MPLAB® C30
C コンパイラー
ユーザーズガイド

マイクロチップデバイスのコード保護機能に関する以下の点に留意ください。

- マイクロチップの製品は各製品独自のマイクロチップデーターシートにある仕様を満たしています。
- 各製品ファミリーは、通常の状態で所定の方法で利用いただければ市場にある類似製品の中で最も安全なファミリーの一つとマイクロチップは信じております。
- 不正かつ非合法な方法を使ったコード保護機能の侵害があります。弊社の理解ではこうした手法は、マイクロチップデーターシートにある動作仕様書以外の方法でマイクロチップ製品を使用することになります。こうした手法を使用した人は、ほとんどの場合、知的財産権の侵害となります。
- マイクロチップはコードの統合性に関心をお持ちの顧客とは協働させていただきます。
- マイクロチップまたは他のセミコンダクターメーカーがコードの安全性を保証したものではありません。コード保護は製品保護が「破られない」ということを保証するものではありません。

コード保護は常に進化します。マイクロチップは、当社製品のコード保護機能を継続的に改善することをお約束いたします。マクロチップのコード保護機能を破ることは、デジタル・ミレニアム著作権法に違反します。こうした行為によるソフトウェアーや著作権に関わる作品への不正アクセスがあった場合、同法に基づき賠償請求する権利があります。

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

マイクロチップは、1OS/TS-16949 を受けました。本社、アリゾナ州チャンドラーとテンペとカリフォルニア州マウンテンビューにあるデザイン及びウエハー施設に対する 2002 年品質システム認証です。弊社の品質システムプロセスと手続きは、PICmicro® 8-bit MCUs、KEELOQ® コードホッピングデバイス、シリアル EEPROMs、マイクロペリフェラル、非揮発性メモリーとアナログ製品を対象としています。更に、開発システムの設計及び製造に関するマイクロチップの品質システムは、2000 年に ISO9001 の認証を受けています。

MPLAB® C30
ユーザーズガイド

目次

MPLAB® C30
ユーザーガイド

はじめに

顧客の皆様への注意

すべての文書には日付が記載されており、このマニュアルも例外ではありません。マイクロチップ社のツールと文書は顧客の皆様のニーズに応えるため日々進化を続けており、このマニュアル内のダイアログおよび / またはツールの説明も変更となる可能性があります。最新のマニュアルは当社の **Web** サイト **(www.microchip.com)** で入手してください。

マニュアルは **"DS"** 番号で識別されます。この番号は各ページ下のページ数の下欄に記載されています。**DS** 番号の番号付けは **"DSXXXXXA"** となっており、**"XXXXX"** は文書番号、**"A"** は改訂番号となっています。

開発ツールの最新の情報は、**MPLAB IDE** のオンラインヘルプを参照してください。「ヘルプ」メニューを選択し、「トピックス」をクリックすると、入手可能なオンラインヘルプファイルのリストが開きます。

序章

このドキュメントは、ユーザーのアプリケーションを開発するために、dsPIC® デバイス用のマイクロチップ MPLAB C30 C コンパイラを使用する際の手助けをすることを目的としています。MPLAB C30 は GCC（GNU コンパイラー群）ベースの言語ツールであり、フリーシフトウエアファウンデーション (FSF) からのソースコードをベースにしています。FSF の詳細につきましては、www.fsf.org をご参照ください。

マイクロチップから供給可能なその他の GNU 言語ツールは以下のとおりです。

- MPLAB ASM30 アセンブラ
- MPLAB LINK30 リンカー
- MPLAB LIB30 ライブラリアン / アーカイバ

本章で説明する内容は以下の通りです。

- 本ガイドについて
- 推奨文献
- トラブルシューティング
- マイクロチップウェブサイト
- 開発システムに関しての顧客への通知サービス
- カスタマサポート

本ガイドについて

本ガイドのレイアウト

本ガイドは、ファームウェアを開発する際の MPLAB C30 の使用方法を説明します。本ガイドのレイアウトは以下の通りです。

- **第 1 章 : コンパイラー概要** – MPLAB C30, 開発ツールおよび特徴について説明します。
- **第 2 章 : MPLAB C30 と ANSI C の違い** – MPLAB C30 構文でサポートされる C 言語と標準 ANSI-89 C との違いについて説明します。
- **第 3 章 : MPLAB C30 を使用する** – コマンドラインからの MPLAB C30 コンパイラの使い方について説明します。
- **第 4 章 : MPLAB C30 実行時環境** – MPLAB C30 の実行時モデルおよびセクション情報、初期化、メモリモデル、ソフトウエアスタック等々について説明します。
- **第 5 章 : データタイプ** – MPLAB C30 で用いられる整数、浮動小数、ポインタのデータ型を説明します。
- **第 6 章 : デバイスサポートファイル** – MPLAB C30 のヘッダ、レジスタ定義ファイルおよび SFR を使用する場合の使用法を説明します。
- **第 7 章 : 割り込み** – 割込みの使用方法を説明します。
- **第 8 章 : アセンブリ言語と C モジュールの混用** – MPLAB C30 と MPLAB ASM30 アセンブル言語モジュールを一緒に用いる場合のガイドラインを説明します。
- **別紙 A: 実装時定義された動作** – ANSI 標準で実装時定義と記載された MPLAB C30 の特殊パラメータについて説明します。
- **別紙 B: MPLAB C30 の診断** – MPLAB C30 が生成するエラーや警告メッセージの一覧を示します。
- **別紙 C: MPLAB C18 と MPLAB C30 の違い** – PIC18XXXXX コンパイラ (MPLAB C18) と dsPIC コンパイラ (MPLAB C30) の違いについて特記します。
- **別紙 D: ASCII 文字セット** – ASCII 文字の一覧を示します。
- **別紙 E: GNU 無料ドキュメントライセンス** – 無料ソフト基金の使用許諾。

本ガイドで用いられる凡例

本マニュアルでは、以下のドキュメント凡例を用いています。

ドキュメント凡例

文字種類	意味	使用例
Arial フォント：		
イタリック文字	参照文献	*MPLAB IDE ユーザーズガイド*
	強調する文字	... は唯一のコンパイラー ...
最初が大文字	ウィンドウ	the Output window
	ダイアログ	the Settings dialog
	メニュー選択	select Enable Programmer
引用	ウィンドウもしくはダイアログ内のフィールド名	“Save project before build”
右矢印を持ったアンダーライン付きイタリック	メニュー選択パス	*File>Save*
太字	ダイアログボタン	**OK** をクリック
	タブ	**Power** タブをクリック
‘b*nnnn*	バイナリ数値。*n* は桁数だけ並べる。	‘b00100, ‘b10
角括弧 **<>**, 内の文字	キーボードのキー	**<Enter>**、**<F1>** を押します
クーリエフォント：		
通常クーリエ	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`
	コマンドラインオプション	`-Opa+, -Opa-`
	ビット値	`0, 1`
イタリッククーリエ	変数の引数	*`file`*`.o` では、*`file`* は有効なファイル名を表す。
`0x`*`nnnn`*	16 進法番号。*n* は 16 の各桁。	`0xFFFF, 0x007A`
角括弧 [ ]	オプション引数	`mcc18 [options]` *`file`* `[options]`
中括弧 {|} とパイプ文字	互いに排他的な配列もしくは OR 選択	`errorlevel {0|1}`
省略 ...	テキストの繰り返し	`var_name [, var_name...]`
	ユーザーが入力するコード	`void main (void) { ... }`

推奨文献

このユーザーズガイドは dsPIC デバイス用の MPLAB C30 コンパイラの使い方について説明しています。MPLAB C30 とその他のツールに関するより詳細な情報については、以下の文書を読むことをお勧めします。

README ファイル

マイクロチップツールの最新の情報については、ソフトウエアに含まれる添付 README ファイル (ASCII テキストファイル) をお読みください。

dsPIC 言語ツールの入門 (DS70094)

dsPIC デジタルシグナルコントローラ (DSC's) 用のマイクロチップ言語ツール (MPLAB ASM30、MPLAB LINK30、MPLAB C30) のインストールと使い方についてのガイドが説明されています。dsPIC シミュレータや MPLAB SIM30 の使用例についても説明されています。

MPLAB ASM30, MPLAB LINK30 とユーティリティユーザーズガイド (DS51317)

dsPIC DSC アセンブラ、MPLAB ASM30、dsPIC DSC リンカー、MPLAB LINK30 および 種々の dsPIC DSC ユーティリティ、MPLAB LIB30 アーカイバ / ライブラリアンも含めて、使い方が説明されています。

GNU HTML ドキュメント

本ドキュメントは言語ツール CD-ROM で供給されます。これは、標準 GNU 開発ツールについて説明しており、MPLAB C30 の元になっています。

汎用とセンサーファミリの dsPIC30F のデータシート (DS70083)

dsPIC30F デジタルシグナルコントローラ (DSC) 用のデータシートです。デバイスとそのアーキテクチャの概要について説明しています。メモリ構成、DSP 命令と周辺機能および電気的特性の詳細について説明しています。

dsPIC30F ファミリリファレンスマニュアル (DS70046)

本ファミリリファレンスマニュアルは dsPICSOF ファミリのアーキテクチャと周辺モジュールの動作について説明しています。

dsPIC30F プログラマリファレンスマニュアル (DS70030)

dsPICS3OF デバイスのプログラマガイドで、プログラマモデルと命令セットを含みます。

C 標準情報

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、形式を規定し、C プログラム言語で表現されたプログラムの解釈を与えています。その目的は、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。

C リファレンスマニュアル

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

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

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

Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J. 07632.

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

トラブルシューティング

本ドキュメントで述べられていない共通問題に関する情報につきましては、README ファイルをご覧ください。

マイクロチップウェブサイト

マイクロチップでは、マイクロチップワールドワイドウエブ (WWW) サイト www.microchip.com で、オンラインサポートを提供しています。このウエブサイトは、お客様がファイルや情報を最も簡単に入手できる手段としてご利用いただけます。サイトをご覧になるには、各自、お好みのブラウザでご覧下さい。マイクロチップウエブサイトは、以下の情報を含みます。

- **製品サポート**－データシート、errata、アプリケーションノート、サンプルプログラム、デザインリソース、ユーザーズガイド、サポート資料、最新ソフトウェアリリースとアーカイブされたソフトウェア
- **一般的なテクニカルサポート**－よくある質問（FAQ）、テクニカルサポート要求、オンラインディスカッショングループ、マイクロチップコンサルタントプログラムメンバーリスト
- **マイクロチップのビジネス**－製品選択、オーダーガイド、最新マイクロチッププレスリリース、セミナーおよびイベントのリスト、マイクロチップセールスオフィス、ディストリビュータ、工場代表者

開発システムに関しての顧客への通知サービス

マイクロチップは、お客様が最小の努力で現在のマイクロチップ製品について最新情報を入手できることをお手伝いできるように、顧客通知サービスを継続して行っています。一度ご登録いただければ、あなたのご指定した製品ファミリもしくはご興味のある開発ツールに関して、変更、更新、改定もしくは正誤表が発行される毎に e- メールでの通知を受けることができます。

登録するには、マイクロチップのウエブサイトにアクセスし、Customer Change Notification をクリックし、指示に従ってご登録ください。

開発システム製品は下記の通り分類されます。

- **コンパイラ** ー マイクロチップ C コンパイラとその他の言語ツールに関する最新情報で、以下を含みます。MPLAB C17、MPLAB C18 もしくは MPLAB C30 C コンパイラ ; MPASM™、MPLAB ASM30 アセンブラ ; MPLINK™、MPLAB LINK30 オブジェクトリンカー ; MPLIB™、MPLAB LIB30 オブジェクトライブラリアン。
- **エミュレータ** ー マイクロチップインサーキットエミュレータ、MPLAB ICE 2000 と MPLAB ICE 4000 に関する最新情報を含みます。
- **インサーキットデバッガ** ー マイクロチップインサーキットデバッガ、MPLAB ICD 2 に関する最新情報を含みます。
- **MPLAB IDE** ー マイクロチップ MPLAB IDE Windows® 用の統合開発環境システムツールに関する最新情報です。この項では MPLAB IDE、MPLAB SIM と MPLAB SIM30 シミュレータ、MPLAB IDE プロジェクトマネジャと全般的な編集、デバッグの特徴に注力しています。
- プログラマ ー マイクロチップデバイスプログラマに関する最新情報で、MPLAB PM3、PRO MATE® II デバイスプログラマと PICSTART® Plus 開発用プログラマを含みます。

カスタマサポート

マイクロチップ製品のユーザーはいくつかのチャンネル経由で、以下のサポートを受けることができます。

- ディストリビュータもしくは販売特約店
- 地域販売オフィス
- 現地アプリケーションエンジニア (FAE)
- テクニカルサポート
- 開発システム情報ホットライン

サポートが必要な場合は、ディストリビュータ、販売特約店もしくは現地アプリケーションエンジニア (FAE) にお電話ください。地域販売オフィスでも顧客のサポートを行っています。販売店とその所在地については本ドキュメントの最後のページを参照してください。

テクニカルサポートは Web サイト http://support.microchip.com にてご利用いただけます。

第 1 章 コンパイラ概要

1.1 序章

デジタルシグナルコントローラ (DSCs) である dsPIC® ファミリは DSP 応用システムで要求される高性能と、組込み応用システムで必要な標準マイコンの特徴を合わせ持っています。

dsPIC DSCs は、最適化 C コンパイラ、アセンブラ、リンカ、およびアーカイバ / ライブラリアンを含んだ完全なソフトウェア開発ツールセットでサポートされています。

本章では、これらのツールの概要紹介や、最適化 C コンパイラの特徴を紹介し、MPLAB ASM30 アセンブラや MPLAB LINK30 リンカと一緒にどのように動作するかも説明しています。アセンブラとリンカは *MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide*, (DS51317) で詳細に説明されています。

1.2 ハイライト

本章で説明されるのは以下の内容です。

- MPLAB C30 の説明
- MPLAB C30 とその他の開発ツール
- MPLAB C30 の特徴

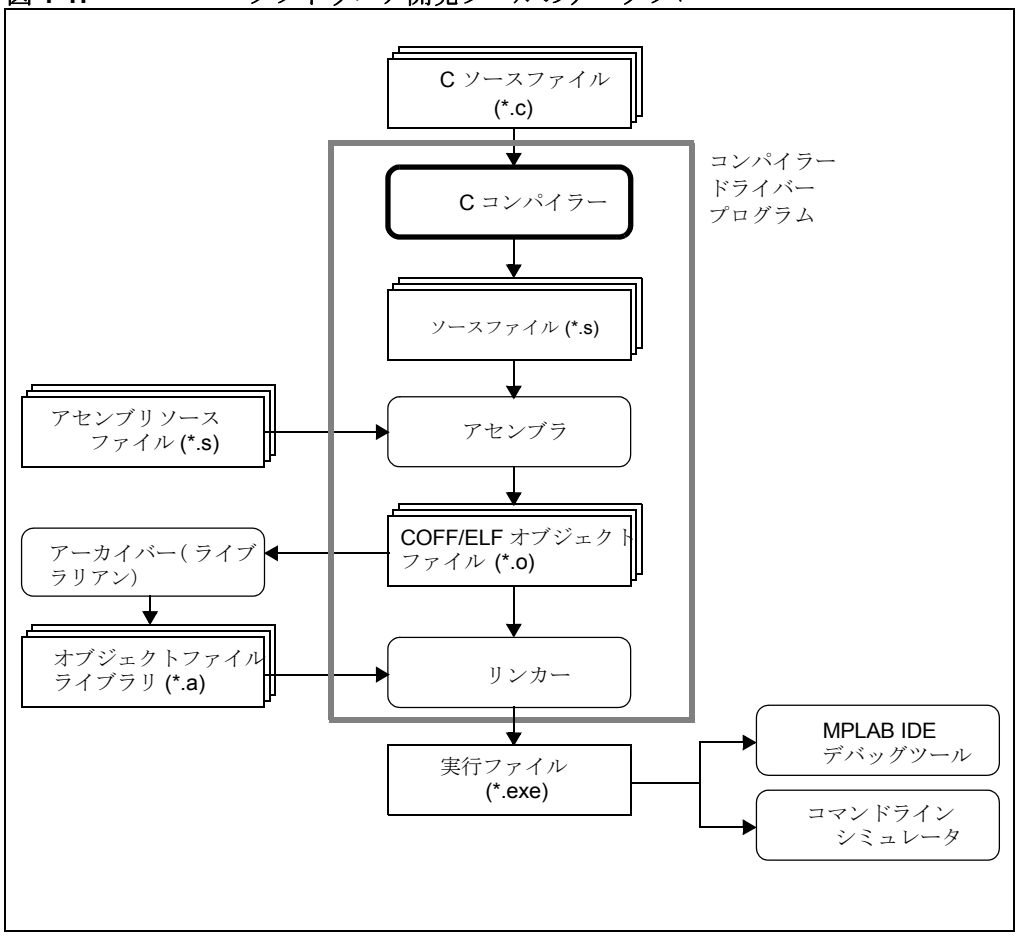
1.3 MPLAB C30 の説明

MPLAB C30 は ANSI x3.159-1989- 準拠の最適化 C コンパイラで、dsPIC 組込みコントロール応用向けの言語拡張を含んでいます。本コンパイラは、C コード開発用プラットフォームを提供する Windows® コンソールアプリケーションです。本コンパイラは、Free Software Foundation から供給される GCC コンパイラから移植しています。

1.4 MPLAB C30 とその他の開発ツール

MPLAB C30 は C ソースファイルをコンパイルし、アセンブル言語ファイルを生成します。これらコンパイラで生成されたファイルは、他のオブジェクトファイルやライブラリと一緒にアセンブル、リンクされて、実行可能な COFF ファイルフォーマットで最終アプリケーションプログラムが生成されます。COFF ファイルは MPLAB IDE にロードされ、テスト・デバッグされ、もしくは変換ユーティリティを用いて COFF ファイルから Intel® hex フォーマットに変換され、コマンドラインシミュレータやデバイスプログラマへのロードに適した形になります。ソフトウエア開発のデータフロー概要については図 1-1 をご参照ください。

図 1-1: ソフトウエア開発ツールのデータフロー



1.5 MPLAB C30 の特徴

MPLAB C30 の C コンパイラは豊富な特徴を持つ最適化コンパイラで、標準 ANSI C プログラムを dsPIC アセンブル言語ソースに変換します。このコンパイラはまた多くのコマンドラインオプションと言語拡張をサポートし、dsPIC デバイスハード機能をすべてアクセス可能であり、コンパイラコードジェネレータを細かく制御できます。この章ではこのコンパイラの主要な特徴について説明します。

1.5.1 ANSI C 標準

MPLAB C30 コンパイラは、十分検証されたコンパイラで、ANSI 規格で定められ、Kernighan and Ritchie's *The C Programming Language* (second edition) で説明されている ANSI C 標準に準拠しています。ANSI 標準は、現在では言語の標準機能になっているオリジナル C 定義への拡張を含みます。これらの拡張により移植性が強化され、能力増強されています。

1.5.2 最適化

本コンパイラは、C ソースから効率的でコンパクトなコードを生成できる多くの進んだ技術を適用する最適化パスを使用しています。最適化パスは、どんな C コードにも適用可能なハイレベルの最適化と、dsPIC デバイスアーキテクチャの特殊機能を利用した dsPIC デバイス専用の最適化を含んでいます。

1.5.3 ANSI 標準ライブラリサポート

MPLAB C30 は完全な ANSI C 標準ライブラリと一緒に供給されます。すべてのライブラリ機能は検証されており、ANSI C ライブラリ標準に適合しています。ライブラリには、ストリング操作、ダイナミックメモリ配置、データ変換、時間管理、数学関数（三角関数、指数関数、双曲線関数）が含まれます。ファイル操作用の標準 I/O 関数も含まれ、コマンドラインシミュレータを用いてホストファイルシステムへの完全アクセスをサポートします。低レベルのファイル I/O 関数用の完全動作可能なソースコードがコンパイラ配布時に提供され、本機能が必要なアプリケーションのスタートポイントとして使用できます。

1.5.4 フレキシブルなメモリモデル

本コンパイラはラージ・スモール両方のコードとデータモデルをサポートします。スモールコードモデルはコール関数のより効率的な形式を利用でき、スモールデータモデルでは、SFR 空間内でのデータアクセス用にコンパクトな命令を使用できます。

本コンパイラは定数データアクセスのために 2 つのモデルを提供します。“constants in data” モデルは、ランタイムライブラリにより初期化されるデータメモリを使用します。“constants in code” モデルは、Program Space Visibility (PSV) ウィンドウからアクセスするプログラムメモリを使用します。

1.5.5 コンパイラドライバ

MPLAB C30 には、強力なコマンドラインドライバプログラムが含まれています。ドライバプログラムを用いることで、アプリケーションプログラムのコンパイル・アセンブル・リンクを 1 ステップでコンパイルできます。（図 1-1 参照）

注意：

第 2 章 MPLAB C30 と ANSI C の違い

2.1 序章

本章では、MPLAB C30 構文でサポートされる C 言語と、1989 年版標準 ANSI C でサポートされる C 言語の違いについて説明します。

2.2 ハイライト

本章では下記について説明します。

- キーワードの違い
- ステートメントの違い
- 表現の違い

2.3 キーワードの違い

本章では、通常の ANSI C と、MPLAB C30 で使用できる C 言語との間でのキーワードの違いを説明します。新しいキーワードは基本 GCC 搭載内容の一部であり、本章では標準 GCC ドキュメントに基づいて説明され、特に、GCC の MPLAB C30 移植後の特殊構文や意味について説明します。

- 変数属性の指定
- 関数の属性を指定する
- インライン関数
- レジスタ変数
- 複素数
- 倍長整数
- `typeof` による型参照

2.3.1 変数属性の指定

MPLAB C30 のキーワード __attribute__ により、特殊な変数属性もしくは構文フィールドを指定することができます。このキーワードに続き、二重括弧内の属性仕様を指定します。現在、以下の属性が変数用としてサポートされています。

- address (addr)
- aligned (alignment)
- deprecated
- far
- mode (mode)
- near
- noload
- packed
- persistent
- reverse (alignment)
- section ("section-name")
- sfr (address)
- space (space)
- transparent_union
- unordered
- unused
- weak

属性指定のやり方として __ (2 つのアンダースコア) をそれぞれのキーワードの前と後ろに置く（例えば‘aligned’ではなく‘__aligned__’）。ことでも指定できます。これを用いれば、同じ名前を持つ可能性のあるマクロに関係なくヘッダファイルで属性を使用できます。

複数の属性を指定するには、二重括弧内でコンマにより区切ります。

例えば以下のように記述します。

```
__attribute__ ((aligned (16), packed)).
```

address (*addr*)

address 属性は変数の絶対アドレスを指定します。この属性は section 属性とは一緒に使用できません。address 属性が優先されます。address 属性を持つ変数は auto_psv 領域には配置できません。（space () 属性または -mconst-in-code オプションの項を参照）auto_psv 領域に配置しようとすると警告が発生しコンパイラはその変数を psv 領域に配置します。

変数が PSV セクションに配置されると、アドレスはプログラマメモリアドレスになります。

```
int var __attribute__ ((address(0x800)));
```

aligned (*alignment*)

この属性は、バイト単位で、変数の最小配列を指定します。alignment は 2 のべき乗値で指定します。例えば、以下のように宣言すると

```
int x __attribute__ ((aligned (16))) = 0;
```

コンパイラは 16 バイト領域でグローバル変数 X を割り当てます。dsPIC デバイスでは、配列オペランドを必要とする DSP 命令やアドレッシングモードをアクセスするために、アセンブラ表記と一緒に用いることができます。

上記例で示されるように、与えられた変数に対してコンパイラで用いる配列を、(バイト単位で) 明確に指定することができます。別のやり方として、配列要素数は省略して、dsPIC デバイスにとって最も有効な配列になるように変数の配列指定を行うこともできます。例えば、以下のように記述することができます。

```
short array[3] __attribute__ ((aligned));
```

配列属性の指定時に配列要素数を省略すると、コンパイラは宣言された変数の配列を、ターゲットマシン上で用いられるデータ形式の最大配列 (dsPIC デバイスの場合は 2 バイト (1 ワード)) に、自動的に設定します。

aligned 属性は配列を増加させるのみですが、packed を指定すると減少させることができます。(以下を参照ください)。aligned された属性は、reverse の属性と競合します。これは両方の属性に対するエラー条件となります。

deprecated

deprecated 属性は、コンパイラによって特別に認識されるように付けられた宣言を生成します。deprecated 関数もしくは変数が使用されると、コンパイラはワーニングを発生します。

deprecated 定義は、いったん定義されるとすべてのオブジェクトファイル内で有効となります。例えば、以下のファイルをコンパイルした場合、

```
int __attribute__((__deprecated__)) i;
int main() {
  return i;
}
```

次のような警告が発生します。

```
deprecated.c:4: warning: ‘i’ is deprecated (declared
 at deprecated.c:1)
```

この場合でも i は通常どおり定義されオブジェクトファイルに生成されます。

far

far 属性はコンパイラーに変数が二アデータ領域 (最初の 8 KB) に配置される必要がないことを伝えます (つまり、その変数はデータメモリ内のどこに配置されてもいいということです)。

mode (*mode*)

この属性は、データ型の宣言の際に、*mode* に対応させてどの型を使うかを指定します。これにより、データ幅に従って整数型もしくは浮動小数点型が指定されます。*mode* で使用できる有効値は以下の通りです。

モード	データ幅	MPLAB C30 Type
QI	8 ビット	char
HI	16 ビット	int
SI	32 ビット	long
DI	64 ビット	long long
SF	32 ビット	float
DF	64 ビット	long double

この属性は、MPLAB C30 でサポートされるターゲット全般にわたって移植可能なコードを記述する場合に役立ちます。例えば、以下の関数は、2 つの 32 ビット符号付き整数を加算し、結果を 1 つの 32 ビット符号付き整数で返します。

```
typedef int __attribute__((__mode__(SI))) int32;
int32
add32(int32 a, int32 b)
  {
        return(a+b);
  }
```

byte もしくは __byte__ のモードを指定することで、1 バイト整数に相当するモードを指示でき、word もしくは __word__ のモードを指定することで、1 ワード整数モードを指示でき、pointer もしくは __pointer__ のモードを指定することで、ポインタを表すモードを指示できます。

near

near 属性はコンパイラーに変数が二アデータ領域（データメモリの最初の 8 KB）に配置されていることを告げます。このような変数は二アデータ領域に配置されていない（もしくはどこに配置されているか不明な）変数よりも効率的にアクセスできることがあります。

```
int num __attribute__ ((near));
```

noload

noload 属性は変数のための領域を割り当てる必要があるが、初期値はロードする必要がないことを示します。この属性は、シリアル EEPROM 読み込みのようにアプリケーションが実行時にメモリに変数をロードするよう設計されている場合に有用です。

```
int table1[50] __attribute__ ((noload)) = { 0 };
```

packed

packed 属性は、aligned 属性で大きな値を設定していない場合に、変数もしくは構造体フィールドが取ることができる最小サイズの配列（1 変数につき 1 バイト、1 フィールドにつき 1 ビット）に圧縮します。

下記構造体では、フィールド x が圧縮されるので、x は a のすぐ次に続いて配置されます。

```
struct foo
{
 char a;
 int x[2] __attribute__ ((packed));
};
```

> **注：** dsPIC デバイスではワードは偶数バイト領域で配列されていますので、packed 属性を用いる際には、ランタイムアドレッシングエラーを避けるように注意しなければなりません。

persistent

persistent 属性は起動時に初期化したりクリアにすべきではない変数を指定します。persistent 属性を持つ変数は、デバイスがリセットされた後にも有効なままで残る状態情報の保存に使用できます。

```
int last_mode __attribute__ ((persistent));
```

reverse (*alignment*)

reverse 属性は変数の最終アドレスプラス 1 の最小の配列を指定します。配列はバイト単位で指定され、2 の累乗にする必要があります。reverse 配列の変数は dsPIC アセンブリ言語で減分 modulo バッファに使用できます。この属性は、アセンブリ言語でアクセス可能な C の変数をアプリケーションが定義する場合に有用です。

```
int buf1[128] __attribute__ ((reverse(256)));
```

reverse 属性は aligned 属性および section 属性と競合します。reverse 配列変数にセクションの指定をしようとしても無視され、警告が発生します。reverse 属性を持つ変数は auto_psv 領域（space() 属性または -mconst-in-code オプションを参照）に配置することはできません。そのような配置を試みると、警告が発生しコンパイラーはその変数を psv 領域に配置します。

section (*"section-name"*)

デフォルト設定では、コンパイラは生成されたオブジェクトを .data や .bss のようなセクション内に配置します。section 属性は、変数（もしくは関数）を特定のセクション内に配置するように指定することで、この動作を上書きします。

```
struct array {int i[32];}
struct array buf __attribute__ ((section( “userdata” ))) = {0};
```

section 属性は address 属性および reverse 属性と競合します。どちらと使用しようと試みても、セクション名は無視され警告が発生します。この属性は space 属性とも競合します。詳しくは space 属性の説明を参照してください。

sfr (*address*)

sfr 属性はコンパイラーに変数が二アデータ領域（データメモリの最初の 8 KB）に割り当てられていることを宣言し、*address* パラメータを用いて変数のランタイムアドレスを指定します。このような変数は二アデータ領域に割り当てられていない（または割り当てられているか分からない）変数よりも効率的にアクセスできる場合があります。

```
extern volatile int __attribute__ ((sfr(0x200)))u1mod;
```

エラーが発生しないようにするため、外部定義であることを指定する必要があります。

space (*space*)

通常、コンパイラーは変数を一般的なデータ領域に割り当てます。space 属性はコンパイラーに変数を特定なメモリ領域に割り当てるよう指示する場合に使用します。メモリ領域については、**セクション 4.6「メモリ空間」**で説明されます。space 属性では、以下の引数が受け入れられます。

data

一般的なデータ領域に変数を割り当てます。一般的なデータ領域内の変数には通常の C ステートメントを使用してアクセスできます。これはデフォルトの割り当てです。

xmemory

変数を X データ領域に割り当てます。X データ領域内の変数には通常の C ステートメントを使用してアクセスできます。xmemory 領域の割り当て例は以下の通りです。

```
int x[32] __attribute__ ((space(xmemory)));
```

ymemory

変数を Y データ領域に割り当てます。Y データ領域内の変数には通常の C ステートメントを使用してアクセスできます。ymemory 領域の割り当て例は以下の通りです。

```
int y[32] __attribute__ ((space(ymemory)));
```

prog

変数をプログラム領域の実行可能コードに指定されているセクションに割り当てます。プログラム領域内の変数は通常の C ステートメントを使用してアクセスできません。プログラマーにより明示的にアクセスする必要があり、通常、テーブルアクセスインラインアセンブリ命令またはプログラム空間可視化ウィンドウを使用してアクセスします。

auto_psv

コンパイラの管理下で、自動 Program Space Visibility(PSV) ウィンドウアクセスに指定されたセクションのプログラムメモリ領域に変数を割り当てます。auto_psv 内の変数は通常の C ステートメントを使用して読み取れ（書き込みは不可）、合計で最大 32K の割り当て空間があります。space(auto_psv) を指定する際は、section 属性を用いてセクション名を付けることはできません。セクション名は無視され、警告が発生します。auto_psv 空間内の変数は特定のアドレスまたは逆配列には配置できません。

> **注意：** auto_psv セクション内の変数は起動時にデータメモリにアップロードされません。この属性は RAM の使用率を削減するのに役立ちます。

psv

変数をプログラム空間の Program Space Visibility(PSV) ウィンドウアクセスのセクションに割り当てます。リンカーは PSVPAG レジスタを設定するだけですべての変数がアクセスできるようにセクションを割り当てます。PSV 空間内の変数はコンパイラーによって管理されず、通常の C ステートメントではアクセスできません。この空間にはプログラマが明示的にアクセスする必要があり、通常、テーブルアクセスインラインアセンブリ命令または PSV ウィンドウを使用します。

eedata

変数を EEData 空間に割り当てます。EEData 空間内の変数には通常の C ステートメントではアクセスできません。この空間にはプログラマが明示的にアクセスする必要があり、テーブルアクセスインラインアセンブリ命令または PSV ウィンドウを使用します。

transparent_union

この属性は、union という関数パラメータに付属して動作し、対応する引数がいずれかの union メンバのタイプを持ちますが、そのタイプが最初の union メンバのタイプであるかのように引渡しされます。その引数は、トランスペアレントな union の最初のメンバの呼び出し方法として関数に引き渡され、union そのものの呼び出し方法は用いません。この引数が引き渡されて正常動作するように、union のすべてのメンバは同じ機械語表現にする必要があります。

unordered

unordered 属性は、この変数の配置が現在の C ソースファイル内の他の変数に連携して相対的に移動してもよいことを示しています。これは ANCI C 準拠ではありませんが、リンカがメモリの隙間を利用しやすくなります。

```
const int __attribute__ ((unordered)) i;
```

unused

この属性は、変数に付属して動作し、変数が用いられない可能性があることを意味します。MPLAB C30 はこのような変数に対しては、未使用変数ワーニングを生成しません。

weak

weak 属性は、weak シンボルとして生成するように宣言を行います。weak シンボルはグローバル定義で取って代わられます。weak が、外部シンボルへの参照に適用される時には、リンキングは不要になります。例えば、以下のようになります。

```
extern int __attribute__((__weak__)) s;
int foo() {
  if (&s) return s;
  return 0; /* possibly some other value */
}
```

上記プログラムで、s が他のモジュールで定義されなければ、プログラムはリンクしますが、s はアドレスが与えられません。条件文は s が定義されていることをチェックし、もしあればその値を返します。そうでなければ、‘0’を返します。この特徴には多くの使用方法がありますが、ほとんどの場合はオプションライブラリと一緒にリンクされるジェネリックコードを生成することに使用されます。

weak 属性は変数と同様に関数へも適用できます。

```
extern int __attribute__((__weak__))
 compress_data(void *buf);
int process(void *buf) {
  if (compress_data) {
    if (compress_data(buf) == -1) /* error */
  }
  /* process buf */
}
```

上記のコードでは、関数 compress_data は、リンクされていれば、他のモジュールから使用可能です。この特徴の使用可否を決めるのはリンクタイム時であり、コンパイルタイム時ではありません。

定義時に weak 属性の与える影響はさらに複雑で、複数のファイルを記述する必要があります。

```
/* weak1.c */
   int __attribute__((__weak__)) i;

void foo() {

  i = 1;
}

/* weak2.c */
int i;

extern void foo(void);

void bar() {
  i = 2;
}

main() {
  foo();
  bar();
}
```

ここで、i の weak2.c での定義はシンボルにとって強い定義になります。リンクエラーは発行されず、両方の i's は同じ記憶位置を参照します。記憶位置は weak1.c 内の i に割り当てられますが、この領域はアクセスできません。

両方の i が同じデータ型を持つか否かのチェックはしません。weak2.c の i を変更して float タイプにしてもリンクはできますが、関数 foo の振る舞いは期待どおりにはならず、foo は 32 ビット浮動小数点の値の最下位部に値を書き込みます。逆に、weak1.c の weak 定義の i のタイプを変更して float にすると、悲劇的な結果になります。32 ビット浮動小数点の値を 16 ビット整数型領域に書き込み、i の直後にある変数を上書きしてしまいます。

weak 定義のみが存在する場合は、リンカは最初のその定義の記憶域を選択し、残りの定義はアクセスできません。

シンボルの型にかかわらず、動作は一致し、関数と変数は同じ様に動作します。

2.3.2 関数の属性を指定する

MPLAB C30 では、プログラムの中で呼ばれる関数についていくつかの事を宣言します。これにより、コンパイラが関数コールを最適化し、コードをより注意深くチェックする手助けになります。

キーワード __attribute__ により、宣言をする際に特殊な属性を規定することができます。このキーワードに続き、二重括弧内に属性を指定します。以下の属性が、関数用として現状サポートされています。

- address (addr)
- alias ("target")
- const
- deprecated
- far
- format (archetype, string-index, first-to-check)
- format_arg (string-index)
- near
- no_instrument_function
- noload
- noreturn
- section ("section-name")
- shadow
- unused
- weak

属性指定のやり方として __ (2 つのアンダースコア) をそれぞれのキーワードの前と後に置く (例えば ‘shadow' ではなく __shadow__) ことでも指定できます。これを用いれば、同じ名前を持つ可能性のあるマクロに関係なくヘッダファイルで属性を使用できます。

二重括弧内でコンマにより区切るか、一つの属性宣言に引続き他の属性宣言をすることで、宣言の中で複数の属性を指定することができます。

address (*addr*)

address 属性は、関数の絶対アドレスを指定します。この属性を section 属性と同時に使用することはできません。address 属性が優先します。

```
void foo() __attribute__ ((address(0x100))) {
...
}
```

alias (*"target"*)

alias 属性は、特定された別名を生成することを宣言します。

この属性を使用すると、ターゲットに対する外部参照になり、リンクフェーズで参照が解決されねばなりません。

const

多くの関数は引数以外の値は検査せず、戻り値以外には影響を与えません。そのような関数は、共通する副表記の省略や算術オペレータのようにループの最適化ができます。これらの関数は属性 const で宣言されなければなりません。例えば、

```
int square (int) __attribute__ ((const int));
```

は、双曲線関数 square は、プログラムが示すよりも少ない回数だけコールするほうが安全であることを示しています。

注意点として、ポインタ引数を持ち、ポイントされたデータを処理する関数は const 宣言してはなりません。同様に、non-const 関数は通例 const 宣言してはなりません。void な戻りタイプを持つ関数に const を適用しても意味がありません。

deprecated

deprecated 属性についての情報は、**セクション 2.3.1「変数属性の指定」**をご参照ください。

far

far 属性はコンパイラに対して、より効率のよい形式のコール命令を用いて関数がコールしないように通知します。

format (*archetype*, *string-index*, *first-to-check*)

format 属性は、関数が、フォーマット文字列に反していないかのタイプチェックを受ける printf, scanf もしくは strftime タイムスタイル変数を取ることを規定します。例えば、以下の宣言を考えてみましょう。

```
extern int
my_printf (void *my_object, const char *my_format, ...)
     __attribute__ ((format (printf, 2, 3)));
```

これによりコンパイラは、my_printf をコールするときの引数が、printf スタイルのフォーマット文字列関数 my_format に合っているかをチェックします。

パラメータ *archetype* はフォーマット文字列がどのように解釈されるかを決定し、printf, scanf もしくは strftime のいずれにするかを決定します。パラメータ string-index はどの引数がフォーマット文字列引数かを指定します（引数は 1 から始まり、左側より数えられます）。一方、*first-to-check* はフォーマット文字列と比較しチェックしようとする最初の引数の番号です。引数がチェックに使用できない場合（vprintf などの場合）、3 番目のパラメータを 0 に指定します。この場合、コンパイラーはフォーマット文字列の整合性のみをチェックします。

上記例では、フォーマット文字列 (my_format) は関数 my_printf の第二変数であり、チェックすべき変数は第三の変数から開始します。したがって、format 属性の正しいパラメータは 2 と 3 になります。

format 属性は、フォーマット文字列を引数とする独自関数を使うとき、MPLAB C30 がこれらの関数へのコールをエラーとしてチェックできるようにします。コンパイラは、(-Wformat を用いて) ワーニングが必要な時はいつでも常に ANSI ライブラリ関数 printf、fprintf、sprintf、scanf、fscanf、sscanf、strftime、vprintf、vfprintf、vsprintf 用のフォーマットチェックを行います。従ってヘッダファイル stdio.h を修正する必要はありません。

format_arg (*string-index*)

format_arg 属性は、関数が printf もしくは scanf スタイルの引数を取り、修正をかけ（例えば、他の言語に変換する等）、printf もしくは scanf スタイルの関数に渡すことを規定します。例えば以下の宣言を考えてみましょう。

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

これは結果を printf、scanf、strftime タイプ関数に渡す my_dgettext をコールする引数が、printf スタイルの文字列引数である my_format にあっているかをコンパイラにチェックさせます。

パラメータ *string-index* は、どの引数がフォーマット文字列引数であるかを規定します。(1 から開始)

format-arg 属性は、フォーマット文字列を修正する独自関数を作成したとき、MPLAB C30 が、独自関数をコールするオペランドを持つ printf, scanf もしくは strftime 関数コールをチェックできるようにします。

near

near 属性はコンパイラに対して、より効率のよい形式のコール命令を用いて関数がコールされうることを通知します。

no_instrument_function

通常コマンドラインオプション -finstrument-function が与えられると、プロファイリング関数コールが、ユーザーによりコンパイルされたほとんどの関数の出入口で生成されます。本属性を持つ関数はこの動作をしないようにします。

noload

noload 属性は空間を関数に割り当てますが、実際のコードはメモリにロードされません。この属性は、実行時にアプリケーションがシリアル EEPROM などから関数をメモリにロードするよう設計されている場合に有用です。

```
void bar() __attribute__ ((noload)) {
...
}
```

noreturn

abort や exit 等のいくつかの標準ライブラリ関数は値を返すことはできません。MPLAB C30 はこれを自動的に認知します。作成したプログラムで値を返すことがない独自関数のときは、noreturn を宣言することで、このことをコンパイラに通知することができます。例えば、以下のように記述します。

```
void fatal (int i) __attribute__ ((noreturn));

void
fatal (int i)
{
  /* Print error message. */
  exit (1);
}
```

noreturn キーワードは fatal が戻りのないことを前提としていることをコンパイラに通知します。fatal の過去の戻り値の有無に関わらず最適化を行います。これによりコードを改良することができます。また、初期化されていない変数による誤った警告を避けることができます。

noreturn 関数が void 以外の戻りタイプをもつことは無意味です。

section (*"section-name"*)

通常コンパイラは、生成したコードを .text セクション内に配置します。しかし、追加のセクションが必要となったり、特殊なセクションに関数が必要となる場合があります。Section 属性は関数が有効となる特定のセクションを指定します。例えば、以下の宣言を検討してみましょう。

```
extern void foobar (void)
__attribute__ ((section (".libtext")));
```

これは関数フーバーを .libtext セクションに配置します。

section 属性は address 属性と競合します。セクション名は無視され警告が発生します。

shadow

shadow 属性はコンパイラに対して、レジスタ待避を、ソフトスタックではなくシャドウレジスタを使用するよう指示します。この属性は通常 interrupt 属性と一緒に用いられます。

```
void __attribute__ ((interrupt, shadow)) _T1Interrupt (void)
interrupt [ ( [ save(list) ] [, irq(irqid) ]
[, altirq(altirqid)] [, preprologue(asm) ]
) ]
```

上記オプションは、指定された関数が割込みハンドラであることを示すために使用します。コンパイラーは、この属性が指定されている場合には、割込みハンドラ内で使用するのに適した形で関数の prologue と epilogue シーケンスを生成します。オプションパラメータ save は、関数の prologue と epilogue 内でそれぞれ保存、回復すべき変数のリストを規定します。オプションパラメータ irq と altirq は、使用される割り込みベクタテーブルの ID を指定します。オプションパラメータ preprologue は、コンパイラで生成される prologue コードの前に追加されるべきアセンブリコードを規定します。割込みの詳細説明と使用例については、**第 7 章「割り込み」**をご参照ください。

unused

この属性は関数に付随し、関数が使用されない可能性があることを意味します。MPLAB C30 はこの関数に対して unused 関数警告を生成しません。

weak

weak 属性の詳細については、**セクション 2.3.1「変数属性の指定」**をご参照ください。

2.3.3 インライン関数

関数 inline を宣言すると、MPLAB C30 が、関数コーラー用のコードにその関数コードを入れ込むように指示できます。これは通常、関数コールのオーバーヘッドを無くすことができプログラムの実行速度を上げることができます。さらに、実際の引数の値が定数である時は、既知の値をコンパイル時に単純化することにより、インライン関数のコードの一部を削減することができます。従ってコードサイズの影響は予測できません。機械語のサイズは、状況により、インライン関数とともに増えたり減ったりします。

注：	関数インライン化は関数の定義が明確 (プロトタイプのみではなく) な時にのみ発生します。一つ以上のソースファイルに関数をインライン化するには、それぞれのソースファイルにインクルードするヘッダファイルに関数定義を置くことになります。

関数 inline を宣言するには、以下のように、宣言文の中で inline キーワードを用います。

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(-traditional もしくは -ansi オプションを使用している場合は、 inline ではなく __inline__ を用いてください。) コマンドラインオプション -finline-functions を用いると、十分簡潔な関数インライン化ができます。コンパイラは、関数サイズの推定結果を基に、この方法で組込むにあたりどの関数が十分簡潔であるかを、自ら発見して決定します。

関数定義の中で、インライン置き換えには不向きなものがいくつかあります。これには、varargs, alloca, 可変長データ , 計算結果の goto、非ローカル goto を使う場合が含まれます。コマンドラインオプション -Winline は、インラインとマークされた関数が展開されない場合にはワーニングを発生し、その理由も出力します。

MPLAB C30 構文では、inline キーワードは関数のリンクには影響を与えません。

関数が inline でかつ static である場合、すべての関数コールがコーラーに組み込まれていて、関数のアドレスが使用されないとすると、関数自身のアセンブラコードは参照されません。このような場合、MPLAB C30 は、コマンドラインオプション -fkeep-inline-functions を指定しない限り、実際に関数のアセンブラコードを生成しません。いくつかのコールは種々の理由により組み込まれません。(特に、関数の定義より前に発生するコールや、定義の中の再帰コールは組み込まれません。) もし組み込まれていないコールが存在したら、関数は通常通りアセンブラコードにコンパイルされます。もしプログラムがその (関数) アドレスを参照するなら、通常の方法で関数もコンパイルされねばなりません。なぜなら、アドレスは inline 化されないからです。関数が static であると宣言され、関数定義が関数の使用以前になされていれば、コンパイラはインライン関数のみ消去します。

inline 関数が static でない場合、コンパイラは、別のソースコードからのコールが発生すると仮定します。グローバルシンボルはプログラム内で一度だけしか定義できないので、関数は他のソースファイル内では定義できず、従って、そのコールは組み込まれません。従って、非 static 関数は常に通常の方法でコンパイルされます。

関数定義で inline と extern を同時に指定した場合は、定義はインラインのみで使用されます。関数のアドレスを明確に参照しても、この場合は、関数はコンパイルされません。そのようなアドレスは、関数を宣言し定義しなかった場合のように、外部参照となります。

このように inline と extern の組合せは、マクロにも同様な影響を与えます。これらのキーワードと一緒にヘッダファイル内に関数定義を置き、ライブラリファイルには、別の定義のコピー (inline と extern の無いもの) を置きます。ヘッダファイル内の定義により、関数へのほとんどのコールはインライン化されます。残ったものは、ライブラリ内の 1 コピーを参照します。

2.3.4 レジスタ変数

MPLAB C30 はいくつかのグローバル変数に指定されたハードレジスタを使用するようにすることができます。

> **注：** レジスタ、とくに WO レジスタの多用は、MPLAB C30's のコンパイル機能を損なう恐れがあります。

普通のレジスタ変数が割り当てられるレジスタを指定することもできます。

- グローバルレジスタ変数はプログラムをでレジスタを予約します。これは、しばしばアクセスされる 2、3 のグローバル変数を持つプログラム言語翻訳器のようなプログラムでは役に立ちます。
- 特定のレジスタ内のローカルレジスタ変数はレジスタを予約しません。コンパイラのデータフロー解析により、指定されたレジスタが使用中か、その他の用途に使用できるかを決定することができます。ローカルレジスタ変数が未使用になったときそのレジスタ変数の内容は消去されます。ローカルレジスタ変数へのリファレンスは消去されるか、移動されるか、簡略化されます。

アセンブラ命令の出力を直接特定のレジスタに書き込みたい場合には、これらのローカル変数を拡張インラインアセンブルと使用すれば便利に使えます。（**第 8 章「アセンブリ言語と C モジュールの混用」**を参照ください）。（これは、インラインアセンブル記述内のオペランドに指定された制約に、指定レジスタが適合した場合に有効です。）

2.3.4.1 グローバルレジスタ変数の定義

MPLAB C30 でグローバルレジスタ変数を以下のように定義できます。

```
register int *foo asm ("w8");
```

ここで、w8 は使用されるレジスタの名前です。ライブラリルーチンが影響を与えないように、関数コールで通常保存回復されるレジスタ (W8-W13) を選択してください。

あるレジスタでグローバルレジスタ変数を指定すると、少なくとも現在のコンパイル内で、完全に本使用のためにそのレジスタを予約することができます。そのレジスタは、現在のコンパイルでは関数のその他の目的には割り当てられません。たとえ、無効になってもこのレジスタの内容は消去されませんが、レファレンスは消去、移動もしくは簡略化されます。

割り込みハンドラもしくは一つ以上のスレッドコントロールからグローバルレジスタ変数をアクセスするのは安全ではありません。なぜなら、（特別に当面のタスクのために再コンパイルしなければ）システムライブラリルーチンがレジスタを他の目的で一時的に使用するかもしれないからです。

グローバルレジスタ変数を使用する関数が、この変数の知識なし（すなわち、変数が宣言されていないソースファイルの中）にコンパイルされた第三の関数 `lose` を経由してグローバル変数を使用する関数 `foo` を呼ぶことは安全ではありません。これは `lose` がレジスタを保存し、他の値をそこに置くかもしれないからです。例えば、グローバルレジスタ変数が `qsort` へ渡す比較関数の中で使用できることは期待できません、なぜなら、`qsort` はそのレジスタに何か他のものを置くかもしれないからです。この問題は `qsort` を同じグローバルレジスタ変数定義と一緒に再コンパイルすることで回避することができます。

もし `qsort` もしくは、グローバルレジスタ変数を実際に使用しないその他のソースファイルを、再コンパイルして、その他の目的で当該レジスタを使用しないようにしたいなら、コンパイラにコマンドラインオプション `-ffixed-`*`reg`* を指定すれば十分です。実際にグローバルレジスタ宣言をソースコードに追加する必要はありません。

グローバルレジスタ変数の値を変更する関数は、この変数無しにコンパイルされた関数からは、安全にコールすることはできません、なぜなら、コーラーが戻り値として期待する値が破壊されているかもしれないからです。従って、グローバルレジスタ変数を使用するプログラム部分への入り口となる関数は、コーラーに従属する値を明確に保存・回復しなければなりません。

ライブラリ関数 longjmp は、setjmp の時に持っている値を、それぞれのグローバルレジスタ変数に回復します。

すべてのグローバルレジスタ変数宣言はすべての関数定義よりも先に行わなければなりません。もしそのような宣言が関数定義の後に現れると、レジスタは、先に定義された関数の中で、他の目的に使用されるかも知れません。

グローバルレジスタ変数は初期値を持ちません、なぜなら、実行ファイルはレジスタに初期値を与える手段を持たないからです。

2.3.4.2 ローカル変数用のレジスタ指定

指定されたレジスタでローカルレジスタ変数を以下のように定義できます。

```
register int *foo asm ("w8");
```

ここで、w8 は使用されるレジスタの名前です。これはグローバルレジスタ変数を定義する構文と同じことに注意してください。但し、ローカル変数の場合は、関数の中に現れます。

そのようなレジスタ変数を定義しても、レジスタは予約できず、フロー制御により変数の値が有効でないと判定したときは、別の用途に利用できるようになっています。この特徴により、残っているレジスタが、ある関数をコンパイルするには少なすぎるということを避けることができます。

このオプションは、指定するレジスタ内に変数を割り当てるコードを、MPLAB C30 が常に生成することを保証するものではありません。このレジスタへの明示的な参照を asm ステートメントにコード化すべきではなく、この変数を常に参照するようにして下さい。

ローカルレジスタ変数に対する割り当ては、使用されないことが明らかになった時に消去されます。ローカルレジスタ変数へのリファレンスは消去、移動もしくは簡略化されます。

2.3.5 複素数

MPLAB C30 は複素数データを扱うことができます。キーワード __complex__ 用いることで、複素整数型、複素浮動小数点型を宣言できます。

例えば、__complex__ float x により、x の実部、虚部がともに float 型の変数であることを宣言できます。__complex__ short int y は、y の実部、虚部がともに short int 型の変数であることを宣言しています。

複素数データ型の定数を記述するには、添え字 ‘i’ もしくは ‘j’ (どちらか一つ；どちらも同等です。) を使用します。例えば、2.5fi は __complex__ float 型であり、3i は __complex__ int 型です。このような定数は純粋に虚数値ですが、実数定数を付加することでどんな複素数値も作ることができます。

複素数値の表記 *exp* の実数部を抽出するには、__real__ *exp* と記述します。同様に虚数部を抽出するには、__imag__ を使用します。例えば、以下のように書きます

```
__complex__ float z;
float r;
float i;

r = __real__ z;
i = __imag__ z;
```

演算子 ‘~’ は、複素数型の値を扱う場合には、複素共役を生成します。

MPLAB C30 は複素自動変数を、非連続的に割り当てます。実数部分をレジスタに、虚数部分をスタックに（逆も可能）設定することも可能です。デバッグ情報フォーマットでは、このような非連続的な割り当てを表現することはできません。従って、MPLAB C30 では非連続な複素変数を、非複素数型の別々な二つの変数として記述します。もし変数の実際の名前が foo ならば、二つの仮想変数は foo$real と foo$imag というように名づけられます。

2.3.6 倍長整数

MPLAB C30 は long int の２倍の長さを持つ整数型をサポートしています。符号付き整数は long long int と記述し、符号なし整数は、unsigned long long int と記述します。long long int 型の整数定数を作るには、整数に添え字 LL を付加します。unsigned long long int, 型の整数定数を作るには、整数に添え字 ULL を付加します。

これらのデータ型は他の整数型のように算術的に使用できます。これらの型の、加算、減算、ビットのブール演算はオープンコードですが、割り算、シフト演算はオープンコードではありません。オープンコードではない演算には MPLAB C30 で提供する特別ライブラリを使用します。

2.3.7 typeof による型参照

表現型を参照する別の方法としては、typeof キーワードがあります。このキーワードを使用する構文としては sizeof に似ていますが、構成は、typedef で定義される型名のように働きます。

typeof に対する引数を記述するには二つの方法があります。表現で記述する方法と型で記述する方法です。表現で記述する例は以下の通りです。

```
typeof (x[0](1))
```

これは、x が関数の配列であり、型は関数そのものの型になります。

変数の型名での記述例は以下の通りです。

```
typeof (int *)
```

ここでは、記述された型は int に対するポインタです。

もし ANSI C プログラムに含めなければならないヘッダファイルを記述するならば、typeof ではなく、__typeof__ と書いてください。

typeof は typedef 名が使用できるところならどこでも使用できます。例えば、宣言内、キャスト内、sizeof もしくは typeof 内です。

- これは、x が指すタイプで y を宣言します。
  ```
  typeof (*x) y;
  ```
- これは、上記の値の配列として y を宣言します。
  ```
  typeof (*x) y[4];
  ```
- これは文字列へのポインタの配列として y を宣言します。
  ```
  typeof (typeof (char *)[4]) y;
  ```
  これは、以下の従来の C 宣言と同等です。
  ```
  char *y[4];
  ```

typeof を用いた宣言の意味を確認し、なぜそれが記述するのに便利な方法なのか、これらのマクロを書き直してみましょう。

```
#define pointer(T) typeof(T *)
#define array(T, N) typeof(T [N])
```

この宣言は以下のように書き直すことができます。

```
array (pointer (char), 4) y;
```

ここで、array (pointer (char), 4) は char に対する 4 つのポインタの配列です。

2.4 文の差異

本セクションでは、通常の ANSI C と MPLAB C30 で受け付けられる C との文の差異について述べます。文の差異は基本 GCC の部分であり、本章での説明は標準 GCC ドキュメントを基にして行い、GCC の MPLAB C30 移植の特定の構文と記号に合わせていきます。

- 値としてのラベル
- 省略されたオペランドを持った条件文
- CASE 範囲

2.4.1 値としてのラベル

現在の関数（もしくは含まれている関数）内でオペレータ‘&&’とともに定義されるラベルのアドレスを入手できます。その値は `void *` のタイプを持っています。この値は定数でありその型の定数が有効であるところではどこでも使用できます。例えば、以下のように設定します。

```
void *ptr;
...
ptr = &&foo;
```

これらの値を使って直接ジャンプできます。これは計算された `goto` 文 , `goto *exp` で実現できます。例えば、以下のように設定します。

```
goto *ptr;
```

タイプ `void *` のどんな表現も可能です。

これらの定数を使用する方法の一つは、ジャンプテーブルとして提供されるスタティック配列を初期化することにあります。

```
static void *array[] = { &&foo, &&bar, &&hack };
```

そうすれば、以下のように、インデックスでラベルを選択できます。

```
goto *array[i];
```

> **注：** これは、添え字が範囲内かどうかはチェックしません。(C 内の配列インデックスはしません。)

ラベル値の配列は `switch` 文の目的と同様な働きをします。`switch` 文はクリーナーであり配列には望ましいものです。

ラベル値の別の使い方は、スレッデッドコード用の翻訳にあります。翻訳関数内のラベルは、スレッデッドコードに保存され高速な分岐処理ができるようにします。

このメカニズムは別の関数内のコードにジャンプするという誤用をされかねません。コンパイラはこれが起こることを防止できませんので、ターゲットアドレスが現在の関数内で有効であることを十分注意して確認しなければなりません。

2.4.2 省略されたオペランドを持った条件文

条件記述の中の中間オペランドは省略できる場合があります。もし第 1 オペランドがゼロでないなら、その値は条件記述の値になります。

従って、例えば

```
x ? : y
```

は、x がゼロでないなら、x の値を、そうでなければ y の値をとります。

この例は、以下の式と全く同等です。

```
x ? x : y
```

この簡単な例の場合、中間のオペランドを省略できる能力は、特に有益ではありません。これが有益になるのは、第 1 オペランドが副作用を持つか、もしくは（マクロ変数の場合）持つかもしれない場合です。オペランドを繰り返し中間の値とするのは二倍の副作用をもたらすことになります。中間オペランドを省略するのは、再計算するという望ましくない影響無しにすでに計算された値を使うということです。

2.4.3 CASE 範囲

単一 CASE ラベル内で連続する値の範囲を、以下のように指定することができます。

```
case low ... high:
```

これは、個別 CASE ラベルの適切な個数、すなわち、*low* から *high* までの一つ一つの整数値、と同じ効果を持ちます。

この特徴は ASCII 文字コードの範囲に対して特に有効です。

```
case ‘A’ ... ‘Z’:
```

注意 : ..., の前後にスペースを入れてください。そうでないと、整数変数をともに使用されたときに誤って構文解析されます。例えば、以下のように記述します。

```
case 1 ... 5:
```

以下のようには記述してはなりません。

```
case 1...5:
```

2.5 表現の差異

このセクションでは、通常の ANSI C と、MPLAB C30 で受け入れられる C 言語の表現の差異を説明します。

2.5.1 バイナリ定数

Ob または OB（‘b’または‘B’の前に数字‘0’が付きます）が付いたバイナリ数のシーケンスはバイナリ整数として解釈されます。バイナリ数は数字‘0’と‘1’から成ります。たとえば、10 進数 255 は Ob11111111 と記述されます。

他の整定数と同様に、バイナリ定数の末尾には、サインされていないことを示す‘u’または‘U’が付きます。または、long であることを示す‘1’または‘L’が末尾に付くこともあります。接尾辞‘ll’または‘LL’が付くと、long long バイナリ定数であることを示します。

第 3 章 MPLAB C30 C コンパイラを使用する

3.1 序章

本章ではコマンドライン上での MPLAB C30 C コンパイラの使用方法について説明します。MPLAB® IDE と一緒に MPLAB C30 を使用する方法については、*dsPIC® Language Tools Getting Started* (DS70094) をご参照ください。

3.2 ハイライト

本章で説明する内容は以下の通りです。

- 概要
- ファイル名規則
- オプション
- 環境変数
- コマンドライン上での一つのファイルのコンパイル
- コマンドライン上での複数のファイルのコンパイル

3.3 概要

コンパイルドライバプログラム (`pic30-gcc`) は、C 言語とアセンブル言語とライブラリアーカイブをコンパイル、アセンブル、リンクします。ほとんどのコンパイラのコマンドラインオプションは、GCC ツールセットのすべての実装と共通です。いくつかのオプションが MPLAB C30 コンパイラ独自のものです。

コンパイラコマンドラインの基本形は以下の通りです。

```
pic30-gcc [options] files
```

注:	コマンドラインオプションとファイル名拡張子は大文字と小文字を区別します。

使用可能なオプションは**セクション 3.5「オプション」**に記載されています。

下記例は、C ソースファイル `hello.c` をコンパイル、アセンブル、リンクして、絶対実行ファイル `hello.exe` を生成します。

```
pic30-gcc -o hello.exe hello.c
```

3.4 ファイル名規則

コンパイラドライバは以下のようなファイル拡張子を認識し、この拡張子は大文字と小文字を区別します。

表 3-1: ファイル名

拡張子	定義
file.c	プリプロセスされるべきC ソースファイル
file.h	ヘッダーファイル（コンパイルもリンクも実行されない）
file.i	プリプロセスの必要ないC ソースファイル
file.o	オブジェクトファイル
file.p	前処理後のアセンブリ言語ファイル
file.s	アセンブラコード
file.S	プリプロセスされるべきアセンブラコード
other	リンカに渡されるファイル

3.5 オプション

MPLAB C30 はコンパイルを制御するために多くのオプションを持っており、それらはすべて大文字と小文字を区別します。

- dsPIC デバイスに特有のオプション
- 出力類の制御オプション
- C の方言を制御するオプション
- ワーニングとエラーの制御オプション
- デバッグ用オプション
- 最適化を制御するオプション
- プリプロセッサを制御するオプション
- アセンブラのオプション
- リンク用オプション
- ディレクトリ検索用オプション
- コード生成変換用オプション

3.5.1 dsPIC デバイスに特有のオプション

メモリモデルに関するより詳細な情報については、**セクション 4.7「メモリモデル」**をご覧ください。

表 3-2: dsPIC デバイスに特有のオプション

オプション	定義
`-mconst-in-code`	定数を auto_psv 領域に置きます。コンパイラは PSV ウインドウを用いてこれらの定数をアクセスできます。**（これはデフォルトです）**
`-mconst-in-data`	定数をデータメモリ領域に置きます。
`-merrata=`*`id[,id]*`*	このオプションは *`id`* で認識される dsPIC 特有のエラッタ対策を有効にします。時折生じる `id` 変更の有効な値で、ある種の変更には必要とされません。「*`id`* of `list`」は現在サポートされているエラッタ識別子をそのエラッタの簡単な説明とともに示します。「*`id`* of `all`」は現在サポートされているエラッタ対策すべてを有効にします。
`-mlarge-code`	ラージコードモデルを使用してコンパイルします。呼び出された関数の局所性については問いません。 このオプションを選択すると、32k 以上のサイズのシングル関数はサポートされずアセンブリタイムエラーが発生します。関数内のすべてのブランチがショート型であるためです。
`-mlarge-data`	ラージデータモデルを使用してコンパイルします。静的変数と外部変数の場所については問いません。
`-mpa`(1)	プロシージャ抽象化の最適化を有効にします。ネストのレベルに制限はありません。
`-mpa=`*`n`*(1)	レベル *n* までのプロシージャ抽象化の最適化を有効にします。*n* がゼロの場合、最適化は無効化されます。*n* が 1 の場合、抽象化は第 1 レベルで許可され、ソースコード内の命令シーケンスはサブルーチンに抽象化されます。n が 2 の場合、第 2 レベルの抽象化が許可され、最初のレベルでサブルーチンに抽象化された命令は 1 レベル分深く抽象化されます。このパターンは *n* の値が 3 以上でも同様に続きます。 正味の効果としては、サブルーチン呼び出しネストの深さが *n* の値を上限に制限されます。
`-mno-pa`(1)	プロシージャ抽象化の最適化を有効にしません。 **（これはデフォルトです）**
`-momf=`*`omf`*	コンパイラーで使用する OMF（Object Module Format）を選択します。*`omf`* は、次のいずれかになります： `coff` COFF オブジェクトファイルを生成します。**（これはデフォルトです）** `elf` ELF オブジェクトファイルを生成します。 ELF オブジェクトに使用するデバッグフォーマットは DWARF 2.0 になります。
`-msmall-code`	スモールコードモデルを使用してコンパイルします。呼び出される関数は近接（コーラの 32k ワード以内）と想定されます。**（これはデフォルトです）**

注 1: プロシージャ抽象化はインラインの逆の動作をします。動作はコンパイル単位中の複数のサイトから共通コードシーケンスを抽出するようにデザインされており、抽出したシーケンスをコードの共通領域に配置します。このオプションは一般的には、生成されたコードのランタイムパフォーマンスを向上させませんが、コードのサイズを明らかに節約できます。`-mpa` でコンパイルされたプログラムはデバッグが困難ですので、このオプションは COFF オブジェクトフォーマットを使用したデバッグの際には使用しないようお奨めします。
プロシージャ抽象化は、アセンブリファイル生成後にコンパイルの個別のフレーズとして呼び出されます。このフレーズは複数のコンパイル単位にわたる最適化は行いません。プロシージャ最適化フレーズが有効化されると、インラインアセンブリコードは有効なマシン命令のみに制限されます。無効なマシン命令または命令シーケンス、またはアセンブラ擬似命令（セクショニングディレクティブ、マクロ、インクルードファイルなど）は使用できませんし、これらを使用すると、プロシージャ抽象化フレーズはうまく動作せず、出力ファイルの生成が禁止されます。

表 3-2: dsPIC デバイスに特有のオプション （続き）

オプション	定義
-msmall-data	スモールデータモデルを使用してコンパイルします。すべての静的変数および外部変数はデータメモリ領域の 8 KB より下に配置されると想定されます。**（これはデフォルトです）**
-msmall-scalar	-msmall-data のように、静的スケーラと外部スケーラ以外はデータメモリ領域の 8 KB より下に配置されると想定されます。**（これはデフォルトです）**
-mtext=*name*	-mtext=*name* を指定するとテキスト（プログラムコード）がデフォルトの .text セクションではなく、*name* という名前のセクションに配置されます。= の前後に空白スペースは入れません。
-msmart-io [=0|1|2]	このオプションは printf、scanf、に渡される format 文字列、およびそれらの f と v のバリエーションについて分析をします。小数点なしの引数を使用すると、整数のみのライブラリ関数バリエーションに変換されます。 -msmart-io=0 はこのオプションを無効化し、-msmart-io=2 はコンパイラの検査を緩めにし関数呼び出しを変数または未知のフォーマットの引数に変換します。-msmart-io=1 **はデフォルトで**、明確な定数字のみを変換します。

注 1: プロシージャ抽象化はインラインの逆の動作をします。動作はコンパイル単位中の複数のサイトから共通コードシーケンスを抽出するようにデザインされており、抽出したシーケンスをコードの共通領域に配置します。このオプションは一般的には、生成されたコードのランタイムパフォーマンスを向上させませんが、コードのサイズを明らかに節約できます。-mpa でコンパイルされたプログラムはデバッグが困難ですので、このオプションは COFF オブジェクトフォーマットを使用したデバッグの際には使用しないようお奨めします。
プロシージャ抽象化は、アセンブリファイル生成後にコンパイルの個別のフレーズとして呼び出されます。このフレーズは複数のコンパイル単位にわたる最適化は行いません。プロシージャ最適化フレーズが有効化されると、インラインアセンブリコードは有効なマシン命令のみに制限されます。無効なマシン命令または命令シーケンス、またはアセンブラ擬似命令（セクショニングディレクティブ、マクロ、インクルードファイルなど）は使用できませんし、これらを使用すると、プロシージャ抽象化フレーズはうまく動作せず、出力ファイルの生成が禁止されます。

3.5.2 出力類の制御オプション

表 3-3: 出力類の制御オプション

オプション	定義
-c	ソースコードをコンパイルもしくはアセンブルしますが、リンクはしません。デフォルトの出力ファイルの拡張子は .o です。
-E	プリプロセスの後、つまりコンパイルが実行される前に停止します。デフォルトの出力ファイルは stdout です。
-o *file*	*file* 内に出力をします。
-S	適切なコンパイルの後、つまりアセンブラ呼び出しの前に停止します。デフォルトの出力ファイル拡張子は .s です。
-v	コンパイルの各ステージで実行されたコマンドを印刷します。
-x	-x オプションを使用して明示的に入力言語を指定できます。 -x *language* 後続の入力ファイルの言語を明示的に指定します（コンパイラーにファイル名接尾語によって決まるデフォルト値を選択させるのではなく）。このオプションは次の -x オプションまで後続の入力ファイルすべてに適用されます。以下の値が MPLAB C30 でサポートされています。 c c-header cpp-output assembler assembler-with-cpp -x none 言語仕様をオフにし、後続のファイルがそのファイル名接尾辞に従って処理されるようにします。これはデフォルトの動作ですが、他の -x オプションがすでに使用されている場合に必要です。 例： pic30-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s 上記例では -x none オプションがない場合、コンパイラはすべての入力ファイルはアセンブラが処理すべきものと解釈します。
--help	コマンドラインオプションの説明を印刷します。

3.5.3 C の方言を制御するオプション

表 3-4: C の方言制御オプション

オプション	定義
`-ansi`	ANSI 規格 C プログラムすべて（のみ）をサポート。
`-aux-info filename`	ヘッダーファイルも含め、コンパイル単位内の宣言されたおよび / または定義されたすべての関数について、プロトタイプ宣言しているファイル名を出力します。このオプションは C 以外の言語では無視されます。宣言に加え、ファイルはコメントによって各宣言（ソースファイルと行）が暗示的なものか、プロトタイプか、プロトタイプ以外かを示し（`I` と `N` は新規、`O` は既存で、行番号とコロンの後の最初にこれらの文字が付きます）、それが宣言によるものか、定義によるものかを示します（2 番目の文字が `C` または `F` になります）。関数定義の場合、宣言に続く引数の K&R スタイルリストがコメント内の宣言の後ろに付きます。
`-ffreestanding`	独立した環境でコンパイルが行なわれることを表明します。これは `-fno-builtin` を意味します。独立した環境には、標準のライブラリが存在するとは限らず、プログラム起動が必ずしもメインにはなりません。最も顕著な例は OS カーネルです。これは `-fno-hosted` と同等です。
`-fno-asm`	`asm, inline` もしくは `typeof` をキーワードと認識しないため、コードの中でこれらの単語を識別子として使用できます。代わりに `__asm__, __inline__` および `__typeof__` をキーワードとして使用します。`-ansi` は暗黙の内に `-fno-asm` を指定します。
`-fno-builtin` `-fno-builtin-`*`function`*	`__builtin_` を接頭辞としない組み込み関数を認識しません。
`-fsigned-char`	型 `char` に符号を付け、`signed char` のようにします。**（これはデフォルトです）**

`-fsigned-bitfields`
`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`
	このオプションは、宣言が符号付き・符号なしのどちらも指定しない場合、ビットフィールドを符号付き・符号なしのどちらにするかを制御します。デフォルトでは、`-traditional` が使用されない限り、このようなビットフィールドは符号付きになります。`-traditional` を使用する場合は常に符号なしになります。

`-funsigned-char`	型 `char` の符号をはずし、`unsigned char` のようにします。
`-fwritable-strings`	書き込み可能データセグメント内に文字列を保存し、その文字列が一意にならないようにします。

3.5.4 ワーニングとエラーの制御オプション

ワーニングは、本来はエラーではないが、危険でエラーになるかもしれないことを示す、診断メッセージです。

-W で始まるオプションで多くの特別なワーニングをリクエストすることができます。例えば、-Wimplicit は、暗黙の宣言についてのワーニングをリクエストします。これらの特別なワーニングオプションも -Wno- で始まる否定形を持っており、ワーニングの出力を停止します。例えば、-Wno-implicit 等です。このマニュアルでは、二つの形の内のデフォルトではない一つのみを記述しています。

以下のオプションは、MPLAB C30 C コンパイラで生成されるワーニングの量と種類を制御します。

表 3-5: -WALL に含まれるワーニング / エラーオプション

オプション	定義
-fsyntax-only	コードの構文をチェックしますが、それ以上のことはしません。
-pedantic	厳密な ANSI C によって要求されるすべてのワーニングを発行し、禁止された拡張子を使用したプログラムを拒絶します。
-pedantic-errors	ワーニングよりもエラーを発行する点を除けば、-pedantic と同様です。
-w	すべてのワーニングメッセージを禁止します。
-Wall	この表でリストされたすべての -W オプションを組み合わせたものです。これにより、マクロの結合内でも、ユーザによっては疑わしく見える構文で、容易に回避（もしくはワーニング出力が出ないように修正することが）できる構文についてもワーニングが出るようになります。
-Wchar-subscripts	配列添え字に char 型を使っている場合にワーニングを出します。
-Wcomment -Wcomments	コメント開始シーケンスである /* が /* スタイルのコメントに現れた時、もしくは Backslash-Newline が // スタイルのコメント内に現れた時は必ずワーニングを発生します。
-Wdiv-by-zero	コンパイル時にゼロでの整数の割算が発生した場合にワーニングを発生します。ワーニングメッセージを禁止するには -Wno-div-by-zero を使用します。ゼロでの浮動小数点の割算はワーニングを発生しません。それは、無限値や NaNs を得るのに理論的な方法であるからです。**（これはデフォルトです。）**
-Werror-implicit-function-declaration	関数が宣言以前に使用された場合にエラーを発生します。
-Wformat	printf や scanf に対するコールをチェックし、与えられた引数が指定されたフォーマット文に対して適切な型であることを確認します。
-Wimplicit	-Wimplicit-int と -Wimplicit-function-declaration の両方を指定することと等価です。
-Wimplicit-function-declaration	関数が宣言以前に使用された場合にワーニングを発生します。
-Wimplicit-int	宣言において型が指定されていない場合にワーニングを発生します。
-Wmain	main の型が疑わしい場合にワーニングを発生します。main は int 型の値を返し、ゼロ、２つもしくは３つの適切な型の引数をもつ外部結合の関数です。

-Wmissing-braces	集合体もしくはユニオンの初期化が完全に括弧でくくられていない時にワーニングを発生します。以下の例では、a についてのイニシャライザは完全に括弧でくくられていませんが、b については完全に括弧でくくられています。
	`int a[2][2] = { 0, 1, 2, 3 };`
	`int b[2][2] = { { 0, 1 }, { 2, 3 } };`

表 3-5: -WALL に含まれるワーニング / エラーオプション （続き）

オプション	定義

-Wmultichar
-Wno-multichar
	複数文字の character 定数が使用された時にワーニングを発生します。通常、そのような定数はタイプミスです。このような定数は実装依存の値を持っているので、移植性のあるコード内では使用すべきではありません。以下の例では複数文字の character 定数の使用例を示しています。
	char
	xx(void)
	{
	return('xx');
	}

-Wparentheses	ある文脈の中で括弧が省略されているとワーニングを発生します。たとえば、真値が期待されている文脈の中で代入がある場合や、演算子が入れ子になっていて、後から見るとその優先度についてしばしば混乱するような場合です。
-Wreturn-type	関数が、int をデフォルトの戻り値の型として定義されている場合にワーニングを発生します。戻り値の型が void でない関数で、戻り値を持たない return 文がひとつでもあればワーニングを発生します。
-Wsequence-point	C 標準の中でシーケンスポイントに違反するような未定義の意味を持つコードに関してワーニングを発生します。 C 標準は、C プログラム内での表現をシーケンスポイントの観点から評価して順序を規定します。そのシーケンスポイントはプログラムの部分ごとに、シーケンスポイント以前に実行されるものと、以後に実行されるものとに分けて順序を表示しています。これらは、以下の場合に発生します。full expression（larger expression の一部ではないもの）の評価の後、第一オペランド &&, ||, ? : の評価の後、もしくは（コンマ）オペレータ、関数がコールされる前（但しその引数とコールされる関数を意味する表現の評価の後）などの場合です。 シーケンスポイントルールで表現されたもの以外に、表現の sub expressions の評価の順番は規定できません。これらすべてのルールは全体的な順序というよりも、部分的な順序のみを記述しています。たとえば、もし二つの関数がそれらの間のシーケンスポイントのない一つの表現でコールされると、関数がコールされる順番は規定できません。 オブジェクトの値に対する修正が、シーケンスポイント間で、有効になった場合、これは規定できません。動作がこれに依存するようなプログラムでは、動作は不定となります。C 標準は、“前段と次のシーケンスポイント間で、オブジェクトは、表現の評価により一度だけストアされた値を修正できます。さらに、前値は、ストアされるべき値を決定するためにのみ読み出される”ということを規定しています。もしプログラムがこのルールを破ったら、このような実装の結果はまったく予想できません。 未定義の動作を持ったコードの例は、a = a++;, a[n] = b[n++] and a[i++] = i; です。いくつかのより複雑な場合はこのオプションでは診断できません、そしてそれは、ときどき間違えて正しいという結果を出しますが、それでも一般には、このような問題をプログラムの中で検出するのにかなり有効であると判っています。

表 3-5: -WALL に含まれるワーニング / エラーオプション （続き）

オプション	定義
-Wswitch	switch 文が列挙型のインデックスを持つ場合、その列挙型により名前付けされた定数に対応する case が不足している場合に常に警告を発します。（default のラベルが存在する場合、この警告は発生しません）同様に、このオプションが使用されている場合に列挙型の範囲外の case ラベルがあると、警告が発生します。
-Wsystem-headers	システムヘッダーファイル内構文が見つかった場合 Print 警告メッセージが発生します。システムヘッダーファイルからの警告は通常、実際的な問題を意味していないと想定されて、コンパイラ出力が難解になるだけなので、出力しないようにします。本コマンドラインオプションは、MPLAB C30 にシステムヘッダーからの警告をユーザーコード内で発生したものと同じように除外するよう指示します。ただし、このオプションに関連して -Wall を使用してもシステムヘッダー内の未知の pragma に対して警告は発生しません。そのため、-Wunknown-pragmas も同時に使用する必要があります。
-Wtrigraphs	３文字表記があると警告を発生します（有効化されていると想定して）。
-Wuninitialized	初期化なしに自動変数が使用されると、警告を発生します。この警告は、最適化の際にのみ評価されるデータフロー情報を必要とするので、最適化が有効になっている場合にのみ生成可能です。レジスタ割り当て候補の変数に対してのみ警告が発生します。従って、volatile として宣言された変数やアドレスが取得されている変数、サイズが 1、2、4、8 バイト以外の変数に対しては警告は発生しません。また、構造体、共用体、配列に対しては、レジスタ化されているものであっても警告は発生しません。値を計算するためだけに使用される変数に対しては、その計算結果自体が使われることがなければ警告が発生しないことがあります。そのような計算は警告が印刷される前にデータフロー分析により削除されることがあるためです。
-Wunknown-pragmas	MPLAB C30 が理解不能な #pragma ディレクティブがある場合に警告を発生します。このコマンドラインオプションを使用すると、システムヘッダーファイル内の未知の pragma に対しても警告が発生します。ただし、-Wall コマンドラインオプションにより有効化されている場合は警告は発生しません。
-Wunused	宣言以外に変数が使用されない場合、関数が静的に宣言されながら定義されていない場合、ラベルが宣言されて使用されない場合、ステートメントが明示的に使用されない結果を計算する場合、必ず警告が発生します。使用されていない関数パラメータに関する警告を得るには、-W と -Wunused の両方を指定する必要があります。void で式をキャストすると、その式に対する警告は抑制されます。同様に、unused 属性は、使用されていない変数、パラメータ、ラベルに対する警告を抑制します。
-Wunused-function	静的関数が宣言されながら、定義されていない場合、または非インライン静的関数が使用されていない場合に必ず警告を発生します。
-Wunused-label	ラベルが宣言されながら使用されていない場合に警告を発生します。この警告を抑制するには、unused 属性を使用します（**セクション 2.3.1「変数属性の指定」**参照）。

表 3-5: -WALL に含まれるワーニング / エラーオプション （続き）

オプション	定義
-Wunused-parameter	関数パラメータが宣言以外に使用されていない場合、必ず警告を発生します。この警告を抑制するには、unused 属性を使用します（**セクション 2.3.1「変数属性の指定」**参照）。
-Wunused-variable	ローカル関数または非定数の静的関数が宣言以外に使用されていない場合に必ず警告を発生します。この警告を抑制するには、unused 属性を使用します（**セクション 2.3.1「変数属性の指定」**参照）。
-Wunused-value	ステートメントが明示的に使用されていない結果を計算した場合に必ず警告を発生します。この警告を抑制するには、void の表現をキャストします。

以下の -W オプションは、-Wall には含まれません。そのうちいくつかは、ユーザが通常疑わしいと思わない（たまにはチェックしたいと思うかもしれない）構文について警告を発生します。構文に関する別の警告は、ある場合はさけることが必要もしくは困難なものであり、警告を制限するようにコードを修正するのは簡単ではありません。

表 3-6: -WALL に含まれないワーニング / エラーオプション

オプション	定義
-W	以下のイベントに対する警告メッセージを印刷します。 • volatile 宣言されていない自動変数が、longjmp をコールすることで変更されることがあります。これらの警告は最適化コンパイル時のみ可能です。コンパイラは setjmp へのコールのみ観察しています。どこで longjmp がコールされるかはわかりません。実際、シグナルハンドラはコードの中でどのポイントでもコールされます。その結果、longjmp は問題が発生する場所でコールされることは実際にはあり得ず、事実上問題ない時にも警告が発生することがあります。 • 関数は、戻り値を持って復帰することも、戻り値を持たずに復帰することもできるようになっています。関数本体が終端に達することは、戻り値を持たずに復帰するとみなされます。 • 式文もしくはコンマ式の左側の部分が全く副作用を持たない。警告を制限するには使われない式を void でキャストします。例えば x[i,j] のような式は警告を発生させますが、x[(void)i,j] では発生しません。 • 符号無し値がゼロと < もしくは <= で比較されている。 • x<=y<=z のような比較が行われている。これは (x<=y ? 1 :0) <= z と同等であり、通常の算術記述とは異なる解釈になります。 • static のような記憶クラス指定子は宣言の中で最初に記述されていない。C 標準によれば、この使い方は陳腐化しています。 • -Wall もしくは -Wunused が規定されると、未使用関数に関する警告が発生します。 • 符号付き値が符号無し値に変換されると、符号付き、符号無しの値間での比較は間違った結果を生み出す可能性があります。（ただし、-Wno-sign-compare が一緒に指定されると、警告は発生しません。）

表 3-6: -WALL に含まれないワーニング / エラーオプション

オプション	定義

	• 集合体の初期化を囲む括弧が部分的にしか記述されていない。例えば、以下のコードは、x.h のイニシャライザの前後に大括弧がないので、その警告を発生します。
	`struct s { int f, g; };`
	`struct t { struct s h; int i; };`
	`struct t x = { 1, 2, 3 };`
	• 集合体はイニシャライザを持ち、すべてのメンバを初期化していない。例えば、以下のコードは、x.h が暗示的にゼロに初期化されるので、その警告を発生します。
	`struct s { int f, g, h; };`
	`struct s x = { 3, 4 };`

-Waggregate-return	構造体や共用体を返す関数が定義されるかコールされた場合に警告を発生します。
-Wbad-function-cast	適合しない型に対して関数コールがキャストされた場合に警告を発生します。例えば、int foof() * にキャストされた時に警告を発生します。
-Wcast-align	ターゲットに必要な配列が増加するように、ポインタがキャストされた場合に警告を発生します。例えば、char* に対して、int * がキャストされた時に警告が発生します。
-Wcast-qual	ポインタの指す型から型修飾子を削除するようにポインタがキャストされた場合に必ず警告が発生します。例えば、const char * に対して、通常の char * がキャストされた時に警告が発生します。
-Wconversion	プロトタイプが無い状態で同じ引数に行ったものとは別の型変換を、プロトタイプが行った場合に警告を発生します。これには固定小数点から浮動少数点への変換、もしくはその逆の変換を含みます。また、デフォルトの拡張と同じ場合以外の固定小数点引数の幅もしくは符号付与の変更変換も含みます。さらに、負の整数定数式が暗示的に符号なしタイプに変換されると、警告を発生します。例えば、x が符号なしの場合､代入 x = -1 に関して警告を発生します。(unsigned) -1 のような明確にキャストされている場合は警告を発生しません。
-Werror	すべての警告をエラーにします。
-Winline	関数が inline 宣言されるか、もしくはそれが -finline-functions オプションとされていて、その関数が inline 化できなかった場合、警告を発生します。
-Wlarger-than-*len*	*len* バイト以上のオブジェクトが定義された場合に必ず警告を発生します。
-Wlong-long -Wno-long-long	long long 型が使用されたときに警告が発生します。これはデフォルトです。警告を禁止するには -Wno-long-long を使用します。-Wlong-long と -Wno-long-long フラグは、-pedantic フラグが使用された場合に考慮されます。
-Wmissing-declarations	グローバル関数が事前の宣言なしに定義された時に警告が発生します。定義自体がプロトタイプを生成しても警告を発生します。
-Wmissing- format-attribute	-Wformat が有効になると、フォーマット属性用の候補になるかもしれない関数について警告を発生します。これらは可能性のある候補のみであって、絶対候補ではないことにご注意ください。このオプションは -Wformat が有効にならない限り有効になりません。

表 3-6: -WALL に含まれないワーニング / エラーオプション

オプション	定義
-Wmissing-noreturn	noreturn 属性の候補になるかもしれない関数についての警告を発生します。これらは可能性のある候補であって、絶対的な候補ではありません。noreturn 属性を追加する前に、十分に注意して関数が実際に復帰することがないということを手作業で検証するべきです。そうでなければ、コード生成上の微妙なバグが入るかもしれません。
-Wmissing-prototypes	グローバル関数が事前のプロトタイプ宣言無しに定義された時に警告を発生します。この警告は、定義自体がプロトタイプを生成しても発生します。(このオプションは、ヘッダファイル内で宣言されなかったグローバル関数を検出するために用いられます。)
-Wnested-externs	extern 宣言が関数内で見つかった時に警告を発生します。
-Wno-deprecated-declarations	deprecated マークされた関数、変数、型の使用については警告を発生しません。
-Wpadded	パディングが、構造体の要素か構造体全体を揃えるためにパディングが行われた場合に警告を発生します。
-Wpointer-arith	関数型のサイズもしくは void のサイズに依存するものに関して警告を発生します。MPLAB C30 は、計算するときの便宜上、void * ポインタと関数に対するポインタ型に 1 のサイズを割り当てます。
-Wredundant-decls	複数宣言が有効で、何も変化させない場合でも、同じスコープ内で一度以上宣言されたら警告を発生します。
-Wshadow	ローカル変数が別のローカル変数を隠す時に警告を発生します。
-Wsign-compare -Wno-sign-compare	符号付値が符号無しに変換された時に、符号付 / 符号無し値間の比較結果が、正しくない結果を生じた時に警告を発生します。この警告は -W; でも有効になります。この警告なしに、他の -w の警告が必要なときは、-W -Wno-sign-compare を使用します。
-Wstrict-prototypes	引数の型を指定しない状態で、関数が宣言もしくは定義された場合に警告を発生します。(古い形の関数定義では、引数の型を指定する宣言が前にあれば、警告を発生することなく、許可されています。)
-Wtraditional	伝統的な C と ANSI C の中での動作が異なるような構文について警告を発生します。 • マクロ本体の中のストリング定数内に現れるマクロ引数。これらは、伝統的な C 内では引数に置き換わりますが、ANSIC 内では定数の一部となります。 • あるブロックで外部宣言された関数でブロックの終端より後ろに使用されるもの。 • long 型のオペランドを持つスイッチステートメント。 • 非静的な関数宣言が静的な関数宣言に続いているもの。この構成はいくつかの伝統的な C コンパイラでは受け入れられません。
-Wundef	未定義の識別子が #if ディレクテブ内で評価された場合に警告を発生します。
-Wunreachable-code	実行されないであろうコードを、コンパイラが検出した場合に警告を発生します。これはこのオプションが、警告を受けたラインの一部が実行される状況になっても警告を発生することがあります。従って、明らかに到達しないコードを削除する際には注意が必要です。例えば、関数がインライン化された時に、一つだけのインライン化された関数のコピー内で到達しないことを、警告が意味することもあります。

表 3-6: -WALL に含まれないワーニング / エラーオプション

オプション	定義
-Wwrite-strings	non-const char * ポインタへのアドレスのコピーにより警告が発生するように、ストリング定数に const char[length] 型を与えます。これらの警告により、ストリング定数へ書き込みしようとするコードを、コンパイル時に見つけることができますが、宣言やプロトタイプ内の定数を使用するときに十分な注意が払われている場合だけ有効です。そうでなければ、迷惑なだけです。-Wall がこれらの警告の要求をしなかったのはこのためです。

3.5.5 デバッグ用オプション

表 3-7: デバッグ用オプション

オプション	定義
-g	デッバグ情報を生成します。 MPLAB C30 は -g オプションを -0 オプションと一緒にサポートし、最適化されたコードをデバッグするようにすることができます。コードの最適化によって、時に驚くべき効果がもたらされます。 – いくつかの宣言された変数は全く存在しないかもしれません。 – 制御フローは予期しなかったところへ一時的に移動することがあります。 – いくつかの文は定数の結果を計算するか、もしくはそれらの値はすでに入手済みということで、実行されない場合があります。 – いくつかの文は、ループの外に移動されているので、別の場所で実行されるかもしれません。 それにもかかわらず、最適化された出力をデバッグが可能です。このため、バグがある可能性のあるプログラムに最適化を実行することは合理的であるといえます。
-Q	コンパイル時にそれぞれの関数名をコンパイラが印刷し、コンパイルが終了する時にそれぞれのパスについての統計値を印刷します。
-save-temps	中間ファイルを消去しません。それらを、現ディレクトリに置き、ソースファイルに基づいて名前をつけます。従って、‘foo.c’を ‘-c -save-temps’でコンパイルすると、以下のようなファイルを生成します。 ‘foo.i’ (事前処理されたファイル) ‘foo.p’ (事前手続きで抽象化されたアセンブリ言語ファイル) ‘foo.s’ (アセンブリ言語ファイル) ‘foo.o’ (オブジェクトファイル)

3.5.6 最適化を制御するオプション

表 3-8: 一般的な最適化オプション

オプション	定義
-O0	最適化しません。 **(これがデフォルトです。)** -O が無いと、コンパイラのゴールは、コンパイルのコストを低減し、デバッグが期待された結果を生成させることになります。ステートメントは独立しています。ステートメント間のブレークポイントでプログラムを停止させると、どんな変数に対しても新しい値を代入したりもしくは関数内の他のステートメントに移るようプログラムカウンタを変更したりできます。そしてソースコードから期待する結果そのものが得られます。コンパイラはレジスタに register 宣言された変数を割り当てるのみです。
-O -O1	最適化します。最適化コンパイルはいくらか時間がかかり、大きい関数に対してはより多くのホストメモリを使用します。-O をつけると、コンパイラーはコードサイズと実行時間を減らそうとします。 -O, が設定されるとコンパイラーは -fthread-jumps と -fdefer-pop をオンします。また -fomit-frame-pointer をオンします。
-O2	より最適化をかけます。MPLAB C30 はスピードとサイズのトレードオフを伴わないほとんどすべてのサポートされた最適化を実行します。-O2 は、ループ展開や (-funroll-loops) と関数インライン展開 (-finline-functions) と厳密なエイリアス最適化 (-fstrict-aliasing) を除くすべての最適化オプションをオンにします。また、メモリオペランドの強制コピーと、(-fforce-mem) フレームポインタ削除 (-fomit-frame-pointer) をオンにします。-O に比べて、このオプションはコンパイルに要する時間を増加させ、生成されたコードのパフォーマンスを向上させます。
-O3	さらに最適化をかけます。-O3 は、-O2 で規定されるすべての最適化をオンし、inline-functions オプションもオンします。
-Os	サイズの最適化をします。-Os は、コードサイズを増加させない、すべての -O2 最適化を可能にします。これ以外にコードサイズを低減させるように設計された最適化も実行します。

以下のオプションは特定の最適化を制御します。-O2 オプションは -fun-roll-loops, -funroll-all-loops と -fstrict-aliasing を除いて、これらのすべての最適化をオンします。

以下のフラグは、最適化の細かい調整（“fine-tuning”）が必要となるまれなケースで使用できます。

表 3-9: 特別な最適化オプション

オプション	定義
-falign-functions -falign-functions=*n*	*n* より大きい次の 2 のべき乗値境界に、関数の開始を揃え、*n* バイトまでスキップアップします。例えば、-falign-functions=32 は次の 32 バイト領域に関数を揃えます。ただし、-falign-functions=24 は、23 バイト以下をスキップすることにより可能な場合のみ、次の 32 バイトに揃えます。 -fno-align-functions と -falign-functions=1 は同じで、関数が整列されないことを意味します。 アセンブラは、*n* が 2 のべき乗の時にのみこのフラグをサポートしますので、*n* は切り上げられます。*n* が指定されないときは、マシンに依存するデフォルト値を使用します。
-falign-labels -falign-labels=*n*	すべての分岐ターゲットを、-falign-functions のように、*n* バイトまでスキップして、2 のべき乗境界に揃えます。このオプションはコードを遅くします。分岐ターゲットが通常のコードフロー内に到達した場合にダミーオペレーションを挿入しなければならないためです。 もし -falign-loops もしくは -falign-jumps が適用されこの値より大きな場合、それらの値が、代わりに使用されます。*n* が指定されない場合は、マシンに依存するデフォルト値（1 になる場合がほとんど）を使用しますが、これは整列をしないことを意味します。
-falign-loops -falign-loops=*n*	ループを、-falign-functions のように、*n* バイトまでスキップして、2 のべき乗境界に揃えます。ループが数多く実行され、ダミーオペレーション実行の代わりとなることが期待されます。*n* が指定されないときは、マシンに依存するデフォルト値を使用します。
-fcaller-saves	関数コールで上書きされるレジスタがあるような場合、そのようなコードに関して、レジスタを保存・回復するための命令を追加します。そのような割り当ては、それ以外の場合に生成されるコードより良いコードになると思われる場合のみ実行されます。
-fcse-follow-jumps	共通部分式を除去する際に、ジャンプ命令のジャンプ先が、そのジャンプ命令以外のパスからは到達されない場合、そのジャンプ命令のジャンプ先を調べます。例えば、CSE が else 文を伴った if ステートメントを見つけた場合、CSE は条件が偽である時のジャンプを追跡します。
-fcse-skip-blocks	これは -fcse-follow-jumps と似ていますが、CSE をブロック条件付きでスキップするジャンプを CSE に追跡させます。CSE が単純 if 文で else ステートメントの無いものを見つけた場合、-fcse-skip-blocks は CSE を、if の本体を飛び越すジャンプを追跡させます。
-fexpensive-optimizations	比較的コストのかかる重要ではない最適化を行います。
-ffunction-sections -fdata-sections	それぞれの関数もしくはデータ項目を出力ファイル自身のセクションに置きます。関数の名前もしくはデータ項目の名前は出力ファイルのセクションの名前を決定します。大きな効果がある場合にのみこれらのオプションを使用します。これらのオプションを指定する時にはアセンブラとリンカーは大きなオブジェクトと実行ファイルを生成し、速度は遅くなります。

表 3-9: 特別な最適化オプション

オプション	定義
-fgcse	グローバル共通部分式除去のパスを実行します。このパスはグローバル定数とコピー伝播も実行します。
-fgcse-lm	-fgcse-lm が有効になると、グローバル共通部分式除去が、自分自身へのストアでのみ消されるロードを移動させようとします。これにより、ロード・ストアシーケンスを含むループがループの外のロードに変更され、ループ内はコピー・ストア に変更されます。
-fgcse-sm	-fgcse-sm が有効になると、ストアモーションパスが、グローバル共通部分式除去の後に実行されます。このパスは、ストアをループ外に移動させようとします。 -fgcse-lm と一緒に使用されると、 ロード・ストアシーケンスを含むループは、ロードはループの前に、ストアはループの後に変更されます。
-fmove-all-movables	ループ内のすべての不変の計算はループ外に移動されます。
-fno-defer-pop	関数が戻ってきたらすぐに関数コールに対して引数を POP します。コンパイラは通常、引数をいくつかの関数コール用のスタックに蓄積し、一度にすべて POP します。
-fno-peephole -fno-peephole2	マシン特有のピープホール最適化を無効にします。ピープホール最適化はコンパイル中の様々な場面で発生します。 -fno-peephole はマシン命令に関するピープホール最適化を無効にし、-fno-peephole2 は高度なピープホール最適化を無効にします。ピープホールを完全に無効化するには、これら両方を使用します。
-foptimize-register-move -fregmove	レジスタ結合を最大化するために move 命令内および、単純命令のオペランドのレジスタ番号を再割当します。 -fregmove と -foptimize-register-moves は同じ最適化です。
-freduce-all-givs	ループ内のすべての汎用帰納変数の能力を低減させます。これらのオプションによりコードが良くなることもあるし、悪くなることもあります。結果はソースコード内のループの構造に高く依存します。
-frename-registers	レジスタ割り当て後にあまったレジスタを利用することで、予定されたコード内でできなかったところを再度割り当てようとします。この最適化は多くのレジスタを持つプロセッサに利益をもたらします。しかし、これによりデバッグができなくなります。変数がもとのレジスタ（“home register”）にとどまっていないためです。
-frerun-cse-after-loop	ループ最適化が実行されたあとに共通部分式除去を再実行します。
-frerun-loop-opt	ループ最適化を 2 度実行します。
-fschedule-insns	dsPIC の Read-After-Write ストール（詳しくは *dsPIC30F Family Reference Manual* を参照してください）を無くすために命令を並べ直します。特に、コードサイズに影響を与えずにパフォーマンスを向上させます。
-fschedule-insns2	-fschedule-insns と同様ですが、レジスタ割り当てが完了した後に命令スケジュールの追加実行を要求します。
-fstrength-reduce	ループの能力低減と繰り返し変数の除去の最適化を実行します。

表 3-9: 特別な最適化オプション

オプション	定義

-fstrict-aliasing	コンパイルされる言語に適用できる最も厳密なエイリアシング規則を、コンパイラに適用します。C 言語については、これが、型表現を基にした最適化を活性化します。特に、ある型のオブジェクトが、ほとんど同じ型でないならば、別の型のオブジェクトとして同じアドレスのところには無いものとされます。例えば、unsigned int は int の別名になり得るが、void* もしくは double にはなりえません。文字型は別の型になりえます。
	以下のようなコードには特に注意してください。
	`union a_union {`
	`  int i;`
	`  double d;`
	`};`
	
	`int f() {`
	`  union a_union t;`
	`  t.d = 3.0;`
	`  return t.i;`
	`}`
	最も最近書き込まれた（“type-punning”と呼ばれる）ものとは異なったユニオンメンバから読み出すことはよくあることです。もしメモリがユニオン型でアクセスされるなら、-fstrict-aliasing を持った type-punning は許されます。従って、上記コードは期待通りに動作します。しかし、このコードは以下のものにはなりません。
	`int f() {`
	`  a_union t;`
	`  int* ip;`
	`  t.d = 3.0;`
	`  ip = &t.i;`
	`  return *ip;`
	`}`

-fthread-jumps	比較によるジャンプの分岐先において、最初の比較に包含されるような別の比較があるかどうかをチェックするように最適化が実行されます。もしそうなると、条件の真か偽が判った時に、最初の分岐は 2 番目の分岐もしくは直後の分岐先のうち、どちらかの分岐先に再設定されます。
-funroll-loops	ループ解除の最適化を実行します。これは、繰り返し回数がコンパイル時もしくは実行時に決定されるループのためだけに使用され、-funroll-loops は -fstrength-reduce と -frerun-cse-after-loop の両方を含みます。
-funroll-all-loops	ループ解除の最適化を実行します。すべてのループに適用され、通常はプログラムの実行をより遅くします。 -funroll-all-loops は -fstrength-reduce と -frerun-cse-after-loop を含みます。

形式 -f*flag* のオプションはマシンに依存しないフラグを指定します。ほとんどのフラグは正・負の形式を持っており、-ffoo の負形式は -fno-foo です。以下の表では、一つの形式のみリストアップされています。(デフォルトではない方)

表 3-10: マシンに依存しない最適化オプション

オプション	定義
-fforce-mem	メモリオペランドを、算術計算する前にレジスタにコピーします。この手順は、すべてのメモリリファレンスを共通部分式にすることで、より良いコードを生成します。それらが、共通部分式では無い時には、命令の組合せで、個別にレジスタに値をロードするようなことをすべきではありません。-O2 オプションはこのオプションをオンします。
-finline-functions	すべての単純関数をそのコーラーにインライン展開します。コンパイラは、どの関数がこのような方法で組込むのに十分単純であるかを決定します。与えられた関数のすべてのコールに展開され、関数が static であると宣言された場合、関数は通常は独立したアセンブラコードとして出力されません。
-finline-limit=*n*	デフォルトでは、MPLAB C30 はインラインされる関数のサイズを制限します。このフラグはインラインと (inline キーワードで) 明確にマークされた関数に対する制限をコントロールします。n はインライン化することのできる関数のサイズを、仮想命令数を単位として示したものです。(パラメータ処理の部分は含まれません)。*n* のデフォルト値は 10000 です。この値を増加すると、コンパイル時間とメモリ使用量は増えますが、インライン化されるコードを増やすことができます。 減少させるとコンパイル時間を早くさせ、インライン化されたコードを少なくします(プログラムの速度は落ちると想定されます。)このオプションは特にインライン化を多く使用するプログラムには有効です。 **注:** 仮想命令は、この特別なコンテキストでは、関数のサイズの抽象的な尺度を意味します。アセンブリ命令のカウント値を表すものではなく、その正確な意味はコンパイラーのリリースごとに変更される可能性があります。
-fkeep-inline-functions	与えられた関数に対するすべてのコールがインライン展開されて、関数が static であると宣言されても、別々の関数のコール可能な実行バージョンを出力します。このスイッチは、extern のインライン関数には影響をあたえません。
-fkeep-static-consts	変数が参照されていなくても、最適化がオンされていない場合に static-consts と宣言された変数を出力します。MPLAB C30 はこのオプションをデフォルトで有効にしています。最適化がオン / オフにかかわらず、変数が参照されたかどうかをコンパイラにチェックさせるには、-fno-keep-static-consts オプションを使用します。
-fno-function-cse	関数アドレスをレジスタに置かず、ある決まった関数をコールするおのおのの命令に関数のアドレスを明示的に含ませます。このオプションを使用するとコード効率は良くありませんが、アセンブラ出力を変更するいくつかの奇妙な技巧は、このオプションを使用しないと、実行される最適化により混乱をきたすことがあります。

表 3-10: マシンに依存しない最適化オプション

オプション	定義
-fno-inline	inline キーワードを無視します。通常このオプションは、コンパイラが inline の関数を拡大するのを防ぐのに使用されます。最適化が有効になっていない場合、関数は全くインライン展開されません。
-fomit-frame-pointer	フレームポインタを必要としない関数のレジスタ内のフレームポインタを保持しません。これにより、命令がフレームポインタを保存・セットアップ・回復することを避けることができます。多くの関数で余分のレジスタを利用できるようにします。
-foptimize-sibling-calls	シブリングとテイル再帰コールを最適化します。

3.5.7 プリプロセッサを制御するオプション

表 3-11: プリプロセッサオプション

オプション	定義

-A*question* (*answer*)	#if *#question(answer)* のようなプリプロセッシング条件でテストされた場合、質問 *question*, に対する回答 *answer* を診断します。-A- は通常ターゲットマシンを記述している標準の診断を無効にします。
	例えば、メインプログラムの関数プロトタイプは以下のように宣言されます。
	#if #environ(freestanding)
	int main(void);
	#else
	int main(int argc, char *argv[]);
	#endif
	-A コマンドラインは二つのプロトタイプから選択するために使用されます。例えば、二つのうちの最初の方を選択するには、以下のようなコマンドラインオプションが使用されます。
	-Aenviron(freestanding)

-A *-predicate =answer*	述部 *predicate* と回答 *answer* の診断をキャンセルします。
-A *predicate =answer*	述部 *predicate* と回答 *answer* の診断を発行します。この形式は、依然としてサポートされている古い形式 -A predicate(answer), よりも好まれます。なぜなら、これはシェル特別文字を使用しないからです。
-C	コメントを廃棄しないようプリプロセッサに指示します。-E オプションと一緒に使用します。
-dD	プリプロセッサに、マクロの正しい順序ですべてのマクロ定義を出力に渡すようプリプロセッサに指示します。
-D*macro*	string 1 を持ったマクロ *macro* を定義します。
-D*macro=defn*	マクロ *macro* を *defn* と定義します。コマンドラインの -D のすべての インスタンスを、-U オプションの以前に処理します。
-dM	プリプロセッシングの終了時に有効になるマクロ定義のリストのみを出力するように、プリプロセッサに指示します。-E オプションと一緒に使用されます。
-dN	マクロ引数とコンテンツが省略されることを除いて、-dD と同様です。#define name のみが出力に含まれます。
-fno-show-column	診断時に、コラム番号を印刷しません。もし診断が、dejagnu のようにコラム番号がわからないようなプログラムによりスキャンされるならば、これは必要かもしれません。
-H	通常動作に加えて、使用されているヘッダファイルの名前を印刷します。

表 3-11: プリプロセッサオプション （続き）

オプション	定義
-I-	-I オプションの前に -I- オプションを指定する、どんなディレクトリも、#include “*file* ” の場合のみに検索され、#include <*file*> の場合には検索されません。 もし追加されるディレクトリが、-I の後に -I- オプション付きで指定されると、これらのディレクトリはすべての #include 命令で検索されます。（通常は、すべての -I ディレクトリはこのように使用されます。） さらに、-I- オプションは（現入力ファイルが入っている）現ディレクトリを、#include “file” で最初に検索するディレクトリとして禁止します。-I- のこの効果を上書きすることはできません。-I を用いると、コンパイラが起動した時の現ディレクトリの検索を指示できます。それは、プリプロセッサがデフォルトで実行するものと正確に同じではありません。しかし、それはしばしば満足できるものです。 -I- はヘッダファイルの標準システムディレクトリの使用を禁止するものではありません。従って、-I- と -nostdinc は独立です。
-I*dir*	ヘッダファイルを検索するために、ディレクトリのリストの頭に、デレクトリ *dir* を付加します。これによりシステムヘッダファイルを上書きすることができます。なぜならこれらのディレクトリはシステムヘッダファイルディレクトリ以前に検索されるからです。もし 1 個以上の -I オプションを使用すると、ディレクトリは左から右へスキャンされ、その後に標準システムディレクトリがスキャンされます。
-idirafter *dir*	ディレクトリ *dir* を第二のインクルードパスに追加します。第二のインクルードパスは、メインのインクルードパス（-I を追加されたパス）のディレクトリ内でヘッダファイルが見つからなかった時に検索されます。
-imacros *file*	通常の入力ファイルを処理する前に、ファイルを入力として処理し、出力は廃棄します。ファイルから生成された出力は廃棄されるので、-imacros *file* の唯一の効果は、メインの入力で使用できるファイルの中でマクロを定義することにあります。コマンドライン上の -D や -U オプションは、それらが書き込まれる順番にかかわらず、-imacros *file* 以前に常に処理されます。すべての -include と -imacros オプションは、それらが書き込まれた順番に処理されます。
-include *file*	通常の入力ファイルを処理する前に、ファイルを入力として処理します。ファイルの内容が先にコンパイルされます。コマンドライン上の -D や -U オプションは、それらが書き込まれる順番にかかわらず、-include *file* 以前に常に処理されます。すべての -include と -imacros オプションは、それらが書き込まれた順番に処理されます。
-iprefix *prefix*	*prefix* を次の -iwithprefix オプションとして指定します。
-isystem *dir*	第二のインクルードパスの始めにディレクトリを追加し、標準システムディレクトリに適用されるものと同じ扱いを受けるため、それをシステムディレクトリとして印をつけます。
-iwithprefix *dir*	第二のインクルードパスの始めにディレクトリを追加します。ディレクトリの名前はプリーフィックスと *dir* を結合することで作られます。ここで、プリーフィックスとは事前に -iprefix で指定されます。もしプリーフィックスが指定されていなければ、コンパイラのインストールパスを含んだディレクトリがデフォルトとして使用されます。
-iwithprefixbefore *dir*	メインのインクルードパスにディレクトリを追加します。ディレクトリの名前は、-iwithprefix の場合と同様に、prefix と *dir* を結合することで作られます。

表 3-11: プリプロセッサオプション （続き）

オプション	定義
-M	それぞれのオブジェクトファイルの依存性を記述するのに適した規則を出力するように、プリプロセッサに指示します。それぞれのソースファイル用に、ターゲットがソースファイルのオブジェクトファイル名であり、依存性が、使用するすべての #include ヘッダーファイルであるようなメイクルールをプリプロセッサが出力します。このルールは単一ラインであるか、もしロングで長ければ \-newline で続きます。ルールのリストは、事前処理された C プログラムではなく、標準出力上に印刷されます。 -M は -E を含みます。（**セクション 3.5.2「出力類の制御オプション」**参照）。
-MD	-M と同様ですが依存性情報はファイルに書き込まれ、コンパイルは継続します。依存性情報を含んだファイルは .d 拡張子を持ったソースファイルと同じ名前を与えられます。
-MF *file*	-M もしくは -MM と一緒に用い依存性が書き込まれるファイルを指定します。-MF スイッチが与えられない場合、事前処理された出力を置くのと同じ場所に、プリプロセッサがルールを置きます。ドライバオプション -MD または -MMD, -MF を用いると、デフォルトの依存性出力ファイルに上書きします。
-MG	無くなったファイルを生成されたファイルのように扱い、ソースファイルと同じディレクトリ内に存在すると仮定します。-MG が指定されると、-M もしくは -MM も指定されねばなりません。 -MG は -MD もしくは -MMD と一緒にはサポートされません。
-MM	-M と同様ですが､出力は #include “*file*” に含まれたユーザヘッダファイルのみ言及します。#include <*file*> に含まれたシステムヘッダファイルは省略されます。
-MMD	-MD と同様ですが、システムヘッダファイルではなく、ユーザヘッダファイルのみ言及します。
-MP	このオプションは、CPP に、メインファイルとは別の依存性に、偽のターゲットを追加するように指示し、なにものにも依存しないようにします。これらのダミールールは、適合すべきメイクファイルを更新することなしにヘッダファイルを削除したら、弾力性を生成するように働きます。下記は、典型的な出力です： test.o: test.c test.h test.h:
-MQ	-MT と同じですが、特別な Make 用の文字を引用します。-MQ ‘$(objpfx)foo.o' は次となります。 $$(objpfx)foo.o: foo.c デフォルトのターゲットは、-MQ で与えられるかのように自動的に引用されます。
-MT *target*	依存性生成により作成されたルールのターゲットを変更します。デフォルトでは、CPP は、どのようなパスも含むメインの入力ファイルの名前を取り、.c のようなファイル添え字を削除し、プラットフォームの通常のオブジェクト添え字を添付します。結果がターゲットです。-MT オプションはターゲットを、指定したストリングと同じものに設定します。複数のターゲットが必要なら、それを -MT に対する単一引数として指定するか、もしくは複数の -MT オプションを使用します。 例えば、: -MT ‘$(objpfx)foo.o’ は $(objpfx)foo.o: foo.c を生成します。

表 3-11: プリプロセッサオプション （続き）

オプション	定義
-nostdinc	ヘッダーファイル用の標準システムディレクトリを検索しません。-I オプションで指定したディレクトリ（および、もし適切であれば現ディレクトリ）のみが検索されます。-I については、（**セクション 3.5.10「ディレクトリ検索用オプション」**参照）-nostdinc と -I- の両方を使用することで、インクルードファイル検索パスは明示的に指定されたディレクトリのみに限定されます。
-P	#line ディレクティブを生成しないようにプリプロセッサに指示します。-E オプションと一緒に使用されます。（**セクション 3.5.2「出力類の制御オプション」**参照）。
-trigraphs	ANSI C の三重文字をサポートします。-ansi オプションはこれを有効にします。
-U*macro*	マクロ *macro* を定義解除し、-U オプションはすべての -D オプションの後ですが、-include オプションや -imacros オプションより先に評価されます。
-undef	（アーキテクチャフラグを含み）非標準のマクロの事前定義をしません。

3.5.8 アセンブラのオプション

表 3-12: アセンブラオプション

オプション	定義
-Wa,*option*	*option* をオプションとして、アセンブラに渡します。*option* がコンマを含んでいる場合、コンマの部分で複数のオプションに分割されます。

3.5.9 リンク用オプション

オプション -c, -S もしくは -E が使用されると、リンカは実行されず、オブジェクトファイル名は引数として使用されません。

表 3-13: リンク用オプション

オプション	定義
-L*dir*	*dir* をコマンドラインオプション -l で指定されるライブラリを検索すべきディレクトリのリストに追加します。
-l*library*	リンク時に *library* と名づけられたライブラリを検索します。 リンカーは標準のライブラリのディレクトリのリストをサーチします。ライブラリのファイル名は、lib*library*.a です。リンカーはこのファイルを正確な名前で指定された時のように使用します。このオプションをコマンドのどこに記述するか、および、リンカープロセスライブラリ、オブジェクトファイルの指定の順番によって違いが現れます。foo.o -lz bar.o はファイル foo.o の後にライブラリ z をサーチしますが、それは bar.o のサーチの前になります。bar.o が libz.a 内の関数を参照した場合、これら関数はロードされません。 検索されたディレクトリはいくつかの標準システムディレクトリと -L で指定されるものを持っています。 通常このようにして見つかったファイルはライブラリファイル（アーカイブファイルで、そのメンバがオブジェクトファイルです。）です。リンカはアーカイブファイルをスキャンし、参照されるが定義されていないシンボルを定義するメンバを探します。しかし、ここで見つかったのが通常のオブジェクトファイルならば、通常の方法でリンクされます。-l オプション (すなわち -lmylib) を使用することと、ファイル名 (すなわち libmylib.a) を指定することの唯一の違いは、-l が、指定された通りにいくつかのディレクトリを検索することです。 デフォルトでは、リンカは、-l オプションで指定されたライブラリを求めて :<install-path>\lib を検索するように指示されます。デフォルト位置にインストールされたコンパイラには、これは c:\pic30_tools\lib です。この動作は、**セクション 3.6「環境変数」**で定義される環境変数を使うことで上書きされます。
-nodefaultlibs	リンク時に標準システムライブラリを使用しません。指定されたライブラリのみがリンカに渡されます。コンパイラは memcmp, memset と memcpy に対してコールを生成します。通常、これらのエントリは標準コンパイラライブラリ内のエントリとして解決されます。これらのエントリポイントは、このオプションが指定される時に何か他のメカニズムにより与えられるべきです。
-nostdlib	リンク時に標準システムスタートアップファイルもしくはライブラリを使用しません。スタートアップファイルは渡されないし、指定されたライブラリのみがリンカに渡されます。コンパイラは memcmp, memset と memcpy に対してコールを生成します。通常、これらのエントリは標準コンパイラライブラリ内のエントリとして解決されます。これらのエントリポイントは、このオプションが指定される時に何か他のメカニズムにより与えられるべきです。
-s	実行可能ファイルからシンボル表や再配置情報を削除します。
-u *symbol*	*symbol* を未定義として、ライブラリモジュールを強制的にリンクさせこの記号を定義します。-u を異なる記号で複数回使用し追加のライブラリモジュールを強制的にロードするのは不正ではありません。
-Wl,*option*	*option* をオプションとしてリンカに渡します。*option* がコンマを含んでいたら、コンマの位置で複数のオプションに分割されます。
-Xlinker *option*	*option* をオプションとしてリンカに渡します。これを使用して、システムに特有なリンカオプションで MPLAB C30 が認識できないオプションを与えることができます。

3.5.10 ディレクトリ検索用オプション

表 3-14: ディレクトリ検索用オプション

オプション	定義
-B*prefix*	このオプションは、実行可能ファイル、ライブラリ、インクルードファイル、コンパイラー自身のデータファイルを探す場所を指示します。コンパイラードライバプログラムは、1 つ以上のサブプログラム、pic30-cpp, pic30-cc1, pic30-as および pic30-ld を実行します。prefix を、それぞれのプログラムの接頭辞としてとらえ、実行させます。 サブプログラムを実行するために、コンパイラドライバは最初に、もしあれば -B prefix を試みます。サブプログラムが見つからない場合、もしくは、-B が指定されていない場合は、ドライバは PIC30_EXEC_PREFIX 環境変数内にある値を使用します。詳細は**セクション 3.6「環境変数」**を参照ください。最後に、ドライバは、サブプログラム用の現状 PATH の環境変数を探します。 効果的にディレクトリの名前を指定する -B 接頭辞はリンカーのライブラリにも適用されます。コンパイラーがこれらのオプションをリンカー用の -L オプションに変換するためです。さらにそれらはプリプロセッサ内のインクルードファイルにも適用します。コンパイラがこれらのオプションを、プリプロセッサ用の -i システムオプションに変換するためです。この場合、コンパイラーはインクルードを接頭辞に付与します。-B 接頭辞と同様に接頭辞を指定する別の方法は、環境変数 PIC30_EXEC_PREFIX を使用することです。
-specs=*file*	コンパイラが標準仕様ファイルを読み込んだ後にファイルを処理します。それはどのスイッチが pic30-cc1, pic30-as, pic30-ld 等に渡されるべきかを決定する際に、pic30-gcc ドライバプログラムが使用するデフォルトを上書きするためです。一つ以上の -specs=*file* がコマンドライン上で指定され、左から右へと順番に処理されます。

3.5.11 コード生成変換用オプション

形式 -*fflag* のオプションはマシンに依存しないフラグを指定します。ほとんどのフラグは正負両方の形式を持ち、-ffoo の負形式は -fno-foo です。以下の表には、形式のうち一つ（デフォルトでない方）しかリストアップされていません。

表 3-15: コード生成変換用オプション

オプション	定義
-fargument-alias -fargument-noalias -fargument-noalias-global	パラメータ間およびパラメータとグローバルデータ間の可能性のある関係を指定します。 -fargument-alias は引数（パラメータ）が互いに別名であり、グローバルストレージとも別名であることを指定します。 -fargument-noalias は引数が互いに別名を持つわけではないが、グローバルストレージとは別名であることを指定します。 -fargument-noalias-globalは非引数が互いに別名を持つわけでもなく、グローバルストレージとも別名を持つわけではないことを指定します。それぞれの言語は、言語標準により要求されるオプションはすべて自動的に使用します。これらのオプションを自分で使用する必要はありません。
-fcall-saved-*reg*	*reg* と名づけられたレジスタを関数で保存できる割り当て可能なレジスタとして扱います。コール全般にわたって有効な一時的なものもしくは変数として割り当てられます。このようにコンパイルされた関数は、レジスタを使用している場合、レジスタ *reg* を保存・復帰します。 このフラグをフレームポインタやスタックポインタと一緒に使用するのはエラーになります。マシンの実行モデルに関わる固定の役割をもっている他のレジスタ用としてこのフラグを使用すると重大な障害を引き起こします。また、このフラグを、関数値が戻ってくるレジスタ用として使用しても重大な障害が起こります。このフラグはすべてのモジュールで首尾一貫していなければなりません。
-fcall-used-*reg*	*reg* と名づけられたレジスタを関数コールで上書きされる割り当て可能なレジスタとして扱います。コール全般にわたっては有効ではない一時的なものもしくは変数として割り当てられます。このようにコンパイルされた関数は、レジスタ *reg* の保存・復帰はしません。 マシンの実行モデルに関わる固定の役割を持っている他のレジスタのためにこのフラグを使用すると重大な障害を引き起こします。このフラグはすべてのモジュールで首尾一貫していなければなりません。
-ffixed-*reg*	*reg* と名づけられたレジスタを固定されたレジスタとして扱い、生成されたコードは（スタックポインタ、フレームポインタ、もしくはその他の固定された役割以外）参照しません。 *reg* はレジスタの名前（例えば、-ffixed-w3）でなければなりません。

-finstrument-functions	関数の出入り口に対するお決まりのコールを生成します。関数の入り口直後と出口直前で以下のプロファイリング関数が現在の関数とそのコールサイトのアドレスとともにコールされます。
	`void __cyg_profile_func_enter (void *this_fn, void *call_site);`
	`void __cyg_profile_func_exit (void *this_fn, void *call_site);`
	最初の引数は現関数のスタートのアドレスで、シンボル表の中で見つけることができます。プロファイリング関数はユーザで用意しなければなりません。
	関数実装はフレームポインタの使用を要求します。ある最適化レベルによりフレームポインタの使用を不可にします。
	-fno-omit-frame-pointer を使用すれば回避できます。

表 3-15: コード生成変換用オプション （続き）

オプション	定義
	この実装は別の関数内で、関数の拡張されたインラインのために実行されます。プロファイリングコールは、概念的には、インライン関数が入出りする場所を指示します。これはそのような関数のアドレス可能なバージョンが利用できなければならないことを意味しています。使用する関数すべてが拡張されたインラインである場合、これはコードサイズの追加拡張を意味します。C コードの中で extern inline が使用されていると、そのような関数のアドレス可能なバージョンが無ければなりません。 関数は、属性により no_instrument_function を与えられますが、その場合、この実装は実施されません。
-fno-ident	#ident ディレクティブを無視します。
-fpack-struct	抜け無しに構造体メンバすべてをまとめます。通常、このオプションは使いません。コードを一部最適化し、構成メンバのオフセットがシステムライブラリとは一致しなくなってしまうためです。 dsPIC デバイスは偶数バイト境界上にこれら語句の配列を要求するため、packed 属性を使用する際は実行時にアドレスエラーが起こらないよう注意する必要があります。
-fpcc-struct-return	レジスタと同じように、長い幅のメモリ内に短い struct や union を返します。この規則はあまり効率がよくありませんが、MPLAB C30 でコンパイルされたファイルとその他のコンパイラーでコンパイルされたファイル間での許容される利点があります。 短い構造体やユニオンはそのサイズと配列が int 型のサイズと配列に適合します。
-fno-short-double	デフォルトでは、コンパイラは float 型に double 型を使用します。このオプションは double 型を long double 型相当にします。このオプションをモジュール間で混用すると、モジュールが、引数パッセージを通して直接か、もしくは共有されている領域を通して間接的に倍精度データを共有している場合には、予期せぬ結果をもたらすことがあります。ライブラリは、どちらかのスイッチ設定をもったプロダクト関数とともに与えられます。
-fshort-enums	可能性のある値の宣言された範囲に必要とされるだけのバイト数を、enum タイプのみに割り当てます。特に、その enum タイプは、十分な余裕を持つ最小の整数型に相当します。
-fverbose-asm -fno-verbose-asm	より読みやすくするために、生成されたアセンブリコード内に追加のコメント情報を入れます。デフォルトは -fno-verbose-asm であり、追加情報は省略され、二つのアセンブラファイルを比較する場合には役に立ちます。
-fvolatile	ポインタ経由のすべてのメモリ参照を volatile と見なします。
-fvolatile-global	外部のグローバルなデータアイテムに対するすべてのメモリ参照を volatile とみなします。このスイッチの使用は静的データに影響しません。
-fvolatile-static	静的データに対するすべてのメモリ参照を volatile と見なします。

3.6 環境変数

この章での変数は、オプションではあるが、もし定義されるとコンパイラで使用されます。コンパイラドライバ、もしくは他のサブプログラムは、もし環境変数の値が設定されていなければ、以下の環境変数のいくつかに対する適切な値を決定することを選択します。ドライバもしくは他のサブプログラムは、MPLAB C30 の組込みに関しての内部知識を利用します。組込み構造が完全であり、すべてのサブディレクトリと実行が同じ相対位置にとどまる限り、ドライバもしくはサブプログラムは値を決定できます。

表 3-16: コンパイラに関係する環境変数

オプション	定義
PIC30_C_INCLUDE_PATH	この変数の値は PATH のように、ディレクトリ内のセミコロンで区切られたリストです。MPLAB C30 がヘッダファイルを検索する際に、-I で指定されるディレクトリの後で標準ヘッダファイルディレクトリの前に、変数内のディレクトリをリーチします。もし環境変数が定義されないと、プリプロセッサは、標準インストールに基づいて適切な値を選択します。デフォルトでは、インクルードファイル用として、以下のディレクトリが検索されます。 <install-path>\include と <install-path>\support\h
PIC30_COMPILER_PATH	PIC30_COMPILER_PATH の値は、PATH と同様に、ディレクトリのセミコロンで区切られたリストです。もし PIC30_EXEC_PREFIX を使用したサブプログラムを見つけることができなかたら、MPLAB C30 は、指定されたディレクトリをサーチします。
PIC30_EXEC_PREFIX	もし PIC30_EXEC_PREFIX が設定されると、それは、接頭辞を指定し、コンパイラで実行されるサブプログラムの名前の中で使用します。この接頭辞がサブプログラムの名前と接合されると、ディレクトリ delimiter は不可されませんが、もし望めば、スラッシュで終わる接頭辞を指定できます。もし MPLAB C30 が、指定された接頭辞を使用しているサブプログラムを見つけられなければ、PATH の中で環境変数を探そうとします。 もし PIC30_EXEC_PREFIX 環境変数が設定されないか、もしくは空の値を設定したら、コンパイラドライバは標準インストールに基づいて適切な値を選択します。もしそのインストールが修正されていなければ、ドライバが、必要とされるサブプログラムを割り当てることができる結果となります。 -B コマンドラインオプションで指定されるその他の接頭辞は、ユーザもしくはドライバで定義される値 PIC30_EXEC_PREFIX より先行します。 通常の環境では、この値は定義せずにおいて、ドライバにサブプログラム自身を位置付けるようにさせるのが一番良いです。
PIC30_LIBRARY_PATH	この変数の値は、PATH と同様に、ディレクトリのセミコロンで区切られたリストです。この変数はディレクトリのリストを指定し、リンカに渡します。この変数の、ドライバのデフォルト評価は以下の通りです。 <install-path>\lib; <install-path>\support\gld.
PIC30_OMF	OMF（オブジェクトモジュールフォーマット）を指定し MPLAB 30 で使用できるようにします。デフォルトで、ツールは COFF オブジェクトファイルを作成します。環境変数 PIC30_OMF が値 elf を持っていれば、ツールは ELF オブジェクトファイル を作成します。
TMPDIR	もし TMPDIR が設定されると一時ファイル生成用として使用するディレクトリを指定します。MPLAB C30 は次のステージの入力として使われるべき、コンパイルの一つのステージの出力を保持するためにテンポラリファイルを使用します。例えば、プリプロセッサの出力が相当し、コンパイラ本体の入力になります。

3.7 事前定義制約

以下のプリプロセッシング記号は MPLAB C30 コンパイラーで定義済みです。

記号	-ansi コマンドラインオプションで定義されているか？
dsPIC30	いいえ
__dsPIC30	はい
__dsPIC30__	はい

ELF 特有のコンパイラーバージョンは以下のプリプロセッシング記号を定義します。

記号	-ansi コマンドラインオプションで定義されているか？
dsPIC30ELF	いいえ
__dsPIC30ELF	はい
__dsPIC30ELF__	はい

COFF 特有のコンパイラーバージョンは以下のプリプロセッシング記号を定義します。

記号	-ansi コマンドラインオプションで定義されているか？
dsPIC30COFF	いいえ
__dsPIC30COFF	はい
__dsPIC30COFF__	はい

3.8 コマンドライン上の一つのファイルをコンパイルする

この章では、一つのファイルをコンパイルして、リンクするやり方を示します。この議論のために、コンパイラは c: ドライブ上の、pic30-tools と呼ばれるディレクトリ内にインストールされているものとします。従って、以下のものが適用されます。

表 3-17: コンパイラに関連したディレクトリ

ディレクトリ	定義
c:\pic30_tools\include	ANSI C ヘッダファイル用のディレクトリをインクルードします。このディレクトリは、標準 C ライブラリシステムヘッダファイルをコンパイラがストアする場所です。PIC30_C_INCLUDE_PATH 環境変数はそのディレクトリを指します。(DOS コマンドプロンプトから set をタイプすればこれをチェックします。)
c:\pic30_tools\support\h	dsPIC デバイス特定のヘッダファイル用のディレクトリをインクルードします。このディレクトリはコンパイラが dsPIC デバイス特定ヘッダファイルを、コンパイラがストアする場所です。PIC30_C_INCLUDE_PATH 環境変数はそのディレクトリを指します。(DOS コマンドプランプトから set をタイプすればこれをチェックします。)
c:\pic30_tools\lib	ライブラリディレクトリです。このディレクトリは、ライブラリと事前コンパイルされたオブジェクトファイルが置かれるところです。
c:\pic30_tools\support\gld	リンカスクリプトディレクトリです。このディレクトリはデバイス特有のリンカスクリプトが見つかる場所です。
c:\pic30_tools\bin	実行可能なディレクトリです。このディレクトリは、コンパイラプログラムが置かれる場所です。PATH 環境変数にはこのディレクトリを含まねばなりません。

以下は、二つの数を足し算する簡単な C プログラムです。

以下のプログラムをテキストエディタで作成し、ex1.c として保存します。

```
#include <p30f2010.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int
main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

プログラムの第一行目は、ヘッダファイル p30f2010.h を含み、ヘッダファイルはすべての特殊関数レジスタ用の定義を与えます。ヘッダファイルの詳細については、**第 6 章「デバイスサポートファイル」**を参照ください。

DOS プロンプトで以下をタイプし、プログラムをコンパイルします。

```
C:\> pic30-gcc -o ex1.o ex1.c
```

コマンドラインオプション -o ex1.o は、出力 COFF 実行可能なファイルの名前を付けます（もし -o オプションが指定されていないと、出力ファイル名は a.exe と名づけられます）。COFF 実行可能なファイルは、MPLAB IDE にロードされます。

もし hex ファイルが必要なら、例えばデバイスプログラマにロードするには、以下のコマンドを使用します。

```
C:\> pic30-bin2hex ex1.o
```

これにより、ex1.hex と名づけられたインテル hex ファイルが生成されます。

3.9 コマンドライン上の複数のファイルをコンパイルする

アプリケーション内の複数ファイルの使い方をデモするために、Add() 関数を add.c と呼ばれるファイルへ移動します。下記のようにします。

ファイル 1

```
/* ex1.c */
#include <p30f2010.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
```

ファイル 2

```
/* add.c */
#include <p30f2010.h>
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

DOS プロンプトで以下をタイプし、両方のファイルをコンパイルします。

```
C:\> pic30-gcc -o ex1.o ex1.c add.c
```

このコマンドは、モジュール ex1.c と add.c をコンパイルします。コンパイルされたモジュールはコンパイラライブラリと一緒にリンクされ、実行可能ファイル ex1.o が生成されます。

第 4 章 MPLAB C30 C コンパイラ実行時環境

4.1 序章

本章では MPLAB C30 C コンパイラ実行時環境について説明します。

4.2 ハイライト

本章で説明する項目は以下の通りです。

- アドレス空間
- コードとデータセクション
- スタートアップと初期化
- メモリ空間
- メモリーモデル
- X と Y のデータ空間
- コードとデータの配置
- ソフトウエアスタック
- C スタックの使い方
- C ヒープの使い方
- 関数コール集合
- レジスタ集合
- ビットリバースとモジュロアドレッシング
- PSV の使い方

4.3 アドレス空間

dsPIC® マイクロコントローラ (MCU) デバイスは、伝統的な PICmicro® MCU の特徴（周辺、ハーバードアーキテクチャ、RISC）と新しい DSP の機能を一緒にしたものです。dsPIC デバイスは、二つの際立ったメモリ領域を持っています。

- プログラムメモリ（図 4-1）は実行可能コードとオプションの定数データを含みます。
- データメモリ（図 4-2）は、外部変数、静的変数、システムスタック、およびファイルレジスタを含みます。データメモリは、二アデータ、そのデータは、データメモリ空間の最初の 8 KB 内のデータであるものと、ファーデータ、そのデータはデータ、メモリ空間の上位 56 KB 内にあるものから構成されます。

プログラムとデータメモリ領域は明確に区別されますが、コンパイラーは Program Space Visibility (PSV) ウィンドウを介してプログラムメモリ内の定数データにアクセスできます。

図 4-1: プログラム空間メモリマップ

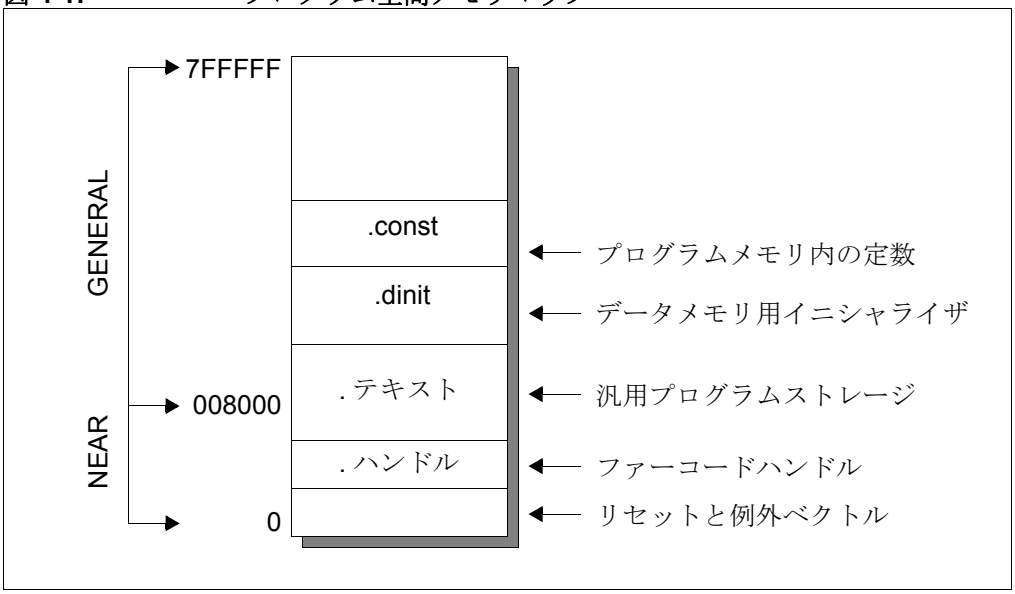
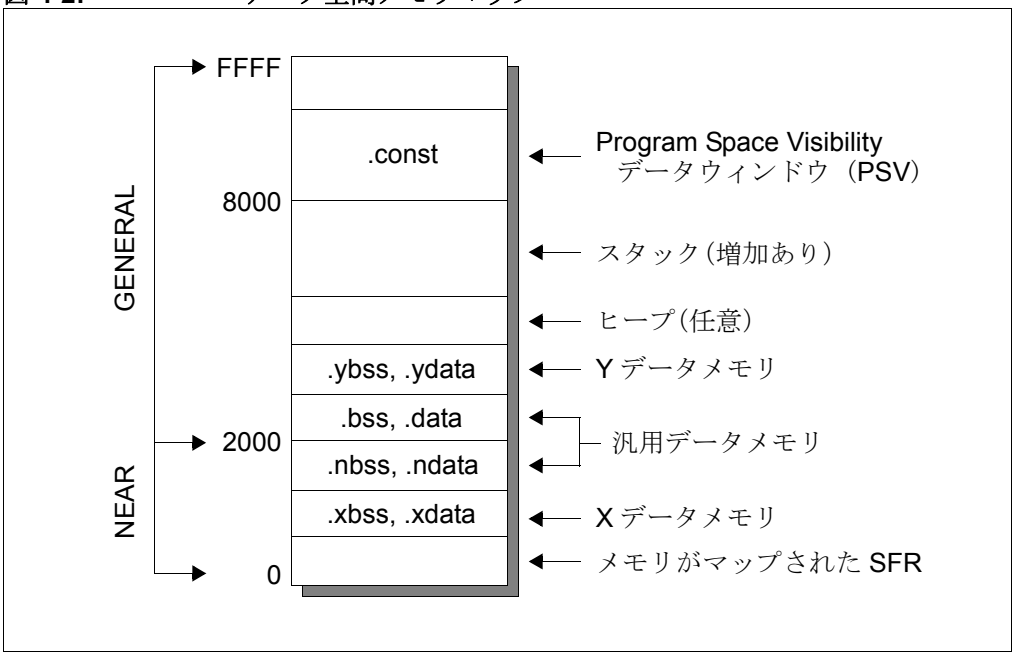


図 4-2: データ空間メモリマップ



4.4 コードとデータセクション

セクションとは、コードもしくはデータの、移動可能なブロックで、dsPIC デバイスメモリの中で、連続した位置を占めます。どんなオブジェクトファイルにも、普通はいくつかのセクションがあります。例えば、ファイルには、プログラムコード用セクションと、初期化されていないデータ用のセクション等々があります。

MPLAB C30 コンパイラは、セクション属性を用いて他で指示されない限り、コードとデータをデフォルトのセクションに置きます（セクション属性の情報については、**セクション 2.3「キーワードの違い」**をご参照ください。）。コンパイラが生成する、すべての実行可能コードは、 `.text` という名前の付けられたセクションに割り当てられ、データは、表 4-1 で示されるように、データのタイプに基づき、異なるセクションに割り当てられます。

表 4-1:　　コンパイラが生成するデータセクション

	初期化されるもの		初期化されないもの	
	変数	ROM 内の定数	RAM 内の定数	変数
ニアの	.ndata	.const	.ndconst	.nbss
ファーの	.data	.const	.dconst	.bss

それぞれのデフォルトセクションと、そのセクションにストアされた情報のタイプを記述したものは、以下にリストされます。

.text

実行可能なコードは、`.text` セクションに割り当てられます。

.data

`far` 属性を持つ初期化された変数は、`.data` セクションに割り当てられます。ラージデータのメモリモデルが選択された時（すなわち、`-mlarge-data` コマンドラインオプションを用いる時）は、これが初期化された変数用のデフォルト位置になります。

.ndata

`near` 属性を持つ初期化された変数は、`.ndata` セクションに割り当てられます。スモールデータのメモリモデルが選択された時（すなわち、`-msmall-data` コマンドラインオプションを用いる時）は、これが初期化された変数用のデフォルト位置になります。

.const

ストリング定数や固定化された変数のような、定数値は、デフォルトの `-mconst-in-code` コマンドライン オプションを用いた時には、`.const`セクションに割り当てられます。このセクションはプログラムメモリに位置するように割り当てられ、PSV ウインドウを使用してアクセスされます。

変数は、`-mconst-in-code` オプションがコマンドライン上に存在するかどうかに関わらず、以下の `section` 属性を使用して `.const` セクションに配置されることがあります。

```
int i __attribute__((space(auto_psv)));
```

.dconst

ストリング定数や固定化された変数のような定数値は、-mconst-in-code コマンドライン オプションを使用せず、デフォルトの -mlarge-data コマンドライン オプションを用いた時には、.dconst セクションに割り当てられます。リンカオプション --no-data-init が指定されていないと、MPLAB C30 のスタートアップコードは、.dinit セクションからのデータをコピーすることでこのセクションを初期化します。.dinit セクションはリンカで生成されプログラムメモリに位置しています。

.ndconst

ストリング定数や定数修飾された変数のような定数値は、-mconst-in-code コマンドラインオプションを使用せず、デフォルトの -msmall-data コマンドラインオプションを用いた時には、.ndconst セクションに割り当てられます。リンカーオプション --no-data-init が指定されていないと、MPLAB C30 のスタートアップコードは、.dinit セクションからのデータをコピーすることでこのセクションを初期化します。.dinit セクションはリンカで生成されプログラムメモリに位置しています。

.bss

far 属性を持った、初期化されない変数は .bss セクションに割り当てられます。ラージデータメモリモデルが選択された時（すなわち、-mlarge-data コマンドラインオプションを用いる時）は、これは初期化されない変数用のデフォルト位置になります。

.nbss

near 属性を持った、初期化されない変数は .nbss セクションに割り当てられます。スモールデータメモリモデルが選択された時（すなわち、-msmall-data コマンドラインオプションを用いる時）は、これは初期化されない変数用のデフォルト位置になります。

.pbss - Persistent Data

デバイスリセットに影響されない RAM 内のデータストレージを要求するアプリケーションは、この目的のために、セクション .pbss を使用します。セクション .pbss は二アデータメモリ内に割り当てられ、libpic30.a 内のデフォルトスタートアップモジュールによっては変更されません。

初期化されない変数は、以下のセクション属性を用いて .pbss セクションに置かれます。

```
int i __attribute__((persistent));
```

永続するデータストレージを利用するために、main() 関数はどのタイプのリセットが発生したかを決定するテストから始まります。RCON リセットコントローラレジスタ内の種々のビットがリセットソースを決定するためにテストされます。詳細は８章の *dsPIC30F Family Reference Manual* (DS70046) をご参照ください。

4.5 スタートアップと初期化

2 つの C ルーチンスタートアップモジュールが、libpic30.a アーカイブ / ライブラリに含まれます。両方のエントリポイントは __reset です。リンカスクリプトは、プログラムメモリ内の位置 0 で GOTO __reset 命令を構成し、その命令はデバイスリセットにより制御が転送されます。

初期のスタートアップモジュール (crt0.o) はデフォルトでリンクされ、以下のような動きをします。

1. スタックポインタ (W15) とスタックポインタリミットレジスタ (SPLIM) は、リンカもしくはカスタムリンカスクリプトで与えられる値を用いて、初期化されます。詳細については、**セクション 4.9「ソフトウエアスタック」**をご参照ください。
2. もし、.const セクションが定義されると、PSVPAG と CORCON レジスタの初期化によって、Program Space Visibility(PSV) ウインドウにマッピングされます。.const セクションは MPLAB IDE 内で“Constants in code space”オプションが選択された時か、もしくは、デフォルトの -mconst-in-code オプションが、MPLAB C30 コマンドラインで指定されたときに定義されることに注意して下さい。
3. セクション .dinit 内のデータ初期化テンプレートが読み出され、初期化されないセクションすべてがクリアされ、プログラムメモリから読み出される値で、すべての初期化されるセクションが初期化されます。データ初期化テンプレートはリンカで生成され、ユーザが定義するセクションと、**セクション 4.4「コードとデータセクション」**でリストアップされる標準セクションをサポートします。

注： 永続するデータセクション .pbss はクリアも初期化もされません。

4. 関数 main はパラメータ無しでコールされます。
5. main から戻ったら、プロセッサはリセットされます。

代替のスタートアップモジュール (crt1.o) は、the -Wl, --no-data-init option が指定された時にリンクされます。ステップ (3)（これは省略されますが）を除いて、同じオペレーション動作をします。代替のスタートアップモジュールは初期モジュールよりもずいぶん小さく、もし初期化が必要ない時は、プログラムメモリを節約するために選択されます。

両方のモジュールのソースコード（dsPIC アセンブリ言語）は、c:\pic30_tools\src ディレクトリ内に与えられます。スタートアップモジュールは、必要であれば、修正できます。例えば、アプリケーションがパラメータと一緒にコールされる main を要求すると、条件付きアセンブリにより指向性は切替えられ、このサポートを行います。

4.6 メモリ空間

静的な外部変数は通常、汎用データメモリ空間に割り当てられます。定数修飾された変数は、constants-in-data メモリモデルが選択されていると汎用データメモリに割り当てられ、constants-in-code メモリモデルが選択されているとプログラムメモリに割り当てられます。

MPLAB C30 は複数の特殊用途メモリ空間を定義して dsPIC DSC のアーキテクチャ上の機能に一致させます。静的な外部変数は、**セクション 2.3.1「変数属性の指定」**で述べられている space 属性を使用して特殊用途メモリ空間に割り当てられることがあります。

data

汎用データ空間。汎用データ空間内の変数には通常の C ステートメントを使用してアクセスできます。これはデフォルトの割り当てです。

xmemory

X データアドレス空間。X データ空間内の変数には通常の C ステートメントを使用してアクセスできます。X データアドレス空間は DSP 指向ライブラリおよび / またはアセンブリ言語命令に対し特に適合性を持っています。

ymemory

Y データメモリ空間。Y データ空間内の変数には通常の C ステートメントを使用してアクセスできます。Y データアドレス空間は DSP 指向ライブラリおよび / またはアセンブリ言語命令に対し特に適合性を持っています。

prog

汎用プログラム空間。通常は実行コードのために確保されています。プログラム空間内の変数には通常の C ステートメントを使用してもアクセスできません。これらの変数には通常、テーブルアクセスインランアセンブリ命令または PSV ウィンドウを使用してプログラマが明示的にアクセスする必要があります。

const

コンパイラーにより管理されたプログラム空間内の領域で、PSV ウィンドウからのアクセスのためにデザインされています。const 空間内の変数は通常の C ステートメントを使用して読み取り可能で（書き込みは不可）、割り当て可能な空間は最大で合計 32K です。

psv

PSV ウィンドウからのアクセスのためにデザインされたプログラム空間です。psv 空間内の変数はコンパイラーでは管理されず、通常の C ステートメントではアクセスできません。これらの変数には通常、テーブルアクセスインランアセンブリ命令または PSV ウィンドウを使用してプログラマが明示的にアクセスする必要があります。PSVPAG レジスタの設定をするだけで PSV エリアの変数にアクセスすることができるようになります。

eedata

EEPROM 空間は 16 ビット幅不揮発性メモリ領域で、プログラムメモリ内の上位アドレスに配置されます。eedata 空間内の変数には通常の C ステートメントではアクセスできませのん。これらの変数には通常、テーブルアクセスインランアセンブリ命令または PSV ウィンドウを使用してプログラマが明示的にアクセスする必要があります。

4.7 メモリモデル

コンパイラはいくつかのメモリモデルをサポートします。アプリケーションに最適なメモリモデルを選択するために、現在使用中の特定 dsPIC デバイスと使用メモリのタイプに基き、コマンドラインオプションが利用できます。

表 4-2: メモリモデルコマンドラインオプション

オプション	メモリー定義	定義
-msmall-data	8KB までのデータメモリ **これはデフォルトです。**	データメモリをアクセスするための PIC18 のような命令の使用を許可します。
-msmall-scalar	8KB までのデータメモリ **これはデフォルトです。**	データメモリ内のスカラーをアクセスするための PIC18 のような命令の使用を許可します。
-mlarge-data	8 KB 以上のデータメモリ	データリファレンス用の間接的アクセスを使用します。
-msmall-code	32K ワードまでのプログラムメモリ。**これはデフォルトです。**	関数ポインタはジャンプテーブルを経由しません。関数コールは RCALL 命令を使用します。
-mlarge-code	32K ワード以上のプログラムメモリ	関数ポインタはジャンプテーブルを経由します。関数コールは CALL 命令を使用します。
-mconst-in-data	データメモリ内に配置された定数	スタートアップコードで、プログラムメモリからコピーされた値
-mconst-in-code	プログラムメモリ内に配置された定数**これはデフォルトです。**	Program Space Visibility (PSV) データウインドウ経由でアクセスされる値

コマンドラインオプションはコンパイルされたモジュールにグローバルに適用されます。個別の変数と関数は、コード生成をよりよく制御するために、near もしくは far として、宣言されます。個別の変数もしくは関数属性の設定については、**セクション 2.3.1「変数属性の指定」**と**セクション 2.3.2「関数の属性を指定する」**をご参照ください。

4.7.1 二アデータとファーデータ

もし変数が二アデータセクションに割り当てられると、コンパイラは、もし変数が二アデータセクションに割り当てられないときよりも、より良い（よりコンパクトな）コードを生成します。もしアプリケーションすべての変数が、二アデータの 8 KB 以内に収まれば、一つ一つのモジュールをコンパイルする時に、デフォルトの -msmall-data コマンドラインオプションを用いて変数を二アデータに置くように、コンパイラに要求できます。もしスカラータイプ（配列でも構造でもない）により使用されるデータの総量が 8 KB 以下であれば、デフォルトの -msmall-scalar が使用できます。このことは、コンパイラに、アプリケーション用のスカラーを二アデータセクションに割り当てるように配慮することを要求します。

もしこれらのグローバルオプションのうちどちらにも適さない場合は、以下の代替が利用できます。

1. -mlarge-data もしくは -mlarge-scalar コマンドライン オプションを用いて、アプリケーションのいくつかをコンパイルできます。この場合、それらのモジュールで使用される変数のみが、ファーデータセクションに割り当てられます。もしこの代替が使用されると、外部定義された変数を用いる場合には注意が必要です。もし、これらのオプションの一つを用いてコンパイルされたモジュールにより使用される変数が、外部で定義されていると、それが定義されているモジュールもまた同じオプションを用いてコンパイルされねばなりません。もしくは、変数宣言と定義がファー属性を付加されていなければなりません。

2. もしコマンドラインオプション -mlarge-data もしくは -mlarge-scalar が使用されたとき、個々の変数は、二ア属性を付けることで far データ空間から除外されます。
3. モジュールスコープを持つコマンドラインオプションを用いる代わりに、個々の変数に、far 属性を付けることでファーデータセクションに置くことができます。

もし、アプリケーション用のすべての二ア変数が 8K の二アデータ空間に収まらない場合は、リンカはエラーメッセージを出力します。

4.7.2 二アコードとファーコード

（互いに半径 32 K ワード以内にある）二ア関数は、そうでない位置にある場合よりもより効率的に互いにコールできます。もしアプリケーション内のすべての関数が二アにあることがわかっていれば、個々のモジュールをコンパイルする時に、関数コールをより効率的な形式を使用するようにコンパイラに指示するために、デフォルトの -msmall-code コマンドライン オプションが使用できます。

このデフォルトオプションが適していない場合、以下の代替が利用できます。

1. -msmall-code コマンドライン オプションを用いてアプリケーションのいくつかのモジュールをコンパイルできます。この場合、それらのモジュール内の関数コールのみが、関数コールのより効率的な形式を使用します。
2. -msmall-code コマンドライン オプションが使用されたとき、far 属性を付けることで個々の関数用の関数コールにロング形式を使用するようにコンパイラに指示します。
3. モジュールスコープを持つコマンドライン オプションを用いる代わりに、関数の宣言や定義に near 属性を付けることにより、関数コールにより効率的な形式を使用して個々の関数をコールするように指示します。

-msmall-code コマンドラインオプションは、-msmall-data コマンドライン オプションとは異なり、前者の場合はコンパイラは、関数が互いに近い位置に割り当てられることを特に保証しませんが、後者の場合、コンパイラは特別のセクションに変数を割り当てます。

もしニアと宣言された関数が、関数コールのより効率的な形式を用いても、コーラにたどり着かない時は、リンカはエラーメッセージを出力します。

4.8 コードとデータの配置

セクション 4.4「コードとデータセクション」で述べられているように、コンパイラーは、.text セクションにコードを置くようにし、使用されているメモリモデルやデータが初期化されているかどうかによって、データはいくつかの名前を付けられたセクションのうちの 1 つに置かれます。モジュールがリンク時に結合する際、リンカーはその属性に基づき様々なセクションの開始アドレスを決定します。

特定の関数もしくは変数が特定のアドレス、もしくはあるアドレス範囲内に置かれなければならないという場合に問題が起こります。この問題は、**セクション 2.3「キーワードの違い」**で述べられている通り、address 属性を使用することで解決できます。例えば、関数 PrintString をプログラムメモリのアドレス 0x8000 に配置するには以下のように宣言します。

```
int __attribute__ ((address(0x8000))) PrintString (const char *s);
```

同様に、変数 Mabonga をデータメモリ内のアドレス 0x1000 に配置するには、以下のようにします。

```
int __attribute__ ((address(0x1000))) Mabonga = 1;
```

コードまたはデータを配置するもう 1 つの方法として、ユーザー定義のセクションに関数または変数を置き、カスタムリンカースクリプト内のセクションの開始アドレスを指定するというやり方があります。具体的には、以下を実行します。

1. C ソース内のコードもしくはデータの宣言を修正し、ユーザー定義のセクションを指定します。
2. ユーザー定義のセクションをリンカスクリプトファイルに追加し、セクションの開始アドレスを指定します。

例えば、関数 PrintString をプログラムメモリ上のアドレス 0x8000 に置くためには、まず、C ソースの中で以下のように関数を宣言します。

```
int __attribute__((__section__(".myTextSection")))
PrintString(const char *s);
```

セクション属性は、デフォルトの .text セクションではなく .myTextSection と名づけられたセクションに関数を置くように指定します。ユーザ定義のセクションが配置される場所は指定しません。この指定は以下のようにカスタムリンカスクリプトで行わなければなりません。デバイス特有のリンカスクリプトをベースに使用する場合は、以下のセクションの定義を追加します。

```
.myTextSection 0x8000 :
  {
        *(.myTextSection);
  } >program
```

これは、出力ファイルが、.myTextSection と名づけられたセクションで、配置 0x8000 から開始するセクションを含み、.myTextSection と名づけられたすべての入力セクションを含むことを指定します。この例では、そのセクション内には一つの関数 PrintString があるので、関数はプログラムメモリの 0x8000 アドレスに配置されます。

同様に、データメモリ内のアドレス 0x1000 に変数 Mabonga を配置するには、最初に、C ソースの中で以下のように変数を宣言します。

```
int __attribute__((__section__(".myDataSection"))) Mabonga = 1;
```

セクション属性は、デフォルトの .data セクションではなく .myDataSection と名づけられたセクションに関数が置くよう指定します。ユーザ定義のセクションが配置される場所は指定しません。その指定は、以下のようにカスタムリンカスクリプトで行わなければなりません。デバイス特有のリンカスクリプトをベースに使用する場合は、以下のセクションの定義を追加します。

```
.myDataSection 0x1000 :
  {
        *(.myDataSection);
  } >data
```

これは、出力ファイルが、.myDataSection と名づけられたセクションで、配置 0x1000 から開始するセクションを含み、.myDataSection と名づけられたすべての入力セクションを含むことを指定します。この例では、そのセクション内には一つの変数 Mabonga があるので、変数はデータメモリの 0x1000 アドレスに配置されます。

4.9 ソフトウエアスタック

dsPIC デバイスは、レジスタ W15 をソフトウエアスタックポインタとして使うように専任化させます。関数コール、割り込み、例外処理を含むすべてのプロセッサのスタック動作は、ソフトウエアスタックを使用します。スタックは、より高位のメモリアドレスへと上に向かって成長します。

dsPIC デバイスはまた、スタックオーバーフロー検出もサポートします。もしスタックポインタリミットレジスタ SPLIM が初期化されると、デバイスはすべてのスタック使用時についてオーバーフローのテストをします。もしオーバーフローが発生したら、プロセッサはスタックエラー例外を発生します。デフォルトでは、これによりプロセッサはリセットされます。_StackError と名づけられた割り込み関数を定義することで、アプリケーションでもスタックエラー例外ハンドラをインストールできます。詳細については**第 7 章「割り込み」**をご覧ください。

C runtime スタートアップモジュールは、スタートアップや初期化シーケンス内でスタックポインタ (W15) とスタックポインタ制限レジスタ (SPLIM) を初期化します。初期値は通常リンカによって与えられ、未使用のデータメモリから得られる最大のスタックを割り当てます。スタックの位置は、リンクマップ出力ファイルにレポートされます。アプリケーションは、少なくとも最小限のサイズのスタックが、--stack リンカーコマンドラインオプションによって保証されます。詳細については、*MPLAB ASM30, MPLAB LINK30 と Utilities User's Guide* (DS51317) をご参照ください。

他の方法として、特定のサイズのスタックを、カスタムリンカスクリプトのユーザ定義セクションで割り当てることができます。以下の例では、データメモリの 0x100 バイトがスタック用に確保されます。C runtime スタートアップモジュールにより使用されるために、__SP_init と __SPLIM_init の 2 つのシンボルが宣言されていることに注意してください。

```
.stack :
  {
      __SP_init = .;
      . += 0x100
      __SPLIM_init = .;
    . += 8
  } >data
```

__SP_init はスタックポインタ (W15) の初期値を定義し、__SPLIM_init はスタックポインタ リミットレジスター (SPLIM) の初期値を定義します。__SPLIM_init は、物理的スタックのリミットより少なくとも 8 バイト下側であり、スタックエラー例外処理用に残しておきます。この値は、もしスタックエラー割り込みハンドラがインストールされていれば、割り込みハンドラ自体が使用するスタック用にまで低減できます。デフォルトの割り込みハンドラは、追加のスタックを使いません。

4.10 C スタック使用方法

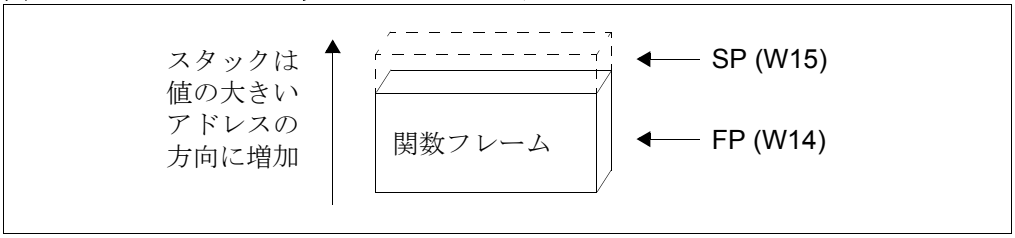
C コンパイラはソフトウエアスタックを使用して、以下の動作を行います。

- 自動変数の割り当て
- 引数の関数への引渡し
- 割り込み関数内でプロセッサ状態を保存
- 関数の戻りアドレスの保存
- 途中の一時結果のストア
- 関数コール間のレジスタの保存

実行時のスタックは低いアドレスから高位のアドレスへと上へ成長していきます。コンパイラは二つのワーキングレジスタを用い、スタックを管理します。

- W15 - これはスタックポインタ (SP) です。スタック上の最初の未使用位置と定義されるスタックのトップを示します。
- W14 - これはフレームポインタ (FP) です。これは現在の関数のフレームを指します。おのおのの関数は、もし要求されれば、スタックの現在のトップから新しいフレームを作成し、自動変数や一時変数をそこに配置します。コンパイラオプション `-fomit-frame-pointer` は FP の使用を制限するために使用されます。

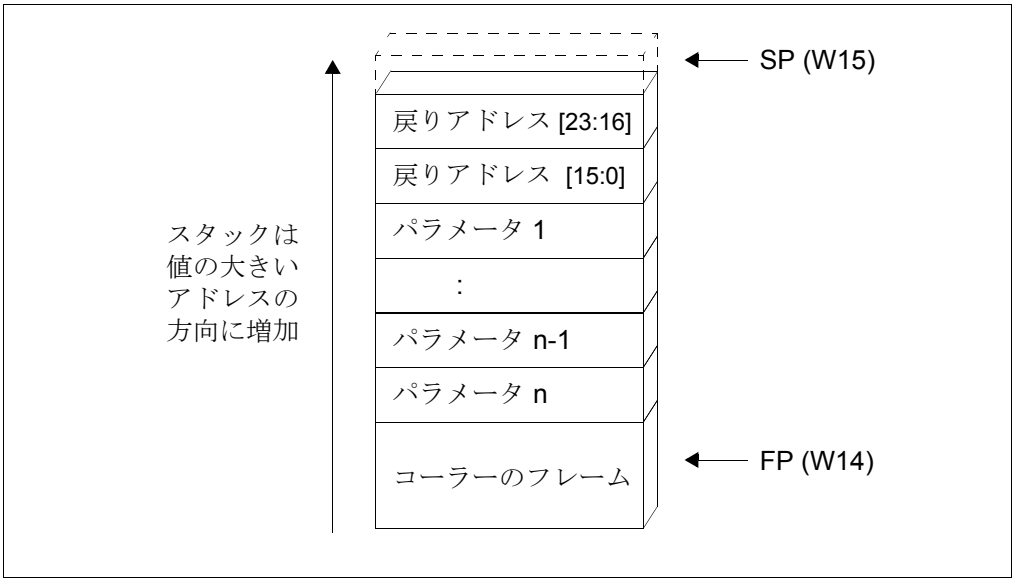
図 4-3:　　スタックとフレームポインタ



C 実行時スタートアップモジュール (`crt0.o` and `crt1.o` in `libpic30.a`) はスタックポインタ W15 を初期化し、スタックの底を示します。また、スタックポインタ制限レジスタを初期化してスタックのトップを示します。スタックは上に向けて成長し、スタックポインタ制限レジスタの値を越えて成長すると、スタックエラートラップが発生します。ユーザはスタックの成長をさらに制限するために、スタックポインタ制限レジスタを初期化することができます。

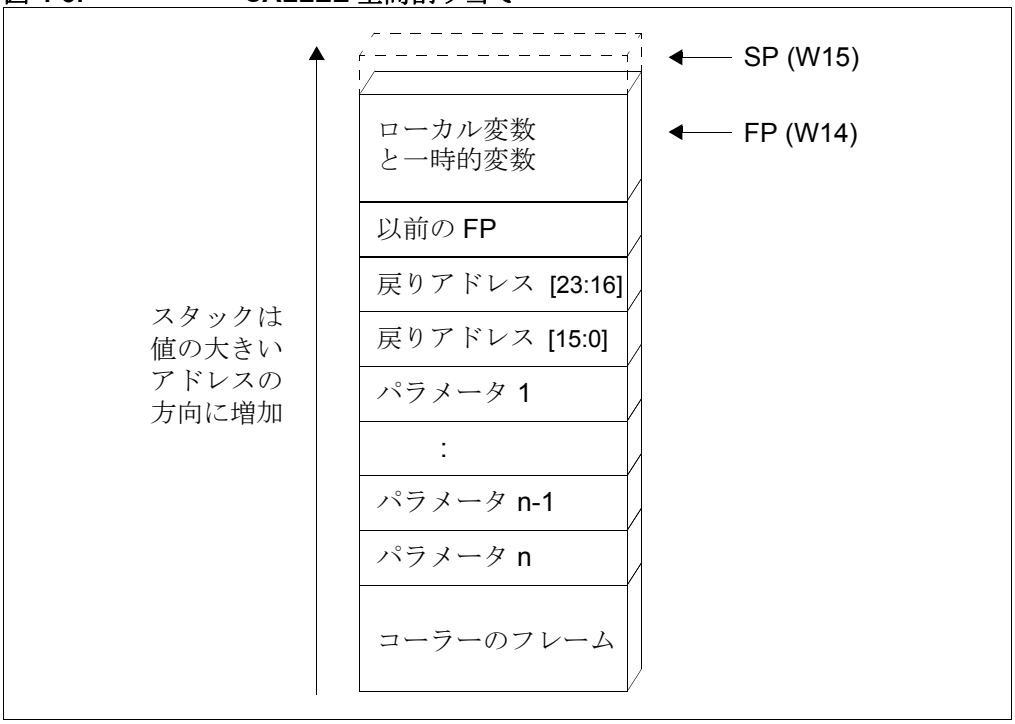
以下の図は関数コールに関する手順を図示しています。 `CALL` もしくは `RCALL` 命令を実行すると、ソフトウエア上の戻りアドレスを push します。図 4-4 をご覧ください。

図 4-4: CALL OR RCALL



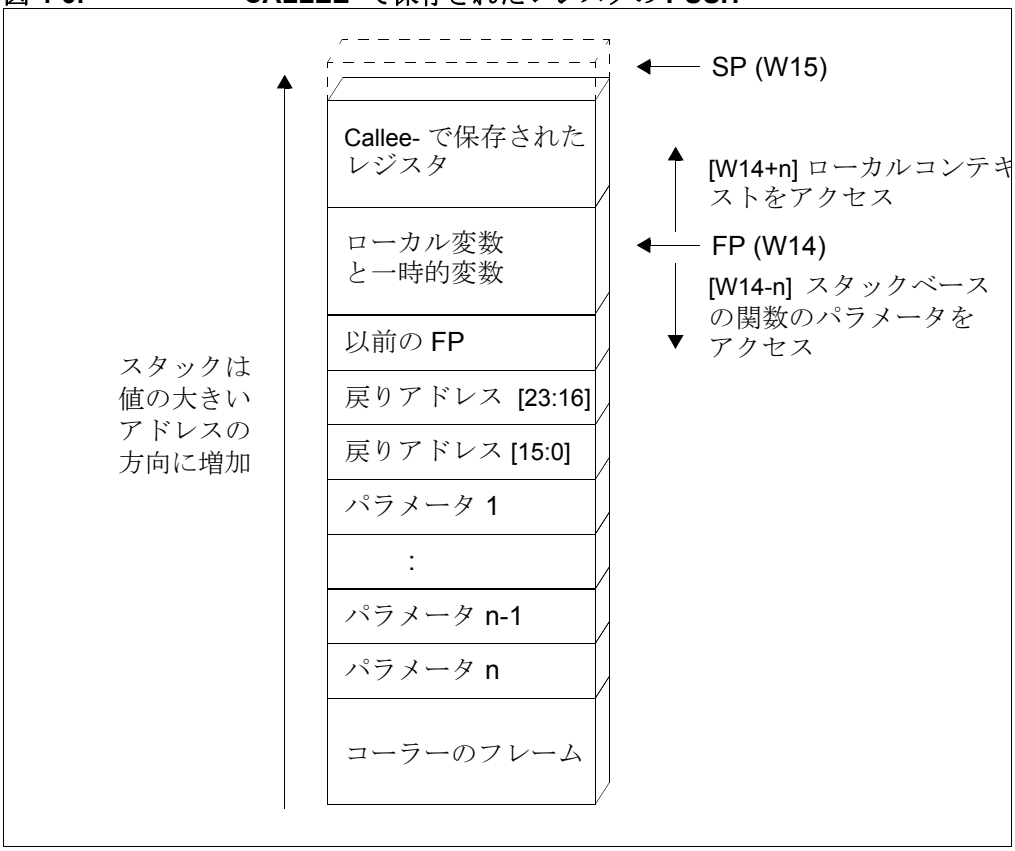
コールされた関数 (callee) はローカルコンテキスト用の空間を割り当てることができます。(図 4-5)。

図 4-5: CALLEE 空間割り当て



最後に、関数で用いられる callee で保存されたレジスタが push されます。（図 4-6）

図 4-6: CALLEE- で保存されたレジスタの PUSH



4.11 C ヒープの使用方法

C 実行時のヒープはデータメモリの初期化されない領域で、標準 C ライブラリのダイナミックメモリ管理関数 calloc、malloc と realloc を用いてダイナミックメモリ割り当てを行う際に使用されます。もし、これらの関数を用いなければ、ヒープを割り当てる必要はありません。デフォルトでは、ヒープは生成されません。

もしダイナミックメモリ割り当てをしたいのなら、直接的にメモリ割り当て関数の一つをコールするか、間接的に標準 C ライブラリの入出力関数を用いるかどちらかのやり方になりますが、ヒープを生成せねばなりません。ヒープは、--heap リンカー コマンドラインオプションを用いて、リンカコマンドライン上でサイズを指定することで生成されます。コマンドラインを用いて、512 バイトのヒープを割り当てる例は以下の通りです。

```
pic30-gcc foo.c -Wl,--heap=512
```

リンカはスタックのすぐ下にヒープを割り当てます。(図 4-2)

もし、標準 C ライブラリ入力・出力関数を使用するなら、ヒープを割り当てねばなりません。もし stdout が使用する唯一のファイルであるなら、ヒープサイズはゼロ、すなわち、以下のようにコマンドラインオプションを用います。

```
-Wl,--heap=0
```

ファイルを開くなら、ヒープサイズは、同時に開くファイルの 1 つのファイルにつき 40 バイトを含まねばなりません。もしヒープメモリが十分でなければ、オープン関数がエラー表示を返します。バッファモードでファイルを開くときには、ファイル 1 つについて、514 バイトのヒープ空間が必要です。もしバッファ用のヒープメモリが十分でなければ、ファイルは非バッファモードで開きます。

4.12 関数コール集合

関数をコールする時は、

- レジスタ WO - W7 はコールする側が保存します。コールする関数は、レジスタ値を保持するためにこれらの値をスタックの上にプッシュしなければなりません。
- レジスタ W8 - W14 はコールされる側が保存します。コールされる関数は、関数が使用するこれらのレジスタすべてを保存しなければなりません。
- レジスタ WO-W4 は関数戻り値に使用されます。

表 4-3: 必要なレジスタ

データタイプ	必要なレジスタ数
char	1
int	1
short	1
pointer	1
long	2 (連続して、偶数レジスタに揃えます)
float	2 (連続して、偶数レジスタに揃えます)
double*	2 (連続して、偶数レジスタに揃えます)
long double	4 (隣接 -4 倍数レジスタに揃えます)
structure	構造体内の 2 バイトあたり 1 レジスタ

* double は -fno-short-double が使用されている場合は long double と同等です。

パラメータは、利用できるレジスタのうち最初に揃えられる連続するレジスタ内に置かれます。コールする関数は、必要ならパラメータを保持しなければなりません。構造体は整列させるという制限がありません。構造体パラメータは、構造体全体を保持するのに十分なレジスタがあれば、レジスタを占有します。関数の結果は WO で始まる連続したレジスタ内に格納されます。

4.12.1 関数パラメータ

最初の 8 個のワーキングレジスタ (W0-W7) は関数パラメータ用に使用されます。パラメータは、レジスタ内で左から右へ順に割り当てられ、一つのパラメータは、適切に整列された最初に利用できるレジスタに割り当てられます。

以下の例では、すべてのパラメータはレジスタに引き渡されますが、宣言に現れる順番通りではありません。このフォーマットを用いると、MPLAB C30 コンパイラが、利用できるパラメータレジスタを最も効率的に使用できます。

例 4-1: 関数コールモデル

```
void
params0(short p0, long p1, int p2, char p3, float p4, void *p5)
{
        /*
        ** W0           p0
        ** W1           p2
        ** W3:W2        p1
        ** W4           p3
        ** W5           p5
        ** W7:W6        p4
        */
        ...
}
```

次の例は、構造 がどのように関数に引き渡されるかを示しています。もし完全な構造が、利用できるレジスタ内に適合していれば、構造はレジスタ経由で引き渡されます。そうでなければ、構造の引数はスタックの上に置かれます。

例 4-2:　　関数コールモデルと構造の引渡し

```
typedef struct bar {
  int i;
  double d;
} bar;

void
params1(int i, bar b) {
        /*
        ** W0          i
        ** W1          b.i
        ** W5:W2       b.d
        */

}
```

可変長引数リストの省略 (...) に対応するパラメータはレジスタには割り当てられません。レジスタに割り当てられないパラメータは、右から左の順にスタックの上にプッシュされます。

次の例では、構造パラメータが大き過ぎるので、レジスタに置くことができません。しかし、これは、次のパラメータがレジスタスポットを使用することを妨げません。

例 4-3:　　関数コールモデルとスタックをベースとした引数

```
typedef struct bar {
  double d,e;
} bar;

void
params2(int i, bar b, int j) {
        /*
        ** W0           i
        ** stack        b
        ** W1           j
        */
}
```

スタックの上に置かれる引数にアクセスすることは、フレームポインタが生成されるかどうかに依存します。一般的にコンパイラは（そのようにするなと言われない限り）フレームポインタを生成し、スタックを基にしたパラメータはフレームポインタ (W14) 経由でアクセスできます。上の例では、b は W14-22 からアクセスできます。フレームポインタのオフセット、－ 22 は、２バイトを前の FP 用、４バイトを戻りアドレス用、16 バイトを b 用に取ることで計算されます。(図 4-6 を参照ください)

フレームポインタを使用しない場合は、アセンブリプログラマは、エントリからプロシジャまでにいくつのスタック空間が使用されるかを知っていなければなりません。もしそれ以上のスペースが使用されなければ、計算は上記の例と同様で、b は W15-20 経由でアクセスでき、4 バイトが戻りアドレス用、16 バイトが b のスタートをアクセスするために用いられます。

4.12.2 戻り値

関数戻り値は、8 もしくは 16 ビットのスカラーは WO に、32 ビットのスカラーは W1 :W0 に、64 ビットのスカラーは W3:W2:W1 :W0 にもどります。集合は間接的に WO 経由で戻り、それは集合値のアドレスを含むように関数コーラーにより設定されます。

4.12.3 複数の関数コールにまたがるレジスタの保存

コンパイラは W8-W15 レジスタが、複数の通常の関数コールにまたがって保存されるようにアレンジします。レジスタ W0-W7 は、寄せ集めのレジスタとして利用できます。割り込み関数については、コンパイラが、すべての必要なレジスタ、すなわち、W0-W15 と RCOUNT が保存されるようにアレンジします。

4.13 レジスタの集合

特定のレジスタは C 実行時環境中で特定の役割を持ちます。レジスタ変数は、表 4-4 に示されるように、1 つもしくはそれ以上のワーキングレジスタを使用します。

表 4-4: レジスタの集合

変数	ワーキングレジスタ
`char, signed char, unsigned char`	フレームポインタとして使用されなければ W0-W13, and W14。
`short, signed short, unsigned short`	フレームポインタとして使用されなければ W0-W13, and W14。
`int, signed int,unsigned int`	フレームポインタとして使用されなければ W0-W13, and W14。
`void * (or any pointer)`	フレームポインタとして使用されなければ W0-W13, and W14。
`long, signed long, unsigned long`	連続する二つのレジスタで、最初のものが {WO, W2, W4, W6, W8, W10, W12}. からのレジスタ。低い番号のレジスタが値の下位 16 ビットを含みます。
`long long, signed long long, unsigned long long`	四つの連続するレジスタで、最初のものが {WO, W4, W8} からのレジスタ。低い番号のレジスタが値の下位 16 ビットを含みます。続く高い番号のレジスタが、続く上位のビットを含みます。
`float`	連続する二つのレジスタで、最初のものが {WO, W2, W4, W6, W8, W10, W12} からのレジスタ。低い番号のレジスタが値の下位 16 ビットを含みます。
`double*`	連続する二つのレジスタで、最初のものが {WO, W2, W4, W6, W8, W10, W12} からのレジスタ。低い番号のレジスタが上位 16 ビットを含みます。
`long double`	四つの連続するレジスタで、最初のものが {WO, W4, W8} からのレジスタ。低い番号のレジスタが値の下位 16 ビットを含みます。続く高い番号のレジスタが、続く上位のビットを含みます。

* `double` は `-fno-short-double` が使用されている場合は `long double` と同等です。

4.14 ビット反転とモジュロアドレッシング

コンパイラーはビット反転やモジュロアドレッシングの使用は直接サポートしません。もしこのうちのどちらかのアドレッシングがレジスタに対して有効にされているとき、コンパイラーが、そのレジスタをポインタとして使用しないようにするのはプログラマの責任です。もしこれらのアドレッシングモードのうちの 1 つが有効の時に割り込みが発生するときには、特に注意しなければなりません。

モジューロアドレッシングに適したメモリ配列のために、C 内の配列を定義できます。増分モジューロバッファとして使用する配列を定義するのに aligned 属性が使用できます。減分モジューロバッファとして使用する配列を定義するのには、reverse 属性が使用できます。これら属性について詳しくは、**セクション 2.3「キーワードの違い」**を参照してください。モジューロアドレッシングについて詳しくは、*dsPICSOF Family Reference Manual* (DS70046) の第 3 章を参照してください。

4.15 PROGRAM SPACE VISIBILITY (PSV) の使用方法

デフォルトでは、コンパイラは、ストリングと定数化され初期化された変数を `.const` セクション内に割り当てるように自動的にアレンジします。`.const` セクションは PSV ウインドウにマッピングされます。PSV 管理は、それを移動しないコンパイラ管理に託され、PSV ウィンドウ自体のサイズでアクセス可能なプログラムのサイズを制限します。

他には、アプリケーションが、それ自体の目的のために自ら、PSV ウィンドウの制御を行うこともあります。アプリケーションの中で PSV を直接コントロールを行う長所として、PSV ウィンドウに永久にマッピングされる 1 つの `.const` セクションを持つよりもさらに柔軟になることを可能にする点が挙げられます。短所は、アプリケーションが PSV コントロールレジスタとビットを管理しなければならないことです。PSV ウィンドウを使わないようにコンパイラーに指示するには、`-mconst-in-data` オプションを指定します。

`space` 属性は PSV ウィンドウ内に配置する変数を定義するのに使用できます。コンパイラーが管理するセクション `.const` 内に特定の変数を配置するには、`space(auto_psv)` 属性を使用します。コンパイラーに管理されていないセクション内の PSV 領域に変数を配置するには、`space(psv)` 属性を使用します。これら属性について詳しくは、**セクション 2.3「キーワードの違い」**を参照してください。

PSV 使用について詳しくは、*MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide*. (DS51317) をご参照ください。

注意：

第 5 章 データタイプ

5.1 序章

本章では MPLAB C30 データタイプについて説明します。

5.2 ハイライト

本章で説明される項目は以下の通りです。

・データの表現
・整数
・浮動少数点
・ポインタ

5.3 データの表現

複数バイトの量が“little endian”フォーマットでストアされ、それは下記を意味します。

- 最低位のアドレスに下位 8 ビットがストアされます。
- 最下位の番号をつけられたビット位置に最下位ビットがストアされます。

例として、0x12345678 のロング値はアドレス 0x100 に以下のようにストアされます。

0x100	0x78	0x56	0x101
0x102	0x34	0x12	0x103

別の例として、0x12345678 のロング値はレジスタ w4 と w5 に以下のようにストアされます。

w4	w5
0x5678	0x1234

5.4 整数

表 5-1 に MPLAB C30 でサポートされる整数データタイプを示します。

表 5-1: 整数データタイプ

タイプ	ビット	最小	最大
char, signed char	8	-128	127
unsigned char	8	0	255
short, signed short	16	-32768	32767
unsigned short	16	0	65535
int, signed int	16	-32768	32767
unsigned int	16	0	65535
long, signed long	32	-2^{31}	$2^{31} - 1$
unsigned long	32	0	$2^{32} - 1$
long long**, signed long long**	64	-2^{63}	$2^{63} - 1$
unsigned long long**	64	0	$2^{64} - 1$

** ANSI-89 extension

実装時定義された、整数の動作については **セクション A.6「整数」**をご参照ください。

5.5 浮動少数

MPLAB C30 は IEEE-754 フォーマットを使用します。表 5-2 にサポートされる浮動少数点データタイプを示します。

表 5-2: 浮動少数データタイプ

タイプ	ビット	E 最小	E 最大	N 最小	N 最大
float	32	-126	127	2^{-126}	2^{128}
double*	32	-126	127	2^{-126}	2^{128}
long double	64	-1022	1023	2^{-1022}	2^{1024}

E = 指数
N = 正規化された（概数）
* double は -fno-short-double が使用されている場合は long double と同等です。

実装時定義された浮動小数の動作については、**セクション A.7「浮動小数点」**の章をご参照ください。

5.6 ポインタ

すべての MPLAB C30 ポインタは、16-bits 幅です。これは全データ空間 (64 KB) のアクセスとスモールコードモデル（32 K のコード）には十分です。ラージコードモデル (>32 K ワードのコード) では、ポインタが解決のハンドルを握ります、つまり、ポインタが、プログラム空間の最初の 32 K ワードに位置付けられる GOTO 命令用のアドレスになります。

第 6 章 デバイスサポートファイル

6.1 序章

本セクションでは MPLAB C30 コンパイルのサポートで使用されるデバイスファイルについて説明します。

6.2 ハイライト

本セクションで説明される項目は以下の通りです。

- プロセッサヘッダファイル
- レジスタ定義ファイル
- SFR の使用
- マクロ使用
- C コードからの EEDATA へのアクセス

6.3 プロセッサヘッダファイル

プロセッサヘッダファイルは言語ツールで配布されます。これらのヘッダファイルは、一つ一つの dsPIC デバイス用に利用できる特別関数レジスタ (SFR's) を定義します。C の中でヘッダファイルを使用するには、

```
#include <p30fxxxx.h>
```

を使います。

ここで xxxx はデバイスパーツ番号に対応します。C ヘッダファイルは、support\h ディレクトリに配置されます。

ヘッダファイルのインクルードは、SFR 名（すなわち CORCON ビット）を使用するために必要です。

例えば、以下の PIC30F2010 用にコンパイルされたモジュールは 2 つの関数を含みます。1 つは PSV ウインドウを有効にするものと、もう 1 つは PSV ウインドウを無効にするものです。

```
#include <p30f2010.h>
void
EnablePSV(void)
{
    CORCONbits.PSV = 1;
}
void
DisablePSV(void)
{
    CORCONbits.PSV = 0;
}
```

プロセッサヘッダファイルの集合は、それぞれの SFR が、そのデバイスのデータシート内に現れるものと同じ名前を使って名づけられます。例えば、コアコントロールレジスタは CORCON となります。もしレジスタが関連のある個別のビットを持っていたら、その SFR のために定義された構造体があり、構造体の名前は SFR の名前と同じで、“bits”が付属されます。例えば、CORCONbits はコア制御 レギスタの名前です。個別のビット（もしくはビットフィールド）はデータシート内の名前を使用して、構造の中で名づけられます。例えば CORCON レジスタの PSV ビットは PSV と名づけます。以下に CORCON の全定義を示します（変更もあります）。

```
/* CORCON: CPU Mode control Register */
extern volatile unsigned int CORCON __attribute__((__near__));
typedef struct tagCORCONBITS {
       unsigned IF      :1;    /* Integer/Fractional mode             */
       unsigned RND     :1;    /* Rounding mode                       */
       unsigned PSV     :1;    /* Program Space Visibility enable     */
       unsigned IPL3    :1;
       unsigned ACCSAT  :1;    /* Acc saturation mode                 */
       unsigned SATDW   :1;    /* Data space write saturation enable  */
       unsigned SATB    :1;    /* Acc B saturation enable             */
       unsigned SATA    :1;    /* Acc A saturation enable             */
       unsigned DL      :3;    /* DO loop nesting level status        */
       unsigned         :4;
} CORCONBITS;
extern volatile CORCONBITS CORCONbits __attribute__((__near__));
```

注： シンボル CORCON と CORCONbits は同じ名前を参照し、リンク時に同じアドレスを決定します。

6.4 レジスタ定義ファイル

セクション 6.3「プロセッサヘッダファイル」で記述されているプロセッサヘッダファイルはそれぞれの部品のすべての SFR's を名づけますが、SFR's のアドレスを定義するものではありません。デバイス特有のリンカスクリプトファイルが個別にそれぞれに用意されていて、support\gld ディレクトリに配置されています。これらのリンカスクリプトファイルが SFR アドレスを定義します。これらのファイルの 1 つを使用するには、以下のリンカコマンドラインオプションを指定します。

```
-T p30fxxxx.gld
```

ここで、xxxx はデバイス部品番号に相当します。

例えば、app2010.c と名づけられたファイルが存在すると仮定し、それが PIC30F2010 部品のアプリケーションを含んでいるとすると、以下のコマンドラインを用いてコンパイル、リンクが実行します。

```
pic30-gcc -o app2010.o -T p30f2010.gld app2010.c
```

-o コマンドラインオプションは、出力 COFF 実行可能ファイルに名前をつけ、-T オプションは PIC30F2010 部品の名前を与えます。もし、p30f2010.gld が現ディレクトリ内に見つからない場合は、リンカは既知のライブラリパスの中を検索します。デフォルトのインストールでは、リンカスクリプトは PIC30_LIBRARAY_PATH に含まれます。参考として、**セクション 3.6「環境変数」**をご覧ください。

6.5 SFRs の使用

SFR's をアプリケーション内で使用するには、従うべき 3 つのステップがあります。

1. 適切なデバイス用のプロセッサヘッダファイルをインクルードします。これは、そのデバイスで利用できる SFR's と一緒にソースコードを提供します。例えば、以下の文は PIC30F6014 部品用のヘッダファイルをインクルードします。

   ```
   #include <p30f6014.h>
   ```

2. 他の C 変数のように SFR's にアクセスします。ソースコードが SFR's に書き込まれるか、SFR's から読み出されます。

   例えば、以下の文は Timer1 用の特殊関数レジスタ内のすべてのビットをゼロにクリアします。

   ```
   TMR1 = 0;
   ```

   この次の文は、T1CON レジスタの 15 番目のビット（‘timer on’ビット）を表し、タイマをスタートする TON と名づけられたビットに 1 をセットします。

   ```
   T1CONbits.TON = 1;
   ```

3. レジスタ定義ファイルもしくは適切なデバイス用のリンカスクリプトと一緒にリンクします。リンカは SFR's のアドレスを与えます。（ビット構造体が、リンク時に SFR と同じアドレスを持つことを思い出してください。）例 6-1 は下記を使っています。

   ```
   p30f6014.gld
   ```

リンカスクリプトの使用の詳細については、*MPLAB ASM30, MPLAB LINK30 と Utilities User's Guide* (DS51317) をご参照ください。

以下の例は、サンプルのリアルタイムクロックです。これはいくつかの SFR's を使用します。これらの SFR's の記述は `p30f6014.h` ファイルで見つけられます。このファイルは、`p30f6014.gld` であるデバイスの特殊リンカスクリプトと一緒にリンクされます。

例 6-1: サンプルリアルタイムクロック

```
/*
** Sample Real Time Clock for dsPIC
**
**  Uses Timer1, TCY clock timer mode
**  and interrupt on period match
*/

#include <p30f6014.h>

/* Timer1 period for 1 ms with FOSC = 20 MHz */
#define TMR1_PERIOD 0x1388

struct clockType
  {
   unsigned int timer;     /* countdown timer, milliseconds */
   unsigned int ticks;     /* absolute time, milliseconds   */
   unsigned int seconds;   /* absolute time, seconds        */
  } volatile RTclock;

void reset_clock(void)
  {
   RTclock.timer = 0;      /* clear software registers  */
   RTclock.ticks = 0;
   RTclock.seconds = 0;

   TMR1 = 0;               /* clear timer1 register     */
   PR1 = TMR1_PERIOD;      /* set period1 register      */
   T1CONbits.TCS = 0;      /* set internal clock source */
   IPC0bits.T1IP = 4;      /* set priority level        */
   IFS0bits.T1IF = 0;      /* clear interrupt flag      */
   IEC0bits.T1IE = 1;      /* enable interrupts         */

   SRbits.IPL = 3;         /* enable CPU priority levels 4-7*/
   T1CONbits.TON = 1;      /* start the timer*/
  }
void __attribute__((__interrupt__)) _T1Interrupt(void)
  { static int sticks=0;

    if (RTclock.timer > 0)   /* if countdown timer is active */
        RTclock.timer -= 1;  /*  decrement it                */
    RTclock.ticks++;         /* increment ticks counter     */
    if (sticks++ > 1000)
    {                           /* if time to rollover         */
    sticks = 0;              /*  clear seconds ticks     */
    RTclock.seconds++;       /*  and increment seconds   */
    }

    IFS0bits.T1IF = 0;       /* clear interrupt flag     */
    return;
  }
```

6.6 マクロの使用

プロセッサヘッダファイルは、特殊関数レジスタ (SFR) に加えて、デジタルシグナルコントローラ (DSCs) の dsPIC30F ファミリ用の役に立つマクロを定義しています。

6.6.1 コンフィギュレーションビット設定マクロ

マクロはコンフィギュレーションビットを設定するため、例えば、マクロを使用して FOSC ビットを設定するために C ソースコードが始まる前に、下記のコードラインが挿入されます。

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

これは、PLL を用いた外部クロックを 16x に設定し、クロックスイッチとフェイルセーフクロックモニタリングを有効にします。

同様に FBORPOR ビットを設定するために、以下のコードラインが挿入されます。

```
_FBORPOR(PBOR_ON & BORV_27 & PWRT_ON_64 & MCLR_DIS);
```

これはブラウンアウトリセットを 2.7 ボルトで有効にし、パワアップタイマーを 64 ミリ秒に初期化し、I/O 用に MCLR ピンを使用するコンフィギュレーションに設定します。

それぞれのコンフィギュレーションビットを有効にする完全な設定リストについては、プロセッサヘッダファイルをご参照ください。

6.6.2 インラインアセンブリ使用マクロ

C 内でアセンブリコードを定義するために使用されるマクロのいくつかが以下にリストアップされています。

```
#define Nop()    {__asm__ volatile ("nop");}
#define ClrWdt() {__asm__ volatile ("clrwdt");}
#define Sleep()  {__asm__ volatile ("pwrsav #0");}
#define Idle()   {__asm__ volatile ("pwrsav #1");}
```

6.6.3 データメモリ割り当てマクロ

データメモリ内に空間を割り当てるために使用されるマクロが以下となります。それらには引数を必要とするものと必要としないものの２つのタイプがあります。

以下のマクロは、整列を指定する引数 N を必要とします。N は、2 の乗数で最小値は 2 です。

```
#define _XBSS(N)   __attribute__((space(xmemory), aligned(N)))
#define _XDATA(N)  __attribute__((space(xmemory), aligned(N)))
#define _YBSS(N)   __attribute__((space(ymemory), aligned(N)))
#define _YDATA(N)  __attribute__((space(ymemory), aligned(N)))
#define _EEDATA(N) __attribute__((space(eedata),  aligned(N)))
```

例えば、32 バイトアドレスに整列される、X メモリ内の初期化されないアレイを宣言するには、以下のように記述します。

```
int _XBSS(32) xbuf[16];
```

特に整列を使用しない、データ EEPROM 内の初期化されないアレイを宣言するには、以下のように記述します。

```
int _EEDATA(2) table1[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
```

以下のマクロは、引数を必要としません。それらは、永続データメモリ内もしくはニアデータメモリ内に変数を配置するために使用されます。

```
#define _PERSISTENT __attribute__((persistent))
#define _NEAR       __attribute__((near))
```

例えば、デバイスリセットされてもその値を保持する 2 つの変数を宣言するには、以下のように記述します。

```
int _PERSISTENT var1,var2;
```

6.6.4 ISR 宣言マクロ

以下のマクロは割り込みサービスルーチーン (ISRs) を宣言するために使用されます。

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

例えば、タイマ 0 割り込み用の ISR を宣言するには、以下のように記述します。

```
void _ISR _INT0Interrupt(void);
```

例えば、高速のコンテキスト保存ができる SPI1 割り込み用の ISR を宣言するには、以下のように記述します。

```
void _ISRFAST _SPI1Interrupt(void);
```

> **注 :** ISRs は、もし表 7-1 にリストアップされている予約名が使用されていると、自動的に割り込みベクタテーブルにインストールされます。

6.7 C コードからの EEDATA へのアクセス

MPLAB C30 はデバイスの EE データ空間内へのデータ配置を可能にする便利なマクロ定義をいくつか提供します。この定義は以下を行なうだけで非常に簡単にできます。

```
int _EEDATA(2) user_data[] =  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

user_data は EE データ空間内に配置され、初期値で 10 ワードを確保します。

dsPIC デバイスはメモリのこの空間にアクセスする手段をプログラマに提供します。最初の方法は、PSV ウィンドウを介したやり方です。二つ目の方法は、特殊なマシン命令（TBLRDx）を使用したやり方です。

6.7.1 PSV を介した EEDATA へのアクセス

コンパイラーは通常、PSV ウィンドウを制御してプログラムメモリ内に格納された定数にアクセスします。また、それ以外にも、PSV ウィンドウは EEDATA メモリへのアクセスにも使用されます。

PSV ウィンドウを使用するには :

- PSVPAG レジスタをアクセスする先のプログラムメモリ内の適切なアドレスに設定する必要があります。EE データでは、OxFF がこれに当たりますが、__builtin_psvpage() 関数を使うのが一番良い方法です。
- PSV ウィンドウも CORCON レジスタの PSV ビットを設定して有効化する必要があります。このビットが設定されていないと、PSV ウィンドウは常に 0x0000 を読み込みます。

例 6-2: PSV を介した EEDATA へのアクセス

```
#include <p30fxxxx.h>
int main(void) {
  PSVPAG = __builtin_psvpage(&user_data);
  CORCONbits.PSV = 1;

  /* ... */

  if (user_data[2]) ;/* do something */

  }
```

この手順は 1 度だけ行います。PSVPAG が変更されない限り、EE データ空間内の変数は、例で示されるように、通常の C 変数としてこれらを参照することで読み込まれます。

> **注 :** このアクセスモデルはコンパイラー管理の PSV (-mconst-in-code) モデルとの互換性はありません。競合が起こらないよう注意してください。

6.7.2 TBLRDx 命令を使用した EEDATA へのアクセス

TBLRDx 命令はコンパイラーで直接サポートされていませんが、インラインアセンブリを介して使用できます。PSV アクセスのように、23- ビットアドレスは SFR 値から形成され、命令の一部としてエンコードされます。前述の例と同じメモリにアクセスするには、以下のコードを使用できます。

TBLRDx 命令を使用するには：

- TBLPAG レジスタをアクセスする先のプログラムメモリ内の適切なアドレスに設定する必要があります。EE データでは、これは 0x7F に当たりますが、`__builtin_tblpage()` 関数を使うのが一番よい方法です。
- TBLRDx 命令は `_asm__` ステートメントのみからアクセスできます。この命令についての情報は、*Programmers Reference Manual* (DS70030) を参照してください。

例 6-3: テーブル読み込みを介した EEDATA へのアクセス

```
#include <p30fxxxx.h>
#define eedata_read(src, dest) {   \
  register int eedata_addr;        \
  register int eedata_val;         \
                                   \
  eedata_addr = __builtin_tbloffset(&src); \
  __asm__("tblrdl [%1], %0" : "=r"(eedata_val) : "r"(eedata_addr)); \
  dest = eedata_val;               \
  }

int main(void) {
  int value;

  TBLPAG = __builtin_tblpage(&user_data);

  eedata_read(user_data[2], value);
  if (value) ; /* do something */

  }
```

6.7.3 追加の情報源

dsPIC30F Family Reference Manual (DS70054) のセクション 5 では、dsPIC デバイスで提供される FLASH プログラムメモリと EE データメモリの使用について有意義な議論がなされています。このセクションは、プログラムメモリと EE データメモリ両方の実行時プログラミングに関する情報を含んでいます。

注意：

第 7 章 割り込み

7.1 序章

割り込み処理は、ほとんどのマイクロコントローラアプリケーションの重要な一面です。割り込みは、実時間で発生するイベントとソフトウエア動作との同期を取るために用いられます。割り込みが発生すると、通常のソフトウエア実行フローは中断され、特殊関数が起動し、そのイベントが処理されます。割り込み処理が終了すると、それ以前のコンテキスト情報が復帰され通常の実行が継続されます。

dsPIC30F デバイスは、内部と外部両方のソースからの複数割り込みをサポートします。さらに、そのデバイスでは、実行中の低プライオリティ割り込みはいつでも高プライオリティ割り込みに上書きされます。

MPLAB C30 コンパイラは C もしくはインラインアセンブリコードの割り込み処理をフルサポートします。この章では割り込み処理の概要について説明します。

7.2 ハイライト

この章では以下の項目について説明します。

- **割り込みサービスルーチンを記述する**－一つもしくはそれ以上の C 関数を割り込みサービスルーチン (ISR's) として指定し、割り込みの発生により起動されるようにします。一般に最高性能を出すために、長時間を要する計算もしくはライブラリコールを必要とするオペレーションはメインアプリケーション内に置きます。この方針は性能を最適化し、割り込みイベントが急に発生した時に、情報を失う可能性を最小にします。
- **割り込みベクタを記述する**－ dsPIC30F デバイスは、割り込みが発生した時にアプリケーションコントロールを転送するために割り込みベクタを使用します。割り込みベクタは、ISR ごとのアドレスを指定し、プログラムメモリ内の固定位置に置かれます。アプリケーションは、割り込みを使用するために、これらの位置に有効な関数アドレスを設定します。
- **割り込みサービスルーチンコンテキストの保存**－割り込みから、割り込み以前と同じ条件状態で戻るようにするため、特殊レジスタなどのコンテキスト情報が保存されなければなりません。
- **レイテンシー**－割り込みがコールされた時から、最初の ISR 命令が実行されるまでの間の時間が、割り込みのレイテンシーです。
- **割り込みのネスティング**－ MPLAB C30 はネスティングされた割り込みをサポートします。
- **割り込みの有効化 / 無効化**－割り込みソースの許可・禁止は二つのレベル、グローバルと個別的、で発生します。

7.3 割り込みサービスルーチンを記述する

この章のガイドラインに従えば、割り込みサービスルーチン (ISRs), も含めて、C 言語構文だけで、すべてのアプリケーションコードを記述することができます。

7.3.1 ISR's を記述するためのガイドライン

ISR's を記述するためのガイドラインは以下の通りです。

- パラメータ無しで、有効な戻りタイプを宣言する。(必須)
- ISR's がメインのラインコードでコールされないようにする。(必須)
- ISR's が、その他の関数をコールしないようにする。(推奨)

MPLAB C30 ISR はその他の C 関数同様、その中にローカル変数を持ったり、グローバル変数にアクセスしたります。しかし、ISR は、パラメータ無し、戻り値無しで宣言される必要があります。このことは、ISR が、ハードウエア割り込みもしくはトラップに対する反応として、メインラインの C プログラムに対して非同期的に起動されるために必要です (すなわち、通常の方法ではコールされず、そのためパラメータや戻り値は適用されないのです)。

ISR's は、ハードウエア割り込みもしくはトラップだけで起動されるべきで、その他の C 関数から起動してはなりません。ISR は、関数から抜け出るためには、通常の RETURN 命令ではなく割り込みからの戻り命令 (RETFIE) を使用します。割り込み処理以外で RETFIE 命令を使用すると、ステータスレジスタのように、プロセッサリソースを破壊します。

最後に、ISR's は他の関数をコールすべきではありません。これは、レイテンシーの問題があるために、推奨されています。詳細については、**セクション 7.6「レイテンシー」**をご参照ください。

7.3.2 ISR's を記述するための構文

C 関数を割り込みハンドラとして宣言するには、関数に割り込み属性でタグ付けします (§ 2.3、__attribute__ keyword の説明を参照)。割り込み属性の構文は以下の通りです。

```
__attribute__((interrupt [(
                          [ save(symbol-list)]
                          [, irq(irqid)]
                          [, altirq(altirqid)]
                          [, preprologue(asm)]
                          )]
              ))
```

interrupt 属性の名前とパラメータの名前は、名前の前後に一対のアンダースコア文字をつけて記述しても構いません。従って、interrupt と __interrupt__ は同等であり、save と __save__ は同等です。

オプションの save パラメータは、ISR への入り口と ISR からの出口で保存・回復されるべき 1 つもしくはそれ以上の変数のリストを指定します。この名前リストは、括弧内に記載し、名前はコンマで区切ります。

もし値が外部で使われないようにするには、ISR 内で修正されるであろうグローバル変数を保存するようにしなければなりません。ISR 内で修正されるグローバル変数は、volatile の属性をもたねばなりません。

オプションの irq パラメータは、指定した割り込みの割り込みベクタを指定し、オプションの altirq パラメータは、指定割り込みの代替え割り込みベクタを指定します。それぞれのパラメータは、括弧でくくられた ID 番号を必要とします。(割り込み ID's のリストに関しては、**セクション 7.4「割り込みベクトルを記述する」**を参照してください。)

オプションの preprologue パラメータは、コンパイラが生成する関数 preprologue の直前に、アセンブリ言語文を生成コードに挿入します。

7.3.3 ISR's のコーディング

以下のプロトタイプは、関数 isr0 が割り込みハンドラであることを宣言します。

```
void __attribute__((__interrupt__)) isr0(void);
```

このプロトタイプが示すように、割り込み関数は、パラメータを取ったりすべきではなく、値を戻したりもしません。コンパイラはすべてのワーキングレジスタを保存し、もし必要であれば、ステータスレジスタやリピートカウントレジスタも保存します。その他の変数は、interrupt 属性のパラメータとして名前を指定することで保存されます。例えば、コンパイラが自動的に変数 var1 や var2 を保存・回復するためには、以下のプロトタイプを使用します。

```
void __attribute__((__interrupt__(__save__(var1,var2)))) isr0(void);
```

コンパイラが（push.s と pop.s 命令を用いて) 高速コンテキスト保存を使用することを要求する場合には、関数に shadow 属性を付けます。**セクション 2.3.2「関数の属性を指定する」**を参照してください。) 例えば以下のように書きます。

```
void __attribute__((__interrupt__, __shadow__)) isr0(void);
```

7.3.4 マクロを用いて単純 ISRs を宣言する

割り込みハンドラが、割り込み属性のオプションパラメータを必要としないなら、簡略化された構文が使用できます。以下のマクロが、デバイス特有のヘッダファイル内で定義されています。

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

例えば、タイマ 0 割り込み用の割り込みハンドラを宣言するには、以下のように書きます。

```
#include <p30fxxxx.h>
void _ISR _INT0Interrupt(void);
```

高速コンテキスト保存を使う SPI1 割り込み用の割り込みハンドラを宣言するには、以下のように書きます。

```
#include <p30fxxxx.h>
void _ISRFAST _SPI1Interrupt(void);
```

7.4 割り込みベクトルを記述する

dsPIC デバイスは 2 つの割り込みベクトル表 - 一次テーブルと代替えテーブル - を持っています。それぞれの表は 62 の割り込みベクトルを含みます。

62 の割り込みソースは、一次と代替割り込みベクトルと関連づけられ、表 7-1 にあるようにプログラムワードを占有します。代替ベクトル名は、INTCON2 レジスタ内の ALTIVT ビットが設定される時に使用されます。

> **注 :** dsPIC デバイスリセットは割り込みベクトル表で取り扱われません。その代わり、デバイスリセットにより、dsPIC プログラムカウンタがクリアされます。これにより、プロセッサがアドレスゼロから実行を始めるようにします。通常リンカースクリプトは、C 実行時のスタートアップモジュールをジャンプ先とする GOTO 命令を 0 番地に配置します。

表 7-1: 割り込みベクトル

IRQ#	ベクトル関数	Primary 名	Alternate 名
n/a	予約	_ReservedTrap0	_AltReservedTrap0
n/a	Oscillator 失敗トラップ	_OscillatorFail	_AltOscillatorFail
n/a	Address エラートラップ	_AddressError	_AltAddressError
n/a	Stack エラートラップ	_StackError	_AltStackError
n/a	Math エラートラップ	_MathError	_AltMathError
n/a	予約	_ReservedTrap5	_AltReservedTrap5
n/a	予約	_ReservedTrap6	_AltReservedTrap6
n/a	予約	_ReservedTrap7	_AltReservedTrap7
0	INTO- 外部割込み 0	_INT0Interrupt	_AltINT0Interrupt
1	ICI- 入力キャプチャ 1	_IC1Interrupt	_AltIC1Interrupt
2	OC1- 出力コンペア 1	_OC1Interrupt	_AltOC1Interrupt
3	TMR1- タイマ 1	_T1Interrupt	_AltT1Interrupt
4	IC2- 入力キャプチャ 2	_IC2Interrupt	_AltIC2Interrupt
5	OC2- 出力コンペア 2	_OC2Interrupt	_AltOC2Interrupt
6	TMR2- タイマ 2	_T2Interrupt	_AltT2Interrupt
7	TMR3- タイマ 3	_T3Interrupt	_AltT3Interrupt
8	SPI1- シリアル周辺インターフェース 1	_SPI1Interrupt	_AltSPI1Interrupt
9	UART1RX-UART1 受信	_U1RXInterrupt	_AltU1RXInterrupt
10	UART1TX-UART1 送信	_U1TXInterrupt	_AltU1TXInterrupt
11	ADC-ADC 変換完了	_ADCInterrupt	_AltADCInterrupt
12	NVM-NVM 書き込み完了	_NVMInterrupt	_AltNVMInterrupt
13	Slave I^2C 割込み	_SI2CInterrupt	_AltSI2CInterrupt
14	Master I^2C 割込み	_MI2CInterrupt	_AltMI2CInterrupt
15	CN- 入力変化割込み	_CNInterrupt	_AltCNInterrupt
16	INT1- 外部割込み 1	_INT1Interrupt	_AltINT1Interrupt
17	IC7- 入力キャプチャ 7	_IC7Interrupt	_AltIC7Interrupt
18	IC8- 入力キャプチャ 8	_IC8Interrupt	_AltIC8Interrupt
19	OC3- 出力コンペア 3	_OC3Interrupt	_AltOC3Interrupt
20	OC4- 出力比較 4	_OC4Interrupt	_AltOC4Interrupt
21	TMR4- タイマ 4	_T4Interrupt	_AltT4Interrupt
22	TMR5- タイマ 5	_T5Interrupt	_AltT5Interrupt
23	INT2- 外部割込み 2	_INT2Interrupt	_AltINT2Interrupt

表 7-1: 割り込みベクトル

IRQ#	ベクトル関数	Primary 名	Alternate 名
24	UART2RX-UART2 受信	_U2RXInterrupt	_AltU2RXInterrupt
25	UART2TX-UART2 送信	_U2TXInterrupt	_AltU2TXInterrupt
26	SPI2- シリアル周辺インターフェース 2	_SPI2Interrupt	_AltSPI2Interrupt
27	CAN 1- 複合 IRQ	_C1Interrupt	_AltC1Interrupt
28	IC3- 入力キャプチャ 3	_IC3Interrupt	_AltIC3Interrupt
29	IC4- 入力キャプチャ 4	_IC4Interrupt	_AltIC4Interrupt
30	IC5- 入力キャプチャ 5	_IC5Interrupt	_AltIC5Interrupt
31	IC6- 入力キャプチャ 6	_IC6Interrupt	_AltIC6Interrupt
32	OC5- 出力比較 5	_OC5Interrupt	_AltOC5Interrupt
33	OC6- 出力比較 6	_OC6Interrupt	_AltOC6Interrupt
34	OC7- 出力比較 7	_OC7Interrupt	_AltOC7Interrupt
35	OC8- 出力比較 8	_OC8Interrupt	_AltOC8Interrupt
36	INT3- 外部割込み 3	_INT3Interrupt	_AltINT3Interrupt
37	INT4- 外部割込み 4	_INT4Interrupt	_AltINT4Interrupt
38	CAN2- 複合 IRQ	_C2Interrupt	_AltC2Interrupt
39	PWM-PWM 周期一致	_PWMInterrupt	_AltPWMInterrupt
40	QEI- ポジションカウンタ比較	_QEIInterrupt	_AltQEIInterrupt
41	DCI-CODEC 変換完了	_DCIInterrupt	_AltDCIInterrupt
42	PLVD- 低電圧検出	_LVDInterrupt	_AltLVDInterrupt
43	FLTA-MPWM 失敗 A	_FLTAInterrupt	_AltFLTAInterrupt
44	FLTB-MPWM 失敗 B	_FLTBInterrupt	_AltFLTBInterrupt
45	予約	_Interrupt45	_AltInterrupt45
46	予約	_Interrupt46	_AltInterrupt46
47	予約	_Interrupt47	_AltInterrupt47
48	予約	_Interrupt48	_AltInterrupt48
49	予約	_Interrupt49	_AltInterrupt49
50	予約	_Interrupt50	_AltInterrupt50
51	予約	_Interrupt51	_AltInterrupt51
52	予約	_Interrupt52	_AltInterrupt52
53	予約	_Interrupt53	_AltInterrupt53

割り込みを処理するために、関数アドレスが、ベクトル表内の対応するアドレスに置かれなければなりません。また、関数は、用いるシステムリソースを保存しなければなりません。関数は、RETFIE プロセッサ命令を用いて、以前のタスクに戻らねばなりません。割り込み関数は、C で記述することもできます。C 関数が割り込みハンドラとして指定されると、コンパイラは、コンパイラが使用するすべてのシステムリソースを待避するようにし、適切な命令を使用して関数から戻るようにアレンジします。コンパイラは、割り込み関数のアドレスで埋められた割り込みベクトル表を、オプションとして作成することができます。

コンパイラが、割り込み関数を指す割り込みベクタを満たすようにアレンジするために、前記の表に記載されたように関数に名前をつけます。例えば、スタックエラーベクタは、以下のように関数が定義されると、自動的にベクタ表にアドレスを生成します。

```
void __attribute__((__interrupt__)) _StackError(void);
```

先行するアンダースコアを使用することに注意してください。同様に alternate スタックエラーベクタは、以下のように関数が定義されると、自動的に満たされます。

```
void __attribute__((__interrupt__)) _AltStackError(void);
```

再び、先行するアンダースコアを使用することに注意してください。

特定のハンドラを持たないすべての割り込みベクタ用として、デフォルトの割り込みハンドラがあらかじめインストールされています。デフォルト割り込みハンドラはリンカが与え、単にデバイスをリセットします。`_DefaultInterrupt` という名前を持った割り込み関数を宣言することで、アプリケーション用のデフォルトの割り込みベクタを生成することができます。

それぞれの表の最後の 9 つの割り込みベクタは事前に定義されたハードウエア関数を持っていません。これらの割り込み用ベクタは前記の表にしめされた名前を用いることで満たすか、もしくはアプリケーションにとってより適切な名前を使用します。または、割り込み属性のパラメータ `irq` もしくは `altirq` を使うことにより適切なベクタエントリを満たすことができます。例えば、関数が 52 の初期割り込みベクタを使用するように指定するには、以下のように使用します。

```
void __attribute__((__interrupt__(__irq__(52)))) MyIRQ(void);
```

同様に、関数が 52 の代替割り込みベクタを使用するように指定するには、以下のように使用します。

```
void __attribute__((__interrupt__(__altirq__(52)))) MyAltIRQ(void);
```

`irq/altirq` に指定する数は45 から 53 の割り込み要求の一つです。割り込み属性の `irq` パラメータを使用すると、コンパイラーは外部シンボル `__Interruptn` を生成し、n はベクトル番号になります。従って、C 識別子 `_Interrupt45` から `_Interrupt53` はコンパイラーによって確保されています。同様に、割り込み属性の `altirq` パラメータが使用されると、`__AltInterruptn` という外部シンボルを生成し、n はベクタ番号です。従って、C 識別子の `_AltInterrupt45` から `_AltInterrupt53` はコンパイラーによって確保されています。

7.5 割り込みサービスルーチンのコンテキスト保存

割り込みは、まさにその性質により、予想できない時に発生します。従って、割り込まれたコードは、割り込みが発生した時に存在したのと同じマシン状態で復帰されねばなりません。

割り込みからの戻りを適切に取り扱うためには、ISR 関数用の設定 (prologue) コードが、ISR の終わりで、復帰用スタック上の、コンパイラが管理するワーキングレジスタと特殊関数レジスタを復帰します。追加変数と特殊関数レジスタが待避・復帰されるように指定するために、`interrupt` 属性のオプション `save` パラメータを使用できます。

あるアプリケーションでは、アセンブリ文を、コンパイラが生成する関数プロローグの直前に、割り込みサービスルーチンに挿入することが必要です。例えば、割り込みサービスルーチンへの入り口で、semaphore がすぐに増加されることが必要であるとします。これは以下のように記述されます。

```
void
__attribute__((__interrupt__(__preprologue__("inc _semaphore"))))
  isr0(void);
```

7.6 レイテンシー

割り込みソースが発生してから ISR コードの最初の命令が実行されるまでの間のサイクル数に影響を与える要素が二つあります。それらは以下の通りです。

- **割り込みのプロセッササービス**－割り込みを認識し、最初の割り込みベクタに分岐するために、プロセサが必要とする総時間です。この値を決定するには、そのプロセッサと使用される割り込みソースに関して、プロセッサのデータシートを参照ください。
- **ISR コード**－ MPLAB C30 は ISR 内で使用するレジスタを保存します。これは、ワーキングレジスタと RCOUNT 特殊関数レジスタを含みます。さらに、もし ISR が通常関数をコールすると、コンパイラは、すべてのワーキングレジスタと RCOUNT を（たとえそれらが ISR 自身の中ですべて明確に使用されるわけでは無いとしても）保存します。コンパイラは、一般的に、どのリソースがコールされた関数で使用されるかわからないので、こうするわけです。

7.7 割り込みのネスティング

dsPIC30F デバイスは割り込みのネスティングをサポートします。プロセッサリソースは ISR 内のスタック上に保存されるので、ネスティングされる ISR's はネスティングされないものと同様にコーディングできます。割り込みのネスティングは、INTCON1 レジスタの NSTDIS（ネスティングされた割り込み禁止）ビットをクリアすることで可能となります。dsPIC30F デバイスはネスティング割り込み可能な状態でリセットから復帰しますので、これがデフォルト状態です。それぞれの割り込みソースは、割り込みプライオリティコントロールレジスタ（IPC*n*）でプライオリティが割り当てられます。もし、プロセッサステータスレベル (ST レジスタ内の CPUPRI フィールド) 内の現在のプロセッサプライオリティレベルと、同じかより大きいプライオリティレベルを持っていて、かつプライオリティレベル未決定の IRQ が発生すると、割り込みがプロセッサに受け付けられます。

7.8 割り込みの有効化 / 無効化

それぞれの割り込みソースは、個別に有効化もしくは無効化できます。それぞれの IRQ に対して 1 つの割り込み有効ビットが、割り込み有効制御レジスタ（IEC*n*）に割り当てられています。割り込み有効ビットを 1 に設定すると、それに対応する割り込みが有効になり、割り込み有効ビットをゼロにクリアすると、それに対応する割り込みが無効になります。dsPIC デバイスがリセットから回復した立ち上がった時は、すべての割り込み有効ビットはゼロクリアされています。さらに、プロセッサは、命令サイクルの指定された番号でのすべての割り込みを無効にする割り込み無効命令（`DISI`）を持っています。

注 :	アドレスエラートラップのようなトラップは、無効化できません。IRQs のみが無効化できます。

DISI 命令は、インラインアセンブリを通して、C プログラム内で使用できます。例えば、以下のようなインラインアセンブリ文は、ソースプログラム内で現れる位置で、指定された DISI 命令を発行します。

```
__asm__ volatile ("disi #16");
```

DISI をこのような方法で使用する場合の欠点は、C プログラマが、C コンパイラがどのように C ソースをマシン命令に変換するかについて、常に判っているわけではないということです。従って、DISI 命令のサイクル数を決定することが困難かもしれません。DISI 命令により、割り込みから保護すべきコードを囲むことで、この困難を回避することができます。最初の命令は、サイクル数を最大値に設定し、次の命令はサイクル数をゼロに設定します。例えば、以下のように記述します。

```
__asm__ volatile("disi #0x3FFF"); /* disable interrupts */
/* ... protected C code ... */
__asm__ volatile("disi #0x0000"); /* enable interrupts */
```

別のやり方は、DISCNT レジスタに直接書き込む方法で、ハードウエア的には DISI 命令と同じ効果を持ちますが、C プログラマがインラインアセンブリを使うことを避けることができるという利点を持ちます。これは、もしインラインアセンブリが関数内で使用されるとコンパイラが実行しない最適化があるので、こちらのほうがより望ましい方法です。つまり、上記に示されるシーケンスの代わりに以下のものが使用できます。

```
DISICNT = 0x3FFF; /* disable interrupts */
/* ... protected C code ... */
DISICNT = 0x0000; /* enable interrupts */
```

第 8 章 アセンブリ言語と C モジュールの混用

8.1 序章

本章ではアセンブリ言語と C モジュールを一緒に使う方法について説明します。C 変数と関数をアセンブリコード内で使用する例と、アセンブリ言語の変数と関数を C 内で使用する例について示します。

8.2 ハイライト

この章では以下の項目について説明します。

- **アセンブリ言語と C モジュールの混用**－個別のアセンブリ言語モジュールがアセンブルされ、それから C モジュールとリンクされます。
- **インラインアセンブリ言語の使用**－アセンブリ言語命令が C コードに直接組み込まれます。インラインアセンブラは、単純（パラメータ無し）アセンブリ言語文と拡張（パラメータ有り）アセンブリ言語文の両方をサポートします。ここで C 変数は、アセンブラ命令のオペランドとしてアクセスできます。

8.3 アセンブリ言語と C 変数と関数の混用

以下のガイドラインは、個別のアセンブリ言語モジュールと C モジュールとのインターフェースを取る方法を示します。

- **セクション 4.13「レジスタの集合」**で述べられているレジスタ集合に従います。特に、レジスタ W0-W7 はパラメータの受け渡しに使用されます。アセンブリ言語関数は、これらのレジスタで、パラメータを受け取り、コールされた関数への引数の引渡しを行います。
- 割り込みハンドリング中にコールされない関数はレジスタ W8-W15 を保持しなければなりません。すなわち、レジスタ中の値は修正前に待避され、コールする関数に戻る前に復帰されねばなりません。レジスタ W0-W7 はそれらの値を保持することなく使用できます。
- 割り込み関数はすべてのレジスタを保持しなければなりません。通常の関数コールとは異なり、割り込みは、プログラムの実行中のどの位置でも発生します。通常のプログラムに戻るときは、すべてのレジスタは割り込みが発生する前の状態でなければなりません。
- C ソースファイルでも参照される個別のアセンブリファイル内で宣言される変数もしくは関数は、アセンブラ指定子 `.global` でグローバル宣言されねばなりません。外部シンボルは少なくとも 1 つのアンダースコアが前になければなりません。C 関数の `main` は、アセンブリの中では、`_main` と名づけられ、逆にアセンブリシンボル `_do_something` は、C 内では `do_something` として参照されます。アセンブリファイルの中で使用される未宣言シンボルは、外部定義されたものとして扱われます。

以下の例では、それらが元々どこで定義されたかにかかわらず、アセンブリ言語と C の両方の中で使用される変数と関数の使用方法について示します。

`ex1.c` ファイルは `foo` と `cVariable` を定義し、アセンブリ言語ファイル内で使用されます。C ファイルは、アセンブリ関数 `asmFunction` をコールする方法と、アセンブリ定義された変数 `asmVariable` のアクセスの方法を示します。

例 8-1: C とアセンブリの混用

```
/*
** file: ex1.c
*/
extern unsigned int asmVariable;
extern void asmFunction(void);
unsigned int cVariable;
void foo(void)
{
  asmFunction();
  asmVariable = 0x1234;
}
```

ex2.s ファイルは、asmFunction と asmVariable を定義し、リンクされたアプリケーション内で使用できます。アセンブリファイルはまた、C 関数 foo のコール方法と、C で定義された変数 cVariable のアクセス方法を示します。

```
;
;  file: ex2.s
;
  .text
  .global _asmFunction
_asmFunction:
  mov #0,w0
  mov w0,_cVariable
  return

  .global _begin
_main:
  call _foo
  return

  .bss
  .global _asmVariable
  .align 2
_asmVariable: .space 2
 .end
```

C ファイル ex1.c では、アセンブリファイル内で宣言されたシンボルに対する外部参照は、標準 extern キーワードを用いて宣言できます。アセンブリソース内の _asmFunction, つまり asmFunction は、void の関数でありそのように宣言されています。

アセンブリファイル ex1.s 内では、シンボル _asmFunction, _main と _asmVariable は、 .global アセンブラ指定子でグローバルに参照できるようになっており、他のどのソースファイルからもアクセスできます。シンボル _main は参照されるのみで、宣言されません。従って、アセンブラはこれを外部リファレンスと設定します。

以下の MPLAB C30 の例では、二つのパラメータでアセンブリ関数をコールする方法を示しています。 call1.c 内の C 関数 main は call2.s 内の asmFunction を、二つのパラメータを用いてコールします。

例 8-2:　　C 内のアセンブリ関数のコール

```
/*
** file: call1.c
*/
extern int asmFunction(int, int);
int x;
void
main(void)
{
 x = asmFunction(0x100, 0x200);
}
```

アセンブリ言語関数は二つのパラメータを加算し、結果を戻します。

```
;
; file: call2.s
;
  .global _asmFunction
_asmFunction:
  add w0,w1,w0
  return
 .end
```

C 内で受け渡しされるパラメータは**セクション 4.12.2「戻り値」**で詳細に説明されています。先の例では、二つの整数引数が W0 と W1 レジスタ内で受け渡しされます。整数の戻り結果はレジスタ W0 経由で渡されます。より複雑なパラメータリストは別のレジスタ群を必要とし、ガイドラインに従って注意深く手書きアセンブリされねばなりません。

8.4　インラインアセンブリ言語を使用する

C 関数内で、コンパイラが生成するアセンブリ言語にアセンブリ言語コードを挿入するために、asm 文が用いられます。インラインアセンブリは二つの形式、単純と拡張、を持ちます。

単純形式では、アセンブラ命令は以下の形式を用いて記述されます。

```
asm ("instruction");
```

ここで、*instruction* は有効なアセンブリ言語構造です。もし、ANSI C プログラム形式でインラインアセンブリを記述する場合は、 asm ではなく、__asm__ と書きます。

注：	インラインアセンブリの単純な形式では、1 つの文字列のみが通過コンパイル可能です。

asm を用いた**拡張**アセンブラ命令では､命令のオペランドは C 表現を用いて指定されます。拡張形式では以下のように表されます。

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
                 [ : [ "constraint"(input-operand) [ , ... ] ]
                      [ "clobber" [ , ... ] ]
               ]
             ]);
```

アセンブラ命令 *template* と、それぞれのオペランドに *constraint* ストリングオペランドを追加することで指定しなければなりません。*template* は命令ニュモニックを指定し、オプションにはオペランド用を指定します。*constraint* ストリングはオペランド制約、例えば、オペランドはレジスタになければならない（通常の場合）とか、もしくはオペランドはニアの値でなければならない、等の制約を指定します。

以下の制約文字が MPLAB C30 でサポートされています。

表 8-1: MPLAB C30 でサポートされる制約文字

文字	制約
=	このオペランドはこの命令用の書き込み専用であることを意味します。以前の値は捨てられ、出力データに置き換えられます。
+	このオペランドは命令により読んだり書いたりされることを意味します。
&	このオペランドは先に書き込まれる オペランドであることを意味し、命令が入力オペランドを用いることで終了する前に修正されます。従って、このオペランドは、入力オペランドもしくはメモリアドレスの一部として使用されるレジスタには置かれません。
g	汎用レジスタではないレジスタを除く、どんなレジスタ、メモリもしくは直接代入整数オペランドが使用できます。
i	直接代入整数オペランド（固定値を持ったもの）が使用できます。これは、その値がアセンブル時にのみ知ることができるシンボリック定数を含みます。
r	汎用レジスタ内にあるものであれば、レジスタオペランドが使用できます。
0, 1, … , 9	指定されたオペランド番号に適合するオペランドが使用できます。もし、digit が同じ alternative 内の文字と一緒に使用されると、digit が最後に来ます。
T	二アもしくはファーデータのオペランドです。
U	二アデータのオペランドです。

例えば、dsPIC デバイスの swap 命令（コンパイラは通常では使わない）の使い方は以下の通りです。

```
asm ("swap %0" : "+r"(var));
```

ここで var はオペランドの C 表現で、入力と出力両方のオペランドです。オペランドはタイプ r の制約で、レジスタオペランドを示します。+r の + はオペランドが入力と出力両方のオペランドであることを示します。

それぞれのオペランドは、括弧内の C 表現によるオペランド制約ストリングにより記述されます。最初のコロンはアセンブラのテンプレートを最初の出力オペランドと分け、次のコロンは最初の入力を（もしあれば）最後の出力オペランドと分けます。コンマは複数ある出力オペランドと入力オペランドを分けます。

もし出力オペランドが無く、入力オペランドがある場合は、2 つの連続するコロンで、出力オペランドが無いことを明確にしなければなりません。コンパイラは、出力オペランド表現が L 値でなければならないことを要求します。入力オペランドは、L 値である必要はありません。コンパイラは、実行される命令用として適切なデータタイプをオペランドが持っているかどうかについてはチェックできません。コンパイラはアセンブラ命令テンプレートを分析せず、それが何を意味するか、もしくはそれが有効なアセンブラ入力であるかどうかは知りません。拡張 asm の特徴は、しばしば、マシン命令（コンパイラ自身が、存在することを知らない命令）用として利用されます。もし出力表現が直接アドレスされないもの（例えば、ビットフィールド）であるとすると、制約がレジスタに与えられねばなりません。その場合、MPLAB C30 は asm の出力としてレジスタを使用し、レジスタを出力にストアします。もし出力オペランドが書き込みのみであるとすると、MPLAB C30 は、命令実行の前にこれらのオペランド内にある値が無効となってしまうため、生成する必要が無いと仮定します。

特定のハードレジスタが付与された命令もあります。このような命令を記述するには、入力オペランドの後に 3 つ目のコロンを付け、付与されたハードレジスタの名前をその後に続けます（文字列がコンマで区切られている場合のみ）。以下は dsPIC デバイスの例です。

```
asm volatile ("mul.b %0"
: /* no outputs */
: "U" (nvar)
: "w2");
```

この場合、オペランド nvar は、ニアデータ空間内で宣言される文字変数で、“U”制約で指定されます。もしアセンブラ命令がフラグ（条件コード）レジスタを変更するならば、書き換えられるレジスタをリストアップするには “cc”を追加します。もしアセンブラ命令が、予測不可能な方法でメモリを修正するなら、書き換えられるレジスタのリストに“memory”を追加します。このことは、アセンブラ命令によりレジスタに置かれる値を MPLAB C30 が保持しないことになります。

複数のアセンブラ命令を、一緒に単一の asm テンプレートに置くには、newlines (\n と書かれます) で区切ります。入力オペランドと出力オペランドのアドレスは、出力レジスタのどれも使用しないことが保証されています。従って、出力レジスタは何度でも読み書き可能です。以下に、テンプレート内の複数命令の例を示します。これはサブルーチン _foo がレジスタ W0 と W1 内の引数を受け付けることを仮定しています。

```
asm ("mov %0,w0\nmov %1,W1\ncall _foo"
: /* no outputs */
: "g" (a), "g" (b)
: "W0", "W1");
```

この例では、制約ストリング“g”が汎用オペランドであることを示しています。出力オペランドが & 制約修正子をもたなければ、MPLAB C30 は、入力は出力が生成される前に使われてしまうという前提で、それを入力オペランドと同じレジスタに割り当てます。もしアセンブラコードが実際に 1 つ以上の命令から構成されるならば、この仮定は正しく無くなってしまいます。そのような場合は、入力オペランドとオーバーラップしないように出力オペランドのために & を使用します。例えば、以下の関数を考えてみましょう。

```
int
exprbad(int a, int b)
{
        int c;

        __asm__("add %1,%2,%0\n sl %0,%1,%0"
                : "=r"(c) : "r"(a), "r"(b));

        return(c);
}
```

この目的は、値 (a + b) << a を計算することです。しかし、計算された値は上書きされたかもしれないし、そうでないかもしれません。正しいコーディングはコンパイラに、オペランド c は、asm 命令が終了する前に、以下に示すような入力オペランドを用いて修正される、ということを通知します。

```
int
exprgood(int a, int b)
{
        int c;

        __asm__("add %1,%2,%0\n sl %0,%1,%0"
                : "=&r"(c) : "r"(a), "r"(b));

        return(c);
}
```

アセンブラ命令が読み書きオペランドを持っているか、もしくはいくつかのビットのみが変更されるようなオペランドを持っている場合は、論理的にその関数を二つの別々のオペランド、すなわち 1 つの入力オペランドと 1 つの書き込みのみの出力オペランド、に分けなければなりません。それらの間の繋がりは、命令実行時には同じ位置にあることが必要であるという制約により表現されます。両方のオペランドの表現方法として同じ C 記述もできますし、異なる記述もできます。例えば、以下の例では、bar を読出し専用ソースオペランドとし、また foo を読み書き用の対象オペランドとする add 命令が示されています。

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "0" (foo), "r" (bar));
```

オペランド 1 用の制約“0”は、オペランド 1 がオペランド 0 と同じ位置を占めねばならないことを意味しています。制約記号は、入力オペランド内でのみ使用が許され、出力オペランドを参照するものでなければなりません。この制約記号だけが、1 つのオペランドがもう 1 つのものと同じ位置にあることを保証できます。foo が両方のオペランドの値であるというだけでは、生成されたアセンブラコード内で同じアドレスになるということを保証するには十分ではありません。以下の例がその説明に役に立つでしょう。

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "r" (foo), "r" (bar));
```

種々の最適化もしくはリロードにより、オペランド 0 と 1 が異なるレジスタに存在することがあります。例えば、コンパイラはあるレジスタ内で foo の値のコピーを見つけ、それをオペランド 1 用に使用しますが、出力オペランド 0 を別のレジスタに生成してしまいます（それを後で foo's 自身のアドレスにコピーします）。

アセンブラコードテンプレート内で参照可能なシンボリック名を使用して入 / 出力オペランドを指定することもできます。これらシンボリック名は制約文字列前の角括弧内で指定され、オペランド番号が後に続く % 記号の代わりに %[name] を使用してアセンブラコードテンプレート内で参照可能です。名前つきオペランドを使用して前述の例を以下のようにコード化できます。

```
asm ("add %[foo],%[bar],%[foo]"
: [foo] "=r" (foo)
: "0" (foo), [bar] "r" (bar));
```

asm の後にキーワード volatile を記述することで、asm 命令が消去されたり、大きく移動されたり、もしくは結合されたりすることを防ぐことができます。例えば、以下のようにします。

```
#define disi(n) \
asm volatile ("disi #%0" \
: /* no outputs */ \
: "i" (n))
```

この場合、制約文字“i”は、disi 命令で要求される直接代入オペランドを示します。

別紙 A 実装時定義動作

A.1 はじめに

本章では、MPLAB C30 の実装時定義動作を取り扱います。ISO 規格では、C 言語に関してベンダーが言語の“実装時定義”関数の詳細を文章化することを義務付けています。

本章で取り扱う項目は以下の通りです :

- 変換
- 環境
- 識別子
- 文字
- 整数
- 浮動小数点
- 配列及びポインタ
- レジスタ
- 構造体、共用体、列挙及びビットフィールド
- 修飾子
- 宣言子
- ステートメント
- プリプロセッサ命令
- ライブラリ関数
- 信号
- ストリーム及びファイル
- tmpfile
- errno
- メモリー
- 異常終了
- 終了
- getenv
- システム
- strerror

A.2 変換

変換の実装時定義動作は ANSI C 規格のセクション G.3.1 で取り上げられています。

連続するスペース文字列は、改行文字以外は維持されますか？それともスペース文字 1 つで置換されますか？ (ISO 5.1.1.2)

空白文字 1 つで置換されます。

診断メッセージはどのように識別されますか？ (ISO 5.1.1.3)

診断メッセージは、ソースファイル名とメッセージに対応する行番号をコロン（':'）で区切って診断メーセージの前に付けることによって識別されます。

異なるクラスのメッセージがありますか？ (ISO 5.1.1.3)

はい。

もしある場合、それが何ですか？ (ISO 5.1.1.3)

出力ファイルの生成を抑制するエラーと、出力ファイルの生成を抑制しない警告です。

各メッセージクラスの変換戻り状態コードは何ですか？

エラーの戻り状態コードは 1、警告の戻り状態コードは 0 です。

診断のレベルを制御することは可能ですか？ (ISO 5.1.1.3)

はい。

可能な場合、どのような形式で制御されますか？ (ISO 5.1.1.3)

コンパイラーコマンドラインオプションを使用して警告メッセージの生成を要請または抑制することができます。

A.3 環境

環境の実装時定義動作は ANSI C 規格のセクション G.3.2 で取り上げられています。

独立プログラムではどのようなライブラリ関数が利用可能ですか？ (ISO 5.1.2.1)

標準 C ライブラリのすべての関数が利用可能です。ただし“ランタイムライブラリ”セクションで説明されているように、ある一定の関数が環境に合わせてカスタマイズされていることが条件となります。

独立プログラムにおけるプログラム終了処理を説明してください。(ISO 5.1.2.1)

関数 `main` が戻されるか関数 `exit` が呼び出された場合、HALT 命令が無限ループで実行されます。この動作はカスタマイズできます。

関数 `main` にパスされる引数 (パラメータ) を説明してください。(ISO 5.1.2.2.1)

`main` にはパラメータはパスされません。

以下のうち、有効な双方向デバイスはどれですか: (ISO 5.1.2.3)

非同期端末　いいえ

対応するディスプレイとキーボード　いいえ

プログラム間接続　いいえ

他にある場合、それは何ですか？　ありません

A.4 識別子

識別子の実装時定義動作は ANSI C 規格のセクション G.3.3 で取り上げられています。

外部結合なしの識別子では、31 以上何文字まで認識可能ですか？ (ISO 6.1.2)

すべての文字が認識可能です。

外部結合ありの識別子では、6 以上何文字まで認識可能ですか？ (ISO 6.1.2)

すべての文字が認識可能です。

外部結合ありの識別子では、大文字と小文字の違いは区別されますか？ (ISO 6.1.2)

はい。

A.5 文字

文字の実装時定義動作は ANSI C 規格のセクション G.3.4 で取り上げられています。

規格で明確に指定されていないソース文字及び実行文字を説明して下さい。(ISO 5.2.1)

ありません。

記載されている文字列に対応するエスケープ文字列の値は何ですか？ (ISO 5.2.2)

表 A-1: エスケープ文字列及び値

文字列	値
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

実行文字セットの 1 つの文字は何ビットですか？ (ISO 5.2.4.2)

8 ビットです。

(文字リテラルと文字列リテラルにおける) 実行文字セットのメンバに対するソース文字セットのメンバのマッピングは何ですか？ (ISO 6.1.3.4)

関数として識別されます。

単純文字のタイプは何ですか？ (ISO 6.2.1.1)

次のどちらでも可能です (ユーザ定義)。デフォルトは `signed char` です。コンパイラーコマンドラインオプションを使用して `unsigned char` をデフォルトにすることができます。

A.6 整数

整数の実装時定義動作は ANSI C 規格のセクション G.3.5 で取り上げられています。

以下の表には様々な型の整数の容量と範囲が記載されています: (ISO 6.1.2.5)

表 A-2: 整数型

記号	サイズ （ビット）	範囲
char	8	-128 … 127
signed char	8	-128 … 127
unsigned char	8	0 … 255
short	16	-32768 … 32767
signed short	16	-32768 … 32767
unsigned short	16	0 … 65535
int	16	-32768 … 32767
signed int	16	-32768 … 32767
unsigned int	16	0 … 65535
long	32	-2147483648 … 2147438647
signed long	32	-2147483648 … 2147438647
unsigned long	32	0 … 4294867295

値が表示できない場合に整数をより短い符号付き整数に変換するとどうなりますか？また、符号なし整数を同じ長さの符号付に変換するとどうなりますか？ (ISO 6.2.1.2)

符号が失われます。エラー信号は出されません。

符号付き整数に対するビット単位の演算の結果はどうなりますか？ (ISO 6.3)

シフト演算子は符号を維持します。その他の演算は符号なしとして扱います。

整数除算の余りの符号は何ですか？ (ISO 6.3.5)

+ です。

負の値の符号付き整数型を右に移動するとどうなりますか？ (ISO 6.3.7)

符号は維持されます。

A.7 浮動小数点

浮動小数点の実装時定義動作は ANSI C 規格のセクション G.3.6 で取り上げられています。

その型の表示可能な値の範囲内にある浮動小数点定数の縮尺後の値は表示可能な近似値ですか？それとも表示可能なより大きな値ですか？それとも表示可能な値のより小さな値ですか？ (ISO 6.1.3.1)

表示可能な近似値です。

以下の表には様々な型の浮動小数点数の容量と範囲が記載されています: (ISO 6.1.2.5)

表 A-3: 浮動小数点型

記号	サイズ（ビット）	範囲
float	32	1.175494e-38 … 3.40282346e+38
double*	32	1.175494e-38 … 3.40282346e+38
long double	64	2.22507385e-308 … 1.79769313e+308

* double は -fno-short-double が使用されている場合は long double と同等です。

整数を元の値を正確に表示することができない浮動小数点数に変換する場合の打ち切りの方向はどうなりますか？ (ISO 6.2.1.3)

切り捨てです。

浮動小数点数をより小さな浮動小数点数に変換する場合の切り捨てあるいは切り上げの方向はどうなりますか？ (ISO 6.2.1.4)

切り捨てです。

A.8 配列およびポインタ

配列及びポインタの実装時定義動作は ANSI C 規格のセクション G.3.7 で取り上げられています。

配列の最大容量、つまり演算子のサイズの型 size_t を維持するのに必要な整数の型は何ですか？ (ISO 6.3.3.4, ISO 7.1.1)

unsigned int.

ポインタを整数型に変換するのに必要な整数のサイズはいくらですか？ (ISO 6.3.4)

16 ビットです。

ポインタを整数にキャスティングする、あるいは整数をポインタにキャスティングするとどうなりますか？ (ISO 6.3.4)

マッピングが識別関数です。

同じ配列の２つのポインタの差 ptrdiff_t を維持するのに必要な整数の型は何ですか？ (ISO 6.3.6, ISO 7.1.1)

unsigned int.

A.9 レジスタ

レジスタの実装時定義動作は ANSI C 規格のセクション G.3.8 で取り上げられています。

ストレージクラス指定子レジスタはどの程度まで実際にレジスタの中のオブジェクトに影響を与えますか？ (ISO 6.5.1)

最適化が無効になっている場合 register ストレージクラスを有効とみなす試みがなされますが、最適化が有効になっている場合は無視されます。

A.10 構造体、共用体、列挙及びビットフィールド

構造体、共用体、列挙及びビットフィールドの実装時定義動作は ANSI C 規格のセクション A.6.3.9 及び G.3.9 で取り上げられています。

共用体オブジェクトのメンバに異なるタイプのメンバを使用してアクセスするとどうなりますか？ (ISO 6.3.2.3)

型変換は適用されません。

構造体のメンバのパディングと整列を説明してください。(ISO 6.5.2.1)

文字はバイトごとに整列しています。その他のすべてのオブジェクトは word ごとに整列しています。

単純 `int` ビットフィールドの型は何ですか？ (ISO 6.5.2.1)

ユーザ定義です。デフォルトでは、`signed int` ビットフィールドです、コマンドラインオプションを使用して `unsigned int` ビットフィールドにすることが可能です。

int 内のビットフィールドの割り当ての順序はどうなっていますか？ (ISO 6.5.2.1)

ビットは低位ビットから高位ビットへ割り当てられます。

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可能ですか？ (ISO 6.5.2.1)

はい。

列挙型の値の表示にはどの整数型が選択されていますか？ (ISO 6.5.2.2)

`int`。

A.11 修飾子

修飾子の実装時定義動作は ANSI C 規格のセクション G.3.10 で取り上げられています。

どのようなアクションが、volatile 修飾子を有するオブジェクトへのアクセスとみなされますか？ (ISO 6.5.3)

オブジェクト名が表示されている場合、アクセスされています。

A.12 宣言子

宣言子の実装時定義動作は ANSI C 規格のセクション G.3.11 で取り上げられています。

演算子、構造体、あるいは共用体のタイプを変更する宣言子の最大数はいくらですか？ (ISO 6.5.4)

制限はありません。

A.13 ステートメント

ステートメントの実装時定義動作は ANSI C 規格のセクション G.3.12 で取り上げられています。

switch ステートメントの case 値の最大数はいくらですか？ (ISO 6.6.4.2)

制限はありません。

A.14 プリプロセッサ

プリプロセッサの実装時定義動作は ANSI C 規格のセクション G.3.13 で取り上げられています。

条件付き取り込みを制御する定数式における単一文字定数の値は、実行文字セットの中の同じ文字定数の値と合致しますか？ (ISO 6.8.1)

はい。

そのような文字定数は負の値を持つ場合もありますか？ (ISO 6.8.1)

はい。

インクルードソースファイルの位置を指定するにはどのような方法が使用されますか？ (ISO 6.8.2)

前処理プログラムがカレントディレクトリを検索し、次にコマンドラインオプションを使用して指定したディレクトリを検索します。

ヘッダはどのように識別されますか？また、場所はどのように指定されますか？ (ISO 6.8.2)

ヘッダは #include 指令で < と > の区切り文字か“と”の区切り文字の間に入れて識別されます。場所はコマンドラインオプションを使用して指定します。

組み込み可能なソースファイルでは引用名はサポートされていますか？ (ISO 6.8.2)

はい。

区切り文字列と外部ソースファイル名間のマッピングは何ですか？ (ISO 6.8.2)

識別関数です。

下記 #pragma 指令の動作を説明してください。(ISO 6.8.6)

表 A-4: #PRAGMA 動作

Pragma	Behavior
#pragma code section-name	コードセクション名をつける。
#pragma code	コードセクション名をデフォルト値にリセットする（viz., .text）。
#pragma idata section-name	初期化データセクション名をつける。
#pragma idata	初期化データセクション名をデフォルト値にリセットする（viz., .data）。
#pragma udata section-name	非初期化データセクション名をつける。
#pragma udata	非初期化データセクション名をデフォルト値にリセットする（viz., .bss）。
#pragma interrupt function-name	関数名を割り込み関数として指定する。

変換のデータと時間が利用可能でない場合の __ DATE __ と __ TIME __ のそれぞれの定義は何ですか？ (ISO 6.8.8)

該当ありません。これらの関数が利用不可の環境ではコンパイラーはサポートしていません。

A.15 ライブラリ関数

ライブラリ関数の実装時定義動作は ANSI C 規格のセクション G.3.14 で取り上げられています。

マクロ NULL が展開される空ポインタ定数は何ですか？ (ISO 7.1.5)

0 です。

± サート関数が表示する診断はどのように認識されますか？また、この関数の終了動作は何ですか？ (ISO 7.2)

アサート関数はファイル名、行番号及びテスト表現をコロン（':'）で区切って表示します。その後 abort 関数を呼び出します。

isalnum、isalpha、iscntrl、islower、isprint 及び isupper 関数でテスト済みの文字は何ですか？ (ISO 7.3.1)

表 A-5: 関数でテスト済みの文字

関数	テストされた文字
isalnum	いずれかの文字または数字 :isalpha 又は isdigit。
isalpha	いすれかの文字：islower 又は isupper。
iscntrl	標準動作制御文字 5 つと、バックスペース、アラートのうちのいずれか：\f, \n, \r, \t, \v, \b, \a。
islower	'a' から 'z' までのいずれかの文字。
isprint	図形文字又は空白文字：isalnum 又は ispunct 又はスペース。
isupper	'A' から 'Z' までのいずれかの文字。
ispunct	以下のいずれかの文字 : ! " # % & ' ( ) ; < = > ? [\] * + , - . / : ^

ドメインエラーの後に数学関数はどの値を返しますか？ (ISO 7.5.1)

NaN です。

数学関数はアンダーフロー範囲エラーのマクロ ERANGE に整数式 errno を設定しますか？ (ISO 7.5.1)

はい。

関数の第 2 引数が 0 の場合ドメインエラーが表示されますか、それとも 0 が返されますか？ (ISO 7.5.6.4)

ドメインエラーが表示されます。

A.16 信号

信号関数用の信号にはどのようなものがありますか？ (ISO 7.7.1.1)

表 A-6: 信号関数

名前	説明
SIGABRT	異常終了
SIGINT	双方向アテンション信号の受信
SIGILL	無効関数の検出
SIGFPE	算術演算の誤り
SIGSEGV	ストレージへの無効アクセス
SIGTERM	プログラムへの終了リクエストの送信

信号関数で認識される各信号のパラメータと使用法を説明してください。(ISO 7.7.1.1)

アプリケーションで定義されます。

信号関数で認識される各信号のデフォルト時の処理とプログラム起動時の処理を説明してください。(ISO 7.7.1.1)

ありません。

信号ハンドラの呼び出し前に signal (sig, SIGDFL); の等しいが実行されない場合、信号の遮断は実施されますか？ (ISO 7.7.1.1)

いいえ。

信号関数特定のハンドラが SIGILL 信号を受信するとデフォルト処理はリセットされますか？ (ISO 7.7.1.1)

いいえ。

A.17 ストリーム及びファイル

テキストストリームの最終行には終端改行文字が必要ですか？ (ISO 7.9.2)

いいえ。

テキストストリームの改行文字のすぐ前に記述された空白文字は、そのテキストストリームが逆読み込みされた場合に表示されますか？ (ISO 7.9.2)

はい。

2 進法ストリームで記述されたデータには空文字はいくつまで付加することができますか？ (ISO 7.9.2)

できません。

付加モードストリームのファイル位置インジケータは最初はファイルの始めか終わりのいずれに配置されますか？ (ISO 7.9.3)

始めです。

テキストススストリームに記述すると関連ファイルがその地点まででで切り捨てられますか？ (ISO 7.9.3)

アプリケーションで定義されます。

ファイルのバッファリングの特徴を説明してください。(ISO 7.9.3)

完全にバッファされます。

長さが 0 のファイルは実際に存在することは可能ですか？ (ISO 7.9.3)

はい。

有効なファイル名を作成するための規則はありますか？ (ISO 7.9.3)

アプリケーションで定義されます。

同じ名前のファイルを同時に開くことは可能ですか？ (ISO 7.9.3)

アプリケーションで定義されます。

開いているファイルに対して削除関数はどのような影響がありますか？ (ISO 7.9.4.1)

アプリケーションで定義されます。

ファイル名変更関数を呼び出す前に既に新しい名前のファイルが存在している場合はどうなりますか？ (ISO 7.9.4.2)

アプリケーションで定義されます。

`fprintf` 関数における `%p` 変換の出力はどのような形式になりますか？ (ISO 7.9.6.1)

16 進数表示です。

`fscanf` 関数における `%p` 変換の入力はどのような形式になりますか？ (ISO 7.9.6.2)

16 進数表示です。

A.18 TMPFILE

プログラムの異常終了時には開いている一時ファイルは削除されますか？ (ISO 7.9.4.3)

はい。

A.19 ERRNO

故障の場合 `fgetpos` 又は `ftell` 関数はマクロ `errno` をどの値に設定しますか？ (ISO 7.9.9.1, (ISO 7.9.9.4)

アプリケーションで定義されます。

`perror` 関数で生成されるメッセージのフォーマットは何ですか？ (ISO 7.9.10.4)

`perror` の引数、コロン、`errno` 値のテキスト記述です。

A.20 メモリー

要求されている容量が 0 の場合、`calloc`、`malloc` または `realloc` 関数の動作はどうなりますか？ (ISO 7.10.3)

ゼロの長さのブロックが割り当てられます。

A.21 異常終了

`abort` 関数が呼び出されると開いているファイルや一時ファイルはどうなりますか？ (ISO 7.10.4.1)

なにも起こりません。

A.22 終了

引数の値が 0 または EXIT_SUCCESS、または EXIT_FAILURE 以外の場合、終了関数によって返される状態は何ですか？ (ISO 7.10.4.3)

引数の値です。

A.23 GETENV

環境名にはどのような制限がありますか？ (ISO 7.10.4.4)

アプリケーションで定義されます。

getenv 関数の呼び出しによって取得された環境リストを変更するにはどのような方法が使用されますか？ (ISO 7.10.4.4)

アプリケーションで定義されます。

A.24 システム

システム関数にパスされる文字列のフォーマットを説明してください。(ISO 7.10.4.5)

アプリケーションで定義されます。

システム関数の実行形態はどのようになっていますか？ (ISO 7.10.4.5)

アプリケーションで定義されます。

A.25 STRERROR

strerror 関数によるエラーメッセージ出力のフォーマットを説明してください。(ISO 7.11.6.2)

単純文字列です。

strerror関数の呼び出しによって返されるエラーメッセージ文字列の内容を説明してください。(ISO 7.11.6.2)

表 A-7: エラーメッセージ文字列

Errno	メッセージ
0	エラーはありません
EDOM	ドメインエラーです
ERANGE	範囲エラーです
EFPOS	ファイル位置エラーです
EFOPEN	ファイルオープン時のエラーです
nnn	エラー #nnn

注意：

別紙 B MPLAB C30 C コンパイラー診断

B.1 はじめに

この別紙では、MPLAB C30 コンパイラーにより生成される最も一般的な診断メッセージを示します。

MPLAB C30 コンパイラーは、エラーと警告、2 種類の診断メッセージを生成できます。それぞれの種類で目的が異なります。

- エラーメッセージはプログラムのコンパイルを不可能にする問題を報告します。MPLABC30 は明らかに問題のあるソースファイルの名前とその行番号を報告します。
- *警告*メッセージは、コンパイルが可能、または実際にコンパイルが処理されている場合に、コード内のその他問題を示唆するような通常とは異なる状態を報告します。警告メッセージもソースファイルの名前と行番号を報告しますが、エラーメッセージとの区別のため、‘warning:’という文字を含みます。

警告メッセージが出るということは、プログラムが本当に本来の動作をしているか、古い機能や MPLAB C30 C の標準機能でない機能が使用をされていないかを確認する必要がある危険ポイントに達していることを示している場合があります。警告メッセージの多くはユーザーが -W オプションを使用してリクエストした時のみ発行されます（例えば、-Wall は様々な有効な警告をリクエストします)。

稀に、コンパイラーが内部エラーメッセージレポートを発行する場合があります。これは、コンパイラー自体がマイクロチップサポートに報告すべき障害を検出したということを意味します。サポートの連絡先の詳細はこの文書に記載されています。

B.2 エラー

記号

\x used with no following HEX digits － \x の後ろに 16 進数字がありません

エスケープシーケンス \x の後には 16 進数字を付けます。

‘&’ constraint used with no register class － ‘&' 制約にレジスタクラスが付いていません

asm ステートメントが無効です。

‘%’ constraint used with last operand － ‘%' 制約に最終オペランドが付いてます

asm ステートメントが無効です。

#elif after #else － #else の後に #elif があります

プリプロセッサ条件式では、#else 節を必ず #elif 節の後に続けます。

#elif without #if － #if が #elif に付いていません

プリプロセッサ条件下では、#elif 節の前に必ず #if 節を付けます。

#else after #else － #else の後に #else があります

プリプロセッサ条件下では、#else 節は 1 度しか使いません。

#else without #if － #if が #else に付いていません

プリプロセッサ条件下では、#else 節の前に必ず #if 節を付けます。

#endif without #if － #endif が #if に付いていません
プリプロセッサ条件下では、#endif 節の前に必ず #if 節を付けます。
#error ‘*message*’ － #error ‘*メッセージ*’
このエラーは #error ディレクティブに反応して表示されます。
#if with no expression －表現式を持たない #if です
定数演算子の値を求めるための表現式が必要です。
#include expects ‘FILENAME’or <FILENAME> － #include は“FILENAME”または <FILENAME> を求めています
#include のファイル名が見つからないか不完全です。引用符か括弧 <> 内にファイル名を入れてください。
‘#’is not followed by a macro parameter －‘#’の後にマクロパラメータが付いていません
ストリングサイズ演算子では‘#' のあとにマクロ引数名をつける必要があります。
‘*#keyword*’expects ‘FILENAME’or <FILENAME> －‘*#keyword*’は“FILENAME”または <FILENAME> を求めています
指定の *#keyword* が引数として引用符内または括弧 <> 内に入れたファイル名を求めています。
‘#’is not followed by a macro parameter －‘#’の後にマクロパラメータが付いていません
演算子‘#’のあとにマクロ引数名を付ける必要があります。
‘##’cannot appear at either end of a macro expansion －‘##’をマクロ展開の最初または末尾に付けないでください
連結演算子‘##’はマクロ展開の最初または末尾に付けてはいけません。

A

a parameter list with an ellipsis can’t match an empty parameter name list declaration －省略符号付きのパラメータリストが空のパラメータ名リスト宣言と一致しません
関数の宣言と定義は同一にする必要があります。
‘*symbol*’after #line is not a positive integer － #line の後の“*symbol*”が正の整数ではありません
#line はソース行番号が正の値であることを求めています。
aggregate value used where a complex was expected －複素数値が必要なところで集合値が使用されています
複素数値が必要なところで集合値を使用しないでください。
aggregate value used where a float was expected －浮動小数点が必要なところで集合値が使用されています
浮動小数点が必要なところで集合値を使用しないでください。
aggregate value used where an integer was expected －整数が必要なところで集合値が使用されています
整数が必要なところで集合値を使用しないでください。
alias arg not a string －エイリアス引数が文字列ではありません
現在の識別子がエイリアスの場合のエイリアス属性の引数は文字列にする必要があります。
alignment may not be specified for ‘*identifier*’－‘*identifier*’の整列を指定しないでください
整列属性は変数でしか使用できません。
‘__alignof’applied to a bit-field －‘__alignof’がビットフィールドに適用されています
‘__alignof’演算子はビットフィールドに適用できません。
alternate interrupt vector is not a constant －代替割り込みベクトルが定数ではありません
割り込みベクトルの値は整定数である必要があります。
alternate interrupt vector number *n* is not valid －代替割り込みベクトルの数値 *n* が無効です
有効な割り込みベクトルの数値が必要です。

ambiguous abbreviation argument －曖昧な省略引数です
指定されたコマンドラインの省略語が曖昧です。
an argument type that has a default promotion can’t match an empty parameter name list declaration. －空のパラメータ名リスト宣言に一致しないデフォルトの拡張を持つ引数です。
関数の宣言と定義は同一にする必要があります。
args to be formatted is not ... －フォーマットする引数は ... ではありません
フォーマット属性の最初のチェックのためのインデックス引数が‘...’の内容で指定されていないパラメータを指定しています。
argument ‘*identifier*’ doesn’t match prototype －引数 ‘*identifier*’ がプロトタイプと一致しません
関数の引数のタイプは、その関数のプロトタイプと同一にする必要があります。
argument of‘asm’is not a constant string －‘asm’の引数が定数の文字列ではありません
‘asm’の引数は定数の文字列にする必要があります。
argument to‘-B’is missing －‘-B’の引数が見つかりません
ディレクトリ名が見つかりません。
argument to‘-l’is missing －‘-l’の引数が見つかりません
ライブラリ名が見つかりません。
argument to‘-specs’is missing －‘-spec’の引数が見つかりません
specs ファイル名が見つかりません。
argument to‘-specs=’is missing －‘-spec=’の引数が見つかりません
specs ファイル名が見つかりません。
argument to‘-x’is missing －‘-x’の引数が見つかりません
言語名が見つかりません。
argument to‘-Xlinker’is missing －‘-Xlinker’の引数が見つかりません
リンカーに渡された引数が見つかりません。
arithmetic on pointer to an incomplete type －ポインタの演算が不完全なタイプです
不完全なタイプのポインタの演算は許可されません。
array index in non-array initializer －配列インデックスは非配列イニシャライザです
非配列イニシャライザに配列インデックスを使用しないでください。
array size missing in‘*identifier*’－‘*identifier*’内の配列サイズが見つかりません
配列サイズが見つかりません。
array subscript is not an integer －配列サブスクリプトが整数ではありません
配列添字は整数にする必要があります。
‘asm’operand constraint incompatible with operand size －‘asm’オペランド制約がオペランドのサイズに適合しません
asm ステートメントが無効です。
‘asm’operand requires impossible reload －‘asm’オペランドが不可能なリロードを要求しています
asm ステートメントが無効です。
asm template is not a string constant － asm テンプレートが文字列定数ではありません
asm テンプレートは文字列定数にする必要があります。
assertion without predicate －アサーションに述語がありません
#assert または #unassert の後には述語を付ける必要があります。識別子を 1 つ付けてください。
‘*attribute*’attribute applies only to functions －‘attribute’属性は関数にのみ適用されます
‘*attribute*’属性は関数にのみ適用します。

B

bit-field ‘*identifier*’ has invalid type －ビットフィールド‘*identifier*’が無効なタイプです
ビットフィールドは列挙型か整数タイプにする必要があります。
bit-field ‘*identifier*’ width not an integer constant －ビットフィールド‘*identifier*’の幅が整定数ではありません
ビットフィールドの幅は整定数にする必要があります。
both long and short specified for ‘*identifier*’ －ロングとショートの両方が‘*identifier*’に指定されています
変数のタイプはロングとショートの両方であってはなりません。
both signed and unsigned specified for ‘*identifier*’ －符号付きと符号なしの両方が‘*identifier*’に指定されています
変数は符号付きと符号なしの両方であってはなりません。
braced-group within expression allowed only inside a function －表現式内の括弧で区切られたグループは関数内でのみ許可されます
表現式内の括弧で区切られたグループを関数以外で使用するのは不正です。
break statement not within loop or switch －ループまたはスイッチ内にないブレークステートメントです
ブレークステートメントはループまたはスイッチ内でのみ使用できます。
__builtin_longjmp second argument must be 1 － __builtin_longjmp の 2 番めの引数は 1 にする必要があります
__builtin_longjmp では 2 番目の引数を 1 にする必要があります。

C

called object is not a function －コールされたオブジェクトは関数ではありません
C 言語では関数のみコールできます。
cannot convert to a pointer type －ポインタータイプに変換できません
表現をポインタータイプに変換できません。
cannot put object with volatile field into register －揮発性フィールドを持つオブジェクトはレジスタに保存できません
揮発性フィールドを持つオブジェクトをレジスタに保存するのは不正です。
cannot reload integer constant operand in ‘asm’ －整定数オペランドを‘asm’にリロードできません
asm ステートメントが無効です。
cannot specify both near and far attributes －二ア属性とファー属性の両方は指定できません
二ア属性とファー属性は互いに排他的関係にあります。1 つの関数または変数に使用できるのはいずれか 1 つのみです。
cannot take address of bit-field ‘*identifier*’ －ビットフィールド‘*identifier*’のアドレスを取得できません
ビットフィールドのアドレス取得を試行するのは不正です。
can’t open ‘*file*’ for writing －書き込み用の‘*file*’が開けません
システムは指定した‘*file*’を開けません。考えられる理由としては、ファイルを開くディスク容量が不足している、ディレクトリが存在しない、または保存先ディレクトリの書き込み許可がないことが挙げられます。
can’t set ‘*attribute*’ attribute after definition －定義後に‘*attribute*’属性を設定できません
記号が定義されている場合、‘*attribute*’属性を使用する必要があります。
case label does not reduce to an integer constant － case ラベルを整定数に判定できません
case ラベルはコンパイル時には整定数である必要があります。
case label not within a switch statement － switch ステートメント内にない case ラベルです
case ラベルは switch ステートメント内に含める必要があります。
cast specifies array type －キャストが配列タイプを指定しています
キャストによる配列タイプの指定は許可されません。

cast specifies function type －キャストが関数タイプを指定しています
キャストによる関数タイプの指定は許可されません。
cast to union type from type not present in union －ユニオン内に存在しないタイプからユニオンタイプをキャストしています
ユニオンタイプをキャストする時はユニオン内に存在するタイプからキャストしてください。
char-array initialized from wide string － wide 文字列で初期化された char-array です
char-array は wide 文字列で初期化してはいけません。通常の文字列を使用してください。
***file: compiler* compiler not installed on this system － *file: compiler* コンパイラーはこのシステムにインストールされていません**
C コンパイラーのみが配布されており、その他の高水準言語はサポートされていません。
complex invalid for‘*identifier*’－複素数が ‘*identifier*’に対して無効です
整数タイプと浮動小数点タイプにのみ複素数修飾子が適用されます。
conflicting types for‘*identifier*’－‘*identifier*’に矛盾したタイプです
identifier に複数の矛盾した宣言が存在します。
continue statement not within loop －継続ステートメントがループ内にありません
継続ステートメントはループ内でのみ使用できます。
conversion to non-scalar type requested －非スカラータイプへの変換がリクエストされました
タイプ変換は、スカラー（集合ではない）タイプへの変換でなくてはいけません。

D

data type of‘*name*’isn’t suitable for a register －‘*name*’のデータタイプがレジスタに適してません
データタイプがリクエストされたレジスタに適していません。
declaration for parameter‘*identifier*’but no such parameter －パラメータ‘*identifier*’が宣言されましたが、そのようなパラメータは存在しません
パラメータリスト内のパラメータのみが宣言可能です。
declaration of‘*identifier*’as array of functions －関数の配列として‘*identifier*’が宣言されました
関数の配列を持つことは不正です。
declaration of‘*identifier*’as array of voids － void の配列として‘*identifier*’が宣言されました
void の配列を持つことは不正です。
‘*identifier*’declared as function returning a function －‘*identifier*’が関数を返す関数として宣言されました
関数は関数を返すべきではありません。
‘*identifier*’declared as function returning an array －‘*identifier*' が配列を返す関数として宣言されました
関数は配列を返すべきではありません。
decrement of pointer to unknown structure －ポインタが未知の構造体まで下がっています
ポインタを未知の構造体まで下げないでください。
‘default’label not within a switch statement － switch ステートメント内にない‘default' ラベルです
default の case ラベルは switch ステートメント内に含める必要があります。
‘*symbol*’defined both normally and as an alias －‘*symbol*’が通常とエイリアスの両方に定義されています
すでに定義されている‘symbol’は他の記号のエイリアスに使用できません。
‘defined’cannot be used as a macro name －‘defined’はマクロ名として使用できません
マクロ名を‘defined’とするべきではありません。
dereferencing pointer to incomplete type －不完全なタイプへのデリファレンスポインタです
デリファレンスポインタは不完全なタイプへのポインタにするべきではありません。
division by zero in #if － #if 内でゼロの除算が行われました
ゼロの除算は計算できません。

duplicate case value －重複したケース値です
ケース値は一意である必要があります。
duplicate label‘*identifier*’－重複したラベル‘*identifier*’です
ラベルはそのスコープ内で一意である必要があります。
duplicate macro parameter‘*symbol*’－重複したマクロパラメータ‘*symbol*’です
‘symbol’がパラメータリストで 2 回以上使用されています。
duplicate member‘*identifier*’－重複したメンバー‘*identifier*’です
構造体は重複したメンバーを持つべきではありません。
duplicate (or overlapping) case value －重複した（またはオーバーラップした）case 値です
case は重複した、またはオーバーラップした値であってはなりません。‘this is the first entry overlapping that value' というエラーメッセージが最初の重複値またはオーバーラップ値が発生した個所に表示されます。case の範囲は MPLAB C30 の ANSI 標準の拡張です。

E

elements of array‘*identifier*’have incomplete type －配列‘*identifier*’のエレメントが不完全なタイプを含んでいます
配列エレメントは完全なタイプを含む必要があります。
empty character constant －空の文字定数です
空の文字定数は不正です。
empty file name in‘#*keyword*’－‘#keyword’内に空のファイル名があります
指定した #keyword の引数として指定したファイル名が空です。
empty index range in initializer －イニシャライザ内に空のインデックス範囲があります
イニシャライザ内で空のインデックス範囲を使用しないでください。
empty scalar initializer －空のスカラーイニシャライザです
スカラーイニシャライザは空にしないでください。
enumerator value for‘*identifier*’not integer constant －‘*identifier*’の列挙型の値が整定数ではありません
列挙値は整定数にする必要があります。
error closing‘*file*’－‘*file*’クローズのエラーです
システムは指定した‘*file*’を閉じることができません。考えられる理由はファイルに書き込むためのディスク容量の不足か、ファイルサイズが大きすぎるかのどちらかです。
error writing to‘*file*’－‘*file*’書き込みエラーです
システムは指定した‘*file*’に書き込むことができません。考えられる理由はファイルに書き込むためのディスク容量の不足か、ファイルサイズが大きすぎるかのどちらかです。
excess elements in char array initializer － char 配列イニシャライザ内のエレメントが多すぎます
イニシャライザ値のステートメント以上の数のエレメントがリスト内にあります。
excess elements in struct initializer －構造体イニシャライザ内のエレメントが多すぎます
イニシャライザ構造体内で過剰な数のエレメントを使用しないでください。
expression statement has incomplete type －表現式ステートメントが不完全なタイプを含んでいます
表現式のタイプが不完全です。
extra brace group at end of initializer －イニシャライザの末尾に不要な中括弧グループがあります
イニシャライザの末尾に不要な中括弧グループを付けないでください。
extraneous argument to‘*option*’option －‘*option*’オプションとは無関係な引数です
指定したコマンドラインオプションにある引数が多すぎます。

F

‘*identifier*’fails to be a typedef or built in type －‘*identifier*’は typedef タイプまたは built in タイプになるのに失敗しました
データタイプは typedef または built- にする必要があります。
field‘*identifier*’declared as a function －フィールド‘*identifier*’が関数として宣言されました
フィールドは関数として宣言されるべきはありません。
field‘*identifier*’has incomplete type －フィールド‘*identifier*’が不完全なタイプを含んでいます
フィールドは完全なタイプを含む必要があります。
first argument to __builtin_choose_expr not a constant － __builtin_choose_expr への最初の引数が定数ではありません
最初の引数はコンパイル時に決定される定数表現でなくてはなりません。
flexible array member in otherwise empty struct －フレキシブル配列メンバーか、空の構造体です
フレキシブル配列メンバーは 1 つ以上の名前が付いたメンバーを持つ構造体の最後のエレメントである必要があります。
flexible array member in union －フレキシブル配列メンバーが ユニオン内にあります
フレキシブル配列メンバーはユニオン内では使用できません。
flexible array member not at end of struct －フレキシブル配列メンバーが構造体の末尾ではありません
フレキシブル配列メンバーは構造体の最後のエレメントである必要があります。
‘for’loop initial declaration used outside C99 mode －‘for’ループの初期宣言が C99 モード外で使用されています
‘for’ループの初期宣言は C99 モード外では無効です。
format string arg follows the args to be formatted －フォーマット文字列引数がフォーマットされる引数の後に付いています
フォーマット属性への引数が矛盾しています。フォーマット文字列引数のインデックスは最初のチェックのための引数のインデックスより短くする必要があります。
format string arg not a string type －フォーマット文字列引数が文字列タイプではありません
フォーマット属性のフォーマット文字列インデックス引数が文字列タイプではないパラメータを指定しています。
format string has invalid operand number －フォーマット文字列は無効なオペランド数を含んでいます
フォーマット属性のオペランド番号引数はコンパイル時には定数でなければなりません。
function definition declared‘register’－‘register’として宣言された関数定義です
関数定義で‘register' 宣言してはなりません。
function definition declared‘typedef’－‘typedef’として宣言された関数定義です
関数定義で‘typedef’としてはなりません。
function does not return string type －関数が文字列タイプを返しません
format_arg 属性は戻り値が文字列タイプの関数でのみ使用できます。
function‘*identifier*’is initialized like a variable －関数‘*identifier*’が変数のように初期化されています
関数を変数のように初期化することは不正です。
function return type cannot be function －関数戻りタイプは関数にはなりません
関数の戻りタイプは関数にはなりません。

G

global register variable follows a function definition －グローバルレジスター変数が関数定義の後に付いています
グローバルレジスター変数は関数定義の前に付きます。
global register variable has initial value －グローバルレジスター変数は初期値を含んでいます
グローバルレジスター変数に初期値を指定しないでください。
global register variable‘*identifier*’used in nested function －グローバルレジスター変数‘*identifier*’がネストされた関数に使用されています
ネストされた関数内でグローバルレジスター変数を使用しないでください。

H

‘*identifier*’has an incomplete type －‘*identifier*’が不完全なタイプを含んでいます
指定した‘*identifier*’に不正なタイプを使用するのは不正です。
‘*identifier*’has both‘extern’and initializer －‘*identifier*’が‘extern’とイニシャライザの両方を含んでいます
‘extern’として宣言された変数は初期化できません
hexadecimal floating constants require an exponent － 16 進法浮動少数点定数は指数を求めています
16 進法浮動少数点定数は指数を含む必要があります。

I

implicit declaration of function‘*identifier*’－関数‘*identifier*’の暗示的宣言です
関数識別子は先行のプロトタイプ宣言または関数定義なしに使用できます。
impossible register constraint in‘asm’－‘asm’に含まれる不可能なレジスター制約です
asm ステートメントが無効です。
incompatible type for argument n of‘*identifier*’－‘*identifier*’の引数 n と互換性のないタイプです
C 言語内で関数をコールする際、実際の引数タイプが正式なパラメータタイプと一致していることを確認してください。
incompatible type for argument *n* of indirect function call －間接的関数コールの引数 *n* と互換性のないタイプです
C 言語内で関数をコールする際、実際の引数タイプが正式なパラメータタイプと一致していることを確認してください。
incompatible types in *operation* － *operation* 内に互換性のないタイプがあります
operation 内で使用されるタイプは互換性のあるものでなくてはなりません。
incomplete‘*name*’option －不完全な‘*name*’オプションです
コマンドラインパラメータ *name* のオプションが不完全です。
inconsistent operand constraints in an‘asm’－‘asm’に含まれる矛盾したオペランド制約です
asm ステートメントが無効です。
increment of pointer to unknown structure －ポインタが未知の構造体まで上がっています
ポインタを未知の構造体まで上げないでください。
initializer element is not computable at load time －イニシャライザエレメントがロード時に計算できません
イニシャライザエレメントはロード時に計算可能である必要があります。
initializer element is not constant －イニシャライザエレメントが定数ではありません
イニシャライザエレメントは定数にする必要があります。
initializer fails to determine size of‘*identifier*’－イニシャライザが‘*identifier*’のサイズ計測に失敗しました
配列イニシャライザがサイズ計測に失敗しました。

initializer for static variable is not constant －静的変数のイニシャライザが定数ではありません
静的変数イニシャライザは定数にする必要があります。
initializer for static variable uses complicated arithmetic －静的変数のイニシャライザが複合演算を使用しています
静的変数イニシャライザは複合演算を使用してはなりません。
input operand constraint contains‘*constraint*’－‘*constraint*’を含むオペランド制約を入力してください
指定した定数は入力オペランドでは無効です。
int-array initialized from non-wide string － wide 文字列以外の文字列で初期化された int-array です
Int-array は wide 文字列以外の文字列で初期化してはいけません。
interrupt functions must not take parameters －割り込み関数はパラメータをとるべきではありません
割り込み関数はパラメータを受け取れません。引数リストが空であることを明示的に宣言するため void を使用する必要があります。
interrupt functions must return void －割り込み関数は void を返すべきです
割り込み関数は *void* の戻りタイプを持つ必要があります。他のタイプは許可されません。
interrupt modifier‘*name*’unknown －割り込み修飾子‘*name*’が不明です
コンパイラーは‘*irq*’、‘*altirq*’または‘*save*’を割り込み属性修飾子として求めています。
interrupt modifier syntax error －割り込み修飾子構文エラー
割り込み属性修飾子に構文エラーがあります。
interrupt pragma must have file scope －割り込み pragma はファイル範囲を持つ必要があります
#pragma はファイル範囲を持つ必要があります。
interrupt save modifier syntax error －割り込み保存修飾子構文エラー
割り込み属性の‘save’修飾子に構文エラーがあります。
interrupt vector is not a constant －割り込みベクターが定数ではありません
割り込みベクトルの値は整定数である必要があります。
interrupt vector number *n* is not valid －割り込みベクターの数値 *n* が無効です
有効な割り込みベクトルの数値が必要です。
invalid #ident directive －無効な #ident ディレクティブ
#ident の後に引用符付きの文字リテラルを付ける必要があります。
invalid arg to‘__builtin_frame_address’－‘__builtin_frame_address’への無効な引数
引数は関数のコーラーのレベル（0 が現在の関数のフレームアドレスを出力すると、1 が現在の関数のコーラーのフレームアドレスを出力、後続の数字もこれに続く）で整数リテラルである必要があります。
invalid arg to‘__builtin_return_address’－‘__builtin_return_address’への無効な引数
レベル引数は整数リテラルである必要があります。
invalid argument for‘*name*’－‘*name*’への無効な引数
コンパイラーは‘data’または‘prog’を空間属性パラメータとして求めています。
invalid character‘*character*’in #if － #if 内に無効な文字‘*character*’があります
このメッセージは制御文字などの印刷されない文字が #if の後に付いた場合に表示されます。
invalid initial value for member‘*name*’－メンバー‘*name*’の初期値が無効です
ビットフィールド‘name’は整数でのみ初期化可能です。
invalid initializer －無効なイニシャライザ
無効なイニシャライザを使用しないでください。

Invalid location qualifier:‘*symbol*’－無効なローケーション修飾子：‘*symbol*’
ロケーション修飾子として、dsPIC では無視される‘*sfr*’または‘*gpf*’を求めています。
invalid operands to binary‘*operator*’－バイナリ‘*operator*’への無効なオペランド
そのバイナリ演算子へのオペランドは無効です。
Invalid option‘*option*’－無効なオプション‘*option*’
指定したコマンドラインオプションが無効です。
Invalid option‘*symbol*’to interrupt pragma －割り込み pragma に無効なオプション‘*symbol*’
割り込み pragma へのオプションとしてシャドウへ保存することを求めています。
Invalid option to interrupt pragma －割り込み pragma に無効なオプション
pragma の末尾の不要部分です。
Invalid or missing function name from interrupt pragma －割り込み pragma の無効か見つからない関数名です
割り込み pragma にはコールする関数名が必要です。
Invalid or missing section name －無効または見つからないセクション名
セクション名は必ず文字または下線 (‘_’) で始め、連続した文字列、下線および / または数字を後に付ける必要があります。‘*access*’、‘*shared*’、‘*overlay*’などの名前には特別な意味があります。
invalid preprocessing directive #‘*directive*’－無効なプリプロセッシングディレクティブ #‘*directive*’
有効なプリプロセッシングディレクティブではありません。スペルをチェックしてください。
invalid preprologue argument －無効なプリプローグ引数
プリプローグオプションがアセンブリステートメントまたは二重引用符でくくられた引数のステートメントを求めています。
invalid register name for‘*name*’－‘*name*’の無効なレジスタ名
ファイル範囲変数‘*name*’が不正なレジスタ名を持つレジスタ変数として宣言されました。
invalid register name‘*name*’for register variable －レジスタ変数の無効なレジスタ名‘*name*’
指定した名前はレジスタ名ではありません。
invalid save variable in interrupt pragma －割り込み pragma 内に無効な保存変数があります
保存には 1 つ以上の記号が必要です。
invalid storage class for function‘*identifier*’－関数‘*identifier*’の無効なストレージクラス
関数は‘register’ストレージクラスを持つわけではありません。
invalid suffix‘*suffix*’on integer constant －整定数の無効な接尾辞‘*suffix*’
整定数の接尾辞に使用できる文字は‘u’、‘U’、‘l’、‘L’のみです。
invalid suffix on floating constant －浮動小数点定数の無効な接尾辞です
浮動小数点定数に使用できる文字は、‘f’、‘F’、‘l’、‘L’のみです。‘L’が 2 つある場合、これらは隣接させ同じケースにする必要があります。
invalid type argument of‘*operator*’－‘*operator*’の無効なタイプの引数
operator への引数のタイプが無効です。
invalid type modifier within pointer declarator －ポインタ宣言子内に無効なタイプの変更子があります
ポインタ宣言子内でタイプ変更子として使用できるのは `const` か `volatile` のタイプのみです。
invalid use of array with unspecified bounds －指定されていない限界を持つ無効な配列が使用されています
指定されていない限界を持つ配列は有効な方法で使用してください。
invalid use of incomplete typedef‘*typedef*’－不完全なタイプ定義‘*typedef*’が使用されています
指定した *typedef* は無効な方法で使用されています。これは許可されません。

invalid use of undefined type‘*type identifier*’－未定義のタイプ‘*type identifier*’の無効な使用
指定した *type* は無効な方法で使用されています。これは許可されません。
invalid use of void expression － void 表現の無効な使用
void 表現は使用しないでください。
“*name*”is not a valid filename －“*name*”は無効なファイル名です
#line は有効なファイル名を求めています。
‘*filename*’is too large －‘*filename*’が大きすぎます
指定したファイルは処理するにはサイズが大きすぎます。恐らくサイズが 4 GB を超えるため、プリプロセッサがそのような大きなサイズのファイルを処理するのを拒否しています。ファイルサイズは 4GB 未満にして下さい。
ISO C forbids data definition with no type or storage class － ISO C ではタイプなしまたはストレージクラスなしのデータ定義を禁止しています
タイプ規則子またはストレージクラス規則子が ISO C 内でのデータ定義に必要です。
ISO C requires a named argument before‘...’－ ISO C は ‘...’の前に名前つき引数を付けることを求めています
ISO C は‘...’の前に名前つき引数を付けることを求めています。

L

label *label* referenced outside of any function －ラベル‘*label*’が関数の外側で参照されました
ラベルは関数内でのみ参照できます。
label‘*label*’used but not defined －ラベル‘*label*’が使用されましたが定義されていません
指定した *ラベル*は使用されましたが定義されていません。
language‘*name*’not recognized －言語‘*name*’は認識されません
C アセンブラを含む許される言語がありません。
***filename*: linker input file unused because linking not done －*ファイル名*：リンク付けがされていないためリンカー入力ファイルが使用されていません**
指定した *ファイル名*がコマンドラインで指定されましたが、リンカー入力ファイルとして認識されました（他のものとして認識されなかったため）。ただし、リンク付けは実行されていません。よって、このファイルは無視されました。
long long long is too long for GCC － long long long は GCC には大きすぎます
MPLAB C30 は `long long` より長い整数に対応していません。
long or short specified with char for‘*identifier*’－ long または short が‘*identifier*’の char で指定されました
long 修飾子および short 修飾子は char タイプと一緒に使用できません。
long or short specified with floating type for‘*identifier*’－ long または short が‘*identifier*’の浮動小数点タイプで指定されました
long 修飾子および short 修飾子は浮動小数点タイプと一緒に使用できません。
long, short, signed or unsigned invalid for‘*identifier*’－‘*identifier*’には、long、short signed、unsigned は無効です
long、short、singed 修飾子は整数タイプでしか使用できません。

M

macro names must be identifiers －マクロ名は識別子にしてください
マクロ名は必ず文字か下線で始め、その後にさらに文字、数字、下線などを付けます。
macro parameters must be comma-separated －マクロパラメータはコンマで区切る必要があります
パラメータリスト内では、パラメータの間にコンマが必要です。
macro‘*name*’passed *n* arguments, but takes just *n* －マクロ‘*name*’は引数を *n* 個渡しましたが、*n* 個しか受け取っていません
マクロ‘name’に渡された引数が多すぎます。

macro‘*name*’requires *n* arguments, but only *n* given －マクロ‘*name*’は *n* 個の引数を要求しましたが、*n* 個しか与えられていません

マクロ‘name’に渡された引数が少なすぎます。

matching constraint not valid in output operand －出力オペランド内では一致制約は無効です

asm ステートメントが無効です。

‘*symbol*’may not appear in macro parameter list －マクロパラメータリスト内に‘*symbol*’が含まれるべきではありません

‘symbol’はパラメータとして認められません。

Missing‘=’for‘save’in interrupt pragma －割り込み pragma 内で保存を示す - が間違っています

この保存パラメータはリストされている変数の前に等符号（=）を付けることを要求します。例えば、`#pragma interrupt isr0 save=var1,var2` のようになります。

missing‘(’after predicate －述語の後に‘(’が見つかりません

#assert または #unassert は ANSWER の前後に括弧を付けるよう要求します。例えば、以下のようになります。`ns#assert PREDICATE (ANSWER)`

missing‘(’in expression －表現内に‘(’が見つかりません

括弧付けが一致しません。始まりの括弧が必要です。

missing‘)’after“defined”－“defined”の後に‘)’が見つかりません

閉じ括弧が必要です。

missing‘)’in expression －表現内に‘)’が見つかりません

括弧付けが一致しません。閉じ括弧が必要です。

missing‘)’in macro parameter list －マクロパラメータリストに‘)’が見つかりません

マクロはカッコにくくられ、コンマで区切られたパラメータを要求しています。

missing‘)’to complete answer －解を完全にするための‘)’が見つかりません

#assert または #unassert は ANSWER の前後に括弧を付けるよう要求します。

missing argument to‘*option*’option －‘*option*’オプションへの引数が見つかりません

指定したコマンドラインオプションは引数を要求しています。

missing binary operator before token‘*token*’－トークン‘*token*’の前にバイナリ演算子が見つかりません

‘token’の前に演算子が必要です。

missing terminating‘*character*’character －終了文字‘*character*’が見つかりません

一重引用符 ' 、二重引用符 " または右山括弧 > などの終了文字が見つかりません。

missing terminating > character －終了文字 > が見つかりません

#include ディレクティブ内には終了文字 > が必要です。

more than *n* operands in‘asm’－‘asm’内に *n* 個以上のオペランドがあります

asm ステートメントが無効です。

multiple default labels in one switch － 1 つの switch 内に複数の default ラベルがあります

各 switch に指定できる default ラベルは 1 つだけです。

multiple parameters named‘*identifier*’－複数のパラメータに‘*identifier*’という名前が付いています

パラメータ名は一意である必要があります。

multiple storage classes in declaration of‘*identifier*’－‘*identifier*’宣言の中に複数のストレージクラスがあります

各宣言に含めることができるストレージクラスは 1 つだけです。

N

negative width in bit-field ‘*identifier*’ －ビットフィールド‘*identifier*’が負の幅です

ビットフィールド幅は負の値にしないでください。

nested function ‘name’ *declared* ‘extern’ －ネストされた関数‘name’が‘extern’宣言されました

ネストされた関数は‘extern’を宣言されるべきではありません。

nested redefinition of ‘*identifier*’ － ‘*identifier*’のネストされた再定義です

ネストされた再定義は不正です。

no data type for mode ‘mode’ －モード‘mode’のデータタイプがありません

モード属性に指定された引数モードは認識できる GCC マシンモードですが、MPLAB C30 では実装されていません。

no include path in which to find ‘*name*’ － ‘*name*’を見つけるためのインクルードパスがありません

インクルードファイル‘*name*’が見つかりません。

no macro name given in #‘*directive*’ directive － #‘*directive*’ディレクティブ内にマクロ名がありません

マクロ名の後には #define, #undef, #ifdef または #ifndef ディレクティブを付ける必要があります。

nonconstant array index in initializer －イニシャライザ内に非定数配列インデックスがあります

イニシャライザ内で使用できるのは定数配列インデックスのみです。

non-prototype definition here －非プロトタイプ定義があります

関数プロトタイプがプロトタイプなしの定義の後に付き、2 つの間で引数の数に矛盾が生じた場合、このメッセージは非プロトタイプ定義の行番号を示します。

number of arguments doesn’t match prototype －引数の数がプロトタイプと一致しません

関数の引数の数は、その関数のプロトタイプと同一にする必要があります。

O

operand constraint contains incorrectly positioned ‘+’ or ‘=’. －オペランド制約が誤った位置の‘+’または‘=’を含んでいます

asm ステートメントが無効です。

operand constraints for ‘asm’ differ in number of alternatives － ‘asm’のオペランド制約の数が他方の数と異なります

asm ステートメントが無効です。

operator “defined” requires an identifier －演算子“defined”には識別子が必要です

“defined”は識別子を求めています。

operator ‘*symbol*’ has no right operand 演算子‘*symbol*’に右のオペランドがありません

プリプロセッサ演算子‘symbol’の右にオペランドが必要です。

output number *n* not directly addressable －出力番号 *n* は直接アドレスできません

asm ステートメントが無効です。

output operand constraint lacks ‘=’ －出力オペランド制約に‘=’がありません

asm ステートメントが無効です。

output operand is constant in ‘asm’ 出力オペランドが‘asm’内で定数です

asm ステートメントが無効です。

overflow in enumeration values －列挙型値内のオーバーフローです

列挙型値は‘int’の範囲内でなくてはいけません。

P

parameter‘*identifier*’declared void －パラメータ‘*identifier*’は void を宣言しました
パラメータは void を宣言してはいけません。
parameter‘*identifier*’has incomplete type －パラメータ‘*identifier*’が不完全なタイプを含んでいます
パラメータは完全なタイプを含む必要があります。
parameter‘*identifier*’has just a forward declaration －パラメータ‘*identifier*’は前方宣言のみを含んでいます
パラメータは完全なタイプを含む必要があります。前方宣言は不完全です。
parameter‘*identifier*’is initialized －パラメータ‘*identifier*’が初期化されました
パラメータを初期化するのは不正です。
parameter name missing －パラメータ名が見つかりません
マクロはパラメータ名を要求しています。間に名前が入っていない 2 つのコンマがないか確認してください。
parameter name missing from parameter list －パラメータ名がパラメータリストから見つかりません
パラメータリスト内にはパラメータ名を含める必要があります。
parameter name omitted －パラメータ名が省かれています
パラメータ名は省くべきではありません。
param types given both in param list and separately －プログラムタイプがパラメータリスト内と個別に両方の形で含まれています
パラメータタイプはパラメータリストのみか個別に含まれるべきです。
parse error －構文解釈エラー
ソース行は構文解析できません。エラーを含んでいます。
pointer value used where a complex value was expected －複素数値が必要なところでポインタ値が使用されています
複素数値が必要なところでポインタ値を使用しないでください。
pointer value used where a floating point value was expected －浮動小数点値が必要なところでポインタ値が使用されています
浮動小数点が必要なところでポインタ値を使用しないでください。
pointers are not permitted as case values －ポインタは case 値として許可されません
case 値は整定数または定数表現でなくてはなりません。
predicate must be an identifier －述語は識別子でなくてはなりません
#assert または #unassert には述語として識別子が 1 つ必要です。
predicate’s answer is empty －述語の answer が空です
#assert または #unassert は述語と括弧を持っていますが、括弧内に必要な answer が入っていません。
previous declaration of‘*identifier*’－‘*identifier*’の以前の宣言です
このメッセージは以前の識別子の宣言の場所で識別されており、現在の宣言とは矛盾します。
***identifier* previously declared here － *identifier* が以前ここで宣言されました**
このメッセージは以前の識別子の宣言の場所で識別されており、現在の宣言とは矛盾します。
***identifier* previously defined here － *identifier* が以前ここで定義されました**
このメッセージは以前の識別子の定義の場所で識別されており、現在の定義とは矛盾します。
prototype declaration －プロトタイプ宣言
関数プロトタイプが宣言された場所で行番号が識別されました。他のエラーメッセージと関連付けて使用されます。

R

redeclaration of ‘*identifier*’ － ‘*identifier*’の再宣言
identifier が重複して宣言されています。
redeclaration of ‘*enum identifier*’ － ‘*enum identifier*’の再宣言
Enums は再宣言するべきではありません。
‘*identifier*’redeclared as different kind of symbol － ‘*identifier*’は異なる種類の記号として再宣言されました
identifier に複数の矛盾した宣言が存在します。
redefinition of ‘*identifier*’ － ‘*identifier*’の再定義
identifier が重複して定義されています。
redefinition of ‘*struct identifier*’ － ‘*struct identifier*’の再定義
Structs は再定義するべきではありません。
redefinition of ‘*union identifier*’ － ‘*union identifier*’の再定義
Unions は再定義するべきではありません。
register name given for non-register variable ‘*name*’ －非レジスタ変数‘*name*’にレジスタ名が付与されました
レジスタとしてマークされていない変数へレジスタのマップが試みられました。
register name not specified for ‘*name*’ － ‘*name*’にレジスタ名が指定されていません
ファイル範囲変数‘*name*’がレジスタの提供なしにレジスタ変数として宣言されました。
register specified for ‘*name*’ isn’t suitable for data type － ‘*name*’に指定されたレジスタがデータタイプに適してません
配列またはその他制約があるため、リクエストされたレジスタが使用できません。
request for member ‘*identifier*’ in something not a structure or union －構造体またはユニオンではない何かのメンバー '*identifier*' のリクエスト
メンバーを持っているのは構造体またはユニオンのみです。そのため、それ以外のメンバーを参照するのは不正です。
requested alignment is not a constant －リクエストされた配列が定数ではありません
aligned 属性の引数はコンパイル時には定数でなければなりません。
requested alignment is not a power of 2 －リクエストされた配列が 2 の乗数ではありません
aligned 属性の引数は 2 の乗数でなければなりません。
requested alignment is too large －リクエストした配列が大きすぎます
リクエストした配列サイズがリンカーが許可するサイズを超えています。サイズは 4096 以下の 2 の乗数にしてください。
return type is an incomplete type －戻りタイプが不完全なタイプです
戻りタイプは完全である必要があります。

S

save variable ‘*name*’ index not constant －保存変数‘*name*’のインデックスが定数ではありません
配列‘name’のサブスクリプトが整定数ではありません。
save variable ‘*name*’ is not word aligned －保存変数‘*name*’がワード配列になっていません
保存されるオブジェクトはワード配列でなくてはなりません。
save variable ‘*name*’ size is not even －保存変数‘*name*’のサイズが均等ではありません
保存されるオブジェクトのサイズは均等でなくてはなりません。
save variable ‘*name*’ size is not known －保存変数‘*name*’のサイズが不明です
保存されるオブジェクトのサイズは知られてなくてはなりません。

section attribute cannot be specified for local variables － section 属性はローカル変数には指定できません
ローカル変数は常にレジスタ内またはスタック上に割り当てます。よって、ローカル変数を名前付きセクションに配置しようと試みるのは不正です。
section attribute not allowed for *identifier* － *identifier* には section 属性は許可されていません
section 属性は関数または変数でしか使用できません。
section of *identifier* conflicts with previous declaration － *identifier* のセクションが以前の宣言と矛盾します
同じ*識別子*の複数の宣言が section 属性を指定している場合、属性の値は一貫している必要があります。
sfr address‘*address*’is not valid － sfr アドレス‘*address*’が有効ではありません
有効なアドレスは 0x2000 未満です。
sfr address is not a constant － sfr アドレスが定数ではありません
SFR アドレスは定数でなくてはなりません。
‘size of’applied to a bit-field －ビットフィールドに‘size of’が適用されています
‘sizeof’はビットフィールドに適用しないでください。
size of array‘*identifier*’has non-integer type －配列のサイズ‘*identifier*’は整数以外のタイプを含んでいます
配列サイズは整数タイプでなくてはなりません。
size of array‘*identifier*’is negative －‘*identifier*’のサイズが負の値です
配列サイズは負の値にすべきではありません。
size of array‘*identifier*’is too large －‘*identifier*’のサイズが大きすぎます
指定した配列が大きすぎます。
size of variable‘*variable*’is too large －変数‘*variable*’のサイズが大きすぎます
変数の最大サイズは 32768 バイトです。
storage class specified for parameter‘*identifier*’－ストレージクラスがパラメータ‘*identifier*’に指定されました
ストレージクラスはパラメータに指定すべきではありません。
storage size of‘*identifier*’isn’t constant －‘*identifier*’のストレージサイズが定数ではありません
ストレージサイズはコンパイル時には定数である必要があります。
storage size of‘*identifier*’isn’t known －‘*identifier*’のストレージサイズが不明です
identifier のサイズが不完全に指定されています。
stray‘*character*’in program －プログラム内の‘*character*’が逸脱しています
ソースプログラムに逸脱した‘*character*’文字を配置しないでください。
strftime formats cannot format arguments － strftime フォーマットは引数で形成されません
archetype パラメータが属性の 3 番目のパラメータである strftime の場合に属性フォーマットを使用していると、最初のパラメータがフォーマット文字列に一致するよう指定され、値は 0 になります。strftime スタイルの関数はフォーマット文字列に一致する入力値を持ちません。
structure has no member named‘*identifier*’－構造は‘*identifier*’という名前のメンバーを持ちません
‘*identifier*’という名前の構造メンバーが参照されましたが、参照先の構造にそのようなメンバーは存在しません。そのような参照は許可されません。
subscripted value is neither array nor pointer －サブスクリプトされた値が配列、ポインタのどちらでもありません
配列またはポインタのみがサブスクリプトされるべきです。
switch quantity not an integer － switch の数量が整数ではありません
switch の数量は整数にする必要があります

symbol‘*symbol*’not defined －記号‘*symbol*’が定義されていません
記号‘symbol’はプラグマ内で使用される前に宣言される必要があります。
syntax error －構文エラー
指定した行に構文エラーがあります。
syntax error‘:’without preceding‘?’－構文エラー：前に‘?’が付かない‘:’
‘:’の前には‘?:’演算子内の‘?’を付ける必要があります。

T

the only valid combination is‘long double’－有効な組み合わせは‘long double’のみです
long 修飾子は double タイプと一緒に使用できる唯一の修飾子です。
this built-in requires a frame pointer －このビルトインにはフレームポインタが必要です
__builtin_return_address はフレームポインタが必要です。
-fomit-frame-pointer オプションは使用しないでください。
this is a previous declaration －以前の宣言です
ラベルが重複している場合、このメッセージは以前の宣言の行数を知らせます。
too few arguments to function －関数への引数が少なすぎます
C 内で関数を呼び出す場合は、関数が要求するより少ない引数を指定しないでください。多く指定してもいけません。
too few arguments to function‘*identifier*’－関数‘*identifier*’への引数が少なすぎます
C 内で関数を呼び出す場合は、関数が要求するより少ない引数を指定しないでください。多く指定してもいけません。
too many alternatives in‘asm’－‘asm’内の選択肢が多すぎます
asm ステートメントが無効です。
too many arguments to function －関数への引数が多すぎます
C 内で関数を呼び出す場合は、関数が要求するより多い引数を指定しないでください。少なく指定してもいけません。
too many arguments to function‘*identifier*’－関数‘*identifier*’への引数が多すぎます
C 内で関数を呼び出す場合は、関数が要求するより多い引数を指定しないでください。少なく指定してもいけません。
too many decimal points in number －数字に 10 進小数点が多すぎます
必要な 10 進小数点は 1 つのみです。
top-level declaration of‘*identifier*’specifies‘auto’－‘*identifier*’の最上位宣言は‘auto’を指定しました
オート変数は関数内でのみ宣言可能です。
two or more data types in declaration of‘*identifier*’－‘*identifier*’の宣言に 2 つ以上のデータタイプがあります
各識別子は 1 つのデータタイプしか持てません。
two types specified in one empty declaration － 1 つの空の宣言内に 2 つのタイプが指定されています
1 つ以上のタイプを指定しないでください。
type of formal parameter *n* is incomplete －仮パラメータのタイプ *n* が不完全です
表示されたパラメータには完全なタイプを指定してください。
type mismatch in conditional expression －条件式内のタイプが一致しません
条件式内のタイプは不一致であってはいけません。
typedef‘*identifier*’is initialized － typedef‘*identifier*’が初期化されました
typedef's を初期化するのは不正です。代わりに __typeof__ を使用してください。

U

‘*identifier*’undeclared (first use in this function) －‘*identifier*’が宣言されていません（この関数内では初めて使用）
指定された識別子を宣言する必要があります。
‘*identifier*’undeclared here (not in a function) －ここでは‘*identifier*’が宣言されていません（関数内以外）
指定された識別子を宣言する必要があります。
union has no member named‘*identifier*’－ユニオンは‘*identifier*’という名前のメンバーを持っていません
‘*identifier*’という名前のユニオンメンバーが参照されましたが、参照先のユニオンにそのようなメンバーは存在しません。そのような参照は許可されません。
unknown field‘*identifier*’specified in initializer －イニシャライザ内で未知のフィールド‘*identifier*’が指定されました
イニシャライザ内で未知のフィールドを使用しないでください。
unknown machine mode‘*mode*’－未知のマシンモード‘*mode*’
モード属性に指定された引数 *mode* は認識可能なマシンモードではありません。
unknown register name‘*name*’in‘asm’－‘asm’内に未知のレジスタ名があります
asm ステートメントが無効です。
unrecognized format specifier －認識されないフォーマット規則子です
フォーマット属性への引数が無効です。
unrecognized option‘*-option*’－認識されないオプション‘*-option*’
指定したコマンドラインオプションが認識されません。
unrecognized option‘*option*’－認識されないオプション‘*option*’
'*option*' は未知のオプションです。
‘*identifier*’used prior to declaration －宣言の前に‘*identifier*’が使用されています
識別子はその宣言の前に使用します。
unterminated #‘*name*’－終端していない #‘*name*’
#endif は #if、#ifdef または #ifndef 条件で終端する必要があります。
unterminated argument list invoking macro‘*name*’－終端していない引数リストがマクロ‘*name*’を呼び出しています
関数マクロの評価で、マクロ拡張完了前にファイルの終わりが検出されました。
unterminated comment －終端していないコメント
コメントターミネ―タをスキャンしている間にファイルの終わりに達しました。

V

‘va_start’used in function with fixed args －固定引数を持つ関数内で‘va_start’が使用されました
変数引数リストを持つ関数内でのみ‘va_start’を使用できます。
variable‘*identifier*’has initializer but incomplete type －変数‘*identifier*’はイニシャライザを持っていますが、タイプが不完全です
不完全なタイプを持つ変数を初期化するのは不正です。
variable or field‘*identifier*’declared void －変数またはフィールド‘*identifier*’は void を宣言しました
変数およびフィールドは void を宣言してはいけません。
variable-sized object may not be initialized －変数サイズのオブジェクトは初期化するべきではありません
変数サイズのオブジェクトを初期化することは不正です。
virtual memory exhausted －仮想メモリが不足しています
エラーメッセージを記述するのに必要なメモリがわずかしか残されていません。

void expression between ‘(’ and ‘)’ － ‘(’ and ‘)’の間に void 表現があります
定数表現が必要ですが、括弧の間に void 表現があります。
‘void’ in parameter list must be the entire list －パラメータリスト内の‘void’は完全なリストでなくてはいけません
‘void’パラメータリスト内のパラメータとして表示される場合は、他のパラメータが存在してはなりません。
void value not ignored as it ought to be － void の値が通常のように無視されませんでした
void 関数の値は表現式には使用できません。

W

warning: -pipe ignored because -save-temps specified －警告：-save-temps が指定されたため、-pipe が無視されました
-pipe オプションは -save-temps オプションと一緒に使用できません。
warning: -pipe ignored because -time specified －警告：-time が指定されたため、-pipe が無視されました
-pipe オプションは -time オプションと一緒に使用できません。
warning:‘-x *spec*’after last input file has no effect －警告：最後の入力ファイルの後の‘-x *spec*’は効果がありません
‘-x’コマンドラインオプションは、そのコマンドライン上で名前が付けられたファイルにのみ影響を及ぼします。そのようなファイルがない場合、このオプションには効果がありません。
weak declaration of‘*name*’must be public －‘*name*’の弱い宣言はパブリックにする必要があります
弱い記号は外部から見えるようにする必要があります。
weak declaration of‘*name*’must precede definition －‘*name*’の弱い宣言は定義より前に来る必要があります
‘*name*’は定義されてから弱い宣言がなされます。
wrong number of arguments specified for *attribute* attribute － *attribute* 属性に誤った引数の数字が指定されています
‘*attribute*’という名前の属性に与えられている引数が少なすぎるか多すぎます。
wrong type argument to bit-complement － bit-complement の引数のタイプが誤っています
この演算子に誤ったタイプの引数を使用しないでください。
wrong type argument to decrement －減分の引数のタイプが誤っています
この演算子に誤ったタイプの引数を使用しないでください。
wrong type argument to increment －増分の引数のタイプが間違っています
この演算子に誤ったタイプの引数を使用しないでください。
wrong type argument to unary exclamation mark －単項感嘆符の引数のタイプが誤っています
この演算子に誤ったタイプの引数を使用しないでください。
wrong type argument to unary minus －単項マイナスの引数のタイプが間違っています
この演算子に誤ったタイプの引数を使用しないでください。
wrong type argument to unary plus －単項プラスの引数のタイプが間違っています
この演算子に誤ったタイプの引数を使用しないでください。

Z

zero width for bit-field‘*identifier*’－ビットフィールド‘*identifier*’の幅がゼロです
ビットフィールドの幅はゼロにしないでください。

B.3 警告

記号

‘/*’ within comment －コメント内に‘/*’があります

コメントマークがコメント内で見つかりました。

‘$’ character(s) in identifier or number －識別子または数字内に‘$’が含まれています

識別子名内のドル記号は、標準の拡張子です。

#‘*directive*’ is a GCC extension － #‘*directive*’ は GCC 拡張子です

#warning、#include_next、#ident、#import、#assert、#unassert ディレクティブは GCC 拡張子であり、ISO C89 の拡張子ではありません。

#import is obsolete, use an #ifndef wrapper in the header file － #import は廃止されています。ヘッダーファイル内では #ifndef ラッパーを使用してください

#import ディレクティブは廃止されています。#import は、インクルードされていないファイルをインクルードするのに使用されていました。代わりに #ifndef ディレクティブを使用してください。

#include_next in primary source file －プライマリソースファイル内に #include_next があります

#include_next は現在のファイルが見つかったディレクトリの後にあるヘッダーファイルディレクティブのリストを検索します。この場合、それ以前にはヘッダーファイルはないため、プライマリソースファイルから検索が始まります。

#pragma pack (pop) encountered without matching #pragma pack (push, <n>) － #pragma pack (push, <n>) とのマッチなしに、#pragma pack (pop) が見つかりました

pack(pop)pragma は、ソースファイル内では後にある pack(push)pragma とペアになる必要があります。

#pragma pack (pop, *identifier*) encountered without matching #pragma pack (push, *identifier*, <n>) － #pragma pack (push, *identifier*, <n>) とのマッチなしに #pragma pack (pop, *identifier*) が見つかりました

pack(pop)pragma は、ソースファイル内では後にある pack(push)pragma とペアになる必要があります。

#warning: *message* － # 警告：*メッセージ*

ディレクティブ #warning はプリプロセッサに警告を発生させプリプロセスを続行させます。#warning の後に続くトークンは警告メッセージとして使用されます。

A

absolute address specification ignored －絶対アドレス仕様が無視されました

MPLAB C30 でサポートされていないため、#pragma ステートメント内のコードセクションの絶対アドレス仕様を無視します。アドレスはキーワード `__attribute__`で定義可能なリンカースクリプト内とコードセクション内で指定する必要があります。

address of register variable‘*name*’ requested －レジスタ変数‘*name*’のアドレス請求

レジスタ指定が変数のアドレスを取得するのを防ぎます。

alignment must be a small power of two, not n －配列は n ではなく、2 の小さな乗数にする必要があります

パック pragma の配列パラメータは 2 の乗数でなければなりません。

anonymous enum declared inside parameter list －アノニマス enum がパラメータリスト内で宣言されました

アノニマス enum が関数パラメータリスト内で宣言されました。通常のよりよいプログラミング慣習としては、パラメータリスト外で enum を宣言します。パラメータリスト内で定義される際、これらは完全なタイプとならないからです。

anonymous struct declared inside parameter list －アノニマス構造体がパラメータリスト内で宣言されました

アノニマスは構造体関数パラメータリスト内で宣言された。通常のよりよいプログラミング慣習としては、パラメータリスト外で構造体を宣言します。パラメータリスト内で定義される際、これらは完全なタイプとならないからです。

anonymous union declared inside parameter list －アノニマスユニオンがパラメータリスト内で宣言されました
アノニマスユニオンが関数パラメータリスト内で宣言された。通常のよりよいプログラミング慣習としては、パラメータリスト外でユニオンを宣言します。パラメータリスト内で定義される際、これらは完全なタイプとならないからです。
anonymous variadic macros were introduced in C99 －アノニマス variadic マクロが C99 に導入されました
引数の変数を受け入れるマクロは C99 の機能です。
argument‘*identifier*’might be clobbered by‘longjmp’or‘vfork’－引数‘*identifier*’は‘longjmp’または‘vfork’によって修飾される可能性があります
引数は longjmp へのコールにより変更される可能性があります。この警告はコンパイルの最適化時にのみ生成されます。
array‘*identifier*’assumed to have one element －配列‘*identifier*’はエレメントを 1 つ持つと仮定しました
配列の長さが明示的に指定されていません。情報がない場合、コンパイラーは配列は 1 つのエレメントを持つと仮定します。
array subscript has type‘char’－配列サブスクリプションはタイプ‘char’を持ちます
配列サブスクリプションはタイプ‘*char*’を持ちます。
array type has incomplete element type －配列タイプは不完全なエレメントタイプを持ちます
配列タイプは不完全なエレメントタイプを持つべきではありません。
asm operand *n* probably doesn’t match constraints － asm オペランド *n* は制約とマッチしない可能性があります
指定した拡張 asm オペランドは制約とマッチしない可能性があります。
assignment of read-only member‘*name*’－読み取り専用メンバー‘*name*’への代入です
メンバー‘name’が const として宣言され、代入では修正できません。
assignment of read-only variable‘*name*’－ 読み取り専用変数‘name' への代入です
‘name' が const として宣言され、代入では修正できません。
‘*identifier*’attribute directive ignored －‘*identifier*’属性ディレクティブが無視されました
名前の付いた属性が不明か、サポートされた属性ではないため無視されました。
‘*identifier*’attribute does not apply to types －‘*identifier*’属性はタイプに適用されません
名前が付いた属性はタイプでは使用できません。無視されます。
‘*identifier*’attribute ignored －‘*identifier*’属性が無視されました
名前の付いた属性が与えられた文脈では意味を成さないため無視されました。
‘*attribute*’attribute only applies to function types －‘*attribute*’属性は関数タイプにのみ適用されます
指定された属性は関数の戻りタイプにのみ適用され、他の宣言には適用されません。

B

backslash and newline separated by space －スペースで区切られたバックスラッシュと newline です
エスケープシーケンスの処理中に、スペースで区切られたバックスラッシュと newline が検出されました。
backslash-newline at end of file －ファイルの最後にバックスラッシュ -newline があります
エスケープシーケンスの処理中に、ファイルの最後にあるバックスラッシュと newline が検出されました。
bit-field‘*identifier*’type invalid in ISO C －ビットフィールド‘*identifier*’タイプは ISO C では無効です
指定した識別子で使用されているタイプは ISO C では有効ではありません。
braces around scalar initializer －スカラーイニシャライザの前後に中括弧があります
イニシャライザの前後に冗長な中括弧が付いています。

built-in function‘*identifier*’declared as non-function －ビルトイン関数‘*identifier*’が非関数として宣言されました
指定された関数はビルトイン関数と同じ名前を持っていますが、関数以外のものとして宣言されました。

C

C++ style comments are not allowed in ISO C89 － C++ スタイルコメントは ISO C89 では許可されていません
C スタイルコメント‘/*’と‘*/’を C++ スタイルコメント‘//’の代わりに使用してください。

call-clobbered register used for global register variable －グローバルレジスタ変数に call-clobbered が使用されています
関数コール (W8-W13) で普通に保存、格納できるレジスタを選択してください。ライブラリルーチンが修飾されずにすみます。

cannot inline function‘main’－関数‘main’をインライン化できません
関数‘main’が *inline* 属性と一緒に宣言されました。別にコンパイルされる C スタートアップコードからメインを呼び出さなくてはならないため、これはサポートされていません。

can’t inline call to‘*identifier*’called from here －ここからコールされた‘*identifier*’へのコールをインライン化できません
コンパイラーは指定された関数へのコールをインライン化できませんでした。

case value‘*n*’not in enumerated type －列挙型タイプにないケース値‘*n*’です
switch ステートメントの制御表現は列挙型ですが、case 表現が列挙型の値に対応しない値 n を持っています。

case value‘*value*’not in enumerated type‘*name*’－列挙型タイプ‘*name*’にない case 値‘*value*’です
‘*value*’は追加の switch case で列挙型タイプ‘*name*’のエレメントではありません。

cast does not match function type －ケースが関数タイプとマッチしません
関数の戻りタイプは関数タイプとマッチしないタイプにキャストされます。

cast from pointer to integer of different size －ポインタから異なるサイズの整数へのキャストです
16 ビット幅ではない整数にポインタがキャストされます。

cast increases required alignment of target type －キャストによりターゲットタイプの必要な配列が増加します
-Wcast-align でコマンドラインオプションをコンパイルする際、コンパイラーはキャストによりターゲットタイプに必要な配列が増加していないことを確認します。例えば、この警告メッセージは char が int へのポインタとしてキャストされると生成されます。char のための配列（バイト配列）が int が求める配列（ワード配列）より少ないためです。

character constant too long －文字定数が長すぎます
文字定数は長すぎてはなりません。

comma at end of enumerator list －列挙子リストの末尾にカンマがあります
列挙子リストの末尾にカンマは不要です。

comma operator in operand of #if － #if のオペランド内にカンマ演算子があります
#if ディレクティブにカンマ演算子は不要です。

comparing floating point with == or != is unsafe －浮動小数点と == または != の比較は安全ではありません
浮動小数点の値は非常に正確な実数の近似となります。同等性をテストする代わりに、比較演算子を使用して 2 つの値の範囲がオーバーラップしているかを見ます。

comparison between pointer and integer －ポインタと整数間の比較です
ポインタタイプは整数タイプと比較されます。

comparison between signed and unsigned －符号付きと符号無しの比較です

比較のオペランドのうち一方が符号付きで、もう一方が符号無しです。符号付きオペランドは符号無しの値として取り扱われ、修正されない可能性があります。

comparison is always *n* －比較は常に *n* です

比較は定数表現のみを含み、コンパイラーは比較のランタイム結果を評価できます。結果は常に *n* です。

comparison is always *n* due to width of bit-field －ビットフィールドの幅に従い、比較は常に *n* です

ビットフィールドを含む比較は常に *n* を評価します。これは、ビットフィールドの幅のためです。

comparison is always false due to limited range of data type －データタイプの制限範囲に従い、比較は常に false です

実行時の比較は常に false と評価されます。これはデータタイプの範囲のためです。

comparison is always true due to limited range of data type －データタイプの制限範囲に従い、比較は常に true です

実行時の比較は常に true と評価されます。これはデータタイプの範囲のためです。

comparison of promoted ~unsigned with constant － promoted~unsigned と定数の比較です

比較のオペランドのうち一方が promoted ~unsigned で、もう一方が定数です。

comparison of promoted ~unsigned with unsigned － promoted~unsigned と符号無しの比較です

比較のオペランドのうち一方が promoted ~unsigned で、もう一方が符号無しです。

comparison of unsigned expression >= 0 is always true －符号無し表現 >= 0 の比較は常に true です

比較表現は符号無しの値とゼロを比較します。符号無しの値は 0 未満にはならないため、実行時の比較は常に true と評価されます。

comparison of unsigned expression < 0 is always false －符号無し表現 < 0 の比較は常に false です

比較表現は符号無しの値とゼロを比較します。符号無しの値は 0 未満にはならないため、実行時の比較は常に false と評価されます。

comparisons like X<=Y<=Z do not have their mathematical meaning － X<=Y<=Z のような比較に数学的な意味はありません

C 表現は対応する数学的表現と常に同じ意味を持つ必要はありません。特に、C 表現 X<=Y<=Z は数学的表現 X £ Y £ Z と同等ではありません。

conflicting types for built-in function ‘*identifier*’ －ビルトイン関数 ‘*identifier*’ と競合するタイプです

指定された関数はビルトイン関数と同じ名前を持っていますが、競合するタイプとともに宣言されました。

const declaration for ‘*identifier*’ follows non-const － ‘*identifier*’ の non-const 宣言の後には const が付きます

指定した識別子は non-const として宣言された後に const として宣言されました。

control reaches end of non-void function －制御が non-void 関数の末尾に達しました

non-void 関数からのすべての exit パスは適切な値を返す必要があります。コンパイラーは明示的な戻り値なしに non-void 関数末端からケースを検出します。従って、戻り値は予想できない場合があります。

conversion lacks type at end of format －変換によりフォーマットの末尾のタイプが欠落しています

printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列のフォーマットフィールドに型指定がないことを検出しました。

concatenation of string literals with __FUNCTION__ is deprecated －文字列リテラルと __FUNCTION__ との結合が重複しています
__FUNCTION__ は __func__（ISO 標準 C99 で定義）と同じように取り扱われます。__func__ は変数で、文字列リテラルではありません。そのため、他の文字列リテラルとは結合しません。

conflicting types for‘*identifier*’－‘*identifier*’に矛盾したタイプです
指定した identifier が複数の矛盾する宣言を含んでいます。

D

data definition has no type or storage class －データ定義がタイプまたはストレージクラスを持っていません
データ定義にタイプとストレージクラスがないことが検出されました。

data qualifier‘*qualifier*’ignored －データ修飾子‘*qualifier*’が無視されました
‘access’、‘shared’、‘overlay’などのデータ修飾子は MPLAB C30 では使用されませんが、MPLAB C17 および C18 との互換性のためにあります。

declaration of‘*identifier*’has‘extern’and is initialized －‘*identifier*’の宣言は ‘extern' を持ち、初期化されています
Extern は初期化すべきではありません。

declaration of‘*identifier*’shadows a parameter －‘*identifier*’の宣言はパラメータをシャドウします
指定した *identifier* 宣言はパラメータをシャドウし、パラメータをアクセス不能にします。

declaration of‘*identifier*’shadows a symbol from the parameter list －‘*identifier*’の宣言はパラメータリストからの記号をシャドウします
指定した identifier 宣言はパラメータリストからの記号をシャドウし、記号をアクセス不能にします。

declaration of‘*identifier*’shadows global declaration －‘*identifier*’の宣言はグローバル宣言をシャドウします
指定した *identifier* 宣言はグローバル宣言をシャドウし、グローバルをアクセス不能にします。

‘*identifier*’declared inline after being called －‘*identifier*’はコールされた後にインラインを宣言しました
指定した関数はコールされた後インラインを宣言しました。

‘*identifier*’declared inline after its definition －‘*identifier*’は定義の後インラインを宣言しました
指定した関数は定義された後インラインを宣言しました。

‘*identifier*’declared‘static’but never defined －‘*identifier*’は‘static’を宣言しましたが、定義されていません
指定した関数は static を宣言しましたが、定義されていません。

decrement of read-only member‘*name*’－読み取り専用メンバー‘*name*’の減分です
メンバー‘name’が const として宣言され、減分では修正できません。

decrement of read-only variable‘*name*’－読み取り専用変数‘*name*’の減分です
‘name’が const として宣言され、減分では修正できません。

‘*identifier*’defined but not used －‘*identifier*’が定義されましたが、使用されていません
指定した関数は定義されましたが、使用されていません。

deprecated use of label at end of compound statement －複合ステートメント末尾でラベル使用が重複しています
ラベルはステートメントの末尾に置くべきではありません。ステートメントの前に付けます。

dereferencing‘void *’pointer －‘void*' ポインタを修飾参照しています
‘void*’ポインタを修飾参照するのは不正です。ポインタを修飾参照する前に、適切なタイプのポインタをキャストしてください。

division by zero －ゼロによる割り算
コンパイル時にゼロによる割り算が検出されました。

duplicate‘const’－重複した‘const’
‘const’修飾子は宣言に 1 度だけ適用します。
duplicate‘restrict’－重複した‘restrict’
‘restrict’修飾子は宣言に 1 度だけ適用します。
duplicate‘volatile’－重複した‘volatile’
‘volatile’修飾子は宣言に 1 度だけ適用します。

E

embedded‘\0’in format －フォーマットに‘\0’が組み込まれています
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列のフォーマットフィールドが‘\0’（ゼロ）を組み込んでいることを検出しました。これはフォーマット文字列処理の早期終了を引き起こすことがあります。
empty body in an else-statement － else- ステートメント内に空の本文があります
else ステートメントは空です。
empty body in an if-statement － if- ステートメント内に空の本文があります
if ステートメントは空です。
empty declaration －空の宣言
宣言には宣言する名前が含まれていません。
empty range specified －空の範囲が指定されました
case 範囲内の値の範囲が空であるため、低位表現の値が上位表現より高く扱われています。case 範囲の構文を再コールしてください。case *low ... high:*。
‘enum *identifier*’declared inside parameter list －‘enum *identifier*’がパラメータリスト内で宣言されました
指定した enum が関数パラメータリスト内で宣言されました。通常のよりよいプログラミング慣習としては、パラメータリスト外で enum を宣言します。パラメータリスト内で定義される際、これらは完全なタイプとならないからです。
enum defined inside parms －パラメータ内で enum が定義されました
enum が関数パラメータリスト内で定義されました。
enumeration value‘*identifier*’not handled in switch －列挙型の値‘*identifier*’は switch 内では取り扱われません
switch ステートメントの制御表現は列挙型タイプですが、すべての列挙型の値は case 表現を持っているわけではありません。
enumeration values exceed range of largest integer －列挙型の値が最大整数の範囲を超えています
列挙型の値は整数として表現されます。列挙型範囲はそのような最大フォーマットを含む MPLAB C30 整数フォーマットで表現不可能なことをコンパイラーが検出しました。
excess elements in array initializer －配列イニシャライザ内のエレメント数を超えています
イニシャライザリスト内のエレメント数が一緒に宣言された配列のエレメントより多くなっています。
excess elements in scalar initializer"); －スカラーイニシャライザ "); 内のエレメント数を超えています
スカラー変数に必要なイニシャライザは 1 つのみです。
excess elements in struct initializer －構造イニシャライザ内のエレメント数を超えています
イニシャライザリスト内のエレメント数が一緒に宣言された構造のエレメントより多くなっています。
excess elements in union initializer － union イニシャライザ内のエレメント数を超えています
イニシャライザリスト内のエレメント数が一緒に宣言された union のエレメントより多くなっています。

extra semicolon in struct or union specified － struct または union 内に余分なセミコロンが指定されています
構造体タイプまたは union タイプが余分なセミコロンを含んでいます。
extra tokens at end of #‘*directive*’directive － #‘*directive*’ディレクティブの末尾に余分なトークンがあります
コンパイラーは #‘directive’ディレクティブを含むソース行で余分なテキストを検出しました。
F
-ffunction-sections may affect debugging on some targets － -ffunction- セクションはある種のターゲットのデバッグに影響を及ぼす可能性があります
-g オプションと -ffunction-sections オプションの両方を指定するとデバッグに問題が発生する可能性があります。
first argument of‘*identifier*’should be‘int’－‘*identifier*’の最初の引数は‘int’にする必要があります
指定した識別子の最初の引数宣言はタイプ int である必要があります。
floating constant exceeds range of‘double’－浮動小数点定数が‘double’の範囲を超えます
浮動小数点定数は‘double’として表現するには大きすぎるか、小さすぎます（振り幅）。
floating constant exceeds range of‘float’－浮動小数点定数が‘float’の範囲を超えます
浮動小数点定数は‘float’として表現するには大きすぎるか、小さすぎます（振り幅）。
floating constant exceeds range of‘long double’－浮動小数点定数が‘long double’の範囲を超えます
浮動小数点定数は‘long double’として表現するには大きすぎるか、小さすぎます（振り幅）。
floating point overflow in expression －表現内で浮動小数点がオーバーフローします
浮動小数点定数表現を畳み込む際に、コンパイラーは表現がオーバーフローしていることを検出しました。従って、これは浮動小数点としては表現できません。
‘*type1*’format,‘*type2*’arg (arg‘*num*’) －‘*type1*’フォーマット、‘*type2*’引数（引数‘*num*’）
フォーマットはタイプ‘type1’ですが、渡される引数はタイプ‘type2’です。問題の引数は、‘num’引数です。
format argument is not a pointer (arg *n*) －フォーマット引数はポインタ (arg *n*) ではありません
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーは指定した引数 *n* はフォーマット指定が指示しているとおりのポインタではないことを検出しました。
format argument is not a pointer to a pointer (arg *n*) －フォーマット引数はポインタ (arg *n*) へのポインタではありません
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーは指定した引数 *n* はフォーマット指定が指示しているとおりのポインタではないことを検出しました。
fprefetch-loop-arrays not supported for this target － fprefetch- ループ配列はこのターゲットをサポートしていません
メモリ先取り指示を作成するオプションはこのターゲットではサポートされていません。
function call has aggregate value －関数コールが集合数を持っています
関数の戻り値が集合数です。
function declaration isn’t a prototype －関数宣言がプロトタイプではありません
`-Wstrict-prototypes` コマンドラインオプションをコンパイルする際、コンパイラーは関数プロトタイプがすべての関数を指定していることを確認しました。この場合、前に付くプロトタイプがない関数定義を検出できます。
function declared‘noreturn’has a‘return’statement －関数は‘noreturn’が‘return’ステートメントを持つと宣言しました
関数が noreturn 属性とともに宣言されました。これは関数が戻り値を返さないことを意味しますが、関数は戻りステートメントを含んでいます。よって、この宣言は矛盾しています。

function might be possible candidate for attribute‘noreturn’－関数は属性‘noreturn’候補である可能性があります
コンパイラーは関数が戻り値を返さないことを検出しました。関数が‘noreturn’属性とともに宣言されていれば、コンパイラーはよりよいコードを作成できます。
function returns address of local variable －関数はローカル変数のアドレスを返します
関数はローカル変数のアドレスを返すべきではありません。関数がこの値を返すと、ローカル変数が再割り当てされるためです。
function returns an aggregate －関数は集合数を返します
関数の戻り値が集合数です。
function‘*name*’redeclared as inline －関数‘*name*’がインラインとして再宣言されました
previous declaration of function‘*name*’with attribute noinline －以前の関数‘*name*’の宣言では、属性は noinline でした
関数‘name’は 2 度目の宣言ではキーワード‘inline' で宣言され、インライン化が行われます。
function‘*name*’redeclared with attribute noinline －関数‘*name*’が属性 noinline で宣言され
previous declaration of function‘*name*’was inline －以前の関数‘*name*’の宣言では、inline でした
関数‘name’は 2 度目の宣言では noinline 属性で宣言され、インライン化できません。
function‘*identifier*’was previously declared within a block －関数‘*identifier*’は以前ブロック内で宣言されました
指定した関数はブロック内の以前の明示的宣言を持っていますが、現在の行で暗示的宣言も持っています。

G

GCC does not yet properly implement‘[*]’array declarators － GCC は‘[*]’配列宣言を適切に実行していません
変数長の配列は現在、コンパイラーではサポートされていません。

H

hex escape sequence out of range － hex エスケープシーケンスが範囲外です
hex シーケンスは 16 進法で 100 未満（10 進法で 256）にする必要があります。

I

ignoring asm-specifier for non-static local variable‘*identifier*’－非静的ローカル変数‘*identifier*’の asm 規則子を無視します
asm 規則子は普通の非レジスタローカル変数と使用すると無視されます。
ignoring invalid multibyte character －無効なマルチバイト文字を無視します
マルチバイト文字を構文解析している際に、コンパイラーはそれを無効と判断しました。無効な文字は無視されます。
ignoring option‘*option*’due to invalid debug level specification －オプション‘*option*’はデバッグレベル仕様が無効なため無視します
デバッグオプションが有効でないデバッグレベルとともに使用されています。
ignoring #pragma *identifier* － #pragma 識別子を無視します
指定した pragma は MLAB C30 コンパイラーではサポートされていないため、無視されます。
imaginary constants are a GCC extention －虚定数は GCC 拡張です
ISO C は虚定数を許可しません。
implicit declaration of function‘*identifier*’－関数‘*identifier*’の暗黙の宣言です
指定した関数は以前の明示的宣言（定義または関数プロトタイプ）を含んでいないため、コンパイラーがその戻りタイプおよびパラメータを予測しました。

increment of read-only member‘*name*’ －読み取り専用メンバー‘*name*’の増分です
メンバー‘name' が const として宣言され、増分では更新できません。
increment of read-only variable‘*name*’ －読み取り専用変数‘*name*’の増分です
‘name’が const として宣言され、増分では更新できません。
initialization of a flexible array member －フレキシブル配列メンバーの初期化です
フレキシブル配列メンバーは静的にではなく、動的に割り当てられるようになっています。
‘*identifier*’initialized and declared‘extern’ －‘identifier' が初期化され Extern 宣言された
Extern は初期化すべきではありません。
initializer element is not constant －イニシャライザエレメントが定数ではありません
イニシャライザエレメントは定数にする必要があります。
inline function‘*name*’given attribute noinline －インライン関数‘*name*’に属性 noinline が付与されています
関数‘name’はインラインとして宣言されましたが、noinline 属性が関数のインライン化を妨げています。
inlining failed in call to‘*identifier*’called from here －ここからコールされた '*identifier*' へのコールをインライン化できません
コンパイラーは指定した関数へのコールをインライン化できませんでした。
integer constant is so large that it is unsigned －静定数が大きすぎて符号を付与できません
整定数値は明示的符号無し修飾子無しのソースコード中に表示されますが、符号付き int としてこの数字を表すことはできません。従って、コンパイラーは自動的に符号無し int としてこれを扱います。
integer constant is too large for‘*type*’type －整定数が‘*type*’タイプには大きすぎます
整定数は符号無し long int では 2^32 - 1、long long int では 2^63 - 1 もしくは符号無し long long int では 2^64 - 1 を超えないようにしてください。
integer overflow in expression －整数が表現内でオーバーフローしています
整定数表現を畳み込む際に、コンパイラーは表現がオーバーフローしていることを検出しました。従って、これは int としては表現できません。
invalid application of‘sizeof’to a function type －関数タイプへの‘sizeof’適用が無効です
sizeof 演算子を関数タイプに適用するのはお奨めできません。
invalid application of‘sizeof’to a void type － void タイプへの‘sizeof’適用が無効です
sizeof 演算子は void タイプに適用すべきではありません。
invalid digit‘digit’in octal constant －無効な数字‘digit’が 8 進定数内にあります
すべての数字を使用されている基数内にする必要があります。例えば、8 進法で使用できるのは 0 から 7 の数字のみです。
invalid second arg to __builtin_prefetch; using zero － __builtin_prefetch; の第二引数に無効な値 が使用されています ; 0 にして下さい
第二引数は 0 または 1 にする必要があります。
invalid storage class for function‘*name*’ －関数‘*name*’の無効なストレージクラス
‘auto’ストレージクラスは上位レベルで定義された関数上で使用するべきではありません。関数が上位レベルで定義されていない場合は、‘static’ストレージクラスは使用するべきではありません。
invalid third arg to __builtin_prefetch; using zero － __builtin_prefetch; への無効な第三引数に 0 が使用されています
第三引数は 0、1、2 または 3 にする必要があります。
‘*identifier*’is an unrecognized format function type －‘*identifier*’は認識されないフォーマット関数タイプです
フォーマット属性として指定した *identifier* はフォーマット関数タイプ、`printf, scanf, strftime` 用のフォーマットとして認識できません。

‘*identifier*’is narrower than values of its type － ‘*identifier*’の幅がそのタイプの値より狭くなっています
構造体のビットフィールドメンバーは列挙型のタイプを持っていますが、フィールドの幅がすべての列挙値を表現するのに不十分です。
‘*storage class*’is not at beginning of declaration － ‘*storage class*’が宣言の最初にありません
指定したストレージクラスが宣言の最初にありません。宣言の最初にはストレージクラスが来る必要があります。
ISO C does not allow extra ‘;’ outside of a function － ISO C は関数外の余分な ‘;’ を許可しません
余分な‘;’が関数の外で見つかりました。ISO C では許可されません。
ISO C does not support ‘++’ and ‘--’ on complex types － ISO C は複素数型の ‘++' および ‘--' はサポートしません
増分演算子と減分演算子は ISO C の複素数型ではサポートされていません。
ISO C does not support ‘~’ for complex conjugation － ISO C は複合接合の ‘~’ はサポートしません
bitwise 否定演算子は ISO C の複合接合では使用できません。
ISO C does not support complex integer types － ISO C は複合整数タイプはサポートしません
__complex__short int などの複合整数タイプは ISO ではサポートされていません。
ISO C does not support plain ‘complex’ meaning ‘double complex’ － ISO C は ‘double complex’ を意味する単純 ‘complex’ はサポートしません
他の修飾子を持たない __complex__ の使用は ISO C でサポートされていない ‘complex double’ と同等です。
ISO C does not support the ‘*char*’ ‘*kind of format*’ format － ISO C は ‘*char*’ ‘*kind of format*’ フォーマットはサポートしません
ISO C は指定した‘kind of format’の指定文字‘char’はサポートしません。
ISO C doesn’t support unnamed structs/unions － ISO C は名前のない struct/union はサポートしません
ISO C ではすべての構造体および / またはユニオンに名前をつける必要があります。
ISO C forbids an empty source file － ISO C は空のソースファイルを禁じています
ファイルは関数またはデータを含んでいません。これは ISO C では許可されません。
ISO C forbids empty initializer braces － ISO C は空のイニシャライザ中括弧を禁じています
ISO C は中括弧内にイニシャライザの値を入れるよう要求しています。
ISO C forbids nested functions － ISO C はネストされた関数を禁じています
関数が他の関数内で定義されています。
ISO C forbids omitting the middle term of a ?: expression － ISO C は ?: 表現の中名辞の省略を禁じています
条件式は‘?' と ‘:' の間に中間式を要求しています。
ISO C forbids qualified void function return type － ISO C は修飾された void 関数戻りタイプを禁じています
void 関数戻りタイプには修飾子を使用しないでください。
ISO C forbids range expressions in switch statements － ISO C は switch ステートメント内の範囲表現を禁じています
ISO C では、単一ケースラベル内に連続した値の範囲を指定することを禁じています。
ISO C forbids subscripting ‘register’ array － ISO C は ‘register’ 配列をサブスクリプトすることを禁じています
ISO C では、‘register’配列のサブスクリプトを禁じています。
ISO C forbids taking the address of a label － ISO C はラベルのアドレスを取得することを禁じています
ISO C では、ラベルのアドレスを取得することは禁じています。
ISO C forbids zero-size array ‘*name*’ － ISO C はゼロサイズの配列 ‘*name*’ を禁じています
‘name’の配列サイズは 0 より大きくする必要があります。

ISO C restricts enumerator values to range of‘int’ － ISO C は‘int’の範囲に列挙型の値を制限しています
列挙型の値の範囲は int タイプの範囲を超えてはなりません。
ISO C89 forbids compound literals － ISO C89 は複合リテラルを禁じています
ISO C89 では、複合リテラルは無効です。
ISO C89 forbids mixed declarations and code － ISO C89 は宣言とコードの混合を禁じています
コードが書き込まれる前に宣言をする必要があります。宣言をコードに含めるべきではありません。
ISO C90 does not support‘[*]’array declarators － ISO C98 は‘[*]’配列宣言をサポートしていません
変数長の配列は ISO C90 ではサポートされていません。
ISO C90 does not support complex types － ISO C90 は複合タイプはサポートしません
__complex__ float x などの複合タイプは ISO C90 ではサポートされていません。
ISO C90 does not support flexible array members － ISO C90 はフレキシブル配列メンバーはサポートしません
フレキシブル配列メンバーは C99 の新しい機能です ISO C90 ではサポートされていません。
ISO C90 does not support‘long long’ － ISO C90 は‘long long’はサポートしません
long long タイプは ISO C90 ではサポートされていません。
ISO C90 does not support‘static’or type qualifiers in parameter array declarators － ISO C90 はパラメータ配列宣言子内の‘static’またはタイプ修飾子はサポートしません
配列を関数のパラメータとして使用する際、ISO C90 は配列宣言子が static またはタイプ修飾子を使用するのを許可しません。
ISO C90 does not support the‘*char*’‘*function*’format － ISO C90 は‘*char*’‘*function*’フォーマットはサポートしません
ISO C は指定した関数フォーマットとしての指定文字‘char’はサポートしません。
ISO C90 does not support the‘*modifier*’‘*function*’length modifier － ISO C90 は‘*modifier*’‘*function*’長さ変更子はサポートしません
指定された修飾子はその関数の長さ修飾子としてはサポートされていません。
ISO C90 forbids variable-size array‘*name*’ － ISO C90 は変数サイズの配列‘*name*’を禁じています
ISO C90 では、配列内のエレメントは整定数表現で指定する必要があります。

L

label‘*identifier*’defined but not used －ラベル‘*identifier*’が定義されましたが、使用されていません
指定したラベルは定義されましたが参照されていません。
large integer implicitly truncated to unsigned type －サイズの大きな整数が暗示的に符号無しタイプに短縮されました
整定数値は明示的符号無し修飾子無しのソースコード中に表示されますが、符号付き int としてこの数字を表すことはできません。従って、コンパイラーは自動的に符号無し int としてこれを扱います。
left-hand operand of comma expression has no effect －カンマ表現の左側のオペランドは効果がありません
比較のオペランドのうち一方が promoted ~unsigned で、もう一方が符号無しです。
left shift count >= width of type －左シフトカウント >= タイプの幅
シフトのカウントはシフトされたタイプ内のビット数を超えてはいけません。そうでないと、シフトは無意味になり、結果が定義されません。
left shift count is negative －左シフトのカウントが負の数です
シフトのカウントは正の数でなくてはなりません。左シフトのカウントが負の数でも、右のシフトを意味しません。それは無意味です。

library function‘*identifier*’declared as non-function －ライブラリ関数‘*identifier*’は非関数として宣言されました
指定された関数はライブラリ関数と同じ名前を持っていますが、関数以外のものとして宣言されました。
line number out of range －行数が範囲外です
#line ディレクティブの行数の制限は、C89 で 32767、C99 で 2147483647 です。
‘*identifier*’locally external but globally static －‘*identifier*’はローカルでは外部ですが、グローバルでは静的です
指定した‘*identifier*’はローカルでは外部ですが、グローバルでは静的です。これは疑わしい状態といえます。
location qualifier‘*qualifier*’ignored －ロケーション修飾子‘*qualifier*’が無視されました
‘grp’、‘sfr’などのロケーション修飾子は MPLAB C30 では使用されませんが、MPLAB C17 および C18 との互換性のためにあります。
‘long’switch expression not converted to‘int’in ISO C － ISO C では、‘long’switch 表現は‘int’に変換されません
ISO C は‘long’switch 表現を‘int’に変換しません。
M
‘main’is usually a function －‘main’は通常、関数です
識別子 main は通常、アプリケーションのメインエントリポイントの名前に使用されます。コンパイラーはこの識別子が変数の名前など他の方法で使用されていることを検出しました。
‘*operation*’makes integer from pointer without a cast －‘*operation*’はキャスト無しのポインタから整数を作成します
ポインタは暗示的に整数に変換されます。
‘*operation*’makes pointer from integer without a cast －‘*operation*’は整数からキャスト無しのポインタを作成します
整数は暗示的にポインタに変換されます。
malformed‘#pragma pack-ignored’－不正な形式の‘#pragma pack-ignored’です
pack pragma の構文が正しくありません。
malformed‘#pragma pack(pop[,id])-ignored’－不正な形式の‘#pragma pack(pop[,id])-ignored’です
pack pragma の構文が正しくありません。
malformed‘#pragma pack(push[,id],<n>)-ignored’－不正な形式の‘#pragma pack(push[,id],<n>)-ignored’です
pack pragma の構文が正しくありません。
malformed‘#pragma weak-ignored’－不正な形式の‘#pragma weak-ignored’です
weak pragma の構文が正しくありません。
‘*identifier*’might be used uninitialized in this function －この関数で初期化解除された‘*identifier*’が使用されている可能性があります
コンパイラーは初期化する前に指定した識別子を使用した可能性のある関数を介した制御パスを検出しました。
missing braces around initializer －イニシャライザの前後に中括弧がありません
イニシャライザの前後に必要な中括弧がありません。
missing initializer －イニシャライザが見つかりません
イニシャライザが見つかりません。
modification by‘asm’of read-only variable‘*identifier*’－‘asm’を使用した読み取り専用変数‘*identifier*’の修正です
const 変数が‘asm’ステートメント内代入の左端にあります。
multi-character *character* constant －複数文字 *character* 定数です
文字定数が 2 つ以上の文字を含んでいます。

N

negative integer implicitly converted to unsigned type －負の整数が暗示的に符号無しタイプに変換されました

負の整定数値はソースコード中に表示されますが、符号付き int としてこの数字を表すことはできません。従って、コンパイラーは自動的に符号無し int としてこれを扱います。

nested extern declaration of‘*identifier*’－‘*identifier*’のネストされた extern 宣言です

指定した *identifier* のネストされた extern 定義があります。

no newline at end of file －ファイルの最後に newline がありません

ソースファイルの最後の行が newline 文字で終わっていません。

no previous declaration for‘*identifier*’－‘*identifier*’の事前の宣言がありません

`-Wmissing-declarations` コマンドラインオプションをコンパイルする際、コンパイラーは関数が定義の前に宣言されていることを確認しました。この場合、前に付く関数宣言なしに関数定義が検出されます。

no previous prototype for‘*identifier*’－‘*identifier*’の事前のプロトタイプがありません

`-Wmissing-prototypes` コマンドラインオプションをコンパイルする際、コンパイラーは関数プロトタイプがすべての関数に指定されていることを確認します。この場合、事前の関数プロトタイプなしに関数定義が検出されました。

no semicolon at end of struct or union － struct または union の末尾にセミコロンがありません

構造体またはユニオン宣言の末尾にセミコロンがありません。

non-ISO-standard escape sequence, ‘*seq*’－ ISO 標準ではないエスケープシーケンス‘*seq*’です

‘seq’が‘\e’または‘\E’で、ISO 標準の拡張子です。シーケンスは文字列または文字定数で使用され、ASCII 文字 <ESC> を表します。

non-static declaration for‘*identifier*’follows static －静的表現の後に‘*identifier*’の静的ではない宣言が付いています

指定した識別子は静的として宣言された後に非静的として宣言されました。

‘noreturn’function does return －‘noreturn’関数が戻り値を返します

noreturn 属性で宣言された関数が戻り値を返します。これは矛盾しています。

‘noreturn’function returns non-void value －‘noreturn’関数が非 void 値を返します

noreturn 属性で宣言された関数が非 void 値を返します。これは矛盾しています。

null format string － null フォーマット文字列

printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列がないことを検出しました。

O

octal escape sequence out of range － octal エスケープシーケンスが範囲外です

octal シーケンスは 8 進法で 400 未満（10 進法で 256）にする必要があります。

output constraint‘*constraint*’for operand *n* is not at the beginning －オペランド *n* の出力制約‘*constraint*’が文頭にありません

拡張 asm の出力制約は文頭に来る必要があります。

overflow in constant expression －定数表現でオーバーフローが発生しています

定数表現の長さがその型の相当値の範囲を超えています。

overflow in implicit constant conversion －暗示的定数変換でオーバーフローが発生しています

暗示的定数変換は数字で表示されますが、符号付き int としてこの数字を表すことはできません。従って、コンパイラーは自動的に符号無し int としてこれを扱います。

P

parameter has incomplete type －パラメータが不完全なタイプを含んでいます
関数パラメータが不完全なタイプを含んでいます。
parameter names (without types) in function declaration －関数宣言内のパラメータ名（タイプ無し）です
関数宣言はタイプではなくパラメータ名をリストします。
parameter points to incomplete type －パラメータは不完全なタイプを指定しています
関数パラメータは不完全なタイプを指定しました。
parameter‘*identifier*’points to incomplete type －パラメータ‘*identifier*’は不完全なタイプを指定しています
指定した関数パラメータは不完全なタイプを指定しました。
passing arg‘*number*’of‘*name*’as complex rather than floating due to prototype －プロトタイプに基づき、浮動値ではなく‘*name*’の引数‘*number*’を複合値として渡しました
プロトタイプは引数‘number’を複合体として宣言しましたが、浮動値が使用されたため、コンパイラーはこれを複合値に変換してプロトタイプと適合するようにしました。
passing arg‘*number*’of‘*name*’as complex rather than integer due to prototype －プロトタイプに基づき、整数ではなく‘*name*’の引数‘*number*’を複合値として渡しました
プロトタイプは引数‘number' を複合値として宣言しましたが、整数値が使用されたため、コンパイラーはこれを複合値に変換してプロトタイプと適合するようにしました。
passing arg‘*number*’of‘*name*’as floating rather than complex due to prototype －プロトタイプに基づき、複合値ではなく‘*name*’の引数‘*number*’を浮動値として渡しました
プロトタイプは引数‘number’を浮動値として宣言しましたが、複合値が使用されたため、コンパイラーはこれを浮動値に変換してプロトタイプと適合するようにしました。
passing arg‘*number*’of‘*name*’as‘float’rather than‘double’due to prototype －プロトタイプに基づき、‘double’ではなく‘*name*’の引数‘*number*’を‘float’として渡しました
プロトタイプは引数‘number’を浮動値として宣言しましたが、double が使用されたため、コンパイラーはこれを浮動値に変換してプロトタイプと適合するようにしました。
passing arg‘*number*’of‘*name*’as floating rather than integer due to prototype －プロトタイプに基づき、整数ではなく‘*name*’の引数 ‘*number*’を浮動値として渡しました
プロトタイプは引数‘number’を浮動値として宣言しましたが、整数が使用されたため、コンパイラーはこれを浮動値に変換してプロトタイプと適合するようにしました。
passing arg‘*number*’of‘*name*’as integer rather than complex due to prototype －プロトタイプに基づき、複合値ではなく‘*name*’の引数 ‘*number*’を整数として渡しました
プロトタイプは引数‘number’を整数として宣言しましたが、複合値が使用されたため、コンパイラーはこれを整数に変換してプロトタイプと適合するようにしました。
passing arg‘*number*’of‘*name*’as integer rather than floating due to prototype －プロトタイプに基づき、浮動値ではなく‘name’の引数 ‘*number*’を整数として渡しました
プロトタイプは引数‘number’を整数として宣言しましたが、浮動値が使用されたため、コンパイラーはこれを整数に変換してプロトタイプと適合するようにしました。
pointer of type‘void *’used in arithmetic －タイプ‘void *’のポインタが演算に使用されました
タイプ‘void' のポインタはサイズがないので、演算には使用すべきではありません。
pointer to a function used in arithmetic －関数へのポインタが演算に使用されました
関数へのポインタは演算に使用すべきではありません。
previous declaration of‘*identifier*’－‘*identifier*’の以前の宣言です
この警告メッセージは他の警告メッセージに関連して表示されます。以前のメッセージは疑わしいコードの場所を示します。このメッセージは最初の宣言または *identifier* の定義を示します。
previous implicit declaration of‘*identifier*’－‘*identifier*’の以前の暗示的宣言です
この警告メッセージは、警告メッセージに“type mismatch with previous implicit declaration”に関連して表示されます。このメッセージは、明示的宣言と矛盾する識別子の暗示的宣言の場所を示します。

R

“*name*” re-asserted －“*name*”が再表明されました
“*name*”の解が重複しています。
“*name*” redefined －“*name*”が再定義されました
“name”は以前定義されていましたが、再度定義されました。
redefinition of‘*identifier*’－‘*identifier*’の再定義
指定した identifier が複数の互換性のない宣言を含んでいます。
redundant redeclaration of‘*identifier*’in same scope －同じ範囲内での‘*identifier*’の冗長な再宣言
指定した識別子は同じ範囲内で再宣言されました。これは冗長です。
register used for two global register variables － 2 つのグローバルレジスタ変数にレジスタが使用されています
2 つのグローバルレジスタ変数が同じレジスタを使用するように定義されています。
repeated‘*flag*’flag in format －フォーマット内の繰り返しの‘*flag*’フラグ
strftime へのコールの引数リストを確認する際、コンパイラーはフォーマット文字列内に繰り返されているフラグがあることを検出しました。
prinff、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフラグ {,+,#,0,-} の 1 つがフォーマット文字列内で繰り返されていることを検出しました。
return-type defaults to‘int’－戻りタイプは‘int’のデフォルトです
明示的関数戻りタイプ宣言がないため、コンパイラーは関数の戻り値を int と想定しました。
return type of‘*name*’is not‘int’－‘*name*' の戻りタイプが‘int' ではありません
コンパイラーは‘name' の戻りタイプとして‘int' を期待しています。
‘return’with a value, in function returning void － void を返すとされた関数が値を返しました
関数は void として宣言されましたが、値を返しました。
‘return’with no value, in function returning non-void －非 void とされた関数が値無しで戻っています
非 void 値を返すことを宣言した関数が、値のない戻りステートメントを含んでいます。これは矛盾しています。
right shift count >= width of type －右シフトカウント >= タイプの幅
シフトのカウントはシフトされたタイプ内のビット数を超えてはいけません。そうでないと、シフトは無意味になり、結果が定義されません。
right shift count is negative －右シフトのカウントが負の数です
シフトのカウントは正の数でなくてはなりません。右シフトのカウントが負の数でも、左のシフトを意味しません。それは無意味です。

S

second argument of‘*identifier*’should be‘char **’－‘identifier’の 2 番目の引数は‘char **’にする必要があります
指定した識別子の 2 番目の引数はタイプ‘char **’である必要があります。
second parameter of‘va_start’not last named argument －‘va_start’の 2 番目のパラメータは最後に名づけられた引数ではありません
‘va_start’の 2 番目のパラメータは最後に名づけられた引数である必要があります。
shadowing built-in function‘*identifier*’－ビルトイン関数‘*identifier*’のシャドウです
指定した関数はビルトイン関数と同じ名前を持っているので、ビルトイン関数をシャドウします。
shadowing library function‘*identifier*’－ライブラリ関数‘*identifier*’のシャドウです
指定した関数はライブラリ関数と同じ名前を持っているので、ライブラリ関数をシャドウします。

shift count >= width of type －シフトカウント >= タイプの幅
シフトのカウントはシフトされたタイプ内のビット数を超えてはいけません。そうでないと、シフトは無意味になり、結果が定義されません。

shift count is negative －シフトのカウントが負の数です
シフトのカウントは正の数でなくてはなりません。負の数の左シフトカウントは右シフトを意味しません。負の数の右シフトカウントも左シフトを意味しません。これらは無意味です。

size of‘*name*’is larger than *n* bytes －‘*name*’のサイズが *n* より大きくなっています
`-Wlarger-than-len` を使用すると、‘name’のサイズが定義された *len* バイトより大きい場合に上記の警告が生成されます。

size of‘*identifier*’is n bytes －‘*identifier*’のサイズは n バイトです
指定した識別子のサイズ（*n* バイト）が指定した `-Wlarger-than-`*len* コマンドラインオプションより大きくなっています。

size of return value of‘*name*’is larger than *n* bytes －‘*name*’の戻り値のサイズが *n* バイトより大きくなっています
`-Wlarger-than-len` を使用すると、‘name’の戻り値のサイズが定義された *len* バイトより大きい場合に上記の警告が生成されます。

size of return value of‘*identifier*’is n bytes －‘*identifier*’の戻り値のサイズは n バイトです
指定した識別子の戻り値のサイズは *n* バイトで、`-Wlarger-than-`*len* コマンドラインオプションで指定したサイズより大きくなっています。

spurious trailing‘%’in format －フォーマットに偽の '%' があります
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列内に偽の‘%' 文字を検出しました。

statement with no effect －効果のないステートメントです
効果のないステートメントです。

static declaration for‘*identifier*’follows non-static －‘identifier' の非静的宣言の後には静的宣言が付いています
指定した識別子は非静的として宣言された後に静的として宣言されました。

string length‘*n*’is greater than the length‘*n*’ISO C*n* compilers are required to support －文字列長‘*n*’は ISO C*n* コンパイラーでサポートする必要がある長さ‘*n*’より長くなっています
ISO C89 の最長の文字列の長さは 509 です。ISO C99 の文字列の最長は 4095 です。

‘struct *identifier*’declared inside parameter list －‘struct *identifier*’がパラメータリスト内で宣言されました
指定した struct が関数パラメータリスト内で宣言されました。通常のよりよいプログラミング慣習としては、パラメータリスト外で struct を宣言します。パラメータリスト内で定義される際、これらは完全なタイプとならないからです。

struct has no members － struct はメンバーを持っていません
構造体は空でメンバーを持っていません。

structure defined inside parms －パラメータ内で union が定義されました
union が関数パラメータリスト内で定義されました。

style of line directive is a GCC extension － line ディレクティブのスタイルは GCC 拡張子です
伝統的 C では、フォーマット‘#line linenum’を使用します。

subscript has type‘char’－サブスクリプトはタイプ ‘char’を持ちます
配列サブスクリプトはタイプ‘*char*’を持ちます。

suggest explicit braces to avoid ambiguous‘else’－曖昧な‘else’を回避するため、明示的中括弧を推奨します
ネストされた if ステートメントは曖昧な else 項を持っています。曖昧さをなくすため、中括弧の使用をお奨めします。

suggest hiding #*directive* from traditional C with an indented # －字下げした # をもって # directive を伝統的 C から隠すようお奨めします
指定したディレクティブは伝統的 C ではなく、# を字下げして隠すことができます。ディレクティブは、その # がカラム 1 にない限り無視されます。
suggest not using #elif in traditional C －伝統的 C で #elif を使用しないようお奨めします
#elif は伝統的 K&R C では使用されるべきではありません。
suggest parentheses around assignment used as truth value －真（truth）の値として使用される代入の前後に中括弧を付けるようお奨めします
代入が真（truth）の値として使用される場合、ソースプログラムのリーダーにその意図を明確にするため、中括弧で囲む必要があります。
suggest parentheses around + or - inside shift －シフト内の + または - の前後に中括弧を付けるようお奨めします
suggest parentheses around && within || － || 内の && の前後に中括弧を付けるようお奨めします
suggest parentheses around arithmetic in operand of | －| のオペランド内の演算の前後に中括弧を付けるようお奨めします
suggest parentheses around comparison in operand of | －| のオペランド内の比較の前後に中括弧を付けるようお奨めします
suggest parentheses around arithmetic in operand of ^ － ^ のオペランド内の演算の前後に中括弧を付けるようお奨めします
suggest parentheses around comparison in operand of ^ － ^ のオペランド内の比較の前後に中括弧を付けるようお奨めします
suggest parentheses around + or - in operand of & － & のオペランド内の + または - の前後に中括弧を付けるようお奨めします
suggest parentheses around comparison in operand of & － & のオペランド内の比較の前後に中括弧を付けるようお奨めします
演算子の順位は C では適切に定義されていますが、明示的な中括弧がなく表現の読み取りが順位ルールのみに頼って行なわれる場合、表現内のオペランドの評価順序を理解する時間が数マイクロ秒ほど長引くことがあります。この場合に重要なのは、シフト内のオペレータ‘+’または‘-’の使用です。中括弧を使用してプログラマの意図を明確に表現すれば、リーダーは不要な手間を省くことができます。プログラマやコンパイラーにとっては曖昧と思えない表現でも中括弧を使用するようお奨めします。

T

‘*identifier*’takes only zero or two arguments －‘identifier' はゼロまたは 2 つの引数のみ取ります
ゼロまたは 2 つの引数のみ期待されています。
the meaning of‘\a’is different in traditional C －伝統的 C では‘\a’の意味は異なります
`-wtraditional` オプションが使用されると、エスケープシーケンス‘\a’はメタシーケンスとして認識されません。値は‘a’のみです。非伝統的なコンパイルでは、‘\a’は ASCII BEL 文字を意味します。
the meaning of‘\x’is different in traditional C －伝統的 C では‘\x’の意味は異なります
`-wtraditional` オプションが使用されると、エスケープシーケンス‘\x’はメタシーケンスとして認識されません。値は‘x’のみです。非伝統的なコンパイルでは、‘\x’は 16 進数エスケープシーケンスを示します。
third argument of‘*identifier*’should probably be‘char **’－‘*identifier*’の 3 番目の引数は‘char **’になる可能性があります
指定した識別子の 3 番目の引数はタイプ‘char **’である必要があります。
this function may return with or without a value －この関数は値を返す時と返さない時があります
non-void 関数からのすべての exit パスは適切な値を返す必要があります。コンパイラーが non-void 関数末端から明示的な戻り値がある時とない時があることを検出しました。従って、戻り値は予想できないことになります。
this target machine does not have delayed branches －このターゲットマシンは遅延分岐を持っていません
`-fdelayed-branch` オプションがサポートされていません。
too few arguments for format －フォーマットの引数が少なすぎます
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーは実際の引数の数がフォーマット文字列で必要な数より少ないことを検出しました。

too many arguments for format －フォーマットの引数が多すぎます
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーは実際の引数の数がフォーマット文字列で必要な数より多いことを検出しました。
traditional C ignores #‘*directive*’with the # indented －伝統的 C は字下げした # が付く #‘*directive*’を無視します
伝統的に、ディレクティブはその # がカラム 1 内にない限り無視されます。
traditional C rejects initialization of unions －伝統的 C は union の初期化を拒否します
伝統的 C では、union は初期化できません。
traditional C rejects the‘ul’suffix －伝統的 C は接尾辞‘ul’を拒否します
接尾辞‘u’は伝統的 C では有効ではありません。
traditional C rejects the unary plus operator －伝統的 C は単項＋演算子を拒否します
単項＋演算子は伝統的 C では有効ではありません。
trigraph ??*char* converted to *char* －3 重文字 ??*char* は *char* に変換されます
3 文字の連続、3 重文字はキーボードにない記号を表現するのに使用します。3 重文字シーケンスは以下のように変換されます。

??(= [| ??) =]	??< = {	??> = }	??= = #	??/ = \	??’ = ^	??! = |	??- = ~

trigraph ??*char* ignored － 3 重文字 ??*char* は無視されました
3 重文字シーケンスが無視されました。*char* は、(、)、<、>、=、/、'、!、または - になります。
type defaults to‘int’in declaration of‘*identifier*’－‘*identifier*’の宣言内ではタイプのデフォルトは‘int’になります
指定した *identifier* の明示的タイプ宣言がないため、コンパイラーはタイプを int と想定しました。
type mismatch with previous external decl previous external decl of‘*identifier*’－タイプが以前の‘*identifier*’の外部宣言とマッチしません
指定した識別子のタイプが以前の宣言とマッチしません。
type mismatch with previous implicit declaration －タイプが以前の暗示的宣言とマッチしません
明示的宣言が以前の暗示的宣言と矛盾します。
type of‘*identifier*’defaults to‘int’－‘*identifier*’のタイプのデフォルトは‘int’です
明示的タイプ宣言がないため、コンパイラーは識別子のタイプを int と想定しました。
type qualifiers ignored on function return type －関数戻りタイプではタイプ修飾子は無視されます
関数戻りタイプと一緒に使用されるタイプ修飾子は無視されます。

U

undefining‘defined’－定義されていない‘defined’です
‘defined’はマクロ名として使用できないので、未定義にはできません。
undefining‘*name*’－定義されていない‘*name*’です
#undef ディレクティブが以前定義されたマクロ名‘name’に使用されました。
union cannot be made transparent － union はトランスペアレントにできません
`transparent_union` 属性が union に適用されましたが、指定した変数が属性の要件を満たしていません。
‘union *identifier*’declared inside parameter list －‘union *identifier*’がパラメータリスト内で宣言されました
指定した union が関数パラメータリスト内で宣言されました。通常のよりよいプログラミング慣習としては、パラメータリスト外で union を宣言します。パラメータリスト内で定義される際、これらは完全なタイプとならないからです。

union defined inside parms －パラメータ内でユニオンが定義されました
ユニオンが関数パラメータリスト内で定義されました。
union has no members －ユニオンはメンバーを持っていません
ユニオンは空でメンバーを持っていません。
unknown conversion type character‘*character*’in format －フォーマット内に未知の変換タイプ文字‘*character*’があります
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列内の変換文字の 1 つが無効（認識不可能）であることを検出しました。
unknown conversion type character 0x*number* in format －フォーマット内に未知の変換タイプ文字 0x*number* があります
printf、scanf などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列内の変換文字の 1 つが無効（認識不可能）であることを検出しました。
unknown escape sequence‘*sequence*’－未知のエスケープシーケンス‘*sequence*’です
‘sequence' は有効なエスケープコードではありません。エスケープコードは‘\’で始まり、n、t、b、r、f、b、\、'、"、a、?、のいずれか 1 つの文字を使用するか、8 進または 16 進の数的シーケンスである必要があります。8 進数では、数的シーケンスは 8 進法で 400 未満にする必要があります。16 進数では、数的シーケンスは‘x’で始まり 16 進法の 100 未満にする必要があります。
unnamed struct/union that defines no instances －インスタンスを定義しない名前無しの struct/union
struct/union は空で名前を持っていません。
unreachable code at beginning of *identifier* － *identifier* の始まりで到達不可能なコード
指定した関数の始まりで到達不可能なコードがあります。
unrecognized gcc debugging option: *char* －認識されない gcc デバッギングオプション：*char*
-d*letters* デバッギングオプションでは、‘char’は有効な文字ではありません。
unused parameter‘*identifier*’－使用されていないパラメータ‘*identifier*’
指定した関数パラメータはその関数内では使用されません。
unused variable‘*name*’－使用されていない変数‘*name*’
指定した変数は定義されましたが使用されていません。
use of‘*’and‘flag’together in format －フォーマット内で‘*’と‘flag’が一緒に使用されています
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列内で‘*’と‘flag’の両方が使用されていることを検出しました。
use of C99 long long integer constants － C99 long long 整定数が使用されています
ISO C89 では、整定数を long long として宣言することは許可されません。
use of‘*length*’length modifier with‘*type*’type character －‘*length*’長さ修飾子が‘*type*’タイプ文字と一緒に使用されています
printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーは指定した長さが指定したタイプとともに不正に使用されていることを検出しました。
‘*name*’used but never defined －‘*name*’は使用されていますが、定義されていません
指定した関数は使用されましたが、定義されていません。
‘*name*’used with‘*spec*’‘*function*’format －‘*name*’は‘*spec*’‘*function*’フォーマットで使用されています
指定した関数のフォーマットの中では変換仕様‘spec’と一緒に使用される‘name’は有効ではありません。
useless keyword or type name in empty declaration －空の宣言の中では役に立たないキーワードまたはタイプ名です
空の宣言は役に立たないキーワードまたはタイプ名を含んでいます。

V

__VA_ARGS__ can only appear in the expansion of a C99 variadic macro － __VA_ARGS__ は C99 variadic マクロの拡張でのみ使用されます
定義済みマクロ __VA_ARGS は省略記号を使用したマクロ定義の代入部分で使用します。

value computed is not used －計算された値は使用されていません
計算された値は使用されていません。

variable‘*name*’declared‘inline’－変数‘*name*’に‘inline’を宣言しました
キーワード‘inline’は関数でのみ使用すべきです。

variable‘%s’might be clobbered by‘longjmp’or‘vfork’－変数 ‘%s’は‘longjmp’または‘vfork’によって修飾される可能性があります
非揮発性自動変数は longjmp へのコールにより変更される可能性があります。この警告はコンパイルの最適化時にのみ生成されます。

volatile register variables don’t work as you might wish －揮発性レジスタ変数が期待通りに動作しませんでした
変数を引数として渡すと、その変数は別のレジスタ（wO-w7）に移行され、引数の伝送のために指定（wO-w7 でない場合）されます。または、コンパイラーは指定したレジスタが適切でないので一時的にその値を他の場所に移すよう指示を発行します。この指示は指定したレジスタが非同期的（つまり、ISR を介して）に修正された場合にのみ発行されます。

W

-Wformat-extra-args ignored without -Wformat － -Wformat がないため -Wformat-extra-args が無視されました
`-Wformat-extra-args` を使用するには、-Wformat を指定する必要があります。

-Wformat-nonliteral ignored without -Wformat － -Wformat がないため -Wformat-nonliteral が無視されました
`-Wformat-nonliteral` を使用するには、-Wformat を指定する必要があります。

-Wformat-security ignored without -Wformat － -Wformat がないため -Wformat-security が無視されました
`-Wformat-security` を使用するには、-Wformat を指定する必要があります。

-Wformat-y2k ignored without -Wformat － -Wformat がないため -Wformat-y2k が無視されました
使用するには、`-Wformat` を指定する必要があります。

-Wid-clash-LEN is no longer supported － -Wid-clash-LEN は現在サポートされていません
オプション `-Wid-clash-LEN` は現在サポートされていません。

-Wmissing-format-attribute ignored without -Wformat － -Wformat がないため -Wmissing-format-attribute が無視されました
`-Wmissing-format-attribute` を使用するには、`-Wformat` を指定する必要があります。

-Wuninitialized is not supported without -O － -Wuninitialized は -O なしではサポートされません
`-Wuninitialized` オプションを使用するには、最適化をオンにする必要があります。

‘*identifier*’was declared‘extern’and later‘static’－‘*identifier*’は‘extern’として宣言され、その後‘static’として宣言されました
指定した識別子は‘extern’として宣言された後に静的として宣言されました。

‘*identifier*’was declared implicitly‘extern’and later‘static’－‘*identifier*’は暗示的に‘extern’として宣言され、その後‘static’として宣言されました
指定した識別子は暗示的に‘extern’として宣言された後に静的として宣言されました。

‘*identifier*’was previously implicitly declared to return‘int’－‘*identifier*’は‘int’を返すために以前暗示的に宣言されました
以前の暗示的宣言に対して不一致があります。

‘*identifier*’ was used with no declaration before its definition － ‘*identifier*’ はその定義の前に宣言に使用されていません

`-Wmissing-declarations` コマンドラインオプションをコンパイルする際、コンパイラーは関数が定義の前に宣言されていることを確認します。この場合、事前宣言の無い関数定義が検出されます。

‘*identifier*’ was used with no prototype before its definition － ‘*identifier*’ はその定義の前にプロトタイプに使用されていません

`-Wmissing-prototypes` コマンドラインオプションをコンパイルする際、コンパイラーは関数プロトタイプがすべての関数に指定されていることを確認します。この場合、コールされる前に関数プロトタイプが無い関数定義が検出されます。

writing into constant object (arg *n*) －定数オブジェクト (arg *n*) への書き込みです

printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーは指定した引数 *n* はフォーマット指定が書き込み先として指定した const オブジェクトが定数であることを検出しました。

Z

zero-length *identifier* format string －ゼロ長 *identifier* フォーマット文字列です

printf、*scanf* などへのコールの引数リストを確認する際、コンパイラーはフォーマット文字列が空 (“”) であることを検出しました。

別紙 C MPLAB C18 と MPLAB C30 C コンパイラ

C.1 はじめに

本章の目的は、MPLAB C18 と MPLAB C30 C コンパイラの違いを明確にすることです。MPLAB C18 コンパイラの詳細については、*MPLAB C18 C Compiler User's Guide* (DS51288) をご参照ください。

本章では、2 つのコンパイラの違いにつき、以下の項目について説明します。

- データフォーマット
- ポインタ
- ストレージクラスと関数の引数
- ストレージ修飾子
- 事前定義されたマクロ名
- 整数拡張
- 数値定数
- 文字列定数
- アノニマス構造体
- アクセスメモリ
- インラインアセンブリ
- Pragma
- メモリーモデル
- コール規則
- スタートアップコード
- コンパイラで管理されるリソース
- 最適化
- オブジェクトモジュールフォーマット
- 実装時定義された動作
- ビットフィールド

C.2 データフォーマット

表 C-1: データフォーマットで使用されるビット数

データフォーマット	MPLAB C18(1)	MPLAB C30(2)
char	8	8
int	16	16
short long	24	-
long	32	32
long long	-	64
float	32	32
double	32	32 or 64(3)

注 1: MPLAB C18 は独自のデータフォーマットを使用し、それは IEEE-754 フォーマットに似ていますが、上位 9 ビットが回転します。（表 C-2 をご参照ください）

2: MPLAB C30 は IEEE-754 フォーマットを使用します。

3: セクション 5.5「浮動少数」をご参照ください。

表 C-2: MPLAB C18 FLOATING-POINT と MPLAB C30 IEEE-754 フォーマット

標準	バイト 3	バイト 2	バイト 1	バイト 0
MPLAB C30	seeeeeee1	$e_0$ddd dddd16	dddd dddd8	dddd dddd0
MPLAB C18	eeeeeeee0	sddd dddd16	dddd dddd8	dddd dddd0

凡例： s = 符号ビット、d = 仮数、e = 指数

C.3 ポインタ

表 C-3: ポインタで用いられるビット数

メモリタイプ	MPLAB C18	MPLAB C30
プログラムメモリ －二ア	16	16
プログラムメモリ －ファー	24	16
データメモリ	16	16

C.4 ストレージクラス

MPLAB C18 では、変数用 overlay、および、関数引数用の auto もしくは static といった非 ANSI ストレージクラス指定子が使用できます。

MPLAB C30 ではこれらの指定子は使用できません。

C.5 スタックの使い方

表 C-4: 使用されるスタックのタイプ

スタック上の項目	MPLAB C18	MPLAB C30
戻りアドレス	ハードウエア	ソフトウエア
ローカル変数	ソフトウエア	ソフトウエア

C.6 ストレージ修飾子

MPLAB C18 は非 -ANSI の far、near、rom および ram のタイプの修飾子を使用します。

MPLAB C30 は非 -ANSI の far、near および space の属性を使用します。

例 C-1: near 変数を定義する

```
C18:  near int gVariable;
C30:  __attribute__((near)) int gVariable;
```

例 C-2: far 変数を定義する

```
C18:  far int gVariable;
C30:  __attribute__((far)) int gVariable;
```

例 C-3: プログラムメモリ内の変数を作成する

```
C18:  rom int gArray[6] = {0,1,2,3,4,5};
C30:  __attribute__((section(".romdata"), space(prog) ))
        int gArray[6] = {0,1,2,3,4,5};
```

C.7 定義されたマクロ名

MPLAB C18 は、選択されたメモリモデルにより、__18CXX, __18F242, ...（__ prefix を持ったほかのプロセッサ）と __SMALL__ もしくは __LARGE__ を定義します。

MPLAB C30 は __dsPIC30 を定義します。

C.8 整数拡張

MPLAB C18 は、両方のオペランドが int より小さくても、一番大きいオペランドのサイズで整数拡張を実行します。MPLAB C18 には標準に適合するように -Oi+ オプションがあります。

ISO で必須とされている int 精度もしくはそれより大きい値で整数拡張を実行します。

C.9 文字列定数

MPLAB C18 はプログラムメモリ内の文字列定数を、.stringtable セクション内でも保持します。MPLAB C18 は文字列関数のいくつかの変数をサポートします。例えば、strcpy 関数は４つの変数を持ち、文字列の、データ / プログラムメモリへの、もしくはデータ / プログラムメモリからのコピーが実施できます。

MPLAB C30 は、PSV ウインドウを通して、その他のデータがアクセスされるように、データメモリもしくはプログラムメモリからの文字列定数のアクセスができます。

C.10 アノニマス構造体

MPLAB C18 は、共有体の中で、非 ANSI のアノニマス構造体をサポートします。

MPLAB C30 はサポートしません。

C.11 アクセスメモリー

dsPIC30F デバイスはアクセスメモリを持ちません。

C.12 インラインアセンブリー

MPLAB C18 は、インラインアセンブリのブロックを識別するために、非 ANSI の _asm と _endasm を使用します。

MPLAB C30 は、関数コールのように見える、非 ANSI の asm を使用します。MPLAB C30 での asm 文の使用に関して詳細は、**セクション 8.4「インラインアセンブリ言語を使用する」**に述べられています。

C.13 PRAGMA

MPLAB C18 は、セクション用（code、romdata、udata、idata）、割り込み用（最優先と低い優先）および変数ロケーション用 (バンク、セクション) の pragma を用います。

MPLAB C30 は、pragma の代わりに非 ANSI 属性を使用します。

表 C-5: MPLAB C18 Pragma と MPLAB C30 属性

Pragma (MPLAB C18)	属性 (MPLAB C30)
#pragma udata [*name*]	__attribute__ ((section ("*name*")))
#pragma idata [*name*]	__attribute__ ((section ("*name*")))
#pragma romdata [*name*]	__attribute__ ((space (prog)))
#pragma code [*name*]	__attribute__ ((section ("*name*"))), __attribute__ ((space (prog)))
#pragma interruptlow	__attribute__ ((interrupt))
#pragma interrupt	__attribute__ ((interrupt, shadow))
#pragma varlocate *bank*	NA*
#pragma varlocate *name*	NA*

*dsPIC デバイスはバンクを持ちません。

例 C-4: データメモリ内のユーザセクションにある初期化されない変数の指定

```
C18:  #pragma udata mybss
      int gi;
C30:  int __attribute__((__section__(".mybss"))) gi;
```

例 C-5: データメモリ内のアドレス 0X100 に変数 MABONGA を置く

```
C18:  #pragma idata myDataSection=0x100;
      int Mabonga = 1;
C30:  int __attribute__((address(0x100))) Mabonga = 1;
```

例 C-6:　　プログラムメモリに置かれる変数を指定する

```
C18:  #pragma romdata const_table
      const rom char my_const_array[10] =
        {0,1,2,3,4,5,6,7,8,9};
C30:  const __attribute__((space(const)))
      char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};
```

注 : MPLAB C30 コンパイラは、プログラム空間の変数をアクセスすることを、直接にはサポートしていません。そのように割り当てられた変数は、通常はテーブルアクセスインラインアセンブリ命令を用いるかもしくはプログラム空間可視化（PSV）ウインドウを用いて、プログラマでが明確にアクセスしなければなりません、PSV ウインドウの詳細については、**セクション 4.15「Program Space Visibility (PSV) の使用方法」**をご参照ください。

例 C-7:　　関数 PRINTSTRING をプログラムメモリのアドレス 0X8000 に配置する

```
C18:  #pragma code myTextSection=0x8000;
      int PrintString(const char *s){...};
C30:  int __attribute__((address(0x8000))) PrintString
        (const char *s) {...};
```

例 C-8:　　コンパイラが変数 VAR1 と VAR2 を自動的に保存・回復する

```
C18:  #pragma interrupt isr0 save=var1, var2
      void isr0(void)
      {
      /* perform interrupt function here */
      }
C30:  void __attribute__((__interrupt__(__save__(var1,var2))))
      isr0(void)
      {
      /* perform interrupt function here */
      }
```

C.14　メモリモデル

MPLAB C18 は非 ANSI のスモールとラージのメモリモデルを使用します。スモールは 16- ビットのポインタを使用し、プログラムメモリを 64 KB (32 KB ワード) 以下に制限します。

MPLAB C30 は非 ANSI のスモールコードとラージコードのモデルを使用します。スモールコードはプログラムメモリを 96 KB (32 KB ワード) 以下に制限します。ラージコードでは、ポインタはジャンプテーブル内を検索します。

C.15　コール規則

MPLAB C18 と MPLAB C30 のコール規則については、多くの差があります。
MPLAB C30 のコール規則については、**セクション 4.12「関数コール集合」**をご参照ください。

C.16 スタートアップコード

MPLAB C18 は 3 つのスタートアップルーチンを持ちます。1 つはユーザデータ初期化を行わないもの、1 つはイニシャライザを持つ変数のみを初期化するもの、もう 1 つはすべての変数を初期化するもの（イニシャライザを持たないものは ANSI 標準で要求されるようにゼロに設定します）です。

MPLAB C30 は 2 つのスタートアップルーチンを持ちます。1 つはユーザデータ初期化を行わないもの、1 つは、永続データセクション内の変数以外のすべての変数を初期化するもの（イニシャライザを持たないものは ANSI 標準で要求されるようにゼロに設定します）です。

C.17 コンパイラで管理されるリソース

MPLAB C18 は以下のような管理されるリソースを持ちます：PC、WREG、STATUS、PROD セクション .tmpdata、セクション MATHDATA、FSRO、FSR1、FSR2、TBLPTR、TABLAT。

MPLAB C30 は以下のような管理されるリソースを持ちます：W0-W15、RCOUNT、SR。

C.18 最適化

以下の最適化がそれぞれのコンパイラの一部です。

MPLAB C18	MPLAB C30
分岐 (-Ob+) コード整頓化 (-Os+) Tail Merging(-Ot+) 未到達コードの除去 (-Ou+) コピー伝達 (-Op+) 冗長ストアの除去 (-Or+) 無効コードの除去 (-Od+)	最適化設定 (-On は n is 1, 2, 3 もしくは s)(1)
文字列マージンの複製 (-Om+)	-fwritable-strings
バンキング (-On+)	なし – バンキングは未使用
WREG コンテンツトラッキング (-Ow+)	すべてのレジスタは自動的にトラックされます
手続きの抽象化 (-Opa+)	手続きの抽象化 (-mpa)

注　1: MPLAB C30 では、これらの最適化設定はほとんどの要求を満たします。詳細調整“fine-tuning”用に追加のフラグが使用できます。詳細については、**セクション 3.5.6「最適化を制御するオプション」**をご参照ください。

C.19 オブジェクトモジュールフォーマット

MPLAB C18 と MPLAB C30 は、交換できない異なる COFF ファイルフォーマットを使用します。

C.20 インプリメンテーション定義された動作

負の符号がついた整数値の右シフトに関して、

- MPLAB C18 は符号ビットを保持しません。
- MPLAB C30 は符号ビットを保持します。

C.21 ビットフィールド

MPLAB C18 のビットフィールドはバイトストレージ境界を越えることはできません。従って、サイズが 8 ビットより大きな値をとることができません。

MPLAB C30 はどんなビットサイズでも、その基礎をなすタイプのサイズまでサポートします。どんな整数タイプもビットフィールドにできます。割り当ては、その基礎をなすタイプに備わったビット境界を越えることはできません。

例 :

```
struct foo {
  long long i:40;
  int j:16;
  char k:8;
} x;

struct bar {
  long long I:40;
  char J:8;
  int K:16;
} y;
```

`struct foo` は、MPLAB C30 を用いると、10 バイトのサイズを持ちます。`i` はビットオフセット 0 (から 39 まで) に割り当てられます。`j` の前に 8 ビットのパディングがあり、`j` はビットオフセット 48 に割り当てられます。もし、`j` が次に利用できるビット 40 に割り当てられるなら、16 ビット整数のストレージ境界を越えることになります。`j` のあとのビットオフセット 64 に、`k` が割り当てられます。構造体は、配列の場合でも、必要な境界整列を保持するために、8 ビットのパディングを含みます。構造体内の一番大きい境界整列は 2 バイトなので、境界整列は 2 バイトになります。

`struct bar` は MPLAB C30 を用いると、8 バイトのサイズを持ちます。`I` はビットオフセット 0 (から 39 まで) に割り当てられます。`J` は、`char` のストレージ境界を越えず、ビットオフセット 40 に割り当てられるので、パッドする必要はありません。K はビットオフセット 48 から割り当てられ、空間を浪費することなく、構造体を満たして生きます。

注意：

別紙 D ASCII 文字セット

表 D-1: ASCII 文字セット

下位 \ 上位	0	1	2	3	4	5	6	7
0	NUL	DLE	Space	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

注意：

別紙 E GNU 無料ドキュメントライセンス

GNU 無料ドキュメントライセンス
バージョン 1.2 2002 11 月

0. 前書き

このライセンスの目的は、マニュアルやテキスト、その他機能的および役に立つドキュメントを、自由と言う意味において自由に作成するために、修正して、もしくは修正しないで、商用目的であれ非商用目的であれ、誰でもコピーしたり再配布したりすることの自由を保証するものです。このライセンスは、著者や出版者のために、その作品に対する信用を得るための方法を保持するものであり、他人によりなされた修正に責任を持つとは見なされません。

このライセンスは、“コピーレフト”であり、ドキュメントの派生物はそれ自体が同様に無料でなければなりません。それは、GNU General Public License を補足するものであり、無料ソフトウエア用に設計されたコピーレフトライセンスです。

このライセンスは、無料ソフトウエア用のマニュアルとして用いるために設計されています。無料ソフトウエアには無料のドキュメントが必要であり、無料のプログラムは、ソフトウエアが与えるのと同じ自由を与えるマニュアルと一緒に供給されねばならないからです。しかし、このドキュメントは、ソフトウエアマニュアルだけには限定されず、主題となる内容もしくは印刷された本として出版されるか否かにかかわらず、原文に関するどんな作品にも用いられます。このライセンスを、主として、教育的もしくは参考的な目的の業務用として推奨します。

1. 適用範囲と定義

このライセンスは、どんなメディアであれ、著作権保有者による、このライセンス条項のもとで配布される、という注意書きを含むどんなマニュアルやその他の作品にも適用されます。そのような注意書きは、全世界で、ロイヤリティ無しのライセンスを、期限無しで、ここに述べられる条件のもとで作品を使用する権利を与えます。以下に示す“ドキュメント”は、そのようなマニュアルもしくは作品を示します。どんな人でもライセンスを受けることができ、“あなた”として述べられます。著作権法のもと、許可を得たうえで、コピーしたり、修正したり、作品を配布する場合、このライセンスを受けることができます。

ドキュメントの“修正版”は、ドキュメントもしくはその一部を含むあらゆる著作物を意味し、逐語複写されたものもしくは、修正および / もしくは他の言語に翻訳されたものどちらかです。

“第二のセクション”は名前を付けられた付録もしくはドキュメントの表書セクションで、ドキュメント全体の主題（もしくは関連事項に関して）出版者と著者の関係を専門に扱う部分であり、全体の主題に直接関わるものではありません。（従って、もしドキュメントが数学の教科書なら、第二のセクションは数学の説明をするものではありません。）関係とは、主題もしくは関連する項目との歴史的繋がりのある事柄、もしくは、それらに関する法律的、商業的、哲学的、倫理的、もしくは政治的地位の事柄である場合もあります。

“不変のセクション”は、ある種の第二のセクションで、そのタイトルが不変のセクションであり、ドキュメントがこのライセンスのもとに発行されていると述べた注意が記載されています。もしセクションが、上記 Secondary の定義に適合しなければ、不変のと記載するのは許されません。ドキュメントによっては、不変のセクションの無いものもあります。もしドキュメントが不変のセクションを認識しないのであれば、それは無いからです。
“カバーテキスト”は、表書（Front-Cover Texts）もしくはあとがき（Back-Cover Texts）と、リストアップされた短い文章で、ドキュメントがこのライセンスのもとに発行されていると述べた注意が記載されています。表書は多くても 5 語、あとがきは多くても 25 語です。
ドキュメントの“透明な”コピーとは、マシンで読み込むことのできるコピーを意味し、その表現フォーマットは一般大衆に利用できるものであり、基本的なテキストエディタもしくは（ピクセルで構成された画像用に）基本的なペイントプログラムもしくは（描画用の）広く利用可能な描画エディタで直接的に改訂するのに適しており、テキストフォーマッタへの入力に適したもの、もしくはテキストフォーマッタへの入力に適する種々のフォーマットへの自動変換に適したものです。別の透明なファイルフォーマットで作られたコピーで、そのマークアップが（もしくはマークアップのないものもあるが）読み取り器による引続き行われる修正を阻止するようなコピーは、透明ではありません。画像フォーマットは、たくさんの量のテキスト用に使用されるとしたら、透明ではありません。透明でないコピーは“不透明”と呼びます。
透明なコピー用の適切なフォーマットの例は、マークアップを持たない普通の ASCII, Texinfo 入力フォーマット、LaTeX 入力フォーマット、一般に利用可能な DTD を用いた SGML もしくは XML, および、人手で修正可能なように設計された、標準に準拠した簡単な HTML, PostScript もしくは PDF です。透明な画像フォーマットには、PNG、XCF および JPG があります。不透明なフォーマットには、独自のワードプロセッサによってのみ読んだり編集したりすることのできる独自のフォーマットを持ったもので、DTD および / もしくはプロセッシングツールが一般的に利用できない SGML もしくは XML、および、出力目的専用のワードプロセッサで生成された、HTML、PostScript もしくは PDF があります。

“タイトルページ”は、印刷された著作物用として、タイトルページ自体、プラス、タイトルページに明記することをライセンスが必要としているものをはっきりと保持するために必要なページを意味します。そのようなタイトルページを持たないフォーマットの作品では、“タイトルページ”は、文章本体の始まりより以前にあり、作品のタイトルが最も大きく表示される場所に近いテキストを意味します。

セクション“XYZ と題をつけられた”は、タイトルがまさしく XYZ もしくは、別の言語で XYZ と訳されるテキストに引続く括弧内に XYZ を含むようなタイトルを持つドキュメントのサブユニットを意味します。（ここで、XYZ は以下に示すような特定のセクションを意味し、それらは謝辞 , 献呈の辞 , 裏書 , もしくは履歴を意味します。） ドキュメントを修正する際にそのようなセクションのタイトルを保存すること（“Preserve the Title”）は、この定義に従って、“XYZ と題をつけられた”セクションを保持することを意味します。
ドキュメントは、このライセンスをそのドキュメントに適用するという注意書きの次に免責事項を含む場合もあります。この免責事項はこのライセンスのリファレンスによって含まれると見なされますが、責任放棄に関してのみであり、これらの免責事項が持つであろうその他の暗示については無効であり、このライセンスの意味には影響を与えません。

2. 逐語的複製

このライセンス、著作権の注意書き、および、ライセンスがドキュメントに適用されることを述べる注意書きがすべての複製物で再製造されるのであれば、またこのライセンスの条件に他のどんな条件をも追加しないのであれば、ドキュメントを、どのようなメディアにおいてでも、商用であれ非商用であれ、複製し配布することができます。複製し配布する複製物を読んだりもしくは更なる複製をすることを、妨害したりコントロールしたりする技術的な手段を用いてはなりません。しかしながら、複製物と引き換えに補償を受けることができます。もし大量の複製物を配布するのであれば、第 3 章の条件に従わねばなりません。

複製物を、上記で述べられているのと同じ条件で、貸与したり、公衆に展示することもできます。

3. 大量の複製

ドキュメントの印刷された複製物（もしくは印刷されたカバーを共通に持つメディアに複製されたもの）を 100 部以上出版し、ドキュメントのライセンス注意書きがカバーテキストを必要とするならば、すべてのカバーテキスト、表紙の表書きや裏表紙の後書きにはっきりと、そのライセンス注意書きを含めなければなりません。両方のカバーははっきりとあなたがそれらの複製物の出版者であることが認識できるようにしなければなりません。その他の記事をカバーに追加することもできます。カバーに限り修正を加えて複製を行うことは、それがドキュメントのタイトルを保持しこれらの条件を満たす限り、別の面から見ても逐語的な複製として扱われます。

もしどちらかのカバー用に要求される文が、あまりにも多くて明確性に適合しないなら、最初の文を実際のカバーに（合理的に適合するようにできるだけ多く）リストアップし、残りを近くのページに続けるようにしなければなりません。

もしドキュメントの不透明な複製物を 100 部以上出版もしくは配布するのであれば、それぞれの不透明な複製物にマシンで読める透明なコピーを一緒に含めるか、それぞれの不透明な複製物内もしくはそれに添えて、一般のネットワークを使用する大衆が、公衆の標準であるネットワークプロトコルを用いて、ドキュメントの完全に透明な複製物を、添付物無しに、ダウンロードするためにアクセスするロケーションを記述しなければなりません。後者のオプションを使用する場合は、多くの不透明な複製物の配布を開始する時は、この透明な複製物が、記述されたロケーションでアクセスできる状態を、公衆向けの版の不透明な複製物を（直接もしくは代理店もしくは小売店を経由して）最後に配布した後すくなくとも 1 年は保持することを保証するという、合理的に注意深い手順を取らねばなりません。

強制ではありませんが、ドキュメントの著者には、大量の複製物を再配布する場合は十分な時間をみて事前にコンタクトし、ドキュメントの最新版を供給する了承を得るべきです。

4. 修正

上記第２章、第３章の条件のもと、このライセンスに忠実に従い、ドキュメントの役割を満たし、従って、修正版の複製物を所有する誰に対しても、修正版の配布と修正する権利をライセンスすることのできる修正版を発行することを条件に、ドキュメントの修正版を複製し配布することができます。さらに、修正版に関して、以下のことをしなければなりません。

a) タイトルページ（およびもしあるならカバー）に、ドキュメントのタイトルから、および以前の版（もしあるなら、ドキュメントの履歴セクションにリストアップされているような）のタイトルとは異なったタイトルを使用します。もしオリジナルの出版者が許可を与えるのなら、以前の版と同じタイトルを使用できます。

b) タイトルページに、著者として、修正版の中で修正の権利に責任のある 1 人もしくはそれ以上の人物もしくは実体のあるものを、（もしリストアップするという要求が著者たちにより免除されない限り）ドキュメントの主な著者のうち少なくとも 5 人（5 人より少なければすべての著者）と一緒にリストアップします。

c) タイトルページに、出版者として、修正版の出版者の名前を述べます。

d) ドキュメントのすべての著作権を保持します。

e) その他の著作権注意書きに近接する、あなたの修正に対する適切な著作権を追加します。

f) 著作権注意書きの直後に、以下の補遺に示される形式で、このライセンスの条項において修正版を使用する公の許可をライセンスする注意書きを含みます。

g) そのライセンス注意書きには、不変のセクションおよび、ドキュメントのライセンス注意書きにあるような要求されるカバーテキストのすべてのリストを保持します。

h) このライセンスの改変されていない複製を含みます。

i) “履歴”と題されたセクションを保持し、そのタイトルを保持し、タイトルページで与えられる修正版の、少なくともタイトル、年、新しい著者および出版者を述べた項目を、それに追加します。もしドキュメントに“履歴”と題されたセクションがない場合は、タイトルページで与えられる修正版の、少なくともタイトル、年、新しい著者および出版者を述べた項目を作成し、以前の文章で述べられたように、修正版を説明している項目を追加します。

j) もしあれば、ドキュメント内で与えられ、ドキュメントの透明な複製に公的にアクセスするためのネットワークロケーション、および、同様にドキュメント内で与えられる、そのドキュメントの元になっている、以前の版用のネットワークロケーションを保持します。これらは、“履歴”セクションに置かれています。ドキュメント自体より少なくとも 4 年前に出版された作品のネットワークロケーション、もしくは、参照している原出版者の許可があれば、ネットワークロケーションは省略できます。

k) “謝辞”もしくは“献呈”と題されたセクション用に、セクションのタイトルを保持し、ここで与えられる、貢献者に対する謝辞および / もしくは献呈のすべての中身や論調を、セクション内に保持します。

l) その文やタイトル内で変更されない、ドキュメントのすべての不変のセクションを保持します。セクション番号もしくはそれに相当するものは、セクションタイトルの一部だとはみなされません。

m) “裏書”と題されたセクションを消去します。そのようなセクションは、修正版には含まれません。

n) 存在するセクションに再タイトル化して、“裏書”と題されるようにするとか、もしくは不変のセクションを持ったタイトルと競合するようにしてはいけません。

o) 免責事項を保持します。

もし修正版が新しい前付セクションもしくは付録を含みそれが、第二のセクションとして認められ、ドキュメントからのマテリアル複製を含まないなら、あなたの判断でそれらのいくつかもしくはすべてのセクションを不変として指名できます。これを行うには、修正版のライセンス通知の中の不変セクションのリストに、そのタイトルを追加します。これらのタイトルは、その他のセクションタイトルとは異なるものでなければなりません。

もし“裏書”と題されたセクションが、種々の団体による修正版の裏書のみを含んでいる場合、例えば、よく見たレビュー文もしくは文が標準の権威ある定義としての機関に承認されているという文言であれば、裏書を追加できます。

最大 5 文字までの 1 節をフロントカバーテキストとして、また最大 25 文字までの 1 節をバックカバーテキストとして、修正版の中のカバーテキストのリストの終わりに追加できます。1 つの団体によって（もしくはアレンジされることにより）フロントカバーテキストの 1 つの節および、バックカバーテキストの 1 つの節のみを追加できます。もしドキュメントが、同じカバーにカバーテキストをすでに含んでいる場合、以前あなたによって追加されたかもしくは、あなたが、その団体のために行動しているの団体によって準備された場合は、さらなる追加はできませんが、古いものを追加した以前の出版者からの明確な許可があれば、古いものと置き換えることができます。

ドキュメントの著者と出版者は、そのライセンスにより、彼らの名前を宣伝に用いたり、修正版の裏書を明言もしくは暗示するしたりする許可を与えてはなりません。

5. ドキュメントを結合する

ドキュメントを、すべてのオリジナルドキュメントのすべての不変のセクションを、修正無しで含め、ライセンス通知の中に、結合された作品の不変のセクションをリストアップし、すべての免責事項を保持すれば、このライセンスのもと（修正版用としては、上の第 4 章で定義された条項のもと）で発行された他のドキュメントと一緒に結合することができます。

結合された作品は、このライセンスの複製を 1 つだけ含めばよく、複数の同じ不変のセクションは、1 つの複製で置き換えられます。もし同じ名前だが異なる内容を持った複数の不変のセクションがあるならば、その終わりの部分に、もし分かるようであれば、セクションのオリジナル著者もしくは出版者を、もしくは他の一意の番号を、括弧の中に入れて追加することで 1 つ 1 つのセクションのタイトルを一意にします。同様な調整を、結合された作品のライセンス通知の中の不変のセクションのリスト内のセクションタイトルに行います。

結合にあたっては、種々のオリジナルドキュメント内の“履歴”と題されたセクションを結合して、1 つの“履歴”と題されたセクションにしなければなりません。同様に、“謝辞”や“献呈”と題されたセクションについても、1 つのセクションにしなければなりません。“裏書”と題されたセクションはすべて消去しなければなりません。

6. ドキュメントの収集

もし他のすべての面において、ドキュメントの 1 つ 1 つを逐語的に複製するために、このライセンスの規則に従うのであれば、ドキュメントと、このライセンスのもとで発行された他のドキュメントから成るコレクションを生成し、種々のドキュメント内にあるこのライセンスの個々の複製を、コレクションに含まれる 1 つの複製で置き換えることができます。

もし、抽出されたドキュメントにこのライセンスの複製を挿入し、そのドキュメントを逐語的に複製することに関するすべてのその他の面においてこのライセンスに従うのであれば、そのようなコレクションから 1 つのドキュメントを抽出し、このライセンスのもとで個別に配布できます。

7. 独立作品の集合

もし結合の結果としての著作権が、個々の著作物が許可する範囲を超えて結合物のユーザの法律上の権利を制限するために使用されないのであれば、ドキュメントもしくはその派生物をその他の分けられた独立のドキュメントもしくは作品と、ストレージもしくは配布メディアのボリューム内もしくはボリューム上で、結合することを、“集合”と呼びます。

もし、第 3 章のカバーテキストで要求される物がドキュメントの複製物に適用されるのであれば、ドキュメントが全体の集合の半分より小さいのであれば、ドキュメントのカバーテキストは、集合内でドキュメントを括弧でくくるカバー上、もしくはもしドキュメントが電子形態であれば電子的にカバーに該当する場所に置かれます。そうでなければ、集合全体を括弧でくくった印刷されたカバーの上に表示されなければなりません。

8. 翻訳

翻訳は修正と見なされますので、第 4 章の条項のもとで、ドキュメントの翻訳物を配布することができます。不変のセクションを翻訳に置き換えるには、著作権所有者からの特別な許可を必要としますが、それらの不変のセクションのオリジナル版に追加して、いくつかのもしくはすべての不変のセクションの翻訳を含めることができます。もし、このライセンスのオリジナルの英語版およびそれらの通知や放棄のオリジナル版を含めるならば、このライセンスの翻訳、ドキュメント内のすべてのライセンス通知およびすべての免責事項をを含めることができます。このライセンスもしくは通知もしくは放棄の翻訳とオリジナル版の間に不一致がある場合は、オリジナル版が優先されます。

このドキュメント内のセクションが“謝辞”“献呈”もしくは“履歴”と題されたものであるなら、そのタイトル（第 1 章）を保持するという要求（第 4 章）は、典型的に実際のタイトルを変更することを要求します。

9. 失効

このライセンスではっきりと与えられていない場合、ドキュメントを複製したり修正したりサブライセンスを与えることはできません。ドキュメントを複製したり修正したりサブライセンスを与えるということ以外の試み・行為は、無効であり、このライセンスのもとで自動的に権利を失効させます。しかし、このライセンスのもとで、あなたから複製や権利を受けた当事者は、ライセンスに忠実に従い続ける限りは、ライセンスを失効されることはありません。

10. このライセンスの将来の改定

Free Software Foundation は、時により、GNU 無料ドキュメントライセンスの新しい改訂版を出版します。そのような新版は、意図は現行版に似ていますが、新しい問題もしくは事柄を述べているもので、詳細は異なります。http://www.gnu.org/copyleft/ を参照下さい。

ライセンスのそれぞれの版は、区別するための版番号を与えられています。もしドキュメントがこのライセンスの特定の番号の版もしくはより後の版が適用されることを指定しているとすると、Free Software Foundation により（草案としてではなく）出版されている、その指定された版もしくはより後の版のどちらかの条項と条件に従うオプションを選択します。もしドキュメントがこのライセンスの版番号を指定しないのであれば、これまで Free Software Foundation により（草案としてではなく）出版されたどの版でも選択できます。

語彙集

Absolute Section

固定（絶対）アドレスを持つセクションで、リンカーによる変更はできません。

Access Memory (PIC18 Only)

バンク選択レジスタ (BSR) の設定に関わらずアクセスできる PIC18XXXXX デバイスの特殊レジスタ。

Address

メモリ内の位置を示す値。

Alphabetic Character

アルファベット文字とは、アラビア文字のことです。(a, b, ...,z, A, B, ...,Z)。

Alphanumeric

アルファニューメリック文字は、アルファベット文字と 10 進数 (0,1,...,9) から構成されます。

ANSI

米国規格協会（American National Standards Institute）は合衆国内での標準を作成・承認することに責任を持つ機関です。

Application

ソフトウェアとハードウェアのセットで、PICmicro マイクロコントローラにより制御されます。

Archive

再配置可能なオブジェクトモジュールの集合。複数のソースファイルをオブジェクトファイルにアセンブルし、アーカイバを用いてオブジェクトファイルを 1 つのライブラリファイルに束ねることで生成されます。ライブラリはオブジェクトモジュールと他のライブラリとリンクされ、実行可能コードを生成します。

Archiver

ライブラリを生成・操作する道具。

ASCII

情報交換用米国標準コード（American Standard Code for Information Interchange）は、それぞれの文字を表示するために 7 バイナリビットを用いてコード化された文字のことです。大文字、小文字、アラビア数字、シンボルおよび制御文字を含みます。

Assembler

アセンブリ言語ソースコードを機械コードに変換する言語ツールです。

Assembly Language

バイナリ機械コードをシンボリック形式で記述するプログラム言語です。

Asynchronous Stimulus

刺激外部入力がシミュレータデバイスになされる際に生成されるデータ。

Breakpoint, Hardware

その実行により停止を引き起こすイベント。

Breakpoint, Software

ファームウェアの実行が停止するアドレス。通常、特殊ブレーク命令により達成されます。

Build

アプリケーションのすべてのソースファイルをコンパイルしリンクすること。

C

記述が少なく、現代のコントロールフローとデータ構造、豊富なオペレータセットを特徴とする、汎用プログラム言語。

Calibration Memory

PICmicro マイクロコントローラ内蔵の RC オシレータまたはその他デバイス周辺機器を較正するのに使用する特殊な関数レジスタまたはレジスタ。

COFF

Common Object File Format。このフォーマットのオブジェクトファイルは機械コード、デバッグおよびその他の情報を含みます。

Command Line Interface

テキストの入力・出力に基づいた、プログラムとそのユーザーとの間の通信手段。

Compiler

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。

Configuration Bits

PICmicro マイクロコントローラのオペレーションモードを設定するためにプログラムされた特殊な目的を持つビット。コンフィギュレーションビットは事前にプログラムされる場合とされない場合があります。

Control Directives

アセンブリ言語コード内のディレクティブ。指定した表現によりアセンブル時にコードがインクルードされたり削除されることがあります。

Cross Reference File

記号の表と、記号を参照するファイルのリストを参照するファイル。記号が定義されている場合、最初にリストされるファイルは定義された場所を示すものです。残りのファイルは記号への参照を含みます。

Data Directives

データディレクティブはプログラムまたはデータメモリのアセンブラの割り当てを制御し、記号により（つまり意味を持った名前により）データ項目を参照する手段を提供します。

Data Memory

Microchip の MCU と DSC デバイスでは、データメモリ (RAM) は、汎用レジスタ (GPRs) と特殊関数レジスタ (SFRs) から成ります。いくつかのデバイスでは、EEPROM データメモリも持っています。

Device Programmer

マイクロコントローラのような電気的にプログラム可能な半導体デバイスをプログラムするのに使われるツールです。

Directives

言語ツールの動作の制御を行う、ソースコード内のステートメント。

Download

ダウンロードとは、ホストからエミュレータ、プログラマ、ターゲットボードなどの他のデバイスへデータを送信するプロセスです。

EEPROM

電気的に消去可能なプログラム読み取り専用メモリ。電気的に消去可能な特殊なタイプの PROM です。データは 1 度に 1 バイト単位で書き込み • 消去されます。EEPROM は電源が切れている間もその内容を保持します。

Emulation

エミュレーションメモリにロードされたソフトウェアの実行プロセス。マイクロコントローラデバイスに常駐するファームウェアのような動作です。

Emulation Memory

エミュレータに含まれるプログラムメモリ。

Emulator

エミュレーションを実行するハードウェア。

Emulator System

MPLAB ICE 2000 および 4000 のエミュレーションシステムは pod、プロセッサモジュール、デバイスアダプタ、ケーブル、MPLAB IDE ソフトウェアを含みます。

EPROM

消去・プログラム可能読み取り専用メモリ（Erasable Programmable Read Only Memory）。通常、紫外線照射により消去可能なプログラム可能読み取り専用メモリです。

Event

アドレス、データ、パスカウント、外部入力、サイクルタイム（fetch、R/W）、タイムスタンプを含むバスサイクルの説明。イベントはトリガー、ブレークポイント、割り込みを説明するのに使用されます。

Export

標準フォーマットで MPLAB IDE からのデータを送信すること。

Extended Microcontroller Mode

拡張マイクロコントローラモードでは、チップ上のプログラムメモリと外部メモリが使用できます。このモードを実行すると、PIC17CXXX または PIC18CXXX デバイスのプログラムメモリアドレスが内部メモリ空間よりも大きくなった場合に、自動的に外部にスイッチされます。

External Label

外部へのリンクを持つラベル。

External Linkage

定義済みのモジュールの外部から参照可能なリンクがある場合、関数または変数は外部リンクを持ちます。

External Symbol

外部リンクを持つ識別子の記号。参照または定義になります。

External Symbol Resolution

リンカーにより実行されるプロセス。すべての入力モジュールからの外部記号定義がすべての外部記号参照を解決するために収集されます。対応する定義をもたない外部記号参照はリンカーのエラーを引き起こし、そのエラーは報告されます。

External Input Line

外部シグナル基づいたイベントを設定するための外部入力シグナル論理プローブ行（TRIGIN）。

External RAM

チップ内の読み取り / 書き込みメモリ。

File Registers

チップに内蔵されたデータメモリで、汎用レジスタ (GPRs) と特殊関数レジスタ (SFRs) を含みます。

Flash

バイト単位ではなく、ブロック単位でデータの書き込み・消去を行なう EEPROM の一種。

FNOP

Forced No Operation。強制 NOP サイクルは 2 サイクルの命令の 2 番目のサイクルです。PICmicro マイクロコントローラのアーキテクチャはパイプラインされているため、物理アドレス空間内の次の命令を事前に取得します。一方で、現在の命令も実行します。ただし、現在の命令がプログラムカウンタを変更した場合、この事前に取得した命令は無視され、強制 NOP サイクルが発生します。

GPR

汎用レジスタ（General Purpose Register）。デバイスのデータメモリ (RAM) の一部で、汎用に用いることができます。

Halt

プログラム実行の停止。停止の実行はブレークポイントでの停止と同じです。

Hex Code

16 進のフォーマットコードでストアされる実行可能命令。Hex コードは Hex ファイルに含まれます。

Hex File

デバイスをプログラムするのに適した、16 進のアドレスや値 (Hex コード) を含んだ ASCII ファイル。

High Level Language

アセンブリよりもさらにプロセッサから離れた、プログラムを作成する言語です。

ICD

インサーキットデバッガ。MPLAB ICD および MPLAB ICD 2 は、それぞれ、PIC16F87X、PIC18FXXXMicrochip デバイス向けのインサーキットデバッガです。これらの ICD は MPLAB IDE とともに動作します。

ICE

インサーキットエミュレータ。MPLAB ICE 2000 および 4000 は MPLAB IDE とともに動作する Microchip のインサーキットエミュレータです。

IDE

統合開発環境（Integrated Development Environment）。MPLAB IDE は Microchip の統合された開発環境です。

Import

データを hex ファイルなどの外部ソースから MPLAB IDE に取り入れること。

Instruction Set

特定のプロセッサが理解する機械語命令の集合。

Instructions

中央処理装置に特定の動作を行うよう指示し、その動作に用いられるデータを含むことができるビットのシーケンス。

Internal Linkage

定義済みの外部モジュールからアクセスできない場合に関数または変数はインターネットリンクを持ちます。

International Organization for Standardization

コンピューティングや通信を含め、多くの商業や技術における標準を設定する機関。

Interrupt

動作中のアプリケーションの実行を中断し、イベントが処理されるために割り込みサービスルーチン (ISR) に制御を渡すことを行う、CPU への信号。

Interrupt Handler

割り込みが発生した時に、特別のコードを処理するルーチン。

Interrupt Request

プロセッサが一時的に通常の命令実行を中断し、割り込みハンドラの実行を開始させるようなイベント。いくつかのプロセッサでは、異なる優先度の割り込みを許可する、いくつかの割り込みリクエストイベントを持ちます。

Interrupt Service Routine

割り込みが発生した際に挿入されるユーザー生成コード。プログラムメモリ内のコードの場所は、通常、発生した割り込みのタイプにより異なります。

IRQ

割り込み要求をご参照ください。

ISO

国際標準化機関（International Organization for Standardization）をご参照ください。

ISR

割り込みサービスルーチンをご参照ください。

Librarian

アーカイバをご参照ください。

Library

アーカイブをご参照ください。

Linker

オブジェクトファイルとライブラリを結合し実行可能コードを生成する言語ツールで、あるモジュールから別のモジュールへの参照を分解します。

Linker Script Files

リンカスクリプトファイルはリンカのコマンドファイル。リンカオプションを定義し、ターゲットプラットフォーム上の利用できるメモリを記述します。

Listing Directives

リストディレクティブはアセンブラリストファイルフォーマットを制御するディレクティブでス。タイトル、ページ付け、その他リスト制御に関する設定を可能にします。

Listing File

リストファイルは ASCII テキストファイルで、ソースファイル内の各 C ソーステートメント、アセンブリ命令、アセンブラディレクティブ、マクロのために生成されるマシンコードを示します。

Local Label

ローカルラベルは LOCAL ディレクティブによりマクロ内で定義されるラベルです。これらのラベルはマクロの例示の与えられたインスタンスに特別に対応するものです。つまり、ローカルとして宣言された記号とラベルは ENDM マクロ発生後にはアクセスできなくなります。

Logic Probes

最大 14 までの論理プローブを Microchip エミュレータと接続できます。論理プローブは外部追跡入力、トリガー出力シグナル、＋ 5V、共通グラウンドを提供します。

Machine Code

プロセッサにより実際に読み込まれ解釈されるコンピュータプログラムの表現。バイナリ機械語コードのプログラムは一連の機械語命令（データが挿入されることもある）で構成されます。特定のプロセッサ用のすべての可能性のある命令の集合は、“命令セット”として知られています。

Machine Language

特定の中央処理装置用の命令のセットで、翻訳無しにプロセッサで使用できるように設計されたもの。

Macro

マクロ命令のこと。短縮された形式の一連の命令を表現する命令。

Macro Directives

実行とデータ割り当てをマクロ本文定義内で制御するディレクティブ。

Make Project

アプリケーションの再ビルド、再コンパイルを行なうコマンド。最新の完全なコンパイル後に変更が生じたソースファイルでのみ実行されます。

MCU

マイクロコントローラユニットマイクロコントローラの略語です。uC ともいいます。

Message

言語ツールオペレーションの潜在的な問題を警告するために表示されるテキスト。メッセージが表示されてもオペレーションは停止しません。

Microcontroller

CPU、RAM、プログラムメモリ、I/O ポートおよびタイマを含む、高度集積チップ。

Microcontroller Mode

マイクロチップコントローラの PIC17CXXX および PIC18CXXX ファミリーの可能なプログラムメモリ構成の 1 つ。マイクロコントローラモードでは、内部実行のみが許可されます。従って、このモードではチップ上のプログラムメモリのみが使用可能です。

Microprocessor Mode

マイクロチップコントローラの PIC17CXXX および PIC18CXXX ファミリーの可能なプログラムメモリ構成の 1 つ。マイクロプロセッサモードでは、チップ上のプログラムメモリは使用されません。プログラムメモリ全体が外部にマップされます。

Mnemonics

直接機械語コードに翻訳できるテキスト命令。Op コードとも言われます。

MPASM Assembler

PICmicro マイクロコントローラデバイス、KEELOQ デバイス、マイクロチップメモリデバイス用のマイクロチップテクノロジー社のリロケータブルマクロアセンブラ。

MPLAB ASM30

マイクロチップ社の、dsPICS0F デジタルシグナルコントローラデバイス用のリロケータブルマクロアセンブラ。

MPLAB C1X

マイクロチップ社からの MPLAB C17 と MPLAB C18 C 両方を指します。MPLAB C17 は PIC17CXXX デバイス用の C コンパイラで、MPLAB C18 C は PIC18CXXX と PIC18FXXXX デバイス用の C コンパイラです。

MPLAB C30

dsPICSOF デジタルシグナルコントローラデバイス用のマイクロチップ社の C コンパイラー。

MPLAB ICD 2

PIC16F87X、PIC18FXXX、dsPICSOFXXXX デバイス用のマイクロチップ社のインサーキットデバッガ。ICD は MPLAB IDE で動作します。各 ICD の主なコンポーネントはモジュールです。完全なシステムは、モジュール、ヘッダー、デモボード、ケーブルおよび MPLAB IDE ソフトウェアを含みます。

MPLAB ICE 2000

MPLAB IDE とともに動作する PICmicro MCU 用のマイクロチップ社のインサーキットエミュレータ。

MPLAB ICE 4000

MPLAB IDE とともに動作する dsPIC DSC 用のマイクロチップ社のインサーキットエミュレータ。

MPLAB IDE

マイクロチップ社の集積化された統合開発環境です。

MPLAB LIB30

MPLAB LIB30 アーカイバ / ライブラリアンは、MPLAB ASM30 もしくは MPLAB C30 C コンパイラのどちらかを用いて生成される COFF オブジェクトモジュールとともに使用されるオブジェクトライブラリアンです。

MPLAB LINK30

MPLAB LINK30 は、マイクロチップ社の MPLAB ASM30 アセンブラとマイクロチップ MPLAB C30 C コンパイラ用のオブジェクトリンカです。

MPLAB SIM

PICmicro MCU デバイスのサポートのもとで MPLAB IDE とともに動作するマイクロチップ社のシミュレータ。

MPLAB SIM30

dsPIC DSC デバイスのサポートのもとで MPLAB IDE とともに動作するマイクロチップ社のシミュレータ。

MPLIB Object Librarian

MPLIB ライブラリアンは MPASM アセンブラ (mpasm または mpasmwin v2.0) または MPLAB C1XC コンパイラーにより作成される CFF オブジェクトモジュール使用のためのオブジェクトライブラリアンです。

MPLINK Object Linker

MPLINK リンカーはマイクロチップ MPASM アセンブラおよびマイクロチップ MPLAB C17 または C18 C コンパイラー用のオブジェクトリンカーです。また、MPLINK リンカーはマイクロチップ MPLIB ライブラリアンとともに使用することもできます。MPLINK リンカーは MPLAB IDE とともに使用するようデザインされていますが、常に一緒に使用する必要はありません。

MRU

最も最近使用されたの意味。MPLAB IDE のメインプルダウンメニューから選択して使用可能なファイルやウィンドウについて言及する場合に使用する表現です。

Nesting Depth

マクロが他のマクロをインクルードできる最大レベル。

Node

MPLAB IDE プロジェクトのコンポーネント。

Non Real-Time

ブレークポイントにあるプロセッサ、1 つの手順の説明、シミュレーションモード内で実行されている MPLAB IDE について言及する際に使用する表現です。

Non-Volatile Storage

電源がオフになっている間もその内容を保持するストレージデバイス。

NOP

オペレーションなし。プログラムカウンタを進める以外になんら効果を持たない命令です。

Object Code

アセンブラまたはコンパイラーにより生成されるマシンコード。

Object File

機械語コードとデバッグ情報もある場合もあるが、それらを含むファイル。すぐに実行可能かもしくは、完全な実行可能なプログラムを生成するには、他のオブジェクトファイル、例えばライブラリ、とリンクが必要な、再配置可能なファイルです。

Object File Directives

オブジェクトファイルを作成する際にのみ使用されるディレクティブ。

Off-Chip Memory

チップ外メモリは PIC17CXXX または PIC18CXXX デバイスのメモリ選択オプションで、ターゲットボード上に常駐するメモリまたはすべてのプログラムメモリがエミュレータにより提供される場所にあります。オプション > 開発モードでアクセスできるメモリタブでチップ外メモリ選択ダイアログボックスを開けます。

Opcodes

実行コード。ニーモニックを参照ください。

Operators

プラス符号‘+' やマイナス符号‘-' のようなシンボルで、十分に定義された表現で記述する際に使用されます。それぞれの演算子は評価の順序を決定するために指定された優先権を持ちます。

OTP

ワンタイムプログラマブル。窓付きパッケージではない EPROM デバイスです。EPROM はメモリを消去するのに紫外線が必要で、窓付きデバイスのみ消去可能です。

Pass Counter

イベント（特定のアドレスでの命令の実行など）発生ごとに値が増えるカウンタです。パスカウント値がゼロに達すると、イベントが発生します。パスカウンタをブレークや追跡論理、複合トリガーダイアログ内の連続するイベントに割り当てることもできます。

PC

パーソナルコンピュータまたはプログラムカウンタ。

PC Host

IBM または IBM と互換性のあるパーソナルコンピュータ。ウィンドウズオペレーティングシステムサポートのもと動作します。

PICmicro MCUs

PIC マイクロコントローラ (MCUs) はマイクロチップ社のマイクロコントローラファミリを指します。

PICSTART Plus

マイクロチップ社の開発デバイスプログラマ。プログラム 8-、14-、28-、40 ピン PICmicro マイクロコントローラ。MPLAB IDE ソフトウェアとともに使用する必要があります。

Pod, Emulator

エミュレーションメモリ、追跡メモリ、イベントタイマー、サイクルタイマー、追跡 / ブレーキングポイント論理を含む外部エミュレータボックス。

Power-on-Reset Emulation

ソフトウェアのリセットプロセス。データ RAM 領域にランダムに値を書き込み、最初の電源投入時に RAM 内の初期化されていない値をシミュレートします。

PRO MATE II

マイクロチップ社のデバイスプログラマ。すべての PICmicro マイクロコントローラとメモリおよび KEELOQ デバイスの大半がプログラムできます。MPLAB IDE 配下またはスタンドアロンで使用できます。

Program Counter

現在実行中の命令のアドレスを含む位置。

Program Memory

命令がストアされる、デバイス内のメモリ領域。また、ダウンロードされたターゲットアプリケーションファームウェアを含むエミュレータまたはシミュレータのメモリも指します。

Project

オブジェクトをビルドするためのソースファイルと命令のセットで、アプリケーションの実行可能なコード。

Prototype System

ユーザーのターゲットアプリケーションまたはターゲットボードを示す用語。

PWM Signals

パルス幅モデュレーションシグナル。特定の PICmicro MCU デバイスは PWM 周辺機器を持っています。

Qualifier

パスカウンタに使用されるか、複合トリガー内の他のオペレーションの前のイベントとして使用されるアドレスまたはアドレス範囲。

Radix

ベース、16 進法、10 進法の数字で、アドレスを指定するのに使用します。

RAM

ランダムアクセスメモリ（データメモリ）。情報にどんな順番でもアクセスできるようなメモリ。

Raw Data

セクションと関連づけられるコードまたはデータを表すバイナリ。

Real-Time

エミュレータまたは MPLAB ICD モードを停止状態から開放する時、プロセッサはリアルタイムモードで実行され、通常のチップと同じ動作をします。リアルタイムモードでは、MPLAB ICE のリアルタイム追跡バッファが有効化され、すべての選択したサイクルから定期的にキャプチャを行い、すべてのブレーク論理が有効化されます。MPLAB ICD のエミュレータでは、プロセッサはリアルタイムで実行され、有効なブレークポイントが停止を引き起こすまで、またはユーザーがエミュレータを停止するまで続きます。シミュレータのリアルタイムは、単純に、シミュレーションを実行するホスト CPU がマイクロコントローラ命令を可能な速く実行することを意味します。

Recursion

関数またはマクロが定義され自身を呼び出すことのできるコンセプトを意味します。再帰マクロを記述するには細心の注意を払う必要があります。再帰からの出口を用意しておかないと、無限ループにはまってしまいます。

ROM

リードオンリーメモリ（プログラムメモリ）。変更できないメモリ。

Run

エミュレータを停止から開放するコマンドで、アプリケーションコードを実行してリアルタイムで変更を行なったり、I/O に対応できるようにします。

SFR

特殊関数レジスタを参照下さい。

Shell

MPASM アセンブラシェルはマクロアセンブラにインターフェースの入力を促します。2 つの MPASM アセンブラシェルがあり、そのうち 1 つは DOS バージョン、もう 1 つは Windows バージョン用です。

Simulator

デバイスの動作をモデル化したソフトウエアプログラム。

Single Step

このコマンドはコードを順を追って実行し、1 度に 1 個の命令を実行します。各命令の後、MPLAB IDE はレジスタウィンドウ、ウォッチ変数、ステータス表示を更新し、命令実行をデバッグできます。また、シングルステップ C コンパイラーソースコードを実行することもできますが、1 つの命令を実行する代わりに MPLAB IDE は上位レベル C ステートメントにより生成されたすべてのアセンブリ レベル命令を実行します。

Skew

命令の実行に関する情報で、プロセッサバスに時間を追って表示されます。例えば、実行された Opcode は以前の命令の実行中の取り込みとしてバスに表示され、ソースデータアドレスとその値、取り込み先データアドレスは Opcode が実際に実行された時に表示されます。取り込み先のデータ値は次の命令が実行された時に表示されます。追跡バッファは 1 つのインスタンスでバス上にある情報をキャプチャします。従って、1 つの追跡バッファエントリは 3 つの命令の実行情報を含みます。1 つの情報からもう 1 つの命令実行サイクル情報へとキャプチャされたサイクルの数を skew と呼びます。

Skid

ハードウェアブレークポイントがプロセッサ停止に使用されると、1 つ以上の追加命令がプロセッサ停止前に実行されることがあります。意図されたブレークポイントの後に実行された追加の命令の数を skid と呼びます。

Source Code

その中に、プログラマによりコンピュータプログラムが書かれる形式。ソースコードは、翻訳されるか、機械語コードか、もしくは翻訳器により実行されるようないくつかの公式プログラム言語で書かれます。

Source File

ソースコードを含む ASCII テキストファイル。

Special Function Registers

I/O プロセッサ関数、I/O ステータス、タイマもしくはその他のモードや周辺機器を制御する専用のレジスタで、データメモリ (RAM) の一部。

Stack, Hardware

関数コールが実行された時に戻りアドレスが格納される PICmicro マイクロコントローラ内の場所。

Stack, Software

戻りアドレス、関数パラメータおよびローカル変数をストアするために、アプリケーションにより使用されるメモリ。このメモリは高級言語内でコードを開発する際にコンパイラにより管理されます。

Static RAM or SRAM

静的ランダムアクセスメモリ。ターゲットボード上で読み込み / 書き込み可能なプログラムメモリで、頻繁にリフレッシュを行なう必要がありません。

Status Bar

ステータスバーは MPLAB IDE ウィンドウの底部に位置し、現在の情報をカーソル位置、開発モードとデバイス、アクティブツールバーで示します。

Step Into

このコマンドは Sigle Step と同じです。Step Into（Step Over の逆）はサブルーチンの CALL 命令の後を続けて実行します。

Step Over

Step Over はサブルーチンを実行せずにコードのデバッグを可能にします。CALL 命令を Step Over すると、次のブレークポイントが CALL の後の命令に設定されます。何らかの理由でサブルーチンがエンドレスループにはまったり、適切な戻り値を返さない場合、次のブレークポイントには達することができません。Step Over コマンドは CALL 命令ハンドリングの部分を除いては Single Step と同じです。

Stimulus

スティミュレータへの入力、つまり外部シグナルへの刺激に応答するために生成されるデータ。多くの場合、データはテキストファイルのアクションのリストのフォームに入力されます。刺激には、非同期、同期（pin）、クロック、レジスタなどがあります。

Stopwatch

実行サイクルを測定するカウンタ。

Symbol

記号はプログラムに関する様々な事柄を説明する際に使用する汎用的なメカニズムです。記号には、関数名、変数名、セクション名、ファイル名、struct/enum/union タグ名などがあります。MPLAB IDE の記号として、主に変数名、関数名、アセンブリラベルが挙げられます。記号がリンクされた後の値はメモリ内の値となります。

System Window Control

システムウィンドウコントロールはウィンドウの左上角といくつかのダイアログに位置します。このコントロールをクリックすると、通常、「最小化」「最大化」「閉じる」の項目を持つメニューが開きます。

Target

ユーザーハードウェアをご参照ください。

Target Application

ターゲットボード上に常駐するソフトウェア。

Target Board

ターゲットアプリケーションを構成する回路とプログラマブルなデバイス。

Target Processor

ターゲットアプリケーションボード上にあるマイクロコントローラデバイス。

Template

ビルドし後でファイルに挿入するテキスト行。MPLAB Editor はテンプレートをテンプレートファイル内に格納します。

Tool Bar

アイコンの列またはカラム。クリックすると MPLAB IDE の機能が実行されます。

Trace

プログラム実行を記録するエミュレータまたはシミュレータの機能。エミュレータはプログラム実行を MLAB IDE の追跡ウィンドウにアップロードされる追跡バッファに記録します。

Trace Memory

追跡メモリはエミュレータに含まれます。追跡メモリは追跡バッファと呼ばれることもあります。

Trigger Output

トリガー出力はどのアドレスまたはアドレス範囲でも生成可能なエミュレータ出力シグナルを参照します。追跡およびブレークポイント設定とは独立しています。トリガー出力にはどのような数字でも設定できます。

Uninitialized Data

初期値を持たずに定義されるデータ。C では

```
int myVar;
```

は、初期化されないデータセクションに駐在する変数を定義します。

Upload

アップロード機能はデータをエミュレータやプログラマなどのツールからホスト PC に転送します。または、ターゲットボードからエミュレータに転送します。

Warning

デバイス、ソフトウェアファイル、装置などに物理的ダメージを与える状況を警告するアラート。

Watch Variable

ウォッチウィンドウでセッションをデバッグしている間にモニターする変数。

Watch Window

ウォッチウィンドウには各ブレークポイントで更新される watch 変数のリストを含みます。

Watchdog Timer

A timer on a PICmicro マイクロコントローラ上のタイマーで、選択可能な長さの時間の後プロセッサをリセットします。WDT は有効 / 無効の切り替えとコンフィギュレーションビットを使用した設定が可能です。

WDT

Watchdog タイマーをご参照ください。

索引

N

O

T

U

V

W

X

注意：

全世界の販売及びサービス拠点

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Alpharetta, GA
Tel: 770-640-0034
Fax: 770-640-0307

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

San Jose
Mountain View, CA
Tel: 650-215-1444
Fax: 650-961-0286

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8676-6200
Fax: 86-28-8676-6599

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-2229-0061
Fax: 91-80-2229-0062

India - New Delhi
Tel: 91-11-5160-8631
Fax: 91-11-5160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Gumi
Tel: 82-54-473-4301
Fax: 82-54-473-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Penang
Tel: 60-4-646-8870
Fax: 60-4-646-5086

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

10/31/05