# MPLAB® C18
# C コンパイラ
# ユーザーズ ガイド

ご注意：この日本語版ドキュメントは、参考資料としてご使用の上、最新情報につきましては、必ず英語版オリジナルをご参照いただきますようお願いします。

**マイクロチップ テクノロジー社 ( 以下、マイクロチップ社 ) デバイスのコード保護機能に関する以下の点にご留意ください。**

- マイクロチップ社製品は、その該当するマイクロチップ社データシートに記載の仕様を満たしています。
- マイクロチップ社では、通常の条件ならびに仕様どおりの方法で使用した場合、マイクロチップ社製品は現在市場に流通している同種製品としては最もセキュリティの高い部類に入る製品であると考えております。
- 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します。マイクロチップ社の確認している範囲では、このような方法のいずれにおいても、マイクロチップ社製品をマイクロチップ社データシートの動作仕様外の方法で使用する必要があります。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。
- マイクロチップ社は、コードの保全について懸念を抱いているお客様と連携し、対応策に取り組んでいきます。
- マイクロチップ社を含むすべての半導体メーカーの中で、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、マイクロチップ社が製品を「解読不能」として保証してい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。マイクロチップ社では、製品のコード保護機能の改善に継続的に取り組んでいます。マイクロチップ社のコード保護機能を解除しようとする行為は、デジタルミレニアム著作権法に抵触する可能性があります。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。



**商標**



再生紙を使用しています。

**QUALITY MANAGEMENT SYSTEM**
**CERTIFIED BY DNV**
**ISO/TS 16949:2002**

*マイクロチップ社では、Chandler および Tempe ( アリゾナ州 )、Gresham ( オレゴン州 ) の本部、設計部およびウエハ製造工場そしてカリフォルニア州とインドのデザインセンターが ISO/TS-16949:2002 認証を取得しています。マイクロチップ社の品質システムプロセスおよび手順は、PIC® MCU および dsPIC® DSC、KEELOQ® コードホッピングデバイス、シリアル EEPROM、マイクロペリフェラル、不揮発性メモリ、アナログ製品に採用されています。また、マイクロチップ社の開発システムの設計および製造に関する品質システムは、ISO 9001:2000 の認証を受けています。*

# MPLAB® C18 C コンパイラ ユーザーズ ガイド

## 目次

# MPLAB® C18 C コンパイラ ユーザーズ ガイド

## 序章

### 顧客の皆様への注意

**ドキュメントはすべて古くなります。本書も例外ではありません。マイクロチップ社のツールおよびマニュアルはユーザーのニーズを満たすため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に記載されているものと異なる場合があります。最新のドキュメントを入手するには、弊社のウェブ サイト (www.microchip.com) をご覧ください。**

**ドキュメントは「DS」番号で識別されています。この識別番号は、各ページのフッタ部分、ページ番号の前に記載されています。DS 番号の表記規則は「DSXXXXXA」で、「XXXXX」が文書番号、「A」が文書のリビジョン レベルを表しています。**

**開発ツールについての最新情報は、MPLAB® IDE のオンライン ヘルプをご覧ください。[ ヘルプ ] メニューを選択して、次に [ トピック ] を選択すると、利用できるオンライン ヘルプ ファイルのリストが表示されます。**

### はじめに

本書では、MPLAB® C18 コンパイラの技術内容について詳しく説明します。本書では MPLAB C18 コンパイラのすべての機能を解説します。次の条件を満たしたプログラマの方を読者に想定しています。

- C プログラムを作成できる
- MPLAB 統合開発環境 (IDE) を使用してプロジェクトを作成、デバッグできる
- コードを作成するターゲット プロセッサのデータシートを読んで理解している

### 本書の構成

本書の構成は次のとおりです。

- **第 1 章「はじめに」** – MPLAB C18 コンパイラの概要と起動方法について説明します。
- **第 2 章「言語の仕様」** – MPLAB C18 コンパイラと ANSI 規格の違いについて説明します。
- **第 3 章「ランタイム モデル」** – MPLAB C18 コンパイラが利用する PIC18 PICmicro® マイクロコントローラのリソースについて説明します。
- **第 4 章「最適化」** – MPLAB C18 コンパイラの最適化機能について説明します。
- **第 5 章「サンプル コード」** – いくつかのサンプル アプリケーションを例にとり、本書で説明した内容に言及しながら ソース コードを説明します。
- **付録 A「COFF ファイル フォーマット」** – マイクロチップ社の COFF フォーマットについて詳しく説明します。
- **付録 B「ANSI 処理系で定義されている動作」** – MPLAB C18 の処理系で定義されている動作を、ANSI 規格の規定に従って説明します。
- **付録 C「コマンドライン オプション一覧」** – すべてのコマンドライン オプションと、それぞれを本書で説明している項番号を一覧で示します。
- **付録 D「MPLAB C18 メッセージ一覧」** – エラー、警告、メッセージを一覧で示します。

- **付録 E「拡張モード」** – 非拡張モードと拡張モードの違いについて説明します。

## 本書で使用される表記

本書では以下の表記上の規則を使用しています。

**表記上の規則**

| 説明 | 意味 | 例 |
|---|---|---|
| **明朝フォント :** | | |
| 斜体文字 | 参考資料 | *MPLAB® IDE User's Guide* |
| **クーリエ フォント :** | | |
| 通常のクーリエ | 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` |
| | コマンド行オプション | `-Opa+, -Opa-` |
| イタリック クーリエ | 変数の引数 | *`file`*`.o`: *`file`* は任意の有効なファイル名 |
| `0b`*`nnnn`* | 2 進数、*n* は 2 進数桁 | `0b00100, 0b10` |
| `0x`*`nnnn`* | 16 進数、*n* は 16 進数桁 | `0xFFFF, 0x007A` |
| 角括弧 : [ ] | 任意の引数 | `mcc18 [ オプション] `*`file`*` [ オプション]` |
| 省略記号 : ... | 繰り返されるテキスト | `var_name [, var_name...]` |
| | ユーザーが定義するコード | `void main (void) { ... }` |
| **アイコン :** | | |
| Full Version Only | この機能は、完全版のソフトウェアでのみサポートされます。 | Full Version Only **1.2.5 モードの選択** |

## PIC18 の開発参考資料

**readme.c18**

このファイルは本ソフトウェアに添付されており、本書に収録されていない最新情報が記載されています。

**PIC18 Configuration Settings Addendum (DS51537)**

MPLAB C18 C コンパイラの `#pragma config` ディレクティブおよび MPASM™ アセンブラの `CONFIG` ディレクティブでサポートされている、マイクロチップ社の PIC18 デバイスの構成ビット設定を一覧にしたものです。

**MPLAB® C18 C Compiler Getting Started (DS51295)**

MPLAB C18 コンパイラのインストール、シンプルなプログラムの作成、MPLAB IDE でのコンパイラの使用について説明しています。

**MPLAB® C18 C Compiler Libraries (DS51297)**

MPLAB C18 のライブラリおよびコンパイル済みオブジェクト ファイルのリファレンス ガイドです。MPLAB C18 C コンパイラで提供されるすべてのライブラリ関数を一覧にまとめ、それぞれの使用方法を詳しく説明しています。

**MPLAB® IDE Quick Start Guide (DS51281)**

MPLAB IDE ソフトウェアのセットアップからプロジェクトの作成、デバイスへの書き込みまでを説明しています。

**MPASM™ Assembler, MPLINK™ Object Linker, MPLIB™ Object Librarian User's Guide (DS33014)**

マイクロチップ社の PICmicro MCU アセンブラ (MPASM)、リンカ (MPLINK)、ライブラリアン (MPLIB) の使用方法を説明しています。

**PICmicro® 18C MCU Family Reference Manual (DS39500)**

MCU の拡張デバイス ファミリを中心に解説しています。拡張 MCU ファミリのアーキテクチャおよび周辺モジュールの動作について説明していますが、個々のデバイスに固有の内容は取り上げていません。

**PIC18 デバイスのデータ シートとアプリケーション ノート**

データ シートには、PIC18 デバイスの動作と電気的仕様が記載されています。アプリケーション ノートには、PIC18 デバイスの具体的な使用法が記載されています。

上記のドキュメントは、マイクロチップ社のウェブ サイト (www.microchip.com) で Adobe Acrobat (.pdf) 形式で入手できます。

## C に関する参考資料

情報システムにおける米国標準 – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

この標準は、形式および C プログラム言語で表現されたプログラムの解釈について規定しています。種々のコンピューティングシステムにおける C 言語プログラムに関して、移植性、信頼性、保守性および効率的な実行を促進することを目的としています。

Harbison, Samuel P. and Steele, Guy L., *C: A Reference Manual,* Fourth Edition. Prentice-Hall, Englewood Cliffs, New Jersey 07632.

C プログラミング言語を非常に詳しく解説しています。この著書は、C 言語、ランタイム ライブラリ、そして正確性、移植性、保守性に重点を置いた C プログラミング スタイルまでを広く解説した、リファレンス マニュアルの決定版です。

Huang, Han-Way. *PIC® Microcontroller: An Introduction to Software & Hardware Interfacing*. Thomson Delmar Learning, Clifton Park, New York 12065.

PIC マイクロコントローラ (MCU) のプログラミングおよび周辺機能とのインターフェースを含め、マイクロチップ社の PIC18 マイクロコントローラ ファミリの内容を余すところなく紹介した入門書です。カレッジ レベルの本書では、PIC MCU アセンブリ言語と MPLAB C18 C コンパイラの両方が使用されています。

Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, New Jersey 07632.

ANSI 規格で定義された C を簡潔に説明しています。C プログラマ向けリファレンスに最適です。

Kochan, Steven G. *Programming In ANSI C*, Revised Edition. Hayden Books, Indianapolis, Indiana 46268.

この著書も、大学で ANSI C を学ぶ学生向けのリファレンスとして定評があります。

Peatman, John B. *Embedded Design with the PIC18F452 Microcontroller*, First Edition. Pearson Education, Inc., Upper Saddle River, New Jersey 07458.

マイクロチップ社の PIC18FXXX ファミリおよび拡張アプリケーション コードの作成を中心に解説しています。

Van Sickle, Ted. *Programming Microcontrollers in C*, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

本書は主に Motorola のマイクロコントローラを中心に解説していますが、マイクロコントローラ向け C プログラミングの基本的な原則に参照できます。

## その他の参考資料

Standards Committee of the IEEE Computer Society – *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, New York 10017.

この規格には、MPLAB C18 で使用している浮動小数点フォーマットについて説明があります。

## マイクロチップ社ウェブ サイト

マイクロチップ社は同社のウェブ サイト (www.microchip.com) でオンライン サポートを行っています。本ウェブ サイトはファイルや情報をいち早く顧客に提供する手段として用いられています。ウェブ サイトにはご使用のブラウザでアクセスでき、下記の情報が含まれます。

- **製品サポート** – データシートと正誤表、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、ユーザーガイド、ハードウェア サポート文書、最新リリース ソフトウェア、ソフトウェア アーカイブ
- **一般的なテクニカル サポート** – よく寄せられる質問 (FAQ)、テクニカル サポートの依頼、オンライン ディスカッション グループ、マイクロチップ社コンサルタント プログラム メンバーのリスト
- **マイクロチップのビジネス** – 製品選択と注文ガイド、マイクロチップ社の最新プレスリリース、セミナとイベントのリスト、マイクロチップの営業オフィス、代理店、工場代理人のリスト

## 開発システムのお客様変更通知サービス

マイクロチップ社のお客様変更通知サービスは、お客様がマイクロチップ社製品の最新情報を入手できるようにします。加入者は、指定した製品ファミリや興味のある開発ツールに関する変更、更新、リビジョン、正誤表があるときは常に E メール通知を受け取ることができます。

登録には、マイクロチップのウェブ サイトにアクセスして、[ お客様変更通知 ] をクリックし、登録指示に従うだけです。

開発システム製品のグループ カテゴリは、以下のとおりです。

- **コンパイラ** – マイクロチップ社製 C コンパイラおよび他の言語ツールについての最新情報です。これには、MPLAB C18 および MPLAB C30 C コンパイラ、MPASM™ および MPLAB ASM30 アセンブラ、MPLINK™ および MPLAB LINK30 オブジェクト リンカ、MPLIB™ および MPLAB LIB30 オブジェクト ライブラリアンが含まれます。
- **エミュレータ** – マイクロチップ社製インサーキットエミュレータの最新情報です。これには、MPLAB ICE 2000 および MPLAB ICE 4000 が含まれます。
- **インサーキット デバッガ** – マイクロチップ社製インサーキット デバッガ MPLAB ICD 2 の最新情報です。
- **MPLAB® IDE** – 開発システムツール向け Windows® 統合開発環境であるマイクロチップ社製 MPLAB IDE の最新情報です。このリストでは、MPLAB IDE、MPLAB SIM シミュレータ、MPLAB IDE Project Manager および一般的な編集およびデバッグ機能が取り上げられています。
- **プログラマ** – マイクロチップ社製プログラマの最新情報です。これには、MPLAB PM3 および PRO MATE® II デバイスプログラマおよび PICSTART® Plus および PICkit™ 開発プログラマが含まれます。

## 顧客サービス

マイクロチップ製品のユーザーは、いくつかのチャネルを介してサポートを受けられます。

- 販売代理店または販売担当者
- 地域の営業所
- フィールド アプリケーション エンジニア (FAE)
- 技術サポート

技術サポートを得るには、販売代理店か販売担当者、フィールド アプリケーション エンジニア (FAE) に連絡してください。地域の営業所でもお客様の手助けをします。営業所と所在地のリストが本文書の最後に記載されています。

技術サポートは弊社ウェブ サイト (http://support.microchip.com) を通して受けることができます。

この他、開発システム情報ラインでは、マイクロチップ社の最新の開発システム ソフトウェア製品をご案内しています。また、このサービスではお客様に現在ご利用可能なアップグレード キットの情報もお知らせしています。

開発システム情報ラインの電話番号 :

1-800-755-2345 – 米国およびカナダ

1-480-792-7302 – その他の国

# 第 1 章 はじめに

## 1.1 概要

MPLAB C18 コンパイラは、PIC18 PICmicro マイクロコントローラ (MCU) 向けのフリースタンディング最適化 ANSI C コンパイラです。このコンパイラは ANSI 標準 X3.159-1989 に準拠していますが、PICmicro MCU の効果的なサポートを優先しているため、この規格とは一部異なる部分があります。このコンパイラは、32 ビット Windows® コンソール アプリケーションであり、マイクロチップ社の MPLAB IDE とも完全な互換性を備えているため、MPLAB ICE インサーキット エミュレータ、MPLAB ICD 2 インサーキット デバッガ、MPLAB SIM シミュレータを利用して、ソースレベルのデバッグが実行できます。

MPLAB C18 コンパイラには次の特長があります。

- ANSI '89 との互換性
- MPLAB IDE との統合により、プロジェクト 管理とソースレベルのデバッグが容易
- 再配置可能なオブジェクト モジュールを生成することで、コード再利用度が向上
- MPASM アセンブラで生成したオブジェクト モジュールと互換性があり、1 つのプロジェクトでアセンブリと C プログラミングを混在できる優れた柔軟性
- 外部メモリへの透過的なリード / ライト アクセス
- 完全な制御が要求される場合に使用されるインライン アセンブリを強力にサポート
- 多重の最適化機能を備えた効率的なコード ジェネレータ エンジン
- PWM、SPI™、I²C™、UART、USART、文字列操作、算術演算ライブラリなど、広範なライブラリをサポート
- データとコードのメモリ配置をユーザー レベルで完全に制御可能

## 1.2 コンパイラの起動

MPLAB IDE で MPLAB C18 コンパイラを使用する方法については、『*MPLAB® C18 C Compiler Getting Started*』(DS51295) を参照してください。MPLAB C18 コンパイラは、コマンドラインからも起動できます。コマンドラインの使用法は次のとおりです。

```
mcc18 [options] file [options]
```

コマンドラインでは、1 つのソース ファイルと任意の数のオプションを指定できます。コマンドライン オプションに `--help` を指定すると、コンパイラで指定可能なすべてのコマンドライン オプションが一覧で表示されます。コマンドライン オプションに `-verbose` を指定すると、コンパイル完了時にバージョン番号およびエラー、警告、メッセージの総数がバナー形式で表示されます。

### 1.2.1 出力ファイルの生成

デフォルトでは、コンパイラによって生成される出力オブジェクト ファイルの名前は *file*.o (*file* はコマンドラインで指定したソース ファイルから拡張子を除いたもの ) となります。コマンドライン オプションに -fo を指定すると、出力オブジェクト ファイル名が変更できます。次に例を示します。

```
mcc18 -fo bar.o foo.c
```

ソース ファイルにエラーがある場合、*file*.err (*file* はコマンドラインで指定したソース ファイル名から拡張子を除いたもの ) という名前のエラー ファイルが生成されます。コマンドライン オプションに -fe を指定すると、エラー ファイルの名前を変更できます。次に例を示します。

```
mcc18 -fe bar.err foo.c
```

### 1.2.2 診断情報の表示

診断情報の表示は、コマンドライン オプションに -w または -nw を指定することで制御できます。コマンドライン オプションの -w では、表示する診断情報のレベル (1、2、3) を設定します。表 1-1 に、各レベルで表示される 診断情報の内容を示します。コマンドライン オプションの -nw を使用すると、特定のメッセージを非表示に設定できます (MPLAB C18 コンパイラで生成されるメッセージの一覧については、**付録 D「MPLAB C18 メッセージ一覧」**を参照するか、コマンドライン オプションで --help-message-list と指定して表示してください )。コマンドライン オプションで --help-message-all と指定すると、すべてのメッセージに関するヘルプが表示されます。特定の 診断情報に関するヘルプを表示するには、コマンドライン オプションの --help-message を使用します。次に例を示します。

```
mcc18 --help-message=2068
```

上記のコマンドを実行すると、次の結果が表示されます。

```
2068: obsolete use of implicit 'int' detected.

The ANSI standard allows a variable to be declared without a base type
being specified, e.g., "extern x;", in which case a base type of 'int'
is implied. This usage is deprecated by the standard as obsolete, and
therefore a diagnostic is issued to that effect.
```

**表 1-1: 診断情報のレベル**

| Warning Level | Diagnostics Shown |
|---|---|
| 1 | Errors (fatal and non-fatal) |
| 2 | Level 1 plus warnings |
| 3 | Level 2 plus messages |

### 1.2.3 マクロの定義

マクロを定義するには、コマンドライン オプション -D を使用します。コマンドライン オプションの -D は、-D*name* または -D*name*=*value* のいずれかの形式で指定できます。-D*name* と指定すると、マクロ *name* が 1 と定義されます。-D*name*=*value* と指定すると、マクロ *name* に *value* の値が定義されます。次に例を示します。

```
mcc18 -DMODE
```

この場合、マクロ MODE の値は 1 と定義されます。

```
mcc18 -DMODE=2
```

この場合、マクロ MODE の値は 2 と定義されます。

一般に、コマンドライン オプション -D は条件付きコンパイルで使用されます。次に例を示します。

```
#if MODE == 1
    x = 5;
#elif MODE == 2
    x = 6;
#else
    x = 7;
#endif
```

### 1.2.4 プロセッサの選択

デフォルトでは、MPLAB C18 コンパイラは汎用の PIC18 PICmicro マイクロコントローラ用にアプリケーションをコンパイルします。特定のプロセッサのみを対象にオブジェクト ファイルを生成する場合は、コマンドライン オプションに -p*processor* (*processor* は使用するプロセッサ名 ) を指定します。例えばオブジェクト ファイルを PIC18F452 でのみ使用する場合は、コマンドライン オプションで -p18f452 と指定します。コマンドライン オプションに -p18cxx を指定すると、明示的に汎用の PIC18 マイクロコントローラを対象にソースをコンパイルできます。


### 1.2.5 モードの選択

MPLAB PIC18 C コンパイラの動作モードには、拡張モード[1] と非拡張モードの 2 つがあります。コンパイラの動作モードを拡張モードとすると、拡張命令 (ADDFSR、ADDULNK、CALLW、MOVSF、MOVSS、PUSHL、SUBFSR、SUBULNK) およびリテラル オフセットによるインデックス アドレス指定が使用できるため、通常は少ない命令数でスタック ベースの変数にアクセスできます。このため、プログラム メモリ イメージが小さくなります。非拡張モードで動作時は、コンパイラは拡張命令もリテラル オフセットによるインデックス アドレス指定も利用できません。コンパイラの動作モードは、コマンドライン オプションの --extended または --no-extended で指定します。コマンドライン オプションの --extended は、拡張命令セットをサポートしたプロセッサを -p オプションで選択しているか、または汎用の PIC18 マイクロコントローラをコンパイル対象とする場合に指定できます (**1.2.4 項「プロセッサの選択」**参照 )。コマンドライン オプションの --no-extended は、汎用マイクロコントローラを含むすべての PIC18 マイクロコントローラで使用できます。コマンドライン オプションに --extended も --no-extended も指定しない場合は、選択したプロセッサの種類にかかわらず、コンパライは非拡張モードで動作します。表 1-2 に、各コマンドライン オプションを指定した場合のコンパイラの動作モードを示します。

1. デモ バージョンの試用期限が過ぎると、コンパイラの拡張モードは使用できません。

**表 1-2: モードの選択**

| | **-p *extended*** | **-p *no-extended*** | **-p18cxx** | **No Processor Specified** |
|---|---|---|---|---|
| --extended | Extended | Error | Extended | Extended |
| --no-extended | Non-Extended | Non-Extended | Non-Extended | Non-Extended |
| Not Specified | Non-Extended | Non-Extended | Non-Extended | Non-Extended |

mcc18 --help としてコンパイラを起動すると、非拡張モード時のコンパイラのヘルプが表示されます。このヘルプで表示されるコマンドライン オプションの中には、拡張モードでは使用できないものが含まれます。拡張モード時のコンパイラのヘルプを表示するには、コマンドラインで mcc18 --extended --help として実行します。

**注 :** その他のコマンドライン オプションについては、本ユーザー ガイドで随時解説します。すべてのコマンドライン オプションの一覧については、**付録 C「コマンドライン オプション一覧」**を参照してください。

# 第 2 章 言語の仕様

## 2.1 データ型と値の制限

### 2.1.1 整数型

MPLAB C18 コンパイラは、ANSI で定義された標準の整数型をサポートしています。標準整数型の値の範囲を表 2-1 に示します。また、MPLAB C18 は 24 ビット整数型の `short long int` ( または `long short int`) を、符号付きと符号なしのいずれもサポートしています。これら整数型の値の範囲も表 2-1 に示しています。

**表 2-1: 整数データ型のサイズと値の制限**

| Type | Size | Minimum | Maximum |
|---|---|---|---|
| `char`(1,2) | 8 bits | -128 | 127 |
| `signed char` | 8 bits | -128 | 127 |
| `unsigned char` | 8 bits | 0 | 255 |
| `int` | 16 bits | -32,768 | 32,767 |
| `unsigned int` | 16 bits | 0 | 65,535 |
| `short` | 16 bits | -32,768 | 32,767 |
| `unsigned short` | 16 bits | 0 | 65,535 |
| `short long` | 24 bits | -8,388,608 | 8,388,607 |
| `unsigned short long` | 24 bits | 0 | 16,777,215 |
| `long` | 32 bits | -2,147,483,648 | 2,147,483,647 |
| `unsigned long` | 32 bits | 0 | 4,294,967,295 |

**注 1:** 修飾子のない *`char`* は、デフォルトで符号付きです。

**2:** コマンドライン オプションに`-k` を指定すると、修飾子のない *`char`* がデフォルトで符号なしになります。

### 2.1.2 浮動小数点型

MPLAB C18 では、`double` または `float` データ型の 32 ビット浮動小数点型が固有でサポートされています。MPLAB C18 では、IEEE-754 浮動小数点標準に準拠して浮動小数点型を表現します。浮動小数点型の値の範囲を表 2-2 に示します。

**表 2-2: 浮動小数点データ型のサイズと値の制限**

| Type | Size | Minimum Exponent | Maximum Exponent | Minimum Normalized | Maximum Normalized |
|---|---|---|---|---|---|
| `float` | 32 bits | -126 | 128 | $2^{-126} \approx 1.17549435e-38$ | $2^{128} * (2-2^{-15}) \approx 6.80564693e+38$ |
| `double` | 32 bits | -126 | 128 | $2^{-126} \approx 1.17549435e-38$ | $2^{128} * (2-2^{-15}) \approx 6.80564693e+38$ |

## 2.2 データ型ストレージ ( エンディアン )

エンディアンとは、マルチバイトの値におけるバイトの並び方をいいます。MPLAB C18 では、データをリトル エンディアン形式で保存します。すなわち、下位バイトが下位アドレスに格納されます ( 下位バイトから順に格納する方式 )。次に例を示します。

```
#pragma idata test=0x0200
long l=0xAABBCCDD;
```

この場合、メモリ レイアウトは次のとおりです。

| Address | *0x0200* | *0x0201* | *0x0202* | *0x0203* |
|---|---|---|---|---|
| **Content** | 0xDD | 0xCC | 0xBB | 0xAA |

## 2.3 ストレージ クラス

MPLAB C18 は、ANSI 標準のストレージ クラス (auto、extern、register、static、typedef) をサポートしています。

### 2.3.1 overlay

MPLAB C18 コンパイラには、overlay というストレージ クラスも用意されています。overlay ストレージ クラスは、コンパイラが非拡張モードで動作する場合のみ適用されます (**1.2.5 項「モードの選択」**参照 )。overlay ストレージ クラスは、ローカル変数に使用できます ( 仮パラメータ、関数定義、グローバル変数には使用できません )。ストレージ クラスに overlay を指定したシンボルは、関数固有の静的なオーバーレイ セクションに配置されます。これらの変数は静的に割り当てられますが、関数に入るたびに初期化されます。次に例を示します。

```
void f (void)
{
  overlay int x = 5;
  x++;
}
```

この例では、x のストレージは静的に割り当てられますが、関数に入るたびに 5 に初期化されます。初期化子がない場合は、関数に入った時点での x の値は未定義となります。

複数の関数で overlay 属性を指定されたローカル ストレージが、同時にアクティブにならないことが確実な場合、MPLINK リンカはこれらをオーバーレイしようとします。次に例を示します。

```
int f (void)
{
  overlay int x = 1;
  return x;
}

int g (void)
{
  overlay int y = 2;
  return y;
}
```

f と g が同時にアクティブにならなければ、x と y で同じメモリ番地を共有できます。次に別の例を示します。

```
int f (void)
{
  overlay int x = 1;
  return x;
}

int g (void)
{
  overlay int y = 2;
  y = f ();
  return y;
}
```

この例では f と g が同時にアクティブになる可能性があるため、x と y はオーバーレイされません。overlay 属性を使用するとローカル変数を静的に割り当てることができるため、通常は少ない命令数で変数にアクセスできる、すなわちプログラム メモリのイメージを小さくできる、という利点があります。しかも一部の変数をオーバーレイできるため、static として宣言した場合よりも変数の割り当てに必要なデータ メモリのサイズが小さくなります。

再帰関数の中に overlay ストレージ クラスのローカル変数が検出されると、MPLINK リンカはエラーを出力して処理を中止します。いずれかのモジュールで関数ポインタを使用した呼び出しが実行されており、いずれかのモジュール ( 前述のモジュール以外を含む ) に overlay ストレージ クラスの変数が検出されると、MPLINK リンカはエラーを出力して処理を中止します。

ローカル変数のデフォルトのストレージ クラスは auto です。この設定は、明示的に static または overlay キーワードを使用するか、暗黙的にコマンドライン オプションに -scs (static ローカル変数 ) または -sco (overlay ローカル変数 ) を指定すると変更できます。これと合わせて、MPLAB C18 では -sca コマンドライン オプションも用意されています。このオプションは、ローカル変数のストレージ クラスを明示的に auto に指定するものです。

### 2.3.2 static 属性の関数の引き数

関数のパラメータには、auto または static のストレージ クラスを指定できます。auto 属性のパラメータはソフトウェア スタックに配置され、再入が可能になります。static 属性のパラメータはグローバルに割り当てられていて直接アクセスできるため、通常はコード サイズは小さくなります。パラメータに static 属性を指定できるのは、コンパイラが非拡張モードで動作する場合のみです (**1.2.5 項「モードの選択」**参照 )。

関数パラメータのデフォルトのストレージ クラスは auto です。この設定は、明示的に static キーワードを使用するか、暗黙的にコマンドライン オプションに -scs を指定すると変更できます。コマンドライン オプションに -sco を指定した場合でも、関数パラメータのストレージ クラスが暗黙的に static に変更されます。

## 2.4 ストレージ修飾子

ANSI 標準のストレージ修飾子 (const、volatile) 以外にも、MPLAB C18 コンパイラには far、near、rom 、ram というストレージ修飾子が用意されています。ANSI C の const および volatile 修飾子と同様に、これらの新しい修飾子も識別子に結合した構文で使用します。表 2-3 に、各ストレージ修飾子を付けて定義したオブジェクトが配置される場所を示します。明示的にストレージ修飾子を付けずにオブジェクトを定義すると、デフォルトのストレージ修飾子として far および ram が適用されます。

**表 2-3: 各ストレージ修飾子によるオブジェクトの配置**

| | rom | ram |
|---|---|---|
| far | Anywhere in program memory | Anywhere in data memory (default) |
| near | In program memory with address less than 64K | In access memory |

### 2.4.1 near/far 属性のデータ メモリ オブジェクト

far 修飾子は、データ メモリ内の変数がメモリ バンクに格納されており、この変数にアクセスする前にバンク スイッチング命令が必要であることを示すために使用します。near 修飾子は、データ メモリ内の変数がアクセス RAM への格納を示すために使用します。

### 2.4.2 near/far 属性のプログラム メモリ オブジェクト

far 修飾子は、プログラム メモリ内の変数がプログラム メモリの任意の番地に存在しうること、あるいはポインタの場合は 64K を超えるプログラム メモリ領域にアクセスできることを示すために使用します。near 修飾子は、プログラム メモリ内の変数が 64K 未満の番地に存在すること、あるいはポインタの場合は 64K までのプログラム メモリ領域にしかアクセスできないことを示すために使用します。

### 2.4.3 ram/rom 修飾子

PICmicro マイクロコントローラはプログラム メモリとデータ メモリで異なるアドレス バスを使用するように設計されているため、MPLAB C18 ではプログラム メモリに配置されたデータとデータ メモリに配置されたデータを区別するための拡張が必要となります。ANSI/ISO C 規格では、コードとデータを別々のアドレス領域に配置することが認められていますが、この方法でも コード領域に配置されたデータを特定するには不十分です。このため、MPLAB C18 には rom および ram 修飾子が用意されています。rom 修飾子は オブジェクトがプログラム メモリに配置されていることを示し、ram 修飾子はオブジェクトがデータ メモリに配置されていることを示します。

ポインタには、データ メモリを指すポインタ (ram ポインタ ) とプログラム メモリを指すポインタ (rom ポインタ ) の 2 種類があります。ポインタは、明示的に rom ポインタとして宣言しない限り ram ポインタとして扱われます。表 2-4 に示すように、ポインタのサイズはポインタの型によって異なります。

> **注 :** rom 変数に書き込む際、コンパイラは TBLWT 命令を使用します。ただし、使用するメモリの種類によっては特別なアプリケーション コードの作成が必要になることがあります。詳しくはデータシートを参照してください。

**表 2-4: ポインタのサイズ**

| Pointer Type | Example | Size |
|---|---|---|
| Data memory pointer | `char * dmp;` | 16 bits |
| Near program memory pointer | `rom near char * npmp;` | 16 bits |
| Far program memory pointer | `rom far char * fpmp;` | 24 bits |

## 2.5 インクルード ファイルの検索パス

### 2.5.1 システム ヘッダ ファイル

#include <filename> でインクルードされるソース ファイルは、環境変数 MCC_INCLUDE で指定したパスおよびコマンドライン オプション -I で指定したディレクトリを対象に検索されます。複数のディレクトリ内を検索する場合は、環境変数 MCC_INCLUDE およびコマンドライン オプション -I に複数の値をセミコロンで区切って指定します。インクルードされるファイルが環境変数 MCC_INCLUDE のディレクトリ リストとコマンドライン オプション -I のディレクトリ リストの両方に存在する場合は、コマンドライン オプション -I で指定したディレクトリにあるファイルがインクルードされます。これにより、環境変数 MCC_INCLUDE の設定をコマンドライン オプション -I で一時的に変更できます。

### 2.5.2 ユーザー ヘッダ ファイル

#include “filename” でインクルードされるソース ファイルは、インクルード元ファイルのあるディレクトリを対象に検索されます。ファイルが見つからない場合は、システム ヘッダ ファイルの場合と同じディレクトリを対象に検索されます (**2.5.1 項「システム ヘッダ ファイル」**参照 )。

## 2.6 定義済みマクロ名

標準の定義済みマクロ名以外にも、MPLAB C18 には次のマクロがあらかじめ定義されています。

__18CXX 定数 1。MPLAB C18 コンパイラを示す。

__*PROCESSOR* 指定したプロセッサを対象にコンパイルされた場合、定数 1。
例えばコマンドライン オプションに -p18c452 と指定してコンパイルした場合、__18C452 は定数 1 と定義される。コマンドライン オプションに -p18f258 と指定してコンパイルした場合、__18F258 は定数 1 と定義される。

__SMALL__ コマンドライン オプションに -ms を指定してコンパイルした場合、定数 1。

__LARGE__ コマンドライン オプションに -ml を指定してコンパイルした場合、定数 1。

__TRADITIONAL18__ 非拡張モードを使用した場合、定数 1
(**1.2.5 項「モードの選択」**参照 )。

__EXTENDED18__ 拡張モードを使用した場合、定数 1
(**1.2.5 項「モードの選択」**参照 )。

## 2.7 ISO との違い

### 2.7.1 汎整数拡張

ISO では、すべての算術演算を int 以上の精度で実行することが義務づけられています。しかし MPLAB C18 のデフォルト設定では、2 つのオペランドが両方とも int より小さい型の場合を含め、大きい方のオペランドのサイズに合わせて算術演算が実行されます。ISO に準拠した方法で算術演算を実行するには、コマンドライン オプションに -Oi を指定します。

次に例を示します。

```
unsigned char a, b;
unsigned i;

a = b = 0x80;
i = a + b; /* ISO requires that i == 0x100, but in C18 i == 0 */
```

これは、定数リテラルの場合も同様です。定数リテラルの型には、該当するグループから、桁あふれなしに定数の値を表現できる最初の型が選択されます。

次に例を示します。

```
#define A 0x10 /* A will be considered a char unless -Oi
                  specified */
#define B 0x10 /* B will be considered a char unless -Oi
                  specified */
#define C (A) * (B)

unsigned i;
i = C; /* ISO requires that i == 0x100, but in C18 i == 0 */
```

### 2.7.2 数値定数

MPLAB C18 は、16 進数 (0x) および 8 進数 (0) の値を示す標準の接頭子以外にも、2 進数の値を表す接頭子 0b をサポートしています。例えば、10 進数の数値 237 は 2 進数の定数 0b11101101 と表記できます。

### 2.7.3 文字列定数

プログラム メモリに配置されたデータは、主に静的文字列として使用されます。このため、MPLAB C18 では文字列定数はすべて自動的にプログラム メモリに配置されます。このような文字列定数は、「プログラム メモリ内の char の配列」(const rom char []) となります。文字列定数はすべて .stringtable セクションと呼ばれる romdata (**2.9.1 項「#pragma *sectiontype*」**参照 ) セクションに配置されます。次の例では、文字列「hello」は .stringtable セクションに配置されます。

```
strcmppgm2ram (Foo, "hello");
```

このように文字列定数はプログラム メモリに配置されるため、文字列を扱う標準関数にはバリエーションがあります。例えば、strcpy 関数は 4 種類あり、データ メモリとプログラム メモリの間で文字列をコピーできるようになっています。

```
/*
 * Copy string s2 in data memory to string s1 in data memory
 */
char *strcpy (auto char *s1, auto const char *s2);

/*
 * Copy string s2 in program memory to string s1 in data
 * memory
 */
char *strcpypgm2ram (auto char *s1, auto const rom char *s2);

/*
 * Copy string s2 in data memory to string s1 in program
 * memory
 */
rom char *strcpyram2pgm (auto rom char *s1, auto const char *s2);

/*
 * Copy string s2 in program memory to string s1 in program
 * memory
 */
rom char *strcpypgm2pgm (auto rom char *s1,
                         auto const rom char *s2);
```

MPLAB C18 を使用する際は、プログラム メモリ内の文字列テーブルを次のように宣言できます。

```
rom const char table[][20] =   { "string 1", "string 2",
                                 "string 3", "string 4" };
rom const char *rom table2[] = { "string 1", "string 2",
                                 "string 3", "string 4" };
```

table の宣言では、長さ 20 文字の 4 つの文字列の配列が宣言されることにより、80 バイトのプログラム メモリが占有されます。table2 は、プログラム メモリへのポインタの配列として宣言されています。同様に、rom 修飾子の前に * を付けることで、ポインタの配列はプログラム メモリに配置されます。table2 の文字列はすべて長さが 9 バイトで、配列の長さは 4 要素なので、table2 は (9*4+4*2) = 44 バイトのプログラム メモリを占有します。ただし、ポインタの使用によって間接性が増すため、table2 へのアクセスは table へのアクセスよりも効率が悪くなります。

MPLAB C18 ではアドレス空間が分離しているため、プログラム メモリ内のデータを指すポインタとデータ メモリ内のデータを指すポインタには互換性がありません。2 つのポインタ型が互換となるのは、両方が互換性のある型のオブジェクトを指しており、かつこれらのオブジェクトが同じアドレス空間に配置されている場合のみです。例えば、プログラム メモリ内の文字列を指すポインタとデータ メモリ内の文字列を指すポインタは、異なるアドレス空間を指しているため、互換性がありません。

プログラム メモリからデータ メモリへ文字列をコピーする関数は、次のように記述できます。

```
void str2ram(static char *dest, static char rom *src)
{
  while ((*dest++ = *src++) != '\0')
    ;
}
```

次のコードでは、PICmicro MCU の C ライブラリを使用して、プログラム メモリ内の文字列を PIC18C452 の USART に送ります。文字列を USART に送るライブラリ関数 putsUSART(const char *str) は、文字列へのポインタを引き数にとりますが、その文字列はデータ メモリ内にある必要があります。

```
rom char mystring[] = "Send me to the USART";

void foo( void )
{
  char strbuffer[21];
  str2ram (strbuffer, mystring);
  putsUSART (strbuffer);
}
```

または、ライブラリ ルーチンを変更してプログラム メモリ内の文字列を読み出すこともできます。

```
/*
 * The only changes required to the library routine are to
 * change the name so the new routine does not conflict with
 * the original routine and to add the rom qualifier to the
 * parameter.
 */
void putsUSART_rom( static const rom char *data )
{
  /* Send characters up to the null */
  do
  {
    while (BusyUSART())
      ;

    /* Write a byte to the USART */
    putcUSART (*data);
  } while (*data++);
}
```

### 2.7.4 stdio.h 関数

stdio.h で定義された出力関数は、プログラム メモリ内のデータ、浮動小数点形式のサポート、MPLAB C18 固有の拡張といった点で、ANSI の定義と異なります。

関数 puts と fputs の出力文字列は、プログラム メモリに格納されます。関数 vsprintf、vprintf、sprintf、printf、fprintf、vfprintf の書式付き出力の文字列は、プログラム メモリに格納されます。

関数 vsprintf、vprintf、sprintf、printf、fprintf、vfprintf は、浮動小数点変換修飾子をサポートしていません。

24 ビット整数およびプログラム メモリ内のデータに関する MPLAB C18 固有の拡張については、『*MPLAB® C18 C Compiler Libraries*』の 4.7 項を参照してください。

## 2.8 言語の拡張仕様

### 2.8.1 匿名構造体

MPLAB C18 では、共用体の中に匿名構造体を使用できます。匿名構造体の形式は次のとおりです。

```
struct { member-list };
```

匿名構造体では、名前のないオブジェクトが定義されます。匿名構造体のメンバーの名前は、その構造体を宣言したスコープ内の他の名前との重複を避ける必要があります。これらのメンバーは、同じスコープ内であれば通常のメンバー アクセス構文なしで直接使用できます。

次に例を示します。

```
union foo
{
  struct
  {
    int a;
    int b;
  };
  char c;
} bar;
char c;

...

bar.a = c; /* 'a' is a member of the anonymous structure
              located inside 'bar' */
```

構造体に対してオブジェクトまたはポインタが宣言されている場合は、匿名構造体にはなりません。次に例を示します。

```
union foo
{
  struct
  {
    int a;
    int b;
  } f, *ptr;
  char c;
} bar;
char c:

...

bar.a = c;      /* error */
bar.ptr->a = c; /* ok */
```

上記の例で bar.a への代入は、メンバー名がどのオブジェクトにも関連付けられていないため、不正となります。

### 2.8.2 インライン アセンブリ

MPLAB C18 には、MPASM アセンブラと同様の構文を使用する内部アセンブラを実装しています。アセンブリ コードのブロックは _asm で開始し、_endasm で終了します。ブロック内の構文は次のとおりです。

```
[label:] [<instruction> [arg1[, arg2[, arg3]]]]
```

内部アセンブラは、次の点で MPASM アセンブラと異なります。

- ディレクティブはサポートされない
- コメントには C または C++ の表記法を使用する
- テーブルの読み出し / 書き込みには、次に列挙する完全なテキスト ニーモニックが必要
  - TBLRD
  - TBLRDPOSTDEC
  - TBLRDPOSTINC
  - TBLRDPREINC
  - TBLWT
  - TBLWTPOSTDEC
  - TBLWTPOSTINC
  - TBLWTPREINC
- 命令オペランドはデフォルトなし – オペランドはすべて明示的な指定が必要
- デフォルトの基数は 10 進数
- リテラルは、MPASM アセンブラの表記ではなく C の基数表記を使用する。例えば、16 進数は H'1234' ではなく 0x1234 と表記する
- ラベルにはコロンが必要
- インデックス アドレス構文 ([]) のサポートはなし – 必ずリテラルとアクセス ビットを指定すること ( 例 : CLRF [2] ではなく CLRF 2,0 と指定 )

次に例を示します。

```
_asm
  /* User assembly code */
  MOVLW 10       // Move decimal 10 to count
  MOVWF count, 0

  /* Loop until count is 0 */
  start:
    DECFSZ count, 1, 0
    GOTO done
    BRA start
  done:
_endasm
```

一般に、インライン アセンブリの使用は最小限にとどめることを推奨します。インライン アセンブリを含む関数は、コンパイラによって最適化されません。アセンブリ コードで記述する部分が多い場合は、MPASM アセンブラでモジュールを作成した後、MPLINK リンカで C モジュールにリンクするようにしてください。

## 2.9 プラグマ

### 2.9.1 #pragma *sectiontype*

セクション宣言プラグマは、MPLAB C18 がそれに関連するタイプの情報を配置するセクションを変更するために使用します。

セクションとは、アプリケーションの一部を特定のメモリ番地に配置したものです。セクションには、コードまたはデータを含めることができます。セクションは、プログラム メモリまたはデータ メモリのいずれかに配置できます。プログラム メモリとデータ メモリそれぞれに 2 種類のセクションがあります。

- プログラム メモリ
  - code – 実行可能な命令を配置する
  - romdata – 変数と定数を配置する
- データ メモリ
  - udata – 静的に割り当てられた、初期化しないユーザー変数を配置する
  - idata – 静的に割り当てられた、初期化するユーザー変数を配置する

セクションには、絶対セクション、割り当て済みセクション、未割り当てセクションの 3 つに分類されます。絶対セクションとは、セクション宣言プラグマの =address で明示的なアドレスを与えられたセクションをいいます。割り当て済みセクションとは、リンカ スクリプトの SECTION ディレクティブで特定のセクションに割り当てられたものをいいます。未割り当てセクションとは、絶対セクションと割り当て済みセクションのいずれにも該当しないものをいいます。

#### 2.9.1.1 シンタックス

*section-directive:*

```
    # pragma udata [attribute-list] [section-name [=address]]
  | # pragma idata [attribute-list] [section-name [=address]]
  | # pragma romdata [overlay] [section-name [=address]]
  | # pragma code [overlay] [section-name [=address]]
```

*attribute-list:*

```
    attribute
  | attribute-list attribute
```

*attribute:*

```
    access
  | overlay
```

*section-name:* C 識別子

*address:* 整定数

#### 2.9.1.2 セクションの内容

code セクションには、プログラム メモリ内の実行可能な命令が配置されます。romdata セクションには、プログラム メモリに割り当てられているデータ ( 通常は rom 修飾子付きで宣言されている変数 ) が配置されます。romdata の使用方法 ( メモリ マップド ペリフェラルの場合など ) についての詳細は、『*MPASM™ Assembler, MPLINK™ Object Linker, MPLIB™ Object Librarian User's Guide*』(DS33014) を参照してください。udata セクションには、データ メモリに静的に割り当てられている、初期化しないグローバル データが配置されます。idata セクションには、データ メモリに静的に割り当てられている、初期化するグローバル データが配置されます。

次の例の各オブジェクトは、表 2-5 に示すセクションにそれぞれ配置されます。

```
rom int ri;
rom char rc = 'A';

int ui;
char uc;

int ii = 0;
char ic = 'A';

void foobar (void)
{
  static rom int foobar_ri;
  static rom char foobar_rc = 'Z';
  ...
}
void foo (void)
{
  static int foo_ui;
  static char foo_uc;
  ...
}

void bar (void)
{
  static int bar_ii = 5;
  static char bar_ic = 'Z';
  ...
}
```

**表 2-5:　各オブジェクトが配置されるセクション**

| Object | Section Location |
|---|---|
| ri | romdata |
| rc | romdata |
| foobar_ri | romdata |
| foobar_rc | romdata |
| ui | udata |
| uc | udata |
| foo_ui | udata |
| foo_uc | udata |
| ii | idata |
| ic | idata |
| bar_ii | idata |
| bar_ic | idata |
| foo | code |
| bar | code |
| foobar | code |

#### 2.9.1.3　デフォルトのセクション

MPLAB C18 では、各セクション タイプにデフォルトのセクションがあります ( 表 2-6 参照 )。

**表 2-6:　デフォルトのセクション名**

| Section Type | Default Name |
|---|---|
| code | .code_*filename* |
| romdata | .romdata_*filename* |
| udata | .udata_*filename* |
| idata | .idata_*filename* |

*filename* は、生成されるオブジェクト ファイルの名前です。例えば、“mcc18 foo.c -fo=foo.o” で生成されるオブジェクト ファイルのデフォルトの code セクション名は、“.code_foo.o” となります。

以前宣言したセクション名を指定すると、MPLAB C18 は該当するタイプのデータを再びそのセクションに割り当てます。セクションの属性は、以前に宣言したものと一致している必要があります。属性が異なると、エラーとなります ( **付録 D「MPLAB C18 メッセージ一覧」**参照 )。

セクション プラグマ ディレクティブで名前を指定しない場合は、該当するタイプのデータは現在のモジュールのデフォルトのセクションに割り当てられることになります。次に例を示します。

```
/*
 * The following statement changes the current code
 * section to the absolute section high_vector
 */
#pragma code high_vector=0x08
...

/*
 * The following statement returns to the default code
 * section
 */
#pragma code
...
```

MPLAB C18 コンパイラがソース ファイルのコンパイルを開始すると、初期化するデータと初期化しないデータのそれぞれにデフォルトのデータ セクションが作成されます。これらデフォルトのセクションは、コンパイラの起動オプションに -Oa+ を指定した場合はアクセス RAM に、指定しない場合はアクセス RAM 以外の RAM に配置されます。コマンドライン オプション -Oa+ は、非拡張モードでのみ有効です (**1.2.5 項「モードの選択」**参照 )。ソース コードで #pragma udata [access] *name* ディレクティブが検出されると、初期化しないデータの現在のデータ セクションは *name* となります。このデータ セクションは、オプションの access 属性を指定した場合はアクセス RAM に配置され、指定しない場合はアクセス RAM 以外の RAM に配置されます。同様に、初期化するデータの場合も #pragma idata [access] *name* ディレクティブが検出されると、現在のデータ セクションが変更されます。

オブジェクトの定義で明示的な初期化子を指定している場合、オブジェクトは初期化するデータの現在のデータ セクションに配置されます。オブジェクトの定義で明示的な初期化子を指定しない場合、オブジェクトは初期化しないデータの現在のデータ セクションに配置されます。例えば次のコードでは、i は初期化するデータの現在のデータ セクションに配置され、u は初期化しないデータの現在のデータセクションに配置されます。

```
int i = 5;
int u;

void main(void)
{
   ...
}
```

オブジェクトの定義で明示的に far 修飾子 (**2.4 項「ストレージ修飾子」**参照 ) を指定すると、オブジェクトはアクセス RAM 以外の RAM に配置されます。同様に、明示的に near 修飾子 (**2.4 項「ストレージ修飾子」**参照 ) を指定すると、コンパイラはそのオブジェクトをアクセス RAM に配置します。オブジェクトの定義で near と far のどちらの修飾子も指定しない場合、コマンドライン オプションに -Oa+ が指定されているかどうかでコンパイラの動作が決定します。

#### 2.9.1.4 予約されているセクション名

表 2-7 に、コンパイラ専用に予約されているセクション名の一覧を示します。

**表 2-7: 予約されているセクション名**

| Section Name | Purpose |
|---|---|
| _entry_scn | Contains a jump to the start-up code. Located at the RESET vector. |
| _startup_scn | Contains the start-up code, which calls the application's main() function. |
| _cinit_scn | Contains the start-up function that performs data initialization. |
| .cinit | Contains a copy of initialized data in program memory that is used by the start-up code to perform the initialization. |
| MATH_DATA | Contains arguments, return values, and temporary locations used by the math library functions. |
| .tmpdata | Contains the compiler temporary variables for the non-interrupt service routine source. |
| *isr*_tmp | Contains the compiler temporary variables for the interrupt service routine, *isr* (see **Section 2.9.2 "#pragma interruptlow fname / #pragma interrupt fname"**). |
| .stringtable | Contains all constant strings (see **Section 2.7.3 "String Constants"**). |

**表 2-7: 予約されているセクション名 ( 続き )**

| Section Name | Purpose |
|---|---|
| .code_*filename* | Contains, by default, the executable content for the file, *filename*. |
| .idata_*filename* | Contains, by default, the initialized data for the file, *filename*. |
| .udata_*filename* | Contains, by default, the uninitialized data for the file, *filename*. |
| .romdata_*filename* | Contains, by default, the data allocated in program memory for the file, *filename*. |
| .config_*address*_*filename* | Contains the configuration settings specified for the given *address* and *filename*. |
| .stack | Contains the software stack. |
| CTYPE | Contains the executable content for the character classification functions (see *MPLAB® C18 C Compiler Libraries*). |
| D100TCYXCODE | Contains the library function Delay100TCYx (see *MPLAB® C18 C Compiler Libraries*). |
| D10KTCYXCODE | Contains the library function Delay10KTCYx (see *MPLAB® C18 C Compiler Libraries*). |
| D10TCYXCODE | Contains the library function Delay10TCYx (see *MPLAB® C18 C Compiler Libraries*). |
| D1KTCYXCODE | Contains the library function Delay1KTCYx (see *MPLAB® C18 C Compiler Libraries*). |
| DELAYDAT1 | Contains uninitialized data used by some of the Delay functions (see *MPLAB® C18 C Compiler Libraries*). |
| DELAYDAT2 | Contains uninitialized data used by some of the Delay functions (see *MPLAB® C18 C Compiler Libraries*). |
| PROG | Contains the executable content of the math library (see *MPLAB® C18 C Compiler Libraries*). |
| SEED_DATA | Contains the initialized data used by rand and srand functions (see *MPLAB® C18 C Compiler Libraries*). |
| SFR_BANKED* | Contains the SFRs located in banked RAM. |
| SFR_UNBANKED* | Contains the SFRs located in access RAM. |
| STDIO | Contains the executable content of the peripheral output routines for the standard library output functions. |
| STDLIB | Contains the executable content of the data conversion functions (see *MPLAB® C18 C Compiler Libraries*). |
| STRING | Contains the memory and string manipulation functions (see *MPLAB® C18 C Compiler Libraries*). |
| UARTCODE | Contains the executable content for the software UART functions (see *MPLAB® C18 C Compiler Libraries*). |
| UARTDATA | Contains uninitialized data used by the software UART functions (see *MPLAB® C18 C Compiler Libraries*). |

* はワイルドカードを示します。

#### 2.9.1.5 セクションの属性

#pragma *sectiontype* ディレクティブには、オプションとして access または overlay という 2 つのセクション属性を含めることができます。

##### 2.9.1.5.1 access

access 属性を指定すると、コンパイラは指定したセクションをデータ メモリのアクセス領域に配置します ( アクセス データ メモリについての詳細は、各デバイスのデータシートまたは『*PICmicro® 18C MCU Family Reference Manual*』(DS39500) を参照してください )。

access 属性を指定したデータ セクションは、リンカ スクリプト ファイルで ACCESSBANK と定義されたメモリ領域に配置されます。このメモリ領域は命令のアクセス ビットでアクセスするため、バンク切り替えは不要です ( 詳細については、各デバイスのデータシートを参照してください )。access セクションに配置した変数は、near キーワードを付けて宣言する必要があります。次に例を示します。

```
#pragma udata access my_access
/* all accesses to these will be unbanked */
near unsigned char av1, av2;
```

##### 2.9.1.5.2 overlay

overlay 属性を指定すると、同じ物理アドレスに複数のセクションを配置できます。このように同じ番地に複数の変数を配置することにより、メモリを節約できます ( ただし両方の変数が同時にアクティブにならないことが条件です )。overlay 属性は access 属性と同時に指定できます。

2 つのセクションをオーバーレイするには、次の 4 つの条件を満たす必要があります。

1. 2 つのセクションが別々のソース ファイルにあること
2. 2 つのセクションが同じ名前であること
3. 片方のセクションで access 属性を指定している場合は、もう片方のセクションにも指定すること
4. 片方のセクションで絶対アドレスを指定している場合は、もう片方のセクションにも同じ絶対アドレスを指定すること

overlay 属性のあるコード セクションは、他の overlay コード セクションと重複する番地に配置できます。次に例を示します。

**file1.c:**

```
#pragma code overlay my_overlay_scn=0x1000
void f (void)
{
 ...
}
```

**file2.c:**

```
#pragma code overlay my_overlay_scn=0x1000
void g (void)
{
 ...
}
```

overlay 属性のあるデータ セクションは、他の overlay データ セクションと重複する番地に配置できます。これは、同時にアクティブになることのない複数の変数で、同じデータ範囲を使用する場合に有用になります。次に例を示します。

**file1.c:**

```
#pragma udata overlay my_overlay_data=0x1fc
/* 2 bytes will be located at 0x1fc and 0x1fe */
int int_var1, int_var2;
```

**file2.c:**

```
#pragma udata overlay my_overlay_data=0x1fc
/* 4 bytes will be located at 0x1fc */
long long_var;
```

overlay セクションの詳細については、『*MPASM™ Assembler, MPLINK™ Object Linker, MPLIB™ Object Librarian User's Guide*』(DS33014) を参照してください。

#### 2.9.1.6 コードの配置

#pragma code ディレクティブが検出されると、次に別の #pragma code ディレクティブが検出されるまで、生成されるすべてのコードは指定されたコード セクションに配置されます。絶対コード セクションを指定すると、コードを特定の番地に配置できます。次に例を示します。

```
#pragma code my_code=0x2000
```

上記の例では、コード セクション my_code が、プログラム メモリの 0x2000 番地に配置されます。

コード セクションは、リンカによって強制的にプログラム メモリ領域に配置されますが、コード セクションを配置するメモリ領域の指定も可能です。セクションを特定のメモリ領域に割り当てるには、リンカ スクリプトの SECTION ディレクティブを使用します。次のリンカ スクリプト ディレクティブは、my_code1 というコード セクションをメモリ領域 page1 に割り当てます。

```
SECTION NAME=my_code1 ROM=page1
```

#### 2.9.1.7 データの配置

MPLAB C18 コンパイラでは、データをデータ メモリとプログラム メモリのいずれにも配置できます。オンチップのプログラム メモリに配置されたデータの読み出しは可能ですが、書き込みにはユーザーが特別なコードを使用する必要があります。一般に、外部プログラム メモリに配置されたデータの読み出しや書き込みには、ユーザーが特別なコードを用意する必要はありません。

次に示す例では、静的に割り当てられた初期化しないデータ (udata) のセクションを絶対アドレス 0x120 番地に配置することを宣言しています。

```
#pragma udata my_new_data_section=0x120
```

rom キーワードは、コンパイラに対して変数をプログラム メモリに配置するように指定します。コンパイラは、この変数を現在の romdata タイプのセクションに割り当てます。次に例を示します。

```
#pragma romdata const_table
const rom char my_const_array[10] = {0, 1, 2, 3, 4, 5,
                                     6, 7, 8, 9};

/* Resume allocation of romdata into the default section */
#pragma romdata
```

romdata セクションはリンカによって強制的にプログラム メモリ領域に配置され、udata セクションと idata セクションはデータ メモリ領域に配置されますが、データ セクションを配置するメモリ領域は指定もできます。セクションを特定のメモリ領域に割り当てるには、リンカ スクリプトの SECTION ディレクティブを使用します。次の例では、udata セクションの my_data がメモリ領域 gpr1 に割り当てられます。

```
SECTION NAME=my_data RAM=gpr1
```

### 2.9.2 #pragma interruptlow *fname* / #pragma interrupt *fname*

interrupt プラグマは高優先度の割り込みサービス ルーチン (ISR) となる関数を宣言し、interruptlow プラグマは低優先度の割り込みサービス ルーチンとなる関数を宣言します。

割り込みが発生すると、現在動作中のアプリケーションの実行を一時停止し、現在のコンテクスト情報を保存してから制御を ISR に渡してイベントを処理します。ISR が完了すると、以前のコンテクスト情報が復元され、アプリケーションの通常の実行が再開されます。割り込みの際には、WREG、BSR、STATUS の内容が最小コンテクストとして保存および復元されます。最小コンテクストの保存と復元には、高優先度の割り込みではシャドー レジスタが使用されますが、低優先度の割り込みではソフトウェア スタックが使用されます。このため、高優先度の割り込みは高速な “return from interrupt” で終了できますが、低優先度の割り込みは通常の “return from interrupt” で終了します。ソフトウェア スタックを使用してコンテクストを保存した場合、1 バイトにつき 2 つの MOVFF 命令が必要 ( ただし WREG は MOVWF 命令と MOVF 命令の 2 つが必要 ) となるため、最小コンテクストの保存に関しては、低優先度の割り込みでは高優先度の割り込みよりも 10 ワード余分にオーバーヘッドが発生します。

割り込みサービス ルーチンが使用する一時データ セクションは、通常の C 関数で使用するものとは区別されています。割り込みサービス ルーチン内の式の評価で必要となる一時データはすべてこのセクションに割り当てられ、他の関数 ( 割り込み関数を含む ) の一時データ セクションの番地と重複することはありません。interrupt プラグマを使用すると、割り込み用の一時データ セクションに名前を付けることができます。このセクションに名前を付けない場合は、*fname*_tmp という名前の udata セクションに一時変数が作成されます。次に例を示します。

```
void foo(void);
...
#pragma interrupt foo
void foo(void)
{
  /* perform interrupt function here */
}
```

この場合、コンパイラは割り込みサービス ルーチン foo の一時変数を foo_tmp という名前の udata セクションに配置します。

#### 2.9.2.1 シンタックス

*interrupt-directive:*

**# pragma interrupt** *function-name* [*tmp-section-name*][**save=***save-list*][**nosave=***nosave-list*]
| **# pragma interruptlow** *function-name* [*tmp-section-name*][**save=***save-list*][**nosave=***nosave-list*]

*save-list:*

*location-specifier*
| *save-list, location-specifier*

*nosave-list:*

*location-specifier*
| *nosave-list, location-specifier*

*location-specifier:*

*symbol-name*
| **section("***section-name***")**

*function-name:* C 識別子 – ISR となる C 関数の名前を指定

*tmp-section-name:* C 識別子 – ISR の一時データを割り当てるセクションの名前を指定

*symbol-name:* C 識別子 – 割り込み処理後に復元される変数の名前を指定

*section-name:* C 識別子、ただし先頭がドット (.) でも可 – 割り込み処理後に復元されるセクションの名前を指定

#### 2.9.2.2 割り込みサービス ルーチン

MPLAB C18 の ISR は、ローカル変数があり、グローバル変数にアクセスできるという点では通常の C 関数と同じですが、パラメータおよび戻り値なしで宣言する必要がある点が異なります。これは、ISR はハードウェア割り込みに応答する形で非同期に呼び出されるためです。ISR とメインラインの関数の両方からアクセスされるグローバル変数は、volatile として宣言してください。

ISR は必ずハードウェア割り込みによって呼び出し、他の C 関数から呼び出さないようにします。ISR が関数から抜ける際には、通常の RETURN 命令ではなく RETFIE (return from interrupt) 命令を使用します。コンテクストとは無関係に高速な RETFIE 命令を実行すると、WREG、BSR、STATUS レジスタの内容が破壊されます。

#### 2.9.2.3 割り込みベクタ

MPLAB C18 では、ISR は自動的には割り込みベクタに配置されません。一般には、割り込みベクタに GOTO 命令を配して制御を ISR に渡します。次に例を示します。

```
#include <p18cxxx.h>

void low_isr(void);
void high_isr(void);

/*
 * For PIC18 devices the low interrupt vector is found at
 * 00000018h. The following code will branch to the
 * low_interrupt_service_routine function to handle
 * interrupts that occur at the low vector.
 */
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
  _asm GOTO low_isr _endasm
}
#pragma code /* return to the default code section */

#pragma interruptlow low_isr
void low_isr (void)
{
  /* ... */
}

/*
 * For PIC18 devices the high interrupt vector is found at
 * 00000008h. The following code will branch to the
 * high_interrupt_service_routine function to handle
 * interrupts that occur at the high vector.
 */
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
  _asm GOTO high_isr _endasm
}
#pragma code /* return to the default code section */

#pragma interrupt high_isr
void high_isr (void)
{
  /* ... */
}
```

詳細な例については、**第 5 章「サンプル コード」**を参照してください。

#### 2.9.2.4 ISR のコンテクスト保存

MPLAB C18 では、デフォルトでコンパイラ管理リソース (**3.4 項「コンパイラ管理リソース」**参照 ) が保存されますが、これ以外にも save= 節で指定した任意のシンボルを関数によって保存および復元できます。

ユーザー定義のグローバル変数 myint を保存するには、次のプラグマ ディレクティブを使用します。

```
#pragma interrupt high_interrupt_service_routine save=myint
```

save= 節には、変数以外にもデータ セクション全体の指定ができます。例えば、mydata という名前のユーザー定義セクションを保存するには、次のようなプラグマ ディレクティブを使用します。

```
#pragma interrupt high_interrupt_service_routine save=section("mydata")
```

これまでのすべての例では、保存する値は 1 つだけでした。save= 節では、複数の変数およびセクションを指定して保存することもできます。割り込みサービス ルーチンで変数 myint およびセクション mydata を使用している場合は、割り込みプラグマ ディレクティブで save=myint, section ("mydata") と指定して、これらを保存するようにしてください。次に例を示します。

```
#pragma interrupt isr save=myint, section("mydata")
```

#### 2.9.2.5 割り込み専用リソースの指定

アプリケーションによっては、コンパイラによって保存される番地を割り込みコンテクストでしか使用しない場合があります。このような番地は、必ずしも保存や復元を必要としません。nosave= 節を使用すると、コンパイラ管理リソースの使用範囲は割り込みサービス ルーチンに限定されており、割り込み実行時に退避する必要がないよう指定できます。高優先度の割り込みの場合、nosave= 節には、FSR0、TBLPTR、TBLPTRU、TABLAT、PCLATH、PCLATU、PROD、section(".tmpdata")、section("MATH_DATA") を指定できます。低優先度の割り込みの場合、nosave= 節には上記の高優先度の割り込みで指定できる項目に加え、WREG、BSR、STATUS も指定対象となります。また、拡張モードでのコンパイル時は、MATH_DATA セクションへの参照として __RETVAL0 も指定できます。

例えば、TBLPTR と TABLAT レジスタが高優先度の割り込み関数 foo のコンテクスト内でしか使用されず、コンパイラによる保存と復元が不要な場合は、次のようなプラグマ ディレクティブを使用します。

```
#pragma interrupt foo nosave=TBLPTR, TABLAT
```

#### 2.9.2.6 遅延

割り込みが発生してから ISR の最初の命令が実行されるまでの時間を、割り込みの遅延といいます。遅延には次の 3 つの要素が影響します。

1. **プロセッサによる割り込みの処理** : プロセッサが割り込みを認識し、割り込みベクタの最初のアドレスに分岐するまでの時間。この値を求めるには、個々のプロセッサおよび使用する割り込みソースに応じて、プロセッサのデータシートを参照してください。
2. **割り込みベクタの実行** : ISR への分岐命令が格納されている割り込みベクタのコードを実行するのに必要な時間。
3. **ISR プロローグ コード** : MPLAB C18 がコンパイラ管理リソース、および save= リストで指定されているデータを保存するのに必要な時間。

#### 2.9.2.7 割り込みの入れ子

低優先度の割り込みは、アクティブなレジスタがソフトウェア スタックに保存されるため、入れ子にできます。高優先度の割り込みサービス ルーチンの場合は、1 レベル分しかないハードウェアのシャドー レジスタを使用するため、一度に複数のインスタンスをアクティブにはできません。

低優先度の割り込みを入れ子にする場合は、ISR の先頭付近に GIEL ビットをセットするステートメントを追加します。詳細については、プロセッサのデータシートを参照してください。

### 2.9.3 #pragma tmpdata [*section-name*]

tmpdata プラグマは、コンパイラが一時変数を作成する現在のセクションを変更します。デフォルトでは、MPLAB C18 はメインライン関数 (ISR 以外の関数 ) の一時変数を、.tmpdata という名前のセクションに作成します。

次に示すステートメントでは、現在の一時データ セクションを user_tmp という名前のセクションに変更します。コンパイラは、このステートメントから次の #pragma tmpdata ディレクティブまでの間にある関数 (ISR を除く ) の一時変数を、すべて user_tmp セクションに作成します。

```
#pragma tmpdata user_tmp
```

次に示すステートメントでは、一時データ セクションをデフォルトの一時データ セクション (.tmpdata) にリセットします。

```
#pragma tmpdata
```

> **注 :** コンパイラが ISR のコンテクストを保存する際、#pragma tmpdata ディレクティブによって作成された一時データ セクションは保存されません。コンパイラが保存するのは、デフォルトの一時データ セクションである .tmpdata のみです。.tmpdata セクションを保存する必要がない場合は、interrupt プラグマの nosave= 節 (**2.9.2 項「#pragma interruptlow fname / #pragma interrupt fname」**参照 ) で指定することで、コンテクスト保存のオーバーヘッドを低減できます。

#### 2.9.3.1 シンタックス

*tmpdata-directive:*

**#pragma tmpdata** [*section-name*]

*section-name*: C 識別子 – コンパイラが一時変数を作成する一時データ セクションの名前を指定

#### 2.9.3.2 ISR 関数と ISR 以外の関数による一時データ セクションの共有

tmpdata プラグマを使用すると、割り込みサービス ルーチンの関数と割り込み以外の関数との間で一時データを共有できます。次の例では、割り込みサービス ルーチン isr と割り込みでない関数 increment のコンパイラ一時変数が、どちらも isr_tmp という名前の udata セクションに配置されます。

```
void increment (int counter);
void isr (void);
#pragma interrupt isr nosave=section(".tmpdata")
void isr (void)
{
  static int foo = 0;
  ...
  increment (foo);
  ...
}
#pragma tmpdata isr_tmp
void increment (int counter)
{
  ...
}
#pragma tmpdata
```

**注 :** ISR によって呼び出された関数の一時データ セクションを tmpdata プラグマで別途指定している場合でも、コンパイラは ISR のコンテクストを保存する際、自動的にデフォルトの一時データ セクションである .tmpdata を保存します。.tmpdata セクションを保存する必要がない場合は、interrupt プラグマの nosave= 節 (**2.9.2 項「#pragma interruptlow fname / #pragma interrupt fname」**参照 ) で指定することで、コンテクスト保存のオーバーヘッドを低減できます。

#### 2.9.3.3 複数の高優先度割り込み

高優先度割り込みは 1 度に 1 つしか処理されないため、同じ一時データ セクションを複数の高優先度割り込みで使用したり、各 ISR によって呼び出される関数と共有したりできます。次に示す例では、高優先度の ISR と、2 つの ISR から呼び出される関数 increment の両方が、コンパイラによって生成される一時データの保存にセクション isr_tmp を使用しています。

```
void increment (int counter);
void isr1 (void);
void isr2 (void);

#pragma interrupt isr1 isr_tmp nosave=section(".tmpdata")
void isr1 (void)
{
  static int foo = 0;
  ...
  increment (foo);
  ...
}

#pragma interrupt isr2 isr_tmp nosave=section(".tmpdata")
void isr2 (void)
{
  static int foo = 0;
  ...
  increment (foo);
  ...
}

#pragma tmpdata isr_tmp
void increment (int counter)
{
  ...
}
#pragma tmpdata
```

#### 2.9.3.4 入れ子の割り込み

入れ子の割り込みでは、一時データの扱いが複雑になります。割り込みを入れ子にした場合は、割り込みサービス ルーチンが使用している一時データの保存には特に注意が必要です。次に示す例では、割り込みサービス ルーチンの一時データ用のセクションは、割り込みサービス ルーチンに入る際に必ず保存しておく必要があります。

```
void increment (int counter);
void isr1 (void);
void isr2 (void);

#pragma interrupt isr1 isr_tmp save=section("isr_tmp") nosave=section(".tmpdata")
void isr1 (void)
{
  static int foo = 0;
  ...
  increment (foo);
  ...
}
#pragma interruptlow isr2 isr_tmp save=section("isr_tmp") nosave=section(".tmpdata")
void isr2 (void)
{
  static int foo = 0;

  INTCONbits.GIE = 1;
  ...
  increment (foo);
  ...
}

#pragma tmpdata isr_tmp
void increment (int counter)
{
   ...
}
#pragma tmpdata
```

### 2.9.4 #pragma varlocate *bank variable-name*<br>#pragma varlocate "*section-name*" *variable-name*

varlocate プラグマは、変数がリンク時にどこに配置されるかをコンパイラに知らせることにより、コンパイラがバンク スイッチングを効率化できるようにします。

varlocate の指定のみでは、コンパイラやリンカで強制的に実行されません。その変数を含むセクションは、リンカ スクリプトを使用するか、変数が定義されたモジュールで絶対セクションを指定することにより、適切なバンクに明示的に割り当てておく必要があります。

#### 2.9.4.1 シンタックス

*variable-locate-directive:*

**# pragma varlocate** *bank variable-name*[, *variable-name*...]<br>| **# pragma varlocate** "*section-name*" *variable-name*[, *variable-name*...]

*bank:* 整定数

*variable-name:* C 識別子

*section-name:* C 識別子

#### 2.9.4.2 # pragma varlocate *bank variable-name* の使用例

1 番目のファイルで、c1 と c2 を明示的にバンク 1 に割り当てます。

```
#pragma udata bank1=0x100
signed char c1;
signed char c2;
```

2 番目のファイルで、c1 と c2 が両方ともバンク 1 に存在することをコンパイラに知らせます。

```
#pragma varlocate 1 c1
extern signed char c1;

#pragma varlocate 1 c2
extern signed char c2;

void main (void)
{
  c1 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c2 += 5;
}
```

2 番目のファイルで c1 と c2 が使用されると、コンパイラはこれらの変数が両方とも同じバンクに存在しており、c1 の直後に c2 を使用する場合は MOVLB 命令を省略できると判断します。

#### 2.9.4.3 # pragma varlocate "*section-name*" *variable-name* の使用例

1 番目のファイルで、c3 と c4 が my_section という名前の udata セクションに作成されます。

```
#pragma udata my_section
signed char c3;
signed char c4;
#pragma udata
```

2 番目のファイルで、c3 と c4 が両方とも my_section という名前の udata セクションに存在することをコンパイラに知らせます。

```
#pragma varlocate "my_section" c3, c4
extern signed char c3;
extern signed char c4;

void main (void)
{
  c3 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c4 += 5;
}
```

2 番目のファイルで c3 と c4 が使用されると、コンパイラはこれらの変数が両方とも同じセクションに存在しており、c3 の直後に c4 を使用する場合は MOVLB 命令を省略できると判断します。

### 2.9.5 #pragma config

#pragma config ディレクティブは、アプリケーションで使用するプロセッサ固有の構成設定 ( 構成ビット ) を指定します。

構成設定は、複数の #pragma config ディレクティブを使用して指定できます。MPLAB C18 は、指定されている構成設定がコンパイル対象のプロセッサで有効であるかどうかを検証します。#pragma config ディレクティブで値が指定されていない構成バイトがある場合は、その設定に関するビットはデフォルトの値となります。

#pragma config ディレクティブで構成バイトを設定すると、その 1 バイトごとに .config_*address*_*filename* (*address* は構成バイトの 16 進数アドレス、*filename* は生成されるオブジェクト ファイルの名前 ) という名前の絶対 romdata セクションがコンパイラによって生成されます。例えば、0x300001 番地の構成バイトに対して構成設定を指定し、コマンドライン オプションで "mcc18 foo.c -fo=foo.o" と指定してソース ファイルをコンパイルすると、.config_300001_foo.o という名前の romdata セクションが作成されます。

### 2.9.5.1 シンタックス

*pragma-config-directive*:
**# pragma config** *setting-list*
*setting-list*:
*setting*
| *setting-list*, *setting*
*setting*:
*setting-name* **=** *value-name*

*setting-name* と *value-name* はデバイス固有であり、コマンドライン オプションの --help-config を使用して指定できます。また、各デバイスで利用できる設定項目とそれぞれの値については、『*PIC18 Configuration Settings Addendum*』(DS51537) にも記載されています。

### 2.9.5.2 使用例

次に、#pragma config ディレクティブの使用例を示します。このサンプルでは、下記の内容を実行しています。

- ウォッチドッグ タイマを有効にする
- ウォッチドッグ タイマのポストスケーラ比を 1:128 に設定する
- HS オシレータを選択する

```
#pragma config WDT = ON, WDTPS = 128
#pragma config OSC = HS
...
void main (void)
{
...
}
```

## 2.10 プロセッサ固有ヘッダ ファイル

プロセッサ固有ヘッダ ファイルとは、レジスタ定義ファイル (**2.11 項「プロセッサ固有レジスタ定義ファイル」**参照 ) で定義された特殊機能レジスタを外部宣言する C ファイルです。例えば、PIC18C452 プロセッサ固有ヘッダ ファイルでは、PORTA が次のように宣言されます。

```
extern volatile near unsigned char PORTA;
```

更に、次のように宣言します。

```
extern volatile near union  {
  struct {
    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
  } ;
  struct {
    unsigned AN0:1;
    unsigned AN1:1;
    unsigned AN2:1;
    unsigned AN3:1;
    unsigned T0CKI:1;
    unsigned SS:1;
    unsigned OSC2:1;
  } ;
  struct {
    unsigned :2;
    unsigned VREFM:1;
    unsigned VREFP:1;
    unsigned :1;
    unsigned AN4:1;
    unsigned CLKOUT:1;
  } ;
  struct {
    unsigned :5;
    unsigned LVDIN:1;
  } ;
} PORTAbits ;
```

最初の宣言では、PORTA が 1 バイト (unsigned char) であることを指定しています。変数がレジスタ定義ファイルで宣言されているため、extern 修飾子が必要です。volatile 修飾子を付けることによって、PORTA に代入された値が変更される可能性があることをコンパイラに知らせます。near 修飾子を指定することにより、このポートはアクセス RAM に配置されます。

2 番目の宣言では、PORTAbits がビット アドレス指定可能な匿名構造体 (**2.8.1 項「匿名構造体」**参照 ) の共用体であることを指定しています。特殊機能レジスタの個々のビットは複数の機能 ( すなわち複数の名前 ) を持つ場合があるため、共用体の中には同じレジスタを参照する構造体が複数定義されています。これらの構造体で定義される各ビットは、どの構造体のものもすべてレジスタ内の同じビットを参照しています。ビットがその位置で 1 つしか機能を持たない場合は、他の構造体の定義は単にパディングされます。例えば、PORTA のビット 6 には 4 つの名前があり、すべての構造体で定義されていますが、ビット 1 とビット 2 には 2 つしか名前がないため、3 番目と 4 番目の構造体では単にパディングされています。

次のいずれの文でも、特殊機能レジスタ PORTA を使用できます。

```
PORTA = 0x34;       /* Assigns the value 0x34 to the port */
PORTAbits.AN0 = 1; /* Sets the AN0 pin high */
PORTAbits.RA0 = 1; /* Sets the RA0 pin high, same as above
                      statement */
```

レジスタを宣言する以外にも、プロセッサ固有ヘッダ ファイルではインライン アセンブリのマクロも定義されます。これらのマクロは、アプリケーションにおいて PICmicro MCU の特定の命令を C コードから実行できるようにするものです。これらの命令はインライン アセンブリ命令としてのインクルードもできますが、便宜上 C のマクロとして提供されています ( 表 2-8 参照 )。

プロセッサ固有ヘッダ ファイルを使用するには、使用するデバイスに適合したヘッダ ファイルをアプリケーションのソース コードにインクルードしてください ( 例 : PIC18C452 を使用する場合は `#include <p18c452.h>`)。プロセッサ固有ヘッダ ファイルは、`c:\mcc18\h` ディレクトリ (`c:\mcc18` はコンパイラのインストール先ディレクトリ ) にあります。または、`#include <p18cxxx.h>` と指定すると、コマンドライン オプション `-p` で選択したプロセッサに該当するプロセッサ固有ヘッダ ファイルが正しくインクルードされます。

**表 2-8: PICmicro® MCU 命令用 C マクロ**

| Instruction Macro(1) | Action |
|---|---|
| `Nop()` | Executes a no operation (`NOP`) |
| `ClrWdt()` | Clears the Watchdog Timer (`CLRWDT`) |
| `Sleep()` | Executes a `SLEEP` instruction |
| `Reset()` | Executes a device reset (`RESET`) |
| `Rlcf(`*`var`*`,` *`dest`*`,` *`access`*`)` (2,3) | Rotates *`var`* to the left through the carry bit |
| `Rlncf(`*`var`*`,` *`dest`*`,` *`access`*`)` (2,3) | Rotates *`var`* to the left without going through the carry bit |
| `Rrcf(`*`var`*`,` *`dest`*`,` *`access`*`)` (2,3) | Rotates *`var`* to the right through the carry bit |
| `Rrncf(`*`var`*`,` *`dest`*`,` *`access`*`)` (2,3) | Rotates *`var`* to the right without going through the carry bit |
| `Swapf(`*`var`*`,` *`dest`*`,` *`access`*`)` (2,3) | Swaps the upper and lower nibble of *`var`* |

**注 1:** 関数でこれらのマクロを使用すると、MPLAB® C18 コンパイラによるその関数の最適化に影響します。

**2:** *`var`* は 8 ビット長 ( すなわち `char`) である必要があり、スタックには配置できません。

**3:** *`dest`* が `0` の場合は結果が `WREG` に格納され、*`dest`* が `1` の場合は結果が *`var`* に格納されます。*`access`* が `0` の場合はアクセス バンクが選択され、`BSR` の値は無視されます。*`access`* が `1` の場合は、`BSR` の値に従ってバンクが選択されます。

## 2.11 プロセッサ固有レジスタ定義ファイル

プロセッサ固有レジスタ定義ファイルとは、ある特定のデバイスのすべての特殊機能レジスタを定義したアセンブリ ファイルです。プロセッサ固有レジスタ定義ファイルをコンパイルするとオブジェクト ファイルとなり、これをアプリケーションにリンクして使用します ( 例えば p18c452.asm をコンパイルすると p18c452.o となります )。このオブジェクト ファイルは p18*xxxx*.lib に含まれています ( 例えば p18c452.o は p18c452.lib に含まれています )。

プロセッサ固有レジスタ定義ファイルのソース コードは、c:\mcc18\src\traditional\proc と c:\mcc18\src\extended\proc の両方のディレクトリにあります。コンパイル後のオブジェクト コードは、c:\mcc18\lib ディレクトリ (c:\mcc18 はコンパイラのインストール先ディレクトリ ) にあります。

例えば、PORTA は PIC18C452 のプロセッサ固有レジスタ定義ファイルで次のように定義されています。

```
SFR_UNBANKED0 UDATA_ACS H'f80'
PORTA
PORTAbits RES 1 ; 0xf80
```

1 行目では、PORTA が存在するファイル レジスタ バンクとそのバンクの開始アドレスを指定しています。PORTA には PORTAbits と PORTA の 2 つのラベルがありますが、どちらも同じ番地 ( この場合は 0xf80) を参照しています。

# 第 3 章 ランタイム モデル

この章では、ランタイム モデルについて説明します。ランタイム モデルとは MPLAB C18 コンパイラの動作の前提となるもので、これには MPLAB C18 コンパイラが PIC18 PICmicro マイクロコントローラのリソースをどのように使用するかといった情報が含まれます。

## 3.1 メモリ モデル

MPLAB C18 は、スモール メモリ モデルとラージ メモリ モデルの両方でライブラリを完全にサポートしています ( 表 3-1 参照 )。コマンドライン オプションに -ms を指定するとスモール メモリ モデルが選択され、-ml オプションを指定するとラージ メモリ モデルが選択されます。どちらのオプションも指定しない場合は、デフォルトでスモール メモリ モデルが選択されます。

**表 3-1: メモリ モデルのまとめ**

| Memory Model | Command-line Switch | Default ROM Range Qualifier | Size of Pointers to Program Space |
|---|---|---|---|
| small | -ms | near | 16 bits |
| large | -ml | far | 24 bits |

スモール モデルとラージ モデルの違いは、プログラム メモリへのポインタのサイズにあります。スモール メモリ モデルでは、プログラム メモリを参照する関数ポインタとデータ ポインタのいずれも 16 ビットを使用します。このためスモール モデルでは、ポインタのアドレス指定は、プログラム メモリの最初の 64K までです。ラージ メモリ モデルでは、24 ビットを使用します。64K を超えるプログラム メモリを使用するアプリケーションでは、ラージ メモリ モデルを使用する必要があります。

プログラム領域へのポインタを宣言する際に near または far 修飾子を使用すると、メモリ モデルの設定を個別にオーバーライドできます。near メモリへのポインタはスモール メモリ モデルと同様に 16 ビットを使用し、far メモリへのポインタはラージ メモリ モデルと同様に 24 ビットを使用します。

次の例で作成されるプログラム メモリへのポインタは、スモール メモリ モデルを使用している場合でも 64K 以上のプログラム メモリ領域をアドレス指定できます。[1]

```
far rom *pgm_ptr;
```

次の例で作成される関数ポインタは、スモール メモリ モデルを使用している場合でも 64K 以上のプログラム メモリ領域をアドレス指定できます。[2]

```
far rom void (*fp) (void);
```

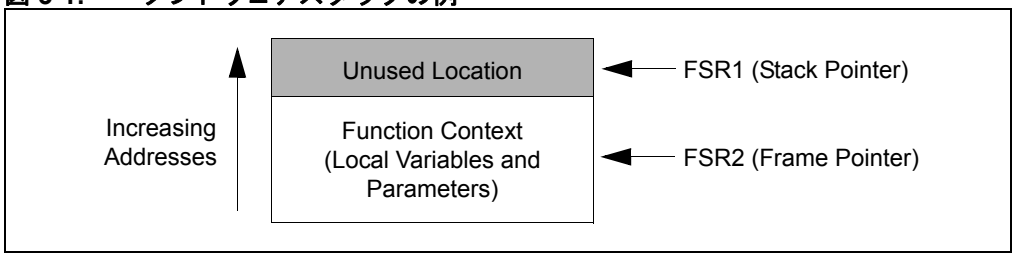
1 つのプロジェクト内で異なるメモリ モデルを使用するファイルが混在している場合、プログラム メモリへのグローバル ポインタは near または far 修飾子を付けて明示的に宣言し、すべてのモジュールで正しくアクセスできるようにする必要があります。MPLAB C18 に同梱されているコンパイル済みのライブラリは、スモール メモリ モデルにもラージ メモリ モデルにも使用できます。

1. スモール メモリ モデル プログラムで far データ ポインタを使用した後は、TBLPTRU バイトをユーザーがクリアする必要があります。MPLAB C18 はこのバイトをクリアしません。
2. スモール メモリ モデル プログラムで far 関数ポインタを使用した後は、PCLATU バイトをユーザーがクリアする必要があります。MPLAB C18 はこのバイトをクリアしません。

## 3.2 呼び出し規則

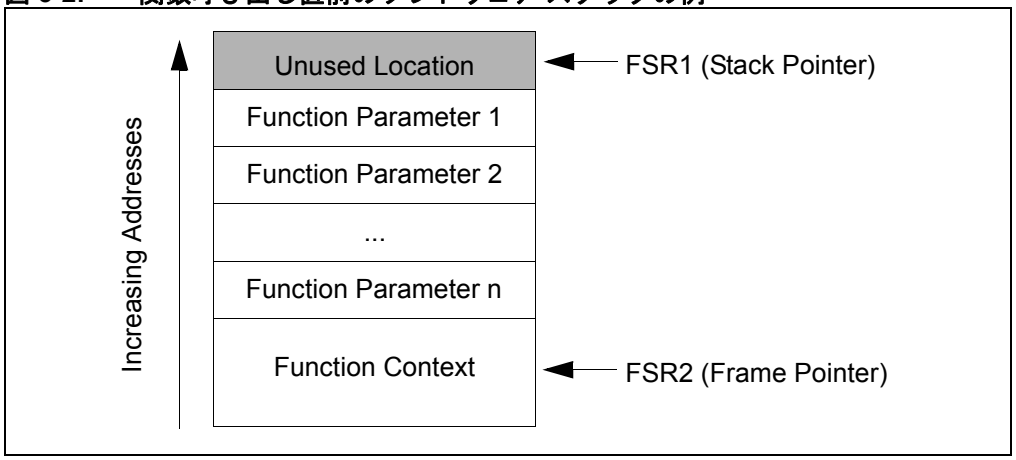
MPLAB C18 のソフトウェア スタックは上方成長スタックのデータ構造をしており、コンパイラが auto ストレージ クラスの関数の引き数とローカル変数を配置する場所です。ソフトウェア スタックは、PICmicro マイクロコントローラが関数呼び出しの戻り番地を格納するハードウェア スタックとは区別されます。図 3-1 に、ソフトウェア スタックの例を示します。

**図 3-1: ソフトウェアスタックの例**




スタック ポインタ (FSR1) は常に、次に利用できるスタック番地を指し示します。MPLAB C18 では、FSR2 をフレーム ポインタとして使用し、ローカル変数およびパラメータへの高速アクセスを可能にしています。関数を実行する際は、その関数のスタック ベースの引き数が右から左の順にスタックにプッシュされ、関数が呼び出されます。関数に入った時点では、関数の最も左の引き数がソフトウェア スタックの一番上に配置されます。図 3-2 に、関数呼び出しの直前のソフトウェア スタックの例を示します。

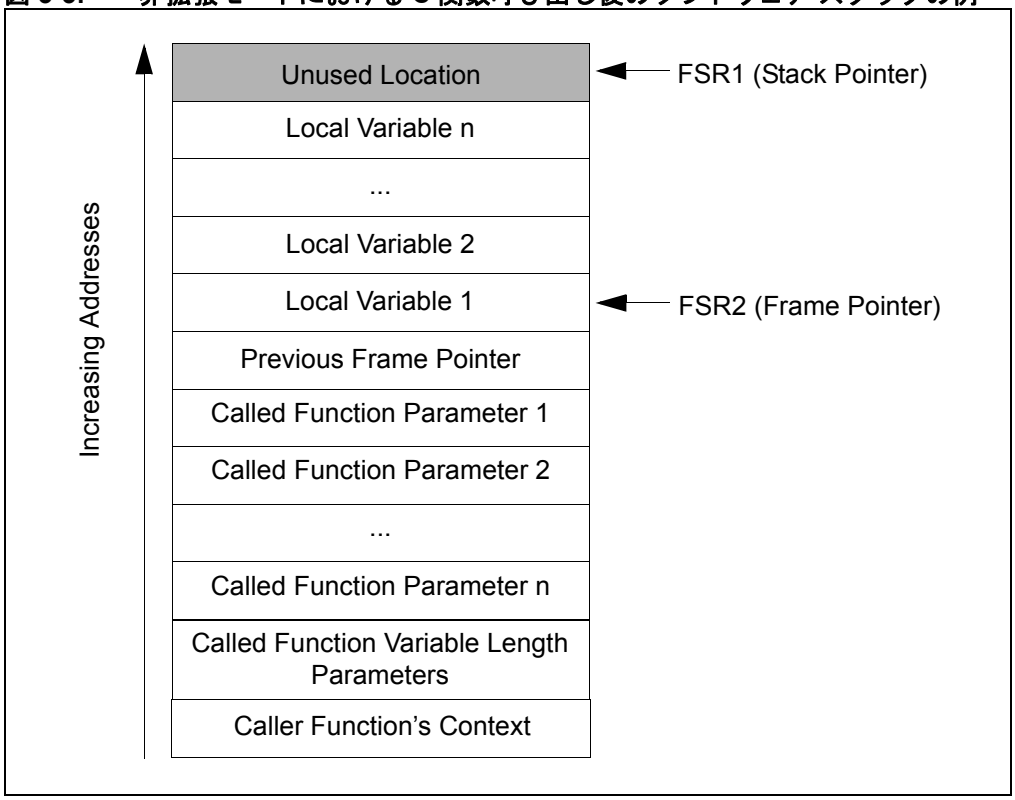
**図 3-2: 関数呼び出し直前のソフトウェア スタックの例**




### 3.2.1 非拡張モード時の規則

非拡張モードの場合、フレーム ポインタはスタック ベースの引き数とスタック ベースのローカル変数の境界となるスタック番地を指し示します。スタック ベースの引き数はフレーム ポインタから負のオフセット位置に格納され、スタック ベースのローカル変数はフレーム ポインタから正のオフセット位置に格納されます。C 関数に入った直後に、呼び出された関数は FSR2 の値をスタックにプッシュし、FSR1 の値を FSR2 にコピーします。これにより、呼び出し元の関数のコンテクストが保存され、現在の関数のフレーム ポインタが初期化されます。次にこの関数のスタック ベースのローカル変数の合計サイズがスタック ポインタに加算され、これらの変数にスタック領域が割り当てられます。スタック ベースのローカル変数とスタック ベースの引き数への参照は、フレーム ポインタからのオフセットを使用して解決されます。図 3-3 に、非拡張モードにおける C 関数呼び出し後のソフトウェア スタックの例を示します。

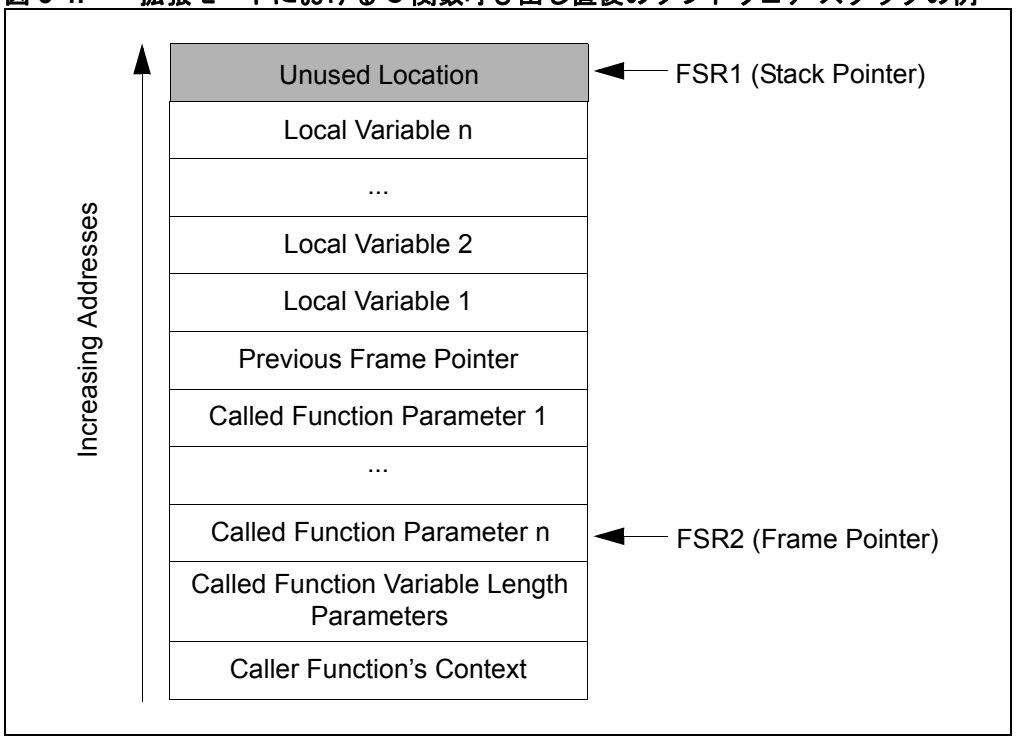
**図 3-3:　非拡張モードにおける C 関数呼び出し後のソフトウェア スタックの例**




### 3.2.2　拡張モード時の規則

拡張モードの場合、フレーム ポインタは関数の最も右の名前付きパラメータの下位バイトを指し示します。ローカル変数とパラメータはどちらもフレーム ポインタから正の ( 負でない ) オフセット位置に格納されるため、コンパイラはリテラル オフセットによるインデックス アドレス指定によってこれらにアクセスできます。呼び出された関数に入ると、FSR2 の値がスタックに保存され、FSR1 の値が FSR2 にコピーされます。そして FSR2 から名前付きパラメータのサイズと保存したフレーム ポインタのサイズの和が差し引かれます。これによって呼び出し元関数のフレーム ポインタが保存され、現在の関数のフレーム ポインタが初期化されます。次に、関数のローカル変数のサイズの合計が FSR1 に加算され、これらローカル変数にスタック領域が割り当てられます。図 3-4 に、拡張モードにおける C 関数呼び出し後ソフトウェア スタックの例を示します。

**図 3-4: 拡張モードにおける C 関数呼び出し直後のソフトウェア スタックの例**




### 3.2.3 戻り値

戻り値が格納される場所は、戻り値のサイズによって異なります。表 3-2 に、サイズ別の戻り値の場所を示します。

**表 3-2: 戻り値**

| Return Value Size | Return Value Location |
|---|---|
| 8 bits | WREG |
| 16 bits | PRODH:PRODL |
| 24 bits | [Non-Extended mode] (AARGB2+2):(AARGB2+1):AARGB2 (1)<br>[Extended mode] __RETVAL2:__RETVAL1:__RETVAL0 (1) |
| 32 bits | [Non-Extended mode] (AARGB3+3):(AARGB3+2):(AARGB3+1):AARGB3 (1)<br>[Extended mode] __RETVAL3:__RETVAL2:__RETVAL1:__RETVAL0 (1) |
| > 32 bits | On the stack, and FSR0 points to the return value |

**注 1:** この場所はコンパイラ専用に予約されています。

### 3.2.4 ソフトウェア スタックの管理

スタックのサイズ決めと配置はリンカ スクリプトの STACK ディレクティブで実行します。STACK ディレクティブには、割り当てられたスタックのサイズを制御する SIZE と位置を制御する RAM という 2 つの引き数があります。次に示す例では 128 バイトのスタックが割り当てられ、そのスタックは gpr3 という名前のメモリ領域に配置されます。

```
STACK SIZE=0x80 RAM=gpr3
```

MPLAB C18 は、256 バイトを超えるスタック サイズをサポートしています。デフォルトのリンカ スクリプトでは、メモリ バンクごとに 1 つのメモリ領域が割り当てられます。スタックセクションはメモリ領域の境界をまたぐことはできないため、256 バイトを超えるスタックを割り当てるには 2 つ以上のメモリ領域を結合する必要があります。例えば、PIC18C452 のデフォルトのリンカ スクリプトには次の定義が含まれます。

```
DATABANK NAME=gpr4 START=0x400 END=0x4ff
DATABANK NAME=gpr5 START=0x500 END=0x5ff
...
STACK SIZE=0x100 RAM=gpr5
```

バンク 4 とバンク 5 に 512 バイトのスタックを割り当てるには、上記の定義を次のように書き換えます。

```
DATABANK NAME=stackregion START=0x400 END=0x5ff PROTECTED
STACK SIZE=0x200 RAM=stackregion
```

256 バイトを超えるスタックを使用する場合は、コンパイラのコマンドライン オプションに -ls を指定する必要があります。ラージ スタックを使用すると、プッシュ / ポップの際にフレーム ポインタの下位バイトだけでなく両方のバイト (FSR2L および FSR2H) を増分または減分する必要があるため、パフォーマンスはやや低下します。

アプリケーションで必要なソフトウェア スタックのサイズは、プログラムの複雑度により異なります。関数呼び出しを入れ子状にする場合は、呼び出し元関数の auto 属性のパラメータと変数をすべてスタックに格納する必要があります。このため、呼び出しツリーのすべての関数に対応できる十分なサイズが必要です。

MPLAB C18 では、パラメータとローカル変数をソフトウェア スタックに配置することも、グローバル メモリから直接利用することもできます。static キーワードを指定すると、ローカル変数または関数パラメータはソフトウェア スタックではなくグローバル メモリに配置されます。[1] 一般に、スタック ベースのローカル変数と関数パラメータにアクセスするには、static 属性のローカル変数や関数パラメータにアクセスするよりも多くのコードが必要です (**2.3.2 項「static 属性の関数の引き数」**参照 )。スタック ベースの変数を使用する関数は、再入や再帰が可能なため、柔軟性に優れています。

### 3.2.5 C とアセンブリの混在

#### 3.2.5.1 アセンブリからの C 関数の呼び出し

アセンブリから C 関数を呼び出す際は、次の点に注意してください。

- static と定義されたものを除き、C 関数にはグローバルな性質が必要です。
- C 関数名は、アセンブリ ファイル内で extern 属性のシンボルとして宣言されている必要があります。
- 関数呼び出しには、CALL または RCALL を使用する必要があります。

1. static パラメータは、コンパイラが非拡張モードで動作する場合のみ有効です。(**1.2.5 項「モードの選択」**参照 )。

#### 3.2.5.1.1 auto パラメータ

auto パラメータは、右から左の順にソフトウェア スタックにプッシュされます。マルチバイト データの場合は、下位バイトから先にソフトウェア スタックにプッシュされます。

**例 3-1:**

次のように C 関数のプロトタイプが宣言されているとします。

```
char add (auto char x, auto char y);
```

この場合、x = 0x61 および y = 0x65 という値で関数 add を呼び出すには、y の値をソフトウェア スタックにプッシュした後で x の値をプッシュする必要があります。戻り値は 8 ビットなので、WREG に格納されます ( 表 3-2 参照 )。
よって、次のようになります。

```
            EXTERN add  ; defined in C module
...
MOVLW 0x65
MOVWF POSTINC1 ; y = 0x65 pushed onto stack
MOVLW 0x61
MOVWF POSTINC1 ; x = 0x61 pushed onto stack
CALL  add
MOVWF result   ; result is returned in WREG
...
```

**例 3-2:**

次のように C 関数のプロトタイプが宣言されているとします。

```
int sub (auto int x, auto int y);
```

この場合、x = 0x7861 および y = 0x1265 という値で関数 sub を呼び出すには、y の値をソフトウェア スタックにプッシュした後で x の値をプッシュする必要があります。戻り値は 16 ビットなので、PRODH:PRODL に格納されます ( 表 3-2 参照 )。
よって、次のようになります。

```
              EXTERN sub  ; defined in C module
...
MOVLW 0x65
MOVWF POSTINC1
MOVLW 0x12
MOVWF POSTINC1        ; y = 0x1265 pushed onto stack
MOVLW 0x61
MOVWF POSTINC1
MOVLW 0x78
MOVWF POSTINC1        ; x = 0x7861 pushed onto stack
CALL  sub
MOVFF PRODL, result
MOVFF PRODH, result+1 ; result is returned in PRODH:PRODL
...
```

#### 3.2.5.1.2 static パラメータ

static パラメータはグローバルに割り当てられるため、直接アクセスが可能です。static パラメータは、コンパイラが非拡張モードで動作時のみ有効です (**1.2.5 項「モードの選択」**参照 )。static パラメータの命名規則は、*__function_name:n* (*function_name* は関数名、*n* は 0 から起算したパラメータの位置 ) です。例えば、次のように C 関数のプロトタイプが宣言されているとします。

```
char add (static char x, static char y);
```

この場合、y の値は __add:1 を使用してアクセスし、x の値は __add:0 を使用してアクセスします。

**注 :** ':' は MPASM アセンブラのラベルで有効な文字でないため、アセンブリ関数では static パラメータへのアクセスはサポートされていません。

### 3.2.5.2 C からのアセンブリ関数の呼び出し

C からアセンブリ関数を呼び出す際は、次の点に注意します。

- 関数ラベルを ASM モジュールで global として宣言する必要があります。
- 関数を C モジュールで extern として宣言する必要があります。
- ASM モジュールで大文字と小文字を区別しない場合は、C モジュールで関数をすべて大文字で宣言する必要があります。
- 関数は、MPLAB C18 コンパイラのランタイム モデルを遵守している必要があります ( 戻り値は表 3-2 に示した位置に格納するなど )。
- C から関数を呼び出す際は、標準の C 関数記法を使用します。

**例 3-3:**

次の関数がアセンブリで記述されているとします。

```
                UDATA_ACS
delay_temp      RES     1

                CODE
asm_delay
          SETF   delay_temp
not_done
          DECF   delay_temp
          BNZ    not_done
done
          RETURN

                GLOBAL asm_delay  ; export so linker can see it
                END
```

この場合、C ソース ファイルから関数 asm_delay を呼び出すには、このアセンブリ関数の外部プロトタイプを追加し、関数を標準の C 関数記法で呼び出す必要があります。

```
/* asm_delay is found in an assembly file */
extern void asm_delay (void);

void main (void)
{
  asm_delay ();
}
```

**例 3-4:**

次の関数がアセンブリで記述されているとします。

```
                INCLUDE "p18c452.inc"

                CODE
asm_timed_delay
not_done
          ; Figure 3-2 is what the stack looks like upon
          ; entry to this function.
          ;
          ; 'time' is passed on the stack and must be >= 0
          MOVLW  0xff
          DECF   PLUSW1, 0x1, 0x0
          BNZ    not_done
done
          RETURN
               ; export so linker can see it
               GLOBAL asm_timed_delay
               END
```

この場合、C ソース ファイルから関数 asm_timed_delay を呼び出すには、このアセンブリ関数の外部プロトタイプを追加し、関数を標準の C 関数記法で呼び出す必要があります。

```
/* asm_timed_delay is found in an assembly file */
extern void asm_timed_delay (unsigned char);

void main (void)
{
  asm_timed_delay (0x80);
}
```

#### 3.2.5.3 アセンブリでの C 変数の使用

アセンブリで C 変数を使用する際は、次の点に注意してください。

- C ソース ファイルにおける C 変数のスコープは、グローバルである必要があります。
- C 変数名は、アセンブリ ファイル内で extern 属性のシンボルとして宣言されている必要があります。

**例 3-5:**

次のコードが C で記述されているとします。

```
unsigned int c_variable;

void main (void)
{
  ...
}
```

この場合、アセンブリから変数 c_variable を変更するには、アセンブリ ソース ファイルにこの変数の外部宣言を追加する必要があります。

```
     EXTERN c_variable  ; defined in C module
MYCODE CODE
asm_function
     MOVLW 0xff
     ; put 0xffff in the C declared variable
     MOVWF c_variable
     MOVWF c_variable+1
done
     RETURN

     ; export so linker can see it
     GLOBAL asm_function
     END
```

#### 3.2.5.4 C でのアセンブリ変数の使用

C でアセンブリ変数を使用する際は、次の点に注意してください。

- 変数を ASM モジュールで global として宣言する必要があります。
- 変数を C モジュールで extern として宣言する必要があります。
- ASM モジュールで大文字と小文字を区別しない場合は、C モジュールで変数をすべて大文字で宣言する必要があります。

**例 3-6:**

次のコードがアセンブリで記述されているとします。

```
MYDATA UDATA
asm_variable RES   2 ; 2 byte variable

     ; export so linker can see it
     GLOBAL asm_variable
     END
```

この場合、C ソース ファイルから変数 asm_variable を変更するには、C ソース ファイルにこの変数の外部宣言を追加する必要があります。変数は、通常の C 変数と同じように使用できます。

```
extern unsigned int asm_variable;

void change_asm_variable (void)
{
  asm_variable = 0x1234;
}
```

## 3.3 スタートアップ コード

### 3.3.1 デフォルトの動作

MPLAB C18 のスタートアップは、reset ベクタ (0 番地 ) から開始します。reset ベクタは、FSR1 と FSR2 を初期化する関数にジャンプしてソフトウェア スタックを参照し、更にオプションとしてプログラム メモリから idata セクション ( データ メモリ初期化データ ) を初期化する関数を呼び出し、アプリケーションの main() 関数の呼び出しをループ実行します。

スタートアップ コードで idata セクションを初期化するかどうかは、アプリケーションにどのスタートアップ コード モジュールをリンクするかで決定されます。c018i.o と c018i_e.o の各モジュールは初期化を実行しますが、c018.o と c018_e.o のモジュールは初期化を実行しません。MPLAB C18 によって提供されるデフォルトのリンカ スクリプトは、非拡張モードを使用する場合は c018i.o、拡張モードを使用する場合は c018i_e.o モジュールとリンクします。

ANSI 規格では、静的なストレージ期間を持つオブジェクトで明示的に初期化されないものは、すべてゼロに設定するよう規定されています。スタートアップ コード モジュールの c018.o/c018_e.o と c018i.o/c018i_e.o は、いずれもこの条件を満たしていません。上記条件を満たした第 3 のスタートアップ モジュールが、c018iz.o および c018iz_e.o として提供されています。アプリケーションにこのスタートアップ コード モジュールをリンクすると、idata セクションの初期化に加え、静的なストレージ期間を持ち明示的に初期化されないすべてのオブジェクトがゼロに設定されます。

データ メモリを初期化するには、まず MPLINK リンカが初期化されたデータ メモリのコピーをプログラム メモリに作成し、これをスタートアップ コードがデータ メモリにコピーします。MPLINK リンカは、プログラム メモリ イメージのコピー先を記述した .cinit セクションを作成します。表 3-3 に、.cinit セクションのフォーマットを示します。

**表 3-3: .cinit のフォーマット**

| Field | Description | Size |
|---|---|---|
| num_init | Number of sections | 16 bits |
| from_addr_0 | Program memory start address of section 0 | 32 bits |
| to_addr_0 | Data memory start address of section 0 | 32 bits |
| size_0 | Number of data memory bytes to initialize for section 0 | 32 bits |
| ... | ... | ... |
| from_addr_*n*(1) | Program memory start address of section *n*(1) | 32 bits |
| to_addr_*n*(1) | Data memory start address of section *n*(1) | 32 bits |
| size_*n*(1) | Number of data memory bytes to initialize for section *n*(1) | 32 bits |

**注 1:** *n* = num_init - 1

スタートアップ コードはまずスタックを準備し、( オプションで ) 初期化するデータをコピーしてから、C プログラムの main() 関数を呼び出します。main() には引き数は渡されません。MPLAB C18 は、次に示すようなループ呼び出しによって main() に制御を渡します。

```
loop:
  // Call the user's main routine
  main();
goto loop;
```

### 3.3.2 カスタマイズ

デバイスの reset 後、コンパイラが生成した他のコードが実行される前にアプリケーション固有のコードを実行する場合は、該当するスタートアップ ファイルを編集して _entry() 関数の先頭に目的のコードを追加してください。

非拡張モード使用時にスタートアップ ファイルをカスタマイズする方法は、次のとおりです。

1. c:\mcc18\src\traditional\startup ディレクトリに移動する (c:\mcc18 はコンパイラのインストール先ディレクトリ )。
2. c018.c、c018i.c、c018iz.c のいずれかを編集し、カスタマイズしたスタートアップ コードを追加する。
3. 編集したスタートアップ ファイルをコンパイルして c018.o、c018i.o、c018iz.o のいずれかを生成する。
4. スタートアップ モジュールを c:\mcc18\lib にコピーする (c:\mcc18 はコンパイラのインストール先ディレクトリ )。

拡張モード使用時にスタートアップ ファイルをカスタマイズする方法は、次のとおりです。

1. c:\mcc18\src\extended\startup ディレクトリに移動する (c:\mcc18 はコンパイラのインストール先ディレクトリ )。
2. c018_e.c 、c018i_e.c 、c018iz_e.c のいずれかを編集し、カスタマイズしたスタートアップ コードを追加する。
3. 編集したスタートアップ ファイルをコンパイルして c018_e.o、c018i_e.o、c018iz_e.o のいずれかを生成する。
4. スタートアップ モジュールを c:\mcc18\lib にコピーする (c:\mcc18 はコンパイラのインストール先ディレクトリ )。

## 3.4 コンパイラ管理リソース

PIC18 PICmicro マイクロコントローラには、MPLAB C18 コンパイラのみが使用する特殊機能レジスタとデータ セクションがあり、これらは汎用のユーザー コードには使用できません。表 3-4 に、これらのリソースとコンパイラによる主な用途を示します。ISR の実行中は、コンパイラ管理リソースはすべて自動的に待避します。

**表 3-4: コンパイラ管理リソース**

| Compiler-Managed Resource | Primary Use(s) |
|---|---|
| PC | Execution control |
| WREG | Intermediate calculations |
| STATUS | Calculation results |
| BSR | Bank selection |
| PROD | Multiplication results, return values, intermediate calculations |
| section .tmpdata(1) | Intermediate calculations |
| FSR0 | Pointers to RAM |
| FSR1 | Stack Pointer |
| FSR2 | Frame Pointer |
| TBLPTR(2) | Accessing values in program memory |
| TABLAT | Accessing values in program memory |
| PCLATH | Function pointer invocation |
| PCLATU(3) | Function pointer invocation |
| section MATH_DATA | Arguments, return values and temporary locations for math library functions |

**注 1:** コンパイラが ISR のコンテクストを保存する際、#pragma tmpdata ディレクティブによって作成された一時データ セクションは保存されません。コンパイラが保存するのは、デフォルトの一時データ セクションである .tmpdata のみです。**2.9.3.3 項「複数の高優先度割り込み」**を参照してください。

**2:** スモール メモリ モデルでは、TBLPTRU は、割り込み関数本体で使用されていれば自動的に保存されます ( 関数呼び出しでのみ使用されている場合は保存されません )。ラージ メモリ モデルでは、TBLPTRU が割り込み関数本体または関数呼び出しのいずれに使用されていても自動的に保存されます。

**3:** スモール メモリ モデルでは、PCLATU は、割り込み関数本体で使用されていれば自動的に保存されます ( 関数呼び出しでのみ使用されている場合は保存されません )。ラージ メモリ モデルでは、PCLATU が割り込み関数本体または関数呼び出しのいずれに使用されていても自動的に保存されます。

# 第 4 章 最適化

MPLAB C18 コンパイラは、最適化コンパイラです。このコンパイラでは、主にコードサイズの縮小を目的とした最適化が実行されます。デフォルトでは MPLAB C18 コンパイラの最適化機能はすべて有効になっていますが、コマンドライン オプション -O- を指定すると、これらをすべて無効にできます。また、MPLAB C18 コンパイラの最適化機能は個別の有効化 / 無効化もできます。表 4-1 に、MPLAB C18 コンパイラの最適化機能、および各機能を有効 / 無効にするコマンドライン オプション、デバッグへの影響の有無、各機能に関する参照先セクションを示します。

**注 :** インライン アセンブリ コードを含む関数は最適化されません。

**表 4-1: MPLAB® C18 の最適化機能**

| Optimization | To Enable | To Disable | Affects Debugging | Section |
|---|---|---|---|---|
| Duplicate String Merging | -Om+ | -Om- | | 4.1 |
| Branches | -Ob+ | -Ob- | | 4.2 |
| Banking | -On+ | -On- | | 4.3 |
| WREG Content Tracking | -Ow+ | -Ow- | | 4.4 |
| Code Straightening | -Os+ | -Os- | | 4.5 |
| Tail Merging | -Ot+ | -Ot- | ✓ | 4.6 |
| Unreachable Code Removal | -Ou+ | -Ou- | ✓ | 4.7 |
| Copy Propagation | -Op+ | -Op- | ✓ | 4.8 |
| Redundant Store Removal | -Or+ | -Or- | ✓ | 4.9 |
| Dead Code Removal | -Od+ | -Od- | ✓ | 4.10 |
| Procedural Abstraction | -Opa+ | -Opa- | ✓ | 4.11 |

## 4.1 重複文字列のマージ

-Om+ / -Om-

重複文字列のマージ機能を有効にすると、2 つ以上の同じリテラル文字列が 1 つの文字列テーブル エントリにまとめられ、プログラム メモリにはデータが 1 つだけ格納されます。次に示すコード例で重複文字列のマージを有効 (-Om+) にすると、文字列「foo」のデータは出力オブジェクト ファイルに 1 つだけ格納され、a と b はどちらもこのデータを参照します。

```
const rom char *a = "foo";
const rom char *b = "foo";
```

コマンドライン オプションに -Om- を指定すると、重複文字列があってもマージされません。

重複文字列をマージしても、ソース コードのデバッグには影響ありません。

## 4.2 分岐最適化

-Ob+ / -Ob-

コマンドライン オプション -Ob+ を指定すると、MPLAB C18 コンパイラは次の分岐最適化を実行します。

1. ある分岐 ( 条件分岐または無条件分岐 ) の分岐先が無条件分岐の場合、前者の分岐先を後者の分岐先に変更できます。
2. RETURN、ADDULNK、SUBULNK 命令への無条件分岐は、RETURN、ADDULNK、SUBULNK 命令にそれぞれ置き換え可能です。
3. 直後の 命令を分岐先としている分岐 ( 条件分岐または無条件分岐 ) は、削除できます。
4. 条件分岐への条件分岐がある場合、これら 2 つの分岐条件が同じなら、前者の分岐先を後者の分岐先に変更できます。
5. 条件分岐の直後に無条件分岐があり、両者の分岐先が同じなら、条件分岐は削除できます ( すなわち無条件分岐だけで十分です )。

コマンドライン オプションに -Ob- を指定すると、分岐は最適化されません。

分岐最適化だけではプログラム サイズが縮小しない場合でも、到達不能コードが顕在化することがあり、これらは 到達不能コード除去 (**4.7 項「到達不能コード除去」**参照 ) で削除できます。分岐を最適化しても、ソース コードのデバッグには影響ありません。

## 4.3 バンク切り替え最適化

-On+ / -On-

バンク切り替え最適化は、バンク選択レジスタ (BSR) に既に適切な値が入っていると判断される場合に、MOVLB 命令を除去します。次に、C ソース コードの例を示します。

```
unsigned char a, b;
a = 5;
b = 5;
```

MPLAB C18 では、バンク切り替え最適化を無効 (-On-) にしてコンパイルすると、変数の代入前に毎回 BSR がロードされます。

```
0x000000 MOVLB a
0x000002 MOVLW 0x5
0x000004 MOVWF a,0x1
0x000006 MOVLB b
0x000008 MOVWF b,0x1
```

上記のコードをバンク切り替え最適化を有効 (-On+) にしてコンパイルすると、MPLAB C18 は BSR の値が変化していないと判断して、2 番目の MOVLB 命令を削除します。

```
0x000000 MOVLB a
0x000002 MOVLW 0x5
0x000004 MOVWF a,0x1
0x000006 MOVWF b,0x1
```

バンク切り替えを最適化しても、ソース コードのデバッグには影響ありません。

## 4.4 WREG 内容追跡

-Ow+ / -Ow-

WREG 内容追跡は、ワーキング レジスタ (WREG) に既に適切な値が入っていると判断される場合に、MOVLW 命令を除去します。次に、C ソース コードの例を示します。

```
unsigned char a, b;
a = 5;
b = 5;
```

MPLAB C18 では、WREG 内容追跡を無効 (-Ow-) にしてコンパイルすると、変数の代入前に毎回 WREG に値 5 がロードされます。

```
0x000000 MOVLW 0x5
0x000002 MOVWF a,0x1
0x000004 MOVLW 0x5
0x000006 MOVWF b,0x1
```

上記のコードを WREG 内容追跡を有効 (-Ow+) にしてコンパイルすると、MPLAB C18 は WREG にはこの時点で既に 5 がロードされていると判断して、2 番目の MOVLW 命令を削除します。

```
0x000000 MOVLW 0x5
0x000002 MOVWF a,0x1
0x000004 MOVWF b,0x1
```

WREG の内容を追跡しても、ソース コードのデバッグには影響ありません。

## 4.5 コード並べ替え

-Os+ / -Os-

コード並べ替えでは、コード シーケンスを実行順に並べ替えます。これによって分岐命令を移動または削除するため、コードサイズが縮小し、コードの効率化につながります。次に、C ソース コードの例を示します。

```
first:
  sub1();
  goto second;
third:
  sub3();
  goto fourth;
second:
  sub2();
  goto third;
fourth:
  sub4();
```

この例では、関数が数字の小さい順、すなわち sub1、sub2、sub3、sub4 の順に呼び出されます。コード 並べ替えを無効 (-Os-) にすると、元のコードの順番をそのまま反映したアセンブリ コードが生成されます。

```
0x000000 first  CALL sub1,0x0
0x000002
0x000004        BRA second
0x000006 third  CALL sub3,0x0
0x000008
0x00000a        BRA fourth
0x00000c second CALL sub2,0x0
0x00000e
0x000010        BRA third
0x000012 fourth CALL sub4,0x0
0x000014
```

コード並べ替えを有効 (-Os+) にすると、コードが並べ替えられ、分岐命令が除去されます。

```
0x000000 first  CALL sub1,0x0
0x000002
0x000004 second CALL sub2,0x0
0x000006
0x000008 third  CALL sub3,0x0
0x00000a
0x00000c fourth CALL sub4,0x0
0x00000e
```

コードを並べ替えても、ソース コードのデバッグには影響ありません。

## 4.6 テール マージ

-Ot+ / -Ot-

テール マージは、同一の命令シーケンスが複数ある場合、1 つのシーケンスにまとめます。次に、C ソース コードの例を示します。

```
if ( user_value )
  PORTB = 0x55;
else
  PORTB = 0x80
```

テール マージを無効 (-Ot-) にしてコンパイルすると、if 文が真の場合と偽の場合の MOVWF PORTB,0x0 が両方とも生成されます。

```
0x000000 MOVF user_value,0x0,0x0
0x000002 BZ 0xa
0x000004 MOVLW 0x55
0x000006 MOVWF PORTB,0x0
0x000008 BRA 0xe
0x00000a MOVLW 0x80
0x00000c MOVWF PORTB,0x0
0x00000e RETURN 0x0
```

一方、テール マージを有効 (-Ot+) にしてコンパイルすると、MOVWF PORTB,0x0 は 1 つだけ生成され、if と else の両方で使用されます。

```
0x000000 MOVF user_value,0x0,0x0
0x000002 BZ 0x8
0x000004 MOVLW 0x55
0x000006 BRA 0xa
0x000008 MOVLW 0x80
0x00000a MOVWF PORTB,0x0
0x00000c RETURN 0x0
```

この最適化機能を有効にしてコンパイルしたソース コードをデバッグすると、複数のソース行がアセンブリ コード上で 1 つのシーケンスを共有するため、どのソース行が実行されているのかをデバッガが判断できず、不適切なソース行がハイライトされる場合があります。

## 4.7 到達不能コード除去

-Ou+ / -Ou-

到達不能コード除去は、通常のプログラム フローでは決して実行されることがないと考えられるコードを除去します。次に、C ソースコードの例を示します。

```
if (1)
{
  x = 5;
}
else
{
  x = 6;
}
```

このコードでは、else の部分に到達することがないのは明白です。到達不能コード除去を無効 (-Ou-) にすると、6 を x に移動する命令と、この命令を迂回する分岐命令を含んだアセンブリ コードが生成されます。

```
0x000000 MOVLB x
0x000002 MOVLW 0x5
0x000004 BRA 0xa
0x000006 MOVLB x
0x000008 MOVLW 0x6
0x00000a MOVWF x,0x1
```

到達不能コード除去を有効 (-Ou+) にすると、else 部分の命令を含まないアセンブリ コードが生成されます。

```
0x000000 MOVLB x
0x000002 MOVLW 0x5
0x000004 MOVWF x,0x1
```

到達不能コードの最適化が実行されると、C ソース コードで一部の行にブレークポイントを設定できなくなる可能性があります。

## 4.8 コピー伝播

-Op+ / -Op-

コピー伝播とは、変数 *x* と *y* の間で *x* ← *y* の代入が実行された場合、以後の命令で *x* と *y* の値がいずれも変更されていなければ、*x* の代わりに *y* を使用するといった変形をいいます。この最適化だけでは命令数の削減にはなりませんが、デッド コード除去 (**4.10 項「デッド コード除去」**参照 ) が有効になります。次に、C ソース コードの例を示します。

```
char c;
void foo (char a)
{
  char b;
  b = a;
  c = b;
}
```

コピー伝播を無効 (-Op-) にすると、元のコードをそのまま反映したアセンブリ コードが生成されます。

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF b,c
0x000006
0x000008        RETURN 0x0
```

コピー伝播を有効 (-Op+) にすると、2 番目の命令で b を c に移動する代わりに a を c に移動します。

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

この状態でデッド コード除去を実行すると、a を b に代入する部分が不要となり、除去されます (**4.10 項「デッド コード除去」**参照 )。

コピーを伝播すると、ソース コードのデバッグに影響する場合があります。

## 4.9 冗長ストア除去

-Or+ / -Or-

1 つの命令シーケンスで *x* ← *y* という代入が複数回出現する場合、その間のコードで *x* と *y* の値のいずれも変更されていなければ、2 回目の代入は除去できます。これは、共通部分式除去の特殊なケースです。次に、C ソース コードの例を示します。

```
char c;
void foo (char a)
{
  c = a;
  c = a;
}
```

冗長ストア除去を無効 (-Or-) にすると、元のコードをそのまま反映したアセンブリ コードが生成されます。

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

冗長ストア除去を有効 (-Or+) にすると、2 回目の c = a の代入が不要になります。

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        RETURN 0x0
```

冗長ストア除去が実行されると、C ソース コードで一部の行にブレークポイントを設定できなくなる可能性があります。

## 4.10 デッド コード除去

-Od+ / -Od-

関数で計算される値のうち、関数の出口に至るいずれのパスでも使用されない値は、デッドと見なされます。計算するのがデッド値のみの命令も、デッドと見なされます。関数のスコープの外部で可視の番地に保存された値は、その値が使用されているかどうか判断できないため、使用されている ( すなわちデッドではない ) と見なされます。次に、**4.8 項「コピー伝播」**の例と同じコードを示します。

```
char c;
void foo (char a)
{
    char b;
    b = a;
    c = b;
}
```

コピー伝播を有効 (-Op+)、デッド コード除去を無効 (-Od-) にすると、**4.8 項「コピー伝播」**で示したアセンブリ コードが生成されます。

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

コピー伝播を有効 (-Op+)、デッド コード除去を有効 (-Od+) にすると、2 番目の命令では、b を c に移動する代わりに a が c に移動されます。これにより b への代入は、デッドとなって除去可能になります。

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        RETURN 0x0
```

デッド コードの最適化が実行されると、C ソース コードで一部の行にブレークポイントを設定できない場合があります。

## 4.11 プロシージャ抽出

Full Version Only

-Opa+ / -Opa-

多くのコンパイラと同様、MPLAB C18 でも 1 つのオブジェクト ファイル内に同じコード シーケンスが何度も出現することが頻繁にあります。プロシージャ抽出では、繰り返されるコードがある場合にそのコードを含むプロシージャを作成し、繰り返されるコードをプロシージャへの呼び出しに置き換えることによって、コード サイズを縮小します。プロシージャ抽出は、1 つのコード セクションのすべての関数を対象に実行されます。[1]

> **注 :** プロシージャ抽出を実行すると、プログラム サイズが縮小される代わりに実行速度が低下する場合があります。

次に、C ソース コードの例を示します。

```
distance -= time * speed;
position += time * speed;
```

1. デモ バージョンの試用期限が過ぎると、プロシージャ抽出は実行されません。

プロシージャ抽出を無効 (-Opa-) にしてコンパイルすると、time * speed に相当するコード シーケンスはすべての命令に対して生成されます。該当する部分を**太字**にして示します。

```
0x000000 main       MOVLB time
0x000002            MOVF time,0x0,0x1
0x000004            MULWF speed,0x1
0x000006            MOVF PRODL,0x0,0x0
0x000008            MOVWF PRODL,0x0
0x00000a            CLRF PRODL+1,0x0
0x00000c            MOVF WREG,0x0,0x0
0x00000e            SUBWF distance,0x1,0x1
0x000010            MOVF PRODL+1,0x0,0x0
0x000012            SUBWFB distance+1,0x1,0x1
0x000014            MOVF time,0x0,0x1
0x000016            MULWF speed,0x1
0x000018            MOVF PRODL,0x0,0x0
0x00001a            MOVWF PRODL,0x0
0x00001c            CLRF PRODL+1,0x0
0x00001e            MOVF WREG,0x0,0x0
0x000020            ADDWF position,0x1,0x1
0x000022            MOVF PRODL+1,0x0,0x0
0x000024            ADDWFC position+1,0x1,0x1
0x000026            RETURN 0x0
```

これに対し、プロシージャ抽出を有効 (-Opa+) にしてコンパイルすると、この 2 つのコード シーケンスはプロシージャとして抽出され、繰り返しコードはそのプロシージャへの呼び出しで置き換えられます。

```
0x000000 main       MOVLB time
0x000002            CALL _pa_0,0x0
0x000004
0x000006            SUBWF distance,0x1,0x1
0x000008            MOVF PRODL+1,0x0,0x0
0x00000a            SUBWFB distance+1,0x1,0x1
0x00000c            CALL _pa_0,0x0
0x00000e
0x000010            ADDWF position,0x1,0x1
0x000012            MOVF PRODL+1,0x0,0x0
0x000014            ADDWFC position+1,0x1,0x1
0x000016            RETURN 0x0
0x000018 __pa_0     MOVF time,0x0,0x1
0x00001a            MULWF speed,0x1
0x00001c            MOVF PRODL,0x0,0x0
0x00001e            MOVWF PRODL,0x0
0x000020            CLRF PRODL+1,0x0
0x000022            MOVF WREG,0x0,0x0
0x000024            RETURN 0x0
```

プロシージャ抽出を 1 回実行しただけでは、すべての繰り返し部分を抽出できません。プロシージャ抽出は、抽出できる部分がなくなるまで ( または最大 4 回まで ) 実行されます。プロシージャ抽出の実行回数は、コマンドライン オプション -pa=*n* で制御できます。プロシージャ抽出により、関数呼び出しの階層が $2^n$ - 1 (*n* はプロシージャ抽出の実行回数 ) だけ増える可能性があります。ハードウェア スタックのリソースに制約のあるアプリケーションでは、コマンドライン オプション -pa=*n* でプロシージャ抽出の実行回数を調整できます。

この最適化機能を有効にしてコンパイルしたソース コードをデバッグすると、複数のソース行がアセンブリ コード上で 1 つのシーケンスを共有するため、どのソース行が実行されているのかをデバッガが判断できず、誤ったソース行がハイライトされる場合があります。

# 第 5 章 サンプル コード

## 5.1 アプリケーション : LED と割り込みを使用する組み込み「HELLO, WORLD!」

次に示すサンプル アプリケーションは、PIC18F452 マイクロコントローラの PORTB に接続された LED を点灯するというものです。このアプリケーションでは、次のコマンドラインを実行してビルドします。

```
mcc18 -p 18f452 -I c:\mcc18\h leds.c
```

c:\mcc18 は、コンパイラのインストール先ディレクトリです。このサンプル アプリケーションは、PICDEM™ 2 デモ ボードで使用できるように設計されています。このサンプルには、次の要素が含まれます。

1. 割り込み処理 (#pragma interruptlow、割り込みベクタ、割り込みサービス ルーチン )
2. システム ヘッダ ファイル
3. プロセッサ固有のヘッダ ファイル
4. #pragma *sectiontype*
5. インライン アセンブリ

```
/*  1 */ #include <p18cxxx.h>
/*  2 */ #include <timers.h>
/*  3 */
/*  4 */ #define NUMBER_OF_LEDS 8
/*  5 */
/*  6 */ void timer_isr (void);
/*  7 */
/*  8 */ static unsigned char s_count = 0;
/*  9 */
/* 10 */ #pragma code low_vector=0x18
/* 11 */ void low_interrupt (void)
/* 12 */ {
/* 13 */   _asm GOTO timer_isr _endasm
/* 14 */ }
/* 15 */
/* 16 */ #pragma code
/* 17 */
/* 18 */ #pragma interruptlow timer_isr
/* 19 */ void
/* 20 */ timer_isr (void)
/* 21 */ {
/* 22 */   static unsigned char led_display = 0;
/* 23 */
/* 24 */   INTCONbits.TMR0IF = 0;
/* 25 */
/* 26 */   s_count = s_count % (NUMBER_OF_LEDS + 1);
/* 27 */
/* 28 */   led_display = (1 << s_count++) - 1;
/* 29 */
/* 30 */   PORTB = led_display;
/* 31 */ }
/* 32 */
/* 33 */ void
/* 34 */ main (void)
/* 35 */ {
/* 36 */   TRISB = 0;
/* 37 */   PORTB = 0;
/* 38 */
/* 39 */   OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT);
/* 40 */   INTCONbits.GIE = 1;
/* 41 */
/* 42 */   while (1)
/* 43 */     {
/* 44 */     }
/* 45 */ }
```

**1 行目：** この行で汎用プロセッサ ヘッダ ファイルをインクルードします。プロセッサは、コマンドライン オプション -p で正しく選択されています (**2.5.1 項「システム ヘッダ ファイル」**および **2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**10 行目：** PIC18 デバイスでは、低優先度の割り込みベクタは 00000018h 番地です。この行から、デフォルトのコード セクションは、low_vector という名前の絶対コード セクション (0x18 番地 ) に変更されます (**2.9.1 項「#pragma *sectiontype*」**および **2.9.2.3 項「割り込みベクタ」**参照 )。

**13 行目：** この行には、ISR にジャンプするインライン アセンブリが含まれます (**2.8.2 項「インライン アセンブリ」**および **2.9.2.3 項「割り込みベクタ」**参照 )。

**16 行目：** この行から、デフォルトのコード セクションに戻ります (**2.9.1 項「#pragma *sectiontype*」**および表 2-6 参照 )。

**18 行目：** この行で、関数 timer_isr を、低優先度の割り込みサービス ルーチンとして指定します。関数 timer_isr に対して、コンパイラが RETURN 命令ではなく RETFIE 命令を生成するためには、この行が必要です (**2.9.2 項「#pragma interruptlow *fname* / #pragma interrupt *fname*」**参照 )。

**19～20行目:** これらの行で関数 timer_isr を定義します。この関数は ISR であるため、パラメータも戻り値もないことに注意してください (**2.9.2.2 項「割り込みサービス ルーチン」**参照 )。

**24 行目：** この行で割り込みフラグ TMR0 をクリアし、プログラムが同じ割り込みを何度も処理するのを回避しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**30 行目：** この行は、特殊機能レジスタ PORTB の、C での書き換え方法を示しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**36-37 行目：** これらの行で、特殊機能レジスタ TRISB と PORTB を初期化しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**39 行目：** この行では割り込み TMR0 を許可し、内蔵 16 ビット クロックをタイマに設定しています。

**40 行目：** この行でグローバル割り込みを許可しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

## 5.2 アプリケーション : 大規模なデータ オブジェクトの作成と USART

次に示すサンプル アプリケーションでは、ユーザーが (HyperTerminal® を使用して ) 0 ～ 9 の 1 桁の数字を入力します。USART から文字を受信すると、データ配列から文字列を出力します。受信した文字が 0 ～ 9 以外の場合は、エラー文字列を出力します。このアプリケーションは、次のコマンドラインを実行してビルドします。

```
mcc18 -p 18f452 -I c:\mcc18\h example2.c
```

c:\mcc18 は、コンパイラのインストール先ディレクトリです。このサンプル アプリケーションは、MPLAB ICD2、PICDEM™ 2 Plus デモ ボード、PIC18F452 デバイスで使用できるように設計されています。このサンプルには、次の要素が含まれます。

1. 大規模なデータ オブジェクトの作成
2. USART に対する読み出しと書き込み
3. 割り込み処理 (#pragma interrupt、割り込みベクタ、割り込みサービス ルーチン )
4. システム ヘッダ ファイル
5. プロセッサ固有のヘッダ ファイル
6. #pragma *sectiontype*
7. インライン アセンブリ

デフォルトでは、MPLAB C18 は 1 つのオブジェクトがバンク境界をまたぐことはないと想定しています。256 バイトを超えるオブジェクトの作成も可能ですが、次の手順を実行してマルチバンク オブジェクトを作成する必要があります。

1. #pragma idata または #pragma udata ディレクティブを使用して、オブジェクトを専用のセクションに配置する。
   ```
   #pragma udata buffer_scn
   static char buffer[0x180];
   #pragma udata
   ```
2. このオブジェクトへのアクセスには、必ずポインタを使用する。
   ```
   char * buf_ptr = &buffer[0];
   ...
   // examples of use
   buf_ptr[5] = 10;
   if (buf_ptr[275] > 127)
   ...
   ```
3. 複数のバンクにまたがる新規リージョンは、リンカ スクリプトで作成する。

   修正前のリンカ スクリプト :
   ```
   DATABANK NAME=gpr2 START=0x200 END=0x2FF
   DATABANK NAME=gpr3 START=0x300 END=0x3FF
   ```
   修正後のリンカ スクリプト :
   ```
   DATABANK NAME=big  START=0x200 END=0x37F PROTECTED
   DATABANK NAME=gpr3 START=0x380 END=0x3FF
   ```
4. オブジェクトのセクション ( 手順 1 で作成したもの ) を、新規リージョン ( 手順 3 で作成したもの ) に割り当てる。リンカ スクリプトに SECTION ディレクティブを追加する。
   ```
   SECTION NAME=buffer_scn RAM=big
   ```

```
/*  1 */ #include <p18cxxx.h>
/*  2 */ #include <usart.h>
/*  3 */
/*  4 */ void rx_handler (void);
/*  5 */
/*  6 */ #define BUF_SIZE 25
/*  7 */
/*  8 */ /*
/*  9 */  * Step #1 - The data is allocated into its own section.
/* 10 */  */
/* 11 */ #pragma idata bigdata
/* 12 */ char data[11][BUF_SIZE+1] = {
/* 13 */   { "String #0\n\r" },
/* 14 */   { "String #1\n\r" },
/* 15 */   { "String #2\n\r" },
/* 16 */   { "String #3\n\r" },
/* 17 */   { "String #4\n\r" },
/* 18 */   { "String #5\n\r" },
/* 19 */   { "String #6\n\r" },
/* 20 */   { "String #7\n\r" },
/* 21 */   { "String #8\n\r" },
/* 22 */   { "String #9\n\r" },
/* 23 */   { "Invalid key (0-9 only)\n\r" }
/* 24 */ };
/* 25 */ #pragma idata
/* 26 */
/* 27 */ #pragma code rx_interrupt = 0x8
/* 28 */ void rx_int (void)
/* 29 */ {
/* 30 */   _asm goto rx_handler _endasm
/* 31 */ }
/* 32 */ #pragma code
/* 33 */
/* 34 */ #pragma interrupt rx_handler
/* 35 */ void rx_handler (void)
/* 36 */ {
/* 37 */   unsigned char c;
/* 38 */
/* 39 */   /* Get the character received from the USART */
/* 40 */   c = ReadUSART();
/* 41 */   if (c >= '0' && c <= '9')
/* 42 */     {
/* 43 */       c -= '0';
/* 44 */       /* Display value received on LEDs */
/* 45 */       PORTB = c;
/* 46 */
/* 47 */       /*
/* 48 */        * Step #2 - This example did not need an additional
/* 49 */        * pointer to access the large memory because of the
/* 50 */        * multi-dimension array.
/* 51 */        *
/* 52 */        * Display the string located at the array offset
/* 53 */        * of the character received
/* 54 */        */
/* 55 */       putsUSART (data[c]);
/* 56 */     }
/* 57 */   else
/* 58 */     {
/* 59 */       /*
/* 60 */        * Step #2 - This example did not need an additional
```

```
/*  61 */        * pointer to access the large memory because of the
/*  62 */        * multi-dimension array.
/*  63 */        *
/*  64 */        * Invalid character received from USART.
/*  65 */        * Display error string.
/*  66 */        */
/*  67 */       putsUSART (data[10]);
/*  68 */
/*  69 */       /* Display value received on LEDs */
/*  70 */       PORTB = c;
/*  71 */     }
/*  72 */
/*  73 */     /* Clear the interrupt flag */
/*  74 */     PIR1bits.RCIF = 0;
/*  75 */ }
/*  76 */
/*  77 */ void main (void)
/*  78 */ {
/*  79 */   /* Configure all PORTB pins for output */
/*  80 */   TRISB = 0;
/*  81 */
/*  82 */   /*
/*  83 */    * Open the USART configured as
/*  84 */    * 8N1, 2400 baud, in polled mode
/*  85 */    */
/*  86 */   OpenUSART (USART_TX_INT_OFF &
/*  87 */              USART_RX_INT_ON &
/*  88 */              USART_ASYNCH_MODE &
/*  89 */              USART_EIGHT_BIT &
/*  90 */              USART_CONT_RX &
/*  91 */              USART_BRGH_HIGH, 103);
/*  92 */
/*  93 */   /* Display a prompt to the USART */
/*  94 */   putrsUSART (
/*  95 */     (const far rom char *)"\n\rEnter a digit 0-9!\n\r");
/*  96 */
/*  97 */   /* Enable interrupt priority */
/*  98 */   RCONbits.IPEN = 1;
/*  99 */
/* 100 */   /* Make receive interrupt high priority */
/* 101 */   IPR1bits.RCIP = 1;
/* 102 */
/* 103 */   /* Enable all high priority interrupts */
/* 104 */   INTCONbits.GIEH = 1;
/* 105 */
/* 106 */   /* Loop forever */
/* 107 */   while (1)
/* 108 */     ;
/* 109 */ }
```

**1 行目：** この行で汎用プロセッサ ヘッダ ファイルをインクルードします。プロセッサは、コマンドライン オプション -p で正しく選択されています (**2.5.1 項「システム ヘッダ ファイル」**および **2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**11 行目：** 大規模なオブジェクトの作成 : 手順 1。#pragma idata ディレクティブを使用し、初期化済みデータ変数 data を専用のセクションに格納します。(**2.9.1 項「#pragma *sectiontype*」**参照 )。

**25 行目：** この行から、デフォルトの初期化済みデータ セクションに戻ります (**2.9.1 項「#pragma *sectiontype*」**および表 2-6 参照 )。

**27 行目：** PIC18 デバイスでは、高優先度の割り込みベクタは 00000008h 番地です。この行から、デフォルトのコード セクションは、rx_interrupt という名前の絶対コード セクション (0x8 番地 ) に変更されます (**2.9.1 項「#pragma *sectiontype*」**および **2.9.3.4 項「入れ子の割り込み」**参照 )。

**32 行目：** この行から、デフォルトのコード セクションに戻ります (**2.9.1 項「#pragma *sectiontype*」**および表 2-6 参照 )。

**34 行目：** この行で、関数 rx_handler を、高優先度の割り込みサービス ルーチンとして指定します。関数 rx_handler に対して、コンパイラが RETURN 命令ではなく RETFIE 命令を生成するためには、この行が必要です (**2.9.2 項「#pragma interruptlow fname / #pragma interrupt fname」**参照 )。

**35 行目：** この行で関数 rx_handler を定義します。この関数は ISR であるため、パラメータも戻り値もないことに注意してください (**2.9.2.3 項「割り込みベクタ」**参照 )。

**45、70 行目：** これらの行は、特殊機能レジスタ PORTB の、C での書き換え方法を示しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**55、67 行目：** 大規模なオブジェクトの作成 : 手順 2。大規模なオブジェクトに間接アクセスします。これらの行で、データ文字列が USART に出力されます。

**74 行目：** この行は、特殊機能レジスタ PIR1 の特定ビットの、C での書き換え方法を示しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**80 行目：** この行で特殊機能レジスタ TRISB を初期化します (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**86 ～ 91 行目：** これらの行で、8N1 として、2400 ボーで、ポーリング モードに設定された USART をオープンします。また、USART の受信割り込みもここで許可します。

**98 行目：** この行で、PIC18 の割り込み優先度付け機能を有効にします (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**101 行目：** この行で、USART の受信割り込みを高優先度の割り込みソースに指定します (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**104 行目：** この行で、高優先度のすべての割り込みを許可します (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

リンカ スクリプト：

```
// This file was originally 18f452i.lkr as distributed with MPLAB C18.
// Modified as follows:
// - combine banks 4 and 5 into PROTECTED DATABANK "largebank"
// - moved stack to gpr3
// - Assign the "bigdata" SECTION into the new "largebank" region

LIBPATH .

FILES c018i.o
FILES clib.lib
FILES p18f452.lib

CODEPAGE   NAME=vectors    START=0x0            END=0x29           PROTECTED
CODEPAGE   NAME=page       START=0x2A           END=0x7DBF
CODEPAGE   NAME=debug      START=0x7DC0         END=0x7FFF         PROTECTED
CODEPAGE   NAME=idlocs     START=0x200000       END=0x200007       PROTECTED
CODEPAGE   NAME=config     START=0x300000       END=0x30000D       PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE       END=0x3FFFFF       PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000       END=0xF000FF       PROTECTED

ACCESSBANK NAME=accessram  START=0x0            END=0x7F
DATABANK   NAME=gpr0       START=0x80           END=0xFF
DATABANK   NAME=gpr1       START=0x100          END=0x1FF
DATABANK   NAME=gpr2       START=0x200          END=0x2FF
DATABANK   NAME=gpr3       START=0x300          END=0x3FF
// Step #3 - Create a new region in the linker script
// This is the databank that will contain the large memory object
DATABANK   NAME=largebank  START=0x400          END=0x5F3          PROTECTED
DATABANK   NAME=dbgspr     START=0x5F4          END=0x5FF          PROTECTED
ACCESSBANK NAME=accesssfr  START=0xF80          END=0xFFF          PROTECTED

SECTION    NAME=CONFIG     ROM=config

// Step #4 - Assign the large memory object's section into the new region
SECTION    NAME=bigdata    RAM=largebank

STACK SIZE=0x100 RAM=gpr3
```

## 5.3 アプリケーション : EEDATA と複数の割り込みソースの利用

次のサンプル アプリケーションでは、PIC18FXX20 64/80L TQFP デモ ボードにおいて PORTD に接続された LED が回転発光します。LED の最初の回転方向は、EEDATA から読み出されます。左下のボタンを押すと LED の回転方向が反転し、更新後の変数 direction が EEDATA に書き込まれます。LED 光の回転速度は、POT に接続された ADC のアナログ チャネル 0 (AN0) で制御します。このアプリケーションは、次のコマンドラインを実行してビルドします。

```
mcc18 -p 18f8720 -I c:\mcc18\h example3.c
```

c:\mcc18 は、コンパイラのインストール先ディレクトリです。このアプリケーションは、MPLAB ICD2、PIC18FXX20 64/80L TQFP デモ ボード、PIC18F8720 デバイスで使用できるように設計されています。このサンプルには、次の要素が含まれます。

1. EEDATA に対する読み出しと書き込み
2. 割り込み駆動型のアクセスと、周辺モジュールのポーリングによるアクセスを併用
3. 割り込み優先度の使用 (#pragma interrupt、#pragma interruptlow、割り込みベクタ、割り込みサービス ルーチン )
4. C コードによる構成ビットの設定
5. #pragma *sectiontype*
6. インライン アセンブリ

EEDATA の読み出しに必要な手順とそれに関連する C コードは次のとおりです。

1. EEDATA にアクセスするために EEPGD をクリアする。
   ```
   EECON1bits.EEPGD = 0;
   ```
2. アドレスを EEADR に格納する。
   EEADR = *addr*;
3. RD ビットをセットして、読み出しを開始する。
   ```
   EECON1bits.RD = 1;
   ```
4. EEDATA レジスタから結果を読み出す。
   ```
   my_variable = EEDATA;
   ```

EEDATA の書き込みに必要な手順とそれに関連する C コードは次のとおりです。

**注 :** 手順 5 ～ 7 の間は、割り込みを禁止しておく必要があります。

1. EEDATA にアクセスするために EEPGD をクリアする。
   ```
   EECON1bits.EEPGD = 0;
   ```
2. EEDATA への書き込みを許可するために WREN をセットする。
   ```
   EECON1bits.WREN = 1;
   ```
3. アドレスを EEADR に書き込む。
   EEADR = *addr*;
4. EEDATA に書き込み値をセットする。
   EEDATA = *value*;
5. EECON2 に 0x55 を書き込む。
   EECON2 = 0x55;
6. EECON2 に 0xAA を書き込む。
   EECON2 = 0xAA;
7. WR ビットをセットして書き込みサイクルを開始する。
   ```
   EECON1bits.WR = 1;
   ```
8. EEIF フラグがセットされるまで待機する。
   ```
   while (!PIR2bits.EEIF)
    ;
   ```
9. EEIF フラグをクリアする。
   ```
   PIR2bits.EEIF = 0;
   ```

```
/*  1 */ #include <p18cxxx.h>
/*  2 */ #include <delays.h>
/*  3 */
/*  4 */ /* Set up the configuration bits */
/*  5 */ #pragma config OSC = HS, OSCS = OFF
/*  6 */ #pragma config PWRT = OFF
/*  7 */ #pragma config BOR = OFF
/*  8 */ #pragma config WDT = OFF
/*  9 */ #pragma config CCP2MUX = OFF
/* 10 */ #pragma config LVP = OFF
/* 11 */
/* 12 */ void tmr2 (void);
/* 13 */ void button (void);
/* 14 */
/* 15 */ #pragma code high_vector_section=0x8
/* 16 */ void
/* 17 */ high_vector (void)
/* 18 */ {
/* 19 */   _asm GOTO button _endasm
/* 20 */ }
/* 21 */ #pragma code low_vector_section=0x18
/* 22 */ void
/* 23 */ low_vector (void)
/* 24 */ {
/* 25 */   _asm GOTO tmr2 _endasm
/* 26 */ }
/* 27 */ #pragma code
/* 28 */
/* 29 */ volatile unsigned current_ad_value;
/* 30 */ int count = 0;
/* 31 */ volatile enum { DIR_LEFT = 0, DIR_RIGHT } direction;
/* 32 */
/* 33 */ #pragma interruptlow tmr2
/* 34 */ void
/* 35 */ tmr2 (void)
/* 36 */ {
/* 37 */   /* clear the timer interrupt flag */
/* 38 */   PIR1bits.TMR2IF = 0;
/* 39 */
/* 40 */   /*
/* 41 */    * if we have reached the repeat count,
/* 42 */    * update the LEDs
/* 43 */    */
/* 44 */   if (count++ < current_ad_value)
/* 45 */     return;
/* 46 */   else
/* 47 */     count = 0;
/* 48 */
/* 49 */   /*
/* 50 */    * Based on the direction, rotate the LEDs
/* 51 */    */
/* 52 */   if (direction == DIR_LEFT)
/* 53 */     {
/* 54 */       _asm RLNCF PORTD, 1, 0 _endasm
/* 55 */     }
/* 56 */   else
/* 57 */     {
/* 58 */       _asm RRNCF PORTD, 1, 0 _endasm
/* 59 */     }
/* 60 */ }
```

```
/*  61 */
/*  62 */ #pragma interrupt button
/*  63 */ void
/*  64 */ button (void)
/*  65 */ {
/*  66 */   direction = !direction;
/*  67 */
/*  68 */   /*
/*  69 */    * Store the new direction in EEDATA.
/*  70 */    * Note that since we are already
/*  71 */    * in the high priority interrupt, we do
/*  72 */    * not need to explicitly enable/disable
/*  73 */    * interrupts around the write cycle
/*  72 */    */
/*  73 */   EECON1bits.EEPGD = 0;  /* WRITE step #1 */
/*  74 */   EECON1bits.WREN = 1;   /* WRITE step #2 */
/*  75 */   EEADR = 0;             /* WRITE step #3 */
/*  76 */   EEDATA = direction;    /* WRITE step #4 */
/*  77 */   EECON2 = 0x55;         /* WRITE step #5 */
/*  78 */   EECON2 = 0xaa;         /* WRITE step #6 */
/*  79 */   EECON1bits.WR = 1;     /* WRITE step #7 */
/*  80 */   while (!PIR2bits.EEIF) /* WRITE step #8 */
/*  81 */     ;
/*  82 */   PIR2bits.EEIF = 0;     /* WRITE step #9 */
/*  83 */
/*  84 */   /* clear the interrupt flag */
/*  85 */   INTCONbits.INT0IF = 0;
/*  86 */ }
/*  87 */
/*  88 */ void
/*  89 */ main (void)
/*  90 */ {
/*  91 */   /*
/*  92 */    * The first thing to do is to read
/*  93 */    * the start direction from data EEPROM.
/*  94 */    */
/*  95 */   EECON1bits.EEPGD = 0;  /* READ step #1 */
/*  96 */   EEADR = 0;             /* READ step #2 */
/*  97 */   EECON1bits.RD = 1;     /* READ step #3 */
/*  98 */   direction = EEDATA;    /* READ step #4 */
/*  99 */
/* 100 */   /*
/* 101 */    * Make all bits on the Port D output
/* 102 */    * bits for the LEDs
/* 103 */    */
/* 104 */   TRISD = 0;
/* 105 */
/* 106 */   /*
/* 107 */    * Make PORTA RA0 input, for the A/D
/* 107 */    * converter
/* 108 */    */
/* 109 */   TRISAbits.TRISA0 = 1;
/* 110 */
/* 111 */   /* PORTB RB0 input for the button */
/* 112 */   TRISBbits.TRISB0 = 1;
/* 113 */
/* 114 */   /* Reset Port D. Set just one bit to on. */
/* 115 */   PORTD = 1;
/* 116 */
/* 117 */   /* Enable interrupt priority */
```

```
/* 118 */   RCONbits.IPEN = 1;
/* 119 */
/* 120 */   /* Clear the peripheral interrupt flags */
/* 121 */   PIR1 = 0;
/* 122 */
/* 123 */   /* Enable the timer interrupt */
/* 124 */   PIE1bits.TMR2IE = 1;
/* 125 */   IPR1bits.TMR2IP = 0;
/* 126 */
/* 127 */   /*
/* 128 */    * Set the button on RB0 to trigger an
/* 129 */    * interrupt.  It is always high priority
/* 130 */    */
/* 131 */   INTCONbits.INT0IE = 1;
/* 132 */
/* 133 */   /* Configure the ADC, most of this is the
/* 134 */    * default settings:
/* 135 */    * Fosc/32
/* 136 */    * AN0 Analog,
/* 137 */    * AN1-15 Digital Channel zero Interrupt
/* 138 */    *   disabled
/* 139 */    * Internal voltage references
/* 140 */    */
/* 141 */
/* 142 */   /* FOSC/32 clock select */
/* 143 */   ADCON2bits.ADCS0 = 1;
/* 144 */   ADCON2bits.ADCS1 = 1;
/* 145 */   ADCON2bits.ADCS2 = 1;
/* 146 */   ADCON2bits.ADCS2 = 1;
/* 147 */
/* 148 */   /* AN0-15, VREF */
/* 149 */   ADCON1 = 0b00001110;
/* 150 */
/* 151 */   /* Enable interrupts */
/* 152 */   INTCONbits.GIEH = 1;
/* 153 */   INTCONbits.GIEL = 1;
/* 154 */
/* 155 */   /* Turn on the ADC */
/* 156 */   ADCON0bits.ADON = 1;
/* 157 */
/* 158 */   /* Enable the timer */
/* 159 */   T2CONbits.TMR2ON = 1;
/* 160 */
/* 161 */   /* Start the ADC conversion */
/* 162 */   while (1)
/* 163 */     {
/* 164 */       /* Give the ADC time to get ready. */
/* 165 */       Delay100TCYx (2);
/* 166 */
/* 167 */       /* start the ADC conversion */
/* 168 */       ADCON0bits.GO = 1;
/* 169 */       while (ADCON0bits.GO)
/* 170 */         ;
/* 171 */       current_ad_value = ADRES;
/* 172 */     }
/* 173 */ }
```

**1 行目：** この行で generic processor header file をインクルードします。プロセッサは、コマンドライン オプション -p で正しく選択されています (**2.5.1 項「システム ヘッダ ファイル」**および **2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**5 ～ 10 行目：** これらの行で、#pragma config ディレクティブを使用して構成ビットをセットします (**2.9.5 項「#pragma config」**参照 )。

**15 行目：** PIC18 デバイスでは、高優先度の割り込みベクタは 00000008h 番地です。この行から、デフォルトのコード セクションは、high_vector_section という名前の絶対コード セクション (0x8 番地 ) に変更されます (**2.9.1 項「#pragma *sectiontype*」**および **2.9.2.4 項「ISRのコンテクスト保存」**参照 )。

**21 行目：** PIC18 デバイスでは、低優先度の割り込みベクタは 00000018h 番地です。この行から、デフォルトのコード セクションは、low_vector_section という名前の絶対コード セクション (0x18 番地 ) に変更されます (**2.9.1 項「#pragma *sectiontype*」**および **2.9.2.4 項「ISRのコンテクスト保存」**参照 )。

**27 行目：** この行から、デフォルトのコード セクションに戻ります (**2.9.1 項「#pragma *sectiontype*」**および表 2-6 参照 )。

**29、31 行目：** これらの行では volatile キーワードを使用して、これらの変数がメイン ラインのコードと割り込みサービス ルーチンの両方で使用することを宣言しています。volatile キーワードがある場合、コンパイラはこれら変数へのアクセスを最適化しません。

**33 行目：** この行で、関数 tmr2 を、低優先度の割り込みサービス ルーチンとして指定します。関数 tmr2 に対して、コンパイラが RETURN 命令ではなく RETFIE 命令を生成するためには、この行が必要です (**2.9.2 項「#pragma interruptlow fname / #pragma interrupt fname」**参照 )。

**34 ～ 35 行目：** これらの行で関数 tmr2 を定義します。この関数は ISR であるため、パラメータも戻り値もないことに注意してください (**2.9.2.3 項「割り込みベクタ」**参照 )。

**38 行目：** この行は、特殊機能レジスタ PIR1 の特定ビットの、C での書き換え方法を示しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**54、58 行目：** これらの行では、ANSI C では直接扱わない処理をインライン アセンブリで実行しています (**2.8.2 項「インライン アセンブリ」**参照 )。

**62 行目：** この行で、関数 button を、高優先度の割り込みサービス ルーチンとして指定します。関数 button に対して、コンパイラが RETURN 命令ではなく RETFIE 命令を生成するためには、この行が必要です (**2.9.2 項「#pragma interruptlow fname / #pragma interrupt fname」**参照 )。

**63 ～ 64 行目：** これらの行で関数 button を定義します。この関数は ISR であるため、パラメータも戻り値もないことに注意してください (**2.9.2.3 項「割り込みベクタ」**参照 )。

**45、70 行目：** これらの行は、特殊機能レジスタ PORTB の、C での書き換え方法を示しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**73 ～ 82 行目：** これらの行は、EEDATA への書き込み方法を示しています。それぞれの行で、特殊機能レジスタまたは特殊機能レジスタ内の 1 つのビットを使用しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**85 行目：** この行は、特殊機能レジスタ INTCON の特定のビットの、C での書き換え方法を示しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**95 ～ 98 行目：** これらの行は、EEDATA からの読み出し方法を示しています。それぞれの行で、特殊機能レジスタまたは特殊機能レジスタ内の 1 つのビットを使用しています (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**118 行目：** この行で、PIC18 の割り込み優先度付け機能を有効にします (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**124 〜 125 行目：** これらの行で TMR2 割り込みを許可し、この割り込みを低優先度の割り込みソースに指定します (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

**152 〜 153 行目：** この行で、すべての割り込みを許可します (**2.10 項「プロセッサ固有ヘッダ ファイル」**参照 )。

# 付録 A COFF ファイル フォーマット

マイクロチップ社の COFF 仕様は、『*Understanding and Using COFF*』(Gintaras R. Gircys © 1988, O'Reilly and Associates, Inc.) に記載された UNIX® System V COFF フォーマットに準拠しています。ここでは、両者で異なる部分について説明します。

## A.1 struct filehdr – ファイル ヘッダ

構造体 filehdr には、ファイルに関する情報が含まれます。COFF ファイルは、必ずこのファイル ヘッダから開始します。ここには、オプション ファイル ヘッダ、シンボル テーブル、セクション ヘッダの開始場所を記述します。

```
typedef struct filehdr
{
   unsigned short f_magic;
   unsigned short f_nscns;
   unsigned long f_timdat;
   unsigned long f_symptr;
   unsigned long f_nsyms;
   unsigned short f_opthdr;
   unsigned short f_flags;
} filehdr_t;
```

### A.1.1 unsigned short f_magic

このマジック ナンバーは、このファイルが従う COFF ファイルの使用の識別に使用します。マイクロチップ社の PICmicro MCU COFF ファイルでは、このナンバーは 0x1240 です。

### A.1.2 unsigned short f_nscns

COFF ファイル内のセクション数を示します。

### A.1.3 unsigned long f_timdat

COFF ファイルの作成日時 ( タイムスタンプ ) を示します。この値は、1970 年 1 月 1 日 0:00 からの秒数で表されます。

### A.1.4 unsigned long f_symptr

シンボル テーブルへのポインタを示します。

### A.1.5 unsigned long f_nsyms

シンボル テーブル内のエントリ数を示します。

### A.1.6 unsigned short f_opthdr

オプション ヘッダのレコード サイズを示します。

### A.1.7 unsigned short f_flags

COFF ファイルに含まれる内容に関する情報です。表 A-1 に、ファイル ヘッダ フラグの種類と その説明、およびそれぞれの値を示します。

**表 A-1: ファイル ヘッダ フラグ**

| Flag | Description | Value |
|---|---|---|
| F_RELFLG | Relocation information has been stripped from the COFF file. | 0x0001 |
| F_EXEC | The file is executable and has no unresolved external symbols. | 0x0002 |
| F_LNNO | Line number information has been stripped from the COFF file. | 0x0004 |
| L_SYMS | Local symbols have been stripped from the COFF file. | 0x0080 |
| F_EXTENDED18 | The COFF file produced utilizing the Extended mode. | 0x4000 |
| F_GENERIC | The COFF file is processor independent. | 0x8000 |

## A.2 struct opthdr – オプション ファイル ヘッダ

構造体 opthdr には、処理系に依存するファイル レベルの情報が含まれます。PICmicro MCU の COFF ファイルでは、ターゲット プロセッサの名前やコンパイラ / アセンブラのバージョンの指定、および再配置タイプの定義に使用します。

なお、このヘッダのレイアウトは処理系固有のものです ( すなわち、マイクロチップ社のオプション ヘッダとSystem Vのオプション ヘッダはフォーマットが異なります)。

```
typedef struct opthdr
{
   unsigned short magic;
   unsigned long vstamp;
   unsigned long proc_type;
   unsigned long rom_width_bits;
   unsigned long ram_width_bits;
} opthdr_t;
```

### A.2.1 unsigned short magic

このマジック ナンバーを使用して、適切なレイアウトを決定します。

### A.2.2 unsigned long vstamp

バージョン スタンプを示します。

### A.2.3 unsigned long proc_type

ターゲット プロセッサのタイプです。表 A-2 に、各プロセッサ タイプ、およびこのフィールドに格納される値を示します。

**表 A-2: プロセッサ タイプ**

| Processor | Value | Processor | Value |
|---|---|---|---|
| PIC18C242 | 0x8242 | PIC18F2620 | 0x2620 |
| PIC18C252 | 0x8252 | PIC18F2680 | 0x2680 |
| PIC18C442 | 0x8442 | PIC18F4220 | 0xA422 |
| PIC18C452(1) | 0x8452 | PIC18F4221 | 0x4221 |
| PIC18C601 | 0x8601 | PIC18F4320 | 0xA432 |
| PIC18C658 | 0x8658 | PIC18F4321 | 0x4321 |
| PIC18C801 | 0x8801 | PIC18F4331 | 0x4331 |
| PIC18C858 | 0x8858 | PIC18F4410 | 0x4410 |
| PIC18F1220 | 0xA122 | PIC18F442 | 0x442F |
| PIC18F1230 | 0x1230 | PIC18F4420 | 0x4420 |
| PIC18F1231 | 0x1231 | PIC18F4431 | 0x4431 |
| PIC18F1320 | 0xA132 | PIC18F4439 | 0x4439 |
| PIC18F1330 | 0x1330 | PIC18F4450 | 0x4450 |
| PIC18F1331 | 0x1331 | PIC18F4455 | 0x4455 |
| PIC18F2220 | 0xA222 | PIC18F448 | 0x8448 |
| PIC18F2221 | 0x2221 | PIC18F4480 | 0x4480 |
| PIC18F2320 | 0xA232 | PIC18F44J10 | 0xE410 |
| PIC18F2321 | 0x2321 | PIC18F4510 | 0x4510 |
| PIC18F2331 | 0x2331 | PIC18F4515 | 0x4515 |
| PIC18F2410 | 0x2410 | PIC18F452 | 0x452F |
| PIC18F242 | 0x242F | PIC18F4520 | 0x4520 |
| PIC18F2420 | 0x2420 | PIC18F4525 | 0x4525 |
| PIC18F2431 | 0x2431 | PIC18F4539 | 0x4539 |
| PIC18F2439 | 0x2439 | PIC18F4550 | 0x4550 |
| PIC18F2450 | 0x2450 | PIC18F458 | 0x8458 |
| PIC18F2455 | 0x2455 | PIC18F4580 | 0x4580 |
| PIC18F248 | 0x8248 | PIC18F4585 | 0x4585 |
| PIC18F2480 | 0x2480 | PIC18F45J10 | 0xE510 |
| PIC18F24J10 | 0xD410 | PIC18F4610 | 0x4610 |
| PIC18F2510 | 0x2510 | PIC18F4620(2) | 0x4620 |
| PIC18F2515 | 0x2515 | PIC18F4680 | 0x4680 |
| PIC18F252 | 0x252F | PIC18F6310 | 0x6310 |
| PIC18F2520 | 0x2520 | PIC18F6390 | 0x6390 |
| PIC18F2525 | 0x2525 | PIC18F6410 | 0x6410 |
| PIC18F2539 | 0x2539 | PIC18F6490 | 0x6490 |
| PIC18F2550 | 0x2550 | PIC18F64J15 | 0xB415 |
| PIC18F258 | 0x8258 | PIC18F6520 | 0xA652 |
| PIC18F2580 | 0x2580 | PIC18F6525 | 0x6525 |
| PIC18F2585 | 0x2585 | PIC18F6527 | 0x6527 |
| PIC18F25J10 | 0xD510 | PIC18F6585 | 0x6585 |
| PIC18F2610 | 0x2610 | PIC18F65J10 | 0xB510 |

**表 A-2: プロセッサ タイプ ( 続き )**

| Processor | Value |
|---|---|
| PIC18F65J15 | 0xB515 |
| PIC18F6620 | 0xA662 |
| PIC18F6621 | 0xA621 |
| PIC18F6622 | 0xF622 |
| PIC18F6627 | 0x6627 |
| PIC18F6680 | 0x6680 |
| PIC18F66J10 | 0xB610 |
| PIC18F66J15 | 0xB615 |
| PIC18F66J60 | 0xB660 |
| PIC18F66J65 | 0xB665 |
| PIC18F6720 | 0xA672 |
| PIC18F6722 | 0x6722 |
| PIC18F67J10 | 0xB710 |
| PIC18F67J60 | 0xB760 |
| PIC18F8310 | 0x8310 |
| PIC18F8390 | 0x8390 |
| PIC18F8410 | 0x8410 |
| PIC18F8490 | 0x8490 |
| PIC18F84J15 | 0xC415 |
| PIC18F8520 | 0xA852 |
| PIC18F8525 | 0x8525 |
| PIC18F8527 | 0x8527 |
| PIC18F8585 | 0x8585 |
| PIC18F85J10 | 0xC510 |
| PIC18F85J15 | 0xC515 |
| PIC18F8620 | 0xA862 |
| PIC18F8621 | 0x8621 |
| PIC18F8622 | 0x8622 |
| PIC18F8627 | 0x8625 |
| PIC18F8680 | 0x8680 |
| PIC18F86J10 | 0xC610 |
| PIC18F86J15 | 0xC615 |
| PIC18F86J60 | 0xC660 |
| PIC18F86J65 | 0xC665 |
| PIC18F8720 | 0xA872 |
| PIC18F8722 | 0x8721 |
| PIC18F87J10 | 0xC710 |
| PIC18F87J60 | 0xC760 |
| PIC18F96J60 | 0xD660 |
| PIC18F96J65 | 0xD665 |
| PIC18F97J60 | 0xD760 |
| PIC18LF2423 | 0x2423 |
| PIC18LF2523 | 0x2523 |
| PIC18LF4423 | 0x4423 |
| PIC18LF4523 | 0x4523 |

**注 1:** コンパイラが非拡張モードで動作時に、汎用プロセッサ用にコンパイルを実行する場合はこのプロセッサが使用されます。

**2:** コンパイラが拡張モードで動作時に、汎用プロセッサ用にコンパイルを実行する場合はこのプロセッサが使用されます。

### A.2.4 unsigned long rom_width_bits

プログラム メモリのビット幅を示します。

### A.2.5 unsigned long ram_width_bits

データ メモリのビット幅を示します。

## A.3 struct scnhdr – セクション ヘッダ

構造体 scnhdr には、個々のセクションに関する情報が含まれます。PICmicro MCU の COFF ファイルでは、セクション名の定義が通常の COFF ファイルとは多少異なります。PICmicro MCU の COFF ファイルでは、セクション名が 8 文字を超える場合があるため、長いセクション名については文字列テーブル エントリが使用できます。

```
typedef struct scnhdr
{
    union
    {
      char _s_name[8] /* section name is a string */
      struct
        {
          unsigned long _s_zeroes
          unsigned long _s_offset
        }_s_s;
    }_s;

    unsigned long s_paddr;
    unsigned long s_vaddr;
    unsigned long s_size;
    unsigned long s_scnptr;
    unsigned long s_relptr;
    unsigned long s_lnnoptr;
    unsigned short s_nreloc;
    unsigned short s_nlnno;
    unsigned long s_flags;
} scnhdr_t;
```

### A.3.1 union _s

文字列または文字列テーブルへの参照を示します。8 文字より短い文字列は直接格納され、それ以上の文字列はすべて 文字列テーブルに格納されます。最初の 4 文字が 0 の場合、最後の 4 バイトは 文字列テーブルへのオフセットと見なされます。これは 厳密には ANSI 規格に準拠していないので、注意が必要です (ANSI 規格では、型の変更は定義されていません )。しかしこの方法は効果的であり、System V レイアウトとのバイナリ互換性を維持するのはこのオプションのみです。またこの実装には、標準の System V で長いシンボル名に使用される構造体をそのまま反映できるという利点があります。

#### A.3.1.1 char s_name[8]

インプレースのセクション名を示します。セクション名が 8 文字未満の場合、セクション名はインプレースで格納されます。

#### A.3.1.2 struct _s_s

セクション名は文字列テーブルに格納されます。セクション名の最初の 4 文字がゼロの場合、最後の 4 文字を文字列テーブルへのオフセットとして、セクション名を検索します。

##### A.3.1.2.1 unsigned long _s_zeroes

セクション名の最初の 4 文字がゼロであることを示します。

##### A.3.1.2.2 unsigned long _s_offset

文字列テーブルに格納されたセクション名のオフセットを示します。

#### A.3.1.3 unsigned long s_paddr

セクションの物理アドレスを示します。

#### A.3.1.4 unsigned long s_vaddr

セクションの仮想アドレスを示します。常に s_paddr と同じ値となります。

### A.3.2 unsigned long s_size

セクションのサイズを示します。

### A.3.3 unsigned long s_scnptr

このセクションに関する COFF ファイル内の原データへのポインタを示します。

### A.3.4 unsigned long s_relptr

このセクションに関する COFF ファイル内の再配置情報へのポインタを示します。

### A.3.5 unsigned long s_lnnoptr

このセクションに関する COFF ファイル内の行番号情報へのポインタを示します。

### A.3.6 unsigned short s_nreloc

このセクションの再配置エントリ数を示します。

### A.3.7 unsigned short s_nlnno

このセクションの行番号エントリ数を示します。

### A.3.8 unsigned long s_flags

セクションのタイプと内容のフラグを示します。セクションのタイプとセクション修飾子を定義するフラグは、s_flags フィールドにビット フィールドとして格納されます。ビット フィールドにはマスクが定義され、アクセスを容易にしています。表 A-3 に、セクション ヘッダ フラグの種類とその説明、およびそれぞれの値を示します。

**表 A-3: セクション ヘッダ フラグ**

| Flag | Description | Value |
|---|---|---|
| STYP_TEXT | Section contains executable code. | 0x00020 |
| STYP_DATA | Section contains initialized data. | 0x00040 |
| STYP_BSS | Section contains uninitialized data. | 0x00080 |
| STYP_DATA_ROM | Section contains initialized data for program memory. | 0x00100 |
| STYP_ABS | Section is absolute. | 0x01000 |
| STYP_SHARED | Section is shared across banks. | 0x02000 |
| STYP_OVERLAY | Section is overlaid with other sections of the same name from different object modules. | 0x04000 |
| STYP_ACCESS | Section is available using access bit. | 0x08000 |
| STYP_ACTREC | Section contains the overlay activation record for a function. | 0x10000 |

## A.4 struct reloc – 再配置エントリ

再配置可能な識別子 ( 変数や関数など ) にアクセスする命令には、再配置エントリが必要です。オフセットが再配置先の番地に格納される System V の再配置データとは異なり、シンボルのベース アドレスに加えるオフセットは再配置エントリに格納されます。このようにする必要があるのは、マイクロチップ社の再配置はアドレス + オフセットの値をデータ ストリームにフィルするだけでなく、シンプルなコード変更にも使用されるためです。この場所にオフセットを格納する方が簡単明瞭であり、ファイル サイズの増加も少なくすみます。

```
typedef struct reloc
{
   unsigned long r_vaddr;
   unsigned long r_symndx;
   short r_offset;
   unsigned short r_type;
} reloc_t;
```

### A.4.1 unsigned long r_vaddr

参照のアドレス ( 原データの開始位置からのバイト オフセット ) を示します。

### A.4.2 unsigned long r_symndx

シンボル テーブルへのインデックスを示します。

### A.4.3 short r_offset

シンボル r_symndx の番地に加える符号付きオフセットを示します。

### A.4.4 unsigned short r_type

再配置タイプ、処理系で定義された値です。表 A-4 に、再配置タイプの種類と説明、および それぞれの値を示します。

**表 A-4: 再配置タイプ**

| Type | Description | Value |
|---|---|---|
| RELOCT_CALL | CALL instruction (first word only on PIC18) | 1 |
| RELOCT_GOTO | GOTO instruction (first word only on PIC18) | 2 |
| RELOCT_HIGH | Second 8 bits of an address | 3 |
| RELOCT_LOW | Low order 8 bits of an address | 4 |
| RELOCT_P | 5 bits of address for the P operand of a PIC17 MOVFP or MOVPF instruction | 5 |
| RELOCT_BANKSEL | Generate the appropriate instruction to bank switch for a symbol | 6 |
| RELOCT_PAGESEL | Generate the appropriate instruction to page switch for a symbol | 7 |
| RELOCT_ALL | 16 bits of an address | 8 |
| RELOCT_IBANKSEL | Generate indirect bank selecting instructions | 9 |
| RELOCT_F | 8 bits of address for the F operand of a PIC17 MOVFP or MOVPF instruction | 10 |
| RELOCT_TRIS | File register address for TRIS instruction | 11 |
| RELOCT_MOVLR | MOVLR bank PIC17 banking instruction | 12 |
| RELOCT_MOVLB | MOVLB PIC17 and PIC18 banking instruction | 13 |
| RELOCT_GOTO2 | Second word of an PIC18 GOTO instruction | 14 |
| RELOCT_CALL2 | Second word of an PIC18 CALL instruction | 14 |
| RELOCT_FF1 | Source register of the PIC18 MOVFF instruction | 15 |
| RELOCT_FF2 | Destination register of the PIC18 MOVFF instruction | 16 |
| RELOCT_SF2 | Destination register of the PIC18 MOVSF instruction | 16 |
| RELOCT_LFSR1 | First word of the PIC18 LFSR instruction | 17 |
| RELOCT_LFSR2 | Second word of the PIC18 LFSR instruction | 18 |
| RELOCT_BRA | PIC18 BRA instruction | 19 |
| RELOCT_RCALL | PIC18 RCALL instruction | 19 |
| RELOCT_CONDBRA | PIC18 relative conditional branch instructions | 20 |
| RELOCT_UPPER | Highest order 8 bits of a 24-bit address | 21 |
| RELOCT_ACCESS | PIC18 access bit | 22 |
| RELOCT_PAGESEL_WREG | Selecting the correct page using WREG as scratch | 23 |
| RELOCT_PAGESEL_BITS | Selecting the correct page using bit set/clear instructions | 24 |
| RELOCT_SCNSZ_LOW<br>RELOCT_SCNSZ_HIGH<br>RELOCT_SCNSZ_UPPER | Size of a section | 25<br>26<br>27 |
| RELOCT_SCNEND_LOW<br>RELOCT_SCNEND_HIGH<br>RELOCT_SCNEND_UPPER | Address of the end of a section | 28<br>29<br>30 |
| RELOCT_SCNEND_LFSR1<br>RELOCT_SCNEND_LFSR2 | Address of the end of a section on LFSR | 31<br>32 |
| RELOCT_TRIS_4BIT | File register address for 4-bit TRIS instruction | 33 |

## A.5 struct syment – シンボル テーブル エントリ

すべての識別子、セクション、関数の開始、関数の終了、ブロックの開始、ブロックの終了に対して、シンボルが作成されます。

```
#define SYMNMLEN 8
struct syment
{
    union
    {
      char _n_name[SYMNMLEN];
      struct
      {
        unsigned long _n_zeroes;
        unsigned long _n_offset;
      } _n_n;
      char *_n_nptr[2];
    } _n;

    unsigned long n_value;
    short n_scnum;
    unsigned long n_type;
    char n_sclass;
    unsigned char n_numaux;
}
```

### A.5.1 union _n

シンボル名は文字列として直接格納されるか、文字列テーブルへの参照として使用されます。8 文字未満のシンボル名はこの共用体に格納され、それ以上のシンボル名はすべて文字列テーブルに格納されます。データ構造がセクション名をシンボル テーブルに格納するように拡張されているのは、この構造がヒントになっています。

#### A.5.1.1 char _n_name [SYMNMLEN]

インプレースのシンボル名 ( 長さが 8 文字未満の場合 ) を示します。

#### A.5.1.2 struct _n_n

シンボル名は文字列テーブルに格納されます。シンボル名の最初の 4 文字がゼロの場合、最後の 4 文字を文字列テーブルへのオフセットとして使用して、シンボル名を検索します。

##### A.5.1.2.1 unsigned long _n_zeros

シンボル名の最初の 4 文字がゼロであることを示します。

##### A.5.1.2.2 unsigned long _n_offset

文字列テーブルに格納されたシンボル名のオフセットを示します。

#### A.5.1.3 char *_n_nptr

オーバーレイを許可します。

### A.5.2 unsigned long n_value

シンボルの値を示します。通常、これはシンボルが存在するセクション内でのシンボルのアドレスです。リンク時の定数 ( マイクロチップ社のシンボル _stksize など ) については、値はアドレスではなくリテラル値となります。一般に、リンカはこの違いを区別せず、アプリケーション コードにおける使用方法のみ異なります。

### A.5.3 short n_scnum

このシンボルが配置されているセクション番号を参照します。

### A.5.4 unsigned long n_type

基本型と派生型を示します。

#### A.5.4.1 シンボルの型

表 A-5 に、基本型とその説明、およびそれぞれの値を示します。

**表 A-5: シンボルの基本型**

| Type | Description | Value |
|---|---|---|
| T_NULL | null | 0 |
| T_VOID | void | 1 |
| T_CHAR | character | 2 |
| T_SHORT | short integer | 3 |
| T_INT | integer | 4 |
| T_LONG | long integer | 5 |
| T_FLOAT | floating point | 6 |
| T_DOUBLE | double length floating point | 7 |
| T_STRUCT | structure | 8 |
| T_UNION | union | 9 |
| T_ENUM | enumeration | 10 |
| T_MOE | member of enumeration | 11 |
| T_UCHAR | unsigned character | 12 |
| T_USHORT | unsigned short | 13 |
| T_UINT | unsigned integer | 14 |
| T_ULONG | unsigned long | 15 |
| T_LNGDBL | long double | 16 |
| T_SLONG | short long | 17 |
| T_USLONG | unsigned short long | 18 |

#### A.5.4.2 派生型

ポインタ、配列、関数は派生型を介して扱われます。表 A-6 に、派生型とその説明、およびそれぞれの値を示します。

**表 A-6: 派生型**

| Derived Type | Description | Value |
|---|---|---|
| DT_NON | no derived type | 0 |
| DT_RAMPTR | pointer to data memory | 1 |
| DT_FCN | function | 2 |
| DT_ARY | array | 3 |
| DT_ROMPTR | pointer to program memory | 4 |
| DT_FARROMPTR | far (24 bit) pointer to program memory | 5 |

### A.5.5 char n_sclass

シンボルのストレージ クラスです。表 A-7 に、ストレージ クラスとその説明、およびそれぞれの値を示します。

**表 A-7: ストレージ クラス**

| Storage Class | Description | Value |
|---|---|---|
| C_EFCN | Physical end of function | 0xFF |
| C_NULL | Null | 0 |
| C_AUTO | Automatic variable | 1 |
| C_EXT | External symbol | 2 |
| C_STAT | Static | 3 |
| C_REG | Register variable | 4 |
| C_EXTDEF | External definition | 5 |
| C_LABEL | Label | 6 |
| C_ULABEL | Undefined label | 7 |
| C_MOS | Member of structure | 8 |
| C_ARG | Function argument | 9 |
| C_STRTAG | Structure tag | 10 |
| C_MOU | Member of union | 11 |
| C_UNTAG | Union tag | 12 |
| C_TPDEF | Type definition | 13 |
| C_USTATIC | Undefined static | 14 |
| C_ENTAG | Enumeration tag | 14 |
| C_MOE | Member of enumeration | 16 |
| C_REGPARM | Register parameter | 17 |
| C_FIELD | Bit field | 18 |
| C_AUTOARG | Automatic argument | 19 |
| C_LASTENT | Dummy entry (end of block) | 20 |
| C_BLOCK | “bb” or “eb” | 100 |
| C_FCN | “bf” or “ef” | 101 |
| C_EOS | End of structure | 102 |
| C_FILE | File name | 103 |
| C_LINE | Line number reformatted as symbol table entry | 104 |
| C_ALIAS | Duplicate tag | 105 |
| C_HIDDEN | External symbol in dmert public library | 106 |
| C_EOF | End of file | 107 |
| C_LIST | Absolute listing on or off | 108 |
| C_SECTION | Section | 109 |

### A.5.6 unsigned char n_numaux

このシンボルの補助エントリ数を示します。

## A.6 struct coff_lineno – 行番号エントリ

実行可能なソース コードのすべての行に対して、そのセクションに関連付けられた行番号テーブル内に coff_lineno エントリが作成されます。すなわち、PICmicro MCU の COFF ファイルでは、すべての命令に coff_lineno エントリが存在することになります。これは、デバッグ情報が絶対リスティング ファイルによるデバッグで利用されることが多いためです。多くの場合、COFF ファイルにはすべての命令に対するエントリが存在しますが、例外もあります。この情報は、System V のフォーマットとは大きく異なります。

```
struct coff_lineno
{
   unsigned long l_srcndx;
   unsigned short l_lnno;
   unsigned long l_paddr;
   unsigned short l_flags;
   unsigned long l_fcnndx;
} coff_lineno_t;
```

### A.6.1 unsigned long l_srcndx

関連するソース ファイルのシンボル テーブル インデックスを示します。

### A.6.2 unsigned short l_lnno

行番号を示します。

### A.6.3 unsigned long l_paddr

この行番号エントリに対応するコードのアドレスを示します。

### A.6.4 unsigned short l_flags

行番号エントリに対するビット フラグを示します。表 A-8 に、ビット フラグとその説明、およびその値を示します。

**表 A-8: 行番号エントリのフラグ**

| Flag | Description | Value |
|---|---|---|
| LINENO_HASFCN | Set if l_fcndx is valid | 0x01 |

### A.6.5 unsigned long l_fcnndx

関連する関数が存在する場合、そのシンボル テーブル インデックスを示します。

## A.7 struct aux_file – ソース ファイルの補助シンボル テーブル エントリ

```
typedef struct aux_file
{
   unsigned long x_offset;
   unsigned long x_incline;
   unsigned char x_flags;
   char _unused[11];
} aux_file_t;
```

### A.7.1 unsigned long x_offset

ファイル名の文字列テーブル オフセットを示します。

### A.7.2 unsigned long x_incline

このファイルがインクルードされている行番号を示します。0 の場合、ファイルはインクルードされていません。

### A.7.3 unsigned char x_flags

.file エントリに対するビット フラグです。表 A-9 に、ビット フラグとその説明、およびその値を示します。

**表 A-9: .file エントリのフラグ**

| Flag | Description | Value |
|---|---|---|
| X_FILE_DEBUG_ONLY | This .file entry was included for debugging purposes only. | 0x01 |

## A.8 struct aux_scn – セクションの補助シンボル テーブル エントリ

```
typedef struct aux_scn
{
   unsigned long x_scnlen;
   unsigned short x_nreloc;
   unsigned short x_nlinno;
   char _unused[12];
} aux_scn_t;
```

### A.8.1 unsigned long x_scnlen

セクションの長さを示します。

### A.8.2 unsigned short x_nreloc

再配置エントリ数を示します。

### A.8.3 unsigned short x_nlinno

行番号数を示します。

## A.9 struct aux_tag – struct/union/enum タグ名の補助シンボル テーブル エントリ

```
typedef struct aux_tag
{
   char _unused[6];
   unsigned short x_size;
   char _unused2[4];
   unsigned long x_endndx;
   char _unused3[4];
} aux_tag_t;
```

### A.9.1 unsigned short x_size

構造体、共用体、または列挙型のサイズを示します。

### A.9.2 unsigned long x_endndx

この構造体タグ、共用体タグ、または列挙型タグの次にあるエントリのシンボル インデックスを示します。

## A.10 struct aux_eos – struct/union/enum の終了を表す補助シンボル テーブル エントリ

```
typedef struct aux_eos
{
   unsigned long x_tagndx;
   char _unused[2];
   unsigned short x_size;
   char _unused2[12];
} aux_eos_t;
```

### A.10.1 unsigned long x_tagndx

構造体、共用体、または列挙型タグのシンボル インデックスを示します。

### A.10.2 unsigned short x_size

構造体、共用体、または列挙型のサイズを示します。

## A.11 struct aux_fcn – 関数名の補助シンボル テーブル エントリ

```
typedef struct aux_fcn
{
   unsigned long x_tagndx;
   unsigned long x_size;
   unsigned long x_lnnoptr;
   unsigned long x_endndx;
   short x_actscnum;
   char _unused[2];
} aux_fcn_t;
```

### A.11.1 unsigned long x_tagndx

戻り値の型に関連付けられた、構造体タグ名または共用体タグ名のシンボル テーブル インデックス ( 戻り値の基本型が構造体または共用体の場合 ) を示します。

### A.11.2 unsigned long x_lnnoptr

この関数の行番号へのファイル ポインタを示します。

### A.11.3 unsigned long x_endndx

この関数の次のエントリのシンボル インデックスを示します。

### A.11.4 short x_actscnum

静的有効化レコード データのセクション番号を示します。

## A.12 struct aux_fcn_calls – 関数呼び出し参照の補助シンボル テーブル エントリ

```
typedef struct aux_fcn_calls
{
   unsigned long x_calleendx;
   unsigned long x_is_interrupt;
   char _unused[12];
} aux_fcn_calls_t;
```

### A.12.1 unsigned long x_calleendx

呼び出し先関数のシンボル インデックスを示します。高位関数の呼び出しの場合、AUX_FCN_CALLS_HIGHERORDER にセットされます。

```
#define AUX_FCN_CALLS_HIGHERORDER ((unsigned long)-1)
```

### A.12.2 unsigned long x_is_interrupt

関数が割り込みかどうか、割り込みの場合はその優先度を指定します。

0: 割り込みしない

1: 低優先度の割り込み

2: 高優先度の割り込み

## A.13 struct aux_arr – 配列の補助シンボル テーブル エントリ

```
#define X_DIMNUM 4
typedef struct aux_arr
{
   unsigned long x_tagndx;
   unsigned short x_lnno;
   unsigned short x_size;
   unsigned short x_dimen[X_DIMNUM];
   char _unused[4];
} aux_arr_t;
```

### A.13.1 unsigned long x_tagndx

配列の要素の型に関連付けられた、構造体タグ名または共用体タグ名のシンボル テーブル インデックス ( 基本型が構造体または共用体の場合 ) を示します。

### A.13.2 unsigned short x_size

配列のサイズを示します。

### A.13.3 unsigned short x_dimen[X_DIMNUM]

最初の 4 つの次元のサイズを示します。

## A.14 struct aux_eobf – ブロックまたは関数の終了を示す補助シンボル テーブル エントリ

```
typedef struct aux_eobf
{
   char _unused[4];
   unsigned short x_lnno;
   char _unused2[14];
} aux_eobf_t;
```

### A.14.1 unsigned short x_lnno

ブロック / 関数の開始行からの相対的終了行の C ソース行番号を示します。

## A.15 struct aux_bobf – ブロックまたは関数の開始を表す補助シンボル テーブル エントリ

```
typedef struct aux_bobf
{
   char _unused[4];
   unsigned short x_lnno;
   char _unused2[6];
   unsigned long x_endndx;
   char _unused3[4];
} aux_bobf_t;
```

### A.15.1 unsigned short x_lnno

開始行の C ソース行番号を示します。開始行を含むスコープを基準とします。

### A.15.2 unsigned long x_endndx

このブロック / 関数の次のエントリのシンボル インデックスを示します。

## A.16 struct aux_var – struct/union/enum 型の変数の補助シンボル テーブル エントリ

```
typedef struct aux_var
{
   unsigned long x_tagndx;
   char _unused[2];
   unsigned short x_size;
   char _unused2[12];
} aux_var_t;
```

### A.16.1 unsigned long x_tagndx

構造体、共用体、または列挙型タグのシンボル インデックスを示します。

### A.16.2 unsigned short x_size

構造体、共用体、または列挙型のサイズを示します。

## A.17 struct aux_field – ビット フィールドの補助エントリ

```
typedef struct aux_field
{
   char _unused[6];
   unsigned short x_size;
   char _unused2[12];
} aux_field_t;
```

### A.17.1 unsigned short x_size

ビット フィールドのサイズ ( 単位 : ビット ) を示します。

ノート :

# 付録 B ANSI 処理系で定義されている動作

## B.1 はじめに

ここでは、MPLAB C18 の実装で定義されている動作について説明します。ISO C 規格では、ベンダーは C 言語の「処理系定義」固有の機能について文書化を義務づけています。

> **注 :** (6.1.2) のように括弧内に示した項番は、ANSI C 規格 X3.159-1989 の項番を示しています。

次に示す処理系定義の動作は、ANSI C 規格の G.3 節に記載されています。

## B.2 識別子

**ANSI C 規格 :** 「外部結合なしの識別子の先頭から (31 以上 ) の認識可能な文字数 (6.1.2)」

「外部結合ありの識別子の先頭から (6 以上 ) の認識可能な文字数 (6.1.2)」

「外部結合ありの識別子での大文字と小文字の区別の有無 (6.1.2)」

**実装 :** MPLAB C18 では、すべての識別子で認識可能な文字数は 31 以上です。外部結合ありの識別子では大文字と小文字が区別されます。

## B.3 文字

**ANSI C 規格 :** 「複数の文字を含む整数文字定数の値、または複数のマルチバイト文字を含むワイド文字定数の値 (6.1.3.4)」

**実装 :** 整数文字定数の値は、最初の文字の 8 ビット値です。ワイド文字はサポートしていません。

**ANSI C 規格 :** 「符号指定なしの `char` の値の範囲は、`signed char` として扱うか、`unsigned char` として扱うか (6.2.1.1)」

**実装 :** 符号指定なしの `char` は、`signed char` と同じ値の範囲となります。MPLAB C18 では、これをコマンドライン オプション (`-k`) で `unsigned char` に変更できます。

## B.4 整数

**ANSI C 規格 :** 「int または unsigned int を使用できる場合は、常に char、short int、int ビット フィールド ( 符号付きと符号なしのいずれも )、または列挙型 を式で使用できる。元の型のすべての値を int で表現できる場合は、その値は int に変換され、それ以外の場合は unsigned int に変換される。これを*汎整数拡張*と呼ぶ。その他の算術型は、汎整数拡張によって変換されることはない。

汎整数拡張では、符号を含めた値がそのまま維持される (6.2.1.1)」

**実装 :** MPLAB C18 では、デフォルトでは汎整数拡張を実行しません。ANSI で定義されている動作を実行するには、コマンドライン オプション -Oi を指定します。**2.7.1 項「汎整数拡張」**を参照してください。

**ANSI C 規格 :** 「整数をより短い符号付き整数に変換した結果、または符号なし整数を同じ長さの符号付き整数に変換した結果、値を表現できない場合 (6.2.1.2)」

**実装 :** 整数型をより短い 整数型に変換すると、値の上位ビットは破棄され、残りのビットは 短い整数型に従って解釈されます。符号なし整数を同じサイズの符号付き整数に変換した場合、符号なし整数のビットは、そのサイズの符号付き整数の規則に従ってそのまま解釈されます。

**ANSI C 規格 :** 「符号付き整数に対するビット単位の演算の結果 (6.3)」

**実装 :** 符号付き整数に対するビット単位の演算子が、同じ型の符号なし整数に対する場合と同様に適用されます ( すなわち、符号ビットも通常のビットと同じ扱いとなります )。

**ANSI C 規格 :** 「整数除算の剰余の符号 (6.3.5)」

**実装 :** 剰余の符号は商の符号と同じです。

**ANSI C 規格 :** 「負の値の符号付き整数型を右シフトした結果 (6.3.7)」

**実装 :** この値は、同じサイズの符号なし整数型の場合と同様にシフトされます ( すなわち、符号ビットは伝播しません )。

## B.5 浮動小数点

**ANSI C 規格 :** 「浮動小数点の型による表現と値の範囲の違い (6.1.2.5)」

「整数値を元の値を正確に表現できない浮動小数点数に変換した場合の切り捨ての方向 (6.2.1.3)」

「浮動小数点数をより小さな浮動小数点数に変換した場合の切り捨てまたは丸めの方向 (6.2.1.4)」

**実装 :** **2.1.2 項「浮動小数点型」**を参照してください。

最近接丸めが実行されます。

## B.6 配列およびポインタ

**ANSI C 規格 :** 「配列の最大サイズを維持するのに必要な整数型、すなわち sizeof 演算子の型 size_t (6.3.3.4, 7.1.1)」

**実装 :** size_t は、unsigned short long int と定義されています。

**ANSI C 規格 :** 「ポインタの整数へのキャスト、またはその逆の結果 (6.3.4)」

**実装 :** 整数には、ポインタを表現する 2 進数値が含まれます。ポインタ型の値が整数型よりも大きい場合は、整数型におさまるように切り捨てられます。

**ANSI C 規格 :** 「同じ配列内の要素への 2 つのポインタ間の差を維持するために必要な整数型 ptrdiff_t (6.3.6, 7.1.1)」

**実装 :** ptrdiff_t は unsigned long short と定義されています。

## B.7 レジスタ

**ANSI C 規格 :** 「レジスタ ストレージ クラス指定子を使用することによって、オブジェクトをレジスタの中に実際に配置できる範囲 (6.5.1)」

**実装 :** レジスタ ストレージ クラス指定子は無視されます。

## B.8 構造体および共用体

**ANSI C 規格 :** 「共用体オブジェクトのメンバーに、別の型のメンバーを使用してアクセスする場合 (6.3.2.3)」

**実装 :** メンバーの値は、アクセスされるメンバーの型であると解釈されたメンバーのアドレスにあるビットとなります。

**ANSI C 規格 :** 「構造体のメンバーのパディングおよび整列方法 (6.5.2.1)」

**実装 :** 構造体メンバーおよび共用体メンバーは、バイト境界にアライメントされます。

## B.9 ビット フィールド

**ANSI C 規格 :** 「「符号指定のない」int 型のビット フィールドを signed int のビット フィールドとして扱うか、unsigned int のビット フィールドとして扱うか (6.5.2.1)」

**実装 :** 「符号指定のない」int 型のビット フィールドは、signed int のビット フィールドとして扱われます。

**ANSI C 規格 :** 「1 ユニットでのビット フィールドの割り当て順序 (6.5.2.1)」

**実装 :** ビット フィールドは、順に最下位ビットから最上位ビットへ割り当てられます。

**ANSI C 規格 :** 「ビット フィールドがストレージ ユニットの境界をまたぐことは可能か (3.5.2.1)」

**実装 :** ビット フィールドはストレージ ユニットの境界をまたぐことはできません。

## B.10 列挙型

**ANSI C 規格 :** 「列挙型の値を表現するために使用される整数型 (6.5.2.2)」

**実装 :** 列挙型のすべての値を表現できる最小の型を使用します。

## B.11 SWITCH 文

**ANSI C 規格 :** 「switch 文における case の値の最大数 (6.6.4.2)」

**実装 :** 値の最大数に制限はありません(ターゲット メモリのみに依存)。

## B.12 プリプロセッサ ディレクティブ

**ANSI C 規格 :** 「インクルード可能なソース ファイルの指定方法 (6.8.2)」

**実装 :** **2.5.1 項「システム ヘッダ ファイル」**を参照してください。

**ANSI C 規格 :** 「インクルード可能なソース ファイルの引用名のサポート (6.8.2)」

**実装 :** **2.5.2 項「ユーザー ヘッダ ファイル」**を参照してください。

**ANSI C 規格 :** 「認識可能な #pragma ディレクティブでの動作 (6.8.6)」

**実装 :** **2.9 項「プラグマ」**を参照してください。

# 付録 C コマンドライン オプション一覧

使用方法 : `mcc18 [options] file [options]`

**表 C-1: コマンドライン オプション一覧**

| Option | Description | Reference |
|---|---|---|
| `-?, --help` | Displays the help screen | 1.2 |
| `-I=<path>` | Add 'path' to include path | 2.5.1, 2.5.2 |
| `-fo=<name>` | Object file name | 1.2.1 |
| `-fe=<name>` | Error file name | 1.2.1 |
| `-k` | Set plain char type to unsigned char | 2.1 |
| `-ls` | Large stack (can span multiple banks) | 3.2.4 |
| `-ms` | Set compiler memory model to small model (default) | 2.6, 3.1 |
| `-ml` | Set compiler memory model to large model | 2.6, 3.1 |
| `-O, -O+` | Enable all optimizations (default) | 4 |
| `-O-` | Disable all optimizations | 4 |
| `-Od+` | Enable dead code removal (default) | 4.10 |
| `-Od-` | Disable dead code removal | 4.10 |
| `-Oi+` | Enable integer promotion | 2.7.1 |
| `-Oi-` | Disable integer promotion (default) | 2.7.1 |
| `-Om+` | Enable duplicate string merging (default) | 4.1 |
| `-Om-` | Disable duplicate string merging | 4.1 |
| `-On+` | Enable banking optimizer (default) | 4.3 |
| `-On-` | Disable banking optimizer | 4.3 |
| `-Op+` | Enable copy propagation (default) | 4.8, 4.10 |
| `-Op-` | Disable copy propagation | 4.8, 4.10 |
| `-Or+` | Enable redundant store elimination (default) | 4.9 |
| `-Or-` | Disable redundant store elimination | 4.9 |
| `-Ou+` | Enable unreachable code removal (default) | 4.7 |
| `-Ou-` | Disable unreachable code removal | 4.7 |
| `-Os+` | Enable code straightening (default) | 4.5 |
| `-Os-` | Disable code straightening | 4.5 |
| `-Ot+` | Enable tail merging (default) | 4.6 |
| `-Ot-` | Disable tail merging | 4.6 |
| `-Ob+` | Enable branch optimizations (default) | 4.2 |
| `-Ob-` | Disable branch optimizations | 4.2 |
| `-sca` | Enable default auto locals (default). Valid for Non-Extended mode only. | 2.3 |
| `-scs` | Enable default static locals. Valid for Non-Extended mode only. | 2.3 |
| `-sco` | Enable default overlay locals (statically allocate activation records). Valid for Non-Extended mode only. | 2.3 |

**表 C-1: コマンドライン オプション一覧 ( 続き )**

| Option | Description | Reference |
|---|---|---|
| -Oa+ | Enable default data in access memory. Valid for Non-Extended mode only. | 2.9.1.3 |
| -Oa- | Disable default data in access memory (default). Valid for Non-Extended mode only. | 2.9.1.3 |
| -Ow+ | Enable WREG tracking (default) | 4.4 |
| -Ow- | Disable WREG tracking | 4.4 |
| -Opa+ | Enable procedural abstraction (default) | 4.11 |
| -Opa- | Disable procedural abstraction | 4.11 |
| -pa=<repeat count> | Set procedural abstraction repeat count (default = 4) | 4.11 |
| -p=<processor> | Set processor (default is generic) | 1.2.4, 2.6, 2.10 |
| -D<macro>[=text] | Define a macro | 1.2.3 |
| -w={1\|2\|3} | Set warning level (default = 2) | 1.2.2 |
| -nw=<n> | Suppress message <n> | 1.2.2 |
| -v | Display version number and exit. | |
| -verbose | Operate verbosely (show banner and other information) | 1.2 |
| --extended | Generate Extended mode code. | 1.2.5 |
| --no-extended | Generate Non-Extended mode code. | 1.2.5 |
| --help-message-list | Display a list of all diagnostic messages | 1.2.2 |
| --help-message-all | Display help for all diagnostic messages | 1.2.2 |
| --help-message=<n> | Display help on diagnostic number <n> | 1.2.2 |
| --help-config | Display help on device-specific configuration settings | 2.9.5 |

# 付録 D MPLAB C18 メッセージ一覧

ここでは、MPLAB C18 コンパイラから出力されるエラー、警告、メッセージを一覧にまとめます。

## D.1 エラー

1000: %s

1002: syntax error, '%s' expected

プリプロセッサの構文に、トークンが不足しています。通常はタイプミス、ディレクティブに必要なオペランドの不足、括弧の不一致などが原因として考えられます。

1013: error in pragma directive

MPLAB C18 がプラグマの構文解析中に、新規の行が見つかりませんでした。プラグマの後に余分なテキストがある場合に発生します。

1014: redundant attribute specifier declaring section '%s'

#pragma sectiontype ディレクティブで、overlay または access 属性が複数回指定されています。

1016: integer constant expected for #line directive

#line プリプロセッサ ディレクティブの行番号オペランドは、整定数である必要があります。

1017: symbol name expected in 'interrupt' pragma

save= 節には、静的に割り当てられたスコープ内のシンボルのうち、割り込み関数で保存および復元するものの名前を、カンマで区切って列記する必要があります。通常は、現在スコープ内にないシンボルを指定している、参照先シンボルを宣言しているヘッダ ファイルをインクルードしていない、シンボル名のタイプミスなどが原因として考えられます。

1018: function name expected in 'interrupt' pragma

interrupt プラグマでは、割り込みとして宣言する関数の名前は、最初のパラメータとして指定する必要があります。関数のシンボルは現在のスコープ内にある必要があり、パラメータも戻り値も持つことはできません。通常は、割り込みとして宣言した関数のプロトタイプがない、またはタイプミスが原因として考えられます。

1019: '%s' is a compiler managed resource - it should not appear in a save= list

割り込み宣言の save= 節で、無効なシンボル名を指定しています。一部のアドレスは、save= リストで指定して保存 / 復元を実行すると、不正なコードとなります。これらのアドレスは特別なコンテクストの保存を必要としないので、save= リストから正しく削除すればエラーが解消されます。

1020: unexpected input following '%s'

プリプロセッサ構文に余分な情報があります。

1021: unterminated comment

C 言語形式のコメント「/*」が正しく終了していません。エラー メッセージには、該当するコメントの開始行が表示されます。

1022: end of file in argument '%s' for macro '%s'

指定されたマクロの指定された引き数を処理中に、EOF が検出されました。通常は、括弧の不足が原因として考えられます。

1023: end of file in valist argument for macro '%s'

指定されたマクロの変数引き数を処理中に、EOF が検出されました。通常は、括弧の不足が原因として考えられます。

1024: macro '%s' expects %d arguments, but only %d found

マクロで指定した引き数の数が不正です。マクロを使用するには、渡した引き数の数とそのマクロに定義された引き数の数が、完全に一致している必要があります。

1025: missing '%c' in header name

#include 文のヘッダ ファイル名を処理中に EOF が検出されました。通常は、#include ディレクティブの行が正しく終了していないことが原因として考えられます。

1026: malformed #include directive

#include の後に「"」または「<」以外のものが検出されました。通常は、ディレクティブのタイプミスが原因として考えられます。

1027: unable to locate '%s'

指定したヘッダ ファイルが、インクルード ファイルの検索パスに見つかりませんでした ( システム ヘッダ ファイルまたはユーザー ヘッダ ファイルのいずれでも )。コマンドライン オプション -I が正しく指定されているかどうか確認してください。この他、ヘッダファイルのタイプミスやアクセス権限が不十分なことが原因として考えられます。

1028: %s without matching #if

一致する #if のないプリプロセッサ ディレクティブが検出されました。通常は、入れ子の不一致またはスペルミスが原因として考えられます。

1029: malformed expression in '%s'

プリプロセッサ ディレクティブの式が不正です。通常は、括弧の不一致またはスペルミスが原因として考えられます。

1030: identifier expected in %s

プリプロセッサ ディレクティブで識別子が必要なところで、C の識別子が検出されませんでした。通常は、識別子のタイプミスが原因として考えられます。

1031: '%c' expected in 'defined'

'defined' プリプロセッサ ディレクティブの後には、括弧または識別子のいずれかが必要です。通常は、括弧がないか識別子のタイプミスが原因として考えられます。

1032: ')' expected in expansion of macro '%s'

マクロを展開する際は、閉じ括弧が必要です。通常は、括弧の不足が原因として考えられます。

1033: preprocessor can only input one file at a time

プリプロセッサの入力で扱えるソース ファイルは 1 つのみです。通常は、コンパイラの実行ファイルがプリプロセッサの起動に失敗したことが原因として考えられます。プリプロセッサを個別に起動する場合は、コマンドラインを訂正してください。

1034: previous definition of macro '%s' does not agree

ANSI 規格では、オブジェクト形式マクロとして現在定義されている識別子を別の #define プリプロセッサ ディレクティブで再定義できるのは、後者の定義がオブジェクト形式マクロの定義であり、2 つの置換リストが同一である場合に限られます。同様に、関数形式マクロとして現在定義されている識別子を別の #define プリプロセッサ ディレクティブで再定義できるのは、後者の関数形式マクロの定義もパラメータの数とスペルが同じであり、2 つの置換リストが同一である場合に限られます。

1035: expecting macro name, received '%s' instead

必要な C 識別子が見つかりませんでした。通常は、識別子のタイプミスが原因として考えられます。

1036: syntax error in macro argument list, expecting ')'

変数引き数リスト (...) の最後に閉じ括弧がありません。

1037: duplicate parameter name '%s' in macro '%s'

マクロのパラメータ名は一意である必要があります。

1038: syntax error in macro argument list

引き数リストに必要なカンマがないか、または変数引き数リスト (...) に必要な閉じ括弧が見つかりません。

1039: illegal character in macro name '%c'

マクロ名の後にスペースまたは開き括弧が見つかりません。通常は、マクロ名のタイプミスが原因として考えられます。

1040: # or ## operator found in simple macro %s

プリプロセッサの文字列展開演算子 (#) とトークン結合演算子 (##) は、関数形式マクロの引き数と共に使用する必要があります。

1041: # operator requires a parameter name as operand

プリプロセッサの文字列展開演算子 (#) にはオペランドとしてパラメータ名が必要ですが、C 識別子が見つかりません。通常は、識別子のタイプミスが原因として考えられます。

1042: filename for %s directive exceeds maximum filename length

プリプロセッサ ディレクティブで指定したファイルの名前が、MAX_FILENAME_PATH_LEN のファイル名の長さの制限を超えています。

1050: section address permitted only at definition

#pragma sectiontype ディレクティブの location 節で絶対アドレスを指定する際は、このセクションを最初に定義するプラグマでのみ指定できます。

1052: section overlay attribute does not match definition

MPLAB C18 では、以前に宣言されたセクションの属性と、現在の #pragma sectiontype ディレクティブで指定する属性とが一致している必要があります。

1053: section share attribute does not match definition

MPLAB C18 では、以前に宣言されたセクションの属性と、現在の #pragma sectiontype ディレクティブで指定する属性とが一致している必要があります。

1054: section type does not match definition

このセクション名は、MPLAB C18 によって既に別の型 (code、idata、udata、romdata) で検出されています。

1055: section access attribute does not match definition

MPLAB C18 では、以前に宣言されたセクションの属性と、現在の #pragma sectiontype ディレクティブで指定する属性とが一致している必要があります。

1070: too many line numbers in section '%s'

COFF ファイル フォーマットでは、1 つのセクションに (32767 * 2 + 1) 行まで含められます。ソース ファイルの行数を減らしてください。

1071: too many relocations in section '%s'

COFF ファイル フォーマットでは、1 つのセクションに (32767 * 2 + 1) リロケーションまで含められます。ソース ファイルの変数の参照数を減らしてください。

1072: too many function calls for ISR '%s'

1 つの ISR から呼び出せる関数は 253 種類までです。出力オブジェクト ファイル フォーマット (COFF) では、補助エントリ数が 255 に制限されています。ISR は 2 つの補助エントリを必要とし、1 種類の関数の呼び出しにも 1 つの補助エントリが必要です。

1073: too many function calls for '%s'

割り込み以外の関数から呼び出せる関数は 254 種類までです。出力オブジェクト ファイル フォーマット (COFF) では、補助エントリ数が 255 に制限されています。割り込み以外の関数は 1 つの補助エントリを必要とし、1 種類の関数の呼び出しにも 1 つの補助エントリが必要です。

1099: %s

ソース コードの #error ディレクティブのメッセージ。

1100: syntax error

関数の型定義が不正です。

1101: lvalue required

オブジェクトを指定する式が必要です。通常は、括弧の不足や「*」演算子の不足が原因として考えられます。

1102: cannot assign to 'const' modified object

const 修飾子を付けたオブジェクトは読み取り専用として宣言されており、変更はできません。

1103: unknown escape sequence '%s'

コンパイラで不明のエスケープ シーケンスです。有効なエスケープ シーケンス文字を ANSI 規格で確認してください。

1104: division by zero in constant expression

コンパイラは、ゼロ除算 ( または剰余を求めるゼロ除算 ) を含む定数式を処理できません。

1105: symbol '%s' has not been defined

定義する前にシンボルが参照されています。通常は、シンボル名のスペルミス、シンボルを宣言しているヘッダ ファイルがない、内部スコープでのみ有効なシンボルへの参照などが原因として考えられます。

1106: '%s' is not a function

関数名以外のシンボルは、割り込み関数として宣言できません。

1107: interrupt functions must not take parameters

プロセッサが割り込みルーチンにジャンプする際にパラメータ渡しは実行されないため、割り込み関数はパラメータなしで宣言する必要があります。

1108: interrupt functions must not return a value

割り込みはプロセッサによって非同期に呼び出されるため、戻り値を返すための呼び出し元ルーチンは存在しません。

1109: type mismatch in redeclaration of '%s'

宣言されているシンボルの型が、以前に宣言された同じシンボルの型と互換性がありません。通常は、修飾子がないか位置が不正であることが原因として考えられます。

1111: undefined label '%s' in '%s'

goto 文で参照されているラベルが関数で定義されていません。通常は、ラベル識別子のスペルミス、スコープ外のラベル ( 別の関数で定義されたラベル ) への参照などが原因として考えられます。

1112: integer type expected in switch control expression

switch 文の制御式は整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1113: integer constant expected for case label value

case ラベルの値は整定数である必要があります。

1114: case label outside switch statement detected

case ラベルの有効範囲は、switch 文の本体内部に限られます。通常は、「}」の位置が不正であることが原因として考えられます。

1159: default label outside switch statement detected

default ラベルの有効範囲は、switch 文の本体内部に限られます。通常は、「}」の位置が不正であることが原因として考えられます。

1115: multiple default labels in switch statement

1 つの switch 文に含めることができる default ラベルは、1 つのみです。通常は、内部の switch 文を閉じる「}」の不足が原因として考えられます。

1116: type mismatch in return statement

戻り値の型が、関数で宣言されている戻り値の型と互換性がありません。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1117: scalar type expected in 'if' statement

if 文の制御式は、スカラ型 ( 整数またはポインタ ) である必要があります。

1118: scalar type expected in 'while' statement

while 文の制御式は、スカラ型 ( 整数またはポインタ ) である必要があります。

1119: scalar type expected in 'do..while' statement

do..while 文の制御式は、スカラ型 ( 整数またはポインタ ) である必要があります。

1120: scalar type expected in 'for' statement

for 文の制御式は、スカラ型 ( 整数またはポインタ ) である必要があります。

1121: scalar type expected in '?:' expression

「?:」演算子の制御式は、スカラ型 ( 整数またはポインタ ) である必要があります。

1122: scalar operand expected for '!' operator

「!」演算子のオペランドは、スカラ型である必要があります。

1123: scalar operands expected for '||' operator

論理和演算子「||」のオペランドは、スカラ型である必要があります。

1124: scalar operands expected for '&&' operator

論理積演算子「&&」のオペランドは、スカラ型である必要があります。

1125: 'break' must appear in a loop or switch statement

break 文は、while、do、for、switch のいずれかの文の中で使用する必要があります。通常は、「}」の位置が不正であることが原因として考えられます。

1126: 'continue' must appear in a loop statement

continue 文は、while、do、for、switch のいずれかの文の中で使用する必要があります。

1127: operand type mismatch in '?:' operator

「?:」演算子の結果オペランドの型は、スカラ型で、かつ互換性のある型である必要があります。

1128: compatible scalar operands required for comparison

比較演算子のオペランドは、互換性のあるスカラ型である必要があります。

1129: [] operator requires a pointer and an integer as operands

配列アクセス演算子「[]」では、一方のオペランドはポインタ、もう一方のオペランドは整数である必要があります。つまり、x[y] の場合には、式 *(x+y) が有効である必要があります。x[y] は、機能的に *(x+y) と等価です。

1130: pointer operand required for '*' operator

間接参照演算子「*」では、オペランドとして void 型以外のオブジェクトへのポインタが必要です。

1131: type mismatch in assignment

代入演算子では、右辺の式の結果と左辺の式の結果は、互換性のある型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1132: integer type expected for right hand operand of '-=' operator

「-=」演算子の左辺がポインタ型の場合、右辺は整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1133: type mismatch in '-=' operator

「-=」演算子のオペランドは、x-=y の場合に式 x=x-y が有効となるような型である必要があります。

1134: arithmetic operands required for multiplication operator

乗算演算子「*」および「*=」のオペランドは、算術型である必要があります。通常は、間接参照演算子「*」や添字演算子「[]」がないことが原因として考えられます。

1135: integer operands required for modulus operator

剰余演算子「%」および「%=」のオペランドは、整数型である必要があります。通常は、間接参照演算子「*」や添字演算子「[]」がないことが原因として考えられます。

1136: integer operands required for shift operator

ビット シフト演算子のオペランドは、整数型である必要があります。通常は、間接参照演算子「*」や添字演算子「[]」がないことが原因として考えられます。

1137: integer types required for bitwise AND operator

「&」演算子と「&=」演算子では、両方のオペランドが整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1138: integer types required for bitwise OR operator

「|」演算子と「|=」演算子では、両方のオペランドが整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1139: integer types required for bitwise XOR operator

「^」演算子と「^=」演算子では、両方のオペランドが整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1140: integer type required for bitwise NOT operator

「~」演算子のオペランドは、整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1141: integer type expected for pointer addition

加算演算子で一方のオペランドがポインタ型の場合、もう一方のオペランドは整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1142: type mismatch in '+' operator

「+」演算子のオペランドは、一方がポインタ型ならもう一方が整数型、または両方のオペランドとも算術型である必要があります。

1143: pointer difference requires pointers to compatible types

2 つのポインタの差を計算する際は、2 つのポインタが互換性のある型のオブジェクトを指し示している必要があります。通常は、括弧の不足や「[]」演算子の不足が原因として考えられます。

1144: integer type required for pointer subtraction

減算演算子の左辺のオペランドがポインタ型の場合、右辺のオペランドは整数型である必要があります。通常は、「*」演算子や「[]」演算子がないことが原因として考えられます。

1145: arithmetic type expected for subtraction operator

減算演算子の左辺のオペランドがポインタ型でない場合は、両方のオペランドが算術型である必要があります。

1146: type mismatch in argument %d

関数呼び出しに対する引き数の型は、対応するパラメータの宣言されている型と互換性がある必要があります。

1147: scalar type expected for increment operator

増分演算子のオペランドは、変更可能なスカラ型の lvalue である必要があります。

1148: scalar type expected for decrement operator

減分演算子のオペランドは、変更可能なスカラ型の lvalue である必要があります。

1149: arithmetic type expected for unary plus

単項プラス演算子のオペランドは、算術型である必要があります。

1150: arithmetic type expected for unary minus

単項マイナス演算子のオペランドは、算術型である必要があります。

1151: struct or union object designator expected

メンバー アクセス演算子「.」のオペランドは構造体 / 共用体、「->」のオペランドは構造体 / 共用体へのポインタである必要があります。

1152: scalar or void type expected for cast

明示的なキャストでは、オペランドの型がスカラ型で、キャストされる型がスカラ型または void 型である必要があります。

1153: cannot assign array type objects

配列型のオブジェクトには、直接代入できません。代入できるのは、配列の要素のみです。

1154: parameter %d in '%s' must have a name

関数の定義では、パラメータに識別子で名前を付ける宣言が必要です。名前の宣言はプロトタイプでは不要ですが、定義では必要です。

1155: 'overlay' symbol '%s' not in function scope

変数は、関数のスコープ内でのみオーバーレイできます。

1156: member '%s' declared as having function type

構造体メンバーと共用体メンバーは、関数型にはできません。通常は、関数ポインタが正しく宣言されていないことが原因として考えられます。

1157: function 'main' should be declared as 'void main (void)'

MPLAB C18 のスタートアップ コードが呼び出す関数 main は、パラメータも戻り値も持つことはできません。関数 main は、必ずパラメータと戻り値なしで宣言する必要があります。

1158: arithmetic operands required for division operator

除算演算子「/」および「/=」のオペランドは、算術型である必要があります。通常は、間接参照演算子「*」や添字演算子「[]」がないことが原因として考えられます。

1160: conflicting storage classes specified

宣言で指定できるストレージ クラスは 1 つのみです。

1161: conflicting base types specified

宣言では、1 つの基本型 (void、int、float など ) のみ指定できます。同じ基本型のインスタンスが複数ある場合もエラーとなります (int int x; など )。

1162: both 'signed' and 'unsigned' specified

1 つの型に指定できるのは、signed と unsigned のいずれか 1 つです。

1163: function must be located in program memory

データ メモリは実行可能でないため、関数はすべてプログラム メモリに配置する必要があります。

1165: reference to incomplete tag '%s'

前方参照の構造体タグまたは共用体タグは、宣言で直接参照できません。宣言できるのは、前方参照タグへのポインタのみです。

1166: invalid type specification

型指定が不正です。通常は、タイプミスや typedef 型の誤用が原因として考えられます ( 例えば、int enum myEnum xyz; は不正な型指定です )。

1168: reference to undefined enumeration tag '%s'

列挙型タグは、他の宣言でそのタグを参照する前に定義しておく必要があります。構造体タグや共用体タグとは異なり、列挙型タグに対する前方参照は実行できません。

1169: anonymous members allowed in unions only

匿名構造体のメンバーは、共用体メンバーとしてのみ宣言できます。

1170: non-integral type bitfield detected

構造体のビットフィールド メンバーは、整数型である必要があります。

1171: bitfield width greater than 8 detected

1 つのビットフィールドは、1 つのストレージ ユニット (MPLAB C18 では 1 バイト ) におさめる必要があります。つまり、1 ビットフィールドに含めることができるのは 8 ビット以下です。

1172: enumeration value of '%s' does not match previous

複数の列挙型タグで同じ列挙型の定数名を使用している場合、列挙型の定数の値は各列挙型で同一である必要があります。

1173: cannot locate a parameter in program memory, '%s'

パラメータはすべてスタックに配置されるため、プログラム メモリ内にはパラメータを配置できません。通常は、プログラム メモリへのポインタの宣言のタイプミスが原因として考えられます。

1174: local '%s' in program memory can not be 'auto'

ローカル変数をプログラム メモリに配置する場合は、static または extern として宣言する必要があります。auto で宣言されたローカル変数はスタックに配置されます。

1175: static parameter detected in function pointer '%s'

関数ポインタでは、パラメータを渡すにはスタックを介する必要があります。静的ローカル変数を有効にしてコンパイルを実行する場合は、関数ポインタのパラメータ、および関数ポインタに番地が割り当てられている関数のパラメータは、明示的に auto として宣言してください。

1176: tag type mismatch in redeclaration of '%s'

構造体、共用体、または列挙型の宣言において、以前宣言された構造体、共用体、または列挙型と名前が同じで型の異なるタグ名を使用しています。ある構造体を共用体として宣言し直した場合や、ある列挙型を構造体として宣言し直した場合などが例として挙げられます。

1177: illegal reference to array of void

配列の要素を void 型にはできません。通常は、型定義にポインタ宣言子「*」がないことが原因として考えられます。例えば、void *array[10]; とするところを void array[10]; とした場合などです。

1178: illegal declaration of object of type void

オブジェクトの型を void 型にはできません。通常は、宣言にポインタ宣言子「*」がないことが原因として考えられます。例えば、void *p; とするところを void p; とした場合などです。

1200: cannot reference the address of a bitfield

構造体のビットフィールド メンバーのアドレスは、直接参照できません。

1201: cannot dereference a pointer to 'void' type

間接参照演算子「*」では、オペランドとして void 型以外のオブジェクトへのポインタが必要です。

1202: call of non-function

関数呼び出しの後置演算子「()」のオペランドは、「関数へのポインタ」型である必要があります。多くの場合、これは関数識別子です。通常は、スコープの括弧の不足が原因として考えられます。

1203: too few arguments in function call

関数を呼び出すには、渡される引き数の数とその関数に宣言されたパラメータの数が、完全に一致している必要があります。

1204: too many arguments in function call

関数を呼び出すには、渡される引き数の数とその関数に宣言されたパラメータの数が、完全に一致している必要があります。

1205: unknown member '%s' in '%s'

構造体タグまたは共用体タグに、要求されている名前のメンバーがありません。通常は、メンバー名のスペルミスや、入れ子の構造体でのメンバーアクセス演算子の不足が原因として考えられます。

1206: unknown member '%s'

構造体または共用体の型に、要求されている名前のメンバーがありません。通常は、メンバー名のスペルミスや、入れ子の構造体でのメンバーアクセス演算子の不足が原因として考えられます。

1207: tag '%s' is incomplete

不完全な構造体タグまたは共用体タグは、メンバー アクセス演算子から参照できません。通常は、シンボル定義における構造体のタグ名のスペルミスが原因として考えられます。

1208: "#pragma interrupt" detected inside function body

interrupt プラグマは、ファイル レベルのスコープでのみ利用できます。

1210: unknown symbol '%s' in interrupt save list

interrupt プラグマの save= リスト内のシンボルは、既に宣言されていること、およびスコープ内にあることが必要です。

1211: missing definition for interrupt function '%s'

割り込みとして宣言された関数が定義されていません。割り込み関数は、関数を割り込みとして宣言したプラグマと同じモジュール内で関数定義する必要があります。

1212: static function '%s' referenced but not defined

関数が static として宣言され、モジュール内の別の場所で参照されているにもかかわらず、この関数の定義が存在しません。通常は、関数定義の際の関数名のスペルミスが原因として考えられます。

1213: initializer list expected

初期化対象のシンボルには、ブレースで囲まれた初期化子のリストが必要ですが、1 つの値の初期化子しかありません。

1214: constant expression expected in initializer

静的に割り当てられたシンボルの初期化子の値は、定数式である必要があります。

1216: string initializer used for non-character array object

文字列リテラルの初期化子は、「char 型の配列」型または「char 型へのポインタ」型 ( いずれの場合も unsigned char を含む ) のオブジェクトの初期化にのみ使用できます。

1218: extraneous initializer values

初期化子の値の数が、初期化対象のオブジェクトの型で指定できる値の数と一致しません。初期化子リストの値が多すぎます。

1219: integer constant expected

整数型の定数式が必要ですが、整数型以外の式または定数のない式しかありません。

1220: initializer detected in typedef declaration of '%s'

typedef 宣言には、初期化子を含めることはできません。

1221: empty initializer list detected

初期化子リストを空にはできません。ブレースの間に、少なくとも 1 つの初期化子の値が必要です。

1222: "#pragma config" detected inside function body

config プラグマは、ファイル レベルのスコープでのみ利用できます。

1223: configuration setting '%s' has already been specified

指定した構成設定は、同じ #pragma config のこれより前、または別の #pragma config で既に設定されています。

1224: configuration setting '%s' not recognized

指定した構成設定は、選択したデバイスでは認識されません。指定した設定が、すべて大文字で正しくスペルされていることを確認してください。選択したデバイスで利用可能な構成設定の詳細については、--help-config を実行してください。

1225: configuration value '%s' not recognized for configuration setting '%s'

指定した構成の値は、選択したデバイスおよび構成設定では認識されません。指定した値が、すべて大文字で正しくスペルされていることを確認してください。選択したデバイスで利用可能な構成設定と値の詳細については、--help-config を実行してください。

1226: cannot specify both #pragma config and _CONFIG_DECL macro

構成設定には、#pragma config ディレクティブまたは _CONFIG_DECL マクロのいずれかのみ指定できます (#pragma config を推奨 )。

1227: cannot specify #pragma config directive when compiling for generic device

#pragma config ディレクティブはプロセッサ固有のディレクティブであり、コマンドライン オプション -p で特定のプロセッサを指定する必要があります。

1228: %s cannot be specified in both interrupt save= and nosave= clauses

指定した場所は、割り込み定義の save= 節にも nosave= 節にも指定できません。save= リストまたは nosave= リストから、その項目を削除してください。

1250: '%s' operand %s must be a literal

オペコードに対して指定したオペランドはシンボル参照ではなく、リテラル値である必要があります。

1251: '%s' operand count mismatch

オペコードに指定したオペランドの数が不正です。MPASM アセンブラとは異なり、MPLAB C18 のインライン アセンブラでは、すべてのオペランドを明示的に指定する必要があります。アクセス ビットやデスティネーション ビットなどのオペランドには、デフォルト値はありません。

1252: invalid opcode '%s' detected for processor '%s'

このオペコードは、ターゲット プロセッサでは使用できません。通常は、命令セットの異なるプロセッサ間で ( 例 : PIC17CXX から PIC18CXX へ ) のインライン アセンブリ コードの移植、オペコードのスペルミスが原因として考えられます。

1253: constant operand expected

インライン アセンブリ オペコードのオペランドは、定数式に帰着する必要があります。定数式とは、リテラル定数または静的割り当てのシンボル参照に、オプションとして整定数をプラス / マイナスしたものです。通常は、インライン アセンブリのオペコードに対するオペランドとしての、動的に割り当てられたシンボル (auto 属性のローカル変数やパラメータ ) の使用が原因として考えられます。

1300: stack frame too large

スタック フレームのサイズが、アドレス指定可能な最大サイズを超えています。通常は、1 つの関数で auto ストレージ クラスに割り当てられたローカル変数が多すぎることが原因として考えられます。

1301: parameter frame too large

パラメータ フレームのサイズが、アドレス指定可能な最大サイズを超えています。通常は、1 つの関数に渡されるパラメータの数が多すぎることが原因として考えられます。

1302: old style function declarations not supported

MPLAB C18 では現在、旧来の K&R スタイルの関数定義はサポートしていません。ANSI 規格で推奨されているインライン パラメータの型宣言を使用してください。

1303: 'near' symbol defined in non-access qualified section

access 属性のないセクションに静的に割り当てられた変数には、アクセス ビットでアクセスできないため、near 修飾子で定義すると不正な番地にアクセスしてしまいます。

1304: illegal use of obsolete 'overlay' storage class for symbol '%s'

拡張モードでは、overlay ストレージ クラスはサポートされません。なお非拡張モードでは、overlay ストレージ クラスはローカル変数にのみ使用できます。

1500: unable to open file '%s'

コンパイラはその名前のファイルを開くことができませんでした。通常は、ファイル名のスペルミスやアクセス権限が不十分なことが原因として考えられます。

1504: redefinition of '%s'

同じ名前の関数は、複数回定義できません。

1505: redeclaration of '%s'

同じ名前の変数は、複数回宣言できません。

1506: function '%s' cannot have 'overlay' storage class specifier

overlay ストレージ クラスは、関数には指定できません。

1507: variable '%s' of 'overlay' storage class cannot have 'near' qualifier

アクセス RAM に overlay ストレージ クラスの変数を配置することは、現在サポートされていません。

1508: inconsistent linkage for %s

この識別子には、内部結合と外部結合の両方が与えられています。

1509: %s cannot have 'extern' storage class

extern ストレージ クラスは、パラメータには指定できません。

1510: %s cannot have 'extern' storage class, block scope, and an initializer

extern ストレージ クラスのブロック スコープ オブジェクトは、明示的な初期化はできません。

1511: ran out of internal memory for temps

これ以上の一時変数割り当てはできません。

1512: redefinition of label '%s'

1 つの関数内では、同じラベルを何度も定義できません。

1513: redefinition of member '%s'

構造体または共用体は、同じ名前のメンバーを複数持つことはできません。

1514: cast of a pointer to floating point is undefined

不正なキャストが要求されました。通常は、代入時に配列の添字を省略したことが原因として考えられます。

1515: redefinition of case value %ld

switch 文では、ある特定の値に対して使用できる case 文は 1 つのみです。

1516: array size must be greater than zero

配列のサイズに指定する定数は、ゼロより大きい値である必要があります。

## D.2 警告

2001: non-near symbol '%s' declared in access section '%s'

access 属性のセクションに静的に割り当てられた変数は、リンカによって常にアクセス データ メモリに配置されるため、ストレージの修飾には常に near 修飾子を使用できます。ストレージに near 修飾子を指定しなくても不正なコードとはなりませんが、本来不要なバンク選択命令が追加される場合があります。

2002: unknown pragma '%s'

不明なプラグマ ディレクティブが検出されました。ANSI/ISO の規定に従い、このプラグマは無視されます。通常は、プラグマ名のスペルミスが原因として考えられます。

2003: _CONFIG_DECL macro has been deprecated; please utilize #pragma config

_CONFIG_DECL マクロはほとんど使用されておらず、現在フェーズ アウトの段階にあります。今後は #pragma config ディレクティブを使用してください。

2025: default overlay locals is unsupported in Extended mode, -sco ignored

拡張モードでは、overlay ストレージ クラスはサポートされていません。

2026: default static locals is unsupported in Extended mode, -scs ignored

拡張モードでは、static ストレージ クラスをデフォルトとすることはできません。

2027: default auto locals is redundant in Extended mode, -sca ignored

拡張モードでは、ローカル変数のデフォルトのストレージ クラスは常に auto です。

2028: default static locals is unsupported in Extended mode, -Ol ignored

拡張モードでは、static ストレージ クラスをデフォルトとすることはできません。

2029: default access RAM is unsupported in Extended mode, -Oa ignored

拡張モードでは、near ストレージ領域をデフォルトとすることはできません。

2052: unexpected return value

戻り値なしとして宣言された関数で、戻り値のある文が検出されました。この戻り値は無視されます。

2053: return value expected

戻り値ありとして宣言された関数で、戻り値が検出されませんでした。この場合、戻り値は定義なしとなります。

2054: suspicious pointer conversion

明示的なキャストなしに、ポインタが整数として、または整数がポインタとして使用されています。

2055: expression is always false

条件文の制御式は、常に偽の定数値となります。

2056: expression is always true

条件文の制御式は、常に真の定数値となります。

2058: call of function without prototype

スコープ内に関数プロトタイプのない関数が呼び出されました。この場合、関数引き数の型チェックが実行できないため、安全ではありません。

2059: unary minus of unsigned value

通常、単項マイナス演算子は符号付きの値にのみ使用します。

2060: shift expression has no effect

値をゼロ ビットだけシフトしても、式の値は変化しません。

2062: '->' operator expected, not '.'

構造体 / 共用体のメンバーに対して、構造体 / 共用体へのポインタを使用してアクセスする際に、「.」演算子が使用されました。

2063: '.' operator expected, not '->'

構造体 / 共用体のメンバーに対して直接アクセスする際に、「->」演算子が使用されました。

2064: static function '%s' not defined

この関数は static として宣言されていますが、この関数の定義が見つかりません。通常は、関数定義の際の関数名のスペルミスが原因として考えられます。

2065: static function '%s' never referenced

この関数は static として定義されていますが、一度も参照されていません。

2066: type qualifier mismatch in assignment

ポインタの代入において、ソース ポインタとデスティネーション ポインタが互換性のある型のオブジェクトを指し示していますが、ソース ポインタが示すオブジェクトの属性が const または volatile であるのに対し、デスティネーション ポインタはそれ以外の属性のオブジェクトを指し示しています。

2068: obsolete use of implicit 'int' detected

ANSI 規格では、基本型を指定しないで変数を宣言できます ( 例えば extern x; とした場合、基本型 int が暗黙的に適用されます )。ただし ANSI 規格ではこの用法を旧式と指摘しているため、診断情報を出力するようにしています。

2069: enumeration value exceeds maximum range

signed long 形式で表現できない列挙型の値が宣言され、列挙型のタグが負の列挙型の値となっています。列挙型の値は unsigned long で表現されますが、負の表現を持つ列挙型の定数を相対比較しても、期待どおりに動作しない場合があります。

2071: %s cannot have 'overlay' storage class; replacing with 'static'

現時点では、パラメータに overlay ストレージ クラスの属性は指定できません。デフォルトのローカル ストレージ クラスが overlay の場合、パラメータには static ストレージ クラスが適用されます。

2072: invalid storage class specifier for %s; ignoring

この宣言には使用不可能なストレージ クラス指示子が使用されています。

2073: null-terminated initializer string too long

ヌル終端の初期化子文字列が長すぎて、配列オブジェクトに入りません。

2074: location %s specified in the interrupt save list is redundant

割り込み宣言の save= 節で、無効なシンボル名を指定しています。このシンボルについては、必要に応じてコンパイラがコンテクスト保存コードを自動的に生成するので、save= リストから適切に削除してください。

2075: location %s specified in the interrupt nosave list is changed by the interrupt, but will not be saved

nosave= 節で指定したシンボル名が、割り込み関数によって変更されました。ただし、このシンボルに関してはコンパイラがコンテクスト保存コードを生成しないため、元の値が失われる場合があります。

2076: %s ignored in the interrupt nosave list, as it is preserved via shadow register

この番地は高優先度割り込みシャドー レジスタによって保存されるため、高優先度割り込みの nosave= 節に指定しても無視されます。

2077: %s is not a valid compiler managed resource for use in the interrupt nosave list

nosave= 節に指定できるコンパイラ管理リソースは、WREG、BSR、STATUS、PROD、FSR0、TBLPTR、TABLAT、section(".tmpdata")、section("MATH_DATA") のみです ( 拡張モード時は __RETVAL0 も指定可能 )。

2100: obsolete use of 'overlay' for symbol '%s', processing as 'auto'

拡張モードでは、overlay ストレージ クラスはサポートされていません。この宣言は、ストレージ クラスに auto が指定されたものとして処理されます。

2101: obsolete use of 'static' storage for parameter '%s', treating as 'auto'

拡張モードでコンパイルを実行する場合、MPLAB C18 ではすべての関数パラメータが auto ストレージ クラスである必要があります。詳細については、MPLAB C18 ユーザーズ ガイドを参照してください。

2102: near range specifier ignored for 'auto' variable '%s'

ストレージ クラスに auto を指定した変数はスタックに配置されるため、変数のストレージに near 修飾子を指定してもアクセス メモリには配置されません。

## D.3 メッセージ

3000: test of floating point for equality detected

2 つの浮動小数点値が同じかをテストしても、期待される結果が得られない場合があります。2 つの式が数学的に等しくても、丸め誤差により演算結果が異なる場合があります。

3002: comparison of a signed integer to an unsigned integer detected

符号付き整数と符号なし整数を比較しても、符号付き整数が負の場合は、期待した結果が得られない場合があります。符号なし整数と、2 進数で等価な符号付きの値の表現を比較するには、最初に符号付きの値を明示的に同じサイズの符号なしの型に修正する必要があります。

# 付録 E 拡張モード

ここでは、非拡張モードと拡張モードの違いについて詳しく説明します。これらのモードの違いは次のとおりです。

- ソース コードの互換性
  - スタック フレームのサイズ
  - static パラメータ
  - overlay キーワード
  - インライン アセンブリ
  - 定義済みマクロ
- コマンドライン オプションの違い
- COFF ファイルの違い

## E.1 ソース コードの互換性

### E.1.1 スタック フレームのサイズ

コンパイラの拡張モード動作時、スタック フレーム ( ローカル変数、パラメータ、およびフレーム ポインタを格納 ) の総サイズは、1 つの関数につき 96 バイトに制限されます。非拡張モードでは、1 つの関数につきローカル変数用に 120 バイト、パラメータ用に 120 バイトを使用できます。

### E.1.2 static パラメータ

コンパイラの拡張モード動作時は、static パラメータはサポートされません。コンパイラの拡張モード動作中に static パラメータが検出されると、警告が出力されます。加えて、これらのパラメータについては、明示的に auto パラメータが指定されたものとして扱われます。これらのパラメータはグローバルには割り当てられず、スタックに格納されます。拡張モードではスタック フレームのサイズが 1 つの関数につき 96 バイトに制限されているため、非拡張モードでは問題がなくても、拡張モードで stack frame too large という診断メッセージが出力される場合があります。この問題を解決するには、関数のパラメータ数を削減する必要があります。

### E.1.3 overlay キーワード

コンパイラの拡張モード動作時は、overlay キーワードはサポートされません。コンパイラの拡張モード動作中に overlay キーワードが検出されると、警告が出力されます。加えて、これらのコードについては、明示的に auto キーワードが指定されたものとして扱われます。static パラメータの場合と同様、overlay ローカル変数はグローバルには割り当てられず、スタックに格納されます。拡張モードではスタック フレームのサイズが 1 つの関数につき 96 バイトに制限されているため、非拡張モードでは、問題がなくても拡張モードで stack frame too large という診断メッセージが出力される場合があります。この問題を解決するには、関数の auto パラメータ数を削減する必要があります。例えば、overlay 変数を static 変数に変更するのも 1 つの方法です。

| | |
|---|---|
| **注 :** | overlay セクション (#pragma overlay) は、コンパイラの動作モードにかかわらずサポートされます。 |

### E.1.4 インライン アセンブリ

コンパイラの拡張モード動作時は、インライン アセンブリに拡張命令 (ADDFSR、ADDULNK、CALLW、MOVSF、MOVSS、PUSHL、SUBFSR、SUBULNK) を使用できます。コンパイラの非拡張モード動作時にインライン アセンブリ内で拡張命令が検出されると、エラーが出力されます。

また、コンパイラの拡張モード動作時は、リテラル オフセットによるインデックス アドレス指定を示す MPASM アセンブラのブラケット付き構文 (CLRF [2] など ) は認識されません。ただし、f オペランドが 0x5F 以下で、なおかつアクセス ビット オペランド (a) がゼロに設定 ( 例 : CLRF 2, 0) されていれば、コンパイラのインライン アセンブリでリテラル オフセットによるインデックス アドレス指定が正しく認識されます。これと同じ命令は、コンパイラの非拡張モード動作時はアクセス RAM への参照として解釈されます。

### E.1.5 定義済みマクロ

定義済みマクロを使用すると、コンパイラの動作モードにかかわらずソース コードの互換性を維持できます。拡張モードでのコンパイル時には定義済みマクロ __EXTENDED18__ が定数 1 となり、非拡張モードでコンパイル時には定義済みマクロ __TRADITIONAL18__ が定数 1 となります。

次に、定義済みマクロの有用な使用例を示します。

1. 定義済みマクロを利用して、非拡張モード時は static パラメータを使用し、拡張モード時は auto パラメータを使用する。

```
#ifdef __EXTENDED18__
#define SCLASS auto
#else
#define SCLASS static
#endif

void foo (SCLASS int bar);
```

2. 定義済みマクロを利用して、非拡張モード時は overlay キーワードを使用し、拡張モード時は auto キーワードを使用する。

```
#ifdef __EXTENDED18__
#define SCLASS auto
#else
#define SCLASS overlay
#endif

void foo (void)
{
SCLASS int bar;

...
}
```

3. 定義済みマクロを利用して、非拡張モード時はインライン アセンブリで拡張命令以外の命令を使用し、拡張モード時はインライン アセンブリで拡張命令を使用する。

```
_asm
#ifdef __EXTENDED18__
PUSHL 5
#else
MOVLW 5
MOVWF POSTINC1, 0
#endif
...
MOVF POSTDEC1, 1, 0
_endasm
```

## E.2 コマンドライン オプションの違い

コンパイラの拡張モード動作時は、次のコマンドライン オプションはサポートされません。

- デフォルトのローカル ストレージ クラス (-scs/-sco/-sca)
  コンパイラの拡張モード動作時は、デフォルトのローカル変数は auto のみサポートされます。
- アクセス メモリへのデフォルトのデータ配置 (-Oa+/-Oa-)
  拡張モード デバイスではアクセス RAM の容量に制約があるため、コンパイラの拡張モード動作時のデフォルトでは、データはアクセス RAM に配置できません。

## E.3 COFF ファイルの違い

### E.3.1 汎用プロセッサ

汎用プロセッサ (-p18cxx) 用にコンパイルを実行する際は、COFF ファイルのオプション ファイル ヘッダで指定したプロセッサ タイプ (proc_type) は、拡張モードでは PIC18F4620 に設定され、非拡張モードでは PIC18C452 に設定されます。

### E.3.2 ファイル ヘッダの f_flags フィールド

コンパイラの拡張モード動作時は、ファイル ヘッダの f_flags の F_EXTENDED18 ビットがセットされた COFF ファイルが生成されます。このビットは、コンパイラの非拡張モード動作時はセットされません。

ノート :

# 用語集

## 数字

### 16 進数

0 ～ 9 の数字と A ～ F ( または a ～ f) のアルファベットを使用する、16 を底とした記数法。A ～ F で 10 進数の 10 ～ 15 を表現する。一番右の桁が 1 の位、次の桁が 16 の位、その次の桁が $16^2 = 256$ の位、となる

### 2 進数

0 ～ 1 の 2 つの数字を使用する、2 を底とした記数法。一番右の桁が 1 の位、次の桁が 2 の位、その次の桁が $2^2 = 4$ の位、となる

### 8 進数

0 ～ 7 の数字を使用する、8 を底とした記数法。一番右の桁が 1 の位、次の桁が 8 の位、その次の桁が $8^2 = 64$ の位、となる

## A

### ANSI

米国規格協会 (American National Standards Institute)

## C

### CPU

中央演算処理装置

## I

### ICD

インサーキット デバッガ

### ICE

インサーキット エミュレータ

### IDE

統合開発環境

### IEEE

電気電子学会 (Institute of Electrical and Electronics Engineers)

### ISO

国際標準化機構 (International Organization for Standardization)

### ISR

割り込みサービス ルーチン

## M

### MPASM アセンブラ

PICmicro マイクロコントローラ ファミリに対応した、マイクロチップ テクノロジー社提供の再配置可能なマクロ アセンブラ

### MPLIB オブジェクト ライブラリアン

PICmicro マイクロコントローラ ファミリに対応した、マイクロチップ テクノロジー社提供のライブラリアン

**MPLINK オブジェクト リンカ**
PICmicro マイクロコントローラ ファミリに対応した、マイクロチップ テクノロジー社提供のリンカ

## R

**RAM**
ランダム アクセス メモリ

**ROM**
読み出し専用メモリ (Read Only Memory)

## あ

**アクセス メモリ**
バンク選択レジスタ (BSR) の設定にかかわらずアクセス可能な、PIC18 PICmicro マイクロコントローラの特別な汎用レジスタ

**アセンブラ**
アセンブリ ソース コードをマシン コードに変換する言語ツール

**アセンブリ**
2 進数のマシン コードを可読性の高い形式に記述したシンボリック言語

**アドレス**
ある情報をメモリのどこに格納するかを識別するためのコード

## え

**エラー ファイル**
MPLAB C18 コンパイラが生成した診断情報を含むファイル

**エンディアン**
マルチバイト オブジェクトにおけるバイトの並び順

## お

**オブジェクト コード**
アセンブラまたはコンパイラによって生成されるマシン コード

**オブジェクト ファイル**
オブジェクト コードを含むファイル。そのまま実行できるものと、他のオブジェクト コード ファイル ( ライブラリなど ) とリンクしてから完全な実行可能プログラムを生成することが必要なものとがある

## か

**拡張モード**
拡張モードでは、コンパイラは拡張命令 (ADDFSR、ADDULNK、CALLW、MOVSF、MOVSS、PUSHL、SUBFSR、SUBULNK) およびリテラル オフセットによるインデックス アドレス指定を利用

## こ

**高級言語**
プログラムを記述するための言語で、プロセッサから見てアセンブリよりも遠い位置関係にあるもの

**コンパイラ**
高級言語で記述されたソース ファイルをマシン コードに変換するプログラム

## さ

**再帰**
自分自身を参照すること ( 自分自身を呼び出す関数など )

**再入**
複数のインスタンスを同時にアクティブにすることが可能な関数。直接または間接再帰、あるいは割り込み処理中の実行によって発生

**再配置可能**
アドレスが特定のメモリ番地に固定されていないオブジェクト

## し

**条件付きコンパイル**
プログラムの一部を、プリプロセッサ ディレクティブによって指定された特定の定数式が真の場合のみコンパイルすること

## す

**ストレージ クラス**
オブジェクトが識別されると、そのオブジェクトのメモリ存続期間を決定

**ストレージ修飾子**
宣言されるオブジェクトの特別な属性を指定 (const など )

## せ

**セクション**
アプリケーションの一部を特定のメモリ番地に配置したもの

**セクション属性**
セクションに与えられた性質 (access セクションなど )

**絶対セクション**
リンカによって変更されない固定アドレスを持つセクション

## ち

**遅延**
イベントが発生してからそれに応答するまでの時間

**致命的エラー**
コンパイルが即時に中止されるようなエラー。これ以降はメッセージの生成もしない

**中央演算処理装置**
デバイスで実行する命令を正しくフェッチし、命令をデコードし、その命令を実行する役割を果たす部分。必要に応じて、算術論理演算装置 (ALU) と組み合わせて命令実行を完了する。プログラム メモリのアドレス バス、データ メモリのアドレス バス、スタックへのアクセスを制御する

## と

**特殊機能レジスタ**
I/O プロセッサ機能、I/O ステータス、タイマなど、さまざまなモードや周辺モジュールを制御するためのレジスタ

**匿名構造体**
名前のないオブジェクト

## ひ

**非拡張モード**
非拡張モード時のコンパイラでは、拡張命令はもとより、リテラル オフセットによるインデックス アドレス指定も利用不可

**非同期**
複数のイベントが同時には発生しないこと。この用語は主に、プロセッサの実行中のいつでも発生する可能性がある割り込みを指すのに使用される

## ふ

**フリースタンディング**
言語規格に厳密に準拠しており、複素数型を使用しておらず、ライブラリに関する条項 (ANSI '89 規格の条項 7) に指定された機能の使用が標準ヘッダ <float.h>、<iso646.h>、<limits.h>、<stdarg.h>、<stdbool.h>、<stddef.h>、<stdint.h> の内容に制限されているプログラムを扱う処理系

**フレーム ポインタ**
スタック ベースの引き数とスタック ベースのローカル変数の境界となるスタック番地を指し示すポインタ

**プラグマ**
コンパイラによって固有の意味を持つディレクティブ

## へ

**ベクタ**
リセットまたは割り込み発生時にアプリケーションのジャンプ先となるメモリ番地

## ま

**マイクロコントローラ**
CPU、RAM、何らかの形式の ROM、I/O ポート、タイマなど多くの機能を集積したチップ

## め

**メモリ モデル**
プログラム メモリを指し示すポインタのサイズに関する規定を記述したもの

## よ

**読み出し専用メモリ**
恒久的に保存されているデータへの高速アクセスが可能なメモリ ハードウェア。ただし、データの追加や変更は不可

## ら

**ライブラリ**
再配置可能なオブジェクト モジュールを集めたもの

**ライブラリアン**
ライブラリを作成、操作するためのプログラム

**ランタイム モデル**
コンパイラの動作の前提となるもの

**ランダム アクセス メモリ**
任意の順に情報にアクセスできるメモリ デバイス

## り

**リトル エンディアン**
あるオブジェクト内で、下位バイトを下位アドレスに格納する方式

**リンカ**
オブジェクト ファイルとライブラリを結合して実行可能コードを作成するプログラム

## わ

**割り当て済みセクション**
リンカ コマンド ファイルでターゲット メモリ ブロックに割り当てられたセクション

**割り込み**
CPU に対する信号の一種。この信号が発生すると、現在動作中のアプリケーションの実行を一時停止し、制御を ISR に渡してイベントを処理する。ISR の実行が完了すると、通常のアプリケーションの実行が再開される

**割り込みサービス ルーチン**
割り込みを処理する関数

ノート :

# 索引

## き



## こ



## さ



## し

# 世界各国での販売およびサービス

## 北米

**本社**
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
テクニカルサポート:
http://support.microchip.com
ウェブアドレス:
www.microchip.com

**アトランタ**
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

**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**
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

**ココモ**
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

**ロサンゼルス**
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

**サンタクララ**
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

**トロント**
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

## アジア / 太平洋

**アジア太平洋支社**
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

**オーストラリア - シドニー**
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

**中国 - 北京**
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

**中国 - 成都**
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

**中国 - 香港 SAR**
Tel: 852-2401-1200
Fax: 852-2401-3431

**中国 - 南京**
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

**中国 - 青島**
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

**中国 - 上海**
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

**中国 - 瀋陽**
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

**中国 - 深川**
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

**中国 - 武漢**
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

**中国 - 厦門**
Tel: 86-592-2388138
Fax: 86-592-2388130

**中国 - 西安**
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

**中国 - 珠海**
Tel: 86-756-3210040
Fax: 86-756-3210049

## アジア / 太平洋

**インド - バンガロール**
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

**インド - ニューデリー**
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

**インド - プネ**
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

**日本 - 横浜**
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

**韓国 - 大邱**
Tel: 82-53-744-4301
Fax: 82-53-744-4302

**韓国 - ソウル**
Tel: 82-2-554-7200
Fax: 82-2-558-5932 または
82-2-558-5934

**マレーシア - クアラルンプール**
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

**マレーシア - ペナン**
Tel: 60-4-227-8870
Fax: 60-4-227-4068

**フィリピン - マニラ**
Tel: 63-2-634-9065
Fax: 63-2-634-9069

**シンガポール**
Tel: 65-6334-8870
Fax: 65-6334-8850

**台湾 - 新竹**
Tel: 886-3-572-9526
Fax: 886-3-572-6459

**台湾 - 高雄**
Tel: 886-7-536-4818
Fax: 886-7-536-4803

**台湾 - 台北**
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

**タイ - バンコク**
Tel: 66-2-694-1351
Fax: 66-2-694-1350

## 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: 45-4450-2828
Fax: 45-4485-2829

**フランス - パリ**
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

**ドイツ - ミュンヘン**
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

**イタリア - ミラノ**
Tel: 39-0331-742611
Fax: 39-0331-466781

**オランダ - ドリューネン**
Tel: 31-416-690399
Fax: 31-416-690340

**スペイン - マドリッド**
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

**英国 - ウォーキンガム**
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08