイラは、1/2/3/4/8 バイト幅の整数データ型とビット型をサポートします。表 5-3 に、整数データ型とそれらのサイズおよび算術型を示します。各型グループの既定値型は太字で表記しています。

表 5-3. 整数データ型

型	サイズ (ビット数)	算術型
__bit	1	符号なし整数
signed char	8	符号付き整数
unsigned char	8	符号なし整数
signed short	16	符号付き整数
unsigned short	16	符号なし整数
signed int	16	符号付き整数
unsigned int	16	符号なし整数
__int24	24	符号付き整数
__uint24	24	符号なし整数
signed long	32	符号付き整数
unsigned long	32	符号なし整数
signed long long	32/64	符号付き整数
unsigned long long	32/64	符号なし整数

__bit 型と__int24 型は、本処理系でのみ利用可能な非標準の型です。PIC18 または拡張ミッドレンジ デバイス向けにビルドする場合、long long 型は 64 ビットの C99 標準型です。しかし、C90 に準拠するプロジェクトまたはその他のデバイスをターゲットとするプロジェクトの場合、本処理系は long long 型のサイズを 32 ビットに制限します。

全ての整数値はリトルエンディアン形式で表現されます(最下位バイト(LSB) は最も低いアドレスに格納されます)。

型に符号属性が指定されない場合、char および__bit 型は unsigned となり、その他の型は全て signed となります。__bit 型に対する符号指定は無意味です。

符号付き値は 2 の補数整数値として格納されます。

各型が保持可能な値のレンジは、『Microchip Unified Standard Library Reference Guide』内の<limits.h> 向け宣言にまとめて記載しています。そこに記載されているシンボルはプリプロセッサ マクロであり、それらは <limits.h>をソースコードにインクルードする事で利用可能になります。C 言語標準ではデータ型のサイズを細部にわたって規定しません。このため先のマクロを利用し、各型で表現可能な値の範囲を制限する事により、本処理系におけるコード移植性を高める事ができます。__int24 型関連のマクロは、本処理系でのみ利用可能な非標準マクロです。long long 型関連の値のレンジは、使用する C 規格に依存します。

exact-width 型関連の値(int8_t、uint32_t 等)を定義するマクロは <stdint.h> 内で定義されています。

5.3.2.1 ビット データ型と変数

MPLAB XC8 C コンパイラは、__bit 型指定子によるビット整数型をサポートします。

ビット変数の振る舞いは通常の unsigned char 型変数とほとんど同じですが、値 0 または 1 を保持する事しかできません。8 個のビット オブジェクトがメモリの 1 つのバイトに格納されるため、ビット変数を使うとフラグを効率的に保存できます。

bit 変数に対する演算は、シングルビット命令(bsf、bcf)を使って実行できます。

```
__ bit init_flag;
```

これらの型は auto 型での利用や関数パラメータとしては利用できませんが、static で修飾する事により関数内でローカルに定義できます。

例:

```
int func(void) {
    static __bit flame_on;
    // ...
}
```

関数のプロトタイプ内で通常の方法で__bit キーワードを使う事により、関数から bit を返す事ができます。返された値は STATUS レジスタのキャリーフラグに保存されます。

bit 型へのポインタを宣言する事も、bit オブジェクトのアドレスをポインタに割り当てる事もできません。さらに、bit 変数を静的に初期化する事もできません。このため、それらの変数にはコード内で非ゼロの開始値(すなわち 1)を代入する必要があります。bit 修飾されたオブジェクトは、__persistent 修飾されていない限り、起動時にクリアされます。

bit 変数に他の整数型(8 ビット以上)を割り当てた場合、最下位ビットだけが使われます。

例:

```
int data = 0x54;
__bit bitvar;
bitvar = data;
```

data の LSb は 0 であるため、bitvar は代入によってクリアされます。これは__bit 型を設定するのであって、_Bool型を設定するのではありません(5.3.3 「ブーリアン 型」参照)。

bit オブジェクトとそれらを保持するセクションに割り当てられるアドレスは全てビットアドレスです。絶対アドレスの bit 変数(5.4.4 「絶対アドレス変数」参照) の場合、コード内でビットアドレスを指定する必要があります。これらのビットアドレスをその他の変数によって使われるバイトアドレスと比較する際は注意が必要です。

5.3.3 ブーリアン型

本コンパイラは、真偽値を保持するために使われる_Bool 型をサポートします。

この型の変数が保持する値は整数ではなく、式内での振る舞いは__bit 型整数とも異なります。値 0 を_Bool 型に変換した結果は 0 (偽)となり、0 以外の値は全て 1 (真) に変換されます。整数__bit 型の場合、値は最下位ビットに切り詰められます。

<stdbool.h>ヘッダ内で true および false マクロが定義されており、これらは_Bool 型および bool マクロ(_Bool 型への拡張)で使えます。
例:

```
#include <stdbool.h>
_Bool motorOn;
motorOn = false;
```

C90 規約でコンパイルする場合、_Bool は使えませんが、<stdbool.h>をインクルードすれば bool 型が使えます。ただし、これは単なる unsigned char 向けの typedef に過ぎません。

5.3.4 浮動小数点データ型

MPLAB XC8 コンパイラは、32 ビットと 24 ビットの浮動小数点型をサポートします。前者は IEEE 754 32 ビット フォーマットであり、後者はこれを 24 ビットに切り詰めたフォーマットです。C99 への準拠を選択した場合、浮動小数点型のサイズは自動的に 32 ビットに設定されます。24 ビットの浮動小数点型が明示的に選択された場合、本コンパイラは C90 ライブラリを使います。表 5-4 に、浮動小数点データ型とそれらのサイズおよび算術型を示します。

表 5-4. 浮動小数点データ型

型	サイズ (ビット数)	算術型
float	24 / 32	Real

.....続き		
型	**サイズ (ビット)**	**算術型**
double	24 / 32	Real
long double	double と同サイズ	Real

float 型と double 型では 24 ビット フォーマットが既定値です。オプション-fshort-float および-fshort-double を使うと、これを明示的に指定する事もできます。32 ビット フォーマットは、double 値に対して-fno-short- double オプションが指定された場合と float 値に対して-fno-short-float オプションを指定された場合に使われます。

変数は float および double キーワードを使って宣言でき、宣言された変数はそれらのキーワードと同じ型の値を保持します。浮動小数点型は常に符号付きとして扱われ、浮動小数点型の指定時に unsigned キーワードを使う事は禁じられています。long double として宣言された型は、double として宣言された型と同じ書式を使います。全ての浮動小数点値はリトルエンディアン形式で表現されます(最下位バイト(LSB) は最も低いアドレスに格納されます)。

本処理系の 32 ビット浮動小数点型は完全な IEEE 処理系に比べて厳格ではなく、以下の規則に従います。

非正規化数(表現できない小さな数) は保持されず、値 0 として解釈されます。-1.17549435E-38 と 1.17549435E-38 の間の値を表現可能な浮動小数点値は 0.0 以外に存在しません。これを非正規レンジと呼びます。ルーチンの非正規結果は 0 にフラッシュされます。負の 0 結果は生成されません。

ルーチンに対する非数(NaN)の引数は無限大として解釈されます。NaN 結果は加算、減算、乗算、除算ルーチンで(たとえ NaN が期待される場合であっても)決して生成されず、代わりに正しい符号を持つ無限大結果が生成されます。負の値の平方根は、「distinguished」 NaN (エラーを返すために使われる既定値 NaN)を返します。

無限大の引数は全ての演算に対して「有効」であり、その符号で表現可能な「最大数」として振る舞います。従って、+inf + -inf の結果は定義上は有効な結果となりませんが、本処理系では 0 です。

表 5-5 に示す 32 ビットおよび 24 ビット浮動小数点型の書式は、どちらも以下に従います。

- 「符号」は符号ビットであり、値の正/負を示します。
- 「バイアスした指数部」は 8 ビット幅であり、127 の下駄を履かせたバイアス形式で保存されます(0 の指数は 127 として保存されます)。
- 「仮数部」は小数点の右側(小数点以下)を表します。小数点の左側には暗黙の 1 ビットが置かれます。この暗黙ビットの値は、浮動小数点値がゼロの場合は「0」、それ以外の場合は常に「1」です。0 値は、0 の指数部によって表されます。

浮動小数点型の値は $(-1)^{符号} \times 2^{(指数部-127)} \times 1.$ 仮数部 として表せます。

表 5-5. 浮動小数点型の書式

書式	符号	バイアスした指数部	仮数部
IEEE 754 32 ビット	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
IEEE 754 24 ビット バージョン	x	xxxx xxxx	xxx xxxx xxxx xxxx

IEEE 754 32 ビット フォーマットの例を表 5-6 に示します。仮数部の最上位ビット(MSb)は暗黙ビット(小数点の左側のビット)です。このビットは、指数部が 0 ではない限り 1 であると見なされます。

表 5-6. IEEE 754 浮動小数点(32 ビット)型の例

書式	値	バイアスした指数部	1. 仮数部	10 進数
32 ビット	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		(251)	(1.302447676659)	—
24 ビット	42123Ah	10000100b	1.001001000111010b	36.557
		(132)	(1.142395019531)	—

表 5-6 の 32 ビット値は以下の手順により計算できます。

符号ビットは「0」、「バイアスした指数部」は 251 であるため、指数部の値は 251 - 127 = 124 です。「1.仮数部」の小数点の右側の 2 進値を 10 進値に変換し、2^{23} で除算する事により、値 0.302447676659 が得られます。この小数値に 1 を加算します。以上の 2 つの値から、浮動小数点値は以下のように求まります。

$-1^{0} \times 2^{124} \times 1.302447676659$

これは以下の値に近似できます。

2.77000e+37

2 進表記の浮動小数点値は時に誤解を招きます。「有限サイズの浮動小数点型は全ての浮動小数点値を厳密に表現できるわけではない」という事に注意が必要です。浮動小数値の指数部のサイズ (ビット数)によって保持可能な値のレンジが決まり、仮数部のサイズ (ビット数)は厳密に表現可能な値の間隔に関係します。24 ビット フォーマットが表現可能な値のレンジは 32 ビット フォーマットとほぼ同じですが、厳密に表現可能な実数軸上の位置間隔は 32 ビット フォーマットよりも大きくなります。

例えば 24 ビットの浮動小数点型は 95000.0 を厳密に表現できますが、厳密に表現可能な次に大きな値は 95002.0 であり、2 つの値の中間の値は丸め処理されるため厳密に表現できません。このため、C コードにおける浮動小数点値の比較では、期待通りの結果が得られない場合があります。

例:

```
volatile float myFloat;
myFloat = 95002.0;
if(myFloat == 95001.0)      // value will be rounded
    PORTA++;                // this line will be executed!
```

この場合、比較した 2 つの値は異なるにも関わらず、if 式の結果は「真」と評価されます。

32 ビット フォーマットの浮動小数点型も 95000.0 を厳密に表現でき、厳密に表現可能な次に大きな値は(約) 95000.00781 であるため、24 ビット フォーマットよりも高精度です。

浮動小数点フォーマットの特性は、『Microchip Unified Standard Library Reference Guide』に含まれる<float.h>向け宣言にまとめて記載しています。そこに記載されているシンボルはプリプロセッサ マクロであり、それらは<float.h>をソースコードにインクルードする事で利用可能になります。C 言語標準では浮動小数点データ型のサイズと書式を細部にわたって規定しません。このため先のマクロを利用し、各型で表現可能な値の範囲を制限する事により、本処理系におけるコード移植性を高める事ができます。

5.3.5 構造体と共用体

MPLAB XC8 C コンパイラは struct 型と union 型をサポートします。構造体と共用体では、各メンバーに適用されるメモリオフセットだけが異なります。

これらの型は少なくとも 1 バイト幅を持ちます。ビットフィールドと_Bool オブジェクトは完全にサポートされます。

__bit 型オブジェクトは構造体と共用体のメンバーにはなれません。

構造体と共用体は、関数の引数および戻り値として自由に渡す事ができます。構造体と共用体を指すポインタは完全にサポートされます。

5.3.5.1 構造体および共用体の修飾子

本コンパイラは、構造体に対する型修飾子をサポートします。修飾子を構造体に対して適用すると、構造体の全てのメンバーがその修飾子を継承します。下の例では、構造体に対して const 修飾子を適用しています。

```
const struct
    { int
    number; int
    *ptr;
} record = { 0x55, &i }; // record place in program memory
```

この場合、構造体の全体がプログラム空間内に配置され、各メンバーは読み出し専用です。通常、構造体を const 修飾する場合、全てのメンバーを忘れずに初期化する必要があります(実行時に初期化できないため)。

構造体を修飾せずに、構造体メンバーを個々に const 修飾した場合、構造体は RAM 内に配置されますが、各メンバーは読み出し専用です。構造体のメンバーを修飾する場合の例を以下に示します。

```
struct {
    const int number;
    int * const ptr;
} record = { 0x55, &i }; // record placed in data memory
```

5.3.5.2 構造体内のビットフィールド

MPLAB XC8 C コンパイラは、構造体の中のビットフィールドを完全にサポートします。

ビットフィールドは通常 unsigned int 型を使って定義されますが、常に 8 ビットワード内に割り当てられます。最初に定義されたビットは、格納先のワードの最下位ビット(LSb) に格納されます。ビットフィールドが宣言された時、使用中の 8 ビット記憶域ユニット内にそのビットフィールドが収まる場合、ビットフィールドはそこに割り当てられます。ビットフィールドがそこに収まりきらない場合は、新しいバイトが構造体内に割り当てられます。

ビットフィールドは 8 ビット記憶域ユニットの境界をまたぐ事ができません。_Bool 型のビットフィールドもサポートされますが、それらは 1 ビットのサイズに制限されます。

構造体が以下の通りに定義された場合、この構造体には 1 バイトだけが割り当てられます。

```
struct {
    unsigned        lo :1;
    unsigned        dummy :6;
    unsigned        hi :1;
} foo;
```

foo が最終的にアドレス 0x10 にリンクされた場合、フィールド lo はアドレス 0x10 の bit 0 に対応し、フィールド hi はアドレス 0x10 の bit 7 に対応します。dummy の LSb はアドレス 0x10 の bit 1 に対応します。

Note: 1 ビットよりも大きなビットフィールドへのアクセスは非常に非効率的です。コードサイズと実行速度を重視する場合、代わりに char 型を使うか char 型構造体メンバーを使う事を推奨します。一部の SFR はビットフィールドとして定義されている事に注意が必要です。それらのビットフィールドの大部分はシングルビットですが、マルチビット オブジェクトも含まれます。

制御レジスタ内のアクティブなビットの間の未使用領域を埋めるために、無名のビットフィールドを宣言する事ができます。
上の例で dummy が決して参照されない場合、この構造体は以下の通りに宣言できます。

```
struct {
    unsigned        lo :1;
    unsigned           :6;
    unsigned        hi :1;
} foo;
```

ビットフィールドを含む構造体は、コンマで区切った各フィールドの初期値のリストを与える事によって初期化できます。

```
struct {
    unsigned        lo : 1;
    unsigned        mid : 6;
    unsigned        hi : 1;
} foo = {1, 8, 0};
```

無名ビットフィールドを含む構造体を初期化する場合、無名メンバーには初期値を与えません。以下に例を示します。

```
struct {
    unsigned        lo  :1;
    unsigned            :6;
    unsigned        hi  :1;
} foo = {1, 0};
```

この場合、メンバーlo および hi は正しく初期化されます。

サイズが 0 のビットフィールドは特殊なケースです。規格によると、これは「直前のビットフィールド(もし存在するなら)が配置されたアロケーション ユニット内にさらなるビットフィールドは追加されない」という事を示します。

5.3.5.3 無名の構造体と共用体

MPLAB XC8 コンパイラは無名の構造体と共用体をサポートします。これらは識別子を持たない C11 (ISO/IEC 9899:2011) コンストラクタであり、そのメンバーにはコンストラクタの識別子を参照せずにアクセスできます。無名構造体および共用体は他の構造体または共用体の内部に配置する必要があります。

例:

```
struct {
    union {
        int x;
        double y;
    };
} aaa;
aaa.x = 99;
```

union は無名であり、そのメンバーは構造体の一部であるかのようにアクセスされます。

無名構造体または共用体で定義されたオブジェクトは、C99 規約を使っている場合にのみ初期化可能です。

5.3.6 ポインタのタイプ

MPLAB XC8 C コンパイラは、下記の 2 種類の基本的ポインタタイプをサポートします。

データポインタ プログラムによって読み出し可能(場合によっては書き込みも可能)なオブジェクトのアドレスを保持します。

関数ポインタ ポインタを介して呼び出し可能な関数のアドレスを保持します。

これらのポインタタイプは正しく使い分ける必要があります。データポインタ(汎用 void * ポインタも含む)を使って関数のアドレスを保持してはいけません。また、関数ポインタを使ってオブジェクトのアドレスを保持してもいけません。

MPLAB XC8 コンパイラはプログラム内でのポインタに対するアドレス割り当てを全て記録しており、ポインタ変数の型に対して追加の特別な修飾子を使うのではなく、標準的な const および volatile 修飾子を利用でき、本来の意味で利用可能です。

各ポインタが保持するアドレスのサイズと書式は、そのポインタがアドレス指定可能な全てのターゲットのセットに基づきます。この情報は、プログラム内で定義されるポインタごとに固有です。従って、C 言語のポインタであても、プログラム内での使われ方に応じて保持可能なアドレスのサイズと書式が異なる可能性があります。

本コンパイラは、全てのターゲットのメモリ位置とサイズを調査し、各ポインタのサイズとスコープを決定します。ターゲットのサイズは、特に配列または構造体で重要となります。例えば、配列の全てのエレメントにアクセスできるように、ポインタをインクリメントできる必要があります。

5.3.6.1 型修飾子とポインタの組み合わせ

ポインタ型の定義に関しては、最初に C 言語の慣例を調べる事を推奨します。

ポインタは他の C オブジェクトと同様に修飾できますが、ポインタには 2 つの量が関係するため、修飾するには注意が必要です。2 つ量の 1 つはポインタ自身です。ポインタは通常の C オブジェクトと同様に扱われ、ポインタにはメモリが予約されます。もう 1 つはターゲットです(複数の場合あり)。ターゲットはポインタの参照先(ポインタが指す対象)です。ポインタ定義の一般的な形態は以下の通りです。

```
target_type_&_qualifiers * pointer’s_qualifiers pointer’s_name;
```

* の右側にある(pointer’s_name の直前の) 修飾子は、ポインタ変数自体に関係します。* の左側にある型と全ての修飾子は、ポインタのターゲットに関係します。ポインタを逆参照し、ポインタ変数から現在のターゲット値を取得するのが*演算子でもあり、対称性からこれは理に適っています。

volatile 修飾子を使ったポインタ定義の 3 つの例を以下に示します。定義内のフィールドは、スペースを挿入して見やすくしています。

```
volatile int *          vip ;
int          * volatile  ivp ;
volatile int * volatile vivp ;
```

最初の例は、名前が vip のポインタです。ポインタ自体(アドレスを保持する変数) は volatile ではありません。しかし、ポインタが逆参照された時にアクセスされるオブジェクトは volatile として扱われます。言い換えると、このポインタを介してアクセス可能なターゲット オブジェクトは外部で変更される可能性があります。

2 番目の例では、ivp という名前のポインタは volatile であり、ポインタが格納するアドレスは外部で変更される可能性があります。しかし、ポインタが逆参照された時にアクセスされるオブジェクトは volatile ではありません。

最後の例は、vivp という名前のポインタです。ポインタ自体が volatile で修飾され、ポインタが指すオブジェクトも volatile です。

1 つのポインタに複数のオブジェクトのアドレスが割り当てられる可能性があるという事に注意が必要です。例えば、関数へのパラメータとして使われるポインタには、その関数が呼び出されるたびに異なるオブジェクト アドレスが割り当てられます。ポインタの定義は、割り当てられる全てのターゲット アドレスに対して有効である事が必要です。

Note: ポインタに関する表現には注意が必要です。「const ポインタ」という表現は、const オブジェクトを指すポインタなのか、ポインタ自体が const なのか曖昧です。定義を明確にするために「const を指すポインタ」と「const ポインタ」を使い分ける事はできますが、これらの表現が普遍的に理解されるとは限りません。

5.3.6.2 ポインタ ターゲット修飾子

ポインタに割り当てたターゲットをコンパイラによって特定のメモリ空間内に配置させたい場合、__rom および __ram ポインタ ターゲット修飾子が使えます。

これらの修飾子は、ポインタの宣言または定義時にのみ使えます。これらの修飾子を通常の変数に対して使う事はできません。これらはポインタ自体の配置に何も影響を及ぼしません。これらの修飾子は常にコンパイラによって強制され、-maddrqual オプション(4.6.1.1 「addrqual オプション」参照)または#pragma addrqual の影響を受けません。

__rom または__ram で修飾されたポインタに対して矛盾するターゲットが割り当てられるとエラーが発生します。

例:

```
const int __rom * in_ptr;
```

プログラムがデータメモリにあったオブジェクトのアドレスを in_ptr に割り当てた場合、エラーが生成されます。__rom の使用は const 修飾子を暗黙的に意味しますが、コードの意味を明確にするため、const を明示的に使う事を推奨します。

これらの修飾子の使用は、ポインタの全ての宣言を通して一貫している必要があり、同じポインタ変数に対して両方の修飾子を使う事は禁じられています。

5.3.6.3 データポインタ

以下で説明する通り、MPLAB XC8 C コンパイラでは複数の種類のポインタが使われます。ローカル OCG 最適化が選択された場合、以下で(ローカル)と示されているポインタ種類のみが考慮されます(5.3.6.3.2 「ローカル最適化時のポインタ種類」参照)。

ベースラインおよびミッドライン デバイスの場合:

- 共通メモリと 2 つの連続した偶数-奇数バンク (例: バンク 0-1、バンク 6-7)にアクセス可能な 8 ビットポインタ
- データメモリ空間の全体にアクセス可能な 16 ビットポインタ (ローカル)
- 最大 256 バイトのプログラム空間データにアクセス可能な 8 ビットポインタ
- 最大 64K バイトのプログラム空間データにアクセス可能な 16 ビットポインタ (ローカル)
- データ空間メモリの全体と最大 64K バイトのプログラム空間データにアクセス可能な複合ターゲット空間 16 ビットポインタ (ローカル)

PIC18 デバイスの場合:

- アクセスバンクにアクセス可能な 8 ビットポインタ
- データメモリ空間の全体にアクセス可能な 16 ビットポインタ (ローカル)
- 最大 256 バイトのプログラム空間データにアクセス可能な 8 ビットポインタ
- 最大 64K バイトのプログラム空間データにアクセス可能な 16 ビットポインタ (ローカル)
- プログラム空間の全体にアクセス可能な 24 ビットポインタ (ローカル)
- データ空間メモリの全体と最大 64K バイトのプログラム空間データにアクセス可能な複合ターゲット空間 16 ビットポインタ
- データ空間メモリの全体とプログラム空間の全体にアクセス可能な複合ターゲット空間 24 ビットポインタ (ローカル)

ソースコードの予備的なスキャンの後に、各データポインタに適切な種類が割り当てられます。ユーザがポインタの種類を指定する事はできません(ポインタに対する代入のみが可能)。C コードは、必要な情報をコンパイラに提供する必要があります。

ポインタに関する情報とそれらのターゲットは、ポインタ リファレンス グラフ(6.3.5 「ポインタ参照グラフ」参照)に示されます。ポインタ リファレンス グラフはアセンブリ リストファイル内に出力されます。

5.3.6.3.1 両方のメモリ空間にアクセスするポインタ

1 つのデータポインタに対してデータ空間内のメモリが割り当てられた 1 つまたは複数のオブジェクトのアドレスと、プログラムメモリ空間内のメモリが割り当てられた 1 つまたは複数のオブジェクトのアドレスの両方が割り当てられる場合、そのポインタの種類は複合ターゲット空間タイプに分類されます(5.3.6.3 「データポインタ」参照)。この場合、実行時にターゲットメモリ空間を指定できるようにアドレスがエンコードされます。これらのポインタは以下の通りにエンコードされます。

ベースライン/ミッドレンジ デバイスにおける 16 ビット複合ターゲット空間ポインタの場合: アドレスの MSb (bit 15)により、そのアドレスが参照するメモリ空間を指定します。このビットがセットされている場合、そのアドレスはプログラム メモリ空間内のターゲットのアドレスを示し、クリアされている場合はデータメモリ内のオブジェクトのアドレスを示します。このアドレスの残りのビットは、ターゲットメモリ空間内のフルアドレスを表します。

PIC18 デバイスにおける 16 ビット複合ターゲット空間ポインタの場合: データ空間アドレスの最高アドレスより上のアドレスは、プログラム空間メモリ内のオブジェクトのアドレスを示します。

PIC18 デバイスにおける 24 ビット複合ターゲット空間ポインタの場合: アドレスの bit 21 により、そのアドレスが参照するメモリ空間を指定します。このビットがセットされている場合、そのアドレスはデータメモリ内のオブジェクトのアドレスを示し、クリアされている場合はプログラムメモリ内のオブジェクトのアドレスを示します。このアドレスの残りのビットは、ターゲットメモリ空間内のフルアドレスを表します。効率上の理由から、メモリ空間を指定するビットの意味はベースライン/ミッドレンジ デバイスとは逆になっています。

アセンブリコードが C ポインタを参照する場合、本コンパイラはベースライン/ミッドレンジ向けに 16 ビット複合ターゲット空間ポインタを採用し、PIC18 向けに 24 ビット複合ターゲット空間ポインタを採用します。これらのポインタは、たとえアセンブリコード内で正しくフォーマットされたアドレスがポインタに割り当てられていても、全てのメモリ領域に制限なくアクセスし、正常に動作します。

5.3.6.3.2 ローカル最適化時のポインタ種類

ローカル最適化が有効にされている場合、ポインタのサイズと種類は、プログラムによって割り当てられたターゲットに基づくのではなく、純粋にポインタの定義に基づいて決まります。

ポインタ ターゲット指定子__ram および__rom (5.3.6.2 「ポインタ ターゲット修飾子」参照)を使うと、プログラムの実行中にポインタに割り当てられるアドレスを、指定したメモリ空間内に確実に配置できます。ポインタ種類によるアドレスレンジ制限に加えてポインタ ターゲット指定子によるメモリ空間指定によって、ポインタのサイズは予測可能となります。

ローカル最適化によって、指定されたターゲットを持つコードで定義されたポインタは、以下のようなサイズを持ち、有効な C の修飾子となります。

`const` *`type`* `*`
データメモリまたはプログラムメモリ内の任意のオブジェクトのアドレスが割り当て可能です。64kB を超えるプログラムメモリを備えた PIC18 向けにビルドされたプログラムでは、これらのポインタは 3 バイト幅です。その他の場合、これらのポインタは 2 バイト幅です。

`__rom` *`type`* `*`
プログラムメモリ内の任意のオブジェクトのアドレスが割り当て可能です。データメモリ内のオブジェクトのアドレスの割り当てが試みられた場合、エラーが発生します。64kB を超えるプログラムメモリを備えた PIC18 向けにビルドされたプログラムでは、これらのポインタは 3 バイト幅です。その他の場合、これらのポインタは 2 バイト幅です。オプションとして、ポインタサイズを変更する事なく const 指定子を__rom と一緒に使う事ができます。

***`type`* `*`**
データメモリ内の任意のオブジェクトのアドレスが割り当て可能です。通常の動作として、const オブジェクトのアドレスがこのタイプのポインタに割り当てられると、コンパイラは警告を出力します。これらのポインタのサイズは常に 2 バイトです。

`__ram` *`type`* `*`
データメモリ内の任意のオブジェクトのアドレスが割り当て可能です。プログラムメモリ内のオブジェクトのアドレスの割り当てが試みられた場合、エラーが発生します。これらのポインタのサイズは常に 2 バイトです。const 指定子が一緒に使われた場合、それはターゲット オブジェクトを読み出し専用として扱う必要があるという事のみ示します。オプションとして、ポインタサイズを変更する事なく const 指定子を__ram と一緒に使う事ができます。

__far に対するポインタ、__eeprom に対するポインタ、関数ポインタのサイズと動作はローカル最適化設定の影響を受けません。

5.3.6.4 関数ポインタ

MPLAB XC8 コンパイラは、関数へのポインタを完全にサポートします。これらのポインタは、ユーザ定義 C 配列に格納された複数の関数アドレスの中の 1 つを呼び出す場合によく使われます(配列をルックアップ テーブルのように使用)。

ベースライン/ミッドレンジ デバイスの場合、関数ポインタのサイズは常に 1 バイトであり、コンパイラが出力するジャンプテーブル内へのオフセットを保持します。このジャンプテーブルは、デスティネーション関数へのジャンプを格納します。

拡張ミッドレンジ デバイスの場合、関数ポインタは常に 16 ビット幅であり、関数のフルアドレスを保持可能です。

PIC18 デバイスの場合、関数ポインタは 16 ビットまたは 24 ビット幅です。ポインタサイズは、ターゲット デバイス上で利用可能なプログラムメモリの容量によって単純に決まります。

データポインタと同様に、関数ポインタに割り当てられるターゲットは追跡されます。全ての関数命令は必ずプログラム メモリ内に格納されるため、関数ポインタの扱いはデータポインタよりも容易です。ポインタ リファレンス グラフ(6.3.5 「ポインタ参照グラフ」参照) は、データポインタに加えて関数ポインタとそれらの全てのターゲットも示します。ターゲットは、ポインタを介して呼び出し可能と考えられる関数の名前です。

ベースライン/ミッドライン デバイスの注目すべき実行時機能として、Null (値 0)を格納した関数ポインタがあります。これを使って関数を間接的に呼び出すと、コードはそれ自身へ分岐するループから抜け出せなくなります。このエンドレスループは、この誤った状況を検出するために使えます。通常、Null 関数を介する関数呼び出しによりコードのクラッシュ等予期しない挙動が生じます。エンドレスループのジャンプ先のラベルは `fpbase` です。

5.3.6.5 特殊なポインタ ターゲット

ポインタと整数は互換ではありません。ポインタに整数を割り当てると警告が出力されます。

例:

```
const char * cp = 0x123; // the compiler will flag this as bad code
```

整数定数(0x123) にはデスティネーションの型、サイズ、メモリ位置に関する情報は何もありません。整数アドレスが割り当てられたポインタが逆参照される時に期待以外の動作が発生する可能性が非常に高くなります(特に、複数のメモリ空間を備える PIC デバイスでは、その可能性が高い)。

ポインタには必ず C オブジェクトのアドレスを割り当てる必要があります。デスティネーション アドレスで定義された C オブジェクトが存在しない場合、目的に適ったオブジェクトをこのアドレスで定義または宣言する必要があります。オブジェクトのサイズは、ポインタによってアクセス可能なメモリ位置のレンジに適合している必要があります。

例として、プログラムメモリ内のアドレス 0x900 から始まる 1000 個のメモリ位置のチェックサムを生成する場合を想定します。このデータを読み出すためにポインタを使います。以下のようなコードを思い付くかもしれません。

```
const char * cp;
cp = 0x900; // what resides at 0x900???
```

そして、ポインタをデータごとにインクリメントします。

これよりも好ましい方法として、以下のコードが使えます。

```
const char * cp;
extern const char inputData[1000] __at(0x900);
cp = inputData;
// cp is incremented over inputData and used to read values there
```

この場合、コンパイラはターゲットのサイズとメモリ空間を特定できます (配列のサイズと型はポインタとターゲットのサイズを示し、オブジェクト(ポインタではない) の `const` 修飾子はターゲットがプログラムメモリ空間内に位置する事を示すため)。この例の場合、`const` 配列は初期値の指定を必要とせず(5.3.8.1 「const 型修飾子」参照)、これらのアドレスで他のオブジェクトの上に配置可能です。

ポインタがデータメモリ内のオブジェクトにアクセスする必要がある場合、ダミーターゲットとして機能する異なるオブジェクトを定義する必要があります。例えば、データメモリ内のアドレス 0x90 で始まる 10 バイトに対してチェックサムを計算する場合、以下のコードが使えます。

```
const char * cp;
extern char inputData[10] __at(0x90);
cp = inputData;
// cp is incremented over inputData and used to read values there
```

extern 宣言はメモリを消費せず、これは既存オブジェクトの上に配置可能です。

ユーザ定義の絶対アドレス オブジェクトは実行時起動コードによってクリアされず、他の絶対アドレス変数の上に配置可能です。

ポインタの比較(減算)には注意が必要です。

例:

```
if(cp1 == cp2)
    ; // take appropriate action
```

C 規格では、2 つのポインタのアドレスが同じオブジェクトのアドレスである場合にのみ、ポインタの比較が許容されます。唯一の例外として、アドレスは配列の終端を越えて 1 エレメントまで拡張できます。

ポインタと整数定数を決して比較してはいけません。そのような比較はさらに危険です。以下に例を示します。

```
if(cp1 == 0x246)
    ; // take appropriate action
```

NULL ポインタは、ポインタに定数を割り当てる事ができる 1 つの事例であり、コンパイラはこれを正しく処理します。NULL ポインタは数値的に 0 と等価ですが、これは C 規格が定める特殊なケースです。NULL マクロとの比較も許容されます。

5.3.7 定数の型と書式

C 言語における定数は即値であり、各種の書式で指定でき、各種の型を割り当てる事ができます。

5.3.7.1 整数定数

整数定数の書式は基数を指定します。MPLAB XC8 コンパイラは、標準の基数指定子に加えて、C コードで 2 進定数を指定するための基数指定子をサポートします。

表 5-7 に、基数を指定するために使う書式を示します。2 進または 16 進基数を指定するための文字は、16 進数の各桁に使う文字と同様に、大文字と小文字を区別しません。

表 5-7. 基数の書式

基数	書式	例
2 進数	0b*number* または 0B*number*	0b10011010
2 進数	0*number*	0763
10 進数	*number*	129
16 進数	0x*number* または 0X*number*	0x2F

全ての整数定数には、オーバーフローせずに値を保持できるように int、long int、long long int 型のいずれかが割り当てられます。8 進または 16 進定数に対しては、符号付きの型では保持可能な値のレンジが小さすぎる場合に、符号なし型(unsigned int、unsigned long int、unsigned long long int)が割り当てられます。

定数の既定値型は、値の後に添え字を追加する事によって変更できます(例: 23U の U が添え字です)。表 5-8 に、添え字と型 (型の割り当て時に考慮される型) の関係を示します。例えば、10 進数の定数に添え字 l を指定した場合、コンパイラは long int 型を割り当てます。しかし、この型ではその定数を保持できない場合、コンパイラは long long int 型を割り当てます。8 進または 16 進定数に対しては、符号なし型も考慮されます。

表 5-8. 添え字に対する型の割り当て

添え字	10 進数	8 または 16 進数
u または U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l または L	long int long long int	long int unsigned long int long long int unsigned long long int
u または U l または L	unsigned long int unsigned long long intint	unsigned long int unsigned long long int
ll または LL	long long int	long long int unsigned long long int
u または U ll または LL	unsigned long long int	unsigned long long int

以下に、定数に割り当てられる既定値型が不適当であるために問題が生じる可能性のあるコードの例を示します。

```
unsigned long int result;
unsigned char shifter;
shifter = 20;
result = 1 << shifter;   // oops!
```

定数である 1 には int 型が割り当てられるため、シフト演算の結果は int 型です。結果を long 型変数 result に割り当てていますが、この定数をいくらシフトしても、シフトの結果は int 型のサイズより大きくなりません。この例では、値 1 は左へ 20 ビットシフトされ、その結果は 0x100000 ではなく 0 です。

下の例では、添え字を使って定数の型を変更する事で、シフト結果に unsigned long 型を持たせています。

```
result = 1UL << shifter;
```

5.3.7.2 浮動小数点定数

浮動小数点定数の型は、添え字を付けないと double 型であり、添え字 f または F を付けると float 型です。添え字 l または L は long double 型を指定しますが、MPLAB XC8 コンパイラはこれを double 型と同じ型であると見なします。

浮動小数点定数は、小数点付きの 10 進数と指数として指定できます。C99 を使っている場合、浮動小数点定数は 16 進数の仮数と 2 を底とする指数として表す事ができます(指数部は p または P で始まる)。

例:

```
myFloat = -123.98E12;
myFloat = 0xFFEp-22; // C99 float representation
```

5.3.7.3 文字定数と文字列定数

文字定数は、シングル クォーテーション「'」で囲みます(例: 'a')。文字定数は int 型ですが、コンパイラによって後で char 型へと最適化される場合があります。

C 規格に準拠するため、本コンパイラは文字または文字配列における拡張キャラクタセットをサポートしません。その代わりとして、バックスラッシュ文字によるエスケープシーケンスを使います。以下に例を示します。

```
const char name[] = "Bj\370rk";
printf("%s's Resum\351", name);     \\ prints "Bjørk's Resumé"
```

マルチバイト文字定数は、この処理系によってサポートされません。

文字列定数(または文字列リテラル) は、ダブルクォーテーション「"」で囲みます(例: "hello world")。文字列定数の型は const char * です。文字列を構成する文字は、const 修飾された全てのオブジェクトと同様に、プログラムメモリ内に保存されます。

よくある警告は、const ターゲットを指定していないポインタに対して、変更できない文字列リテラルを割り当てた事に起因します。

例:

```
char * cp = "hello world\n";
```

5.3.6.1 「型修飾子とポインタの組み合わせ」を参照し、ポインタを以下の通りに修飾します。

```
const char * cp = "hello world\n";
```

文字列による配列 (ポインタではない) の定義と初期化は例外です。

例:

```
char ca[]= "hello world\n";
```

これは文字列の文字を実際に RAM 配列にコピーします(前述の例のように文字のアドレスをポインタに割り当てるのではありません)。この文字列リテラルは読み出し専用のままですが、配列は読み書き可能です。

MPLAB XC8 コンパイラは同じ文字の並びを持つ複数の文字列に対して同じ記憶域位置とラベルを使います。しかし、データ空間内に配置された配列を初期化するために文字列が使われる場合は例外です。以下に例を示します。

```
if(strncmp(scp, "hello", 6) == 0)
    fred = 0;
if(strcmp(scp, "world") == 0)
    fred--;
if(strcmp(scp, "hello world") == 0)
    fred++;
```

文字列"world" 内の文字と文字列"hello world" 内の最後の 6 文字(最後の 1 文字は Null 終端文字) はメモリ内の同じ文字によって表されていても、文字列"hello" は文字列"hello world" 内の同じ文字とオーバーラップしません(Null 文字の配置が異なるため)。

5.3.8 標準型修飾子

本コンパイラは、組込みアプリケーションの開発で重要となる標準修飾子 const および volatile をサポートします。

5.3.8.1 const 型修飾子

const 型修飾子は、そのオブジェクトが読み出し専用である(変更されない) という事をコンパイラに伝えるために使います。const 宣言されたオブジェクトに対して変更が試みられた場合、コンパイラは警告またはエラーを出力します。

const で修飾された auto オブジェクトとパラメータ オブジェクトの扱いは異なりますが、const を使って宣言されたその他の全てのオブジェクトはプログラムメモリ空間にリンクされる特別な psect 内に配置されます。これらのオブジェクトは絶対アドレスオブジェクトにもできます。プログラムメモリ内の指定アドレスに配置するには、__at(*address*)コンストラクタを使います。下記の例は、オブジェクト tableDef をアドレス 0x100 に配置します。

```
const int tableDef[] __at(0x100) = { 0, 1, 2, 3, 4};
```

実行時に値を代入する事はできないため、通常 const オブジェクトは宣言時に初期化する必要があります。

例:

```
const int version = 3;
```

上記は version を、値 3 を保持する読み出し専用の int 型変数として定義します。しかし、非初期化絶対アドレス extern const オブジェクトを定義でき、プログラムメモリ内でオブジェクトを特定位置で他のオブジェクト上に配置する必要がある場合は便利です。

例:

```
extern const char checksumRange[0x100] __at(0x800);
```

上記は、checksumRange をプログラムメモリ内のアドレス 0x800 に配置された 0x100 文字の配列として定義します。しかし、この定義では HEX ファイル内にはデータを配置した事になりません。

5.3.8.2 volatile 型修飾子

volatile 型修飾子は、そのオブジェクトが一連のアクセスの間でその値を保持するとは保証できないという事をコンパイラに伝えます。この情報は、volatile が宣言されたオブジェクトに対する明らかに重複した参照が最適化によって削除されてしまう(それによってプログラムの動作が変わってしまう)事を防ぎます。

ハードウェアによって変更可能な SFR またはハードウェアを駆動する SFR は全て volatile として修飾されます。また、割り込みコードによって変更される可能性のある全ての変数にも、この修飾子を使う必要があります。

例:

```
#include <xc.h>
volatile static unsigned int  TACTL __at(0x160);
```

volatile 修飾子は、そのオブジェクトに対するアトミックなアクセスを保証しません。8 ビット PIC MCU アーキテクチャは、命令あたり最大で 1 バイトのデータにアクセスする事しかできないため、大部分のオブジェクトの読み書きを完了するには複数命令サイクルが必要です。

volatile オブジェクトにアクセスするためにコンパイラが生成するコードは、通常の変数にアクセスするためのコードと異なる場合があります。一般的に、volatile オブジェクト向けのコードではサイズが増加し、実行速度が低下します。従って、この修飾子は必要な場合にのみ使うべきです。しかし、この修飾子を必要な時に使わないと、期待した動作が得られない可能性があります。

通常 volatile キーワードは、未使用のグローバル変数が削除されてしまう事を防ぐために使われます。volatile ではない変数が一切使われていない場合や、使われていても効果が全くない場合、それらの変数はコンパイラがコードを生成する前に削除される可能性があります。

volatile 変数の識別子のみを記述した C 命令文は、その変数のメモリ位置を読み出してその結果を破棄するコードを生成します。例えば、if PORTB; は完全な命令文であり、PORTB を読み出すアセンブリ コードが生成されます。

一部の変数は、修飾されていなくても volatile であるとして扱われます。プロジェクトがアセンブリコードを含む場合、5.11.3.6 「未定義シンボル」を参照してください。

5.3.9 特殊な型修飾子

特定アドレス空間への静的記憶期間オブジェクトの配置をユーザが制御できるようにするため、MPLAB XC8 C コンパイラは特殊な型修飾子をサポートします。

5.3.9.1 bank 型修飾子

オブジェクトを特定 RAM バンクに配置するために、__bank(*n*)型修飾子と-maddrqual コンパイラ オプションを使います。

PIC デバイスのデータメモリは、複数の RAM バンクで構成されます。本コンパイラは、オブジェクトを利用可能バンクの 1 つに自動的に配置しますが、ユーザによるバンク指定が必要な場合もあります。例えば、アセンブリコードがオブジェクトにアクセスする前にバンクを選択する場合です。この型修飾子を使うと、オブジェクトをバンク 0～バンク 3 のいずれかに配置できます(現時点ではバンク 3 まで選択可能)。

この型修飾子は、静的記憶期間を持つ任意の変数で使えます。下記の例は、playMode をバンク 1 内に配置します。

```
__bank(1) unsigned char playMode;
```

コンパイラ オプション-maddrqual は、これらの修飾子の効果を制御します(4.6.1.1 「addrqual オプション」参照)。このオプションにより、これらの修飾子が使われた時の本コンパイラの対応を設定できます。既定値では、これらの修飾子は無視されます。無視された修飾子はエラーも警告も生成せず、効果はありません。

5.3.9.2 eeprom 型修飾子

__eeprom 型修飾子は、EEPROM を備えたベースラインおよびミッドレンジ デバイスでオブジェクトを EEPROM 内に配置するために使います。選択されたデバイスでこの修飾子がサポートされない場合、この修飾子を使うと警告が生成されます。デバイスが EEPROM を備えているかどうかは、データシートで確認してください。

この型修飾子は、全ての静的記憶期間オブジェクトで使えます。以下の例は、inputData 配列を EEPROM 内に配置します。

```
__eeprom unsigned char inputData[3];
```

EEPROM へのアクセス方法は 5.4.5 「EEPROM 内の変数」を参照してください。

5.3.9.3 far 型修飾子

__far 型修飾子は変数を外部メモリ内に配置するために使います。この修飾子は-maddrqual コンパイラ オプションにより制御されます。

一部の PIC18 デバイスは外部メモリをサポートします。使用するハードウェアが外部メモリをサポートする場合、最初に-mram オプションでそのメモリを指定する必要があります(4.6.1.17 「ram オプション」参照)。以下の例は、追加のデータメモリを 20000h～2FFFFh に配置します。

```
-mram=default,+20000-2FFFF.
```

RAM レンジに追加されたメモリは、__far で修飾された変数専用に使われます。外部メモリへのアクセスは通常のデータメモリへのアクセスよりも効率が劣るため、実行速度は遅くなり、コードサイズは増加します。デバイスの外部プログラムメモリ空間内に配置された unsigned int オブジェクトの例を以下に示します。

```
__far unsigned int farvar;
```

この修飾子は、コンパイラ オプション-maddrqual により制御されます。このオプションは、PIC18 での__far 型修飾子の効果を指定します(4.6.1.1 「addrqual オプション」参照)。このオプションにより、この修飾子が使われた時の本コンパイラの対応を設定できます(既定値では、これらの修飾子は無視されます)。無視された修飾子はエラーも警告も生成せず、効果はありません。

PIC10/12/16 デバイスおよび外部メモリをサポートしない全ての PIC18 デバイスでは、この修飾子は無視されます。

5.3.9.4 near 型修飾子

__near 型修飾子は変数を共通メモリ内に配置するために使います。この修飾子は-maddrqual コンパイラ オプションにより制御されます。

一部の 8 ビット PIC アーキテクチャは、バンクの選択に関係なく常にアクセス可能なデータメモリを実装しています。バンク型メモリにアクセスするにはバンク選択命令が必要ですが、この共通メモリにはバンク選択が不要であるため、コードサイズと実行時間を削減できます。PIC18 デバイスは、アクセスバンク メモリとしてこのメモリを参照します。ミッドレンジおよびベースライン デバイスの一部は、非常に小量の共通メモリを備えています。これらに比べて PIC18 デバイスの共通メモリは大容量ですが、その大きさはデバイスごとに異なります。詳細はデバイスのデータシートを参照してください。

この型修飾子は、静的記憶期間を持つ全ての変数で使えます。

例:

```
__near unsigned char fred;
```

この修飾子は、コンパイラ オプション-maddrqual により制御されます。(4.6.1.1 「addrqual オプション」参照)。このオプションにより、この修飾子が使われた時の本コンパイラの対応を設定できます(既定値では、これらの修飾子は無視されます)。無視された修飾子はエラーも警告も生成せず、効果はありません。

本コンパイラは、一部の一時オブジェクト向けに共通メモリを使う必要があります。共通メモリの残りの空間は汎用的に使えます。本コンパイラは頻繁にアクセスされるユーザ定義オブジェクトを共通メモリ内に自動的に配置します。従って、この修飾子はオブジェクトを特別に共通メモリ内に配置する必要がある場合にのみ使います。

5.3.9.5 persistent 型修飾子

__persistent 型修飾子は、実行時起動コードによって変数がクリアされる事を防ぐために使います。この修飾子を持つ変数は、他の変数とは異なるメモリ領域に保存されます。

既定値では、明示的に初期化されない静的記憶期間の C 変数は起動時にクリアされます。これは C 言語の定義に準拠しています。しかし、リセット事に一部のデータを保持する事が必要な状況もあります。

以下の例では、静的ローカル オブジェクト intvar は起動時にクリアされません。

```
void test(void)
{
static __persistent int intvar;  /* must be static */
```

```
// ...
}
```

5.3.9.6 ram および rom ポインタ ターゲット修飾子

__ram および__rom 修飾子は、ポインタのアクセス先メモリ空間を制限するために使います。これらの修飾子は、そのポインタ自体の配置には影響を及ぼしません。詳細は 5.3.6.3 「データポインタ」を参照してください。

5.3.9.7 section 修飾子

__section()修飾子は、そのオブジェクトをユーザが指定するセクション内に配置します。この修飾子が指定されない場合、コンパイラはそのオブジェクトを既定値セクション内に配置します。この修飾子の詳細な使い方は 5.14.3 「割り当てられたセクションの変更とリンク」参照してください。

5.4 メモリの割り当てとアクセス

ユーザが定義したオブジェクトは、自動的にメモリの特定領域に割り当てられます。必要に応じて、この割り当てを変更できます。以下では、各種メモリ領域とそれらの割り当てについて説明します。

5.4.1 アドレス空間

全ての 8 ビット PIC デバイスはハーバード アーキテクチャを採用しており、データメモリ(RAM)空間とプログラムメモリ空間は分離されています。一部のデバイスは EEPROM も実装しています。

データメモリ (データシートでは汎用レジスタファイルと呼ぶ) は、アセンブリ命令中のビット割り当て幅制約から、複数のバンクで構成されます。バンクは、SFR 内の 1 つまたは複数のビットを設定する 1 つまたは複数の命令によって「選択」されます。デバイスの正確な動作についてはデータシートを参照してください。

汎用 RAM と SFR はどちらも同じデータ空間を共有し、利用可能な全ての RAM バンク内に配置可能です。PIC18 デバイスでは全ての SFR が 1 つの RAM バンク内に配置されますが、ミッドレンジおよびベースライン デバイスでは SFR は各バンクの低い側のアドレス領域内に配置されます。これらのデバイスでは、SFR の配置のために汎用メモリは断片化されるため、大部分の C オブジェクトのサイズは制限されます。

拡張ミッドレンジ デバイスでは、リニアアドレス指定モードによりこの断片化の問題を克服します。この場合、汎用メモリには 1 つの連続したメモリチャンクとしてアクセス可能です。従って、これらのデバイス向けにコンパイルする場合、オブジェクトの最大許容サイズは通常増加します。PIC18 デバイスを使う場合も、定義されたオブジェクトは一般的にデータメモリ全体を使う事ができます。

多くのデバイスは、バンクの選択に関係なくアクセス可能なメモリ領域を備えています。この領域を共通メモリと呼びます。PIC18 のデータシートでは、このメモリが配置されているバンクをアクセスバンクと呼びます。このため、共通メモリはしばしばアクセスバンク メモリと呼ばれます。これらの領域に保存されたオブジェクトにアクセスする場合、バンクを事前に選択するコードは不要であるため、通常は高速でありコードサイズも削減されます。本コンパイラは、可能な限りこのメモリを使うよう試みます。

プログラムメモリ空間は基本的に実行可能コード用に使われますが、データを格納する事もできます。プログラムメモリ内にデータを配置して読み出す方法は、デバイスファミリに応じて各種存在しますが、それらのオブジェクトは全て読み出し専用となります。

5.4.2 データメモリ内のオブジェクト

ほとんどの変数は、最終的にデータメモリ内に配置されます。自動記憶期間オブジェクトと静的記憶期間オブジェクトではメモリの割り当て方法が根本的に異なるため、以下では別々に説明します。

以下で説明する状況に加えて、一部の最適化は、通常ならプログラムメモリ内に配置される const 修飾されたオブジェクトを、より効率的にアクセスできるデータメモリ内へキャッシュしようと試みます(4.6.6.8 「cacheconst オプション」参照)。

5.4.2.1 静的記憶期間オブジェクト

オブジェクトに割り当てられたメモリ位置がプログラムの実行中に固定されたままである(すなわちスタック上の空間に割り当てられない) 場合、そのオブジェクトを静的期間オブジェクトと呼びます。これらのオブジェクトは、コンパイラによっていずれかの RAM バンクに割り当てられます。

この割り当ては 2 ステップで実行されます。すなわち、コード ジェネレータが各オブジェクトを特定のセクション内に配置し、次にリンカがそれらのセクションを事前に決められたメモリ領域内に配置します。セクションの概要は 5.14.1 「コンパイラが生成する psect」を参照してください。

コード ジェネレータは、バンク選択が効率的に処理できるように各オブジェクトを格納するバンクを決定します。この際、コード ジェネレータはオブジェクトの実際のアドレスを知りません。

コンパイラは、プログラムの開始時に格納される値に基づいて、オブジェクトを 3 つのカテゴリに分類して扱います。各オブジェクト カテゴリは、以下のいずれかのセクション ファミリを持ちます(5.14.1 「コンパイラが生成する psect」参照)。

nv これらのセクションは、__persistent で修飾されたオブジェクトを格納します。これらのオブジェクトの値は実行時起動コードによってクリアされません。

bss これらのセクションは、非初期化オブジェクトを格納します。これらのオブジェクトの値は実行時起動コードによってクリアされます。

data これらのセクションは、初期化オブジェクトの RAM イメージを格納します。実行時起動コードは非 0 値をそれらに複製します。

実行時起動コードの動作については、5.9 「メイン、スタートアップ、リセット」を参照してください。

5.4.2.1.1 static オブジェクト

全ての static オブジェクトは静的記憶期間オブジェクトです。関数内で定義されスコープが関数内部に制限されるローカル static オブジェクトもこれに含まれます。ローカル static オブジェクトもポインタによって参照可能であり、ポインタを介して明示的に変更されない限り、それらを内部で定義している関数が次に呼び出されるまで値を保持します。

static オブジェクトは、一度代入された初期値をプログラムの実行中に保持します。従って、これらのオブジェクトは初期化 auto オブジェクトよりも効率的なコードを生成します。なぜなら、初期化 auto オブジェクトの場合、それらを内部で定義しているブロックの実行が開始されるたびに値が代入されるためです。しかし、auto オブジェクトとは異なり、static オブジェクトの初期化子は定数式である事が必要です。

const として指定された全ての static 変数は、プログラムメモリ内に保存されます。

5.4.2.1.2 オブジェクト サイズの制限

静的記憶期間オブジェクトのサイズは、利用可能なデバイスメモリのサイズによって制限されますが、これ以外の要件によって制限される場合もあります。

拡張ミッドレンジ デバイス向けにコンパイルする場合、一般的に各オブジェクトのサイズは利用可能なデータメモリの総容量によってのみ制限されます。利用可能な RAM バンクの 1 つに収まりきらないオブジェクトは複数バンクにまたがって配置され、デバイスのリニア データメモリ機能を使ってアクセスされます。通常、リニア メモリアクセスはオブジェクトに直接アクセスするよりも低速です。

PIC18 デバイス向けにコンパイルする場合も、一般的に各オブジェクトのサイズは利用可能なデータメモリの容量によってのみ制限されます。命令セットは、複数の RAM バンクにまたがるオブジェクトを許容しますが、そのようなオブジェクトにアクセスするコードのサイズは一般的に大きくなり、実行速度は低下します。

ベースラインおよびその他のミッドレンジ デバイスの場合、オブジェクトは 1 つのバンク内に完全に収まる必要があります。このため、オブジェクトのサイズは、RAM バンク内で利用可能な最大空間のサイズ以下に制限されます。

5.4.2.1.3 既定値のメモリ割り当てを変更する

静的記憶期間オブジェクトの既定値メモリ割り当ては、以下のいずれかの方法で変更できます。

- メモリ位置を予約する
- 指定子を使う
- オブジェクトを絶対アドレス指定にする
- オブジェクトを固有のセクション内に配置し、そのセクションをリンクする

データメモリの特定位置を他の目的に使うために、オブジェクトがそれらのメモリ位置に割り当てられる事を防ぐ必要がある場合、メモリ調整オプションを使ってそれらのメモリ位置を予約するのが最善の方法です。
詳細な方法は 4.6.1.18 「reserve オプション」を参照してください。

__bank( )指定子を使うと、オブジェクトを特定の RAM バンク内に配置できます(5.3.9.1 「bank 型修飾子」参照)。

少数のオブジェクトだけデータ空間メモリ内の特定アドレスに配置する場合、それらのオブジェクトを絶対アドレス指定にできます(5.4.4 「絶対アドレス変数」参照)。絶対アドレス オブジェクトのアドレスは既知であるため、それらのオブジェクトは通常のメモリ割り当て手順に従いません。

__section( )指定子を使ってオブジェクトを固有のセクション内に配置し、そのセクションを必要な位置でリンクできます(5.14.3 「割り当てられたセクションの変更とリンク」参照)。

© 2022 Microchip Technology Inc. and its subsidiaries

5.4.2.2 自動記憶期間オブジェクト

自動記憶期間オブジェクト(auto オブジェクト、パラメータ オブジェクト、一時変数等) は、一般的にコンパイラが実装するスタック上の空間に割り当てられます。

MPLAB XC8 コンパイラは 2 種類のスタック (コンパイルド スタックとソフトウェア スタック)を実装します(5.2.4.2 「データスタック」参照)。各 C 関数は、その内部の自動記憶期間オブジェクト向けにこれらのスタックの 1 つを使うようコンパイルされます。表 5-9 に、スタックの選択が関数の再入可能性に及ぼす影響を示します。

表 5-9. 関数モデルの実装

使用データスタック	関数モデル	対応デバイスファミリ
コンパイルド スタック	再入不可	全デバイス
ソフトウェア スタック	再入可能	拡張ミッドレンジ、PIC18

再入可能関数モデルをサポートしないデバイス向けにコンパイルする場合、全ての関数はコンパイルド スタックを使うようにエンコードされ、再入不可関数となります。

拡張ミッドレンジおよび PIC18 デバイスの場合、既定値によりコンパイラは全ての関数向けに再入不可モデルを使いますが、`-mstack` オプション(4.6.1.22 「stack オプション」参照)を使う事でコンパイラによる関数モデルの割り当てを変更できます。このオプションの引数で `software` を指定すると、コンパイラは各関数向けに常に再入可能モデル (ソフトウェア スタック)を選択します。このオプションを `hybrid` に設定すると、コンパイラは各関数のコンパイル方法を自動的に選択します。関数が再帰呼び出しを利用していない場合、本コンパイラは再入不可モデル(コンパイルド スタック)を使います。複数のコールグラフに現れる関数(すなわちメインライン コードと割り込みコードの両方から呼び出される関数)または 1 つのコールグラフでループ内に現れる関数(すなわち再帰的に呼び出される関数) 向けに本コンパイラは再入可能モデルを使います。`hybrid` モードにより、プログラム内で再帰的呼び出しが使える一方、その必要がない関数には効率的なコンパイルド スタックが使えます。

あるいは、関数の定義時に関数指定子を使う事で、個々の関数の関数モデルを変更する事もできます。`__compiled` または`__nonreentrant` 指定子(どちらも意味は同じ)を使う事で、その関数向けにコンパイルド スタックを使うよう指定します。これらの指定子は、他の関数には影響を及ぼしません。あるいは、`__software` または`__reentrant` 指定子を使う事で、その関数向けにソフトウェア スタックを使うよう指定します。

関数指定子は`-mstack` オプションの設定よりも優先されます。例えば`__mstack=compiled` オプションが指定されている場合に 1 つの関数に対して`__software` (または`__reentrant`) 指定子が使われた場合、その関数のみソフトウェア スタックを使い、その他の全ての関数はコンパイルド スタックを使います。これらの関数指定子は、`hybrid` モードで本コンパイラが選択した全てのモードを上書きします。

`-mstack=compiled` オプションまたは`__compiled` (`__nonreentrant`) 指定子によってコンパイルド スタックを使うよう指定されている関数がプログラムの複数のコールグラフに現れる場合、関数複製機能が自動的に有効になります(5.8.7 「関数の複製」参照)。再入不可関数を複製する事でその関数は複数のコールグラフ内で呼び出し可能となりますが、その関数を再帰的に呼び出す事はできません。

関数内で定義された auto オブジェクトは宣言された順番にメモリに配置される必要があるのに対し、パラメータは常に辞書式順序に基づいてメモリに配置されます。実際、1 つの関数の auto オブジェクトが多数の RAM バンク内に配置される場合があります。

標準の `const` 修飾子を auto オブジェクトに使う事で、それらのオブジェクトを読み出し専用にできます。しかし、それらのオブジェクトはスタック上で保存されない可能性があります。本コンパイラによるそのようなオブジェクトの配置方法は、5.1.2.4 「const auto オブジェクト」を参照してください。

5.4.2.2.1 オブジェクト サイズの制限

コンパイルド スタックは、異なる RAM バンク内に配置された複数のブロックで構成可能であるため、スタックの総サイズはデバイスで利用可能なメモリのみによって概ね制限されます。しかし、スタック内の個々のオブジェクトのサイズは、RAM バンク内で利用可能な最大空間のサイズ以下に制限されます。

ソフトウェア スタックは常にメモリの単一ブロックに配置されますが、このメモリブロックはバンク境界をまたいでも構いません。ソフトウェア スタックの最大サイズは、一般的に、残されている空きデータ空間の容量によって制限されます。コンパイル時も実行時もスタック オーバーフローは監視されませんが、関数が定義しているスタックベース オブジェクトの数が多すぎるとコンパイラエラーが生成されます。

各関数が定義可能なスタックデータの量は PIC 世代別の命令セットにより制限されるため、コンパイラはこれらのデータ割り当ての超過を検出する事ができます。この制限は PIC18 に対して 127 バイト、拡張ミッドレンジ デバイスに対して通常 31 バイトです。後者の制限は超過しても構いませんが、31 バイトが追加されるたびにオブジェクトへのアクセスにオフセットが要求されるため、コンパイラによって出力される命令の数が増加します。

© 2022 Microchip Technology Inc. and its subsidiaries

再入可能関数が他の再入可能関数を呼び出す場合、呼び出し先関数へパラメータがロードされるとスタックポインタはインクリメントします。これにより、新しい top-of-stack 位置から呼び出し元関数によって定義されたスタックベース オブジェクトまでのオフセットが増加します。このオフセットが過大になると、呼び出し元関数内のスタックベース オブジェクトへのアクセスが試みられた時に警告(1488)またはエラーが生成されます。呼び出し先再入可能関数が値を返す場合、これもスタック上に配置される可能性があるため、同様の状況が生じます。これらの理由により、スタック深さの全部を全ての関数向けに使えない場合があります。

5.4.2.2.2 既定値のメモリ割り当てを変更する

全ての自動記憶期間オブジェクトは 1 つのスタック上に配置されるため、それらを個々に移動させる手段はありません。それらを絶対アドレスで指定する事はできず、__section()指定子を使って固有のセクションに割り当てる事もできません。

5.4.3 プログラム空間内のオブジェクト

静的記憶期間変数と const 修飾された変数はプログラムメモリ内に配置されます。一部の auto オブジェクトもプログラムメモリ内に配置される場合があります(5.1.2.4 「const auto オブジェクト」参照)。一部の最適化(4.6.6.8 「cacheconst オプション」参照) は、const 修飾されたオブジェクトを、より効率的にアクセスできるデータメモリ内にキャッシュしようと試みます。

プログラムメモリ内のデータへのアクセスは、データメモリ内のオブジェクトにアクセスするよりも大幅に低速であり、アクセスに関連するコードも大きくなります。

拡張ミッドレンジ デバイスはプログラムメモリを直接読み出す事ができますが、コンパイラはデータを retlw 命令として保存します。データの読み出しには 2 通りの方法が使われます。 1 つはミッドレンジデバイスのように retlw を命令として call する事で W に読み出す方法。もう 1 つはレジスタ間接によるアドレス指定読み出しです。コンパイラは、データを読み出す際に最も効率の良いアクセス方法を選択します。

PIC18 デバイスの場合、データはプログラムメモリ内に個々のバイトとして保存できます。これは、テーブル読み出し命令を使って読み出せます。

その他の 8 ビット PIC デバイスでは、プログラム空間を直接読み出す事はできません。これらのデバイスの場合、本コンパイラは retlw 命令を使ってプログラムメモリ内にデータを保存します。この命令を呼び出すと W レジスタ内の 1 バイトのデータが返されます。コンパイラは、プログラムメモリが直接読み出されているかのように見せるために必要なコードを生成します。

5.4.3.1 オブジェクト サイズの制限

const 修飾されたオブジェクトのサイズは、利用可能なデバイスメモリのサイズによって制限されますが、これ以外の要件によって制限される場合もあります。

ベースライン PIC デバイスの場合、1 つの const オブジェクトの最大サイズは 255 バイトです。しかし、オブジェクトの総サイズがデバイスで利用可能なプログラムメモリのサイズを超えない限り、const オブジェクトを必要なだけいくつでも定義できます。

ミッドレンジ デバイスの場合、1 つの const オブジェクトの最大サイズは、利用可能なプログラムメモリのサイズによって制限されます。

PIC18 デバイスの場合、const 修飾されたオブジェクトの最大サイズは、利用可能プログラムメモリのサイズまたは 0xFFFF のどちらか小さい方によって制限されます。このため、デバイスが実装しているプログラムメモリのサイズがいくら大きくても、オブジェクトのサイズは 64kB 未満に制限されます。しかし、プログラムメモリ内の「アドレス 0～データメモリの最高アドレスに等しいアドレス」までの領域は、通常このデータを保持するために使われないという事に注意が必要です。

const データ自体に加えて、プログラムメモリ内のデータにアクセスするための短いコードがコンパイラによって出力される場合があります。この追加コードは、const 修飾オブジェクトの大きさまたは個数に関係なく 1 度だけインクルードされますが、const オブジェクトの最大サイズがさらに制限される可能性があります。

5.4.3.2 既定値のメモリ割り当てを変更する

const 修飾オブジェクトの既定値メモリ割り当ては、以下の方法のいずれかにより変更できます。

- メモリ位置を予約する
- オブジェクトを絶対アドレス指定にする
- オブジェクトを固有のセクション内に配置し、そのセクションをリンクする

- キャッシュによる最適化を使う

1 つまたは複数のプログラムメモリ アドレスを他の目的に使うために、変数がそれらのアドレスに割り当てられる事を防ぐ必要がある場合、メモリ調整オプションを使ってそれらのアドレスを予約するのが最善の方法です。詳細な方法は 4.6.1.18 「reserve オプション」を参照してください。

少数の const オブジェクトだけプログラム空間メモリ内の特定アドレスに配置する場合、それらのオブジェクトを絶対アドレス指定にできます。絶対アドレス変数については、5.4.4 「絶対アドレス変数」を参照してください。

プログラムメモリ内のオブジェクトは__section()指定子を使って固有のセクションに割り当てる事もでき、このセクションを必要な位置でリンクする事ができます(5.14.3 「割り当てられたセクションの変更とリンク」参照)。

プログラムメモリ内の一部のオブジェクトは、キャッシュ最適化(4.6.6.8 「cacheconst オプション」参照)を有効にする事により、データメモリ内でキャッシュできます。個々のオブジェクトをキャッシュする事はできません。どのオブジェクトがこの最適化によってキャッシュされるかは、プログラムを変更して再ビルドすると変化する可能性があります。

5.4.4 絶対アドレス変数

オブジェクトは、__at(*address*)コンストラクタを使って宣言する事により、特定アドレスに配置できます。*address* は、その変数のメモリ内の位置を表す整数定数です。そのような変数を絶対アドレス変数と呼びます。

変数の絶対アドレス化は、オブジェクトをユーザ定義位置に配置できる最も容易な方法ですが、そのアドレスはコンパイルの前に既知である事が必要であり、オブジェクトごとに再配置を指定する必要があります。

5.4.4.1 データメモリ内の絶対アドレス オブジェクト

静的記憶期間オブジェクトとファイルスコープを有するオブジェクト(すなわち、関数内で定義された static オブジェクトとスタックベース オブジェクトを除く全てのオブジェクト) はデータメモリ内の絶対アドレスに配置できます。

例:

```
volatile unsigned char Portvar __at(0x06);
```

上記は、データメモリ内のアドレス 06h に配置された変数 Portvar を宣言します。注意として、__at()コンストラクタは定義内の変数識別子の前または後に置く事ができますが、C90 規約に互換とする場合、識別子の後に置く必要があります。

Note: 絶対アドレス オブジェクトの定義によってメモリが断片化され、リンカがその他のオブジェクトを配置できなくなる可能性があります。絶対アドレス オブジェクトを定義する必要がある場合、それらを RAM バンクの上端または下端に配置する事で、残りのメモリ領域が小さなチャンクに断片化されないよう心掛けます。

アドレスが汎用 RAM (GPR) 内である場合、コンパイラは絶対アドレス オブジェクト向けの記憶域を使用中としてマークします。PIC18 デバイスでは、アドレス 0 は GPR の一部であると見なされないので注意が必要です。このアドレスをオブジェクトのアドレスとして使うと NULL ポインタと同義になります。多くの場合、コンパイラはこのアドレスを内部定義オブジェクト向けに使います。従って、絶対アドレス オブジェクトをアドレス 0 に配置してはいけません。

絶対アドレス変数同士の間のオーバーラップは検出されません。必要ならば、複数の絶対アドレス変数を同じアドレスで定義しても害はありません。絶対アドレス オブジェクトのアドレスがデバイスメモリの外部またはリンカクラスによって定義された GPR の外部であっても、警告は出力されません。

RAM 内の絶対アドレス変数はそれらの定義時に初期化できず、実行時起動コードによってクリアもされません。絶対アドレス変数の定義後に、必要に応じてメインライン コード内の適切な位置でそれらの変数に初期値を代入します。

絶対アドレス指定によってオブジェクトを共通(非バンク型)メモリ内に配置しようと試みてはいけません。この目的には、常に__near 修飾子(5.3.9.4 「near 型修飾子」参照)を使う必要があります。

絶対アドレス bit 変数(5.3.2.1 「bit データ型と変数」参照)を定義する場合、アドレスはビットアドレスで指定する必要があります。ビットアドレスは、バイトアドレスを 8 倍してから、そのバイト内のビットオフセットを加算する事により求まります。下記の例は、mode という名前の bit 変数をアドレス 0x50 のビット位置#2 に配置します。

```
bit mode __at(0x282);
```

拡張ミッドレンジ デバイス向けにコンパイルする場合、絶対アドレス オブジェクトのアドレスは、通常のバンク型メモリアドレスまたはリニアアドレスで指定できます。リニアアドレスは、バンク型メモリの最高アドレスより上のアドレスから始まるため、どちらのアドレスタイプが指定されているかは明確に判別できます。

例:

```
int input[100] __at(0x2000);
```

上記の例では、input がリニアアドレス空間内のアドレス 0x2000 (すなわちバンク型アドレス空間におけるバンク 0 RAM 内のアドレス 0x20) に配置される事は明確です。

5.4.4.2 プログラムメモリ内の絶対アドレス オブジェクト

静的記憶期間を持つ const 修飾オブジェクトとファイルスコープを持つオブジェクトは、プログラムメモリ内の絶対アドレスに配置できます。

例:

```
const int settings[] __at(0x200) = { 1, 5, 10, 50, 100 };
```

上記により、配列 settings がプログラムメモリ内のアドレス 0x200 に配置されます。

注意として、__at()コンストラクタは定義内の変数識別子の前または後に置く事ができますが、C90 規約に準拠する場合は識別子の後に置く必要があります。

非初期化 extern const オブジェクトは絶対アドレスに配置でき、出力ファイルに現れない位置確保用オブジェクトを定義する場合に使えます。

5.4.5 EEPROM 内の変数

EEPROM を内蔵したデバイス向けに、本コンパイラはこのメモリにアクセスするための各種方法を提供します。以下では、それらの方法について説明します。

5.4.5.1 EEPROM 変数

ベースラインおよびミッドレンジ デバイス向けにコンパイルする場合、__eeprom 修飾子を使うと、EEPROM 空間内に名前付き C オブジェクトを配置できます(5.3.9.2 「eeprom 型修飾子」参照)。

__eeprom で修飾されたオブジェクトは、通常の RAM ベース オブジェクトと同様に、必要に応じてクリアまたは初期化されますが、初期化処理は実行時起動コードに含まれません。初期値は HEX ファイルに格納され、デバイスのプログラミング時に EEPROM に書き込まれます。従って、プログラムの実行時に EEPROM の内容を変更してからデバイスをリセットした場合、次の起動時にこれらのオブジェクトはコード内で指定された初期値を格納していません。

以下の例は、EEPROM 内に 2 つの配列を定義します。

```
__eeprom char regNumber[10] = "A93213";
__eeprom int lastValues[3];
```

これらのオブジェクトの初期値は HEX ファイルに格納されています。lastValues の初期値として 0 が使われます。

__eeprom 修飾されたオブジェクトにアクセスするコードは、RAM ベース オブジェクトにアクセスするコードよりも大幅に長く低速になります。EEPROM 内の値を複雑な式で使う場合、「can't generate code」エラー メッセージを防ぐため、それらの値を通常の RAM ベース オブジェクトへ複製してから使う事を推奨します。

5.4.5.2 EEPROM の初期化

EEPROM データ領域へのプログラマによる書き込みをサポートするデバイスでは、__EEPROM_DATA( )マクロを使って、EEPROM への書き込み準備が完了した HEX ファイル内に値を格納する事ができます。このマクロを使って実行時に EEPROM に書き込む事はできません。

例:

```
#include <xc.h>
__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

このマクロは 8 個のデータ値を表す 8 個のパラメータを持ちます。各値のサイズは 1 バイトである事が必要です。未使用値は 0 で指定する必要があります。

__EEPROM_DATA()マクロ引数はアセンブリコードへと展開されます。そのため引数内で記述した定数以外の式、演算子、トークンは全てアセンブリコードで書かれている必要があります (6.1 「MPLAB XC8 アセンブリ言語」参照)。

このマクロを複数回呼び出す事で、EEPROM データを必要なだけ定義できます。このマクロは関数定義の外に置く事を推奨します。

このマクロによって定義された値は、__eeprom で修飾されたオブジェクトと EEPROM 空間を共有しますが、それらのオブジェクトを初期化するために使う事はできません。このマクロによってデータ値を保持するために使われるセクションは、__eeprom 修飾オブジェクトによって使われるセクションとは異なります。これらのセクションのリンク順序は、必要に応じて調整できます(4.6.12 「リンカオプション」参照)。

便宜のために、ターゲットデバイス上で利用可能な EEPROM のバイト数を表す_EEPROMSIZE マクロが利用できます。

5.4.5.3 EEPROM アクセス関数

ライブラリ関数 eeprom_read()および eeprom_write()を使うと、プログラムの実行中に EEPROM を読み書きできます。

これらの関数は、EEPROM を実装したミッドレンジ デバイスで利用できます(9.2.4 「eeprom_read 関数」と 9.2.5 「eeprom_write 関数」参照)。

便宜のために、ターゲットデバイス上で利用可能な EEPROM のバイト数を表す_EEPROMSIZE マクロが利用できます。

5.4.5.4 EEPROM アクセスマクロ

EEPROM アクセス関数のマクロ バージョンも利用できます(9.2.6 「EEPROM_READ マクロ」と 9.2.7 「EEPROM_WRITE マクロ」参照)。

5.4.6 レジスタ内の変数

MPLAB XC8 コンパイラでは、レジスタ内の変数の配置を直接制御する事はできません。register キーワード (auto 変数でのみ使用可能) は報告なしで無視され、変数の配置には何も影響を及ぼしません。

一部の引数は、メモリ位置ではなく W レジスタを介して関数に渡されますが、その関数に関連するコードによって W レジスタ内の値が使えるようにするため、通常それらの値は関数内部のコードによってメモリに書き戻されます。どのパラメータ変数がレジスタを使えるかは、5.7.5 「関数のパラメータ」を参照してください。

5.4.7 動的メモリ割り当て

動的メモリ割り当ては、実行時に任意サイズのメモリブロックをプログラムから要求および解放するための手段です。MPLAB XC コンパイラがサポートする大部分のデバイスでは、何らかの形で動的メモリ割り当てが可能です。

プログラムは malloc()または calloc()ライブラリ関数を呼び出す事で、実行時にメモリのブロックを割り当てる事ができます。オブジェクトの静的または自動的な割り当て(variable-length automatic array は除く)とは異なり、動的割り当てでは、要求サイズは定数式でなくても構いません。割り当てられた全てのメモリブロックにまたがる連続したメモリ領域をまとめてヒープと呼びます。

プログラムは、free()ライブラリ関数を使う事で、不要になったメモリブロックを解放できます。解放されたメモリは、後で必要に応じて再割り当てできます。既に割り当て済みのメモリブロックのサイズを変更する必要がある場合、realloc()関数が使えます。

ターゲット デバイスとオプションの選択に応じて、MPLAB XC コンパイラは動的割り当ての完全なスキームまたはシンプルなスキームを実装できます。また、動的メモリ割り当てが全く許容されない場合もあります。本コンパイラにおける動的メモリ割り当ての詳細な動作については、次の 5.4.7.1 「PIC デバイスの動的メモリ割り当て」で説明します。動的メモリ割り当て関数の構文と使用法は、『Microchip Unified Standard Library Reference Guide』に記載しています。

5.4.7.1 PIC デバイスの動的メモリ割り当て

PIC18 または拡張ミッドレンジ デバイス向けプロジェクトで C99 言語規格が選択された場合、MPLAB XC8 C コンパイラはシンプルな動的メモリ割り当てスキームを実装します。その他のデバイスでは、メモリ構成上の制約により、動的メモリ割り当ては実装されません。C90 言語規格が選択されている場合、動的メモリ割り当ては全てのデバイスで利用できません。

シンプルな動的メモリ割り当てスキームには全ての標準メモリ割り当てライブラリ関数が使えますが、そのアルゴリズムはリソースの消費量を削減するようシンプル化が図られています。

このスキームはデータおよびプログラムメモリ容量の小さなデバイスにとって有利ですが、メモリの無駄な消費を防ぐため、動的メモリ割り当ては慎重に使う必要があります。

メモリ割り当て量は、ヒープ向けに予約された最大サイズ以内であれば、必要なだけ割り当て可能です。割り当て後に解放されたメモリブロックはリンクされたリスト内で維持され、後で追加のメモリが必要になった時の再割り当て用に利用可能です。これらの解放されたブロックの管理(マージ方法) は基本的な手法に限られます。

プログラムの構造は動的メモリ割り当て動作の効率に影響を及ぼします。新しいメモリブロックを割り当てる前に割り当て済みのメモリブロックを解放できれば、解放されたメモリブロックをより効率良く再使用できます。メモリブロックの割り当てと解放をどのような順序で行っても、ヒープ内の未使用メモリは断片化され、将来の割り当てに利用できなくなる可能性があります。可能な限り動的メモリ割り当てを使わないのが賢明です。

0 バイトのメモリを割り当てる場合であっても、1 バイトのメモリが消費され、そのメモリへのポインタが使用可能な場合は通常の方法で返されます。

`realloc()`を使ったメモリの再割り当ては、可能であれば常に要求されたサイズの新しいブロックを割り当てます。そして、既存のデータを新しいブロックに複製した後に古いブロックを解放します。

プログラムの実行中にヒープが拡大し、他のメモリ領域を上書きするかデバイスのメモリを超過する危険性があります。これらの状況を防ぐため、`-mheap` オプションを使ってヒープの最大許容サイズを指定する事ができます(4.6.1.11 「heap オプション」参照)。実行時にメモリ割り当てを試みるとヒープが最大サイズを超えてしまう場合、その割り当ては実行されず、割り当て関数は Null ポインタを返す事で、この状況を知らせます。

ヒープを使うプログラムがスタックベース オブジェクトのメモリ割り当て用にソフトウェア スタックも使う場合があります(プログラム内の各コールグラフ(例: メインライン コードと割り込み関数) はソフトウェア スタックを 1 つずつ使います)。ヒープとスタックはどちらも実行時にサイズが変化するため、静的割り当てオブジェクトによって使われないデバイスメモリ領域を両方で共有する必要があります。ソフトウェア スタックの最大サイズは、`-mstack`オプション(4.6.1.22 「stack オプション」参照)を使って設定できます。`-mstack` と `-mheap`オプションの両方に対して引数 `auto` を指定すると、本コンパイラは空きメモリ空間をスタックとヒープに均等に分配します。

ヒープに割り当てられたメモリは、`heap` という名前の psect によって保持されます。

プログラムによって**使われる**ヒープメモリの最大量はコンパイル時に未知ですが、ヒープ向けに**予約された**最大メモリ量は、コンパイル後にコンパイラが出力するメモリ使用量サマリ内に示されます。

5.4.8 メモリモデル

MPLAB XC8 C コンパイラは、メモリへの変数の割り当てを変更するために固定されたメモリモデルを使いません。メモリ割り当ては完全に自動的に行われ、ユーザがメモリモデルを制御する事はできません。

5.5 演算子と命令文

MPLAB XC8 C コンパイラは全ての ANSI 演算子をサポートし、それらの一部は処理系定義の振る舞いをします(11. 「処理系定義の振る舞い」参照)。以下では、しばしば誤解される事のある演算コードと、本コンパイラが出力可能な追加の演算について説明します。

5.5.1 整数拡張

整数拡張は、一部の式の値の型を変更します。MPLAB XC8 C コンパイラは常に C 規格に準拠して整数拡張を実行しますが、そのような振る舞いを予期していなかったユーザに混乱を招く可能性があります。

演算子に対して複数のオペランドが存在する場合、通常それらは厳密に同じ型である事が必要です。必要に応じてコンパイラはそれらが同じ型となるよう自動的にオペランドを変換します。この変換はサイズの大きい方の型に揃えるため、情報の損失はありません。しかし、型の変換によって期待とは異なるコードの振る舞いが生じる場合があります。これらは標準型変換に分類されます。

演算子の両側のオペランドが同じ型であっても、標準型変換の前に一部のオペランドがより大きな型に無条件に変換される場合があります。この変換を整数拡張と呼びます。本コンパイラは整数拡張を必要に応じて実行します。この動作を制御する(または無効にする)ためのオプションはありません。

整数拡張は列挙型、signed char および unsigned char 型、short int 型、ビットフィールド型を signed int 型または unsigned int 型のどちらかへ暗黙的に変換します。変換した結果が signed int によって表現可能であれば signed int に変換し、そうでなければ unsigned int に変換します。

以下に例を示します。

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

a - b の unsigned char 型の結果は 206 であり、10 未満ではありません。しかし、a と b は減算の前に整数拡張によって signed int 型に変換されます。従って、これらのデータ型による減算結果は-50 (10 未満)であり、if 文の本体は実行されます。

減算結果を unsigned にするには、キャスト(型変換)を適用します。以下の例では、比較を unsigned int 型として実行させています。

```
if((unsigned int)(a - b) < 10)
  count++;
```

ビット単位の補数演算子「~」でもよく問題が起こります。この演算子は値をビット単位で反転します。

例:

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

c の値が 0x55 である場合、~c は 0xAA になるとよく勘違いされます。しかし実際の結果は 0xFFAA (コンパイラが 16 ビット int を使う場合)または 0xFFFFFFAA (コンパイラが 32 ビット int を使う場合)であり、この例の比較結果は「偽」です。状況によっては、このような問題に対してコンパイラは不適合比較エラーを発行します。この問題にもキャストを使って対処できます。

以上のように、整数拡張によって演算は char 型オペランドではなく int 型オペランドを使って実行されます。しかし、オペランドの型が char であっても int であっても演算結果が異ならない場合があります。その場合、本コンパイラはコード効率の向上のために整数拡張を適用しない事があります。

例:

```
unsigned char a, b, c;
a = b + c;
```

厳密に言うと、この命令文では b と c を unsigned int に拡張した後に加算を実行し、加算結果を a の型にキャストした後に結果に代入します。この場合、加算が int として実行されても char として実行されても a に代入される値は同じであるため、コンパイラは前者を使ってエンコードする可能性があります。

上の例で a が unsigned int 型であった場合、C 規格に準拠するために整数拡張が必要です。

5.5.2 ローテート

ローテート演算子は C 言語に含まれていませんが、C コード内でローテート演算を実行する事は可能です。

本コンパイラは、シフト演算子と論理演算子を使ってローテート演算を実行している式を検出し、それらを効率的にコンパイルします。

例:

```
c = (c << 1) | (c >> 7);
```

c が unsigned かつ非 volatile である場合、本コンパイラは 1 ビットの左ローテート演算が意図されていると判断し、PIC MCU のローテート命令を使って出力をエンコードします。2 ビットの左ローテートを実装するコードの例を以下に示します。

```
c = (c << 2) | (c >> 6);
```

このコード最適化は、char より大きな整数型に対しても有効です。この最適化が適用できない場合、またはこのコードを他のコンパイラに移植する場合、シフト演算とビット単位 OR 演算を使ってローテートが実装されます。

5.5.3 switch 文

本コンパイラは、各種のストラテジを使って switch 文をエンコードします。既定値では、本コンパイラは switch 文の中で使われる case 値に基づくストラテジを選択します。各 switch 文ごとに、別々にストラテジが割り当てられます。

ストラテジのタイプは#pragma switch ディレクティブ(5.13.3.10 「#pragma switch ディレクティブ」参照)を使って指定できます。1 つのタイプに対して複数のストラテジが存在する場合があります。

各 switch 文に関する情報(スイッチされる値と case 値のリスト) がアセンブリ リストファイルに出力されます(6.3.4 「switch 文情報」参照)。

case レンジ(lo_value … hi_value:)指定は C90 および C99 規約の機能ではないため、プログラム内で使わない事を推奨します。p1 フロントエンドが使われ、かつ CCI が有効ではない場合、本コンパイラは case レンジを使って switch 文を正しくコンパイルします。p1 フロントエンドは、C90 規約向けにビルドする場合または PIC18 および拡張ミッドレンジ デバイス以外のデバイス向けにビルドする場合に使われます。その他の場合、case レンジ指定は評価されません。

5.6 レジスタの使用

本コンパイラが C ソースコードから生成するアセンブリコードは、PIC MCU レジスタセットの中の特定のレジスタを使います。非常に重要な事として、本コンパイラは「本コンパイラによって生成されたコードだけがこれらのレジスタの内容を変更できる」と想定します。

本コンパイラが使用する特別なレジスタの一覧を表 5-10 に示します。

表 5-10. コンパイラが使うレジスタ

レジスタ名	適用可能デバイス
WREG	全ての 8 ビットデバイス
STATUS	全ての 8 ビットデバイス
PCLATH	ミッドレンジ デバイス
PCL、PCLATH、PCLATU	全ての PIC18 デバイス
BSR	拡張ミッドレンジ、PIC18
FSR	ミッドレンジ デバイス(拡張ミッドレンジは除く)
FSR0L、FSR0H、FSR1L、FSR1H	拡張ミッドレンジ、PIC18
FSR2L、FSR2H	全ての PIC18 デバイス
TBLPTRL、TBLPTRH、TBLPTRU、TABLAT	全ての PIC18 デバイス
PRODL、PRODH	全ての PIC18 デバイス
btemp、wtemp、ttemp、ltemp、lltemp	拡張ミッドレンジ、PIC18

本コンパイラは、*x*temp レジスタを変数として扱います。これらのレジスタは、割り込みコード内で使われた場合に、他のレジスタと同様に保存されます。lltemp レジスタは、拡張ミッドレンジまたは PIC18 デバイス向けにのみ利用可能です。

レジスタ自体が C 言語の lvalue (代入先)である場合、本コンパイラはレジスタ値に対する変更を認識しません。例えば、命令文 WREG = 0; が clrf 命令を使ってエンコードされた場合、コンパイラはこれを W レジスタの変更として見なしません。インライン アセンブリコードによってこれらのレジスタを直接変更してはいけません。下記の例では、インライン アセンブリコードを使って STATUS レジスタ内の ZERO ビットを変更しようとしています。

```
#include <xc.h>
void getInput(void)
```

```
{
    asm("bcf ZERO"); //do not write using inline assembly code
    process(c);
```

表 5-10 に示したレジスタが割り込みコード内で使われる場合、それらは割り込み発生時にハードウェアまたはソフトウェアで保存/復元されます(5.8.4 「コンテクスト スイッチ」参照)。

5.7 関数

関数は C 言語規格に従った通常の方法で書きます。以下では、関数に関連する処理系固有の機能について説明します。

5.7.1 関数指定子

関数のリンクに影響を及ぼす標準 C 指定子 static に加え、以下で説明する各種の非標準関数指定子が存在します。

5.7.1.1 interrupt 指定子

__interrupt()指定子は、その関数が割り込みサービスルーチンであり、それに適するようエンコードする必要がある事を指定します。割り込み関数の詳細は 5.8.1 「割り込みサービスルーチンの書き方」に記載しています。

5.7.1.2 inline 指定子

inline 関数指定子は、その関数に対する呼び出しを、可能であれば関数のボディに置き換える事をコンパイラに指示します。これによりプログラムの実行速度が向上する可能性があります。

インライン関数に関する規則は C99 言語規格と少し異なり、inline 関数指定子を使った(しかし extern ではない)関数向けのファイルスコープ宣言は関数の外部定義として扱われます。これは、コードをビルドするためにその関数の外部定義を追加する必要はないという事を意味します。

インライン化の候補として指定された関数の例を以下に示します。

```
inline int combine(int x, int y)
  { return 2 * x - y;
}
```

その関数をインライン化するか(それとも関数定義を呼び出すか)どうかは、コンパイラが判断します。inline を指定してもその関数がコンパイラによってインライン化されるとは限りません。従って、インライン化されると勝手に思い込まないよう注意が必要です。インライン アセンブリを含む関数はインライン化されません。生成される一部のアセンブリコードも関数のインライン化を阻止します。インライン関数が static オブジェクトを参照している場合、C 規格に従うよう警告が出力されるか、インライン化に成功しません。

コンパイラによってインライン化された関数に対する全ての関数呼び出しは、呼び出し先関数のボディに置き換えられたかのようにエンコードされます。これはアセンブリコード レベルで実行されます。インライン化にはアセンブリ オプティマイザが有効である事が必要であり、高レベルの最適化が必要であるためライセンスなしのコンパイラでは関数のインライン化はできません。たとえ高レベルの最適化が有効であっても、コンパイラは inline 指定された関数を通常の方法でエンコードする場合もあります。

インライン化に成功すると、その関数の呼び出しに関連する呼び出し/戻りシーケンスが不要となるため、プログラムの実行速度は向上します。また、呼び出し命令が実行されないため、ハードウェア スタックの使用量も削減されます。しかし、コールグラフはインライン化の前に生成されるため、インライン化によるスタックサイズの減少はコールグラフに反映されません。

インライン化された関数のボディに関連するアセンブリコードが非常に小さい場合、コードサイズは減少する可能性があります。しかし、インライン化された関数のボディが呼び出し/戻りシーケンスよりも大きい場合、コードサイズは増加します。関数が小さなアセンブリコードしか生成しない場合にのみ、inline 指定子の使用を検討すべきです。関数のボディ内の C コードが大きければ生成されるアセンブリコードのサイズも大きいとは限りません(3.5.13 「関数のサイズを調べる方法を教えてください」参照)。

この指定子は、#pragma inline ディレクティブと同じ効果を有します。オプティマイザが「1 度だけ呼び出される」小さなコードを暗黙的にインライ化する場合もあります(5.13.3.4 「#pragma Intrinsic ディレクティブ」と 6.2 「アセンブリレベルの最適化」参照)。

xc8-cc フラグの-Wpedantic を使う場合、inline キーワードは使えなくなりますが、__inline キーワードは使えます。

5.7.1.3 reentrant / nonreentrant 指定子

__reentrant および__nonreentrant 関数指定子は、関数のスタックベース変数(auto およびパラメータ) 向けに使う関数モデル(表 5-11 参照)を指定します。エイリアスとして__reentrant の代わりに software を使い、__nonreentorant の代わりに__compiled を使う事もできます。これらの指定子は、既定値の割り当てを変更する必要がある場合にのみ使います。

表 5-11. スタック関連の関数指定子

指定子	スタックベース変数のメモリ割り当て
__compiled、__nonreentrant	常にコンパイルド スタックを使用、関数は再入不可
__software、__reentrant	ソフトウェア スタックを使用(サポートされる場合)、関数は再入可能

常に再入可能としてエンコードされる関数の例を以下に示します。

```
__reentrant int setWriteMode(int mode)
{
    accessMode = (mode!=0);
    reutrn mode;
}
```

これらの指定子は、-mstack オプションによる設定を上書きします(4.6.1.22 「stack オプション」参照)。どちらの関数指定子も使われず、かつ-mstack オプションも指定されなかった(または hybrid として指定された)場合、その関数のスタックベース変数向けに使うスタックはコンパイラによって選択されます。

__reentrant 指定子は、ターゲット デバイスがソフトウェア スタックをサポートしている場合にのみ効力を有します。関数によっては、再入可能にできない場合があります。割り込み関数は常にコンパイルド スタックを使う必要がありますが、そこから呼び出される関数はソフトウェア スタックを使えます。ベースラインおよびミッドレンジ デバイス向けにエンコードされる関数は常に再入不可モデル(従ってコンパイルド スタック)を使います。

__software(__reentrant)指定子を繰り返し使うと、ソフトウェア スタックのサイズが大きく増加し、オーバーフローが生じる恐れがあります。コンパイル時にソフトウェア スタックのサイズを正確に知る事はできないため、コンパイラは他の目的で使われるメモリが上書きされそうかどうかを示す警告を出す事ができません。

各デバイスで利用可能なデータスタックについては 5.2.4.2 「データスタック」を参照してください。

5.7.1.4 外部関数

プロジェクトの C ソースファイルの外部で定義された関数(例: 分離したブートローダ プロジェクトまたはアセンブリ モジュール内で定義された関数)を使う場合、コンパイラがそれらの関数への呼び出しを適切にエンコードできるようにするために宣言が必要です。

関数宣言の例を以下に示します。これが宣言である事を明確に示すために、この例では extern 指定子が使われていますが、この指定子を使わなくても構いません。

```
extern int clockMode(int);
```

5.7.2 実行可能コードのメモリ割り当て

関数に関連するコードは、常にプログラムメモリ内に配置されます。

ベースラインおよびミッドレンジ デバイスの場合、プログラムメモリは複数のページで構成されます。プログラムメモリのアドレスはページ境界またぐ箇所でも連続的に処理できますが、あるページ内のコードから別のページ内のラベルへの呼び出しまたはジャンプには、同一ページ内の呼び出し/ジャンプよりも長い命令シーケンスが必要になります。プログラムメモリと命令セットに関する詳細は、使用するデバイスのデータシートを参照してください。

PIC18 デバイスのプログラムメモリはページで構成されません。call および goto 命令は 2 ワード命令であり、それらのデスティネーションに制限はありません。

各関数向けに生成されたコードは、最初にコンパイラによってセクション内に配置されます(5.14.1 「コンパイラが生成する psect」参照)。プログラムメモリがページで構成される場合、短い形態の call および jump 命令が使えるため、オプティマイザは複数の関数を同一セクション内に配置しようと試みます。ミッドレンジでは、これらのセクションはプログラムメモリ内の任意位置にリンク可能です(5.14.2 「既定値リンカクラス」参照)。ただし、ベースライン デバイスでは、関数のエントリポイントは各ページの先頭から 256 アドレス以内に制限されます。

各セクションのベース名を以下に示します全てのプログラムメモリ セクション名のリストは、5.14.1.1 「プログラム空間内の psect」を参照してください。

maintext 関数 main( )向けに生成されたコードはこのセクションに格納されます。この psect には一部の最適化と機能は適用されません。

text*n* これらのセクション(*n* は 10 進数の番号) は、特殊なリンク位置を要求しないその他の全ての実行可能コードを格納します。

5.7.3 既定値の関数メモリ割り当てを変更する

関数の既定値メモリ割り当ては、以下の方法のいずれかにより変更できます。

- メモリ位置を予約する
- 関数を絶対アドレスにする
- 関数を固有のセクション内に配置し、そのセクションをリンクする

データメモリの特定位置を他の目的に使うために、関数がそれらのメモリ位置に割り当てられる事を防ぐ必要がある場合、メモリ調整オプションを使ってそれらのメモリ位置を予約する事を推奨します(4.6.1.18 「reserve オプション」参照)。

個々の関数を既知のアドレスに明示的に配置する最も容易な方法は、絶対アドレス変数と同様に__at(*address*)コンストラクタを使う事です。

絶対アドレス関数に関連するコードが他の絶対アドレス関数のコードとオーバーラップする場合、コンパイラは警告を出力します。絶対アドレス オブジェクトのアドレスがデバイスのメモリの外部またはリンカクラスによって定義されたメモリの外部であっても、警告は出力されません。コンパイラは通常の関数コードを絶対アドレス関数上には配置しません。

以下の例は、絶対アドレス関数をアドレス 400h に配置します。

```
int  __at(0x400) mach_status(int mode)
{
    /* function body */
}
```

注意として、__at( )コンストラクタは定義内の関数識別子の前に置く必要があります(C90 言語規格向けにコードをビルドする場合、この位置では規則に違反します)。しかし、宣言内では__at( )を識別子の後に置く事ができます。その場合、定義内の__at( )コンストラクタは完全に削除できます。
例:

```
int mach_status(int mode)  __at(0x400); // declaration indicates address

int mach_status(int mode) { ...} // defintition can omit the __at()
```

このコンストラクタを割り込み関数で使う事はできません。リセットおよび割り込みベクタ位置の移動方法は、4.6.1.3 「codeoffset オプション」を参照してください。これはブートローダ等のアプリケーション向けに使えます。

絶対アドレス関数向けに生成されたコードは、その関数専用のセクション内に配置されます。セクション名の書式を以下に示します(全てのセクション名を含むリストは 5.14.1 「コンパイラが生成する psect」参照)。

***xxx*_text** 絶対アドレス関数向けのセクションを定義します。*xxx* は、その関数に対応するアセンブリ シンボルです(例: 絶対アドレス関数 rv( )は_rv_text という名前の psect 内に配置される)。

__section( )指定子を使って関数をユーザ定義 psect に割り当て(5.14.3 「割り当てられたセクションの変更とリンク」参照)、その psect を必要な位置でリンクする事ができます。この方法を使うと、関数を最も柔軟に固定アドレスに配置できます (別のセクションの直後またはアドレスレンジ内の任意位置に配置可能)。割り込み関数の場合、この指定子は割り込み関数ボディの位置にのみ影響を及ぼします。実行可能ではないオブジェクト(const オブジェクト等)を保持するために使われるセクション内に関数を配置してはいけません。そうした場合、実行時エラーが発生するか、関数のデバッグに支障をきたす可能性があります。

どのような方法で関数を配置するにせよ、アドレスの選択には注意を払う必要があります。可能であれば、関数をプログラムメモリ ページの上端または下端に配置する事で、メモリの断片化を防ぐと共に、リンクエラーの可能性を低減できます。

最初のページ (リセットおよび割り込みコードを格納したページ) 内に関数を配置する事で、psect マージ時の最適化処理を助ける事ができます。

5.7.4 関数のサイズ制限

全てのデバイスにおいて、通常の関数向けに生成されるコードのサイズは利用可能なプログラムメモリのサイズによってのみ制限されます。しかし、1 つの関数がページ境界にまたがって配置された場合や、複数ページにわたる場合、関数内で発生するジャンプがページ間ジャンプのシーケンスとなり効率が低下します。詳細は 5.7.2 「実行可能コードのメモリ割り当て」を参照してください。

割り込み関数のサイズは 1 ページ以下に制限され、複数ページに分割する事はできません(5.8.1 「割り込みサービスルーチンの書き方」参照)。

5.7.5 関数のパラメータ

MPLAB XC8 は、決められた方法で引数を関数に渡します。この方法は引数のサイズと数に応じて異なる他、その関数で使われるスタックモデルによっても異なります。

Note: しばしば「引数」と「パラメータ」は同じ意味で使われますが、一般的に「引数」は関数に渡される値を指し、「パラメータ」は引数を保持するために関数によって定義される変数を指します。

5.7.5.1 コンパイルド スタックのパラメータ

コンパイルド スタックを使う再入不可関数の場合、コンパイラは W レジスタ内または呼び出される関数のパラメータメモリ内で引数を渡します。

最初のパラメータのサイズが 1 バイトである場合、それは W レジスタ内で渡されます。その他の全てのパラメータは、パラメータメモリ内で渡されます。パラメータメモリはコンパイルド スタック内に配置されます(5.2.4.2.1 「コンパイルド スタックの動作」参照)。

パラメータは、シンボル?_*function* からのオフセットとして参照されます (*function* は、その内部でパラメータを定義している関数(すなわち呼び出し先関数) の名前)。

auto 変数とは異なり、パラメータ変数はそれらが関数のプロトタイプ内に現れる順番通りにメモリに割り当てられます。これは、関数のパラメータは常に同一 RAM バンク内に配置される事を意味します。これに対し、関数向けの auto 変数は複数のバンクにまたがって任意の順番で配置できます。

可変引数リスト(プロトタイプ内で ellipsis(省略記号)を使って定義)を持つ関数の無名パラメータの引数も、名前付きパラメータと一緒にパラメータ用メモリ内に配置されます。

以下に関数プロトタイプの例を示します。

```
void test(char a, int b);
```

関数 test( )はパラメータメモリ内でパラメータ b を受け取り(?_test と?_test+1 の 2 バイトを使用)、パラメータ a を W レジスタ内で受け取ります。

コンパイルド スタックを使う複数の関数が同じ関数ポインタを介して間接的に呼び出される可能性がある場合、コンパイラは特別な対応を取る必要があります。そのような例を以下に示します。この例では sp_add、sp_sub、sp_null がポインタ fp を介して呼び出される可能性があります。

```
int (*funcs[])(int, int) = {sp_add, sp_sub, sp_null};
int (*fp)(int, int);
fp = funcs[getOperation()];
result = fp(37, input);
```

このような場合、コンパイラは 3 つの関数を「仲間」として配列内で扱います。

全ての「仲間」関数へのパラメータはメモリ上で整列されます(同じメモリ領域が利用されます)。この約束事によってどの関数が呼び出されてもコンパイラは同様のパラメータ処理を行え、関数依存性が無くなります。ただし、同じメモリ領域が使われるため「仲間」関数から「仲間」関数を呼び出す事はできません。呼び出した場合、呼び出し元関数のパラメータが上書きされてしまいます。そのような呼び出しが試みられた場合はエラーが出力されます。

5.7.5.2 ソフトウェア スタックのパラメータ

ソフトウェア スタックを使う関数に渡される大部分の引数は、スタック上で渡されます(5.2.4.2.2 「ソフトウェア スタックの動作」参照)。

引数は呼び出し元関数によってスタックへプッシュされます。プッシュされる順番は、関数のプロトタイプ内で対応するパラメータが現れる順番とは逆です。その後、呼び出し先関数は必要に応じて auto 変数または一時変数向けの記憶域を割り当てるため、スタックポインタの値を増加させて領域確保を行います。

拡張ミッドレンジ デバイスの場合、最初の引数がバイトサイズであれば、W レジスタを引数用に使います。ただし関数の引数の個数が可変の場合を除きます。再入可能関数が `external` である場合(5.7.2 「実行可能コードのメモリ割り当て」参照)、W レジスタは引数の保持用に決して使われません。PIC18 デバイス向けにコンパイルする場合、W レジスタは再入可能関数の引数用に決して使われません。

可変引数リスト(プロトタイプ内で ellipsis を使って定義)を取る関数内の無名パラメータ向け引数は、ソフトウェア スタック上で名前付きパラメータ向け引数の前に配置されます。関数の全ての引数がプッシュされた後に、無名パラメータの総サイズがスタックへプッシュされます。関数あたり最大で 256 バイトの非プロトタイプ パラメータが許容されます。

フレームポインタは存在しないため、手書きアセンブリコード内で関数パラメータ(または他のスタックベース オブジェクト) にアクセスする事は推奨しません。

5.7.6 関数の戻り値

関数からの戻り値はレジスタに書き込まれるか、その関数によって使われるスタック上に配置されます。そのメカニズムは、使われる関数モデルに応じて異なります。

5.7.6.1 コンパイルド スタックを使う関数の戻り値

コンパイルド スタックを使う再入不可関数の場合、戻り値は W レジスタまたはその関数のパラメータメモリを使って呼び出し元関数に渡されます。関数のパラメータに割り当てられたメモリ(関数がリターンする時点で不要となる)は、関数のコードとデータ要求を削減するために再使用されます。

シングルバイトの戻り値は W レジスタ内で関数から戻されます。1 バイトよりも大きな値は、関数のパラメータメモリ領域内で戻されます。

例:

```
int returnWord(void)
{
    return 0x1234;
}
```

上記の関数は、`?_returnWord` 内で値 0x34 を返し、`?_returnWord+1` 内で値 0x12 を返します。

PIC18 では、戻り値のサイズが 4 バイトより大きくて 8 バイトより小さい場合、パラメータ領域のアドレスも FSR0 レジスタ内に配置されます。

`__bit` 型の値を返す関数は、STATUS レジスタ内のキャリービットを使って値を返します。

5.7.6.2 ソフトウェア スタックを使う関数の戻り値

ソフトウェア スタックを使う関数は、`btemp` 変数を介して呼び出し元関数に戻り値(サイズは 4 バイト以下)を返します。拡張ミッドレンジ デバイスの場合、可変数の引数を持たない関数は、W レジスタを使ってバイトサイズの値を返します。サイズが 4 バイトより大きなオブジェクトは、スタック上で返されます。PIC18 デバイスの場合、`__bit` 型の値を返す再入可能関数は、`btemp0` 内の bit 0 を使って値を返します(その他のデバイスは STATUS レジスタ内のキャリービットを使用)。

フレームポインタは存在しないため、手書きアセンブリコード内で戻り値の位置または他のスタックベース オブジェクトにアクセスする事は推奨しません。

5.7.7 関数の呼び出し

全ての 8 ビット PIC デバイスは、関数の戻りアドレスを保存するためにハードウェア スタックを使います。このスタックの最大深さはデバイスごとに異なります。

一般的に、C 関数が呼び出された時に、制御をその関数に渡すために、呼び出しアセンブリ命令が使われます。1 つの呼び出しはハードウェア スタックの 1 レベルを使います。これは、呼び出し先コードが `return` または `retlw` 命令を実行した後に解放されます。ネストされた関数呼び出しによりプログラムのスタック使用量は増加します。ハードウェア スタックがオーバーフローすると、関数の戻りアドレスが上書きされ、コードは最終的に実行に失敗します。

PIC18 デバイスの場合、`call` 命令が関数呼び出しの唯一の方法になります。他の 8 ビット PIC デバイスの場合、`-mstackcall` オプション(4.6.1.23 「stackcall オプション」参照)を使い、コンパイラによりハードウェアスタックがオーバーフローする可能性のある、ネストの多い呼び出しを検出した場合の対応を制御できます。

このオプションが有効である場合、コンパイラは警告を出力せずに、マネージド スタックを使う方法(ハードウェア スタックを必要としないルックアップ テーブル方式) に自動的に切り換えます。

このルックアップ方式は、関数を呼び出すのではなく、関数のアドレスへ直接ジャンプします。これが実行される前に、特殊な「return」命令(ジャンプ命令として実装される) のアドレスが呼び出し先関数のワークに一時的に保存されます。この return 命令により、制御を呼び出し元関数へ戻す事ができます。

この関数呼び出し方法により、ベースラインおよびミッドレンジ デバイスの限られたスタックでオーバーフローを生じる事なく関数を深くネストする事ができます。しかし、この方法によりメモリ使用量は増加し、プログラムの実行速度は低下します。

5.7.7.1 間接呼び出し

ポインタを使って関数が間接的に呼び出される場合、コンパイラは各種の方法を使って関数を呼び出します。

全ての PIC18 および拡張ミッドレンジ デバイスは、関数ポインタ内の値を使って、プログラム カウンタに適切なアドレスを書き込みます。PIC18 デバイスの場合、TOS レジスタに関数ポインタアドレスをロードし、呼び出し実行に際しては `Return` 命令を実行します。拡張ミッドレンジ デバイスの場合、`callw` 命令が使われます。間接呼び出しが可能な関数の数は、デバイスの利用可能メモリ量によってのみ制限されます。

全てのベースラインおよびミッドレンジ デバイスは、ジャンプ命令でロードされるルックアップ テーブルを使います。ルックアップ テーブルコードが呼び出され、テーブル内で適切なジャンプを実行するためにオフセットが使われます。間接的に呼び出される関数が増えるにつれてテーブルのサイズは増加しますが、0xFF バイトを超える事はできません。これにより、間接的に呼び出しが可能な関数の数が制限されます(通常、関数の数は約 120 以下に制限されます)。この制限は、プログラムが定義可能な関数ポインタの数に影響を及ぼさない事に注意が必要です。この数は、デバイス上で利用可能なメモリ量により制限されます。

間接呼び出しは`-mstackcall` オプションの影響を受けません。また、ベースラインおよびミッドレンジ デバイスにおける間接呼び出しの深さは、ハードウェア スタックの深さにより制限されます。

5.8 割り込み

MPLAB XC8 コンパイラは、C コードから割り込みを完全に処理するための機能を備えています。多くの場合、割り込み関数は割り込みサービス ルーチン(ISR)と呼ばれます。

割り込みの処理方法は、デバイスファミリごとに異なります。大部分のベースライン デバイスは割り込み機能を一切実装していません。ミッドレンジ デバイスは、全ての割り込み要因にリンクした 1 つのベクタ位置を備えています。一部の PIC18 デバイスは 2 つの独立した割り込みベクタを備えており、1 つは低優先度割り込み要因に割り当てられ、もう 1 つは高優先度割り込み要因に割り当てられています。一部の PIC18 デバイスは 1 つのベクタ割り込みコントローラ(VIC)モジュールを備えると共に、高優先度または低優先度割り込み関数のアドレスを格納可能な 1 つまたは複数の割り込みベクタテーブル(IVT)をサポートします。

VIC モジュールを備えたデバイス上の IVT の動作は、`MVECEN` コンフィグレーション ビットをクリアする事により無効にできます。これを無効にした場合、デバイスは 2 つの優先度と 2 つのベクタ位置で動作するレガシーモードとなります。このビットは、割り込み関数のプログラミング方法を指定するためにコンパイラによっても使われます。レガシーモードの場合、このビットは `OFF` に設定されます。レガシーモードではベクタテーブルは無効にされますが、ベクタ位置は再配置可能なままです。既定値では、ベクタ位置は 0x8 と 0x18 です(VIC モジュールを備えない PIC18 デバイスと同じ位置)。

PIC18 デバイスが実装する優先度スキームも、`IPEN` SFR ビットをクリアする事により無効にできます。無効にした場合、デバイスはミッドレンジ互換モードで動作し、アドレス 0x8 に配置された 1 つの割り込みベクタのみを使います。

割り込みを使う場合に従う必要がある一般的な手順は以下の通りです。これらの手順の詳細は、その後の各項目で説明します。

割り込み機能を備えた拡張ベースライン デバイス、ミッドレンジ デバイス、ミッドレンジ互換モードで動作する PIC18 デバイスの場合:

- 全ての割り込み要因を処理する 1 つの割り込み関数を書きます。
- メインライン コード内の適切な位置で、必要な割り込み要因のマスクを解除します(対応する SFR 内でその割り込み要因のイネーブルビットをセットします)。
- コード内の適切な位置で、グローバル割り込みイネーブルを有効にする事で、割り込みの生成を許可します。

VIC モジュールを備えない PIC18 デバイス、レガシーモードで動作する PIC18 デバイスの場合:

- 各割り込み要因に割り当てる優先度を決めます。デバイスがレガシーモードで動作する場合、必要なデュアル割り込みベクタのセット数を決めます。
- 必要に応じて MVECEN コンフィグレーション ビットを設定します。VIC モジュールを備えたデバイスでこのビットが OFF に設定されていない場合、コンパイラはベクタテーブルをエンコードします。
- 使用する各優先度を処理する割り込み関数を、優先度ごとに 1 つずつ書きます。最大で 2 つの割り込み関数を定義できます。レガシーモードで動作するデバイスの場合、ベクタセットあたり 2 つの割り込み関数を定義できます。未使用割り込みが偶発的にトリガされた場合に対応するため、両方の割り込み関数を実装するか、-mundefints オプションを使って既定値アクションを提供する事を推奨します(4.6.1.25 「undefints オプション」参照)。
- 優先度を各割り込み要因に割り当てるため、SFR 内の適切なビットに書き込むコードを書きます。
- デバイスがレガシーモード動作する場合、必要に応じてコード内の適切な位置で IVTBASE レジスタに書き込む事により、デュアルベクタのセットを選択します。割り込みが有効にされている時に IVTBASE レジスタに書き込んではいけません。-mivt オプション(4.6.1.12 「ivt オプション」参照)を使って初期ベクタを選択する事もできます。
- コード内の適切な位置で、必要な割り込み要因を有効にします。
- コード内の適切な位置で、グローバル割り込みイネーブルを有効にします。

VIC モジュールを使うデバイスの場合:

- 各割り込み要因に割り当てる優先度と、必要な割り込みベクタテーブルの数を決めます。
- 割り込み関数を必要な数だけ書きます。割り込みの応答を高速にするため、各割り込み要因向けに専用の関数を書きます(必要ならば 1 つの関数で複数の割り込み要因を処理する事も可能)。未使用割り込み要因が偶発的にトリガされた場合に対応するため、1 つまたは複数の関数を追加するか、-mundefints オプションを使って既定値アクションを提供する事を推奨します(4.6.1.25 「undefints オプション」参照)。
- 必要な優先度を各割り込み要因に割り当てるため、SFR 内の適切なビットに書き込むコードを書きます。
- 複数の割り込みベクタテーブルを使う場合、コード内の適切な位置で IVTBASE レジスタに書き込む事により IVT を選択します。割り込みが有効にされている時に IVTBASE レジスタに書き込んではいけません。-mivt オプション(4.6.1.12 「ivt オプション」参照)を使って初期 IVT を選択する事もできます。
- コード内の適切な位置で、必要な割り込み要因を有効にします。
- コード内の適切な位置で、グローバル割り込みイネーブルを有効にします。

割り込み関数を C コードから直接呼び出してはいけません (使われる RETURN 命令が異なるため)。しかし、割り込み関数から他の関数(ユーザ定義関数とライブラリ関数の両方)を呼び出す事はできます。

「割り込みコード」とは、割り込みが発生した結果として実行される全てのコードを指します。これには、ISR から呼び出される関数とライブラリコードも含まれます。割り込みコードは、割り込まれた命令が実行された時点で完了します。これはメインライン コードと大きく異なります。メインライン コードとは、独立したアプリケーションにおいて通常リセット後に実行される主幹プログラムです。

5.8.1 割り込みサービスルーチンの書き方

ISR のプロトタイプと内容は、ターゲットデバイスとコンパイルするプロジェクトに応じて異なります。ISR を書く際は、以下のガイドラインに従う必要があります。

VIC モジュールを備えないデバイスの場合:

- __interrupt()指定子を使って各 ISR プロトタイプを書きます。
- 戻り値およびパラメータの型として void を使います。
- ターゲット デバイスが割り込み優先度をサポートする場合、優先度指定を引数として表現します。各関数で __interrupt()に対して low_priority (または__low_priority) 引数または high_priority (または __high_priority) 引数を使います。
- ISR のボディ内で、処理される各割り込み要因の割り込みフラグと割り込みイネーブルを確認する事により割り込み要因を特定し、対応する条件割り込み処理コードを記述します。

レガシーモードで動作するデバイスの場合:

- __interrupt()指定子を使って各 ISR プロトタイプを書きます。
- void を戻り型として使い、割り込み要因を識別する必要がある場合は void または char 型引数のパラメータリストを指定します。VIC モジュールを備えないデバイスとの間の移植性を確保する必要がある場合、パラメータリストを void に設定する事を推奨します。

- ISR プロトタイプ内の__interrupt()指定子への引数として、割り込み関数の要因に対して割り当てられた割り込み優先度を、low_priority (または __low_priority) または high_priority (または __high_priority)を使って指定します。必要に応じて、関数のアドレスを格納する IVT のベースアドレスを base() (または__base())を使って指定します。VIC モジュールを備えないデバイスとの間の移植性を確保する必要がある場合、このベースアドレスは既定値のままにしておく事を推奨します。
- ISR が複数の要因を処理する場合、関数のパラメータ(指定されている場合) から割り込みの要因を特定するか、処理される各要因の割り込みフラグと割り込みイネーブルを確認する事により割り込みの要因を特定します。

VIC モジュールを使うデバイスの場合:

- __interrupt()指定子を使って各 ISR プロトタイプを書きます。
- void を戻り型として使い、割り込み要因を識別する必要がある場合は void または char 型引数のパラメータリストを指定します。
- ISR プロトタイプ内の__interrupt()指定子に対する引数として、各割り込み関数が処理する要因を、irq() (または__irq())を使って指定します。また、関数の要因に割り当てる優先度を、low_priority (または __low_priority) または high_priority (または high_priority)を使って指定します。必要に応じ、関数のアドレスを格納する IVT のベースアドレスを、base() (または__base())を使って指定します。
- ISR が複数の要因を処理する場合、関数のパラメータ(指定されている場合) から割り込みの要因を特定するか、処理される各要因の割り込みフラグと割り込みイネーブルを確認する事により割り込みの要因を特定します。

全てのデバイス:

- 割り込み要因が処理された後に、ISR ボディの内部で、対応する割り込みフラグをクリアします。
- ISR ボディの内部で割り込みを再度有効にしてはいけません。これは ISR から戻る際に自動的に実行されます。
- ISR はできるだけ短くシンプルにします。複雑なコードは一般的に多くのレジスタを使いがちであるため、コンテクスト スイッチコードのサイズが増加します。

割り込み優先度が使われる場合、優先度を指定していない ISR は既定値により高優先度となります。コードを読みやすくするため、ISR の優先度を常に指定する事を推奨します。

VIC モジュールを備えないデバイスで__interrupt()指定子に対して引数 irq()または base()を渡すと、コンパイラはエラーを出力します。レガシーモードに設定されたデバイスでこの指定子を使う場合、引数 irq()を指定するとコンパイラはエラーを出力します。引数 base()は、必要であれば使っても構いません。

VIC モジュールを備えたデバイスは、各割り込みを番号で識別します。この番号は、ベクタテーブルが有効な場合に__interrupt()への引数 irq()で指定できます。あるいは、その番号に対応するコンパイラ定義シンボルを使う事ができます。ウェブブラウザで pic_chipinfo.html または pic18_chipinfo.html ファイルを開いてターゲット デバイスを選択すると、全ての割り込み番号、シンボル、説明を記載した一覧が表示されます。これらのファイルは、本コンパイラのインストール ディレクトリの下の docs ディレクトリに保存されています。

割り込み関数は常に再入不可関数モデルを使います。これらの関数は、再入を指定する全てのオプションまたは関数指定子を無視します。

本コンパイラは割り込み関数を他の関数とは異なる方法で処理し、関数と専用のリターン命令によって使われる全てのレジスタを退避/復元するコードを生成します。

IVT を使わないコード向けに書かれた割り込み関数の例を以下に示します。この割り込み関数は、割り込みイネーブルビット(例: TMR0IE)と割り込みフラグビット(例: TMR0IF)を監視する事により、割り込みの要因を確認します。周辺モジュールに関連する割り込みフラグは、その周辺モジュールが割り込みを生成するように設定されていなくてもアサートされる場合があるため、割り込みフラグだけでなく割り込みイネーブルビットも確認する必要があります。

```
int tick_count;
void __interrupt(high_priority) tcInt(void)
{
    if (TMR0IE && TMR0IF) { // any timer 0 interrupts?
        TMR0IF=0;
        ++tick_count;
    }
    if (TMR1IE && TMR1IF) { // any timer 1 interrupts?
        TMR1IF=0;
        tick_count += 100;
    }
    // process other interrupt sources here, if required
    return;
}
```

以下も同じ関数のコードですが、ベクタテーブルを使うデバイス向けに分割および変更されています。各 ISR に割り当てられている割り込み要因は 1 つだけであるため、割り込みコードは要因を判別する必要がなく、複数の要因が割り当てられた場合より高速です。

```
void __interrupt(irq(TMR0),high_priority) tc0Int(void)
{
    TMR0IF=0;
    ++tick_count;
    return;
}
void __interrupt(irq(TMR1),high_priority) tc1Int(void)
{
    TMR1IF=0;
    tick_count += 100;
    return;
}
```

1 つの関数で複数の割り込み要因を処理する場合、引数 irq()で複数の割り込み要因を指定し、関数パラメータを使って要因番号を保持します。

例:

```
void __interrupt(irq(TMR0,TMR1),high_priority) tInt(unsigned char src)
{
    switch(src)
    { case
    IRQ_TMR0:
        TMR0IF=0;
        ++tick_count;
        break;
    case
        IRQ_TMR1:T
        MR1IF=0;
        tick_count += 100;
        break;
    }
    return;
}
```

この例では、割り込みが発生した時に VIC モジュールがパラメータ src に割り込み要因番号をロードします。

特別な割り込み要因シンボル default を使うと、その ISR が任意の(すなわち irq()を使って明示的に指定されたのではない) 割り込みベクタへリンクされる事を示す事ができます。また、-mundefints オプション(4.6.1.25 「undefints オプション」参照)により、未使用ベクタ位置を配置する事も可能です。

既定値では、割り込みベクタテーブルは IVTBASE レジスタのリセット値に等しいアドレス(レガシーアドレス: 0x8)に配置されます。__interrupt()に対する引数 base()を使って、その関数向けに異なるテーブル ベースアドレスを指定できます。この引数には、コンマで区切った複数のアドレスを指定できます。ベースアドレスを IVTBASE レジスタのリセット値より低いアドレスに設定する事はできません。

既定値により(または必要に応じて)、コンパイラは実行時起動コード内で IVTBASE レジスタを初期化します。-mivt オプション(4.6.1.12 「ivt オプション」参照)を OFF にする事により、この機能を無効にできます。このオプションにより、初期ベクタテーブル向けにこのレジスタの初期アドレスを指定する事もできます。ベクタ割り込みが有効である場合、このオプションを使ってアドレスを指定しないと、ベクタテーブル位置は、プログラム内で使われる最低のテーブルアドレス(__interrupt()に対する引数 base()により指定) に設定されます。

-

引数 base()を使って複数の割り込みベクタテーブルを実装する場合、各テーブルに十分なメモリを割り当てる必要があります。コンパイラが割り込みベクタのオーバーラップを検出すると、エラーメッセージが出力されます。

timer 0 および timer 1 割り込み要因を処理する 2 つの ISR の割り込み関数プロトタイプの例を以下に示します。これらは独立した 2 つのベクタテーブル (ベースアドレスは 0x100 と 0x200) に配置されるよう設定されます。その他の全ての割り込み要因は、両方のベクタテーブル内の低優先度 ISR `defIsr()`により処理されます。

これらの ISR がアクティブになるには、最初に IVTBASE レジスタに 0x100 または 0x200 がロードされる必要があります。このレジスタ内のアドレスを変更する事により、アクティブになるベクタテーブルを選択できます。

```
void __interrupt(irq(TMR0,TMR1),base(0x100)) timerIsr(void)
{...}
void __interrupt(irq(TMR0,TMR1),base(0x200)) altTimerIsr(void)
{...}
void __interrupt(irq(default),base(0x100,0x200),low_priority) defIsr(void)
{...}
```

レガシーモードで動作するデバイス向けのベクタテーブルは 2 つのベクタ(高優先度および低優先度割り込みベクタ)しか備えていません。テーブルのベースアドレスからこれらのベクタへのオフセットは、それぞれ 0x8 および 0x18 です。引数 `base()`は、ベクタテーブルのベースアドレスを指定するのであって、テーブル内のベクタのアドレスを指定するのではありません。以下のコードでは、レガシーモードのデバイスの両方の優先度の ISR を、テーブルのベースアドレスを 0x2000 へ移動させて定義しています。これらの ISR はアドレス 0x2008 および 0x2018 に配置されます。

```
void __interrupt(base(0x2000), high_priority) highIsr(void)
{...}

void __interrupt(base(0x2000), low_priority) lowIsr(void)
{...}
```

レガシーモードで動作するデバイスでも、複数の割り込みベクタテーブルを定義できます。各テーブルに対して異なるベースアドレスを使い、高優先度および低優先度割り込み向けに最大で 2 つの割り込み関数を定義できます。

5.8.2 既定値の割り込み関数メモリ割り当てを変更する

通常の関数に比べると、割り込み関数に関連するコードを移動させるのは困難です。なぜなら、割り込みルーチンはデバイスによって厳密に定義されたエントリポイントを持つからです。

割り込み関数を移動させるには`__section()`指定子(5.14.3 「割り当てられたセクションの変更とリンク」参照)が使えますが、割り込みエントリポイントは既定値ベクタ位置のままです。

ベクタ位置を移動させる方法は、5.8.3 「割り込みベクタの指定」を参照してください。

5.8.3 割り込みベクタの指定

VIC モジュールを備えないデバイスの場合、割り込みベクタは完全に自動的に配置されます。コンパイラは、割り込みコード エントリポイントを固定されたベクタ位置へリンクします。通常、エントリポイント コードは割り込みコンテクスト スイッチを実行するコードの全部または一部であり、割り込み関数のボディはエントリポイントとは異なる位置に配置されます。

実行時にこれらの割り込みベクタの位置を変更する事はできず、ベクタへリンクされたコードを変更する事もできません。プログラムの実行中に代替の割り込み関数を選択して使う事はできません。プログラム内に割り込みベクタより多くの割り込み関数が存在する場合、エラーとなります。

VIC モジュールを備えたデバイスの場合、実行時により柔軟に割り込み関数を実行できます。IVT が有効である場合、これらのデバイスは 1 つの割り込みベクタテーブルを使います。各テーブルエントリはアドレスを保持し、そのエントリに対応する割り込みがトリガされた時に読み出され、デバイスはそのアドレスへジャンプします。各割り込み関数に対応するベクタテーブル エントリは、`interrupt()`に対する引数 `irq()` (または`__irq()`) および `base()` (または`__base()`) 内の情報に基づいて、コンパイラが自動的に作成します(5.8.1 「割り込みサービスルーチンの書き方」参照)。

ベクタテーブル内のアドレスを実行時に変更する事はできませんが、複数のテーブルを作成しておき、デバイスにテーブルを切り換えさせる事ができます。IVTBASE レジスタに保存されているベクタテーブルのベースアドレスを変更する事により、アクティブなベクタテーブルを変更できます。これらのレジスタはアトミックに変更できないため、変更する前に全ての割り込みを無効にしておく必要があります。下の例に、割り込みを無効にしてから IVTBASE レジスタの値を変更し、最後に割り込みを再び有効にする方法を示します。

```
di();    // disable all interrupts
IVTBASEU = 0x0;
IVTBASEH = 0x2;
IVTBASEL = 0x0;
ei();    // re-enable interrupts
```

`-mivt` オプション(4.6.1.12 「ivt オプション」参照)を使ってこのレジスタをプリロードする事もできます。

VIC モジュールを備えた PIC18 デバイスがレガシーモードで動作する場合、ベクタテーブルは無効になり、標準型 PIC18 デバイスが備えるデュアル優先度ベクタが使われます。これらのベクタ位置はアドレスではなく命令を保持しますが、標準型 PIC18 デバイスとは異なり、プログラムは IVTBASE レジスタを使ってベクタ位置を任意のアドレスに割り当てる事ができます。また、各ベースアドレスに対して 2 つの割り込み関数を定義する事もできます。

IVTBASE レジスタの機能と `-mcodeoffset` オプション(4.6.1.3 「codeoffset オプション」参照)を混同しないよう注意が必要です。このオプションは、リセットに関連するコードを移動させますが、リセット時にデバイスがジャンプする先のアドレスを変更するのではありません。VIC を備えないデバイスでは、このオプションは割り込みに関連するコードも移動させますが、やはり割り込みベクタアドレスは変更しません。VIC を備えたデバイスの場合、レガシーモードで動作する場合であっても、割り込みに関連するコードはこのオプションの影響を受けません。その代わりに、それらのデバイスでは IVTBASE レジスタが、ベクタテーブルが使用すると想定されるアドレス (レガシーモードで動作する場合は割り込みベクタのアドレス)を制御します。

VIC を備えたデバイス向けにブートローダによってロードされるアプリケーションを書く場合、通常は `-mcodeoffset` オプションと IVTBASE レジスタの両方を使って、リセットおよび割り込みエンドポイントをシフトさせる必要があります。このオプションはリセット位置をシフトさせ、割り込み関数が適切な`__base()`アドレス引数を使っていればレジスタの変更はシフトされたベクタテーブルが正しくアクセスされる事を保証します。この要件は、レガシーモードで動作するデバイスを対象とします。

プロジェクト内で明示的に指定されていない割り込みベクタには、既定値の関数アドレスを割り当てる事ができます。そうするには、既定値をその `irq()`割り込み要因として使う割り込み関数を定義するか、 `-mundefints` オプション(4.6.1.25 「undefints オプション」参照)を使って既定値命令を割り当てます。

5.8.4 コンテクスト スイッチ

コンパイラは、割り込みが発生した時に現在のステータスを保存し割り込みから戻った時にステータスを復元するコードを、自動的にプロジェクトへリンクします。

5.8.4.1 割り込み時のコンテクスト保存

一部のレジスタは、割り込み発生時にハードウェアによって自動的に保存されます。その他の割り込み関数によって使われるレジスタやコンパイラの一時オブジェクト等、ハードウェアによって保存されない情報はコンパイラによって生成されたコード内で保存されます。このコードをコンテクスト保存またはコンテクスト スイッチコードと呼びます。

割り込み発生時にハードウェアまたはソフトウェアによって保存される必要があるレジスタについては、5.6 「レジスタの使用」を参照してください。

拡張ミッドレンジ PIC デバイスは WREG、STATUS、BSR、FSRx レジスタをハードウェアで保存します(特殊なシャドーレジスタを使用)。このため、これらのレジスタをソフトウェアで保存する必要はありません。ソフトウェアによって保存する事が必要になる可能性があるレジスタは、BTEMP レジスタ(3)です。これはレジスタのように動作するコンパイラの一時保存位置です。

その他のミッドレンジ PIC プロセッサは、割り込み発生時に PC 全体(PCLATH レジスタを除く) のみ保存します。WREG、STATUS、FSR、PCLATH レジスタと全ての BTEMP レジスタは、コンパイラによって生成されたコードによって保存される必要があります。

既定値では、PIC18 の高優先度割り込み関数は内部シャドーレジスタを使って W、STATUS、BSR レジスタを保存します。PIC18 の低優先度割り込みの場合、またはシャドーレジスタが使えない場合、割り込みコードによって使われている全てのレジスタはソフトウェアによって保存されます。

PIC18 デバイスが VIC モジュールを備えている場合、追加で FSRx、PCLATHx、PRODx レジスタがシャドーレジスタに保存されます。使われているその他のレジスタは全てソフトウェアによって保存されます。低優先度割り込みと高優先度割り込み向けに別々のシャドーレジスタが利用できます。

一部の古いデバイスで MPLAB ICD デバッガ向けにコンパイルする場合、デバッガがこれらのシャドーレジスタを使うため、コンパイラはシャドーレジスタを使いません。一部のエラッタ回避策も、シャドーレジスタの使用を回避します(4.6.1.10 「errata オプション」参照)。

コンパイラは、割り込み関数またはその割り込み関数が呼び出す全ての関数がどのレジスタとオブジェクトを使うのか正確に特定し、それらのレジスタとオブジェクトを適切に保存します。

割り込み関数内のインライン アセンブリコードによるレジスタの使用はコンパイラによって検出されません。割り込み関数内(または割り込み関数から呼び出される関数内) にインライン アセンブリコードが含まれる場合、使われているレジスタを保存および復元するためのアセンブリコードを追加する事が必要になる場合があります。

[3] これらのレジスタは、コンパイラによって割り当てられるメモリ位置ですが、コードを生成するためにレジスタであるかのように扱われます。通常これらは、再入可能コードを生成する際に使われます。

ソフトウェアスタックが使われている場合、コンテクスト スイッチコードはスタックポインタレジスタも初期化するため、割り込み用に予約されているスタックの領域にもアクセスします。ソフトウェア スタックの詳細は5.4.2.2.1 「オブジェクト サイズの制限」を参照してください。

5.8.4.2 コンテクストの復元

ソフトウェアによって保存された全てのオブジェクトは、割り込み関数がリターンする前にソフトウェアによって自動的に復元されます。コンテクストは保存時とは逆順に復元されます。

ソフトウェアスタックが使われている場合、コンテクスト復元コードは、割り込みが発生する前に使われていたスタックの領域にアクセスできるように、スタックポインタレジスタも復元します。
ソフトウェア スタックの詳細は5.4.2.2.1 「オブジェクト サイズの制限」を参照してください。

5.8.5 割り込みの有効化

2 つのマクロが利用できます。<xc.h>をインクルードすると、それらのマクロは利用可能な全ての割り込みのマスクを制御します。2 つのマクロとは、全ての割り込みを有効に(またはアンマスク)する ei()と、全ての割り込みを無効に(またはマスク)する di()です。

全てのデバイスにおいて、これらのマクロは INTCON または INTCON0 レジスタ内の GIE ビットに影響を及ぼします。これらのマクロは、プログラム内で必要とされる割り込み向けの適切な割り込みイネーブルビットを有効にした後に使います。

例:

```
ADIE = 1;  // A/D interrupts will be used
PEIE = 1;  // all peripheral interrupts are enabled
ei();      // enable all interrupts
// ...
di();      // disable all interrupts
```

Note: 割り込み関数自体の中で割り込みを再有効化してはいけません。割り込みは、retfie 命令の実行時にハードウェアによって自動的に再有効化されます。割り込み関数自体の中で割り込みを再有効化すると、異常動作が生じます。

5.8.6 割り込みルーチンからオブジェクトへのアクセス

他の関数からアクセスされるオブジェクトを割り込みルーチンから読み書きするのは安全ではありません。

コンパイラは、割り込みルーチン内で参照される全ての変数を volatile として扱いますが、コードの移植性を高めるため、volatile 指定子を使ってこれらの変数の型を明示的に示す事を推奨します(5.3.8.2 「volatile 型修飾子」参照)。コンパイラは、volatile オブジェクトに対する最適化の実行を制限します(5.12 「最適化」参照)。

オブジェクトが明示的に volatile として指定されていても、コンパイラはそれらがアトミックにアクセスされる事を保証できません。これは特に、マルチバイト オブジェクトに対する演算に当てはまりますが、シングルバイト オブジェクトまたはビット オブジェクトであっても多くの演算は 1 命令で実行できません。

アクセスがアトミックである事を保証できない限り、メインライン コード内において割り込み関数によって使われるオブジェクトを変更する前に割り込みを無効にする必要があります。アトミックかどうかは、アセンブラ リスト ファイルで命令文に対して生成されたコードを見れば分かります。しかし、プログラムの開発が進むにつれ命令が変更される可能性があり、これは特に最適化が有効である場合に当てはまります。

5.8.7 関数の複製

MPLAB XC8 コンパイラは、複数のコールグラフから呼び出される再入不可関数向けに生成されたコードを複製する機能を備えています。

コールグラフはメインライン コード向けに 1 つと、割り込みが定義されていれば各割り込み関数向けに 1 つずつ存在します。コンパイラは、割り込みはいつでも発生する可能性があると想定します。コンパイルド スタックを使うようコード化された関数は再入不可であるため、そのような関数がメインライン コードと割り込みコードから呼び出されると、異常動作が発生します。コンパイラは、複数のコールグラフから呼び出される全ての再入不可関数向けに生成されたコードを複製します。これにより、それらの関数は再入可能であるかのように見えますが、再帰呼び出しはできません。

コンパイラは、再入可能モデルを使って関数をコンパイルする事もできますが、このモデルは一部のデバイスでしか利用できません。また、再入可能モデルを使うとコードの効率が低下する可能性があります。-mstack オプションまたは__nonreentrant 指定子を使うと、コンパイラが再入可能モデルを選択するのを防ぐ事ができます。各関数にどちらの関数モデルが使われたのかを調べる方法は、6.3.3 「関数情報」を参照してください。

関数が複製された場合、メインライン コードはオリジナルの関数向けに生成されたコードを呼び出し、各割り込みコールグラフ内のコードはそれぞれ専用に複製された関数コードを呼び出します。複製はコンパイラが生成するコード内でのみ行われ、C ソースコードそのものは複製されません。

複製されたコードとその関数によって定義されているオブジェクトには一意の識別子が使われます。複製されたコードの識別子には、オリジナル コードと同じ識別子に接頭辞 i1 が付きます。PIC18 の場合、複製された関数には接頭辞として i1 (低優先度割り込み用)と i2 (高優先度割り込み用)を使います。

説明のため、プログラム内で main()関数と割り込みコード内の関数はどちらも input()という名前の関数を呼び出す場合を想定します。C 関数 input()向けに生成されたアセンブリ コードは、アセンブリ ラベル_input を使います。複製された関数のコードは、ラベルとして i1_input を使います。input()が一時変数を参照する場合、生成されたコードはシンボル??_input を使い、複製されたコードは??i1_input を使います。生成された関数コード内のローカルラベルも、同様に複製されます。アセンブリ リストファイル内のコールグラフには、これらの関数が別々に書かれた関数であるかのうに呼び出される事が示されます。これらのシンボルは、マップファイルのシンボルテーブルにも示されます。

ライブラリ関数に関連するコードも同様に複製されます。これには、暗黙的に呼び出されるルーチン (C 演算子に関連する除算または浮動小数点演算等)も含まれます。

5.8.7.1 複製の無効化

複数のコールグラフから呼び出される再入不可関数の自動的な複製は、特殊なプラグマを使って禁止できます。

メインライン コードから関数が呼び出されている時に割り込みが発生しない事が保証される場合にのみ、関数の複製を無効にしても構いません。通常、これを保証するには関数を呼び出す前に割り込みを無効にします。関数が呼び出された後に、その内部で割り込みを無効にするのでは不十分です。その関数を実行している時に割り込みが発生すると、コードの実行は失敗します。割り込みを無効にする詳細な方法は、5.8.5 「割り込みの有効化」を参照してください。

```
#pragma interrupt_level 1
```

このプラグマは、複製を禁止する関数の宣言の前に置く必要があります。このプラグマは、この後に最初に定義される関数にのみ影響を及ぼします。

例えば、メインライン コードと割り込み関数の両方から呼び出される関数 read()は、割り込みが無効にされている時にのみメインライン コードから呼び出されると想定します。この場合、以下のコードにより read()の複製を禁止する事ができます。

```
#pragma interrupt_level 1
int read(char device)
{
    // ...
}
```

メインライン コード内では、通常この関数は以下の通りに呼び出されます。

```
di();  // turn off interrupts
read(IN_CH1);
ei();  // re-enable interrupts
```

#pragma interrupt_level の後で指定される値は、どの割り込みに対してその関数を複製しないのかを示します。ミッドレンジ デバイスの場合、この値は常に 1 です。PIC18 デバイスの場合、この値は 1 (低優先度割り込み関数向け)または 2 (高優先度割り込み関数向け) として指定できます。両方の割り込み優先度で複製を無効にする場合、プラグマを 2 回使って 1 と 2 を指定する必要があります。以下の関数は、低優先度および高優先度のどちらの割り込み関数から呼び出されても複製されません。

```
#pragma interrupt_level 1
#pragma interrupt_level 2 int
timestwo(int a) {
        return a * 2;
}
```

5.9 メイン、スタートアップ、リセット

リセット後に、プログラムはコンパイラによって追加された実行時起動コードを最初に実行し、その後に制御が main()関数に渡されます。このシーケンスについて、以下で説明します。

5.9.1 main 関数

main は特別な識別子です。プログラム内に main()という名前の関数が 1 つだけ存在する事が必要です。これは、プログラム内で最初に実行される C 関数です。

main 関数はホストによって呼び出されるのではないため、コンパイラは main()の最後に特別なコードを挿入します。このコードは、この関数が終了した時(すなわち main()内でリターン命令文が実行された時)またはコード実行が main()の最後の右波カッコ「}」に達した時に実行されます。この特別なコードにより、実行はアドレス 0 (全ての 8 ビット PIC デバイスにおけるリセットベクタ位置) へジャンプします。これは本質的にソフトウェア リセットを実行します。ソフトウェア リセット後のレジスタの状態は、ハードウェア リセット後の状態とは異なる可能性があります。

main()関数を終了させない事を推奨します。このために、決して終了しないループ コンストラクタ(例: while(1))で main()内のコードを囲むか、コードの最後にこのループを追加します。

例:

```
int main(void)
{
    // your code goes here
    // finished that, now just wait for interrupts
    while(1)
        continue;
}
```

5.9.2 実行時起動コード

C プログラムが main()関数の実行を開始する前に、特定のオブジェクトを初期化してデバイスを特定の状態にしておく必要があります。そのために、実行時起動コードは以下のタスクを実行します(実行に特別な順序はありません)。

- 定義時に値が代入されるグローバル変数の初期化
- ユーザコードで初期化していないグローバル変数
- レジスタまたはデバイス ステートの一般設定

従来方式のコンパイルではプリコンパイル済みのコードをリンクする必要がありましたが、MPLAB XC8 C コンパイラはユーザ プログラムに必要な実行時起動コードを自動的に決定します。使用するファイルとプロセスの制御方法の詳細は、4.3.2 「起動と初期化」に記載しています。以下では、実行時起動コードが実行するタスクについて説明します。

実行時起動コードは、デバイスがリセットを終了した直後であり、レジスタはパワーオン リセット値を保持していると想定します。プログラムがブートローダによって呼び出されるアプリケーションである場合、ブートローダの実行後に、実行時起動コードが正しく実行されるよう RAM バンク 0 が選択されている事が必要です。これは、ブートローダの最後に適切なコードシーケンスをインライン アセンブリとして配置する事により達成できます。

表 5-12 に、起動時および電源投入時コードによって使われる重要なアセンブリラベルを示します。

表 5-12. 重要なアセンブリラベル

ラベル	位置
reset_vec	リセットベクタ位置(0x0)
powerup	電源投入時コードの先頭(このコードが使われる場合のみ)
start	実行時起動コードの先頭(startup.s 内)
start_initialization	C 初期化起動コードの先頭(C 出力コード内)

5.9.2.1 オブジェクトの初期化

実行時起動コードは、プログラムの実行開始前に全ての静的記憶期間オブジェクトに初期値を格納します。

例:

```
int input = 88;
```

上記のコードでは、初期値(0x88)がプログラムメモリ内にデータとして保存され、その値は実行時起動コードによって input 向けに予約されたメモリに複製されます。効率を高めるため、複数の初期値がデータのブロックとして保存され、ループによって複製されます。

絶対アドレス変数は決して初期化されず、明示的に値が代入される必要があります(その値が正常なプログラム実行に必要な場合)。

オブジェクトの初期化は-Wl,--no-data-init を使って無効にできますが、オブジェクトに格納された初期値を必要とするコードは実行に失敗します。

auto オブジェクトは動的に生成されるため、それらを初期化するためのコードは、それらを定義している関数内に置く必要があります。実行時起動コードは、これらのオブジェクトの事を考慮しません。

Note: 初期化される auto 変数は、特にオブジェクトのサイズが大きい場合にコードの性能に影響を及ぼすため、代わりに static オブジェクトが使えないか検討すべきです。

リセット中に内容を保持する必要があるオブジェクトには__persistent 修飾子を付ける必要があります。そのようなオブジェクトはメモリの別の領域でリンクされ、実行時起動コードよって変更されません。

オブジェクトを初期化するコードの実行によって、STATUS レジスタの初期状態は失われてしまいます。このレジスタからリセットの原因を特定する必要がある場合、STATUS レジスタの初期情報を別途保持する事もできます。

関連リンク

5.9.2.4 「ステータス レジスタの保持」

5.9.2.2 オブジェクトのクリア

値が代入されない静的記憶期間オブジェクトは、main()関数の実行前に実行時起動コードによってクリアされる必要があります。
例:

```
int output;
```

実行時起動コードは、非初期化オブジェクトによって使われる全てのメモリ位置をクリアします。従って、main()が実行される前のそれらの値は 0 です。

絶対アドレス変数は決してクリアされないため、明示的に値 0 を代入する必要があります(それが正常なプログラム実行に必要な場合)。

オブジェクトのクリアは、-Wl,--no-data-init を使って無効にできますが、オブジェクトに格納された初期値を必要とするコードは実行に失敗します。

リセット中に内容を保持する必要があるオブジェクトは__persistent で修飾する必要があります。そのようなオブジェクトはメモリの別の領域でリンクされ、実行時起動コードよって変更されません。

オブジェクトをクリアするコードの実行によって、STATUS レジスタの初期状態は失われてしまいます。このレジスタからリセットの原因を特定する必要がある場合、STATUS レジスタの初期情報を別途保持する事もできます。

関連リンク

5.3.9.5 「persistent 型修飾子」
5.9.2.4 「ステータス レジスタの保持」

5.9.2.3 デバイスステートの設定

一部の PIC デバイスのプログラムメモリにはオシレータ校正定数が書き込み済みです。これらのデバイスでは、この校正値をロードするためのコードが自動的に実行時起動コードに追加されます(5.2.11 「オシレータ校正定数」参照)。

プログラムがソフトウェア スタックを使う場合、スタックポインタ(FSR1)も実行時起動コードによって初期化されます(5.4.2.2.1 「オブジェクト サイズの制限」参照)。

5.9.2.4 ステータス レジスタの保持

-mresetbits オプション(4.6.1.19 「resetbits オプション」参照) は、STATUS レジスタ内の一部のビットを、それらが後続のプログラムによって変更されてしまう前に保存します。後で保存されたビットの状態を調べる事で、リセットの原因をいつでも特定できます。このオプションは、PIC18 デバイス向けには使えません。

STATUS レジスタは 8 ビット幅のアセンブリ変数___resetbits に保存されますが、このレジスタ内のバンク選択ビット(ターゲット デバイスに存在する場合) は正確に保存されない可能性があります。コンパイラはアセンブリ シンボル___powerdown と___timeout も定義します。これらのシンボルは、STATUS レジスタ内のパワーダウン(PD) およびタイムアウト(TO)ビットを表し、必要に応じてアセンブリコード内で使えます。これらの定義と、コンパイラが生成したレジスタ保存用コードは、実行時起動ファイル startup.s で確認できます。

<xc.h>をインクルードすれば、C コードから前述のシンボルにアクセスする事もできます。それらのシンボルと等価な C 識別子の先頭にはアンダースコア文字が 2 個だけ付きます(例: __resetbits)。シンボルの割り当ての詳細は 5.11.3.3 「等価アセンブリ シンボル」を参照してください。 以下のミッドレンジ デバイス向けサンプルコードは、保存されたビットの状態を main()の先頭部で確認します。リセットの正確な原因を特定するには、その他のレジスタ(PCON、PCON0 等) 内のビットの状態も確認する必要があるかもしれません。記録される全てのリセット要因はデバイス データシートを参照してください。

```
#include <xc.h>

int
main(void)
{
  // how did we get here?
  if(__timeout == 0 && __powerdown == 0)
    handleWDT_timeout();    // WDT wake-up from sleep
  // proceed with remaining code
}
```

コンパイラはコード内の前述のシンボルが使われていないか調べ、それらが存在すれば自動的に-mresetbits オプションを有効にします。必要に応じて、この機能を手動で有効にする事もできます。

5.9.3 電源投入時コード

ハードウェア コンフィグレーションによっては、リセット後の最初の数命令サイクル中に特殊な初期化が必要になる事がしばしばあります。これを行うため、電源投入時コードによりリセットベクタへのフックが提供されます。

このコードは、ユーザ定義アセンブラ モジュール内で提供できます。電源投入時コードのテンプレートとして powerup.S ファイルが使えます。このファイルは、コンパイラ インストール ディレクトリ内の pic/sources ディレクトリに保存されています。詳細は、このファイル内のコメントを参照してください。

このファイルを作業ディレクトリにコピーし、ソースファイルとしてプロジェクトにインクルードします。特別なリンカ オプションやその他のコードは不要です。コンパイラは、ユーザが電源投入時コードを定義している事を検出し、このコードが powerup という名前の psect (セクション) に格納されていれば、リセット後にこのコードを自動的に実行します。

このコードはラベル start へジャンプする goto 命令で終了する必要があります。全てのユーザ定義アセンブリコードと同様に、このコードでもプログラムメモリのページとデータメモリのバンクを考慮する事と、ターゲット デバイスに適用される全てのエラッタ回避策に配慮する事が必要です。

5.10 ライブラリ

MPLAB XC8 C コンパイラは、ユーザのコード開発を支援する関数、マクロ、型、オブジェクトのライブラリを提供します。

本コンパイラは C 言語標準ライブラリとデバイスに固有のライブラリ リソースを提供します。ライブラリには、浮動小数点演算や整数除算等のタスクを実行するためにプログラムから暗黙的に呼び出される C コードも含まれます。これらのコードはソース中に含まれる関数とは直接的な対応関係を持ちません。加えて、関数レベルのプロファイリング、ミッドレンジ デバイスが備える EEPROM へのアクセス、REAL-ICE トレース/ログ等に関連する各種のライブラリ関数が利用できます。これらは、必要に応じてプログラムと一緒にビルドされます。これらの関数は、どのライブラリ ファイルにも含まれていません。

コンパイラが提供するライブラリとは別に、ユーザが書いたソースコードからユーザ独自の関数を作成する事もできます。

5.10.1 標準ライブラリ

Microchip Unified Standard ライブラリは、C99 言語仕様によって提供される標準 C 言語ヘッダによって定義された関数と、それらを使うために必要な全ての型およびプリプロセッサ マクロを含みます。このライブラリは、全ての Microchip 社製 C コンパイラと一緒に提供されます。

ライブラリ内の関数の挙動、それらの関数へのインターフェイス、型およびマクロの用途は、『Microchip Unified Standard Library Reference Guide 』に記載されています。

5.10.2 スマート I/O コード

I/O 関数の `print` および `scan` ファミリ関数に関連するライブラリコードは、プロジェクトにおけるそれらの関数の用法に基づいて、コンパイルごとにコンパイラによってカスタマイズできます。これにより、プログラム イメージにリンクされる冗長なライブラリコードが削減されるため、プログラムメモリとデータメモリの使用量を削減できます。

スマート出力(print ファミリ) 関数には以下が含まれます。

`printf`	`snprintf`	`sprintf`
`vprintf`	`vsnprintf`	`vsprintf`

スマート入力(scan ファミリ) 関数には以下が含まれます。

`scanf`	`sscanf`	`vscanf`	`vsscanf`

プロジェクトをビルドするたびに、コンパイラはプロジェクトの C ソースコードを解析し、スマート I/O 関数への呼び出しを検出します。書式文字列内に存在する変換指定が全ての呼び出しで照合され、それらが存在すればプログラム イメージ出力内で関連する機能を備えたライブラリ ルーチンのインクルードがトリガされます。

例として、プログラムが以下の呼び出しのみを含む場合、

```
printf("input is:%d\n", input);
```

スマート I/O が有効であれば、コンパイラはプログラム内で `printf` によって`%d` プレースホルダのみが使われている事を検出します。これにより `printf` を定義するライブラリ ルーチンがリンクされ、これには少なくとも 10 進整数の print を処理可能な基本関数が含まれます。

プログラムに以下の呼び出しが追加された場合、

```
printf("input is:%f\n", ratio);
```

コンパイラは、`printf` によって`%d` および`%f` プレースホルダの両方が使われている事を検出します。この場合、リンクされるライブラリ ルーチンには、プログラムの全ての要件を満たすために機能が追加されます。

以下では、本コンパイラにおけるスマート I/O 機能の動作について説明します。`<stdio.h>` ヘッダに含まれる全ての I/O 関数の構文と使用法は、『Microchip Unified Standard Library Reference Guide』に記載されています。

5.10.2.1 PIC デバイス向けのスマート I/O

PIC MUC 向け MPLAB XC8 コンパイラを使う場合、スマート I/O 関数の書式文字列を処理するコードは 2 つのソースファイル(入力用の `doprnt.c` と出力用の `doscan.c`) 内で定義され、それらの書式文字列内で使用可能な全ての変換指定子、length 修飾子、精度指定子、フィールド幅、フラグが実装されます。これらの機能は、プロジェクト内での I/O 関数の用法に基づいて条件付きでプログラムイメージにインクルードされる個々のコード セクションにより実装されます。スマート出力関数によって使われる機能の検出は、スマート入力関数とは独立して行われます。しかし、それらの関数の間にその他の違いはなく、書式コードは共有されます。例えば `printf` 関数によって使われる書式文字列を処理するコードは、`vsnprintf` 関数によって使われる物と同じであり、デスティネーションに書き込むコードだけが異なります。

I/O 関数への呼び出し内の書式文字列が文字列リテラルではない場合、コンパイラは I/O 関数のどの機能が使われているのか判別できません。この場合、書式コードのより完全なバージョンがプログラム イメージにインクルードされます。書式文字列が文字列リテラルでなくても、コンパイラはある程度の憶測を立てる事ができます。特に、書式文字列に続く引数の数と型から、どのプレースホルダが有効なのかを判別できる場合があります。これにより、たとえ書式文字列が文字列リテラルでなくても、生成される書式コードのサイズと複雑さを最小限に維持できます。

例として、プログラムが以下の呼び出しのみを含む場合、

```
printf(myFormatString, 4, 6);
```

コンパイラは、引数の型から少なくとも「出力関数に対して浮動小数点型のプレースホルダは要求されない」という事を判断し、それらをプログラム イメージから除外できます。

スマート I/O ライブラリコードの解析とカスタマイズは自動的に行われます。この動作を制御するためのオプションはありません。プログラム イメージにリンクされた I/O 機能は、プログラム内のスマート I/O 関数が使う書式文字列を調整する事によってのみ制御できます。プログラム内のスマート I/O 関数に対する呼び出し内で浮動小数点指定子および修飾子を使わない事により、コードサイズを最も効果的に削減できます。

スマート I/O 関数はビルドのたびにオリジナルのソースコードから再生成されますが、それらは最初に(リンカに渡される前に)ライブラリ ファイル内へパッケージされます。このため、例えばユーザ独自の(非スマート) `printf()` 関数を定義する必要がある場合、スマート I/O 関数を通常の方法で同名のユーザ定義関数に置き換える事ができます。

5.10.3 ユーザ定義ライブラリ

ユーザ定義ライブラリを作成してプログラムにリンクできます。多数のソースファイルを扱うよりもライブラリ ファイルの方が管理が容易でコンパイル時間も短くて済みますが、そのプロジェクトで使うターゲット デバイスとオプションに対応している必要があります。ライブラリを異なるプロジェクトで使うには、バージョンの異なるライブラリを作成して管理する事が必要になる場合があります。

ライブラリは、ライブラリアン `xc8-ar` を使って作成できます(8.1 「アーカイバ/ライブラリアン」参照)。

ユーザ定義ライブラリは、ビルド後にコマンドラインでソースファイルと一緒に使えるようになります。IDE プロジェクトにライブラリを追加するか、オプションを使って追加するライブラリを指定できます。

コマンドラインで指定されたライブラリ ファイルは、C 標準ライブラリの前に(しかし全てのプロジェクト モジュールの後で)、未解決シンボルについてスキャンされます。従ってそれらの内容は、C 標準ライブラリ内の定義を再定義する可能性があります。

5.10.4 ライブラリ ルーチンの使用

ライブラリ ファイルがプロジェクトに含まれる場合、参照されているライブラリ関数とオブジェクトは自動的にプログラムにリンクされます。あるライブラリ ファイルから 1 つの関数だけを使う場合、そのライブラリ内の他の関数はインクルードされません。

プログラムは、使用する全てのライブラリ関数またはシンボルに対する宣言を必要とします。標準ライブラリは、ソースファイルにインクルード可能な標準 C ヘッダ(.h ファイル)と一緒に提供されます。各ライブラリ関数に対応するヘッダについては、C 言語の教科書または 9.1 「ライブラリ サンプルコード」を参照してください。ユーザ独自のライブラリを作成する場合、通常はユーザがライブラリ ヘッダを書きます。

ヘッダファイルはライブラリ ファイルではありません。ライブラリ ファイルにはプリコンパイル済みコード(通常は関数と変数の定義)が収められています。ヘッダファイルはライブラリ ファイル内の関数、変数、型の宣言(ライブラリ ファイル内の定義に対応)を提供します。ヘッダはプリプロセッサ マクロを定義する事もできます。

5.11 C 言語とアセンブリ言語の併用

C コードと一緒にアセンブリコードを使う方法には以下の 2 通りがあります。

- アセンブリコードを別のアセンブリソース モジュールに格納する
- アセンブリコードを C コードにインライン化する

Note: プロジェクトに含めるアセンブリコードが多ければ多いほど、管理には手間がかかります。また、コンパイラがアップデートされた時に、アセンブリコードを修正する必要が生じる可能性があります。C 言語で書かれたコードは、これらの影響を受けません。
アセンブリコードを追加する必要がある場合、インライン アセンブリとして C コード内に含めるのではなく、別のアセンブリ モジュール内で独立したコードとして書く事を推奨します。

5.11.1 アセンブリ言語モジュールとの連携

モジュール全体をアセンブリコードで記述する事もできます。理想的には、そのようなコードはアセンブリ ソースファイルに格納すべきです。これらのファイルをプロジェクトにインクルードして C ソースファイルと一緒にビルドする事で、リンカはこれらのソースファイルからプログラム イメージを生成できます。

完全なアセンブリコードのみを格納したソースファイルには拡張子「.s 」を使います。プリプロセッサによる前処理が必要なプリプロセッサ ディレクティブが含まれる場合は拡張子「.S」を使う必要があります。-xassembler-with-cpp オプション(4.6.2.6 「X: ソース言語オプションの指定」参照)を使うと、拡張子に関係なく全てのアセンブリ ソースファイルに前処理を適用できます。

既定値ではアセンブリ モジュールはアセンブラによって最適化されませんが、-fasmfile オプション(4.6.6.7 「asmfile オプション」参照) によって最適化を有効にできます。

5.11.2 インライン アセンブリ

命令文 asm();を使う事で、アセンブリ命令を C コードに直接埋め込む(インライン化する)事ができます。

これらの命令は、関数呼び出しと同様にカッコで囲まれた文字列として書かれますが、関数のように呼び出されるのではありません。通常は文字列内に 1 つの命令を配置しますが、「\n」文字を使って区切る事により複数のアセンブリ命令を指定する事もできます(例: asm("movlw 55\nmovwf _x");)。複数の命令文を使い、各命令文に命令を 1 つだけ配置した方がコードは読みやすくなります。

asm()形式のインライン アセンブリは、C ソースコード内のどこに配置しても、全ての C 制御フロー構造に合うように埋め込まれます。

例:

```
unsigned int var;
int main(void)
{
    var = 1;
    asm("bcf 0,3");
    asm("BANKSEL _var");
    asm("rlf (_var)&07fh");
    asm("rlf (_var+1)&07fh");
}
```

インライン アセンブリコードは、アセンブラ オプティマイザによって最適化されません。

インライン アセンブラコードを使う場合、コンパイラが生成するコードと相互作用しない事が非常に重要です。コード ジェネレータは、アセンブリコードによるレジスタの使用を検出できないため、レジスタが使われるかどうか、あるいは書き換えられるかどうかを調べません。代わりにコンパイラはインラインアセンブリに遭遇した時点でそれまでのバンク選択トレース情報が無効になると判断するため、インラインアセンブリ側で現在のバンクを指定している全ての SFR や SFR 内のビットを保存し、復旧させる必要はありません。

コンパイラによって使われるレジスタの説明は、5.6 「レジスタの使用」に記載しています。どのレジスタがアセンブリ前後のコードによって使われるのか疑わしい場合、-Wa,-a オプション使ってプログラムをコンパイルする事で、生成されたアセンブリコードを調べる事ができます。プログラムの他の部分が変更されると、コンパイラによって使われるレジスタとコードストラテジも変更されるという事に注意が必要です。

C 関数がメインライン コードと割り込みコードから呼び出される場合、その関数は複製される可能性があります(5.8.7 「関数の複製」参照)。コンパイラによって生成されたラベルが複製されないようにするために特殊な接頭辞が使われますが、これは C 関数内の手書きインライン アセンブリコード内で定義されたラベルには適用されません。従って、複製される可能性のある関数が含まれる場合、インライン アセンブリ向けにアセンブリラベルを定義するべきではありません。

5.11.3 アセンブリコードと C コードの間の相互作用

MPLAB XC8 C コンパイラは、C コードがユーザ定義アセンブリコードの要求に従う事ができるように設計された各種の機能を備えています。また、手書きのアセンブリコードが C コードから生成されたアセンブリコードと干渉しないようにするために守るべきガイドラインがいくつか存在します。

5.11.3.1 スタンドアロン アセンブリコードの書き方

C コードから呼び出し可能なアセンブリコードを書く事ができます。そのようなコードを書く場合、以下のガイドラインに従う必要があります。

- コード内で SFR を使う場合、<xc.inc>アセンブリ ヘッダファイルをインクルードする必要があります。#include を使ってこのヘッダファイルをインクルードする場合、ソースファイルを前処理する必要があります。
- 実行可能アセンブリコード向けに適切な psect を定義または選択する必要があります(初歩的なガイドラインは 5.14.1 「コンパイラが生成する psect」参照)。
- コードの名前(ラベル) の先頭には 1 つのアンダースコア文字を付けます。
- コードのラベルは、他のモジュールからグローバルにアクセス可能となるようにします。

- 引数の受け渡しがモデル化される可能性のあるコードには適切な C 等価プロトタイプを選択します。
- コードとの間で値の受け渡しが必要である場合、可能であれば再入可能コードを書きます(5.11.3.2 「パラメータを持つ再入可能アセンブリコードの書き方」参照)。再入可能にできなければ、値を渡すために通常の変数を使います。
- オプションにより、シグネチャ値を使って関数の呼び出し時に型のチェックを可能にします。
- バンク選択命令を使い、全ての変数シンボルのアドレスをマスクします。

下記の例は、8 ビットの引数を PORTB の内容に加算し、その結果を 8 ビット値として返すミッドレンジ デバイス向けのアセンブリコードです。他のデバイスを使う場合もコードも同様です。

```
#include <xc.inc>
GLOBAL _add             ; make _add globally accessible
SIGNAT  add,4217        ; tell the linker how it should be called
; everything following will be placed into the code psect
PSECT code
; our routine to add to ints and return the result
_add:
    ; W is loaded by the calling function;
    BANKSEL (PORTB)               ; select the bank of this object
    addwf BANKMASK(PORTB),w    ; add parameter to port

    ; the result is already in the required location (W) so we can
    ; just return immediately
    return
```

コードは定義済みの psect (code) 内に配置され、<xc.inc>をインクルードした後に利用可能となります。このセクションは CODE リンカクラスに属するため、コード用に確保されたメモリ領域に自動的に配置されます(ユーザが既定値リンカ オプションを調整する必要はありません)。デバイスは、このセクションを使って実行可能コードを保持できます。

ユーザ独自の psect を作成する必要がある場合、以下のコードがミッドレンジ デバイス向けに使えます。

```
PSECT mytext,local,class=CODE,delta=2
```

このセクションで使われる delta フラグ(=2) は、psect が配置されるメモリ空間がワードアドレス指定可能である事を示します(PIC10/12/16 デバイスの場合)。PIC18 デバイスの場合、プログラムメモリはバイトアドレス指定が可能ですが、命令はワード境界に整列する必要があります。このため、セクション定義には以下に示すコードを使います。この場合、delta の値は 1 (既定値設定であるため、下のコードでは省略)ですが、reloc (アラインメント) フラグを 2 に設定する事で、セクションがワード境界アドレスから始まるように指定しています。

```
PSECT text0,class=CODE,reloc=2
```

PSECT アセンブラ ディレクティブで使うフラグの詳細は、6.1.9.36 「PSECT ディレクティブ」を参照してください。C 識別子とアセンブリコードで使われる識別子の間の割り当ては、5.11.3 「アセンブリコードと C コードの間の相互作用」を参照してください。アセンブリコード領域でのコード名は、C 識別子「add」に対応させるために「_add 」とする必要があります。このコードは他のモジュールから呼び出されるため、GLOBAL アセンブラ ディレクティブ (6.1.9.21 「GLOBAL ディレクティブ」参照)を使ってこのラベルをグローバルにアクセス可能とする必要があります。

また、このコードが正しく呼び出される事をリンカがチェックできるようにするため、このコードでは SIGNAT ディレクティブ(6.1.9.40 「SIGNAT ディレクティブ」参照)を使っています。

引数を渡すために W レジスタが使われます。パラメータを渡す方法については、5.7.5 「関数のパラメータ」を参照してください。

BANKSEL ディレクティブを使ってバンクを正しく選択し、BANKMASK マクロを使って全てのアドレスを適切なサイズへとマスクします。

C コードからアセンブリコードを呼び出すために、そのコードの宣言が必要です。以下のコードは、アセンブリコードの動作を宣言した後に、そのコードを呼び出しています。

```
// declare the assembly routine so that the call from C code is correct
extern unsigned char add(unsigned char a);

int main(void) {
    volatile unsigned char result;
```

```
    result = add(5); // call the assembly routine
}
```

5.11.3.2 パラメータを持つ再入可能アセンブリコードの書き方

拡張ミッドレンジおよび PIC18 デバイス向けの手書きアセンブリコードは、ソフトウェア スタックを使う事で C コードからの再入可能呼び出しができるように書く事ができます。そのようなコードでは、必要に応じてパラメータを受け取る事、値を返す事、独自のローカル オブジェクトを定義する事が可能です。

そのようなコードを作成する場合、以下の手順に従う必要があります。

1. C ソースコード内で、アセンブリコード向けの C プロトタイプを宣言し、パラメータと戻り値の型を適切に選択します。
2. <xc.inc> アセンブリ ヘッダファイルをインクルードします。#include を使ってインクルードする場合、ソースファイルの拡張子として「.S」を使う必要があります。
3. 必要に応じ、stack_auto *name*,*size* マクロ(*name* は任意の有効なアセンブラ識別子、*size* は変数のバイト数)を使って C の auto に相当するワーク変数サイズを定義します。
4. 必要に応じ、ワーク変数の各パラメータを定義します。パラメータは先の auto 相当変数サイズ宣言の後で定義する必要があり、その順序は C プロトタイプ内でそれらが現れる順序に一致する必要があります。
5. コード内の全ての命令より前で、alloc_stack マクロを使ってスタックを 1 度初期化します。
6. 各リターン命令の直前で、restore_stack マクロを使ってスタックを復元します。

5.11.1 「アセンブリ言語モジュールとの連携」で説明した通りの通常の方法でアセンブリコードを書きます。

各 auto 変数とパラメータ変数は、スタックポインタ(SFR1) に対して一意のオフセットで配置されます。前述のガイドラインに従う場合、シンボル name_offset が使えます。これには、その名前を持つ変数のスタックポインタ オフセットが代入されます。これらのマクロは、auto およびパラメータ変数の両方に対して存在します。

コードが値を返す場合、これはそのコードを呼び出すコードによって期待される位置に配置される必要があります。C-callableコードの完全な説明は、5.7.6.2「ソフトウェア スタックを使う関数の戻り値」を参照してください。要約すると、サイズが 1～4 バイトのオブジェクトの場合、これらは btemp として参照される一時変数 + オフセットに書き込まれる必要があります。ユーザが前述のマクロを使う場合、このシンボルは自動的にユーザのコードにリンクされます。

コードの実行中にスタックポインタを勝手に調整しない事を推奨します。各 auto およびパラメータ変数のオフセットを定義するシンボルは、スタック ポインタが変更されていないと想定します。しかし、アセンブリコードが他の再入可能コードを呼び出す場合(それらが C コード内で定義されていようがアセンブリコード内で定義されていようが関係なく)、引数をスタックへプッシュし、関数を呼び出し、スタックから戻り値を削除するアセンブリコードを書く必要があります。

以下のサンプルコードは、PIC16F1xxx デバイス向けに書かれた再入可能アセンブリコード「_inc」です。このコードの引数と戻り値は、下記の C プロトタイプにより記述されます。

```
extern reentrant int inc(int foo);
```

このコードは、渡された int 引数より 1 大きい int 値を返します。このコードは auto 変数「x」を、説明のためだけに使います。

```
#include <xc.inc>
PSECT text2,local,class=CODE,delta=2
GLOBAL _inc
_inc:
    stack_auto x,2      ;an auto called 'x'; 2 bytes wide
    stack_param foo,2   ;a parameter called 'foo'; 2 bytes wide
    alloc_stack
    ;x = foo + 1;
    moviw   [foo_offset+0]FSR1
    addlw   low(01h)
    movwf   btemp+0
    moviw   [foo_offset+1]FSR1
    movwf   btemp+1
    movlw   high(01h)
    addwfc  btemp+1,f
    movf   btemp+0,w
    movwi   [x_offset+0]FSR1
```

```
    movf    btemp+1,w
    movwi   [x_offset+1]FSR1
    ;return x;
    moviw   [x_offset+0]FSR1
    movwf   btemp+0
    moviw   [x_offset+1]FSR1
    movwf   btemp+1
    restore_stack
    return
```

以下のサンプルコードは、PIC18 デバイス向けに書かれた再入可能アセンブリコード「_add」です。このコードの引数と戻り値は、下記の C プロトタイプにより記述されます。

```
extern reentrant int add(int base, int index);
```

このコードは、渡された引数 base および index の int 加算値より 1 大きい int 値を返します。このコードは auto 変数「tmp 」および「result」を、説明のためだけに使います。

```
#include <xc.inc>
psect        text1,class=CODE,space=0,reloc=2
GLOBAL _add
_add:
    stack_auto tmp,2      ;an auto called 'tmp'; 2 bytes wide
    stack_auto result,2   ;an auto called 'result'; 2 bytes wide
    stack_param base,2    ;a parameter called 'base'; 2 bytes wide
    stack_param index,2   ;a parameter called 'index'; 2 bytes wide
    alloc_stack
    ;tmp = base + index;
    movlw   base_offset
    movff   PLUSW1,btemp+0
    movlw   base_offset+1
    movff   PLUSW1,btemp+1
    movlw   index_offset
    movf    PLUSW1,w,c
    addwf   btemp+0,f,c
    movlw   index_offset+1
    movf    PLUSW1,w,c
    addwfc  btemp+1,f,c
    movlw   tmp_offset
    movff   btemp+0,PLUSW1
    movlw   tmp_offset+1
    movff   btemp+1,PLUSW1
    ;result = tmp + 1;
    movlw   tmp_offset
    movf    PLUSW1,w,c
    addlw   1
    movwf   btemp+0,c
    movlw   tmp_offset+1
    movff   PLUSW1,btemp+1
    movlw   0
    addwfc  btemp+1,f,c
    movlw   result_offset
    movff   btemp+0,PLUSW1
    movlw   result_offset+1
    movff   btemp+1,PLUSW1
    ;return result;
    movlw   result_offset
    movff   PLUSW1,btemp+0
    movlw   result_offset+1
    movff   PLUSW1,btemp+1
    restore_stack
    return
```

5.11.3.3 等価アセンブリ シンボル

大部分の C オブジェクトは、対応する等価アセンブリ シンボルに割り当てられます。

この時にアセンブリコード領域と C コード領域で同一のシンボル名が使われたとしても、互いに干渉する事はありません。例えば、シンボル main がアセンブリコード領域で定義された場合、これは C コード領域にある main シンボルとは区別され異なる位置(アドレス)を参照します。

C 関数の名前は同一名を持つアセンブラ ラベルに割り当てられますが、名前の前に 1 個のアンアースコアが付加されます。従って、関数 main()はアセンブリラベル_main を定義します。

ベースライン PIC デバイスでは、関数向けに代替アセンブリコード領域シンボルを使う事ができます。これらのデバイスでは、call 命令のデスティネーションはプログラムメモリ ページの前半部に制限されます。従ってコンパイラは関数を 2 つの部分に分けてエンコードします。

例: ベースライン デバイス向け C 関数 add()

```
entry__add:
ljmp    _add
```

ラベル entry__add は関数のエントリポイントであり、常にプログラムメモリ ページの前半部に配置されます。ラベル_add に対応する実際の関数ボディは ljmp 命令の分岐先となりプログラムメモリ空間の全域にジャンプ可能です (6.1.1.7 「ロングジャンプと呼び出し」参照)。

アセンブリコードからベースライン デバイスのコードを呼び出そうとする場合、デバイスにおけるこの制限と、コンパイラが C 関数向けにこの制限に対応する方法に気を付ける必要があります。手書きアセンブリコードは常に通常のアセンブリ等価関数ラベルではなく entry__*funcName* ラベルを呼び出す必要があります。

C 関数が static で修飾され、プログラム内に全く同名の static 関数が複数存在する場合、「最初」の関数の名前が通常のアセンブリ シンボルに割り当てられ、後続の関数は以下の形態を持つ特殊なシンボルに割り当てられます。

_functionName@fileName$F*number*

functionName: 関数の名前、*fileName*: 関数を格納したファイルの名前、*number*: 一意の連続番号

この場合、「最初」の定義は複雑です。通常、main()を定義するソース モジュール内にシンボルが存在する場合、そのシンボルが最初に処理されます。このモジュール内にシンボルが存在しない場合、コマンドラインでソースファイルが指定された順番によって処理順が決まります。

例として、プログラムが同名の 2 つの static 関数(add())を含む場合を想定します。1 つは main.c ファイル内にあり、もう 1 つは lcd.c ファイル内にあります。最初の関数は、アセンブリラベル_add を生成します。2 つ目の関数は、例えば_add@lcd$F38 というラベルを生成するかもしれません。

静的記憶期間を持つ C 変数の名前も、同一名を持つアセンブリラベルに割り当てられますが、名前の前にアンダースコアが付加されます。従って、変数 result はアセンブリラベル_result を定義します。

C 変数が static で修飾されている場合、プログラム内に全く同名の変数が複数存在する可能性があります。関数の外で定義された static 変数に適用される規則は、static 関数に対して適用される規則と同様です。「最初」の変数は、変数名の先頭にアンダースコアが追加された名前を持つシンボルに割り当てられ、その他の変数は以下の形態の名前を持つシンボルに割り当てられます。

_variableName@fileName$F*number*

variableName: 変数の名前、*fileName*: 変数を格納したファイルの名前、*number*: 一意の連続番号

全てのローカルな(すなわち関数定義内で定義された) static 変数は、以下の形態のアセンブリ名を持ちます。

functionName@variableName

output という名前を持つ static 変数が関数 read()内に存在し、同名の static 変数が関数 update()内でも定義されている場合、アセンブリ内のシンボルにはそれぞれシンボル read@output と update@output を使ってアクセスできます。

再入可能モデルを使う関数は、auto およびパラメータ変数へのアクセスを可能にするシンボルを一切定義しません。アセンブリコード内でこれらへのアクセスを試みてはいけません。これらの変数には特殊なシンボルが定義されますが、それは再入不可モデルを使う関数によって定義されます。これらのシンボルについて以下で説明します。

コンパイルド スタック上のパラメータおよび auto 変数へ容易にアクセスできるようにするため、各変数に対して一意のシンボルを割り当てる決まった置換方式が使われます。シンボルは以下の書式を持ちます。

functionName@variableName

関数 main()が foobar という名前の auto 変数を定義している場合、アセンブリ内でシンボル main@foobar を使ってこの C 変数にアクセスできます。

関数パラメータは、auto 変数と同じシンボル割り当て方法を使います。read()という名前の関数が channel という名前のパラメータを持つ場合、そのパラメータに対応するアセンブリ シンボルは read@channel です。

関数の戻り値に対して C 識別子は割り当てられません。関数の戻り値は、その関数のパラメータ変数(それらが存在する場合)と同じメモリを共有します。戻り値向けに使われるアセンブリ シンボルの書式は?_*functionName* です(*functionName*: 値を返す関数の名前)。従って、関数 getPort()の戻り値は、アセンブリ シンボル?_getPort が保持しているアドレスに配置されます。この戻り値のサイズが 1 バイトよりも大きい場合、各バイトにアクセスする

ためにシンボルにオフセットが追加されます(例: ?_getPort+1)。

中間結果を保持するためにコンパイラが一時変数を生成する場合、それらは auto 変数であるかのように振る舞います。それらに対応する C 変数は存在しないため、アセンブリ シンボルは関数+オフセットの auto ブロックを表すシンボルに基づきます。このシンボルの書式は??_*functionName* です(*functionName*: そのシンボルが使われている関数の名前)。例えば関数 main()が一時変数を使う場合、それらは、シンボル??_main からのオフセットとしてアクセスされます。

5.11.3.4 アセンブリコードからレジスタへのアクセス

分離したアセンブリ モジュールでは、SFR 定義に自動的にアクセスできません。これらのレジスタ定義へアクセスするために、アセンブリ ヘッダファイル<xc.inc> が使えます。このファイルを C コード内のインライン アセンブリ向けに使ってはいけません。使った場合、<xc.h> 内の定義と競合します。

このファイルをインクルードするには、アセンブラの INCLUDE ディレクティブ (6.1.9.23 「INCLUDE ディレクティブ」参照)または C プリプロセッサの#include ディレクティブを使います。後者を使う場合、アセンブリ ソースファイルの拡張子を「.S 」にする必要があります。

このヘッダファイル内のレジスタ向けシンボルは、<xc.h>をインクルードした場合に C コード領域内で使われる識別子(例: PORTA、EECON1 等) と同じであるかのように見えますが、それらはアセンブリコード領域内の異なるシンボルです。ただし、それらは同じメモリ位置に割り当てられます。

レジスタ内のビットの名前は「*registerName*,*bitNumber*」として定義されます。 例えば RA0 ビットは PORTA,0 として定義されます。

SFR を使うミッドレンジ デバイス向けアセンブリ モジュールの例を以下に示します。

```
#include <xc.inc>
GLOBAL _setports
PSECT text,class=CODE,local,delta=2
_setports:
    movlw 0xAA
    BANKSEL (PORTA)
    movwf BANKMASK(PORTA)
    BANKSEL (PORTB)
    bsf RB1
```

C コード内のインライン アセンブリからレジスタ定義にアクセスする場合、<xc.h> ヘッダを C モジュールにインクルードする必要があります。このヘッダに含まれる情報は、インライン アセンブリ シンボルと C コードからアクセス可能な通常のシンボルの両方を定義します。

レジスタ名に使われるシンボルは、<xc.inc> によって定義されているシンボルと同じです。上のサンプルコードをインライン アセンブリとして書き換えた例を以下に示します。

```
#include <xc.h>

int main(void) {
        asm("movlw 0xAA");
        asm("BANKSEL (PORTA)");
        asm("movwf " ___mkstr(BANKMASK(PORTA)));
        asm("BANKSEL (PORTB)");
        asm("bsf " ___mkstr(BANKMASK(PORTB)) ","  ___mkstr(_PORTB_RB1_POSN));
}
```

BANKMASK()はプリプロセッサ マクロであり、C 文字列リテラル内で展開されないという事に注意が必要です。インライン アセンブリ内で BANKMASK()マクロを使う場合、___mkstr()マクロ (先頭に 3 個のアンダースコアが付く事に注意、<xc.h>をインクルードすると定義される) を使ってマクロの置換リストを文字列へと変換します。RB1 のビット位置を定義するマクロも同様に展開されます。

コードジェネレータは、SFR 名称がインラインアセンブリ内で使われている事を検知しないため、このような措置が必要です。コンパイラによって使われるレジスタのリストと詳細は 5.6 「レジスタの使用」に記載しています。

5.11.3.5 絶対アドレス psect

MPLAB XC8 コンパイラは、絶対アドレス psect のアドレス境界 (abs、ovrld、PSECT フラグを使って定義) を特定し、コンパイル処理の前にそのデータメモリが C ソースによって使われないよう予約します。アセンブリコードによって要求されるデータメモリは絶対アドレス psect を使う必要がありますが、プログラムメモリ内に配置される psect は絶対アドレスでなくても構いません。

以下のサンプルコードは、アセンブリコード ファイルに格納され、データ空間内のアドレス 0x110 に配置する必要があるテーブルを定義します。

```
PSECT lkuptbl,class=RAM,space=1,abs,ovrld
ORG 110h
lookup:
    DS 20h
```

プロジェクトのコンパイル時に、メモリ空間 1 (データメモリ) 内のアドレスレンジ 0x110～0x12F が使われるという事が記録され、C ソースをコンパイルする前にその領域が予約されます。絶対アドレス psect は常にアドレス 0 から始まります。そのような psect では、ORG ディレクティブを使って非 0 の開始アドレスを指定できます。このディレクティブの詳細は 6.1.9.30 「ORG ディレクティブ」を参照してください。

5.11.3.6 未定義シンボル

C ソースコード内で定義されたオブジェクトがアセンブリコード内でのみアクセスされる場合、コンパイラはそのオブジェクトが使われないと判断してそのオブジェクトを削除してしまう可能性があり、その場合は未定義シンボルエラーが発生します。

この問題に対処するため、MPLAB XC8 コンパイラはアセンブリ定義を持たないアセンブリコード内のシンボル (通常はラベル)を検出します。C ソースコード内でこれらのシンボルが検出された場合、削除される事を防ぐため、これらのシンボルは自動的に volatile (5.3.8.2 「volatile 型修飾子」参照) として扱われます。

例えば、C プログラムが 1 つの変数を以下の通りに定義し、

```
int input;
```

この変数が C コード内で全く使われない場合を想定します。以下の例では、アセンブリ モジュールは GLOBAL アセンブラ ディレクティブを使ってこのシンボル(先頭にアンダースコア文字を付加)を単純に宣言した後に、このシンボルを使います(5.11.3 「アセンブリコードと C コードの間の相互作用」参照)。

```
GLOBAL _input, _raster
PSECT text,local,class=CODE,reloc=2
_raster:
    movf   _input,w
```

5.12 最適化

MPLAB XC8 コンパイラによる最適化は、アセンブリコードへ変換される前の C ソースコードに対して実行される C レベルの最適化と、コンパイラが生成したアセンブリコードに対して実行されるアセンブリレベルの最適化に大きく分類できます。

C レベルの最適化は、コード生成フェイズの早期に実行され、フローオン効果が得られます(1 つの最適化を実行した結果から新たな最適化の余地が発生する場合があります)。これらの最適化はデバッグ情報が生成される前に適用されるため、プログラムのソースレベルのデバッグに対する影響は比較的小さくて済みます。

これらの最適化の一部はコード生成処理に不可欠であるため、オプションによって無効にする事はできません。特定の最適化を無効にする方法は後で説明します。

ライセンスなしのコンパイラを使う場合、一部の最適化レベルは無効になります(4.6.6 「最適化を制御するためのオプション」参照)。たとえそれらを有効にしても、最適化は非常に特殊な条件が成立する場合にのみ適用されます。このため、コードの一部の行は最適化されるのに、他の同じような行は最適化されないといった結果が生じる場合があります。

最適化レベルに応じて利用可能な最適化機能が決まります(表 5-13 参照)。

表 5-13. 最適化レベルの設定

レベル	利用可能な最適化
O0	• 基本的な最適化
O1	• 最小限のコード ジェネレータ最適化
O2	• 全ての一般コード ジェネレータ最適化 • 最小限のアセンブリ最適化

.....続き

レベル	利用可能な最適化設定
03 (ライセンスが必要)	• 全ての一般および速度重視コード ジェネレータ最適化 • 全ての一般および速度重視アセンブラ最適化
0s (ライセンスが必要)	• 全ての一般およびサイズ重視コード ジェネレータ最適化 • 全ての一般およびサイズ重視アセンブラ最適化

アセンブリレベルの最適化については、6.2 「アセンブリレベルの最適化」を参照してください。最小限のコード ジェネレータ最適化は以下により構成されます。

- 非標準キーワードまたはコンパイラ ディレクティブを使わずにオブジェクトを RAM バンクに割り当てるためのプログラム全体の解析
- 定数式の単純化および畳み込み
- 評価式ツリー最適化による効率的アセンブリコード生成
- 変数の数値内容が特定可能な場合に定数伝播が実行されます。volatile ではなく値が厳密に特定可能な変数は、数値で置き換えられます。初期化されないグローバル変数は、値が代入されるまで 0 であると見なされます。
- 到達不能な C 命令文は、アセンブリコードが生成される前に削除されます。これにより、C レベルで後続の最適化が適用可能となります。

以下は、C レベルでのより高度な最適化の内容です。これらは C 式または C 式から生成されるコードをシンプル化します。これらは、モジュールごとに適用されるのではなく、プログラム全体に対して適用されます。

- コンパイラは、アセンブリコードを生成する際に現在の RAM バンクを追跡します。これにより、コンパイラはバンク選択命令の数を削減できます。
- コードが生成される前に、全ての評価式ツリーに対して演算強度の軽減や式変換が適用されます。これは、式を等価でより効率的な演算に置き換えます。
- プログラム内の未使用変数は削除されます。これは全ての変数に適用されます。削除された変数向けにメモリは予約されません。これらの変数はどのリストファイルおよびマップファイルにも示されず、デバッグ情報にも現れません(デバッガ内で監視されません)。未使用変数が見つかると、警告が出力されます。volatile で修飾されたグローバル オブジェクトは決して削除されません(5.3.8.2 「volatile 型修飾子」参照)。変数のアドレスを取得する事や、手書きアセンブリコード内で変数のアセンブリコード領域シンボルを参照する事も、変数の使用に含まれます。
- 変数に対する冗長な(後で使われる事のない) 代入は、その変数が volatile でない限り、削除されます。代入命令文は、オリジナルのソースコード内に元々存在しなかったかのように、完全に削除されます。代入命令文に対応するコードは生成されないため、デバッガ内でその行にブレークポイントを設定する事はできません。
- プログラム内の未使用関数は削除されます。直接的にも間接的にも呼び出される事がなく、そのアドレスが取得または参照される事もない関数は、未使用と見なされます。関数全体が、オリジナルのソースコード内に元々存在しなかったかのように削除されます。その関数に対応するコードは生成されないため、デバッガ内でその関数内の行にブレークポイントを設定する事はできません。分離した手書きアセンブリコード内で、関数のアセンブリコード領域シンボルが参照されていれば、その関数は削除対象になりません。条件は GLOBAL ディレクティブ内のシンボルを使っているという事のみです。
- 関数内の未使用のリターン式は削除されます。その関数を呼び出している全ての箇所で戻り値を放棄する場合、戻り値は未使用であると見なされます。戻り値の計算に関連するコードは削除され、関数は戻り値の型が void であるかのようにコード化されます。
- 読み出される前に値が代入される変数は、実行時起動コードによってクリアまたは初期化されません。これには非 auto 変数のみが考慮され、他のコードがそれらの変数を読み出す前にそれらの変数に値が代入される場合、それらの変数は__persistent として扱われます(5.3.9.1 「bank 型修飾子」参照)。全ての__persistent オブジェクトは実行時起動コードによってクリアされないため、この最適化はプログラムの起動速度を向上させます。
- ポインタサイズは、それらがアクセス可能なターゲット オブジェクトに適するように最適化されます。コンパイラはポインタ変数に対する全ての代入を追跡し、各ポインタがアクセス可能なターゲットのリストを保持します。各ターゲットのメモリ空間が既知であり、ポインタに必要なサイズをメモリから逆参照する事で、ポインタごとのカスタマイズが可能となります。

- 特定ターゲットのみのポインタ参照は、ターゲットオブジェクトの直接アクセスと置き換え可能です。これはデータへのポインタや関数へのポインタにも適用されます。
- const 修飾されたオブジェクトは、データメモリ内への配置が考慮されます(プログラムによって使われないメモリ領域が利用できる場合)。

MPLAB X IDE 等の統合開発環境において変数をウォッチする場合、最適化処理によりレジスタ変数として値を保持していると、不正な値を示す場合があります。ELF/DWARF デバッグファイル フォーマットを使うと、このような現象を最小限に抑えられる場合があります。デバッグ中のコード内でレジスタが使われているかどうかは、アセンブリ リストファイルで確認できます。

5.13 前処理

全ての C ソースファイルはコンパイル前に処理されます。前処理済みのファイルはコンパイル後に削除されますが、-E オプションを使うとこのファイルを保存して後で調べる事ができます(4.6.2.2 「E: 前処理のみ」参照)。

拡張子が「.S 」のアセンブリ ソースファイルは前処理されます。

5.13.1 プリプロセッサ ディレクティブ

XC8 コンパイラは、標準ディレクティブに加えて各種の特別なプリプロセッサ ディレクティブをサポートします。それらの全てを表 5-14 に示します。

表 5-14. プリプロセッサ ディレクティブ

ディレクティブ	意味	例
#	プリプロセッサ NULL ディレクティブ(動作なし)	番号

#define	プリプロセッサ マクロを定義します。	`#define SIZE (5)`
		`#define FLAG`
		`#define add(a,b) ((a)+(b))`

#elif	#else #if の短縮形	#ifdef 参照
#else	定数式が「偽」であれば続くソース行をインクルードします。	#if 参照
#endif	条件付きのソース インクルードを終了します。	#if 参照
#error	エラーメッセージを生成します。	#error Size too big

#if	定数式が「真」であれば続くソース行をインクルードします。	`#if SIZE < 10`
		`  c = process(10)`
		`#else`
		`  skip();`
		`#endif`

#ifdef	プリプロセッサ シンボルが定義されている場合に続くソース行をインクルードします。	`#ifdef FLAG`
		`  do_loop();`
		`#elif SIZE == 5 skip_loop();`
		`#endif`

#ifndef	プリプロセッサ シンボルが未定義の場合に続くソース行をインクルードします。	`#ifndef FLAG`
		`  jump();`
		`#endif`

#include	テキストファイルをソースにインクルードします。	`#include <stdio.h>` `#include “project.h”`
#line	リスティングの行番号とファイル名を指定します。	#line 3 final

.....続き

ディレクティブ	意味	例
#*nn* filename	*nn*: 番号、*filename*: ソースファイル 指定されたソースファイルの指定行番号の内容	#20 init.c
#pragma	コンパイラに固有のオプション	5.13.3 「プラグマ ディレクティブ」参照
#undef	プリプロセッサ シンボルを未定義にします。	#undef FLAG
#warning	警告メッセージを生成します。	#warning Length not set

引数を使ったマクロ展開は、#演算子を使って続く引数を文字列に変換でき、##演算子を使って引数の両側の字句(式)を連結できます。2 つの式を連結する場合、式がマクロ展開を必要とする場合は、以下の例のように 2 つのマクロを使う事を考慮します。

```
#define __paste1(a,b)    a##b
#define __paste(a,b)     __paste1(a,b)
```

paste マクロを使って、それら自身がさらなる展開を必要とする可能性のある 2 つの式を連結します。一度展開されたマクロ識別子が連結後に現れた場合、それは再度展開されないという事に注意が必要です。

5.13.1.1 プリプロセッサの算術演算

プリプロセッサ マクロ置換式はテキストであり、型を使いません。それらがインクルード指示(後述) への制御式の一部でない場合、マクロはプリプロセッサによって評価されません。マクロがテキストとして展開され、前処理が完了すると、展開後は C ソースコードと一緒にコードジェネレータによって評価可能な「式」となります。展開後の「式」内のトークンは型を継承し、通常の方法により汎用的整数からの拡張や型変換の対象となります。

マクロが条件付きインクルード指示（#if または#elif）の制御式の一部である場合、マクロはプリプロセッサによって評価される必要があります。この評価の結果は、同じシーケンス記述の C ソースの結果とは異なる事がよくあります。例えばプリプロセッサは、制御式内のリテラル値にサイズを割り当てますが、そのサイズは<stdint.h>内の intmax_t のサイズに依存します。

MPLAB XC8 C コンパイラでは、このサイズは PIC18 デバイス向けに C99 でコンパイルする場合にのみ 64 ビットであり、これ以外の場合に 32 ビットです。

以下のコードは期待通りに動作しない可能性があります。プリプロセッサは MAX が 32 ビット乗算の結果(0xF4240)であると評価します。しかし、long int 変数として定義された max には値 0x4240 が代入されます(定数 1000 は signed int 型であるため、C コード領域の乗算も 16 ビット signed int 型を使って実行される)。

```
#define MAX 1000*1000
...
#if MAX > INT16_MAX       // evaluation of MAX by preprocessor
    long int max = MAX;   // evaluation of MAX by code generator
#else
    int max = MAX;        // evaluation of MAX by code generator
#endif
```

C コード領域でのオーバーフローは、マクロ内で定数接尾辞を使う事により防ぐ事ができます(5.3.7 「定数の型と書式」参照)。例えば、マクロ式内の値の後の接尾辞 L は、C コンパイラによって long 型として解釈される必要がある事を示します。しかし、この接尾辞はプリプロセッサがこの値を評価する方法には影響を及ぼしません(プリプロセッサがこの値を評価する必要がある場合)。

下の例は、C 式内で 0xF4240 として評価されます。

```
#define MAX 1000*1000L
```

5.13.2 定義済みマクロ

コンパイラ ドライバは、プリプロセッサに対して特定のシンボルを定義する事で、デバイスのタイプ等に応じた条件付きコンパイルを可能にします。表 5-15 に示すシンボルは、ドライバによって定義される比較的一般的なシンボルです。

各シンボルの定義時の値は、明記がない限り 1 です。

表 5-15. 定義済みマクロ

シンボル	設定条件
ERRATA_4000_BOUNDARY	ERRATA_4000 を適用する場合
HI_TECH_C	入力 C ソースが HI-TECH-C 互換記述の場合
MPLAB_ICD	PIC18 以外のデバイスで MPLAB ICD デバッガ向けにビルドする場合 (値は MPLAB ICD 2 に対して 2、MPLAB ICD 3 に対して 3)
_CHIPNAME	特定のデバイスタイプ(例: _16F877)が選択された場合
BANKBITS	PIC18 以外のデバイス向けにビルドする場合 (値 0/1/2 は、利用可能バンクまたは RAM の数 1/2/4 を示す)
BANKCOUNT	PIC18 以外のデバイス向けにビルドする場合 (実装されているデータメモリのバンク数を示す)
COMMON	共通 RAM が存在する場合
_COMMON_ADDR_	共通メモリが存在する場合(共通メモリの開始アドレスを示す)
_COMMON_SIZE_	共通メモリが存在する場合(共通メモリのサイズを示す)
_EEPROMSIZE	非 PIC18 デバイス向けにビルドする場合(EEPROM の利用可能バイト数を示す)
_EEPROM_INT	PIC18 以外のデバイス向けにビルドする場合 (デバイスが EEPROM にアクセスするために NVMREG 経由でアクセスする場合は値 2 (_NVMREG_INT)を、EEREG 経由でアクセスする場合は値 1 (_REG_INT)を、レジスタインターフェイスを使わない場合は値 0 (_NOREG_INT)を持つ)
_ERRATA_TYPES	エラッタ回避策が適用される場合 (-merrata オプション(4.6.1.10 「errata オプション」) 参照)
_*FAMILY*_FAMILY_	PIC18 デバイス向けにビルドする場合 (コンパイラの pic/dat/ini ディレクトリに保存されている対応する.ini ファイル内の FAMILY フィールドによって示される PIC18 ファミリを示す)
_FLASH_ERASE_SIZE	常時 (フラッシュ プログラムメモリの消去ブロックのサイズを示す。フラッシュ データメモリを表すのではない)
_FLASH_WRITE_SIZE	常時 (フラッシュ プログラムメモリの書き込みブロックのサイズを示す。フラッシュ データメモリを表すのではない)
GPRBITS	非 PIC18 デバイス向けにビルドする場合 (値 0/1/2 は、利用可能バンクまたは汎用 RAM の数 1/2/4 を示す)
GPRCOUNT	非 PIC18 デバイス向けにビルドする場合 (値は汎用 RAM を格納したバンクの数を示す)
_HAS_FUNCTIONLEVELPROF_	-finstrument-functions が指定され、ターゲットがプロファイリングをサポートする場合
_HAS_INT24	常時
_HAS_OSCVAL_	ターゲット デバイスがオシレータ校正レジスタを備えている場合
MPC	Microchip PIC MCU ファミリ向けにコンパイルする場合
_OMNI_CODE_	OCG コンパイラを使ってコンパイルする場合
_PIC12	ベースライン デバイス(12 ビット命令) 向けにビルドする場合

.....続き

シンボル	設定条件
_PIC12E	拡張ベースライン デバイス(12 ビット命令)向けにビルドする場合
_PIC12IE	割り込みを備えた拡張ベースライン デバイス向けにビルドする場合
_PIC14	ミッドレンジ デバイス(14 ビット命令) 向けにビルドする場合
_PIC14E	拡張ミッドレンジ デバイス(14 ビット命令) 向けにビルドする場合
_PIC14EX	バンクが拡張された拡張ミッドレンジ デバイス(14 ビット命令) 向けにビルドする場合
_PIC18	PIC18 デバイス(16 ビット命令) 向けにビルドする場合
PROGMEM	フラッシュメモリを備えたミッドレンジ デバイス向けにビルドする場合 ターゲット デバイスが備えるフラッシュメモリのタイプを以下の通りに示す: 0xFF: 未知 0xF0: なし 0: 読み出し専用 1: ワード書き込み、消去は自動 2: ブロック書き込み、消去は自動 3: ブロック書き込み、消去は手動
_RAMSIZE	PIC18 デバイス向けにビルドする場合 (データメモリの利用可能バイト数を示す)
_ROMSIZE	常時 (利用可能なプログラムメモリの量を示す。PIC18 デバイスではバイト単位、その他のデバイスではワード単位)
__CHIPNAME と __CHIPNAME__	特定のデバイスタイプ(16F877 等)が選択された場合
__CLANG__	clang フロントエンドが使われる場合(-std=c99)
__CODECOV	常時 (値は CC_NONE に対して 0、CC_RAM に対して 1)
__DATABANK	EEPROM またはフラッシュメモリが実装されている場合 (どのバンクに EEDATA/PMDATA レジスタが格納されているかを示す)
__DATE__	常時 (現在の日付を文字列リテラル(例: "May 21 2004")として示す)
__DEBUG	MPLAB X IDE を使ってデバッグビルドを実行する場合
__EXTMEM	デバイスが外部メモリを備えている場合 (外部メモリのサイズを示す)
__FILE__	常時 (ソースファイルが前処理される事を示す)
__FLASHTYPE	フラッシュメモリを備えたデバイス(PIC18 を除く) 向けにビルドする場合 (ターゲット デバイスが備えるフラッシュメモリのタイプを示す。下の_PROGMEM 参照)
__LINE__	常時 (このソース行番号を示す)
__J_PART	PIC18 「J」シリーズデバイス向けにビルドする場合
__MPLAB_ICDX__	ICD デバッガ向けにコンパイルする場合 (*X* の値 2/3/4 はそれぞれ Microchip 社の MPLAB ICD 2/ ICD 3/ ICD 4 を示す)
__MPLAB_ICE__	任意のインサーキット エミュレータを示す
__MPLAB_PICKIT*X*__	PICkit™向けにコンパイルする場合 (*X* の値 2/34 はそれぞれ Microchip 社の MPLAB PICkit 2/ PICkit 3/ PICkit 4 を示す)
__MPLAB_REALICE__	Microchip 社の MPLAB REAL ICE™インサーキット エミュレータ向けにコンパイルする場合
__MPLAB_SNAP__	Microchip 社の MPLAB Snap インサーキット デバッガ向けにコンパイルする場合

.....続き

シンボル	設定条件
__OPTIMIZE_SPEED__	速度重視の最適化を使う場合
__OPTIMIZE_SPACE__と __OPTIMIZE_SIZE__	サイズ重視の最適化を使う場合
__OPTIMIZE_NONE__	最適化をどれも有効にしない場合
__OPTIM_FLAGS	常時 (最適化が有効である事を示す(この表の下の説明参照))
__PICCPRO__と __PICC__	PIC10/12/14/16 デバイス向けにビルドする場合
__PICC18__	C18 互換モードではない場合
__RESETBITS_ADDR	STATUS レジスタが保存される場合 (このレジスタの保存先のアドレスを示す)
__SIZEOF_TYPE__	常時 (指定されたタイプ(例: __SIZEOF_INT__または__SIZEOF__INT24__) のサイズをバイト数で示す)
__STACK	常時 (グローバル スタック設定を示す: __STACK_COMPILED (0x1) = コンパイルド スタック、__STACK_HYBRID (0x2) = ハイブリッド スタック、__STACK_REENTRANT (0x4) = ソフトウェア スタック)
__STRICT	-Wpedantic オプションが有効な場合
__TIME__	常時 (現在の時刻を文字列リテラル(例: “08:06:31”) として示す)
__TRADITIONAL18__	PIC18 デバイス向けにビルドする場合 (非拡張命令セットが選択される事を示す)
__XC	常時 (Microchip 社向け MPLAB XC コンパイラが使われる事を示す)
__XC8	常時 (Microchip 社の 8 ビットデバイス向け MPLAB XC コンパイラが使われる事を示す)
__XC8_VERSION	常時 (コンパイラのバージョン番号に 1000 を乗算した値を示す(例: v1.00 は 1000 として示される))

表 5-16 に示すコンパイラ定義マクロをビット幅 AND マスクとして使う事により__OPTIM_FLAGS が保持している最適化タイプが分かります。

表 5-16. 最適化フラグ

マクロ	値	意味
__OPTIM_NONE	0x0	最適化を何も適用しない (ソースコード通りの生成)
__OPTIM_ASM	0x1	C コードに対するアセンブラ最適化
__OPTIM_ASMFILE	0x2	アセンブリ ソースコードに対するアセンブラ最適化
__OPTIM_SPEED	0x20000	速度の最適化
__OPTIM_SPACE	0x40000	サイズの最適化
__OPTIM_SIZE	0x40000	サイズの最適化
__OPTIM_DEBUG	0x80000	正確なデバッグのための最適化
__OPTIM_LOCAL	0x200000	ローカル最適化

5.13.3 プラグマ ディレクティブ

特定のコンパイル時ディレクティブを使う事により、コンパイラの挙動を変更できます。これらは C 標準の #pragma 機能を使って実装されます。このプラグマの書式は以下の通りです。

```
#pragma directive options
```

directive はキーワードセットの中の 1 つを指定します。一部のキーワードには options を続けて指定します。表 5-17 にキーワードの一覧を示します。他の場所で説明されていないキーワードについては、以下で説明します。

表 5-17. プラグマ ディレクティブ

ディレクティブ	意味	例
addrqual	修飾子に対するコンパイラの対応を指定	#pragma addrqual require
config	コンフィグレーション ビットの指定	#pragma config WDT=ON
inline	可能であれば関数をインライン化する	#pragma inline(getPort)
intrinsic	関数がインラインである事を指定	#pragma intrinsic(_delay)
interrupt_level	割り込みコードとメインライン コードからの呼び出しを許可	#pragma interrupt_level 1
pack	構造体パッキング(アライメント)指定	#pragma pack 1
printf_check	printf 形式の書式文字列チェックの有効化	#pragma printf_check(printf) const
psect	コンパイラが生成する psect の名前変更	#pragma psect nvBANK0=my_nvram
regsused	関数によって使われるレジスタの指定	#pragma regsused myFunc wreg,fsr
switch	switch 命令文向けコード生成の指定	#pragma switch direct
warning	メッセージ パラメータの制御	#pragma warning disable 299,407

5.13.3.1 #pragma addrqual ディレクティブ

このディレクティブを使うと、非標準メモリ修飾子に対するコンパイラの対応を制御できます。このプラグラマは -maddrqual オプションと等価であり、どちらも同じ引数を使います(4.6.1.1 「addrqual オプション」参照)。

このプラグマは C プログラムの全体に影響を及ぼします。このプラグマが必要な場合、1 度だけ発行します。複数回発行した場合、最後のプラグマによってコンパイラの対応が決まります。

例:

```
#pragma addrqual require
__bank(2) int foobar;
```

5.13.3.2 #pragma config ディレクティブ

このディレクティブを使うと、デバイス コンフィグレーション ビットを PIC18 ターゲットデバイス向けに指定できます。詳細は 5.2.5 「コンフィグレーション ビットへのアクセス」を参照してください。

5.13.3.3 #pragma inline ディレクティブ

#pragma inline ディレクティブは、指定された関数に対する呼び出しを可能な限り高速にするようコンパイラに指示します。このプラグマは、inline 関数指定子と同じ効果を有します。

5.13.3.4 #pragma intrinsic ディレクティブ

#pragma intrinsic ディレクティブは、その関数が本来コンパイラによってインライン化される関数であるという事をコンパイラに示すために使います。このディレクティブをユーザ定義コードで使ってはいけません。

intrinsic プラグマを使う既存のライブラリ関数の再定義を試みてはいけません。そのようなコードの独自のバージョンを開発する必要がある場合、このプラグマを使っている元の関数と同じ名前を使ってはいけません。
例えば memcpy( )の独自バージョンを開発する必要がある場合、異なる名前(例: sp_memcpy( ))を与える必要があります。どの関数がこのプラグマを使っているかは、標準ヘッダファイルで確認できます。

5.13.3.5 #pragma interrupt_level ディレクティブ

`#pragma interrupt_level` ディレクティブを使うと、メインライン コードと割り込みコードから呼び出される関数の複製を防ぐ事ができます(5.8.7.1 「複製の無効化」参照)。

5.13.3.6 #pragma pack ディレクティブ

全ての 8 ビット PIC デバイスは、メモリに対してバイトアクセスしかできないため、構造体内でメモリ オブジェクトを整列させる必要はありません。このプラグマを使っても、何の効果も得られません。

5.13.3.7 #pragma printf_check ディレクティブ

特定のライブラリ関数は、`printf()`形式の書式文字列とそれに続く可変数の引数を受け付けます。書式文字列は実行時に解釈されますが、コンパイル時に引数との整合性をチェックできます。

このディレクティブは、指定された関数に対するこのチェックを有効にします。例えば、システム ヘッダファイル `<stdio.h>` は以下のディレクティブを使って、`printf()`に対するこのチェックを有効にします。

```
#pragma printf_check(printf) const
```

このディレクティブは、printf 形式の書式文字列を受け付けるユーザ定義関数向けにも使えます。

関数名の後の修飾子により、可変引数リスト内のパラメータを自動的に変換できます。上の例は、RAM 内の文字列に対する全てのポインタを、`const char *`型のポインタへと変換します。

このオプションの効果が見えるようにするには、警告レベルを-1 以下に設定する必要があります(4.6.4.2 「warn オプション」参照)。

5.13.3.8 #pragma psect ディレクティブ

`#pragma psect` はオブジェクトと関数を新しい psect (セクション) へとリダイレクトするために使われましたが、`__section()`指定子に置き換えられました(5.14.3 「割り当てられたセクションの変更とリンク」参照)。

この指定子は`#pragma psect` ディレクティブと同じタスクを実行しますが、より容易に使える上に、psect のリンク先に関する制約も緩和されます。常に`__section()`指定子を使って変数と関数を固有の psect に配置する事を推奨します。

このプラグマの一般的な形態は以下の通りです。

```
#pragma psect standardPsect=newPsect
```

これにより、通常は標準 psect (`standardPsect`) に格納される内容が `newPsect` という名前の新しい psect に格納されます。この新しい psect のフラグと属性は `standardPsect` と同じですが、名前が異なります。従って、オリジナルの psect の内容の配置に影響を与える事なく、この新しい psect を明示的に配置できます。

リダイレクトされる標準 psect の名前に番号が含まれる場合(例: `text0`、`text1`、`text2`)、psect 名の中の番号の位置でプレースホルダ%%u を使う必要があります(例: `text%%u`)。

5.13.3.9 #pragma regsused ディレクティブ

`#pragma regsused` ディレクティブを使うと、コード ジェネレータからは見えない(例えばアセンブリコードで書かれた) 関数向けにレジスタの使用を指定できます。このディレクティブは、C コード内で定義された関数に対して効果を有しません(そのような関数のレジスタ使用はコンパイラによって正確に特定できるため、このプラグマを使う必要はありません)。このプラグマは、レジスタをシャドーレジスタに保存できるデバイスに対しても効果を有しません。

このプラグマの一般的な形態は以下の通りです。

```
#pragma regsused routineName registerList
```

`routineName`: レジスタの使用を定義する関数またはルーチンの C 等価名
`registerList`: スペースで区切ったレジスタ名のリスト(表 5-10 参照)

リストに含まれないレジスタは、関数またはルーチンで使われないと見なされます。コードジェネレータは関数呼び出しの間もそれらのレジスタの値は保存されていると見なします。実際の関数やルーチンがこれらリスト外のレジスタを使ってしまうと、プログラム実行の信頼性が低下する可能性があります。

レジスタの名前では大文字と小文字は区別されません。レジスタ名が認識されない場合、警告が出力されます。レジスタを何も指定しないと、指定された関数またはルーチンはレジスタを一切使いません。このプラグマが使われない場合、コンパイラは全てのレジスタは外部関数によって使われると見なします。

例えば、_search という名前のルーチンが PIC18 向けアセンブリコードで書かれている場合、C ソース内で以下を書く事により、

```
extern void search(void);
#pragma regsused search wreg status fsr0
```

このルーチンが W レジスタ、STATUS レジスタ、FSR0 レジスタを使うという事を示します。fsr0 は、FSR0L と FSR0H の両方を指定します。必要に応じて、これらを別々に指定する事もできます。

コンパイラは、どのレジスタとオブジェクトが割り込み関数によって保存される必要があるのか判別できます。例えば割り込みルーチンがアセンブリ ルーチンを呼び出す場合、このプラグマを使ってコンテクスト スイッチコードをカスタマイズできます。

5.13.3.10 #pragma switch ディレクティブ

通常、コンパイラは switch 命令文をエンコードする際に、可能な限り小さなコード生成しようと試みます。#pragma switch ディレクティブを使うと、これとは異なるストラテジを使うようコンパイラに指示できます。

switch プラグマは、後続の全てのコードに影響を及ぼします。このプラグマの一般的な形態は以下の通りです。

```
#pragma switch switchType
```

switchType: 表 5-18 から 1 つを選択 (PIC18 デバイス向けには、現在「space」だけが使えます)

表 5-18. swithch のタイプ

タイプ	概要
speed	最速のスイッチ方式を使う
space	最小コードのスイッチ方式を使う
time	固定遅延のスイッチ方式を使う
auto	最小コードのスイッチ方式を使う(既定値)
direct (廃止)	固定遅延のスイッチ方式を使う
simple (廃止)	シーケンシャル XOR 方式を使う

#pragma switch ディレクティブに対して time オプションを指定した場合、コンパイラはテーブル索引型のスイッチ方式を採用します。この場合、各スイッチの実行時間は全て同じになるため、タイミングが重要な場合(例: ステートマシン) に適します。

auto オプションを使うと、既定値動作に戻す事ができます。

アセンブリ リストファイルには、各 switch 命令文に関する情報(選択されたストラテジ) が示されます(6.3.4 「switch 文情報」参照)。

5.13.3.11 #pragma warning ディレクティブ

このプラグマを使うと、コンパイラの一部のメッセージ(警告、エラー等)を制御できます。clang フロントエンドを使って C99 プロジェクトをコンパイルする場合、このプラグマの効果は得られません。コンパイラが使用するメッセージ システムの完全な情報は、4.5 「コンパイラのメッセージ」を参照してください。

5.13.3.11.1 warning disable プラグマ

warning disable プラグマを使うと、一部の警告メッセージを無効にできます。

このプラグマは、パーサーまたはコード ジェネレータによって生成される警告(すなわち C コードに直接関係する警告) に対してのみ有効です。このプラグマの位置は、パーサーに対してのみ意味を持ちます。すなわち、パーサー警告番号は、警告の特定のインスタンスをターゲットとしてコードの 1 つのセクションに対してのみ無効にできます。コード ジェネレータによって生成される警告の特定のインスタンスを個々に制御する事はできません。このプラグマは、複数のモジュールにまたがる C プログラム全体のコンパイルが完了するまで効力を持ちます。

無効にされた警告は、warning push および warning pop プラグマを使って保存および復元できます。warning push および warning pop プラグマはネスト可能です。

下のサンプルコードは、通常であれば const オブジェクト アドレスが非 const オブジェクトに対するポインタに割り当てられた事に関連する警告(359)を生成します。

```
int readp(int * ip)
    { return *ip;
}
const int i = 'd';
int main(void) {
    unsigned char c;
#pragma warning disable 359
    readp(&i);
#pragma warning enable 359
}
```

これと同じ効果は、下のコードを使っても得られます。

```
#pragma warning push
#pragma warning disable 359
    readp(&i);
#pragma warning pop
```

この場合、メッセージ システムの状態は warning push プラグマによって保存されます。警告 359 が無効にされた後、この警告をトリガするソースコードの後で、メッセージ システムの状態は warning pop プラグマを使って復元されます。

5.13.3.11.2 warning error/warning プラグマ

一部のメッセージのタイプを変更する事ができます。

メッセージのタイプを変更するには、warning プラグマを使う必要があります。このプラグマは、パーサーまたはコード ジェネレータによって生成されるメッセージに対してのみ有効です。プラグマの位置は、パーサーに対してのみ意味を持ちます。すなわち、パーサー メッセージ番号は、メッセージの特定のインスタンスをターゲットとしてコードの 1 つのセクションに対してのみメッセージのタイプを変更できます。コード ジェネレータによって生成されるメッセージの特定のインスタンスを個々に制御する事はできません。このプラグマは、C プログラム全体のコンパイルが完了するまで効力を持ちます。

下の例は、前の例で生成される警告を、関数 main()内のインスタンス向けに、エラーに変換します。

```
int main(void)
    { unsigned char
    c;
    #pragma warning error 359
    readp(&i);
}
```

このコードをビルドすると、警告 359 はエラーとして出力されます。

5.14 プログラムのリンク

本コンパイラは、コンパイル シーケンスを早期に停止するよう要求されない限り、自動的にリンカを呼び出します。

リンカは、コマンドライン ドライバから取得したオプションを直接使って動作します。これらのオプションはデバイスのメモリと、メモリ内のセクションの配置方法を指定します。リンカスクリプトは使われません。

リンカは、コマンドライン ドライバから取得したオプションを使って動作し、メモリ領域とセクションの配置位置を指定するリンカスクリプトを使います。

ユーザはリンカに渡されるリンカ オプションを調整できます。しかし、そのような調整は特殊な状況でのみ必要になります(4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)。

リンカは、セクションとコード内の一部のオブジェクトに割り当てられたメモリの詳細を示すマップファイルを生成します。マップファイルはメモリに関する最良の情報源です。

関連リンク
7.3 「マップファイル」

5.14.1 コンパイラが生成する psect

コード ジェネレータは、コードとデータを標準名を持つ psect (またはセクション) 内に配置します。これらの psect は、その後に既定値のリンカオプションとして配置されます。リンカは、コンパイラが生成するこれらの psect をユーザが定義した psect と全く同様に扱います。PSECT アセンブラ ディレクティブ(6.1.9.36 「PSECT ディレクティブ」参照)を使う事により、アセンブリコード内で psect を作成できます。

psect 名は大文字と小文字を区別します。一部の psect には特殊な命名規則を使います(例: 非初期化オブジェクトを保持する bss psect 等)。非初期化オブジェクトが存在する場合、それらをバンク 0 データメモリ内に配置し、その他のオブジェクトをバンク 1 メモリに配置する事が必要になる場合があります。これら 2 つのオブジェクト グループは、リンカがそれらを別々に制御できるようにするために、異なる RAM バンク内に配置する必要があります。

psect 名の一般的な形態は以下の通りです。

```
[bit]psectBaseNameCLASS[div]
```

psectBaseName: psect のベース名(5.14.1.2 「データ空間 psect」に記載)
CLASS: その psect がリンクされるリンカクラス(5.14.2.2 「データメモリ クラス」参照) に由来する名前(例: BANK0)

psect が__bit 変数を保持する場合、接頭辞「bit」を使います。従って、例えば bssBANK0、bssBANK1、bitbssBANK0 等の名前を持つ psect が、初期化されない変数を保持するためにコンパイラによって定義されます。

注意: 同様に、__eeprom 修飾された変数は、例えば bssEEDATA または dataEEDATA という名前の psect を定義できます。クラス接尾辞「EEDATA」を使う全ての psect は HEX ファイル内に配置され、デバイスへのプログラミング時に EEPROM に書き込まれます。

data psect を予約済み領域の前後に分割する必要がある場合、psect 名に文字 l と h を(上記の*[div]*として) 使う事で、上位部と下位部を識別します。従って、分割された psect の名前として、例えば bssBANK0l および bssBANK0h 等が使われます。

プロジェクト内のオブジェクトまたはコードがどの psect で保持されるかは、アセンブリ リストファイルで調べる事ができます(6.3.1 「一般的な書式」参照)。

5.14.1.1 プログラム空間内の psect

以下の psect はプログラムメモリ内に配置されます。

checksum	この psect は、-mchecksum オプションを使って要求されたチェックサムの位置をマーキングするために使います。詳細は 4.6.1.2 「checksum オプション」を参照してください。チェックサムの値はリンカの実行後に加算されるため、この psect の内容はアセンブリ リストファイルで示されず、関連する情報もマップファイルで示されません。この psect に関連する既定値リンカオプションを変更してはいけません。
cinit	この psect は C 実行時起動コード内初期化処理によって使われます。この psect 内のコードは、コード ジェネレータによって C プログラム向けに生成されたコードと一緒に出力されます。この psect は、他の psect の要求と干渉しない限り、1 つのプログラムメモリ ページ内の任意位置へリンクできます。
config	この psect は、コンフィグレーション ワードを保存するために使います。この psect は、HEX ファイル内の特定の位置に保存される必要があります。この psect に関連する既定値リンカオプションを変更してはいけません。
const	const として宣言されたオブジェクトおよび変更不可の文字列リテラルを保持するために使われる PIC18 専用の psect です。この psect は、プログラム内の const データの総量が 64kB を超える場合に使われます。この psect は、他の psect の要求と干渉しない限り、1 つのプログラムメモリ ページ内の任意位置へリンクできます。
eeprom (PIC18: eeprom_data)	この psect は、EEPROM メモリ内の初期値を保存するために使われます。この psect に関連する既定値リンカオプションを変更してはいけません。
end_init	この psect は、ベースラインおよびミッドレンジ デバイスの C 初期化実行時起動コードによって使われます。この psect は、制御を main()関数へ移すためのコードを保持します。

idata	この psect は、全ての初期化される変数の ROM イメージを格納します。これらの psect に関連付けられたクラス名は、これらの psect の内容が複製される先の RAM ベース データ psect のクラスを表します。これらの psect は、他の psect の要求と干渉しない限り、1 つのプログラムメモリ ページ内の任意アドレスでリンクできます。
idloc	この psect は、ID ロケーション ワードを保存するために使います。この psect は、HEX ファイル内の特定の位置に保存される必要があります。この psect に関連する既定値リンカオプションを変更してはいけません。
init	この psect は、実行時起動アセンブリ モジュール内のアセンブリコードによって使われます。`init` と `cinit` 内のコードにより、実行時起動コードが定義されます。割り込みコードが定義されていない場合、リセットベクタ コードはこの psect 内へ「フォールスルー」する可能性があります。この psect 向けのリンカオプションは変更しない事を推奨します。
intcode と **intcodelo**	これらの psect は、高優先度(既定値) および低優先度の割り込みサービスルーチン向けの実行可能コードを格納し、それぞれアドレス 0x8 および 0x18 で割り込みベクタへリンクされます。これらの psect に関連する既定値リンカオプションを変更してはいけません。ブートローダを使ってコードを移動させる場合、4.6.1.3 「codeoffset オプション」を参照してください。
intentry	この psect は、割り込みベクタへリンクされる割り込みサービスルーチン向けのエントリコードを格納します。ミッドレンジ デバイスの場合、このコードは必要なレジスタを保存し、C 実行時起動コード内初期化処理へジャンプします。拡張ミッドレンジ デバイスの場合、この psect は割り込み関数の本体を格納します。PIC18 デバイスは `intcode` psect を使います。 この psect は、割り込みベクタでリンクされる必要があります。この psect に関連する既定値リンカオプションを変更してはいけません。ブートローダを使ってコードを移動させる場合、`-mcodeoffset` オプション(4.6.1.3 「codeoffset オプション」)を参照してください。
ivt0xn	この psect は、割り込みベクタテーブルを使うデバイスまたはレガシーモードで動作するデバイス向けにアドレス *n* に配置されるベクタテーブルを格納します(5.8.1 「割り込みサービスルーチンの書き方」参照)。
jmp_tab	このベースライン専用 psect は、ジャンプアドレスと関数戻り値を保存するために使います。この psect に関連する既定値リンカオプションを変更してはいけません。
maintext	この psect は、`main()`関数向けのアセンブリコードを保持するために使います。`main()`向けのコードはプログラム エントリポイントを含むため、他のコードから分離されます。実行時起動コードはこの psect に「フォールスルー」する可能性があるため、この psect に関連する既定値リンカオプションを変更してはいけません。
mediumconst	`const` 宣言されたオブジェクトと文字列リテラルを保持するために使う PIC18 専用 psect です。この psect は、プログラム内の `const` データの総量が 255 バイトを超え、かつ 64kB 未満である場合に使われます。この psect は、他の psect の要求と干渉しない限り、プログラムメモリの下位 64kB 内の任意位置へリンクできます。PIC18 デバイスの場合、psect は最高 RAM アドレスよりも上のアドレスに配置される必要があります。
powerup	この psect は、ユーザが提供する起動ルーチン向けの実行可能コードを格納します。 この psect に関連する既定値リンカオプションを変更してはいけません。
reset_vec	この psect は、リセットベクタに関連するコードを保持するために使います。 この psect に関連する既定値リンカオプションを変更してはいけません。ブートローダを使う場合のコードの移動に関しては、`-mcodeoffset` オプション(4.6.1.3 「codeoffset オプション」)を参照してください。
reset_wrap	プログラムメモリの先頭位置にオシレータ校正値がロードされた後に実行されるコードを格納するベースライン デバイス向け psect です。 この psect に関連する既定値リンカオプションを変更してはいけません。
smallconst	この psect は、`const` が宣言されたオブジェクトと文字列リテラルを保持します。この psect は、プログラム内の `const` データの総量が 255 バイトを超える場合に使われます。 この psect は、0x100 境界をまたがない限り、また他の psect の要求と干渉しない限り、プログラムメモリ内の任意位置へリンクできます。PIC18 デバイスの場合、この psect は最高 RAM アドレスよりも上のアドレスに配置される必要があります。

strings この psect は、const オブジェクト向けに使います。
この psect は、全ての無名文字列リテラルも格納します。
この psect は、0x100 境界をまたがない限り、また他の psect の要求と干渉しない限り、プログラムメモリ内の任意位置へリンクできます。

stringtext*n* stringtext*n* psect (*n* は 10 進数) は、拡張ミッドレンジ デバイスで const オブジェクト向けに使います。これらの psect は、他の psect の要求と干渉しない限り、プログラムメモリ内の任意位置へリンクできます。

textn これらの psect (*n* は 10 進数) は、特別なリンク位置を要求しないその他全ての実行可能コードを格納します。これらの psect は、他の psect の要求と干渉しない限り、プログラムメモリ ページ内の任意位置へリンクできます。一部の PIC18 デバイスに対し、コンパイラはエラッタ回避策として疑似ページ境界を適用します。実行可能メモリ内のページの存在とそれらのサイズは、既定値の CODE リンカクラスで確認します。

temp この psect は、コンパイラが定義する一時変数を格納します。この psect は共通メモリ内にリンクする必要があり、他の psect の要求と干渉しない限り、このメモリ内の任意位置にリンクできます。

***xxx*_text** 絶対アドレス指定された関数向けの psect です。*xxx* は、その関数に対応するアセンブリ シンボルです。例えば関数 rv( )が絶対アドレス指定された場合、この関数に関連するコードは _rv_text という名前の psect に格納されます。
これらの psect は C ソースコード内で指定されたアドレスに配置済みであるため、これらを配置するためのリンカオプションを変更してはいけません。

***xxx*_const** 絶対アドレス指定された const オブジェクト向けの psect です。
xxx は、そのオブジェクトに対応するアセンブリ シンボルです。例えば配列 nba が絶対アドレス指定された場合、この配列内の値は_nba_const という名前の psect に格納されます。これらの psect は C ソースコード内で指定されたアドレスに配置済みであるため、これらを配置するためのリンカオプションを変更してはいけません。

5.14.1.2 データ空間内の psect

以下の psect はデータメモリ内に配置されます。

nv これらの psect は、__persistent で修飾された変数を保存するために使います。それらの変数は、起動時にクリアも変更もされません。これらの psect は配置先の RAM バンク内の任意位置にリンクできますが、デバイスがサポートする共通(非バンク型)メモリとオーバーラップしてはいけません。

bss これらの psect は、初期化されない変数を格納します。
これらの psect は配置先の RAM バンク内の任意位置にリンクできますが、デバイスがサポートする共通(非バンク型)メモリとオーバーラップしてはいけません。

data これらの psect は、初期化される変数の RAM イメージを格納します。
これらの psect は配置先の RAM バンク内の任意位置にリンクできますが、デバイスがサポートする共通(非バンク型)メモリとオーバーラップしてはいけません。

cstack これらの psect は、コンパイルド スタックを格納します。スタックにはプログラム全体の auto およびパラメータ変数が格納されます。これらの psect は配置先の RAM バンク内の任意位置にリンクできますが、デバイスがサポートする共通(非バンク型)メモリとオーバーラップしてはいけません。

stack この psect は、ソフトウェア スタック向けのプレースホルダとして使います。このスタックは動的であり、コンパイラはそのサイズを知る事ができません。5.2.4.2 「データスタック」で説明した通り、この psect はスタックが可能な限り大きくなれるように、残された空きデータ空間に割り当てられます。この psect はデータメモリ内の任意位置にリンクできますが、この psect の既定値リンカオプションを調整すると、ソフトウェア スタックのサイズが制限されてしまう可能性があります。ソフトウェア スタックがオーバーフローすると異常動作が生じる可能性があります。

5.14.2 既定値リンカクラス

リンカは、psect がリンク可能なメモリ領域を表すためにクラスを使います。

クラスはリンカオプション(7.1.1 「A: リンカクラスの定義」参照) によって定義されます。コンパイラ ドライバは、選択されたターゲット デバイスに基づいて、そのようなオプションの既定値設定をリンカに渡します。リンカクラスの名前は大文字と小文字を区別します。

一般的に psect には、そのクラスのメモリ領域から空きメモリが割り当てられます。psect に対するクラスの指定は、PSECT ディレクティブで class フラグを使って行います(6.1.9.36.3 「class フラグ」参照)。別の方法として、リンカオプションを使って psect をそのクラスに対応したメモリ位置に明示的に配置する事もできます(7.1.17 「P: psect を配置する」参照)。

クラスは 1 つまたは複数のメモリ領域を表す事ができます。2 つのクラスが連続している場合でも、1 つ目の終了アドレスと 2 つ目の開始アドレスの間にはクラス境界が形成され、それぞれの psect 間においてクラス境界を越える事はできません。同じアドレスをカバーする複数クラスは作成可能ですが、異なる範囲に分割され、異なる境界を持つとされます。これにより、コンテンツがメモリ内のどこにあるか、またはアクセスするデータがどこにあるかにより対応する psect が分かります。1 つのクラスから割り当てられたメモリは、同じメモリアドレスを指定する他のクラスからも予約されます。

リンカにとっては、クラス名やそれが定義するメモリには意味がありません。

-mreserve オプション(4.6.1.18 「reserve オプション」参照)を使うか、-mram および-mrom オプション(4.6.1.17 「ram オプション」および 4.6.1.20 「rom オプション」参照)を使ってメモリレンジを予約する事で、これらのクラスから特定のメモリ領域を除外できます。

クラスのメモリ領域からメモリを予約する以外は、クラスによって指定されたアドレス境界を変更または削除してはいけません。

5.14.2.1 プログラムメモリ クラス

コンパイラによって定義される以下のリンカクラスは、プログラムメモリ空間内の領域を表します。デバイスによっては、存在しないクラスがあります。

CODE ターゲット デバイス上のプログラムメモリ ページへマッピングされるレンジで構成され、実行可能コードを格納する psect 向けに使われます。
ベースライン デバイスの場合、ジャンプテーブル経由でアクセスされるコードだけがこのクラスを使えます。

ENTRY ベースライン デバイスで call 命令を介してアクセスされる実行可能コードを格納する psect 向けのクラスです。呼び出しは、ベースライン デバイス上のページの前半部に対してのみ可能です。
このクラスはページ全体にわたって定義されますが、このクラスが保持する psect はページの前半部で始まるように配置されます。
このクラスはミッドレンジ デバイスでも使われ、0x100 境界に整列した多数の 0x100 ワード長レンジを格納します。

STRING 0x100 境界に整列した 0x100 ワード長のレンジを格納します。このクラスは、内容が 0x100 ワード境界を越えない psect 向けに使えます。

STRCODE プログラムメモリ全体をカバーする単一メモリレンジを定義します。このクラスは、内容が任意のページに格納されページ境界をまたぐ可能性がある psect 向けに使えます。

CONST 0x100 境界に整列した 0x100 ワード長のレンジで構成されます。このクラスは、内容が 0x100 ワード境界を越えない psect 向けに使えます。

5.14.2.2 データメモリ クラス

コンパイラによって定義される以下のリンカクラスは、データ空間メモリ内の領域を表します。デバイスによっては、存在しないクラスがあります。

RAM ターゲット デバイスの全ての汎用 RAM メモリをカバーするレンジ(共通(非バンク型)メモリは含まず)で構成されます。
このクラスは、汎用 RAM バンク内に配置する必要がある psect 向けに使えます。

BIGRAM 拡張ミッドレンジ デバイスのリニア データメモリまたは PIC18 デバイスで利用可能な全てのメモリ空間をカバーするように設計された単一のメモリレンジで構成されます。
このクラスは、内容がリニアアドレス指定を使ってアクセスされる psect または単一 RAM バンク内に格納する必要がない psect 向けに適しています。

ABS1 ターゲット デバイスの全ての汎用 RAM メモリ(共通(非バンク型)メモリを含む)をカバーするレンジで構成されます。
このクラスは、汎用 RAM 内に配置される必要がある(しかし任意のバンクまたは共通メモリ内に配置可能な) psect 向けに使えます。

BANK*x* (*x* はバンク番号) — 各バンク番号のクラスは、そのバンク内の汎用 RAM (共通(非バンク型)メモリを含まず)をカバーする単一レンジで構成されます。

COMMON 共通(非バンク型) RAM をカバーする単一メモリレンジで構成されます(共通(非バンク型) RAM を備える全てのミッドレンジ デバイス向け)

COMRAM 共通(非バンク型)RAM をカバーする単一メモリレンジで構成されます(共通(非バンク型) RAM を備える全ての PIC18 デバイス向け)

SFR*x* (*x* は番号) — 各番号のクラスは、バンク x 内の SFR メモリをカバーする単一のレンジで構成されます。これらのクラスは汎用の RAM ではないため、通常はユーザが指定する必要はありません。

5.14.2.3 その他のクラス

コンパイラによって定義される以下のリンカクラスは、特殊な用途向けのメモリ領域を表します。デバイスによっては、存在しないクラスがあります。

CONFIG コンフィグレーション ビットデータ向けに予約されたメモリをカバーする単一レンジで構成されます。このクラスは汎用 RAM を表さないため、通常ユーザが使う必要はありません。

IDLOC HEX ファイル内の ID ロケーション データ向けに予約されたメモリをカバーする単一レンジで構成されます。このクラスは汎用 RAM を表さないため、通常ユーザが使う必要はありません。

EEDATA ターゲット デバイスの EEPROM (存在する場合)をカバーする単一レンジで構成されます。このクラスは、EEPROM へ書き込まれるデータを格納する psect 向けに使います。

5.14.3 割り当てられたセクションの変更とリンク

`__section()`指定子を使うと、オブジェクトまたは関数をユーザ定義 psect (またはセクション) へリダイレクトさせる事ができます。

5.14.1 「コンパイラが生成する psect」には、オブジェクトとコードを保持するためにコンパイラが使う既定値セクションを記載しています。関数またはオブジェクトによって使われる既定値セクションは必要に応じて変更できます。例えば、オブジェクトが既存のコンパイラ機能では対応できない特殊なリンクを要求する場合、セクションを既定値から変更する必要があります。

オブジェクト向けに`__section()`指定子によって作成された新しいセクションにはどのリンカクラスも割り当てられません。加えて、コンパイラは新しいセクションの最終的な位置に関して何も仮定しません。`__section()`で指定されたオブジェクトは、任意の RAM バンクにリンクできます。通常、コンパイラは新しいセクションがどこに配置されるかリンカ次第であり、予め知る事はできません。そのため、この指定子を使わなかったオブジェクトは指定子を使ったオブジェクトに比べアクセスするコード効率が低下する場合があります。

新しいセクションは、他のオブジェクトがメモリに割り当てられた後にリンクされるため、`__section()`指定子を使うとメモリエラーが発生する可能性もあります。その場合、新しいセクション向けにメモリを予約する必要があります(4.6.1.18 「reserve オプション」参照)。

関数向けに`__section()`指定子によって作成された新しいセクションは、同じフラグを継承します。しかし、関数に関連するリンクの制約は比較的緩いため、これは生成されるコードに対して非常に小さな影響しか及ぼしません。新しいセクションの名前には、有効なアセンブリコード領域識別子を指定する必要があります。この名前には英数字とアンダースコア「_」だけが使えます。アセンブラ ディレクティブ、制御、ディレクティブ フラグと同じ名前を使う事はできません。新しいセクションに格納された実行可能コードをアセンブラによって最適化する必要がある場合、セクション名にサブストリング「text」を含める必要があります(例: `usb_text`)。そのような名前を持たないセクションは、アセンブラ オプティマイザによって変更されません。

`__section()`指定子を使うオブジェクトは、実行時起動コードによって通常の方法でクリアまたは初期化されます(4.3.2 「起動と初期化」参照)。コンパイラは、初期化オブジェクト向けに、初期値を格納した追加の新しいセクションを自動的に割り当てます。このセクションには、指定されたセクションの名前に接頭辞 `i` を付けた名前が与えられます。このセクションはプログラムメモリ内に保存される必要があり、リンカオプションを使って明示的に配置する事が必要になる場合があります。

既定値とは異なるセクションに割り当てられるオブジェクトと関数の例を以下に示します。

```
int __section("myData") foobar;
int __section("myCode") helper(int mode) { /* ...*/ }
```

通常、新しく作成したセクションはリンカオプションを使って明示的に配置します。例えば、前記で作成したセクションを配置する場合、以下のドライバ オプションが使えます。

```
-Wl,-PmyData=0200h
-Wl,-AMYCODE=50h-3ffh
-Wl,-PmyCode=MYCODE
```

これはセクション「myData」をアドレス 0x200 に配置し、セクション「myCode」をリンカクラス MYCODE によって表される0x50～0x3ff 内の任意位置に配置します。-Wlドライバ オプション(4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)を使って渡す事ができるリンカオプションについては、7.1 「動作」を参照してください。

ベースラインまたはミッドレンジ PIC デバイスで、プログラムメモリ内のメモリレンジ向けに新しいクラスを作成する場合、そのクラスがワードアドレス指定可能なメモリを定義するという事をリンカに知らせる必要があります。これには、リンカの-D オプション (クラスが持つデルタ(アドレス増分)値を示す)を使います。

例:

```
-Wl,-DMYCODE=2
```

このオプションを PIC18 向けに使ってはいけません(PIC18 はバイトアドレス指定可能なプログラムメモリを備えるため)。

新しいセクションで特別なフラグを設定する場合、ソースコード内で新しいセクションの定義を提供する必要があります。例えば、myCode セクションを 100h の整数倍のアドレスに配置する場合、ソースファイル内に以下のコードを書きます。

```
asm("PSECT mycode,reloc=100h");
int __section("myCode") helper(int mode) { /* ...*/ }
```

reloc、size、limit psect フラグ(6.1.9.36.16 「reloc フラグ」参照) は、この方法により再定義できます。再定義によってアセンブラ警告メッセージが出力される場合がありますが、この場合は無視して構いません。

5.14.4 ライブラリ モジュールの置き換え

ライブラリアン xc8-ar (8.1 「アーカイバ/ライブラリアン」参照)を必要とせずにライブラリ ルーチンをユーザ独自のルーチンに容易に置き換える事ができます。

プロジェクト内のソースファイルがライブラリ内のルーチンまたはオブジェクトと同名のルーチンまたはオブジェクトの定義を含んでいる場合、ライブラリ内の定義はソースファイルからの定義によって置き換えられます。なぜなら、リンカはライブラリ ファイルを検索する前に、全てのソース モジュールで定義をスキャンするからです。

この方法を使って C ライブラリ関数をアセンブリコードで書かれた等価な関数で置き換える事はできません。これが必要な場合、ライブラリ ファイルを編集または新規作成するために xc8-ar ライブラリアンを使う必要があります。

5.14.5 シグネチャの確認

シグネチャは関数の戻り値の型、関数のパラメータの数、関数の呼び出し方法に影響するその他の情報から計算される 16 ビット値です。このシグネチャは、C 関数が参照または定義される時に自動的に生成され、出力コード内に配置されます。シグネチャの不一致(すなわち関数の定義とその関数の呼び出され方が矛盾する事を示す)が見つかると、リンカによって報告されます。

アセンブリ ルーチンを書く必要がある場合、それと等価な C プロトタイプに対応したシグネチャをそのルーチンに含める事を推奨します。関数の正しいシグネチャを特定する最も容易な方法は、同じプロトタイプでダミーの C 関数を書き、-Wa,-a オプションを使ってアセンブリ リストファイルを調べる事です。

例えば、「_widget」という名前のアセンブリ ルーチンがあり、その等価 C プロトタイプが char 型の引数を受け取り、char 型の値を返す場合を想定します。そのような関数の正しいシグネチャ値を見つけるには、下記のようなテスト用関数を書き、テストプログラムの一部としてこの関数をビルドした後に、アセンブリ リストファイルを確認します。

```
char myTestFunction(char arg1) {
}
```

アセンブリ リストファイル内に示されるアセンブリコードは、シグネチャ値が 4217 である事を示す以下の行を含みます。

```
SIGNAT  _myTestFunction,4217
```

_widget を含むアセンブリ ソースコードに、同様の SIGNAT ディレクティブを含めます。

C ソースファイル内でこのアセンブリ ルーチンを以下のように誤って宣言した場合、

```
extern char widget(long);
```

この宣言によって生成されるシグネチャはアセンブリ ルーチン内で指定されたシグネチャと異なるため、リンカによりエラーが出力されます。

5.14.6 リンカによって定義されるシンボル

リンカは、オプションによって明示的にリンクされたセクションがメモリ内のどこに配置されたのかを特定するために使う特殊なシンボルを定義します。これらのシンボルは、必要に応じて C コードまたはアセンブリコード内で使えます。

セクションのリンクアドレスは、下記の名前を持つグローバル シンボルの値から得られます。

__L*name* (先頭に 2 個のアンダースコア)

name: セクションの名前

例えば__LbssBANK0 は、bssBANK0 セクションの下位境界です。セクションの最高アドレス(すなわちリンクアドレス + サイズ) は、下記のシンボルにより表されます。

__H*name*

セクションのロードアドレスとリンクアドレスが異なる場合、ロード開始アドレスはシンボル__B*name* によって表されますが、PIC デバイスではこれらはめったに使われません。

セクションがリンカクラスによって暗黙的にリンクされる場合(すなわちリンカ オプション-P (7.1.17 「P: psectを配置する」参照) によって明示的にメモリに配置されない場合)、そのセクションに対応する特殊シンボルにはアドレスが代入されず、値は 0 となります。-P リンカオプションを使ってセクションがリンクされた場合にのみ、アドレスがシンボルに代入されます。このオプションは、セクションをリンカクラス内の任意位置に単純に配置できます (例: -PmyConstData=CONST)。

アセンブリコードは、これらのシンボル(名前の先頭に 2 個のアンダースコアが付く事に注意)をグローバルに宣言する事により、これらのシンボルを使う事ができます。

例:

```
GLOBAL  __Lidata
```

C コードは、以下の通りにシンボルを宣言する事により、これらのシンボルを使う事ができます。

```
extern char * _Lidata;
```

C コード領域では先頭のアンダースコアは 1 個だけである事に注意してください。C コード内での識別子はアドレスを表すため、通常はポインタ型を選択します。

6. マクロアセンブラ

MPLAB XC8 C コンパイラは、全ての 8 ビット PIC デバイス向けソースファイルをサポートするアセンブラを備えています。このアセンブラは、プロジェクト内の全てのアセンブリ ソースファイル向けに、コンパイラ ドライバ xc8-cc によって自動的に実行されます。

以下では、アセンブリ ソースファイルまたは C コード内のインライン アセンブリ向けに本アセンブラがサポートするディレクティブ (アセンブラ疑似命令や制御文) について説明します。

「アセンブラ」は、人間が読めるニーモニックをマシン語コードに変換するためのツールを表す用語として一般的に使われますが、アセンブラが読み出すソースコードは「アセンブラ」と呼ばれる事もあれば「アセンブリ」と呼ばれる事もあります。「アセンブリ」の方が広く用いられるため、本書では言語を指す意味で「アセンブリ」を使います。従って本書では、アセンブリ言語(または単にアセンブリ)、アセンブリ リスティング、アセンブラ オプション、アセンブラ ディレクティブ、アセンブラ オプティマイザという用語を使います。

本章の主な内容は以下の通りです。

- MPLAB XC8 のアセンブリ言語
- アセンブリレベルの最適化
- アセンブリ リストファイル

6.1 MPLAB XC8 のアセンブリ言語

以下には、マクロアセンブラがサポートするソース言語に関する情報を記載しています。

全てのオペコード ニーモニックとオペランド構文は、ターゲット デバイスに応じて異なるため、デバイス データシートで確認する必要があります。以下では追加のニーモニック、命令セットからの逸脱、アセンブラ ディレクティブについて説明します。

コンパイラからの中間生成アセンブリ ソースコード、手書きアセンブリ ソースコード、手書きアセンブリ モジュール、C コード内のインライン アセンブリコードから同じアセンブラによってオブジェクトが生成されます。

6.1.1 アセンブリ命令からの逸脱

MPLAB XC8 アセンブラでは、Microchip 社データシートで指定されたアセンブリ言語とは少し異なる形式の言語を使う事ができます。こちらの形式の方が一般的に読みやすいですが、データシートで指定された形式も使えます。以下では、許容される命令仕様からの逸脱と、デバイス命令セットに追加して使える疑似命令について説明します。

これらは、C コード内のインライン アセンブリコードまたは手書きアセンブリ モジュール内で使えます。

6.1.1.1 デスティネーションとアクセス オペランド

バイト型ファイルレジスタ命令のデスティネーションを指定するために、表 6-1 に示す 2 通りのスタイルが使えます。「Wreg デスティネーション」では命令の結果が W レジスタに書き込まれ、「ファイルレジスタ デスティネーション」では命令のファイルレジスタ オペランドによって指定されたレジスタに結果が書き込まれます。通常このオペランドは、データシート内で,d によって表されます。

表 6-1. デスティネーション オペランドのスタイル

スタイル	Wreg デスティネーション	ファイルレジスタ デスティネーション
XC8	,w	,f
MPASM	,0	,1

例 (バンク選択とアドレスマスクを無視):

```
addwf    foo,w      ;add wreg to foo, leaving the result in wreg
addwf    foo,f      ;add wreg to foo, updating the content of foo
addwf    foo,0      ;add wreg to foo, leaving the result in wreg
addwf    foo,1      ;add wreg to foo, updating the content of foo
```

デスティネーション オペランドを必要とする命令に対しては、常にデスティネーション オペランドを指定する事を推奨します。デスティネーション オペランドを省略すると、デスティネーションはファイルレジスタであると見なされます。

PIC18 デバイスの場合、RAM アクセスビットを指定するために表 6-2 に示す 2 通りのスタイルのオペランドが使えます。「バンク型アクセス」の場合、命令内で指定されたファイルレジスタ アドレスは現在選択されているバンク内のオフセットを示します。「非バンク型アクセス」の場合、ファイルレジスタ アドレスはアクセスバンク(または共通メモリ) 内のオフセットを示します。

表 6-2. RAM アクセス オペランドのスタイル

スタイル	バンク型アクセス	非バンク型アクセス
XC8	,b	,c or ,a
MPASM	,1	,0

通常このオペランドは、データシート内で,a によって表されます。

命令オペランドの前に「c:」を付ける事で、そのアドレスがアクセスバンク内である事を示す事もできます。
例:

```
addwf    bar,f,c        ;add wreg to bar in common memory
addwf    bar,f,a        ;add wreg to bar in common memory
addwf    bar,1,0        ;add wreg to bar in common memory
addwf    bar,f,b        ;add wreg to bar in banked memory
addwf    bar,1,1        ;add wreg to bar in banked memory
btfsc    c:bar,3           ;test bit three in the common memory symbol bar
```

RAM アクセス オペランドまたは共通メモリ領域指定(C:)を常に指定する事を推奨します。これらが指定されない場合、命令アドレスは絶対アドレスでありアクセスバンクの上位側半分に位置する(すなわちアドレスはマスクされてはならない)と見なされ、命令はアクセスバンク RAM を使います。その他の全ての状況において、命令はバンク型メモリにアクセスします。

XC8 スタイルを使う場合、PIC18 命令に対してデスティネーション オペランドと RAM アクセス オペランドは任意の順番で指定できます。例えば、以下の命令は同じです。

```
addwf   foo,f,c
addwf   foo,c,f
```

1 つの命令内では、全てのオペランドに対して同じスタイルを使う必要があります(例: addwf bar,1,c は数値と文字の混在により違反)。また、プログラム全体を通して同じスタイルを使う事を推奨します。

例えば以下のサンプルコードでは、最初の命令は W レジスタの内容を絶対アドレスに書き込み、次の命令は識別子によって表されるアドレスに書き込みます。この例では、バンク選択とマスキングが使われています。これらの命令の PIC18 オペコード(foo に割り当てられたアドレスは 0x516 であり、bar に割り当てられたアドレスは 0x55 であると想定)が以下に示されています。

```
6EE5   movwf 0FE5h           ;write to access bank location 0xFE5
6E55   movwf bar,c           ;write to access bank location 0x55
0105   BANKSEL(foo)          ;set up BSR to access foo
6F16   movwf                 ;write to foo (banked)
6F16   movwf BANKMASK(foo)   ;defaults to banked access
```

最初の 2 つの命令オペコードの RAM アクセスビット (オペコードの bit 8) はクリアされますが、最後の 2 つの命令ではセットされる事に注意してください。

6.1.1.2 バンクとページの選択

BANKSEL( )疑似命令を使うと、指定されたオペランドのバンクを選択するための命令を生成できます。オペランドは、データメモリ内のオブジェクトのシンボルまたはアドレスである事が必要です。

ターゲット デバイスに応じて、生成されるコードは 1 つまたは複数のビット命令を使って適切なレジスタ内のビットをセット/クリアするか、movlb 命令を使います(拡張ミッドレンジまたは PIC18 デバイスの場合)。ミッドレンジまたはベースライン デバイスの場合、この疑似命令は複数の命令に展開される場合があるため、この疑似命令を btfsc および btfss 命令の直後で使ってはいけません。

例:

```
movlw 20
BANKSEL(_foobar)    ;select bank for next file instruction
movwf BANKMASK(_foobar) ;write data and mask address
```

同様に、PAGESEL( )疑似命令を使うと、アドレス オペランドのページを選択するためのコードを生成できます。現在のページを選択する場合、オペランドとしてロケーション カウンタが使えます。

ターゲット デバイスに応じて、生成されるコードは 1 つまたは複数の命令を使って適切なレジスタ内のビットをセット/クリアするか、movlp 命令を使います(拡張ミッドレンジ デバイスの場合)。このディレクティブは複数の命令に展開される可能性があるため、btfs*x* 命令の直後で使ってはいけません。

例:

```
fcall _getInput
PAGESEL $        ;select this page
```

PIC18 ターゲット向けにコンパイルする場合、このディレクティブは受け入れられますが、何の効果も持ちません(対応するコードは何も生成されません)。PIC18 は、8 ビットデバイス間の移植を容易にする事だけを目的に、このディレクティブをサポートします。

6.1.1.3 アドレスマスク

大部分の命令で使われるファイルレジスタ アドレスには、アドレスからバンク情報を除外するためにマスクを使う必要があります。これを正しく行わないと、リンカ フィックスアップ エラー(3.6.8 参照) が発生します。

全ての MPLAB XC8 アセンブリ識別子は、フルアドレスを表現します。このアドレスには、そのオブジェクトのバンク情報が含まれています。ファイルレジスタ オペランドを使う 8 ビット PIC 命令セットのほぼ全てが、このオペランド値が現在選択中のバンク内のオフセットである事を期待します。バンクのサイズはデバイスファミリに応じて異なるため、このオフセットの幅も異なります。

識別子またはアドレス オペランドには BANKMASK( )マクロが使えます。このマクロは、適切なマスクを使った AND 演算により、アドレス内のバンク情報を除外します。このマクロは、<xc.inc>をインクルードする事により利用できます。このマクロはターゲット デバイスのバンクサイズに応じてマスクを調整するため、このマクロを使うと Microchip 社製デバイス間のアセンブリコードの移植性が向上します。このマクロの使用例は 6.1.1.2 「バンクとページの選択」に記載しています。

オペランドがフルアドレスである事を期待する命令(例: PIC18 の movff 命令)でこのマクロを使ってはいけません。

6.1.1.4 movfw 疑似命令

MPASM によって実装されていた movfw 疑似命令は、MPLAB XC8 アセンブラには実装されていません。同じ機能を実行する標準 PIC 命令を代わりに使う必要があります。

以下の MPASM 命令:

```
movfw foobar
```

は、以下の標準 PIC 命令に直接対応します。

```
movf foobar,w
```

6.1.1.5 movff/movffl 命令

movff 命令は機種依存命令であり、拡張データメモリを備える PIC18 デバイス向けにのみ使えます。この命令は、movffl 命令の代用としても機能します。

それらのデバイスで movff 命令向けの出力を生成する場合、アセンブラはオペランド シンボルを保持している psect を確認します。ソースオペランドを保持する psect とデスティネーション オペランドを保持する psect の両方がフラグ lowdata を指定している場合、この命令は 2 ワードの movff 命令としてエンコードされます。1 つのオペランドが絶対アドレス指定であり、かつそのアドレスがメモリの下位 4kB 内にある場合も、2 ワードの短縮形式が使えると見なされます。その他の全ての状況では、この命令は 3 ワードの movffl 命令としてエンコードされます。

アセンブリ リストファイルには、この命令がどちらの形式でエンコードされたかに関係なく、常に movff ニーモニックが示されます。オペコードワードの数を確認する事で、どちらの形式でエンコードされたかが分かります。

6.1.1.6 割り込みからの戻りモード

PIC18 命令 retfie の直後に「f」(コンマなし)を指定する事で、シャドーレジスタからの読み値を対応するレジスタに複製するよう指示できます。この修飾子を付けないと、これらのレジスタはシャドーレジスタからの値によって更新されません。この構文は、ベースラインおよびミッドレンジ デバイスには対応しません。

以下の例に、この命令の両方の形式と、それらが生成するオペコードを示します。

```
0011  retfie f    ;shadow registers copied
0010  retfie      ;return without copy
```

デバイス データシートに記載されているオペランド「0」および「1」を代わりに使う事もできます。

6.1.1.7 ロングジャンプと呼び出し

アセンブラは、標準 PIC MCU アセンブリ命令へと展開されるニーモニックとして fcall と ljmp を認識します。

ベースラインおよびミッドレンジ デバイスの場合、これらの命令はそれぞれ標準の call および goto 命令へと展開されますが、デスティネーションがプログラムメモリの別のページ内である場合、ミッドレンジ デバイスの PCLATH レジスタまたはベースライン デバイスの STATUS レジスタ内のビットを設定するための命令も生成されます。ページ選択命令は call または goto 命令の直前に生成されるか、その前の fcall/ljmp ニーモニックの一部として(それらの直後に) 生成されます。

PIC18 デバイスの場合、これらのニーモニックは他の PIC デバイスとの互換性のためだけに使われ、常に標準の PIC18 call および goto 命令として展開されます。

アセンブリコードを最終的な実行位置に依存しないようにする場合、これらの特別なニーモニックを可能な限り使う必要があります。ページ選択命令が生成されるかどうかと、それらがどこに配置されるかは、前後のソースコードに依存します。call または goto 命令のデスティネーションが現在のページ内であると判定された場合、PCLATH ビットを設定するための追加コードは最適化によって削除される可能性があります。

fcall および ljmp ニーモニックに対するオペランドは、どのターゲット デバイスであってもマスクされない事が必要です。デスティネーション アドレスとページ (デバイスにページが存在する場合)を指定するためにフルアドレスが必要です。ニーモニックが展開されると、展開されたコード内の call または goto 命令で使われるアドレスは自動的にマスクされます。ソースコード内で call または goto 命令を直接使う場合、常にオペランドに対して適切なマスクを適用する必要があります。

ミッドレンジ デバイスでアセンブリ リストファイル内に示される fcall 命令の例を下に示します。fcall 命令は 5 つの命令に展開された事が分かります。この例では、PCLATH レジスタ内のビットをセット/クリアする 2 つのビット命令が使われています。このレジスタ内のビットは、fcall の前に選択されていたページを再選択するために、呼び出しの後にもセット/クリアされます。

```
 13 0079 3021                       movlw   33
 14 007A 120A   158A  2000          fcall   _phantom
        120A   118A
 15 007F 3400                       retlw   0
```

fcall および ljmp 命令は複数の命令へと展開される可能性があるため、これらの命令の直前にスキップする可能性のある命令(例: btfsc 命令)を使ってはいけません。

fcall および ljmp 命令は、それらを格納している psect は 1 ページよりも小さいと想定します。現在の psect が 1 ページよりも大きい場合、これらの命令を使って現在の psect 内のラベルへ制御を移動させてはいけません。既定値のリンカ オプションは、1 ページよりも大きなコード psect を許容しません。

PIC18 デバイスの場合、標準の call 命令の後に,f を指定する事により W、STATUS、BSR レジスタを対応するシャドーレジスタへプッシュするよう指定できます。これは、デバイス データシートに記載されている「,1」構文に置き換わります。

6.1.1.8 相対分岐

PIC18 デバイスは、条件付き相対分岐命令(例: bz、bnz)を実装しています。これらの命令では、goto 命令に比べてジャンプ先レンジが制限されます。

状況によっては、アセンブラは相対分岐命令を反対条件の相対分岐へ変更して goto 命令へ変換する場合があります。
例:

```
   bz error
   ;next
```

上記は下記に変換される場合があります。

```
   bnz l18
   goto error
 l18:
   ;next
```

この変換により、条件分岐が goto 命令と同じジャンプ先レンジを持てるようにします。

この変換は C コード内のインライン アセンブリコード、分離したモジュール内の手書きアセンブリコード、ASMOPT ディレクティブ(6.1.9.2 「ASMOPT ディレクティブ」参照) によって保護されたアセンブリソースには適用されません。

6.1.2 命令文の書式

有効な命令文書式を表 6-3 に示します。

label フィールドはオプションであり、指定する場合は 1 つの識別子を格納する必要があります。*label* は別の行に書くか(表内の書式 1)、ニーモニックの前に置く事ができます(表内の書式 2)。

書式 3 は、特定のアセンブラ ディレクティブ(MACRO、SET、EQU 等) 向けにのみ使えます。*name* フィールドは必須であり、1 つの識別子を格納する必要があります。

アセンブリ ソース ファイルが最初にプリプロセッサによって処理される場合、そのソースファイルにはプリプロセッサ ディレクティブを形成する行を含める事ができます。これらのディレクティブの詳細な書式については、5.13.1 「プリプロセッサ ディレクティブ」を参照してください。

以下に書式を示しますが、順序が保たれているなら、行内の桁位置には影響を受けません。

表 6-3. アセンブラ命令文の書式

書式	フィールド 1	フィールド 2	フィールド 3	フィールド 4
書式 1	*label*:			
書式 2	*label*:	*mnemonic*	*operands*	*; comment*
書式 3	name	*pseudo-op*	*operands*	*; comment*
書式 4	*; comment only*			
書式 5	空白行			

6.1.3 文字

標準の 7 ビット ASCII 文字セットを使います。識別子ではアルファベットの大文字と小文字が区別されますが、ニーモニックと予約済みワードでは区別されません。タブはスペースと等価です。

6.1.3.1 デリミタ

全ての数値と識別子はタブやスペース、英数字以外の文字、行末(EOL) 文字のいずれかによって区切る必要があります。

6.1.3.2 特殊文字

一部の文字は特殊な意味を持ちます。マクロボディ内では、文字「&」をトークンの連結用に使います。マクロボディ内でビット単位&演算子を使う場合、代わりに&&または and を使います。マクロ引数リスト内では、マクロ引数を山カッコ(<と>)で囲みます。

6.1.4 コメント

アセンブリコードのコメントは、文字列の一部でも文字定数でもないセミコロンで開始します。

例:

```
   movlw 22   ;this value will ensure there is a good safety margin
```

アセンブリ ファイルが最初に C プリプロセッサによって処理される場合、そのファイルには C または C++形式のコメント(標準の/* ... */ および//構文)を含める事ができます。

プリプロセッサ ディレクティブ(特に#define ディレクティブ) 内でアセンブリ コメント(;comment) の使用は避けます。C プリプロセッサはマクロ置換の前にアセンブリ コメントを削除しないため、それらのコメントは置換後のテキスト内に残され、ビルドエラーの原因となる可能性があります。これらの状況では、常に C または C++形式のコメントを使う必要があります。

6.1.4.1 特殊なコメント文字列

コード ジェネレータは、コンパイラが生成するアセンブリ命令に対して各種のコメント文字列を付加します。これらのコメントは、通常アセンブラ オプティマイザにより使われます。

コメント文字列 ;volatile は、命令内でアクセスされるメモリ位置が C ソースコード内で volatile として宣言された変数割り当てであるかのように振る舞います。このコメント文字列が存在する場合、この位置へのアクセスが冗長であるかのように見えても、アセンブラ オプティマイザはこのアクセス文を削除しません。

このコメント文字列を手書きアセンブリソース内で使う事で、アセンブリコード内で定義およびアクセスされる位置に対して同じ効果が得られます。

コメント文字列 ;wreg free は、一部の call 命令に付加されます。この文字列は、W レジスタに関数パラメータは書き込まれなかった(すなわち未使用である)という事を示します。この文字列が存在する場合、W レジスタを冗長にロードする関数呼び出しの前に、アセンブラ命令に対して最適化を行う事ができます。

6.1.5 定数

6.1.5.1 数値定数

アセンブラは符号付き 32 ビット精度で全ての算術演算を実行します。

全ての数値の既定値の基数は 10 です。数値の後に基数指定子を付ける事で、他の基数を指定できます(表 6-4 参照)。

表 6-4. 数値の基数

基数	書式
2 進数	数字 0 または 1 の後に B
8 進数	数字 0～7 の後に O、Q、o、q のいずれか
10 進数	数字 0～9 の後に D、d または指定子なし
16 進数	数字 0～9 およびアルファベット A～F の前に 0x、または後に H または h

識別子と区別するため、16 進数の先頭は数字である事が必要です(例: 0ffffh)。16 進値に使うアルファベットは大文字でも小文字でも構いません。

2 進定数の後には大文字の B が必要です。小文字の b は一時的な(数値)ラベルの逆参照用に使われます。

式内では、実数は通常の書式で受け入れられ、IEEE 32 ビット フォーマットとして解釈されます。

6.1.5.2 文字定数と文字列

文字定数は、シングルクオートで(')で囲まれた 1 個の文字を格納します。

複数文字の並びから成る文字定数(文字列)はキャリッジ リターンまたは改行文字を含まず、シングルクオート(')またはダブルクオート(") によって囲まれます(前後のクオートは同じである事が必要です)。

6.1.6 識別子

アセンブリ識別子は、メモリ位置または数値を表すユーザ定義シンボルです。シンボルには文字(アルファベット、数字、特殊文字($、?、_))を何個でも含める事ができます。

数字と$文字は、識別子の先頭文字として使えません。アルファベットの大文字と小文字は区別されます(例: Fred と fred は異なるシンボルです)。

識別子の例を以下に示します。

```
An_identifier
an_identifier
```

```
an_identifier1
?$_12345
```

アセンブリ コード ニーモニック(例: movlw、return)、アセンブラ ディレクティブ(例: SET、LIST)、ディレクティブ引数トークン(例: hex、push)、演算子(例: mod、nul)と同じシンボル(大文字/小文字の区別なし)を識別子として使ってはいけません。

1 個以上のアンダースコア文字で始まる識別子は C コードからアクセスされる可能性があります。そのようなシンボルは、C コード識別子と干渉しないよう注意する必要があります。アンダースコア文字で始まらない識別子は、アセンブリコード領域のみからアクセス可能です。C コード領域とアセンブリコード領域の間のシンボルの対応については、5.11.3.3 「等価アセンブリ シンボル」を参照してください。

6.1.6.1 識別子の意味

識別子に対して各種データ型の実装を試みる他のアセンブラとは異なり、本アセンブラはどのシンボルにも意味を付加せず、シンボルの用法に対して制限する事も期待する事もしないという事に注意する必要があります。

psect (プログラム セクション) の名前と通常のシンボルは分離され、オーバーラップも可能な名前空間を使いますが、本アセンブラはシンボルが何を表すために使われるのか一切気にしません。ビット型を含む全てのデータ型のアドレスを定義するために特殊な構文は不要であり、提供もされません。また、本アセンブラはシンボルが複数のコンテクストで使われても警告を発行しません。命令およびアドレスモード構文は、アセンブラが正しいコードを生成するために必要な全ての情報を提供します。

6.1.6.2 アセンブラが生成する識別子

LOCAL ディレクティブがマクロブロック内で使われる場合、そのマクロの各展開内で指定される各識別子を置き換えるために、アセンブラは一意のシンボルを生成します。これらの一意シンボルは??*nnnn* (*nnnn* は 4 桁の番号)の書式を持ちます。ユーザが同じ書式のシンボルを定義してはいけません。

6.1.6.3 ロケーション カウンタ

アクティブなプログラム セクション内の現在位置には、シンボル$を介してアクセスできます。このシンボルは現在実行中の命令のアドレスへと展開されます(このアドレスは、この命令の実行時にプログラム カウンタ(PC) に格納されているアドレスとは異なります)。

例:

```
goto $   ;endless loop
```

上記のコードは、そのコード自身へジャンプしてエンドレスループを形成します。シンボル$とオフセットを使う事により、相対ジャンプのデスティネーションを指定できます。

$に加算するアドレス オフセットの扱いは、ターゲット デバイスに応じて異なります。ベースラインおよびミッドレンジ デバイスの場合、プログラムメモリ位置はワードアドレスで指定されるため、このオフセットは現在位置からの命令数を表します。PIC18 デバイスの場合、プログラムメモリはバイトアドレスで指定されるため、このシンボルへのオフセットは現在位置からのバイト数を表します。PIC18 命令はワード境界に整列する必要があるため、ロケーション カウンタに対するオフセットは 2 の倍数である事が必要です。全てのオフセット値は、最も近い 2 の倍数に切り捨てられます。

例:

```
goto       $+2   ;skip...
movlw      8     ;to here for PIC18 devices, or
movwf       foo  ;to here for baseline and mid-range devices
```

ベースラインおよびミッドレンジ デバイスの場合、movlw 命令はスキップされます。PIC18 デバイスの場合、goto $+2 は次の命令へジャンプします(nop 命令であるかのように機能します)。

6.1.6.4 レジスタ シンボル

定義済みアセンブリ ヘッダファイルをインクルードする事により、アセンブリ モジュール内のコードから特殊機能レジスタにアクセスできます。以下の行をアセンブラ ソースファイルに追加する事により、適切なファイルをインクルードできます。

```
#include <xc.inc>
```

前処理を適用するため、このソースファイルの名前には拡張子.S を付ける必要があります。このヘッダファイルには、ターゲット デバイスに固有のヘッダファイルをソースファイルへインクルードするための適切なコマンドが含まれています。

これらのデバイス固有ヘッダファイルには、全てのバイトサイズまたはマルチバイト サイズ レジスタ向けの EQU 宣言と、バイトレジスタ内の名前付きビット向けの#define マクロが含まれています。

6.1.6.5 シンボリック ラベル

ラベルとは、現在の psect 内の現在のアドレス値が代入されるシンボリック エイリアスです。リンクが実行されるまで、ラベルに値は代入されません。

ラベル定義は、有効なアセンブリ識別子(後にコロン : が必要)で構成されます。定義は、独立した 1 行に書く事もできますが、命令またはアセンブラ ディレクティブの左側に書く事もできます。アセンブリコード内で 2 通りの方法で書かれたラベルの例を以下に示します。

```
frank:
  movlw   1
  goto        fin
simon44: clrf _input
```

ラベル frank には最終的に movlw 命令が代入され、simon44 には clrf 命令のアドレスが代入されます。それらがどのように定義されたかに関係なく、アセンブラによって生成されるアセンブリ リストファイルは常にラベルを単独で 1 行内に示します。

ラベルはアセンブリコード内で使えます。他の命令で絶対アドレスを使うよりも、ラベルを使う事を推奨します。そうする事で、ラベルをジャンプ型命令のターゲット位置として使う事ができます。あるいは、レジスタにアドレスをロードするために使う事もできます。

変数と同様に、ラベルにもスコープがあります。既定値により、ラベルは、それが定義されているモジュール内の任意位置で使えます。ラベルは、ラベルの定義よりも前に位置するコードから使う事もできます。他のモジュールからのラベルへのアクセスを可能にするには、GLOBAL ディレクティブ(6.1.9.21 「GLOBAL ディレクティブ」参照)を使います。

6.1.7 式と演算子

式には、単項演算子(オペランドは 1 つ、例: not)または二項演算子(オペランドは 2 つ、例: +)が使えます。式内で使える演算子を表 6-5 に示します。

式内の演算子は左から右へ向かって順番に評価されます。従って 5 + 1 * 2 の結果は 12 です。丸カッコを使って演算子の実行順を変更できます。例えば 5 + (1 * 2)の結果は 7 です。

等価演算子(例: =)および関係演算子(例: >=)を除き、表 6-5 内の全ての演算子は定数式および再配置可能式内で自由に組み合わせる事ができます。再配置可能式は、リンク時に(psect が配置されシンボル値が決定された後に)リンカによって評価されます。

表 6-5. アセンブリ演算子

演算子	目的	例
*	乗算	movlw 4*33,w
+	加算	bra $+1
-	減算	DB 5-2
/	除算	movlw 100/4
= または eq	等価	IF inp eq 66
> または gt	符号評価付き「大なり」	IF inp > 40
>= または ge	符号評価付き「以上」	IF inp ge 66
< または lt	符号評価付き「小なり」	IF inp < 40
<= または le	符号評価付き「以下」	IF inp le 66
<> または ne	符号評価付き「不等価」	IF inp <> 40
low	オペランドの下位バイト	movlw low(inp)
high	オペランドの上位バイト	movlw high(1008h)

.....続き

演算子	目的	例
highword	オペランドの上位 16 ビット	DW highword(inp)
mod	剰余	movlw 77mod4
& または and	ビット単位 AND	clrf inp&0ffh
^	ビット単位 XOR (排他的 OR)	movf inp^80,w
|	ビット単位 OR	movf inp|1,w
not	ビット単位補数	movlw not 055h,w
<< または shl	左シフト	DB inp>>8
>> または shr	右シフト	movlw inp shr 2,w
rol	左ローテート	DB inp rol 1
ror	右ローテート	DB inp ror 1
float24	実数オペランドの 24 ビット バージョン	DW float24(3.3)
nul	マクロ引数が Null かどうかのテスト	

6.1.8 プログラム セクション(psect)

プログラム セクション(psect) は、コードまたはデータのセクションです。psect により、プログラムの各部を 1 つのグループにまとめる事ができます。同じソースファイル内で離れた位置にあるコードや異なるモジュール内のコードも 1 つの psect にまとめる事ができます。

psect には識別するための名前と各種の属性が与えられます。psect の定義には PSECT アセンブラ ディレクティブを使います。このディレクティブには引数として名前を指定する必要があります。名前に続けて、psect の属性を定義するためにコンマで区切った引数のリストを指定できます。メモリ内の psect の配置を制御するためのリンカ オプションの説明は 7. 「リンカ」に記載しています。これらのオプションには、-Wl ドライバ オプション(4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)を使ってドライバからアクセスできます。

コード ジェネレータが定義する全ての psect のリストは 5.14.1 「コンパイラが生成する psect」に記載しています。

abs (絶対アドレス)として定義されない限り、psect は再配置可能です。

PSECT ディレクティブによって明示的に特定 psect 内に配置されなかったコードまたはデータは、既定値(無名)の psect 内に配置されます。ユーザがこの psect のリンク位置を制御する事はできないため、コードおよびオブジェクトの前に常に PSECT ディレクティブを配置する事を推奨します。

アセンブリコードを書く際は、コンパイラが生成する既存の psect またはユーザが独自に生成する psect が使えます。コンパイラが生成する psect を使う場合、リンカ オプションを調整する必要はありません。

ユーザ独自の psect を生成する場合、それらを既存のリンカクラスに割り当てる事を推奨します(5.14.2 「既定値リンカクラス」と 7.1.2 「C: psect に対するリンカクラスの割り当て」参照)。そうしない場合、それらを正しく配置するためにリンカ オプションの指定が必要になる可能性があります。

psect のサイズと配置は重要です。デバイスのページ境界をまたがない psect 内に全ての実行可能コードを配置する事でコードを書くのが容易となります。また、データ psect がバンク境界をまたがない場合も同様です。psect のリンク位置は、psect の内容にアクセスするアセンブリコードと一致している必要があります。

6.1.9 アセンブラ ディレクティブ

アセンブラ ディレクティブまたは疑似命令は、命令ニーモニックと同様の方法で使われます。PAGESEL と BANKSEL を除くこれらのディレクティブは命令を生成しません。DB、DW、DDW ディレクティブはデータバイトを現在の psect 内に配置します。アセンブラ ディレクティブの一覧を表 6-6 に示します。

表 6-6. アセンブラ ディレクティブ

ディレクティブ	目的
ALIGN	出力を指定された境界に整列させます。
ASMOPT	後続のコードをアセンブラによって最適化するかどうかを制御します。
BANKISEL	間接アクセス向けにオペランドのバンクを選択するためのコードを生成します。 (一部のデバイスのみ)
BANKSEL	オペランドのバンクを選択するためのコードを生成します。
CALLSTACK	コールスタックの残りの深さを示します。
[NO]COND	リスティング ファイル内の条件付きコードのインクルードを制御します。
CONFIG	コンフィグレーション ビットを指定します。
DABS	絶対アドレス記憶域を定義します。
DB	定数バイトを定義します。
DDW	2 倍幅(32 ビット) 定数ワードを定義します。
DEBUG_SOURCE	デバッグ情報を制御します。
DLABS	リニアメモリ絶対アドレス記憶域を定義します。
DS	記憶域を予約します。
DW	定数ワードを定義します。
ELSE	IF との組み合わせで条件付きアセンブリを実装します。
ELSIF	IF との組み合わせで条件付きアセンブリを実装します。
ENDIF	条件付きアセンブリを終了します。
END	アセンブリを終了します。
ENDM	マクロ定義を終了します。
EQU	シンボル値を定義します。
ERROR	ユーザ定義エラーを生成します。
[NO]EXPAND	リスティング ファイル内のアセンブラマクロの展開を制御します。
EXTRN	他のモジュール内で定義されたグローバル シンボルとリンクします。
FILE	後続のアセンブリコードを格納したソースファイルを示します。
GLOBAL	シンボルを他のモジュールからアクセスできるようにします。または、他のモジュール内で定義された他のグローバルシンボルへの参照を可能にします。
IF	条件付きアセンブリを実装します。
INCLUDE	指定されたファイルの内容をテキストとしてインクルードします。
IRP	リストを使ってコードのブロックを繰り返します。
IRPC	文字リストを使ってコードのブロックを繰り返します。
LINE	後続のアセンブリコードを格納した現在のソースファイルの行番号を示します。
[NO]LIST	リスティング ファイルのオプションを定義します。
LOCAL	ローカルタブを定義します。
MACRO	マクロを定義します。
MESSG	ユーザ定義アドバイザリ メッセージを生成します。

.....続き

ディレクティブ	目的
ORG	現在の psect 内のロケーション カウンタを設定します。
PAGELEN	リスティング ファイルページの長さを指定します。
PAGESEL	このページ向けに PCLATH ビットを設定するためのセット/クリア命令を生成します。
PAGEWIDTH	リスティング ファイルページの幅を指定します。
PROCESSOR	このファイルをどのデバイス向けにアセンブルするのか定義します。
PSECT	psect を宣言または再開します。
RADIX	数値定数の基数を指定します。
REPT	コードのブロックを n 回繰り返します。
SET	シンボル値を定義または再定義します。
SIGNAT	関数シグネチャを定義します。
SUBTITLE	リスティング ファイル向けにプログラムのサブタイトルを指定します。
TITLE	リスティング ファイル向けにプログラムのタイトルを指定します。
WARN	ユーザ定義警告を生成します。

6.1.9.1 ALIGN ディレクティブ

ALIGN ディレクティブは、現在の psect 内で後続の全ての内容(例: データ記憶域、コード等)を指定されたオフセット境界へ整列させます。境界は、このディレクティブ後のバイト数として指定されます。

例えば、psect 内で出力を 2 バイト(偶数)アドレスに整列させる場合、以下のコードが使えます。

```
ALIGN 2
```

psect が偶数アドレスで始まる場合、後続の内容は必ず偶数の絶対アドレスで始まります(すなわち、整列は現在の psect 内でのみ行われます)。psect の整列については、6.1.9.36.16 「reloc フラグ」を参照してください。

ALIGN ディレクティブは、psect の長さを特定数の倍数にするために使う事もできます。例えば、上の例の ALIGN ディレクティブを psect の最後に配置した場合、その psect の長さは常に偶数バイト数となります。

6.1.9.2 ASMOPT ディレクティブ

ASMOPT *action* ディレクティブは、アセンブリコードの処理時に、アセンブラ オプティマイザを選択的に制御します。指定可能な *action* を表 6-7 に示します。

表 6-7. ASMOPT ディレクティブの action

action	目的
off	後続のコードに対するアセンブラ オプティマイザの適用を無効にします。
on	後続のコードに対するアセンブラ オプティマイザの適用を有効にします。
pop	アセンブラ最適化設定の状態を復元します。
push	アセンブラ最適化設定の状態を保存します。

ASMOPT off ディレクティブの後のコードは変更されません。ASMOPT on ディレクティブの後のコードに対して、アセンブラは許可されている最適化を実行します。

ASMOPT push および ASMOPT pop ディレクティブを使うと、アセンブラ オプティマイザの状態をスタックに保存し、後で復元する事ができます。これらは、コードの小さなセクションに対してオプティマイザを無効にする必要があるものの、オプティマイザがそれ以前に無効にされたかどうか不明である場合に役立ちます。

例:

```
ASMOPT PUSH  ;store the state of the assembler optimizers
ASMOPT OFF   ;optimizations must be off for this sequence
movlw 0x55
movwf EECON2
movlw 0xAA
movwf EECON2
ASMOPT POP   ;restore state of the optimizers
```

6.1.9.3 BANKISEL ディレクティブ

BANKISEL ディレクティブは、オペランドのメモリ位置の間接アクセス向けに正しい RAM バンクを選択するコードを生成するために使います。オペランドは、データメモリ内に配置されたオブジェクトのシンボルまたはアドレス値です。

例:

```
PROCESSOR 12F510

#include <xc.inc>

PSECT udata_bank1
myVar:
  DS  1

PSECT code
setMode:
  movlw myVar
  movwf FSR       ;Load the address of myVar info FSR
  BANKISEL myVar  ;Select the correct bank for myVar
  movlw 055h
  movwf INDF      ;Indirectly write to myVar
```

このディレクティブは、ミッドレンジ デバイスの IRP ビットまたはベースライン デバイスの STATUS ビットを、シンボル引数向けに適切に設定します。その他の全てのデバイスでは、このディレクティブは無視されます。

6.1.9.4 BANKSEL ディレクティブ

BANKSEL ディレクティブは、オペランドの RAM バンクを選択するためのコードを生成するために使えます。オペランドは、データメモリ内に配置されたオブジェクトのシンボルまたはアドレスである事が必要です(6.1.1.2 「バンクとページの選択」参照)。

6.1.9.5 CALLSTACK ディレクティブ

CALLSTACK *depth* ディレクティブは、プログラム内のその特定位置で利用可能なコールスタックのレベル数をアセンブラに示します。

アセンブラ オプティマイザは、このディレクティブを使って、手続きの抽象化等の変換が可能かどうかを判断します。

6.1.9.6 COND ディレクティブ

COND ディレクティブは、アセンブリ リスティング ファイル内の条件付きコードをインクルードします。相補的な NOCOND ディレクティブは、リスティング ファイル内の条件付きコードをインクルードしません。

6.1.9.7 CONFIG ディレクティブ

CONFIG ディレクティブを使うと、コンフィグレーション ビット(またはヒューズ)と ID ロケーション レジスタをアセンブリ ソースファイル内で指定できます。コンフィグレーション ビット(またはヒューズ)は、基本的なデバイスの動作(オシレータモード、ウォッチドッグ タイマ、プログラミング モード、コード保護等)を指定します。プログラムを正常に実行するには、これらのビットを正しく設定する必要があります。

このディレクティブの書式は以下の通りです。

```
CONFIG setting = value
CONFIG register = literal_value
```

setting : コンフィグレーション設定記述子(例: WDT)

value : 状態を指定するテキスト記述(例: OFF)または数値

setting または *value* トークンまたは *setting = value* 式をダブルまたはシングル クォーテーションで囲む事により、プリプロセッサが実行するマクロ置換からそれらを保護できます。

例:

```
CONFIG "WDT = ON"         ;turn on watchdog timer
CONFIG "FEXTOSC" = "ECH" ;external clock oscillator mode, high power PFM
CONFIG WDTPS = 0x1A       ;specify the timer postscale value
```

純粋に数値のように見える value トークンであっても、実際には値のテキスト記述である場合があります。
例:

```
config WDTPS = 32
```

32 は、一部のデバイス向けのテキスト記述であり、ウォッチドッグ タイマのポストスケールが 1:32 である事を示します(数値の 32 を示すのではありません)。このような場合、コンフィグレーション レジスタの対応するビットには、数値として 32 が書き込まれるのではない可能性があります。アセンブラは最初に *value* が定義済みの文字列を表しているかどうか確認し、そうではない場合にのみ *value* は数値定数であると見なし、他の数値定数オペランドと同様の制約を *value* に適用します。

コンフィグレーション マクロ内で使われる OFF および ON トークンが 0 および 1 と等価であると見なしてはいけません。多くの場合、それは誤りです。

register フィールドはコンフィグレーション レジスタまたは ID ロケーション レジスタの名前であり、必ず *value* で数値定数を指定する必要があります。
例:

```
CONFIG CONFIG1L = 0x8F
```

利用可能な *setting*、*register*、*value* フィールドの設定は、デバイスの chipinfo ファイル(例: pic_chipinfo.html 、pic18_chipinfo.html) に記載されています。このファイルはコンパイラ インストール ディレクトリ内の docs ディレクトリに保存されています。ターゲット デバイスへのリンクをクリックすると、このプラグマ向けの適切な設定と値が表示されます。詳細はデバイスのデータシートを参照してください。

各コンフィグレーションの設定用に CONFIG ディレクティブを 1 つずつ使う事も、複数の設定をコンマで区切って 1 つのディレクティブで指定する事もできます。このディレクティブは、デバイスを完全に設定し終えるまで必要に応じて何度でも使えます。

下の例に、1 つのコンフィグレーション レジスタ内の個々の設定を指定して書き込む方法と、レジスタ全体として書き込む方法を示します。

```
; PIC18F67K22
; VREG Sleep Enable bit :Enabled
; LF-INTOSC Low-power Enable bit :LF-INTOSC in High-power mode during Sleep
; SOSC Power Selection and mode Configuration bits :High Power SOSC circuit selected
; Extended Instruction Set :Enabled
CONFIG RETEN = ON, INTOSCSEL = HIGH, SOSCSEL = HIGH, XINST = ON

; Alternatively
CONFIG CONFIG1L = 0x5D

; IDLOC @ 0x200000
CONFIG IDLOC0 = 0x15
```

不適正なプログラム動作を防ぐため、コンフィグレーション ワード内の全てのビットに書き込む必要があります。どのビットも既定値/未プログラム状態のままにしてはいけません。コンフィグレーション ビットの既定値状態は全てが論理 High であるとは限りません。既定値状態が論理 Low のビットも存在します。詳細はデバイスのデータシートを参照してください。

MPLAB X IDE を使うと、内部ツールを利用して必要なプラグマを生成し、それらをソースコードにコピー&ペーストできます。MPLAB X IDE の「Configuration Bits」ウィンドウの詳細は『MPLAB® X IDE User's Guide 』を参照してください。

6.1.9.8 DABS ディレクティブ

DABS ディレクティブを使うと、指定したアドレスでメモリの 1 つまたは複数のバイトを予約できます。このプラグマの一般的な書式は以下の通りです。

```
DABS memorySpace, address, bytes [,symbol]
```

memorySpace: 予約するメモリ空間を表す番号

address: 予約する領域の開始アドレス

bytes: 予約するバイト数

symbol: オプションとして、そのアドレスに配置されるオブジェクトの名前

symbol を指定すると、コード内の予約されたメモリにアクセスできます。このシンボルは自動的にグローバルにアクセス可能とされ、このディレクティブで指定されたアドレスが割り当てられます。下の例では、シンボル foo がこのディレクティブによって定義されています。

```
DABS 1,0x100,4,foo
```

このシンボルは、コード内で以下の例のように使えます。

```
    movlw 20
    movwf BANKMASK(foo)
```

DABS ディレクティブは現在の psect 内だけでなくどの位置にあるメモリでも予約できるという点で、DS ディレクティブとは異なります。実際、これらのディレクティブは、現在選択されている psect とは全く無関係に、どこにでも配置できます。psect 内で DS ディレクティブによって定義されたオブジェクトにはリンカによって空きメモリが割り当てられますが、DABS ディレクティブを使う場合は、割り当てるアドレスを指定する必要があります。

memorySpace の番号は、psect の space フラグ オプションで指定される番号と同じです(6.1.9.36.18 「space フラグ」参照)。

リンカは、オブジェクトファイルからこの DABS 関連の情報を読み出し、予約されたアドレスが他のメモリ配置向けに使われないように確保します。

6.1.9.9 DB ディレクティブ

DB ディレクティブは、プログラムメモリ内のバイトを初期化するために使います。

このディレクティブは、コンマで区切られた引数のリストを受け取ります。各引数には数値(例: 55)、文字定数(例: 'T')、文字列(例: "level"または'level')が使えます。各引数または文字列引数の各文字は 1 バイトとして、現在の psect 内の連続するプログラムメモリ位置に書き込まれます。

書き込まれる値のエンコードは、このディレクティブを格納している psect の delta フラグによって決まります。例:

```
PSECT myBytes,class=CODE,delta=2
alabel:
  DB  'X',1,2,3,4,
```

この例では、delta psect フラグ(6.1.9.36.4 「delta フラグ」参照) によって指定されるプログラムメモリ内のアドレス単位は 2 バイトです(ベースラインおよびミッドレンジ PIC デバイスの場合)。この場合、DB ディレクティブはプログラムメモリの各ワードを以下の通りに初期化します(16 進表記、各ワードの上位バイトは 0 に設定)。

```
0058 0001 0002 0003 0004
```

しかし、PIC18 デバイスの場合、プログラムメモリ psect の delta フラグは 1 に設定されるため、パディングは発生しません。

```
PSECT myBytes,class=CODE,delta=1
alabel:
  DB  'X',1,2,3,4
```

この場合、以下の(16 進)データがプログラム メモリ向けに HEX ファイル内で定義されます。

```
58 01 02 03 04
```

文字列は、シングルまたはダブルクオートで囲まれた文字の並びで構成されます。アセンブラは終端文字を追加しません。NULL 終端文字列が必要な場合、DB ディレクティブに対して終端文字を別の引数として指定します。例:

```
PSECT myConst,class=CODE,delta=1
bytes:
        DB "a terminated string",0
```

上記は以下を定義します。

```
61 20 74 65 72 6D 69 6E 61 74 65 64 20 73 74 72 69 6E 67 00
```

DB ディレクティブを使ってデータメモリ内にオブジェクトを作成する事はできません。オブジェクトを作成するには DS ディレクティブ(6.1.9.13 「DS ディレクティブ」参照)を使います。

6.1.9.10 DDW ディレクティブ

DDW ディレクティブは、プログラムメモリの 32 ビットワードを初期化するために使います。

このディレクティブは、コンマで区切られた引数のリストを受け取ります。各引数には数値(例: 55)、文字定数(例: 'T')、文字列(例: "level"または'level')が使えます。各引数または文字列引数の各文字は、4 バイトとして現在の psect 内の連続するプログラムメモリ位置に書き込まれます。

例:

```
PSECT myWords,class=CODE,delta=2
alabel:
  DDW 'X',1,2,398,0x8005FFFF
```

この場合、DDW ディレクティブはプログラムメモリの各ワードを以下の通りに初期化します(16 進表記):

```
00000058 00000001 00000002 0000018E 8005FFFF
```

文字列は、シングルまたはダブルクオートで囲まれた文字の並びで構成されます。アセンブラは、終端文字を追加しません。NULL 終端文字列が必要な場合、DDW ディレクティブに対して終端文字を別の引数として指定します。
例:

```
PSECT myConst,class=CODE,delta=1
words:
        DDW "a terminated string",0
```

上記は以下を定義します。

```
00000061 00000020 00000074 00000065 00000072 0000006D 00000069 0000006E 00000061 00000074
00000065 00000064 00000020 00000073 00000074 00000072 00000069 0000006E 00000067 00000000
```

DDW ディレクティブを使ってデータメモリ内にオブジェクトを作成する事はできません。オブジェクトを作成するには DS ディレクティブ(6.1.9.13 「DS ディレクティブ」参照)を使います。

6.1.9.11 DEBUG_SOURCE ディレクティブ

DEBUG_SOURCE *action* ディレクティブは、アセンブラによるオブジェクトコードの生成方法をアセンブリソースまたは C ソースのデバッグに適するように制御します。指定可能な *action* を表 6-8 に示します。

表 6-8. DEBUG_SOURCE ディレクティブの action

action	目的
C	C ソースのデバッグ向けにコードを生成します。
asm	アセンブリソースのデバッグ向けにコードを生成します。
pop	アセンブラ デバッグソース設定の状態を復元します。
push	アセンブラ デバッグソース設定の状態を保存します。

このディレクティブは、デバッグに影響を及ぼす可能性のある情報の生成を制御します。例えば、ラベルに関連するデバッグ情報により、MPLAB X IDE の「Disassembly」画面に表示されるアセンブリマクロの完全な展開が制限される可能性があります。このディレクティブはコードのデバッグにのみ影響を及ぼし、アセンブルされたコードの動作には影響を及ぼしません。

このディレクティブが指定されない場合、asm 設定が採用されます。この設定は、PIC アセンブラを使ってビルドされるプロジェクトまたは C プロジェクトの一部であるマクロを使うアセンブリコード向けに使います。C 設定は、MPLAB XC8 C コンパイラから出力されるアセンブリコード内で、C コードから生成されたアセンブリコード向けに自動的に使われます。

DEBUG_SOURCE push および DEBUG_SOURCE pop ディレクティブを使うと、デバッグソース設定の状態をスタックに保存し、後で復元する事ができます。これらは、コードの小さなセクションに対して特定のデバッグソース状態を適用する必要があるものの、それ以前のデバッグソース状態が不明である場合に役立ちます。

例:

```
DEBUG_SOURCE push    ;store the state of the debug source setting
DEBUG_SOURCE asm     ;ensure proper debugging of the macro used below
    MY_UNLOCK_MACRO  ;this should expand in the Disassembly View
DEBUG_SOURCE pop     ;restore state of the debug source setting
```

6.1.9.12 DLABS ディレクティブ

リニアアドレス指定をサポートするデバイスでは、指定したリニアアドレスでメモリ内の 1 つまたは複数のバイトを予約するために DLABS ディレクティブを使う必要があります。通常、このディレクティブは、1 つのデータバンク内に収まらずリニアアドレス指定によってアクセスする必要がある大きなオブジェクトにメモリを割り当てるために使います。このプラグマの一般的な書式は以下の通りです。

```
DLABS memorySpace, address, bytes [,symbol]
```

引数 *memorySpace* は、リニアメモリ空間を表す番号です。この番号はバンク型データ空間と同じ番号です。引数 *address* は、予約する領域の開始アドレスです。これはリニアアドレスでもバンク型アドレスでも構いません。
引数 *bytes* は予約するバイトの数です。symbol は、オプションとして、そのアドレスに配置されるオブジェクトの名前を指定します。

symbol を指定すると、ユーザコード内の予約されたメモリにアクセスできます。このシンボルは自動的にグローバルにアクセス可能とされ、このディレクティブで指定されたアドレスがリニアアドレス形式で割り当てられます。下の例では、シンボル foo がこのディレクティブによって定義されています。

```
DLABS 1,0x120,128,foo
```

このシンボルにはリニアアドレス 0x20A0 が割り当てられ、下の例のようにコード内で使えます。

```
    movlw low(foo)
    movwf FSR1L
    movlw high(foo)
    movwf FSR1H
```

memorySpace 番号は、psect の space フラグ オプションで指定される番号と同じです(6.1.9.36.18 「space フラグ」参照)。

リンカは、オブジェクトファイルからこの DLABS 関連の情報を読み出し、予約されたアドレスが他のメモリ配置向けに使われないように確保します。

6.1.9.13 DS ディレクティブ

DS *units* ディレクティブは、指定されたサイズの空間を予約しますが、初期化はしません。引数は、予約するアドレス単位の数を指定する *units* だけです。アドレス単位は、このディレクティブを保持する psect で使われているフラグによって決まります。

通常このディレクティブは、データメモリ内の RAM ベース オブジェクト向けにバイトを予約するために使います(オブジェクトを格納する psect の space フラグが 1 に設定されている場合)。このディレクティブを格納している psect がビット psect (psect の bit フラグが 1)である場合、指定された数のビットが予約されます。プログラムメモリにリンクされた psect 内で使われる場合、このディレクティブはロケーション カウンタを移動させますが、HEX ファイル出力内に何も配置しません。ミッドレンジおよびベースライン デバイスの場合、プログラムメモリ内のアドレス単位のサイズは 2 バイトである事に注意が必要です(6.1.9.36.4 「delta フラグ」 参照)。この場合、DS 疑似命令は、そのインスタンス内で指定された数のワードを予約する働きをします。

通常、オブジェクトはラベルを使って定義され、その後に DS ディレクティブを使ってそのラベル位置が予約されます。

例:

```
PSECT myVars,space=1,class=BANK2
alabel:
  DS 23    ;reserve 23 bytes of memory
PSECT myBits,space=1,bit,class=COMRAM
xlabel:
  DS 2+3   ;reserve 5 bits of memory
```

6.1.9.14 DW ディレクティブ

DW ディレクティブは、プログラムメモリの 16 ビットワードを初期化するために使います。

このディレクティブは、コンマで区切られた引数のリストを受け取ります。各引数には数値(例: 55)、文字定数(例: 'T')、文字列(例: "level"または'level')が使えます。各引数または文字列引数の各文字は 2 バイトとして、現在の psect 内の連続するプログラムメモリ位置に書き込まれます。

例:

```
PSECT myWords,class=CODE,delta=2
alabel:
  DW  'X',1,2,398,5472
```

この場合、DW ディレクティブはプログラムメモリの各ワードを以下の通りに初期化します(16 進表記):

```
0058 0001 0002 018E 1560
```

文字列は、シングルまたはダブルクオートで囲まれた文字の並びで構成されます。アセンブラは、終端文字を追加しません。NULL 終端文字列が必要な場合、DW ディレクティブに対して終端文字を別の引数として指定します。
例:

```
PSECT myConst,class=CODE,delta=1
words:
        DW "a terminated string",0
```

上記は以下を定義します。

```
0061 0020 0074 0065 0072 006D 0069 006E 0061 0074 0065 0064 0020 0073 0074 0072 0069 006E
0067 0000
```

DW ディレクティブを使ってデータメモリ内にオブジェクトを作成する事はできません。オブジェクトを作成するには DS ディレクティブ(6.1.9.13 「DS ディレクティブ」参照)を使います。

6.1.9.15 END ディレクティブ

END *label* ディレクティブは必須ではありませんが、使う場合はプログラムの最後に配置する必要があります。このディレクティブはアセンブリを終了させます。このディレクティブの後にはたとえ空白行であっても存在してはいけません。

引数として式が提供された場合、その式はプログラムのエントリポイントを定義するために使われます。これは、アセンブラによって生成されるオブジェクト ファイル内の開始レコードに保存されます。これが使われるかどうかはリンカによって決まります。

例:

```
END start_label ;defines the entry point
```

または

```
END               ;do not define entry point
```

6.1.9.16 EQU ディレクティブ

EQU 疑似命令はシンボルを定義し、その値を式で指定します。

例:

```
thomas EQU 123h
```

識別子 thomas には値 123h が与えられます。EQU は、シンボルが未定義である場合にのみ有効です。値の再定義については 6.1.9.39 「SET ディレクティブ」を参照してください。

このディレクティブは、指定されたシンボル向けにメモリを予約**しません。**データメモリを予約するには DS ディレクティブ(6.1.9.13 「DS ディレクティブ」参照)を使います。EQU は、プリプロセッサの#define ディレクティブ(5.13.1 「プリプロセッサ ディレクティブ」参照)と同様の機能を実行します。

6.1.9.17 ERROR ディレクティブ

ERROR ディレクティブは、ユーザ定義のビルド時エラーメッセージを生成し、アセンブラを停止させます。このディレクティブが実行されると、文字列として指定されたメッセージが出力されます。通常このディレクティブは、無効な条件を検出するために、条件付きで使われます。

例:

```
IF MODE
  call process
ELSE
  ERROR "no mode defined"
ENDIF
```

6.1.9.18 EXPAND ディレクティブ

EXPAND ディレクティブは、マクロ展開によって生成されたコードをアセンブラ リスティング ファイル内に表示します。相補的な NOEXPAND ディレクティブは、マクロ展開によって生成されたコードをアセンブラ リスティング ファイル内で非表示にします。

6.1.9.19 EXTRN ディレクティブ

EXTRN *identifier* 疑似命令は GLOBAL (6.1.9.21 「GLOBAL ディレクティブ」参照) に似ていますが、他のモジュール内で定義されたグローバル シンボルとのリンク用にのみ使われます。同じモジュール内で定義されたシンボルに対して EXTRN を使うとエラーが生成されます。

6.1.9.20 FILE ディレクティブ

FILE ディレクティブは、後続のアセンブリコードを格納しているソースファイルを指定します。

例:

```
FILE "X1_start.s"
```

このディレクティブはプロジェクトに関連するデバッグ情報をビルドするために使われ、通常は C コード ジェネレータによって中間生成アセンブリ ファイル内に配置されます。これらを手書きアセンブリコード向けに要求してはいけません。

6.1.9.21 GLOBAL ディレクティブ

GLOBAL *identifier_list* ディレクティブは、コンマで区切ったシンボルのリストを宣言します。シンボルが現在のモジュール内で定義されている場合、それらはパブリックとされます。シンボルが現在のモジュール内で定義されていない場合、それらは外部モジュール内で定義されたパブリック シンボルに対する参照とされます。従って、同じシンボルを 2 つのモジュール内で使う場合、GLOBAL ディレクティブを少なくとも 2 回使う必要があります。シンボルを定義するモジュール内でそのシンボルをパブリックにするために 1 回と、そのシンボルを使う別のモジュール内で外部定義とリンクするために 1 回使います。

例:

```
GLOBAL  lab1,lab2,lab3
```

6.1.9.22 IF、ELSIF、ELSE、ENDIF ディレクティブ

これらのディレクティブは、条件付きアセンブリを実装します。

IF と ELSIF に対する引数は、絶対式である事が必要です。式の値が非 0 である場合、次の ELSE、ELSIF、ENDIF に遭遇するまで後続のコードがアセンブルされます。式の値が 0 である場合、次の ELSE、ELSIF、ENDIF に遭遇するまでコードは出力されません。ELSE では、コンパイルの条件が逆転します。ENDIF は、条件付きアセンブリ ブロックを終了させます。条件付きアセンブリ ブロックはネストできます。

これらのディレクティブは、C 命令文の if とは異なり、実行時に評価するコードを出力しません。これらのディレクティブは、コードのビルド時にのみ評価されます。加えて、条件式が真であっても偽であってもアセンブリコードは常にスキャンされて解釈されますが、命令に対応するマシンコードは条件が一致した場合にのみ出力されます。従って、アセンブラ ディレクティブ(例: EQU) は条件式の状態に関係なく処理されるため、IF コンストラクタの中で使ってはいけません。

例:

```
IF ABC
    goto aardvark
ELSIF DEF
    goto denver
ELSE
    goto grapes
```

```
ENDIF
ENDIF
```

この例では、ABC が非 0 である場合、最初の goto 命令がアセンブルされ、2 番目と 3 番目の goto 命令はアセンブルされません。ABC が 0 かつ DEF が非 0 である場合、2 番目の goto 命令がアセンブルされ、1 番目と 3 番目の goto 命令はアセンブルされません。ABC と DEF の両方が 0 である場合、3 番目の goto 命令がアセンブルされます。

6.1.9.23 INCLUDE ディレクティブ

INCLUDE "*filename*" ディレクティブは、指定されたファイルの内容と置き換えられます。

例:

```
INCLUDE "options.inc"
```

コンパイラ ドライバは、アセンブラに検索パスを渡しません。このため、インクルード ファイルが現在の作業ディレクトリ内に存在しない場合、*filename* にはファイルのフルパスを指定する必要があります。

拡張子.S を持つアセンブリ ソースファイルは前処理されるため、INCLUDE ディレクティブの代わりにプリプロセッサ ディレクティブ#include が使えます。

6.1.9.24 IRP および IRPC ディレクティブ

IRP および IRPC ディレクティブは、引数リストのメンバーごとに 1 だけ回繰り返します (後述の REPT ではブロックを指定回数繰り返します)。

IRP の場合、通常のマクロ引数リストで指定された各メンバーを繰り返します。IRPC の場合、1 つの引数内の各文字を繰り返します。繰り返しごとに、引数は 1 つの仮パラメータと置換されます。

例:

```
IRP number,4865h,6C6Ch,6F00h
    DW number
ENDM
```

上記は以下へと展開されます。

```
DW 4865h
DW 6C6Ch
DW 6F00h
```

通常のマクロと同様の方法でローカルラベルと山カッコが使えます。

IRPC ディレクティブは、非空白類文字の文字列から 1 文字ずつ置換します。

例:

```
IRPC char,ABC
    DB 'char'
ENDM
```

上記は以下へと展開されます。

```
DB 'A'
DB 'B'
DB 'C'
```

6.1.9.25 LINE ディレクティブ

LINE ディレクティブは、後続のアセンブリコードを格納した現在のソースファイルの行番号を指定します。

例:

```
LINE 26
```

このディレクティブはプロジェクトに関連するデバッグ情報をビルドするために使われ、通常は C コード ジェネレータによって中間生成アセンブリ ファイル内に配置されます。これらを手書きアセンブリコード向けに要求してはいけません。

6.1.9.26 LIST ディレクティブ

LIST ディレクティブは、リスティング出力の生成を制御します。

以前に NOLIST ディレクティブによってリスティングが無効にされていた場合、LIST ディレクティブはリスティングを有効に戻します。

LIST ディレクティブのオプションを指定する事でアセンブリコードのリスト出力を制御する事もできます。それらのオプションを表 6-9 に示します。

表 6-9. LIST ディレクティブのオプション

オプション	既定値	概要
c=*nnn*	80	ページ(列) の幅を設定します。
n=*nnn*	59	ページの長さを設定します。
t=ON|OFF	OFF	リスト出力の行を切り詰めます。既定値は行をラップします。
p=*device*	n/a	デバイスのタイプを設定します。
x=ON|OFF	OFF	マクロ展開を ON/OFF します。

6.1.9.27 LOCAL ディレクティブ

LOCAL *label* ディレクティブを使うと、与えられたマクロの各展開に対して一意のラベルを定義できます。LOCAL ディレクティブの後で指定された全てのシンボルは、マクロの展開時にアセンブラが生成する一意のシンボルに置換されます。

例:

```
down MACRO count
  LOCAL more
  more: decfsz count
  goto more
ENDM
```

上記の例では、ラベル more はアセンブラが生成した一意のラベルに置き換わります。

例:

```
down foobar
```

上記は下記へと展開されます。

```
??0001 decfsz foobar
goto ??0001
```

2 度目に呼び出された場合、ラベル more は??0002 へと展開され、重複定義シンボルエラーは回避されます。

6.1.9.28 MACRO および ENDM ディレクティブ

MACRO … ENDM ディレクティブは、アセンブリ マクロの定義を提供します。オプションで引数を指定する事もできます。識別子への簡易な値の割り当てについては 6.1.9.16 「EQU ディレクティブ」を参照してください。プリプロセッサの#define マクロ ディレクティブについては 5.13.1 「プリプロセッサ ディレクティブ」を参照してください。こちらも引数が使えます。

MACRO ディレクティブの前にはマクロ名が必要であり、後にはコンマで区切った仮引数のリストをオプションとして指定できます。マクロを利用する時、マクロ名をアセンブリ命令と同様に使う必要があり、その後に仮パラメータ向けに置換される引数のリストが続きます。

例:

```
;macro: movlf - Move a literal value into a nominated file register
;args:arg1 - the literal value to load
;       arg2 - the NAME of the source variable
movlf   MACRO   arg1,arg2
  movlw arg1
  movwf arg2 mod 080h
ENDM
```

上記例では movlf というマクロがボディ内の 2 つの命令へと展開され、movlf に続く仮パラメータが利用時の引数と置換されます。

例:

```
movlf 2,tempvar
```

上記は下記へと展開されます。

```
movlw 2
movwf tempvar mod 080h
```

&文字はマクロ引数と他のテキストを連結するために使えますが、展開後にこの文字は削除されます。

例:

```
loadPort MACRO port, value
  movlw value
  movwf PORT&port
ENDM
```

呼び出された時に port が例えば A であった場合、PORTA がロードされます。マクロ内では& トークンは特別な意味を持つため、ビット単位 AND 演算子(&) は使えません(代わりに and が使えます)。

コメントを 2 個のセミコロン「;;」で始める事により、マクロ展開でのコメントの生成を抑止できます(従ってマクロを保存するためのスペースを節約できます)。

マクロの呼び出し時に、引数のリストはコンマで区切る必要があります。引数にコンマ(またはその他のデリミタ(スペース等))を含める必要がある場合、それらの文字を山カッコ(< と>) で囲みます。

引数の先頭にパーセント文字「%」を付けると、その引数は式として評価され、文字列ではなく 10 進数として渡されます。これにより、マクロボディ内での引数の評価に応じて異なる結果を得る事ができます。

nul 演算子をマクロ内で使う事で、マクロ引数をテストできます。

例:

```
IF nul     arg3 ;argument was not supplied.
...E
LSE              ;argument was supplied
...EN
DIF
```

マクロ内での一意のローカルラベルの使用については、6.1.9.27 「LOCAL ディレクティブ」を参照してください。

既定値では、アセンブリ リストファイルの上ではマクロ未展開(拡張命令)のように示します。EXPAND アセンブラディレクティブ(6.1.9.18 「EXPAND ディレクティブ」参照)を使うと、リスティング ファイル内にマクロの展開を表示する事ができます。

6.1.9.29 MESSG ディレクティブ

MESSG ディレクティブは、ユーザ定義のビルド時アドバイザリ メッセージを生成します。このディレクティブが実行されても、アセンブラのビルドは続行されます。このディレクティブが実行されると、文字列として指定されたメッセージが出力されます。通常このディレクティブは、無効な条件を検出するために、条件付きで使われます。

例:

```
IF MODE
  call process
ELSE
  MESSG "no mode defined"
ENDIF
```

6.1.9.30 ORG ディレクティブ

ORG ディレクティブは、現在の psect 内のロケーション カウンタの値を変更します。従って、ORG によって設定されるアドレスはその psect のベースアドレスに対する相対アドレスであり、ベースアドレスはリンク時まで不確定です。

Note: 誤って使われた ORG ディレクティブは、ロケーション カウンタをユーザが指定した絶対アドレスに移動させません。このディレクティブが配置されている psect が絶対アドレス指定されかつオーバーレイされている場合にのみ、ロケーション カウンタは指定されたアドレスへ移動します。オブジェクトを特定アドレスに配置するには、それらを専用のpsect 内に配置し、リンカ オプション-P (7.1.17 「P: psect を配置する」参照)を使ってそのpsect を必要なアドレスでリンクする必要があります。通常、プログラム内でORGディレクティブを使う必要はありません。

ORG に対する引数は、絶対アドレス値または現在の psect に対する相対アドレス値である事が必要です。どちらの場合も、現在のロケーション カウンタは引数によって指定された値に設定されます。ロケーション カウンタを後戻りさせる事はできません。

例:

```
ORG 100h
```

上記は、ロケーション カウンタを現在の psect の開始アドレス + 100h へ移動させます。実際のアドレスは、リンク時まで不確定です。

ORG ディレクティブを使ってロケーション カウンタを絶対アドレス値に設定するには、このディレクティブを絶対アドレス指定されかつオーバーレイされた psect から使う必要があります。

例:

```
PSECT absdata,abs,ovrld
ORG 50h
;this is guaranteed to reside at address 50h
```

6.1.9.31 PAGE ディレクティブ

PAGE ディレクティブは、リスティング出力内で改ページをトリガします。ソース内の Ctrl-L (フォームフィード)文字によっても改ページが発生します。

6.1.9.32 PAGELEN ディレクティブ

PAGELEN *nnn* ディレクティブは、アセンブリ リスティングの長さ(行数)を設定します。

6.1.9.33 PAGESEL ディレクティブ

PAGESEL ディレクティブを使うと、アドレス オペランドのページを選択するためのコードを生成できます(6.1.1.2 「バンクとページの選択」参照)。

6.1.9.34 PAGEWIDTH ディレクティブ

PAGEWIDTH *nnn* ディレクティブは、アセンブリ リスティングの幅を指定された文字数に設定します。

6.1.9.35 PROCESSOR ディレクティブ

アセンブラソースが特定の 1 つのデバイスにのみ適用可能である場合、そのモジュール内で PROCESSOR ディレクティブを使う必要があります。ビルド時は、常に-mcpu オプションを使ってターゲット デバイスを指定する必要があります。このディレクティブで指定されたデバイスと-mcpu で指定されたデバイスが一致しない場合、エラーが生成されます。

例:

```
PROCESSOR 18F4520
```

6.1.9.36 PSECT ディレクティブ

PSECT ディレクティブは、プログラム セクションを宣言または再開します。

このディレクティブには引数として psect 名を渡します。オプションとして、コンマで区切ったフラグのリストを指定できます。これらのフラグは psect の属性を指定します。使用可能なフラグを表 6-10 に示します。

psect 名は通常のアセンブリ シンボルとは別の名前空間内にあるため、psect 名には通常のアセンブリ識別子と同じ識別子が使えます。しかし、psect 名にアセンブラ ディレクティブ、キーワード、psect フラグのいずれかと同じ名前を持たせる事はできません。

psect が 1 度宣言された後に、再度 PSECT ディレクティブを使ってその psect を再開できますが、フラグを繰り返し指定する必要はありません(フラグは以前の宣言から継承されます)。同じ psect に対してフラグが異なる PSECT ディレクティブが使われると、エラーが生成されます(例外: reloc、size、limit フラグは再指定可能)。

表 6-10. PSECT ディレクティブのフラグ

フラグ	意味
abs	psect は絶対アドレス指定
bit	psect はビット オブジェクトを保持する
class=*name*	psect のクラス名の指定

.....続き

フラグ	意味
delta=*size*	アドレス単位のサイズ
global	psect はグローバル(既定値)
inline	psect の内容(関数)は呼び出し時にインライン化可能
keep	psect はインライン化後に削除されない
limit=*address*	psect の上限アドレス(PIC18 のみ)
local	psect は一意であり同名を持つ他の psect とリンクしない
lowdata	psect 全体が 0x1000 より下のアドレス領域内に配置される
merge=*allow*	この psect のマージを許可/禁止する
noexec	psect は実行可能コードを格納しなし(デバッグ用)
note	psect はプログラム イメージ内に出力されるべきデータを一切含まない
optim=*optimizations*	psect 向けに許容する最適化を指定
ovrld	psect は他のモジュール内の同名 psect とオーバーラップする
pure	psect は読み出し専用
reloc=*boundary*	psect は指定された境界で始まる
size=*max*	psect の最大サイズ
space=*area*	psect が格納されるメモリ領域を指定
split=*allow*	psect の分割を許可/禁止する
with=*psect*	psect を指定された psect と同じページ内に配置する

PSECT ディレクティブの使用例を以下に示します。

```
; swap output to the psect called fred
PSECT fred
; swap to the psect bill, which has a maximum size of 100 bytes and which is global
PSECT bill,size=100h,global
; swap to joh, which is an absolute and overlaid psect that is part of the CODE linker class,
; and whose content has a 2-byte word at each address
PSECT joh,abs,ovrld,class=CODE,delta=2
```

6.1.9.36.1 abs フラグ

abs フラグは、現在の psect が絶対アドレス指定である(すなわちアドレス 0 で始まる)として定義します。これは、その psect に対するこのモジュールの配置がアドレス 0 から始まるという事を意味しません。なぜなら、他のモジュールが同じ psect に配置する可能性があるからです(6.1.9.36.14 「ovrld フラグ」も参照)。

abs フラグが指定された psect の再配置はできません。そのような psect の再配置を試みるリンカ オプションが発行されると、エラーが発生します。

6.1.9.36.2 bit フラグ

bit フラグは、その psect が 1 ビット幅のオブジェクトを保持する事を指定します。そのような psect のスケール値は 8 であり(すなわち、その psect 内の各バイトは 8 個のアドレス単位を含む)、この psect に対する全てのアドレスはビットアドレスです。psect のスケール値が 1 ではない場合、マップファイル内にスケール値が示されます(7.3 「マップファイル」参照)。

6.1.9.36.3 class フラグ

class フラグは、その psect に割り当てるリンカクラス名を指定します。クラスによって、psect を配置可能なアドレスレンジが決まります。

クラス名を使う事で、リンク時にローカル psect の配置が可能となります。なぜなら、それらの psect は-P リンカ オプション内でそれらの独自名によって常に参照されるとは限らないからです(例: 同一名のローカル psect が複数存在する場合)。

psect を特定アドレスではなく特定アドレスレンジ内の任意アドレスに配置可能な場合、クラス名は役立ちます。通常、ユーザが定義した psect にクラスを割り当てると、その psect をメモリ内に配置するための特別なリンカ オプションを指定する必要はなくなります。

詳細は 7.1.1 「A: リンカクラスの定義」を参照してください。

6.1.9.36.4 delta フラグ

delta フラグは、アドレス指定単位のサイズを定義します。アドレス指定単位とは、各アドレスに割り当てられるデータバイトの数です。

ミッドレンジおよびベースライン デバイスの場合、プログラムメモリ空間はワード単位でアドレス指定するため、この空間内の psect の delta は 2 である事が必要です。すなわち、プログラムメモリ内の各アドレスの内容を定義するために、HEX ファイル内で 2 バイトのデータが必要です。このため、HEX ファイル内のアドレスは、プログラムメモリ内のアドレスと一致しません。

これらのデバイスのデータメモリ空間はバイト単位でアドレス指定するため、この空間内の psect の delta は 1 である事が必要です。これは delta 値の既定値です。

PIC18 デバイスの場合、全てのメモリ空間はバイト単位でアドレス指定するため、全ての psect の delta は 1 (既定値)である事が必要です。

不適切な delta 値を使って psect を再定義すると、アセンブラがフェイズエラーを発行する可能性があります。

6.1.9.36.5 global フラグ

global フラグは、その psect はリンク時に他のモジュール内の同名のグローバル psect と連結するする必要がある事を示します。

local フラグを使わない限り、psect は既定値によりグローバルであると見なされます。

6.1.9.36.6 inline フラグ

このフラグは廃止される予定です。代わりに optim フラグを使う事を推奨します。

inline フラグは、psect の内容がインライン化可能である事をアセンブラに知らせるために、コード ジェネレータによって使われます。これにより inline psect の内容は複製され、その psect 内で定義されている関数への呼び出しを置き換えるために使われます。

6.1.9.36.7 keep フラグ

このフラグは廃止される予定です。代わりに optim フラグを使う事を推奨します。

keep フラグは、アセンブラ オプティマイザによるインライン化の後にその psect が削除されてしまう事を防ぎます。このフラグを指定しないと、インライン化が指定された psect (6.1.9.36.6 「inline フラグ」参照)は、インライン化処理の後に削除される可能性があります。

6.1.9.36.8 limit フラグ

limit フラグは、psect の最高アドレスに対する制限値を指定します。この psect がメモリ内に配置された時に最高アドレスがこの制限値を超えると、エラーが生成されます。このフラグは、現在のところ PIC18 デバイスのビルド時にのみ利用可能です。

6.1.9.36.9 local フラグ

local フラグを使って定義された psect は、リンク時に他のモジュール内の他の local psect と (たとえ同名の psect が存在しても) 結合されません。1 つのモジュール内に 2 つの local psect が存在する場合、それらは同じ psect を参照します。local psect は、global psect と(たとえそれらが別のモジュール内であっても) 同じ名前を持つ事はできません。

local psect とリンカクラスが割り当てられていない psect (6.1.9.36.3 「class フラグ」参照) は、-P リンカ オプションを使ってアドレスにリンクする事はできません。なぜなら、同一名の psect が複数存在する可能性があるからです。通常、これらの psect は class フラグを定義し、そのクラスのアドレスレンジ内の任意位置に配置されます。

6.1.9.36.10 merege フラグ

このフラグは廃止される予定です。代わりに optim フラグを使う事を推奨します。

merge フラグは、他の psect とのマージ方法を制御します。このフラグには 0 または 1 を指定できます。0 を指定した場合、psect は最適化中にアセンブリ オプティマイザによってマージされません。1 を指定した場合、psect は最適化中に他の psect とマージされる可能性があります。オプティマイザは、この psect と他の psect をマージする事で最適化の効果が得られると判断した場合にのみマージを実行します。このフラグが指定されていない場合、マージは実行されません。

通常、マージはコードベース psect (text psect) に対してのみ実行されます。

6.1.9.36.11 noexec フラグ

noexec フラグは、psect が実行可能コードを格納していない事を示します。この情報は、デバッグ用にのみ使われます。

6.1.9.36.12 note フラグ

note フラグは、コンパイラまたはデバッガツール向けの内容を格納した特殊な psect によって使われます。この psect の内容は最終的なプログラム出力へ複製されません。note フラグを指定する場合、他の psect フラグの使用は禁止されます。他のフラグが使われた場合、警告が生成されます。

6.1.9.36.13 optim フラグ

optim フラグは、その psect の内容に対して許可する最適化の種類を指定します。

最適化の種類は、コロンで区切ったリストで指定できます(表 6-11 参照)。空白のリストは、その psect に対して最適化を一切許可しない事を意味します。

表 6-11. optim フラグのオプション

オプション名	許容する最適化
inline	psect の内容がインライン化される事を許容する
jump	ジャンプベースの最適化を実行する
merge	psect の内容が他の類似する psect とマージされる事を許容する(PIC10/12/16 デバイスのみ)
pa	手続きの抽象化を実行する
peep	ピープホール最適化を実行する
remove	psect の内容が完全にインライン化された後にその psect が削除される事を許容する
split	psect がサイズ制限を超える場合にその psect が分割される事を許容する(PIC10/12/16 デバイスのみ)
empty	この psect に対する最適化を無効にする

例:

```
PSECT myText,class=CODE,reloc=2,optim=inline:jump:split
```

上記の psect 定義は、myText psect の内容に対するインライン化、ジャンプベース最適化、分割を許容します。

例:

```
PSECT myText,class=CODE,reloc=2,optim=
```

上記の定義は、この psect に対する最適化をオプティマイザの設定に関係なく全て無効にします。

optim フラグは、以下の各フラグの代わりに使えます。

merge、split、inline、keep

6.1.9.36.14 ovrld フラグ

ovrld フラグは、リンク時にこの psect の内容を他のモジュールからの psect の内容とオーバーレイする必要がある事をリンカに知らせます。通常、同一名の psect はモジュールをまたいで連結されます。同一モジュール内のオーバーレイされた psect に対する配置は常に連結されます。

このフラグと abs フラグ(6.1.9.36.1 「abs フラグ」参照) の組み合わせにより、絶対アドレス確定済みの psect を定義できます(その psect 内で定義された全てのシンボルは絶対アドレスとなります)。

6.1.9.36.15 pure フラグ

pure フラグは、この psect は実行時に変更されないという事をリンカに指示します。例えば、ROM 内に配置された psect がこれに該当します。このフラグはリンカとターゲット システムに依存するため、その有用性は制限されます。

6.1.9.36.16 reloc フラグ

reloc フラグを使うと、psect を特定の境界に整列させるための要件を指定できます。境界は 2 のべき乗値(例: 2、8、0x40)で指定する必要があります。例えば、フラグ reloc=100h は、この psect が 0x100 の倍数のアドレス(例: 0x100、0x400、0x500)で始まる必要がある事を示します。

PIC18 の命令はワード境界に整列するため、PIC18 上で実行可能コードを格納する全ての psect の reloc 値は 2 である事が必要です。PIC18 のその他のセクションおよびその他のデバイスの全てのセクションには、通常既定値の reloc 値(=1) が使えます。

6.1.9.36.17 size フラグ

size フラグを使うと、psect の最大サイズを指定できます(例: size=100h)。psect のサイズは、全てのモジュールからの psect が結合された後にリンカによって確認されます。

6.1.9.36.18 space フラグ

space フラグは、psect が配置されるメモリ空間を指定するために使います。プログラムメモリ内に配置された psect とデータメモリ内に配置された psect の space 値は異なるため、例えばプログラムメモリ空間内のアドレス 0 とデータメモリ空間内のアドレス 0 は異なる位置として識別されます。

space フラグの値とメモリ空間の関係を表 6-12 に示します。

表 6-12. space フラグの値

space フラグの値	メモリ空間
0	プログラムメモリ、PIC18 デバイスの EEPROM
1	データメモリ
2	予約済み
3	ミッドレンジ デバイスの EEPROM
4	コンフィグレーション ビット
5	IDLOC
6	Note

バンク型データメモリ空間を備えたデバイスの各バンクの space 値は同じです。この psect 内のオブジェクトにはバンク番号を含むフルアドレスが使われます。このため、データメモリ空間内の各位置は一意に識別可能です。例えばバンクのサイズが 0x80 バイトである場合、バンク 0 のオブジェクトにはアドレス 0〜0x7F が割り当てられ、バンク 1 内のオブジェクトにはアドレス 0x80〜0xFF が割り当てられます。

6.1.9.36.19 split フラグ

このフラグは廃止される予定です。代わりに optim フラグを使う事を推奨します。

split フラグには 0 または 1 を指定できます。 0 を指定した場合、psect は最適化中にアセンブリ オプティマイザによって分割されません。1 を指定した場合、psect は分割される可能性があります。分割は、この psect の他の属性が分割を許容し、かつ、この psect のサイズが大きすぎて利用可能メモリ内に収まらない場合に実行されます。このフラグが指定されない場合、この psect が分割可能かどうかは psect がマージ可能かどうかに基づきます(6.1.9.36.10 「merge フラグ」参照)。

6.1.9.36.20 with フラグ

with フラグを使うと、psect を別の psect と同じページ内に配置できます。例えば、with=text は、この psect を text psect と同じページ内に配置します。

他の psect と一緒に配置された各 psect のサイズの合計は withtotal として参照されます。

6.1.9.37 RADIX ディレクティブ

RADIX *radix* ディレクティブは、アセンブラ ソースファイル内で指定される数値定数の基数を制御します。使用可能な基数を表 6-13 に示します。

表 6-13. RADIX ディレクティブのオペランド

基数	意味
dec	10 進値定数
hex	16 進値定数
oct	8 進値定数

6.1.9.38 REPT ディレクティブ

REPT ディレクティブは無名マクロを一時的に定義し、それを引数によって指定された回数で繰り返し展開します。

例:

```
REPT 3
    addwf fred,w
ENDM
```

上記は以下へと展開されます。

```
addwf fred,w
addwf fred,w
addwf fred,w
```

6.1.9.24 「IRP および IRPC ディレクティブ」を参照してください。

6.1.9.39 SET ディレクティブ

SET ディレクティブは、シンボルの再定義をエラーなしで許容するという点を除けば、EQU ディレクティブ(6.1.9.16 「EQU ディレクティブ」参照)と同じです。

例:

```
thomas SET 0h
```

このディレクティブは、指定されたシンボル向けにメモリを予約**しません**。データメモリを予約するには DS ディレクティブ(6.1.9.13 「DS ディレクティブ」参照)を使います。SET ディレクティブは、プリプロセッサの#define ディレクティブ(5.13.1 「プリプロセッサ ディレクティブ」参照)と同様の機能を実行します。

6.1.9.40 SIGNAT ディレクティブ

SIGNAT ディレクティブは、16 ビットのシグネチャ値をラベルに割り当てるために使います。リンク時に、リンカは特定のラベルに対して、定義された全てのシグネチャが一致するかどうか確認します。一致しない場合、リンカはエラーを生成します。SIGNAT ディレクティブを使う事で、リンク時に関数プロトタイプと呼び出しの間の整合性を確認させる事ができます。

例:

```
SIGNAT _fred,8192
```

上記は、シグネチャ値 8192 をシンボル_fred に割り当てます。いずれかのオブジェクトファイル内で_fred のシグネチャ値が異なる場合、リンカはエラーを報告します。

多くの場合、このディレクティブは C ソースから呼び出されるアセンブリ ルーチンで使われます。MPLAB XC8 C コンパイラによって使われるシグネチャ値を特定するための最も容易な方法は、アセンブリ ルーチン向けに必要なプロトタイプと同じプロトタイプを C ルーチンで書く事です。そして、コード ジェネレータによって決定されたその関数のシグネチャ ディレクティブの引数をアセンブリ リストファイル内に記載します。

6.1.9.41 SPACE ディレクティブ

SPACE *nnn* ディレクティブは *nnn* 行の空白をアセンブリ リスティング出力内に配置します。

6.1.9.42 SUBTITLE ディレクティブ

SUBTITLE "*string*"ディレクティブは、アセンブリ リスティング ファイルの各ページの先頭(タイトルの下)に表示されるサブタイトルを定義します。サブタイトルはシングルまたはダブルクオートで囲む必要があります。

6.1.9.43 TITLE ディレクティブ

TITLE "*string*"ディレクティブ キーワードは、アセンブリ リスティング ファイルの各ページの先頭に表示されるタイトルを定義します。タイトルはシングルまたはダブルクオートで囲む必要があります。

6.1.9.44 WARN ディレクティブ

WARN ディレクティブは、ユーザ定義のビルド時警告メッセージを生成します。このディレクティブが実行されても、コンパイラのビルドは続行されます。このディレクティブが実行された時に出力される警告は、文字列により指定します。通常このディレクティブは、無効な条件を検出するために、条件付きで使われます。

例:

```
IF MODE
  movwf modeQ
ELSE
  WARN "MODE is zero - write skipped"
ENDIF
```

6.2 アセンブリレベルの最適化

コード ジェネレータが C コードに対してが直接実行する最適化に加えて、アセンブラがアセンブリコードに対して最適化を実行します。

-fasmfile ドライバ オプション(4.6.6.7 「asmfile オプション」参照)が有効である場合、アセンブラは手書きアセンブリソース モジュールに対してのみ最適化を実行します。いかなる場合も、C コードに追加されたインライン アセンブリコード(5.11.2 「インライン アセンブリ」参照) に対して最適化は実行されません。

本アセンブラが実行可能な最適化を以下に挙げます。特に明記しない限り、これらの最適化はレベル 3 および s でのみ有効となるため、ライセンスなしのコンパイラでは利用できません(「最適化を制御するためのオプション」参照)。アセンブリコード レベルの最適化には以下が含まれます。

- 小さなルーチンをインライン化する事で、ルーチンの呼び出しを不要にします。1 度しか呼び出されない非常に小さなルーチン(通常は数個の命令)は、コードサイズに悪影響を及ぼさないように変更されます。これにより、コードサイズを大きく増加させる事なくコード実行を高速にできます。
- 関数を明示的にインライン化するためにインライン指定子(5.7.1.2 「inline 指定子」参照)を使います。
- 複数回現れるアセンブリコード シーケンスに対して手続きの抽象化が実行されます。これはインライン化とは逆の処理です。コードシーケンスは、ラベル PL*x* (*x* は番号)を使う呼び出し可能コードへと抽象化されます。元のコードシーケンスの全てのインスタンスは、このコードへの呼び出しに置き換えられます。この最適化はコードサイズをかなり削減しますが、コード実行速度にはあまり影響を及ぼしません。しかし、デバッグには悪影響を及ぼします。
- Jump-to-jump 型の最適化が最初に適用され、汎用テンプレートに従う条件付きコードシーケンスに関連する出力が整理されます。Jump-to-jump 型の最適化は、ジャンプ命令からジャンプ命令へのジャンプを削除します。この最適化は最適化レベル 2 で有効になります(ライセンスなしコンパイラで利用可能)。
- 到達不能コードを削除します。他の最適化によって一部のコードが孤立し、通常の実行ではそれらのコードが到達不能になる可能性があります(リターン命令後の命令等)。全てのラベルはエントリポイントになり得ると見なされ、ラベルの後のコードは常に到達可能であると見なされます。
- ピープホール最適化が全ての命令に対して実行されます。これらの最適化は、各命令の実行の直前/直後の状態を評価し、命令削除の可能性を探ります。これらの最適化は、各ステップで 1 つまたは複数の命令を変更または削除します。例えば、W が値 0 を格納する事が既知である場合、W を特定のアドレスへ移動させる命令 (movwf) は、clrf 命令に置き換えられる可能性があります。
- 他の最適化を可能にするために psect のマージが実行される可能性があります。同じ psect 内のコードは同じプログラムメモリ ページ内に配置される事が保証されます。このため、同じ psect 内であれば、呼び出しまたはジャンプの実行前にページ選択ビットを設定する必要はなくなります。ljmp および fcall 命令を使っているコードには、この最適化が効果的です(6.1.1 「アセンブリ命令からの逸脱」参照)。

アセンブリ最適化を適用すると、しばしば MPLAB X IDE 等のツールを使ったデバッグに障害が生じます。コードをデバッグする際は、アセンブリ最適化が無効になるよう最適化レベルを下げる事が必要になるかもしれません(4.6.6 「最適化を制御するためのオプション」参照)。アセンブラ最適化は、コードサイズを大幅に削減可能ですが、一般的に RAM 使用量に対する効果はほとんどありません。

6.3 アセンブリ リストファイル

本アセンブラは、ユーザからの要求に応じてアセンブリ リストファイルを生成します。通常、アセンブリ リストファイルの生成を要求するために、xc8-cc ドライバ オプション-Wa,-a を使います(4.6.10 「アセンブラ オプション」参照)。

アセンブリ リストファイルは、C ソースコードとアセンブリ ソースコードの両方から生成したアセンブリ出力を示します。アセンブラ オプティマイザが有効である場合、アセンブリ出力はアセンブリ ソースコードと異なる可能性があります。それでも、アセンブリ リストファイルはアセンブリ プログラミングに役立ちます。

リストファイルは人間が読める形式で出力されます。アセンブラがリストファイルをソースファイルとして使う事はできません。アセンブリ リストファイルは、アセンブリ出力ファイルとは異なり、アドレスおよびオペコード データを含みます。

アセンブラ オプティマイザは一部の式を単純化すると共に、明確化のために一部のアセンブラ ディレクティブをリスティング ファイルから削除します(これらのディレクティブは実際のアセンブリ出力ファイルには含まれます)。コンパイラが生成したコードをアセンブリ リストファイルで確認する際は、アセンブラ オプティマイザを無効にする事で、コンパイラが生成したディレクティブの全てをリストファイル内で確認できます。確認した後は、開発を継続するためにオプティマイザを再度有効にする事を忘れないよう注意が必要です。オプティマイザの制御に関する詳細は 4.6.6 「最適化を制御するためのオプション」に記載しています。

リンク処理が正常に完了した場合、リンカによって更新されたリスティング ファイルには絶対アドレスとシンボル値が格納されます。従って、アセンブラ リストファイルを使って命令の位置と正確なオペコードを調べる事ができます。

リンカがリストファイルを更新しなかった場合、アセンブリ リストファイル内のアドレスまたはオペコードの後にアポストロフィ「'」が示されます。ほとんどの場合、これはコンパイラエラーまたはコンパイラ オプション (リンク処理の前にコンパイルを停止)が原因で発生します。

例:

```
85  000A' 027F                      subw    127,w
86  000B' 1D03                      skip
87  000C' 2800'                     goto    u15
```

これらのアポストロフィは、アドレスは絶対アドレスではなくそれらを格納している psect 内のアドレス オフセットを示しているに過ぎないという事と、オペコードはフィックスアップされていないという事を示します。フィックスアップされていないオペコード内のアドレス フィールドは値 0 で示されます。

アセンブラは、渡された各アセンブリファイルに対して 1 つのアセンブリ リストファイルを生成します。従って、プロジェクト内の全ての C ソースコード (p-code ベースのライブラリコードを含む) 向けに 1 つのファイルが生成されます。このファイルは、実行時起動コードの一部を形成する C 初期化コードの一部も含みます。各アセンブリ ソースファイルに対しても 1 つのファイルが生成されます。通常、各プロジェクトには少なくとも 1 つのアセンブリファイルが存在します。プロジェクトは実行時起動ファイルの一部を含み、そのファイル名は通常 startup.s です。

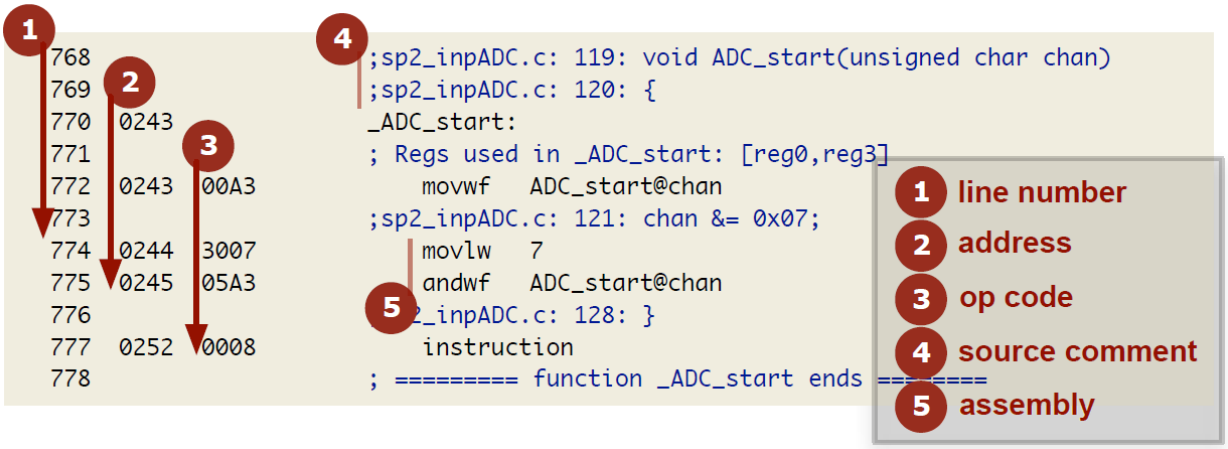
6.3.1 一般的な書式

アセンブリ リストファイルの一般的な書式を図 6-1 に示します。

行番号はアセンブリ リストファイル内の行番号です(C ソースまたはアセンブリ ソースファイル内の行番号ではありません)。セミコロンで始まるコードは、コード ジェネレータによって追加されたコメントです。そのようなコメントには、オリジナルのソースコード(生成されたアセンブリコードに対応)またはコード ジェネレータによって挿入されたコメント(動作の説明)が含まれます。

各関数の出力の前に、コード ジェネレータによって要約されたその関数の詳細情報が示されます。これにはレジスタの使用、ローカル変数、呼び出し先関数、呼び出し元関数に関連する情報が含まれます。

図 6-1. アセンブリ リスティング ファイルの一般書式



6.3.2 psect 情報

アセンブリ リストファイルを使ってデータ オブジェクトまたはコード セクションが格納されている psect の名前を調べる事ができます。

グローバル シンボルに関しては、マップファイル内のシンボルテーブル (シンボルごとの psect 名を示す)で確認できます。モジュールに対してローカルなシンボルに関しては、リストファイル内にシンボルの定義が示されます。ラベルは、後にコロン「:」が付いたシンボルの名前として示されます。このコードがどの psect に配置されているかは、リストファイル上で上に遡って最初に現れる PSECT アセンブラ ディレクティブを探す必要があります。このディレクティブで指定されている名前が、このコードが配置されている psect の名前です(6.1.9.36 「PSECT ディレクティブ」参照)。

6.3.3 関数情報

各 C 関数の情報は、関数のアセンブリラベルの前に表示されます(直後にコロン「:」が付いた関数の名前を探してください)。そこには、その関数によって使われるリソースの一般情報が示されます。代表的な例を図 6-2 に示します。ほとんどの情報は説明を要しませんが、特別なコメントが後に続きます。

位置は *offset*[space]として示されます。例えば 42[BANK0]は、変数がバンク 0 メモリ空間内でコンパイルド スタック コンポーネントに対して 42 バイト オフセットした位置に配置された事を意味します(5.2.4.2.1 「コンパイルド スタックの動作」参照)。これは、その変数がアドレス 42 に配置された事を意味するのではありません。アドレス 42[BANK0]を持つ全てのオブジェクトは同じ位置に配置されます。

多くの場合、ポインタ変数はポインタが参照可能なターゲットと一緒に示され、ターゲットは矢印「->」の後に示されます(6.3.5 「ポインタ参照グラフ」参照)。ポインタのサイズは動的に変化するため、この情報の auto および parameters セクションは特に役立ちます(5.3.6 「ポインタのタイプ」参照)。この情報は、各ポインタ変数に割り当てられた実際のバイト数を示します。

追跡されたオブジェクト情報は、一般的にプログラム作成者にとって重要ではありません。これは、関数の開始および終了時に選択されている RAM バンクの既知の状態を示します。また、関数内でバンク選択ビットが変更されたかどうかも示します。

ハードウェア スタック情報は、この関数が単独で消費したスタックレベル数と、この関数自身とこの関数が呼び出す全ての関数が消費した総スタックレベル数を示します。これは、インライン化されていない関数に対してのみ有効である事に注意が必要です。

再入不可モデルを使用するとして示されている関数は、auto およびパラメータ変数をコンパイルド スタックに割り当てます。再入可能として示されている関数は、スタックベース変数をソフトウェア スタックに割り当てます。

図 6-2. 関数情報

```
4064 ;; *************** function _render *****************
4065 ;; Defined at:
4066 ;;                line 29 in file "draw.c"
4067 ;; Parameters:    Size  Location     Type
4068 ;;                None
4069 ;; Auto vars:     Size  Location     Type
4070 ;;  lll              4   42[BANK0 ] long
4071 ;;  x                2   46[BANK0 ] volatile int
4072 ;;  cp               1   41[BANK0 ] PTR unsigned char
4073 ;;                 -> inputData(2),
4074 ;; Return value:  Size  Location     Type
4075 ;;                None               void
4076 ;; Registers used:
4077 ;;                wreg, fsr0l, fsr0h, status,2, status,0, pclath, cstack
4078 ;; Tracked objects:
4079 ;;                On entry : 17F/0
4080 ;;                On exit  : 0/0
4081 ;;                Unchanged: FFE00/0
4082 ;; Data sizes:     COMMON   BANK0   BANK1   BANK2
4083 ;;      Params:         0       0       0       0
4084 ;;      Locals:         0       7       0       0
4085 ;;      Temps:          0       5       0       0
4086 ;;      Totals:         0      12       0       0
4087 ;;Total ram usage:       12 bytes
4088 ;; Hardware stack levels used:    1
4089 ;; Hardware stack levels required when called:    4
4090 ;; This function calls:
4091 ;;                _lrv
4092 ;;                ___altofl
4093 ;;                ___awdiv
4094 ;;                ___awmod
4095 ;; This function is called by:
4096 ;;                _main
4097 ;; This function uses a non-reentrant model
```

1. function's name
2. file name and line number of definition
3. size, location and type of parameters
4. size, location and type of auto variables
5. size, location and type of return value
6. registers that the function code used
7. selected GPR bank on entry and exit
8. RAM memory summary for entire function
9. hardware stack requirements
10. functions called by this function
11. which functions call this function
12. how the function was encoded

6.3.4 switch 文情報

各 switch 文向けに生成されたコードと一緒に、その命令文がどのようにエンコードされたのかを示す情報が示されます。本コンパイラは switch 文向けに各種のストラテジを使う事ができ、その中から適切なストラテジを選択します(5.5.3 「switch 文」参照)。必要に応じ、プラグマを使ってユーザが特定タイプのストラテジを指定する事もできます(5.13.3.10 「#pragma switch ディレクティブ」参照)。switch 文情報の標準的な例を図 6-3 に示します。

図 6-3. switch 文情報

```
206         ; Switch size 1, requested type "space"
207         ; Number of cases is 4, Range of values is 0 to 254
208         ; switch strategies available:
209         ; Name         Instructions Cycles
210         ; simple_byte           13     7 (average)
211         ; jumptable            260     6 (fixed)
212         ; rangetable           259     6 (fixed)
213         ; spacedrange          516     9 (fixed)
214         ; locatedrange         255     3 (fixed)
215         ;       Chosen strategy is simple_byte
```

1. size of the switched value
2. number and range of the case values
3. all switch strategies and their attributes
4. the strategy choosen for this switch statement

6.3.5 ポインタ参照グラフ

アセンブリ リストファイルに含まれるその他の重要情報として、ポインタ参照グラフがあります。ポインタ参照グラフは、プログラムに含まれる各ポインタと、ポインタがプログラム全体を通して参照可能な各ターゲットを記載したリストです。各ターゲットのサイズと型およびポインタ変数自体のサイズと型が示されます。

例えば下記のコードは、C コード内の task_tmr という名前のポインタを示しています。これは、関数 timer_intr()に対してローカルなポインタです。また、このポインタは unsigned int 型に対するポインタであり、1 バイト幅です。このポインタのターゲットは 1 つだけ存在し、それは task という名前の構造体内のメンバー(timer_count)です。このターゲット変数は BANK0 クラス内に配置され、2 バイト幅です。

```
timer_intr@task_tmr   PTR unsigned int  size(1); Largest target is 2
-> task.timer_count(BANK0[2]),
```

ポインタ参照グラフは、データ オブジェクトに対するポインタと関数に対するポインタの両方を示します。

6.3.6 コールグラフ

コールグラフもアセンブリ リストファイル内の重要情報です。コールグラフは、コンパイルド スタックを使ってスタックベース変数(関数パラメータ、auto 変数、一時変数)を扱う事ができる全ての 8 ビットデバイス向けに生成されます。コンパイルド スタックの詳細は「コンパイルド スタックの動作」を参照してください。

関数ごとの呼び出し情報を示すコールグラフ テーブルがマップファイル内に示されます。このテーブルの後に、プログラム全体に関する従来形式のコールグラフが表示されます。コールグラフはコード ジェネレータによって生成され、コンパイルド スタック内で関数の auto-パラメータ ブロック(APB)のオーバーラップを可能にするために使われます。関数がソフトウェア スタックを使う場合、コールグラフは使われません。コールグラフからは以下の情報が得られます。

- プログラム内で「ルート」ノードとなる(コールツリーのトップに位置し自動的に呼び出される) 関数
- プログラムの実行中に呼び出された(またはポインタを介し間接的に呼び出された可能性がある)とリンカが見なした関数
- プログラム内の関数呼び出しの階層構造
- 各関数の APB 内の auto 変数およびパラメータ領域のサイズ
- コンパイルド スタック内での各関数の APB のオフセット
- コールツリーの推定深さ

これらの機能について以下で説明します。

6.3.6.1 コールグラフのテーブル

コールグラフ テーブルの標準的な例(一部抜粋)を図 6-4 に示します。コールグラフは、リストファイル内の **Call Graph Tables:** の下に示されます。

図 6-4. コールグラフのテーブルの書式

```
 Call Graph Tables:

--------------------------------------------------------------------------------
 (Depth) Function                Calls       Base Space   Used Autos Params    Refs
 ---------------------------------------------------------------------------------
 (0) _main                                                   12    12      0   34134
                                           43 BANK0          5     5      0
                                            0 BANK1          7     7      0
                                _aOut
                              _initSPI
--------------------------------------------------------------------------------
 (1) _aOut                                                    2     0      2      68
                                            2 BANK0          2     0      2
                                _SPI
                    _GetDACValue (ARG)
 --------------------------------------------------------------------------------
 (1) _initSPI                                                 0     0      0       0
 --------------------------------------------------------------------------------
 (2) _SPI                                                     2     2      0      23
                                            0 BANK0          2     2      0
...

 Estimated maximum stack depth 6
 --------------------------------------------------------------------------------
```

コールグラフ テーブルは関数 `main()`で始まります。関数名は常にアセンブリ形式で示されます。従って関数 `main()`は、シンボル`_main` とし示されます。常に `main()`関数が 1 つのコールツリーのルートとなります。割り込み関数は別のツリーを形成します。

`main()`から呼び出される(または呼び出し可能な)全ての関数は、**Calls** 列の下に示されます。図 6-4 の例では、`main()`は `aOut()`と `initSPI()`を呼び出しています。これらは図中のオレンジ色のボックス内に示されています。関数名の隣にアスタリスク(*)が表示される場合、その関数はポインタを介して間接的に呼び出されます。コールグラフには、実際に呼び出された関数だけでなく呼び出される可能性がある関数も含まれます。

例:

```
int test(int a)
    { if(a)
        foo();
    else
        bar();
}
```

上の例では `test()`の下で `foo()`または `bar()`のどちらかが呼び出されます。`a` が常に「真」である場合、関数 `bar()`は決して呼び出されませんが、コールグラフには示されます。

これらの呼び出し先の関数に加えて、`main()`向けにコンパイルド スタック内に割り当てられるメモリの情報も示されます。このメモリは、`main()`内で定義されているスタックベース変数向けに使われる他、必要に応じて関数の戻り値向けにも一時的に使われます。

図中のオレンジ色のボックスには、`main()`が 12 個の auto および一時変数を定義している事が **Autos** 列に示されています。パラメータは何も定義されていません(**Params** 列の値は 0)。

main()内のローカル オブジェクトに対する参照がアセンブリコード内に全部で 34134 個存在する事が **Refs** 列に示されています。**Used** 列には、ローカル記憶域の総バイト数(**Autos** 列と **Params** 列の合計)が示されます。

ターゲット デバイス上の利用可能メモリの全てを活用するために、コンパイルド スタックは 1 つのメモリ空間内の 1 つのメモリブロック内に配置されるのではなく、異なるメモリ空間内の複数のメモリブロック内に分割して配置される可能性があります。この分割状態は、各関数のメモリ使用量サマリ行の下に示されます。この例では、main() 向けの 5 バイトの auto オブジェクトがコンパイルド スタックのバンク 0 コンポーネント内に配置され(**Space** 列)、このスタック内へのオフセットは 43 (**Base** 列)である事が示されています。また、7 バイトの auto オブジェクトがコンパイルド スタックのバンク 1 データ コンポーネント内にオフセット 0 で配置された事が示されています。**Space** 列に示される名前は、スタックのこのセクションを保持するリンカクラスの名前と同じです。

main()に関する情報(オレンジ色のボックス) の下には、main()から呼び出される各関数(aOut()と initSPI())に関する情報が同様に表示されます。分かりやすくするため、図にはこのプログラムの最初に現れる数個の関数しか示していません。

各関数の名前の前には、その関数の呼び出しスタックの深さがカッコ内に示されます。関数は、プログラム内の複数の位置から呼び出される可能性があり、スタック深さはそれらの位置ごとに異なる可能性があります。コールテーブル内の関数の位置に関係なく、各関数に対して常に最大呼び出し深さが示されます。main()関数の呼び出しスタックの深さは常に 0 です。割り込み関数の開始呼び出しスタック深さは、最悪条件を想定して、前のコールグラフ ツリーの開始深さ + 1 とされます。関数が再帰呼び出しを行う場合、スタック深さは**(?)**として示され、関数名の後に**(recursive)**と表示されます。

コールグラフ内の各ツリーの最後に、そのツリーで推定される最大スタック深さが示されます。スタック深さに関するさらに詳細な情報は、スタック ガイダンス機能(5.2.15 「スタック ガイダンス」参照) により提供されます。コールグラフ内のいずれかの関数がソフトウェア スタックを使う場合、スタック深さは表示されません。この場合、コールグラフの最後にプログラム全体の推定スタック深さが 1 つだけ示されます。プログラム内に再帰関数呼び出しが存在する場合、最大スタック深さは unknown として示されます。上の例では、推定最大スタック深さは 6 です。ターゲット デバイスのハードウェア スタックの深さは、そのデバイスのデータシートで確認する必要があります(5.2.4.1 「関数戻りアドレス用スタック」参照)。コールグラフに表示されるスタック深さは、プログラムのスタック使用量を見積もるための目安として使えます。以下の理由により、プログラムのスタック総使用量に関する確定的な値を得る事はできません。

- コールツリーの一部は到達不能であるかもしれません。そうであれば、そのツリーの実際のスタック使用量は減少します。
- main()またはその他のツリーに対する割り込みの位置は基本的に特定不可能です。なぜなら、main()のコールツリー内のどの位置で割り込み(またはその他の関数呼び出し)が発生するかは未知だからです。コンパイラは、最大 main()深さで割り込みが発生するという最悪条件を想定します。
- アセンブラ オプティマイザが関数呼び出しを関数へのジャンプに置き換えたかもしれません。そうであれば、そのツリーのスタック使用量は減少します。
- アセンブラの最適化処理(手続きの抽象化) により、抽象化されたコードへの呼び出しが追加されたかもしれません。そうであれば、スタック深さは増加します。この最適化によって最大スタック深さを超えないかどうかは確認されます。
- インライン化された関数は呼び出されないため、その関数によるスタック使用量は減少します。

PIC10/12/16 デバイス向けにのみ、ハードウェア スタックを管理するようコンパイラを設定できます(4.6.1.23 「stackcall オプション」参照)。このモードが選択されると、コンパイラはデバイスの最大スタック深さを超える可能性があると判断した場合に呼び出しをジャンプへ変換します。コールテーブル内に示される推定スタック深さは、この機能によるスタックの節約分を反映するため、スタック深さと呼び出し深さは一致しません。

実行時起動ルーチンは main()を呼び出すのではなく main()へジャンプするため、main()自体はスタックのレベルを消費しないという事に注意が必要です。

最大スタック深さを超えたと判断され、かつ、スタックがコンパイラによって管理されない場合、コード ジェネレータは警告を生成します。前述の理由により、この警告も、スタックに問題が生じる可能性がある事を示すガイドラインとして提供されます。

6.3.6.2 コールグラフのクリティカル パス

リストファイル内のコールグラフの直前に、コールグラフ内で検出されたメモリ使用のクリティカル パスが示されます。各メモリ空間および各コールグラフ向けにクリティカル パスが表示されます。次のページの例では、「Critical Paths under _main in BANK0」の下に、バンク 0 メモリ内の main()関数(コールグラフのルート) のクリティカル パスが示されています。main()関数向けに 1 つのコールグラフが存在し、各割り込み関数向けにも 1 つずつのコールグラフが存在します。これらの各コールグラフは、デバイスが定義する全てのメモリ空間向けに示されます。

クリティカル パスは、連続する 1 つのブロック内で一緒にスタックされる APB の最大レンジを表します。基本的に、クリティカル パスは、APB がその特定メモリ空間内でプログラムのメモリ使用量に寄与している関数を特定します。

そのメモリ空間内でのこれらの関数のメモリ使用量を削減できれば、そのメモリ空間におけるプログラムの総メモリ使用量を削減できます。

この情報の例を以下に示します。

```
3793 ;; Critical Paths under _main in BANK0
3794 ;;
3795 ;;   _main->_foobar
3796 ;;   _foobar->  flsub
3797 ;;   flsub->  fladd
```

この例は、main()関数から始まる全てのコールグラフ パス(main()が foobar()を呼び出し、foobar()が__flsub()を呼び出し、__flsub()が__fladd()を呼び出すパス) がバンク 0 RAM の最大のメモリブロックを使っている事を示しています。各関数のメモリ使用量は、コールグラフ テーブル内に示されます。

クリティカル パス内に現れない関数によって使われるメモリは、クリティカル パス内の関数によって使われるメモリと完全にオーバーラップします。そのような関数のメモリ使用量を削減しても、プログラム全体のメモリ使用量には影響を及ぼしません。

6.3.6.3 コールグラフの表記

コールテーブルの後にコールグラフが示されます。これは、main()関数と全ての割り込み関数の完全なコールツリーを表示します。これは、コールグラフのテーブルで示された情報の一部を異なる形式で示します。このグラフの一例を以下に示します。

```
 Call Graph Graphs:

_main (ROOT)
  _initSPI
  _aOut
    _SPI
    _GetDACValue
      _  ftadd
        _  ftpack
        _  ftmul (ARG)
...
```

インデントにより呼び出しの深さが示されます。上図では、main()から aOut()が呼び出され、そこからGetDACValue()が呼び出され、さらにそこから ftadd()等が呼び出される事が分かります。関数名の隣にアスタリスク(*) が示される場合、その関数はポインタを介して間接的に呼び出されます。

6.3.6.4 引数(ARG)ノード

コールツリーとコールグラフの両方において、別の関数に対する引数を取得するために呼び出される関数の名前の後に注釈として(ARG)が示されます。例えば、以下のコードでは、関数 input()は関数 process()への引数を取得するために呼び出されています。

```
result = process(input(0x7));
```

このようなコードが main()関数内に現れる場合、コールグラフは以下のように示されます。

```
_main (ROOT)
  _input
  _process
    _input (ARG)
```

これは、main()が input()と process()を呼び出し、さらに input()は process()向けの引数式として呼び出される事を示します。

コールグラフ内のこれらの引数(ARG)ノードは、プログラムが使用する総スタック深さに寄与しませんが、コンパイルド スタックの生成に関与します。前述のツリーでは、引数ノード(ARG)関数は深さ 2 の位置に示されますが、実際に使われるコールスタックの深さは 2 ではなく 1 です。

なぜなら、呼び出しで見ると実際には再入になっておらず、ハードウェアスタックを使うのは戻りアドレスだけだからです。

データの破損を防ぐため、コンパイラは、関数および関数の「引数」向けパラメータ領域がコンパイルド スタック内の一意アドレスに配置される事を保証する必要があります。関数の戻り値もそのパラメータ領域に保存されるため、たとえパラメータが何も存在していなくても、コンパイラはこの事を考慮する必要があります。関数のパラメータは、その関数が実際に呼び出される前(引数が渡される時) に「アクティブ」になり、その関数の戻り値の格納位置は関数が戻った後も戻り値が処理されている間は「アクティブ」なままです。

データ配置の観点から、コンパイラは、関数のパラメータがロードされた時点でその関数が「呼び出された」と見なし、その戻り値が不要になるまで「呼び出し中である」と見なします。従って「再入」の定義は、データ領域割り当てという観点では、スタックの呼び出し深さという観点よりも大幅に広がります。

6.3.7 シンボルテーブル

各アセンブリ リストファイルの最後にシンボルテーブルが示されますこれは、マップファイル内で示されるシンボルテーブル(7.3.2.6 「シンボルテーブル」参照)とは、以下の 2 点で異なります。

- リストファイルの生成元であるアセンブリ モジュールに関連するシンボルだけが示される
- そのモジュールに関連するローカル シンボルとグローバル シンボルの両方が示される

各シンボルは、割り当てられたアドレスと一緒に示されます。

7. リンカ

以下では、リンカの動作と使用法について説明します。

リンカのアプリケーション名は hlink です。コンパイラ ドライバ xc8-cc が必要な全ての引数を指定して自動的にリンカを実行するため、通常はリンカを直接呼び出す必要はありません。リンカの直接使用は複雑であるため、リンク処理要件に関する正確な知識が必要です。psect が正しくリンクされないと、異常動作が発生します。

7.1 動作

リンカに対するコマンドの書式は以下の通りです。

```
hlink [options] files
```

options では何も指定しないか、次に示すリンカ オプション(大文字/小文字を区別)を指定します。各オプションは、リンカの特定の動作を変更します。*files* では、1 つまたは複数のオブジェクトファイルと、必要に応じてライブラリ ファイル(拡張子は.a)を指定します。

リンカによって認識されるオプションを表 7-1 に示します。各オプションの説明は表の後に記載しています。

表 7-1. リンカのコマンドライン オプション

オプション	効果
-8	8086 スタイルのセグメントを使う(オフセット アドレス形式)
-A*class=low-high ,...*	クラスのアドレスレンジを指定する
-C*psect=class*	グローバル psect のクラス名を指定する
-C*baseaddr*	ベースアドレスを指定してバイナリ出力ファイルを生成する
-D*class=delta*	クラスのデルタ値を指定する
-D*symfile*	旧式のシンボルファイルを生成する
-E*errfile*	errfile にエラーメッセージを書き込む
-F	シンボル レコードのみの.o ファイルを生成する
-G *spec*	セグメント セレクタの計算方法を指定する
-H *symfile*	シンボルファイルを生成する
-H+ *symfile*	拡張シンボルファイルを生成する
-I	未定義シンボルを無視する
-J *num*	処理を中止する最大エラー数を設定する
-K	関数パラメータ領域と auto 変数領域のオーバーレイを防ぐ
-L	.o ファイル内で再配置アイテムを保存する
-LM	.o ファイル内でセグメント再配置アイテムを保存する
-N	マップファイル内のシンボルテーブルをアドレス順に並べる
-Nc	マップファイル内のシンボルテーブルをクラスアドレス順に並べる
-Ns	マップファイル内のシンボルテーブルを空間アドレス順に並べる
-M*mapfile*	ファイル名を指定してリンクマップを生成する
-O*outfile*	出力ファイルの名前を指定する
-P*spec*	psect のアドレスとリンク順を指定する

.....続き	
オプション	**効果**
-Q*processor*	マップファイルに表示するデバイスタイプを指定する
-S	シンボルファイル内でのシンボルの一覧表示を禁止する
-S*class=limit*[,*bound*]	psect のクラスに対するアドレス制限と開始境界を指定する
-U*symbol*	指定されたシンボルを未定義シンボルとしてシンボルテーブルに入力する
-V*avmap*	ファイル *avmap* を使って Avocet 形式のシンボルファイルを生成する
-W*warnlev*	警告レベル(-9～9)を設定する
-W*width*	マップファイルの幅(>=10)を指定する
-X	シンボルファイルからローカルシンボルを削除する
-Z	シンボルファイルから自明なローカルシンボルを削除する
--DISL=*list*	無効にするメッセージを指定する
--EDF=*path*	メッセージ ファイルの位置を指定する
--EMAX=*number*	エラーの最大数を指定する
--NORLF	リストファイルの再配置を禁止する
--VER	バージョン番号を出力した後に停止する

標準入力がファイルである場合、そのファイルはコマンドライン引数を格納していると見なされます。コマンドラインは、\を使って複数行に分割できます。この方法により、hlink コマンドをほぼ無制限の長さで指定できます。例えば、以下の内容を格納した x.lnk という名前のリンクコマンド ファイルは、

```
-Z -Ox.o -Mx.map \
-Ptext=0,data=0/,bss,nvram=bss/.\
x.o y.o z.o
```

以下のどちらかの方法でリンカに渡す事ができます。

```
hlink @x.lnk
hlink < x.lnk
```

一部のリンカオプションには、メモリアドレスまたはサイズを指定する必要があります。それらの構文はどれも似ています。既定値により、数値は 10 進値であると見なされます。16 進数としての解釈を要求するには、数値の後に H または h を追加する必要があります。例えば 765FH は 16 進数として扱われます。

通常、リンカはビルド時に MPLAB XC8 C コンパイラ(xc8-cc)または MPLAB XC8 PIC アセンブラ(pic-as)ドライバから呼び出され、既定値リンカ オプションのセットが渡されます。これらの場合、リンカを直接実行しないため、リンカオプションを変更または追加する必要がある場合は、-Wl (4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)または-Xlinker (4.6.11.9 「xlinker オプション」参照)ドライバ オプションを使います。

7.1.1 A: リンカクラスの定義

-A*class=low-high* オプションを使うと、1 つまたは複数のアドレスレンジを 1 つのリンカクラスに割り当てる事ができます。これらのレンジは連続していなくても構いません。

例:

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

上記は、CODE という名前のクラスに、2 つに分離したアドレスレンジを割り当てます。

-P オプションにより、クラス名をアドレスとして使って psect をこれらのレンジ内の任意位置に配置できます(7.1.17 「P: psect を配置する」参照)。
例:

```
-PmyText=CODE
```

あるいは、定義時に class フラグ(6.1.9.36.3 「class フラグ」参照) によって CODE クラスが指定された psect は、自動的にこのレンジ内へリンクされます(別のオプションによって psect が明示的に配置される場合を除く)。

同一サイズの複数の連続したアドレスレンジが存在する場合、それらのレンジの数を指定できます(文字 x の後で指定)。

例:

```
-ACODE=0-0FFFFhx16
```

上記は、アドレス 0 から始まる 16 個の連続したレンジ(各レンジのサイズは 64K バイト)を指定します。これらのレンジは連続していますが、psect は 64K 境界をまたぐ事はできません。従って、以下を指定した場合とでは psect の配置が異なる可能性があります。

```
-ACODE=0-0FFFFFh
```

上記を指定した場合、64K 境界は存在しません。

-A オプションは、アドレスのメモリ空間を指定しません。psect が 1 つのクラスに割り当てられると、その psect に割り当てられているメモリ空間(6.1.9.36.18 「space フラグ」参照) がそのクラスに割り当てられます。

7.1.2 C: psect に対するリンカクラスの割り当て

-C*psect=class* オプションを使うと、psect を特定のクラスに割り当てる事ができます。psect クラスはオブジェクト ファイル内で指定されるため、通常はこのオプションを使って指定する必要はありません(6.1.9.36.3 「class フラグ」参照)。

7.1.3 D: クラスのデルタ値の定義

-D*class=delta* オプションを使うと、指定されたクラスに属する psect の delta 値を定義できます。*delta* 値は数値である事が必要です。この値は、psect 内のオブジェクトのアドレス指定単位(アドレスの増分)をバイト数で示します。大部分の psect では、定義時に delta フラグ値(6.1.9.36.4 「delta フラグ」参照)が指定されるため、このオプションを指定する必要はありません。

7.1.4 D: 旧式のシンボルファイルを定義する

-D*symfile* オプションは、旧式のシンボルファイルを生成するために使います。旧式のシンボルファイルは ASCII ファイルであり、各行でシンボルのリンクアドレスに続いてシンボル名が示されます。

7.1.5 E: エラーファイルを指定する

-E*errfile* オプションを指定すると、リンカはエラーメッセージを画面(既定値のエラー出力先) に表示せずに指定されたファイルに書き込みます。

7.1.6 F: シンボルのみのオブジェクト ファイルを生成する

通常、リンカはプログラムコードおよびデータバイトとシンボル情報を含んだオブジェクト ファイルを生成します。しかし、シンボルのみのオブジェクト ファイルを生成し、その後のリンカ実行時にそのファイルを再使用してシンボル値を提供する事が必要になる場合があります。そのような場合、-F オプションを使うと出力ファイルからデータとコードを取り除き、シンボルレコードだけを残す事ができます。

このオプションは、1 つのプロジェクトの一部 (分割したビルド)を別のプロジェクトと共有する場合に使えます(例: ブートローダとアプリケーションの間での共有)。1 つのプロジェクト向けのファイルをこのオプションを使ってコンパイルする事で、シンボル専用のオブジェクト ファイルを生成し、そのファイルを他のプロジェクト向けのファイルとリンクする事ができます。

7.1.7 G: 代替セグメント セレクタを使う

セグメント化された(またはバンク切り換え型の) psect を使ってプログラムをリンクする場合、リンカは 2 通りの方法でセグメント アドレス(またはセレクタ)を各セグメントに割り当てる事ができます。セグメントは、psect の連続したグループとして定義され、各 psect はグループ内の直前の psect と連結したリンクアドレスとロードアドレスを持ちます。セグメントのセグメント アドレス(またはセレクタ) の値は、リンカによってセグメント型再配置が処理される時に生成されます。

既定値により、セグメント セレクタはそのセグメントのベース ロードアドレスをそのセグメントの再配置アドレス単位で除算する事により生成されます。再配置アドレス単位は、アセンブラ レベルで psect に与えられる reloc フラグに基づきます(6.1.9.36.16 「reloc フラグ」参照)。-G*spec* オプションにより、セグメント セレクタを別の方法で計算できます。-G オプションの引数の例を以下に示します。

```
A/10h-4h
```

A はセグメントのロードアドレスを表し、/ は除算を表します。これは、「psect のロードアドレスを 10 HEX で除算し、そこから 4 HEX を減算する」事を意味します。この計算式は、A の代わりに N 、/の代わりに* (乗算)を使い、定数を減算ではなく加算する事により変更できます。トークン N は、リンカによって割り当てられるセグメントの序数によって置き換えられます。

例:

```
N*8+4
```

これは、「セグメント番号に 8 を乗算し、そこに 4 を加算する」事を意味します。その結果がセグメント セレクタです。この例の場合、定義されたセグメントの番号に対して 4、12、20、...という並びでセグメント セレクタが割り当てられます。

各psect のセレクタはマップファイル内に示されます(7.3.2.2 「モジュールごとのpsect 情報」参照)。

7.1.8 H: シンボルファイルを生成する

-H*symfile* オプションは、シンボル ファイルを生成するようリンカに指示します。引数 *symfile* を任意に指定する事で、データを受け取るファイルの名前を指定できます。既定値のファイル名は l.sym です。

7.1.9 H+: 拡張シンボルファイルを生成する

-H+*symfile* オプションは、拡張シンボルファイルを生成するようリンカに指示します。このファイルは、標準シンボルファイルに加えて、各シンボルに割り当てられたクラス名と、各クラスが使用するメモリレンジを示すセグメント セクションを提供します。この拡張書式ファイルは、コードをデバッガと連動して実行する場合に推奨します。引数 *symfile* を指定する事で、シンボルファイルの名前を指定する事もできます。既定値のファイル名は l.sym です。

7.1.10 I: 未定義シンボルを無視する

通常、未定義シンボルへの参照を解決できないと重大エラーが発生しますが、-I オプションを指定すると未定義シンボルは警告として扱われます。

7.1.11 J: 最大エラー数を指定する

リンカは、特定数のエラー(警告は除く) が発生した時点でオブジェクト ファイルの処理を停止します。既定値の最大エラー数は 10 ですが、-J*errcount* オプションを使って変更できます。

7.1.12 L: ロード時の再配置を許容する

通常リンカは、出力ファイルを生成する時に再配置情報を保存しません(ファイルは absolute であるため)。しかし、ロード時にプログラムのさらなる「再配置」が実行される場合があります。-L オプションを指定すると、入力内の各再配置レコードに対して 1 つの Null 再配置レコードが出力ファイル内で生成されます。

7.1.13 LM: ロード時のセグメント再配置を許容する

-L オプションと同様に、-LM オプションは出力ファイル内で再配置レコードを保存しますが、セグメント再配置のみを保存します。

7.1.14 M: マップファイルを生成する

-M*mapfile* オプションを指定すると、リンカは指定された名前を持つファイル内にリンクマップを生成します(ファイル名が省略された場合、標準出力上でリンクマップを生成します)。マップファイルの書式は 7.3 「マップファイル」に記載しています。

7.1.15 N: シンボルテーブルの並び順を指定する

既定値では、マップファイル内のシンボル テーブルの並び順は名前に基づきます。-N オプションを指定すると、シンボルの値に基づいて並び換える事ができます。-Ns および-Nc オプションも同様の効果を持ちますが、シンボルは空間値またはクラスによってグループ化されます。

7.1.16 O: 出力ファイル名を指定する

このオプションにより、オブジェクト ファイル向けの出力ファイルの名前を指定できます。

7.1.17 P: psect を配置する

psect は、-P*spec* オプションによってリンカに提供された情報に基づいて、互いにリンクされてアドレスが割り当てられます。-P オプションの引数は、コンマで区切られた以下の書式により指定されます。

```
-Ppsect=linkaddr+min/loadaddr+min,psect=linkaddr/loadaddr,...
```

全ての値は省略可能であり、省力された場合は以前の値に応じて既定値が適用されます。psect のリンクアドレスは、実行時にその psect にアクセス可能なアドレスです。ロードアドレスは、出力ファイル(HEX またはバイナリファイル等)内でその psect が始まるアドレスです。8 ビット PIC デバイスでは、ロードアドレスを滅多に使いません。アドレスはアドレス値、他の psect の名前、クラス、特殊なトークンで指定できます。

このオプションへの引数にコンマが含まれる場合があります。このため、このオプションを MPLAB XC8 C コンパイラ(xc8-cc)または MPLAB XC8 PIC アセンブラ(pic-as)ドライバからリンカへ渡す場合、引数内のコンマによって予期せぬ結果が生じる事を防ぐため、-Wl オプション(4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)を使うのではなく、-Xlinker ドライバ オプション(4.6.11.9 「xlinker オプション」参照)を使う事が必要になる場合があります。

このオプションの最も一般的な例を以下に示します。

```
-Ptext10=02000h
```

上記は、psect text10 の開始アドレスを 0x2000 に配置(リンク)します。

```
-PmyData=AUXRAM
```

上記は、psect myData を、リンカクラス AUXRAM のアドレスレンジ内の任意位置に配置します(リンカクラスのアドレスレンジは-A オプション(7.1.1 「A: リンカクラスの定義」参照)で指定する必要があります)。

```
-PstartCode=0200h,endCode
```

上記は、startCode の開始アドレスを 0x200 に配置し、その直後に endCode を配置します。このオプションの追加のバリアントは滅多に必要になりません。以下では、それらのバリアントについて説明します。
特定アドレス以下のリンクアドレスまたはロードアドレスを許容できない場合、接尾辞+*min* により最小アドレスを指定できます。

リンクアドレスが負の値である場合、psect は逆順にリンクされます(トップの psect は指定アドレス - 1 に配置され、次の psect はメモリ内の最初の psect の前に配置されます)。

ロードアドレスを完全に省略した場合、ロードアドレスはリンクアドレスと同じアドレスに設定されます。スラッシュ「/」文字の後でアドレスが何も指定されない場合、ロードアドレスは直前の psect のロードアドレスと連結されます。

例:

```
-Ptext=0,data=0/,bss
```

上記のオプションが処理された後、text psect のリンクアドレスとロードアドレスは 0 となり、data psect のリンクアドレスは 0 でロードアドレスは text に続くアドレスとなります。bss psect のリンクアドレスとロードアドレスは data と連結されます。

ロードアドレスがピリオド「.」として指定された場合、リンカはロードアドレスをリンクアドレスと同じアドレスに設定します。

psect の最終的なリンクアドレスとロードアドレスはマップファイル内に示されます(7.3.2.2 「モジュールごとの psect 情報」参照)。

7.1.18 Q: デバイスを指定する

-Q*processor* オプションを使うと、デバイスのタイプを指定できます。これは、単にマップファイルに表示するための情報として扱われます。このオプションの引数は、デバイスを説明する文字列です。このオプションはリンカの動作に何も影響を及ぼしません。

7.1.19 S: シンボルファイルからシンボル情報を省略する

-S オプションを指定すると、リンカによって生成されるシンボルファイル内のシンボル情報は省略されます。セグメント情報は残されます。

7.1.20 S: クラスに上限アドレスを設定する

- *Sclasslimit*[,*bound*]=オプションを使うと、psect のクラスに対して上限アドレスを設定できます。以下の例は、psect の CODE クラスのアドレスを 400h 未満に制限します。

```
-SCODE=400h
```

psect に対する上限アドレスは、アセンブリコード内で limit フラグ(6.1.9.36.8 「limit フラグ」参照)を使って設定する必要があります。

引数 *bound* (境界)を指定した場合、psect のクラスは bound の倍数のアドレスから始まります。以下の例では、psect の FARCODE クラスは 1000h の倍数アドレスで始まり、6000h 未満に制限されます。

```
-SFARCODE=6000h,1000h
```

7.1.21 U: 未定義シンボルを追加する

-U*symbol* オプションは、指定されたシンボルを、リンカのシンボル テーブル内に未定義シンボルとして入力します。これは、ライブラリに後からリンクする場合や、既定値で後のモジュールがリンクされるように順序が調整されているライブラリからモジュールをリンクする場合に役立ちます。

7.1.22 V: Avocet 形式のシンボルファイルを生成する

Avocet 形式のシンボルファイルを生成する場合、リンカが Avocet メモリ識別子に対して psect 名を割り当てる事ができるようにするため、-V*avmap* オプションを使ってマップファイルを指定する必要があります。通常、*avmap* ファイルはコンパイラと一緒に提供されるか、必要時にコンパイラによって自動的に生成されます。

7.1.23 W: 警告レベルまたはマップファイル幅を指定する

-W*num* オプションを使うと、警告レベル(*num* = -9～9)またはマップファイルの幅(*num* >= 10)を設定できます。

-W9 を指定した場合、全ての警告メッセージが抑制されます。-W0 が既定値です。警告レベルを-9 (-W-9) に設定すると、最も広範囲の警告メッセージが生成されます。

7.1.24 X: シンボルファイルからローカルシンボルを省略する

-X オプションを使うと、シンボルファイルからローカルシンボルを省略できます。グローバル シンボルは常にシンボルファイル内で示されます。

7.1.25 Z: シンボルファイルから自明なシンボルを省略する

コンパイラによって生成される一部のローカルシンボルは、デバッグには無関係です。-Z オプションを使うと、アルファベット k、l、f、L、S、u のいずれか 1 文字の後に数字列が続く全てのローカルシンボルをシンボルファイルから省略できます。

7.1.26 disl

--disl=*messages* オプションは、特定番号のメッセージを無効にするために、主にコマンドライン ドライバ xc8-cc によって使われます。*messages* は、コンパイル中に無効にするメッセージ番号のコンマで区切られたリストです。

コンパイラのメッセージ システムの詳細は、4.5 「コンパイラ メッセージ」を参照してください。

7.1.27 edf

--edf=*file* オプションは、メッセージ記述ファイルのパスを指定するために、主にコマンドライン ドライバ xc8-cc によって使われます。既定値ファイルは、コンパイラのインストール ディレクトリ内の dat ディレクトリ内に格納されます。

コンパイラのメッセージ システムの詳細は、4.5 「コンパイラ メッセージ」を参照してください。

7.1.28 emax

--emax=*number* オプションは、エラーの許容最大数を指定するために、主にコマンドライン ドライバ xc8-cc によって使われます。エラーの数がこの数を超えると、アセンブラは停止します。既定値のエラー数は 10 です。

このオプションは、コマンドライン ドライバ xc8-cc で-fmax-errors ドライバ オプションを使ってコンパイルする場合に適用されます。コンパイラのメッセージ システムの詳細は、4.5 「コンパイラ メッセージ」を参照してください。

7.1.29 norlf

--norlf オプションを使うと、アセンブラによって生成されるアセンブリ リストファイルに対してリンカがフィックスアップを適用する事を防ぐ事ができます。通常このオプションは、プレリンクの実行時にコマンドライン ドライバ xc8-cc によって使われますが、最終リンクの実行時には省略されます(これにより、リストファイルには最終的な絶対アドレスが示されます)。
フィックスアップ エラーの解決を試みる場合、アセンブリ リストファイルをフィックスアップして絶対アドレスが計算できるようにするため、このオプションを無効にする必要があります。コンパイラ ドライバが 1 に設定されたプリプロセッサ マクロ__DEBUG を検出した場合、このオプションはビルド時に無効になります。このマクロは、MPLAB X IDE 内でデバッグビルドを選択した時に 1 に設定されます。従って、フィックスアップ エラーに遭遇する場合は、常にこのオプションが選択されています。

7.1.30 ver

--ver オプションを指定すると、リンカのバージョンとビルドを示す情報が出力されます。コマンドラインで他のオプションとファイルが指定されていても、リンカはこのオプションを処理した後に停止します。

7.2 psect と再配置

リンカは再配置可能オブジェクト ファイル(拡張子は.o)とオブジェクト ファイル(拡張子は.a) の両方を読む事ができます。ライブラリ ファイルは、複数のオブジェクト ファイルを 1 つのファイルに梱包した物であり、開梱後は個々のオブジェクト ファイルと同様の方法で処理されます。

各オブジェクト ファイルは複数のレコードを含みます。各レコードには、保持している情報の種類に応じた型が割り当てられます。ターゲット デバイスとそのコンフィグレーションに関する一般情報を保持する型、データを保持可能な型、プログラム デバッグ情報を保持する型等があります。

オブジェクト ファイル内の多くの情報は psect (プログラム セクション) に関連します。psect はアセンブリコード領域のコンストラクタであり、命令またはデータを格納するブロックです。プログラムに寄与する全ての物が psect 内に配置されます。psect の概要は 6.1.8 「プログラム セクション」を参照してください。psect 内でデータを保持するために使われる特定のレコード型が存在します。各オブジェクト ファイルの大部分は、実行可能コードと一部のオブジェクトを格納した psect レコードで構成されます。

リンカは以下のタスクを実行します。

- 参照される全ての再配置可能オブジェクト ファイルの内容を 1 つにまとめる。
- オブジェクト ファイルに格納されている psect を利用可能なデバイスメモリ内へ再配置する。
- psect 内のシンボル参照をフィックスアップする。

再配置により、psect をターゲット デバイスのメモリへ割り当てます。

ターゲット デバイスのメモリ仕様は、リンカ オプションを介してリンカに渡されます。これらのオプションは、コマンドライン ドライバ xc8-cc により生成されます。ソースファイル内でオプションを指定するためのリンカスクリプトや手段は存在しません。既定値のリンカ オプションを調整する必要はほとんどありません。必要に応じてドライバ オプション-Wl (4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)を使ってリンカ オプシ ョンを変更できますが、注意が必要です。

psect が最終的なメモリ位置に配置された後に、psect 内のシンボル参照を絶対アドレス値に置き換える事ができます。この処理をフィックスアップと呼びます。

リンカは 1 つのオブジェクト ファイルを出力します。再配置は完了済みであり、全てのコードとオブジェクトにはアドレスが割り当て済みであるため、このオブジェクト ファイルは絶対アドレス化済みです。

7.3 マップファイル

マップファイルは、psect のメモリ割り当てに関する情報と、psect 内のシンボルに割り当てられたアドレスを示します。

7.3.1 マップファイルの生成

MPLAB X IDE を使ってコンパイルする場合、マップファイルは既定値により生成されます。コマンドラインからドライバを使う場合、マップファイルの生成を要求するために-Wl,-Map オプションを使います(4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)。通常マップファイルには拡張子.mapが与えられます。

マップファイルはリンカ アプリケーションにより生成されます。リンカが実行される前にビルドが停止した場合、マップファイルは生成されません。リンカでエラーが発生してもマップファイルは生成されます。このマップファイルは未完成ですが、エラーの原因究明に役立ちます。しかし、エラーの数が多すぎたり重大エラーが発生したためにリンカが実行を完了できなかった場合、マップファイルは生成されません。-fmax-errors オプションにより、リンカが停止するまでに許容する最大エラー数を増やす事ができます。

7.3.2 マップファイルの内容

マップファイルは以下のセクションで構成されます。

- コンパイラ名とバージョン番号
- リンカを呼び出すために使われたコマンドラインのコピー
- リンクされた最初のファイル内のオブジェクト コードのバージョン番号
- デバイスタイプ
- ペアレント オブジェクト ファイルごとの psect サマリ
- CLASS ごとの psect サマリ
- セグメント サマリ
- 未使用アドレスレンジのサマリ
- シンボルテーブル
- 各関数の情報サマリ
- 各モジュールの情報サマリ

次の 7.3.2.1 「一般情報」には、マップファイルの例 (一部抜粋)とその説明を記載しています。

7.3.2.1 一般情報

マップファイルの先頭には、リンカの実行に関連する一般情報が示されます。

プログラムを解析する際は、必ずマップファイルの最上段に表示されるコンパイラのバージョン番号を見て正しいコンパイラが使われた事を確認する必要があります。

-mcpu オプション(4.6.1.5 「cpu オプション」参照)で選択されたデバイスまたは IDE 内で選択されたデバイスが **Machine type** として表示される事が必要です。

Object code version は、アセンブラによって生成された再配置可能オブジェクト ファイルのファイル フォーマットに対応します。アセンブラまたはリンカが別々に更新されていなければ、これに注意を払う必要はありません。

標準的なマップファイルの例(一部抜粋)を以下に示します。

```
Linker command line:
--edf=/Applications/Microchip/XC8/2.20/dat/en_msgs.txt -cs -h+main.sym -z \
-Q16F946 -ol.o -Mmain.map -ver=XC8 -ACONST=00h-0FFhx32 \
-ACODE=00h-07FFhx4 -ASTRCODE=00h-01FFFh -AENTRY=00h-0FFhx32 \
-ASTRING=00h-0FFhx32 -ACOMMON=070h-07Fh -ABANK0=020h-06Fh \
-ABANK1=0A0h-0EFh -ABANK2=0120h-016Fh -ABANK3=01A0h-01EFh \
-ARAM=020h-06Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh \
-AABS1=020h-07Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh -ASFR0=00h-01Fh \
-ASFR1=080h-09Fh -ASFR2=0100h-011Fh -ASFR3=0180h-019Fh \
-preset_vec=00h,intentry,init,end_init -ppowerup=CODE -pfunctab=CODE \
-ACONFIG=02007h-02007h -pconfig=CONFIG -DCONFIG=2 -AIDLOC=02000h-02003h \
-pidloc=IDLOC -DIDLOC=2 -AEEDATA=00h-0FFh/02100h -peeprom_data=EEDATA \
-DEEDATA=2 -DCODE=2 -DSTRCODE=2 -DSTRING=2 -DCONST=2 -DENTRY=2 -k \
startup.o main.o

Object code version is 3.10

Machine type is 16F946
```

Linker command line: の下には、 直近のビルド向けにリンカに渡された全てのコマンドライン オプションとファイルが示されます。これらはリンカ オプションであってコマンドライン ドライバ オプションではない事に注意してください。

リンカ オプションは複雑ですが、これらの既定値設定からの変更が必要になる事はほとんどありません。リンカ オプションは、選択されたターゲット デバイスと指定されたドライバ オプションに基づいて、コマンドライン ドライバ xc8-cc によって設定されます。多くの場合、マップファイル内でリンカ オプションを調べる事により、ドライバ オプションが正しかった事を確認できます。例えば、ドライバ オプションによってメモリのある領域を予約するよう指定した場合、リンカ オプションが変更された事をマップファイル内で確認できます。

既定値のリンカ オプションを変更する必要がある場合、ドライバ オプション-Wl (4.6.11.8 「Wl: リンカに直接オプションを渡すためのオプション」参照)を使って、ドライバから間接的に変更できます。このオプションを使った場合、マップファイルの内容が正しく変更された事を常に確認する必要があります。

7.3.2.2 モジュールごとの psect 情報

マップファイル内の次のセクションには、出力に寄与したモジュールのリストと、これらのモジュールによって定義された psect に関する情報が示されます。

このセクションは、以下に示す行で始まります。

```
Name   Link Load Length Selector  Space  Scale
```

この行の下の左端(**Name** 列) にオブジェクト ファイル(拡張子は.o) のリストが表示されます。ソースモジュールから生成されたオブジェクト ファイルとオブジェクト ライブラリ ファイル(拡張子は.a) から抽出されたオブジェクト ファイルの両方が示されます。後者の場合、ライブラリ ファイルの名前がオブジェクト ファイルリストの前に示されます。

オブジェクト ファイル名の右(**Name** 列) に、そのオブジェクト ファイルからプログラム内へリンクされた psect が示されます。その psect に関する情報がその右側の各列に示されます。

リンカは、2 種類のアドレス (リンクアドレスとロードアドレス)を扱います。一般的に言うと、psect のリンクアドレス(**Link** 列) は、実行時にその psect にアクセスするためのアドレスです。

ロードアドレス(**Load** 列) は出力ファイル(HEX またはバイナリファイル) 内での psect の開始アドレスであり、多くの場合リンクアドレスと同じです。psect がビット値を保持するために使われる場合、ロードアドレスは無関係であり、代わりにバイトアドレスへと変換されたリンクアドレス (ビット単位のアドレス)を保持するためにロードアドレスが使われます。

psect の長さ(**Length** 列) は、その psect で使われる単位で示されます。

Selector 列は一般的にあまり使われず、PC デバイス向けにコンパイルする場合は無関係です。

Space 列は重要であり、その psect が配置されたメモリ空間を示します。複数の分離されたメモリ空間を備えるハーバード アーキテクチャ デバイス(PIC10/12/16) の場合、この Space 列とアドレスの組み合わせにより正確な保存位置が指定されます。Space 列の値 0 はプログラムメモリを示し、値 1 はデータメモリを示します(6.1.9.36.18 「space フラグ」参照)。

Scale 列は、1 バイトあたりのアドレス単位の数を示します。空白はスケール値が 1 である事を示します。ビット オブジェクトを保持する psect のスケール値は 8 です。ビット オブジェクトを保持する psect の **Load** 列は、ロードアドレスを示すのではなくバイトの単位に変換されたリンクアドレスを示します(6.1.9.36.2 「bit フラグ」参照)。

以下に、マップファイル内の表示例を示します。

```
Name     Link  Load  Length Selector  Space  Scale
ext.o    text    3A    3A   22        30     0
         bss     4B    4B   10        4B     1
         rbit    50  A       2         0     1         8
```

これは、リンカが処理したファイルの 1 つが ext.o という名前を持つ事を示しています。

このオブジェクト ファイルは 3 つの psect (text 、bss、rbit)を含んでいます。
text psect はアドレス 3A でリンクされ、bss psect はアドレス 4B でリンクされています。text の Length が 22 ワードであるため、問題があるように見えます。しかし text と bss は異なるメモリ空間内に配置されているため(text の Space は 0、bss の Space は 1)、問題はありません。

rbit psect はビット オブジェクトを格納しており、これは Scale 値 (=8) により確認できます。rbit は bss の上にリンクされており、これ問題があるかのように見えます。rbit と bss の **Space** 列の値は同じですが、rbit はビット オブジェクトを格納するためそのリンクアドレスの単位はビットです。rbit の **Load** 列は、バイト単位に変換されたリンクアドレス(50h/8 => Ah)を示しています。

オブジェクト ファイルリストの下にラベル **COMMON** が表示される場合があります。これは、プログラム全体の psect からプログラムへの寄与を示します。C 言語ベースのプロジェクトの場合、ここにはコンパイルド スタックに関する情報が示されます。

7.3.2.3 クラスごとの psect 情報

マップファイル内の次のセクションにも前述と同じ psect 情報が示されますが、このセクションではそれらの情報がクラスごとに示されます。このセクションは、以下に示す行で始まります。

```
TOTAL   Name  Link  Load  Length
```

この行の下に、クラス名とそのクラスに属する psect の情報が示されます(6.1.9.36.3 「class フラグ」参照)。このセクションの情報は、表示の仕方が異なるだけで前述のモジュールごとの psect 情報と同じであり、追加された情報はありません。

7.3.2.4 セグメントリスト

マップファイル内の次のセクションには、セグメントのリストが示されます。通常、ユーザはこのセクションを無視して構いません。

セグメントとは同じメモリ空間内に連続して配置された psect の概念的なグループであり、リンカは psect を配置するための手掛かりとしてセグメントを使います。セグメントに関連するアセンブラ ディレクティブは存在せず、セグメントを制御する事は一切できません。

このセクションは、以下に示す行で始まります。

```
SEGMENTS   Name Load Length  Top  Selector  Space  Class
```

セグメントの名前には、連続したグループ内で最も低いリンクアドレスを持つ psect の名前が使われます。従って、同じ名前を持つ psect とセグメントが存在するため、混乱を招く恐れがあります。マップファイルのこのセクションから psect 情報を読み取らないでください。

繰り返しますが、マップファイルのこのセクションの内容は無視して構いません。

7.3.2.5 未使用アドレスレンジ

マップファイル内の次のセクションには、何も割り当てられて**いない**(従ってまだ利用可能な)メモリのサマリが示されます。このセクションは以下の行で始まります。

```
UNUSED ADDRESS RANGES
```

この下に、クラスとそのクラス内でまだ利用可能なメモリのリストが示されます。1 つのクラス内に複数の利用可能メモリレンジが存在する場合、各レンジは別々の行に示されます。クラス内に存在するページ境界は示されません。しかし、**Largest block** 列には連続した最大の空き空間が示されます。これには、そのメモリレンジ内のページ境界が考慮されます。psect をメモリ内に配置できなかった場合(例: cant-find-space 型エラーが発生した場合)、その理由を調べるためにこの情報が役立ちます。

メモリは複数のクラスに属する場合があるため、未使用レンジの単純な合計は空き空間の実際の総量とは異なります。

7.3.2.6 シンボルテーブル

マップファイル内の次のセクションには、プログラムが定義したグローバル シンボルがアルファベット順に示されます。このセクションは以下の行で始まります。

```
Symbol Table
```

このテーブルには以下のシンボルが示されます。

- グローバル アセンブリラベル
- グローバル EQU/SET アセンブラ ディレクティブ ラベル
- リンカによって定義されたシンボル

アセンブリ シンボルは GLOBAL アセンブラ ディレクティブ(6.1.9.21 「GLOABAL ディレクティブ」参照) によりグローバルとなります。

リンカによって定義されたシンボルは EQU ディレクティブと同様に振る舞います。しかしそれらのシンボルはリンク処理中にリンカによって定義されたシンボルであり、どのソースファイルまたは中間生成ファイルにも現れません(5.14.6 「リンカによって定義されるシンボル」参照)。

各シンボルには、そのシンボルが定義されている psect とそのシンボルに割り当てられた値(通常はアドレス) が一緒に示されます。そのシンボルがコードまたはデータのどちらを表すのかを示す情報も、どのメモリ空間内に配置されているのかを示す情報も、シンボルに対してエンコードされません。

シンボルの psect が(abs)として示される場合、そのシンボルは psect に直接関連付けられていません。絶対アドレス C 変数またはアセンブリコード内で EQU ディレクティブを使って定義されたシンボルがこれに該当します。

シンボルテーブルは各アセンブラ リストファイルにも示されます。マップファイル内に示されるシンボルテーブルとは異なり、それらシンボル テーブルには、対応するモジュール内で定義されたシンボルだけが表示されます(ローカルシンボルも表示)。

7.3.2.7 関数情報

シンボルテーブルの次に、プログラム内の各関数に関連する情報が示されます。この情報は、アセンブリ リストファイル内に示される関数情報と同じです。しかし、マップファイルには、全ての関数の情報が 1 箇所にまとめて表示されます。

この情報の詳細は、6.3.3 「関数情報」を参照してください。

7.3.2.8 モジュール情報

マップファイル内の最後のセクションには、各モジュールのコード使用量サマリが示されます。プログラム内の各モジュール向けに表示される情報の例を以下に示します。

```
Module      Function          Class         Link    Load    Size
main.c
            init         CODE               07D8    0000    1
            main         CODE               07E5    0000    13
            getInput     CODE               07D9    0000    4
main.c estimated size:18
```

上の例には、モジュール名 main.c が示されています。特別なモジュール名 shared は、プログラム メモリに割り当てられたデータ オブジェクトと、特定のモジュールに固有ではないコード用に使われます。

次に、各モジュールによって定義されたユーザ定義関数とライブラリ関数が示されます。各関数の右側には、psect のクラス、psect のリンクアドレスとロードアドレス、そのサイズ(PIC18 デバイスの場合はバイト数、その他の 8 ビットデバイスの場合はワード数) が示されます。

関数リストの後に、そのモジュールによって使われるプログラムメモリの推定サイズが示されます。

8. ユーティリティ

以下では、本コンパイラに同梱されるユーティリティ アプリケーションの一部について説明します。

ここには比較的よく使われるアプリケーションを記載していますが、通常それらを直接実行する必要はありません。それらの機能の一部は、コマンドライン引数または MPLAB X IDE 内のプロジェクト プロパティの選択に基づいてコマンドライン ドライバから間接的に呼び出されます。

8.1 アーカイバ/ライブラリアン

アーカイバ/ライブラリアン プログラムは、各種の中間生成ファイルから 1 つのファイル(ライブラリ アーカイブ ファイルと呼ぶ)を生成する機能を備えています。ライブラリ アーカイブは個々のファイルを扱うよりも容易に管理でき、保存容量も節約できます。

アーカイバは、コンパイラが必要とする全てのライブラリ アーカイブ タイプをビルドでき、既存アーカイブの書式を検出できます。

8.1.1 アーカイバ/ライブラリアンの使い方

アーカイバ プログラム(xc8-ar) は、ライブラリ アーカイブ ファイルの作成と編集用に使います。基本的なコマンド書式は以下の通りです。

```
xc8-ar [options] file.a [file.p1 file.o ...]
```

file.a は、作成または編集するライブラリ アーカイブ ファイルを指定します。

file.a の後に、必要に応じて、指定されたコマンドによって要求されるモジュール(p-code (.p1)またはオブジェクト(.o)モジュール等) が続きます。C ソースからビルドされた内容に対しては p-code モジュールを使い、アセンブリソースからビルドされた内容に対してはオブジェクト モジュールを使う必要があります。

options は、アーカイバ/ライブラリアンを制御するためのオプションです。表 8-1 にそれらのオプションを示します。

表 8-1. アーカイバ コマンドライン オプション

オプション	効果
-d *modules*	モジュールの削除
-m *modules*	モジュールの並び換え
-p	モジュールのリスト表示
-r *modules*	モジュールの置き換え
-t	モジュールとシンボルのリスト表示
-x *modules*	モジュールの抽出
--target *device*	ターゲット デバイスの指定

モジュールを置換または抽出する場合、モジュールの名前を指定する必要があります。名前が指定されないと、アーカイブ内の全てのモジュールが置換または抽出されます。

アーカイブ ファイルの作成または既存アーカイブへのファイルの追加は、アーカイブ内のモジュールを置き換えるようアーカイバに要求する事により実行されます。アーカイブ内にそれらのモジュールが存在しない場合は、アーカイブへの追加となります。

オブジェクト(拡張子は.o) および p-code (拡張子は.p1) モジュールを同じアーカイブに追加する事ができます。

アーカイバは、各モジュールをコマンドラインで指定された順番に取り込んでライブラリ アーカイブを作成します。アーカイブの更新時も、モジュールの順番は保持されます。新しいモジュールは最後尾に追加されます。

アーカイブ内のモジュールの順番は、リンカに対して重要な意味を持ちます。アーカイブ内のあるモジュールが同じアーカイブ内の他のモジュールによって定義されたシンボルを参照する場合、シンボルを定義しているモジュールは、シンボルを参照しているモジュールよりも順番が後である事が必要です。

-d オプションを使うと、指定されたモジュールはアーカイブから削除されます。このオプションでモジュール名を指定しないとエラーが発生します。

-p オプションは、アーカイブ ファイル内のモジュールのリストを表示します。

-m オプションは、コマンドラインからモジュール名のリストを取得し、アーカイブ ファイル内のそれらのモジュールの順番を、コマンドラインで指定され順番に並び換えます。コマンドラインで指定されなかったモジュールは、順番が変更されたモジュールの後に、元の順番のまま残されます。

8.1.1.1 例

以下では、ライブラリアンの使用例を示します。

```
xc8-ar -r myLib.a ctime.p1 init.p1
```

上記のコマンドは、モジュール ctime.p1 および init.p1 を格納した myLib.a という名前のライブラリを作成します。

下記のコマンドは、ライブラリ lcd.a からオブジェクト モジュール a.p1 を削除します。

```
xc8-ar -d lcd.a a.p1
```

8.2 Hexmate

Hexmate アプリケーションは、Intel HEX ファイル向けに設計されたリンク後処理用ユーティリティです。

Hexmate はコンパイラ ドライバによって自動的に呼び出されますが、スタンドアロン アプリケーションとして実行する事も可能です。

8.2.1 Hexmate の使い方

Hexmate は、Intel HEX ファイル向けの各種タスクを実行します。これには以下が含まれます。

- 複数の Intel HEX ファイルを 1 つの Intel HEX ファイルにマージする
- 可変長ハッシュ値(CRC、SHA 等)を計算して保存する
- 未使用メモリ位置へ既知のデータシーケンスを書き込む
- INHX32 ファイルから他の INHX フォーマット(INHX8M 等)へ変換する
- HEX ファイル内の特定の(または部分的な) オペコード シーケンスを検出する
- 特定の(または部分的な) オペコード シーケンスを検索/置換する
- HEX ファイル内で使われているアドレスのマップを提供する
- HEX ファイル内のデータレコード長を変更または固定する
- Intel HEX ファイル内のチェックサムを検証する

Hexmate の代表的な応用例には以下が含まれます。

- ビルド時にブートローダまたはデバッグ モジュールをメイン アプリケーション内へマージする
- プログラムメモリの特定レンジに対してハッシュ値を計算し、その値をプログラムメモリまたは EEPROM に保存する
- 未使用メモリ領域に命令(誤ってフェッチした際に既知の位置へプログラム カウンタを進める命令等)を書き込む
- 指定されたアドレスにシリアル番号を書き込む
- 指定されたアドレスに文字列(タイムスタンプ等)を保存する
- 特定のメモリアドレスに初期値を保存する(例: EEPROM の初期化)
- 不適正な命令や制限されている命令を検出する
- 特定ブートローダの要件に適合するように HEX ファイルを調整する

8.2.2 Hexmate のコマンドライン オプション

Hexmate は、コマンドラインで指定された HEX ファイルをプログラムのソースファイルから生成された出力とマージするために、コマンドライン ドライバ xc8-cc によって呼び出されます。 Hexmate を直接実行しなくても、コンパイラ ドライバ オプションを使ってその他の Hexmate 機能の一部を要求する事はできますが、以下で説明するオプションを使ってコマンドライン上で Hexmate を実行する事により、完全な制御が可能となります。

以下のコマンド書式により、Hexmate を直接実行できます。

```
hexmate [specs,]file1.hex [...[specs,]fileN.hex] [options]
```

file1.hex～*fileN*.hex は、Hexmate を使ってマージする入力 Intel HEX ファイルのリストです。
options は、コマンドライン上のどこで指定しても構いません。これらのオプションを表 8-2 に示します。

HEX ファイルを 1 つだけ指定した場合、マージには無意味ですが、オプションによって指定されたアクションを HEX ファイルに対して実行できます。

Hexmate は、レコードタイプの特定のサブセットのみを格納した一般的な 8/16/32 ビットに対応する書式(インテル HEX)を読み書き可能です。フォーマットについては、8.2.2.16 「format」で説明します。

表 8-2. Hexmate のコマンドライン オプション

オプション	効果
--edf=*file*	メッセージ記述ファイルを指定します。
--emax=*n*	エラーの許容最大数を設定します(エラーの数がこの値を超えると実行は停止します)。
--msgdisable=*number*	指定された番号のメッセージを無効にします。
--sla=*address*	タイプ 5 レコードの開始リニアアドレスを設定します。
--ssa=*address*	タイプ 3 レコードの開始セグメント アドレスを設定します。
--ver	バージョンおよびビルド情報を表示した後に停止します。
-addressing=*units*	全ての Hexmate オプションのアドレス フィールドに適用するアドレス単位を設定します。
-break	新しいレコードが設定されたアドレスで開始できるよう連続データを分割します。
-ck=spec	ハッシュ値を計算して保存します。
-fill=*spec*	未使用メモリ位置に既知の値を書き込みます。
-find=*spec*	特定のコードシーケンスを検索します。
-find=*spec*,delete	特定のコードシーケンスを検索し、見つかれば削除します(使用には注意が必要)。
-find=*spec*,replace=*spec*	コードシーケンスを新しいコードシーケンスで置換します。
-format=*type*	最大データレコード長を指定または INHX バリアントを選択します。
-help	全てのオプションを表示または特定オプションのヘルプ メッセージを表示します。
-logfile=*file*	出力と各種結果の Hexmate 解析をファイルに保存します。
-mask=*spec*	メモリレンジとビットマスクの論理積(AND)を取ります。
-o*file*	出力ファイルの名前を指定します。
-serial=*spec*	シリアル番号またはコードシーケンスを指定アドレスに保存します。
-size	結果として得られる HEX イメージ内のデータのバイト数を報告します。
-string=*spec*	ASCII 文字列を指定アドレスに保存します。
-strpack=*spec*	文字列パッキングを使って ASCII 文字列を指定アドレスに保存します。
-w*level*	警告の感度を調整します。
+	必要に応じてオプションの前に「+」を付ける事で、そのアドレスレンジ内の他のデータを上書きできます。

ドライバ xc8-cc または IDE を使ってプロジェクトをコンパイルする場合、ログファイルが既定値により生成されます。ログファイルの名前はプロジェクトの名前と同じであり、拡張子は.hxl です。

© 2022 Microchip Technology Inc. and its subsidiaries

オプションのフォーマットまたは基数については、各オプションの説明を参照してください。これらのオプション内で指定される全てのアドレス フィールドは、HEX ファイル アドレスとして入力されるという事に注意が必要です。これは、-addressing オプションを使って変更可能です。

8.2.2.1 処理指定とファイル名

Hexmate は、INHX32 または INHX8M フォーマットを使う Intel HEX ファイルを処理可能です。コマンドライン上で指定される各 HEX ファイルに追加の処理指定を適用する事で、それらのファイルの処理方法に対して制約または条件を設定できます。

それらの処理指定は、ファイル名の前で指定する必要があります。処理指定のリストとファイル名の間はコンマで区切ります。

処理指定 r*Start-End* を使うと、レンジ制限を適用できます。*Start* と *End* は、どちらも 16 進値であると見なされます。Hexmate は制限されたアドレスレンジ内のデータのみを処理します。

例:

```
r100-1FF,myfile.hex
```

上記は myfile.hex を入力として使い、そのファイルから 0x100～1FF (上限/下限アドレスを含む) のアドレスレンジ内にあるデータのみを処理します。

処理指定 s*Offset* により、アドレスシフトを適用できます。*Offset* は、16 進値であると見なされます。アドレスシフトを使う場合、この HEX ファイルから読み出されたデータは、出力ファイル内で新しいアドレスへシフトされます(シフト量は Offset で指定)。Offset は、正または負の値で指定できます。

例:

```
r100-1FFs2000,myfile.hex
```

上記は、myfile.hex 内の 0x100～1FF からのデータブロックを、出力ファイル内の新しいアドレスレンジ 0x2100～21FF へシフトします。

実行可能コードのセクションのシフトには注意が必要です。プログラム コードは、それが位置に依存しない場合にのみシフトさせても構いません。

8.2.2.2 上書き接頭辞

入力ファイル名またはオプションの前に+ 演算子を付けると、そのファイルからのデータまたはそのオプションによって生成されるデータは優先的に出力ファイルに書き込まれます(同じアドレス上に存在する全てのデータを上書きします)。

例えば、input.hex はアドレス 0x1000 にデータを格納しているとします。

```
input.hex +-string@1000="My string"
```

上記を指定した場合、-string オプションで指定されたデータが出力ファイル内のアドレス 0x1000 に書き込まれます。

```
+input.hex -string@1000="My string"
```

上記を指定した場合、input.hex 内のアドレス 0x1000 に格納されているデータが最終出力にコピーされます。

このオプションが指定されていない場合、2 つのソースが同じ位置に異なるデータを保存しようと試みると Hexmate はエラーを発行します。

8.2.2.3 edf

--edf=*file* は、警告またはエラーメッセージを表示する際に使うメッセージ記述ファイルを指定します。このオプションの引数は、メッセージ ファイルへのフルパスで指定する必要があります。Hexmate はメッセージ ファイルの内部コピーを使うため、通常このオプションを指定する必要はありません。 このオプションは、内容が更新された別のファイルを指定する場合に使います。

メッセージ ファイルは MPLAB XC8 C コンパイラに同梱され、コンパイラの pic/dat ディレクトリに保存されています。英語版のファイル名は en_msgs.txt です。

8.2.2.4 emax

--emax=*num* オプションは Hexmate が表示するエラーの最大数を設定します。エラーがこの数を超えると、Hexmate は実行を終了します(設定例: --emax=25)。既定値では、最大 20 個のエラーが表示されます。

8.2.2.5 msgdisable

--msgdisable=*number* オプションを使うと、Hexmate の実行中にエラー、警告、アドバイザリ メッセージを無効にできます。

無効にするメッセージは、コンマで区切ったメッセージ番号のリストとして指定します。このリスト内の全てのエラーメッセージ番号は、それらの後に:off 引数が指定されない限り、無視されます。

例:

```
--msgdisable=2031,963
```

メッセージリストが 0 として指定された場合、全ての警告は無効になります。

8.2.2.6 sla

--sla=*address* オプションを使うと、INHX32 または INHX032 出力ファイル内のタイプ 5 レコード内でリニア開始アドレス(SLA)を指定できます。例えば、--sla=0x10000 を指定した場合、出力 HEX ファイルはタイプ 5 レコード (ペイロード 0x10000)を格納します。

例:

```
:040000**05**00010000F6
```

このオプションが使われた時、入力ファイル内の全ての入力 SLA レコードの構文とチェックサムが確認されます。それらが有効ではない場合、エラーが発行されます。その後、これらの SLA レコードは警告メッセージ**なし**で廃棄されます。出力ファイルのフォーマットが INHX32 または INHX032 ではない場合、警告が発行され、SLA レコードは書き込まれません。フォーマットが INHX32 または INHX032 である場合、オプションによって指定された値を格納した 1 つの SLA レコードだけが出力ファイルに書き込まれます。

このオプションが使われない場合、入力ファイル内の全ての入力 SLA レコードの構文とチェックサムが確認されます。これらのレコードによって指定されたアドレスの間に矛盾がなければ、そのアドレスで唯一の SLA レコードが出力に書き込まれます。入力ファイル内の SLA レコードの間に競合がある場合、警告メッセージが出力され、SLA レコードは出力に**書き込まれません**。入力ファイル内に SLA レコードが存在しない場合、SLA レコードは出力に書き込まれません。

8.2.2.7 ssa

--ssa=*address* オプションを使うと、INEX16 ファイル内のタイプ 3 レコード内のセグメント開始アドレス(SSA)を指定できます。例えば--ssa=0x10000 を指定すると、出力 HEX ファイルはタイプ 3 レコード(ペイロードは 0x10000)を格納します。

例:

```
:040000**03**00010000F8
```

このオプションを使う場合、入力ファイル内の全ての SSA レコードの構文とチェックサムが確認されます。それらが有効ではない場合、エラーが発行されます。その後、これらの SSA レコードは警告メッセージ**なし** で廃棄されます。出力ファイルのフォーマットが INHX16 ではない場合、警告が発行され、SSA レコードは書き込まれません。フォーマットが INHX16 である場合、オプションによって指定された値を格納した 1 つの SSA レコードだけが出力ファイルに書き込まれます

このオプションを使わない場合、入力ファイル内の全ての入力 SSA レコードの構文とチェックサムが確認されます。これらのレコードによって指定されたアドレスの間に矛盾がなければ、そのアドレスで唯一の SSA レコードが出力に書き込まれます。入力ファイル内の SSA レコードの間に競合がある場合、警告メッセージが出力され、SSA レコードは出力に**書き込まれません**。入力ファイル内に SSA レコードが存在しない場合、SSA レコードは出力に書き込まれません。

8.2.2.8 ver

--ver オプションを指定すると、Hexmate はバージョンおよびビルド情報を出力した後に停止します。

8.2.2.9 addressing

-addressing=*units* オプションを使うと、Hexmate のコマンドライン オプションのアドレス単位を既定値の 1 から最大 4 まで変更できます。

既定値では、Hexmate オプション内で指定される全てのアドレス引数はバイトアドレスであると見なされます(Intel HEX ファイルはバイトアドレスを使用するため)。例えば、オプション-mask=0F@0-FF を指定した場合、アドレス 0x0 から 0xFF までの全ての HEX ファイルデータに対してマスクが実行されます。一部のデバイス アーキテクチャでは、ネイティブのアドレス フォーマットはバイトアドレス指定ではありません。例えば、ある HEX ファイルがアドレス 0x200 と 0x201 に値が 0x0F と 0x55 のバイトを格納していると想定します。この HEX ファイルがミッドレンジ PIC デバイスにロードされた場合、これらの 2 バイトはデバイス内のアドレス 0x100 で 1 ワードを形成します。この場合、各デバイスアドレスでのワード値は、HEX ファイル内で 2 バイト値に展開されて 2 つのアドレスに格納されます。Hexmate オプション内でデバイスアドレスを指定したい場合、このオプションを使って HEX ファイルアドレスとデバイスアドレスの間の対応を指定できます。

このオプションでは、各デバイスアドレス位置に保存される HEX ファイル内のバイト数を示すパラメータとして unit だけを指定します。unit の設定可能レンジは 1〜4 です。8 ビット AVR デバイスとベースライン、ミッドレンジ、24 ビット PIC デバイスに対しては、unit = 2 (-addressing=2) が使えます。

これにより、全ての Hexmate オプションでデバイスアドレスを指定できます。その他の全ての Microchip 社製デバイスに対しては、既定値のアドレス単位(-addressing=1)を指定するか、このオプションを完全に省略できます。これらのデバイスでは、HEX ファイルとデバイスアドレスは同じであり、それらの間の対応付けは不要です。

8.2.2.10 break

-break オプションには、コンマで区切った 16 進アドレスのリストを指定します。HEX ファイル内でこれらのアドレスのいずれかに達すると現在のデータレコードは完了し、そのアドレスから新しいデータレコードが再開します。

例えば、このオプションを指定しない場合の Hexmate の出力が以下であると想定します。

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:10001000ADDEADDEADDEADDEADDEADDEADDEADDE88
:10002000ADDEADDEADDEADDEADDEADDEADDEADDE78
...
```

-break=16 オプションを指定した場合、出力は以下となります。

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:06001000ADDEADDEADDE49
:10001600ADDEADDEADDEADDEADDEADDEADDEADDE82
:10002600ADDEADDEADDEADDEADDEADDEADDEADDE72
...
```

データレコードを分割する事で、プログラム空間の機能的に異なる領域を区別する事ができます。一部の HEX ファイルリーダーは、この方法で配置されたレコードに依存します。

8.2.2.11 ck

-ck オプションは、ハッシュ値の計算用に使います。このオプションの書式は以下の通りです。

```
-ck=start-end@dest[+offset][wWidth][tCode[.Base]][gAlgorithm][pPolynomial][rRevWidth]
[sSkipWidth[.SkipBytes]][oXORvalue]
```

- *start* と *end* は、ハッシュを計算するアドレスレンジを 16 進値で指定します。これらのアドレスがチェックサムと Fletcher アルゴリズム向けデータ幅の倍数ではない場合、値 0 を不足する入力ワード該当位置にパディングします。
- *dest* は、ハッシュ結果の保存位置を指定する 16 進アドレスです。このアドレスは、ハッシュが計算されるアドレス領域の外である事が必要です。
- *offset* は、ハッシュ計算で使う初期 16 進値です(指定は任意)。SHA アルゴリズムに対してこの引数は不要です。
- *Width* は、結果値の幅を 10 進値で指定します(指定は任意)。大部分のアルゴリズムは 1〜4 のバイト幅で結果を計算できます。SHA アルゴリズムの場合、この値はビット幅を表します。正の値で Width を指定した場合、結果はビッグエンディアンのバイト順で保存されます。負の値で Width を指定した場合、結果はリトルエンディアンのバイト順で保存されます。Width を指定しなかった場合、2 バイト幅の結果がリトルエンディアンのバイト順で保存されます。Fletcher アルゴリズムでは幅は固定されるため、この Width 引数は不要ですが、この引数を使って結果のエンディアンを変更できます。
- *Code* は、結果内の各バイトの直後に追加する 16 進コードシーケンスです(指定は任意)。ハッシュ結果の各バイトを命令内に埋め込む必要がある場合、またはハッシュ値をパディングして実行時にデバイスから読み取れるようにする必要がある場合、この機能が使えます。例えば、ミッドレンジ PIC デバイスに対して t34 を指定すると、結果の各バイトは retlw 命令(ビットシーケンス 0x34xx) 内に埋め込まれます。コードシーケンスが複数バイトを指定する場合、これらはハッシュバイトの後にビッグエンディアン順に保存されます。例えば tAABB は、ハッシュバイトの直後に 0xAA を追加し、次のアドレスに 0xBB を追加します。t0000 を指定した場合、ハッシュの各バイトの後に 2 個の 0x00 バイトが保存されます。*Code* 引数では、コードシーケンスに続けて.*Base* をオプションとして指定できます。*Base* は、後続のコードシーケンスを追加する前に出力するハッシュバイトの数です。例えば t11.2 と指定した場合、ハッシュ結果の各 *2* バイトの後に 0x11 のバイトが出力されます。
- *Algorithm* は、結果の計算用に使う Hexmate ハッシュ アルゴリズムを選択するための 10 進整数です。選択可能なアルゴリズムのリストを表 8-3 に示します。この引数を指定しなかった場合、既定値アルゴリズム(8 ビットチェックサムの加算、表 8-3 内のアルゴリズム 1)が使われます。
- *Polynomial* は、CRC アルゴリズム用に使う多項式を指定するための 16 進値です。

- *RevWidth* は、バイト順を反転して読み出すワード幅を指定します。このオプションで非 0 の値を指定した場合、ハッシュ値を計算する時に各ワード内のバイトは逆順に読み出されます。ワードは HEX ファイル内でアドレスに対して整列します。現時点では、*RevWidth* は 0 または 2 である事が必要です。0 を指定した場合、この機能は無効になります(r サブオプションが何も指定されなかった場合と同じ)。このサブオプションは、スキャナ モジュールを介してデータを受け取る PIC ハードウェア CRC モジュールによって生成された CRC を照合するために Hexmate が使われる状況を想定しています。この機能は全てのタイプのハッシュに対して使えますが、表 8-3 内のチェックサム アルゴリズム(-4～4)を使う場合、効果はありません。
- *SkipWidth* は、ハッシュ計算中にスキップするワード幅を指定します(指定は任意)。このオプションで非 0 の値を指定した場合、各ワード内の最高アドレスのバイトは、ハッシュ値の計算中にスキップされます。ワードは HEX ファイル内でアドレスに対して整列します。現時点では、この幅は 0 (スキップ機能は無効、s サブオプションが何も指定されなかったのと同じ)または 2 以上の値である事が必要です。この *SkipWidth* に続けて.*SkipBytes* を任意に指定できます。*SkipBytes* は、各ワード内でスキップするバイトの数を示します。例えば s4.2 と指定した場合、各 4 バイトワード内のアドレスが高い側の 2 バイトがスキップされます。MPLAB XC16 C コンパイラによって HEX ファイルに追加されたファントムバイト(0x00)がハッシュ計算で処理されるのを防ぐには、s4 を指定します。
- *XORvalue* は、ハッシュ結果を保存する前の XOR 演算に使う 16 進値を指定します。

1 文字の引数トークンは大文字と小文字を区別しません。例えば w2 と W2 はどちらも有効な幅指定引数です。

チェックサム計算に対するこのオプションの標準的な例を以下に示します。

```
-ck=0-1FFF@2FFE+2100w-2g2
```

これにより、アドレスレンジ 0～0x1FFF に対してチェックサム(16 ビット加算)が計算され、チェックサム結果がアドレス 0x2FFE に書き込まれます。チェックサム値には 0x2100 のオフセットが適用されます。結果は 2 バイト幅であり、リトルエンディアン形式で保存されます。

バイト順反転(r)およびスキップ(s)機能は、処理中のシーケンス内の開始バイトには影響せず、HEX ファイルアドレスに整列したワードに対してのみ影響を与えます。言い換えると、ワードの位置は-ck オプションで指定された開始および終了アドレスの影響を受けません。

例:

```
-ck=0-5@100w2g5p1021s2
```

上記は、ハッシュ値の計算時に、HEX アドレス 0～5 で 1 バイトおきにスキップします(s2)。これが下記の HEX レコード(下線部のデータ)に対して適用された場合、

```
:10000000640025000030A75007600770078006400 1C
```

ハッシュはバイト 64、25、03 (16 進)から計算されます。同じ HEX レコードを異なる開始～終了アドレスレンジ(1～6)で処理する例を以下に示します。

```
-ck=1-6@100w2g5p1021s2
```

この場合、ハッシュはバイト 25、03、75 から計算されます。これらの機能は、デバイス上で ROM 上のデータ格納様式を模倣するため、これらの機能が使うワードはデバイスアドレスで整列されます(すなわち HEX ファイルアドレスに対して整列されます)。

表 8-3. Hexmate ハッシュ アルゴリズムの選択

選択値	アルゴリズムの概要
-5	ビット順を反転した巡回冗長検査(CRC)結果
-4	初期値から 32 ビット値を減算
-3	初期値から 24 ビット値を減算
-2	初期値から 16 ビット値を減算
-1	初期値から 8 ビット値を減算
1	初期値に 8 ビット値を加算
2	初期値に 16 ビット値を加算

.....続き

選択値	アルゴリズムの概要
3	初期値に 24 ビット値を加算
4	初期値に 32 ビット値を加算
5	巡回冗長検査(CRC)
7	Fletcher のチェックサム(8 ビット計算、2 バイトの結果幅)
8	Fletcher のチェックサム(16 ビット計算、4 バイトの結果幅)
10	SHA-2 (現在は SHA256 のみサポート)

ハッシュ計算に使われるアルゴリズムの詳細は、8.2.3 「ハッシュ値の計算」を参照してください。

8.2.2.12 fill

`-fill` オプションは、HEX ファイルの内の未使用(未指定)メモリ位置に既知の値を書き込むために使います。このオプションの書式は以下の通りです。

```
-fill=[wwidth:]fill_expr@address[:end_address]
```

各引数は以下の意味を持ちます。

wwidth *fill_expr* 内の各定数の幅を 10 進値(1〜9)で指定します。この幅が指定されない場合の既定値は 2 です。例えば`-fill=w1:0x55@0:0xF` が指定された場合、アドレスレンジ 0〜0xF 内の全ての未使用バイトにバイト値 0x55 が書き込まれます。

例:

```
:10000000FBEF3FF05555555555555555555555555DB
```

`-fill=w2:0x55@0:0xF` が指定された場合、同じくアドレスレンジ 0〜0xF 内の全ての未使用バイトに値 0x0055 が書き込まれます。

例:

```
:10000000FBEF3FF05500550055005500550055005500D9
```

fill_expr 書き込み値を定義します。この値は *const* (最初の未使用メモリ位置に書き込むベース値)とオプションの *increment* (増分値) により定義されます。ベース値として 1 バイトを超える値が指定された場合、それらの値はリトルエンディアンのバイト順で保存されます。以下に使用例を示します。

- *const* だけを指定した場合、その値が繰り返し書き込まれます。例えば`-fill=0xBEEF@0-0x1FF` は、アドレスレンジ 0〜0x1FF 内の未使用メモリ位置の全てに値 0xBEEF を繰り返し書き込みます。

  例:

  ```
  :10000000FBEF3FF0EFBEEFBEEFBEEFBEEFBEEFBEC9
  ```

- *const*`+=`*increment* を指定した場合、メモリに書き込まれる値は *increment* ずつ増加します。例えば`- fill=0xBEEF+=1@0:0x1FF` は 0xBEEF、0xBEF0、0xBEF1、0xBEF2、...と 1 ずつ増加する値を書き込みます。

  例:

  ```
  :10000000FBEF3FF0F1BEF2BEF3BEF4BEF5BEF6BEAE
  ```

  const は、スキャンされる各メモリ位置で(その位置が実装されているかどうか、または未使用かどうかに関係なく)増加します。

- *const*`-=`*increment* を指定した場合、メモリに書き込まれる値は *increment* ずつ減少します。例えば`-fill=0xBEEF-=0x10@0:0x1FF` は 0xBEEF、0xBEDF、0xBECF、0xBEBF、...と 0x10 ずつ減少する値を書き込みます。

  例:

  ```
  :10000000FBEF3FF0CFBEBFBEAFBE9FBE8FBE7FBE79
  ```

  const は、スキャンされる各メモリ位置で(その位置が実装されているかどうか、または未使用かどうかに関係なく)減少します。

- *const*,*const*,...,*const* とコンマで区切った値のリストを指定した場合、それらの値の並びが繰り返しメモリに書き込まれます。例えば- fill=0xDEAD,0xBEEF@0:0x1FF は値 0xDEAD、0xBEEF、0xDEAD、0xBEEF、...と 2 つの値を繰り返し書き込みます。

  例:

```
:10000000FBEF3FF0ADDEEFBEADDEEFBEADDEEFBE2F
```

@*address* *fill_expr* を書き込むアドレスを指定します。例えば -fill=0xBEEF@0x1000 は、addressing 値が 1 に設定されていれば、バイト値 0xEF をアドレス 0x1000 に書き込みます。

```
:01100000EF00
```

上の例に対して -addressing=2 オプションが追加で指定された場合、アドレス 0x2000 と 0x2001 に 2 バイトが書き込まれます。

```
:02200000EFBE31
```

:*end_address* *fill_expr* を書き込むメモリ領域の終了アドレスを指定します。例えば - fill=0xBEEF@0xF0:0xFF はアドレスレンジ 0〜0xFF 内の未使用アドレスに値 0xBEEF を書き込みます。

```
:1000F000EFBEEFBEEFBEEFBEEFBEEFBEEFBE98
```

アドレスレンジ(-addressing 値の倍数) が書き込み値の幅の倍数ではない場合、最後の書き込み位置には書き込み値の一部だけが使われ、警告が発行されます。

書き込み値はワード境界に整列するため、書き込み値の幅の倍数アドレスで書き込みが始まります。書き込み値が命令オペコードである場合、この整列により命令は正しく実行されます。同様に、書き込みシーケンスの総長が 1 よりも大きい場合(たとえ幅が 1 に指定されていても)、書き込みシーケンスはその総長に対して整列されます。例えば下記の fill オプションは、2 の倍数ではないアドレスから始まる 2 バイトの書き込みシーケンスを指定します。

```
-fill=w1:0x11,0x22@0x11001:0x1100c
```

この結果として下記の HEX レコードが得られます。この場合、開始アドレスには書き込みシーケンスの第 2 バイトが書き込まれます。

```
:0C10010022112211221122112211221B1
```

これに対し、-fill=w1:0x11,0x22@0x11000:0x1100c (開始アドレスは 2 の倍数)を指定した場合の結果は以下となります。

```
:0D10000011221122112211221122112A0
```

全ての fill 定数(幅指定を除く) は、通常の C 構文に従って(符号なしの) 2/8/10/16 進数として表現可能です。例えば 10 進数の 1234 は 16 進数の 0xFF00 として表現できます(FF00 は違反)。

8.2.2.13 find

-find=*opcode* オプションは、指定されたオペコードまたはコードシーケンスを検索し、その結果をログファイルに保存します。このオプションの書式は以下の通りです。

```
-find=Findcode[mMask]@Start-End[/Align][w][t"Title"]
```

- *Findcode* は、検索対象の 16 進コードシーケンスです。例えば clrf 命令(オペコードは 0x01F1)を検索する場合、01F1 をシーケンスとして指定します。HEX ファイル内では、これはバイトシーケンス F1 01 (HEX アドレス 0 の 0xF1 と HEX アドレス 1 の 0x01)として現れます。
- *Mask* は任意に指定可能です。これは、*Findcode* 値に対して適用するビットマスクを指定します。マスクを指定する事で、*Findcode* 値内の一部を検索条件から除外できます。これはリトルエンディアンのバイト順で入力します。
- Start と *End* は、検索するアドレスレンジを制限します。
- *Align* は任意に指定可能です。これを指定した場合、この値の倍数のアドレスで始まるコードシーケンスのみが検索されます。

- w を指定した場合、Hexmate はコードシーケンスが検出された時に警告を発行します。
- *Title* は任意に指定可能です。これにより、検索されるコードシーケンスにタイトルを付ける事ができます。タイトルを定義する事で、ログレポートとメッセージが読みやすくなります。タイトルは検索結果に影響を及ぼしません。

全ての数値引数は 16 進値であると見なされます。

以下に例を示します。
-find=1234@0-7FFF/2w を指定すると、2 バイトアドレス境界で始まるコードシーケンス「1234h」(HEX ファイルには 34 12 として保存される)がアドレスレンジ 0h～7FFFh 内で検索され、このシーケンスが見つかるたびに警告が発行されます(w が指定されているため)。

-find=1234M0F00@0-7FFF/2wt"ADDXY"は、オプションは前の例と同じですが、検索対象のコードシーケンスに対してマスク値 00OFh が適用されます。この場合、オペコード 123xh (*x* は任意の数字)が検索されます。バイトマスクのバイト幅は、検索するオペコードのバイト幅と同じである事が必要です。さらに t"ADDXY"によってタイトルが指定されているため、Hexmate が生成するメッセージまたはレポート内でこのオペコードは「ADDXY」という名前で参照されます。

要求された場合、ログファイルは全ての検索結果を格納します。-find オプションは、長さが 1～8 バイトの HEX データを受け付けます。-find は、次に説明する replace または delete と組み合わせて使う事もできます。

8.2.2.14 find と delete

-find オプションで delete を指定すると、一致したシーケンスを出力ファイルから削除できます。これにより、対応するデータは完全に削除されます(値が 0 のバイトに置き換えられるのではありません)。

この機能は、-find オプションの後に,delete を追加する事で利用できます。

例:

```
-find=ff@7fe0-7fff,delete
```

この機能を使うには十分な注意が必要です。一般的に、実行可能コードを削除する事は推奨しません。

8.2.2.15 find と replace

-find オプションで replace を指定すると、一致したシーケンス(またはその一部)を新しいコードに置き換える事ができます。

この機能は、-find オプションの後に,replace=*spec* を追加する事で利用できます。

例:

```
-find=spec,replace=Code[mMask]
```

- *Code* は、-find の検索条件に一致したシーケンスを置き換える 16 進コードシーケンスです。
- *Mask* は任意に指定可能なビットマスクであり、一致したコードシーケンスを *Code* 内のどのビットで置き換えるのか指定します。この機能は、例えば 16 ビット命令内の 4 ビットのみを置換する必要がある場合に便利です。残りの 12 ビットは未変更のまま残す事ができます。

例:

```
-find=ff@7fe0-7fff,replace=55
```

この機能を使うには十分な注意が必要です。

8.2.2.16 format

-format=*type*[,*length*]オプションを使うと、出力する INHX ファイルの書式を指定すると共に、最大レコード長を調整できます。

type 引数は、生成する INHX ファイルの書式(表 8-4 参照)を指定します。例えば INHX16 または INHX32 ファイルから INHEX8 ファイルへ変換する場合、入力ファイルにレコードタイプ 2、3、4、5 のいずれかが含まれていると、変換はできません。そのような場合に、このオプションを使って入力ファイルの書式を変更する事ができます。HEX ファイルの書式は、そのファイル内に存在するレコードのタイプによって決まります。例えばレコードタイプ 0 および 1 のみが存在する場合、その HEX ファイルの書式は INHX8M と見なされます。タイプ 4 または 5 も存在する場合は INHX32 と見なされます。タイプ 2 または 3 のレコードとタイプ 4 または 5 のレコードが一緒に存在する場合、そのファイルはどの書式にも当てはまりません。指定された出力書式内で許容されないレコードは削除されます。

length 引数を使うと、データレコードあたりの最大バイト数を指定できます。有効な長さは 1〜16 (10 進値)です(既定値は 16)。

format オプションがサポートする書式(*type*)を表 8-4 に示します。INHX032 は、実際に使われる INHX 書式ではありません。この書式を選択すると INHX32 ファイルが生成されますが、上位アドレス情報は 0 へと初期化されます。この書式は、一部のデバイス プログラマによって要求されます。

表 8-4. HEX ファイルの書式

type	有効なレコードタイプ	注釈
INHX8M	0、1	16 ビット幅アドレス フィールド
INHX16	0、1、2、3	20 ビット幅アドレス フィールド
INHX32	0、1、4、5	32 ビット幅アドレス フィールド
INHX032	0、1、4、5	上位アドレスを 0 に初期化した INHX32

8.2.2.17 help

-help は、全ての Hekmate オプションを表示します。-help のパラメータとして他の Hexmate オプションを指定すると、そのオプションに関する詳細なヘルプ情報が表示されます。

例:

```
-help=string
```

上記は、-string Hexmate オプションに関する追加のヘルプ情報を表示します。

8.2.2.18 logfile

-logfile オプションは、HEX ファイルの統計データを指定された名前のファイルに保存します。

例:

```
-logfile=output.hxl
```

上記は、Hexmate が生成中の HEX ファイルを解析し、その結果をファイル output.hxl に保存します。

8.2.2.19 mask

-mask=*spec* オプションは、指定されたマスク値と指定されたメモリレンジ内のデータの間の論理積(AND)を取ります。このオプションは、HEX データ上で ROM の未実装ビット位置をブランクにしたい場合に使えます。このオプションの書式は以下の通りです。

```
-mask=hexcode@start-end
```

上記は、*hexcode* で指定された値とアドレスレンジ *start*〜*end* 内のデータの AND を取ります。全ての値は 16 進値であると見なされます。マルチバイトのマスク値は、リトルエンディアンのバイト順で入力できます。

8.2.2.20 o : 出力ファイルの指定

-o*file* オプションは、生成された Intel HEX 出力を指定したファイルに保存します。

例:

```
-oprogram.hex
```

上記は、生成された出力をファイル program.hex に保存します。出力ファイルには、入力ファイルのいずれかと同じ名前を指定できますが、その場合、その入力ファイルの内容は完全に置き換えられます。

ファイル名を指定しないと出力は生成されません。これは、HEX ファイルのサイズを調べるためだけに Hexmate を使う場合に役立ちます。このオプションを何も指定しなかった場合、出力 HEX ファイルは標準出力ストリームへ出力されます。

8.2.2.21 serial

-serial=*specs* オプションは、特定の HEX 値シーケンスを指定されたアドレスに保存します。このオプションの書式は以下の通りです。

```
-serial=Code[+/-Increment]@Address[+/-Interval][rRepetitions]
```

- *Code* は、保存する 16 進値シーケンスです。最初に指定されたバイトは最低アドレスに保存されます。
- *Increment* を指定すると、繰り返しのたびに *Code* の値をインクリメントできます(任意に指定可能)。
- *Address* は、このコードを保存する位置です(繰り返す場合は最初の保存位置)。
- *Interval* は、繰り返しのたびにこのコードに適用するアドレスシフト量を指定します(任意に指定可能)。
- *Repetitions* は、このコードを繰り返す回数を指定します(任意に指定可能)。

Repetitions(既定値により 10 進値)を除く全ての数値引数は 16 進値であると見なされます。

例:

```
-serial=000001@EFFE
```

上記は、HEX コード 0x00001 をアドレス 0xEFFE に保存します。

例:

```
-serial=0000+2@1000+10r5
```

上記は、5 個のコードを保存します。最初のコード 0000 はアドレス 0x1000 に保存されます。後続のコード値は 0x2 ずつ増分され、+0x10 のアドレス間隔で保存されます。

8.2.2.22 size

-size オプションは、標準出力へ出力される HEX イメージ内のデータバイト数を報告します。このサイズはログファイルが要求された場合にも記録されます。

8.2.2.23 string

-string オプションは、ASCII 文字列を指定されたアドレスに埋め込みます。このオプションの書式は以下の通りです。

```
-string@Address[tCode]=”Text”
```

- *Address* は、文字列の保存アドレスを 16 進値で指定します。
- *Code* を指定すると、文字列内の各バイトの直後にバイトシーケンスを追加できます。これにより、文字列のバイトを命令内でエンコードできます(RETLW:0x34 等)。
- *Text* は、ASCII へ変換されて埋め込まれる文字列です。

例:

```
-string@1000=”My favorite string”
```

上記は、文字列「My favorite string」(Null 終端文字を含む) の ASCII データをアドレス 0x1000 に保存します。

例:

```
-string@1000t34=”My favorite string”
```

上記は、前の例と同じ文字列内の各バイトの直後に HEX コード 0x34 を追加して保存します。

8.2.2.24 strpack

-strpack=*spec* オプションは、-string と似た機能を実行しますが、2 つの重要な違いがあります。

-string は各文字に対応する完全なバイトを保存しますが、-strpack は各文字の下位 7 ビットのみを保存します。7 ビット文字のペアは、別々のバイトとしてではなく連結された 1 つの 14 ビットワードとして保存されます。これは文字列パッキングと呼ばれます。これは、プログラムメモリが 14 ビットワードとしてアドレス指定されるミッドレンジ PIC デバイスでしばしば役立ちます。このオプションを使う場合、エンコードされた文字が実行時に正しく読み出されて解釈される必要があります。

-string オプションとのもう 1 つの違いは、-strpack オプションでは t 指定子が使えないという事です。

8.2.2.25 w:警告レベルを指定する

`-wlevel` オプションは、警告レベルしきい値を設定します。*level* は、-9～9 のレベルに設定できます(例: `-w5`)。

警告レベルは、適正かどうか疑わしい要求またはファイル内容に対して Hexmate がどの程度こだわるのかを示します。各警告にはレベルが設定されており、レベルが高いほど警告メッセージの重要度が増します。警告メッセージのレベルがこのオプションで設定されたしきい値を超えると、その警告は出力されます。既定値の警告レベルしきい値は 0 であり、通常の全ての警告が表示されます。

8.2.3 ハッシュ値の計算

ハッシュ値は小さな固定サイズの値であり、任意サイズのデータブロック内の全ての値から計算され、それらの値を代表する値として使われます。データブロックをコピーした際に、新しいブロックから計算されたハッシュ値と元のハッシュ値を比較し、それらが一致した場合は高い確度でコピーは有効であると見なす事ができます。ハッシュ値の計算には様々なアルゴリズムが使われます。アルゴリズムが複雑であるほど検証の信頼性は向上しますが、組み込み環境(特に小型機器)で複雑なアルゴリズムを使うと、計算負荷が過大となる場合があります。

Hexmate を使うと、HEX ファイルに格納されたプログラム イメージのハッシュを計算する事ができます。このハッシュを同じ HEX ファイルに埋め込み、プログラム イメージと一緒にターゲット デバイスに書き込む事ができます。実行時に、ターゲット デバイスは内部のメモリに保存されたプログラム イメージに対して同様のハッシュ アルゴリズムを実行できます。保存されていたハッシュ値と計算されたハッシュ値が一致すれば、デバイスメモリ内のプログラムは有効なプログラム イメージであると見なす事ができます。

Hexmate は、ハッシュ値の計算用に各種ハッシュ アルゴリズム (チェックサム、巡回冗長検査(CRC)等)を実装しています。

`xc8-cc` または MPLAB X IDE を使ってプロジェクトをビルドする場合、コンパイラ ドライバの`-mchecksum` オプションと引数により、Hexmate を起動してハッシュ計算用に適切なオプションを渡す事ができます。Hexmate を直接実行する場合、`-ck` オプションによりハッシュ計算を要求します(8.2.2.11 「ck」参照)。

未使用メモリ位置が存在するメモリに対してハッシュ値を計算する場合、注意が必要です。`xc8-cc` または MPLAB X IDE を使ってプロジェクトをビルドする場合、未プログラム状態のデバイスメモリと照合するため、ハッシュ値を要求すると自動的に Hexmate に対して未使用メモリ位置への書き込みが要求されます。Hexmate を直接実行する場合、`-fill` オプション(8.2.2.12 「fill」参照)を使ってこれらの位置に既知の値を書き込むよう配慮します。

Hexmate は任意の Intel HEX ファイルからハッシュ値を生成できます。そのファイルがどのコンパイラによって生成され、どのデバイスに書き込まれるかは関係ありません。しかし、実行時に読み出し可能なメモリ位置はターゲット デバイスのアーキテクチャによって制限される場合があります。そのような場合、2 つのハッシュ値が同様の方法で計算されて互いに一致するようにするため、Hexmate のハッシュ計算方法を変更する必要が生じます。加えて、一部のコンパイラは、デバイスメモリ内に存在しないパディング バイトまたはファントム バイトを HEX ファイルに挿入する可能性があります。そのような場合、Hexmate はハッシュ値を計算する際にこれらのバイトを無視する必要があります。以下では、可能な対策について説明します。

デバイスによっては、プログラムメモリの幅全体を読み出せない場合があります。例えばベースラインおよびミッドレンジ PIC デバイスは、各プログラムメモリ位置の下位バイトのみ読み出す事ができます。しかし HEX ファイルは各プログラムメモリ ワードに対して 2 バイトを格納し、Hexmate はハッシュ値の計算時にこれらのバイトの両方に対して通常の処理を行います。このため、`xc8-cc` または MPLAB X IDE を使ってプロジェクトのビルドを実行する場合、`-mchecksum` オプションに対して `skip=2` サブオプションを使う事で、Hexmate に各プログラムワードの MSB をスキップさせます。Hexmate を直接実行する場合、`-ck` オプションに対して `s2` サブオプションを使う事で、各 2 バイトワードの MSB をスキップさせます。この種の検証プロセスは、各プログラムワードの MSB の破損に配慮しないという事に注意が必要です。

一部のデバイスはハードウェア CRC モジュールを備えており、CRC ハッシュ値を計算可能です。必要に応じ、スキャナ モジュールを使ってプログラムメモリ データを CRC モジュールへ供給する事で、計算を自動化できます。スキャナ モジュールは各プログラムメモリ ワードの MSB を最初に読み出すため、Hexmate にも命令ワード内の HEX ファイルバイトを逆順に処理させる必要があります。このため、`xc8-cc` または MPLAB X IDE を使ってプロジェクトのビルドを実行する場合、`-mchecksum` オプションに対して `revword=2` サブオプションを使う事で、Hexmate に 2 バイトワード内のバイトの処理順序を逆転させます。Hexmate を直接実行する場合、`-ck` オプションに対して `r2` サブオプションを使う事で、Hexmate に 2 バイトワード内のバイトの処理順序を逆転させます。

また、HEX ファイル内でエンコードされた Hexmate ハッシュ値を実行時に読み出す方法についても配慮が必要です。

ベースラインおよびミッドレンジ PIC デバイスは、`retlw` 命令を使ってデータをプログラムメモリに保存する必要があります。従って、これらのデバイスは、Hexmate によって計算されたハッシュ値の各バイトを保存するために 1 つの命令を必要とします。`xc8-cc` または MPLAB X IDE を使ってプロジェクトのビルドを実行する場合、`-mchecksum` オプションに対して `code=34` サブオプションを使う事で、Hexmate にハッシュ値の各バイトを `retlw` 命令内に保存させる事ができます(`retlw` 命令は `0x34`*nn* としてエンコードされます(*nn* は実行時に WREG

に書き込まれる 8 ビットデータ値))。Hexmate を直接実行する場合、-ck オプションに対して t34 サブオプションを使う事で、Hexmate に各バイトを retlw 命令の一部として保存させます。

dsPIC および PIC24 デバイスは、プログラムメモリの全幅を読み出す事ができず、このメモリに書き込まれるデータは、通常 HEX ファイルデータの各 4 バイトあたり 2 バイトのチャンクに保存されます。xc8-cc または MPLAB X IDE を使ってこれらのデバイス向けにプロジェクトのビルドを実行する場合、-mchecksum オプションに対して code=0000.2 サブオプションを使う事で、Hexmate にハッシュデータの各 2 バイトに対してバイトシーケンス 0x00 0x00 を追加させます。Hexmate を直接実行する場合、-ck オプションに対して t0000.2 サブオプションを使う事で、Hexmate に HEX ファイルの各 4 バイトの下位 2 バイトにハッシュ値の 2 バイトを保存させ、上位 2 バイトを 0 に設定させます。

8.2.3.1 ハッシュ アルゴリズム

以下では、Hexmate によって使われるアルゴリズムの例を示します。これらのアルゴリズムは、実行時に対応するハッシュ値を計算するためにも使えます。これらの例は、ターゲット デバイスと状況に応じて変更が必要になる場合があります。

8.2.3.1.1 加算アルゴリズム

Hexmate は、プログラム イメージ内の特定レンジでデータ値を合計する各種のシンプルなチェックサム アルゴリズムを備えています。これらのアルゴリズムは、アルゴリズム サブオプション内のセレクタ値 1/2/3/4 に対応し、それぞれプログラム イメージ内のデータを 1/2/3/4 バイトを単位として読み出します。この合計値は、同じオプションによってアルゴリズムに提供される初期値(オフセット) に加算されます。最終的なチェックサム値が切り詰められる幅(1/2/3/4 バイトのいずれか)もこのオプションにより指定されます。Hexmate は、チェックサムを自動的に HEX ファイル内に保存します(アドレスはチェックサム オプションにより指定)。

以下に示す関数は、任意のデータサイズ(read_t) およびチェックサム幅(result_t) の組み合わせで動作するようカスタマイズできます。

```
#include <stdint.h>
typedef uint8_t read_t;        // size of data values read and summed
typedef uint16_t result_t;     // size of checksum result

// add to offset, n additions of values starting at address
data,
// truncating and returning the result
// data: the address of the first value to sum
// n:    the number of sums to perform
// offset: the intial value to which the sum is added
result_t ck_add(const read_t *data, unsigned n, result_t offset)
{
    result_t chksum;
    chksum = offset;
    while(n--) {
        chksum += *data;
        data++;
    }
    return chksum;
}
```

read_t および result_t 型定義は、それぞれデータ読み出し/合計値とチェックサム結果値の幅に適合するよう調整する必要があります。MPLAB XC8 を使う場合、サイズが 1 であれば char 型を使い、サイズが 4 であれば long 型を使います(その他のサイズに対しても適切な型を使います)。オフセットを使わない場合、そのパラメータは削除できます(ループの前で chksum に 0 を代入)。

例えば、アドレスレンジ 0x100〜0x7f に対して、1 バイト幅値の加算による 2 バイト幅チェックサムの計算を 0x20 のオフセットで開始する場合を想定します。チェックサムは、リトルエンディアン形式で 0x7fe と 0x7ff に保存されます。

プロジェクトをビルドする際に、以下のオプションを指定します。

MPLAB X IDE では、**[XC8 Linker]** カテゴリ内で**[Additional options]** オプション カテゴリの**[Checksum]**フィールドで先頭の = の右に以下の情報を入力します。

```
-mchecksum=100-7fd@7fe,offset=20,algorithm=1,width=-2
```

Hexmate を直接実行する場合、以下のオプションを使います。

```
-ck=100-7fd@7fe+20g1w-2
```

以下の MPLAB XC8 コードを PIC デバイス向けに調整します。このコードは ck_add()を呼び出し、実行時のチェックサムとコンパイル時に Hexmate が保存したチェックサムを比較します。

```
extern const read_t ck_range[0x6fe/sizeof(read_t)] __at(0x100);
extern const result_t hexmate __at(0x7fe);
result_t result;

result = ck_add(ck_range, sizeof(ck_range)/sizeof(read_t), 0x20);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

このコードでは、プレースホルダ配列 ck_range を使ってチェックサムを計算するメモリレンジを指定し、変数 hexmate は Hexmate によるチェックサム結果の保存位置に割り当てられます。これらのオブジェクトはどちらも extern かつ絶対アドレスであるため、追加のデバイスメモリを消費しません。これらのオブジェクトのアドレスとサイズは、Hexmate に渡すオプションと一致するように調整します。

Hexmate は任意のアドレスレンジに対してチェックサムを計算できますが、上記コード内のテスト関数 ck_add() は合計レンジの開始および終了アドレスが read_t 幅の倍数である事を前提とします。read_t が 1 である場合、これは問題になりません。チェックサムの仕様をこの前提に適合させる事を推奨します。そうしないと、開始および/または終了データの部分的な読み出しを実行するためにテストコードを変更する必要があり、コードは大幅に複雑になります。

8.2.3.1.2 減算アルゴリズム

Hexmate は、プログラム イメージ内の特定レンジでデータ値を減算する各種のチェックサム アルゴリズムを備えています。これらのアルゴリズムは、アルゴリズム サブオプション内のセレクタ値-1/-2/-3/-4 に対応し、それぞれプログラム イメージ内のデータを 1/2/3/4 バイト量として読み出します。この点を除けば、減算アルゴリズムは加算アルゴリズムと同じです。加算アルゴリズムの詳細は 8.2.3.1.1 「加算アルゴリズム」参照してください。

以下に示す関数は、任意のデータサイズ(read_t) およびチェックサム幅(result_t) の組み合わせで動作するようカスタマイズできます。

```
#include <stdint.h>
typedef uint8_t read_t;        // size of data values read and subtracted
typedef uint16 t result t;     // size of checksum result

// add to offset n subtractions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to subtract
// n:    the number of subtractions to perform
// offset: the intial value to which the subtraction is added
result_t ck_sub(const read_t *data, unsigned n, result_t offset)
{
    result_t chksum;
    chksum = offset;
    while(n--) {
        chksum -= *data;
        data++;
    }
    return chksum;
}
```

例えば、アドレスレンジ 0x0～0x7fd に対して、2 バイト幅値の加算による 4 バイト幅チェックサムの計算を 0x0 のオフセットで開始する場合を想定します。チェックサムは、リトルエンディアン形式で 0x7fe と 0x7ff に保存されます。

xc8-cc を使ってプロジェクトをビルドする場合、以下のオプションを指定します。

MPLAB X IDE では、**[XC8 Linker]** カテゴリ内で**[Additional options]** オプション カテゴリの**[Checksum]** フィールドで先頭の = の右に以下の情報を入力します。

```
-mchecksum=0-7fd@7fe,offset=0,algorithm=-2,width=-4
```

Hexmate を直接実行する場合、以下のオプションを使います。

```
-ck=0-7fd@7fe+0g-2w-4
```

以下の MPLAB XC8 コードを PIC デバイス向けに調整します。このコードは ck_sub( )を呼び出し、実行時のチェックサムとコンパイル時に Hexmate が保存したチェックサムを比較します。

```
extern const read_t ck_range[0x7fe/sizeof(read_t)] __at(0x0);
extern const result_t hexmate __at(0x7fe);
result_t result;

result = ck_sub(ck_range, sizeof(ck_range)/sizeof(read_t), 0x0);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

8.2.3.1.3 Fletcher アルゴリズム

Hexmate は Fletcher のチェックサムを実装する各種アルゴリズムを備えています。これらのアルゴリズムは比較的複雑ですが、巡回冗長検査(CRC)に近い信頼性を CRC よりも軽い計算負荷で提供します。このアルゴリズムには 2 種類があり(アルゴリズム サブオプションのセレクタ値 7 と 8 に対応)、それぞれ 1 バイト計算/2 バイト結果と 2 バイト計算/4 バイト結果のアルゴリズムを実装します。Hexmate は、チェックサムを自動的に HEX ファイル内に保存します(アドレスはチェックサム オプションにより指定)。

以下に示す関数は、1 バイト幅加算を実行し、2 バイト結果を生成します。

```
#include <stdint.h>
typedef uint16 t result t;     // size of fletcher result

result_t
fletcher8(const unsigned char * data, unsigned int n )
{
    result_t sum = 0xff, sumB = 0xff;
    unsigned char tlen;
    while (n) {
        tlen = n > 20 ?20 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xff) + (sum >> 8);
        sumB = (sumB & 0xff) + (sumB >> 8);
    }
    sum = (sum & 0xff) + (sum >> 8);
    sumB = (sumB & 0xff) + (sumB >> 8);
    return sumB << 8 | sum;
}
```

例えば、アドレスレンジ 0x100～0x7fb に対して 2 バイト幅 Fletcher ハッシュの計算を、0x20 のオフセットで開始する場合を想定します。チェックサムは、リトルエンディアン形式で 0x7fc～0x7ff に保存されます。

プロジェクトをビルドする際に、以下のオプションを指定します。

MPLAB X IDE では、**[XC8 Linker]** カテゴリ内で**[Additional options]** オプション カテゴリの**[Checksum]**フィールドで先頭の = の右に以下の情報を入力します。width サブオプションで幅を制御する事はできませんが、引数の符号によって結果のエンディアンを指定する事ができます。

```
-mchecksum=100-7fb@7fc,offset=20,algorithm=8,width=-4
```

Hexmate を直接実行する場合、以下のオプションを使います。w サブオプションで幅を制御する事はできませんが、引数の符号によって結果のエンディアンを指定する事ができます。

```
-ck=100-7bd@7fc+20g8w-4
```

このコードは、8.2.3.1.1 「加算アルゴリズム」で示したのと同様の方法で呼び出せます。

下記のコードは、2 バイト加算/4 バイト結果の Fletcher アルゴリズムを実行します。

```
#include <stdint.h>
typedef uint32 t result t;     // size of fletcher result

result_t
fletcher16(const unsigned int * data, unsigned n)
{
```

```
    result_t sum = 0xffff, sumB = 0xffff;
    unsigned tlen;
    while (n) {
        tlen = n > 359 ?359 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xffff) + (sum >> 16);
        sumB = (sumB & 0xffff) + (sumB >> 16);
    }
    sum = (sum & 0xffff) + (sum >> 16);
    sumB = (sumB & 0xffff) + (sumB >> 16);
    return sumB << 16 | sum;
}
```

8.2.3.1.4 CRC アルゴリズム

Hexmate は信頼性の高い巡回冗長検査(CRC)を実装する各種アルゴリズムを備えています。このアルゴリズムには 2 種類があり(アルゴリズム サブオプションのセレクタ値 5 と-5 に対応)、それぞれ標準の CRC 計算と式のビット順を反転した CRC 計算を実装します。反転アルゴリズムは、データの最下位ビットから順番に計算します。

使用する多項式と初期値はオプションで指定できます。Hexmate は、CRC 結果を自動的に HEX ファイル内に保存します(アドレスはチェックサム オプションにより指定)。

一部のデバイスでは、実行時にハードウェア CRC モジュールを使って CRC を計算できます。これらのモジュールは、スキャナ モジュールを使ってプログラムメモリからデータを読み出せます。Hexmate と CRC/スキャナ モジュールのバイト処理順を一致させるため、反転した 2 ワード幅を指定する必要があります。これにより、HEX ファイル内の各 2 バイトワードは正順で読み出されますが、それらのワード内のバイトは逆順で処理されます。MPLAB XC8 コンパイラを使う場合はサブオプション `revword=2` を使い、Hexmate を直接実行する場合は`-ck` に対して `r2` サブオプションを使います。

以下に示す関数は、任意の結果幅(`result_t`)向けにカスタマイズできます。この関数は、`POLYNOMIAL` マクロによって指定された多項式を使って CRC ハッシュ値を計算します。

```
#include <stdint.h>
typedef uint16_t result_t;     // size of CRC result
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(result_t))
#define MSb     ((result_t)1 << (WIDTH - 1))

result_t
crc(const unsigned char * data, unsigned n, result_t remainder)
    { unsigned pos;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        remainder ^= ((result_t)data[pos] << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    return remainder;
}
```

`result_t` 型定義は、結果幅に適合するよう調整する必要があります。MPLAB XC8 を使う場合、例えばサイズが 1 であれば `char` 型を使い、サイズが 4 であれば `long` 型を使います。

例えば、アドレスレンジ 0x0～0xFF に対して、2 バイト幅 CRC ハッシュ値の計算を、0xFFFF の初期値で開始する場合を想定します。結果は 0x100 と 0x101 にリトルエンディアン形式で保存します。

プロジェクトをビルドする際に、以下のオプションを指定します。

MPLAB X IDE では、**[XC8 Linker]** カテゴリ内で**[Additional options]** オプション カテゴリの**[Checksum]**フィールドで先頭の `=` の右に以下の情報を入力します。

```
-mchecksum=0-FF@100,offset=0xFFFF,algorithm=5,width=-2,polynomial=0x1021
```

Hexmate を直接実行する場合、以下のオプションを使います。

```
-ck=0-ff@100+ffffg5w-2p1021
```

以下の MPLAB XC8 コードを PIC デバイス向けに調整します。このコードは crc() を呼び出し、実行時のハッシュ結果とコンパイル時に Hexmate が保存したハッシュ結果を比較します。

```
extern const unsigned char ck_range[0x100] __at(0x0);
extern const result_t hexmate __at(0x100);
result_t   result;

result = crc(ck_range, sizeof(ck_range), 0xFFFF);
if(result != hexmate){
    // something's not right, take appropriate action
    ck_failure();
}
// data verifies okay, continue with the program
```

反転した CRC 結果は、入力データと最終結果を反転するか多項式を反転する事により計算できます。以下に示す関数は、任意の結果幅(result_t) 向けにカスタマイズできます。crc_reflected_IO()関数は、データストリームのビット位置を反転する事により反転した CRC ハッシュ値を計算します。もう 1 つの crc_reflected_poly()関数はデータストリームを変更せずに多項式を反転させます。多項式は、どちらの関数でも POLYNOMIAL マクロを使って指定されます。どちらの関数も reflect()関数を使ってビット順を反転させます。

```
#include <stdint.h>
typedef uint16_t result_t;     // size of CRC result
typedef unsigned char read_t;
typedef unsigned int reflectWidth;
// This is the polynomial used by the CRC-16 algorithm we are using.
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(result_t))
#define MSb     ((result_t)1 << (WIDTH - 1))
#define LSb     (1)
#define REFLECT_DATA(X)       ((read_t) reflect((X), 8))
#define REFLECT_REMAINDER(X)  (reflect((X), WIDTH))

reflectWidth
reflect(reflectWidth data, unsigned char nBits)
{
    reflectWidth reflection = 0;
    reflectWidth reflectMask = (reflectWidth)1 << nBits - 1;
    unsigned char bitp;
    for (bitp = 0; bitp != nBits; bitp++)
        { if (data & 0x01) {
            reflection |= reflectMask;
        }
        data >>= 1;
        reflectMask >>= 1;
    }
    return reflection;
}

result_t
crc_reflected_IO(const unsigned char * data, unsigned n, result_t remainder)
    { unsigned pos;
    unsigned char reflected;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++)
        { reflected =
        REFLECT_DATA(data[pos]);
        remainder ^= ((result_t)reflected << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    remainder = REFLECT_REMAINDER(remainder);
    return remainder;
}
```

```
result_t
crc_reflected_poly(const unsigned char * data, unsigned n, result_t remainder)
    { unsigned pos;
    unsigned char bitp;
    result_t rpoly;
    rpoly = reflect(POLYNOMIAL, WIDTH);
    for (pos = 0; pos != n; pos++) {
        remainder ^= data[pos];
        for (bitp = 8; bitp > 0; bitp--)
            { if (remainder & LSb) {
                remainder = (remainder >> 1) ^ rpoly;
            } else {
                remainder >>= 1;
            }
        }
    }
    return remainder;
}
```

例えば、アドレスレンジ 0x0〜0xFF に対して、2 バイト幅の反転した CRC ハッシュ値の計算を、0xFFFF の初期値で開始する場合を想定します。結果は 0x100 と 0x101 にリトルエンディアン形式で保存します。

プロジェクトをビルドする際に、以下のオプションを指定します。
MPLAB X IDE では、**[XC8 Linker]**カテゴリ内で**[Additional options]** オプション カテゴリの**[Checksum]**フィールドで先頭の = の右に以下の情報を入力します。 この場合、アルゴリズムの選択値は-5 です。

```
-mchecksum=0-FF@100,offset=0xFFFF,algorithm=-5,width=-2,polynomial=0x1021
```

Hexmate を直接実行する場合は以下のオプションを使います。この場合もアルゴリズムの選択値は-5 です。

```
-ck=0-ff@100+ffffg-5w-2p1021
```

プロジェクト内で `crc_reflected_IO()`または `crc_reflected_poly()`を前述の通りに呼び出します。

8.2.3.1.5 SHA アルゴリズム

Hexmate はセキュアハッシュ アルゴリズム(SHA)を実装しています。セレクタ値 10 により SHA256 アルゴリズム(SHA-2 アルゴリズムの 256 ビット バリアント)が選択されます。

SHA256 の実装用コードは Hexmate がサポートする他のアルゴリズムよりも複雑です。SHA256 の結果は、その名が示す通り 256 ビット(32 バイト)です。このアルゴリズムのパブリックドメイン実装コードは、サードパーティ ウェブサイト(github.com/B-Con/crypto-algorithms 等)からダウンロードできます。

以下のオプションは、アドレスレンジ 0x0〜0x1FF に対して SHA256 ハッシュ値を計算し、結果をアドレス 0x1000 にリトルエンディアン形式で保存します。

```
-ck=0-1ff@1000g10w-256
```

9. ライブラリ関数

プログラムの開発を支援するため、MPLAB XC8 C コンパイラには関数、マクロ、型定義を含む各種ライブラリが同梱されます。

標準 C ライブラリの詳細は別の文書『Microchip Unified Standard Library Reference Guide』に記載しています。この文書の内容は全ての MPLAB XC C コンパイラに対応しています。

以下では、MPLAB XC8 に固有のライブラリ関数およびマクロについて説明します。

9.1 ライブラリ サンプルコード

以下には、大部分のライブラリ関数向けのサンプルコードを記載しています。これらのサンプルコードは、ライブラリ関数の呼び出し方を示す事を目的として記載しています。コードにその他の機能が含まれていても、それらは必ずしも完全でも実用的でもありません。サンプルコードのエンコード方法と実行時の動作は、ターゲット デバイスに応じて異なる場合があります。

サンプルコードは MPLAB X IDE 等が備えるシミュレータを使って実行できます。ハードウェア上でも実行できますが、使用するデバイスとハードウェアのセットアップを変更する必要があります。これらのサンプルコードには、ハードウェア上でコードを実行するために必要なデバイス コンフィグレーション ビットを示していません。それらはデバイスごとに異なります。MPLAB X IDE を使う場合、その内蔵ツールを利用してコンフィグレーション ビットの初期化に必要なコードを容易に生成し、これをコピーしてプロジェクトのソースコードにペーストできます。MPLAB X IDE の「Configuration Bits」ウィンドウの詳細は『MPLAB® X IDE User's Guide 』を参照してください。

ライブラリ サンプルコードの多くは `printf()`関数を使います。この関数を使ってユーザが選択した周辺モジュールへ出力させるには、サンプルコードにコードを追加する事が必要になる場合があります。

サンプルコードを MPLAB X IDE シミュレータで実行する場合、`printf()`関数により出力を USART (一部のデバイスでは UART) へ送信し、ウィンドウに表示させる事ができます。これを実行するには以下の手順が必要です。

- MPLAB X IDE 内で USART IO 機能を有効にします(IDE では複数の USART から 1 つを選択できる場合があります)。
- プロジェクトコード内で、IDE で選択したのと同じ USART を初期化して有効にします。
- プロジェクトコード内で、その USART へ 1 バイトを送信する「print-byte」関数を定義します。
- その print-byte 関数を `printf()`関数から呼び出します。

一部のコンパイラでは、上記の USART 初期化と print-byte 関数を実装した汎用コードが提供される場合があります。その他のコンパイラでは、多くの場合、Microchip Code Configurator (MCC)を使ってこのコードを生成する事ができます。使用するターゲット デバイス向けに MCC が利用可能かどうか確認してください。利用できない場合、類似するデバイス向けの MCC 出力をそのターゲット デバイス向けに調整できる場合があります。通常、MCC による既定値の UART 設定はシミュレータ上で正常に動作しますが、そのままでは最終アプリケーションで使えない可能性があります。USART の設定が済めば、`printf()`に加えて、`stdout` へ書き込む標準 I/O ライブラリ関数が使えるようになります。

一部のライブラリ サンプルコードは、`scanf()`関数も使います。この関数を使ってユーザが選択した周辺モジュールからデータを読み出すには、サンプルコードにコードを追加する事が必要になる場合があります。

サンプルコードを MPLAB X IDE シミュレータで実行する場合、`scanf()`関数によってテキストファイル入力を USART 経由で読み出す事ができます。これを実行するには以下の手順が必要です。

- MPLAB X IDE 内で USART IO 機能を有効にします。
- プロジェクトコード内で、IDE で選択したのと同じ USART を初期化して有効にします。
- プロジェクトコード内で、その USART から 1 バイトを読み出す「read-byte」関数を定義します。
- その read-byte 関数を `scanf()`関数から呼び出します。
- 入力データを格納したテキストファイルを用意し、レジスタ経由にて、前述の USART に対応する受信レジスタにこのファイルの内容を渡します。

一部のコンパイラでは、上記の USART 初期化と read-byte 関数を実装した汎用コードが提供される場合があります。その他のコンパイラでは、多くの場合、Microchip Code Configurator (MCC)を使ってこのコードを生成する事ができます。通常、MCC による既定値の UART 設定はシミュレータ上で正常に動作しますが、そのままでは最終アプリケーションで使えない可能性があります。USART の設定が済めば、`scanf()`に加えて、`stdin` から読み出す標準 I/O ライブラリ関数が使えるようになります。

MPLAB X IDE の詳細は『MPLAB® X IDE User's Guide』を参照してください。MCC ツールの詳細は『MPLAB® Code Configurator v3.xx User's Guide』を参照してください。

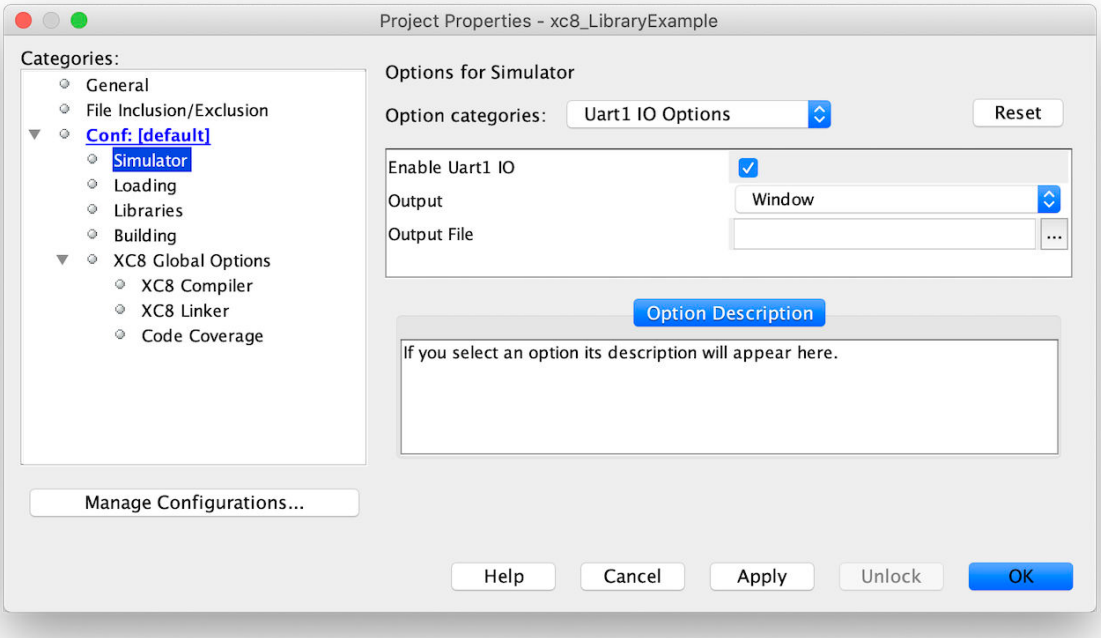
以下では、本コンパイラに固有の実装について詳細に説明します。

9.1.1 8 ビット PIC MCU 向けサンプルコード

MPLAB X IDE シミュレータでサンプルコードを実行する場合、IDE の UART IO 機能を使って stdout ストリームからの出力を表示できます。この UART IO 機能は大部分のデバイスで利用可能です。適切な設定により、printf() の出力および stdout へ書き込むその他の関数からの出力をシミュレータ上でのプログラム実行中に IDE に表示する事が可能となります。

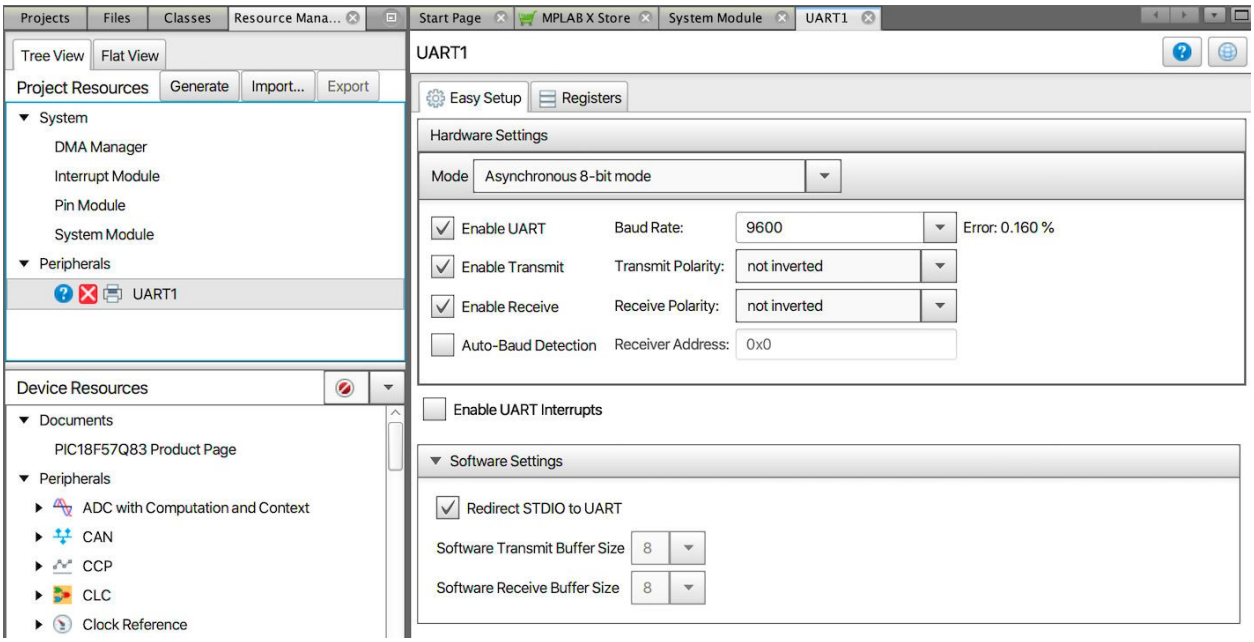
UART IO 機能は、**Project properties > Simulator> Uartx IO Options** 内の**[Enable Uartx IO]**チェックボックスで有効にします(図 9-1 参照)。複数の UART の中から 1 つを選択できる場合があります。その場合、コード内で書き込み先として使われる UART を選択します。出力は IDE 内のウィンドウに表示できる他、ダイアログ内の設定に応じてホストマシンへファイルを送信する事もできます。

図 9-1. MPLAB X IDE 内で UART IO を有効にする



MCC によって生成された UART 初期化関数と print-byte 関数は、シミュレータ上で使えます。**[UARTx]**タブ内の**[Redirect STDIO to UART]**チェックボックス(図 9-2 参照)を有効にした場合、printf()によって使われる print-byte 関数は MCC が生成する UART へのデータ送信用関数を呼び出します。既定値の通信設定はシミュレータで使えますが、UART をハードウェア上で使う場合はこれらの設定を変更する事が必要になる可能性があります。

図 9-2. MCC を使った UART の初期化



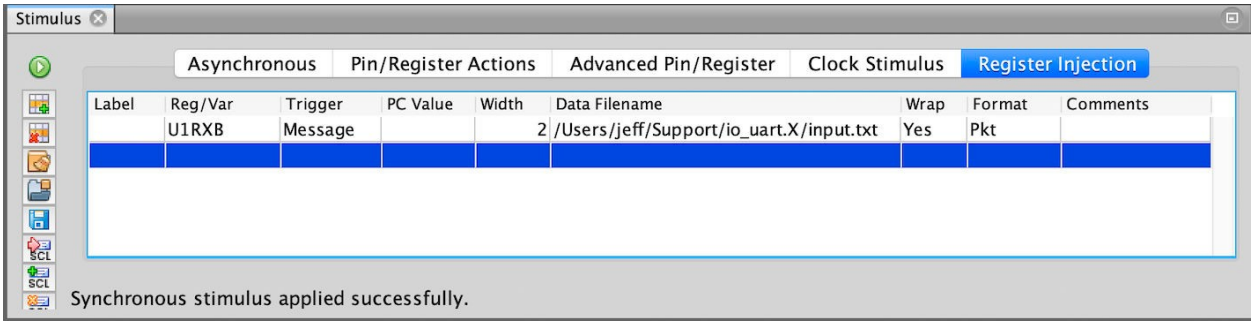
同じ UART IO 機能により、stdin ストリームからの入力の読み出しが可能となります。適切な設定により、シミュレータでのプログラムの実行中に scanf()および stdin から読み出すその他の関数によってファイルからの入力を取得する事が可能になります。

scanf()を使うサンプルコードを MPLAB X IDE シミュレータで実行する場合、UART を前述の通りに設定する必要があります。printf()の出力用に使う UART と同じ UART を入力用に使う事ができます。プログラムに文字入力を提供するため、適切な拡張子(.txt 等)を持つ空白のファイルを作成してプロジェクトに追加します。エディタでこのファイルを開き、プログラムへ送信する文字を書き込みます。これらの文字は、ダブルクオートで囲んだ文字列(例: "here is my input")として指定します。入力の読み出し方法を制御するコマンドをこのファイルに追加する事もできます(『MPLAB® X IDE User's Guide』参照)。

次に、**Window > Simulator > Stimulus** と操作してダイアログを開き、**[Register Injection]**を選択します(図 9-3 参照)。USART によって使われる受信レジスタの名前を**[Reg/Var]**フィールドに入力します(入力に応じてポップアップリストに一致するレジスタ名が表示されます)。このレジスタ名はデバイスごとに異なります。このレジスタの名前はデバイスのデータシートで確認してください。多くの場合、この名前は MCC によって作成された UARTx_Read()関数によって返されるため、そちらで確認する事もできます。**[Trigger]**フィールドが **Message** に設定され、**[Format]**フィールドが **Pkt** に設定されている事を確認します。**[Data Filename]**フィールドに、入力文字を書き込んだテキストファイルの名前とパスを入力します。最後に、左端の最上段にある緑のボタンをクリックする事によりスティミュラスを適用し、ダイアログの最下段に表示されるメッセージを確認します。緑のボタンは、スティミュラスの適用を ON/OFF します。

図 9-3 では、PIC18 デバイスの UART1 (受信レジスタとして U1RXB を使用) 向けにスティミュラスが正しく設定された状態を示しています。

図 9-3. UART によって読み出すファイルを指定する



スティミュラスが正しく設定されていれば、scanf()および stdin から読み出すその他の関数は指定されたファイルから入力を取得できます。

9.2 <xc.h>デバイス固有関数

ヘッダファイル<xc.h>はターゲット デバイスに固有の型、マクロ、関数で構成されます。<xc.h>は、特殊機能レジスタへのアクセスを提供するデバイス固有ヘッダファイルも含みます。

9.2.1 CLRWDT マクロ

このマクロはウォッチドッグ タイマをクリアします。

Include

<xc.h>

プロトタイプ

```
void CLRWDT(void);
```

注釈

このマクロは clrwdt 命令を実行します。

例

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
void main(void) {
  unsigned char c;

  SYSTEM_Initialize();
  WDTCON0bits.SWDTEN = 1;

  while (1) {
    c = PORTA;
    LATB = c;
    CLRWDT();
  }
}
```

9.2.2 di マクロ

このマクロは割り込みを無効にします。

Include

<xc.h>

プロトタイプ

```
void di(void);
```

注釈

このマクロは、INTCON レジスタ内の GIE ビットをクリアします。MCC が生成する INTERRUPT_GlobalInterruptDisable()関数は、di()マクロと同じタスクを実行します。

例

サンプルコード内の printf()または scanf() (および stdin または stdout ストリームを読み書きするその他の関数) の使用に関しては、9.1 「ライブラリ サンプルコード」を参照してください。

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>
unsigned int count;
void  __interrupt() myIsr (void) {
    if(INTCONbits.TMR0IE == 1 && INTCONbits.TMR0IF == 1)
        { INTCONbits.TMR0IF = 0;
        TMR0 = timer0ReloadVal;
        count++;
    }
```

```
}

unsigned int getTicks(void)
  { unsigned int val;

  di();         /* disable interrupts */
  val = count; /* protected non-atomic operation */
  ei();         /* re-enable interrupts */
  return val;
}

int main(void) {
  SYSTEM_Initialize();                /* MCC generated code */
  INTERRUPT_GlobalInterruptEnable();   /* MCC alternative to ei() */
  INTERRUPT_PeripheralInterruptEnable();
  while(getTicks() < 0x400)           /* wait for 0x400 overflows */
    continue;
  INTERRUPT_GlobalInterruptDisable();  /* MCC alternative to di() */
  printf("done");
  while(1)
    continue;
}
```

9.2.3 ei マクロ

このマクロは割り込みを有効にします。

Include

<xc.h>

プロトタイプ

```
void ei(void);
```

注釈

このマクロは、INTCON レジスタ内の GIE ビットをセットします。MCC が生成する INTERRUPT_GlobalInterruptEnable()関数は、ei()マクロと同じタスクを実行します。

例

サンプルコード内の printf()または scanf() (および stdin または stdout ストリームを読み書きするその他の関数)の使用に関しては、9.1 「ライブラリ サンプルコード」を参照してください。

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

unsigned int count;

void  interrupt() myIsr (void) {
    if(INTCONbits.TMR0IE == 1 && INTCONbits.TMR0IF == 1)
        { INTCONbits.TMR0IF = 0;
        TMR0 = timer0ReloadVal;
        count++;
    }
}

unsigned int getTicks(void)
  { unsigned int val;


  di();         /* disable interrupts */
  val = count; /* protected non-atomic operation */
  ei();         /* re-enable interrupts */
  return val;
}

int main(void) {
  SYSTEM_Initialize();                /* MCC generated code */
  INTERRUPT_GlobalInterruptEnable();   /* MCC alternative to ei() */
  INTERRUPT_PeripheralInterruptEnable();
  while(getTicks() < 0x400)           /* wait for 0x400 overflows */
    continue;
  INTERRUPT_GlobalInterruptDisable();  /* MCC alternative to di() */
```

```
  printf("done");
  while(1)
    continue;
}
```

9.2.4 eeprom_read 関数

この関数は、EEPROM メモリ空間からデータを読み出します。

Include

<xc.h>

プロトタイプ

```
unsigned char eeprom_read(unsigned char address);
```

引数

address EEPROM 内の読み出しアドレス

注釈

この関数は、EEPROM を備えた全てのミッドレンジ デバイス向けに使えます。PIC18 デバイスの場合、このルーチンを呼び出すと、レガシーの PIC18 周辺モジュール ライブラリ内の等価な関数(別途ダウンロードしてインストールする必要あり)が呼び出されます。PIC18 デバイス向けには、可能であれば MPLAB MCC を使って EEPROM アクセスコードを生成する事を推奨します。

この関数は、要求された動作を実行する前に、既に実行中の EEPROM への書き込みが完了するまで待機します。

例

以下の例では、1 つのマクロによってデバイスの EEPROM に保存されたデータを、3 種類の関数によって読み出しています。

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

__EEPROM_DATA(0x00, 0x20, 0x40, 0x60, 0x80, 0x00, 0x00, 0x00);

void main(void)
  { unsigned char
  byte;

  SYSTEM_Initialize();

  byte = DATAEE_ReadByte(2);  /* MCC-generated function */
  printf("Byte at address 2 in EEPROM is %x\n", byte);
  byte = eeprom_read(3);
  printf("Byte at address 3 in EEPROM is %x\n", byte);
  byte = EEPROM_READ(4);
  printf("Byte at address 4 in EEPROM is %x\n", byte);

  while (1) {
  }
}
```

出力例

```
Byte at address 2 in EEPROM is 40
Byte at address 3 in EEPROM is 60
Byte at address 4 in EEPROM is 80
```

9.2.5 eeprom_write 関数

この関数は、EEPROM メモリ空間にデータを書き込みます。

Include

<xc.h>

プロトタイプ

```
void eeprom_write(unsigned char address, unsigned char value);
```

引数

address	EEPROM 内の書き込みアドレス
value	EEPROM に書き込む値

注釈

この関数は、EEPROM を備えた全てのミッドレンジ デバイス向けに使えます。PIC18 デバイスの場合、このルーチンを呼び出すと、レガシーの PIC18 周辺モジュール ライブラリ内の等価な関数(別途ダウンロードしてインストールする必要あり)が呼び出されます。PIC18 デバイス向けには、可能であれば MPLAB MCC を使って EEPROM アクセスコードを生成する事を推奨します。

この関数は、要求された動作を実行する前に、既に実行中の EEPROM への書き込みが完了するまで待機します。この関数は、EEPROM 内の指定されたアドレス(address)に指定された値(value)を書き込みますが、この処理は関数から戻った時点では完了せず、EEPROM に書き込まれた新しいデータが有効になるまでに一定の時間が必要です。EEPROM に関する詳細情報は、ターゲット デバイスのデータシートを参照してください。

例

```
#include <xc.h>

int main(void)
{
  unsigned char data = 0x55;
  unsigned char addr = 0x20;

  eeprom_write(addr, data);
}
```

9.2.6 EEPROM_READ マクロ

このマクロは、EEPROM メモリ空間からデータを読み出します。

Include

<xc.h>

プロトタイプ

unsigned char EEPROM_READ(*scalar* address);

引数

address	EEPROM 内の読み出しアドレス

注釈

このマクロは、EEPROM を備えた全てのミッドレンジ デバイス向けに使えます。PIC18 デバイス向けには、可能であれば MPLAB MCC を使って EEPROM アクセスコードを生成する事を推奨します。

eeprom_read 関数とは異なり、このマクロは要求された動作を実行する前に既に実行中であった EEPROM への書き込みが完了するまで待機しません。このマクロは、指定されたアドレス(address) の値を返します。

例

以下の例では、1 つのマクロによってデバイスの EEPROM に保存されたデータを、3 種類の関数によって読み出しています。

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

__EEPROM_DATA(0x00, 0x20, 0x40, 0x60, 0x80, 0x00, 0x00, 0x00);

void main(void)
  { unsigned char
  byte;

  SYSTEM_Initialize();
```

```
  byte = DATAEE_ReadByte(2);  /* MCC-generated function */
  printf("Byte at address 2 in EEPROM is %x\n", byte);
  byte = eeprom_read(3);
  printf("Byte at address 3 in EEPROM is %x\n", byte);
  byte = EEPROM_READ(4);
  printf("Byte at address 4 in EEPROM is %x\n", byte);

  while (1) {
  }
}
```

出力例

```
Byte at address 2 in EEPROM is 40
Byte at address 3 in EEPROM is 60
Byte at address 4 in EEPROM is 80
```

9.2.7 EEPROM_WRITE マクロ

このマクロは、EEPROM メモリ空間にデータを書き込みます。

Include

<xc.h>

プロトタイプ

void EEPROM_WRITE(*scalar* address, *scalar* value);

引数

address	EEPROM 内の書き込みアドレス
value	EEPROM に書き込む値

注釈

このマクロは、EEPROM を備えた全てのミッドレンジ デバイス向けに使えます。PIC18 デバイス向けには、可能であれば MPLAB MCC を使って EEPROM アクセスコードを生成する事を推奨します。

この関数は、要求された動作を実行する前に、既に実行中の EEPROM への書き込みが完了するまで待機します。このマクロは指定された値(value)を EEPROM 内の指定されたアドレス(address) に書き込みます。

例

```
#include <xc.h>

int main (void)
{
  unsigned char data = 0x55;
  unsigned char addr = 0x20;

  EEPROM_WRITE(addr, data);
}
```

9.2.8 __EEPROM_DATA マクロ

このマクロは、プログラミング時にデバイスの EEPROM レジスタに値を保存します。

Include

<xc.h>

プロトタイプ

__EEPROM_DATA(value, value, value, value, value, value, value, value);

引数

value	EEPROM に書き込む 8 個の値

注釈

このマクロは、プログラミング時にデバイスの EEPROM レジスタに初期値を保存するために使います。

このマクロを呼び出すたびに、EEPROM に書き込む 8 バイトのブロック(8 個の値)を指定する必要があります。このマクロを繰り返し呼び出す事で、複数のブロックを保存できます。

このマクロは EEPROM アドレス 0 から書き込みを開始し、呼び出されるたびにアドレスを 8 ずつ自動的にインクリメントします。

例

以下の例では、1 つのマクロによってデバイスの EEPROM に保存されたデータを、3 種類の関数によって読み出しています。

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

__EEPROM_DATA(0x00, 0x20, 0x40, 0x60, 0x80, 0x00, 0x00, 0x00);

void main(void)
  { unsigned char
  byte;

  SYSTEM_Initialize();

  byte = DATAEE_ReadByte(2);  /* MCC-generated function */
  printf("Byte at address 2 in EEPROM is %x\n", byte);
  byte = eeprom_read(3);
  printf("Byte at address 3 in EEPROM is %x\n", byte);
  byte = EEPROM_READ(4);
  printf("Byte at address 4 in EEPROM is %x\n", byte);

  while (1) {
  }
}
```

出力例

```
Byte at address 2 in EEPROM is 40
Byte at address 3 in EEPROM is 60
Byte at address 4 in EEPROM is 80
```

9.2.9 _delay ビルトイン関数

このビルトイン関数は、実行を遅延させます。

Include

<xc.h>

プロトタイプ

```
void _delay(unsigned long cycles);
```

引数

cycles　遅延の命令サイクル数

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインラインアセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として指定された命令サイクル数の遅延を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な定数式である事が必要です。

_delay()ビルトイン関数は、ループと nop 命令を使って遅延を実装します。

指定された遅延時間(cycles) が定数式ではない場合、または 50,463,240 命令サイクルより長い場合、エラーとなります。この関数を複数回呼び出す事で、より長い遅延を実装できます。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/

unsigned char count;

void main(void)
  { SYSTEM_Initialize(
  );

  while (1)
    { LATA =
    0xFF;                /* wait 100 cycles*/
    _delay(100);
    LATA = 0xAA;         /* wait 100 cycles, clearing the watchdog */
    _delaywdt(100);
    LATA = 0x55;         /* wait 30 cycles */
    _delay3(10);
    LATA = 0xFF;         /* wait 800 micro seconds */
    __delay_us(800);
    LATA = 0x00;         /* wait 800 micro seconds, clearing the watchdog */
    __delaywdt_us(800);
    LATA = count;        /* wait half a second */
    __delay_ms(500);
    LATA = ~count;       /* wait half a second, clearing the watchdog */
    __delaywdt_ms(500);
    count++;
  }
}
```

9.2.10 _delaywdt ビルトイン関数

このビルトイン関数は、実行を遅延させます。

Include

<xc.h>

プロトタイプ

```
void _delaywdt(unsigned long cycles);
```

引数

cycles	遅延の命令サイクル数

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインラインアセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として指定された命令サイクル数の遅延を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な定数式である事が必要です。

_delay()ビルトイン関数は、ループと clrwdt 命令を使って遅延を実装します。

指定された遅延時間(cycles)が定数式ではない場合、または 50,463,240 命令サイクルより長い場合、エラーとなります。この関数を複数回呼び出す事で、より長い遅延を適用できます。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/
```

```
/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/

unsigned char count;

void main(void)
  { SYSTEM_Initialize(
  );

  while (1)
    { LATA =
    0xFF;                /* wait 100 cycles*/
    _delay(100);
    LATA = 0xAA;         /* wait 100 cycles, clearing the watchdog */
    _delaywdt(100);
    LATA = 0x55;         /* wait 30 cycles */
    _delay3(10);
    LATA = 0xFF;         /* wait 800 micro seconds */
    __delay_us(800);
    LATA = 0x00;         /* wait 800 micro seconds, clearing the watchdog */
    __delaywdt_us(800);
    LATA = count;        /* wait half a second */
    __delay_ms(500);
    LATA = ~count;       /* wait half a second, clearing the watchdog */
    __delaywdt_ms(500);
    count++;
  }
}
```

9.2.11 _delay3 ビルトイン関数

このビルトイン関数は、実行を遅延させます。

Include

<xc.h>

プロトタイプ

```
void _delay3(unsigned char loops);
```

引数

loops	遅延ループの回数(1 ループは 3 サイクル)

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインラインアセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として指定された命令サイクル数の 3 倍の遅延を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な 257 未満の定数式である事が必要です。

_delay3()ビルトイン関数は、ループを使って遅延を実装します。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/

unsigned char count;

void main(void)
  { SYSTEM_Initialize(
  );

  while (1) {
```

```
    LATA = 0xFF;
    _delay(100);         /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);      /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);         /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);     /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);  /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);     /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);  /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.12 __builtin_software_breakpoint ビルトイン関数

このビルトイン関数は、ソフトウェア ブレークポイントをトリガします。

- この関数は全てのデバイスで使えますが、8 ビット ベースライン PIC デバイスで使った場合、効果はありません。

Include

<xc.h>

プロトタイプ

```
void   __builtin_software_breakpoint(void);
```

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。デバッガを使ってコードを実行している時にこの関数が呼び出されると、無条件でソフトウェア ブレークポイントがトリガされます。

ソフトウェア ブレークポイント コードは、ミッドレンジおよび PIC18 デバイス向けにのみ生成されます。ベースライン デバイスはこの方法によるソフトウェア ブレークポイントをサポートしないため、このビルトイン関数が使われても無視します。

例

以下のサンプルコードには、MCC が生成した関数に対して 2 通りのブレークポイントが追加されています。

```
static i2c1_fsm_states_t I2C1_DO_BUS_COLLISION(void)
{
  // Clear bus collision status flag
  I2C1_MasterClearIrq();

  I2C1_Status.error = I2C1_FAIL;
  switch (I2C1_Status.callbackTable[I2C1_WRITE_COLLISION]
(I2C1_Status.callbackPayload[I2C1_WRITE_COLLISION])) {
  case I2C1_RESTART_READ:
    return I2C1_DO_SEND_RESTART_READ();
  case I2C1_RESTART_WRITE:
    __debug_break();                /* break when debugging only */
    return I2C1_DO_SEND_RESTART_WRITE();
  default:
    __builtin_software_breakpoint(); /* unconditional break */
    return I2C1_DO_RESET();
  }
}
```

9.2.13 __debug_break ビルトイン関数

このビルトイン関数は、デバッグビルド向けにソフトウェア ブレークポイントをトリガします。

Include

<xc.h>

プロトタイプ

```
void   __debug_break(void);
```

注釈

このビルトイン関数は、コード ジェネレータによってデバッグビルド向けに展開されますが、製品向けビルドの場合は無視されます。デバッガを使ってコードを実行している時にこの関数が呼び出されると、無条件でソフトウェア ブレークポイントがトリガされます。

ソフトウェア ブレークポイント コードは、ミッドレンジおよび PIC18 デバイス向けにのみ生成されます。ベースライン デバイスはこの方法によるソフトウェア ブレークポイントをサポートしないため、このビルトイン関数が使われても無視します。

例

以下のサンプルコードには、MCC が生成した関数に対して 2 通りのブレークポイントが追加されています。

```
static i2c1_fsm_states_t I2C1_DO_BUS_COLLISION(void)
{
  // Clear bus collision status flag
  I2C1_MasterClearIrq();

  I2C1_Status.error = I2C1_FAIL;
  switch (I2C1_Status.callbackTable[I2C1_WRITE_COLLISION]
(I2C1_Status.callbackPayload[I2C1_WRITE_COLLISION])) {
  case I2C1_RESTART_READ:
    return I2C1_DO_SEND_RESTART_READ();
  case I2C1_RESTART_WRITE:
    __debug_break();                /* break when debugging only */
    return I2C1_DO_SEND_RESTART_WRITE();
  default:
    __builtin_software_breakpoint(); /* unconditional break */
    return I2C1_DO_RESET();
  }
}
```

9.2.14 __delay_ms ビルトイン関数

このビルトイン関数は、実行を ms 単位で遅延させます。

Include

<xc.h>

プロトタイプ

```
void   __delay_ms(unsigned long time);
```

引数

time	遅延時間(単位は ms)

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインライン アセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として ms 単位で指定された遅延時間を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な定数式である事が必要です。

このマクロを使う前に、プリプロセッサ マクロ_XTAL_FREQ の定義が必要です。このマクロでは、システムによって使われるオシレータ周波数(単位は Hz)を指定する必要があります。このマクロは、これらの遅延の挙動を制御するだけであり、デバイスの実行速度には影響を及ぼしません。

__delay_ms()ビルトイン関数は、ループと nop 命令を使って遅延を実装します。

指定された遅延時間が定数式ではない場合、または長すぎる場合、エラーとなります。この関数を複数回呼び出す事で、より長い遅延を適用できます。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/
```

```
/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/

unsigned char count;

void main(void)
  { SYSTEM Initialize(

  while (1)
    { LATA =
    0xFF;               /* wait 100 cycles*/
    _delay(100);
    LATA = 0xAA;        /* wait 100 cycles, clearing the watchdog */
    _delaywdt(100);
    LATA = 0x55;        /* wait 30 cycles */
    _delay3(10);
    LATA = 0xFF;        /* wait 800 micro seconds */
    __delay_us(800);
    LATA = 0x00;        /* wait 800 micro seconds, clearing the watchdog */
    __delaywdt_us(800);
    LATA = count;       /* wait half a second */
    __delay_ms(500);
    LATA = ~count;      /* wait half a second, clearing the watchdog */
    __delaywdt_ms(500);
  }
}
```

9.2.15 __delaywdt_ms ビルトイン関数

このビルトイン関数は、実行を ms 単位で遅延させます。

Include

<xc.h>

プロトタイプ

```
void   __delaywdt_ms(unsigned long time);
```

引数

time	遅延時間(単位は ms)

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインラインアセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として ms 単位で指定された遅延時間を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な定数式である事が必要です。

このマクロを使う前に、プリプロセッサ マクロ_XTAL_FREQ の定義が必要です。このマクロでは、システムによって使われるオシレータ周波数(単位は Hz)を指定する必要があります。このマクロは、これらの遅延の挙動を制御するだけであり、デバイスの実行速度には影響を及ぼしません。

__delaywdt_ms()ビルトイン関数は、ループと clrwdt 命令を使って遅延を実装します。

指定された遅延時間が定数式ではない場合、または長すぎる場合、エラーとなります。この関数を複数回呼び出す事で、より長い遅延を適用できます。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/
```

```
unsigned char count;

void main(void)
  { SYSTEM_Initialize(

  while (1)
    { LATA =
    0xFF;                /* wait 100 cycles*/
    _delay(100);
    LATA = 0xAA;         /* wait 100 cycles, clearing the watchdog */
    _delaywdt(100);
    LATA = 0x55;         /* wait 30 cycles */
    _delay3(10);
    LATA = 0xFF;         /* wait 800 micro seconds */
    __delay_us(800);
    LATA = 0x00;         /* wait 800 micro seconds, clearing the watchdog */
    __delaywdt_us(800);
    LATA = count;        /* wait half a second */
    __delay_ms(500);
    LATA = ~count;       /* wait half a second, clearing the watchdog */
    __delaywdt_ms(500);
    count++;
}
```

9.2.16 __delay_us ビルトイン関数

このビルトイン関数は、実行を µs 単位で遅延させます。

Include

<xc.h>

プロトタイプ

```
void   __delay_us(unsigned long time);
```

引数

time	遅延時間(単位は µs)

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインラインアセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として µs 単位で指定された遅延時間を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な定数式である事が必要です。

このマクロを使う前に、プリプロセッサ マクロ_XTAL_FREQ の定義が必要です。このマクロでは、システムによって使われるオシレータ周波数(単位は Hz)を指定する必要があります。このマクロは、これらの遅延の挙動を制御するだけであり、デバイスの実行速度には影響を及ぼしません。

__delay_us()ビルトイン関数は、ループと nop 命令を使って遅延を実装します。

指定された遅延時間が定数式ではない場合、または長すぎる場合、エラーとなります。この関数を複数回呼び出す事で、より長い遅延を適用できます。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/

unsigned char count;
```

```
void main(void)
  { SYSTEM Initialize(

  while (1)
    { LATA =
    0xFF;                /* wait 100 cycles*/
    _delay(100);
    LATA = 0xAA;         /* wait 100 cycles, clearing the watchdog */
    _delaywdt(100);
    LATA = 0x55;         /* wait 30 cycles */
    _delay3(10);
    LATA = 0xFF;         /* wait 800 micro seconds */
    __delay_us(800);
    LATA = 0x00;         /* wait 800 micro seconds, clearing the watchdog */
    __delaywdt_us(800);
    LATA = count;        /* wait half a second */
    __delay_ms(500);
    LATA = ~count;       /* wait half a second, clearing the watchdog */
    __delaywdt_ms(500);
  }
}
```

9.2.17 __delaywdt_us ビルトイン関数

このビルトイン関数は、実行を µs 単位で遅延させます。

Include

<xc.h>

プロトタイプ

```
void   __delaywdt_us(unsigned long time);
```

引数

time	遅延時間(単位は µs)

注釈

このビルトイン関数は、コード ジェネレータによって展開されます。呼び出されると、このルーチンはインラインアセンブリの遅延シーケンスへと展開されます。このシーケンスは、引数として µs 単位で指定された遅延時間を適用するコードで構成されます。引数は、変数または関数呼び出しを含まずコンパイル時に完全に評価可能な定数式である事が必要です。

このマクロを使う前に、プリプロセッサ マクロ_XTAL_FREQ の定義が必要です。このマクロでは、システムによって使われるオシレータ周波数(単位は Hz)を指定する必要があります。このマクロは、これらの遅延の挙動を制御するだけであり、デバイスの実行速度には影響を及ぼしません。

__delaywdt_us()ビルトイン関数は、ループと clrwdt 命令を使って遅延を実装します。

指定された遅延時間が定数式ではない場合、または長すぎる場合、エラーとなります。この関数を複数回呼び出す事で、より長い遅延を適用できます。

例

以下のサンプルコードに、全てのビルトイン遅延ルーチンを示します。

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC.*/

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected.*/

unsigned char count;

void main(void)
  { SYSTEM_Initialize(
  );
```

```
  while (1)
    { LATA =
    0xFF;                /* wait 100 cycles*/
    _delay(100);
    LATA = 0xAA;         /* wait 100 cycles, clearing the watchdog */
    _delaywdt(100);
    LATA = 0x55;         /* wait 30 cycles */
    _delay3(10);
    LATA = 0xFF;         /* wait 800 micro seconds */
    __delay_us(800);
    LATA = 0x00;         /* wait 800 micro seconds, clearing the watchdog */
    __delaywdt_us(800);
    LATA = count;        /* wait half a second */
    __delay_ms(500);
    LATA = ~count;       /* wait half a second, clearing the watchdog */
    __delaywdt_ms(500);
  }
}
```

9.2.18 __fpnormalize 関数

この関数は、浮動小数点値を正規化します。

Include

<xc.h>

プロトタイプ

```
__fpnormalize(double value);
```

value 正規化する浮動小数点値

注釈

この関数を使うと、32 ビット浮動小数点数(デバイス側の計算結果による)を「厳密ではない」浮動小数点数値(5.3.4 「浮動小数点データ型」参照) に変換します。

この関数は、渡された値が非正規化数(表現不能なほど小さい値)であれば符号に拘わらず正の 0 にフラッシュします。

例

```
#include <xc.h>

int main(void)
{
  double input_fp;
  // read in a floating-point value from an external source
  input_fp = getFP();
  // ensure it is formatted using the relaxed rules
  input_fp = __fpnormalize(input_fp);
}
```

9.2.19 NOP マクロ

これは無演算のマクロです。

Include

<xc.h>

プロトタイプ

```
void NOP(void);
```

注釈

このマクロは nop 命令を実行します。nop 命令は、例えば、デバイスが特定状態である時にコードが確実に実行できるよう短い遅延を適用するために使えます。あるいは、デバッグ用にブレークポイントを実装する位置として便利に使えます。

例

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/

void main(void)
  { SYSTEM_Initialize();

  INTERRUPT_GlobalInterruptEnable();
  WeatherStation_initialize();

  while (1) {
    if (tmr1_tick == 1)
      { WeatherStation_Print(
      ); tmr1_tick = 0;
    }
    SLEEP(); // wait for interrupt
    NOP();   // ensure instruction pre-fetched while sleeping is safe to execute on awakening
  }
}
```

9.2.20 __osccal_val ビルトイン関数

このビルトイン関数は、内部オシレータの校正定数を返します。

Include

<xc.h>

プロトタイプ

```
unsigned char   __osccal_val(void);
```

注釈

このビルトイン関数は、オシレータの設定値を格納した retlw 命令のアドレスに対応するラベルへと展開されます。この関数を呼び出すと、デバイスのオシレータ設定値が返されます。返された値は任意の式内で使えます。

この関数は、プログラムメモリ内にオシレータ設定値が書き込み済みで出荷されるデバイス向けにのみ使えます。この関数は、実行時起動コードによって自動的に呼び出されるため(ユーザが無効に設定した場合を除く、4.6.1.15「oscca オプション」参照)、内部オシレータを校正するためにこの関数を明示的に呼び出す必要はありません。

例

```
#include <xc.h>

int main(void)
{
  OSCCAL = __osccal_val();
}
```

9.2.21 READTIMERx マクロ

このマクロはタイマモジュールを読み出します。

Include

<xc.h>

プロトタイプ

```
unsigned short READTIMERx(void);
```

戻り値

対応する TMR レジスタの値

注釈

READTIMERx()マクロは PIC18 プロジェクト向けに使えます。この関数は、TMR*x* レジスタ(*x* は 0、1、3 のいずれか)の保持値を返します。

例

```
#include <xc.h>

int main(void)
{
  while(READTIMER0() != 0xFF)
    continue;
}
```

9.2.22 RESET マクロ

このマクロはデバイスをリセットします。

Include

<xc.h>

プロトタイプ

void RESET(void);

注釈

このマクロは、デバイスに reset 命令が実装されている場合には reset 命令を出力します。デバイスが reset 命令を実装していない場合、ウォッチドック タイマを有効とする前提で、自分自身へのジャンプ命令を配置します。デバイスはウォッチドック タイマのオーバーフローによりリセットされます。使用するデバイスに reset 命令が実装されているかどうかは、デバイス データシートで確認してください。

例

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/

void main(void)
  { SYSTEM_Initialize();
  DISP_Initialize();

  INTERRUPT_GlobalInterruptEnable();

  while (1) {
    if(DISP_needsUpdate)
      { if(DISP_update() ==
      0)
        RESET();  // something's gone horribly wrong
    }
  }
}
```

9.2.23 SLEEP マクロ

このマクロは、デバイスをスリープモードへ移行させます。

Include

<xc.h>

プロトタイプ

void SLEEP(void);

注釈

このマクロは sleep 命令を実行します。

例

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/

void main(void)
  { SYSTEM_Initialize();

  INTERRUPT_GlobalInterruptEnable();
```

```
  WeatherStation_initialize();

  while (1) {
    if (tmr1_tick == 1)
      { WeatherStation_Print(
      ); tmr1_tick = 0;
    }
    SLEEP(); // wait for interrupt
    NOP();   // ensure instruction pre-fetched while sleeping is safe to execute on awakening
  }
}
```

9.2.24 WRITETIMER マクロ

このマクロはタイマモジュールに書き込みます。

Include

<xc.h>

プロトタイプ

void WRITETIMER*x*(int n);

注釈

WRITETIMER*x*()マクロは PIC18 プロジェクト向けに使えます。このマクロは 2 バイト引数の両方のバイトを TMR*x* レジスタ(*x* は 0、1、3 のいずれか) に書き込みます。このマクロを使うと、TMR*x* レジスタに正しいバイト順で書き込む事ができます。このレジスタの全体に直接アクセスした場合、正しいバイト順は保証されません。

例

```
#include <xc.h>

int main(void)
{
  WRITETIMER0(0x4A);
}
```

10. エラーおよび警告メッセージ

以下では、MPLAB XC8 C コンパイラのエラー、警告、アドバイザリ メッセージについて説明します。これには、以前の HI-TECH C コンパイラと全てのコンパイラ バージョンからのメッセージが含まれます。従って、実際にご使用になるコンパイラ バージョンとは無関係なメッセージも記載されています。

各メッセージの前に、一意のメッセージ番号がカッコ付きで示されます。この番号は、コンパイラがメッセージを発行する際にも示されます。以下では、メッセージをこの番号順に記載しています。

各メッセージの末尾には、そのメッセージを生成可能なアプリケーションの名前がカッコ付きで示されます。一部のメッセージの説明では、そのメッセージの原因となり得るコードまたはオプションの例が示されます。メッセージ テキスト内の「*」は、コンパイラによって状況に応じた適切な文字列に置換されます。

C またはアセンブリ ソースコード内の 1 つの問題によって複数のメッセージが生成される可能性があります。その場合、メッセージが表示された順番に問題を解消する必要があります。

10.1 メッセージ 0～499

(1) too many errors (*) (全アプリケーション)
エラーの数が上限値を超えたため、実行中のコンパイラ アプリケーションは直ちに処理を停止します。引き続き残りの未コンパイル ソースファイルは処理されますが、後続のコンパイラ アプリケーションは実行されません。許容最大エラー数は、`-fmax-errors` オプション(4.6.4.1 「max-errors オプション」により調整できます。

(2) error/warning (*) generated but no description available (全アプリケーション)
実行中のコンパイラ アプリケーションがメッセージ (アドバイザリ/警告/エラー)を発行したものの、メッセージ記述ファイル(MDF) 内に該当するメッセージ テキストが存在しません。原因として、MDF が最新ではないか、そのメッセージが選択された言語に翻訳されていない可能性が考えられます。

(3) malformed error information on line * in file * (全アプリケーション)
コンパイラは選択された言語向けのメッセージ記述ファイル(MDF)を読み出せませんでした。MDF が破損している可能性があります。

(100) unterminated #if[n][def] block from line * (プリプロセッサ)
`#if` で始まるブロックが対応する`#endif` で終了していません。
例:

```
#if INPUT        /* error flagged here */
int main(void)
{
    run();
}                /* no #endif was found in this module */
```

(101) #* cannot follow #else (プリプロセッサ)
`#else` または`#elif` が`#else` と同じ条件ブロック内で使われています。これらは`#if` の後でのみ使えます。
例:

```
#ifdef FOO
    result = foo;
#else
    result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
    result = next(0);
#endif
```

(102) #* must be in an #if (プリプロセッサ)
#elif、#else、#endif ディレクティブの前に対応する#if 行が存在しません。対応する#if 行が存在しているのにこのエラーが発生する場合、余分な#endif が使われていないか、またはコメントが正しく終了しているか等を確認してください。
例:

```
#ifdef FOO
    result = foo;
#endif
    result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
    result = next(0);
#endif
```

(103) #error: * (プリプロセッサ)
このエラーはプログラマによって生成され、意図したエラー生成の原因となっているディレクティブが存在する事を示します。通常これはコンパイル時間定義等を確認するために使われます。このディレクティブを削除すればエラーは解消しますが、削除する前にそのディレクティブがそこで使われている理由を確認してください。

(104) preprocessor #assert failure (プリプロセッサ)
プリプロセッサ ディレクティブ#assert に対する引数が 0 であると評価されました。これはプログラマに起因するエラーです。
例:

```
#assert SIZE == 4 /* size should never be 4 */
```

(105) no #asm before #endasm (プリプロセッサ)
#endasm 演算子の前に対応する#asm が存在しません。
例:

```
void cleardog(void)
{
    clrwdt
    #endasm /* in-line assembler ends here, only where did it begin?*/
}
```

(106) nested #asm directives (プリプロセッサ)
#asm ディレクティブがネストされています。#endasm ディレクティブの書き忘れまたはスペルミスがないか確認してください。
例:

```
#asm
    MOVE  r0, #0aah
#asm      ; previous #asm must be closed before opening another
    SLEEP
#endasm
```

(107) illegal # directive “*” (プリプロセッサ、パーサー)
コンパイラはこの#ディレクティブを認識しません。ディレクティブ トークンのスペルミスが原因かもしれません。
例:

```
#indef DEBUG  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] without an argument (プリプロセッサ)
プリプロセッサ ディレクティブ#if、#ifdef、#ifndef に対して引数が指定されていません。#if に対する引数は式で指定し、#ifdef または#ifndef に対する引数は 1 つの名前で指定する必要があります。
例:

```
#if            /* oops -- no argument to check */
output = 10;
#else
output = 20;
#endif
```

(109) #include syntax error (プリプロセッサ)
#include に対する引数(ファイル名) の構文が正しくありません。#include に対する引数には、ダブルクオート " "または山カッコ< >で囲んだファイル名を指定する必要があります。この引数にスペースを含めてはいけません。閉じ側のクオートまたは山カッコが抜けていないか確認してください。この行には、コメント以外は何も追加してはいけません。
例:

```
#include stdio.h  /* oops -- should be:#include <stdio.h> */
```

(110)) too many file arguments; usage: cpp [input [output]] (プリプロセッサ)
cpp に対するファイル引数は最大で 2 つまでしか指定できません。プリプロセッサをコンパイラ ドライバから実行している時にこのエラーが発生した場合、Microchip 社の技術サポートにお問い合わせください。

(111) redefining preprocessor macro “*” (プリプロセッサ)
マクロが元の定義とは異なる内容で再定義されています。意図的にマクロを再定義する必要がある場合、再定義の前に#undef を使って元の定義を無効にする必要があります。
例:

```
#define ONE 1
/* elsewhere:*/
/* Is this correct?It will overwrite the first definition.*/
#define ONE one
```

(112) #define syntax error (プリプロセッサ)
マクロの定義に構文エラーがあります。マクロまたは仮パラメータの名前が文字で始まらないか、閉じカッコ「)」が抜けている可能性があります。
例:

```
#define FOO(a, 2b) bar(a, 2b)  /* 2b is not to be!*/
```

(113) unterminated string in preprocessor macro body (プリプロセッサ、アセンブラ)
マクロ定義内に、クオートで閉じられていない文字列が存在します。

(114) illegal #undef argument (プリプロセッサ)
#undef に対する引数は無効な名前です。名前は英文字で始まる必要があります。
例:

```
#undef 6YYY /* this isn’t a valid symbol name */
```

(115) recursive preprocessor macro definition of “*” defined by “*” (プリプロセッサ)
このマクロは、展開された時にそれ自身の再帰的展開が生じるような形で定義されています。

(116) end of file within preprocessor macro argument from line * (プリプロセッサ)
マクロ引数の終端がありません。マクロ呼び出しの末尾の閉じカッコが抜けている可能性があります。このメッセージには、マクロ引数の開始行番号が示されます。
例:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* oops -- where is the closing bracket? */
```

(117) misplaced constant in #if (プリプロセッサ)
#if 式内の定数の位置が構文的に正しくありません。演算子が欠落している可能性があります。
例:

```
#if FOO BAR /* oops -- did you mean:#if FOO == BAR ?*/
```

(118) stack overflow processing #if expression (プリプロセッサ)
#if 式の評価中にプリプロセッサがスタックを使い果たしました。式を単純化する必要があります。カッコで囲まれた副次式の数が多すぎる可能性があります。

(119) invalid expression in #if line (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(120) operator “*” in incorrect context (プリプロセッサ)
#if 式内の演算子の配置が正しくありません(例: 2 つの二項演算子の間に値が存在しない)。
例:

```
#if FOO * % BAR == 4 /* what is "* %" ?*/
#define BIG
#endif
```

(121) expression stack overflow at operator “*” (プリプロセッサ)
#if 行内の式を評価するためにスタック (サイズは 128) が使い果たされました。式が非常に複雑であるためにスタックのオーバーフローが発生した可能性があります。式を単純化してください。

(122) unbalanced parenthesis at operator “*” (プリプロセッサ)
#if 式内の開きカッコと閉じカッコの数が一致しません。
例:

```
#if ((A) + (B) /* oops -- a missing ), I think */
#define ADDED
#endif
```

(123) misplaced “?” or “:”; previous operator is “*” (プリプロセッサ)
#if 式内でコロン演算子と?演算子が正しく対応していません。
例:

```
#if XXX :YYY /* did you mean:#if COND ?XXX :YYY */
```

(124) illegal character “*” in #if (プリプロセッサ)
#if 式内で無効な文字が使われています。英文字、数字、有効な演算子を構成する文字だけが使えます。
例:

```
#if YYY /* what are these characters doing here?*/
int m;
#endif
```

(125) illegal character (* decimal) in #if (プリプロセッサ)
#if 式内で印字不可能文字が使われています。英文字、数字、有効な演算子を構成する文字だけが使えます。
例:

```
#if ^S YYY /* what is this control characters doing here?*/
int m;
#endif
```

(126) strings can’t be used in #if (プリプロセッサ)
#if 式内でプリプロセッサが許容しない文字列が使われています。
例:

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if (プリプロセッサ)
プリプロセッサ式内の defined()疑似関数には、引数として 1 つの名前を指定する必要があります。この名前は英文字で始まり、丸カッコで囲まれている必要があります。

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
input = read();
#endif
```

(128) illegal operator in #if (プリプロセッサ)
#if 式内で無効な演算子が使われています。構文が正しいか確認してください。
例:

```
#if FOO = 6 /* oops -- should that be:#if FOO == 5 ?*/
```

(129) unexpected “\” in #if (プリプロセッサ)
#if 文内で無効なバックスラッシュ文字が使われています。
例:

```
#if FOO == \34
#define BIG
#endif
```

(130) unknown type “*” in #[el]if sizeof() (プリプロセッサ)
プリプロセッサ sizeof()内で不明な型が使われています。プリプロセッサ sizeof()は、基本型または基本型に対するポインタを評価する事しかできません。
例:

```
#if sizeof(unt) == 2 /* should be:#if sizeof(int) == 2 */ i
= 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof() (プリプロセッサ)
#if 式内の sizeof()に対する引数で無効な型の組み合わせが使われています。
例:

```
/* To sign, or not to sign, that is the error.*/
#if sizeof(signed unsigned int) == 2
i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof() (プリプロセッサ)
#if 式内の sizeof()で型が指定されていません。プリプロセッサ式内の sizeof()に対する引数は、基本型または基本型に対するポインタである事が必要です。
例:

```
#if sizeof() /* oops -- size of what?*/ i
= 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof() (プリプロセッサ)
sizeof()式の評価中にプリプロセッサで内部エラーが発生しました。型指定子が正しいか確認してください。これは内部エラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(134) syntax error in #[el]if sizeof() (プリプロセッサ)
#if 式内の sizeof()に対する引数で構文エラーが見つかりました。原因としてカッコの数の不一致等が考えられます。
例:

```
#if sizeof(int == 2)  // oops - should be:#if sizeof(int) == 2
i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if (プリプロセッサ)
プリプロセッサは式内の演算子を認識できませんでした。これは内部エラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(137) strange character “*” after ## (プリプロセッサ)
トークン連結演算子##の後で英数字以外の文字が使われています。この演算子の結果は有効なトークンとなる必要があるため、オペランドは英数字だけを含むトークンである事が必要です。
例:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(138) strange character (*) after ## (プリプロセッサ)
トークン連結演算子##の後で印字不可能文字が使われています。この演算子の結果は有効なトークンとなる必要があるため、オペランドは英数字だけを含むトークンである事が必要です。
例:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(139) end of file in comment (プリプロセッサ)
コメント内でファイル終端(EOF)に達しました。コメント終了フラグが抜けていないか確認してください。
例:

```
/* Here the comment begins.I’m not sure where I end, though
}
```

(140) can’t open * file “*”:* (ドライバ、プリプロセッサ、コード ジェネレータ、アセンブラ)
指定されたコマンドファイルを開く事ができませんでした。コマンドライン上のファイル名とパスが正しいスペルで指定されているか確認してください。
例:

```
xc8 @communds
```

正しくは:

```
xc8 @commands
```

(141) can’t open * file “*”:* (全アプリケーション)
出力ファイルを生成できませんでした。コマンドライン上のファイル名とパスが正しいスペルで指定されているか確認してください。

(144) too many nested #if blocks (プリプロセッサ)
`#if`、`#ifdef` 等のブロックのネストの深さが許容最大値(32)を超えています。

(146) #include filename too long (プリプロセッサ)
インクルード ファイルの名前が長すぎるため、ファイルの検索中に内部バッファのサイズを超えました。このバッファサイズは 4096 バイトであるため、このエラーが発生する事はほとんどありません。

(147) too many #include directories specified (プリプロセッサ)
インクルード ファイルの検索用に指定されたディレクトリの数が最大許容数(7)を超えています。ドライバで指定するディレクトリの数を減らしてください。

(148) too many arguments for preprocessor macro (プリプロセッサ)
マクロのパラメータの数が C 規格の最大数(31)を超えています。

(149) preprocessor macro work area overflow (プリプロセッサ)
マクロ展開の総長が内部テーブルのサイズを超えています。このテーブルのサイズは通常 32768 バイトです。従って、マクロ展開の総長が 32K バイトを超える事はできません。

(150) illega l“__” preprocessor macro “*” (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(151) too many arguments in preprocessor macro expansion (プリプロセッサ)
マクロ呼び出しで指定された引数の数が多すぎます。指定可能な最大数は 31 です。

(152) bad dp/nargs in openpar(): c = * (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(153) out of space in preprocessor macro * argument expansion (プリプロセッサ)
マクロ引数が内部バッファのサイズを超えています。このバッファのサイズは通常 4096 バイトです。

(155) work buffer overflow concatenating “*” (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(156) work buffer “*” overflow (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(157) can’t allocate * bytes of memory (コード ジェネレータ、アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(158) invalid disable in preprocessor macro “*” (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(159) too many calls to unget() (プリプロセッサ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(161) control line “*” within preprocessor macro expansion (プリプロセッサ)
マクロの展開時に#で始まるプリプロセッサ制御行が見つかりました。これは許容されません。

(162) #warning: * (プリプロセッサ、ドライバ)
この警告は、ユーザが定義した#warning プリプロセッサ ディレクティブによって生成される場合と、ドライバによるマップファイルの読み出しで問題が生じたために生成される場合があります。後者の場合、Microchip 社技術サポートに詳細をお問い合わせください。

(163) unexpected text in control line ignored (プリプロセッサ)
制御行の末尾に余分な文字が書かれていた場合、この警告が生成され、余分なテキストは無視されます。そのようなテキストは、C 規格に従ってコメント内に書く事を推奨します。
例:

```
#if defined(END)
#define NEXT
#endif END     /* END would be better in a comment here */
```

(164) #include filename “*” was converted to lower case (プリプロセッサ)
#include で指定されたファイル名は小文字へと変換されました。
例:

```
#include <STDIO.H> /* oops -- should be:#include <stdio.h> */
```

(165) #include filename “*” does not match actual name (check upper/lower case) (プリプロセッサ)
Windows バージョンの場合、この警告は#include で指定されたのと同じスペルのファイルが見つかり、実際にインクルードされましたが、ファイル名の大文字/小文字は完全に一致しなかった事を意味します。例えば、#include “code.c”と指定した場合、Code.c が見つかればインクルードされます。Linux バージョンの場合、この警告はファイルが見つからなかった場合に生成されます。

(166) too few values specified with option “*” (プリプロセッサ)
プリプロセッサ(cpp) -S オプションに渡された値のリストが不完全です。プリプロセッサをコンパイラ ドライバから呼び出した場合、この問題は発生しません。このオプションには char、short、int、long、float、double 型のサイズを指定する値を渡す必要があります。

(167) too many values specified with -S option; “*” unused (プリプロセッサ)
-S プリプロセッサ オプションに渡された値の数が多すぎます。メッセージ(166)を参照してください。

(168) unknown option “*” (全アプリケーション)
指定されたオプションが認識されないためエラーが発生しました。

(169) strange character (*) after ## (プリプロセッサ)
#の後の文字が適切ではありません。

(170) symbol “*” in undef was never defined (プリプロセッサ)
#undef への引数として渡されたシンボルが未定義です。一部のコンパイラでは、この警告を無効にできます。この警告は、例えば以下のコードにより回避できます。
例:

```
#ifdef SYM
#undef SYM /* only undefine if defined */
#endif
```

(171) wrong number of preprocessor macro arguments for “*” (* instead of *) (プリプロセッサ)
マクロが不適正な数の引数を使って呼び出されています。
例:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)        /* oops -- only two arguments required */
```

(172) formal parameter expected after # (プリプロセッサ)
文字列化演算子#の後には仮マクロ パラメータが必要です。この演算子はプリプロセッサ制御行の先頭で使われる#とは異なる事に注意してください。
例:

```
#define str(x) #y  /* oops -- did you mean x instead of y?*/
```

トークンを文字列化する場合、専用のマクロを定義する必要があります。

例:

```
#define  mkstr (x) #x
```

上記を定義した後に__mkstr__(*token*)を使う事で、*token* を文字列へ変換します。

(173) undefined symbol “*” in #if; 0 used (プリプロセッサ)
#if 式内のシンボルは定義済みのプリプロセッサ マクロではありません。この式を評価するために、このシンボルの評価値は 0 として扱われました。一部のコンパイラでは、この警告を無効にできます。
例:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
#define GOOD
#endif
```

(174) multi-byte constant “*” isn’t portable (プリプロセッサ)
マルチバイト定数は移植可能ではないため、コンパイラの後続の処理で排除されます。
例:

```
#if CHAR == ’ab’
#define MULTI
#endif
```

(175) division by zero in #if; zero result assumed (プリプロセッサ)
#if 式内に 0 による除算が存在し、その結果は 0 として扱われました。
例:

```
#if foo/0  /* divide by 0: was this what you were intending?*/
int a;
#endif
```

(176) missing newline (プリプロセッサ)
行末の改行文字が欠落しています。最終行を含む全ての行の行末には改行文字が必要です。この問題は通常エディタによって生じます。

(177) symbol “*” in -U option was never defined (プリプロセッサ)
プリプロセッサに対して-U オプションで指定されたマクロは内部で定義された物ではないため、未定義化はできません。

(179) nested comments (プリプロセッサ)
この警告は、ネストされたコメントが見つかった場合に発行されます。直前のコメント終了記号が抜けているか正しく書かれていない可能性があります。
例:

```
output = 0; /* a comment that was left unterminated
flag = TRUE; /* next comment:
hey, where did this line go?*/
```

(180) unterminated comment in included file (プリプロセッサ)
インクルードされたファイル内で始まったコメントは、そのファイル内で終了する必要があります。

(181) non-scalar types can’t be converted to other types (パーサー)
構造体、共用体、配列を異なる型へ変換する事はできません。
例:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types (パーサー)
この式は、非互換な型の間の変換(例: 構造体型から整数型への変換等)を意味します。
例:

```
struct LAYOUT layout;
int i;
layout = i;          /* int cannot be converted to struct */
```

たとえ構造体が int 型のみを格納している場合でも、その構造体を int 型変数に代入する事はできません(その逆も不可)。

(183) function or function pointer required (パーサー)
関数または関数ポインタのみが関数呼び出しにより呼び出せます。
例:

```
int a, b, c, d;
a = b(c+d);   /* b is not a function -- did you mean a = b*(c+d) ? */
```

(184) calling an interrupt function is illegal (パーサー)
割り込み関数を他の関数から呼び出す事はできません。割り込み関数は、ハードウェア(またはソフトウェア)割り込みによってのみ呼び出し可能です。なぜなら、これらの関数は割り込みからの呼び出し専用の開始および終了コードを使うからです。割り込み関数から通常の関数を呼び出す事はできます。

(185) function does not take arguments (パーサー、コード ジェネレータ)
パラメータを持たない関数が、1 つまたは複数の引数を使って呼び出されています。
例:

```
int get_value(void);
int main(void)
{
int input;
input = get_value(6); /* oops --
parameter should not be here */
}
```

(186) too many function arguments (パーサー)
関数呼び出しの引数の数が多すぎます。
例:

```
void add(int a, int b);
add(5, 7, input);      /* call has too many arguments */
```

(187) too few function arguments (パーサー)
関数呼び出しの引数の数が少なすぎます。
例:

```
void add(int a, int b);
add(5);                 /* this call needs more arguments */
```

(188) constant expression required (パーサー)
この文脈では、コンパイル時に定数として評価可能な式が必要です。
例:

```
int a;
switch(input)
{ case a:/* oops!
cannot use variable as part of a case label */
input++;
}
```

(189) illegal type for array dimension (パーサー)
配列の次元は整数型または列挙値である事が必要です。
例:

```
int array[12.5];  /* oops -- twelve and a half elements, eh?*/
```

(190) illegal type for index expression (パーサー)
配列の添字は整数値または列挙値で指定する必要があります。
例:

```
int i, array[10];
i = array[3.5];   /* oops --
exactly which element do you mean?*/
```

(191) cast type must be scalar or void (パーサー)
型変換(丸カッコで囲まれた抽象的型宣言子)では、スカラー型(配列でも構造体でもない型)または `void` 型を指定する必要があります。
例:

```
lip = (long [])input; /* oops -- possibly: lip = (long *)input */
```

(192) undefined identifier “*” (パーサー)
このシンボルは定義も宣言もされていません。確かに定義済みである場合、スペルミスがないか確認してください。

(193) not a variable identifier “*” (パーサー)
この識別子は変数ではありません。これは変数以外の何らかのオブジェクト (ラベル等)であると思われます。

(194) “)” expected (パーサー)
閉じカッコ「)」がこの位置で必要です。式内でこのカッコが抜けているか、その他の構文エラーが存在する可能性があります。このエラーは、コードが最初に意味を成さなくなる行で発生します。従って、このエラーは不完全な式の直後の命令文で発生する可能性があります。
例:

```
if(a == b  /* the closing parenthesis is missing here */
b = 0;   /* the error is flagged here */
```

(195) expression syntax (パーサー)
この式の構文は正しくないため、コンパイラによる解析はできません。
例:

```
a /=% b;  /* oops -- possibly that should be: a /= b; */
```

(196) struct/union required (パーサー)
ピリオド「.」の前に構造体または共用体識別子が必要です。
例:

```
int a;
a.b = 9; /* oops -- a is not a structure */
```

(197) struct/union member expected (パーサー)
ピリオド「.」または矢印「->」の後に構造体または共用体メンバーの名前が必要です。

(198) undefined struct/union “*” (パーサー)
指定された構造体または共用体タグは未定義です。
例:

```
struct WHAT what; /* a definition for WHAT was never seen */
```

(199) logical type required (パーサー)
if、while 文または論理演算子(!、&&等)に対するオペランドとして使われる式はスカラー整数型である事が必要です。
例:

```
struct FORMAT format;
if(format)            /* this operand must be a scaler type */
    format.a = 0;
```

(200) taking the address of a register variable is illegal (パーサー)
レジスタ宣言された変数にはメモリ内の記憶域を割り当てられないため、&演算子によるアドレス取得はできません。
例:

```
int * proc(register int in)
{
    int * ip = &in;
    /* oops -- in cannot have an address to take */
    return ip;
}
```

(201) taking the address of this object is illegal (パーサー)
&演算子のオペランドであった式がメモリ記憶域を示す式(lvalue: 名前を持つオブジェクト等)ではないため、そのアドレスを定義する事はできません。
例:

```
ip = &8;  /* oops -- you cannot take the address of a literal */
```

(202) only lvalues can be assigned to or modified (パーサー)
lvalue (アドレス指定可能な記憶域を直接示す識別子または式)のみが代入または変更可能です。
例:

```
int array[10];
int * ip;
char c;
array = ip;   /* array is not a variable, it cannot be written to */
```

型変換から lvalue は得られません。

例:

```
/* the contents of c cast to int is only a intermediate value */
(int)c = 1;
```

しかし、ポインタを使ってこれを書く事ができます。

```
*(int *)&c = 1
```

(203) illegal operation on bit variable (パーサー)
ビット変数に対して無効な演算が使われています。一部の演算はビット変数に対して使えません。
例:

```
bit   b;
int * ip;
ip = &b;  /* oops -- cannot take the address of a bit object */
```

(204) void function can't return a value (パーサー)
void 関数は値を返す事ができないため、return 文で戻り値を指定してはいけません。
例:

```
void run(void)
{
    step();
    return 1; /* either run should not be void, or remove the 1 */
}
```

(205) integral type required (パーサー)
この演算子には整数型のオペランドが必要です。

(206) illegal use of void expression (パーサー)
void 式は値を持たないため、値を持つ式が必要な場合に使う事はできません(算術演算子に対するオペランドとして使ってはいけません)。

(207) simple type required for "*" (パーサー)
この演算子のオペランドは単純な型(すなわち配列でも構造体でもない型)である事が必要です。

(208) operands of "*" not same type (パーサー)
この演算子のオペランドは異なるポインタ型です。
例:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;  /* result of ?: will be int * or char * */
```

おそらく、このコードは以下を意図した物と思われます。

```
cp = flag ?(char *)ip : cp2;
```

(209) type conflict (パーサー)
この演算子のオペランドの型は矛盾しています。

(210) bad size list (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(211) taking sizeof bit is illegal (パーサー)
C 言語の__bit 型に対する sizeof()演算子の使用は禁止されています。sizeof()演算子は、与えられた型のサイズ(その型のオブジェクトを保存するために必要なバイト数)を返します。従って、この演算子を__bit 型に使っても意味がありません。

(212) missing number after pragma "pack" (パーサー)
pragma pack の引数は 10 進数である事が必要です。これは構造体内の各メンバーの整列方法を指定します。一部のプロセッサは整列を強要し、奇数アドレスに対するワードフェッチは正しく動作しないため、pragma pack の使用には注意が必要です。
例:

```
#pragma pack /* what is the alignment value */
```

以下は正しい使用例です。

```
#pragma pack 2
```

(214) missing number after pragma “interrupt_level” (パーサー)
pragma interrupt_level は、割り込みレベルを示す引数を必要とします。この引数の値はミッドレンジ デバイス対して 1、PIC18 デバイスに対して 1 または 2 です。

(215) missing argument to pragma “switch” (パーサー)
pragma switch には引数として auto、direct、simple のいずれかが必要です。
例:

```
#pragma switch /* oops -- this requires a switch mode */
```

以下は正しい使用例です。

```
#pragma switch simple
```

(216) missing argument to pragma “psect” (パーサー)
pragma psect には引数として *oldname = newname* が必要です(*oldname*: 既存(コンパイラにとって既知)の psect 名、*newname*: 新しい名前)。
例:

```
#pragma psect /* oops -- this requires an psect to redirect */
```

以下は正しい使用例です。

```
#pragma psect text=specialtext
```

(218) missing name after pragma “inline” (パーサー)
inline pragma の後に関数の名前が必要です。名前がコード ジェネレータによって認識されればその関数は展開され、その他の関数は変更されません。
例:

```
#pragma inline /* what is the function name?*/
```

以下は正しい使用例です。

```
#pragma inline memcpy
```

(219) missing name after pragma “printf_check” (パーサー)
printf_check pragma の後に関数の名前が必要です。これは、その関数に対して printf 形式の書式文字列チェックを指定します。
例:

```
#pragma printf_check /* what function is to be checked?*/
```

以下は正しい使用例です。

```
#pragma printf_check sprintf
```

全ての標準 printf 形式関数向けのプラグマは既に<stdio.h>に含まれています。

(220) exponent expected (パーサー)
浮動小数点定数は、e または E の後に 1 桁以上の値を必要とします。
例:

```
float f;
f = 1.234e; /* oops -- what is the exponent?*/
```

(221) hexadecimal digit expected (パーサー)
0x の後に 1 桁以上の 16 進値(0〜9、A〜F (または a〜f))が必要です。
例:

```
a = 0xg6; /* oops -- was that meant to be a = 0xf6 ?*/
```

(222) binary digit expected (パーサー)
0b 書式指定子の後に 2 進値が必要です。
例:

```
i = 0bf000; /* oops -- f000 is not a base two value */
```

(223) digit out of range (Parser, Assembler)
この値のいずれかの桁が基数以上となっています(例: 8 進値に 8 以上の桁、10 進値に A～F の桁が含まれている)。8 進値は 0 で始まる数字列で示され、16 進値は 0X または 0x で始まります。
例:

```
int a = 058; /* leading 0 implies octal which has digits 0 - 7 */
```

(224) illegal “#” directive (パーサー)
無効な#プリプロセッサ ディレクティブが見つかりました。コード内のどこかにディレクティブのスペルミスがあるかもしれません。

(225) missing character in character constant (パーサー)
2 つのシングルクオートの間に文字がありません。
例:

```
char c = ''; /* the character value of what?*/
```

(226) char const too long (パーサー)
シングルクオートで囲まれた文字定数に複数の文字を含める事はできません。
例:

```
c = ’12’; /* oops -- only one character can be specified */
```

(227) “.” expected after “..”(パーサー)
連続した 2 個のピリオド(. .)が見つかりました。関数プロトタイプ内で可変数のパラメータを示すために使う ellipsis シンボル(3 個のピリオドによる省略記号)を意図しているのであれば、ピリオドをもう 1 個追加する必要があります。

あるいは、構造体メンバー演算子(1 個のピリオド)を意図しているのであれば、ピリオドを 1 個削除する必要があります。

(228) illegal character (*) (パーサー)
この文字は C コード内で使用禁止です。有効な文字には英文字、数字、演算子を構成する文字が含まれます。
例:

```
c = a; /* oops -- did you mean c = ’a’; ?*/
```

(229) unknown qualifier “*” given to -A (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(230) missing argument to -A (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(231) unknown qualifier “*” given to -I (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(232) missing argument to -I (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(233) bad -Q option “*” (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(234) close error (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

© 2022 Microchip Technology Inc. and its subsidiaries

(236) simple integer expression required (パーサー)
変数に絶対アドレスを割り当てるために使う__at()演算子の後には単純な整数式が必要です。
例:

```
int address;
char LOCK __at(address);
```

(237) function “*” redefined (パーサー)
このモジュール内で 1 つの関数に対して複数の定義が見つかりました。関数オーバーロードは禁止されています。
例:

```
int twice(int a)
{
    return a*2;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
    return a*2;
}
```

(238) illegal initialization (パーサー)
typedef 宣言は初期化可能な記憶域を予約しないため、この宣言を初期化する事はできません。
例:

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(239) identifier “*” redefined (from line *) (パーサー)
この識別子は同じスコープ内で既に定義されています。再定義はできません。
例:

```
int a; /* a filescope variable called "a" */
int a;  /* attempting to define another of the same name */
```

同一名の変数を異なるスコープで定義する事はできますが、推奨はしません。

(240) too many initializers (パーサー)
このオブジェクトに対して初期化子が多すぎます。オブジェクト(配列または構造体) の定義に対して初期化子の数が適正であるか確認してください。
例:

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(241) initialization syntax (パーサー)
このオブジェクトの初期化は構文的に正しくありません。波カッコとコンマの位置および数が正しいか確認してください。
例:

```
int iarray[10] = {{’a’, ’b’, ’c’};
/* oops -- one two many {s */
```

(242) illegal type for switch expression (パーサー)
switch()演算子には整数型または列挙値の式が必要です。
例:

```
double d;
switch(d) { /* oops -- this must be integral */
case ’1.0’:
    d = 0;
}
```

(243) inappropriate break/continue (パーサー)
適切な制御構造で囲まれていない break または continue 文が見つかりました。continue は while、for、do while ループ内でのみ使う事ができ、break はそれらのループ内または switch 文内でのみ使う事ができます。
例:

```
switch(input)
{ case 0:
    if(output == 0)
        input = 0xff;
    } /* oops! this should not be here; it closed the switch */
    break;     /* this should be inside the switch */
```

(244) “default” case redefined (パーサー)
switch()文内で複数の default ラベルが使われています。このラベルは 1 つしか使えません。
例:

```
switch(a) {
default:     /* if this is the default case...*/
    b = 9;
    break;
default:     /* then what is this?*/
    b = 10;
    break;
```

(245) “default” case not in switch (パーサー)
switch 文で囲まれていない default ラベルが見つかりました。default ラベルは、switch 文のボディ内でのみ使えます。

この default ラベルの前に switch 文が存在する場合、switch コード内の閉じカッコの数が多すぎるのかもしれません。閉じカッコによって default ラベルの前で switch 文が終了している可能性があります。メッセージ(246)を参照してください。

(246) case label not in switch (パーサー)
switch 文で囲まれていない case ラベルが見つかりました。case ラベルは、switch 文のボディ内でのみ使えます。

case ラベルの前に switch 文が存在する場合、switch コード内の閉じカッコの数が多すぎるのかもしれません。閉じカッコによって case ラベルの前で switch 文が終了している可能性があります。
例:

```
switch(input)
{ case ’0’:
    count++;
    break;
case ’1’:
    if(count>MAX)
        count= 0;
    }          /* oops -- this shouldn’t be here */
    break;
case ’2’:    /* error flagged here */
```

(247) duplicate label “*” (パーサー)
この関数内で複数の同名ラベルが使われています。これらのラベルのスコープは各ラベルを格納しているブロック内に限定されるのではなく、関数全体に及ぶという事に注意してください。
例:

```
start:
if(a > 256)
    goto end;
start:           /* error flagged here */
if(a == 0)
    goto start; /* which start label do I jump to?*/
```

(248) inappropriate “else” (パーサー)
if 文と対応しない else キーワードが見つかりました。カッコの欠落等による構文エラーの可能性があります。
例:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
    c = 0;
/* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
    c = 0xff;
```

(249) probable missing “}” in previous block (パーサー)
コンパイラは関数またはその他の宣言と思われる内容を見つけましたが、その前の関数が閉じ波カッコによって終了していません。直前の関数内のどこかで閉じカッコ(最後の閉じカッコであるとは限らない)が欠落しているかもしれません。
例:

```
void set(char a)
{
    PORTA = a;
    /* the closing brace was left out here */
void clear(void) /* error flagged here */
{
    PORTA = 0;
}
```

(251) array dimension redeclared (パーサー)
配列の次元が以前の宣言とは異なる非 0 値として再宣言されています。宣言済みの配列のサイズ(次元) は、0 として再宣言する事のみ可能です。
例:

```
extern int array[5];
int array[10];        /* oops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype (パーサー)
関数定義の引数が以前のプロトタイプと一致しません(メッセージに示される引数番号は左端の引数を 0 とします)。
例:

```
/* this is supposedly calc’s prototype */
extern int calc(int, int);
int calc(int a, long int b)  /* hmmm -- which is right?*/
{                            /* error flagged here */
    return sin(b/a);
}
```

(253) argument list conflicts with prototype (パーサー)
関数定義内の引数リストが以前のプロトタイプと一致しません。引数の数と型が全て一致するか確認してください。
例:

```
extern int calc(int);   /* this is supposedly calc’s prototype */
int calc(int a, int b) /* hmmm -- which is right? */
{                       /* error flagged here */
    return a + b;
}
```

(254) undefined *:“*” (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(255) not a member of the struct/union “*” (パーサー)
この識別子は指定された構造体または共用体型のメンバーではありません。
例:

```
struct {
    int a, b, c;
} data;
```

```
if(data.d)    /* oops -- there is no member d in this structure */
    return;
```

(256) too much indirection (パーサー)
ポインタ宣言は 16 レベルの間接参照しか持つ事ができません。

(257) only “register” storage class allowed (パーサー)
関数パラメータ向けに使える記憶域クラスは `register` のみです。
例:

```
void process(static int input)
```

(258) duplicate qualifier (パーサー)
この型指定内に同じ修飾子が 2 つ存在します。これは単純な誤りの他に `typedef` の使用を介して発生する可能性があります。余分な修飾子を削除してください。
例:

```
typedef volatile int vint;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) object can’t be qualified both far and near (パーサー)
1 つの型を `far` と `near` の両方で修飾する事は禁じられています。
例:

```
far near int spooky;  /* oops -- choose far or near, not both */
```

(260) undefined enum tag “*” (パーサー)
この `enum` タグは未定義です。
例:

```
enum WHAT what; /* a definition for WHAT was never seen */
```

(261) struct/union member “*” redefined (パーサー)
この `struct` または `union` 内で同じメンバー名が既に使われています。
例:

```
struct {
    int a;
    int b;
    int a; /* oops -- a different name is required here */
} input;
```

(262) struct/union “*” redefined (パーサー)
構造体または共用体が再定義されています。
例:

```
struct {
    int a;
} ms;
struct {
    int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members can’t be functions (パーサー)
関数を構造体または共用体のメンバーにする事はできません。関数に対するポインタならメンバーになれます。
例:

```
struct {
    int a;
    int get(int);  /* should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type (パーサー)
ビットフィールドの型は int（または unsigned) にしかできません。
例:

```
struct FREG {
    char b0:1;   /* these must be part of an int, not char */
    char   :6;
    char b7:1;
} freg;
```

(265) integer constant expected (パーサー)
構造体宣言内のメンバー名の後のコロンは、そのメンバーがビットフィールドである事を示します。コロンの後には、そのビットフィールドのビット数を定義する整数定数が必要です。
例:

```
struct {
    unsigned first:/* oops -- should be: unsigned first; */
    unsigned second;
} my_struct;
```

ビットフィールドを持つ構造体を宣言する場合の例を以下に示します。

```
struct {
    unsigned first :4; /* 4 bits wide */
    unsigned second:4; /* another 4 bits */
} my_struct;
```

(266) storage class illegal (パーサー)
構造体または共用体メンバーに記憶域クラスを与える事はできません。メンバーの記憶域クラスは、構造体の記憶域クラスによって決まります。
例:

```
struct {
    /* no additional qualifiers can be present with members */
    static int first;
} ;
```

(267) bad storage class (コード ジェネレータ)
記憶域クラスが無効な変数定義が見つかりました。
例:

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters cannot be static */
{
    return foo * a;
}
```

(268) inconsistent storage class (パーサー)
宣言内で記憶域クラスが競合しています。宣言内では記憶域クラスを 1 つだけ指定する必要があります。
例:

```
extern static int where; /* so is it static or extern?*/
```

(269) inconsistent type (パーサー)
宣言内では基本型を 1 つだけ指定できます。
例:

```
int float input; /* is it int or float?*/
```

(270) variable can’t have storage class “register” (パーサー)
関数パラメータと auto 変数だけが register 修飾子を使って宣言できます。
例:

```
register int gi;      /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
    return input + gi;
}
```

(271) type can't be long (パーサー)
int と double だけが long と一緒に修飾できます。
例:

```
long char lc; /* what?*/
```

(272) type can't be short (パーサー)
int だけが short と一緒に修飾できます。
例:

```
short float sf; /* what?*/
```

(273) type can't be both signed and unsigned (パーサー)
型修飾子 signed と unsigned の両方を同じ宣言内で使う事はできません(それらは互いに真逆の意味を持ちます)。
例:

```
signed unsigned int confused;  /* which is it?*/
```

(274) type can't be unsigned (パーサー)
浮動小数点型は unsigned にできません。
例:

```
unsigned float uf; /* what?*/
```

(275) "..." illegal in non-prototype argument list (パーサー)
ellipsis シンボル(省略記号) は、プロトタイプ引数リスト内の最後のアイテムとしてのみ使えます。このシンボルを単独で使う事はできません。また、型を持たない引数名(すなわち K&R 形式の非プロトタイプ関数定義) の後で使う事もできません。
例:

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
    int a, b;
{
```

(276) type specifier required for prototyped argument (パーサー)
プロトタイプ引数には型指定子が必要です。識別子のみは許容されません。

(277) can't mix prototyped and non-prototyped arguments (パーサー)
1 つの関数宣言内では、プロトタイプ形式の引数(丸カッコ内で型と一緒に指定)または K&R 形式の引数(丸カッコ内で名前のみ指定し、関数ボディ開始前の宣言リスト内で引数の型を指定) のどちらかだけが使えます(形式の異なる引数が 1 つの宣言内で混在してはいけません)。
例:

```
int plus(int a, b) /* oops -- a is prototyped, b is not */
int b;
{
    return a + b;
}
```

(278) argument "*" redeclared (パーサー)
この引数は同じ引数リスト内で複数回宣言されています。
例:

```
/* cannot have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal (パーサー)
宣言内で関数の引数に初期化子を持たせる事はできません。引数は、関数が呼び出された時に呼び出し元関数から与えられた値により初期化されます。
例:

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal (パーサー)
関数の配列を定義する事はできません。関数へのポインタの配列は定義可能です。
例:

```
int * farray[]();  /* oops -- should be: int (* farray[])(); */
```

(281) functions can’t return functions (パーサー)
関数が関数を返す事はできません。関数ポインタを返す事は可能です。関数ポインタを返す関数は `int (*(`*name*`())())()`として宣言できますが、多数のカッコが必要であるため注意が必要です。

(282) functions can’t return arrays (パーサー)
関数はスカラー(単純)型または構造体を返す事しかできず、配列を返す事はできません。

(283) dimension required (パーサー)
多次元配列では、最上位(最初)の次元のみ値を指定しなくても構いません。その後の全ての次元には定数式を指定する必要があります。
例:

```
/* This should be, for example: int arr[][7] */
int get_element(int arr[2][])
{
    return array[1][6];
}
```

(284) invalid dimension (パーサー)
指定された配列の次元は無効です。0 より大きな値を指定する必要があります。
例:

```
int array[0]; // oops -- you cannot have an array of size 0
```

(285) no identifier in declaration (パーサー)
宣言内に識別子がありません。このエラーは、閉じカッコの欠落等によってコンパイラが混乱した場合にも発生する可能性があります。
例:

```
void interrupt(void) /* what is the name of this function?*/
{
}
```

(286) declarator too complex (パーサー)
この宣言子は複雑すぎるためコンパイラで処理できません。宣言を単純化できないか検討してください。単純化する事でコードの管理も容易になります。

(287) arrays of bits or pointers to bit are illegal (パーサー)
`__bit` オブジェクトの配列または `bit` 変数に対するポインタは禁止されています。
例:

```
bit barray[10]; /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) the type 'void' is applicable only to functions (パーサー)
変数は `void` にできません。関数のみが `void` にできます。
例:

```
int a;
void b;  /* this makes no sense */
```

(289) the specifier 'interrupt' is applicable only to functions (パーサー)
`interrupt` 修飾子は関数以外に対して適用できません。
例:

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(290) illegal function qualifier(s) (パーサー)
関数に対して適用された修飾子は、この文脈では無意味です。一部の修飾子は、値(const または volatile) に対して使った場合にのみ意味を持ちます。アスタリスク「*」(その関数は修飾されたオブジェクトへのポインタを返すという事を示す)を書き忘れた可能性があります。
例:

```
const char ccrv(void) /* const * char ccrv(void) perhaps?*/
{                     /* error flagged here */
    return ccip;
}
```

(291) K&R identifier “*” not an argument (パーサー)
K&R 形式の引数宣言子内で使われているこの識別子は、関数名の後のカッコ内で指定されていません。
例:

```
int process(input)
int unput;        /* oops -- that should be int input; */
{
}
```

(292) a function is not a valid parameter type (パーサー)
関数を関数パラメータにする事はできません。関数に対するポインタであればパラメータにできます。宣言内で*が欠落している可能性があります。

(293) bad size in index_type() (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(294) can’t allocate * bytes of memory (コード ジェネレータ、Hexmate)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(295) expression too complex (パーサー)
この式によってコンパイラの内部スタックでオーバーフローが発生しました。式を整理するか 2 つに分割する必要があります。

(296) out of memory (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(297) bad argument (*) to tysize() (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(298) end of file in #asm (プリプロセッサ)
#asm ブロック内でファイルの終端(EOF) に遭遇しました。#endasm が欠落しているかスペルミスの可能性があります。
例:

```
#asm
MOV  r0, #55
MOV  [r1], r0
}               /* oops -- where is the #endasm */
```

(300) unexpected end of file (パーサー)
C モジュール内で予期せぬファイルの終端(EOF) に遭遇しました。
例:

```
int main(void)
{
    init();
    run();    /* is that it? What about the close brace */
```

(301) end of file on string file (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(302) can't reopen "*":* (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(303) can't allocate * bytes of memory (line *) (パーサー)
パーサーは文字列のソートとマージを試みましたが、最長の文字列にメモリを割り当てる事ができませんでした。このモジュール内の文字列の数または長さを削減してください。

(306) can't allocate * bytes of memory for * (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(307) too many qualifier names (パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(308) too many case labels in switch (コード ジェネレータ)
`switch` 文内の `case` ラベルの数が多すぎます。1 つの `switch` 文内で使える `case` ラベルの最大数は 511 です。

(309) too many symbols (アセンブラ、パーサー)
アセンブラのシンボルテーブルに対してシンボルの数が多すぎます。プログラム内のシンボルの数を減らしてください。

(310) "]" expected (パーサー)
配列宣言内または配列インデックスを使う式内で閉じ角カッコが欠落しています。
例:

```
process(carray[idx);  /* oops --
should be: process(carray[idx]); */
```

(311) 1) closing quote expected (パーサー)
指定された文字列の閉じ側のクオートが欠落しています。

(312) "*" expected (パーサー)
パーサーはこのトークンを期待しています。

(313) function body expected (パーサー)
関数宣言で K&R 形式の引数(カッコ内で引数名のみ指定、型指定なし)が使われた場合、その後に関数ボディが続く必要があります。
例:

```
/* the function block must follow, not a semicolon */ int get_value(a, b);
```

(314) ";" expected (パーサー)
命令文からセミコロンが欠落しています。セミコロンで終了していない命令文の後で閉じ波カッコまたはキーワードが見つかりました。
例:

```
while(a) {
    b = a--  /* oops -- where is the semicolon?*/
}          /* error is flagged here */
```

Note: セミコロンが欠落した命令文の直後が閉じ波カッコまたはキーワードではない場合、通常は後続のコードで他のエラーが発行されます(パーサーは、そのコードを前の命令文の続きだと見なすため)。

(315) "{" expected (パーサー)
開き波カッコがこの位置で必要です。このエラーは、開き波カッコが欠落した関数定義によって発生する可能性があります。
例:

```
/* oops! no opening brace after the prototype */
void process(char c)
```

```
    return max(c, 10) * 2; /* error flagged here */
}
```

(316) “}” expected (パーサー)
閉じ波カッコがこの位置で必要です。このエラーは、閉じ波カッコが欠落した初期化配列によって発生する可能性があります。
例:

```
char carray[4] = { 1, 2, 3, 4;  /* oops -- no closing brace */
```

(317) “(” expected (パーサー)
開き丸カッコ「(」がこの位置で必要です。この丸カッコは `while`、`for`、`if`、`do`、`asm` キーワードの直後のトークンである事が必要です。
例:

```
if a == b /* should be: if(a == b) */
b = 0;
```

(318) string expected (パーサー)
`asm` 文に対するオペランドは、丸カッコで囲まれた文字列である事が必要です。
例:

```
asm(nop); /* that should be asm("nop");
```

(319) while expected (パーサー)
`do` 文の最後にはキーワード `while` が必要です。
例:

```
do {
    func(i++);
}           /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
    end();
```

(320) “:” expected (パーサー)
`case` ラベルの後または `default` キーワードの後のコロンが欠落しています。これは、コロンの位置で誤ってセミコロンがタイプ入力された場合によく発生します。
例:

```
switch(input) {
case 0;         /* oops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected (パーサー)
`goto` の後にラベルを表す識別子が必要です。
例:

```
if(a)
      goto 20;
      /* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or “{” expected (パーサー)
キーワード `enum` の後に `enum` タグとして定義された(あるいは定義される予定の)識別子または開き波カッコが必要です。
例:

```
enum 1, 2;  /* should be, for example: enum {one=1, two }; */
```

(323) struct/union tag or “{” expected (パーサー)
`struct` または `union` キーワードの後に構造体あるいは共用体を表す識別子または開き波カッコが必要です。
例:

```
struct int a;  /* this is not how you define a structure */
```

以下は正しいコード例です。

```
struct {
    int a;
} my_struct;
```

(324) too many arguments for printf-style format string (パーサー)
書式文字列の引数が多すぎます。これは無害ですが、書式文字列が正しく表現されない可能性があります。
例:

```
/* oops -- missed a placeholder?*/
printf("%d - %d", low, high, median);
```

(325) error in printf-style format string (パーサー)
この位置の書式文字列内にエラーが存在します。文字列は printf()形式の書式文字列として解釈され、構文的に正しくありません。このため、実行時に予期せぬ挙動が生じる可能性があります。
例:

```
printf("%l", lll);  /* oops -- possibly: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string (パーサー)
この書式指定子には long 型の引数が必要です。書式指定子および対応する引数の数と順番を確認してください。
例:

```
printf("%lx", 2); // possibly you meant: printf("%lx", 2L);
```

(327) long long int argument required in printf-style format string (パーサー)
この書式指定子には long long 型の引数が必要です。書式指定子および対応する引数の数と順番を確認してください。
例:

```
printf("%llx", 2);  // possibly you meant: printf("%llx", 2LL);
```

(328) int argument required in printf-style format string (パーサー)
printf 形式の書式指定子には整数型の引数が必要です。書式指定子および対応する引数の数と順番を確認してください。
例:

```
printf("%d", 1.23); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string (パーサー)
この引数に対応する printf 書式指定子は%f (またはこれに類似)であり、浮動小数点型の式を必要とします。書式指定子または printf に対する引数に過不足がないか確認してください。
例:

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required in printf-style format string (パーサー)
この書式指定子にはポインタ引数が必要です。書式指定子および対応する引数の数と順番を確認してください。

(331) too few arguments for printf-style format string (パーサー)
この書式文字列に対する引数の数が少なすぎます。実行時に無意味な値が出力または変換される可能性があります。
例:

```
printf("%d - %d", low); /* oops! where is the other value to print?*/
```

© 2022 Microchip Technology Inc. and its subsidiaries

(332) “interrupt_level” should be 0 to 7 (パーサー)
pragma interrupt_level の引数のレンジは 0～7 ですが、ミッドレンジ デバイス向けにはレベル 1 のみが使え、PIC18 デバイス向けにはレベル 1 または 2 のみが使えます。
例:

```
#pragma interrupt_level 9 /* oops -- the level is too high */
void interrupt isr(void)
{
    /* isr code goes here */
}
```

(333) unrecognized qualifier name after “strings” (パーサー)
pragma strings に渡された修飾子が認識されませんでした。
例:

```
/* oops -- should that be #pragma strings const ?*/
#pragma strings cinst
```

(334) unrecognized qualifier name after “printf_check” (パーサー)
#pragma printf_check に渡された修飾子が認識されませんでした。
例:

```
/* oops -- should that be const not cinst?*/
#pragma printf_check(printf) cinst
```

(335) unknown pragma “*” (パーサー)
不明の pragma ディレクティブに遭遇しました。
例:

```
#pragma rugsused myFunc w  /* I think you meant regsused */
```

(336) string concatenation across lines (パーサー)
2 行にまたがって書かれた文字列は連結されます。これが期待した結果であるかどうか確認してください。
例:

```
char * cp = "hi"
"there";    /* this is okay, but is it what you had intended? */
```

(337) line does not have a newline on the end (パーサー)
ファイルの最終行の行末に改行文字(オペレーティング システムに依存)がありません。一部のエディタはこのようなファイルを生成し、インクルード ファイルで問題が生じる場合があります。C 規格は、全てのソースファイルが完全な行のみで構成される事を要求します。

(338) can’t create * file “*” (全アプリケーション)
アプリケーションは指定されたファイルを作成しようと(または開こうと) 試みましたが、失敗しました。ファイルの名前とパスが正しいか確認してください。

(339) initializer in extern declaration (パーサー)
キーワード extern を含む宣言内で初期化子が使われています。オブジェクトを初期化するにはそれを定義する(すなわち、それに記憶域を割り当てる)必要があるため、extern 記憶域クラスは上書きされます。
例:

```
extern int other = 99; /* if it’s extern and not allocated
storage, how can it be initialized?*/
```

(340) string not terminated by null character (パーサー)
char 型配列が、配列より大きな文字列リテラルで初期化されています。従って、この配列には Null 終端文字を追加するため空きがありません。
例:

```
char foo[5] = "12345"; /* the string stored in foo won’t have
a null terminating, i.e.
foo = [’1’, ’2’, ’3’, ’4’, ’5’] */
```

(343) implicit return at end of non-void function (パーサー)
関数が値を返すよう宣言されていますが、その実行パスは return 文を経ずに関数ボディの終端に達する(従って関数は値なしでリターンする)可能性があります。値を指定した return 文を挿入するか、関数が値を返す必要がない場合はその関数を void として宣言してください。
例:

```
int mydiv(double a, int b)
{
    if(b != 0)
        return a/b;   /* what about when b is 0? */
}                 /* warning flagged here */
```

(344) non-void function returns no value (パーサー)
関数は値を返すよう宣言されていますが、return 文で戻り値が指定されていません。
例:

```
int get_value(void)
{
    if(flag)
        return val++;
    return;
    /* what is the return value in this instance?*/
}
```

(345) unreachable code (パーサー)
コードのこのセクションは決して実行されません(そこに到達可能な実行パスが存在しません)。
例:

```
while(1)          /* how does this loop finish? */
    process();
flag = FINISHED; /* how do we get here?*/
```

(346) declaration of “*” hides outer declaration (パーサー)
外部宣言と同じ名前を持つオブジェクトが宣言されています(この関数またはブロックより前に外部で宣言済みです)。これはエラーではありませんが、外部変数は意図した通りに使われなくなる可能性があります。
例:

```
int input;         /* input has filescope */
void process(int a)
{
    int input;       /* local blockscope input */
    a = input;       /* this will use the local variable. Is this right?*/
```

(347) external declaration inside function (パーサー)
関数が extern 宣言を含んでいます。これはエラーではありませんが、関数宣言のスコープがその関数ボディ内に制限されるため、常に望ましくありません。この後にコンパイラが同じファイル内でその extern オブジェクトの別の宣言、使用、定義のいずれかに遭遇した場合、そのオブジェクトは以前の宣言をもう持っていないため、コンパイラはその宣言が矛盾しないかどうか確認できません。これにより、プログラムの異常な挙動またはリンク時のシグネチャ エラーが生じる可能性があります。また、これは同一オブジェクトに対する以前の宣言を隠蔽し、やはりコンパイラの型チェックを妨げます。一般的な規則として、extern 変数および関数は他の全ての関数の外部で宣言します。
例:

```
int process(int a)
{
    /* this would be better outside the function */
    extern int away;
    return away + a;
}
```

(348) auto variable “*” should not be qualified (パーサー)
auto 変数を near または far 等で修飾してはいけません。auto 変数の記憶域クラスは、スタックの構成によって暗黙的に定義されます。auto 変数を static で修飾する事はできますが、その変数はもはや auto ではなくなります。

(349) non-prototyped function declaration for “*” (パーサー)
関数が旧式(K&R 形式)の引数を使って宣言されています。全ての関数に対してプロトタイプ宣言を使う事を推奨します。
例:

```
int process(input)
int input;         /* warning flagged here */
{
}
```

上記のコードは、以下の通りに書き換える事を推奨します。

```
int process(int input)
{
}
```

(350) unused * “*” (from line *) (パーサー)
このオブジェクトは、コンパイル中の関数またはモジュール内で全く使われていません。このオブジェクトは冗長であるか、このオブジェクトを使うはずであったコードがコンパイルから除外されたか、オブジェクト名のスペルミスが原因として考えられます。rcsid および sccsid シンボルは未使用であっても何も報告されないという事に注意が必要です。

(352) float parameter coerced to double (パーサー)
非プロトタイプ関数のパラメータが float として宣言されていますが、コンパイラはこれを double に変換しました。これは C 言語の既定値型変換規則「非プロトタイプ関数に浮動小数点値が渡された場合は double に変換する」に従うためです。関数の宣言はこの規則に従う必要があります。
例:

```
double inc_flt(f) /* f will be converted to double */
float f;          /* warning flagged here */
{
    return f * 2;
}
```

(353) sizeof external array “*” is zero (パーサー)
外部配列のサイズが 0 であると評価されました。外部宣言内でその配列の次元が明示的に指定されていない可能性があります。

(354) possible pointer truncation (パーサー)
far 修飾されたポインタが既定値ポインタまたは near 修飾されたポインタに割り当てられたか、既定値ポインタが near 修飾されたポインタに割り当てられています。これにより、ポインタサイズが切り詰められて情報が失われる可能性があります(使用するメモリモデルに依存)。

(355) implicit signed to unsigned conversion (パーサー)
符号付き値がより大きな符号なし型に代入または変換されました。ANSI C の「値の保持」規則に従い、符号付き値は最初に符号付きのままターゲット型のサイズに変換され、その後に符号なし値に変換されます(その結果、ビットパターンは変更されません)。これにより、予期せぬ符号拡張が発生する可能性があります。これを防ぐには、最初に符号付き値を等価な符号なし値に変換する事が必要です。
例:

```
signed char sc;
unsigned int ui;
ui = sc;    /* if sc contains 0xff, ui will contain 0xffff for example */
```

上記は、char 変数をより大きな型へと符号拡張します。符号拡張を望まない場合、型変換を使います。

例:

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer (パーサー)
浮動小数点値が整数型に代入または変換されました。これにより、浮動小数点値が切り詰められる可能性があります。型変換を使う事でこの警告を回避できます。
例:

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

下記の型変換を使う事で、上記の式が持つ意図を明確に示す事ができます。

```
i = (int)dd;
```

(357) illegal conversion of integer to pointer (パーサー)
整数がポインタ型に代入または変換されました。通常これは、誤った変数が使われた事を意味します。しかし、これが真に意図した動作である場合、型変換(キャスト)を使う事で、この変換が必要である事をコンパイラに知らせ、警告を回避する事ができます。この警告は、&アドレス演算子の欠落が原因で生成される場合もあります。
例:

```
int * ip;
int i;
ip = i;    /* oops -- did you mean ip = &i ? */
```

下記の型変換を使う事で、上記の式が持つ意図を明確に示す事ができます。

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer (パーサー)
ポインタが整数型に代入または変換されました。通常これは、誤った変数が使われた事を意味します。しかし、これが真に意図した動作である場合、型変換を使う事で、この変換が必要である事をコンパイラに知らせ、警告を回避する事ができます。この警告は、* 間接参照演算子の欠落が原因で生成される場合もあります。
例:

```
int * ip;
int i;
i = ip;    /* oops -- did you mean i = *ip ? */
```

下記の型変換を使う事で、意図を明確に示す事ができます。

```
i = (int)ip;
```

(359) illegal conversion between pointer types (パーサー)
特定型のポインタ(特定種類のオブジェクトを指すポインタ)が異なる型のポインタに変換されました。通常これは、誤った変数が使われた事を意味します。しかし、これが真に意図した動作である場合、型変換を使う事で、この変換が必要である事をコンパイラに知らせ、警告を回避する事ができます。
例:

```
long input;
char * cp;
cp = &input; /* is this correct?*/
```

これは、マルチバイト変数内のバイトにアクセスするためによく使われる方法です。これが意図した動作である事を示すために、下記の型変換が使えます。

```
cp = (char *)&input; /* that's better */
```

同じ型のオブジェクトを指すポインタ同士の間の変換で修飾子が互いに異なっている場合にも、この警告が生成される可能性があります。

例:

```
char * cp;
/* yes, but what sort of characters?*/
cp = "I am a string of characters";
```

文字列リテラルの既定値型が const char *である場合、この警告は非常に重要です。これは、以下の通りに書く必要があります。

例:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

ポインタ型から修飾子を省略すると、しばしば致命的エラーが発生します。

(360) array index out of bounds (パーサー)

配列が 0 未満の定数または配列内の要素数以上の定数によってインデックスされています。ポインタ変数を介して配列要素にアクセスする事で、この警告は回避できます。

例:

```
int i, * ip, input[10];
i = input[-2];           /* oops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];              /* this is okay */
```

(361) function declared implicit int (パーサー)

名前が未定義の関数が呼び出された場合、コンパイラは引数未指定(K&R 形式) のパラメータを使って自動的にその関数を整数型として宣言します。その後にその関数が定義され、関数の型と引数が先の暗黙的な宣言と異なっていた場合、コンパイラエラーが発生します。これを防ぐには、全ての関数をそれらが使われる前に定義します(あるいは、少なくとも宣言します)。また、可能であればプロトタイプ パラメータを使う事を推奨します。関数を使用前に宣言する場合、宣言の前に適切なキーワード(extern または static)が必要です。

例:

```
/* I can prevent an error arising from calls below */
extern void set(long a, int b);
int main(void)
{
    /* at this point, a prototype for set() has already been seen */
    set(10L, 6);
}
```

(362) redundant "&" applied to array (パーサー)

アドレス演算子&が配列に対して適用されています。配列の名前でアドレスは取得されるため、&は不要であり無視されます。

例:

```
int array[5];
int * ip;
/* array is a constant, not a variable; the & is redundant.*/
ip = &array;
```

(363) redundant "&" or "*" applied to function address (パーサー)

アドレス演算子&が関数に対して適用されています。関数の名前でアドレスは取得されるため、&は不要であり無視されます。

例:

```
extern void foo(void);
int main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
    bar = foo; /* the & is redundant */
}
```

(364) attempt to modify object qualified * (パーサー)

const または code 宣言されたオブジェクトに対してプログラムが代入または変更を実行する事は一切できません。そのようなオブジェクトの変更を試みられた場合の影響は、コンパイラごとに異なります。

例:

```
const int out = 1234; /* "out" is read only */
out = 0;              /* oops -- writing to a read-only object */
```

(365) pointer to non-static object returned (パーサー)
関数が auto 変数等の非 static 変数へのポインタを返しています。auto 変数に割り当てられた記憶域は関数が戻った時点で無効となるため、これはエラーとなる可能性があります。
例:

```
char * get_addr(void)
{
    char c;
    /* returning this is dangerous; the pointer could be dereferenced */
    return &c;
}
```

(366) operands of “*” not same pointer type (パーサー)
この演算子のオペランドは異なるポインタ型です。誤ったポインタが使われている可能性があります。しかし、これが真に意図した動作である場合、型変換を使う事でこのエラーメッセージを回避できます。

(367) identifier is already extern; can’t be static (パーサー)
この関数は既に(おそらく暗黙的に) extern として宣言されており、static としての再宣言は無効です。
例:

```
int main(void)
{
    /* at this point the compiler assumes set is extern...*/
    set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
    PORTA = a + b;
}
```

(368) array dimension on “*[]” ignored (プリプロセッサ)
関数パラメータで指定された配列次元は無視されました。この引数は渡される時にポインタへと変換されるため、任意サイズの配列を渡す事ができます。パラメータから次元を削除するか、ポインタ構文を使ってパラメータを定義する事により、このメッセージを回避できます。
例:

```
/* param should be:"int array[]" or "int *" */
int get_first(int array[10])
{                           /* warning flagged here */
    return array[0];
}
```

(369) signed bitfields not supported (パーサー)
符号なしのビットフィールドのみがサポートされます。ビットフィールドが int 型として宣言された場合、コンパイラはそれを符号なしとして扱います。
例:

```
struct {
    signed int sign: 1;    /* oops -- this must be unsigned */
    signed int value: 7;
} ;
```

(370) illegal basic type; int assumed (パーサー)
基本型から修飾付き基本型への型変換は認識されず、その基本型は int 型であると見なされました。
例:

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) ’a’;
```

(371) missing basic type; int assumed (パーサー)
この宣言には基本型が含まれないため、int 型であると見なされました。この宣言はエラーではありませんが、意図を明確に示すために基本型を含める事を推奨します。
例:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) "," expected (パーサー)
この位置でコンマが必要です。宣言リスト内の 2 つの識別子の間にコンマが脱落している可能性があります。あるいは、直前の型名がスペルミスによって識別子として解釈された可能性があります。
例:

```
unsigned char a;
/* thinks: chat & b are unsigned, but where is the comma?*/
unsigned chat b;
```

(373) implicit signed to unsigned conversion (パーサー)
signed 型が指定されましたが、暗黙的に unsigned 型へ変換されました。
例:

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed (パーサー)
基本型から修飾付き基本型への型変換が指定されておらず、int 型であると見なされました。
例:

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(375) unknown FNREC type "*" (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(376) bad non-zero node in call graph (リンカ)
コールグラフ内の最上層のノードが下層から参照されています。プログラム内に間接的な再帰呼び出しが存在する可能性があります。コンパイルド スタックが使われる場合、そのような呼び出しは許容されません。

(378) can't create * file "*" (Hexmate)
このタイプのファイルは生成できませんでした。そのファイル(またはこの名前により指定されるファイル)が既に使用中ではないか確認してください。

(379) bad record type "*" (リンカ)
これは内部コンパイラエラーです。オブジェクト ファイルが有効かどうか確認してください。Microchip 社技術サポートに詳細をお問い合わせください。

(380) unknown record type (*) (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(381) record "*" too long (*) (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(382) incomplete record: type = *, length = * (Dump、Xstrip)
このメッセージは dump または xstrip ユーティリティにより生成され、オブジェクト ファイルが無効であるか切り詰められている事を示します。Microchip 社技術サポートに詳細をお問い合わせください。

(383) text record has length (*) too small (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(384) assertion failed: file *, line *, expression * (リンカ、パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(387) illegal or too many -G options (リンカ)
複数のリンカ-g オプションが存在するか、-g オプションの後で引数が何も指定されていません。セグメント アドレスの計算方法を指定するために引数が必要です。

(388) duplicate -M option (リンカ)
リンカに対してマップファイルの名前が 2 度指定されました。コンパイラ ドライバを使っている場合、この問題は発生しません。リンカを手動で呼び出す場合、コマンドライン上でこのオプションを 1 度だけ指定してください。

(389) illegal or too many -O options (リンカ)
このリンカ-o フラグは無効であるか、別の-o オプションが指定されています。リンカに対する-o オプションの直後に(スペースを挟まずに)ファイル名を指定する必要があります。

(390) missing argument to -P (リンカ)
リンカに渡された-p オプションの数が多すぎるか、-p オプションの後で引数が何も指定されていません。1 つの-p オプションに対して複数の引数をコンマで区切って指定する事で、-p オプションの数を削減する事ができます。

(391) missing argument to -Q (リンカ)
-Q リンカオプションには、デバイスのタイプを示す引数が必要です。

(392) missing argument to -U (リンカ)
-U(未定義化)オプションには 1 つの引数が必要です。

(393) missing argument to -W (リンカ)
-W オプション (リスト幅) には数値の引数が必要です。

(394) duplicate -D or -H option (リンカ)
リンカに対してシンボルファイル名が 2 度指定されました。コンパイラ ドライバを使っている場合、この問題は発生しません。リンカを手動で起動する場合、コマンドライン上でこれらのオプションのどちらかを 1 度だけ指定してください。

(395) missing argument to -J (リンカ)
-j リンカオプションには引数(処理を中止させる最大エラー数を指定) が必要です。

(397) usage: hlink [-options] files.obj files.lib (リンカ)
コマンドライン リンカが不適切な方法で使われました。リンカを直接実行していないのにこのメッセージが表示される場合、これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(398) output file can't be also an input file (リンカ)
リンカの入力ファイルの 1 つ対してリンカ出力の書き込みが試みられました。入力ファイルからの読み出しと出力ファイルへの書き込みを同時に実行する事はできません。

(400) bad object code format (リンカ)
これは内部コンパイラエラーです。オブジェクト ファイルのオブジェクト コード フォーマットが無効です。そのファイルが有効なオブジェクト ファイルかどうか確認してください。Microchip 社技術サポートに詳細をお問い合わせください。

(402) bad argument to -F (Objtohex)
Objtohex に対する-F オプションの引数が無効です。このツールを直接呼び出していないのにこのメッセージが表示される場合、これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(403) bad -E option:“*” (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(404) bad maximum length value to -<digits> (Objtohex)
OBJTOHEX *-n,m* (HEX 長さ/丸め処理) オプションの *n* の値が無効です。

(405) bad record size rounding value to -<digits> (Objtohex)
OBJTOHEX *-n,m* (HEX 長さ/丸め処理) オプションの *m* の値が無効です。

(406) bad argument to -A (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(407) bad argument to -U (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(408) bad argument to -B (Objtohex)
このオプションには整数型引数(基数は 8、10、16 のいずれか) が必要です。このツールを直接呼び出していないのにこのメッセージが表示される場合、これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(409) bad argument to -P (Objtohex)
このオプションには整数型引数(基数は 8、10、16 のいずれか) が必要です。このツールを直接呼び出していないのにこのメッセージが表示される場合、これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(410) bad combination of options (Objtohex)
OBJTOHEX に対して指定されたオプションの組み合わせが無効です。

(412) text does not start at 0 (Objtohex)
コードは 0 から開始する必要があります。ここでは、そうなっていません。

(413) write error on “*” (アセンブラ、リンカ、Cromwell)
このファイルで書き込みエラーが発生しました。ディスク容量が不足している可能性があります。

(414) read error on “*” (リンカ)
このファイルの読み出し中にリンカがエラーに遭遇しました。

(415) text offset too low in COFF file (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(416) bad character (*) in extended TEKHEX line (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(417) seek error in “*” (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(418) image too big (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(419) object file is not absolute (Objtohex)
OBJTOHEX に渡されたオブジェクト ファイルは再配置可能アイテムを含んでいます。これは、オブジェクト ファイルが正しくないか、リンカまたは OBJTOHEX に渡されたオプションが無効である可能性を示しています。アセンブラからのオブジェクト出力ファイルは再配置可能です(絶対アドレス化されていません)。リンカのオブジェクト ファイル出力は絶対アドレス化されています。

(420) too many relocation items (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(421) too many segments (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(422) no end record (リンカ)
このオブジェクト ファイルには終了レコードがありません。このファイルはオブジェクト ファイルではない可能性があります。コンパイラによって生成されたオブジェクト ファイルでこの問題が発生する場合、Microchip 社の技術サポートにお問い合わせください。

(423) illegal record type (リンカ)
オブジェクト ファイル内にエラーが存在します。オブジェクト ファイルに問題がない場合、リンカで内部エラーが発生した可能性があります。コンパイラによって生成されたオブジェクト ファイルでこの問題が発生する場合、Microchip 社の技術サポートにお問い合わせください。

(424) record too long (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(425) incomplete record (Objtohex, Libr)
OBJTOHEX またはライブラリアンに渡されたオブジェクト ファイルは破損しています。Microchip 社技術サポートに詳細をお問い合わせください。

(427) syntax error in list (Objtohex)
OBJTOHEX が読み出したリスト内に構文エラーが存在します。このリストは、オプションに対する応答として標準入力から読み出されます。

(428) too many segment fixups (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(429) bad segment fixups (Objtohex)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(430) bad specification (Objtohex)
OBJTOHEX に提供されたリストは構文的に正しくありません。

(431) bad argument to -E (Objtoexe)
このオプションには整数型引数(基数は 8、10、16 のいずれか)が必要です。`objtoexe` を直接呼び出している場合、この引数を確認してください。そうではない場合、これは内部コンパイラエラーであると考えられるため、Microchip 社技術サポートに詳細をお問い合わせください。

(432) usage: objtohex [-ssymfile] [object-file [exe-file]] (Objtohex)
コマンドライン ツール `objtohex` が正しく使われていません。このツールを直接呼び出していないのにこのメッセージが表示される場合、これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(434) too many symbols (*) (リンカ)
シンボルテーブル内のシンボルの数が多すぎます(許容最大数はメッセージ内の(*)により示されます)。一部のグローバル シンボルをローカル シンボルに変更する事でシンボルの数を減らしてください。

(435) bad segment selector “*” (リンカ)
リンカに対するセグメント指定オプション(`-G`)が無効です。
例:

```
-GA/f0+10
```

上記では基数の指定が抜けています。下記は訂正例です。

```
-GA/f0h+10
```

(436) psect “*” re-orged (リンカ)
この psect の開始アドレスが複数回指定されています。

(437) missing “=” in class spec (リンカ)
クラス指定には等号(=)が必要です(例: `-Ctext=ROM`)。

(438) bad size in -S option (リンカ)
`-S` オプションで指定されたアドレスは無効です。これは有効な値(基数は 8、10、16 のいずれか)である事が必要です。基数は値の後の `O` (8 進)または `H` (16 進)により指定します。16 進値は、値の前の `0x` により指定する事もできます。値と基数の大文字/小文字は区別されません。10 進値が既定値です。
例:

```
-SCODE=f000
```

上記では基数の指定が抜けています。下記は訂正例です。

```
-SCODE=f000h
```

(439) bad -D spec:“*” (リンカ)
処理指定`-D` (クラスに対して `delta` 値を指定) の書式が無効です。
例:

```
-DCODE
```

上記では `CODE` クラスに対して `delta` 値が指定されていません。下記は `CODE` クラスの `delta` 値を 2 に設定します。

```
-DCODE=2
```

(440) bad delta value in -D spec (リンカ)
処理指定`-D` に対して指定された `delta` 値が無効です。この値は整数(基数は 8、10、16 のいずれか)である事が必要です。

(441) bad -A spec:“*” (リンカ)
処理指定`-A` (リンカに対してアドレスレンジを指定) の書式が無効です。
例:

```
-ACODE
```

上記では `CODE` クラスに対してアドレスレンジが指定されていません。下記は `CODE` クラスに対してアドレスレンジ `0h`〜`1fffh` を指定します。

```
-ACODE=0h-1fffh
```

(442) missing address in -A spec (リンカ)
処理指定`-A` (リンカに対してアドレスレンジを指定)の書式が無効です。
例:

```
-ACODE=
```

上記では `CODE` クラスに対してアドレスレンジが指定されていません。下記は `CODE` クラスに対してアドレスレンジ `0h`〜`1fffh` を指定します。

```
-ACODE=0h-1fffh
```

(443) bad low address “*” in -A spec (リンカ)
処理指定`-A` で指定された下限アドレスは無効です。これは有効な値(基数は 8、10、16 のいずれか)である事が必要です。基数は値の後の `O` (8 進)または `H` (16 進)により指定します。16 進値は、値の前の `0x` により指定する事もできます。値と基数の大文字/小文字は区別されません。10 進値が既定値です。
例:

```
-ACODE=1fff-3fffh
```

上記では下限アドレス(`1fff`)に対して基数が指定されていません。下記は訂正例です。

```
-ACODE=1fffh-3fffh
```

(444) expected “-” in -A spec (リンカ)
-A リンカ オプション内の下限アドレスと上限アドレスの間に「-」がありません。
例:

```
-AROM=1000h
```

下記は正しい例です。

```
-AROM=1000h-1fffh
```

(445) bad high address “*” in -A spec (リンカ)
処理指定-A で指定された上限アドレスは無効です。これは有効な値(基数は 8、10、16 のいずれか)である事が必要です。基数は値の後の O (8 進)または H (16 進)により指定します。16 進値は、値の前の 0x により指定する事もできます。値と基数の大文字/小文字は区別されません。10 進値が既定値です。
例:

```
-ACODE=0h-ffff
```

上記では上限アドレス(ffff)に対して基数が指定されていません。下記は訂正例です。

```
-ACODE=0h-ffffh
```

(446) bad overrun address “*” in -A spec (リンカ)
処理指定-A で指定されたオーバーラン アドレスは無効です。これは有効な値(10 進、8 進、16 進のいずれか)である事が必要です。基数は値の後の O (8 進)または H (16 進)により指定します。16 進値は、値の前の 0x により指定する事もできます。値と基数の大文字/小文字は区別されません。10 進値が既定値です。
例:

```
-AENTRY=0-0FFh-1FF
```

上記ではオーバーラン アドレス(1FF)に対して基数が指定されていません。下記は訂正例です。

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address “*” in -A spec (リンカ)
処理指定-A で指定されたロードアドレスは無効です。これは有効な値(10 進、8 進、16 進のいずれか)である事が必要です。基数は値の後の O (8 進)または H (16 進)により指定します。16 進値は、値の前の 0x により指定する事もできます。値と基数の大文字/小文字は区別されません。10 進値が既定値です。
例:

```
-ACODE=0h-3fffh/a000
```

上記ではロードアドレス(000)に対して基数が指定されていません。下記は訂正例です。

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count “*” in -A spec (リンカ)
処理指定-A で指定された繰り返し数は無効です。
例:

```
-AENTRY=0-0FFhxf
```

上記では基数が指定されていません。下記は訂正例です。

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec:* (リンカ)
処理指定-A が無効です。処理指定-A の正しいコード例を以下に示します。

```
-AROM=1000h-1FFFh
```

(450) psect “*” was never defined, or is local (リンカ)
-P オプション内で指定されたこの psect は、プログラム内のどのモジュールでも定義されていないか、local psect フラグを使って定義されていても class フラグが指定されなかったためにアドレスにリンクできません。アセンブリ リストファイルで、この psect が存在する事および local フラグが指定されていない事を確認してください。

(451) bad psect origin format in -P option (リンカ)
-p オプション内の origin フォーマットが有効な値(10 進、8 進、16 進のいずれか)ではないか、既存 psect の名前ではありません。16 進値の末尾には H が必要です。
例:

```
-pbss=f000
```

上記では基数が指定されていません。下記は正しい例です。

```
-pbss=f000h
```

(452) bad “+” (minimum address) format in -P option (リンカ)
リンカの-p オプション内の最小アドレス値の書式が正しくありません。
例:

```
-pbss=data+f000
```

上記では基数が指定されていません。下記は正しい例です。

```
-pbss=data+f000h
```

(453) missing number after “%” in -P option (リンカ)
-p オプション内の%演算子(丸め処理境界)の後に値が必要です。

(454) link and load address can’t both be set to “.” in -P option (リンカ)
psect のリンクアドレスとロードアドレスの両方がピリオド「.」で指定されています。この方法で指定できるのはどちらか 1 つのアドレスだけです。
例:

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

上記のオプションはどちらも有効かつ等価です。しかし、下記の用法は不明です。

```
-Pmypsect=./.
```

(455) psect “*” not relocated on 0x* byte boundary (リンカ)
この psect は要求された境界上に再配置されません。この psect が再配置可能かどうかと、-p オプションが正しいかどうかを確認してください。

(456) psect “*” not loaded on 0x* boundary (リンカ)
-p オプションで指定されたロードアドレスは、この psect の再配置要件に適合しません。例えば、psect を 4K バイト境界に整列させる必要がある場合、その psect を 100H で開始する事はできません。

(459) remove failed; error:*, * (Xstrip)
中間生成ファイルの削除時に出力ファイルの生成に失敗しました。

(460) rename failed; error:*, * (Xstrip)
中間生成ファイルの名前変更時に出力ファイルの生成に失敗しました。

(461) can’t create * file “*” (Assembler, Code Generator)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(464) missing key in avmap file (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(465) undefined symbol “*” in FNBREAK record (リンカ)
再入不可関数向けの FNBREAK レコード内で未定義のシンボルが見つかりました。これが手書きアセンブリコードではない場合、Microchip 社の技術サポートにお問い合わせください。

(466) undefined symbol “*” in FNINDIR record (リンカ)
再入不可関数向けの FNINDIR レコード内で未定義のシンボルが見つかりました。これが手書きアセンブリコードではない場合、Microchip 社の技術サポートにお問い合わせください。

(467) undefined symbol “*” in FNADDR record (リンカ)
再入不可関数向けの FNADDR レコード内で未定義のシンボルが見つかりました。これが手書きアセンブリコードではない場合、Microchip 社の技術サポートにお問い合わせください。

(468) undefined symbol “*” in FNCALL record (リンカ)
再入不可関数向けの FNCALL レコード内で未定義のシンボルが見つかりました。これが手書きアセンブリコードではない場合、Microchip 社の技術サポートにお問い合わせください。

(469) undefined symbol “*” in FNROOT record (リンカ)
再入不可関数向けの FNROOT レコード内で未定義のシンボルが見つかりました。これが手書きアセンブリコードではない場合、Microchip 社の技術サポートにお問い合わせください。

(470) undefined symbol “*” in FNSIZE record (リンカ)
再入不可関数向けの FNSIZE レコード内で未定義のシンボルが見つかりました。これが手書きのアセンブリコードではない場合、Microchip 社の技術サポートにお問い合わせください。

(471) recursive function calls:(リンカ)
これらの関数は互いに再帰的に呼び出されています。これらの関数の一部は静的に配置されたローカル変数(コンパイルド スタック)を使います。reentrant キーワードを使うか(コンパイラがこれをサポートする場合)、コードを変更する事で再帰呼び出しを防ぐ必要があります。
例:

```
int test(int a)
{
    if(a == 5) {
    /* recursion cannot be supported by some compilers */
        return test(a++);
     }
    return 0;
}
```

(472) non-reentrant function “*” appears in multiple call graphs: rooted at “*” and “*” (リンカ)
この関数は、メインライン コードと割り込みコードの両方から呼び出される可能性があります。reentrant キーワードを使うか(コンパイラがこれをサポートする場合)、コードを変更する事でローカル変数またはパラメータの使用を防ぐか、関数を複製する必要があります。
例:

```
void interrupt my_isr(void)
{
    scan(6);     /* scan is called from an interrupt function */
}
void process(int a)
{
    scan(a);     /* scan is also called from main-line code */
}
```

(473) function “*” is not called from specified interrupt_level (リンカ)
この関数は、同じ割り込みレベルの割り込み関数から決して呼び出されません。
例:

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```

(474) no psect specified for function variable/argument allocation (リンカ)
リンカに対して auto/パラメータ ブロックに関する情報を指定する FNCONF アセンブラ ディレクティブが見つかりません。これは必要に応じて標準実行時ファイル内で提供されます。このエラーは、正しい実行時起動モジュールがリンクされなかった可能性がある事を示します。実行時起動モジュールが手書きである場合、FNCONF ディレクティブが使われている事を確認してください。

(475) conflicting FNCONF records (リンカ)
リンカは 2 つの競合する FNCONF ディレクティブに遭遇しました。このディレクティブは 1 度だけ指定される必要があり、通常全てのプログラムにリンクされる標準実行時起動コードに含まれています。

(476) fixup overflow referencing * * (location 0x* (0x*+*), size *, value 0x*) (リンカ)
リンカに対してアイテムの再配置(フィックスアップ)が要求されましたが、このアイテムは再配置後にメモリ空間に正しく格納できない可能性があります。詳細はエラーメッセージ(1356) の説明を参照してください。

(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) (リンカ)
リンカに対してアイテムの再配置(フィックスアップ)が要求されましたが、このアイテムは再配置後にメモリ空間に正しく格納できない可能性があります。詳細はエラーメッセージ(1356) の説明を参照してください。

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(479) circular indirect definition of symbol “*” (リンカ)
このシンボルは外部シンボルを参照し、参照シンボルがこのシンボルに対する循環参照になっています。

(480) function signatures do not match:* (*):0x*/0x* (リンカ)
指定された関数のシグネチャが別のモジュール内でのシグネチャと一致しません。すなわち、この関数の定義はモジュール間で異なります。例えば、一方のモジュールでのみプロトタイプ宣言されている可能性があります。各モジュール内でその関数がどのように宣言されているか調べ、それらが矛盾していないか確認してください。
例:

```
extern int get_value(int in);
/* and in another module:*/
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol “*” psect conflict (リンカ)
複数の psect 内で共通シンボルが定義されています。

(482) symbol “*” is defined more than once in “*” (アセンブラ)
このシンボルは複数回定義されています。シンボルが同じモジュール内で複数回定義されている場合、このエラーはアセンブラによって発行されます。
例:

```
_next:
    MOVE r0, #55
```

```
    MOVE [r1], r0
_next:          ; oops -- choose a different name
```

シンボル(C またはアセンブラ) が異なるモジュール内で複数回定義されている場合、この警告はリンカによって発行されます。エラーメッセージにはモジュールの名前が示されます。コンパイル後は、しばしば C 識別子名の前に 1 つのアンダースコアが追加される事に注意してください。

(483) symbol “*” can’t be global (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(484) psect “*” can’t be in classes “*” and “*” (リンカ)
1 つの psect を複数のクラスに割り当てる事はできません。このエラーは、アセンブラ モジュール内で PSECT ディレクティブに対する class=オプションが競合しているか、リンカに対して-C オプションが使われた事により発生します。
例:

```
psect final,class=CODE
finish:
    /* elsewhere:*/
psect final,class=ENTRY
```

(485) unknown “with” psect referenced by psect “*” (リンカ)
この psect の with フラグで指定された psect が存在しません。
例:

```
psect starttext,class=CODE,with=rext
; was that meant to be with text?
```

(486) psect “*” selector value redefined (リンカ)
この psect の selector 値が複数回定義されています。

(487) psect “*” type redefined:*/* (リンカ)
別のモジュールがこの psect に対して異なるタイプを定義しています。原因として、非互換のオブジェクト モジュールをリンクしようとしている可能性があります(例: 386 フラットモデル コードと 8086 リアルモード コードのリンク等)。

(488) psect “*” memory space redefined:*/* (リンカ)
1 つのグローバル psect が 2 つの異なるメモリ空間内で定義されています。どちらかの psect の名前を変更するか、それらが同じ psect である場合は space フラグを使ってそれらを同じメモリ空間内に配置してください。
例:

```
psect spdata,class=RAM,space=0
    ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect “*” memory delta redefined:*/* (リンカ)
global psect が 2 つの異なる delta 値で定義されています。
例:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class “*” memory space redefined:*/* (リンカ)
1 つのクラスが 2 つの異なるメモリ空間内で定義されています。どちらかのクラスの名前を変更するか、それらが同じクラスである場合はそれらを同じメモリ空間内に配置してください。

(491) can’t find 0x* words for psect “*” in segment “*” (リンカ)
リンカの主要タスクの 1 つは、プログラムから生成されるコードとデータのブロック(または psect)をターゲット デバイスのメモリ内に配置する事です。このエラーは、psect の 1 つを格納するために十分なサイズの空きメモリ領域が見つからなかった事を示します。

このメッセージには、リンカが配置を試みた psect の名前と、セグメントの名前(通常はコンパイラ ドライバから-A リンカオプションによって定義されたクラスの名前) が示されます。通常プログラムコードに関連する psect の名前は text (または text を含んだ名前)です。bss または data 等の名前は変数ブロックを参照します。このエラーには 2 つの原因が考えられます。

1 つは、プログラムまたはプログラムのデータのサイズがターゲット デバイス上のメモリ空間のサイズを超えた場合(言い換えると、デバイスのメモリの一部が完全に書き込まれた場合)です。これが原因である場合、指定された psect のサイズを削減する必要があります。

もう 1 つは、psect に必要な空きメモリ容量は十分に存在するものの、それが断片化されているために psect を 1 つの連続したブロック内に配置できない場合です。リンカは psect を複数の領域に分けて配置する事ができません。このため、各 psect を格納可能な連続したメモリブロックが見つからないとこのエラーが発生します。その場合、可能であれば psect を小さく分割する必要があります。

利用可能な空きメモリ領域は、マップファイル内の**「UNUSED ADDRESS RANGES」**セクションで調べる事ができます。このタイトルの下で、このエラーメッセージに示されたセグメントの名前を探します。セグメント名が見つからない場合、その psect 向けに利用可能な全てのメモリは割り当て済みです。セグメント名が見つかった場合、その下にメモリの各空きブロックのアドレスレンジが示されます。これらの各ブロックのサイズとエラーメッセージに示されたワード数を比較します。

コードを格納した psect のサイズは、コンパイラの全ての最適化を適用するか、プログラムの構成を見直す事で削減できる場合があります。断片的な空きメモリを利用する目的でコード psect を複数の小さな psect に分割する場合、大きな関数は相互に呼び出し可能な小さな関数に分割する必要があります。これらの関数は、新しいモジュール内に配置する事が必要になる場合があります。

データを格納した psect のサイズもコンパイラの最適化によって削減できる可能性はありますが、その効果はコード psect ほど大きくありません。プログラムを書き直して変数を削減する事が必要になる場合もあります。既定値のリンカ オプションを変更する必要がある場合、ドライバ オプション-Wl を使ってドライバから間接的に変更できます。データ psect が配置できない場合、通常は変数の総サイズを削減する必要があります。

例として、下記のメッセージが表示された場合を想定します。

```
Can’t find 0x34 words (0x34 withtotal) for psect text in segment CODE (error)
```

マップファイルで未使用メモリのレンジを調べます。

例:

```
UNUSED ADDRESS RANGES
CODE             00000244-0000025F
                 00001000-0000102f
RAM              00300014-00301FFB
```

CODE セグメント内には、0x1c (= 0x25f - 0x244+1)バイトのブロックと 0x30 バイトのブロックが存在します。これらのブロックは、どちらも 0x34 バイトの text psect を格納できません。しかし利用可能メモリの総サイズは 0x34 バイト以上あるため、text psect 内へとエンコードされる関数を 2 つの小さな関数に分割してプログラムがそれらを正しくリンクできれば、この psect を格納できる可能性があります。

(492) attempt to position absolute psect “*” is illegal (リンカ)
この psect は絶対アドレス指定されており、-P オプションでアドレスを指定してはいけません。この psect の abs フラグまたは-P リンカオプションのどちらかを削除する必要があります。

(493) origin of psect “*” is defined more than once (リンカ)
この psect の origin が複数回定義されています。この psect を指定している-p リンカ オプションが複数存在している可能性があります。

(494) bad -P format “*/*” (リンカ)
リンカに対して指定された-P オプションの書式が正しくありません。このオプションは psect の配置を指定します。
例:

```
-Ptext=10g0h
```

上記は誤りであり、下記が訂正例です。

```
-Ptext=10f0h
```

(495) use of both “with=” and “INCLASS/INCLASS” allocation is illegal (リンカ)
psect が with フラグを使って定義されている場合、その psect のリンクと位置の両方を 1 つのクラス内として指定する事はできません。

(497) psect “*” exceeds max size:*h > *h (リンカ)
この psect のバイト数は、size フラグによって指定された最大許容値を超えています。

(498) psect “*” exceeds address limit:*h > *h (リンカ)
この psect の最高アドレスは、limit フラグによって指定された上限を超えています。この psect のリンク位置を変更するか、psect 内のコードまたはデータの量を削減する必要があります。

(499) undefined symbol:(アセンブラ、リンカ)
このメッセージの後に示されたシンボルはリンク時に未定義です。原因としてスペルミスがあるか、適切なモジュールとのリンクに失敗した可能性があります。

10.2 メッセージ 500～999

(500) undefined symbols:(リンカ)
このメッセージの後に示されたシンボルはリンク時に未定義です。原因としてスペルミスがあるか、適切なモジュールとのリンクに失敗した可能性があります。

(501) program entry point is defined more than once (リンカ)
リンカに渡されたオブジェクト ファイル内で複数のエントリポイントが定義されています。end エントリポイントは END ディレクティブの後で指定されます。実行時起動コードはエントリポイントを定義します。
例:

```
powerup:
goto start
END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * (リンカ)
オブジェクト ファイルは無効なサイズのレコードを格納しています。そのファイルは切り詰められたか、オブジェクト ファイルではない可能性があります。Microchip 社技術サポートに詳細をお問い合わせください。

(503) ident records do not match (リンカ)
リンカに渡されたオブジェクト ファイル内の ident レコードが一致しません。これらのファイルは異なるデバイスタイプ向けのファイルです。

(504) object code version is greater than *.*(リンカ)
オブジェクト モジュールのオブジェクト コードバージョンは、リンカがサポート可能な最高バージョンを超えています。適切なリンカを使っているか確認してください。リンカにパッチが未適用である場合、Microchip 社技術サポートにお問い合わせください。

(505) no end record found inobject file (リンカ)
オブジェクト ファイルは終了レコードを格納していません。そのファイルは切り詰められたか、オブジェクト ファイルではない可能性があります。コンパイラによって生成されたオブジェクト ファイルでこの問題が発生する場合、Microchip 社の技術サポートにお問い合わせください。

(506) object file record too long:*+* (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(507) unexpected end of file in object file (リンカ)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(508) relocation offset (*) out of range 0..*-*-1 (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(509) illegal relocation size:* (リンカ)
リンカが読み出したオブジェクト コード フォーマットにエラーが存在します。古いバージョンのリンカを使っているか、アセンブラまたはリンカに内部エラーが存在する可能性があります。コンパイラによって生成されたオブジェクト ファイルでこの問題が発生する場合、Microchip 社の技術サポートにお問い合わせください。

(510) complex relocation not supported for -R or -L options (リンカ)
リンカに対して複雑な再配置を格納したファイルと一緒に-R または-L オプションが指定されました。

(511) 1) bad complex range check (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(512) unknown complex operator 0x* (リンカ)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(513) bad complex relocation (リンカ)
リンカは、構文的に正しくない複雑な再配置を実行するよう要求されました。オブジェクト ファイルが破損している可能性があります。

(514) illegal relocation type:* (リンカ)
オブジェクト ファイルは無効な再配置タイプを持つ再配置レコードを格納しています。このファイルは切り詰められたか、オブジェクト ファイルではない可能性があります。コンパイラによって生成されたオブジェクト ファイルでこの問題が発生する場合、Microchip 社の技術サポートにお問い合わせください。

(515) unknown symbol type * (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(516) text record has bad length:*-*-(*+1) < 0 (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(520) function “*” is never called (リンカ)
この関数は決して呼び出されません。動作上の問題はありませんが、この関数を削除する事でメモリ空間を節約できます。この関数が呼び出される必要がある場合、ソースコードを確認してください。一部のアセンブラ ライブラリコードは決して呼び出されませんが、実際には実行されます。この場合、ルーチンは特別な順序でリンクされているため、プログラムの実行は 1 つのルーチンから次のルーチンに繋がって実行されます。

(521) call depth exceeded by function “*” (リンカ)
コールグラフによると、その関数は指定されたよりも深くネストされています。

(522) library “*” is badly ordered (リンカ)
このライブラリの順番は適切ではありません。これでも正しくリンクされますが、順番を最適化する事でより高速なリンクが可能です。

(523) argument to -W option (*) illegal and ignored (リンカ)
リンカオプション-w に対する引数がレンジ外です。このオプションは、警告レベル(レンジは-9～9)またはマップファイルの幅(レンジは 10 以上)を制御します。

(524) unable to open list file “*”:* (リンカ)
リンカは、リストファイルのフィックスアップ(絶対アドレス化)を試みましたが、このリストファイルを開く事ができませんでした。コンパイル処理中にアセンブリ リストファイルが生成されたか確認してください。あるいは、リンク処理からアセンブラ リストファイル生成オプションを削除してください。

(525) too many address (memory) spaces; space (*) ignored (リンカ)
アドレス空間の数(PSECT アセンブラ ディレクティブで指定)は、現在 16 以下に制限されています。

(526) psect “*” not specified in -P option (first appears in “*”) (リンカ)
この psect はリンカに対する-P または-A オプション内で指定されていません。この psect は、プログラムの最後でリンクされましたが、それが本来意図された位置ではないと考えられます。

(528) no start record; entry point defaults to zero (リンカ)
開始レコードを格納したオブジェクト ファイルがリンカに渡されませんでした。プログラムの開始アドレスは 0 に設定されました。これは無害かもしれませんが、END ディレクティブを使って起動モジュール内で開始アドレスを定義する事を推奨します。

(529) usage: objtohex [-Ssymfile] [object-file [HEX-file]] (Objtohex)
コマンドライン ツール objtohex が正しく使われていません。このツールを直接呼び出していないのにこのメッセージが表示される場合、これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(593) can’t find 0x* words (0x* withtotal) for psect “*” in segment “*” (リンカ)
メッセージ(491)を参照してください。

(594) undefined symbol:(リンカ)
このメッセージの後に示されたシンボルはリンク時に未定義です。原因としてスペルミスがあるか、適切なモジュールとのリンクに失敗した可能性があります。

(595) undefined symbols:(リンカ)
このメッセージの後に示されたシンボルはリンク時に未定義です。原因としてスペルミスがあるか、適切なモジュールとのリンクに失敗した可能性があります。

(596) segment “*” (*-*) overlaps segment “*” (*-*) (リンカ)
これらの 2 つのセグメントの間でコードまたはデータがオーバーラップしています。-P リンカオプションによって割り当てられているアドレスを確認してください。

(599) No psect classes given for COFF write (Cromwell)
COFF ファイルを生成するには、CROMWELL に対してプログラムメモリ psect クラスを指定する必要があります。-N オプションを使っているか確認してください。

(600) No chip arch given for COFF write (Cromwell)
COFF ファイルを生成するには、CROMWELL に対してデバイス アーキテクチャを指定する必要があります。-P オプションを使っているか確認してください。

(601) Unknown chip arch “*” for COFF write (Cromwell)
COFF ファイルの生成用に指定されたデバイス アーキテクチャは CROMWELL によって認識されませんでした。-P オプションを使っているか、およびアーキテクチャが正しく指定されているか確認してください。

(602) null file format name (Cromwell)
CROMWELL に対する-I または-O オプションでは、ファイル フォーマットを指定する必要があります。

(603) ambiguous file format name “*” (Cromwell)
CROMWELL に対して指定された入力または出力フォーマットが不明です。これらのフォーマットは、それぞれ-i および-o キーオプションにより指定されます。

(604) unknown file format name “*” (Cromwell)
CROMWELL に対して不明の出力フォーマットが指定されています。
例:

```
cromwell -m -P16F877 main.HEX main.sym -ocot
```

cot という出力ファイルタイプは存在しません。cof の間違いかもしれません。

(605) did not recognize format of input file (Cromwell)
CROMWELL への入力ファイルのフォーマットは Cromwell マップファイル(CMF)、COD、Intel HEX、Motorola HEX、COFF、OMF51、ELF、UBROF、HI-TECH のいずれかである事が必要です。

(606) inconsistent symbol tables (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(607) inconsistent line number tables (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(608) bad path specification (Cromwell)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(609) missing device spec after -P (Cromwell)
CROMWELL に対する-p オプションでは、デバイス名を指定する必要があります。

(610) missing psect classes after -N (Cromwell)
-N オプションにより CROMWELL に対して psect クラスの名前のリストを提供する必要があります。

(611) 1) too many input files (Cromwell)
CROMWELL による変換用に指定された入力ファイルの数が多すぎます。

(612) too many output files (Cromwell)
CROMWELL に対して指定された出力ファイル フォーマットの数が多すぎます。

(613) no output file format specified (Cromwell)
CROMWELL に対して出力フォーマットを指定する必要があります。

(614) no input files specified (Cromwell)
CROMWELL に対して変換する入力ファイルを指定する必要があります。

(616) option -Cbaseaddr is illegal with options -R or -L (リンカ)
リンカオプション-Cbaseaddr とリンカオプション-R または-L を一緒に使う事はできません。

(618) error reading COD file data (Cromwell)
入力 COD ファイルの読み出しでエラーが発生しました。コマンドラインでファイル名とパスを正確なスペルで入力してください。

(619) I/O error reading symbol table (Cromwell)
COD ファイルは、指定されたレコード内に無効なフォーマットを含んでいます。

(620) filename index out of range in line number record (Cromwell)
COD ファイルは、指定されたレコード内に無効な値を含んでいます。

(621) error writing ELF/DWARF section “*” on “*” (Cromwell)
このセクションをこのファイルへ書き込む際にエラーが発生しました。コマンドラインでファイル名とパスを正確なスペルで入力してください。

(622) too many type entries (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(623) bad class in type hashing (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(624) bad class in type compare (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(625) too many files in COFF file (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(626) string lookup failed in COFF: get_string() (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(627) missing “*” in SDB file “*” line * column * (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(629) bad storage class “*” in SDB file “*” line * column * (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(630) invalid syntax for prefix list in SDB file “*” (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(631) syntax error at token “*” in SDB file “*” line * column * (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(632) can’t handle address size (*) (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(633) unknown symbol class (*) (Cromwell)
CROMWELL は COFF、Microchip COFF、ICOFF ファイルのシンボルテーブル内で不明なシンボルクラスに遭遇しました。

(634) error dumping “*” (Cromwell)
CROMWELL への入力ファイルのタイプが非サポートであるか、そのファイルは画面に出力できません。

(635) invalid HEX file “*” on line * (Cromwell)
指定された HEX ファイル内に無効な行が存在します。コンパイラによって生成された HEX ファイルでこの問題が発生する場合、Microchip 社の技術サポートにお問い合わせください。

(636) error in Intel HEX file “*” on line * (Cromwell, Hexmate)
この Intel HEX ファイル内のこの行でエラーが見つかりました。HEX ファイルは破損している可能性があります。

(637) unknown prefix “*” in SDB file “*” (Cromwell)
これは内部コンパイラ警告です。Microchip 社技術サポートに詳細をお問い合わせください。

(638) version mismatch:0x* expected (Cromwell)
入力された Microchip COFF ファイルは CROMWELL を使って生成された物ではありません。

(639) zero bit width in Microchip optional header (Cromwell)
入力された Microchip COFF ファイル内のオプションヘッダは、プログラムメモリまたはデータメモリ空間の幅が 0 ビットであると示しています。

(668) prefix list did not match any SDB types (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(669) prefix list matched more than one SDB type (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(670) bad argument to -T (Clist)
-T オプションの引数(タブサイズを指定)が存在しないか、書式が正しくありません。このオプションには 10 進整数の引数が必要です。

(671) argument to -T should be in range 1 to 64 (Clist)
-T オプションの引数(タブサイズを指定)が許容レンジを超えています。この引数の許容レンジは 1～64 (10 進整数、1 と 64 を含む)です。

(673) missing filename after * option (Objtohex)
このオプションには有効なファイル名が必要です。このオプションに対してファイル名を正しいスペルで指定してください。

(674) too many references to “*” (Cref)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(679) unknown extraspecial:* (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(680) bad format for -P option (コード ジェネレータ)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(681) bad common spec in -P option (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(685) bad putwsize() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(686) bad switch size (*) (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(687) bad pushreg “*” (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(688) bad popreg “*” (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(689) unknown predicate “*” (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(691) interrupt functions not implemented for 12 bit PIC MCU (コード ジェネレータ)
12 ビット(ベースライン) PIC MCU は割り込みをサポートしません。

(692) more than one interrupt level is associated with the interrupt function “*” (コード ジェネレータ)
割り込み関数には割り込みレベルを 1 つだけ割り当てる事ができます。この関数で interrupt_level プラグマが 1 つだけ使われている事を確認してください。割り込み関数から呼び出されるメインライン関数では、このプラグマを複数回使えます。
例:

```
#pragma interrupt_level 0
#pragma interrupt_level 1 /* oops -- which is it to be:0 or 1?*/
void interrupt isr(void)
{
```

(693) 0 (default) or 1 are the only acceptable interrupt levels for this function (コード ジェネレータ)
設定可能な割り込みレベルは 0 または 1 のみです。全ての interrupt_level プラグマがこれらのレベルを使っている事を確認してください。
例:

```
#pragma interrupt_level 2 /* oops -- only 0 or 1 */
void interrupt isr(void)
{
    /* isr code goes here */
}
```

(694) no interrupt strategy available (コード ジェネレータ)
このデバイスは割り込みサービスルーチン中のレジスタの保存/復元をサポートしません。

(695) duplicate case label (*) (コード ジェネレータ)
switch 文内に同じ値を持つ 2 つの case ラベルが存在します。
例:

```
switch(in) {
case ’0’:/* if this is case ’0’...*/
    b++;
    break;
case ’0’:/* then what is this case?*/ b-
    -;
    break;
}
```

(696) out-of-range case label (*) (コード ジェネレータ)
この case ラベルの値は制御式から得られる値のレンジ内ではないため、このラベルは決して選択されません。

(697) non-constant case label (コード ジェネレータ)
この switch 文内の case ラベルの値が定数ではありません。

(698) bit variables must be global or static (コード ジェネレータ)
bit 変数は auto にはできません。コードまたは関数のブロックに対してローカルなスコープを持つビット変数が必要な場合、その変数を static として修飾する必要があります。
例:

```
bit proc(int a)
{
    bit bb;        /* oops --  this should be: static bit bb; */
    bb = (a > 66);
    return bb;
}
```

(699) no case labels in switch (コード ジェネレータ)
switch 文内に case ラベルが存在しません。
例:

```
switch(input) {
}              /* there is nothing to match the value of input */
```

(700) truncation of enumerated value (コード ジェネレータ)
このコンパイラがサポートする最大値よりも大きな列挙値が検出され、切り詰められました。
例:

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(701) unreasonable matching depth (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(702) regused(): bad arg to G (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(703) bad GN (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(704) bad RET_MASK (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(705) bad which (*) after I (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(706) bad which in expand() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(707) bad SX (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(708) bad mod “+” for how = “*” (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(709) metaregister “*” can’t be used directly (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(710) bad U usage (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(711) 1) bad how in expand() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(712) can’t generate code for this expression (コード ジェネレータ)
このエラーは、C 式が複雑すぎたためにコード ジェネレータがコンパイルできなかった事を示します。コードを生成するには、コード ジェネレータがその式のコンパイル方法を知っている事と、十分なリソース (レジスタ、一時記憶位置等)が利用可能である事が必要です。一時変数を使って中間結果を保持する事により式を単純化する事で、しばしばこの問題を回避する事ができます。

このエラーは、コンパイルされるコードが異常である場合にも発行される可能性があります。例えば、`const` 修飾されたオブジェクトに書き込む事は禁止されており、そのようなコードが見つかると警告メッセージが発行されます。しかし、コード ジェネレータはそのような書き込みを実行するためのコードを生成しようと試みて失敗する可能性があります。

このエラーは、`intrinsic` プラグマを使う関数の再定義が試みられた場合にも発生する可能性があります。

(713) bad initialization list (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(714) bad intermediate code (コード ジェネレータ)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(715) bad pragma “*” (コード ジェネレータ)
コード ジェネレータは渡された pragma ディレクティブを理解できません。その pragma ディレクティブはターゲット デバイス向けに実装されていない可能性があります。

(716) bad argument to -M option “*” (コード ジェネレータ)
コード ジェネレータは渡された-M オプションを理解できません。コード ジェネレータが標準コンパイラ ドライバによって呼び出された場合、この問題は発生しません。

(718) incompatible intermediate code version; should be *.*(コード ジェネレータ)
p1 によって生成された中間コードファイルのバージョンは、このコード ジェネレータに適合しません。同じディレクトリに 1 つまたは複数の非互換バージョンのコンパイラがインストールされているか、エラーによって一時ファイルが破損した可能性があります。TEMP 環境変数の設定を確認してください。TEMP が長いパス名を参照している場合、これを短く変更してください。Microchip 社技術サポートに詳細をお問い合わせください。

(720) multiple free:* (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(721) element count must be constant expression (コード ジェネレータ)
配列内の要素数は定数式によって指定される必要があります。const 修飾された変数は定数式を形成しません。
例:

```
const unsigned char mCount = 5;
int mDeadtimeArr[mCount]; // oops -- the size cannot be a variable
```

(722) bad variable syntax in intermediate code (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(723) function definitions nested too deep (コード ジェネレータ)
このエラーは C コードでは発生しません(C 言語では関数のネストは不可能であるため)。Microchip 社技術サポートに詳細をお問い合わせください。

(724) bad op (*) in revlog() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(726) bad op “*” in uconval() (コード ジェネレータ)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(727) bad op “*” in bconfloat() (コード ジェネレータ)
This is an internal code generator error.Microchip 社技術サポートに詳細をお問い合わせください。

(728) bad op “*” in confloat() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(729) bad op “*” in conval() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(730) bad op “*” (Code Generator)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

© 2022 Microchip Technology Inc. and its subsidiaries

(731) expression error with reserved word (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(732) initialization of bit types is illegal (コード ジェネレータ)
__bit 型の変数は初期化できません。
例:

```
__bit b1 = 1; /* oops! b1 must be assigned after its definition */
```

(733) bad string “*” in pragma “psect” (コード ジェネレータ)
コード ジェネレータに渡された#pragma psect ディレクティブは、無効な書式の文字列を含んでいます。
例:

```
#pragma psect text  /* redirect text psect into what?*/
```

以下のコードが正しい使用例です。

```
#pragma psect text=special_text
```

(734) too many “psect” pragmas (コード ジェネレータ)
#pragma psect ディレクティブの数が多すぎます。

(735) bad string “*” in pragma “stack_size” (コード ジェネレータ)
stack_size プラグマに対する引数の書式が正しくありません。このプラグマの後で最大許容スタックサイズを表す値を指定する必要があります。

(737) unknown argument “*” to pragma “switch” (コード ジェネレータ)
#pragma switch ディレクティブに対して無効な switch コード生成方法が指定されています。有効な引数は auto、simple、direct です。

(739) error closing output file (コード ジェネレータ)
ファイルを閉じる時にコンパイラがエラーを検出しました。Microchip 社技術サポートに詳細をお問い合わせください。

(740) zero dimension array is illegal (コード ジェネレータ)
配列が 0 次元となる宣言がコード ジェネレータに渡されました。

(741) bitfield too large (* bits) (コード ジェネレータ)
ビットフィールド内の最大ビット数は、記憶域ユニットの幅と同じ 8 です。
例:

```
struct {
    unsigned flag :1;
    unsigned value :12; /* oops -- that’s larger than 8 bits wide */
    unsigned cont :6;
} object;
```

(742) function “*” argument evaluation overlapped (リンカ)
この関数呼び出しの引数により、2 つの関数がオーバーラップします。これは、下記例のような呼び出しで発生します。
例:

```
void fn1(void)
{
    fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
    return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
    return one+two+three;
}
```

fn1 が fn3 を呼び出す際に 2 つの引数を評価するために fn2 が呼び出され、その fn2 は fn3 を呼び出しています。この種の呼び出しシーケンスを避けるため、プログラムの構造を変更する必要があります。

(743) divide by zero (コード ジェネレータ)
0 による除算を含む式がコードから削除されました。

(744) static object “*” has zero size (コード ジェネレータ)
サイズが 0 の static オブジェクトが宣言されました。

(745) nodecount = * (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(746) object “*” qualified const but not initialized (コード ジェネレータ)
const として宣言されたこのオブジェクトは、定義時に初期化されません(初期値が提供されていません)。C プログラムからこのオブジェクトに書き込む事はできないため、誤って初期値が省略されてしまった可能性があります。

(747) unrecognized option “*” to -Z (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(748) variable “*” possibly used before being assigned a value (コード ジェネレータ)
この変数は値が代入される前に使われた可能性があります。これは auto 変数であるため、その値は予測不可能です。
例:

```
int main(void)
{
    int a;
    if(a)         /* oops -- ’a’ has never been assigned a value */
        process();
}
```

(749) unknown register name “*” used with pragma (リンカ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(750) constant operand to || or && (コード ジェネレータ)
論理演算子 | | または&&に対する 1 つのオペランドが定数です。式内の丸カッコの位置が正しいか確認してください。グローバル オプティマイザが有効かつオペランドの 1 つが auto または static ローカル変数である場合、コード ジェネレータがその値を追跡した時にこのメッセージが生成される可能性があります。
例:

```
{
    int a;
    a = 6;
    if(a || b) /* a is 6, therefore this is always true */
        b++;
```

(751) arithmetic overflow in constant expression (コード ジェネレータ)
コード ジェネレータが定数式を評価した結果、その値は式の型に対して大きすぎると評価されました。多くの場合、この警告は符号付きデータ型に値を代入するコードによって発生します。
例:

```
signed char c;
c = 0xFF;
```

符号付き 8 ビット変数 c には-128～127 の値しか代入できません。上記のコードでは 255 が代入されるため、このレンジを超えています。この変数内の全てのビットをセットする必要がある場合、下記の 2 通りの方法が使えます。

```
c = ~0x0;
c = -1;
```

これらは、変数のサイズに関係なく、その中の全てのビットをセットします。

この警告は、中間生成値のオーバーフローによって生成される場合もあります。

例:

```
unsigned int i; /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

240 x 137 = 32880 であるため、unsigned int に保存できそうに思われますが、実際には警告が発行されます。その理由は、240 と 137 の両方が signed int 値であるからです。この場合、乗算結果も signed int 値となり、signed int では 32880 を保持できません。両方のオペランドが定数であるため、コード ジェネレータはコンパイル時にこの式を評価可能ですが、これは全ての ANSI C 規則に従う必要があります。符号なしの乗算結果は、少なくとも 1 つのオペランドを符号なしに指定する事により得られます。

例:

```
i = 240u * 137; /* force at least one operand
to be unsigned */
```

(752) conversion to shorter data type (コード ジェネレータ)

rvalue よりも lvalue の方がデータ型が短いため、この式は切り詰められる可能性があります。

例:

```
char a;
int b, c;
a = b + c;  /* int to char conversion can result in truncation */
```

(753) undefined shift (* bits) (コード ジェネレータ)

データ型のビット数以上のシフトが試みられました。多くのプロセッサでは、これによって不定の結果が生じます。これは移植性に問題のあるコードであり、C 規格はその結果を未確定として扱います。

例:

```
int input;
input <<= 33;  /* oops -- that shifts the entire value out */
```

(754) bitfield comparison out of range (コード ジェネレータ)

ビットフィールドがそのレンジを超えた値と比較されました。例えば、2 ビットのビットフィールド(値のレンジは 0〜3)と値 5 の比較結果は決して「真」とはなりません。

例:

```
struct {
    unsigned mask :2; /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
    return (value.mask == 6); /* test can
}
```

(755) divide by zero (コード ジェネレータ)

定数式に 0 による除算が含まれています。

例:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch (コード ジェネレータ)

if、for、while 文等による条件分岐の結果が常に同じです(条件分岐を使う意味がありません)。これは、変数と定数式を含む比較等で生じる可能性があります。変数がローカルスコープ(auto または static)を持ち、かつグローバル オプティマイザが有効で最適化レベルが 2 以上、警告レベルしきい値が既定値レベル(0)未満に設定されている場合、コード ジェネレータからこのメッセージが発行される可能性があります。

グローバル オプティマイザは、可能な限りローカル変数の内容を追跡します。C コードがこれらのローカル変数と定数を比較している場合、コンパイル時にその結果が推測され、比較を行う意味がなければ比較を省略したコードが出力されます。

例:

```
{
int a, b;
a = 5;
/* this can never be false; always perform the true statement */
```

```
if(a == 5)
    b = 6;
```

上記からは、単純に a を 5 に設定し b を 6 に設定するコードが生成され、if(a == 5)に対応するコードは生成されません。a がグローバル変数である場合、他の関数(特に割り込み関数)が a を変更する可能性があるため、この変数を追跡する事はできません。

この警告は、コンパイラによる最適化とは無関係に発行される場合もあります。例えば、式内のカッコの位置が不適切であるかカッコが抜けているために正しい評価結果が得られない可能性がある場合に発行されます。

さらに、while(1)等のコードが使われていた場合にも発行される可能性があります。この場合、for(;;)を使って無限ループを生成する事で、この警告を防ぐ事ができます。

通常のループでも、同様の状況が発生します。

```
{
int a, b;
/* this loop must iterate at least once */
for(a=0; a!=10; a++)
    b = func(a);
```

上記の場合、コード ジェネレータは a に値 0 が代入された直後に a が値 10 と比較されている事を検出します。for ループ中に a は変更されるため、この比較コードを削除する事はできませんが、コード ジェネレータはループの最初の実行時に比較が実行されないようにコードを調整します。これによりコードのサイズは削減されませんが、実行速度が向上します。

(758) constant conditional branch: possible use of “=” instead of “==” (コード ジェネレータ)

if またはその他の条件コンストラクタ内で、定数を変数に代入する式が使われています。例えば、本来「==」(比較)を使うべき位置で誤って「=」(代入)を使っている可能性があります。

```
int a, b;
/* this can never be false; always perform the true statement */
if(a = 4)
    b = 6;
```

上記は if の条件式で a に値 4 を代入しています。代入式の論理値は常に「真」であるため、比較は行われず常に b に 6 が代入されます。上記は下記のように書き直す事ができます。

```
/* this can never be false;
always perform the true statement */
if(a == 4)
    b = 6;
```

この場合、a が 4 に等しい場合にのみ b に 6 が代入されます。

(759) expression generates no code (コード ジェネレータ)

この式は出力コードを生成しません。関数内にカッコの抜け等がないか確認してください。

例:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

一部のデバイスでは、ハードウェア フラグをクリアするために特殊機能レジスタを読み出す必要があります。そのような場合、コード ジェネレータは変数 ID のみを含む命令文のコードを生成します。この状況は volatile 修飾された変数に対して発生する可能性があります。通常、生成されたコードは変数を読み出すだけで他に何もしません。

(760) portion of expression has no effect (コード ジェネレータ)

この式の一部は、式の値に対して何の効果も持ちません。

例:

```
int a, b, c;
a = b,c; /* "b" has no effect, was that meant to be a comma?*/
```

(761) size of yields 0 (コード ジェネレータ)
コード ジェネレータは、オブジェクトのサイズが 0 である事を検出しました。ポインタの宣言内に問題がある事はほぼ確実であり、長さ 0 の配列に対してポインタが宣言されている可能性があります。一般的に、配列に対するポインタはほとんど使われません。長さが不明のオブジェクトの配列に対するポインタが必要である場合、1 つのオブジェクトに対するポインタだけを使い、そのポインタにインデックスを付けるかそのポインタをインクリメントする事ができます。

(762) constant truncated when assigned to bitfield (コード ジェネレータ)
ビットフィールド構造体メンバーに代入されている定数値が大きすぎます。
例:

```
struct INPUT
    { unsigned
    a :3; unsigned
    b :5;
} input_grp;
input_grp.a = 0x12;  /* oops -- 0x12 cannot fit into a 3-bit wide object */
```

(763) constant left operand to “?:” operator (コード ジェネレータ)
条件演算子?の左辺オペランドは定数であるため、 三項演算子? : の結果は常に同じです。

```
a = 8 ? b : c; /* this is the same as saying a = b; */
```

(764) mismatched comparison (コード ジェネレータ)
変数または式と定数が比較されていますが、定数の値は変数または式の値のレンジ内ではありません。
例:

```
unsigned char c;
if(c > 300)      /* oops -- how can this be true? */
    close();
```

(765) degenerate unsigned comparison (コード ジェネレータ)
結果が常に「真」または常に「偽」となる比較(符号なし値と 0 の間の比較) が存在します。
例:

```
unsigned char c;
if(c >= 0)
    ...
```

符号なし値は 0 未満にならないため、上記は常に真です。

(766) degenerate signed comparison (コード ジェネレータ)
結果が常に「真」または常に「偽」となる比較(符号付き値とその型での負の最大値との比較) が存在します。
例:

```
char c;
if(c >= -128)
    ...
```

8 ビット符号付き char の負の最大値は-128 であるため、上記は常に真です。

(767) constant truncated to bitfield width (コード ジェネレータ)
ビットフィールド構造体メンバーに対して定数の値が大きすぎます。
例:

```
struct INPUT
    { unsigned
    a :3; unsigned
    b :5;
} input_grp;
input_grp.a |= 0x13;   /* oops -- 0x13 to large for 3-bit wide object */
```

(768) constant relational expression (コード ジェネレータ)
常に「真」または常に「偽」となる関係式が存在します。例えば、符号なし値と負値との比較や、変数とその変数が表現可能な最大値より大きな値との比較等がこれに該当します。
例:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
b = 9;
```

(769) no space for macro definition (アセンブラ)
アセンブラの実行中にメモリが足りなくなりました。

(772) include files nested too deep (アセンブラ)
マクロ展開とインクルード ファイルの処理により、アセンブラの内部スタックが使い果たされました。開く事ができるマクロとインクルード ファイルの最大数は 30 です。

(773) macro expansions nested too deep (アセンブラ)
アセンブリコード内のマクロ展開のネストが深すぎます。マクロとインクルード ファイルは同時に最大 30 までネスト可能です。

(774) too many macro parameters (アセンブラ)
このマクロ定義に対するマクロ パラメータの数が多すぎます。

(776) can’t allocate space for object “*” (offs:*) (アセンブラ)
アセンブラの実行中にメモリが足りなくなりました。

(777) can’t allocate space for opnd structure within object “*” (offs:*) (アセンブラ)
アセンブラの実行中にメモリが足りなくなりました。

(780) too many psects defined (アセンブラ)
定義された psect の数が多すぎます。

(781) can’t enter abs psect (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(782) REMSYM error (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(783) “with” psects are cyclic (アセンブラ)
2 つの psect の両方が `with` フラグを使ってお互いを指定しています(例: psect A は `with` フラグで psect B を指定し、psect B は `with` フラグで psect A を指定)。psect の `with` フラグは、その psect を指定された他の psect と同じメモリページ内に配置する必要がある事を示します。`with` フラグは、どちらか一方の psect でのみ指定する必要があります。他方の psect 宣言から `with` フラグを削除してください。
`with` フラグを使ったアセンブラ宣言の例を以下に示します。

```
psect my_text,local,class=CODE,with=basecode
```

上記は `my_text` という名前の psect を定義し、これを `basecode` という名前の psect と同じページ内に配置します。

(784) overfreed (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(785) too many temporary labels (アセンブラ)
このアセンブリ ファイル内の一時ラベルの数が多すぎます。一時ラベルは最大で 2000 個まで許容されます。

(787) can't handle "v_rtype" of * in copyexpr (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(788) invalid character "*" in number (アセンブラ)
数値に 0～9 または 0～F のレンジ外の文字が使われています。

(790) end of file inside conditional (アセンブラ)
`if` に対応する `endif` が見つかる前にファイル終端(EOF)に遭遇しました。

(793) unterminated macro argument (アセンブラ)
マクロに対する引数の終端がありません。マクロ引数は山カッコ< >で囲む必要があります。

(794) invalid number syntax (アセンブラ)
数値の構文が無効です。例えば、8 進値で 8 または 9 が使われている場合、このエラーが示されます。

(796) use of LOCAL outside macros is illegal (アセンブラ)
`LOCAL` ディレクティブはマクロの内部でのみ使えます。これは、マクロの各呼び出しに対して一意となるローカルラベルを定義します。

(797) syntax error in LOCAL argument (アセンブラ)
アセンブラマクロ内で `LOCAL` アセンブラ ディレクティブを使って定義されたシンボルの構文が正しくありません。シンボルおよびその他のアセンブラ識別子の全てがターゲット デバイスのアセンブリ言語に適合している事を確認してください。

(798) use of macro arguments in a LOCAL directive is illegal (アセンブラ)
マクロ内の `LOCAL` ディレクティブの後のラベル リストにそのマクロに対する仮パラメータを含める事はできません。
例:

```
mmm MACRO a1
MOVE        r0, #a1
LOCAL        a1    ; oops -- the parameter cannot be used with LOCAL
ENDM
```

(799) REPT argument must be >= 0 (アセンブラ)
`REPT` ディレクティブに対する引数は 0 より大きな値である事が必要です。
例:

```
REPT -2                         ; -2 copies of this code?*/
MOVE            r0, [r1]++
ENDM
```

(800) undefined symbol "*" (アセンブラ)
このシンボルはこのモジュール内で定義されておらず、かつ `GLOBAL` として指定もされていません。

(801) range check too complex (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(802) invalid address after END directive (アセンブラ)
アセンブラ `END` ディレクティブの後で指定されたプログラムの開始アドレスは現在のファイル内のラベルである事が必要です。

(803) undefined temporary label (アセンブラ)
未定義の一時ラベルが参照されています。一時ラベルは 0 以上の番号を持つ必要があります。

(804) write error on object file (アセンブラ)
アセンブラがオブジェクト ファイルへの書き込みに失敗しました。これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(806) attempted to get an undefined object (*) (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(807) attempted to set an undefined object (*) (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(808) bad size in add_reloc() (アセンブラ)
これは内部コンパイラエラーです。詳細は Microchip 社技術サポートにお問い合わせください。

(809) unknown addressing mode (*) (アセンブラ)
不明のアドレス指定モードがアセンブリ ファイル内で使われました。

(811) “cnt” too large (*) in display() (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(814) device type not defined (アセンブラ)
デバイスタイプ(例: -16c84)が定義されていません。これはコマンドライン、デバイス アセンブラ ディレクティブ、LIST アセンブラ ディレクティブのいずれかで定義する必要があります。

(815) syntax error in chipinfo file at line * (アセンブラ)
chipinfo ファイル内のこの行で非標準の構文が使われています。

(816) duplicate ARCH specification in chipinfo file “*” at line * (アセンブラ、ドライバ)
chipinfo ファイルのデバイス セクション内に複数の ARCH 値が存在します。ARCH 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(817) unknown architecture in chipinfo file at line * (アセンブラ、ドライバ)
chip INI ファイルの読み出し中に不明のデバイス アーキテクチャ (ファミリ) に遭遇しました。

(818) duplicate BANKS for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の BANKS 値が存在します。BANKS 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(819) duplicate ZEROREG for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の ZEROREG 値が存在します。ZEROREG 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(820) duplicate SPAREBIT for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の SPAREBIT 値が存在します。SPAREBIT 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(821) duplicate INTSAVE for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の INTSAVE 値が存在します。INTSAVE 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(822) duplicate ROMSIZE for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の ROMSIZE 値が存在します。ROMSIZE 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(823) duplicate START for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の START 値が存在します。START 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(824) duplicate LIB for “*” in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に複数の LIB 値が存在します。LIB 値は 1 つだけである事が必要です。chipinfo ファイルをエディタで編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(825) too many RAMBANK lines in chipinfo file for “*” (アセンブラ)
chipinfo ファイルのデバイス セクション内の RAMBANK フィールドの数が多すぎます。数を減らしてください。

(826) inverted ram bank in chipinfo file at line * (アセンブラ、ドライバ)
chipinfo ファイル内の RAM フィールドで指定された HEX 値の順番が正しくありません。2 つ目の HEX 値は 1 つ目の HEX 値より大きい事が必要です。

(827) too many COMMON lines in chipinfo file for “*” (アセンブラ)
chipinfo ファイル内で共通メモリ (アクセスバンク)を指定している行の数が多すぎます。

(828) inverted common bank in chipinfo file at line * (アセンブラ、ドライバ)
chipinfo ファイル内の COMMON フィールドで指定された HEX 値の順番が正しくありません。2 つ目の HEX 値は 1 つ目の HEX 値より大きい事が必要です。chipinfo INI ファイルを変更していないのにこのエラーが発生する場合、Microchip 社技術サポートにお問い合わせください。

(829) unrecognized line in chipinfo file at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に不明の行が存在します。chipinfo INI を編集していないのにこのエラーが発生する場合、Microchip 社技術サポートにお問い合わせください。

(830) missing ARCH specification for “*” in chipinfo file (アセンブラ)
chipinfo ファイルのデバイス セクション内に ARCH 値がありません。デバイスのアーキテクチャを指定する必要があります。chipinfo ファイルを変更していないのにこのエラーが発生する場合、Microchip 社技術サポートにお問い合わせください。

(832) empty chip info file “*” (アセンブラ)
chipinfo ファイル内にデータがありません。chipinfo ファイルを編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートにお問い合わせください。

(833) no valid entries in chipinfo file (アセンブラ)
chipinfo ファイル内に有効なデバイス記述がありません。

(834) page width must be >= 60 (アセンブラ)
リストファイルのページ幅は 60 文字以上である事が必要です。これより小さな幅では、適正書式のリストファイルを生成できません。
例:

```
LIST C=10 ; the page width will need to be wider than this
```

(835) form length must be >= 15 (アセンブラ)
-F オプションで指定されたフォーム長は 15 行以上である事が必要です。この長さを 0 に設定する事は可能です。その場合、ページングが完全に OFF になります。既定値は 0 です(ページングなし)

(836) no file arguments (アセンブラ)
ファイルが指定されずにアセンブラが呼び出されました。何もアセンブルできません。

(839) relocation too complex (アセンブラ)
この式内の複雑な再配置が大きすぎてオブジェクト ファイルに挿入できません。

(840) phase error (アセンブラ)
アセンブラは 1 つのシンボルに対して 2 つの異なるパスで 2 つの異なる値を計算しました。通常これは psect が異なる delta 値を使って再定義された事が原因で発生します。

(841) bad source/destination for movfp/movpf instruction (アセンブラ)
movfp/movpf 命令で指定された絶対アドレスが長すぎます。

(842) bad bit number (アセンブラ)
ビット番号はレンジが 0〜7 の絶対式である事が必要です。

(843) a macro name can't also be an EQU/SET symbol (アセンブラ)
マクロと同名の EQU または SET シンボルが見つかりました。これは許容されません。
例:

```
getval MACRO
MOV  r0, r1
ENDM
getval EQU 55h ; oops -- choose a different name to the macro
```

(844) lexical error (アセンブラ)
入力内に不明の文字またはトークンが見つかりました。

(845) symbol "*" defined more than once (アセンブラ)
このシンボルは複数回定義されています。シンボルが同じモジュール内で複数回定義されている場合、このエラーはアセンブラによって発行されます。
例:

```
_next:
MOVE        r0, #55
MOVE        [r1], r0
 next:           ; oops -- choose a different name
```

シンボル(C またはアセンブラ)が異なるモジュール内で複数回定義されている場合、この警告はリンカによって発行されます。エラーメッセージには、モジュールの名前が示されます。コンパイル後は、しばしば C 識別子名の前に 1 つのアンダースコアが追加される事に注意してください。

(846) relocation error (アセンブラ)
2 つの再配置可能値を互いに加算する事はできません。再配置可能値に定数を加算する事と、同じ psect 内の 2 つの再配置可能アドレスを減算する事は可能です。絶対アドレス値は、アセンブル時にアセンブラがその値を知る事ができる位置で使われる必要があります。

(847) operand error (アセンブラ)
このオペコードに対するオペランドが無効です。オペランドの正しい書式をアセンブラのリファレンス マニュアルで確認してください。

(848) label defined in this module has also been declared EXTRN (アセンブラ)
アセンブリラベルの定義と、同じシンボルに対する EXTRN 宣言が、同じモジュール内に存在します。
他のモジュールからこのシンボルへのアクセスを可能とするには、EXTRN の代わりに GLOBAL を使う必要があります。

(849) illegal instruction for this device (アセンブラ)
このデバイスがサポートしていない命令です。

(850) PAGESEL not usable with this device (アセンブラ)
選択されたデバイスで PAGESEL 疑似命令を使う事はできません。

(851) illegal destination (アセンブラ)
この命令のデスティネーション(,f または,w)が正しくありません。

(852) radix must be from 2 - 16 (アセンブラ)
RADIX アセンブラ ディレクティブで指定される基数は、2 (2 進数)〜16 (16 進数) の範囲内である事が必要です。

(853) invalid size for FNSIZE directive (アセンブラ)
FNSIZE アセンブラ ディレクティブの引数は正の定数である事が必要です。

(855) ORG argument must be a positive constant (アセンブラ)
ORG アセンブラ ディレクティブの引数は正の定数または正の定数に割り当てられたシンボルである事が必要です。
例:

```
ORG -10  /* this must a positive offset to the current psect */
```

(856) ALIGN argument must be a positive constant (アセンブラ)
ALIGN アセンブラ ディレクティブの引数は、非 0 の正の整数である事が必要です。

(857) use of both local and global psect flags is illegal with same psect (リンカ)
local psect が global psect と同じ名前を持つ事はできません。
例:

```
psect text,class=CODE       ; the text psect is implicitly global
MOVE        r0, r1
; elsewhere:
psect text,local,class=CODE
MOVE        r2, r4
```

psect のスコープが明示的に指定されない場合、その psect は既定値により global とされます。

(859) argument to C option must specify a positive constant (アセンブラ)
LIST アセンブラディレクティブの C=オプション (リスト出力の列幅を設定) に対するパラメータは、正の 10 進定数である事が必要です。
例:

```
LIST C=a0h ; constant must be decimal and positive,
try:LIST C=80
```

(860) page width must be >= 49 (アセンブラ)
LIST アセンブラ ディレクティブのページ幅サブオプションは、49 以上の値を指定する必要があります。

(861) argument to N option must specify a positive constant (アセンブラ)
LIST アセンブラ ディレクティブの N オプション (リスト出力のページ長さを設定) に対するパラメータは、正の 10 進定数である事が必要です。
例:

```
LIST N=-3  ; page length must be positive
```

(862) symbol is not external (アセンブラ)
シンボルは EXTRN として宣言済みですが、現在のモジュールでも定義されています。

(863) symbol can't be both extern and public (アセンブラ)
extern として宣言されたシンボルはインポートされ、public として宣言されたシンボルは現在のモジュールからエクスポートされます。1 つのシンボルを extern と public の両方にする事はできません。

(864) argument to "size" psect flag must specify a positive constant (アセンブラ)
PSECT アセンブラ ディレクティブの size フラグに対するパラメータは、正の定数である事が必要です。
例:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect flag “size” redefined (アセンブラ)
PSECT アセンブラ ディレクティブの size フラグの値が以前の PSECT ディレクティブと異なります。
例:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) argument to “reloc” psect flag must specify a positive constant (アセンブラ)
PSECT アセンブラ ディレクティブの reloc フラグに対するパラメータは、正の定数である事が必要です。
例:

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect flag “reloc” redefined (アセンブラ)
PSECT アセンブラ ディレクティブの reloc フラグの値が以前の PSECT ディレクティブと異なります。
例:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) argument to “delta” psect flag must specify a positive constant (アセンブラ)
PSECT アセンブラ ディレクティブの delta フラグに対するパラメータは、正の定数である事が必要です。
例:

```
PSECT text,class=CODE,delta=-2  ; negative delta value doesn’t make sense
```

(869) psect flag “delta” redefined (アセンブラ)
psect の delta オプションが同じモジュール内で複数回再定義されています。

(870) argument to “pad” psect flag must specify a positive constant (アセンブラ)
PSECT アセンブラ ディレクティブの pad フラグに対するパラメータは、非 0 で正の整数である事が必要です。

(871) argument to “space” psect flag must specify a positive constant (アセンブラ)
PSECT アセンブラ ディレクティブの space フラグに対するパラメータは、正の定数である事が必要です。
例:

```
PSECT text,class=CODE,space=-1 ; space values start at zero
```

(872) psect flag “space” redefined (アセンブラ)
PSECT アセンブラ ディレクティブの space フラグの値が以前の PSECT ディレクティブと異なります。
例:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) a psect can only be in one class (アセンブラ)
1 つの psect を複数のクラスに割り当てる事はできません。この位置で psect が定義されていますが、他の位置で定義されたクラスと異なります。psect のクラスは class フラグにより指定されます。
例:

```
psect text,class=CODE
```

class フラグで異なるクラス名を指定している他の psect 定義を探してください。

(874) a psect can only have one “with” option (アセンブラ)
psect と一緒に配置できる他の psect は 1 つだけです。一緒に配置する psect の名前は with フラグにより指定されます。with フラグで異なる psect 名を指定している他の psect 定義を探してください。
例:

```
psect bss,with=data
; elsewhere
psect bss,with=lktab  ; oops -- bss is to be linked with two psects
```

(875) bad character constant in expression (アセンブラ)
文字定数が複数個の文字を格納しているか、何も格納していません。
例:

```
MOV r0, #'12'     ; '12' specifies two characters
```

(876) syntax error (アセンブラ)
構文エラーが検出されました。各種の原因が考えられます。

(877) yacc stack overflow (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(878) -S option used:"*" ignored (ドライバ)
このアセンブリ ファイルは-S オプション付きでドライバに渡されました。これは既にアセンブリ ファイルであるため、ドライバは何もやる事がありません。

(880) invalid number of parameters.Use "* –HELP" for help (ドライバ)
コンパイラ ドライバに対するコマンドラインの使い方が正しくありません。

(881) setup succeeded (ドライバ)
--setup ドライバ オプションによりコンパイラは正常にセットアップされました。

(883) setup failed (ドライバ)
--setup ドライバ オプションによるコンパイラのセットアップに失敗しました。このオプションに対するdirectory 引数 (スペルが正しいか、ホスト オペレーティング システムに対して構文的に正しいか、そのディレクトリが実際に存在しているか等)を確認してください。

(884) please ensure you have write permissions to the configuration file (ドライバ)
ドライバは XML コンフィグレーション ファイルにアクセスできなかったため、--setup ドライバ オプションによるコンパイラのセットアップは失敗しました。このファイルに書き込む権限を有しているか確認してください。ドライバは以下の順番でコンフィグレーション ファイルを探します。

- 環境変数 XC_XML によって指定されたファイル
- ディレクトリ「/etc 」が書き込み可能であれば、その中の xc.xml ファイル
- ホーム ディレクトリ内の.xc.xml ファイル

ファイルが何も見つからなかった場合、このエラーが発生します。

(889) this * compiler has expired (ドライバ)
このコンパイラのデモ期間は終了しました。

(890) contact Microchip to purchase and re-activate this compiler (ドライバ)
コンパイラのデモ インストールの評価期間は終了しました。再度有効にするには、コンパイラを購入する必要があります。評価期間に関してご不明な点がございましたら、Microchip 社の技術サポートにお問い合わせください。

(891) can't open psect usage map file "*":* (ドライバ)
ドライバはこのマップファイルを開く事ができませんでした。ドライバ オプション--summary=*file* を使うと、psect 使用量を示すマップファイルが生成されます。このファイルが別のアプリケーション内で開かれていないか確認してください。

(892) can't open memory usage map file "*":* (ドライバ)
ドライバはこのマップファイルを開く事ができませんでした。ドライバ オプション--summary=*file* を使うと、メモリ使用量を示すマップファイルが生成されます。このファイルが別のアプリケーション内で開かれていないか確認してください。

(893) can't open HEX usage map file "*":* (ドライバ)
ドライバはこのマップファイルを開く事ができませんでした。ドライバ オプション`--summary=`*`file`* を使うと、HEX 使用量を示すマップファイルが生成されます。このファイルが別のアプリケーション内で開かれていないか確認してください。

(894) unknown source file type "*" (ドライバ)
入力ファイルの拡張子が無効です。ドライバは以下のファイル拡張子を持つファイルのみ識別可能です
`.as、.c、.obj、.usb、.p1、.lib、.hex`

(895) can't request and specify options in the one command (ドライバ)
ドライバ オプション`--getoption` と`--setoption` を同時に使う事はできません。

(896) no memory ranges specified for data space (ドライバ)
ターゲット デバイスのデータ空間メモリ向けに内部メモリレンジも外部メモリレンジも指定されていません。

(897) no memory ranges specified for program space (ドライバ)
ターゲット デバイスのプログラム空間メモリ向けに内部メモリレンジも外部メモリレンジも指定されていません。

(899) can't open option file "*" for application "*":* (ドライバ)
`--getoption` または`--setoption` ドライバ オプションによって指定されたオプション ファイルを開く事ができませんでした。`--getoption` オプションを使った場合、ファイル名のスペルが正しいかどうかと、そのファイルが実際に存在しているかどうかを確認してください。`--setoption` オプションを使った場合、指定された位置でこのファイルが作成可能かどうかと、このファイルが他のアプリケーションによって使われていないかどうかを確認してください。

(900) exec failed:* (ドライバ)
このサブコンポーネントの実行に失敗しました。ファイルが実際に存在しているか確認してください。コンパイラの再インストールを試してください。

(902) no chip name specified; use "* –CHIPINFO" to see available chip names (ドライバ)
どのデバイス向けにビルドを行うのか選択されずにドライバが呼び出されました。`–CHIPINFO` オプションを指定してドライバを実行すると、ビルド向けに選択可能な全てのデバイスのリストが表示されます。

(904) illegal format specified in "*" option (ドライバ)
このオプションの使い方が正しくありません。--help オプションを使うかマニュアルを参照し、このオプションの正しい用法を確認してください。

(905) illegal application specified in "*" option (ドライバ)
このオプションに対して不明のアプリケーションまたは本コンパイラに含まれていないアプリケーションが指定されました。

(907) unknown memory space tag "*" in "*" option specification (ドライバ)
このメモリオプションに対するパラメータとして指定された文字列は、どの有効なタグとも一致しませんでした。本書内のこのオプションに関する説明を参照し、このデバイス向けにどのタグが有効なのか確認してください。

(908) exit status = * (ドライバ)
実行されたサブコンポーネントの 1 つがエラーコードを返しました。この問題に関する追加のメッセージがそのサブコンポーネントから発行されているので、そちらを参照してください。

(913) "*" option can cause compiler errors in some standard header files (ドライバ)
このオプションを使うと、標準ヘッダファイル内で使われている修飾子の一部が無効になり、エラーが発生します。この問題とその解消法の詳細は、本書内のこのオプションに関する項目を参照してください。

(915) no room for arguments (プリプロセッサ、パーサー、コード ジェネレータ、リンカ、Objtohex)
コード ジェネレータはこれ以上のメモリを割り当てる事ができません。

(917) argument too long (プリプロセッサ、パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(918) *: no match (プリプロセッサ、パーサー)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(919) * in chipinfo file “*” at line * (ドライバ)
chipinfo ファイル内で指定されているパラメータが無効です。

(920) empty chipinfo file (ドライバ、アセンブラ)
chipinfo ファイルを開く事はできましたが、内容が何もありません。コンパイラの再インストールを試してください。

(922) chip “*” not present in chipinfo file “*” (ドライバ)
選択されたデバイスは、コンパイラの chipinfo ファイル内に存在しません。このデバイスに対するサポートが利用可能かどうか、またはコンパイラ バージョンのアップグレードが必要かどうかは、Microchip 社の技術サポートにお問い合わせください。

(923) unknown suboption “*” (ドライバ)
このオプションに対して不明のサブオプションが指定されています。単純なスペルミスの可能性があります。そうではない場合、`--help` オプションを使って、使用可能なサブオプションを確認してください。

(924) missing argument to “*” option (ドライバ)
このオプションには追加の引数が必要です。このオプションの用法を確認してください。

(925) extraneous argument to “*” option (ドライバ)
このオプションに対して不要な引数が指定されています。このオプションの用法を確認してください。

(926) duplicate “*” option (ドライバ)
このオプションが複数回指定されています。このオプションは 1 度しか指定できません。

(928) bad “*” option value (ドライバ、アセンブラ)
このオプションの引数は 16 進整数である事が必要です。

(929) bad “*” option ranges (ドライバ)
このオプションには、レンジ指定形式のパラメータ(*`start_of_range-end_of_range`*)が必要です。

(930) bad “*” option specification (ドライバ)
このオプションに対するパラメータが正しく設定されていません。`--help` オプションを指定してドライバを実行するか、本書内のドライバの説明を参照して、このオプションの正しい用法を確認してください。

(931) command file not specified (ドライバ)
このアプリケーションに対するコマンドファイルがコマンドライン上の「@」または「<」の後で指定されていません。

(939) no file arguments (ドライバ)
コマンドライン上で入力ファイルが指定されずにドライバが呼び出されました。サードパーティ製 IDE を使ってビルドしている時にこのメッセージが表示された場合、コンパイルするソースファイルまたはリンクするオブジェクト ファイルを IDE が検証できなかったために、それらがコマンドラインから除外された可能性があります。

(940) *-bit * placed at * (Objtohex)
要求された計算の結果を示します。

(941) bad “*” assignment; USAGE: (Hexmate)**
Hexmate に対するオプションの指定が正しくないか不完全です。メッセージに示される指示に従ってオプションを正しく指定してください。

(942) unexpected character on line * of file “*” (Hexmate)
ファイルに無効な文字が含まれています。ファイルが破損している可能性があります。例えば、Intel HEX ファイルには 16 進値、コロン(:)、行書式の ASCII 表現のみ含まれている事が必要です。その他の文字が含まれていると、このエラーが発生します。

(944) data conflict at address *h between * and * (Hexmate)
Hexmate に対するソースが同じアドレスに異なるデータを保存するよう要求しています。データソースを他のデータに上書きするには、「+」指定子を使う必要があります。競合する 2 つのソースが同じソースである場合、そのソースはエラーを含んでいる可能性があります。

(945) range (*h to *h) contained an indeterminate value (Hexmate)
このレンジ計算には解決不可能な値が含まれます。これはレンジ計算結果自体をアドレスレンジ内に配置している場合に発生する可能性があります。

(948) result width must be between 1 and 4 bytes (Hexmate)
要求されたバイトサイズは無効です。チェックサム結果は 1〜4 バイト幅である事が必要です。-CKSUM オプションに対するパラメータを確認してください。

(949) start of range must be less than end of range (Hexmate)
-CKSUM オプションで終了アドレスより高い開始アドレスが指定されています。パラメータが不完全であるか指定順が正しくありません。

(951) start of fill range must be less than end of range (Hexmate)
-FILL オプションで終了アドレスより高い開始アドレスが指定されています。パラメータが不完全であるか指定順が正しくありません。

(953) unknown -HELP sub-option:* (Hexmate)
-HELP に無効なサブオプションが渡されました。サブオプションのスペルを確認してください。また、サブオプションを何も指定せずに-HELP を使う事で、全てのオプションを含むリストを表示できます。

(956) -SERIAL value must be between 1 and * bytes long (Hexmate)
指定されたシリアル番号はレンジ外です。このオプションが許容するバイト数で保存可能かどうか確認してください。

(958) too many input files specified; * file maximum (Hexmate)
引数として指定されたファイルの数が多すぎます。1 つのコマンドでマージするのではなく、複数回に分けてこれらのファイルをマージしてください。

(960) unexpected record type (*) on line * of “*” (Hexmate)
Intel HEX ファイルに無効なレコード型が含まれます。Intel HEX の書式仕様を参照して有効なレコード型を確認してください。

(962) forced data conflict at address *h between * and * (Hexmate)
Hexmate に対して複数のソースが異なるデータを同じアドレスに保存するよう要求しています。複数のソースが「+」指定子を使って同じアドレスにデータを保存しています。そこに実際に保存されたデータが期待した通りのデータではない可能性があります。

(963) range includes voids or unspecified memory locations (Hexmate)
ハッシュ (チェックサム) レンジ内のデータ内容にギャップが存在します。ハッシュ計算のアドレスレンジ内に未使用バイトのギャップが存在すると、実行時に計算されたハッシュ値がコンパイル時に計算されたハッシュ値と一致しない可能性があります。未使用位置に既知の値を書き込む事でこの問題を解消してください。

(964) unpaired nibble in -FILL value will be truncated (Hexmate)
-FILL オプションに対して指定された 16 進コードには不完全バイトが含まれています。不完全バイト (ニブル) は無視されます。

(965) -STRPACK option not yet implemented; option will be ignored (Hexmate)
このオプションは現在利用できず、無視されます。

(966) no END record for HEX file “*” (Hexmate)
Intel HEX ファイルに END 型のレコードが含まれていません。この HEX ファイルは不完全です。

(967) unused function definition “*” (from line *) (パーサー)
この static 関数は、コンパイル中のモジュール内で呼び出されていません。static 関数は他のモジュールから呼び出されないため、この警告はその関数がプログラム全体で全く使われないという事を意味します。原因として、この関数は冗長であるか、この関数を呼び出すはずであったコードがコンパイルから除外されたか、関数名のスペルミスが考えられます。

(968) unterminated string (アセンブラ)
文字列定数の閉じ側のクオートが脱落している可能性があります。

(969) end of string in format specifier (パーサー)
printf()形式関数の書式指定子が正しくありません。

(970) character not valid at this point in format specifier (パーサー)
printf()形式の書式指定子が無効な文字を含んでいます。

(971) type modifiers not valid with this format (パーサー)
型修飾子をこの書式で使う事はできません。

(972) only modifiers “h” and “l” valid with this format (パーサー)
この printf 書式指定子には修飾子として h (short) および l (long)のみが使えます。

(973) only modifier “l” valid with this format (パーサー)
この書式には修飾子として l (long) のみが使えます。

(974) type modifier already specified (パーサー)
この型修飾子は、この型で既に指定済みです。

(975) invalid format specifier or type modifier (パーサー)
printf 形式文字列内の書式指定子または修飾子は、この書式に対して使えません。

(976) field width not valid at this point (パーサー)
printf()形式書式指定子内のこの位置でフィールド幅を指定する事はできません。

(978) this identifier is already an enum tag (パーサー)
struct または union キーワードの後のこの識別子は、既に列挙型向けのタグとして指定されています。従って、この識別子は enum キーワードの後でしか使えません。
例:

```
enum IN {ONE=1, TWO};
struct IN {             /* oops -- IN is already defined */
    int a, b;
};
```

(979) this identifier is already a struct tag (パーサー)
union または enum キーワードの後のこの識別子は、既に構造体向けのタグとして指定されています。従って、この識別子は struct キーワードの後でしか使えません。
例:

```
struct IN {
    int a, b;
};
enum IN {ONE=1, TWO}; /* oops -- IN is already defined */
```

(980) this identifier is already a union tag (パーサー)
struct または enum キーワードの後のこの識別子は、既に共用体向けのタグとして指定されています。従って、この識別子は union キーワードの後でしか使えません。
例:

```
union IN {
    int a, b;
};
enum IN {ONE=1, TWO}; /* oops -- IN is already defined */
```

(981) pointer required (パーサー)
この位置でポインタが必要です。
例:

```
struct DATA data;
data->a = 9;      /* data is a structure, not a pointer to a structure */
```

(982) unknown op “*” in nxtuse() (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(983) storage class redeclared (パーサー)
以前に static として宣言された変数が extern として再宣言されています。

(984) type redeclared (パーサー)
関数またはオブジェクトの型が再宣言されています。このエラーは、2 つの不整合な宣言が存在する場合、または暗黙的な宣言の後に不整合な宣言が行われた場合に発生します。
例:

```
int a;
char a;  /* oops -- what is the correct type?*/
```

(985) qualifiers redeclared (パーサー)
関数または変数が異なる修飾子を使って再宣言されています。

(986) enum member redeclared (パーサー)
列挙体のメンバーが異なる値で複数回定義されています。そのメンバーが同じリスト内で 2 回使われていないか、あるいはそのメンバーの名前が複数の enum リスト内で使われていないか確認してください。

(987) arguments redeclared (パーサー)
この関数に渡されたパラメータのデータ型はその関数のプロトタイプと一致しません。

(988) number of arguments redeclared (パーサー)
この関数宣言内の引数の数は、同じ関数の以前の宣言と一致しません。

(989) module has code below file base of *h (リンカ)
このモジュールは指定されたアドレスより下のコードを格納していますが、バイナリ出力ファイルを生成してこのアドレスに割り当てるよう指定するために-C オプションが使われています。これは、このモジュールからのコードをそのファイルの先頭より前に配置する必要がある事を意味します。アセンブリ ファイル内で psect ディレクティブが抜けていないか確認してください。

(990) modulus by zero in #if; zero result assumed (プリプロセッサ)
#if 内の剰余演算に 0 による除算が含まれています。その結果は 0 であると見なされました。
例:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
#define INTERESTING
#endif
```

(991) integer expression required (パーサー)
enum 宣言内でメンバーに値を代入する事はできますが、代入可能な式は int 型として評価可である事が必要です。
例:

```
enum {one = 1, two, about_three = 3.12};
/* no non-int values allowed */
```

(992) can't find op (アセンブラ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(993) some command-line options are disabled (ドライバ)
コンパイラはデモモードで動作しています。一部のコマンドライン オプションは無効にされています。

(994) some command-line options are disabled and compilation is delayed (ドライバ)
コンパイラはデモモードで動作しています。一部のコマンドライン オプションは無効にされており、コンパイルは低速です。

(995) some command-line options are disabled; code size is limited to 16kB, compilation is delayed (ドライバ)
コンパイラはデモモードで動作しています。 一部のコマンドライン オプションは無効にされており、コンパイルは低速であり、最大許容コードサイズは 16KB 以下に制限されます。

10.3 メッセージ 1000～1499

(1015) missing "*" specification in chipinfo file "*" at line * (ドライバ)
デバイスに対してこの属性が定義されていません。

(1016) missing argument* to "*" specification in chipinfo file "*" at line * (ドライバ)
この属性の値が chipinfo ファイル内で指定されていません。

(1017) extraneous argument* to "*" specification in chipinfo file "*" at line * (ドライバ)
chipinfo ファイル内で処理指定に対する引数の数が多すぎます。

(1018) illegal number of "*" specification* (* found; * expected) in chipinfo file "*" at line * (ドライバ)
この属性は特定回数指定される必要がありますが、このデバイスに対して必要な回数指定されていません。

(1019) duplicate "*" specification in chipinfo file "*" at line * (ドライバ)
この属性は 1 度だけ定義される必要がありますが、このデバイスに対して複数回定義されています。

(1020) unknown attribute "*" in chipinfo file "*" at line * (ドライバ)
chipinfo ファイルは、このコンパイラ バージョンでは認識できない属性を含んでいます。このコンパイラの異なるバージョン向けの chipinfo ファイルまたはドライバが誤って使われていないか確認してください。

(1021) syntax error reading “*” value in chipinfo file “*” at line * (ドライバ)
この値は chipinfo ファイル内でこのデバイス向けに正しく定義されていません。ユーザがこのファイルを変更する場合、注意が必要です。このファイルの冒頭に記載されているコメントを参照して正しい値を定義してください。

(1022) syntax error reading “*” range in chipinfo file “*” at line * (ドライバ)
このレンジは chipinfo ファイル内でこのデバイス向けに正しく定義されていません。ユーザがこのファイルを変更する場合、注意が必要です。このファイルの冒頭に記載されているコメントを参照して正しい値を定義してください。

(1024) syntax error in chipinfo file “*” at line * (ドライバ)
chipinfo ファイル内のこの行に構文エラーが存在します。

(1025) unknown architecture in chipinfo file “*” at line * (ドライバ)
chipinfo ファイル内のこの行でコンパイラには認識不可能なアーキテクチャが定義されています。

(1026) missing architecture in chipinfo file “*” at line * (アセンブラ)
chipinfo ファイルのデバイス セクション内に ARCH 値がありません。デバイスのアーキテクチャを指定する必要があります。chipinfo ファイルを変更していないのにこのエラーが発生する場合、Microchip 社技術サポートにお問い合わせください。

(1027) activation was successful (ドライバ)
コンパイラのアクティベーションに成功しました。

(1028) activation was not successful - error code (*) (ドライバ)
コンパイラのアクティベーションに失敗しました。

(1029) compiler not installed correctly - error code (*) (ドライバ)
コンパイラのアクティベーション情報が見つかりませんでした。このコンパイラは実行できません。コンパイラが正しく完全にインストールされていない可能性があります。メッセージに示されたエラーコードが原因究明に役立つ場合があります。このエラーに関して Microchip 社にお問い合わせになる際は、このエラーコードをお知らせください。

(1030) Hexmate - Intel HEX editing utility (Build 1.%i) (Hexmate)
実行中の Hexmate のバージョン番号を示します。

(1031) USAGE:* [input1.HEX] [input2.HEX]...[inputN.HEX] [options] (Hexmate)
Hexmate の使用法を示します。

(1032) use –HELP=<option> for usage of these command line options (Hexmate)
-HELP オプションにオプション名を渡すと、そのオプションに関する詳細情報が表示されます。

(1033) available command-line options:(Hexmate)
このメッセージの下に、このアプリケーション向けに利用可能なオプションのリストが表示されます。

(1034) type “*” for available options (Hexmate)
ヘルプ情報の取得方法を示します。これにより、このアプリケーション向けに利用可能なオプションまたはそれらの用法に関する追加の情報が得られます。

(1035) bad argument count (*) (パーサー)
関数に対する引数の数が不適切です。これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1036) bad “*” optional header length (0x* expected) (Cromwell)
この COFF ファイル内のオプションヘッダの長さが正しくありません。

(1037) short read on * (Cromwell)
このデータの読み出しは、指定された長さを読み出す前に終了しました。

(1038) string table length too short (Cromwell)
COFF 文字列テーブルに対して指定された長さが短すぎます。

(1039) inconsistent symbol count (Cromwell)
シンボルテーブル内のシンボルの数が COFF ヘッダ内で指定された数を超えています。

(1040) bad : record 0x*, 0x* (Cromwell)
レコードの型とレコードの値が適合していません。

(1041) short record (Cromwell)
ファイルの読み出し中に、レコードの 1 つが指定された長さに達する前に終了しました。

(1042) unknown * record type 0x* (Cromwell)
このレコードの型指定子は、このファイル書式向けに有効などの型にも一致しません。

(1043) unknown optional header (Cromwell)
この Microchip COFF ファイルの読み出し時に、ファイルヘッダ内のオプションヘッダの長さが不適正でした。

(1044) end of file encountered (Cromwell、Linker)
ファイルの終端に達しましたが、データが足りません。この入力ファイルが切り詰められていないか確認してください。

(1045) short read on block of * bytes (Cromwell)
UBROF レコードからバイトデータのブロックを読み出しましたが、必要な長さを読み出す前にブロックが終了しました。

(1046) short string read (Cromwell)
UBROF レコードから文字列を読み出しましたが、指定された長さを読み出す前に文字列が終了しました。

(1047) bad type byte for UBROF file (Cromwell)
この UBROF ファイルは不正なレコードで始まっています。

(1048) bad time/date stamp (Cromwell)
この UBROF ファイルの時刻/日付スタンプが正しくありません。

(1049) wrong CRC on 0x* bytes; should be * (Cromwell)
この UBROF ファイル内の終了レコードの CRC 値が不適合です。

(1050) bad date in 0x52 record (Cromwell)
この UBROF ファイル内のデバッグレコードは日付が不正なコンポーネントを含んでいます。

(1051) bad date in 0x01 record (Cromwell)
この UBROF ファイル内のプログラム開始レコードまたはセグメント レコードは日付が不正なコンポーネントを含んでいます。

(1052) unknown record type (Cromwell)
この URBOF ファイルの読み出し時にレコードの型を判別できませんでした。

(1053) additional RAM ranges larger than bank size (ドライバ)
要求された追加の RAM ブロックはバンクのサイズを超えています。そのブロックを複数に分割してバンク境界をまたがないようにする必要があります。

(1054) additional RAM range out of bounds (ドライバ)
カスタム RAM 設定で定義された RAM メモリレンジが制限を超えています。

(1055) RAM range out of bounds (*) (ドライバ)
デバイス コンフィグレーション ファイルまたはカスタム設定により定義された RAM メモリレンジが制限を超えています。

(1056) unknown chip architecture (ドライバ)
コンパイラは非サポートまたは無効なアーキテクチャのデバイス向けにコンパイルを試みています。

(1058) assertion (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1059) rewrite loop (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1081) static initialization of persistent variable “*” (パーサー、コード ジェネレータ)
`persistent` 変数に初期値が代入されました。初期値はコンパイラの起動コードの実行中に変数に代入されますが、`persistent` 修飾子はコンパイラの起動コードによってこの変数が変更されない事を要求するため、これは矛盾する動作です。

(1082) size of initialized array element is zero (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1088) function pointer “*” is used but never assigned a value (コード ジェネレータ)
関数ポインタにアドレスが設定されないまま関数を呼び出しています。
例:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to?*/
```

(1089) recursive function call to “*” (コード ジェネレータ)
この関数に対する再帰呼び出しが見つかりました。この呼び出しは、直接的または間接的(関数ポインタを使用)にこの関数を呼び出しているか、この関数をコールグラフに含む別の関数を呼び出している可能性があります。

(1090) variable “*” is not used (コード ジェネレータ)
この変数は宣言されていますが、プログラムによって使われていません。プログラムからこの変数を削除できる可能性があります。

(1091) main function “*” not defined (コード ジェネレータ)
`main` 関数が定義されていません。全ての C プログラムには `main()`関数が必要です。

(1094) bad derived type (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1095) bad call to typeSub() (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1096) type should be unqualified (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1097) unknown type string “*” (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

© 2022 Microchip Technology Inc. and its subsidiaries

(1098) conflicting declarations for variable “*” (*:*)(パーサー、コード ジェネレータ)
1 つの変数に対して型情報が異なる複数の宣言が検出されたか、変数の宣言と定義の間で型情報が一致していない可能性があります。
例:

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1118) bad string “*” in getexpr(J) (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1119) bad string “*” in getexpr(LRN) (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1121) expression error (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1137) match() error:* (コード ジェネレータ)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1157) W register must be W9 (アセンブラ)
この位置で必要なワーキング レジスタは W9 ですが、他のワーキング レジスタが選択されました。

(1159) W register must be W11 (アセンブラ)
この位置で必要なワーキング レジスタは W11 ですが、他のワーキング レジスタが選択されました。

(1178) the “*” option has been removed and has no effect (ドライバ)
このオプションは、コンパイラのこのバージョンでサポートされないため無視されました。コンパイラの`--help` オプションを使うかマニュアルを参照して、代わりに使えるオプションを見つけてください。

(1179) interrupt level for function “*” cannot exceed * (コード ジェネレータ)
この関数の割り込みレベルが高すぎます。各関数には固有の割り込みレベルが割り当てられます。このレベルは、コールグラフと再入呼び出しされた関数の解析時に考慮されます。`interrupt_level` プラグマを使っている場合、そこで指定されている値を確認してください。

(1180) directory “*” does not exist (ドライバ)
セットアップ オプションで指定されているディレクトリが存在しません。そのディレクトリを作成した後にやり直してください。

(1182) near variables must be global or static (コード ジェネレータ)
`near` として修飾された変数は、`static` としても修飾されているか、グローバルである事が必要です。`auto` 変数を `near` として修飾する事はできません。

(1183) invalid version number (アクティベーション)
アクティベーションを試みましたが、指定されたシリアル番号に対応するバージョン番号が Microchip 社のアクティベーション サーバ データベース上で見つかりませんでした。

(1184) activation limit reached (Activation)
指定されたシリアル番号のアクティベーション数がライセンスで許可されている最大数を超えています。

(1185) invalid serial number (アクティベーション)
アクティベーションを試みましたが、一致するバージョン番号が Microchip 社のアクティベーション サーバ データベース上で見つかりませんでした。

(1186) license has expired (ドライバ)
このコンパイラのライセンス期限は終了しています。

(1187) invalid activation request (ドライバ)
コンパイラは正しくアクティベートされていません。

(1188) network error * (アクティベーション)
コンパイラのアクティベーション ソフトウェアは、ネットワーク経由で Microchip 社のアクティベーション サーバに接続できませんでした。

(1190) FAE license only - not for use in commercial applications (ドライバ)
このコンパイラは FAE ライセンスを使ってアクティベートされました。このコンパイラを商用アプリケーションの開発向けに使う事は承認されていません。

(1191) licensed for educational use only (ドライバ)
このコンパイラは教育機関向けライセンスを使ってアクティベートされました。このライセンスは教育機関向けにのみ提供され、このコンパイラを商用アプリケーションの開発向けに使う事は承認されていません。

(1192) licensed for evaluation purposes only (ドライバ)
このコンパイラは評価用ライセンスを使ってアクティベートされました。

(1193) this license will expire on * (ドライバ)
コンパイラは期限付きの評価用にインストールされています。この日付で評価期間は終了します。

(1195) invalid syntax for “*” option (ドライバ)
追加のパラメータを指定可能なコマンドライン オプションに対して不適切または不完全なパラメータが渡されました。例えば、オプションに対して 2 つの整数パラメータを指定する必要がある時に、パラメータの 1 つに文字列が渡された場合、または 1 つのパラメータだけが渡された場合、このエラーが発生する可能性があります。

(1198) too many “*” specifications; * maximum (Hexmate)
指定されたオプションの数が多すぎます。可能であれば、複数のコマンドラインに分けてこれらのオプションを指定してください。

(1199) compiler has not been activated (ドライバ)
このコンパイラの評価期間は終了しました。有効なシリアル番号を使ってアクティベートするまで、このコンパイラは使えません。Microchip 社からこのソフトウェアを購入してシリアル番号を取得してください。

(1200) Found %0*lXh at address *h (Hexmate)
`-FIND` オプションで指定されたコードシーケンスがこのアドレスで見つかりました。

(1201) all FIND/REPLACE code specifications must be of equal width (Hexmate)
このオプションの検索、置換、マスク属性は全て同じバイト幅である事が必要です。このオプションに渡したパラメータを確認してください。例えば、1234h (2 バイト) の検索で FFh (1 バイト) のマスクを指定するとこのエラーが発生します。このマスクを 00FFh (2 バイト) として指定すればエラーを回避できます。

(1202) unknown format requested in -FORMAT:* (Hexmate)
不明または未サポートの INHX フォーマットが要求されました。サポートされる INHX フォーマットをマニュアルで確認してください。

(1203) unpaired nibble in * value will be truncated (Hexmate)
このオプションに対するデータが完全なバイトとして渡されませんでした。データが不完全であるか、先頭の 0 が省略された可能性があります。例えば、値として Fh を指定した場合、有意ビット数は 4 しかありません(完全な 1 バイトではありません)。値として 0Fh を指定する事で完全な 1 バイト(有意ビット数は 8)を渡す事ができます。

(1204) * value must be between 1 and * bytes long (Hexmate)
このオプションに渡されたデータの長さが不適正です(このオプションが要求するレンジ内ではありません)。

(1205) using the configuration file *; you can override this with the environment variable HTC_XML (ドライバ)
これはコンパイラのセットアップ時に選択されたコンパイラ コンフィグレーション ファイルです。 このファイルは、HTC_XML 環境変数を使って変更できます。 このファイルは、コンパイラがインストールされている場所を特定するために使われます。

(1207) some of the command line options you are using are now obsolete (ドライバ)
ドライバに渡されたコマンドライン オプションにこのコンパイラ バージョンで廃止済みのオプションが含まれていますが、オプション削除猶予期間中はそれらのオプションもドライバによって処理されます。

(1208) use –help option or refer to the user manual for option details (ドライバ)
廃止済みのオプションが検出されました。--help を使うかマニュアルを参照して、代わりに使えるオプションを見つけてください。

(1209) An old MPLAB tool suite plug-in was detected.(ドライバ)
ドライバに渡されたオプションは、かつて Microchip MPLAB v8 IDE からこのコンパイラの旧バージョンに渡されていた旧式のオプションです。これらのオプションは既に廃止されていますが、現在でも認識されます。IDE に最新の Microchip オプション プラグインをインストールする事を推奨します。

(1210) Visit the Microchip website (www.microchip.com) for a possible upgrade (ドライバ)
弊社のウェブサイトにアクセスし、コンパイラ メッセージに表示された問題に対処するためのアップグレードが入手可能かどうか確認してください。当ウェブサイト内の MPLAB XC8 C コンパイラのページに移動し、バージョン アップグレード用のダウンロード ファイルを探してください。現在使用中のバージョンが最新である場合、Microchip 社技術サポートにお問い合わせください。

(1212) Found * (%0*lXh) at address *h (Hexmate)
-FIND オプションで指定されたコードシーケンスがこのアドレスで見つかりました。

(1213) duplicate ARCH for * in chipinfo file at line * (アセンブラ、ドライバ)
chipinfo ファイルのデバイス セクション内に複数の ARCH 値が存在します。ARCH 値は 1 つだけである事が必要です。chipinfo ファイルを編集していないにも関わらずこのエラーが発生する場合、Microchip 社の技術サポートに詳細をお問い合わせください。

(1218) can’t create cross reference file * (アセンブラ)
アセンブラは相互参照ファイルの作成に失敗しました。ファイルのパスが正しいか確認してください。

(1228) unable to locate installation directory (ドライバ)
コンパイラはインストール ディレクトリを見つける事ができませんでした。

(1230) dereferencing uninitialized pointer “*” (コード ジェネレータ)
値が未割り当てのポインタが逆参照されています。これにより実行時の動作に問題が生じる可能性があります。

(1235) unknown keyword * (ドライバ)
USB 記述ファイル内のトークンが認識されませんでした。

(1236) invalid argument to *:* (ドライバ)
オプションに対して無効な引数が渡されました。オプションの用法、構文、パラメータの許容レンジを確認してください。

(1237) endpoint 0 is pre-defined (ドライバ)
USB ファイル内でエンドポイント 0 の定義が試みられました。

(1238) FNALIGN failure on * (リンカ)
2 つの関数の auto/パラメータ ブロックが FNALIGN ディレクティブを使って整列されていますが、一方の関数が他方の関数を呼び出しています。このため、整列はされません。これは、1 つの関数ポインタに各関数のアドレスが割り当てられ、その中の 1 つの関数が他の関数を呼び出す場合に発生します。
例:

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* !*/
int (*ip)(int);
ip = one;
ip(23);
ip = two;      /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets (コード ジェネレータ)
関数ポインタにアドレスが設定されないまま関数を呼び出しています。
例:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to?*/
```

(1240) unknown algorithm type (%i) (ドライバ)
-E*file* または-E+*file* オプションで指定されたエラーファイルを開く事ができませんでした。ファイルまたはディレクトリが正しく指定され、ファイルが読み出し専用に設定されている事を確認してください。

(1241) bad start address in * (ドライバ)
--CHECKSUM オプションで指定されたレンジ開始アドレスを読み出せませんでした。この値は 16 進値である事が必要です。

(1242) bad end address in * (ドライバ)
--CHECKSUM オプションで指定されたレンジ終了アドレスを読み出せませんでした。この値は 16 進値である事が必要です。

(1243) bad destination address in * (ドライバ)
--CHECKSUM オプションで指定されたデスティネーション アドレスを読み出せませんでした。この値は 16 進値である事が必要です。

(1245) value greater than zero required for * (Hexmate)
-FIND オプションの *Align* オペランドは正の値である事が必要です。

(1246) no RAM defined for variable placement (コード ジェネレータ)
バンク型 RAM メモリに対応するメモリが指定されていません。

(1247) no access RAM defined for variable placement (コード ジェネレータ)
アクセスバンク メモリに対応するメモリが指定されていません。

(1248) symbol (*) encountered with undefined type size (コード ジェネレータ)
コード ジェネレータに対して変数の配置が要求されましたが、その変数のサイズが不明です。これは内部コンパイラ エラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1250) could not find space (* byte*) for variable * (コード ジェネレータ)
コード ジェネレータは、この変数向けにバンク型 RAM 内の空き領域を見つける事ができませんでした。

(1253) could not find space (* byte*) for auto/param block (コード ジェネレータ)
コード ジェネレータは、auto およびパラメータ変数を保持する psect 向けに RAM 内の空き領域を見つける事ができませんでした。

(1254) could not find space (* byte*) for data block (コード ジェネレータ)
コード ジェネレータは、初期化変数を保持するデータ psect 向けに RAM 内の空き領域を見つける事ができませんでした。

(1255) conflicting paths for output directory (ドライバ)
-O または--OUTDIR オプションによってコンパイラに渡された出力ディレクトリのパスは矛盾しています。
例:

```
--outdir=../../   -o../main.HEX
```

(1256) undefined symbol “*” treated as HEX constant (アセンブラ)
シンボルまたは 16 進値のどちらとも解釈可能なトークンが定義済みのどのシンボルとも一致しないため、16 進値として解釈されました。16 進値である事を明確に示すため、値の先頭に 0 を追加するか、0x 等の基数指定子を使ってください。
例:

```
MOV  a, F7h ; is this the symbol F7h, or the HEX number 0xF7?
```

(1257) local variable “*” is used but never given a value (コード ジェネレータ)
定義済みの auto 変数が式内で使われていますが、最初に使われる前に C コード内で値が代入されていません。auto 変数は起動時にクリアもされず初期値も定義されません。
例:

```
int main(void)
    { double src,
    out;
    out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function “*” (コード ジェネレータ)
コード ジェネレータによるコールツリー解析結果は、ハードウェア スタックがオーバーフローする可能性を示しています。これはガイドラインに過ぎませんが割り込み、アセンブラ オプティマイザ、プログラム構造によってはスタックが実際にオーバーフローする恐れがあります。スタック使用量は C プログラムに基づいて推測され、アセンブリコードによるコールツリーを含んでいません。

(1259) can’t optimize for both speed and space (ドライバ)
最適化オプションとして速度重視とサイズ重視の両方がドライバに対して指定されています。
例:

```
--opt=speed,space
```

(1260) macro “*” redefined (アセンブラ)
同一名のマクロに対して複数の定義が見つかりました。
例:

```
MACRO fin
    ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
    reti
ENDM
```

(1261) string constant required (アセンブラ)
DS または DSU ディレクティブには文字列引数が必要です。
例:

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1262) object “*” lies outside available * space (コード ジェネレータ)
絶対アドレス変数がターゲット デバイス向けに定義されたメモリ領域の外に配置されました。
例:

```
int data   at(0x800);     /* oops -- is this the correct address? */
```

(1264) unsafe pointer conversion (コード ジェネレータ)
構造体へのポインタが、定義の異なる構造体へのアドレスで初期化されています。
例:

```
struct ONE {
    unsigned a;
    long b;          /* !*/
```

```
} one;
struct TWO {
    unsigned a;
    unsigned b;      /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;    /* oops -- was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * into * bytes at 0x* (リンカ)
詳細はエラーメッセージ(1356)の説明を参照してください。

(1268) fixup overflow storing 0x* in * bytes at * (リンカ)
詳細はエラーメッセージ(1356)の説明を参照してください。

(1273) Omniscient Code Generation not available in Free mode (ドライバ)
Free (無償)モードのコンパイラでは先進機能を使う事はできません。

(1275) the qualifier “*” is only applicable to functions (パーサー)
関数定義にしか使えない修飾子が変数定義向けに使われています。

```
interrupt int dacResult; /* oops --
the interrupt qualifier can only be used with functions */
```

(1276) buffer overflow in DWARF location list (Cromwell)
ELF/DWARF デバッグファイル向けのバッファがオーバーフローしました。Microchip 社技術サポートに詳細をお問い合わせください。

(1278) omitting “*” which does not have a location (Cromwell)
保存位置を持たないこれらの変数はデバッグ出力から省略されます。Microchip 社技術サポートに詳細をお問い合わせください。

(1284) malformed mapfile while generating summary:CLASS expected but not found (ドライバ)
メモリ使用量サマリを生成するために読み出されたマップファイルの書式が正しくありません。マップファイルは編集されたか破損している可能性があります。そうではない場合、コンパイラ エラーの可能性があるため、Microchip 社の技術サポートに詳細をお問い合わせください。

(1285) malformed mapfile while generating summary: no name at position * (ドライバ)
メモリ使用量サマリを生成するために読み出されたマップファイルの書式が正しくありません。マップファイルは編集されたか破損している可能性があります。そうではない場合、コンパイラ エラーの可能性があるため、Microchip 社の技術サポートに詳細をお問い合わせください。

(1286) malformed mapfile while generating summary: no link address at position * (ドライバ)
メモリ使用量サマリを生成するために読み出されたマップファイルの書式が正しくありません。マップファイルは編集されたか破損している可能性があります。そうではない場合、コンパイラ エラーの可能性があるため、Microchip 社の技術サポートに詳細をお問い合わせください。

(1287) malformed mapfile while generating summary: no load address at position * (ドライバ)
メモリ使用量サマリを生成するために読み出されたマップファイルの書式が正しくありません。マップファイルは編集されたか破損している可能性があります。そうではない場合、コンパイラ エラーの可能性があるため、Microchip 社の技術サポートに詳細をお問い合わせください。

(1288) malformed mapfile while generating summary: no length at position * (ドライバ)
メモリ使用量サマリを生成するために読み出されたマップファイルの書式が正しくありません。マップファイルは編集されたか破損している可能性があります。そうではない場合、コンパイラ エラーの可能性があるため、Microchip 社の技術サポートに詳細をお問い合わせください。

(1289) line range limit exceeded, possibly affecting ability to debug code (Cromwell)
設定された制限を超える長さのアセンブリコード出力が C 命令文から生成されました。これは、CROMWELL によって生成されたデバッグ情報が正確ではない可能性がある事を意味します。この警告はコード自体の潜在的エラーを示す物ではありません。

(1290) buffer overflow in DWARF debugging information entry (Cromwell)
ELF/DWARF デバッグファイル向けのバッファがオーバーフローしました。Microchip 社技術サポートに詳細をお問い合わせください。

(1291) bad ELF string table index (Cromwell)
CROMWELL に渡された ELF ファイルは不適正であるため使えません。

(1292) malformed define in .SDB file * (Cromwell)
CROMWELL に渡された SDB ファイルは不適正であるため使えません。

(1293) couldn't find type for "*" in DWARF debugging information entry (Cromwell)
CROMWELL に渡された SDB ファイルからシンボルの型を特定する事ができませんでした。マップファイルは編集されたか破損している可能性があります。そうではない場合、コンパイラ エラーの可能性があるため、Microchip 社の技術サポートに詳細をお問い合わせください。

(1294) there is only one day left until this license expires (ドライバ)
コンパイラは現在デモモードで動作しています。コンパイラの評価用ライセンスはあと 1 日で期限切れとなり、PRO モードで使う事ができなくなります。Free(無償)モードで使い続ける事はできますが、出力バイナリコードのサイズは大きくなります。

(1295) there are * days left until this license will expire (ドライバ)
コンパイラは現在デモモードで動作しています。このメッセージはコンパイラの評価用ライセンスの期限を示しています。期限が切れると PRO モードで使う事ができなくなります。Free(無償)モードで使い続ける事はできますが、出力バイナリコードのサイズは大きくなります。

(1296) source file "*" conflicts with "*" (ドライバ)
コンパイラは同一ベース名を持つ複数のソースファイルに遭遇しました。これは、それらのファイルが異なるディレクトリに保存されている場合にのみ発生します。コンパイラと IDE は、ソースファイルのベース名に基づいて中間生成ファイルの名前を決定し、それらを常に同じディレクトリに保存します。このため、同一ベース名のソースファイルが複数存在すると、このエラーが発生します。全てのソースファイルのベース名が異なっているか確認してください。

(1297) option * not available in Free mode (ドライバ)
一部のオプションは、Free(無償)モードのコンパイラでは使えません。Free モードで使えなくなるのは一般的にコンパイラの実行方法に関連するオプション(例: `--getoption`、`--setoption`)であり、コード生成に関連するコンパイラ機能は影響を受けません。

(1298) use of * outside macros is illegal (アセンブラ)
一部のアセンブラ ディレクティブ(`EXITM` 等)は、マクロ定義の中でのみ使用可能です。

(1299) non-standard modifier "*" - use "*" instead (パーサー)
非標準の printf プレースホルダ修飾子が使われています。このメッセージが推奨している修飾子を使ってください。例えば、値を `char` 型として出力するよう指定する場合、`b` の代わりに標準の `hh` 修飾子を使う事を推奨します。

(1300) maximum number of program classes reached; some classes may be excluded from debugging information (Cromwell)
CROMWELL にはコマンドラインからクラス名のリストが渡されます。リストに含まれるクラス名の数が多すぎた場合、一部のクラス名だけが使われ、デバッグ情報の精度が低下する可能性があります。

(1301) invalid ELF section header; skipping (Cromwell)
CROMWELL は ELF セクションヘッダ内に無効なセクションを見つけました。このセクションはスキップされます。

(1302) could not find valid ELF output extension for this device (Cromwell)
拡張子がターゲットのデバイスファミリ向けではありません。

(1303) invalid variable location detected:* - * (Cromwell)
SDB ファイルからシンボルの位置を特定する事ができませんでした。

(1304) unknown register name:“*” (Cromwell)
SDB ファイル内でこのシンボルの位置として指定されているレジスタの名前が認識されませんでした。

(1305) inconsistent storage class for variable:“*” (Cromwell)
SDB ファイル内でこのシンボル向けに指定されている記憶域クラスが認識されませんでした。

(1306) inconsistent size (* vs *) for variable:“*” (Cromwell)
SDB ファイル内で指定されているこのシンボルのサイズは、その型のサイズと一致しません。

(1307) psect * truncated to * bytes (ドライバ)
スタックまたはヒープ向けの psect に要求されたサイズを割り当てる事ができませんでした。この psect は利用可能なメモリ空間に収まるよう切り詰められます。

(1308) missing/conflicting interrupts sub-option; defaulting to “*” (ドライバ)
`--INTERRUPT` オプションに対するサブオプションが何も指定されていないか正しく指定されていません。
例:

```
--INTERRUPTS=single,multi
```

上記では、`--INTERRUPTS` に対してシングルベクタとマルチベクタの両方が指定されています。

(1309) ignoring invalid runtime * sub-option (*) using default (ドライバ)
`--RUNTIME` オプションに対するサブオプションが正しく指定されていません。
例:

```
--RUNTIME=default,speed:0y1234
```

上記では、0y1234 を 0x1234 に訂正する必要があります。

(1310) specified speed (*Hz) exceeds max operating frequency (*Hz); defaulting to *Hz (ドライバ)
`--RUNTIME` オプションの `perform` サブオプションで指定された周波数は選択されているデバイスに対して高すぎます。
例:

```
--RUNTIME=default,speed:0xffffffff
```

値が大きすぎます。

(1311) missing configuration setting for config word *; using default (ドライバ)
ソースコード内でこのコンフィグレーション ワードに設定値が提供されなかったため、既定値が使われます。

(1312) conflicting runtime perform sub-option and configuration word settings; assuming *Hz (ドライバ)
`--RUNTIME` オプションの `perform` サブオプションで指定された値はコンフィグレーション ワードの設定と矛盾しているため、既定値周波が選択されました。

(1313) * sub-options (“*”) ignored (ドライバ)
このサブオプションには引数が不要であり、指定された引数は無視されます。
例:

```
--OUTPUT=intel:8
```

上記では「`:8`」は不要です。

(1314) illegal action in memory allocation (コード ジェネレータ)
これは内部エラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1315) undefined or empty class used to link psect * (リンカ)
クラスに割り当てられたアドレスレンジ内に psect を配置するよう要求されましたが、そのクラスは未定義であるかメモリレンジが何も割り当てられていません。

(1316) attribute “*” ignored (パーサー)
この属性は有効ですが、パーサーはこの属性を実装していません。この属性はパーサーによって無視され、効果を持ちません。Microchip 社技術サポートに詳細をお問い合わせください。

(1317) missing argument to attribute “*” (パーサー)
この属性に必要な引数が指定されていません。Microchip 社技術サポートに詳細をお問い合わせください。

(1318) invalid argument to attribute “*” (パーサー)
この属性に対する引数が正しく指定されていません。Microchip 社技術サポートに詳細をお問い合わせください。

(1319) invalid type “*” for attribute “*” (パーサー)
パーサーに無効なオプションが渡されました。Microchip 社技術サポートに詳細をお問い合わせください。

(1320) attribute “*” already exists (パーサー)
同じ属性オプションが複数回パーサーに渡されました。Microchip 社技術サポートに詳細をお問い合わせください。

(1321) bad attribute -T option “%s” (パーサー)
パーサーに渡されたこの属性オプションは正しく指定されていません。Microchip 社技術サポートに詳細をお問い合わせください。

(1322) unknown qualifier “%s” given to -T (パーサー)
属性オプション内で不明の修飾子が指定されています。Microchip 社技術サポートに詳細をお問い合わせください。

(1323) attribute expected (パーサー)

`__attribute__`ディレクティブが使われていますが、属性が何も指定されていません。
例:

```
int rv (int a)  __attribute__  (())  /* oops -- what is the attribute?*/
```

(1324) qualifier “*” ignored (パーサー)
この修飾子は有効ですが、一部のコンパイラまたはターゲット デバイスでは実装できません。これらの修飾子は無視されます。

(1342) whitespace after “\” (プリプロセッサ)
バックスラッシュ文字と改行文字の間の空白類文字は無視されます。

(1343) hexfile data at address 0x* (0x*) overwritten with 0x* (Objtohex)
このアドレスは追加データによって上書きされようとしています。これは、複数のコードセクションが同じアドレスに配置されてしまう事を意味します。

(1346) can’t find 0x* words for psect “*” in segment “*” (largest unused contiguous range 0x%lX) (リンカ)
メッセージ(491)も参照してください。このメッセージには、リンカが利用できる未使用で最大の連続したブロックのレンジも示されます。このメッセージは、psect を格納可能な 1 つの連続した空間が存在しない場合に発行されます。メモリにバンクまたはページが存在する場合、一般的に、連続した空き空間の最大サイズは未使用空間の総サイズよりも大幅に小さくなります。

(1347) can’t find 0x* words (0x* withtotal) for psect “*” in segment “*” (largest unused contiguous range 0x%lX) (リンカ)
メッセージ(593)も参照してください。このメッセージには、リンカが利用できる未使用で最大の連続したブロックのレンジも示されます。このメッセージは、psect を格納可能な 1 つの連続した空間が存在しない場合に発行されます。メモリにバンクまたはページが存在する場合、一般的に、連続した空き空間の最大サイズは未使用空間の総サイズよりも大幅に小さくなります。

(1348) enum tag “*” redefined (from *:*)(パーサー)
同じ名前を持つ複数の enum タグが定義されています。このメッセージには、以前の定義も示されます。
例:

```
enum VALS { ONE=1, TWO, THREE };
enum VALS { NINE=9, TEN }; /* oops -- is VALS the right tag name?*/
```

(1349) initialization of absolute variable "*" in RAM is not supported (コード ジェネレータ)
__at(*address*)を使って定義された絶対アドレス変数は、定義時に初期値を代入する事はできません。プログラム内の適切な位置でこの変数に値を代入する必要があります。
例:

```
int foobar __at(0x20) = 0x55;  /* oops --
 you cannot assign a value to an absolute variable */
```

(1350) pointer operands to “-” must reference the same array (コード ジェネレータ)
ANSI 規格に準拠して 2 つのアドレスの間で減算を行う場合、それらは同じオブジェクトのアドレスである事が必要です。
例:

```
int * ip;
int fred, buf[20];
ip = &buf[0] - &fred; /* oops --
 second operand must be an address of a "buf" element */
```

(1352) truncation of operand value (0x*) to * bits (アセンブラ)
アセンブラ命令に対するオペランドが大きすぎるため切り詰められました。
例:

```
movlw  0x321  ; oops -- is this the right value?
```

(1354) ignoring configuration setting for unimplemented word * (ドライバ)
ターゲット デバイス上に存在しないワードに対してコンフィグレーション ワードの設定が指定されました。
例:

```
__CONFIG(3, 0x1234);  /* config word 3 does not exist on an 18C801 */
```

(1355) in-line delay argument too large (コード ジェネレータ)
インライン遅延シーケンス_delay に対して要求された命令サイクル数の値が大きすぎます。このシーケンスを複数回使う事で必要な長さの遅延時間を達成してください。
例:

```
#include <xc.h>
int main(void) {
    delay(0x400000); /* oops -- cannot delay by this number of cycles */
}
```

(1356) fixup overflow referencing * * (0x*) into * byte* at 0x*/0x* -> 0x* (* */0x*) (リンカ)**
「フィックスアップ」は、シンボリック参照を絶対アドレス値を持つオペランドへと置き換えるためにリンカによって実行されます。この処理は、psect (プログラムのセクションまたはブロック)を利用可能メモリに配置した後に実行されます。シンボルの値が大きすぎるためにアセンブラ命令内に収まらない場合、「フィックスアップ オーバーフロー」が発生します。例えば、あるアセンブラ命令がアドレス保持用に 8 ビットフィールドを備えているのに対し、このアドレスを表現するシンボルの値が 0x110 であった場合、この値を命令内にエンコードする事はできません。

フィックスアップ エラーは、しばしば手書きアセンブリコードによって発生します。多くの場合、ファイルレジスタ命令内で完全なバンク型データアドレスをマスクし忘れたか、ジャンプまたは呼び出し命令内でデスティネーションアドレスを正しくマスクしなかった事によりこのエラーが発生します。C ソースから生成されたアセンブリコードの場合、switch()文等のコンストラクタから生成されたアセンブリコードのブロックが大きすぎるためにジャンプ命令のスパンが長くなってしまった事によりこのエラーがしばしば発生します。既定値のリンカ オプションを調整した場合にも、このようなエラーが発生する可能性があります。

これらのエラーの原因を特定するための手順は以下の通りです。

- デバッグビルドを実行します(MPLAB X IDE 内でメニューから **Debug** > **Discrete Debugger Operation** > **Build for Debugging** を選択するか、コマンドラインで`-D  DEBUG` オプションを使用)。
- 対応するアセンブラ リストファイルを開きます(MPLAB X IDE 内の「Project Properties」でリストファイルの生成(**XC8 Compiler** > **Preprocessing and Messaging** > **Generate the ASM listing file**)を有効に設定するか、コマンドラインで`-Wa,-a` オプションを使用)。
- エラーメッセージに示されたアドレス位置の命令を見つけます。

例として、以下のエラーメッセージが表示されたと想定します。

```
main.c:4:(1356)(linker) fixup overflow referencing psect bssBANK1 (0x100) into 1 byte at
0x7FF0/0x1 -> 0x7FF0 (main.obj 23/0x0)
```

リンクされたファイルは `main.obj` である事から、`main.lst` という名前のアセンブリ リストファイルを調べる必要がある事が分かります。エラーが発生したメモリ位置は 0x7FF0 です。このメッセージからは、この命令が 1 バイトの値(このサイズは最も近いバイトに丸め処理される)を期待していたのに対して値が 0x100 であった事も分かります。

このアセンブリ リストファイル内で、エラーメッセージに示されたアドレス(0x7FF0)を探します。以下が、そのアドレスで見つかったリストファイルの内容です。

```
61  007FF0 6F00       movwf   _foobar,b       ;#
```

このアドレスでは、アセンブラ命令(`movwf`)がシンボル(`_foobar`)を参照しています。このシンボルを、このファイルの最後に示されるシンボルテーブル内で探します。以下が見つかったシンボルテーブルの内容です。

```
Symbol Table                    Tue Oct 28 11:06:37 2014
 foobar 0100
```

この例では、手書きの PIC18 `movwf` 命令によって問題が発生しています(メモリのバンクに対して 8 ビットのオフセットが取られたが、明らかにアドレス 0x100 はこのサイズを超えている)。

この命令は、以下の通りに書き直す必要があります。

```
MOVWF        BANKMASK(_foo)
```

これにより、バンク情報を格納しているアドレスの先頭ビットがマスクされます。

手書きではなくコンパイラによって生成されたアセンブラ命令によってこのエラーが発生した場合、アセンブラ リストファイル内のその命令から遡ってその命令を生成した C 命令文を特定し、C コード内でエラーに対処する必要があります。

(1357) fixup overflow storing 0x* in * byte* at 0x*/0x* -> 0x* (* */0x*) (リンカ)**
メッセージ 1356 を参照してください。

(1358) no space for * temps (*) (コード ジェネレータ)
コード ジェネレータは、このプログラムの一時変数(スクラッチ変数)を保持するために必要な空間を見つける事ができませんでした。

(1359) no space for * parameters (コード ジェネレータ)
コード ジェネレータは、特定関数のパラメータ変数を保持するために必要な空間を見つける事ができませんでした。

(1360) no space for auto/param * (コード ジェネレータ)
コード ジェネレータは、特定関数の auto 変数を保持するために必要な空間を見つける事ができませんでした。レジスタ内で渡される一部のパラメータも、この auto 領域内の空間に割り当てる事が必要になる場合があります。

(1361) syntax error in configuration argument (パーサー)
`#pragma config` に対する引数が正しくありません。
例:

```
#pragma config WDT     /* oops -- is WDT on or off? */
```

(1362) configuration setting *=* redefined (コード ジェネレータ)
同じ config プラグマ設定が異なる値で複数回指定されています。
例:

```
#pragma config WDT=OFF
#pragma config WDT=ON     /* oops -- is WDT on or off? */
```

(1363) unknown configuration setting (* = *) used (ドライバ)
ターゲット デバイスに対して不明のコンフィグレーション値と設定が使われています。不明のコンフィグレーション レジスタ番号が使われた場合も、このメッセージが発行されます。
例:

```
#pragma config WDR=ON     /* oops -- did you mean WDT?*/
#pragma config CONFIG1L=0x46 /* oops -- no 1L register on a 18F4520 */
```

(1364) can’t open configuration registers data file * (ドライバ)
コンフィグレーション値を格納したファイルが見つかりませんでした。

(1365) missing argument to pragma “varlocate” (パーサー)
#pragma varlocate に対する引数が正しくありません。

```
#pragma varlocate    /* oops -- what do you want to locate & where? */
```

(1366) syntax error in pragma “varlocate” (パーサー)
#pragma varlocate に対する引数が正しくありません。

```
#pragma varlocate fred     /* oops -- which bank for fred? */
```

(1367) end of file in _asm (パーサー)
_asm _endasm ブロック内でファイルの終端(EOF)マーカーに遭遇しました。

(1368) assembler message:* (アセンブラ)
これは、アセンブラソース内の MESSG ディレクティブによって生成されたアセンブラ アドバイザリ メッセージです。

(1369) can’t open proc file * (ドライバ)
選択されたデバイス向けの proc ファイルを開く事ができませんでした。

(1370) peripheral library support is not available for the * (ドライバ)
選択されたデバイス向けの周辺モジュール ライブラリが利用できません。

(1371) float type can’t be bigger than double type; double has been changed to * bits (ドライバ)
-fshort-double オプションの使用により、double 型のサイズが float 型よりも小さくなってしまいます。C 規格はこれを許容しません。double 型のサイズはメッセージに示されたサイズまで拡大されました。

(1372) interrupt level cannot be greater than * (コード ジェネレータ)
interrupt_level が選択されているデバイスに対して高すぎます。
例:

```
#pragma interrupt_level 4
// oops - there aren't that many interrupts on this device
```

(1374) the compiler feature “*” is no longer supported; * (ドライバ)
コンパイラはこの機能をサポートしなくなりました。

(1375) multiple interrupt functions (* and *) defined for device with only one interrupt vector (コード ジェネレータ)

これらの関数は両方とも割り込み関数ですが、ターゲット デバイスは 1 つの割り込みベクタ(従って 1 つの割り込み関数)しかサポートしません。

例:

```
interrupt void isr_lo(void) {
// ...
}
interrupt void isr_hi(void) {   // oops, cannot define two ISRs
// ...
}
```

(1376) initial value (*) too large for bitfield width (*) (コード ジェネレータ)

構造体内のビットフィールドに対する初期値がビットフィールドの幅に対して大きすぎます。

例:

```
struct {
unsigned flag :1;
unsigned mode :3;
} foobar = { 1, 100 };   // oops, 100 is too large for a 3 bit object
```

(1377) no suitable strategy for this switch (コード ジェネレータ)

コンパイラは、C コードの switch 文をエンコードするために使うストラテジを、そのコードと#pragma switch ディレクティブによる選択に基づいて決定する事ができませんでした。異なるストラテジを選択する必要があるかもしれません。

(1378) syntax error in pragma “*” (パーサー)

このプラグマに対する引数が無効です。

例:

```
#pragma addrqual ingore // oops -- did you mean ignore?
```

(1379) no suitable strategy for this switch (コード ジェネレータ)

コンパイラは各種ストラテジの内の 1 つを使って switch()文をエンコードします。このストラテジは case 値の数と値、switch 式、switch プラグマに基づいて選択されます。このエラーは、switch()文のエンコード用にどのストラテジも利用できなかった事を示します。Microchip 社技術サポートに詳細をお問い合わせください。

(1380) unable to use switch strategy “*” (コード ジェネレータ)

コンパイラは各種ストラテジの内の 1 つを使って switch()文をエンコードします。このストラテジは case 値の数と値、switch 式、switch プラグマに基づいて選択されます。このエラーは、要求されたストラテジが switch()文のエンコード用に使えなかった事を示します。Microchip 社技術サポートに詳細をお問い合わせください。

(1381) invalid case label range (パーサー)

case 値のレンジが正しく指定されていません。レンジは整数定数を使って昇順に(最小値...最大値の順番で)指定する必要があります。

例:

```
case 0 ...-2:// oops -- do you mean -2 ...0  ?
```

(1385) * “*” is deprecated (declared at *:*)(パーサー)

attribute ((deprecated))としてマーキングされた変数または関数が使われています。

例:

```
char __attribute ((deprecated)) foobar;
foobar = 9;                 // oops -- this variable is near end-of-life
```

(1386) unable to determine the semantics of the configuration setting “*” for register “*” (パーサー、コード ジェネレータ)
コンフィグレーション ビットに対して指定された値は、データシートに記載されているどの設定にも対応していません。コンパイラは指定の設定を試みますが、データシートを再度確認すべきです。
例:

```
#pragma config OSC=11
// oops -- there is no direct association for that value on an 18F2520
// either use OSC=3 or OSC=RC
```

(1387) in-line delay argument must be constant (コード ジェネレータ)
_delay インライン関数には引数として定数式を渡す必要があります。
例:

```
int delay_val = 99;
_delay(delay_val);   // oops, argument must be a constant expression
```

(1388) configuration setting/register of “*” with 0x* will be truncated by 0x* (パーサー、コード ジェネレータ)
コンフィグレーション ビットに書き込まれた値は大きすぎるか、規定の値になっていない可能性があります。
例:

```
#pragma config WDTPS=138 // oops -- do you mean 128?
```

(1389) attempt to reprogram configuration * “*” with * (is *) (パーサー、コード ジェネレータ)
書き込み済みのコンフィグレーション ビットに、元とは異なる設定が上書きされました。
例:

```
#pragma config WDT=ON
#pragma config WDT=OFF // oops -- watchdog on or off?
```

(1390) identifier specifies insignificant characters beyond maximum identifier length (パーサー)
設定された最大長を超える長さの識別子が使われています。長すぎる識別子は正しく認識されず、コードは正常に実行されない可能性があります。最大識別子長さは-N オプションを使って調整できます。
例:

```
int theValueOfThePortAfterTheModeBitsHaveBeenSet;
// oops, make your symbol shorter or increase the maximum
// identifier length
```

(1391) constant object size of * exceeds the maximum of * for this chip (コード ジェネレータ)
ターゲット デバイスに対して大きすぎる const オブジェクトが定義されています。
例:

```
const int array[200] = { ...};  // oops -- not on a Baseline part!
```

(1392) function “*” is called indirectly from both mainline and interrupt code (コード ジェネレータ)
この関数は、メインライン コードと割り込みコードの両方から呼び出されています。この警告が発行された場合、選択されたデバイスに対してそのようなコードはコンパイラの制限に違反する事を意味します。

(1393) possible hardware stack overflow detected; estimated stack depth:* (コード ジェネレータ)
デバイス上に割り当てられたスタックよりも多くのスタックがプログラムによって使われる可能性があるとコンパイラが推定しました。そのような状況が実際に発生した場合、コード実行は失敗する可能性があります。割り込みが使われる場合、コンパイラはスタックの使用量を推定する事しかできません。これは最悪条件でのスタック使用量として推定されます。この警告が発行された場合、関数呼び出しのネスティングを確認してください。

(1394) attempting to create memory range (* - *) larger than page size * (ドライバ)
ターゲット デバイスのページサイズよりも大きなプログラムメモリ「ページ」がメモリ設定に含まれている事をコンパイラ ドライバが検出しました。ほとんどの場合、これは--ROM オプションを使って既定値メモリ設定が変更された場合に発生します。データシートでデバイスのページサイズを調べ、--ROM オプションを使って指定された全ての連続メモリ領域がデバイスのページ境界をまたがない事を確認してください。
例:

```
--ROM=100-1fff
```

ページサイズが 800h である場合、上記は下記の通りに複数ページに分割する必要があります。

```
--ROM=100-7ff,800-fff,1000-17ff,1800-1fff
```

(1395) notable code sequence candidate suitable for compiler validation suite detected (*) (コード ジェネレータ)
このメッセージはエラーでも警告でもありません(コードのエラーを示す物ではありません)。コンパイラは、特定の内部コードテンプレートの組み合わせに遭遇した事を検出可能な内部チェック機能を備えています。コードのコンパイル中に特定コードの組み合わせが見つかると、このメッセージが発行されます。このメッセージが表示された場合、お客様のコードを弊社サポートチームにご提供いただければ幸いです。弊社はこの情報をコンパイラのテストプロセスに役立てます。

(1396) “*” positioned in the * memory region (0x* - 0x*) reserved by the compiler (コード ジェネレータ)
一部のメモリ領域がコンパイラ用に予約されます。通常これらの領域は、コード内で定義されている変数の割り当て用には使われません。しかし、変数を絶対アドレス化する事により、これらの領域内に変数を配置する事が可能となり、通常ならリンカによって発行されるエラーを回避する事ができます。絶対アドレス変数はどの位置にも配置可能であり、他のオブジェクトの上にすら配置できます。コード ジェネレータからのこの警告は、コンパイラが予約する予定のメモリ内に配置された絶対アドレス変数が検出された事を示します。この警告が発行された場合、その絶対アドレス変数を異なる位置に移動させる必要があります。通常このメッセージは、変数が共通メモリ空間内に配置される時に発行されます。
例:

```
char shared  at(0x7);    // oops, this memory is required by the compiler
```

(1397) unable to implement non-stack call to “*”; possible hardware stack overflow (コード ジェネレータ)
コンパイラは、call アセンブリ命令とハードウェア スタックを使わずに(すなわちルックアップ テーブルを使って) C 関数の呼び出しをエンコードする必要がありましたが、それができませんでした。関数がポインタを介して間接的に呼び出される場合、call 命令が必要かもしれませんが、ハードウェア スタックが既に使い果たされている場合、追加の呼び出しによってスタック オーバーフローが発生します。

(1401) eeprom qualified variables can’t be accessed from both interrupt and mainline code (コード ジェネレータ)
全ての eeprom 変数は、再入不可ルーチンを介してアクセスされます。 割り込みコードとメインライン コードの両方から eeprom 修飾された変数へのアクセスが試みられるとエラーが発生する可能性があります。 割り込み関数から eeprom 変数へのアクセスは避けてください。

(1402) a pointer to eeprom can’t also point to other data types (コード ジェネレータ)
ポインタは、EEPROM 空間と通常のデータ空間の両方にターゲットを持つ事はできません。

(1403) pragma “*” ignored (パーサー)
このプラグマは効果を持たず、コンパイラによって無視されます。このメッセージは C18 互換モードでのみ発行されます。
例:

```
#pragma varlocate "mySection" fred  // oops -- not accepted
```

(1404) unsupported:* (パーサー)

一部のコード機能がサポートされない事を示すために非サポート__attribute__が使われました。

このメッセージには非サポートの(使ってはいけない) 機能が示されます。

(1405) storage class specifier “*” ignored (パーサー)
指定された記憶域クラスは不要であり、コンパイラによって無視されます。このメッセージは、C18 互換モードでのみ発行されます。
例:

```
int procInput(auto int inValue)  // oops -- no need for auto
{ ...
```

(1406) auto eeprom variables are not supported (コード ジェネレータ)
eeprom として修飾された変数は auto にできません。eeprom として修飾された static ローカル オブジェクトを定義する事は可能です。
例:

```
int main(void) {
eeprom int mode; // oops -- make this static or global
```

(1407) bit eeprom variables are not supported (コード ジェネレータ)
eeprom として修飾された変数は bit 型にできません。
例:

```
eeprom bit myEEbit;    // oops -- you cannot define bits in EEPROM
```

(1408) ignoring initialization of far variables (コード ジェネレータ)
far として修飾された変数に初期値を代入する事はできません。この後のコード内で値を代入してください。
例:

```
far int chan = 0x1234; // oops -- you cannot assign a value here
```

(1409) warning number used with pragma “warning” is invalid (パーサー)
warning プラグマで使われたメッセージ番号が 0 未満であるか最高メッセージ番号を超えています。
例:

```
#pragma warning disable 1316 13350   // oops -- possibly number 1335?
```

(1410) can’t assign the result of an invalid function pointer (コード ジェネレータ)
コンパイラは、関数ポインタへと型変換される定数を介して一部の関数が呼び出される事を許容しますが、全ての関数でそれを許容するわけではありません。指定されたアドレスは、このデバイスに対して無効な場合があります。
例:

```
foobar += ((int (*)(int))0x0)(77);
// oops -- you cannot call a function with a NULL pointer
```

(1411) Additional ROM range out of bounds (ドライバ)
-mrom オプションで指定されたプログラムメモリは、このデバイスがサポートする内部または外部メモリのレンジ外です。
例:

```
-mrom=default,+2000-2ffff
```

上記の 2ffff は大きすぎます。2fff の誤りかもしれません。

(1412) missing argument to pragma “warning disable” (パーサー)
#pragma warning disable の後では、無効にするメッセージをコンマで区切って指定する必要があります。
例:

```
#pragma warning disable  // oops -- what messages are to be disabled?
```

下記は正しい例です。

```
#pragma warning disable 1362
```

(1413) pointer comparisons involving address of “*”, positioned at address 0x0, may be invalid (コード ジェネレータ)
アドレス 0 に配置された絶対アドレス オブジェクトが自身のアドレスを取得した場合、取得したアドレスが NULL をチェックするコードによって NULL ポインタと判断されてエラーが発生する可能性があります。
例:

```
int foobar  at(0x00);
int  * ip;
int main(void)
{
    ip = &foobar;  // oops -- 0 is not a valid address
```

(1414) option * is defunct and has no effect (ドライバ)
このオプションはもうサポートされていません。これは無視されます。
例:

```
xc8 --chip=18f452 --cp=24 main.c
```

`--cp` オプションはもう不要です。

(1415) argument to “merge” psect flag must be 0 or 1 (アセンブラ)
この psect フラグには 0 または 1 を代入する必要があります。
例:

```
PSECT myTxt,class=CODE,merge=true  ; oops -- I think you mean merge=1
```

(1416) psect flag “merge” redefined (アセンブラ)
psect に対して以前指定されたのとは異なる `merge` フラグ値が再定義されました。
例:

```
psect mytext,class=CODE,merge=1
; and later
psect mytext,class=CODE,merge=0
; Oops, can mytext be merged or not?
```

(1417) argument to “split” psect flag must be 0 or 1 (アセンブラ)
この psect フラグには 0 または 1 を代入する必要があります。
例:

```
psect mytext,class=CODE,split=5
```

`split` フラグの引数は 0 または 1 である事が必要です。

(1418) Attempt to read “control” qualified object which is Write-Only (コード ジェネレータ)
書き込み専用レジスタに対する読み出しが試みられました。
例:

```
state = OPTION; // oops -- you cannot read this register
```

(1419) using the configuration file *; you can override this with the environment variable XC_XML (ドライバ) これはコンパイラのセットアップ中に選択されたコンパイラ コンフィグレーション ファイルです。 このファイルは、`XC_XML` 環境変数を使って変更できます。 このファイルは、コンパイラがインストールされている場所を特定するために使われます。メッセージ(1205)を参照してください。

(1420) ignoring suboption “*” (ドライバ)
指定されたサブオプションはこの処理系において無効であり、無視されます。
例:

```
--RUNTIME=default,+ramtest
```

(1421) the qualifier xdata is not supported by this architecture (パーサー)
指定された修飾子はこの処理系において無効であり、無視されます。
例:

```
_ xdata int coeff[2]; // that has no meaning for this target
```

(1422) the qualifier __ydata is not supported by this architecture (パーサー)
指定された修飾子はこの処理系において無効であり、無視されます。
例:

```
__ydata int coeff[2]; // that has no meaning for this target
```

(1423) case ranges are not supported (ドライバ)
GCC 形式の case 値レンジの使用は CCI (Common C Interface) 規格に準拠していません。規格に準拠するため、個々の case ラベルと値を使う必要があります。
例:

```
switch(input) {
case 0 ...5:     // oops -- ranges of values are not supported
    low();
```

(1424) short long integer types are not supported (パーサー)
short long 型の使用は CCI 規格に準拠していません。代わりに long 型を使う必要があります。
例:

```
short long typeMod;    // oops -- not a valid type for CCI
```

(1425) __pack qualifier only applies to structures and structure members (パーサー)
指定された修飾子は、構造体または構造体メンバーに対してのみ有効です。この修飾子は無視されます。
例:

```
__pack int c; // oops -- there aren't inter-member spaces to pack in an int
```

(1426) 24-bit floating point types are not supported; * have been changed to 32-bits (ドライバ)
CCI 規格に準拠するため、浮動小数点型は 32 ビット幅である事が必要です。これらの型は 32 ビット幅としてコンパイルされます。
例:

```
-fshort-double=24
```

この double サイズを設定する事はできません。

(1427) machine-dependent path specified in name of included file; use -I instead (プリプロセッサ)
CCI 規格に準拠するため、ヘッダファイル名の指定にディレクトリ区切り文字を含める事はできません。
例:

```
#inlcude <inc\lcd.h>    // oops -- do not indicate directories here
```

パス情報を削除し、-I オプションを使ってパスを指定してください。

例:

```
#include <lcd.h>
```

パスは-Ilcd オプションを使って指定します。

(1428) "*" is not supported; this feature will be ignored (ドライバ)
指定されたオプションはサポートされておらず、コンパイルに対して何の効果も有しません。
例:

```
xc8-cc -mcpu=18f4520 --html main.c
```

--html は有効なオプションではありません。

(1429) attribute "*" is not understood by the compiler; this attribute will be ignored (パーサー)
この属性はこの処理系において有効ではありません。この属性は無視されます。
例:

```
int x  attribute  ((deprecate)) = 0;
```

deprecate は deprecated の誤りです。

(1430) section redefined from “*” to “*” (パーサー)
1 つのオブジェクトを複数のセクションに配置しようとしています。
例:

```
int   section("foo")   section("bar") myvar;  // oops -- which section should it be in?
```

(1431) the __section specifier is applicable only to variable and function definitions at file-scope (パーサー)
__section()指定子を使ってローカル オブジェクトを配置する事はできません。
例:

```
int main(void) {
int   __section("myData") counter;  // oops -- you cannot specify a section for autos
```

(1432) “*” is not a valid section name (パーサー)
__section()で指定されたセクション名は無効です。セクション名は標準の C 識別子規則に準拠する必要があります。
例:

```
int __section("28data") counter;  // oops -- name cannot start with digits
```

(1433) function “*” could not be inlined (アセンブラ)
この関数はインライン化できませんでした。通常の方法で呼び出されます。
例:

```
int inline getData(int port)   // sorry -- no luck inlining this
{   //...
```

(1434) missing name after pragma “intrinsic” (パーサー)
intrinsic プラグマには関数名が必要です。このプラグマが必要になる事はほとんどありません。関数のインライン化には inline キーワードまたはプラグマが使えます。
例:

```
#pragma intrinsic    // oops -- what function is intrinsically called?
```

(1435) variable “*” is incompatible with other objects in section “*” (コード ジェネレータ)
起動時の初期化方法が異なる変数を同じ psect 内に配置する事はできません。例えば、起動時にクリアされる変数と起動時に非 0 の初期値が代入される変数は、別々の psect 内に配置する必要があります。同様に、ビット オブジェクトとバイト オブジェクトを同じ psect 内に格納する事もできません(char 型、int 型も同様)。
例:

```
int   section("myData") input;   // okay
int   section("myData") output;  // okay
int   section("myData") lvl = 0x12;  // oops -- not with uninitialized
bit   section("myData") mode;    // oops again -- no bits with bytes
// each different object to their own new section
```

(1436) “*” is not a valid nibble; use hexadecimal digits only (パーサー)
__IDLOC()の引数は、16 進数の桁(0～9、A～F) のみで構成される(基数指定子等、他の文字を一切含まない)事が必要です。16 進数の桁以外の文字は、対応する位置に 0 として書き込まれます。
例:

```
__IDLOC(0x51); // oops -- you cannot use the 0x radix modifier
```

(1437) CMF error * (Cromwell、リンカ)
Cromwell またはリンカによって読み出された CMF ファイルは無効です。このファイルを編集したり手動で生成したりしていない場合、これは内部エラーです。Microchip 社技術サポートに詳細をお問い合わせください。

(1438) pragma “*” options ignored (パーサー)
このプラグマに対して非サポートのオプションが使われています。このオプションは無視されます。
例:

```
#pragma inline=forced  // oops -- no options allowed with this pragma
```

(1439) message:* (パーサー)
これは、いずれかのプラグマ ディレクティブによって生成されたアドバイザリ メッセージです。このメッセージはIAR C 言語拡張を使っている場合にのみ出力されます。
例:

```
#pragma message "this is a message from your programmer"
```

(1440) big-endian storage is not supported by this compiler (パーサー)
変数に対して__big_endian IAR 拡張が指定されています。このコンパイラは、ビッグエンディアン形式の保存書式をサポートしません。この指定を削除し、その他のコードがこのエンディアン指定に頼っていないか確認してください。
例:

```
_ big_endian int volume; // oops -- this won’t be big endian
```

(1441) use __at() instead of '@' and ensure the address is applicable (パーサー)
IAR C 拡張の使用時に@アドレス指定子が使われています。新しいアーキテクチャでは、指定された全てのアドレスが正しくない可能性があります。デバイス データシートを参照してアドレスを見直してください。この警告が再度表示されないようにするため、修正したアドレスに対して__at()指定子を使ってください。

(1442) type used in definition is incomplete (パーサー)
オブジェクトの定義時には完全な型を使う必要があります。不完全な型を使ってオブジェクトが定義されると、このメッセージが発行されます。
例:

```
typedef struct foo foo_t;
foo_t x;   // oops -- you cannot use foo_t until it is fully defined
struct foo {
int i;
};
```

(1443) unknown --EXT sub-option “*” (ドライバ)
--EXT オプションに対するサブオプションが無効です。
例:

```
xc8 --chip=18f8585 x.c --ext=arm --ext=cci
```

正しいサブオプションは iar、cci、xc8 です。

(1444) respecified C extension from “*” to “*” (ドライバ)
--EXT オプションが異なる引数で複数回使われています。コンパイラが受け入れる C 拡張は、このオプションが最後に使われた時の引数によって決まります。
例:

```
xc8 --chip=18f8585 x.c --ext=iar --ext=cci
```

どちらの C 拡張を使いたいのか不明です。

(1445) #advisory:* (プリプロセッサ)
これは、いずれかのプラグマ ディレクティブによって生成されたアドバイザリ メッセージです。
例:

```
#advisory "please listen to this good advice"
```

(1446) #info:* (プリプロセッサ)
これは、いずれかのプラグマ ディレクティブによって生成されたアドバイザリ メッセージです。このメッセージは#advisory メッセージ(1445)と同じです。
例:

```
#info "the following is for your information only"
```

(1447) extra -L option (-L*) ignored (プリプロセッサ)
このエラーは、重複する-L オプションがプリプロセッサに渡された事により発生します。このアプリケーションを直接実行したのではない場合、これは内部エラーであると考えられます。Microchip 社技術サポートに詳細をお問い合わせください。

© 2022 Microchip Technology Inc. and its subsidiaries

(1448) no dependency file type specified with -L option (プリプロセッサ)
このエラーは、不正な-L オプションがプリプロセッサに渡された事により発生します。このアプリケーションを直接実行したのではない場合、これは内部エラーであると考えられます。Microchip 社技術サポートに詳細をお問い合わせください。

(1449) unknown dependency file type (*) (プリプロセッサ)
このエラーは、不明の依存情報ファイルがプリプロセッサに渡された事により発生します。このアプリケーションを直接実行したのではない場合、これは内部エラーであると考えられます。Microchip 社技術サポートに詳細をお問い合わせください。

(1450) invalid --*-spaces argument (*) (Cromwell)
Cromwell に渡されたオプションは有効なメモリ空間に対応していません。spaces 引数は、空間を現す有効な番号である事が必要です。
例:

```
--data-spaces=a
```

a は有効なデータ空間番号ではありません。

(1451) no * spaces have been defined (Cromwell)
Cromwell には、各番号のメモリ空間の型を示す情報を渡す必要があります。これには--code-spaces および--data-spaces オプションを使います。このアプリケーションを直接実行したのではない場合、これは内部エラーであると考えられます。Microchip 社技術サポートに詳細をお問い合わせください。

(1452) one or more spaces are defined as data and code (Cromwell)
Cromwell に渡されたオプションは、メモリ空間をコード空間とデータ空間の両方として定義しています。このアプリケーションを直接実行したのではない場合、これは内部エラーであると考えられます。Microchip 社技術サポートに詳細をお問い合わせください。
例:

```
--code-space=1,2  --data-space=1
```

空間 1 がコード空間とデータ空間の両方で指定されています。

(1453) stack size specified for non-existent * interrupt (ドライバ)
-mstack オプションにより各割り込み向けにスタックの最大サイズが指定されましたが、コンパイラは対応する割り込み関数の定義を見つける事ができませんでした。これは、そのスタック空間は決して使われない事を意味します。各割り込みに対して割り込み関数が作成されているか確認してください。
例:

```
-mstack=reentrant:20:20:auto
```

上記は 2 つの割り込みスタックを要求していますが、コンパイラは両方の割り込み関数定義を見つける事ができません。

(1454) stack size specified (*) is greater than available (*) (ドライバ)
-mstack オプションにより各スタックの最大サイズが指定されましたが、要求されたメモリの総量が利用可能メモリのサイズを超えています。
例:

```
-mstack=software:1000:1000:20000
```

小型デバイスに対してスタック空間が大きすぎます。

(1455) unrecognized stack size “*” in “*” (ドライバ)
-mstack オプションにより各スタックの最大サイズが指定されましたが、1 つまたは複数のサイズが無効な値です。このオプションでは 10 進値でサイズを指定するか、トークン auto (既定値サイズ)を指定する必要があります。
例:

```
-mstack=software:30:all:default
```

10 進値または auto 以外は使えません。

(1456) too many stack size specifiers (ドライバ)
-mstack オプションで指定されたソフトウェア スタックの最大サイズの数が多すぎます。最大スタックサイズの指定は必須ではありません。指定する場合、各割り込み向けに 1 つずつと、メインライン コード向けに 1 つだけ指定します。
例:

```
-mstack=reentrant:20:20:auto
```

割り込みを 1 つしか実装していないデバイスに対してサイズの数が多すぎます。

(1457) local variable “*” cannot be made absolute (コード ジェネレータ)
ローカル変数(auto 変数、パラメータ、static ローカル オブジェクトを含む) のアドレスを指定する事はできません。
例:

```
int pushState(int a) {
int cnt  at(0x100); // oops -- you cannot specify an address ...
```

(1458) Omniscient Code Generation not available in Standard mode (ドライバ)
このメッセージは、標準動作モードで最適化が全て有効になっていない事を警告するために表示されます。

(1459) peripheral library support is missing for the * (ドライバ)
周辺モジュール ライブラリのコードは選択されたデバイス向けではありません。周辺モジュール ライブラリをリンクするオプションを無効にしてください。

(1460) function-level profiling is not available for the selected chip (ドライバ)
関数プロファイリングは PIC18 または拡張ミッドレンジ デバイス向けにのみ利用可能です。その他のデバイス向けに関数プロファイリングを使ってはいけません。

(1461) insufficient h/w stack to profile function “*” (コード ジェネレータ)
関数プロファイリングは、ハードウェア スタックの 1 レベルを使います。スタックの全てがプログラムによって使われているため、全ての関数をプロファイリングする事はできません。メッセージに示された関数にはプロファイリング コードは埋め込まれず、MPLAB X IDE によって表示されるプロファイリング情報に反映されません。

(1462) reentrant data stack model option conflicts with stack management option and will be ignored (コード ジェネレータ)
マネージド スタック オプションを使うと、ハードウェア スタック深さを超える関数呼び出しをルックアップ テーブル方式の呼び出しへ変換できます。しかし、再入可能関数モデルが有効である場合、このオプションは有効にできません。このオプションと再入可能関数モデル オプションの両方が有効にされた場合、このメッセージが生成され、マネージド スタック オプションが無効にされた状態でコンパイルされます。再入可能関数モデル オプションまたはマネージド スタック オプションのどちらかを無効にしてください。

(1463) reentrant data stack model not supported on this device; using compiled stack for data (コード ジェネレータ)
ターゲット デバイスは再入可能関数をサポートしません。プログラムは、スタックベース データをコンパイルド スタック内に配置するようコンパイルされます。

(1464) number of arguments passed to function “*” does not match function's prototype (コード ジェネレータ)
引数を持った関数が呼び出されましたが、関数定義のパラメータリストが空でした(パラメータリストが void ではありません)。
例:

```
int test();    // oops--this should define the parameters
...
test(12, input);
```

(1465) the stack frame size for function “*” (* bytes) has exceeded the maximum allowable (* bytes) (コード ジェネレータ)
この関数の auto パラメータおよび一時変数向けソフトウェア スタック要求サイズが最大許容サイズを超えている事をコンパイラが検出しました。最大許容サイズは、拡張ミッドレンジ デバイスに対して 31、PIC18 デバイスに対して 127 です。これらの変数のサイズまたは数を削減してください。あるいは、auto オブジェクトの代わりに static ローカル オブジェクトが使えないか検討してください。
例:

```
reentrant int addOffset(int offset) {
int report[400];    // oops--this will never fit on the software stack
```

(1466) registers * unavailable for code generation of this expression (コード ジェネレータ)
コンパイラは、この命令文からコードを生成する事ができませんでした。これは基本的に「can't generate code」エラーメッセージ(712)と同じですが、原因は必要なレジスタが利用できなかった事に関連します。対応策についてはメッセージ(712) の説明を参照してください。

(1467) pointer used for writes includes read-only target "*" (コード ジェネレータ)
const 修飾されていないポインタを用いて const 修飾されたターゲットへの書き込みをしています。これは異常な動作や他のエラーメッセージの原因となる可能性があります。
例:

```
void keepTotal(char * cp) {
    *cp += total;
}
char c;
const char name[] = "blender";
keepTotal(&c);
keepTotal(&name[2]); // oops--will write a read-only object
```

(1468) unknown ELF/DWARF specification (*) in --output option (ドライバ)
ELF サブオプションが不明のフラグを使っています。
例:

```
–output=elf:3
```

値が 3 の elf フラグは存在しません。

通常、elf サブオプションとそのフラグは MPLAB X IDE プラグインにより発行されます。このエラーが発生した場合、コンパイラと IDE の詳細を Microchip 社の技術サポートにお知らせください。

(1469) function specifier "reentrant/software" used with "*" ignored (コード ジェネレータ)
この関数に対して reentrant (または software)指定子が使われましたが、この関数にはソフトウェア スタックが使えません。この指定子は無視され、この関数はコンパイルド スタックを使います。
例:

```
reentrant int main(void) // oops--main cannot be reentrant
{ ...
```

(1470) trigraph sequence "??*" replaced (プリプロセッサ)
プリプロセッサは、ソースコード内にある特殊文字に対する 3 文字表記を置換しました。この 3 文字表記が意図的に使われた物かどうか確認してください。
例:

```
char label[] = "What??!"; // you do know that's a trigraph
// sequence, right?
```

(1471) indirect function call via a NULL pointer ignored (コード ジェネレータ)
コンパイラは、NULL 以外に有効なターゲットを持たない関数ポインタを検出しました。このポインタは関数の呼び出し用に使われていますが、呼び出しは実行されません。
例:

```
int (*fp)(int, int);
result = fp(8,10);  // oops--this pointer has not been initialized
```

(1472) --CODEOFFSET option ignored:* (ドライバ)
コンパイラは-mcodeoffset オプションの指定を無視します。このメッセージには、このオプションが無視された理由(このオプションは既に指定済み、または、コンパイルモードがこのオプションの使用をサポートしていない)が示されます。

(1474) read-only target "*" may be indirectly written via pointer (コード ジェネレータ)
これはメッセージ(1467)と同じですが、エラーが要求される状況向けに生成されます。このメッセージは、1 つ以上の読み出し専用ターゲットに対して書き込みに利用可能なポインタを使っている事を示します。
例:

```
const char c = 'x';
char * cp = &c; // will produce warning 359 about address assignment
```

```
*cp = 0x44;     // oops--you ignored the warning above, now you are
// actually going to write using the pointer?
```

(1478) initial value for “*” differs to that in *:*(コード ジェネレータ)
このオブジェクトは、異なる初期値で複数回定義されています。静的記憶期間の非初期化オブジェクト(および、その全ての要素またはメンバー)は暗黙的に値 0 で初期化される事に注意してください。
例:

```
char myArray[5] = { 0 };
// elsewhere
char myArray[5] = {0,2,4,6,8};  // oops--previously initialized
// with zeros, now with different values
```

(1479) EEPROM data not supported by this device (パーサー)
eeprom 修飾子が使われましたが、ターゲット デバイスは EEPROM を備えていません。この修飾子は全て無視されます。
例:

```
eeprom int serialNo;          // oops--no EEPROM on this device
```

(1480) initial value(s) not supplied in braces; zero assumed (コード ジェネレータ)
オブジェクトを初期化するために代入演算子が使われていますが、波カッコ内で値が指定されていません。このオブジェクトは値 0 で初期化されます。
例:

```
int xy_map[3][3] = { };   // oops--did you mean to supply values?
```

(1481) call from non-reentrant function, “*”, to “*” might corrupt parameters (コード ジェネレータ)

同じ関数ポインタによって複数の関数が間接的に呼び出し可能である場合、それらの関数は「buddy」関数と呼ばれ、それらの関数に対するパラメータはメモリ上で整列配置されます。これによって、ポインタ呼び出し関数によらず、同様のパラメータロードが可能になります。しかしこれは、buddy 関数同士の間で互いに呼び出す事は直接的であれ間接的であれ不可能である事を意味します。

例:

```
// fpa can call any of these, so they are all buddies
int (*fpa[])(int) = { one, two, three };
int one(int x) {
    return three(x+1);  // oops--one() cannot call buddy three()
}
```

(1482) absolute object * overlaps * (リンカ)
絶対アドレス オブジェクトが予約されましたが、他の絶対アドレス オブジェクトによって予約済みのメモリとオーバーラップしています。
例:

```
unsigned char nfo[6] __at(0x80);
unsigned char nfo2[6]  at(0x7b);  //oops--this overlaps nfo
```

(1483) __pack qualifier ignored (パーサー)
__pack 修飾子は、auto または static ローカル構造体に対して効果を有しません。この修飾子は無視されました。
例:

```
int setInput(void) {
    __pack struct {       //oops--this will not be packed
        unsigned x, y;
    } inputData;
...
```

(1484) the branch errata option is turned on and a BRW instruction was detected (アセンブラ)
選択されているデバイスでこの命令を使うと異常動作が発生する可能性があります。このデバイス向けに発行されたエラッタを参照し、使用デバイスのリビジョンに対してこの制約が適用されるかどうか確認してください。制約が適用される場合、手書きアセンブリコードからこの命令を削除してください。
例:

```
btfsc status,2
brw  next      ;oops--this instruction cannot be safely used
call update
```

(1485) * mode is not available with the current license and other modes are not permitted by the NOFALLBACK option (ドライバ)
このコンパイラのライセンスでは、要求されたコンパイラ動作モードが使えません。--nofallback オプションが有効であるため、コンパイラは下位の動作モードへフォールバックせずにこのエラーを生成しました。コンパイラ ライセンスがその動作モードをサポートしているにも関わらずこのエラーが発生した場合、コンパイラが正しくアクティベートされていない可能性があります。

(1486) size of pointer cannot be determined during preprocessing.Using default size * (プリプロセッサ)
プリプロセッサはポインタ型のサイズを特定できません。プリプロセッサによって評価する必要がある式内で sizeof 演算子を使わないでください。
例:

```
#if sizeof(int *) == 3   // oops - you can't take the size of a pointer type
#define MAX 40
#endif
```

(1488) the stack frame size for function “*” may have exceeded the maximum allowable (* bytes) (コード ジェネレータ)
このメッセージは、関数のソフトウェア スタックデータが理論的な最大許容サイズを超えた場合に発行されます。このスタック空間の外に配置されたデータには、一部の命令によってのみアクセス可能です。この警告が発行されてもコードが期待通りに動作し、制限を超えたデータを取得できる場合もあります。そのような場合、この警告を無視して構いません。この警告は、関数が大きなオブジェクト(構造体等)をスタック上で返す再入可能関数を呼び出す場合に発生する可能性があります。命令が制限を超えたデータにアクセスできない場合、この警告に加えて、ビルド処理のアセンブリ段階で追加のエラーメッセージが発行されます。その場合、関数によって定義されているスタックデータの量を削減する事を検討する必要があります。

(1489) unterminated IF directive at end of psect * (アセンブラ)
アセンブラはこの psect の終端に達しましたが、直前に遭遇した IF または ELSIF ディレクティブに対応する ENDIF ディレクティブが見つかりません。
例:

```
psect mytext,class=CODE,reloc=2
movlw 20h
IF TEST_ONLY
    movlw 00h
    movwf _mode,c ; oops--where does the IF end?
psect nexttext,class=CODE,reloc=2
```

(1490) ENDIF not inside an IF directive (アセンブラ)
アセンブラは ENDIF ディレクティブに遭遇しましたが、これに対応する IF または ELSIF ディレクティブが存在しません。
例:

```
psect mytext,class=CODE,reloc=2
movlw 20h
IF TEST_ONLY
    movlw 00h
ENDIF
ENDIF    ; oops--what does this terminate?
```

(1491) runtime sub-option “*” is not available for this device (ドライバ)
--RUNTIME オプションに対して指定されたサブオプションは、選択されているデバイスでは利用できません。
例:

```
xc8 --CHIP=MCP19114 --RUNTIME=+osccal main.c
```

このデバイスでは osccal サブオプションは使えません。

(1492) using updated 32-bit floating-point libraries; improved accuracy might increase code size (コード ジェネレータ)
このアドバイザリ メッセージは、32 ビット浮動小数点ライブラリの変更によってコードサイズが増加する可能性がある事をユーザに警告します。

(1493) updated 32-bit floating-point routines might trigger “can't find space” messages appearing after updating to this release; consider using the smaller 24-bit floating-point types (リンカ)
このアドバイザリ メッセージは、32 ビット浮動小数点ライブラリの変更によって「Can’t Find Space」メッセージが発行される可能性がある事をユーザに警告します。

(1494) invalid argument to normalize32 (アセンブラ)
NORMALIZE32 演算子がリテラル定数でないオペランドに対して使われています。
例:

```
NORMALIZE(_foobar)  ; oops--that must be a literal constant operand
```

(1495) ADDFSR/SUBFSR instruction argument must be 0-3 (アセンブラ)
この命令に対するオペランドは、リテラル定数(レンジは 0〜3)である事が必要です。
例:

```
addfsr 1, 6 ; oops--the offset must be between 0 to 3
```

(1496) arithmetic on pointer to void yields Undefined Behavior (コード ジェネレータ)
ポインタに対する演算を実行するにはターゲット オブジェクトのサイズを知る必要がありますが、汎用(void *)ポインタの場合、このサイズは不明です。
例:

```
void * vp;
vp++;   // oops—how can this be incremented without knowing what it points to?
```

(1497) more than one *interrupt function defined (コード ジェネレータ)
同じ優先度で複数の関数を定義する事はできません。
例:

```
void interrupt lo_isr(void) {  // oops – was this meant to be a low_priority interrupt?
  …
}
void interrupt hi_isr(void) {
  …
}
```

(1498) pointer (*) in expression may have no targets (コード ジェネレータ)
NULL を格納したポインタが逆参照されています。これ以前に、有効なアドレスをポインタに代入する必要があります。
例:

```
char * cp, c;
c = *cp;    // oops –what is cp pointing to?
```

(1499) only decimal floating-point constants can be suffixed “f” or “F”
整数に対して浮動小数点定数向けの接尾辞が使われています。
例:

```
float myFloat = 100f*3.2; // oops – is ‘100f’ mean to be a hex or floating-point value?
```

10.4 メッセージ 1500～1999

(1500) invalid token in #if expression (プリプロセッサ)
不正なプリプロセッサ式が存在します。
例:

```
#define LABEL
#define TEST 0
#if (LABEL == TEST) // oops--LABEL has no replacement text
```

(1504) the PIC18 extended instruction set was enabled but is not supported by this compiler (パーサー) MPLAB XC8 コンパイラは、PIC18 拡張命令セットを使ったコードの生成をサポートしません。 拡張命令セットのコンフィグレーション ビットは常に無効にしておく必要があります。
例:

```
#pragma config XINST=ON // oops--this must be disabled at all times
```

(1505) interrupts not supported by this device (コード ジェネレータ)
割り込みをサポートしないデバイスに対して割り込み関数が定義されています。
例:

```
void interrupt myIsr(void)  // oops--nothing will trigger this
{ ...}
```

(1506) multiple interrupt functions (* and *) defined at interrupt level * (コード ジェネレータ)
同じ優先度に対して複数の割り込み関数が定義されています。
例:

```
void interrupt low_priority
isr(void)
{ ...}
void interrupt low_priority    // oops--you can have two ISRs
loisr(void)                // with the same priority
{ ...}
```

(1507) asmopt state popped when there was no pushed state (アセンブラ)
アセンブラ オプティマイザの状態がアセンブリコード内でポップされていますが、以前のプッシュ済み状態は存在しません。
例:

```
movlw  20h
movwf LATB
opt asmopt_pop; oops--there was never a state pushed
```

(1508) specifier “ ram” ignored (パーサー)
このポインタ ターゲット指定子は通常の変数に対して使えません。この指定子は無視されます。この定義がポインタ型を指定する事を意図した物ではない事を確認してください。
例:

```
_ ram int ip;   // oops -- was this meant to be a pointer?
```

(1509) specifier “ rom” ignored (パーサー)
このポインタ ターゲット指定子は通常の変数に対して使えません。この指定子は無視されます。この定義がポインタ型を指定する事を意図した物ではない事を確認してください。
例:

```
const   rom int cip;   // oops -- was this meant to be a pointer?
```

(1510) non-reentrant function “*” appears in multiple call graphs and has been duplicated by the compiler (コード ジェネレータ)
この関数はメインライン コードと割り込みコードの両方から呼び出されているため、この関数向けに生成されたコード出力は複製されました。これは異常動作の可能性を示すメッセージではありません。関数のコード出力が複製されるのを防ぐ必要がある場合、ハイブリッド スタックモデルの使用(可能な場合)またはソースコードの変更を検討してください。

(1511) stable/invariant mode optimizations no longer implemented; option will be ignored (ドライバ)
このオプションは廃止済みであるため無視されました。

(1512) stable/invariant mode optimizations no longer implemented; specifier will be ignored (コード ジェネレータ)
この指定子は廃止済みであるため無視されました。

(1513) target “*” of pointer “*” not in the memory space specified by * (コード ジェネレータ)
この命令文によってアドレスが代入されたポインタは、ポインタ ターゲット指定子を使って定義されています。この代入は、メモリ空間指定子と矛盾するアドレスをポインタに代入する可能性があります。
例:

```
_ rom int * ip;
int foobar;
ip = &foobar;  // oops -- foobar is in data memory, not program memory
```

(1514) “__ram” and “__rom” specifiers are mutually exclusive (パーサー)
同一ポインタに対して__ram および__rom ポインタ ターゲット指定子の両方が使われていますが、これは無意味です。ポインタのターゲットが両方のメモリ空間内に存在する場合、これらの指定子を使ってはいけません。
例:

```
// oops -- you can’t limit ip to only point to objects in ram and
// also only point to objects in rom
_ ram  rom int * ip;
```

(1515) disabling OCG optimizations for this device is not permitted (ドライバ)
メモリの制限により、このデバイス向けにプロジェクトをビルドできません。OCG カテゴリの最適化を有効にしてください。

(1516) compiler does not support 64-bit integers on the target architecture
メモリの制約により、このターゲット デバイスは 64 ビット整数をサポートできません。代わりに、より小さなサイズの整数が使われます。この警告を防ぐには、long long 型よりも小さな型を選択する必要があります。
例:

```
long long int result;  // oops - this will not be 64-bits wide
```

(1517) peripheral library support only available for C90 (ドライバ)
レガシーの周辺モジュール ライブラリは C90 規格向けにビルドされており、他の C 規格向けには正しく機能しません。

(1518) * function call made with an incomplete prototype (*) (コード ジェネレータ)
関数が呼び出されていますが、この関数向けの完全なプロトタイプが見つかりません。宣言内のパラメータリストが空白になっていないか確認してください。
例:

```
void foo(); // oops -- how will this call be encoded?
int main(void)
{
    foo();
}
```

(1519) note-psects will ignore optimisation-related psect flags (アセンブラ)
note フラグを使う psect は最適化できません。このフラグの他に最適化を要求するフラグが追加で指定されていても、それらは無視されます。通常 note フラグを使う psect は、プロジェクトコードに関係しないデバッグ情報を格納します。

(1520) malformed mapfile while generating summary: no space at position * (ドライバ)
コンパイル後にメモリ使用量サマリを生成する際にマップファイルから読み出された psect 情報が正しくありません。

(1521) internal error encountered creating DWARF information; contact Microchip support with details (Cromwell)
これは内部コンパイラエラーです。Microchip 社技術サポートに詳細をお問い合わせください。

© 2022 Microchip Technology Inc. and its subsidiaries

(1522) RAM access bit operand not specified, assuming * (アセンブラ)
アセンブリ命令に対して RAM アクセス オペランドが指定されていないため、アセンブラは命令のデスティネーション位置をメッセージに示した通りに想定しました。コードを意図した通り確実に動作させるため、常に RAM アクセスビットを明確に指定してください。
例:

```
    movwf input   ; oops - use for example movwf input,b to indicate banked access, etc
```

(1523) debug_source state popped when there was no pushed state (アセンブラ)
debug_source 設定の状態がポップされていますが、以前の状態がプッシュされていません。

```
DEBUG_SOURCE asm
    MY_UNLOCK_MACRO
DEBUG_SOURCE pop   ; oops - there was no prior push of the debug source state
```

(1600) "*" argument :* (Hexmate)
Hexmate オプションに対する引数にエラーが存在します。このメッセージには、エラーの原因となった引数とその問題が示されます。

(1601) "*" argument :* (Hexmate)
Hexmate オプションに対する引数に警告が存在します。このメッセージには、警告の原因となった引数とその潜在的問題が示されます。

(1602) contents of the hex-data (*) do not conform with the chosen output type (*) (Hexmate)
HEX ファイルに書き込まれるデータが存在しますが、このデータは選択されている HEX ファイル フォーマットに対して無効です(例: データのアドレスがサポートされるフォーマットに対して大きすぎる)。異なる出力フォーマットを使うか、問題となっているデータのソースを確認してください。

(1604) Storage Area Flash has been enabled by the configuration bits; ensure this memory is not used for program code (アセンブラ)
コンフィグレーション ビットの設定は、このデバイス上の記憶域フラッシュメモリを有効にしています。このメモリは、プログラムの他の部分によって使われないように、オプション(-mreserve 等)を使って予約する必要があります。詳細はデバイスのデータシートを参照してください。
例:

```
#pragma config SAFEN=ON ;make sure ordinary program code does not use this space
```

(1605) Block Table Read Protection has been enabled by the configuration bits; this may affect variable initialization and reading constants in program memory (アセンブラ)
コンフィグレーション ビットの設定は、ブロックテーブル読み出し保護機能を有効にしています。これは、プログラムメモリから読み出すコードに影響を及ぼし、異常動作の原因となる可能性があります。詳細はデバイスのデータシートを参照してください。
例:

```
#pragma config EBTRB=ON  ;remember this might affect reading program memory
```

10.5 メッセージ 2000～2499

(2000) * attribute/specifier has a misplaced keyword (*) (パーサー)
属性トークンが不適切な位置で使われています。
例:

```
// oops -- 'base' is a token which has specific meaning
void  __interrupt(irq(base)) isr(void)
```

(2001) * attribute/specifier has a misplaced parenthesis (パーサー)
この属性コンストラクタ内で使われている丸カッコの配置が正しくありません。余分なカッコや位置が不適切なカッコが存在しないか確認してください。
例:

```
void __interrupt(irq((TMR0)) isr(void) // oops -- one too many '('s
```

(2002) __interrupt attribute/specifier has conflicting priority-levels (パーサー)
1 つの割り込み関数定義に対して複数の優先度が割り当てられています。
例:

```
//oops -- is it meant to be low or high priority?

void __interrupt(irq(TMR0), high_priority, low_priority) tc0Int(void)
```

(2003) * attribute/specifier has a duplicate keyword (*) (パーサー)
この属性内で同じトークンが複数回使われています。これらのトークンのいずれかが他の何かを意図した物ではないか確認してください。
例:

```
//oops -- using high_priority twice has no special meaning

void __interrupt(irq(TMR0), high_priority, high_priority) tc0Int(void)
```

(2004) __interrupt attribute/specifier has an empty “irq” list (パーサー)
__interrupt()指定子に対する引数 irq()は、割り込みベクタの番号またはシンボルをコンマで区切ったリストとして渡す必要があります。この関数を割り込み要因へリンクするには、少なくとも 1 つの値またはシンボルを指定する必要があります。
例:

```
//oops -- irq() does not indicate the interrupt source

void __interrupt(irq(),high_priority) tc0Int(void)
```

(2005) __interrupt attribute/specifier has an empty “base” list (パーサー)
__interrupt()指定子に対する引数 base()の指定は必須ではありませんが、指定する場合は割り込みベクタテーブルのアドレスをコンマで区切ったリストとして渡す必要があります。ベクタテーブルを配置するには、少なくとも 1 つのアドレスを指定する必要があります。ISR にベースアドレスを指定しなかった場合、そのベクタは IVTBASE レジスタのリセット値と同じアドレスに配置された割り込みベクタテーブル内に配置されます。
例:

```
//oops -- base() was used but did not indicate a vector table address

void  __interrupt(irq(TMR0), base()) tc0Int(void)
```

(2006) __interrupt attribute/specifier has a duplicate “irq” (*) (パーサー)
__interrupt()指定子に対する引数 irq()が複数回使われています。
例:

```
//oops -- is one of those sources wrong?

void __interrupt(irq(TMR0,TMR0)) tc0Int(void)
```

(2007) __interrupt attribute/specifier has a duplicate “base” (*) (パーサー)
__interrupt()指定子に対して同じ引数 base()が複数回使われています。
例:

```
//oops -- is one of those base addresses wrong?

void __interrupt(irq(TMR0), base(0x100,0x100)) tc0Int(void)
```

(2008) unknown “irq” (*) in __interrupt attribute/specifier (パーサー)
__interrupt()指定子に対する引数 irq()で使われているシンボルまたは値は、このデバイス上の割り込み要因に対応していません。
例:

```
//oops -- what interrupt source is TODO?

void   __interrupt(irq(TODO),high_priority) tc0Int(void)
```

(2009) * attribute/specifier has a misplaced number (*) (パーサー)
属性内で数値が使われていますが、属性に数値は期待していません。
例:

```
//oops -- this specifier requires specific argument, not a number
void   __interrupt(0) isr(void)
```

(2010) __interrupt attribute/specifier contains a misplaced interrupt source name (*) (パーサー)
割り込み要因名は、irq()に対する引数としてのみ使えます。
例:

```
//oops -- base() needs a vector table address

void __interrupt(irq(TMR0), base(TMR0)) tc0Int(void)
```

(2011) __interrupt attribute/specifier has a base (*) not supported by this device (パーサー)
__interrupt()指定子に対する引数 base()で指定されたアドレスは、ターゲット デバイスに対して無効です。例えば、IVTBASE レジスタのリセット値より低いアドレスが指定されると、このエラーが発生します。
例:

```
//oops -- the base() address is too low

void __interrupt(irq(TMR0), base(0x00)) tc0Int(void)
```

(2012) * attribute/specifier is only applicable to functions (パーサー)
__interrupt()指定子が関数以外に対して使われています。
例:

```
// oops -- foobar is an int, not an ISR

_ __interrupt(irq(TMR0)) int foobar;
```

(2013) argument “*” used by “*” attribute/specifier not supported by this device (パーサー)
この指定子の引数はターゲット デバイスに対して無効です。
例:

```
// oops -- base() can’t be used with a device that does not

// support vectored interrupts

void  __interrupt(base(0x100)) myMidrangeISR(void)
```

(2014) interrupt vector table @ 0x* already has a default ISR “*” (コード ジェネレータ)
ベクタテーブル内で指定されていないベクタ位置に対しては、既定値割り込み関数を 1 つだけ指定できます。これが 2 回指定されている場合、各既定値に対して正しい base()アドレスが指定されているか確認してください。
例:

```
void __interrupt(irq(default), base(0x100)) tc0Int(void) { ...

void __interrupt(irq(default), base(0x100)) tc1Int(void) { ...
// oops -- did you mean to use different different base() addresses?
```

(2015) interrupt vector table @ 0x* already has an ISR (*) to service IRQ * (*) (パーサー、コード ジェネレータ)
同じベクタテーブル内の特定割り込み要因に対して複数の割り込み関数が指定されています。
例:

```
void __interrupt(irq(TMR0), base(0x100)) tc0Int(void) { ...

void __interrupt(irq(TMR0), base(0x100)) tc1Int(void) { ...
// oops -- did you mean to use different different base() addresses?
```

(2016) interrupt function “*” does not service any interrupt sources (コード ジェネレータ)
割り込み関数が定義されていますが、この関数で処理する割り込み要因が指定されていません。引数 irq()を使って割り込み要因を指定してください。
例:

```
//oops -- what interrupt does this service?

void __interrupt(low_priority, base(0x100)) tc0Int(void)
```

(2017) config programming has disabled multi-vectors, “irq” in __interrupt attribute/specifier is ignored (コード ジェネレータ)
割り込み関数が引数 irq()を使って割り込み要因を指定していますが、ベクタテーブルはコンフィグレーション ビットによって無効にされています。ベクタ割り込みを有効にするか、__interrupt()指定子内で priority キーワードを使って割り込み要因を指定してください。
例:

```
#pragma config MVECEN=0

void __interrupt(irq(TMR0), base(0x100)) tc0Int(void)

// oops -- you cannot disable the vector table then allocate interrupt

// functions a vector source using irq()
```

(2018) interrupt vector table @ 0x* has multiple functions (* and *) defined at interrupt level * (コード ジェネレータ)
レガシーモードで動作するデバイス向けのプログラムにおいて、同じ割り込み優先度を持つ複数の関数が同じベクタテーブル内に存在します。レガシーモードの場合、各ベクタテーブルには、各優先度に対して最大で 1 つの割り込み関数しか格納できません。
例:

```
#pragma config MVECEN=0

void __interrupt(high_priority) tc0Int(void) {...

void __interrupt(high_priority) tc1Int(void) {...
```

(2019) * interrupt vector in table @ 0x* is unassigned, will be programmed with a * (コード ジェネレータ)
レガシーモードで動作するデバイス向けのプログラムにおいて、このベクタテーブル内の 1 つの割り込みベクタにアドレスが書き込まれていません。コンパイラは、-mundefints オプションによって指定されたアドレスをこのベクタに書き込みます

(2020) IRQ * (*) in vector table @ 0x* is unassigned, will be programmed with the address of a * (コード ジェネレータ)
このベクタテーブル内の割り込みベクタにアドレスが書き込まれていません。コンパイラは、-mundefints オプションによって指定されたアドレスをこのベクタに書き込みます。

(2021) invalid runtime “*” sub-option argument (*) (ドライバ)
--RUNTIME オプションのサブオプションに対する引数が無効です。
例:

```
--RUNTIME=default,+ivt:reset
```

ivt サブオプションの引数は数値アドレスである事が必要です。

(2022) runtime sub-option “ivt” specifies a base address (0x*) not supported by this device (ドライバ)
ivt サブオプションで指定されたアドレスは、ターゲット デバイスに対して無効です。例えば、IVTBASE レジスタのリセット値より低いアドレスが指定されると、このエラーが発生します。

(2023) IVT @ 0x* will be selected at startup (コード ジェネレータ)
複数の IVT が定義されたソースコードにおいて、起動時に選択するテーブルのアドレスが指定されていません(--RUNTIME オプションに対する ivt サブオプションで指定)。これが指定されない場合、コンパイラは最低アドレスを持つ IVT を選択します。このオプションを使う場合、常にテーブルアドレスを指定する事を推奨します。

(2024) runtime sub-option “ivt” specifies an interrupt table (@ 0x*) that has not been defined (ドライバ)
--RUNTIME オプションに対する ivt サブオプションにより IVT アドレスが指定されていますが、ソースコード内でこのアドレスに対して ISR が何も指定されていません。オプション内のこのアドレスが正しいかどうか、または__interrupt()指定子に対する引数 base()が正しく指定されているかどうかを確認してください。
例:

```
--RUNTIME=+ivt:0x100
```

ソースコード内でこのベースアドレスは使われていません。

(2025) qualifier * on local variable “*” is not allowed and has been ignored (パーサー)
auto またはローカル static 変数に対してこの修飾子は使えません。このメッセージは、これらの変数に与えられた修飾子が無視された事を示します。
例:

```
near int foobar; // oops -- auto variables cannot use near
```

(2026) variables qualified “*” are not supported for this device (パーサー)
この変数修飾子はこのターゲット デバイスでは使えません。
例:

```
eeprom int serialNo; // oops -- can’t use eeprom with PIC18 devices
```

(2027) initialization of absolute variable “*” in * is not supported (コード ジェネレータ)
この変数は、メモリ空間内で絶対アドレス変数として指定できません。
例:

```
eeprom char foobar   at(0x40) = 99; // oops - absolute can’t be eeprom
```

(2028) external declaration for identifier “*” doesn't indicate storage location (コード ジェネレータ)
外部オブジェクト(例: アセンブリコード内で定義されたオブジェクト) の宣言内で、そのオブジェクトを格納するメモリ空間が指定されていません。このオブジェクトへのアクセスは失敗する可能性があります。必要に応じて const またはバンク指定子を使ってください。
例:

```
extern int tapCounter; // oops - how does the compiler access this?
```

(2029) a function pointer cannot be used to hold the address of data (パーサー)
関数ポインタは関数のアドレスしか保持できません(変数またはオブジェクトのアドレスを保持する事はできません)。
例:

```
int (*fp)(int);
int foobar;
fp = &foobar;  // oops - a variable’s address cannot be assigned
```

(2030) a data pointer cannot be used to hold the address of a function (パーサー)
データポインタ(汎用 void *ポインタを含む)を使って関数のアドレスを保持する事はできません。
例:

```
void *gp;
int myFunc(int);
gp = foobar;  // oops - a function’s address cannot be assigned
```

(2033) recursively called function might clobber a static register it has allocated in expression (コード ジェネレータ)
再帰的に呼び出される関数内で定義された式によってレジスタが使われており、その式は同じ関数の呼び出しを必要とするより大きな式の一部となっています。このレジスタは上書きされ、コード実行に失敗する可能性があります。
例:

```
unsigned long fib_rec(unsigned long n)
{
    // the temporary result of the LHS call to fib_rec() might
    // store the result in a temp that is clobbered during the RHS
    // call to the same function
    return ((n > 1) ?(fib_rec(n-1) + fib_rec(n-2)) : n);
}
```

(2034) 24-bit floating-point types are not CCI compliant; use 32-bit setting for compliance (パーサー)
CCI は 24 ビット浮動小数点型をサポートしません。CCI に準拠する必要がある場合、-no-short-float および -no-short-double オプションを使う事で、float および double 型に対して IEEE 標準の 32 ビット浮動小数点型を適用してください。

(2035) use of sizeof() in preprocessor expressions is deprecated; use __SIZEOF_*__ macro to avoid this warning (プリプロセッサ)
プリプロセッサによる評価が必要な式内での sizeof()の使用はサポートされなくなりました。代わりに、コンパイラによって定義されるプリプロセッサ マクロ(__SIZEOF_INT__等)が使えます。これは、通常の方法で使われる C 演算子 sizeof()には影響しません。
例:

```
#if (sizeof(int) > 2)   // oops -- use (  SIZEOF_INT   > 2) instead
```

(2036) use of @ is not compliant with CCI, use __at() instead (パーサー)
CCI は、@ *address* コンストラクタを使う絶対アドレス関数およびオブジェクトの定義をサポートしません。代わりに、定義内の識別子の後で__at(*address*)を使います。
例:

```
int foobar @ 0x100; // oops -- use   at(0x100) instead
```

(2037) short long integer types are not compliant with CCI (パーサー)
CCI は 3 バイトの short long 型をサポートしません。代わりに、等価な long int 型の使用を検討してください。
例:

```
short long input; // oops -- consider input to be long when using CCI
```

(2038) use of short long integer types is deprecated; use __int24 or __uint24 to avoid this warning (パーサー)
short long 型指定子は、移植性の高い__int24 型(short long の代替)と__uint24 型(unsigned short long の代替)に置換されました。
例:

```
short long input; // oops -- use   int24 as the type for input
```

(2039) __int24 integer type is not compliant with CCI (パーサー)
CCI は 3 バイトの__int24 型をサポートしません。代わりに long int 型を使ってください。
例:

```
__int24 input; // oops -- use a long type when using CCI
```

(2040) __uint24 integer type is not compliant with CCI (パーサー)
CCI は 3 バイトの__uint24 型をサポートしません。代わりに unsigned long int 型を使ってください。
例:

```
__uint24 input; // oops -- use an unsigned long type when using CCI
```

(2041) missing argument after "*" (ドライバ)
このオプションは引数を必要としますが、コマンドラインで引数が何も指定されていません。
例:

```
xc8-cc -mcpu=18f4520 -Wl,-Map main.c
```

-Map オプションにはマップファイルの名前を指定する必要があります(例: -Wl,-Map=proj.map)。

(2042) no target device specified; use the -mcpu option to specify a target device (ドライバ)
どのデバイス向けにビルドを行うのか選択されずにドライバが呼び出されました。-mprint-devices オプションを指定してドライバを実行すると、ビルド用に選択可能な全てのデバイスが表示されます。
例:

```
xc8-cc main.c
```

-mcpu オプションを使って、どのデバイス向けにビルドするのか指定する必要があります。

(2043) target device was not recognized (ドライバ)
トップレベル ドライバは、-mcpu オプションで指定されたデバイスのファミリを認識できませんでした。
例:

```
xc8-cc -mcpu=pic io.c
```

-mprint-devices によって表示されたデバイス名の 1 つを正確に指定する必要があります。

(2044) unrecognized option “*” (ドライバ)
トップレベル ドライバは指定されたオプションを認識できませんでした。このオプションは、この後のコンパイラ ツールチェーンに渡されますが、エラーまたは予期せぬ振る舞いの原因となる可能性があります。

(2045) could not find executable “*” (ドライバ)
トップレベル ドライバは、指定されたコンパイラツールを通常の位置で見つける事ができませんでした。コンパイラ インストール ディレクトリ内のファイルまたはディレクトリが他の場所に移動していないか確認してください。

(2046) identifier length must be between * and *; using default length * (ドライバ)
識別子の最大有意長として指定された文字数は無効であるため、既定値の 255 が適用されました。
例:

```
-N=16
```

この長さは 32～255 の整数で指定する必要があります。

(2047) 24-bit floating point types are not supported when compiling in C99 (ドライバ)
C99 規格向けにコンパイルする場合、`float` および `double` 型は 32 ビット幅である事が必要です。 24 ビット浮動小数点型が必要である場合、C90 準拠ライブラリ(`-mc90lib` オプション)を選択できるかもしれません。これを選択できない場合は、C90 向けにコンパイルする必要があります。
例:

```
xc8-cc -mcpu=18f4520 -fshort-double main.c
```

C99 規格で 24 ビット `double` 型を使う事はできません。

(2048) C language extension “*” is not supported and will be ignored (ドライバ)
指定された言語拡張はサポートされません。
例:

```
xc8-cc -mcpu=16f1937 -mext=iar main.c
```

この言語拡張はサポートされません。

(2049) C99 compliant libraries are currently not available for baseline or mid-range devices, or for enhanced mid-range devices using a reentrant stack; using C90 libraries (ドライバ)
現在、一部のデバイスでは C99 準拠ライブラリを利用できません。それらのデバイスでは、C90 準拠ライブラリを使ってソースコードを C99 規格向けにビルドできます。あるいは、C90 規格向けにビルドする事も可能です。

(2050) use of the -mcci option is deprecated; use -mext=cci to avoid this warning (ドライバ)
CCI (Common C Interface)を選択するには、常に`-mext=cci` オプションを使ってください。

(2051) The current license does not permit the selected optimization level* (ドライバ)
このコンパイラのライセンスでは、要求されたコンパイラ動作モードを使えません。
例:

```
xc8-cc -mcpu=18f4520 -Os main.c
```

コンパイラのライセンスを取得していない場合、レベル「`s`」の最適化を選択する事はできません。

(2052) The current license does not permit the selected optimization level and other levels are not permitted by the NOFALLBACK option (ドライバ)
このコンパイラのライセンスでは、要求されたコンパイラ動作モードを使えません。`--nofallback` オプションが有効であるため、コンパイラは下位の最適化レベルへフォールバックせずにこのエラーを生成しました。コンパイラ ライセンスが要求した最適化レベルをサポートしているにも関わらずこのエラーが発生する場合、コンパイラが正しくアクティベートされていない可能性があります。

(2053) function “*” is never called (コード ジェネレータ)
指定されたインライン関数は 1 度も呼び出されていないため、この関数のコードは生成されません。このメッセージは、メッセージ(520)とは異なり、インライン関数を対象とします。全てのインライン関数に対してこのメッセージを無効にできます。その場合も、その他の全ての未使用関数に対してメッセージ(520)を発行できます。

(2054) the language standard “*” is not supported; using C99 (ドライバ)
コンパイラは-std オプションで指定された言語規格をサポートしていません。代わりに C99 規格が使われます。
例:

```
xc8-cc -mcpu=12f510 -std=c11 main.c
```

C11 規格を選択する事はできません。

(2056) use of the -fmode option is deprecated; use -O to control optimizations and avoid this warning (ドライバ)

実行する最適化を指定するためにモード(-fmode オプション)と最適化(-O オプション)の両方を指定する必要はなくなりました。 全ての最適化は、コンパイラの-O オプションを使って最適化レベルを選択する事により制御可能です。ただし、ライセンスなしのコンパイラでは一部の最適化レベルを利用できません。

(2057) The XC8 compiler installation appears to be corrupted. Please reinstall and try again (ドライバ)

コンパイラは、インストールに何らかの問題がある事を検出しました。コンパイラ アプリケーションが削除されたか移動された可能性があります。

(2058) function "*" cannot be inlined with code coverage enabled (コード ジェネレータ)
コードカバレッジ機能が有効である場合、関数のインライン化はできません。このメッセージは、コードカバレッジ機能が有効である限りこれらの関数はインライン化されないという事をユーザに示すためのアドバイザリ メッセージです。

(2059) conflicting * register values found in Start Segment Address record (3) (Hexmate)
Hexmate は Hex ファイル内の全てのタイプ 3 レコードをそのまま通過させますが、これらのレコード内で CS または IP レジスタ向けに指定された値に矛盾があると、このエラーを発行します。

(2060) CRC polynomial unspecified or set to 0 (Hexmate)
この警告は、多項式値が 0 に設定された状態で Hexmate を使って CRC ハッシュ値を計算しても意味のあるハッシュ結果は得られないという事を示します。通常、これはチェックサム オプションで多項式値を設定し忘れた場合に発生します。

(2061) word width required when specifying reserve byte order hash (Hexmate)
Hex ファイル内の CRC 読み出しデータワードを逆順に計算する場合、Hexmate の -CK オプションに対する r サブオプションでワード幅を指定する必要があります。コンパイラ ドライバを使う場合、これは-mchecksum オプションに対する revword サブオプションを使って指定します。

(2062) word width must be * when specifying reserve byte order hash (Hexmate)
Hex ファイル内の CRC 読み出しデータワードを逆順に計算する場合、ワード幅はこのメッセージ内に示された値のいずれかである事が必要です。この値は、Hexmate の-CK オプションに対する r サブオプションで指定します。コンパイラ ドライバを使う場合、これは-mchecksum オプションに対する revword サブオプションを使って指定します。

(2063) * address must be a multiple of the word width when performing a reverse byte order hash (Hexmate)
Hex ファイル内の CRC 読み出しデータワードを逆順に計算する場合、開始アドレスと終了アドレスはワード幅 (Hexmate の-CK オプションまたはコンパイラの-mchecksum オプションにて指定) の倍数である事が必要です。

(2064) PIC18 extended instruction cannot be used when the standard instruction set is selected (アセンブラ)
PIC18 アセンブリ プロジェクトは、標準命令セットまたは拡張命令セットのどちらかを使うようにセットアップする必要があります。 -misa オプションを使って拡張命令セットを有効にせずに拡張命令を使うと、このメッセージが表示されます。

(2065) offset out of range (アセンブラ)
この拡張 PIC18 命令のファイルレジスタ アドレスはレンジ外です。
例:

```
clrf [200] ; oops
; for extended instructions, the file operand must be less than, for example, 0x60
```

(2066) MESSG directive:* (アセンブラ)
これは MESSG アセンブラ ディレクティブの出力メッセージです。

(2067) ERROR directive:* (アセンブラ)
これは ERROR アセンブラ ディレクティブの出力です。

(2068) use of the opt control "*" is deprecated; use the corresponding directive (アセンブラ)
OPT *CONTROL* 形式のアセンブラ制御は廃止となりました。代わりに、OPT トークンを省略した等価なディレクティブが利用できます(例: OPT TITLE "My great project"の代わりに TITLE "My great project"を使用)。

(2069) use of the radix option of the list control is deprecated; use the radix directive (アセンブラ)
以前は LIST アセンブラ制御により、LIST オプションに対する引数 r を使って入力ソースを指定できましたが、現在は代わりに RADIX ディレクティブを使う必要があります(例: OPT LIST r=hex の代わりに RADIX hex を使用)。

(2070) device specified by the PROCESSOR directive conflicts with that set by the -mcpu option (アセンブラ)
-mcpu ドライバ オプションは、ビルドのターゲット デバイスを設定します。必要に応じて PROCESSOR ディレクティブを使う事で、ソースファイルが特定デバイス向けにのみビルドされるようにできます。-mcpu オプションと PROCESSOR ディレクティブで指定されたデバイスが一致しないと、このメッセージが表示されます。

(2071) could not find record containing hash starting address 0x* (Hexmate)
Hexmate に対し、HEX ファイル内に存在しないアドレスで始まるデータからハッシュを計算するよう要求されました。

(2072) only SHA256 is currently supported (set width control to 256) (Hexmate)
SHA ハッシュ アルゴリズムが選択されている場合、width サブオプションは 256 または-256 のどちらかに設定するか、完全に省略する必要があります。

(2073) when compiling for C90, specifying an output file for dependencies is not supported and will be ignored (ドライバ)
clang フロントエンドのみが依存性情報を格納したファイルを生成できます。
例:

```
xc8-cc -mcpu=18f4520 -MF depfile -std=c90 main.c
```

-std=c90 と一緒に-MF オプションは使えません。

(2074) word size for byte skip with hash calculation must be * (Hexmate)
-CK オプションの s サブオプション(ハッシュ計算中のバイトスキップに適用するワードサイズを指定) に対する引数が無効な値です。
例:

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s1
```

s に対する引数は 1 より大きい事が必要です。

(2075) word size required when requesting byte skip with hash calculation (Hexmate)

-CK オプションの s サブオプション(ハッシュ計算中のバイトスキップに適用するワードサイズを指定) には引数が必要です。

例:

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s.2
```

s の直後に値が必要です(例: s4.2)。

(2076) number of bytes for byte skip with hash calculation must be * (Hexmate)
各ワード内でスキップするバイトの数が無効です。
例:

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s4.4
```

スキップするバイトの数は 4 (ワード幅) 未満である事が必要です(例: s4.2)。

(2077) number of bytes required when requesting byte skip with hash calculation (Hexmate)
-CK オプションの s サブオプション内の「.」の後に引数が必要です。この引数は、ハッシュ値の計算中に各ワード内でスキップするバイトの数を指定します。
例:

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s4.
```

s 引数内の「.」の後に値が必要です (例: s4.2)。

(2078) the number of hash bytes to which the trailing code is appended (*) must be no greater than the hash width (*) (Hexmate)
指定されたバイト数のハッシュ値の後にコードを追加するよう要求されましたが、このバイト数はハッシュ値全体のバイト数を超えています。
例:

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021t00.4
```

ハッシュ値が 2 バイト長しかない場合、4 バイトごとにコードを追加するよう指定しても意味がありません。例えば、t00.1 と指定すると、各バイトにコードを追加できます。

(2079) the hash width (*) must be a multiple of the number of hash bytes appended with a trailing code (*) (Hexmate)
指定されたバイト数のハッシュ値の後にコードを追加するよう要求されましたが、このバイト数はハッシュ値の幅の約数ではありません。
例:

```
hexmate main.hex -ck=0-ff@100+ffffg5w-4p1021t00.3
```

ハッシュ値が 4 バイト長である場合、3 バイトごとにコードを追加するように指定しても意味がありません。例えば、t00.2 と指定すると 2 バイトごとにコードを追加できます。

(2080) WARN:* (アセンブラ)
プログラマによって意図的に生成される警告となるアセンブリ ディレクティブが存在します。ソースコードを調べ、この警告についてさらに調査が必要かどうか判断してください。ディレクティブが適切でない場合、削除する事を検討してください。
例:

```
WARN "I have not yet confirmed that this is the correct threshold"
movlw 22
movwf threshold,c
```

(2081) the device architecture "*" does not match previously encountered object files "*" (リンカ)
異なるターゲット デバイス向けにビルドされたオブジェクト ファイルがリンカに渡されました。これは、プリコンパイル済みのオブジェクト ファイルまたはアセンブラコードからビルドされたライブラリが使われている場合にのみ発生します。全てのモジュールが同じデバイス向けにビルドされている事を確認してください。

(2082) can't create temporary file (ドライバ)
ドライバは一時ファイルの生成に失敗しました。ディスク容量の問題や、一時ファイルの生成先ディレクトリへのアクセス権の問題等、各種の原因が考えられます。

(2083) the current license does not permit the "*" feature (ドライバ)
コンパイラ ライセンスが承認していない機能が使われています。一部のコンパイラ機能には PRO コンパイラ ライセンスが必要です。適切なライセンスを保有しているのにこのメッセージが表示される場合、コンパイラが正しくインストールされているか、およびライセンスの期限が切れていないか確認してください。

```
xc8-cc -mcpu=16f1937 -mchp-stack-usage main.c perip.c
```

例えば、スタック ガイダンス機能を使うには PRO コンパイラ ライセンスが必要です。

(2084) absolute object "*" has an address that lies within memory utilized by the compiler and will likely lead to code failure (コード ジェネレータ)
__at()を使って宣言された絶対アドレス オブジェクトが、コンパイラによって使われるアドレスに配置されています。このオブジェクトは、コンパイラが生成したコードによって実行時に破壊され、エラーが発生する可能性があります。
例:

```
int   at(0x0) myVar;  // oops - not a good address
```

(2085) the BANKISEL directive has no effect with the currently selected device and will be ignored (アセンブラ)
BANKISEL アセンブラ ディレクティブは、ベースライン デバイスとミッドレンジ デバイスにのみ必要です。その他のデバイスではこのディレクティブは無視されますが、デバイス データシートを参照して、オブジェクトの間接アクセスを実行するコードがそのデバイスに対して有効かどうか確認してください。
例:

```
  BANKISEL myVar ;oops - this device does not use separate indirect access bank bits
  movwf INDF
```

(2086) memory for a heap has been reserved in response to the detection of calls to malloc/calloc/realloc/free function(s) (ドライバ)
-mheap オプションで auto サイズ値が設定されている場合に、標準の動的メモリ割り当て関数に対する呼び出しが検出されました。コンパイラが決定したサイズのメモリがヒープ用に予約されました。

(2087) memory for the * software stack has been reserved in response to the detection of functions built for reentrancy (ドライバ)
-mheap オプションで auto サイズ値が設定されている場合に、再入可能モデル (ソフトウェア スタック)を使ってビルドされた関数が検出されました。コンパイラが決定したサイズのメモリが、このスタック用に予約されました。

11. 処理系定義の振る舞い

以下では、C 言語規格に示される「処理系定義の振る舞い」に対して本コンパイラが選択した振る舞いについて説明します。

11.1 概要

ISO C に準拠する処理系は、「処理系定義」(implementation-defined)として定義された振る舞いをどのように選択したか明確に示す必要があります。以下には全ての領域における「処理系定義の振る舞い」と本コンパイラで選択された振る舞い、および ISO/IEC 9899:1999 (aka C99)または ISO/IEC 9899:1990 (aka C90) 規格内の対応するセクション番号を丸カッコで囲んで記載しています。

11.2 コンパイル

ISO 規格	処理系
「解析メッセージの識別方法(3.10、5.1.1.3)」	
	コマンドライン上でコンパイルを行う場合、既定値により以下の書式が使われます。 文字列(warning)は、警告メッセージ向けにのみ表示されます。 *filename*: *function*() *linenumber*: *source line* ^ (*ID*) *message* (warning) または filename: linenumber: (ID) message (warning) *filename* はコードを格納したファイルの名前です(メッセージが特定のファイルに関連しない場合は空白)。*linenumber* はコードの行番号です(メッセージが特定のファイルに関連しない場合は空白)。*ID* はメッセージを識別するための一意番号です。*message* は解析メッセージです。
「コンパイル フェイズ 3 において、改行文字を除く空白類文字の各並びを保持するか一つの空白文字に置き換えるか(5.1.1.2)」	
	本コンパイラは、先頭または中間位置に存在する空白類文字の各並びを 1 つの空白文字に置き換えます。末尾の空白類文字の並びは改行文字に置き換えられます。

11.3 環境

ISO 規格	処理系
「コンパイル フェイズ 1 での、物理的なソースファイルのマルチバイト文字と対応するソース キャラクタセットのマッピング方法(5.1.1.2)」	
	ソースファイル内のマルチバイト文字をサポートしません。
「フリースタンディング環境(ターゲット デバイス環境)においてプログラム開始時に呼び出される関数の名前と型(5.1.2.1)」	
	int main (void);
「フリースタンディング環境におけるプログラム終了処理の効果(5.1.2.1)」	
	リセットベクタ位置への分岐によりソフトリセットを実装します。
「main 関数を定義可能な代替方法(5.1.2.2.1)」	
	void main (void);

.....続き

ISO 規格	処理系
「main 関数の argv 引数が指す文字列の値(5.1.2.2.1)」	
	main に引数を渡しません。 argc または argv への参照は未定義です。
「対話型装置の構成(5.1.2.3)」	
	アプリケーション定義です。
「シグナルのセット、それらの意味、既定の処理方法(7.14)」	
	シグナルは未実装です。
「計算例外に対応する SIGFPE、SIGILL、SIGSEGV 以外のシグナル値(7.14.1.1)」	
	シグナルは未実装です。
「プログラム開始時に実行される signal(sig, SIG_IGN); と同等なシグナル(7.14.1.1)」	
	シグナルは未実装です。
「getenv 関数によって使われる環境名のセットと環境リストを変更するための方法(7.20.4.5)」	
	ホスト環境はアプリケーション定義です。
「system 関数による文字列の実行方法(7.20.4.6)」	
	ホスト環境はアプリケーション定義です。

11.4 識別子

ISO 規格	処理系
「識別子に使えるマルチバイト文字と、それらの文字と国際文字名(UCN)との間の対応(6.4.2)」	
	そのようなマルチバイト文字はありません。
「識別子の有意先頭文字数(5.2.4.1、6.4.2)」	
	全ての文字が有意です。

11.5 文字

ISO 規格	処理系
「1 バイト内のビット数(C90 3.4、C99 3.6)」	
	8 ビットです。
「実行キャラクタセットの要素の値(C90/C99 5.2.1)」	
	実行キャラクタセットは ASCII です。
「標準の各英字エスケープ シーケンスに割り当てられた実行キャラクタセットの要素の一意値(C90/C99 5.2.2)」	
	実行キャラクタセットは ASCII です。
「基本実行キャラクタセットに含まれない文字を格納した char オブジェクトの値(C90 6.1.2.5、C99 6.2.5)」	

© 2022 Microchip Technology Inc. and its subsidiaries

.....続き

ISO 規格	処理系
	char オブジェクトの値は、ソース キャラクタセット内の文字の 8 ビットバイナリ表現です。 つまり、コンパイルしません。
「signed char 型と unsigned char 型のどちらが単なる char 型と同じレンジ、表現、振る舞いを持つか(C90 6.1.2.5、C90 6.2.1.1、C99 6.2.5、C99 6.3.1.1)」	
	既定値により、単なる char 型と機能的に等価なのは unsigned char 型です。 - funsigned-char および-fsigned-char オプションを使って明示的に型を指定できます。
「文字定数と文字列リテラル内のソース キャラクタセットの要素から実行キャラクタセットの要素への対応付け(C90 6.1.3.4、C99 6.4.4.4、C90、C99 5.1.1.2)」	
	ソース キャラクタセットのバイナリ表現は、実行キャラクタセットへと維持されます。
「複数の文字を格納した整数文字定数の値。または、シングルバイト実行キャラクタセットに割り当てられていない 1 文字またはエスケープ シーケンスを格納した整数文字定数の値(C90 6.1.3.4、C99 6.4.4.4)」	
	直前の値は 8 ビット左にシフトされ、次の文字のビットパターンが取り込まれます。最終的な結果は int 型です。 結果が int 型で表現できなくなると警告が出力され、その値は int 型のサイズに切り詰められます。
「複数のマルチバイト文字を格納したワイド文字定数の値。または、拡張実行キャラクタセットに割り当てられていない 1 つのマルチバイト文字またはエスケープ シーケンスを格納したワイド文字定数の値(C90 6.1.3.4, C99 6.4.4.4)」	
	ソースファイル内のマルチバイト文字をサポートしません。
「拡張実行キャラクタセットのメンバーに割り当てる 1 つのマルチバイト文字で表現されたワイド文字定数を、対応するワイド文字コードに変換するために使用される現在のロケール(C90 6.1.3.4、C99 6.4.4.4)」	
	ソースファイル内のマルチバイト文字をサポートしません。
「ワイド文字列リテラルを対応するワイド文字コードに変換するために使う現在のロケール(C90 6.1.4、C99 6.4.5)」	
	ワイド文字列をサポートしません。
「実行キャラクタセットに割り当てられていないマルチバイト文字またはエスケープ シーケンスを格納した文字列リテラルの値(C90 6.1.4, C99 6.4.5)」	
	ソースファイル内のマルチバイト文字をサポートしません。

11.6 整数

ISO 規格	処理系
「処理系で使われる拡張整数型(C99 6.2.5)」	
	__bit キーワードはシングルビット整数型を指定します。 __int24 および__uint24 キーワードは、それぞれ符号付きおよび符号なしの 24 ビット整数型を指定します。
「符号付き整数型の表現に符号と絶対値、2 の補数、1 の補数のどれを使うか。また、異常値はトラップ表現か通常値か(C99 6.2.6.2)」	
	全ての整数型は 2 の補数として表現され、全てのビットパターンは通常値です。
「精度の等しい拡張整数型同士の順位付け(C99 6.3.1.1)」	

.....続き	
ISO 規格	**処理系**
	同じ精度を持つ拡張整数型は存在しません。
「整数型の値を符号付き整数型に変換した時にその値が符号付き整数型で表現できなかった場合の結果、あるいはその場合に生成されるシグナル(C90 6.2.1.2, C99 6.3.1.3)」	
	値 X を幅 N の型に変換した結果の値は、X の 2 の補数の最下位 N ビットです。つまり、X は N ビットに切り詰められます。シグナルは生成されません。
「符号付き整数に対するビット単位演算の結果(C90 6.3、C99 6.5)」	
	右シフト演算子(>>)は符号付き値を符号拡張します。 従って、`signed int` 型の値 0x0124 を持つオブジェクトを 1 ビット右シフトした値は 0x0092 であり、値 0x8024 を 1 ビット右シフトした値は 0xC012 です。 符号なし整数値を右シフトした結果の MSb は常にクリアされます。 符号付きまたは符号なし値を左シフト(<<)した結果の LSb は常にクリアされます。 その他のビット単位演算は、オペランドが符号なしであるかのように振る舞います。

11.7 浮動小数点数

ISO 規格	処理系
「浮動小数点型演算の精度と、浮動小数点型結果を返す<math.h>および<complex.h>内のライブラリ関数の精度(C90 および C99 5.2.4.2.2)」	
	精度は未知です。
「FLT_ROUNDS の非標準値によって特徴付けられる丸め処理の振る舞い(C90 および C99 5.2.4.2.2)」	
	そのような値を使いません。
「FLT_EVAL_METHOD の非標準の負値によって指定される評価手法(C90/C99 5.2.4.2.2)」	
	そのような値を使いません。
「整数を浮動小数点型に変換して元の値を厳密に表現できない場合の丸め処理の方向(C90 6.2.1.3、C99 6.3.1.4)」	
	整数は、最も近い浮動小数点表現へと丸め処理されます。
「浮動小数点値をより低精度の浮動小数点値に変換する際の丸め処理の方向(C90 6.2.1.4、6.3.1.5)」	
	浮動小数点値は、切り捨てる方向で低精度の値へと丸め処理されます。
「浮動小数点型定数に対して、最も近い表現可能な値を選択するか、最も近い表現可能な値のすぐ隣のより大きなまたはより小さな表現可能値を選択するか(C90 6.1.3.1、C99 6.4.4.2)」	
	該当せず: FLT_RADIX は 2 のべき乗です。
「FP_CONTRAC プラグマによって禁止されない場合に、浮動小数点型の式を短縮するかどうかと、どのように短縮するか(C99 6.5)」	
	このプラグマは実装していません。
「FENV_ACCESS プラグマの既定値ステート(C99 7.6.1)」	
	このプラグマは実装していません。
「追加の浮動小数点例外、丸め処理モード、環境、分類と、それらのマクロ名(C99 7.6、7.12)」	

.....続き

ISO 規格	処理系
	どれもサポートしません。
「FP_CONTRACT プラグマの既定値ステート(C99 7.12.2)」	
	このプラグマは実装していません。
「丸め処理結果が IEC 60559 準拠の処理系における算術結果と実質的に等しい場合に「不正確」浮動小数点例外が発生するかどうか(C99 F.9)」	
	例外は発生しません。
「結果がわずかに不正確であるものの、IEC 60559 準拠の処理系では不正確とはならない場合、「アンダーフロー」(および「不正確結果」)浮動小数点例外が発生するかどうか(C99 F.9)」	
	例外は発生しません。

11.8 配列とポインタ

ISO 規格	処理系
「ポインタから整数への(またはその逆の)変換の結果(C90 6.3.4、C99 6.3.2.3)」	
	プログラム全体を通してポインタ変数にデータメモリまたはプログラムメモリのどちらか一方のメモリ内のオブジェクトのアドレスのみが代入される場合、整数をポインタへ変換する際に整数のアドレスが変更される事なくポインタ変数へ複製されます。 プログラム全体を通してポインタ変数にデータメモリとプログラムメモリの両方のメモリ内のオブジェクトのアドレスが代入される場合、整数値が const 型を指すポインタへと明示的に型変換されると整数値の MSb はセットされます。これ以外の場合、MSb はセットされません。整数の残りのビットは、変更される事なくポインタ変数へ代入されます。 ポインタを整数へ変換する場合、ポインタが保持している値は変更される事なく整数に代入されます。整数がポインタのサイズよりも大きい場合、通常の切り捨てが適用されます。
「同じ配列の要素を指す 2 つのポインタの減算結果のサイズ(C90 6.3.6、C99 6.5.6)」	
	符号付き整数結果のサイズは、減算式内のポインタ オペランドと同じです。

11.9 ヒント

ISO 規格	処理系
「register 記憶域クラス指定子の有効性(C90 6.5.1、C99 6.7.1)」	
	register 記憶域クラス指定子は効果を有しません。
「inline 関数指定子の有効性(C99 6.7.4)」	
	ライセンスありのコンパイラで最適化レベルを 2 以上に設定した場合、関数はインライン化される可能性があります。その他の状況で関数がインライン化される事はありません。

11.10 構造体、共用体、列挙体、ビットフィールド

ISO 規格	処理系
「単なる int 型のビットフィールドは signed int 型または unsigned int 型のどちらとして扱われるか(C90 6.5.2、C90 6.5.2.1、C99 6.7.2、C99 6.7.2.1)」	
	単なる int 型のビットフィールドを符号なし整数型として扱います。 符号付き整数ビットフィールドはサポートしません。
「_Bool、signed int、unsigned int の他に許容されるビットフィールドの型(C99 6.7.2.1)」	
	signed char および unsigned char 型が許容されます。
「ビットフィールドが記憶域単位の境界をまたぐ事ができるかどうか(C90 6.5.2.1、C99 6.7.2.1)」	
	ビットフィールドが記憶域単位の境界をまたぐ事はできません。記憶単位(例: 8bit) の境界をまたぐ可能性のある全てのビットフィールドは、新しい単位の LSb 位置に配置されます。
「ユニット内のビットフィールドの割り当て順(C90 6.5.2.1、C99 6.7.2.1)」	
	構造体で最初に定義されているビットフィールドは、記憶域単位内の LSb 位置に配置されます。後続のビットフィールドは、上位側のビットへ向かって配置されます。
「構造体の非ビットフィールド メンバーの整列(C90 6.5.2.1、C99 6.7.2.1)」	
	整列は実行されません。
「各列挙型と互換性のある整数型(C90 6.5.2.2、C99 6.7.2.2)」	
	列挙型を表現する型は、列挙値に応じて選択されます。列挙値に負の値が含まれる場合は符号付き型が選択され、その他の場合は符号なし型が選択されます。 列挙値が char 型により保持可能であれば char 型が選択され、その他の場合は int 型が選択されます。 列挙値は int 型により保持可能なレンジ内である事が必要です。そうではない場合、値は切り詰められます。

11.11 修飾子

ISO 規格	処理系
「volatile 修飾された型を持つオブジェクトへのアクセスは何によって構成するか(C90 6.5.3、C99 6.7.3)」	
	volatile 修飾されたオブジェクトの識別子に対する各参照がオブジェクトへのアクセスを構成します。

11.12 前処理ディレクティブ

ISO 規格	処理系
「ファイル名、ヘッダ定義名の両方の書式において、文字の並びをヘッダまたは外部ソースファイルの名前に対応付ける方法(C90 6.1.7、C99 6.4.7)」	
	区切り文字の間の文字の並びは、ホスト環境向けのファイル名を表す文字列であると見なします。
「条件付きインクルードを制御する定数式の中の文字定数の値が実行キャラクタセット内の同じ文字定数の値と一致するかどうか(C90 6.8.1, C99 6.10.1)」	
	一致します。

.....続き

ISO 規格	処理系
「条件付きインクルードを制御する定数式の中の文字定数(1 文字) は負値を持つ事ができるかどうか(C90 6.8.1、C99 6.10.1)」	
	負値を持つ事ができます。
「# include 文の中の< >で囲まれたヘッダを検索する場所と、その場所の指定方法またはヘッダの識別方法(C90 6.8.2、C99 6.10.2)」	
	プリプロセッサは、-I オプションを使って指定された全てのディレクトリを検索し、-nostdinc オプションが使われていなければ、次に標準のコンパイラ インクルード ディレクトリ<*install directory*>/pic/include を検索します。
「# include 文の中の""で囲まれたソースファイルを検索する方法(C90 6.8.2、C99 6.10.2)」	
	コンパイラは、最初にインクルード ファイルを格納しているディレクトリを検索し、次に< >で囲まれたヘッダ向けの検索ディレクトリを検索します。
「前処理トークンをヘッダ名に組み込む方法(C90 6.8.2、C99 6.10.2)」	
	全てのトークン(空白類文字を含む) はヘッダファイル名の一部であると見なされます。区切り文字の囲みの中のトークンに対してマクロ展開を実行しません。
「#include 処理のネストの制限(C90 6.8.2、C99 6.10.2)」	
	制限はありません。
「#演算子が文字定数または文字列リテラル内の国際文字名の最初の\文字の前に\文字を挿入するかどうか(6.10.3.2)」	
	挿入しません。
「認識された各非 STDC #pragma ディレクティブに対する振る舞い(C90 6.8.6、C99 6.10.6)」	
	詳細は 5.13.3 「プラグマ ディレクティブ」を参照してください。
「コンパイル時の日付が得られない場合の __DATE__の定義と、コンパイル時の時刻が得られない場合の __TIME__の定義(C90 6.8.8、C99 6.10.8)」	
	コンパイル時の日付と時刻は常に得られます。

11.13 ライブラリ関数

ISO 規格	処理系
「ISO 仕様の第 4 節が要求する最小限のライブラリ機能の他にフリースタンディング環境プログラムが利用できるライブラリ機能(5.1.2.1)」	
	ターゲット デバイスに対応するコンパイラ ユーザガイドを参照してください。
「assert マクロが出力する解析情報の書式(7.2.1.1)」	
	Assertion failed:*expression* (*file*:*func*:*line*)
「fegetexceptflag 関数が保存する浮動小数点例外フラグの表現(7.6.2.2)」	
	未実装です。
「feraiseexcept 関数がオーバーフロー例外またはアンダーフロー例外の他に不正確結果例外を生成するかどうか(7.6.2.3)」	
	未実装です。
「"C"と""の他に setlocale 関数への引数として使える文字列(7.11.1.1)」	

.....続き

ISO 規格	処理系
	ありません。
「FLT_EVAL_METHOD マクロの値が 0 未満か 2 よりも大きい場合に float_t と double_t に定義される型(7.12)」	
	未実装です。
「この国際規格が要求する数学関数以外の数学関数の定義域エラー(7.12.1)」	
	ありません。
「定義域エラーが発生した場合に数学関数が返す値(7.12.1)」	
	定義域エラーが発生した場合、errno を EDOM に設定します。
「オーバーフローおよび/またはアンダーフロー レンジエラーが発生した時に、数学関数が errno を ERANGE マクロの値に設定するかどうか(7.12.1)」	
	設定します。
「fmod 関数の第 2 引数が 0 である場合に定義域エラーが発生するか、それとも 0 が返されるか(7.12.10.1)」	
	最初の引数が返されます。
「remquo 関数が商を削減する際に使う法の 2 を底とする対数(7.12.10.3)」	
	未実装です。
「シグナルハンドラの呼び出し前に signal(sig, SIG_DFL); と等価な動作が実行されるかどうか。これが実行されない場合、シグナルのブロッキングが実行されるかどうか(7.14.1.1)」	
	シグナルは未実装です。
「NULL マクロが展開される先の NULL ポインタ定数(7.17)」	
	((void*)0)
「テキストストリームの最終行に終端改行文字が必要かどうか(7.19.2)」	
	ストリームは未実装です。
「改行文字の直前でテキスト ストリームに書き出された空白文字は読み出し時に現れるかどうか(7.19.2)」	
	ストリームは未実装です。
「バイナリ ストリームに書き込まれるデータに付加可能な NULL 文字の数(7.19.2)」	
	ストリームは未実装です。
「アペンドモード ストリームのファイル位置インジケータがファイルの先頭または末尾位置に初期設定されるかどうか(7.19.3)」	
	ストリームは未実装です。
「テキスト ストリームへの書き込み時に、関連するファイルがその位置を越えたところで切り詰められるかどうか(7.19.3)」	
	ストリームは未実装です。
「ファイル バッファリングの特性(7.19.3)」	
	ファイルハンドルは未実装です。
「長さが 0 のファイルが実際に存在するかどうか(7.19.3)」	
	ファイルハンドルは未実装です。

.....続き

ISO 規格	処理系
「ファイルの命名規則(7.19.3)」	
	ファイルハンドルは未実装です。
「同じファイルを複数回開く事ができるかどうか(7.19.3)」	
	ファイルハンドルは未実装です。
「ファイル内のマルチバイト文字に使うエンコーディングの種類と選択(7.19.3)」	
	ファイルハンドルは未実装です。
「ファイルを開く際の remove 関数の影響(7.19.4.1)」	
	ファイルハンドルは未実装です。
「rename 関数の呼び出し前に新しい名前のファイルが存在する場合の影響(7.19.4.2)」	
	ファイルハンドルは未実装です。
「プログラムが異常終了した時に、開いていた一時ファイルが削除されるかどうか(7.19.4.3)」	
	ファイルハンドルは未実装です。
「tmpnam 関数の呼び出し回数が TMP_MAX を超えた場合の動作(7.19.4.4)」	
	ファイルハンドルは未実装です。
「どのモード変更がどのような状況で許容されるか(7.19.5.4)」	
	ファイルハンドルは未実装です。
「無限大または NaN を出力するために使うスタイルと、そのスタイルを NaN のために出力する場合の n 文字シーケンスの意味(7.19.6.1、7.24.2.1)」	
	NaN は nan として出力され、文字シーケンスは出力されません。 無限大は[-/+]inf として出力されます。
「fprintf または fwprintf 関数での%p 変換のための出力(7.19.6.1, 7.24.2.1)」	
	機能的に%1x と等価です。
「fscanf または fwscanf 関数内の%[変換向けの scanlist 内で、-文字が先頭文字でも末尾文字でもなく先頭の^文字に続く第 2 文字でもない場合の-文字の解釈(7.19.6.2, 7.24.2.2)」	
	ストリームは未実装です。
「fscanf または fwscanf 関数内の%p 変換によって照合する文字の並びのセット(7.19.6.2、7.24.2.2)」	
	ストリームは未実装です。
「障害時に fgetpos、fsetpos、ftell 関数が errno マクロに設定する値(7.19.9.1、7.19.9.3、7.19.9.4)」	
	ストリームは未実装です。
「strtod、strtof、strtold、wcstod、wcstof、wcstold 関数によって変換された文字列内の n 文字シーケンスの意味(7.20.1.3, 7.24.4.1.1)」	
	n 文字シーケンスに意味は与えません。
「アンダーフローが発生した時に strtod、strtof、strtold、wcstod、wcstof、wcstold 関数が errno を ERANGE に設定するかどうか(7.20.1.3、7.24.4.1.1)」	
	設定しません。
「要求されたサイズが 0 である場合に calloc、malloc、realloc 関数は NULL ポインタを返すか、それとも割り当てられたオブジェクトへのポインタを返すか(7.20.3)」	

.....続き

ISO 規格	処理系
	この割り当てに成功した場合(メモリに空きがある場合) は要求したサイズは 1 バイトに増加されてポインタを返し、成功しなかった場合は NULL を返します。
「abort 関数が呼び出された時に、開いていた出力ストリームをフラッシュするのか閉じるのか、一時ファイルを削除するかどうか(7.20.4.1)」	
	ストリームは未実装です。
「abort 関数がホスト環境に返す終了ステータス(7.20.4.1)」	
	ホスト環境はアプリケーション定義です。
「引数が NULL ポインタではない場合に system 関数が返す値(7.20.4.5)」	
	ホスト環境はアプリケーション定義です。
「ローカル タイムゾーンと夏時間(7.23.1)」	
	アプリケーション定義です。
「clock_t および time_t で表現可能な時間のレンジと精度(7.23)」	
	time_t 型は秒数を保持するために使われ、long 型として定義されます。 clock_t は unsigned long 型として定義されます。
「clock 関数の計時期間(7.23.2.1)」	
	アプリケーション定義です。
「"C"ロケールでの strftime、strfxtime、wcsftime、wcsfxtime 関数に対する%Z 指定子の置換文字列(7.23.3.5、7.23.3.6、7.24.5.1、7.24.5.2)」	
	これらの関数は未実装です。
「IEC 60559 準拠の処理系において三角関数、双曲線関数、e を底とする指数関数、e を底とする対数関数、エラー関数、対数ガンマ関数によって不正確結果例外が発生するかどうか。発生するならば、いつ発生するか(F.9)」	
	発生しません。
「<math.h>内の関数が丸め処理方向モードを尊重するかどうか(F.9)」	
	丸め処理モードは強制されません。

11.14 アーキテクチャ

ISO 規格	処理系
「ヘッダ<float.h>、<limits.h>、<stdint.h>内で指定されたマクロに割り当てられる値または式(C90 および C99 5.2.4.2、C99 7.18.2、7.18.3)」	
	『Microchip Unified Standard Library Reference Guide』内の<float.h>、<limits.h>、<stdint.h>の説明を参照してください。
「規格で明示的に指定されていない場合のオブジェクト内のバイトの数、順序、エンコーディング(C99 6.2.6.1)」	
	リトルエンディアン(最下位バイトから先に格納)です。
「sizeof 演算子の結果の値(C90 6.3.3.4、C99 6.5.3.4)」	
	結果の型は unsigned int と等価です。

12. 改訂履歴

リビジョン A (2018 年 3 月)

- 本書の初版は『MPLAB® XC8 C Compiler User's Guide』(DS 50002053)を基に編集されました。

リビジョン B (2019 年 3 月)

- ベクタ割り込みを備えたデバイスを使う場合の-mcodeoffset オプションの動作を明確化しました。
- 新しいコードカバレッジ機能に関する情報を追加しました。
- コンパイラ動作モードと最適化レベル制御オプション-O の説明を更新しました。
- コンパイラ コマンドライン オプションに関連する MPLAB X IDE 「Project Property」ダイアログの説明とスクリーンショットを追加しました。
- long long 型のサポートに関する情報を更新しました。
- サンプルコード内の誤りを訂正しました。
- __at()コンストラクタの使用法を明確化しました。
- その他の修正と改善を図りました。

リビジョン C (2020 年 2 月)

- 文書全体を新しい書式に変更しました。
- 標準ライブラリの説明を更新しました。
- 新しい movff 関連エラッタの情報を追加しました。
- アセンブラ分岐命令変換の新しい振る舞いを反映した情報を更新しました。
- Hexmate の検索/置換コマンドの構文に関連する説明を変更しました。
- オブジェクトと p-code モジュールのライブラリ アーカイブ内での共存を可能とする変更の詳細を記載しました。
- SHA256 サブオプションをメモリ使用量サマリ オプションに追加しました。
- printf 形式の関数をユーザ定義関数によって置き換える事が可能となりました。
- 新しい CONFIG ディレクティブを追加しました。
- ディレクティブと同様に扱われるようになったアセンブラ制御に関連する変更の詳細を記載しました。
- デバイスファミリ パック(DFP)オプションの使用法に関する情報を増補しました。
- その他の修正と改善を図りました。

リビジョン D (2021 年 2 月)

- 新しいスタック ガイダンス機能に関する情報を追加しました。
- const 修飾オブジェクトの配置について明確化しました。
- 新しい Hexmate ハッシュ値オプションに関する情報を追加しました。
- Hexmate の一部の既存機能に関する説明を増補しました。
- ライブラリ サンプルに printf()の使用に関する情報を追加しました。
- 更新された-x ドライバ オプションの説明を記載しました。
- -mcmacros および-mclink ドライバ オプションを追加しました。
- MPLAB X IDE 「Porject Porperties」ダイアログのスクリーンショットを更新しました。
- スイッチ文内の case レンジの使用法を明確化しました。
- PIC18 IVT デバイスがレガシーモードで動作する場合の割り込みに関する情報を追加しました。
- ステータス レジスタの保存機能に関する説明を増補しました。
- DABS および DLABS ディレクティブの使用法を明確化しました。
- 新しいアセンブラ ディレクティブ WARN および DEBUG_SOURCE を追加しました。

リビジョン E (2021 年 8 月)

- 標準 C ライブラリ関数に関する説明を本書から削除しました。これらの説明は別の文書『Microchip Unified Standard Library Reference Guide』に記載しています。
- `-fcacheconst` オプションと新しい定数キャッシュ機能の説明を追加しました。
- `-mheap` オプションとヒープおよび動的メモリ割り当て機能の説明を追加しました。
- `-mstack` オプションの動作に関する情報を更新しました。
- 第 4 世代デバッガツールのサポートを反映するため、`-mdebugger` デバッガ オプションを更新しました。
- MPLAB X IDE 「Porject Porperties」ダイアログの「Analysis」オプションに関連するスクリーン ショットを更新しました。
- 新しい `BANKISEL` ディレクティブに関する情報を追加しました。
- 以前は記載していなかった `FILE` および `LINE` アセンブラ ディレクティブに関する情報を追加しました。
- `DB`、`DW`、`DDW` アセンブラ ディレクティブ向けの文字列サポートの説明を追加しました。
- Hexmate の INHX16 フォーマット HEX ファイルの扱いに関する更新情報を追加しました。
- Hexmate の `-CK` オプションに、ハッシュの最終結果と指定された値の間で XOR を計算する新しい 0 サブオプションを追加しました。

Microchip 社ウェブサイト

Microchip 社はウェブサイト(www.microchip.com)を通してオンライン サポートを提供しています。当ウェブサイトでは、お客様に役立つ情報やファイルを簡単に見つけ出せます。以下を含む各種の情報をご覧になれます。

- **製品サポート** - データシートとエラッタ、アプリケーション ノートとサンプル プログラム、設計リソース、ユーザガイドとハードウェア サポート文書、最新のソフトウェアと過去のソフトウェア
- **技術サポート** – よく寄せられる質問(FAQ)、技術サポートのご依頼、オンライン ディスカッション グループ、Microchip 社のコンサルタント プログラムおよびメンバーリスト
- **ご注文とお問い合わせ** - 製品セレクタと注文ガイド、最新プレスリリース、セミナー/イベントの一覧、お問い合わせ先(営業所/販売代理店)の一覧

顧客変更通知サービス

Microchip 社のお客様向け変更通知サービスは、お客様に Microchip 社製品の最新情報をお届けする配信サービスです。ご興味のある製品ファミリまたは開発ツールに関する変更、更新、リビジョン、エラッタ情報をいち早くメールにてお知らせします。

http://www.microchip.com/pcn にアクセスし、登録手続きをしてください。

カスタマサポート

Microchip 社製品をお使いのお客様は、以下のチャンネルからサポートをご利用になれます。

- 正規代理店
- 技術サポート

サポートは正規代理店にお問い合わせください。各地の営業所もご利用になれます。本書の最後のページに各国の営業所の一覧を記載しています。

技術サポートは以下のウェブページからもご利用になれます。www.microchip.com/support

Microchip 社のデバイスコード保護機能

Microchip 社製デバイスのコード保護機能について以下の点にご注意ください。

- Microchip 社製品は、該当する Microchip 社データシートに記載の仕様を満たしています。
- Microchip 社では、通常の条件ならびに仕様に従って使った場合、Microchip 社製品のセキュリティ レベルは、現在市場に流通している同種製品の中でも最も高度であると考えています。
- しかし、コード保護機能を解除するための不正かつ違法な方法が存在する事もまた事実です。
  弊社の理解では、こうした手法は全て Microchip 社データシートにある動作仕様書以外の方法で Microchip 社製品を使用する事です。このような行為は知的所有権の侵害に該当する可能性が非常に高いと言えます。
- Microchip 社はコードの保全性に懸念を抱いているお客様と連携して対応策に取り組んでまいります。
- Microchip 社を含む全ての半導体メーカーで、自社のコードのセキュリティを完全に保証できる企業はありません。コード保護機能とは、Microchip 社が製品を「解読不能」として保証するものではありません。

コード保護機能は常に進歩しています。Microchip 社では、常に製品のコード保護機能の改善に取り組んでいます。Microchip 社のコード保護機能の侵害は、デジタル ミレニアム著作権法に違反します。そのような行為によってソフトウェアまたはその他の著作物に不正なアクセスを受けた場合は、デジタル ミレニアム著作権法の定めるところにより損害賠償訴訟を起こす権利があります。

法律上の注意点

本書に記載されているデバイス アプリケーション等の情報は、ユーザの便宜のためにのみ提供されるものであり、更新によって無効となる事があります。お客様のアプリケーションが仕様を満たす事を保証する責任は、お客様にあります。

Microchip 社は、明示的、暗黙的、書面、口頭、法定のいずれであるかを問わず、本書に記載されている情報に関して、状態、品質、性能、商品性、特定目的への適合性をはじめとする、いかなる類の表明も保証も行いません。Microchip 社は、本書の情報およびその使用に起因する一切の責任を否認します。Microchip 社の明示的な書面による承認なしに、生命維持装置あるいは生命安全用途に Microchip 社の製品を使用する事は全て購入者のリスクとし、また購入者はこれによって発生したあらゆる損害、クレーム、訴訟、費用に関して、Microchip 社は擁護され、免責され、損害をうけない事に同意するものとします。特に明記しない場合、暗黙的あるいは明示的を問わず、Microchip 社が知的財産権を保有しているライセンスは一切譲渡されません。

商標

Microchip 社の名称とロゴ、Microchip ロゴ、Adaptec、AnyRate、AVR、AVR ロゴ、AVR Freaks、BesTime、BitCloud、chipKIT、chipKIT ロゴ、CryptoMemory、CryptoRF、dsPIC、FlashFlex、flexPWR、HELDO、IGLOO、JukeBlox、KeeLoq、Kleer、LANCheck、LinkMD、maXStylus、maXTouch、MediaLB、megaAVR、Microsemi、Microsemi ロゴ、MOST、MOST ロゴ、MPLAB、OptoLyzer、PackeTime、PIC、picoPower、PICSTART、PIC32 ロゴ、PolarFire、Prochip Designer、QTouch、SAM-BA、SenGenuity、SpyNIC、SST、SST ロゴ、SuperFlash、Symmetricom、SyncServer、Tachyon、TimeSource、tinyAVR、UNI/O、Vectron、XMEGA は米国およびその他の国における Microchip Technology Incorporated の登録商標です。

AgileSwitch、APT、ClockWorks、The Embedded Control Solutions Company、EtherSynch、FlashTec、Hyper Speed Control、HyperLight Load、IntelliMOS、Libero、motorBench、mTouch、Powermite 3、Precision Edge、ProASIC、
ProASIC Plus、ProASIC Plus ロゴ、Quiet-Wire、SmartFusion、SyncWorld、Temux、TimeCesium、TimeHub、TimePictra、TimeProvider、WinPath、ZL は米国における Microchip Technology Incorporated の登録商標です。

Adjacent Key Suppression、AKS、Analog-for-the-Digital Age、Any Capacitor、AnyIn、AnyOut、Augmented Switching、BlueSky、BodyCom、CodeGuard、CryptoAuthentication、CryptoAutomotive、CryptoCompanion、CryptoController、dsPICDEM、dsPICDEM.net、Dynamic Average Matching、DAM、ECAN、Espresso T1S、EtherGREEN、IdealBridge、
In-Circuit Serial Programming、ICSP、INICnet、Intelligent Paralleling、Inter-Chip Connectivity、JitterBlocker、maxCrypto、maxView、memBrain、Mindi、MiWi、MPASM、MPF、MPLAB Certified ロゴ、MPLIB、MPLINK、MultiTRAK、NetDetach、Omniscient Code Generation、PICDEM、PICDEM.net、PICkit、PICtail、PowerSmart、PureSilicon、QMatrix、REAL ICE、Ripple Blocker、RTAX、RTG4、SAM-ICE、Serial Quad I/O、simpleMAP、SimpliPHY、SmartBuffer、SMART-I.S.、storClad、SQI、SuperSwitcher、SuperSwitcher II、Switchtec、SynchroPHY、Total Endurance、TSHARC、USBCheck、VariSense、VectorBlox、VeriPHY、ViewSpan、WiperLock、XpressConnect、ZENA は米国およびその他の国における Microchip Technology Incorporated の商標です。

SQTP は米国における Microchip Technology Incorporated のサービス マークです。

Adaptec ロゴ、Frequency on Demand、Silicon Storage Technology、Symmcom はその他の国における Microchip Technology Incorporated の登録商標です。

GestIC は、その他の国における Microchip Technology Germany II GmbH & Co. KG (Microchip Technology Inc.の子会社)の登録商標です。

その他の商標は各社に帰属します。

© 2021, Microchip Technology Incorporated, Printed in the U.S.A.,

All Rights Reserved.ISBN: 978-1-5224-8896-5

品質管理システム

Microchip 社の品質管理システムについては www.microchip.com/quality をご覧ください。

各国の営業所とサービス

北米

本社
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel:480-792-7200
Fax:480-792-7277
技術サポート：
http://www.microchip.com/support
URL:
www.microchip.com

アトランタ
Duluth, GA
Tel:678-957-9614
Fax:678-957-1455

オースティン、TX
Tel:512-257-3370

ボストン
Westborough, MA
Tel:774-760-0087
Fax:774-760-0088

シカゴ
Itasca, IL
Tel:630-285-0071
Fax:630-285-0075

ダラス
Addison, TX
Tel:972-818-7423
Fax:972-818-2924

デトロイト
Novi, MI
Tel:248-848-4000

ヒューストン、TX
Tel:281-894-5983

インディアナポリス
Noblesville, IN
Tel:317-773-8323
Fax:317-773-5453
Tel:317-536-2380

ロサンゼルス
Mission Viejo, CA
Tel:949-462-9523
Fax:949-462-9608
Tel:951-273-7800

ローリー、NC
Tel:919-844-7510

ニューヨーク、NY
Tel:631-435-6000

サンノゼ、CA
Tel:408-735-9110
Tel:408-436-4270

カナダ - トロント
Tel:905-695-1980
Fax:905-695-2078

アジア /太 平洋

オーストラリア - シドニー
Tel:61-2-9868-6733

中国 - 北京
Tel:86-10-8569-7000

中国 - 成都
Tel:86-28-8665-5511

中国 - 重慶
Tel:86-23-8980-9588

中国 - 東莞
Tel:86-769-8702-9880

中国 - 広州
Tel:86-20-8755-8029

中国 - 杭州
Tel:86-571-8792-8115

中国 - 香港 SAR
Tel:852-2943-5100

中国 - 南京
Tel:86-25-8473-2460

中国 - 青島
Tel:86-532-8502-7355

中国 - 上海
Tel:86-21-3326-8000

中国 - 瀋陽
Tel:86-24-2334-2829

中国 - 深圳
Tel:86-755-8864-2200

中国 - 蘇州
Tel:86-186-6233-1526

中国 - 武漢
Tel:86-27-5980-5300

中国 - 西安
Tel:86-29-8833-7252

中国 - 厦門
Tel:86-592-2388138

中国 - 珠海
Tel:86-756-3210040

アジア /太 平洋

インド - バンガロール
Tel:91-80-3090-4444

インド - ニューデリー
Tel:91-11-4160-8631

インド - プネ
Tel:91-20-4121-0141

日本 - 大阪
Tel:81-6-6152-7160

日本 - 東京
Tel:81-3-6880- 3770

韓国 - 大邱
Tel:82-53-744-4301

韓国 - ソウル
Tel:82-2-554-7200

マレーシア - クアラルンプール
Tel:60-3-7651-7906

マレーシア - ペナン
Tel:60-4-227-8870

フィリピン - マニラ
Tel:63-2-634-9065

シンガポール
Tel:65-6334-8870

台湾 - 新竹
Tel:886-3-577-8366

台湾 - 高雄
Tel:886-7-213-7830

台湾 - 台北
Tel:886-2-2508-8600

タイ - バンコク
Tel:66-2-694-1351

ベトナム - ホーチミン
Tel:84-28-5448-2100

ヨーロッパ

オーストリア - ヴェルス
Tel:43-7242-2244-39
Fax:43-7242-2244-393

デンマーク - コペンハーゲン
Tel:45-4485-5910
Fax:45-4485-2829

フィンランド - エスポー
Tel:358-9-4520-820

フランス - パリ
Tel:33-1-69-53-63-20
Fax:33-1-69-30-90-79

ドイツ - ガーヒング
Tel:49-8931-9700

ドイツ - ハーン
Tel:49-2129-3766400

ドイツ - ハイルブロン
Tel:49-7131-72400

ドイツ - カールスルーエ
Tel:49-721-625370

ドイツ - ミュンヘン
Tel:49-89-627-144-0
Fax:49(-89/-627)-144/-44

ドイツ - ローゼンハイム
Tel:49-8031-354-560

イスラエル - ラーナナ
Tel:972-9-744-7705

イタリア - ミラノ
Tel:39-0331-742611
Fax:39-0331-466781

イタリア - パドヴァ
Tel:39-049-7625286

オランダ - ドリューネン
Tel:31-416-690399
Fax:31-416-690340

ノルウェー - トロンハイム
Tel:47-7288-4388

ポーランド - ワルシャワ
Tel:48-22-3325737

ルーマニア - ブカレスト
Tel:40-21-407-87-50

スペイン - マドリッド
Tel:34-91-708-08-90
Fax:34-91-708-08-91

スウェーデン - ヨーテボリ
Tel:46-31-704-60-40

スウェーデン - ストックホルム
Tel:46-8-5090-4654

イギリス - ウォーキンガム
Tel:44-118-921-5800
Fax:44-118-921-5820

© 2022 Microchip Technology Inc. and its subsidiaries