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

MPLAB® XC32 C/C++ コンパイラ ユーザガイド

© 2015 Microchip Technology Inc.

Microchip 社製デバイスのコード保護機能に関して次の点にご注意ください。

- Microchip 社製品は、該当する Microchip 社データシートに記載の仕様を満たしています。
- Microchip 社では、通常の条件ならびに仕様に従って使用した場合、Microchip 社製品のセキュリティ レベルは、現在市場に流通している同種製品の中でも最も高度であると考えています。
- しかし、コード保護機能を解除するための不正かつ違法な方法が存在する事もまた事実です。弊社の理解ではこうした手法は、Microchip 社データシートにある動作仕様書以外の方法で Microchip 社製品を使用する事になります。このような行為は知的所有権の侵害に該当する可能性が非常に高いと言えます。
- Microchip 社は、コードの保全性に懸念を抱くお客様と連携し、対応策に取り組んでいきます。
- Microchip 社を含む全ての半導体メーカーで、自社のコードのセキュリティを完全に保証できる企業はありません。コード保護機能とは、Microchip 社が製品を「解読不能」として保証するものではありません。

コード保護機能は常に進歩しています。Microchip 社では、常に製品のコード保護機能の改善に取り組んでいます。Microchip 社のコード保護機能の侵害は、デジタル ミレニアム著作権法に違反します。そのような行為によってソフトウェアまたはその他の著

本書に記載されているデバイス アプリケーション等に関する情報は、ユーザの便宜のためにのみ提供されているものであり、更新によって無効とされる事があります。お客様のアプリケーションが仕様を満たす事を保証する責任は、お客様にあります。Microchip 社は、明示的、暗黙的、書面、口頭、法定のいずれであるかを問わず、本書に記載されている情報に関して、状態、品質、性能、商品性、特定目的への適合性をはじめとする、いかなる類の表明も保証も行いません。Microchip 社は、本書の情報およびその使用に起因する一切の責任を否認します。Microchip 社の明示的な書面による承認なしに、生命維持装置あるいは生命安全用途に Microchip 社の製品を使用する事は全て購入者のリスクとし、また購入者はこれによって発生したあらゆる損害、クレーム、訴訟、費用に関して、Microchip 社は擁護され、免責され、損害をうけない事に同意するものとします。暗黙的あるいは明示的を問わず、Microchip社が知的財産権を保有しているライセンスは一切譲渡されません。

商標

Microchip 社の名称と Microchip ロゴ、dsPIC、FlashFlex、KEELOQ、KEELOQ ロゴ、MPLAB、PIC、PICmicro、PICSTART、PIC^{32} ロゴ、rfPIC、SST、SST ロゴ、SuperFlash、UNI/O は、米国およびその他の国における Microchip Technology Incorporated の登録商標です。

FilterLab、Hampshire、HI-TECH C、Linear Active Thermistor、MTP、SEEVAL、Embedded Control Solutions Company は、米国における Microchip Technology Incorporated の登録商標です。

Silicon Storage Technology は、その他の国における Microchip Technology Incorporated の登録商標です。

Analog-for-the-Digital Age、Application Maestro、BodyCom、chipKIT、chipKIT ロゴ、CodeGuard、dsPICDEM、dsPICDEM.net、dsPICworks、dsSPEAK、ECAN、ECONOMONITOR、FanSense、HI-TIDE、In-Circuit Serial Programming、ICSP、Mindi、MiWi、MPASM、MPF、MPLAB 認証ロゴ、MPLIB、MPLINK、mTouch、Omniscient Code Generation、PICC、PICC-18、PICDEM、PICDEM.net、PICkit、PICtail、REAL ICE、rfLAB、Select Mode、SQI、Serial Quad I/O、Total Endurance、TSHARC、UniWinDriver、WiperLock、ZENA、Z-Scale は、米国およびその他の国における Microchip Technology Incorporated の登録商標です。

SQTP は、米国における Microchip Technology Incorporated のサービスマークです。

GestIC と ULPP は、その他の国における Microchip Technology Germany II GmbH & Co. & KG (Microchip Technology Incorporated の子会社) の登録商標です。

その他、本書に記載されている商標は各社に帰属します。

©2013, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-63276-672-4

QUALITY MANAGEMENT SYSTEM CERTIFIED BY DNV

== ISO/TS 16949 ==

Microchip 社では、Chandler および Tempe (アリゾナ州)、Gresham (オレゴン州) の本部、設計部およびウェハー製造工場そしてカリフォルニア州とインドのデザインセンターが ISO/TS-16949:2009 認証を取得しています。Microchip 社の品質システム プロセスおよび手順は、PIC® MCU および dsPIC® DSC、KEELOQ® コード ホッピング デバイス、シリアル EEPROM、マイクロペリフェラル、不揮発性メモリ、アナログ製品に採用されています。さらに、開発システムの設計と製造に関する Microchip 社の品質システムは ISO 9001:2000 認証を取得しています。

MICROCHIP

MPLAB® XC32 C/C++ コンパイラ ユーザガイド

目次

序章 9

本書の構成 10
表記規則 11
推奨参考資料 12

第 1 章 コンパイラの概要

1.1 はじめに 15
1.2 対応デバイス 15
1.3 MPLAB XC32 C/C++ コンパイラについて 15
1.4 コンパイラとその他の開発ツール 17

第 2 章 CCI (Common C Interface)

2.1 はじめに 19
2.2 背景 - 移植可能なコードの必要性 19
2.3 CCI の使い方 22
2.4 ANSI 規格からの改良 23
2.5 ANSI 規格の拡張 31
2.6 コンパイラ機能 46

第 3 章 プロジェクトをビルドするための手引き

3.1 はじめに 47
3.2 コンパイラのインストールと有効化 47
3.3 コンパイラの起動 49
3.4 ソースコードの書き方 52
3.5 アプリケーションを思い通りに実行する方法 61
3.6 コンパイル プロセスを理解する 65
3.7 動作に問題のあるコードの修正 72

第 4 章 XC32 ツールチェーンと MPLAB X IDE

4.1 はじめに 75
4.2 MPLAB X IDE とツールのインストール 75
4.3 MPLAB X IDE のセットアップ 76
4.4 MPLAB X IDE プロジェクト 77
4.5 プロジェクトのセットアップ 79
4.6 プロジェクト例 88

第 5 章 コンパイラのコマンドライン ドライバ

5.1 はじめに 91
5.2 コンパイラの起動 91
5.3 C コンパイルのシーケンス 95
5.4 C++ コンパイルのシーケンス 97

© 2015 Microchip Technology Inc.

5.5 ランタイム ファイル 101
5.6 起動と初期化 103
5.7 コンパイラ出力 104
5.8 コンパイラのメッセージ 105
5.9 ドライバ オプションの説明 106

第 6 章 ANSI C 規格の問題

6.1 はじめに 131
6.2 ANSI C 規格からの逸脱 131
6.3 ANSI C 規格に対する拡張 131
6.4 処理系定義のふるまい 132

第 7 章 デバイスに固有の機能

7.1 はじめに 133
7.2 サポートするデバイス 133
7.3 デバイス ヘッダファイル 133
7.4 スタック 134
7.5 ID の格納位置 135
7.6 C コードから SFR を使う 136

第 8 章 サポートするデータ型と変数

8.1 はじめに 139
8.2 識別子 139
8.3 データの表現 139
8.4 整数データの型 140
8.5 浮動小数点データの型 142
8.6 構造体と共用体 144
8.7 ポインタの型 146
8.8 複素数データ型 148
8.9 定数の型と書式 148
8.10 標準型修飾子 151
8.11 コンパイラに固有の修飾子 152
8.12 変数属性 152

第 9 章 メモリの割り当てとアクセス

9.1 はじめに 157
9.2 アドレス空間 157
9.3 データメモリ内の変数 158
9.4 auto 変数のメモリ割り当てとアクセス 161
9.5 プログラムメモリ内の変数 163
9.6 レジスタ内の変数 164
9.7 動的なメモリ割り当て 164
9.8 メモリモデル 165

第 10 章 演算子と命令文

10.1 はじめに 167
10.2 整数拡張 167
10.3 型の参照 169

© 2015 Microchip Technology Inc.

10.4 値としてのラベル 170
10.5 条件演算子のオペランド 171
10.6 case のレンジ指定 171
第 11 章 レジスタの使用
11.1 はじめに 173
11.2 レジスタの使用 173
11.3 レジスタの用法 173
第 12 章 関数
12.1 関数の書き方 175
12.2 関数属性と指定子 176
12.3 関数コードのメモリ割り当て 180
12.4 既定値の関数メモリ割り当てを変更する 180
12.5 関数のサイズ制限 180
12.6 関数のパラメータ 181
12.7 関数の戻り値 183
12.8 関数の呼び出し 183
12.9 関数のインライン展開 184
第 13 章 割り込み
13.1 はじめに 187
13.2 割り込み動作 187
13.3 割り込みサービスルーチンの書き方 188
13.4 ハンドラ関数に例外ベクタを割り当てる 193
13.5 例外ハンドラ 195
13.6 割り込みサービスルーチンのコンテクスト スイッチング 197
13.7 レイテンシ 198
13.8 割り込みのネスト 198
13.9 割り込みの有効化 / 無効化 198
13.10 ISR に関する注意点 198
第 14 章 メイン、スタートアップ、リセット
14.1 はじめに 199
14.2 main 関数 199
14.3 スタートアップ コード 200
14.4 On Reset ルーチン 214
第 15 章 ライブラリ ルーチン
15.1 ライブラリ ルーチンの使用 215
第 16 章 C/C++ とアセンブリ言語の併用
16.1 はじめに 217
16.2 アセンブリ言語と C 変数 / 関数の併用 217
16.3 インライン アセンブリ言語の使い方 220
16.4 定義済みマクロ 224
第 17 章 最適化
17.1 はじめに 227

第 18 章 前処理
18.1 はじめに ... 229
18.2 C/C++ コードのコメント ... 229
18.3 プリプロセッサ ディレクティブ ... 230
18.4 プラグマ ディレクティブ ... 232
18.5 定義済みマクロ ... 233
第 19 章 プログラムのリンク
19.1 はじめに ... 237
19.2 ライブラリ シンボルの置換 ... 237
19.3 リンカによって定義されるシンボル ... 237
19.4 既定値リンカスクリプト ... 238
補遺 A. 組み込みコンパイラ互換モード
A.1 はじめに ... 257
A.2 互換モードでのコンパイル ... 257
A.3 構文互換性 ... 258
A.4 データ型 ... 259
A.5 演算子 ... 259
A.6 拡張キーワード ... 260
A.7 組み込み関数 ... 261
A.8 プラグマ ... 262
補遺 B. 処理系定義のふるまい
B.1 はじめに ... 263
B.2 ハイライト ... 263
B.3 概要 ... 263
B.4 翻訳 ... 264
B.5 環境 ... 264
B.6 識別子 ... 265
B.7 文字 ... 265
B.8 整数 ... 266
B.9 浮動小数点数 ... 267
B.10 配列とポインタ ... 268
B.11 ヒント ... 268
B.12 構造体、共用体、列挙体、ビットフィールド ... 268
B.13 修飾子 ... 269
B.14 宣言子 ... 269
B.15 式文 ... 269
B.16 前処理ディレクティブ ... 270
B.17 ライブラリ関数 ... 270
B.18 アーキテクチャ ... 274
補遺 C. 推奨しない機能
C.1 はじめに ... 275
C.2 指定レジスタへの変数の格納 ... 275

© 2015 Microchip Technology Inc.

補遺 D. ビルトイン関数
D.1 はじめに 277
D.2 ビルトイン関数の説明 278
D.3 ビルトイン DSP 関数 280
補遺 E. ASCII キャラクタセット
補遺 F. 改訂履歴
改訂履歴 289
サポート 291
はじめに 291
myMicrochip 変更通知サービス 291
Microchip 社のウェブサイト 292
Microchip フォーラム 292
カスタマサポート 292
Microchip Technology 社への問い合わせ 292
用語集 293
索引 313
各国の営業所とサービス 326

NOTE:

© 2015 Microchip Technology Inc.

序章

注意

どのような文書でも内容は時間が経つにつれ古くなります。本書も例外ではありません。Microchip 社の製品は、お客様のニーズを満たすために常に改良を重ねており、実際のダイアログやツールが本書の説明とは異なる場合があります。

開発ツールに関する**最新情報**については、MPLAB® IDE または MPLAB X IDE のヘルプをご覧ください。[Help] メニューから [Topics] または [Help Contents] を選択すると、利用できるヘルプファイルのリストが表示されます。

最新版の PDF は、弊社のウェブサイト (http://www.microchip.com) でご覧になれます。文書は DS 番号「DSXXXXXA」で識別します。「XXXXX」は文書番号、「A」は文書のリビジョン レベルを表します。この識別番号は、各ページのフッタ部分、ページ番号の前に記載しています。

MPLAB® XC32 C/C++コンパイラ関連文書とサポート情報は以下の項目に記載しています。

- 本書の構成
- 表記規則
- 推奨参考資料

本書の構成

本書には、GNU 言語ツールを使って 32 ビット アプリケーションのコードを作成する方法を記載しています。本書の構成は以下の通りです。

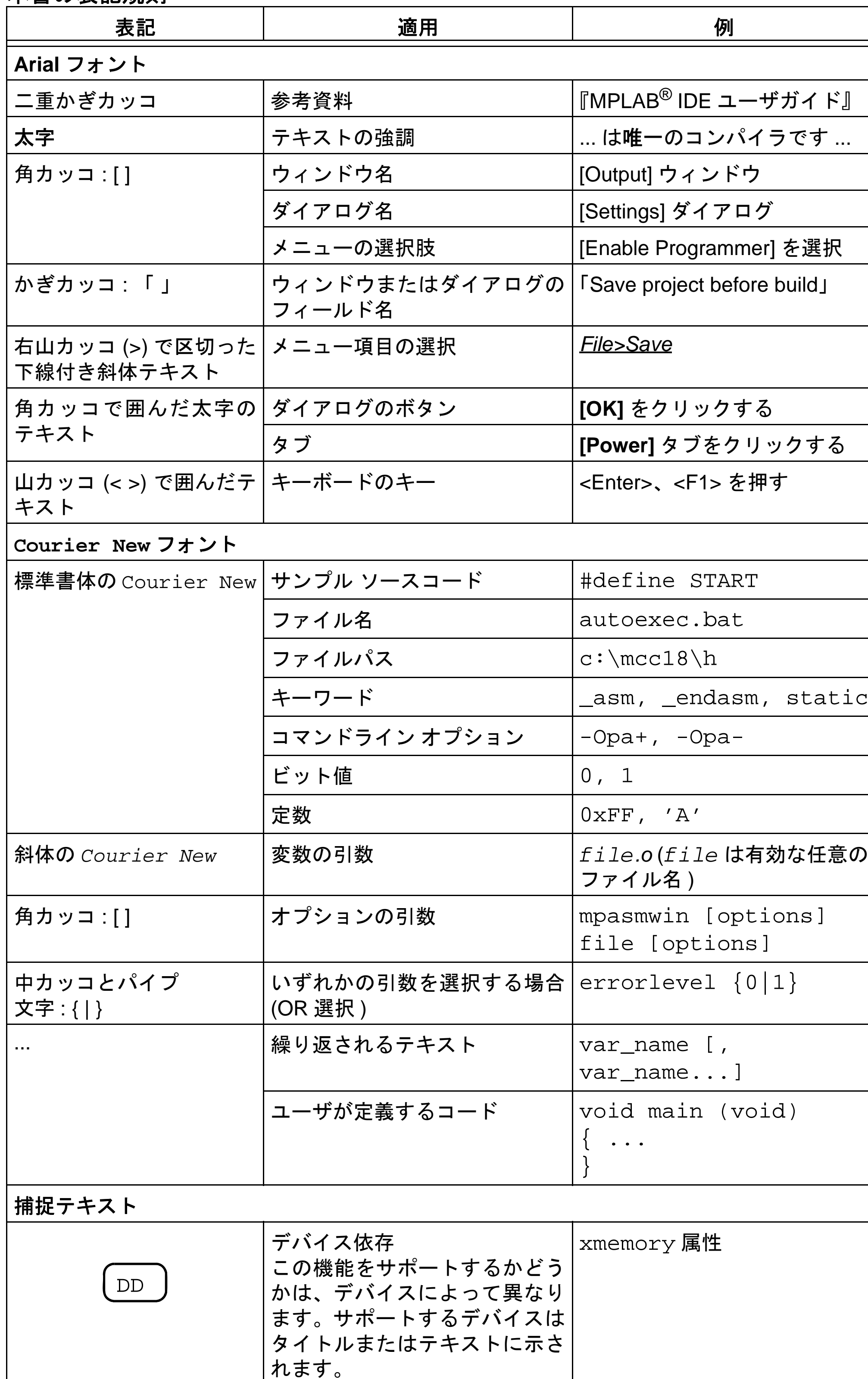
- **第 1 章 コンパイラの概要** - コンパイラ、開発ツール、機能セットに関する説明
- **第 2 章 CCI (Common C Interface)** - 移植性のあるコードを作成するために必要な知識
- **第3章 プロジェクトをビルドするための手引き** - プロジェクトをビルドする際によく起こる状況に関するヘルプとリファレンス
- **第4章 XC32ツールチェーンとMPLAB X IDE** - ツールチェーンとIDEをセットアップするためのガイド
- **第 5 章 コンパイラのコマンドライン ドライバ** - コマンドラインから本コンパイラを使う方法
- **第 6 章 ANSI C 規格の問題** - 本コンパイラと標準 ANSI-89 C がサポートする C/C++ 言語構文の違い
- **第 7 章 デバイスに固有の機能** - コンパイラ ヘッダおよびレジスタ定義ファイルの説明と SFR の使い方
- **第8章 サポートするデータ型と変数** - コンパイラの整数およびポインタデータ型に関する説明
- **第 9 章 メモリの割り当てとアクセス** - コンパイラ ランタイムモデルに関する説明 (セクション、初期化、メモリモデル、ソフトウェア スタック等に関する情報を含む)
- **第 10 章 演算子と命令文** - 演算子と命令文に関する説明
- **第 11 章 レジスタの使用** - SFR へのアクセス方法と使い方
- **第 12 章 関数** - 関数の詳細
- **第 13 章 割り込み** - 割り込みの使い方
- **第 14 章 メイン、スタートアップ、リセット** - C/C++ コードの重要なエレメント
- **第 15 章 ライブラリ ルーチン** - ライブラリの使い方
- **第 16 章 C/C++ とアセンブリ言語の併用** - コンパイラで 32 ビット アセンブリ言語モジュールを使うためのガイドライン
- **第 17 章 最適化** - 最適化オプションに関する説明
- **第 18 章 前処理** - 前処理の詳細
- **第 19 章 プログラムのリンク** - ワークのリンクに関する説明
- **補遺 A. 組み込みコンパイラ互換モード** - 互換モードでコンパイラを使う場合の注意点
- **補遺 B. 処理系定義のふるまい** - ANSI 規格で「処理系定義」とされているコンパイラ固有パラメータに関する説明
- **補遺 C. 推奨しない機能** - 旧式機能の詳細
- **補遺 D. ビルトイン関数** - C コンパイラのビルトイン関数の一覧
- **補遺 E. ASCII キャラクタセット** - ASCII キャラクタセットのテーブル
- **補遺 F. 改訂履歴** - 本書の各リビジョンに関する情報

© 2015 Microchip Technology Inc.

表記規則

本書では以下の表記規則を適用します。

本書の表記規則



表記	適用	例
Arial フォント		
二重かぎカッコ	参考資料	『MPLAB® IDE ユーザガイド』
太字	テキストの強調	... は**唯一**のコンパイラです ...
角カッコ : [ ]	ウィンドウ名	[Output] ウィンドウ
	ダイアログ名	[Settings] ダイアログ
	メニューの選択肢	[Enable Programmer] を選択
かぎカッコ : 「 」	ウィンドウまたはダイアログのフィールド名	「Save project before build」
右山カッコ (>) で区切った下線付き斜体テキスト	メニュー項目の選択	*File>Save*
角カッコで囲んだ太字のテキスト	ダイアログのボタン	**[OK]** をクリックする
	タブ	**[Power]** タブをクリックする
山カッコ (< >) で囲んだテキスト	キーボードのキー	<Enter>、<F1> を押す
Courier New フォント		
標準書体の Courier New	サンプル ソースコード	#define START
	ファイル名	autoexec.bat
	ファイルパス	c:\mcc18\h
	キーワード	_asm, _endasm, static
	コマンドライン オプション	-Opa+, -Opa-
	ビット値	0, 1
	定数	0xFF, 'A'
斜体の *Courier New*	変数の引数	*file.o* (*file* は有効な任意のファイル名)
角カッコ : [ ]	オプションの引数	mpasmwin [options] file [options]
中カッコとパイプ文字 : { | }	いずれかの引数を選択する場合 (OR 選択)	errorlevel {0|1}
...	繰り返されるテキスト	var_name [, var_name...]
	ユーザが定義するコード	void main (void) { ... }
捕捉テキスト		
DD	デバイス依存 この機能をサポートするかどうかは、デバイスによって異なります。サポートするデバイスはタイトルまたはテキストに示されます。	xmemory 属性

推奨参考資料

本書には、MPLAB XC32 C/C++ コンパイラの使い方を記載しています。本書に関連する参考資料として、Microchip 社が提供する以下の文書を推奨します。

リリースノート (Readme ファイル)
Microchip 社製ツールの最新情報は、ソフトウェアに付属するリリースノート (HTML ファイル) でご覧になれます。

MPLAB® XC32 Assembler, Linker and Utilities User's Guide (DS50002186**)**
32 ビット アセンブラ、オブジェクト リンカ、オブジェクト アーカイバ / ライブラリアン、各種ユーティリティの使い方を記載したガイド書です。

32-Bit Language Tools Libraries (DS51685)
MPLAB XC32 C/C++ コンパイラと一緒に提供される全てのライブラリ関数の詳細な説明と使い方を記載しています。

Dinkum Compleat ライブラリ
Dinkum Compleat ライブラリは、複数のヘッダ (ライブラリ ファシリティを宣言または定義するためにユーザ プログラムにインクルードするファイル) で構成されます。Dinkumライブラリは MPLAB X IDE アプリケーションで利用できます([My MPLAB X IDE] タブの [References & Featured Links] セクション)。

PIC32 コンフィグレーション 設定
MPLAB XC32 C/C++コンパイラの`#pragma config`がサポートするMicrochip PIC32 デバイスのコンフィグレーション ビット設定の一覧です。

デバイスに固有の文書
Microchip 社のウェブサイトは、32 ビット デバイスの機能や特長を記載した文書を豊富に提供しています。これには以下が含まれます。

- 個々のデバイスおよびデバイスファミリのデータシート
- ファミリ リファレンス マニュアル
- プログラマ リファレンス マニュアル

C 言語規格の情報
American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd.Street, New York, New York, 10036.

この規格は、C プログラミング言語の記法を定め、プログラムの解釈方式を規定しています。その目的は、様々なコンピューティング システム上での C 言語プログラムの移植性、信頼性、保守性、および実行効率性を高める事です。

C++ 言語規格の情報
Stroustrup, Bjarne, *C++ Programming Language:Special Edition*, 3rd Edition.Addison-Wesley Professional; Indianapolis, Indiana, 46240.

ISO/IEC 14882 C++ 規格 ISO C++ は、ISO (The International Standards Organization) が ANSI (The American National Standards Institute)、BSI (The British Standards Institute)、DIN (The German national standards organization) と協力して定めた規格です。

この規格は書式を指定し、プログラミング言語 C++ プログラミング言語の記法を定め、プログラムの解釈方式を規定しています。その目的は、様々なコンピューティング システム上での C++ 言語プログラムの移植性、信頼性、保守性、および実行効率性を高める事です。

© 2015 Microchip Technology Inc.

C 言語のリファァレンス 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.

GCC 文書

http://gcc.gnu.org/onlinedocs/

http://sourceware.org/binutils/

NOTE:

© 2015 Microchip Technology Inc.

第 1 章　コンパイラの概要

1.1 はじめに

本章には、MPLAB XC32 C/C++ コンパイラに関する以下の内容を記載しています。

- 対応デバイス
- MPLAB XC32 C/C++ コンパイラについて
- コンパイラとその他の開発ツール

1.2 対応デバイス

MPLAB XC32 C/C++ コンパイラは、全ての Microchip 社製 32 ビットデバイスを完全にサポートします。

1.3 MPLAB XC32 C/C++ コンパイラについて

MPLAB XC32 C/C++ コンパイラは機能を完備した最適化コンパイラとして、ANSI C 準拠プログラムを 32 ビットデバイス向けのアセンブリ言語ソースコードに変換します。加えて、本コンパイラは多数のコマンドライン オプションと、32 ビット デバイス ハードウェア機能へのフルアクセスを可能にする言語拡張をサポートし、コードジェネレータの微調整も可能です。

本コンパイラは Free Software Foundation が提供する GCC コンパイラをベースにしています。

本コンパイラは 32/64 ビット Windows®、Linux、Apple OS X を含む一般的なオペレーティング システム向けに利用可能です。

本コンパイラは 3 種類の動作モード (Free、Standard、PRO) で動作可能です。Standard およびPRO動作モードはライセンスが必要なモードであり、有効化のためにアクティベーション キーとインターネット接続が必要です。Free モードにはライセンスは不要です。基本的なコンパイラ動作、サポート デバイス、利用可能メモリは全てのモードで同じです。コンパイラが採用する最適化レベルだけがモードによって異なります。

1.3.1 用語の用法

本書では、「コンパイラ」という用語をしばしば MPLAB XC32 C/C++ コンパイラが提供するアプリケーション群の全てまたは一部を指す意味で使います。多くの場合、ある動作がパーサとコード ジェネレータのどちらのアプリケーションによって実行されたか知る事は重要ではありません。そのような場合、「コンパイラ」が実行したと言えば十分です。

また、「コンパイラ」をコマンドライン ドライバ (または単にドライバ) を指す意味で使う場合もあります。なぜなら、このアプリケーションは常にコンパイル処理を呼び出すために実行するからです。MPLAB XC32 C/C++ コンパイラ パッケージ向けのドライバは `xc32-gcc` と呼びます。C/ASM プロジェクト向けのドライバも `xc32-gcc` です。C/C++/ASM プロジェクト向けのドライバは `xc32-g++` です。**5.9「ドライバ オプションの説明」**では、各種のドライバと、それらのオプションについて説明します。上記の用法に従い、特に明記していない場合、「コンパイラ オプション」は「コマンドライン ドライバオプション」を指します。

同様に「コンパイル」は、ソースコードから実行可能バイナリイメージを生成するプロセスに含まれる全てまたは一部の手順を指す意味で使います。

1.3.2 ANSI C 規格

本コンパイラは、ANSI C 規格 (ANSI 仕様書 (ANSI x3.159-1989) で定義され、Kernighan/Ritchie 著『The C Programming Language』(第 2 版) で記述されている規格) への準拠を完全に検証済みです。ANSI 規格はオリジナルの C 言語定義に対する拡張機能を含みます。現在、それらは C 言語の標準機能となっています。これらの拡張により、移植性と性能が向上します。加えて、PIC32 MCU 組み込み制御アプリケーション向けの言語拡張も含まれます。

1.3.3 最適化

本コンパイラは、様々な先進技法を採用した一連の高機能な最適化パスを使う事で、C/C++ ソースから高効率でコンパクトなコードを生成します。最適化パスには、全ての C/C++ コードに適用可能な高水準の最適化と、デバイス アーキテクチャの特定機能を利用する PIC32 MCU 専用の最適化を含みます。

最適化の詳細については**第 17 章「最適化」**を参照してください。

1.3.4 ANSI 標準ライブラリのサポート

本コンパイラは、完全な ANSI C 標準ライブラリ付きで配布されます。全てのライブラリ関数は検証済みであり、ANSI C ライブラリ規格に準拠します。このライブラリは文字列操作、動的メモリ割り当て、データ変換、計時用の関数と数学関数 (三角、指数、双曲線) を含みます。また、ファイル操作用の標準 I/O 関数が配布時に含まれます。これらの関数は、コマンドライン シミュレータを使ってホスト ファイルシステムへのフルアクセスをサポートします。さらに、低水準ファイル I/O 機能向けのソースコードもコンパイラの配布時に提供されます。このソースコードは、この機能を必要とするアプリケーション向けの初期コードとして使えます。

1.3.5 ISO/IEC C++ 規格

本コンパイラは、2003 標準 C++ ライブラリ付きで配布されます。

> **Note:** プロジェクト プロパティ内でMPLAB XC32のシステム インクルード ディレクトリ(例: `/pic32mx/include/`)を指定しないでください。`xc32-gcc` および `xc32-g++` コンパイル ドライバは、XC libc または Dinkumware libc と、それらに対応するインクルード ファイル ディレクトリを自動的に選択します。システム インクルード ファイルのパスを手動で追加すると、この自動処理が阻害され、不適切な libc インクルード ファイルがプロジェクト向けにコンパイルされる恐れがあります。その場合、インクルード ファイルとライブラリの間で不整合が生じます。プロジェクト プロパティへのシステム インクルードパスの追加は、以前から非推奨の行為である事に注意してください。

1.3.6 コンパイラ ドライバ

本コンパイラは強力なコマンドライン ドライバ プログラムを備えています。ドライバ プログラムを使うと、アプリケーション プログラムを 1 つの手順でコンパイル / アセンブル / リンクできます。

1.3.7 関連文書

本 C コンパイラは、MPLAB IDE (v8.xx 以上) と MPLAB X IDE の両方でサポートされます。C++ に関しては、MPLAB X IDE (v1.40 以上) が必要です。本書では、どちらの IDE も単に「MPLAB IDE」と呼びます。

特定のデバイス (および特定のコンパイラ) にだけ関係する機能には、「DD」マーク (**「序章」**参照) と、適用デバイスを識別するテキストが表記されます。

© 2015 Microchip Technology Inc.

1.4 コンパイラとその他の開発ツール

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

- MPLAB XC32 アセンブラ / リンカ - 『MPLAB® XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) 参照
- MPLAB IDE v8.xx と MPLAB X IDE (C++ には MPLAB X IDE v1.40 以上が必要)
- MPLAB シミュレータ
- 全ての Microchip 社製デバッグツールとプログラマ
- 32 ビットデバイスをサポートするデモボードとスタータキット

NOTE:

© 2015 Microchip Technology Inc.

第 2 章　CCI (Common C Interface)

2.1 はじめに

CCI (Common C Interface) は全ての MPLAB XC C コンパイラで利用でき、それらのコンパイラ間のコード移植性を高めます。例えば、CCI 準拠コードを使うと、MPLAB XC8 C コンパイラを使う PIC18 MCU から MPLAB XC32 C/C++ コンパイラを使う PIC32 MCU へコードを容易に移植できます。

CCI は、ソースコードが ANSI 規格に準拠済みである事を前提とします。CCI を使う場合、ユーザの責任で CCI に準拠したコードを書く必要があります。レガシー プロジェクトは、CCI に準拠させるための移行が必要です。プロジェクトをビルドする際は、コンパイラ オプションを設定してコンパイラ動作をインターフェイスに適合させる必要があります。

本章では以下の項目について説明します。

- 背景 - 移植可能なコードの必要性
- CCI の使い方
- ANSI 規格からの改良
- ANSI 規格の拡張
- コンパイラの機能

2.2 背景 - 移植可能なコードの必要性

プログラマーなら誰でも移植可能なソースコードを書きたいと思います。

移植性とは、書いた時とは異なる実行環境で同じソースコードをコンパイル / 実行できるという事です。コードが完璧な移植性を持ち、しかも柔軟に変更可能であれば、そのコードを新しい環境で実行するために費やす時間と労力を削減できます。

多くの場合、組み込みエンジニアは異なるターゲット デバイス間のコード移植性を考えますが、これは移植性の一部に過ぎません。異なるコンパイラを使って、同じコードを同じターゲットデバイス向けにコンパイルする事があります。その場合、コンパイラ間の違いによってコードのコンパイル時または実行時に問題が生じる可能性があるため、これも移植性として考える必要があります。

同じターゲット デバイスと同じベンダーのコンパイラだけを使う場合でも、コンパイラの動作を調整せずにバージョンを単純にアップデートすると、コードの挙動が変化する可能性があります。

コードが真に柔軟であるためには、ターゲットとツールだけでなく、時が経つと変化する要因に対しても移植性を維持する必要があります。

この移植性は、プログラマーだけで達成する事はできません。なぜならば、コンパイラのベンダーは異なる技術に基づく製品を開発したり、機能やコード構文を変更したり、コンパイラの動作を改良したりするからです。コンパイラが大きく改良されるたびに、多くの無防備なプロジェクトが使えなくなりました。

コンパイラの変更を管理してコードの移植性を向上させるために、C 言語の規格が開発されました。ANSI (American National Standards Institute) は、プログラミング言語を含む様々な技術分野に向けて規格を公開しています。ANSI C 規格は、C プログラミング言語の標準として広く一般に採用されています。

© 2015 Microchip Technology Inc.

2.2.1 ANSI 規格

コンパイラ ベンダーには新しいターゲット デバイスへの対応やコード生成の改善に向けた開発自由度が必要である一方、プログラマーはソースコードが従来通りに動作する事を期待します。従って、ANSI C 規格には、これらの相反する目標を調和させる事が求められます。両方の要求を満たす事ができれば、ソースコードに移植性を持たせる事ができます。

この規格は、C プログラムが従う必要のある構文のみならず、C プログラムを解釈するための意味論的規則も詳細に定義します。従って、ANSI C 準拠コンパイラは、この規格が定義する C プログラム関数に準拠している必要があります。

この規格は、一連のツールとコード実行環境として「処理系」(implementation) を定義しています。これらが変化した場合 (例 : ターゲット デバイスを変更した場合や、ビルドに使うコンパイラのバージョンをアップデートした場合)、以前とは異なる処理系を使う事になります。

この規格は、外から見えるプログラムの挙動という意味で、「ふるまい」(behavior) という用語を使います。これはプログラムの書き方には関係ありません。

この規格は相反する複数の目標を達成しようとするため、一部の仕様は曖昧に見えます。例えばこの規格は、「`int` 型は少なくとも 16 ビット値を保持できなければならない」と規定していますが、実際に使える `int` 型のサイズについては述べていません。また、符号付き整数の右シフト操作の結果は、たとえ ANSI C 規格に準拠していても、処理系によって異なる可能性があります。

規格が厳格すぎると、コンパイラは異なるデバイス アーキテクチャに順応できなくなります[1]。しかし、規格が緩すぎると、コンパイラやアーキテクチャに応じて結果が大きく変化するため、規格としての意味が薄れます。

ANSI C 規格は、ふるまいが完全に定義されないソースコードを以下のグループに分類します。

処理系定義 (Implementation-defined behavior)
これは不定 (unspecified behavior) であり、どのようにふるまいを選択するかは、各処理系によって定義されます。

不定 (Unspecified behavior)
規格は 2 つ以上の選択肢を提供しますが、各インスタンスでどのふるまいを選択するかは規定しません。

未定義 (Undefined behavior)
規格はふるまいに関して一切規定しません。

規格に厳密に準拠するコードは不定、未定義、処理系定義のふるまいに依存する出力を生成しません。前術の例で挙げた `int` のサイズは、処理系によって定義されるふるまいのカテゴリに属します。`int` のサイズは使用するコンパイラ、そのコンパイラの使い方、ターゲット デバイスに依存します。

全ての MPLAB XC コンパイラは ANS X3.159-1989 プログラミング言語規格に準拠します (脚注に記載したように XC8 コンパイラが再帰を実装できない点は除く)。これは一般的に C89 規格と呼びます。後の C99 規格の一部の機能もサポートされます。

1. 事例 : ミッドレンジ PIC® マイクロコントローラはデータスタックを備えていません。このデバイスをターゲットとする場合、再帰を実装できないため、コンパイラは ANSI C 規格に準拠できません。これは、ANSI C 規格がミッドレンジ デバイスおよびツールに対して厳格すぎる事を示しています。

© 2015 Microchip Technology Inc.

自立処理系 (freestanding implementation) - 一般的に組み込みアプリケーションと呼ぶ - の場合、規格は言語に対する規格外拡張を許容します。しかし、当然ながら、規格はそれらの指定方法または動作方法を規定しません。プログラマーがデバイス ハードウェアに密接に関与する場合、デバイス セットアップと割り込みを指定するための手段が必要です。また、小型デバイスの多くは複雑なメモリ アーキテクチャを有するため、それらを使うための手段も必要です。これを規格によって一貫した方法で提供する事はできません。

ANSI C 規格はプログラマーとコンパイラベンダーに相互理解を提供しますが、プログラマーは使用ツールの処理系定義ふるまいと移植性 (すなわち C 言語に対する規格外拡張の必要性) を考慮する必要があります。どちらの状況もコードの移植性に影響する可能性があります。

2.2.2 CCI (Common C Interface)

CCI (Common C Interface) は ANSI C 規格を補完します。MPLAB XC C コンパイラで CCI を使うと、全ての Microchip 社製デバイスで一貫した結果が容易に得られます。

CCI は以下の利点を提供します。これらは全て移植性を考慮して設計されています。

ANSI C 規格からの改良

CCI は、ANSI C 規格の下では処理系定義であると見なされるコードのふるまいを明確に定義します。例えば、符号付き整数の右シフトは CCI によって完全に定義されます。デバイスの特性と密接に関係する多くの処理系定義動作 (`int` のサイズ等) は、CCI によって定義されないという事に注意してください。

規格外拡張のための統一された構文

CCI 規格外拡張のほとんどは、統一された構文でキーワードを使って実装します。これらは、コンパイラに固有の処理系であるキーワード、マクロ、属性を置換します。キーワードの処理系はコンパイラごとに異なる場合があり、キーワードへの引数はデバイスごとに異なる場合があります。

コーディングのガイドライン

CCI は、コードを他のデバイスまたは他のコンパイラに移植可能とするためのコードの書き方に関するアドバイスを提供します。アドバイスを無視する事はできますが、そのコードは CCI に準拠しません。

2.3 CCI の使い方

CCI は、処理系定義のふるまいを改良すると共に、言語拡張の構文を標準化する事によって、移植性を高めます。

ユーザは CCI に準拠するかどうか (有効にするかどうか) を選択できます。これは新しいプロジェクトを開始する際に選択しますが、既存プロジェクトを変更して CCI に準拠させる事もできます。

プロジェクトを CCI に準拠させるには以下が必要です。

CCI を有効にする
プロジェクト内で MPLAB IDE ウィジェット「Use CCI Syntax」を選択するか、これに等価なコマンドライン オプションを使います。

各モジュールに <xc.h> をインクルードする
一部の CCI 機能は、コンパイラがこのヘッダを検出した場合にのみ有効です。

ANSI に準拠している事を確認する
ANSI C 規格に準拠していないコードは CCI に準拠しません。

CCI による ANSI C 規格からの改良を順守する
CCIは、ANSI C規格の下で処理系定義であった一部のふるまいを明確に定義します。

CCI による言語拡張を使う
コンパイラ固有の言語拡張ではなく CCI による拡張を使います。

この後の各セクションでは、CCI による ANSI C 規格からの改良、ANSI C 規格の拡張、その他のコンパイラ オプションと使用方法について詳細に説明すると共に、ガイドラインを提供します。

本書に記載していない処理系定義ふるまいまたは規格外拡張は CCI の範囲外です。例えば GCC ケースレンジ、ラベルアドレス、24 ビット `short long` 型は CCI の範囲外です。これらの機能を使うプログラムは CCI に準拠しません。CCI を有効にして CCI の範囲外の機能を使った場合、コンパイラが警告またはエラーを示す可能性があります。

 © 2015 Microchip Technology Inc.

2.4 ANSI 規格からの改良

以下の各項目では、ANSI C 規格における処理系定義のふるまいを CCI がどのように改良するかについて説明します。

2.4.1 ソースコードの書き方

CCI を使う場合、7 ビット ASCII セットに含まれる文字を使ってソースコードを書く必要があります。行末には「ラインフィード」(\n) または「キャリッジ リターン (\r) + ラインフィード」が使えます。文字定数または文字列リテラルでエスケープ文字を使う事により、基本のキャラクタセットに含まれない拡張文字を表現できます。

2.4.1.1 例

エスケープ文字を使って文字列定数を定義する例を以下に示します。

```
const char myName[] = "Bj\370rk\n";
```

2.4.1.2 コンパイラの違い

CCI 適用前の全てのコンパイラでも同じキャラクタセットを使いました。

2.4.1.3 CCI への移行

対応は不要です。

2.4.2 main のプロトタイプ

main() 関数のプロトタイプは以下の通りです。

```
int main(void);
```

2.4.2.1 例

main() の定義例を以下に示します。

```
int main(void)
{
    while(1)
        process();
}
```

2.4.2.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラは、この関数に void リターン型を使いました。

2.4.2.3 CCI への移行

各プログラムは 1 つの main() 関数を定義します。以前に 8 ビットのターゲット デバイス向けにコンパイルした全てのプロジェクトでは、main() のリターン型を確認する必要があります。

2.4.3 ヘッダファイルの指定

ディレクトリ区切り文字を使ったヘッダファイル指定は CCI に準拠しません。

2.4.3.1 例

CCI に準拠する 2 つのインクルード ディレクティブの例を以下に示します。

```
#include <usb_main.h>
#include "global.h"
```

2.4.3.2 コンパイラの違い

CCI 適用前の全てのコンパイラ バージョンでは、ヘッダファイル指定にディレクトリ区切り文字が使えました。Windows 形式の区切り文字「\」を使ったコードを他のホスト オペレーティング システムの下でコンパイルすると、互換性の問題が生じました。CCI の下では、ディレクトリ区切り文字を使わない事が必要です。

2.4.3.3 CCI への移行

このディレクティブ内のヘッダファイル名以外のテキストを全て削除します。コンパイラのインクルード検索パスまたは MPLAB IDE の等価なパスにディレクトリパスを追加します。コンパイラは、このオプションで指定されたディレクトリを検索します。

例えば、

```
#include <inc/lcd.h>
```

を

```
#include <lcd.h>
```

に変更し、MPLAB IDE プロジェクト プロパティ内でコンパイラのヘッダ検索パスに inc ディレクトリへのパスを追加するか、コマンドラインで以下を指定します。

```
-Ilcd
```

2.4.4 インクルード検索パス

CCI の下でインクルードするヘッダファイルは、コンパイラの検索パスで見つかる必要があります。

ヘッダファイルを山カッコ「< >」で囲んで指定した場合、最初に -I オプション (または等価な MPLAB IDE オプション) で指定したパスが検索され、次に標準のコンパイラ インクルード ディレクトリが検索されます。-I オプションは指定順に検索されます。

ファイルをダブルクォーテーション「" "」で囲んで指定した場合、最初に現在作業中のディレクトリが検索されます。MPLAB X プロジェクトの場合、現在作業中のディレクトリは C ソースファイルが保存されているディレクトリです。このディレクトリでファイルが見つからない場合、ヘッダファイルを山カッコで囲んで指定した場合と同じディレクトリが検索されます。

上記以外のヘッダファイル検索パス指定オプションは CCI に準拠しません。

2.4.4.1 例

以下のディレクティブを使ってヘッダファイルをインクルードした場合、

```
#include "myGlobals.h"
```

現在作業中のディレクトリ、-I オプションで指定したパス、標準のコンパイラ ディレクトリのいずれかにヘッダファイルを保存する必要があります。これ以外の場所に保存した場合、CCI に準拠しません。

2.4.4.2 コンパイラの違い

CCI の下でもコンパイラの動作は変化しません。これは純粋にコーディングのガイドラインです。

2.4.4.3 CCI への移行

-I オプション以外のヘッダファイル検索パス指定オプション (または、これに等価な MPLAB IDE オプション) は全て -I オプションに変更する必要があります。ヘッダファイルは上記のディレクトリに保存する必要があります。

 © 2015 Microchip Technology Inc.

2.4.5 識別子の有意先頭文字数

CCI における内部および外部識別子の有意先頭文字数は 255 以上です。これは ANSI C 規格の要件を拡張します (ANSI C 規格は、オブジェクトの識別に使う有意文字数をこれよりも少なく規定しています)。

2.4.5.1 例

以下の 2 つの長い変数名 (どちらも 72 文字) は、CCI の下で一意に識別されます。

```
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningFast;
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningSlow;
```

2.4.5.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、有意文字数の既定値は 31 でした (オプションで拡張可能)。

CCI 適用前の 16 および 32 ビットコンパイラでは、有意文字数の制限はありませんでした。

2.4.5.3 CCI への移行

対応は不要です。命名に関する制限を緩和できます。

2.4.6 データ型のサイズ

CCI は、C 言語の基本的なデータ型 (例 : char、int、long 等) のサイズを完全には定義しません。設計により、これらの型はターゲット デバイスのレジスタとその他のアーキテクチャ機能のサイズを反映します。型を指定する事で、デバイスはその型のオブジェクトに効率的にアクセスできます。しかし、ANSI C 規格は、これらの型に対して最小サイズ要件だけを規定しています (<limits.h> で指定)。

プロジェクト内で固定サイズの型が必要な場合、<stdint.h> 内で定義された型 (例 : uint8_t または int16_t) を使います。これらの型は、たとえ CCI の範囲外であっても、全ての XC コンパイラで一貫して定義されます。

基本的に、C 言語では 2 つの型のグループが選べます。1 つはターゲット デバイスに合わせたサイズとフォーマットを提供し、もう 1 つはターゲットに関係なく固定されたサイズを提供します。

2.4.6.1 例

以下に変数定義の例を示します。変数 native のサイズは、ターゲット デバイス上での効率的なアクセスを可能にします。変数 fixed のサイズは明確に指定され、ターゲット デバイスによっては効率的にアクセスできない場合でも、サイズは固定されたままです。

```
int native;
int16_t fixed;
```

2.4.6.2 コンパイラの違い

これは、CCI 適用前のコンパイラによって実装された型と一致します。

2.4.6.3 CCI への移行

ターゲット デバイスに関係なく固定サイズの型が必要な場合、<stdint.h> で定義された型の 1 つを使います。

2.4.7 通常の char 型

CCI では、符号の有無を指定しない通常の char 型は unsigned char です。一般的に、char 型を定義する際は、オブジェクトの符号の有無を明示的に指定する事を推奨します。

2.4.7.1 例

```
char foobar;
```

上の例は、foobar という名称の unsigned char オブジェクトを定義します。

2.4.7.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラは、char を常に符号なし型として扱いました。

CCI 適用前の 16 および 32 ビット コンパイラは、既定値の char 型として signed char を使いました。それらのコンパイラでは、-funsigned-char オプションによって既定値の型を unsigned char 型に変更しました。

2.4.7.3 CCI への移行

16 または 32 ビット コンパイラを使う場合、通常の char として定義したオブジェクトの定義は、全て見直す必要があります。「signed」を意図していた全ての char は、下例のように明示的に定義し直す必要があります。

```
signed char foobar;
```

XC16/32 コンパイラでは -funsigned-char オプションを使って char の型を変更できますが、XC8 はこのオプションをサポートしません。従って、このオプションを使ったコードは、厳密には CCI に準拠しません。

2.4.8 符号付き整数の表現

符号付き整数の値は、整数の 2 の補数を取る事によって求まります。

2.4.8.1 例

以下の例は、変数 test に十進数の「-28」を代入します。

```
signed char test = 0xE4;
```

2.4.8.2 コンパイラの違い

CCI 適用前の全てのコンパイラでも、上記の方法で符号付き整数を表現しました。

2.4.8.3 CCI への移行

対応は不要です。

© 2015 Microchip Technology Inc.

2.4.9　整数の変換

整数型をサイズが不十分な符号付き整数に変換すると、そのサイズに収めるために元の値は最上位ビットから切り詰められます。

2.4.9.1　例

下の例では、代入によって値が切り詰められます。

```
signed char destination;
unsigned int source = 0x12FE;
destination = source;
```

CCI の下では、代入後の destination の値は -2 (0xFE) です。

2.4.9.2　コンパイラの違い

CCI 適用前の全てのコンパイラでも、上記と同じ方法で整数の変換を実行しました。

2.4.9.3　CCI への移行

対応は不要です。

2.4.10　整数値のビット演算

符号付き値に対するビット演算は 2 の補数表現 (符号ビットを含む) に従います。**2.4.11「符号付き値の右シフト」**も参照してください。

2.4.10.1　例

以下に、ビット単位 AND 演算における負の量の例を示します。

```
signed char output, input = -13;
output = input & 0x7E;
```

CCI の下では、代入後の output の値は 0x72 です。

2.4.10.2　コンパイラの違い

CCI 適用前の全てのコンパイラでも、上記と同じ方法でビット演算を実行しました。

2.4.10.3　CCI への移行

対応は不要です。

2.4.11　符号付き値の右シフト

符号付き値の右シフトでは符号拡張を行います。これは、元の値の符号を保持します。

2.4.11.1　例

以下に、ビット単位 AND 演算における負の量の例を示します。

```
signed char input, output = -13;
output = input >> 3;
```

CCI の下では、代入後の output の値は -2 (0xFE) です。

2.4.11.2　コンパイラの違い

CCI 適用前の全てのコンパイラでも、上記の方法で符号付き整数を表現しました。

2.4.11.3　CCI への移行

対応は不要です。

2.4.12　型の異なるメンバーを使ってアクセスする共用体メンバーの変換

型の異なる複数のメンバーを定義している共用体 (union) に対して、あるメンバー識別子を使って別のメンバーの内容にアクセスを試みた場合、動作 (結果に対して何らかの変換が適用されるかどうか) は ANSI C 規格において処理系定義です。CCI では変換は一切適用されず、アラインメント等の条件が無効であろうとなかろうと、共用体オブジェクトのバイトは、アクセス先のメンバーと同じ型のオブジェクトとして解釈されます。

2.4.12.1　例

複数のメンバーを定義する共用体の例を以下に示します。

```
union {
    signed char code;
    unsigned int data;
    float offset;
} foobar;
```

コードが `data` の読み出しによって `offset` の抽出を試みた場合、正しい読み値は保証されません。

```
float result;
result = foobbar.data;
```

2.4.12.2　コンパイラの違い

CCI 適用前の全てのコンパイラでも、他のメンバーを介してアクセスされる共用体メンバーを変換しませんでした。

2.4.12.3　CCI への移行

対応は不要です。

2.4.13　既定値の `int` 型ビットフィールド

CCI では、符号の有無を指定しない通常の `int` として指定したビットフィールドの型は、`unsigned int` を使って定義した場合と同じです。これは `int`、`signed`、`signed int` が同義的である他のオブジェクトとは大きく異なります。全てのビットフィールド定義では、符号の有無を明示的に指定する事を推奨します。

2.4.13.1　例

サイズを指定した符号なし整数のビットフィールドを含む構造体タグの例を以下に示します。

```
struct OUTPUTS {
    int direction :1;
    int parity    :3;
    int value     :4;
};
```


© 2015 Microchip Technology Inc.

2.4.13.2　コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、ビットフィールドに int 型を使うと警告が出され、そのビットフィールドは unsigned int 型として実装されました。

CCI適用前の16および32ビットコンパイラでは、intを使って定義したビットフィールドは signed int 型として実装されました (-funsigned-bitfields オプションを指定しなかった場合)。

2.4.13.3　CCI への移行

通常のint型を使ってビットフィールドを定義した全てのコードは見直す必要があります。それらのビットフィールドに対して符号付きの量を意図していた場合、以下のように signed int に変更する必要があります。

```
struct WAYPT {
    int log          :3;
    int direction    :4;
};
```

上のコードから、以下のようにビットフィールドの型を signed int に変更します。

```
struct WAYPT {
    signed int log       :3;
    signed int direction :4;
};
```

2.4.14　記憶域ユニットの境界を越えるビットフィールド

ANSI C 規格では、ビットフィールドが記憶域ユニットの境界を越える事ができるかどうかは処理系定義です。CCI では、ビットフィールドは記憶域ユニットの境界を越えません。追加の記憶域ユニットが構造体に割り当てられ、ビットのパディングによってギャップが埋められます。

記憶域ユニットのサイズは、基本のデータ型 (例 : int) に基づくため、コンパイラごとに異なる事に注意してください。ビットフィールドの型は、基本のデータ型によって決まります。8 ビットコンパイラの記憶域ユニットのサイズは 8 ビットであり、16 ビットコンパイラの場合は 16 ビット、32 ビットコンパイラの場合は 32 ビットです。

2.4.14.1　例

以下に、複数のビットフィールドを含む構造体の定義例を示します。

```
struct {
        unsigned first  :6;
        unsigned second :6;
} order;
```

XC8 コンパイラで CCI を使う場合、記憶域の割り当て単位は 1 バイトです。ビットフィールド first に割り当てられた最初の記憶域ユニットには 2 ビットしか残らないため、ビットフィールド second には追加の記憶域ユニットが割り当てられます。従って、この構造体 order のサイズは 2 バイトです。

2.4.14.2　コンパイラの違い

CCI 適用前の全てのコンパイラも上記と同じ割り当てを行います。

2.4.14.3　CCI への移行

対応は不要です。

2.4.15　ビットフィールドの割り当て順

ANSI C規格は、ビットフィールドを記憶域ユニットに割り当てる際のメモリへのビット割り当て順を指定していません。CCI では、定義された最初のビットは、対応する記憶域ユニットの最下位ビットに割り当てられます。

2.4.15.1　例

以下に、複数のビットフィールドを含む構造体の定義例を示します。

```
struct {
        unsigned lo  :1;
        unsigned mid :6;
        unsigned hi  :1;
} foo;
```

ビットフィールド lo は、構造体 foo に割り当てられた記憶域ユニットの最下位ビットに割り当てられます。ビットフィールド mid は、次の最下位 6 ビットに割り当てられ、ビットフィールド hi は、同じ記憶域ユニットバイトの最上位ビットに割り当てられます。

2.4.15.2　コンパイラの違い

以上は、CCI 適用前の全てのコンパイラの動作と同じです。

2.4.15.3　CCI への移行

対応は不要です。

2.4.16　NULL マクロ

NULL マクロは <stddef.h> 内で定義されます。しかし、この定義は処理系定義です。CCI の下では、NULL の定義は（0）です。

2.4.16.1　例

下の例は、NULL マクロを使ってポインタに NULL ポインタ定数を代入します。

```
int * ip = NULL;
```

NULL の値（0）は暗黙的に目的の型に変換されます。

2.4.16.2　コンパイラの違い

CCI 適用前の 32 ビットコンパイラでは、NULL の定義は（(void *)0）でした。

2.4.16.3　CCI への移行

対応は不要です。

2.4.17　浮動小数のサイズ

CCI の下では、浮動小数型のサイズは 32 ビット以上である事が必要です。

2.4.17.1　例

下の例が定義する outY のサイズは 32 ビット以上です。

```
float outY;
```

2.4.17.2　コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、24 ビットの float および double 型が使えました。

2.4.17.3　CCI への移行

8 ビットコンパイラを使う場合、CCI モードを有効にすると float および double 型のサイズは自動的に 32 ビットになります。float または double 型のサイズを 24 ビットとして想定していたソースコードは見直す必要があります。

その他のコンパイラでは、移行のための対応は不要です。

© 2015 Microchip Technology Inc.

2.5 ANSI 規格の拡張

以下では、CCI が提供するデバイスに固有の ANSI 規格に対する拡張について説明します。

2.5.1 汎用ヘッダファイル

CCI では、コンパイラおよびデバイスに固有の型と SFR の全てを、1 つのヘッダファイル `<xc.h>` を使って宣言する必要があります。このファイルは、CCI に準拠する各モジュールにインクルードする必要があります。一部の CCI 定義は、コンパイラがこのヘッダを検出した場合にのみ有効です。

2.5.1.1 例

以下のようにこのヘッダファイルをインクルードする事で、CCI への準拠と SFR へのアクセスを可能にします。

```
#include <xc.h>
```

2.5.1.2 コンパイラの違い

CCI 適用前の一部の 8 ビットコンパイラは、`<htc.h>` を上記と等価なヘッダとして使いました。CCI 適用前の 16 および 32 ビットコンパイラは、上記と同じ目的のために各種のヘッダを使いました。

2.5.1.3 CCI への移行

```
#include <htc.h>
```

上記は、CCI 適用前の 8 ビットコンパイラで使っていたヘッダファイルです。下記はデバイスファミリに固有のヘッダファイルの例です。

```
#include <p32xxxx.h>
#include <p30fxxxx.h>
#include <p33Fxxxx.h>
#include <p24Fxxxx.h>
#include "p30f6014.h"
```

これらを以下に変更します。

```
#include <xc.h>
```

2.5.2 絶対アドレッシング

CCI では、`__at()` 修飾子を使う事によって、変数と関数を絶対アドレスに配置できます。スタックベース (`auto` およびパラメータ) 変数は `__at()` 指定子を使えません。

2.5.2.1 例

下の例は、2 つの変数と 1 つの関数を絶対アドレス指定します。

```
int modify(int x) __at(0x9D001000);
const char keys[] __at(0xA0000200) = { 'r', 's', 'u', 'd'};
int modify(int x) {
  return x * 2 + 3;
}


Alternatively:
const char keys[] __at(0xA0000200) = { 'r', 's', 'u', 'd'};
__at(0x9D001000)int modify(int x) {
  return x * 2 + 3;
}
```

2.5.2.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、絶対アドレスを指定するために @ シンボルを使いました。

CCI 適用前の 16 および 32 ビットコンパイラでは、オブジェクトのアドレスを指定するために address 属性を使いました。

2.5.2.3 CCI への移行

可能であれば、オブジェクトと関数の絶対アドレス指定は避けるべきです。

XC8 では、絶対アドレスのオブジェクト定義は以下のように変更します。

```
int scanMode @ 0x200;
```

上記を以下に変更

```
int scanMode __at(0x200);
```

XC16/32 では、以下のようにコードを変更します。

```
int scanMode __attribute__(address(0x200)));
```

上記を以下に変更

```
int scanMode __at(0x200);
```

2.5.2.4 注意

現在の XC8 では、__at() および __section() 指定子の両方を 1 つのオブジェクトに適用した場合、__section() 指定子は無視されます。

2.5.3 far オブジェクトおよび関数

__far 修飾子は、変数または関数が「far メモリ」内に配置可能性である事を示すために使えます。far メモリが厳密に何によって構成されるかは、ターゲット デバイスによって異なります。一般的に、このメモリにアクセスするには複雑なコードが必要です。far 修飾子付きのオブジェクトを扱う表現は、比較的低速で大きなサイズのコードを生成します。

この修飾子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

一部のデバイスは far メモリを実装していません。その場合、この修飾子は無視されます。スタックベース (auto およびパラメータ) 変数は __far 指定子を使えません。

2.5.3.1 例

__far を使って修飾した変数と関数の例を下に示します。

```
__far int serialNo;
__far int ext_getCond(int selector);
```

2.5.3.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラは、__far と同じ意味で far 修飾子を使いました。関数は far として修飾できません。

CCI 適用前の 16 ビットコンパイラは、変数と関数の両方に far 属性を使いました。

CCI 適用前の 32 ビットコンパイラは、変数と関数の両方に far 属性を使いました。

 © 2015 Microchip Technology Inc.

2.5.3.3　CCI への移行

8 ビットコンパイラの場合、全ての far 修飾子を以下のように変更します。

```
far char template[20];
```
上記を以下に変更
__far (すなわち __far char template[20];)

16 および 32 ビットコンパイラの場合、全ての far 属性を以下のように変更します。

```
void bar(void) __attribute__ ((far));
int tblIdx __attribute__ ((far));
```

上記を以下に変更

```
void __far bar(void);
int __far tblIdx;
```

2.5.3.4　注意

なし

2.5.4　near オブジェクト

__near 修飾子は、変数または関数が「near メモリ」内に配置可能である事を示すために使えます。near メモリが厳密に何によって構成されるかは、ターゲット デバイスによって異なります。一般的に、このメモリには簡潔なコードでアクセスできます。near 修飾子付きのオブジェクトを扱う表現は、比較的高速で小さなサイズのコードを生成します。

この修飾子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

一部のデバイスは near メモリを実装していません。その場合、この修飾子は無視されます。スタックベース (auto およびパラメータ) 変数は __near 指定子を使えません。

2.5.4.1　例

__near を使って修飾した変数と関数の例を下に示します。

```
__near int serialNo;
__near int ext_getCond(int selector);
```

2.5.4.2　コンパイラの違い

CCI 適用前の 8 ビットコンパイラは、__near と同じ意味で near 修飾子を使いました。関数は near として修飾できません。

CCI 適用前の 16 ビットコンパイラは、変数と関数の両方に near 属性を使いました。

CCI 適用前の 32 ビットコンパイラは、変数と関数の両方に near 属性を使いました。

2.5.4.3 CCI への移行

8 ビットコンパイラの場合、全ての near 修飾子を以下のように変更します。

```
near char template[20];
```

上記を以下に変更
__near (すなわち __near char template[20];)

16 および 32 ビットコンパイラの場合、全ての near 属性を以下のように変更します。

```
void bar(void) __attribute__ ((near));
int tblIdx __attribute__ ((near));
```

上記を以下に変更

```
void __near bar(void);
int __near tblIdx;
```

2.5.4.4 注意

なし

2.5.5 persistent オブジェクト

__persistent 修飾子は、スタートアップ コードによって変数をクリアしない事を示すために使えます。

この修飾子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

2.5.5.1 例

__persistent を使って修飾した変数の例を下に示します。

```
__persistent int serialNo;
```

2.5.5.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラは、__persistent と同じ意味で persistent 修飾子を使いました。

CCI 適用前の 16 および 32 ビットコンパイラは、クリアしない変数を示すために persistent 属性を使いました。

2.5.5.3 CCI への移行

8 ビットコンパイラの場合、全ての persistent 修飾子を以下のように変更します。

```
persistent char template[20];
```

上記を以下に変更
__persistent (すなわち __persistent char template[20];)

16 および 32 ビットコンパイラの場合、全ての persistent 属性を以下のように変更します。

```
int tblIdx __attribute__ ((persistent));
```

上記を以下に変更

```
int __persistent tblIdx;
```

2.5.5.4 注意

なし

© 2015 Microchip Technology Inc.

2.5.6 X および Y データ オブジェクト

__xdata および __ydata 修飾子は、変数が特殊なメモリ領域内に配置可能性である事を示すために使えます。X および Y メモリが厳密に何によって構成されるかは、ターゲット デバイスによって異なります。一般的に、これらのメモリには別々のバスを使って別々にアクセスできます。このようなメモリは、一部の DSP 命令向けによく使われます。

これらの修飾子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

一部のデバイスはこれらのメモリを実装していません。その場合、これらの修飾子は無視されます。

2.5.6.1 例

__xdata と __ydata を使って修飾した 2 つの変数の例を下に示します。

```
__xdata char data[16];
__ydata char coeffs[4];
```

2.5.6.2 コンパイラの違い

CCI 適用前の 16 ビットコンパイラは、変数に xmemory および ymemory メモリ空間属性を使いました。

その他のコンパイラでは、等価な指定子は定義されていません。

2.5.6.3 CCI への移行

16 ビットコンパイラの場合、全ての space 属性 (xmemory または ymemory) を以下のように変更します。

```
char __attribute__((space(xmemory)))template[20];
```

上記を以下に変更
__xdata または __ydata (すなわち __xdata char template[20];)

2.5.6.4 注意

なし

2.5.7 バンク化されたデータ オブジェクト

__bank(num) 修飾子は、変数が特定のメモリバンクに配置可能である事を示すために使えます。num はバンクの番号を表します。バンク指定メモリが厳密に何によって構成されるかは、ターゲット デバイスによって異なります。通常このメモリはデータメモリの一部であり、限られたアドレス幅フィールドでアセンブリ命令の使用を可能にします。

これらの修飾子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

一部のデバイスはバンク化されたデータメモリを実装していません。その場合、この修飾子は無視されます。データバンクの実装数はデバイスごとに異なります。

2.5.7.1 例

__bank( ) を使って修飾した変数の例を下に示します。

```
__bank(0) char start;
__bank(5) char stop;
```

2.5.7.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラは、より制限されるものの、上記と同じメモリ配置を示すために 4 つの修飾子 (`bank0`、`bank1`、`bank2`、`bank3`) を使いました。

その他のコンパイラには、等価な指定子は定義されていません。

2.5.7.3 CCI への移行

8 ビットコンパイラの場合、全ての `bankx` 修飾子を以下のように変更します。

```
bank2 int logEntry;
```

上記を以下に変更
`__bank` (すなわち `__bank(2) int logEntry;`)

2.5.7.4 注意

なし

2.5.8 オブジェクトのアラインメント

`__align(`*`alignment`*`)`指定子は、指定した*`alignment`*値の倍数のメモリアドレスに変数をアラインメントする必要がある事を示すために使えます。*`alignment`* 値は 2 のべき乗値である事が必要です。正の値はオブジェクトの開始アドレスをアラインメントし、負の値はオブジェクトの終了アドレスをアラインメントします。

この指定子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

2.5.8.1 例

`__align()`修飾子を使って終了アドレスを8の倍数にアラインメントした変数と、開始アドレスを 2 の倍数にアラインメントした変数の例を下に示します。

```
__align(-8) int spacer;
__align(2) char coeffs[6];
```

2.5.8.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラはアラインメント機能を実装していません。

CCI適用前の16および32ビットコンパイラでは、変数に`aligned`属性を使いました。

2.5.8.3 CCI への移行

16 および 32 ビットコンパイラの場合、全ての `aligned` 属性を以下のように変更します。

```
char __attribute__((aligned(4)))mode;
```

上記を以下に変更
`__align` (すなわち `__align(4) char mode;`)

2.5.8.4 注意

現在、XC8 はこの機能を未実装です。

 © 2015 Microchip Technology Inc.

2.5.9 EEPROM オブジェクト

__eeprom修飾子は、変数をEEPROM内に配置する必要がある事を示すために使えます。

この修飾子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

一部のデバイスは EEPROM を実装していません。そのようなデバイスにこの修飾子を使うと、警告が生成されます。スタックベース (auto およびパラメータ) 変数は __eeprom 指定子を使えません。

2.5.9.1 例

__eeprom を使って修飾した変数の例を下に示します。

```
__eeprom int serialNos[4];
```

2.5.9.2 コンパイラの違い

CCI適用前の8ビットコンパイラは、一部のデバイスでこの意味を示すためにeeprom修飾子を使いました。

CCI 適用前の 16 ビットコンパイラは、EEPROM 向けに使われるメモリ空間に変数を割り当てるために space 属性を使いました。

2.5.9.3 CCI への移行

8 ビットコンパイラの場合、全ての eeprom 修飾子を以下のように変更します。

```
eeprom char title[20];
```

上記を以下に変更
__eeprom (すなわち __eeprom char title[20];)

16 ビットコンパイラの場合、全ての eedata space 属性を以下のように変更します。

```
int mainSw __attribute__ ((space(eedata)));
```

上記を以下に変更

```
int __eeprom mainSw;
```

2.5.9.4 注意

XC8 は、全ての PIC18 デバイスに対して __eeprom 修飾子を実装しません。その他の 8 ビットデバイスでは、この修飾子は期待通りに機能します。

2.5.10 割り込み関数

`__interrupt(type)`指定子は、関数が割り込みサービスルーチンとして機能する事を示すために使えます。`type` は、割り込み関数に関する情報を示すキーワードのリスト (区切り文字はコンマ) です。

現在利用可能な割り込みの型は以下の通りです。

<empty>
既定値の割り込み関数を実装します。

low_priority
低優先度の割り込み要因に対応する割り込み関数 (XC8 - PIC18 専用)

high_priority
高優先度の割り込み要因に対応する割り込み関数 (XC8)

save(シンボルのリスト)
指定されたシンボルのリストを開始時に保存し、終了時に復元します (XC32)。

irq(*irqid*)
この割り込みに対応する割り込みベクタを指定します (XC32)。

altirq(*altirqid*)
この割り込みに対応する代替割り込みベクタを指定します (XC32)。

preprologue(*asm*)
コンパイラが生成した全ての割り込みコードの前に実行するアセンブリ コードを指定します (XC32)。

shadow
割り込みサービスルーチンでシャドーレジスタを使ってコンテキストを切り換えられるようにします (XC32)。

auto_psv
割り込みサービスルーチンは PSVPAG レジスタを設定し、終了時に復元します (XC32)。

no_auto_psv
割り込みサービスルーチンは PSVPAG レジスタを設定しません (XC32)。

この指定子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

割り込みを実装できないデバイスもあります。そのようなデバイスにこの修飾子を使うと警告が生成されます。`__interrupt` 指定子への引数がターゲット デバイスに対して意味を成さない場合、コンパイラは警告またはエラーを生成します。

2.5.10.1 例

`__interrupt` を使って修飾した関数の例を下に示します。

```
__interrupt(low_priority) void getData(void) {
    if (TMR0IE && TMR0IF) {
        TMR0IF=0;
        ++tick_count;
    }
}
```


© 2015 Microchip Technology Inc.

2.5.10.2　コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、一部のデバイス向けに、`__interrupt` と同じ意味で `interrupt` および `low_priority` 修飾子を使いました。割り込みルーチンは既定値で高優先度でした。

CCI 適用前の 16 および 32 ビットコンパイラでは、割り込み関数を定義するために `interrupt` 属性を使いました。

2.5.10.3　CCI への移行

8 ビットコンパイラの場合、全ての `interrupt` 修飾子を以下のように変更します。

```
void interrupt myIsr(void)
void interrupt low_priority myLoIsr(void)
```

上記をそれぞれ以下に変更

```
void __interrupt(high_priority) myIsr(void)
void __interrupt(low_priority) myLoIsr(void)
```

16 ビットコンパイラの場合、全ての `interrupt` 属性を以下のように変更します。

```
void __attribute__((interrupt,auto_psv,(irq(52)))) myIsr(void);
```

上記を以下に変更

```
void __interrupt(auto_psv,(irq(52)))) myIsr(void);
```

32 ビットコンパイラの場合、`__interrupt()` キーワードは 2 つのパラメータ (ベクタ番号と IPL 値 (オプション)) を使います。`interrupt` 属性を使っているコードは以下のように変更します。

```
void __attribute__((vector(0), interrupt(IPL7AUTO), nomips16))
myisr0_7A(void) {}
void __attribute__((vector(1), interrupt(IPL6SRS), nomips16))
myisr1_6SRS(void) {}
/* Determine IPL and context-saving mode at runtime */
void __attribute__((vector(2), interrupt(), nomips16))
myisr2_RUNTIME(void) {}
```

上記を以下に変更

```
void __interrupt(0,IPL7AUTO) myisr0_7A(void) {}
void __interrupt(1,IPL6SRS) myisr1_6SRS(void) {}
/* Determine IPL and context-saving mode at runtime */
void __interrupt(2) myisr2_RUNTIME(void) {}
```

2.5.10.4　注意

なし

2.5.11 オブジェクトのパッキング

`__pack` 指定子は、メモリギャップを使って構造体メンバーをアラインメントしない事 (または個々の構造体メンバーをアラインメントしない事) を示すために使えます。

この指定子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

一部のコンパイラは、一部のデバイスに対してアラインメント ギャップによる構造体のパディングをサポートしません。そのようなデバイスでは、この指定子を使っても無視されます。

2.5.11.1 例

`__pack`を使って修飾した構造体と、1つのメンバーを明示的にパックした構造体の例を以下に示します。

```
__pack struct DATAPOINT {
    unsigned char type;
    int value;
} x-point;
struct LINETYPE {
    unsigned char type;
    __pack int start;
    long total;
} line;
```

2.5.11.2 コンパイラの違い

`__pack` 指定子は、XC8 で利用可能な新しいCCI 指定子です。デバイスメモリは全てのデータ オブジェクトに対してバイト アドレス指定可能であるため、この指定子は効果を有しません。

CCI 適用前の 16 および 32 ビットコンパイラでは、メモリギャップを使って構造体メンバーをアラインメントしない事を示すために `packed` 属性を使いました。

2.5.11.3 CCI への移行

XC8 の場合、移行のための対応は不要です。

16 および 32 ビットコンパイラの場合、全ての `packed` 属性を以下のように変更します。

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

上記を以下に変更

```
struct DOT
{
    char a;
    __pack int x[2];
};
```

または、必要に応じて構造体全体をパックする事もできます。

2.5.11.4 注意

なし

© 2015 Microchip Technology Inc.

2.5.12 旧式オブジェクトの指定

__deprecate 指定子は、そのオブジェクトが将来性のない旧式オブジェクトである事 (新規設計に使うべきではない事) を示すために使えます。この指定子は、コンパイラの拡張または機能が非サポートになる可能性がある事、または、新しい改良型機能の使用が推奨される事を示すために、コンパイラ ベンダーによって使われます。

この指定子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

2.5.12.1 例

__deprecate キーワードを使う関数の例を下に示します。

```
void __deprecate getValue(int mode)
{
//...
}
```

2.5.12.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラは deprecate 機能を実装していませんでした。

CCI 適用前の 16 および 32 ビットコンパイラでは、できるだけ使用を避けるべきオブジェクトを示すために、deprecated 属性 (スペルが異なるので注意) を使いました。

2.5.12.3 CCI への移行

16 および 32 ビットコンパイラの場合、全ての deprecated 属性を以下のように変更します。

```
int __attribute__(deprecated) intMask;
```

上記を以下に変更

```
int __deprecate intMask;
```

2.5.12.4 注意

なし

2.5.13 セクションへのオブジェクトの割り当て

__section() 指定子は、オブジェクトを名前付きセクション (XC8 の用語では psect) に配置する必要がある事を示すために使えます。一般的にこの指定子は、既存のコンパイラ機能ではアドレス指定できない特殊なリンクをオブジェクトが要求する場合に使います。

この指定子の意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

2.5.13.1 例

__section キーワードを使う変数の例を下に示します。

```
int __section("comSec") commonFlag;
```

2.5.13.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、オブジェクトを新しいセクション (または psect) にリダイレクトするために #pragma psect ディレクティブを使いました。このプラグマと __section() 指定子の違いは以下の通りです。

__section() を使って作成した新しい psect は、プラグマとは異なり、psect のフラグを継承しません (プラグマの場合、このフラグによってオブジェクトがメモリに割り当てられていました)。これは、新しい psect がデータバンクを含む全てのメモリ領域内でリンクできる事を意味します。コンパイラは、新しいセクション内のオブジェクトの位置について何も仮定しません。プラグマを使って新しい psect へリダイレクトしたオブジェクトは、常に同じメモリ領域内 (その領域内の任意のアドレス) でリンクする必要がありました。

__section() 指定子を使うと、異なる psect 内に配置されるようオブジェクトを初期化できます。新しい psect でもオブジェクトは初期化されます。このために、初期値を格納する追加の psect を自動的に割り当てる必要があります (追加 psect の名前 = 接頭辞「i」+ 新しい psect の名前)。プラグマは、初期化されるオブジェクトには使えません。

__section() を使って異なる psect に割り当てたオブジェクトは、プラグマを使った場合とは異なり、スタートアップ コードによってクリアされます。

現在の XC8 コンパイラでは、__section() 指定子を使って作成した全ての新しい psect 向けにメモリを予約し、リンカ オプションを使って割り当てる必要があります。

CCI 適用前の 16 および 32 ビットコンパイラでは、異なるデスティネーション セクション名を示すために section 属性を使いました。__section() 指定子は、この属性と同様に機能します。

2.5.13.3 CCI への移行

XC8 の場合、全ての #pragma psect ディレクティブを以下のように変更します。

```
#pragma psect text%%u=myText
int getMode(int target) {
//...
}
```

上記を以下の __section() 指定子に変更

```
int __section ("myText") getMode(int target) {
//...
}
```

16 および 32 ビットコンパイラの場合、全ての section 属性を以下のように変更します。

```
int __attribute__((section("myVars"))) intMask;
```

を、以下に変更

```
int __section("myVars") intMask;
```


© 2015 Microchip Technology Inc.

2.5.13.4　注意

XC8 の場合、__section() 指定子を割り込み関数と一緒に使う事はできません。

2.5.14　コンフィグレーション ビットの指定

#pragma config ディレクティブは、デバイスのコンフィグレーション ビットを設定するために使えます。このプラグマの形態は以下の通りです。

```
#pragma config setting = state|value
#pragma config register = value
```

setting はコンフィグレーション設定記述子 (例 : WDT)、*state* は記述値 (例 : ON)、*value* は数値です。レジスタ トークンは、コンフィグレーション ワードレジスタ (例 : CONFIG1L) の全体を表現できます。

このディレクティブの意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

2.5.14.1　例

このプラグマを使ってコンフィグレーション ビットを指定する例を以下に示します。

```
#pragma config WDT=ON, WDTPS = 0x1A
```

2.5.14.2　コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、#pragma config が未サポートのターゲットに対して __CONFIG() マクロを使いました。

CCI 適用前の 16 ビットコンパイラでは、コンフィグレーション設定を指定するために各種のマクロを使いました。

CCI 適用前の 32 ビットコンパイラは、#pragma config をサポートしました。

2.5.14.3　CCI への移行

8 ビットコンパイラの場合、全ての __CONFIG() マクロは以下のように変更します。

```
__CONFIG(WDTEN & XT & DPROT)
```

上記を以下に変更

```
#pragma config WDTE=ON, FOSC=XT, CPD=ON
```

#pragma config を既に使っていたい場合、移行は不要です。

16 ビットコンパイラの場合、全ての _FOSC() または _FBORPOR() マクロを以下のように変更します。

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

上記を以下に変更

```
#pragma config FCKSMEM = CSW_ON_FSCM_ON,  FPR = ECIO_PLL16
```

32 ビットコードの場合、移行のための対応は不要です。

2.5.14.4　注意

なし

2.5.15 マニフェスト マクロ

CCI は、コンパイラとターゲット デバイスの特性を明示するマクロの一般形態を定義しています。これらのマクロは、コンパイラまたはターゲット デバイスに基づいて代替のソースコードを条件付きでコンパイルするために使えます。

各種のマニフェスト マクロの詳細を ?2-1 に示します。

表 2-1: CCI が定義するマニフェスト マクロ

名称	意味 (定義されている場合)	例
`__XC__`	MPLAB XC コンパイラを使用	`__XC__`
`__CCI__`	CCI 準拠コンパイラを使用 (CCI は有効)	`__CCI__`
`__XC##__`	## で指定した XC コンパイラを使用 (## は 8、16、32 のいずれか)	`__XC8__`
`__DEVICEFAMILY__`	ターゲット デバイスのファミリを指定	`__dsPIC30F__`
`__DEVICENAME__`	ターゲット デバイスの名前を指定	`__18F452__`

2.5.15.1 例

デバイスが EEPROM を備えているかどうかに応じて条件付きでコンパイルするためのコード例を下に示します。

```
#ifdef __XC32__
void __interrupt(__auto_psv__) myIsr(void)
#else
void __interrupt(low_priority) myIsr(void)
#endif
```

2.5.15.2 コンパイラの違い

これらの CCI マクロの一部は新たに追加された物です (例 : `__CCI__`)。その他の CCI マクロは、CCI 適用前と名称は異なりますが意味は同じです (例 : `__18F452` は `__18F452__` に変更)。

2.5.15.3 CCI への移行

コンパイラ定義マクロを使う全てのコードは見直しが必要です。古いマクロは期待通りに機能しますが、それらは CCI に準拠しません。

2.5.15.4 注意

なし

© 2015 Microchip Technology Inc.

2.5.16 インライン アセンブリ

asm() ステートメントは、アセンブリコードを C 言語コードに挿入するために使えます。引数は、1 つのアセンブリ命令を表す C 言語の文字列リテラルです。当然ですが、引数で指定する命令はデバイスに固有です。

このステートメントの意味については、下の「コンパイラの違い」に記載した CCI 適用前のキーワードを参照してください。

2.5.16.1 例

MOVLW 命令を挿入する例を示します。

```
asm("MOVLW _foobar");
```

2.5.16.2 コンパイラの違い

CCI 適用前の 8 ビットコンパイラでは、インライン アセンブリコードを挿入するために asm() または #asm ...#endasm コンストラクタを使いました。

16 および 32 ビットコンパイラでは、使用する構文に変更はありません。

2.5.16.3 CCI への移行

8 ビットコンパイラの場合、#asm ...#endasm の全てのインスタンスを以下のように変更します (#asm ブロック内の各命令を別々の asm() ステートメントで指定)。

```
#asm
    MOVLW  20
    MOVWF _i
    CLRF   Ii+1
#endasm
```

上記を以下に変更

```
asm("MOVLW 20");
asm("MOVWF _i");
asm("CLRF Ii+1");
```

16 または 32 ビットコードの場合、移行のための対応は不要です。

2.5.16.4 注意

なし

2.6 コンパイラ機能

以下では、ソースコードに直接関係しないコンパイラのオプションと機能について説明します。

2.6.1 CCI を有効にする

ここでは、MPLAB X IDE を使って CCI 適用プロジェクトをビルドする場合を想定します。CCI への準拠を有効にするための MPLAB X IDE プロジェクト プロパティ内のウィジェットは、「Compiler」カテゴリ内の *Use CCI Syntax* です。MPLAB IDE v8 では、[Compiler] タブから同じ名前のウィジェットが利用できます。

IDE を使わない場合のコマンドライン オプションは「`--EXT=cci`」(XC8 向け) または「`-mcci`」(XC16/32 向け) です。

2.6.1.1 コンパイラの違い

このオプションは以前には実装されていませんでした。

2.6.1.2 CCI への移行

このオプションを有効にします。

2.6.1.3 注意

なし

© 2015 Microchip Technology Inc.

第 3 章　プロジェクトをビルドするための手引き

3.1 はじめに

本章には、Microchip 社製 32 ビットデバイス向けプロジェクトのビルド時によく起こる疑問や問題を挙げ、それらに対処するためのヘルプと参照情報を記載しています。各セクションの冒頭にあるリンクをクリックする事で、興味のある項目を素早く見つける事ができます。一部の項目には、複数のセクションからリンクしています。

- コンパイラのインストールと有効化
- コンパイラの起動
- ソースコードの書き方
- アプリケーションを思い通りに実行する方法
- コンパイル プロセスを理解する
- 動作に問題のあるコードの修正

3.2 コンパイラのインストールと有効化

以下では、コンパイラのインストールまたは有効化の際に生じる疑問について詳しく解説します。

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

3.2.1 コンパイラはどのようにインストールして有効にするのですか

XC コンパイラのインストーラは、インストールとライセンスの有効化を同時に行います。インストールガイドとして『Installing and Licensing MPLAB XC C Compilers (DS52059)』が www.microchip.com から入手できます。この文書には、シングルユーザ ライセンスとネットワーク ライセンスに関する詳細と、コンパイラを評価用に有効にする方法が記載されています。

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

コンパイラが正しくインストールされていない、または正しく動作していないと思われる場合、最善の方法は、起こり得る問題を分離するために、MPLAB X IDE を使わずにコンパイラの動作を確認する事です。コマンドラインからコンパイラを起動して、動作を確認します。コードを実際にコンパイルする必要はありません。

端末またはコマンドライン プロンプトから、オプション「-status」を指定してライセンス マネージャ「xclm」を起動します。このオプションを指定すると、ライセンス マネージャはユーザのシステムにインストールされている全ての MPLAB XC ライセンスを表示します。例として 32 ビット Windows の場合、以下をコマンドラインにタイプします (パスは実際のインストール パスに変更します)。

```
"C:\Program Files\Microchip\xc32\v1.00\bin\xclm" -status
```

ライセンス マネージャが起動し、コンピュータ上で利用可能な全ての MPLAB XC コンパイラ ライセンスが表示されます。使用するコンパイラのライセンスが有効として表示される事を確認します (例 : Product:swxc32-pro)。ライセンスが有効でなくてもコンパイラは動作しますが、Free モードでしか使えません。エラーが表示される場合やコンパイラが Free モードである事を示す場合、有効化は成功していません。

3.2.3 同じコンパイラの複数のバージョンをインストールできますか

はい、可能です。本コンパイラとインストール プロセスは、同一コンパイラの複数のバージョンをインストールできるように設計されています。MPLAB X IDE を使う場合、IDE 内のオプションを変更する事で、容易にバージョンを切り換える事ができます (**3.3.4「ビルドに使うコンパイラはどのように選択するのですか」**参照)。

コンパイラは、バージョンに対応する名前が付いたディレクトリにインストールされます。これはインストーラによって指定される既定値ディレクトリです。MPLAB XC32 コンパイラの v1.00 と v1.10 は、以下のように別々のディレクトにインストールされます。

```
C:\Program Files\Microchip\xc32\v1.00\
C:\Program Files\Microchip\xc32\v1.10\
```

 © 2015 Microchip Technology Inc.

3.3 コンパイラの起動

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

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

以下も参照してください。

- コンパイラが特定のメモリ領域を使わないようにする方法を教えてください
- デバッガを使うには、コンパイルする際に何をする必要がありますか
- プロジェクト内でライブラリ ファイルを使う方法を教えてください
- コンパイラはどの最適化レベルを適用しますか

3.3.1 MPLAB IDE 内からコンパイルする方法を教えてください

プロジェクトのセットアップ方法に関しては以下を参照してください。

- **4.5「プロジェクトのセットアップ」** - MPLAB X IDE

3.3.2 コマンドラインからコンパイルする方法を教えてください

コンパイラ ドライバの名前は全ての 32 ビットデバイスに対して `xc32-gcc` です (例 : Windows の場合、ドライバのファイル名は `xc32-gcc.exe` です)。このアプリケーションは、コンパイルのあらゆる局面で呼び出されます。このファイルは、コンパイラ ディストリビューションの「`bin`」ディレクトリに格納されます。各コンパイラ アプリケーション (アセンブラ、リンカ等) を別々に呼び出して実行しないでください。プロジェクトが複数のソースファイルで構成されている場合でも、1 つの命令でコンパイルとリンクを実行できます。

ドライバについては **5.2「コンパイラの起動」**で説明します。複数のドライバをインストールした場合のドライバの選択方法については、**3.3.4 ビルドに使うコンパイラはどのように選択するのですか**を参照してください。ドライバに対するコマンドライン オプションの詳細については、**5.9「ドライバ オプションの説明」**を参照してください。ドライバに渡す事ができるファイルの一覧と説明は **5.2.2「入力ファイルのタイプ」**に記載しています。

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

make ユーティリティ (`make` 等) を使うと、コンパイルは 2 段階のプロセスとして実行されます。最初のステップで中間ファイルを生成し、次のステップで最終的なコンパイルとリンクを実行して 1 つのバイナリ出力を生成します。これについては **5.3.2「マルチステップの C コンパイル」**で説明します。

3.3.4 ビルドに使うコンパイラはどのように選択するのですか

コンパイルおよびインストール プロセスは､同時に複数のコンパイラをインストールできるように設計されています。MPLAB X IDE を使う場合、プロジェクトの作成後にプロジェクト プロパティ内の設定を変更するだけで､異なるコンパイラを使ってそのプロジェクトをビルドできます。

MPLAB X IDE でプロジェクトのビルドに使うコンパイラを選択するには、[Project Properties] ウィンドウ (*File>Project Properties*) を開いて、コンフィグレーション カテゴリ (`Conf:[default]`) を選択します｡MPLAB XC32 コンパイラ バージョンのリストが右端の [Compiler Toolchain] に表示されます｡必要な MPLAB XC32 コンパイラを選択した後に､XC32 グローバル オプションと XC32 コンパイラおよび XC32 リンカ カテゴリを選択する事により､そのコンパイラ向けの制御項目が表示されます。これらの右側にオプションのウィンドウ枠が表示されます。各カテゴリは複数のウィンドウ枠を持ち､これらはウィンドウ枠の上端近くにあるプルダウン メニューから選択できます。

3.3.5 コンパイラの動作モードはどのように変更できますか

コンパイラの動作モード (Free、Evaluation、Standard、PRO) は最適化レベルに基づきます (**第 17 章「最適化」**参照)｡動作モードはコマンドライン オプションとして指定できます (**5.9.7「最適化を制御するためのオプション」**参照)。MPLAB X IDE を使ってビルドする場合､[Project Properties] ウィンドウを開いてコンパイラ名 (C 言語プロジェクトでは xc32-gcc、C++ 言語プロジェクトでは xc32-g++) をクリックし、[Optimization] オプション カテゴリを選択して最適化レベルを設定します (**4.5.3「xc32-gcc (32 ビット C コンパイラ)」**参照)。プロジェクトをビルドする際に、ライセンスを受けた動作モードでは利用できないオプションを選択すると､コンパイラが警告メッセージを出力します。コンパイラは、そのオプションを無効にしたままコンパイルを続けます。

3.3.6 ライブラリのビルド方法を教えてください

他のアプリケーションで再使用できる関数とデータは、それらの C ソースとヘッダファイルを他のプロジェクトにコピーして使えるようにしておくと便利です。あるいは､これらのモジュールをオブジェクト ファイルにビルドし､添付のヘッダファイルと一緒にライブラリ アーカイブにパッケージしておくと､他のアプリケーションに組み込む事ができます。

ライブラリの方がファイル数を少なくできるため便利ですが､ライブラリには保守が必要です。MPLAB XC32 は「*.a」ライブラリ アーカイブを使います。既存のプロジェクトをコンパイラ ツールチェーンの新しいリリースへ移動する際は､ライブラリ オブジェクトをリビルドする必要があるという事に注意が必要です。

コンパイラ ドライバを使う場合､ライブラリに含める全てのファイルをコマンドラインで入力する事により、ライブラリのビルドを開始できます。これらのどのファイルにも `main()` 関数やコンフィグレーション ビット (またはこれに類するデータ) の設定が含まれていない事が必要です。

ユーザ独自のライブラリを作成する方法については、**5.5.1.2「ユーザ定義ライブラリ」**を参照してください。

© 2015 Microchip Technology Inc.

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

コマンドラインで `--help` オプションを使うと、全てのコンパイラ オプションを含むリストが表示されます。本書の **5.9「ドライバ オプションの説明」**にも全てのオプションを記載しています。MPLAB X IDE 内でコンパイルする場合、**4.5「プロジェクトのセットアップ」**を参照してください。

3.3.8 MPLAB IDE 内のビルド オプションについて知りたいのですが、何を見ればよいですか

MPLAB X IDE の [Project Properties] ウィンドウ内にあるウィジェットと制御項目 (XC32 オプション) のほとんどは、コマンドライン ドライバ オプションまたはサブオプションと一対一に対応します。XC32 オプションと、それらに対応するコマンドライン オプションの一覧は **4.5「プロジェクトのセットアップ」**に記載しています。

3.3.9 MPLAB IDE のデバッグビルドでは何が違うのですか

コマンドライン デバッグビルドと MPLAB X IDE デバッグビルドの主な違いは、デバッグを選択した時にプリプロセッサ マクロ `__DEBUG` が 1 に設定される事です。デバッグビルドではない場合、このマクロは定義されません。

このマクロに対して `#ifdef` ディレクティブ等を使う事で、ソース内のコードを条件付きでコンパイルできます (**5.9.8「プリプロセッサを制御するためのオプション」**)。これにより、開発段階のコードにリリース版コードとは異なる挙動を持たせる事ができます。デバッグビルドを実行すると、一部のコンパイラエラーの解析がより容易になります。

MPLAB X IDE では、デバッグビルドの実行時にのみ、デバッガ向けにメモリが予約されます。**3.5.3「デバッガを使うには、コンパイルする際に何をする必要がありますか」**を参照してください。

3.4 ソースコードの書き方

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

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

3.4.1 C 言語の仕様

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

- 式はどのタイミングでキャストすべきですか
- 明示的な型変換によって式の結果が変化してしまう事はありますか
- 英数字以外の文字をプログラムに入力する方法はありますか
- 別のソースファイル内で変数定義を使う方法について教えてください
- 既存のコードを異なるアーキテクチャのデバイスに移植する方法を教えてください

3.4.1.1 式はどのタイミングでキャストすべきですか

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

説明のために以下の例を使います。

```
unsigned long l;
unsigned short s;

s = l;
```

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

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

```
s = (short)l;
```

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

© 2015 Microchip Technology Inc.

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

```
int i, j;
float fl;

fl = i/j;
```

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

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

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

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

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

はい、あります。コンパイラは常に整数拡張を使い、これを無効にする方法はありません (**10.2「整数拡張」**参照)。加えて、2 項演算子のオペランドの型は、C 規格に従って共通の型に変換されます。オペランドの型が変更されると最終的な式の値が変化する可能性があるため、バイナリ演算子を処理する際に適用される C 規格の型変換規則を理解する事が非常に重要です。オペランドの型は、キャストによって手動で変更できます (**3.4.1.1「式はどのタイミングでキャストすべきですか」**参照)。

3.4.1.3 英数字以外の文字をプログラムに入力する方法はありますか

ANSI 規格と MPLAB XC C コンパイラは、ソース キャラクタセットにおいて、文字および文字列リテラル内の拡張キャラクタセットをサポートしません。エスケープシーケンスを使ってこれらの文字を入力する方法については、**8.9「定数の型と書式」**を参照してください。

3.4.1.4 別のソースファイル内で変数定義を使う方法について教えてください

他のソースファイルで定義されている変数が `static` (**9.3.2「`static` 変数」**参照) でも `auto` (**9.4「`auto` 変数のメモリ割り当てとアクセス」**参照) でもない場合、現在のファイルにその変数の宣言を追加する事で、その変数にアクセスできます。この宣言は、以下のように、キーワード `extern` と、変数定義内で指定されている型と変数名で構成します。

```
extern int systemStatus;
```

これは C 言語規格の一部であり、詳細は C 言語の参考書等に記載されています。

現在のファイル内のどの位置で宣言するかによって変数のスコープが決まります。すなわち、ある関数の中で変数を宣言した場合、変数のスコープはその関数の内部に制限されます。関数の外で宣言した場合、その変数には現在のファイル内の全ての関数からアクセスできます。

多くの場合、ヘッダファイル内で宣言し、それらのファイルを C ソースコードにインクルード (`#include`) します (**18.3「プリプロセッサ ディレクティブ」**参照)。

© 2015 Microchip Technology Inc.

3.4.1.5 既存のコードを異なるアーキテクチャのデバイスに移植する方法を教えてください

Microchip 社製デバイスのアーキテクチャには 3 種類あります。8 ビットデバイスはプログラムメモリとデータメモリを分離したハーバード アーキテクチャを採用しています。16 ビットデバイスは改良型ハーバード アーキテクチャを採用し、やはりプログラムメモリとデータメモリ向けに別々のバスを備えます。32 ビットデバイスは MIPS アーキテクチャを採用しています。同じアーキテクチャ ファミリの中の異なるデバイスに移植する場合、アプリケーション コードの変更は最小限で済みます。しかし、異なるアーキテクチャ ファミリのデバイスに移植する場合、コードの大幅な変更が必要になる場合があります。

異なるアーキテクチャ間の移植を容易にするために、CCI (Common C Interface) が開発されました。CCI に準拠したスタイルでコードを書く事により、上位デバイスへの移植が容易になります。CCI の詳細については、**第 2 章「CCI (Common C Interface)」**を参照してください。

3.4.2 デバイスに固有の機能

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

- コンフィグレーション ビットはどのように設定するのですか
- リセットの原因はどのように特定できますか
- リセットの原因はどのように特定できますか
- SFR にはどのようにアクセスできますか
- コンパイラが特定のメモリ領域を使わないようにする方法を教えてください
- デバッガを使うには、コンパイルする際に何をする必要がありますか

3.4.2.1 コンフィグレーション ビットはどのように設定するのですか

これらは、マクロまたはプラグマを使ってコード内で設定します。MPLAB IDE の以前のバージョンでは、ダイアログ内でこれらのビットを設定できました。しかし MPLAB X IDE では、ソースコード内でそれらを指定する必要があります。コンフィグレーション ビットは、ソースコード内で `#pragma` コンフィグを使って設定します。`#pragma` コンフィグの詳細は **7.4.1「コンフィグレーション ビットへのアクセス」**を参照してください。

3.4.2.2 リセットの原因はどのように特定できますか

リセットの原因は、リセット制御 (RCON) レジスタ内のビットに基づいて特定できます。しかし、これらのビットは、`main` の前に実行されるスタートアップ コードによって、すぐに上書きされてしまいます (**14.3「スタートアップ コード」**参照)。実行再開時にアプリケーションがリセットの原因を特定できるよう、C コードからアクセス可能な位置に RCON レジスタの内容を保存しておく事ができます。RCON レジスタに関する詳細については、ターゲット デバイスのデータシートを参照してください。

3.4.2.3 SFR にはどのようにアクセスできますか

メモリマップ上の SFR に割り当てられた変数を定義したヘッダファイルがコンパイラと一緒に配布されます (**7.6「C コードから SFR を使う」**)。これらは C 変数であるため、他の C 変数と同様に扱えます (同じ構文を使ってアクセスできます)。

SFR 内の各ビットにアクセスする事もできます。このために、SFR 全体をマップする構造体内でビットフィールドが利用できます。**8.6.2「構造体内のビットフィールド」**を参照してください。

通常、これらの変数にはデバイス データシートで指定されている名前が割り当てられます。これらの名前が認識されない場合、**3.4.2.4「SFR とその中のビットを表す名前はどのようにして見つけるのですか」**を参照してください。

 © 2015 Microchip Technology Inc.

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

特殊機能レジスタ (SFR) とその中のビットには、そのレジスタ専用に割り当てられた変数を使ってアクセスします (**3.4.2.3「SFR にはどのようにアクセスできますか」**)。しかし、これらの変数の名前がターゲット デバイスのデータシートで指定されている名前と一致しない場合があります。

これらの特殊な変数へのアクセスは、そのデバイスに固有のヘッダファイルによって可能になります。従って、このヘッダファイルを調べれば、それらの変数の名前が分かります。最初に、データシートに記載されている SFR 名を探します。見つからない場合、その SFR が何を表すのかに基づいて探します (多くの場合、ヘッダ内のコメントには、そこに書いてあるマクロが何を実行するのか詳しく書かれています)。

3.4.3 メモリ割り当て

以下は、ソースコードがメモリ割り当てに及ぼす影響についての質問です。

- 変数を特定のアドレスに配置する方法を教えてください
- 関数を特定のアドレスに配置する方法を教えてください
- 変数をプログラムメモリ内に配置する方法を教えてください
- コンパイラが特定のメモリ領域を使わないようにする方法を教えてください
- 予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか

3.4.3.1 変数を特定のアドレスに配置する方法を教えてください

`address` 属性 (**8.12「変数属性」**参照) または CCI の `__at()` コンストラクタ (**2.5.2「絶対アドレッシング」**参照) を使って変数を絶対アドレス指定する方法が最も簡単です。そのように指定したアドレスは、生成されるコード内で変数のシンボルに対して優先的に使われます。アドレスを直接指定するため、オブジェクトの位置を完全に制御できますが、絶対アドレス変数同士がオーバーラップしないよう注意する必要があります。

コンパイラによって配置される auto 変数については **9.4「auto 変数のメモリ割り当てとアクセス」**を参照してください。コンパイラによって配置される非 auto 変数については **9.3.1「非 auto 変数のメモリ割り当て」**を参照してください。コンパイラによって配置されるプログラム空間変数については **9.5「プログラムメモリ内の変数」**を参照してください。

3.4.3.2 関数を特定のアドレスに配置する方法を教えてください

`address` 属性 (**12.2.1「関数属性」**参照) を使って関数を絶対アドレス指定する方法が最も簡単です。そのように指定したアドレスは、生成されるコード内で関数のシンボルに対して優先的に使われます。アドレスを直接指定するため、関数の位置を完全に制御できますが、絶対アドレス関数同士がオーバーラップしないよう注意する必要があります。

3.4.3.3 変数をプログラムメモリ内に配置する方法を教えてください

`const` 修飾子は、その変数が読み出し専用である事を示します。const 修飾子付きの変数のプログラムメモリ (フラッシュ) への割り当てに関しては、**5.9.1「PIC32 専用オプション」**内の `-membedded-data` オプションを参照してください。結果として、auto 変数または関数パラメータを除く `const` 修飾子付きの全ての変数はプログラムメモリ内に配置され、貴重なデータ RAM の使用量を節約できます (**9.5「プログラムメモリ内の変数」**参照)。`const` 修飾付きの変数は絶対アドレス指定する事もできます。

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

ビルド時にデバイスのデータメモリとプログラムメモリの使用レンジを調整するためのオプションはありませんが、リンカスクリプト内でメモリのブロックを予約する事ができます。リンカスクリプトの詳細は『MPLAB XC32 Assembler, Linker and Utilities User’s Guide』(DS50002186) を参照してください。

3.4.4 変数

以下では、プログラム内での変数と型の定義および使用方法に関連する質問に答えます。

- 予期した通りの浮動小数点型結果が得られません。なぜですか
- 変数の個々のビットにはどのようにアクセスできますか
- 変数とマクロの命名方法を教えてください
- 割り込みとメインライン コードでデータを共有する方法を教えてください
- 変数を特定のアドレスに配置する方法を教えてください
- 変数をプログラムメモリ内に配置する方法を教えてください
- 変数をローテートする方法を教えてください
- 変数と関数の位置を調べる方法を教えてください

3.4.4.1 予期した通りの浮動小数点型結果が得られません。なぜですか

まず、MPLAB IDE 内で浮動小数点型変数をウォッチし、それらの型とサイズが定義に合致している事を確認します。MPLAB XC32 では、float および double 型は既定値により 32 ビット浮動小数点型です。long double 型は 64 ビットの浮動小数点型です。

浮動小数点型のサイズは double 型に合わせる事ができます。**8.5「浮動小数点データの型」**を参照してください。

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

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

3.4.4.2 変数の個々のビットにはどのようにアクセスできますか

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

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

1 行目は、整数 var 内の bit (ビット番号) に対応するビットがセットされているかどうかを確認します。2 行目は、var 内の bit に対応するビットをセットします。3 行目は、var 内の bit に対応するビットをクリアします。別の方法として、整数型変数の共用体とビットフィールドを含む構造体 (**8.6.2「構造体内のビットフィールド」**参照) を以下のように定義する事もできます。

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

これにより、var.byte を使って変数 byte にアクセスする事も、var.bitv.b0 ～ var.bitv.b7 を使ってこの変数内の個々のビットにアクセスする事もできます。

© 2015 Microchip Technology Inc.

3.4.4.3 変数とマクロの命名方法を教えてください

C 規格は識別子の先頭から一定数の文字だけが有意であると示唆していますが、その文字数がいくつかは実際に述べておらず、この数はコンパイラごとに異なります。MPLAB XC32 には有意先頭文字数の制限はありませんが、CCI には制限があります (**2.4.5「識別子の有意先頭文字数」**参照)。CCI に準拠する名前を使うと、Microchip 社製デバイスの異なるアーキテクチャ間での移植性が向上します。

2 つの識別子が非有意部でのみ異なる場合、それらは同じオブジェクトとして見なされ、ほぼ確実にコードエラーが発生します。

3.4.5 関数

以下では、関数に関連する質問に答えます。

- 関数の最適なサイズは何で決まりますか
- 関数のサイズを調べる方法を教えてください
- 各関数で使われているリソースを調べる方法を教えてください
- 変数と関数の位置を調べる方法を教えてください
- C 言語では割り込みをどのように使いますか
- 未使用の関数が削除されてしまう事を防ぐ方法を教えてください
- 関数をインライン展開する方法を教えてください

3.4.5.1 関数の最適なサイズは何で決まりますか

一般的に言うと、関数のソースコードは小さくまとめるべきです。そうする事でコードが読みやすくなり、デバッグも容易です。少数のタスクだけを実行する関数は、コードを書く事もデバッグする事も非常に容易です。また、関数を小さくする事で、コーディング エラーの原因になりかねない副作用も抑える事ができます。他方、組み込みプログラミングの領域では、多数の小さな関数を呼び出すとメモリとスタックが不足する恐れがあり、しばしば妥協が強いられます。

関数のサイズは、メモリのページングで問題を生じる可能性があります (**12.5「関数のサイズ制限」**参照)。関数を小さくする事で、リンカはより容易に関数をメモリに割り当てる事ができ、エラーを防げます。

3.4.5.2 未使用の関数が削除されてしまう事を防ぐ方法を教えてください

これは、関数に `__attribute__((keep))` を適用する事で防げます。`keep` 属性は、リンカの `--gc-sections` オプションによって関数が削除される事を (たとえその関数が未使用であっても) 防ぎます。`--gc-sections` オプションを使ったセクション ガベージ コレクションの詳細は、『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

3.4.5.3　関数をインライン展開する方法を教えてください

XC32 コンパイラは、非最適化時にいかなる関数もインライン展開しません。

関数のインライン展開を宣言する事により、関数呼び出しを高速化するよう XC32 コンパイラに指示する事ができます。XC32 がこれを達成できる 1 つの方法は、関数のコードを呼び出し元のコードに統合する事です。これは、関数呼び出しのオーバーヘッドを排除する事によって実行を高速化します。加えて、実際の引数が全て定数である場合、それらの既知の値によってコンパイル時にコードを単純化できます (インライン関数の一部のコードは含めなくて済みます)。コードサイズへの影響を予測する事は困難です。関数のインライン展開によってオブジェクト コードのサイズが増加するか減少するかは、状況に応じて異なります。

関数のインライン展開を宣言するには、以下のように関数の宣言内で inline キーワードを使います。

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

インラインかつ静的な関数への呼び出しを全て呼び出し元に統合する事によってその関数のアドレスが全く使われなくなる場合、その関数自体のアセンブラコードは参照されません。その場合、XC32 はその関数のアセンブラコードを出力しません。各種の理由により、一部の呼び出しは統合できません (例 : 関数の定義よりも前の呼び出しや定義内の再帰的呼び出しは統合できません)。未統合の呼び出しが存在する場合、関数は通常通りにアセンブラコードへコンパイルされます。プログラムが関数のアドレスを参照する場合も、通常通りにコンパイルされます (そのような関数はインライン展開できないため)。

関数のインライン展開を有効にするには、「O1」以上の最適化レベルが必要です。

3.4.6　割り込み

以下では、割り込みおよび割り込みサービスルーチンに関する質問に答えます。

- C 言語では割り込みをどのように使いますか
- 割り込みルーチンを高速にする方法を教えてください
- 割り込みとメインライン コードでデータを共有する方法を教えてください

3.4.6.1　C 言語では割り込みをどのように使いますか

まず､ターゲット デバイスがどの割り込みハードウェアを備えているか知る必要があります。32 ビットデバイスは、複数の分離した割り込みベクタ アドレス領域を備え、優先度スキームを適用します。詳細は **13.2「割り込み動作」**を参照してください。

C ソースコードでは、interrupt 属性を使う事で、関数を割り込みサービスルーチンとして機能させる事ができます。それらの関数は、関数の本文コードを実行する前と後でプログラム コンテキストを保存および復元し、各種のリターン命令を使います。割り込み関数の詳細な書き方については、**13.3「割り込みサービスルーチンの書き方」**を参照してください。割り込みベクタテーブルを実装するには、vector または at_vector 属性を使います。interrupt および vector 属性の使用法を簡素化するため､「sys/attribs.h」ヘッダファイル内で __ISR() マクロが提供されます。

割り込み条件が発生する前に、プログラムは周辺モジュールを正しく設定し、割り込みを有効にしておく必要があります。詳細は **13.9「割り込みの有効化 / 無効化」**を参照してください。

その他の割り込み関連の作業 (割り込みベクタの指定、コンテキストの保存、ネスティング等) と、その他の考慮すべき事項については**第 13 章「割り込み」**を参照してください。

© 2015 Microchip Technology Inc.

3.4.7 アセンブリコード

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

- アセンブリコードと C コードを組み合わせる方法を教えてください
- アセンブリ ソースファイルには命令以外に何が必要ですか
- アセンブリ ソースファイルには命令以外に何が必要ですか
- アセンブリコードから SFR にアクセスする方法を教えてください
- アセンブリコードを書く際に考慮すべき事柄は何ですか

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

理想的には、全ての手書きアセンブリコードは、呼び出し可能な分離したルーチンとして書くべきです。そうする事で、コンパイラが生成するアセンブリコードと手書きアセンブリコードの相互作用をある程度防ぐ事ができます。そのような手書きコードは、分離したアセンブリ モジュールに収めた上で、プロジェクトに追加できます (**16.2「アセンブリ言語と C 変数 / 関数の併用」**参照)。

必要に応じて `asm` 命令 (単純 / 拡張の 2 通りの形態のどちらか) を使う事で、アセンブリコードを C コードにインライン展開できます。この命令の形態に関する説明と例は **16.3「インライン アセンブリ言語の使い方」**に記載しています。

複数の単純な命令をインライン展開するマクロも利用できます (**16.4「定義済みマクロ」**参照)。レジスタの内容とデバイスの状態を変更する複雑なインライン アセンブリ コードは慎重に扱う必要があります。コンパイラによって使われるレジスタに関しては、**第 11 章「レジスタの使用」**を参照してください。

3.4.7.2 アセンブリ ソースファイルには命令以外に何が必要ですか

アセンブリコードには、命令そのもの以外にアセンブラ ディレクティブが必要です。全てのディレクティブの動作に関する説明は、『MPLAB® XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) に記載されています。以下では、2 つのよく使われるディレクティブについて説明します。

全てのアセンブリコードは、`.section` ディレクティブを使って 1 つのセクション内に置く必要があります。これにより、リンカはアセンブリコードをひとまとめに扱ってメモリに配置できます。詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) 内の「Linker Processing」を参照してください。

もう 1 つのよく使われるディレクティブは `.global` です。これは、シンボルを複数のソースファイルからアクセスできるようにするために使います。このディレクティブの詳細についても、上記のユーザガイドを参照してください。

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

大部分の C オブジェクトはアセンブリコードからアクセスできます。C ソース内のシンボルと、そのソースから生成されるアセンブリコード内のシンボルは、互いに対応付けされます。手書きアセンブリコードは、コンパイラが生成したアセンブリコード内の等価シンボルにアクセスする必要があります。詳細は **16.2「アセンブリ言語と C 変数 / 関数の併用」**を参照してください。

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

アセンブリコードからアクセスされる全ての C 変数は、`volatile` で修飾されているかのうように扱われます (**8.10.2「`volatile` 型修飾子」**参照)。C コード内では `volatile` 修飾子を指定する事を推奨します。そうする事で、そのオブジェクトが外部コードからアクセス可能である事を明確に示せます。

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

アセンブリコードから SFR にアクセスするための最も安全な方法は、対応する SFR と同じアドレスを持つシンボルをアセンブリコード内で定義する事です。XC32 コンパイラの場合、xc.h インクルード ファイルは前処理済みアセンブリコードまたは C/C++ コードから使えます。

たとえ SFR へのアクセスを要求するコードであっても、C コードのコンパイルによって生成されたシンボルにアクセスできるという保証はありません。

3.4.7.5 アセンブリコードを書く際に考慮すべき事柄は何ですか

アセンブリコードを手書きする場合、ユーザが管理しなければならない各種の事柄があります。

- 手書きしたアセンブリコードは、全て 1 つのセクション内に置く必要があります。詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) 内の「Linker Processing」を参照してください。
  C コードにインライン展開したアセンブリコードは、コンパイラが生成したアセンブリコードと同じセクション内に置かれます。これを別のセクションへ移動しない事が必要です。
- アセンブリコード内でレジスタに書き込む場合、そのレジスタはコンパイラが生成したコードによって既に使われていない事が必要です。アセンブリコードを別のモジュール内で書く場合、コンパイラは全てのレジスタがそれらのルーチンによって使われると想定するため、これはそれほど大きな問題になりません (**第 11 章「レジスタの使用」**参照)。しかしインライン アセンブリの場合、コンパイラはそのような想定をしません (**16.2「アセンブリ言語と C 変数 / 関数の併用」**参照)。従って、周囲のコンパイラ生成コードで既に使われているリソースをアセンブリコード内で使う (書き込む) 場合、それら全てのリソースを慎重に保存および復元する必要があります。

© 2015 Microchip Technology Inc.

3.5 アプリケーションを思い通りに実行する方法

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

- 出力ポートでグリッチが発生する原因は何ですか
- ブートローダとダウンロード可能アプリケーションのリンク方法を教えてください
- デバッガを使うには、コンパイルする際に何をする必要がありますか
- 割り込みとメインライン コードでデータを共有する方法を教えてください
- コードの悪用を防ぐ方法を教えてください
- Printf を使って周辺モジュールにテキストを出力する方法を教えてください
- コードに遅延を実装する方法を教えてください
- 変数をローテートする方法を教えてください

3.5.1 出力ポートでグリッチが発生する原因は何ですか

ほとんどの場合、グリッチは通常変数を使ってポートの一部のビットまたはポートそのものにアクセスする事によって発生します。これらの変数には `volatile` 修飾子を使う必要があります。**8.10.2「`volatile` 型修飾子」**を参照してください。

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

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

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

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

MPLAB X IDE内でダウンロード可能プロジェクトを使う事で、ブートローダ向けHEXファイルの内容をアプリケーションのコードにマージできます。詳細は MPLAB X IDE ユーザガイドを参照してください。これにより、1 つのイメージ内でブートローダとアプリケーションのコードを含む 1 つの HEX ファイルが得られます。このアプリケーションからオーバーラップに関する警告が出力された場合、メモリがブートローダとダウンロード可能アプリケーションの両方によって使われている事を示していないか確認します。

アプリケーション ノート『PIC32 向けブートローダ』(AN1388) を参照してください。この文書は Microchip 社ウェブサイトからダウンロードできます。

[http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en554836].

3.5.3 デバッガを使うには、コンパイルする際に何をする必要がありますか

MPLAB XC32 コンパイラでビルドしたコードのデバッグには PICkit 3 インサーキット デバッガ、MPLAB ICD 3 インサーキット デバッガ、MPLAB REAL ICE インサーキット エミュレータ等のデバッガが使えます。これらのデバッガは、デバイスのデータメモリとプログラムメモリの一部をデバッグ用に使います。従って、プログラムコードがこれらのデバッグ用リソースを使わないようにする事が重要です。

コンパイラにデバッグ情報を生成させるために、コマンドライン オプション -g (**5.9.6「デバッグのためのオプション」**参照) を使います。このオプションを指定すると、コンパイラはデバッガ向けにメモリを予約し、プログラムコードをこの領域に書き込みません。

MPLAB X IDE で「Debug Run」を実行すると、適切なデバッガ オプションが指定されます。[Run]、[Bulid Project]、[Clean]、[Build] のいずれかを実行する場合、デバッガ オプションは指定されません。

デバイスメモリの一部がデバッガ向けに予約されると、プログラム向けに利用できるメモリは減少します。このため、デバッガを選択した時にコードまたはデータがデバイスに収まりきらなくなる可能性があります。32 ビットデバイスでは、デバッグ用に一部のブート フラッシュメモリを使います。加えて、デバッグツールは一部のデータメモリ (RAM) を使うため、アプリケーションにおける変数の割り当てに影響する可能性があります。

デバッガが使うメモリの位置は MPLAB X、デバッグツール、ターゲット デバイスによって決まります。プロジェクトを IDE の新しいバージョンへ移動すると、必要リソースが変化する可能性があります。このため、デバッガ向けにメモリを手動で予約しない事 (または、どのメモリを使うかコード内で一切想定しない事) が必要です。デバッガ要件の要約が MPLAB IDE ヘルプファイルに記載されています。

コンパイラが予約したリソースがデバッガのメモリ要求を満たしているかどうかは、マップファイルまたはメモリ使用量レポートでブートフラッシュ、アプリケーションフラッシュ、データメモリの使用量を調べる事で確認できます。

MPLAB X IDE でマップファイルを作成するには、[Project Properties] ウィンドウ (*File>Project Properties*) を開いて、リンカカテゴリ (xc32-ld) をクリックします。[Option Categories] で [Diagnostics] を選択します。[Generate map file] の横に、マップファイルのパスと名前を入力します。マップファイルは、プロジェクト フォルダ内に保存する事を推奨します。

[Debug Run] を実行するとマップファイルが生成されます。マップファイルは一般的なテキストビューワを使って閲覧できます。

3.6.14「予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか」も参照してください。

 © 2015 Microchip Technology Inc.

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

割り込みとメインライン コードの両方からアクセスされる変数は、プログラムによって破壊されたり誤読されたりしやすくなります。volatile修飾子(**8.10.2「volatile 型修飾子」**参照) を使う事で、その変数に対して最適化を実行しないようコンパイラに指示できます。これにより、この問題をある程度解消できます。

もう 1 つの問題は、コンパイラ / デバイスがデータにアトミックにアクセスできるかどうかに関係します。32 ビット PIC デバイスでは、これが問題になる事はほとんどありません。アトミックなアクセスとは、1 命令で変数の全体にアクセスする事を意味します。このようなアクセスは割り込まれません。変数がアトミックにアクセスされるかどうかは、アセンブラ リストファイルを見ると分かります。詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。1 命令で変数にアクセスしていれば、それはアトミックです。変数へのアクセス方法はステートメントに応じて様々です。通常、これらの問題を完全に防ぐ最善の方法は、メインライン コードで変数にアクセスする前に割り込みを無効にし、アクセス後に割り込みを再度有効にする事です。詳細は **13.9「割り込みの有効化 / 無効化」**を参照してください。特殊機能レジスタ (SFR) に書き込む際は、SET/CLR/INV レジスタを使います (**7.6「C コードから SFR を使う」**参照)。

3.5.5 コードの悪用を防ぐ方法を教えてください

フラッシュ プログラムメモリを備えた多くのデバイスでは、このメモリの全域または一部の領域を書き込み保護できます。書き込み保護を有効にするには、デバイス コンフィグレーション ビットを正しく設定する必要があります。**7.4.1「コンフィグレーション ビットへのアクセス」**と **2.5.14「コンフィグレーション ビットの指定」**(CCI を使う場合) およびターゲット デバイスのデータシートを参照してください。

また、プログラムメモリの未使用領域を既定値の未プログラム状態のまま残さずに値を書き込んでおく事で、第三者が未使用メモリ領域にコードを書き込む事を防げます。未使用領域には、何らかの命令に対応する値を書き込んでも、全てのビットをセットしても構いません。そうする事で、それらの値は変更できなくなります。プログラムがこれらの領域に達し、そこに書き込まれている値から実行を始めた場合に何が起こるのか (どのような命令が実行されるのか) を考慮する必要があります。

未使用メモリへの値の充填には `--fill` 命令を使います。この命令の使い方については **5.9.10「リンクのためのオプション」**を参照してください。

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

`printf` 関数は 2 つの働きをします。すなわち、1) フォーマット文字列とユーザが指定したプレースホルダに基づいてテキストをフォーマットし、2) このフォーマット済みテキストをデスティネーション (またはストリーム) へ送信 (出力) します。`printf` の出力先は選択できます (LCD、SPI モジュール、USART 等)。

ANSI C 関数 `printf` の詳細については、『32-bit Language Tool Libraries』(DS51685) を参照してください。

`-msmart-io` オプション (**5.9.1「PIC32 専用オプション」**参照) を使うと、`printf` 関数に渡されるフォーマット文字列を静的に解析できます。また、`-Wformat` オプションを使う事で、関数に渡される引数の型が指定のフォーマット文字列に適合しない場合の警告を指定できます (**5.9.5「警告とエラーを制御するためのオプション」**参照)。

ユーザ独自の `printf` 型関数を作成する場合、`format` および `format_arg` 属性を使う必要があります (**12.2.1「関数属性」**参照)。

© 2015 Microchip Technology Inc.

3.5.7 コードに遅延を実装する方法を教えてください

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

Microchip 社は PIC32 デバイスでのソフトウェア遅延の使用を推奨しません。なぜならば L1 キャッシュ、プリフェッチ キャッシュ、フラッシュ待機ステートの設定等、タイミングに影響する変数が多数存在するからです。PIC32 デバイスでは、2 命令サイクル周期でインクリメントするコアタイマをポーリングする方法が使えます。

3.5.8 変数をローテートする方法を教えてください

C 言語にはローテートのための演算子はありませんが、シフトおよびビット単位 OR 演算子を使ってローテートできます。32 ビットデバイスはローテート命令を備えるため、コンパイラはシフトと OR を使ってローテートを表現する事を試み、そのように表現したローテート命令を出力コード内で使います。

CCI の下で符号付き変数を使う場合、**2.4.10「整数値のビット演算」**と **2.4.11「符号付き値の右シフト」**を参照してください。

以下は C コードの例です。

```
int rotate_left (unsigned a, unsigned s)
{
  return (a << s) | (a >> (32 - s));
}
```

コンパイラは、上のコードから以下に類似するアセンブリ命令を生成します。

```
rotate_left:
   subu      $2,$0,$5
   jr        $31
   ror       $2,$4,$2
```


© 2015 Microchip Technology Inc.

3.6 コンパイル プロセスを理解する

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

- Free、Standard、PRO モードの違いは何ですか
- コードを小さくする方法を教えてください
- RAM の使用量を削減する方法を教えてください
- コードを高速にする方法を教えてください
- コンパイラは全ての内容をどのようにメモリに配置するのですか
- 割り込みルーチンを高速にする方法を教えてください
- C 変数はどこまで大きくできますか
- コードにはどの最適化レベルが適用されるのですか
- コンパイラはどのデバイスをサポートしますか
- コンパイラが生成するコードを見る方法を教えてください
- 警告 / エラー メッセージのセットアップ方法を教えてください
- 関数のサイズを調べる方法を教えてください
- 各関数で使われているリソースを調べる方法を教えてください
- 変数と関数の位置を調べる方法を教えてください
- 予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか
- メモリの残量を調べる方法を教えてください
- ライブラリのビルド方法を教えてください
- MPLAB IDE のデバッグビルドでは何が違うのですか
- 未使用の関数が削除されてしまう事を防ぐ方法を教えてください
- プロジェクト内でライブラリ ファイルを使う方法を教えてください
- コンパイラはどの最適化レベルを適用しますか

3.6.1 Free、Standard、PRO モードの違いは何ですか

これらのモード (またはエディション) では、コンパイル時に実行する最適化のレベルが異なります (**第 17 章「最適化」**参照)。Free または Standard モードで動作するコンパイラは、PRO モードがサポートする全てのデバイスをサポートします。どのモードでも、ターゲット デバイスが備える全てのメモリが使えます。モードによって異なるのは、コンパイラが生成する出力コードのサイズと実行速度です。Free モードの出力コードの効率は Standard モードに比べて劣り、Standard モードの出力コードの効率は PRO モードよりも劣ります。

3.6.2 コードを小さくする方法を教えてください

これには各種の方法がありますが、結果はプロジェクトごとに異なります。以下に記載する方法が作業中のコードに適用できるかどうかは、コンパイラが生成したアセンブリコードをアセンブリ リストファイルを使って観察する事で確認できます。リストファイルについては、『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

できるだけ小さなサイズのデータ型を使う事で、それらにアクセスするために必要なコードを小さくできます (RAM の使用量も減少します)。例えば、本コンパイラでは `short` 整数型が使えます。全てのデータ型とサイズについては**第 8 章「サポートするデータ型と変数」**を参照してください。

浮動小数点型には 2 種類のサイズがあり、それらについては後で説明します。可能な限り、浮動小数点型変数の代わりに整数型変数を使います。多くのアプリケーションでは、変数の値をスケーリングする事で、浮動小数点型演算を排除できます。

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

ループまたは条件付きコードを使う場合、以下のように「強い」停止条件を使います。

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

上記は以下よりも推奨されます。

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

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

場合によっては、0 に向かってデクリメントするループカウンタを使った方が、同じ繰り返し数で 0 からカウントアップするよりも効率的です。その場合、上記は以下のように書き換える事ができます。

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

コードをできるだけ小さくするには、使用中のコンパイラ エディションで利用できる全ての最適化機能を有効にする必要があります (**第 17 章「最適化」**参照)。PRO エディションの場合、`-Os` オプション (**5.9.7「最適化を制御するためのオプション」**参照) を使うとコードのサイズを最適化できます。PRO 以外のエディションでは、利用可能な最高の最適化レベルを選択します。

ターゲット デバイスが圧縮 ISA モード (MIPS16 または microMIPS 等) をサポートする場合、それらが使えないか検討します。`-mips16` または `-mmicromips` オプションをプロジェクトに適用する事で、これらのモードをコンパイラの既定値にできます。`mips16` または `micromips` 関数属性を使う事で、関数レベルでのモードを変更します。ライブラリの最適化 / 圧縮済みバージョンをリンカオプションで選択する事もできます。コンパイラが何を最適化してくれるのか理解する事で、コンパイラによる最適化を上手に活用し、コンパイラが最適化してくれる C コードをわざわざ手作業で最適化するといった時間の無駄を省けます。例えば、4 の乗算演算を 2 ビットのシフト演算に変更する必要はありません。この種の最適化はコンパイラが検出してくれます。

 © 2015 Microchip Technology Inc.

3.6.3 RAM の使用量を削減する方法を教えてください

global または static 変数の代わりに auto 変数が使えないか検討します。同時にアクティブにならない auto 変数同士はメモリの割り当てを共有できます。auto 変数のメモリ割り当てはスタックで行います (**9.4「auto 変数のメモリ割り当てとアクセス」**参照)。

関数との間で大きなオブジェクトを授受するよりも、それらのオブジェクトを参照するポインタを授受した方が効率的です。これは特に、大きな構造体を授受する場合に効果的です。

プログラム全体を通して変更する必要のないオブジェクトは、const 修飾子を使ってプログラムメモリ内に配置できます (**9.5「プログラムメモリ内の変数」**参照)。これにより貴重な RAM を節約できますが、実行は遅くなります。

3.6.4 コードを高速にする方法を教えてください

実行速度はコードのサイズに大きく依存します。従って多くの場合、コードサイズの削減を図る事で、実行時間が短縮します。コードサイズの削減方法については、**3.6.2「コードを小さくする方法を教えてください」**と **3.6.6「割り込みルーチンを高速にする方法を教えてください」**を参照してください。しかし、コードサイズは増加するものの、一部のシーケンスを高速化できる方法があります。

一部のコンパイラ エディション (**第 17 章「最適化」**参照) では、実行速度を最適化するために -O3 オプションが使えます (**5.9.7「最適化を制御するためのオプション」**参照)。このオプションは、場合によっては、より高速でありながらよりサイズの大きな出力を採用します。

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

3.6.5 コンパイラは全ての内容をどのようにメモリに配置するのですか

ほとんどの場合、アセンブリ命令とコードおよびデータの両方に関係するディレクティブは、複数のセクションにグループ分けされて、デバイスメモリを表現する複数のコンテナ内に配置されます。-ai オプションを使うと、オブジェクトが配置されたセクションに関する情報をアセンブラ リストファイル内で見る事ができます。

例外として、絶対アドレス指定された変数は定義時に指定されたアドレスに配置され、セクションには配置されません。絶対アドレス変数を設定するには、address() 属性を使います (**8.12「変数属性」**参照)。

3.6.6 割り込みルーチンを高速にする方法を教えてください

割り込みコードに関しては、**3.6.2「コードを小さくする方法を教えてください」**に記載した提案を参考にしてください。多くの場合、コードは小さいほど高速です。

コンパイラは、ISR 内に書かれたコードに加えて、コンテキストを切り換えるためのコードを生成します。これは、割り込みの発生直後とリターン直後に実行されます。割り込みの処理時間にはこれらの実行時間も含みます。このコードは、ISR 内で使うレジスタだけを保存するよう最適化されます。このため、ISR 内で使うレジスタの数が少ないほどコンテキスト切り換えコードの実行時間は短くなります。

一般的に、コードが単純なほど要求リソースは減少します。割り込みコード内でコンパイラがどのレジスタを使っているかは、アセンブリ リストファイルを使って調べる事ができます。リストファイルについては『MPLAB® XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

ISR から他の関数を呼び出すと、関数呼び出しによる追加のオーバーヘッドが生じるだけでなく、コンパイラは汎用レジスタを (その関数によって使われているかどうかに関係なく) 全て保存します。これを防ぐには、ISR 内ではフラグをセットするだけでリターンし、メインライン コードでフラグをチェックする事によって割り込みを処理します。このようにして複雑な割り込み処理コードを ISR の外に出す事で、ISR によるレジスタの使用を防ぐ事ができます。割り込みとメインライン コードが共有する変数には、常に `volatile` 修飾子 (**8.10.2「volatile 型修飾子」**参照) を使います (**3.5.4「割り込みとメインライン コードでデータを共有する方法を教えてください」**参照)。

3.6.7 C 変数はどこまで大きくできますか

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

この疑問に答えるには、その変数がどのメモリ空間に配置されるのか知る必要があります。既定値の `-membedded-data` オプションを使うと、`const` で修飾されたオブジェクトはプログラムメモリ内に配置され、その他のオブジェクトはデータメモリ内に配置されます。プログラムメモリ内のオブジェクトのサイズについては、**9.5.1「const 変数のサイズの制限」**を参照してください。データメモリ内のオブジェクトは、大まかに auto 変数と非 auto 変数に分類されます。これらのサイズについては、**9.3.1「非 auto 変数のメモリ割り当て」**と **9.3.3「非 auto 変数のサイズの制限」**を参照してください。

3.6.8 コードにはどの最適化レベルが適用されるのですか

利用できるコードの最適化レベルは、コンパイラのエディションによって異なります (**第 17 章「最適化」**参照)。最適化オプションについては **5.9.7「最適化を制御するためのオプション」**を参照してください。

3.6.9 コンパイラはどのデバイスをサポートしますか

通常、新型デバイスはコンパイラのリリースごとに追加されます。お使いのコンパイラ リリースがサポートするデバイスについては、readme ファイルを参照してください。

 © 2015 Microchip Technology Inc.

3.6.10 コンパイラが生成するコードを見る方法を教えてください

オプションを指定してアセンブリ リストファイルをセットアップする事で、コードに関する大量の情報をファイルに保存できます。この情報には、プログラムのほぼ全体に対応するアセンブリ出力 (プログラムにリンクしたライブラリ ルーチンを含む)、セクション情報、シンボルリスト等が含まれます。

リストファイルは以下の方法で生成できます。

- コマンドラインでオプションを指定して、基本的なリストファイルを生成します。
  `-Wa, -a=`*`projectname`*`.lst`
- MPLAB X IDE を使う場合、プロジェクトを右クリックし、[Properties] を選択します。[Project Properties] ウィンドウ内の [Categories] の下で [xc32-as] をクリックします。[Option categories] から [Listing file options] を選択し、[List to file] がチェックされている事を確認します。

アセンブリ リストファイルの既定値の拡張子は「`.lst`」です。

リストファイルについては『MPLAB® XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

3.6.11 関数のサイズを調べる方法を教えてください

関数のサイズ (その関数向けに生成されるアセンブリコードのサイズ) は、アセンブリ リストファイルで調べる事ができます。アセンブリ リストファイルの作成方法については、**3.6.10「コンパイラが生成するコードを見る方法を教えてください」**を参照してください。

3.6.12 各関数で使われているリソースを調べる方法を教えてください

アセンブリ リストファイルには、ライブラリ関数を含む各 C 関数に関する情報が含まれています。アセンブリ リストファイルの作成方法については、**3.6.10「コンパイラが生成するコードを見る方法を教えてください」**を参照してください。

関数呼び出しに関する情報は、MPLAB X IDE 内のコールグラフ (*<u>Window>Output>Call Graph</u>*) で見る事ができます。コールグラフはデバッグモード中にのみ見る事ができます。関数を右クリックし、[Show Call Graph] を選択すると、その関数の呼び出し元と呼び出し先が表示されます。

関数が使う auto/ パラメータ / テンポラリ変数は、他の関数からのそれらの変数とオーバーラップする可能性があります。これは、これらの変数がコンパイラによってコンパイルされたスタックに格納されるためです (**9.4「`auto` 変数のメモリ割り当てとアクセス」**参照)。

3.6.13 変数と関数の位置を調べる方法を教えてください

変数と関数の配置先は、アセンブラが生成したアセンブリ リストファイルまたはリンカが生成したマップファイルで調べる事ができます。マップファイルはグローバル シンボルだけを示しますが、アセンブリ リストファイルはローカルシンボルを含む全てのシンボルを示します。

C 識別子とアセンブリコード内のシンボルは互いに対応付けされており、これらのシンボルはマップファイルとリストファイルの両方で示されます。変数に対応するシンボルには、その変数の最下位バイトのアドレスが示されます。関数に対応するシンボルには、その関数向けに生成された最初の命令のアドレスが示されます。

アセンブリ リストファイルとリンカ マップファイルについては『MPLAB® XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

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

ほとんどの変数と関数は、リンカスクリプト内で定義されているセクション内に配置されます。リンカスクリプトの詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。しかし、一部の変数と関数はアドレスレンジ内のどこかにリンクされるのではなく、特定のアドレスに明示的に配置されます (**3.4.3.1 変数を特定のアドレスに配置する方法を教えてください** と **3.4.3.2 関数を特定のアドレスに配置する方法を教えてください** 参照)。

アセンブリ リストファイルを調べる事で、オブジェクトとコードを保持しているセクションの名前を特定できます。マップファイル内のリンカ オプションを調べる事で、セクションが明示的にリンクされているのか、それともクラス内のどこかにリンクされているのか判別できます。これらのファイルの詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

3.6.15 メモリの残量を調べる方法を教えてください

メモリ使用量の概要は、コンパイルした後に `--report-mem` オプションを使ってコンパイラから取得するか、MPLAB X IDE の [Dashboard] ウィンドウで見る事ができます。これらはメモリの使用量と残量を示しますが、そのメモリが連続した 1 ブロックなのか、それとも複数の小さなチャンクに分割されているのか示しません。小さなブロックに分割された空きメモリは、大きなサイズのオブジェクト用には使えません。このため、オブジェクトに対して空きメモリの総量が明らかに十分であっても、メモリ不足エラーが発生する場合があります。

リンカマップ ファイルを調べる事で、各リンカクラス内で利用可能なメモリ残量を正確に特定できます。メモリにページ区分が存在する場合、このファイルは、そのクラス内で最大の連続したブロックも示します。マップファイルの詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

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

ユーザ独自のライブラリ ファイルをビルドする方法については、**3.3.6「ライブラリのビルド方法を教えてください」**を参照してください。コンパイル時に、コンパイラは自動的に全ての適用可能標準ライブラリをビルドプロセスにインクルードします。従って、ユーザがこれらのファイルを制御する必要はありません。

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

```
xc32-gcc -mprocessor=32MZ2048ECH100 main.c int.c mylib.a
```

MPLAB IDE を使ってプロジェクトをビルドする場合、プロジェクト内に表示される「Libraries」フォルダに 1 つまたは複数のライブラリ ファイルを追加します。それらのファイルは、追加された順番に従って検索されます。IDE は、ビルド シーケンス中の適切な時点で、それらのファイルを確実にコンパイラに渡します。

 © 2015 Microchip Technology Inc.

3.6.17 コンパイラはどの最適化レベルを適用しますか

利用できる最適化レベルは、コンパイラのエディションによって決まります (**第 17 章「最適化」**参照)。最適化オプションについては、**5.9.7「最適化を制御するためのオプション」**を参照してください。

3.6.18 デバッガを選択するとメモリが不足します。どうしてですか

ハードウェア ツール デバッガ (PICkit 3 インサーキット デバッガ、MPLAB ICD 3 インサーキット デバッガ、MPLAB REAL ICE インサーキット エミュレータ等) を使う場合、オンボード デバッグ エグゼクティブを格納するためのメモリが必要です。

3.7 動作に問題のあるコードの修正

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

- 警告 / エラー メッセージのセットアップ方法を教えてください
- プログラム内でエラーまたは警告の原因となっているコードを見つける方法を教えてください
- 不要な警告が生成されないようにする方法を教えてください
- LED を点滅させる事すらできません。どうしてですか
- 割り込み使用時の変数の破損とコードエラーの原因は何ですか
- コンパイラの起動
- 割り込み使用時の変数の破損とコードエラーの原因は何ですか
- 予約したメモリにオブジェクトが割り当てられてしまいます。なぜですか

3.7.1 警告 / エラー メッセージのセットアップ方法を教えてください

メッセージ出力の制御については **5.9.5「警告とエラーを制御するためのオプション」**を参照してください。

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

エラーがソースコードに関係する構文エラーであれば、ほとんどの場合、コンパイラは問題を起こしているコード行を示すメッセージを生成します。MPLAB IDE 内でコンパイルしている場合、メッセージをダブルクリックすると、エディタは問題のある行へジャンプします。しかし、問題のコードがいつも簡単に見つかるとは限りません。

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

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

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

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

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

各メッセージの冒頭に表示されるカッコ内の右側に、そのメッセージを生成したアプリケーションの名前が表示されます。エラーを生成したアプリケーションを知る事で、問題の解析が容易になります。コンパイラ アプリケーションの名前は**第 4 章「XC32 ツールチェーンと MPLAB X IDE」**に記載しています。

© 2015 Microchip Technology Inc.

コンパイラが生成したアセンブリコードは、アセンブリ リストファイルで見る事ができます。リンカがオブジェクトをどこに配置しようとするかは、マップファイルを見ると分かります。リストファイルとマップファイルの詳細は『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。

3.7.3 不要な警告が生成されないようにする方法を教えてください

警告はコードエラーを生じる可能性がある状況を示します。エラーが報告された場合、必ずコードをチェックして、それがエラーの原因となり得るかどうか確認します。多くの場合、警告が報告されてもコードは有効です。この場合、以下の方法で警告を抑止できます。

- オプションの `-Wno-` バージョンを使う事で特定の警告を抑止できます。
- `-w` オプションを使うと、全ての警告を抑止できます。
- MPLAB X IDE では、各ツールカテゴリの下の [Project Properties] で警告を抑止します。[Suppressible Messages] タブ ([Tool Options] ウィンドウで [Embedded] ボタンをクリックして開く) も参照してください。

詳細は **5.9.5「警告とエラーを制御するためのオプション」**を参照してください。

3.7.4 LED を点滅させる事すらできません。どうしてですか

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

- デバイスのコンフィグレーション レジスタが **7.4.1「コンフィグレーション ビットへのアクセス」**で説明したように正しく設定されている事を確認します。また、これらのレジスタ内の全てのビットが既定値状態のままではなく明示的に指定されている事も確認します。全ての設定機能は、デバイスのデータシートに記載されています。例えば、オシレータソースを指定するコンフィグレーション ビットの設定が正しくない場合、デバイスクロックが動作すらしない事もあります。
- 内部オシレータを使っている場合、コンフィグレーション ビットに加えて、オシレータの周波数とモードを設定するために初期化が必要な SFR があるかもしれません。詳細は、**7.5「ID の格納位置」**とデバイス データシートを参照してください。
- ウォッチドッグ タイマによってデバイスがリセットする事を防ぐため、コンフィグレーション ビットでタイマを OFF にするか、コード内でタイマをクリアします。ウォッチドッグ タイマはライブラリ関数を使って操作できます。詳細は『32-bit Language Tool Libraries』(DS51685) を参照してください。デバイスがリセットしている場合、プログラム内の LED を点滅させるコード行まで到達できません。テストプログラムが動作するようになるまで、デバイスリセットの原因となり得る他の全ての機能を OFF にします。
- ポートビットに対応するデバイスピンは、しばしば他の周辺モジュールと多重化されています。例えば、1 本のピンをポート内の 1 つのビットに接続する事も、アナログ入力として使う事も、コンパレータ出力として使う事もできます。LED に接続したピンが使用中のポートに内部で接続されていなければ、LED は期待通りに動作しません。デバイス データシート内のポート機能テーブルには各ピンの全ての用途が記載されているため、ピンを共有している周辺モジュールを特定するために役立ちます。

3.7.5 割り込み使用時の変数の破損とコードエラーの原因は何ですか

通常これは、割り込みとメインライン コードの両方によって使われる変数が存在する事によって発生します。コンパイラが変数へのアクセスを最適化する場合、またはアクセスが割り込みルーチンによって中断される場合、変数が破損する恐れがあります。詳細は **3.5.4「割り込みとメインライン コードでデータを共有する方法を教えてください」**を参照してください。

NOTE:

© 2015 Microchip Technology Inc.

第 4 章　XC32 ツールチェーンと MPLAB X IDE

4.1 はじめに

MPLAB X IDE に以下の 32 ビット言語ツールを組み合わせる事により、PIC32 MCU ファミリデバイス向けのアプリケーション コードを GUI 環境で開発できます。

- MPLAB XC32 C コンパイラ
- MPLAB XC32 アセンブラ
- MPLAB XC32 オブジェクトリンカ
- MPLAB XC32 オブジェクト アーカイバ / ライブラリアンとその他の 32 ビット ユーティリティ

本章の内容は以下の通りです。

- MPLAB X IDE とツールのインストール
- MPLAB X IDE のセットアップ
- MPLAB X IDE プロジェクト
- プロジェクトのセットアップ
- プロジェクト例

4.2 MPLAB X IDE とツールのインストール

MPLAB X IDE と 32 ビット言語ツールを使うには、以下をインストールする必要があります。

- MPLAB X IDE (Microchip 社ウェブサイトから無償で入手可能)
- MPLAB XC32 C/C++ コンパイラ (全ての 32 ビット言語ツールを含む)
  このコンパイラには無償版(FreeおよびEvaluationエディション)と有償版(Standard および Pro エディション) があり、どちらも Microchip 社ウェブサイトから入手できます。

32 ビット言語ツールは、既定値により、以下のディレクトリにインストールされます。

- Windows OS の場合 : `C:\Program Files\Microchip\xc32\`*`x.xx`*
- Mac OS の場合 : `Applications/microchip/xc32/`*`x.xx`*
- Linux OS の場合 : `/opt/microchip/xc32/`*`x.xx`*

`x.xx` はバージョン番号です。

以下に示す各ツールの実行ファイルは、「`bin`」サブディレクトリに保存されます。

- C コンパイラ : `xc32-gcc.exe`
- アセンブラ : `xc32-as.exe`
- オブジェクトリンカ : `xc32-ld.exe`
- オブジェクト アーカイバ / ライブラリアン : `xc32-ar.exe`
- その他のユーティリティ : `xc32-`*`utility`*`.exe`

全てのデバイス インクルード (ヘッダ) ファイルは、「/pic32mx/include/proc」サブディレクトリに保存されます。これらのファイルは、「xc.h」ヘッダファイルをインクルード (`#include`) する際に自動的に組み込まれます。

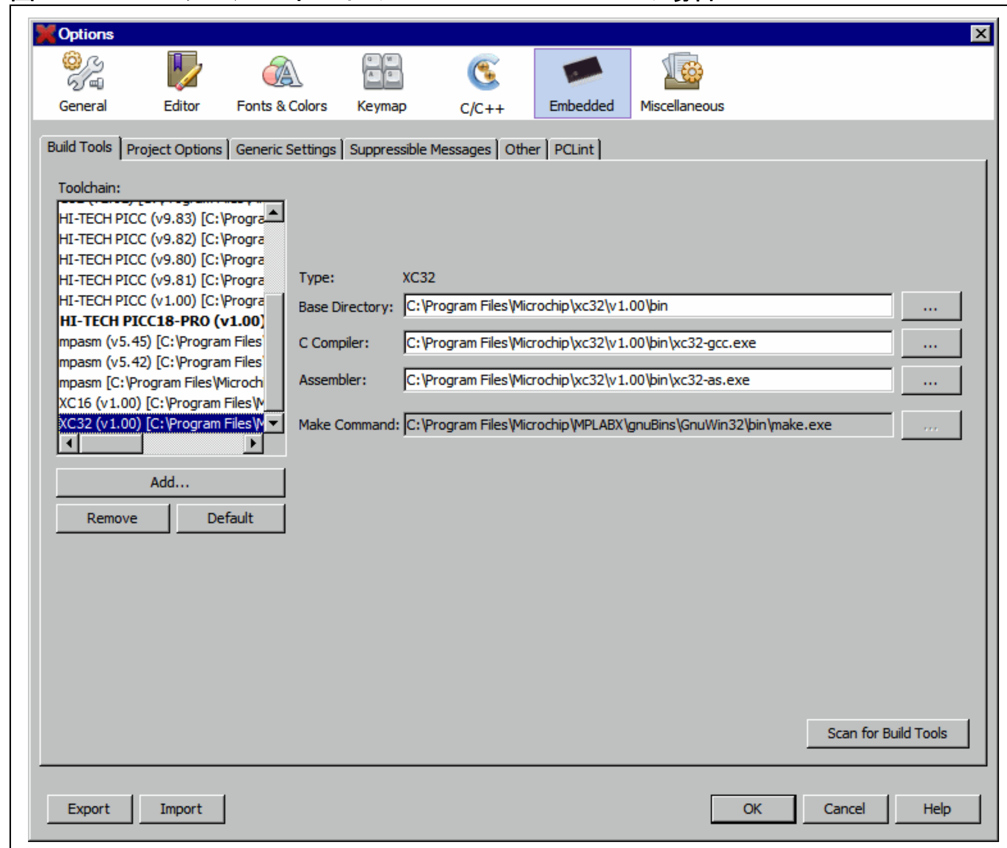
サンプルコードは「examples」ディレクトリに保存されます。

4.3 MPLAB X IDE のセットアップ

MPLAB X IDE を PC にインストールした後に起動し、以下の手順で 32 ビット言語ツールを選択します。

1. MPLAB X IDE のメニューバーで *Tools>Options* を選択して [Options] ダイアログを開きます。[Embedded] ボタンをクリックし、[Build Tools] タブを選択します。
2. [Tool Collection:] の下で [XC32...] をクリックします。パスが実際のインストール場所を正しく指している事を確認します。
3. **[OK]** をクリックします。

図 4-1: XC32 ツールスイートのパス - Windows OS の場合

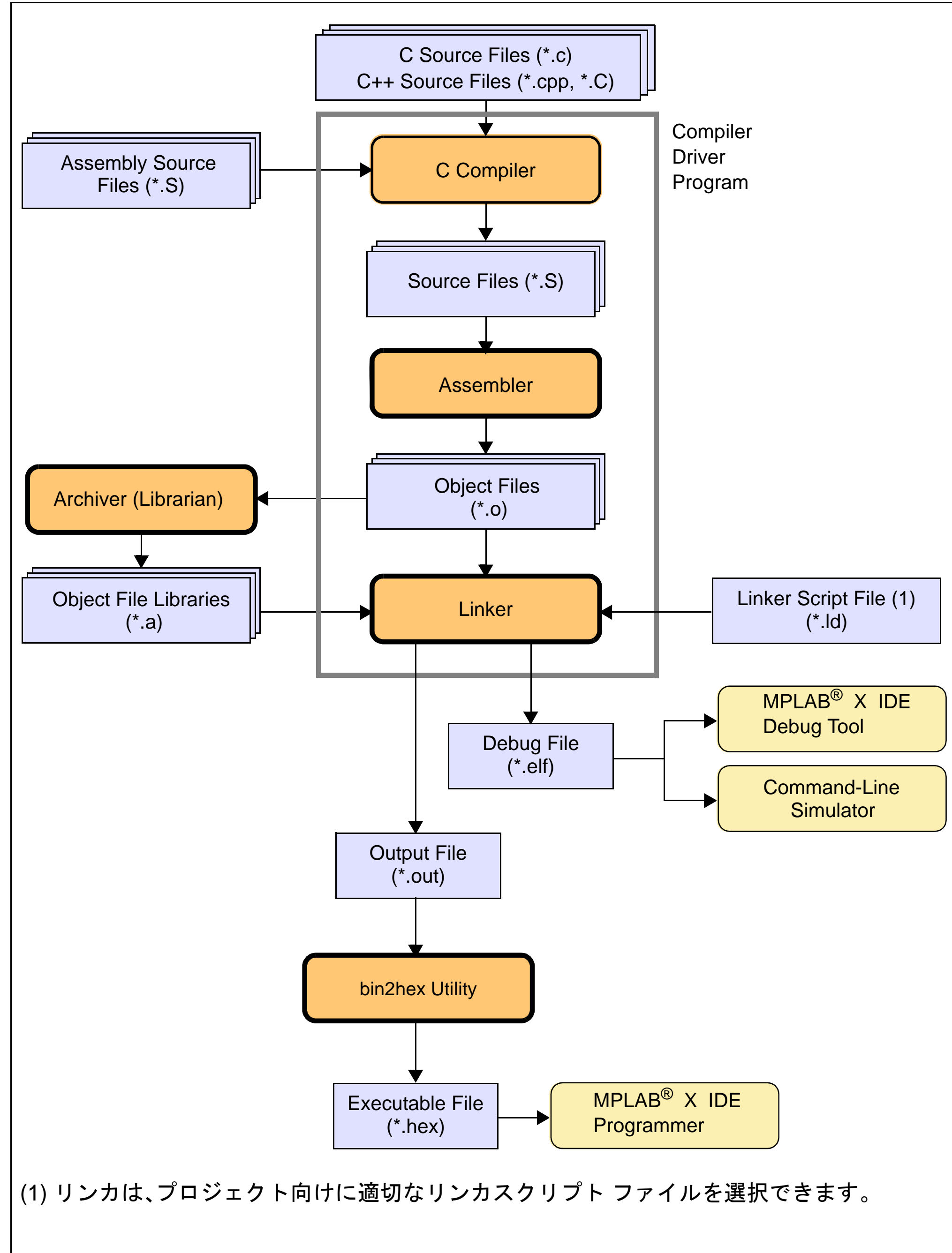


© 2015 Microchip Technology Inc.

4.4 MPLAB X IDE プロジェクト

MPLAB X IDE におけるプロジェクトとは、アプリケーションのビルドに必要なファイル一式の事であり、それらのファイルは各種のビルドツールに関連付けられています。以下に一般的な MPLAB X IDE プロジェクトを示します。

図 4-2: コンパイラとプロジェクトの関係



(1) リンカは、プロジェクト向けに適切なリンカスクリプト ファイルを選択できます。

この図では、C ソースファイル (*.C) をコンパイラへの入力として示しています。コンパイラは、アセンブラに入力するソースファイル (*.S) を生成します。コンパイラの詳細については、コンパイラのマニュアルを参照してください。

図では、アセンブリ ソースファイル (*.S) を C プリプロセッサへの入力として示しています。アセンブラは、リンカまたはアーカイバに入力するオブジェクト ファイル (*.o) を生成します。アセンブラの詳細については、アセンブラのマニュアルを参照してください。

アーカイバ / ライブラリアンを使う事で、オブジェクト ファイルをライブラリにアーカイブできます。アーカイバの詳細については、アーカイバ / ライブラリアンのマニュアルを参照してください。

リンカはオブジェクト ファイル (*.o)、任意のライブラリ ファイル (*.a)、リンカスクリプト ファイル (*.ld) (汎用リンカスクリプトは自動的に追加される) を使ってプロジェクト出力ファイル (*.out) を生成します。

リンカは以下の 2 つの出力ファイルを生成します :

(1) デバッグファイル (`.elf`) - シミュレータとデバッグツールが使用

(2) プロダクション出力ファイル (`.out`) - 実行ファイル (`.hex`) を生成するために bin2hex ユーティリティに入力。

リンカスクリプト ファイルの詳細とオブジェクトリンカの使い方については、リンカのマニュアルを参照してください。

プロジェクトおよび関連するワークスペースの詳細は、MPLAB X IDE のマニュアルを参照してください。

© 2015 Microchip Technology Inc.

4.5 プロジェクトのセットアップ

MPLAB X IDE プロジェクトの最初のセットアップには、内蔵のプロジェクト ウィザード (*File>New Project*) を使います。このウィザードでは、32 ビット言語ツールを使う言語ツールスイートを選択できます。このウィザードと MPLAB X IDE プロジェクトの詳細は、MPLAB X IDE のマニュアルを参照してください。

プロジェクトをセットアップした後は、MPLAB X IDE 内でツールのプロパティを設定できます。

1. MPLAB X IDE メニューバーから *File>Project Properties* 選択すると、プロジェクトのビルド オプションを設定 / 確認するためのウィンドウが開きます。
2. 「Conf:[*default*]」の下で、[Tool Collection] からセットアップするツールを選択します。
   - XC32 (グローバル オプション)
   - xc32-as (32 ビットアセンブラ)
   - xc32-gcc (32 ビット C コンパイラ)
   - xc32-g++ (32-bit C++ Compiler)
   - xc32-g++

4.5.1 XC32 (グローバル オプション)

全ての 32 ビット言語ツールに対するグローバル オプションを設定します。「ページ機能オプション」も参照してください。

表 4-1: XC32 (グローバル オプション) の全てのオプション カテゴリ

オプション	概要	コマンドライン
Use legacy lib	v1.12 以前のフォーマットのライブラリを使う場合にチェックを付けます。新しい (HI-TECH) ライブラリ フォーマットを使う場合はチェックを外します。	`-legacy-libc`
Don’t delete intermediate files	中間生成ファイルを削除せずにオブジェクト ディレクトリに格納します。それらのファイルにはソースファイルに基づく名前が付けられます。	`-save-temps=obj`
Use Whole-Program and Link-Time Optimizations	この機能を有効にすると、ビルドに対して以下の制約を適用します。 - ファイルごとのビルド設定を無視する - インクリメンタル ビルドしない (フルビルドのみ)	

4.5.2 xc32-as (32 ビットアセンブラ)

MPLAB X IDE 内でコマンドライン オプションの一部を指定できます。カテゴリを選択してからアセンブラ オプションを設定します。その他のオプションについては、MPLAB XC32 アセンブラのマニュアルを参照してください。「ページ機能オプション」も参照してください。

表 4-2: xc32-as の汎用オプション カテゴリ

オプション	概要	コマンドライン
Have symbols in production build	MPLAB X 内でソースレベル デバッグ向けのデバッグ情報を生成します。	`--gdwarf-2`
Keep local symbols	チェックを付けると、ローカルシンボル (`.L` で始まる大文字のみのラベル) を保持します。チェックを外すと、ローカルシンボルを破棄します。	`--keep-locals`
Exclude floating-point library	浮動小数点演算のサポートを必要としないアプリケーションでは、このライブラリを除外する事でコードサイズを削減できます。	`-mno-float`
Preprocessor macro definitions	コンパイラの `-D` オプションにより、プロジェクトに固有のプリプロセッサ マクロ定義を渡します。	
Assembler symbols	シンボル「*sym*」を値「*value*」に定義します。	`--defsym` *sym=value*
Preprocessor Include directories	MPLAB X プロジェクト ディレクトリからの相対パスです。	

表 4-2: xc32-as の汎用オプション カテゴリ (続き)

オプション	概要	コマンドライン
Assembler Include directories	MPLAB X プロジェクト ディレクトリからの相対パスです。ディレクトリをディレクトリのリストに追加します。アセンブラは、このリストを使って、`.include` ディレクティブで指定されたファイルを検索します。検索パスに含めるディレクトリは、必要に応じていくつでも追加できます。現在作業中のディレクトリが必ず最初に検索され、その後に `-I` ディレクトリが指定された順番 (左から右) に検索されます。	`-I`

表 4-3: xc32-as のその他のオプション カテゴリ

オプション	概要	コマンドライン

Diagnostics level	[Output] ウィンドウに表示する警告を選択します。
	- 警告を表示する
	- 警告を表示しない
	- 致命的な警告を表示する
		
		`--warn`
		`--no-warn`
		`--fatal-warnings`

Include source code	高水準言語リスティングの場合、チェックを付けます。高水準リスティングは、コンパイラによるアセンブリ ソースコードの生成を要求します (`-g` 等のデバッグ オプションをコンパイラに対して指定し、アセンブリ リスティング (`-al`) を要求します)。通常のリストスティングの場合、チェックを外します。	`-ah`
Expand macros	チェックを付けると、リスティング内でマクロを展開します。チェックを外すと、マクロを折り畳みます。	`-am`
Include false conditionals	チェックを付けると、リスティングに条件式 (`.if`、`.ifdef`) を含めます。チェックを外すと、条件式を省略します。	`-ac`
Omit forms processing	チェックと付けると、リスティング ディレクティブ `.psize`、`.eject`、`.title`、`.sbttl` によって実行される全ての形態の処理を OFF にします。チェックを外すと、リスティング ディレクティブによって処理されます。	`-an`
Include assembly	チェックを付けると、アセンブリ リスティングを含めます。この `-a` サブオプションは、他のサブオプションと一緒に使えます。チェックを外すと、アセンブリ リスティングを除外します。	`-al`
List symbols	チェックを付けると、リスティングにシンボルテーブルを含めます。チェックを外すと、リスティングからシンボルテーブルを除外します。	`-as`
Omit debugging directives	チェックを付けると、リスティングからデバッグ ディレクティブを除外します。これにより、リスティングをクリーンにできます。チェックを外すと、デバッグ ディレクティブをリスティングに含めます。	`-ad`
List to file	ファイルにリスティングする場合、このオプションを使います。リストファイルには、asm ファイルの名前に「`.lst`」を付け足した名前が付けられます。	`-a=${CURRENT_QUOTED_IF_SPACED_OBJECT_FILE_MINUS_EXTENSION}.lst`

4.5.3 xc32-gcc (32 ビット C コンパイラ)

MPLAB XC32 C コンパイラは MPLAB X IDE と連携しますが、IDE とは別に入手する必要があります。製品バージョンを購入するか、無償バージョン (機能制限付き) をダウンロードできます。詳細は Microchip 社ウェブサイト (www.microchip.com) を参照してください。MPLAB X IDE 内でコマンドライン オプションの一部を指定できます。カテゴリを選択した後にコンパイラ オプションを設定します。その他のオプションについては『MPLAB® X IDE ユーザガイド』(DS52027/DS50002027) を参照してください。この文書は Microchip 社ウェブサイトから入手できます。

© 2015 Microchip Technology Inc.

「ページ機能オプション」も参照してください。

表 4-4: xc32-gcc の一般カテゴリ

オプション	概要	コマンドライン
Have symbols in production build	プロダクション ビルドイメージでのデバッグ向けにビルドします。	`-g`
Enable App IO	REAL ICE による APPIN/APPOUT デバッグ機能をサポートします。	`-mappio-debug`
Isolate each function in a section	このオプションは、参照されない関数を削除するために、しばしばリンカの `--gc-sections` オプションと一緒に使います。チェックを付けると、各関数を出力ファイル内の別々のセクションに配置します。出力ファイル内のセクションの名前は、関数の名前によって決まります。Note: このオプションを指定すると、アセンブラとリンカが生成するオブジェクトと実行ファイルのサイズが増加し、実行速度が低下する可能性があります。 チェックを外すと、複数の関数を 1 つのセクションに配置します。	`-ffunction-sections`
Place data into its own section	このオプションは、参照されない静的に割り当てられた変数を削除するために、しばしばリンカの `--gc-sections` オプションと一緒に使います。 チェックを付けると、各データアイテムを出力ファイル内の別々のセクションに配置します。セクションの名前は、データアイテムの名前によって決まります。このオプションを指定すると、アセンブラとリンカが生成するオブジェクトと実行ファイルのサイズが増加し、実行速度が低下する可能性があります。	`-fdata-sections`
Use indirect calls	フルレンジの呼び出しを有効にします。	`-mlong-calls`
Generate 16-bit code	既定値により、MIPS16 命令セット向けにコードを生成します (コードサイズを削減)。	`-mips16`
Exclude floating-point library	浮動小数点演算のサポートを必要としないアプリケーションでは、このライブラリを除外する事でコードサイズを削減できます。	`-mno-float`

表 4-5: xc32-gcc の最適化カテゴリ

オプション	概要	コマンドライン
Optimization Level	最適化レベルを選択します。コンパイラのエディションによっては、一部の最適化レベルをサポートしない場合があります。これは `-On` オプションと等価です (*n* は以下の通り)。 • 0 - 最適化しない : コンパイルに要する時間とリソースを削減し、デバッグで予測通りの結果が得られる事を目標とします。 • 1 - 最適化 : 最適化によってコンパイルに要する時間が増え、より大きな関数にはより多くのホストメモリを消費します。コンパイラはコードサイズと実行時間の削減を試みます。 • 2 - より高レベルの最適化 : コンパイラは、サポートするほとんど全ての最適化(サイズと速度のトレードオフは含まず) を実行します。 • 3 - 速度を優先したさらなる最適化 (O2 の上位バージョン) • 4 - サイズを優先したさらなる最適化 (O2 の上位バージョン)	`-On`

表 4-5: xc32-gcc の最適化カテゴリ (続き)

オプション	概要	コマンドライン
Unroll loops	このオプションを指定すると、しばしば実行速度は向上しますが、コードサイズは増加します。 チェックを付けると、ループ展開の最適化を実行します。これは、コンパイル時または実行時に繰り返し数を特定可能なループに対してのみ実行されます。 チェックを外すと、ループを展開しません。	`-funroll-loops`
Omit frame pointer	チェックを付けると、フレームポインタを必要としない関数に対してレジスタ内でフレームポインタを保持しません。 チェックを外すと、フレームポインタを保持します。	`-fomit-frame-pointer`

Pre-optimization instruction scheduling	最適化レベルの既定値 :
	- 無効
	- 有効
		
		`-fno-schedule-insns`
		`-fschedule-insns`

Post-optimization instruction scheduling	最適化レベルの既定値 :
	- 無効
	- 有効
		
		`-fno-schedule-insns2`
		`-fschedule-insns2`

表 4-6: xc32-gcc の処理およびメッセージ カテゴリ

オプション	概要	コマンドライン
Preprocessor macros	コンパイラの -D オプションにより、プロジェクトに固有のプリプロセッサ マクロ定義を渡します。	
Include directories	これらのディレクトリ内でプロジェクトに固有のインクルード ファイルを検索します。	
Make warnings into errors	チェックを付けると、エラーと警告に基づいてコンパイルを停止します。 チェックを外すと、エラーだけに基づいてコンパイルを停止します。	`-Werror`
Additional warnings	チェックを付けると、全ての警告を有効にします。 チェックを外すと、警告を無効にします。	`-Wall`
support-ansi	チェックを付けると、厳密な ANSI C が要求する警告を出力します。 チェックを外すと、全ての警告を出力します。	`-ansi`
strict-ansi	厳密な ISO C および ISO C++ が要求する警告を出力します (禁止されている拡張を使う全てのプログラムと、ISO C および ISO C++ に従わないその他の一部のプログラムを拒絶します)。	`-pedantic`
Use CCI syntax	CCI 構文 (**第 2 章「CCI (Common C Interface)」**参照) のサポートを有効にします。	`-mcci`
Use IAR syntax	他のツールチェーン ベンダーが使用する構文のサポートを有効にします。	`-mext=IAR`

© 2015 Microchip Technology Inc.

4.5.4 xc32-g++

MPLAB X IDE 内でコマンドライン オプションの一部を指定できます。カテゴリを選択してからリンカ オプションを設定します。その他のオプションについては、32 ビットデバイス向け MPLAB オブジェクトリンカのマニュアルを参照してください。「ページ機能オプション」も参照してください。

表 4-7: xc32-g++ C++ 専用のカテゴリ

オプション	概要	コマンドライン
Generate run time type descriptor information	C++ 実行時型識別機能 (「dynamic_cast」と「typeid」) が使う仮想関数を持つ各クラスに関する情報を生成します。言語のそのようなパーツを使わない場合、このオプションを無効にする事でメモリ空間を節約できます。例外処理も同じ情報を使いますが、例外処理は必要に応じて情報を生成するという事に注意してください。このオプションを無効にしても、「dynamic_cast」演算子は実行時型情報を必要としないキャスト(void * または一義的ベースクラスへのキャスト) 向けに使えます。	`-frtti`
Enable C++ exception handling	C++ 例外処理を有効にします。例外を伝播させるために追加のコードを生成します。	`-fexceptions`
Check that the pointer returned by operator 'new' is non-null	割り当てられているストレージの変更を試みる前に、演算子「new」によって返されたポインタが非 NULL である事を確認します。	`-fcheck-new`
Generate code to check for violation of exception specification	実行時に例外仕様の違反をチェックするためのコードを生成しません。このオプションは C++ 規格に違反しますが、プロダクション ビルドにおけるコードサイズの削減に役立ちます。	`-fenforce-eh-specs`

表 4-8: xc32-g++ の一般カテゴリ

オプション	概要	コマンドライン
Have symbols in production build	プロダクション ビルドイメージでのデバッグ向けにビルドします。	`-g`
Enable App IO	REAL ICE による APPIN/APPOUT デバッグ機能をサポートします。	`-mappio-debug`
Isolate each function in a section	ターゲットが任意のセクションをサポートする場合、各関数を出力ファイル内の別々のセクションに配置します。出力ファイル内のセクションの名前は関数の名前によって決まります。このオプションをリンカの `--gc-sections` オプションと組み合わせる事で、参照されない関数を削除できます。	`-ffunction-sections`
Place data into its own section	ターゲットが任意のセクションをサポートする場合、各データアイテムを出力ファイル内の別々のセクションに配置します。出力ファイル内のセクションの名前はデータアイテムの名前によって決まります。このオプションをリンカの `--gc-sections` オプションと組み合わせる事で、参照されない変数を削除できます。	`-fdata-sections`
Use indirect calls	フルレンジの呼び出しを有効にします。	`-mlong-calls`
Generate 16-bit code	既定値により、MIPS16 命令セット向けにコードを生成します (コードサイズを削減)。	`-mips16`
Exclude floating-point library	浮動小数点演算のサポートを必要としないアプリケーションでは、このライブラリを除外する事でコードサイズを削減できます。	`-mno-float`

表 4-9: xc32-g++ の最適化カテゴリ

オプション	概要	コマンドライン
Optimization Level	最適化レベルを選択します。コンパイラのエディションによっては、一部の最適化レベルをサポートしない場合があります。これは -O*n* オプションと等価です (*n* は以下の通り)。 • 0 - 最適化しない : コンパイルに要する時間とリソースを削減し、デバッグで予測通りの結果が得られる事を目標とします。 • 1 - 最適化 : 最適化によってコンパイルに要する時間が増え、より大きな関数にはより多くのホストメモリを消費します。コンパイラはコードサイズと実行時間の削減を試みます。 • 2 - より高レベルの最適化 : コンパイラは、サポートするほとんど全ての最適化(サイズと速度のトレードオフは含まず) を実行します。 • 3 - 速度を優先したさらなる最適化 (O2 の上位バージョン) • 4 - サイズを優先したさらなる最適化 (O2 の上位バージョン)	-O*n*
Unroll loops	チェックを付けると、ループ展開の最適化を実行します。これは、コンパイル時または実行時に繰り返し数を特定可能なループに対してのみ実行されます。 チェックを外すと、ループを展開しません。	-funroll-loops
Omit frame pointer	チェックを付けると、フレームポインタを必要としない関数に対してレジスタ内でフレームポインタを保持しません。 チェックを外すと、フレームポインタを保持します。	-fomit-frame-pointer
Pre-optimization instruction scheduling	最適化レベルの既定値 : - 無効 - 有効	 -fno-schedule-insns -fschedule-insns
Post-optimization instruction scheduling	最適化レベルの既定値 : - 無効 - 有効	 -fno-schedule-insns2 -fschedule-insns2

表 4-10: xc32-g++ の最適化カテゴリ

オプション	概要	コマンドライン
Preprocessor macros	コンパイラの -D オプションにより、プロジェクトに固有のプリプロセッサ マクロ定義を渡します。	
Include directories	これらのディレクトリ内でプロジェクトに固有のインクルード ファイルを検索します。	
Make warnings into errors	チェックを付けると、エラーと警告に基づいてコンパイルを停止します。 チェックを外すと、エラーだけに基づいてコンパイルを停止します。	-Werror
Additional warnings	チェックを付けると、全ての警告を有効にします。 チェックを外すと、警告を無効にします。	-Wall
support-ansi	チェックを付けると、厳密な ANSI C が要求する警告を出力します。 チェックを外すと、全ての警告を出力します。	-ansi
strict-ansi	厳密な ISO C および ISO C++ が要求する警告を出力します (禁止されている拡張を使う全てのプログラムと、ISO C および ISO C++ に従わないその他の一部のプログラムを拒絶します)。	-pedantic

© 2015 Microchip Technology Inc.

表 4-10: xc32-g++ の最適化カテゴリ (続き)

オプション	概要	コマンドライン
Use CCI syntax	CCI 構文 (**第 2 章「CCI (Common C Interface)」**参照) のサポートを有効にします。	`-mcci`
Use IAR syntax	他のツールチェーン ベンダーが使用する構文のサポートを有効にします。	`-mext=IAR`

4.5.5 xc32-ld (32 ビットリンカ)

MPLAB X IDE 内でコマンドライン オプションの一部を指定できます。カテゴリを選択してからリンカ オプションを設定します。その他のオプションについては、32 ビットデバイス向けMPLABオブジェクト リンカのマニュアルを参照してください。「ページ機能オプション」も参照してください。

表 4-11: xc32-ld の一般カテゴリ

オプション	概要	コマンドライン
Heap Size (bytes)	ヒープのサイズをバイト数で指定します。C プログラムが使う実行時ヒープに *size* バイトを割り当てます。ヒープは未使用のデータメモリから割り当てられます。メモリ容量が不足する場合、エラーが報告されます。	`--heap` *size*
Minimum stack size (bytes)	スタックの最小サイズをバイト数で指定します。既定値では、リンカは全ての未使用データメモリを実行時スタックに割り当てます。2 つのグローバル シンボル (`__SP_init` と `__SPLIM_init`) を宣言する事によってスタックを割り当てる事もできます。このオプションを使うと、指定した最小サイズ以上のスタックを確保できます。実際のスタックサイズは、リンクマップ出力ファイル内で示されます。最小サイズが得られない場合、エラーが報告されます。	`--stack` *size*
Allow overlapped sections	チェックを付けると、セクション アドレスのオーバーラップを確認しません。 チェックを外すと、オーバーラップを確認します。	`--check-sections` `--no-check-sections`
Remove unused sections	チェックを付けると、未使用入力セクションのガベージコレクションを有効にしません (一部のターゲット デバイスにのみ適用)。 チェックを外すと、ガベージ コレクションを有効にします。	`--no-gc-sections` `--gc-sections`
Use response file to link	リンカ オプションを、コマンドラインからではなくファイルで渡します。Windows システムでオブジェクト ファイルを多数指定すると Windows OS によるコマンドライン長の制限を超えてしまう場合、このオプションを使うとプロジェクトに多数のオブジェクト ファイルをリンクできます。	`True`
Additional driver options	この GUI 内で用意されていないドライバ オプションは、ここにタイプします。ここに入力した文字列は、ドライバ起動命令内でそのまま使われます。	

表 4-12: xc32-ld のライブラリ カテゴリ

オプション	概要	コマンドライン
Optimization level of Standard Libraries	最適化レベルを選択します。コンパイラのエディションによっては、一部の最適化レベルをサポートしない場合があります。これは -On オプションと等価です (n は以下の通り)。 • 0 - 最適化しない : コンパイルに要する時間とリソースを削減し、デバッグで予測通りの結果が得られる事を目標とします。 • 1 - 最適化 : 最適化によってコンパイルに要する時間が増え、より大きな関数にはより多くのホストメモリを消費します。コンパイラはコードサイズと実行時間の削減を試みます。 • 2 - より高レベルの最適化 : コンパイラは、サポートするほとんど全ての最適化(サイズと速度のトレードオフは含まず) を実行します。 • 3 - 速度を優先したさらなる最適化 (O2 の上位バージョン) • 4 - サイズを優先したさらなる最適化 (O2 の上位バージョン)	`-On`
System Libraries	プロジェクト ファイルにリンクするライブラリを追加します。複数のライブラリを追加できます。	`--library=name`
Library directories	ライブラリの検索パスにライブラリ ディレクトリを追加します。複数のディレクトリを追加できます。	`--library-path="name"`
Exclude Standard Libraries	チェックを付けると、リンク時に標準のシステム起動ファイルまたはライブラリを使わずに、コマンドラインで指定されたライブラリ ディレクトリだけを使います。 チェックを外すと、標準のシステム起動ファイルまたはライブラリを使います。	`-nostdlib`
Do no link startup code	既定値の起動コードを除外し、プロジェクトが提供するアプリケーション専用の起動コードを使います。	`-nostartfiles`
Generate 16-bit code	MIPS16 命令セット向けにプリコンパイル済みのライブラリをリンクする事で、コードサイズを削減します。	`-mips16`
Exclude floating-pointlibrary	浮動小数点演算のサポートを必要としないアプリケーションでは、このライブラリを除外する事でコードサイズを削減できます。	`-mno-float`

表 4-13: xc32-ld の診断カテゴリ

オプション	概要	コマンドライン
Generate map file	マップファイルを生成します。	`-Map="file"`
Display memory usage	チェックを付けると、メモリ使用量のレポートを出力します。 チェックを外すと、レポートを出力しません。	`--report-mem`
Generate cross-reference file	チェックを付けると、相互参照テーブルを生成します。 チェックを外すと、このテーブルを生成しません。	`--cref`
Warn on section realignment	チェックを付けると、アライメントのためにセクション変更を開始する際に警告します。 チェックを外すと、警告しません。	`--warn-section-align`
Trace Symbols	トレースシンボルを追加 / 削除します。	`--trace-symbol=symbol`

© 2015 Microchip Technology Inc.

表 4-14: xc32-ld のシンボルおよびマクロ カテゴリ

オプション	概要	コマンドライン
Linker symbols	出力ファイル内に、絶対アドレス (*expr*) を格納するグローバル シンボルを生成します。複数のシンボルをコマンドラインで定義する場合、このオプションは必要に応じて何度でも使えます。これに関連して、一部の形態の算術演算が *expr* に対してサポートされます (16 進定数または既存シンボルの名前を与えるか、「+」または「-」を使って 16 進定数またはシンボルの加減算が可能)。	`--defsym=sym`
Preprocessor macro definitions	リンカマクロを追加します。	`-Dmacro`

Symbols	出力内のシンボル情報を指定します。
	- 全てを保持する
	- デバッグ情報を取り除く
	- 全てのシンボル情報を取り除く
		
		—
		`--strip-debug (-S)`
		`--strip-all (-s)`

4.5.6 ページ機能オプション

[Properties] ページの [Options] セクションは、全てのツール向けに以下の機能を提供します。

表 4-15: ページ機能オプション

Reset	ページを既定値にリセットします。
Additional options	コマンドライン(非GUI)フォーマットでオプションを入力します。
Option Description	オプション名をクリックすると、そのオプションに関する情報をこのウィンドウに表示します(一部のオプションの情報はこのウィンドウに表示されません)。
Generated Command Line	オプション名をクリックすると、そのオプションに等価なコマンドライン オプションをこのウィンドウに表示します。

4.6 プロジェクト例

このプロジェクト例では、2 つの C コードファイルを持つ MPLAB X IDE プロジェクトを作成します。

- プロジェクト ウィザードの実行
- ビルド オプションの設定
- プロジェクトのビルド
- 出力ファイル
- コードの開発

4.6.1 プロジェクト ウィザードの実行

MPLAB X IDE 内で *File>New Project* を選択してウィザードを起動します。

1. **プロジェクトの選択 :** カテゴリに [Microchip Embedded] を選択し、プロジェクトに [Standalone Project] を選択します。**[Next>]** をクリックして次へ進みます。
2. **デバイスの選択 :** dsPIC30F6014 を選択します。**[Next>]** をクリックして次へ進みます。
3. **ヘッダの選択 :** このデバイスにはヘッダが存在しないため、これはスキップします。
4. **ツールの選択:** リストから開発ツールを選択します。選択したデバイスをサポートしているツールは、ツール名の横の色付きの丸マークで示されます。マークの上にマウスカーソルを置くと、サポート情報がテキストで示されます。**[Next>]** をクリックして次へ進みます。
5. **コンパイラの選択 :** XC32 ツールチェーンのバージョンを選択します。**[Next>]** をクリックして次へ進みます。
6. **プロジェクト名とフォルダの選択 :** プロジェクト名 (例 : `MyXC32Project`) を入力します。次に、プロジェクト フォルダのパスを選択します。**[Finish]** をクリックして、プロジェクトの作成とセットアップを終了します。

プロジェクト ウィザードを完了すると、[Project] ウィンドウにプロジェクト ツリーが表示されます。プロジェクトに関する詳細は、MPLAB X IDE のマニュアルを参照してください。

4.6.2 ビルド オプションの設定

File>Project Properties を選択するか、プロジェクト名を右クリックして [Properties] を選択すると、[Project Properties] ダイアログを開きます。

1. [Conf:[default]>XC32 (Global Options)] の下で [xc32-gcc] を選択します。
2. [Conf:[default]>XC32 (Global Options)] の下で [xc32-ld] を選択します。
3. [Option Categories] から [Diagnostics] を選択し、ファイル名 (例 : `example.map`) を [Generate map file] に入力します。
4. ダイアログの下端にある **[OK]** をクリックして、ビルド オプションを承認します (ダイアログは閉じます)。

4.6.3 プロジェクトのビルド

プロジェクト ツリー内でプロジェクト名「MyXC32Project」を右クリックし、ポップアップ メニューから [Build] を選択します。[Output] ウィンドウにビルド結果が表示されます。

ビルドが正常に完了しなかった場合、以下を確認します。

1. これまでの手順を再確認します。言語ツールを正しく設定したか、プロジェクト ファイルとビルド オプションは全て正しいか確認してください。
2. サンプル ソースコードを変更する場合、[Output] ウィンドウの [Build] タブでソースコード内の構文エラーを調べます。見つかったエラーをクリックすると、そのエラーを含むソースコード行へジャンプします。エラーを修正した後に再ビルドしてください。

© 2015 Microchip Technology Inc.

4.6.4 出力ファイル

MPLAB X IDE 内でプロジェクト出力ファイルを開くと、内容を表示する事ができます。

1. *File>Open File* を選択します。[Open] ダイアログ内でプロジェクト ディレクトリを見つけます。
2. [Files of type] の下で [All Files] を選択すると、全てのプロジェクト ファイルが表示されます。
3. *File>Open File* を選択します。[Open] ダイアログ内で [example.map] を選択します。**[Open]** をクリックすると、MPLAB X IDE のエディタ ウィンドウにリンカマップ ファイルが表示されます。このファイルの詳細は、リンカのマニュアルを参照してください。
4. *File>Open File* を選択します。[Open] ダイアログ内で、プロジェクト ディレクトリに戻ってから、*dist>default>production* ディレクトリへ移動します。HEX ファイルは 1 つ (MyXC32Project.X.production.hex) しかありません。これがプライマリ出力ファイルです。**[Open]** をクリックすると、MPLAB X IDE のエディタ ウィンドウにこの HEX ファイルが表示されます。このファイルの詳細は、ユーティリティのマニュアルを参照してください。
   他に「MyXC32Project.X.production.elf」というファイルもあります。このファイルはデバッグ情報を含み、デバッグツールはこのファイルを使ってコードをデバッグします。デバッグファイルのタイプの選択については、**4.5.1 XC32 (グローバル オプション)** を参照してください。

4.6.5 コードの開発

多くの場合、アプリケーション コードはエラーを含んでおり、最初は正しく動作しません。従って、コードの開発を助けるデバッグツールが必要です。デバッグには前述の出力ファイルと、MPLAB X IDE と連携する各種のデバッグツールを使います。これには Microchip 社製またはサードパーティ製のシミュレータ、インサーキット エミュレータ、インサーキット デバッガが選べます。これらのツールの使い方については、各ツールのマニュアルを参照してください。デバッグする際は、*Debug>Debug Project* を使ってコードを実行 / デバッグします。詳細は、MPLAB X IDE ユーザガイドを参照してください。

開発が完了したコードはデバイスにプログラミングします。これにも MPLAB X IDE と連携する各種のプログラマが使えます。これらのツールの使い方については、各ツールのマニュアルを参照してください。プログラミングする際は、デバッグ ツールバー上の [Make and Program Device Project] ボタンを使います。このボタンについては、MPLAB X IDE ユーザガイドを参照してください。

NOTE:

 © 2015 Microchip Technology Inc.

第 5 章　コンパイラのコマンドライン ドライバ

5.1　はじめに

コマンドライン ドライバ (xc32-gcc または xc32-g++) は、C/C++ コードの生成 / アセンブリ / リンク等を含む全てのコンパイル手順を実行するために呼び出されるアプリケーションです。IDE を使ってコンパイルを支援する場合でも、IDE は最終的に xc32-gcc(C プロジェクト向け) または xc32-g++(C++ プロジェクト向け) を呼び出します。

コンパイラの内部アプリケーションはコマンドラインから明示的に呼び出せますが、複雑な内部アプリケーションの使用を隠蔽し、全てのコンパイル手順に一貫したインターフェイスを提供する xc32-gcc または xc32-g++ ドライバの使用を推奨します。

本章では、コンパイル中にドライバが実行する手順、ドライバによって処理および生成可能なファイル、コンパイラの動作を制御するためのコマンドライン オプションについて説明します。また、これらのコマンドライン オプションと MPLAB IDE の [Build Options] ダイアログ内の設定との対応についても説明します。

本章には、コマンドラインからのドライバの使用方法に関する以下の項目を記載しています。

- コンパイラの起動
- C コンパイルのシーケンス
- C++ コンパイルのシーケンス
- ランタイム ファイル
- 起動と初期化
- コンパイラ出力
- コンパイラのメッセージ
- ドライバ オプションの説明

5.2　コンパイラの起動

次の項目では、コマンドラインからコンパイラを起動して実行する方法について説明します。また、コンパイラが使う環境変数と入力ファイルについても、その後の項目で説明します。

5.2.1　コンパイル ドライバのコマンドライン フォーマット

コンパイル ドライバ プログラム (xc32-gcc) は、C およびアセンブリ言語モジュールとライブラリ アーカイブをコンパイル / アセンブル / リンクします。モジュールソースが C++ で書かれている場合、xc32-g++ ドライバを使います。ほとんどのコンパイラ用コマンドライン オプションは GCC ツールセットの全ての処理系と共通ですが、本コンパイラに固有のオプションもあります (XC32 コンパイラは GCC ツールセットを使いますが、XC8 は使いません)。

コンパイラ用コマンドラインの基本形態は以下の通りです。

```
xc32-gcc [options] files
xc32-g++ [options] files
```

例えば、C ソースファイル hello.c をコンパイル / アセンブル / リンクして完全な実行ファイル hello.elf を生成する場合、以下の命令を実行します。

```
xc32-gcc -o hello.elf hello.c
```

© 2015 Microchip Technology Inc.

また、C++ ソースファイル hello.cpp をコンパイル / アセンブル / リンクして完全な実行ファイル hello.elf を生成する場合は以下の命令を実行します。

```
xc32-g++ -o hello.elf hello.cpp
```

利用可能なオプションについては **5.9「ドライバ オプションの説明」**で説明します。通常はファイル名の前で options (先頭に「-」を付けて識別) を指定しますが、必須ではありません。

files には複数のファイルを指定でき、C/C++ およびアセンブラのソースファイルと再配置可能なオブジェクト ファイル (.o) またはアーカイブ ファイルを混ぜて一度に指定する事ができます。しかし、それらのファイルを指定する順番が重要です。この指定順は、コードまたはデータがメモリ内に配置される順番またはシンボルの検索順に影響します。一般的に、アーカイブ ファイルはソースファイルの後で指定します。ファイルのタイプについては **5.2.2「入力ファイルのタイプ」**を参照してください。

Note:	コマンドライン オプションとファイルの拡張子は、大文字と小文字を区別します。

ライブラリはユーザ定義オブジェクト コードのリストです。リンカは、標準 C ライブラリに加えて、これらのライブラリも検索します。これらのファイルの検索順は指定した順番によって決まります。通常、これらのファイルはソースファイルの後で指定しますが、そうする事が必須ではありません。

本書では、コンパイラ アプリケーションがコンソールの検索パス内に保存されているか、適切な環境変数が指定されている事を前提とします。そうではない場合、全てのアプリケーションの実行時にフルパスを指定する必要があります。

環境変数

以下で説明する環境変数は定義してもしなくても構いません。しかし定義した場合は、その定義がコンパイラによって使われます。コンパイラ ドライバまたは他のサブプログラムは、以下の環境変数の一部に対して、それらが定義されていなければ適切な値を自動的に決定します。ドライバまたは他のサブプログラムは、コンパイラのインストール状態に関する内部情報を利用します。インストールしたディレクトリ構造が失われていない場合 (すなわち、全てのサブディレクトリと実行ファイルが元のままの相対位置を保っている場合)、ドライバまたはサブプログラムは適切な環境変数値を決定できます。「XC32」変数は新規プロジェクト向けですが、「PIC32」変数はレガシー プロジェクト向けに使えます。

表 5-1: コンパイラ関連の環境変数

オプション	定義
XC32_C_INCLUDE_PATH PIC32_C_INCLUDE_PATH	この変数の値は、セミコロンで区切ったディレクトリのリストです (PATH と同様)。コンパイラはヘッダファイルを検索する際に、最初に -I で指定されたディレクトリを検索した後、この変数で指定されているディレクトリを検索し、その後で標準のヘッダファイル ディレクトリを検索します。この環境変数が未定義である場合、プリプロセッサは標準のインストール状態に基づいて適切な値を選択します。既定値では、以下のディレクトリでインクルード ファイルを検索します。 <install-path>\pic32mx\include
XC32_COMPILER_PATH PIC32_COMPILER_PATH	PIC32_COMPILER_PATH の値は、セミコロンで区切ったディレクトリのリストです (PATH と同様)。コンパイラはサブプログラムを検索する際に、PIC32_EXEC_PREFIX を使って見付からなければ、このオプションで指定されたディレクトリを検索します。

© 2015 Microchip Technology Inc.

表 5-1: コンパイラ関連の環境変数 (続き)

オプション	定義
XC32_EXEC_PREFIX PIC32_EXEC_PREFIX	PIC32_EXEC_PREFIX は、コンパイラが実行するサブプログラムの名前に使う接頭辞を指定します。この接頭辞は、ディレクトリ区切り文字を挟まずにサブプログラム名に追加されますが、必要であればスラッシュで終わる接頭辞を指定する事もできます。指定された接頭辞を使ってサブプログラムを見付ける事ができなかった場合､コンパイラはユーザの PATH 環境変数を調べます。PIC32_EXEC_PREFIX 環境変数が設定されていないか空白値に設定されている場合、コンパイラ ドライバは標準インストール状態に基づいて適切な値を選択します。インストール状態が変更されていなければ、ドライバは必要なサブプログラムを見付ける事ができます。-B コマンドライン オプションで他の接頭辞を指定した場合、ユーザまたはドライバが定義した PIC32_EXEC_PREFIX の値よりもそちらが優先されます。通常の状況では、この値を定義せずに、ドライバ自身にサブプログラムを見付けさせるのが最善の方法です。
XC32_LIBRARY_PATH PIC32_LIBRARY_PATH	この変数の値は、セミコロンで区切ったディレクトリのリストです (PATH と同様)。この変数は、リンカに渡すディレクトリのリストを指定します。この変数のドライバの既定値は以下の通りです。 <install-path>\lib; <install-path>\pic32mx\lib
TMPDIR	TMPDIR は、一時ファイル向けに使うディレクトリを指定します。コンパイラは、一時ファイルを使ってコンパイルの 1 つの処理段の出力を保持し、その値を次の処理段への入力値として使います。例えば、プリプロセッサの出力は本来のコンパイラへの入力として使われます。

5.2.2 入力ファイルのタイプ

コンパイル ドライバは以下のファイル拡張子を認識します (大文字と小文字を区別)。

表 5-2: ファイル名

拡張子	定義
file.c	前処理が必要な C ソースファイル
file.cpp	前処理が必要な C++ ソースファイル
file.h	ヘッダファイル (コンパイルもリンクもしない)
file.i	前処理済みの C ソースファイル
file.o	オブジェクト ファイル
file.ii	前処理済みの C++ ソースファイル
file.s	アセンブリ言語ソースファイル
file.S	前処理が必要なアセンブリ言語ソースファイル
その他	リンカに渡すファイル

コンパイラには、ソースファイル名に関する制約はありません。しかしオペレーティング システムによる制約 (大文字 / 小文字の区別、名前の長さ、その他) には注意が必要です。IDE を使う場合、アセンブリ ソースファイルのベース名は、そのファイルを使うプロジェクトのベース名と同じにしない事が必要です。同じベース名を使うと、ビルド処理中にソースファイルが一時ファイルによって上書きされます。

コンピュータ プログラムに関する説明では、「ソースファイル」と「モジュール」という用語がよく使われます。しばしばこれらは同じ意味で使われますが、本書の場合、これらの用語はコンパイル シーケンスにおける異なる段階でのソースファイルを指します。

「ソースファイル」とは、プログラムの全部または一部が書かれたファイルです。これらのファイルは C/C++ コード、プロセッサ ディレクティブ、命令を含みます。ソースファイルは、ドライバによって内部でプリプロセッサに渡されます。

「モジュール」とは、入力されたソースファイルに対するプリプロセッサの出力であり、`#include` プリプロセッサ ディレクティブによって指定された全てのヘッダファイル (または他のソースファイル) がインクルードされています。これらのファイルからは、全てのプリプロセッサ ディレクティブおよび命令 (デバッグ用の一部の命令は除く) が削除されています。これらのモジュールは、後続のコンパイラ アプリケーションに渡されます。従って、モジュールは複数のソースファイルとヘッダファイルから生成されます。モジュールは、しばしば変換ユニット (translation unit) とも呼ばれます。これらの用語はアセンブリ ファイルに対しても適用します。なぜならば、アセンブリ ファイルにも他のヘッダファイルとソースファイルをインクルードするからです。

© 2015 Microchip Technology Inc.

5.3 C コンパイルのシーケンス

5.3.1 シングルステップの C コンパイル

1つのコマンドライン命令を使って1つまたは複数のファイルをコンパイルできます。

5.3.1.1 1 つの C ファイルをコンパイルする方法

以下では、1 つのファイルをコンパイル / リンクする方法を示します。ここでは、コンパイラの <install-dir>/bin ディレクトリがユーザの PATH 変数に追加済みであると想定します。その他のディレクトリは以下の通りです。

- <install-dir>/pic32mx/include: 標準 C ヘッダファイル用ディレクトリ
- <install-dir>/pic32mx/include/proc: PIC32 デバイス専用ヘッダファイル用ディレクトリ
- <install-dir>/pic32mx/lib: 標準ライブラリおよび起動ファイル用ディレクトリ構造
- <install-dir>/pic32mx/include/peripheral: PIC32 周辺モジュール ライブラリ インクルード ファイル用ディレクトリ
- <install-dir>/pic32mx/lib/proc: デバイス固有のリンカスクリプト フラグメント、レジスタ定義ファイル、コンフィグレーション データが検索されるディレクトリ

以下は、2 つの値を加算する単純な C プログラムです。適当なテキストエディタを使ってこのプログラムを書き、ex1.c として保存します。

```
#include <xc.h>
#include <plib.h>

// Device-Specific Configuration-Bit settings
// SYSCLK = 80 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
//
#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1,
FWDTEN = OFF
#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_8

unsigned int x, y, z;

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

int
main(void)
{
  /* Configure the target for maximum performance at 80 MHz.*/
  SYSTEMConfigPerformance(80000000UL);
  x = 2;
  y = 5;
  z = add(x,y);
  return 0;
}
```

このプログラムの 1 行目で、ヘッダファイル xc.h をインクルードします。このファイルは、ターゲット デバイス上の全ての特殊機能レジスタ (SFR) に関する定義を提供します。

以下をコマンドラインに入力する事で、ファイルをコンパイルします。

```
xc32-gcc –mprocessor=32MX795F512L -o ex1.out ex1.c
```

コマンドライン オプション -o ex1.out は、出力する実行ファイルの名前を指定します (-o オプションを指定しない場合、出力ファイル名は a.out です)。実行ファイルは MPLAB IDE に読み込む事ができます。

HEX ファイルをデバイス プログラマに読み込む必要がある場合、以下の命令を使います。

```
xc32-bin2hex ex1.out
```

これは ex1.hex という名前のインテル HEX 形式ファイルを生成します。

5.3.1.2 複数の C ファイルをコンパイルする

以下では、複数のファイルを 1 手順でコンパイル / リンクする方法を示します。アプリケーション内で複数のファイルを使う方法を説明するために、以下のサンプルコードでは Add() 関数を add.c という名前の別のファイルに移動しています。

File 1

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

// Device-Specific Configuration-Bit settings
// SYSCLK = 80 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
//
#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1,
FWDTEN = OFF
#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_8

int main(void);
unsigned int add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
  /* Configure the target for maximum performance at 80 MHz.*/
  SYSTEMConfigPerformance(80000000UL);
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
```

File 2

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


© 2015 Microchip Technology Inc.

以下をコマンドラインに入力する事で、両方のファイルをコンパイルします。

```
xc32-gcc -mprocessor=32MX795F512L -o ex1.out ex1.c add.c
```

この命令は 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 ライブラリとリンクされ、実行ファイル ex1.out が生成されます。

5.3.2 マルチステップの C コンパイル

make ユーティリティと IDE (MPLAB IDE 等) を使うと、複数のソースファイルを含むプロジェクトのインクリメンタル ビルドが可能です。プロジェクトをビルドする際に、それらのツールは直近のビルドからどのソースファイルが変更されたか認識し、必要なファイルだけをコンパイルする事で、コンパイル時間を短縮します。

例えば、2 つのソースファイルをコンパイルする場合、直近のビルド後に片方のファイルだけが変更されたのであれば、他方の未変更ソースファイルから中間生成ファイルを再生成する必要はありません。

makeユーティリティを使ってコンパイラを起動する場合、中間生成ファイル(.oファイル) を使うように make ファイルを設定し、中間生成ファイルを生成するために -c オプション (**5.9.2「出力の種類を制御するためのオプション」**参照) を指定する必要があります。通常 make ユーティリティはコンパイラを複数回呼び出します (中間生成ファイルを生成するために各ソースファイルに対して 1 回と、次段のコンパイル処理のために 1 回)。

以下は、make ユーティリティを使ってファイル ex1.c と add.c をコンパイルする場合のコマンドラインの例です。

```
xc32-gcc -mprocessor=32MX795F512L -c ex1.c
xc32-gcc -mprocessor=32MX795F512L -c add.c
xc32-gcc -mprocessor=32MX795F512L -o ex1.out ex1.o add.o
```

5.4 C++ コンパイルのシーケンス

5.4.1 シングルステップの C++ コンパイル

1つのコマンドライン命令を使って1つまたは複数のファイルをコンパイルできます。

5.4.1.1 1 つの C++ ファイルをコンパイルする

以下では、1 つのファイルをコンパイル / リンクする方法を示します。ここでは、コンパイラの <install-dir>/bin ディレクトリがユーザの PATH 変数に追加済みであると想定します。その他のディレクトリは以下の通りです。

- <install-dir>/pic32mx/include: 標準 C++ ヘッダファイル用ディレクトリ
- <install-dir>/pic32mx/include/proc: PIC32 デバイス専用ヘッダファイル用ディレクトリ
- <install-dir>/pic32mx/lib: 標準ライブラリおよび起動ファイル用ディレクトリ構造
- <install-dir>/pic32mx/include/peripheral: PIC32 周辺モジュール ライブラリ インクルード ファイル用ディレクトリ
- <install-dir>/pic32mx/lib/proc: デバイス固有のリンカスクリプト フラグメント、レジスタ定義ファイル、コンフィグレーション データが検索されるディレクトリ

以下は、単純な C++ プログラムの例です。適当なテキストエディタを使ってこのプログラムを書き、ex2.cpp として保存します。

```
/* ex2.cpp */
#include <xc.h>      // __XC_UART
#include <plib.h>    // SYSTEMConfigPerformance()

#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <iterator>
#include <functional>
#include <numeric>
using namespace std;

// Device-Specific Configuration-bit settings
#pragma config FPLLMUL=MUL_20, FPLLIDIV=DIV_2, FPLLODIV=DIV_1,
FWDTEN=OFF
#pragma config POSCMOD=HS, FNOSC=PRIPLL, FPBDIV=DIV_8

template <class T>
inline void print_elements (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}

template <class T>
inline void insert_elements (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i)
    {
        coll.insert(coll.end(),i);
    }
}

int main(void) {

    // Configure the target for max performance at 80 MHz.
    SYSTEMConfigPerformance (80000000UL);

    // Direct stdout to UART 1 for use with the simulator
    __XC_UART = 1;

    deque<int> coll;
    insert_elements(coll,1,9);
    insert_elements(coll,1,9);

    print_elements(coll, "on entry:");
```


© 2015 Microchip Technology Inc.

```
    //sort elements
    sort (coll.begin(), coll.end());

    print_elements(coll, "sorted:");

    //sorted reverse
    sort (coll.begin(), coll.end(), greater<int>());

    print_elements(coll, "sorted >:");

    while(1);
}
```

このプログラムの 1 行目で、ヘッダファイル xc.h をインクルードします。このファイルは、ターゲット デバイス上の全ての特殊機能レジスタ (SFR) に関する定義を提供します。2 行目ではヘッダファイル plib.h をインクルードします。このファイルは、周辺モジュール ライブラリに必要なプロトタイプを提供します。

以下をコマンド プロンプトに入力する事で、ファイルをコンパイルします。

```
xc32-g++ -mprocessor=32MX795F512L -Wl,--defsym=_min_heap_size=0xF000
-o ex2.elf ex2.cpp
```

オプション -o ex2.elf は出力する実行ファイルの名前を指定します。この elf ファイルは MPLAB IX DE に読み込む事ができます。

HEX ファイルをばデバイス プログラマに読み込む必要がある場合、以下の命令を使います。

```
xc32-bin2hex ex2.elf
```

これは ex2.hex という名前のインテル HEX 形式ファイルを生成します。

5.4.2 複数の C++ ファイルをコンパイルする

以下では、複数の C および C++ ファイルを 1 手順でコンパイル / リンクする方法を示します。

File 1

```
/* main.cpp */
#include <xc.h>     // __XC_UART
#include <plib.h>   // SYSTEMConfigPerformance()

#include <iostream>
using namespace std;

// Device-Specific Configuration-bit settings
#pragma config FPLLMUL=MUL_20, FPLLIDIV=DIV_2, FPLLODIV=DIV_1,
FWDTEN=OFF
#pragma config POSCMOD=HS, FNOSC=PRIPLL, FPBDIV=DIV_8

// add() must have C linkage
extern "C" {
extern unsigned int add(unsigned int a, unsigned int b);
}

int main(void) {
    int myvalue = 6;

    // Configure the target for max performance at 80 MHz.
    SYSTEMConfigPerformance (80000000UL);

    // Direct stdout to UART 1 for use with the simulator
    __XC_UART = 1;

    std::cout << "original value:" << myvalue << endl;
    myvalue = add (myvalue, 3);
    std::cout << "new value:      " << myvalue << endl;

    while(1);
}
```

File 2

```
/* ex3.c */
unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

以下をコマンドラインに入力する事で、両方のファイルをコンパイルします。

```
xc32-g++ -mprocessor=32MX795F512L -o ex3.elf main.cpp ex3.c
```

この命令はモジュール `main.cpp` と `ex3.c` をコンパイルします。コンパイルされたモジュールはC++向けのコンパイラ ライブラリとリンクされ、実行ファイル`ex3.elf`が生成されます。

Note:	プロジェクト内の C++ ソースファイルに対して必要な C++ 対応ライブラリをリンクするために、xc32-gcc の代わりに xc32-g++ ドライバを使います。

 © 2015 Microchip Technology Inc.

5.5 ランタイム ファイル

コマンドラインで指定した C/C++ およびアセンブリ ソースファイルに加えて、コンパイラが生成したソースファイルとプリコンパイル済みライブラリファイルも、ドライバによってプロジェクト内へコンパイルさる場合があります。これらのファイルには以下が含まれます。

- C/C++ 標準ライブラリ ルーチン
- 暗黙的に呼び出される算術ルーチン
- ユーザ定義ライブラリ ルーチン
- スタートアップ コード

> **Note:** 一部の PIC32 では、デバイス コンフィグレーション ビット (BOOTISA) を使って、MIPS32® モードまたは microMIPS™ ISA モードでブートするよう選択できます。これらのデバイスの BOOTISA ビットを microMIPS モード向けに設定した場合、xc32-gcc/g++ コンパイル ドライバに対して `-mmicromips` オプションを指定する事により、スタートアップ コードの microMIPS バージョンをリンクさせます。BOOTISA ビットを MIPS32 モード向けに設定した場合、コンパイル ドライバに対して `-mno-micromips` オプションを指定する事により、スタートアップ コードのMIPS32バージョンをリンクさせます。

5.5.1 ライブラリ ファイル

ドライバは、選択されたターゲット デバイスに適した C/C++ 標準ライブラリ ファイルの名前と、その他のドライバ オプションを決定します。

オプションの組み合わせが異なる複数のターゲット ライブラリ (multilibs と呼ぶ) がビルドされます。アプリケーションをコンパイル / リンクするためにコンパイラ ドライバを呼び出すと、ドライバは同じオプションを使ってビルドされたターゲット ライブラリのバージョンを選択します。

既定値では、32 ビット言語ツールはディレクトリ `<install-directory>/lib/gcc/` にそれらのライブラリを保存し、ディレクトリ `<install-directory>/<pic32mx>/lib` にターゲット専用のライブラリを保存します。これらの両方のディレクトリ構造は、オプションの組み合わせに応じて各 multilib を格納するためのサブディレクトリを含みます。これらのサブディレクトリは以下の通りです。

```
./.
./size
./speed
./mips16
./micromips
./no-float
./mips16/no-float
./micromips/no-float
./size/mips16
./size/micromips
./size/no-float
./size/mips16/no-float
./size/micromips/no-float
./speed/mips16
./speed/micromips
./speed/no-float
./speed/mips16/no-float
./speed/micromips/no-float
```

コンパイラと一緒に配布されるターゲット ライブラリは、以下のコマンドライン オプションに対応してビルドされます。

- サイズ / 速度 (-Os/-O3)
- MIPS16/MIPS32/microMIPS ISA モード (-mips16/-mno-mips16/-mmicromips)
- ソフトウェア浮動小数点 / 浮動小数点非サポート (-msoft-float/-mno-float)

これらのオプションの指定に応じて、以下の例のように multilibs サブディレクトリが選択されます。

1. `xc32-gcc foo.c`
   `xc32-g++ foo.cpp`

   この例では、コマンドライン オプションを何も指定していません (つまり、既定値のコマンドライン オプションが使われます)。この場合、「.」サブディレクトリが使われます。

2. `xc32-gcc -Os foo.c`
   `xc32-g++ -Os foo.cpp`

   この例では、コマンドライン オプション -Os により、サイズ優先の最適化を指定しています。この場合、「./size」サブディレクトリが使われます。

3. `xc32-gcc -O2 foo.c`
   `xc32-g++ -O2 foo.cpp`

   この例では、コマンドライン オプション -O2 により、サイズ優先でも速度優先でもない最適化を指定しています。この場合、「.」サブディレクトリが使われます。

4. `xc32-gcc -Os -mips16 foo.c`
   `xc32-g++ -Os -mips16 foo.cpp`

   この例の場合、サイズ優先の最適化 (-Os) と MIPS16 コード (-mips16) 向けのコマンドライン オプションを指定しています。この場合、「./size/mips16」サブディレクトリが使われます。

5.5.1.1 標準ライブラリ

C/C++ 標準ライブラリは、標準化された関数のセット (文字列、算術、入出力ルーチン等) を含みます。これらの関数のレンジについては**第 15 章「ライブラリ ルーチン」**に記載しています。

これらのライブラリは、コードジェネレータの出力コードによって暗黙的に呼び出される C/C++ ルーチンも含みます。これらのルーチンは浮動小数点演算等のタスクを実行し、ソースコード内の C/C++ 関数呼び出しには直接対応しません。

5.5.1.2 ユーザ定義ライブラリ

必要に応じ、ユーザ定義ライブラリを作成してプログラムとリンクできます。ライブラリ ファイルの方が扱いが容易でコンパイル時間も短くて済みますが、そのプロジェクトで使うターゲット デバイスとオプションに対応している必要があります。ライブラリを異なるプロジェクトで使うには、バージョンの異なるライブラリを作成する必要があります。

プロジェクトをビルドする際に検索する必要があるユーザ定義ライブラリは、コマンドラインでソースファイル名と一緒に指定します。

標準 C/C++ ライブラリ関数と同様に、ユーザ定義ライブラリ内の全ての関数の宣言もヘッダファイルに追加する必要があります。通常は、1 つまたは複数のヘッダファイルを作成し、それらをライブラリ ファイルと一緒にパッケージングします。これらのヘッダファイルは、必要に応じてソースコードにインクルードできます。

© 2015 Microchip Technology Inc.

5.5.2 周辺モジュール ライブラリの関数

PIC32 周辺モジュールの多くは、コンパイラツールと一緒に提供される周辺モジュール ライブラリ関数によってサポートされます。それらの関数の詳細は『32-Bit Language Tools Libraries』(DS51685) を参照してください。

5.6 起動と初期化

> **Note:** 一部の PIC32 では、デバイス コンフィグレーション ビット (BOOTISA) を使って、MIPS32® または microMIPS™ ISA モードのどちらかでブートするよう選択できます。これらのデバイスの BOOTISA ビットを microMIPS モード向けに設定した場合、xc32-gcc/g++ コンパイル ドライバに対して `-mmicromips` オプションを指定する事により、スタートアップ コードの microMIPS バージョンをリンクさせます。BOOTISA ビットを MIPS32 モード向けに設定した場合、コンパイル ドライバに対して `-mno-micromips` オプションを指定する事により、スタートアップ コードのMIPS32バージョンをリンクさせます。

C の場合 :

起動モジュールは 1 つしかなく、これを使って C ランタイム環境を初期化します。このためのソースコードは `<install-directory>/pic32-libs/libpi32c/startup/crt0.S` 内にあり、ライブラリ `<install-directory>/pic32mx/lib/crt0.o` 内にプリコンパイルされます。デバイスファミリ間のアーキテクチャの違いをサポートするため、これらのモジュールの multilib バージョンが用意されています。

C++ の場合 :

5 つのオブジェクト ファイルのコードを順番にリンクする事で、C++ ランタイム環境を初期化するための 1 つの初期化ルーチンを生成します。

このためのソースコードは`<install-directory>/pic32-libs/libpic32/startup` 内にあります。

PIC32のプリコンパイル済み起動オブジェクトは`<install-directory>/pic32mx/lib/` 内にあり、それらのファイル名は `cpprt0.o`、`crti.o`、`crtn.o` です。

GCC のプリコンパイル済み起動オブジェクトは `<install-directory>/lib/gcc/pic32mx/<gcc-version>/` 内にあり、それらのファイル名は `crtbegin.o` と `crtend.o` です。デバイスファミリ間のアーキテクチャの違いと最適化設定をサポートするため、これらのモジュールの multilib バージョンが用意されています。

これらの起動モジュール内のコードの実際の動作については、**14.3「スタートアップ コード」**を参照してください。

5.7 コンパイラ出力

コンパイラは、コンパイル中に各種のファイルを生成します。これらの多くは中間生成ファイルであり、その一部はコンパイル完了後に削除されます。しかし、多くは残されて、デバイスのプログラミングまたはデバッグ用に使われます。

5.7.1 出力ファイル

コンパイル ドライバは、以下の拡張子 (大文字と小文字は区別) を持つ出力ファイルを生成可能です。

表 5-3: ファイル名

拡張子	定義
`file.hex`	実行ファイル
`file.elf`	ELF デバッグファイル
`file.o`	オブジェクト ファイル (中間生成ファイル)
`file.s`	アセンブリコード ファイル (中間生成ファイル)
`file.i`	前処理済み C ファイル (中間生成ファイル)
`file.ii`	前処理済み C++ ファイル (中間生成ファイル)
`file.map`	マップファイル

多くの出力ファイルは、元のソースファイルと同じベース名を持ちます。例えば、ソースファイル `input.c` からオブジェクト ファイル `input.o` が生成されます。

`-o`オプションを使って名前を指定しない場合、メイン出力ファイルは名前`a.out`を持つ ELF ファイルです。

IDE (MPLAB IDE 等) を使う場合、通常は各アプリケーション向けに生成されるプロジェクトファイルを使ってコンパイラのオプションを指定します。ユーザが指定しない限り、プロジェクト内で使われる出力ファイルのベース名にはプロジェクト名が使われます。詳細は、実際に使う IDE のマニュアルを参照してください。

> **Note:** 本書では、「プロジェクト名」は IDE 内で作成されるプロジェクトの名前を指します。

本コンパイラは、Microchip 社製開発ツールで使う各種フォーマットの出力ファイルを直接生成できます。

既定値では、`xc32-gcc` と `xc32-g++` は ELF 出力ファイルを生成します。ファイルの出力またはファイル名の変更方法については、**5.9「ドライバ オプションの説明」**を参照してください。

5.7.2 診断ファイル

コンパイラは 2 つの重要なファイルを生成します。すなわち、アセンブラがアセンブリ リストファイルを生成し、リンカがマップファイルを生成します。

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

アセンブラでリストファイルを生成するためのオプションは `-a` (ドライバに渡す場合は `-Wa,-a`) です。このオプションには多数のバリエーションがあり、それらについては『MPLAB XC32 Assembler, Linker and Utilities User's Guide』(DS50002186) を参照してください。コンパイラからオプションを渡す方法については、**5.9.9「アセンブリングのためのオプション」**を参照してください。

© 2015 Microchip Technology Inc.

各ビルドに対して 1 つのリストファイルが生成されます。各変換ユニット (translation unit) に対して 1 つのアセンブラ リストファイルが生成されます。これはリンク前のアセンブラリストであるため、最終的なアドレスを示しません。従って、各ソースファイルに対してリストファイルが必要な場合、これらのファイルは別々にコンパイルする必要があります (**5.3.2「マルチステップの C コンパイル」**参照)。これは、MPLAB IDE を使ってビルドする場合に適用されます。各リストファイルには、モジュール名に拡張子 `.lst` を追加した名前が付けられます。

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

リンカでマップファイルを生成するためのオプションは `-Map` *`file`* (ドライバに渡す場合は `-Wl,-Map=file`) です。これについては『MPLAB Assembler, Linker and Utilities for PIC32 User's Guide』を参照してください。コンパイラからオプションを渡す方法については、**5.9.10「リンクのためのオプション」**を参照してください。

リンカの実行が正常に完了すると、プロジェクトのビルド時に 1 つのマップファイルが生成されます。

5.8 コンパイラのメッセージ

メッセージには 3 タイプがあります。以下では、各タイプのメッセージが出力された時のコンパイラの挙動について説明します。

警告メッセージ : コンパイルは可能だが正常とは認められない (コード実行時に障害が生じる可能性のある) ソースコードまたは状況を示します。警告の原因となったコードまたは状況は調査する必要があります。しかし、そのモジュールのコンパイルは続行され、残りのモジュールもコンパイルされます。

エラーメッセージ : 異常またはコンパイルが不可能なソースコードを示します。そのモジュール内の残りのソースコードに対するコンパイルは試みられますが、後続のモジュールはコンパイルされないまま処理が終了します。

致命的エラーメッセージ : コンパイルを続行できない (コンパイル処理を即座に停止する必要がある) 状況を示します。

コンパイラのエラー、警告、コメント出力を制御するためのオプションについては、**5.9.4「C++ の方言を制御するためのオプション」**を参照してください。

5.9 ドライバ オプションの説明

全ての 1 文字オプションは、前にダッシュ記号「-」を付ける事で識別します (例 : -c)。一部の 1 文字オプションでは、オプション文字に続けて (スペースを挿入する事なく) 追加のデータフィールドを指定します (例 : -Idir)。オプションは大文字と小文字を区別します。従って -c と -C は異なるオプションです。

コンパイラは、コンパイルを制御するためのオプションを多数備えており、それらは全て大文字と小文字を区別します。

- PIC32 専用オプション
- 出力の種類を制御するためのオプション
- C 言語の方言を制御するためのオプション
- C++ の方言を制御するためのオプション
- デバッグのためのオプション
- 最適化を制御するためのオプション
- プリプロセッサを制御するためのオプション
- アセンブリングのためのオプション
- リンクのためのオプション
- ディレクトリを検索するためのオプション
- コード生成規則のオプション

5.9.1 PIC32 専用オプション

これらのオプションはデバイスに固有のオプションであり、コンパイラに固有の物ではありません。

表 5-4: PIC32 専用のオプション

オプション	定義
-G *num*	サイズが *num* バイト以下のグローバルおよび静的アイテムを、通常よりも小さなデータまたは bss セクションに入れます。これは、単一命令によるデータへのアクセスを可能にします。全てのモジュールは、同じ -G *num* 値を使ってコンパイルする必要があります。
-mappio-debug	MPLAB ICD 3 デバッガおよび MPLAB REAL ICE インサーキット エミュレータ向けに APPIN/APPOUT デバッグ ライブラリ関数を有効にします。この機能により、DBPRINTF および関連する関数とマクロは『32-bit Language Tool Libraries』(DS51685) に記載されている方法で使えるようになります。このオプションは、ターゲットの PIC32 デバイスが APPIN/APPOUT 機能をサポートしている場合のみ有効にできます。
-mcci	Microchip 社の CCI (Common C Interface) コンパイルモードを有効にします。
-mcheck-zero-division -mno-check-zero-division	ゼロによる整数除算をトラップします(トラップしません)。既定値は -mcheck-zero-division です。
-membedded-data -mno-embedded-data	最初に変数を読み出し専用データセクション内に割り当てる事を試み、それができなければ小さなデータセクションに割り当てる事を試み、それもできなければデータ内に割り当てます。これにより、コードは既定値より少し遅くなりますが、実行時の RAM 要求量が減少するため、一部の組み込みシステムに適しています。
-mframe-header-opt	受け取ったフレームヘッダを使わない関数があれば、コンパイラはそれらの関数の一部の命令を省略します。この機能を使うと、通常は実行速度とコードサイズの両方が向上します。

 © 2015 Microchip Technology Inc.

表 5-4: PIC32 専用のオプション (続き)

オプション	定義
-minterlink-compressed	MIPS16 および microMIPS コードとリンク互換性を持つコードを生成します。
-mips16 -mno-mips16	MIPS16 コードを生成します (生成しません)。これは PRO エディションのコンパイラでのみ利用できます。
-mjals -mno-jals	分岐遅延スロット命令が16 ビットになり得る事を認識する事により、microMIPS 向けに jals 命令を生成します (生成しません)。これは、「16 ビットの分岐遅延スロット命令をサポートする jals 命令は存在しないため、リンク処理中に関数呼び出しは現在のモードを切り換える事ができない」という事を意味します。microMIPS オブジェクト / ライブラリと MIPS32 オブジェクト / ライブラリのリンクを試みた時にリンクエラーが発生する場合、-mno-jals を使う必要があるかもしれません。
-mlong-calls -mno-long-calls	jal 命令の使用を無効にします(有効にします)。jal を使って関数を呼び出した方が効率的ですが、呼び出す側と呼び出される側が同じ 256 MB セグメント内に存在する事が必要です。このオプションは、abicalls コードに対して効力を持ちません。 既定値は -mno-long-calls です。
-mmemcpy -mno-memcpy	非自明のブロック移動に対して memcpy() を適用します (適用しません)。既定値は -mno-memcpy です。この場合、GCC は固定サイズのコピーの大部分をインライン展開できます。
-mmicromips -mno-micromips	microMIPS™ 命令を生成します (生成しません)。この機能は PRO エディションのコンパイラでのみ利用できます。microMIPS 圧縮 ISA モードでブートするようターゲット デバイスを設定した場合(例: #pragma config BOOTISA=MICROMIPS)、リンク時に -mmicromips オプションを使って microMIPS 起動コードを指定します。
-mno-float	ソフトウェア浮動小数点ライブラリを使いません。
-mno-peripheral-libs	現在、-mno-peripheral-libs が既定値であり、-mperipheral-libs はオプションです。既定値の場合、デバイス専用リンカスクリプトで指定されている周辺モジュール ライブラリをリンクします。リンク時に標準の周辺モジュール ライブラリは使いません。
-mprocessor	コンパイルのターゲット デバイスを選択します (例 : -mprocessor=32MX360F512L)。
-msmart-io=[0|1|2]	このオプションは、printf および scanf 関数とそれらの「f」および「v」バリエーションに渡される書式文字列の静的な解析を試みます。非浮動小数点フォーマットの引数が使われている場合、それらの引数は、ライブラリ関数の整数専用バリエーションを使うために変換されます。多くのアプリケーションでは、この機能を使う事でプログラムメモリの使用量を削減できます。 -msmart-io=0 はこのオプションを無効にします。-msmart-io=2 を指定するとコンパイラは寛容になり、変数または未知フォーマットの引数を持つ関数呼び出しを変換します。既定値は -msmart-io=1 です。この場合、コンパイラは、浮動小数点サポートは不要であると判明した場合にのみ変換します。
-muninit-const-in-rodata -mno-uninit-const-in-rodata	未初期化 const 変数を読み出し専用データセクションに入れます。このオプションは、-membedded-data と組み合わせた場合にのみ効力を持ちます。

5.9.2 出力の種類を制御するためのオプション

以下のオプションは、コンパイラが生成する出力の種類を制御します。

表 5-5: 出力の種類を制御するオプション

オプション	定義
-c	ソースファイルをコンパイルまたはアセンブルしますが、リンクはしません。既定値のファイル拡張子は .o です。
-E	前処理段の後（本来のコンパイルの実行前）に停止します。既定値の出力ファイルは stdout です。
-fexceptions	例外処理を有効にします。C++ で書かれた例外ハンドラと相互動作する必要がある C コードをコンパイルする場合、このオプションを有効にする事が必要になる場合があります。
-o file	出力を file に保存します。
-S	本来のコンパイルが正常に完了した後(アセンブラを起動する前)に停止します。既定値のファイル拡張子は .s です。
-v	コンパイルの各処理段で実行された命令を表示します。
-x	以下のように -x オプションを指定する事で、入力言語を明示的に指定できます。 -x *language* ファイル名の拡張子に基づいてコンパイラに既定値言語を選ばせるのではなく、明示的に言語を指定します。指定された言語は、次に -x オプションが指定されるまで、後続の全ての入力ファイルに対して適用されます。コンパイラは以下の値をサポートします。 c c++ c-header cpp-output assembler assembler-with-cpp -x none 言語に関する全ての指定を無効にします。後続のファイルはファイル名の拡張子に従って処理されます。これは既定値の挙動ですが、これ以外の -x オプションを指定した後に既定値に戻すには、-x none オプションを指定する必要があります。 例：xc32-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s 上記を指定した場合、-x none を指定しない限り、コンパイラは全ての入力ファイルをアセンブラ向けであると見なします。
--help	コマンドライン オプションの説明を表示します。

 © 2015 Microchip Technology Inc.

5.9.3 C 言語の方言を制御するためのオプション

以下のオプションは、コンパイラで使う C 言語の方言の種類を定義します。

表 5-6: C 方言の制御オプション

オプション	定義
-ansi	ANSI 規格に準拠する C プログラムだけをサポートします。
-aux-info filename	filename で指定された名前を持つファイルに、変換ユニット内で宣言および / または定義されている全ての関数 (ヘッダファイル内の関数も含む) のプロトタイプ宣言を出力します。C 以外の言語では、このオプションは全く無視されます。このファイルは宣言を示すだけでなく、コメント内に各種の情報も示します (情報の内容 : 各宣言元 (ソースファイルと行)、宣言が暗黙的 / プロトタイプ / 非プロトタイプのどれであったか (行番号とコロンの後の最初の文字 (I/N (new)/O (old)) のいずれか)、それが宣言であったのか定義であったのか (その次の文字 (C または F))。関数定義の場合、宣言の後のコメント内に、K&R スタイルの引数のリスト (その後にそれらの引数の宣言が続く) も示されます。
-fcheck-new / -fno-check-new (default)	演算子 new が返したポインタが非 NULL かどうかチェックします。
-fenforce-eh-specs (default) / -fno-enforce-eh-specs	実行時に例外仕様の違反をチェックするためのコードを生成します (生成しません)。-fno-enforce-eh-specs オプションは C++ 規格に違反しますが、プロダクション ビルドにおけるコードサイズの削減に役立つ場合があります (「NDEBUG」を定義した場合に類似)。しかし、これによってユーザコードが例外仕様に違反する例外をスローできるようになるという事はありません。コンパイラはあくまでも仕様に基づいて最適化するため、ユーザコードが予期せぬ例外をスローすると未定義の挙動が生じます。
-ffreestanding	コンパイルを自立環境で実行するよう指定します。これは暗黙のうちに -fno-builtin を指定します。自立環境では、標準ライブラリが存在するとは限らず、プログラムが必ずしも main で開始されるとは限りません。最も分かりやすい例が OS カーネルです。これは -fno-hosted と等価です。
-fno-asm	asm、inline、typeof をキーワードとして認識しません。コードはこれら3つの単語を識別子として使う事ができます。これらに代わるキーワードとして __asm__、__inline__、__typeof__ が使えます。-ansi は暗黙のうちに -fno-asm を指定します。
-fno-builtin -fno-builtin-function	名前が接頭辞「__builtin_」で始まらないビルトイン関数を認識しません。
-fno-exceptions	C++ 例外処理を無効にします。このオプションは、例外の伝播に必要な追加コードの生成を無効にします。
-fno-rtti	ランタイム型識別機能を有効 / 無効にします。-fno-rtti オプションは、C++ランタイム型識別機能(dynamic_cast と typeid) が使う仮想関数を持つ各クラスに関する情報の生成を無効にします。C++ 言語のこれらの機能を使わない場合、このフラグを指定する事でメモリ空間をある程度節約できます。例外処理も同じ情報を使いますが、例外処理は必要に応じてこれらの情報を生成するという事に注意してください。ランタイム型識別機能を無効にしても、dynamic_cast 演算子はランタイム型情報を要求しないキャスト (void * または一義的ベースクラスへのキャスト) 向けに使えます。

表 5-6: C 方言の制御オプション (続き)

オプション	定義
-fsigned-char	char型を符号付きとして(signed charと同様に)扱います。**(これが既定値です)**
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	これらのオプションは、宣言で符号付き / 符号なしが指定されていない場合に、ビットフィールドの符号付き / 符号なしを制御します。既定値では、そのようなビットフィールドは符号付きです。しかし -traditional を指定すると、常に符号なしとして扱います。
-funsigned-char	char 型を符号なしとして (unsigned char と同様に) 扱います。
-fwritable-strings	文字列を書き込み可能データセグメントに保存します。それらの文字列は一意に保たれません (同じ文字列を複数個保存可能)。

5.9.4 C++ の方言を制御するためのオプション

以下のオプションは、コンパイラで使う C++ 言語の方言の種類を定義します。

表 5-7: C++ 方言の制御オプション

オプション	定義
-ansi	ANSI 規格に準拠する C++ プログラムだけをサポートします。
-aux-info filename	filenameで指定された名前を持つファイルに、変換ユニット内で宣言および/または定義さている全ての関数 (ヘッダファイル内の関数も含む) のプロトタイプ宣言を出力します。C++ 以外の言語では、このオプションは完全に無視されます。このファイルは宣言を示すだけでなく、コメント内に各種の情報も示します (情報の内容 : 各宣言元 (ソースファイルと行)、宣言が暗黙的 / プロトタイプ / 非プロトタイプのどれであったか (行番号とコロンの後の最初の文字 (I/N (new)/O (old)) のいずれか)、それが宣言であったのか定義であったのか (その次の文字 (C または F))。関数定義の場合、宣言の後のコメント内に、K&R スタイルの引数のリスト (その後にそれらの引数の宣言が続く) も示されます。
-ffreestanding	コンパイルを自立環境で実行するよう指定します。これは暗黙のうちに -fno-builtin を指定します。自立環境では、標準ライブラリが存在するとは限らず、プログラムが必ずしも main で開始されるとは限りません。最も分かりやすい例が OS カーネルです。これは -fno-hosted と等価です。
-fno-asm	asm、inline、 typeof をキーワードとして認識しません。コードはこれら3つの単語を識別子として使う事ができます。これらに代わるキーワードとして __asm__、__inline__、__typeof__ が使えます。-ansi は暗黙のうちに -fno-asm を指定します。
-fno-builtin -fno-builtin-function	名前が接頭辞「__builtin_」で始まらないビルトイン関数を認識しません。
-fsigned-char	char 型を符号付きとして (signed char と同様に) 扱います。**(これが既定値です)**
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	これらのオプションは、宣言で符号付き / 符号なしが指定されていない場合に、ビットフィールドの符号付き / 符号なしを制御します。既定値では、そのようなビットフィールドは符号付きです。しかし -traditional を指定すると、常に符号なしとして扱います。
-funsigned-char	char 型を符号なしとして (unsigned char と同様に) 扱います。
-fwritable-strings	文字列を書き込み可能データセグメントに保存します。それらの文字列は一意に保たれません (同じ文字列を複数個保存可能)。

 © 2015 Microchip Technology Inc.

5.9.5 警告とエラーを制御するためのオプション

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

-Wで始まるオプションを使う事で、特定の警告を要求できます(例: -Wimplicitは暗黙的な宣言に対する警告を要求)。これらの各オプションには、その警告を無効にするための -Wno- で始まる否定形が存在します (例 : -Wno-implicit)。本書では、そのように 2 形態あるオプションのうち、既定値ではない方のオプションについて説明します。

以下のオプションは、コンパイラが生成する警告の量と種類を制御します。

表 5-8: -Wall によって暗黙的に有効になる警告とエラー

オプション	定義
-fsyntax-only	コードの構文をチェックするだけで、それ以上の事はしません。
-pedantic	厳密な ANSI C が要求する全ての警告を出力します。
-pedantic-errors	厳密な ANSI C 規格が要求する全てのエラーを出力します。
-w	全ての警告メッセージを抑止します。
-Wall	このオプションは、ユーザによっては疑わしいと見なす可能性のある構文や、簡単に警告を回避できる (または警告が出ないように修正できる) ような構文に関する警告を全て有効にします。マクロに関連する構文であっても、警告を出力します。一部の警告フラグは、-Wall によって暗黙的に有効にならないと言う事に注意してください。そのような警告フラグには、普段は疑わしいと見なされないものの時にはチェックが必要になる可能性がある構文に対する物と、場合によっては警告が必要になるものの回避が困難 (コードを変更して簡単に警告を抑止できない) 構文に対する物があります。それらの一部は -Wextra によって有効になりますが、多くは個別に有効にする必要があります。
-Waddress	メモリアドレスの疑わしい使用に関して警告します。これには、条件式内での関数アドレスの使用(例: void func(void); if (func))と、文字列リテラルのメモリアドレスに対する比較(例: if (x == "abc"))が含まれます。通常、このようなメモリの使用に対してはプログラマー エラーが報告されます。関数のアドレスは常に「真」として評価されるため、それらを条件式内で使うと、通常は「プログラマーが関数呼び出し内でカッコを書き忘れた」と報告されます。また、文字列リテラルに対する比較は未定義の挙動を生じ、C での移植性に欠けるため、通常は「プログラマーが strcmp の使用を意図した」と報告されます。
-Wchar-subscripts	配列の添え字が char 型である場合に警告します。
-Wcomment	コメントの開始を示す文字列 /* が /*.....*/ 型のコメント内で使われた場合、またはバックスラッシュ + 改行が /...../ 型のコメント内で使われた場合に警告します。
-Wdiv-by-zero	コンパイル時の 0 による整数除算に対して警告します。この警告メッセージを抑止するには -Wno-div-by-zero を使います。0による浮動小数点除算に関しては警告しません (無限大と NaN を得るための正規の方法であるため)。**(これが既定値です)**
-Wformat	printf や scanf 等に対する呼び出しにおいて、提供された引数の型が指定された書式文字列に適合しているかどうかチェックします。

表 5-8: -Wall によって暗黙的に有効になる警告とエラー (続き)

オプション	定義
-Wimplicit	これは -Wimplicit-int と -Wimplicit-function-declaration の両方を指定した場合と等価です。
-Wimplicit-function-declaration	関数が宣言される前に使われた場合に警告を出力します。
-Wimplicit-int	宣言で型が指定されていない場合に警告を出力します。
-Wmain	main の型が疑わしい場合に警告を出力します。main は int 型の値を返し、適切な型の 0 または 2 または 3 個の引数を受け取る外部結合の関数である事が必要です。

-Wmissing-braces	集合体または共用体の初期化子のカッコが不完全である場合に警告を出力します。以下の例の場合、a の初期化子のカッコは不完全ですが、b は完全です。
	`int a[2][2] = { 0, 1, 2, 3 };`
	`int b[2][2] = { { 0, 1 }, { 2, 3 } };`

-Wno-multichar	複数文字の char 型定数が使われている場合に警告を出力します。通常、そのような定数はタイプミスです。そのような定数は処理系定義であるため、移植性が求められるコードで使うべきではありません。以下に、複数文字の char 型定数の使用例を示します。
	`char`
	`xx(void)`
	`{`
	`return('xx');`
	`}`

-Wparentheses	特定の文脈内でカッコが省略された場合に警告を出力します (例 : 真理値が期待される文脈内での代入、優先順位にしばしば混乱を招くような演算子の使い方)。
-Wreturn-type	既定値で int 型になる戻り値を持つ関数が定義された場合、常に警告を出力します。戻り値の型が void ではない関数内に戻り値を持たない return 文が存在する場合にも警告を出力します。

© 2015 Microchip Technology Inc.

表 5-8: -Wall によって暗黙的に有効になる警告とエラー (続き)

オプション	定義
-Wsequence-point	C 規格の副作用完了点規則に違反するために未定義の意味を持つ可能性のあるコードに対して警告を出力します。C 規格は C プログラム内の式を評価する順序を副作用完了点に基づいて定義します。これは、副作用完了点の前後で実行されるプログラム部分の間の順序を表します。副作用完了点は以下の位置で発生します。 - 完全式 (より大きな式の一部ではない式) の評価後 - 演算子「&&」、「||」、「?」、「:」、「,」(コンマ) の第 1 オペランドの評価後 - 関数を呼び出す前 (しかし引数と呼び出す関数を指定する式の評価後) - その他の特定箇所 副作用完了点規則の他には、式の副次式を評価する順序を規定する物はありません。これらの規則は全て総合的な順序ではなく部分的な順序だけを定義します。例えば、2 つの関数が 1 つの式内で間に副作用完了点を持たずに呼び出される場合、それらの関数が呼び出される順序は未定義です。しかし、C 規格委員会は、関数呼び出しはオーバーラップしないと規定しています。2 つの副作用完了点の間でオブジェクトの値を変更した場合、その変更がどの時点で効力を生じるかは規定されていません。これに依存するプログラムのふるまいは未定義です。C 規格は、「オブジェクトの保存値は、直前の副作用完了点と次の副作用完了点の間で、式の評価によって一度だけ変更される。」と規定しています。さらに、「先の値は、保存する値を決定するためにのみ読み出される。」と規定しています。プログラムがこの規則に違反する場合、どのような処理系であっても結果は全く予測不可能です。以下は、未定義のふるまいをするコードの例です。 a = a++;, a[n] = b[n++] and a[i++] = i; このオプションは、あまり複雑なケースを診断しません。場合によっては誤診断が生じますが、プログラム内の上記のような問題を検出するにはかなり効果的である事が分かっています。
-Wswitch	switch文が列挙型のインデックスを持ち、その列挙体の中で挙げられている 1 つまたは複数のコードに case が付けられていない場合、警告を出力します (default ラベルがある場合、この警告は出力されません)。このオプションを指定した場合、列挙体の外側の case ラベルによっても警告が出力されます。
-Wsystem-headers	システム ヘッダファイル内の構文に関する警告メッセージを出力します。通常は、システムヘッダからの警告は抑止されます。なぜならば、それらは実際の問題を示さず、コンパイラ出力を読みにくくするだけだからです。このコマンドライン オプションを指定すると、コンパイラはシステムヘッダからの警告を、あたかもそれらがユーザコード内で発生したかのように出力します。しかし、このオプションと一緒に -Wall を指定すると、システムヘッダ内の未知のプラグマに関する警告は出力されないという事に注意してください。その場合、-Wunknown-pragmas も指定する必要があります。
-Wtrigraphs	3 文字表記のサポートが有効になっている場合、3 文字表記を検出すると警告を出力します。

表 5-8: `-Wall` によって暗黙的に有効になる警告とエラー (続き)

オプション	定義
-Wuninitialized	自動変数が初期化されずに使われると警告を出力します。これらの警告は、最適化時にのみ計算されるデータ フロー情報を必要とするため、最適化を有効にしている場合のみ出力されます。これらの警告は、レジスタに割り当てられる候補となっている変数に対してのみ出力されます。従って、volatile 宣言されている変数、アドレスが取られている変数、サイズが 1/2/4/8 バイト以外の変数に対しては、この警告は出力されません。また、構造体、共用体、配列に対しては、たとえそれらがレジスタ内に配置されていても、この警告は出力されません。一度も使われる事のない値を計算するためにだけ使われている変数に対しても、この警告は出力されません。なぜならば、そのような計算は、警告が出力される前にデータ フロー解析によって削除されるからです。
-Wunknown-pragmas	コンパイラにとって未知の #pragma ディレクティブが検出された場合に警告を出力します。このコマンドライン オプションを指定すると、システム ヘッダファイル内に未知のプラグマが存在する場合にも警告を出力します。しかし、-Wall コマンドライン オプションのみによってこの警告が有効にされている場合、ヘッダファイル内の未知のプラグマに関する警告は出力されません。
-Wunused	宣言以外では使われていない変数、静的関数として宣言されたのに定義されていない関数、宣言されたのに使われていないラベル、演算結果が明示的に使われていない命令文に対して警告を出力します。未使用の関数パラメータに関する警告を出力するには、-W と -Wunused の両方を指定する必要があります。式を void にキャストする事で、式に対するこの警告は抑止されます。同様に、unused 属性は未使用の変数、パラメータ、ラベルに対するこの警告を抑止します。
-Wunused-function	宣言された静的関数が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われていない場合に警告を出力します。
-Wunused-label	宣言されたのに使われていないラベルがあると警告を出力します。この警告を抑止するには、unused 属性を使います。
-Wunused-parameter	宣言以外では使われていない関数パラメータがあると警告を出力します。この警告を抑止するには、unused 属性を使います。
-Wunused-variable	宣言以外では使われていないローカル変数または非定数の静的変数があると警告を出力します。この警告を抑止するには、unused 属性を使います。
-Wunused-value	計算した結果が明示的に使われていない命令文があると警告を出力します。この警告を抑止するには、式を void にキャストします。

 © 2015 Microchip Technology Inc.

以下の -W オプションは、-Wall を指定しても有効になりません。それらの一部は、普段は疑わしいと見なされないものの、時にはチェックが必要になる可能性がある構文に対する警告です。その他は、場合によっては必要になるものの、回避が困難 (コードを変更して簡単に警告を抑止できない) 構文に対する警告です。

表 5-9: -Wall によって暗黙的に有効にならない警告とエラー

オプション	定義

-W	以下のイベントに対する追加の警告メッセージを出力します。
	• 非 volatile の自動変数が longjmp に対する呼び出しによって変更される可能性がある。これらの警告は、最適化コンパイルを行う場合にのみ出力可能です。コンパイラは、setjmp に対する呼び出しを監視するだけであり、longjmp がどこで呼び出されるかは関知しません。実際、シグナルハンドラは、コード内のどこででも longjmp を呼び出す事ができます。その結果、実際には問題がなくても警告が出力される可能性があります。なぜならば、longjmp が問題を起こすような位置で呼び出される事は実際にはあり得ないためです。
	• 関数が return value; と return; のどちらによってでも終了できるようになっている。リターン文を一切渡さない関数本体の終了は return; として扱われます。
	• 式文またはコンマ式の左辺が一切の副作用を持たない。この警告を抑止するには、使われていない式を void にキャストします。例えば、x[i,j] は警告を生じますが、x[(void)i,j] は警告を生じません。
	• 符号なし値が < または <= を使って 0 と比較されている。
	• x<=y<=z のような比較が行われている。これは (x<=y ?1 :0) <= z と等価であり、通常の数学的表記法とは解釈が異なります。
	• 記憶域クラス指定子 (static 等) が宣言内の最初に置かれていない。C 規格によると、これは旧式 (obsolescent) の用法です。
	• このオプションと一緒に -Wall または -Wunused も指定した場合、未使用の引数に対して警告を出力します。
	• 符号付き値と符号なし値の比較において、符号付き値が符号なし値に変換されると不正な結果が生じる可能性がある。(ただし、-Wno-sign-compare も指定した場合、警告は出力されません)
	• 集合体の初期化子を囲むカッコが完全ではない。例えば以下のコードでは、x.h に対する初期化子の前後のカッコが抜けているため、この警告が出力されます。
	`struct s { int f, g; };`
	`struct t { struct s h; int i; };`
	`struct t x = { 1, 2, 3 };`
	• 集合体の初期化子が一部のメンバーしか初期化しない。例えば以下のコードでは、x.h が暗黙的に 0 に初期化されるため、この警告が出力されます。
	`struct s { int f, g, h; };`
	`struct s x = { 3, 4 };`

-Waggregate-return	構造体または共用体を返す関数が定義されるか呼び出された場合に警告を出力します。
-Wbad-function-cast	関数呼び出しが不適合な型にキャストされた場合に警告を出力します。例えば、int foof() が何らかのポインタ型 (*) にキャストされると警告を出力します。

© 2015 Microchip Technology Inc.

表 5-9: -Wall によって暗黙的に有効にならない警告とエラー (続き)

オプション	定義
-Wcast-align	ターゲットに必要なアラインメントのサイズが増加するようなポインタのキャストに対して警告を出力します。例えば、char * が int * へキャストされると警告を出力します。
-Wcast-qual	型指定子を削除するようなポインタのキャストに対して警告を出力します。例えば、const char * が char * にキャストされると警告を出力します。
-Wconversion	プロトタイプの存在によって生じる型変換が、プロトタイプが存在しなかった場合に同じ引数に対して生じるであろう型変換と異なる場合に警告を出力します。これには、固定小数点から浮動小数点への (またはその逆の) 変換と、固定小数点型引数のビット幅または符号の有無を変更する変換が含まれます (ただし既定値の拡張 (promotion) と同じ場合は除く)。また、負の整数定数式が暗黙的に符号なし型に変換される場合にも警告を出力します。例えば、x が符号なしの場合、代入式 x = -1 に対して警告を出力します。しかし、(unsigned) -1 のような明示的キャストに対しては、警告は出力されません。
-Werror	全ての警告をエラーにします。
-Winline	関数がインライン展開できない場合に、関数がインライン関数として宣言されるか、-finline-functions オプションが指定されると警告を出力します。
-Wlarger-than-len	len バイトよりも大きなオブジェクトが定義されると警告を出力します。
-Wlong-long -Wno-long-long	long long 型が使われると警告を出力します。これは既定値です。この警告メッセージを抑止するには、-Wno-long-long を指定します。-Wlong-long および -Wno-long-long フラグは、-pedantic フラグが指定されている場合にのみ考慮されます。
-Wmissing-declarations	グローバル関数が先に宣言される事なく定義されると警告を出力します。定義そのものがプロトタイプを提供する場合でも、警告を出力します。
-Wmissing-format-attribute	-Wformat が指定されている場合、書式属性の候補になる可能性のある関数に対しても警告を出力します。これらは候補になる可能性があるというだけであり、絶対にそうなるという事ではありません。-Wformat が指定されていない場合、このオプションは効力を持ちません。
-Wmissing-noreturn	noreturn 属性の候補になる可能性のある関数に対して警告を出力します。これらは候補になる可能性があるというだけであり、絶対にそうなるという事ではありません。関数に noreturn 属性を追加する前に、その関数が決してリターンしないという事を手作業で慎重に確認する必要があります。そうしないと、潜在性のコード生成バグを招く恐れがあります。
-Wmissing-prototypes	グローバル関数が先にプロトタイプ宣言される事なく定義されると警告を出力します。定義そのものがプロトタイプを提供する場合でも、警告を出力します。このオプションを使うと、ヘッダファイル内で宣言されていないグローバル関数を検出できます。
-Wnested-externs	関数の内部で extern 宣言が見付かると警告を出力します。
-Wno-deprecated-declarations	deprecated 属性が指定された (つまり将来のサポートが保証されない) 関数、変数、型が使われていても警告を出力しません。
-Wpadded	構造体のエレメントまたは構造体全体をアラインメントするためのパディングが構造体に含まれている場合に警告を出力します。

 © 2015 Microchip Technology Inc.

表 5-9: -Wall によって暗黙的に有効にならない警告とエラー (続き)

オプション	定義
-Wpointer-arith	関数型のサイズまたは void のサイズに依存する何かがあれば警告を出力します。void * ポインタと関数へのポインタを使う計算の便宜を図るため、コンパイラはこれらの型のサイズを 1 にします。
-Wredundant-decls	同一スコープ内で何かが複数回宣言された場合、たとえ複数回の宣言が有効であり、それらによって何も変更されなくても、警告を出力します。
-Wshadow	あるローカル変数が別のローカル変数をシャドー化している場合に警告を出力します。
-Wsign-compare -Wno-sign-compare	符号付き値と符号なし値の比較において、符号付き値が符号なし値に変換されると不正な結果が生じる可能性がある場合に警告を出力します。この警告は -W によっても有効になります。-W を指定した時にこの警告だけを抑止する必要がある場合、-W -Wno-sign-compare を使います。
-Wstrict-prototypes	引数の型が指定されずに関数が宣言または定義されると警告を出力します (旧式の関数定義は、引数の型を指定する宣言が前にあれば、警告を生じる事なく使えます)。
-Wtraditional	伝統的な C と ANSI C でふるまいが異なる特定の構文に対して警告を出力します。 • マクロ本体の中の文字列定数の内部にマクロ引数が現れる (これらの引数は伝統的 C では置換され、ANSI C では定数の一部になります)。 • あるブロック内で extern 宣言されている関数がブロックの終端よりも後で使われている。 • switch 文が long 型のオペランドを持つ。 • 静的な関数宣言の後に非静的な関数宣言が存在する (一部の伝統的 C コンパイラはこの構文を受け入れません)。
-Wundef	#if ディレクティブ内で未定義の識別子が評価されると警告を出力します。
-Wunreachable-code	決して実行される事のない (到達不能の) コードを検出すると警告を出力します。この警告が出力されても、到達不能と見なされたコードの一部の行は実行される場合があります。従って、到達不能コードを削除する際は、十分な注意が必要です。例えば、この警告は、インライン展開された複数の関数コピーの中の 1 つだけがその行に到達できないという事を意味しているかもしれません。
-Wwrite-strings	文字列定数のアドレスが const 指定なしの char * 型ポインタにコピーされた時に警告が出力されるよう、文字列定数に const char[length] 型を与えます。コンパイル時にこれらの警告を出力すると、文字列定数への書き込みを試みる事ができるコードを見付けるために役立ちます。しかし、これを行うには、宣言とプロトタイプで const を非常に注意深く扱う必要があります。そうしないと迷惑な警告が出力されるだけです。これらの警告が -Wall によって暗黙的に有効にならないのは、このためです。

5.9.6 デバッグのためのオプション

以下のオプションはデバッグ用に使います。

表 5-10: デバッグ オプション

オプション	定義
-g	デバッグ情報を生成します。-g を -O と一緒に使う事で、最適化されたコードをデバッグできます。コードの最適化により、以下のような全く予想外の結果が時として生じます。 • 宣言された変数が全く存在しない • 制御のフローが一時的に予想外の動きを示す • 一部の命令文が実行されない(それらの結果が不変または既に取得済みである場合) • 一部の命令文がループの外へ出されたために異なる位置で実行される このような結果が生じる可能性はありますが、最適化された出力をデバッグする事は可能です。このオプションを使う事で、バグを含んでいるかもしれないプログラムに対する最適化の実行は合理的な試みとなります。
-Q	コンパイラは、各関数をコンパイルするたびに関数名を表示し、各パスが終了するたびにパスに関する統計情報を表示します。
-save-temps -save-temps=cwd	中間生成ファイルを削除せずに、現在作業中のディレクトリに保存します。それらのファイルにはソースファイルと同じベース名が付けられます。例えば、-c -save-temps を指定して foo.c をコンパイルすると、以下のファイルが生成されます。 foo.i (前処理ファイル) 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)
-save-temps=obj	これは -save-temps=cwd に似ていますが、-o オプションが指定されている場合、一時ファイルはオブジェクト ファイルに基づきます。 -o オプションが指定されていない場合、-save-temps=obj スイッチの動作は -save-temps と同じです。 以下に例を示します。 xc32-gcc -save-temps=obj -c foo.c xc32-gcc -save-temps=obj -c bar.c -o dir/xbar.o xc32-gcc -save-temps=obj foobar.c -o dir2/yfoobar 上記はfoo.i、foo.s、dir/xbar.i、dir/xbar.s、dir2/yfoobar.i、dir2/yfoobar.s、dir2/yfoobar.o を生成します。

5.9.7 最適化を制御するためのオプション

以下のオプションは、コンパイラの最適化機能を制御します。

表 5-11: 一般的な最適化オプション

オプション	エディション	定義
-O0	全エディション	**最適化せず (これが既定値です)** -O を指定しない場合、コンパイラはコンパイルに要する時間とリソースを削減し、デバッグで予測通りの結果が得られる事を目標とします。各命令文は互いに独立しています。命令文と命令文の間にあるブレークポイントでプログラムを停止させると、任意の変数に対して新しい値を代入する事や、プログラム カウンタをその関数内の他の命令文へ変更する事ができ、ソースコードから期待した通りの結果が得られます。コンパイラは、register 宣言された変数だけをレジスタに割り当てます。
-O -O1	全エディション	最適化レベル 1 最適化を実行するとコンパイルに要する時間が増え、より大きな関数にはより多くのホストメモリが要求されます。 -O を指定すると、コンパイラはコードサイズと実行時間の削減を試みます。-O を指定すると、コンパイラは -fthread-jumps、-fdefer-pop、-fomit-frame-pointer を有効にします。

 © 2015 Microchip Technology Inc.

表 5-11: 一般的な最適化オプション (続き)

オプション	エディション	定義
-O2	STD、PRO	最適化レベル 2 コンパイラは、サポートする最適化の中でサイズと速度のトレードオフに関与しない最適化処理のほとんど全てを実行します。 -O2 はループ展開 (-funroll-loops)、関数のインライン展開 (-finline-functions)、厳格なエイリアシングの最適化 (-fstrict-aliasing) を除く全ての最適化オプションを有効にします。また、メモリオペランドの強制コピー (-fforce-mem) とフレームポインタの排除(-fomit-frame-pointer) も有効にします。このオプションを指定すると、-O に比べてコンパイル時間は長くなりますが、生成されるコードの性能は向上します。
-O3	PRO	最適化レベル 3 -O3 は、-O2 が適用する全ての最適化オプションに加えて、inline-functions オプションを有効にします。
-Os	PRO	サイズの最適化 -Os は、-O2 が適用する最適化オプションの中でコードサイズが一般的に増加しないオプションだけを全て有効にした上に、コードサイズの削減を目的とするさらなる最適化を実行します。

以下で説明するオプションは、各種の最適化機能を個別に制御します。-O2 オプションは、これらの個別オプションのうち -funroll-loops、-funroll-all-loops、-fstrict-aliasing を除く全てのオプションを有効にします。

最適化の微調整が必要になる稀なケースでは、以下のフラグが使えます。

表 5-12: 個別最適化オプション

オプション	定義
-falign-functions -falign-functions=n	関数の開始位置を n よりも大きくて n に最も近い 2 のべき乗バイト境界に、n バイト未満のスキップ幅で整列させます。例えば -falign-functions=32 は関数を次の 32 バイト境界に整列させますが、-falign-functions=24 は 23 バイト以下のスキップ幅で可能な場合にのみ関数を次の 32 バイト境界に整列させます。 -fno-align-functions と -falign-functions=1 は等価であり、どちらも関数を整列させない事を意味します。アセンブラは、n が 2 のべき乗値である場合にのみこのフラグをサポートするため、n は切り上げられます。n を指定しないと、ターゲット デバイスに固有の既定値が使われます。
-falign-labels -falign-labels=n	全ての分岐先を -falign-functions と同様に最大 n バイトのスキップ幅で 2 のべき乗バイト境界に整列させます。このオプションを使うと、コードの通常のフローで分岐先まで達するためにダミー演算を挿入する必要があるため、コードは単純に遅くなります。 -falign-loops または -falign-jumps が適用可能であり、しかもそれらの方が -falign-labels の値よりも大きい場合、それらの値が代わりに使われます。n を指定しないと、ターゲット デバイスに固有の既定値(ほとんどの場合は1 (整列させない))が使われます。
-falign-loops -falign-loops=n	ループを -falign-functions と同様に最大 n バイトのスキップ幅で 2 のべき乗バイト境界に整列させます。ループの実行回数が多ければ、ダミー演算による損失を埋め合わせる事ができる可能性があります。n を指定しないと、ターゲット デバイスに固有の既定値が使われます。
-fcaller-saves	関数呼び出しの前後にレジスタを退避 / 復元するための命令を追加する事により、関数呼び出しによって上書きされるレジスタに対する値の割り当てを可能にします。そのような割り当ては、結果としてコードが改善されると見込まれる場合にのみ行われます。
-fcse-follow-jumps	共通部分式除去 (CSE: Coommon Subexpression Elimination) では、ジャンプ命令のジャンプ先に到達する経路がそのジャンプ命令以外に存在しなければ、そのジャンプ命令を調べます。例えば、else 節を持つ if 文を検出すると、CSE は評価した条件式が「偽」の場合のジャンプを追跡します。

表 5-12: 個別最適化オプション (続き)

オプション	定義
-fcse-skip-blocks	これは -fcse-follow-jumps に似ていますが、CSE は条件付きでブロックをスキップするジャンプを追跡します。 -fcse-skip-blocks を指定した場合、else 節を持たない単純な if 文を検出すると、CSE は if 文本体を飛び越すジャンプを追跡します。
-fexpensive-optimizations	比較的コストのかかる各種の小さな最適化を実行します。
-ffunction-sections -fdata-sections	各関数またはデータアイテムを出力ファイル内の別々のセクションに配置します。出力ファイル内のセクションの名前は、関数またはデータアイテムの名前によって決まります。これらのオプションは、明らかな利点が得られる場合にのみ使います。これらのオプションを指定すると、アセンブラとリンカが生成するオブジェクトおよび実行ファイルのサイズが増加し、実行速度は低下する可能性があります。
-fgcse	グローバルな共通部分式除去パスを実行します。このパスは、グローバルな定数伝播とコピー伝播も実行します。
-fgcse-lm	-fgcse-lm を有効にすると、グローバルな共通部分式除去は単にストアだけ実行されるロードをループ外に移動しようと試みます。これにより、ループ内のロード / ストア シーケンスは、ループの外のロードとループ内のコピー / ストアに変更できます。
-fgcse-sm	-fgcse-sm を有効にすると、グローバルな共通部分式除去の後にストア移動パスが実行されます。このパスは、ストアをループの外へ移動させようと試みます。このオプションを -fgcse-lm と一緒に使うと、ループ内のロード / ストア シーケンスをループ前のロードとループ後のストアに変更できます。
-fmove-all-movables	ループ内にある全ての不変計算をループの外へ移動します。
-fno-defer-pop	関数がリターンすると、即座にその関数呼び出しに使った引数をポップします。コンパイラは通常、複数回の関数呼び出しに使った引数をスタックに蓄積し、それらを同時にポップします。
-fno-peephole -fno-peephole2	デバイスに固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、コンパイル中の各所で発生します。-fno-peephole はデバイス固有の命令に対するピープホール最適化を無効にし、-fno-peephole2 は高水準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を完全に無効にするには、両方のオプションを使います。
-foptimize-register-move -fregmove	レジスタをできるだけ束ねるために、move 命令および他の単純な命令のオペランドにおけるレジスタ番号の再割り当てを試みます。-fregmove と -foptimize-register-moves は同じ最適化を行います。
-freduce-all-givs	ループ内の全ての一般的誘導変数 (帰納変数) の強度を弱めます。これらのオプションは、コードを改善する事もあれば改悪する事もあります。効果はソースコード内のループの構造によって大きく異なります。
-frename-registers	レジスタの割り当て後に余ったレジスタを使って、コンパイル予定のコード内の誤った依存性の回避を試みます。多数のレジスタを備えるプロセッサには、この最適化が非常に効果的です。しかし、変数は元のレジスタに留まらないため、デバッグができなくなる可能性があります。
-frerun-cse-after-loop	ループ最適化を実行した後に、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 回実行します。

 © 2015 Microchip Technology Inc.

表 5-12: 個別最適化オプション (続き)

オプション	定義
`-fschedule-insns`	必要なデータが利用できる状態になっていないために生じる命令ストールを防ぐため、命令の実行順の変更を試みます。
`-fschedule-insns2`	`-fschedule-insns` と似ていますが、レジスタの割り当て後に、命令スケジューリングの追加パスを要求します。
`-fstrength-reduce`	ループ強度の削減と繰り返し変数の除去による最適化を実行します。

`-fstrict-aliasing`	このオプションを指定すると、コンパイラは、コンパイル中の言語に適用可能なエイリアシング規則の中で最も厳格な規則を想定します。C 言語の場合、このオプションは式の型に基づく最適化を有効にします。特に、型が異なるオブジェクト同士が同じアドレスを共有する事はないと想定されます(型がほとんど同じである場合を除く)。例えば、`unsigned int` は `int` のエイリアスになれますが、`void*` や `double` のエイリアスにはなれません。文字型は他の全ての型のエイリアスになれます。以下のコードには特に注意が必要です。
	`union a_union {`
	`  int i;`
	`  double d;`
	`};`
	
	`int f() {`
	`  union a_union t;`
	`  t.d = 3.0;`
	`  return t.i;`
	`}`
	共用体メンバーを読み出す場合、直近に書き込んだメンバーとは異なるメンバーから読み出す事はよくあります (これを「型のパンニング」と呼ぶ)。`-fstrict-aliasing` を指定した場合でも、`union` 型を介してメモリにアクセスする場合は型のパニングが許容されます。従って、上記のコードは期待通りに動作します。しかし、下記のコードは期待通りに動作しません。
	`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事により、最適化を実行します。そうである場合、条件式の真偽に応じて、最初の分岐の分岐先を2番目の分岐の分岐先かその直後のどちらかに変更します。
`-funroll-loops`	ループ展開の最適化を実行します。これは、コンパイル時または実行時に繰り返し数が特定可能なループに対してのみ実行されます。`-funroll-loops` を指定すると、暗黙的に `-fstrength-reduce` と `-frerun-cse-after-loop` の両方が有効になります。
`-funroll-all-loops`	ループ展開の最適化を実行します。これは全てのループに対して実行され、通常はプログラムの実行速度が低下します。`-funroll-all-loops` を指定すると、`-fstrength-reduce` と `-frerun-cse-after-loop` が有効になります。
`-fuse-caller-save`	コンパイラは呼び出し元退避レジスタモデルを適用します。処理間の最適化と組み合わせる事で、より効率的なコードを生成できます。

-fflag の形態のオプションは、デバイスに非依存のフラグを指定します。大部分のフラグには肯定形と否定形の両方があります。-ffoo の否定形は -fno-foo です。以下の表には、既定値ではない方だけを記載しています。

表 5-13: デバイスに依存しない最適化オプション

オプション	定義
-fforce-mem	メモリオペランドに対して算術演算を実行する前に、それらをレジスタにコピーします。これは、全てのメモリ参照を潜在的な共通部分式にする事により、生成されるコードを改善します。メモリ参照が共通部分式ではない場合、命令の組み合わせによって個別にレジスタに書き込む事を防ぐ必要があります。-O2 オプションは、このオプションを有効にします。
-finline-functions	全ての単純な関数を、それらの呼び出し元にインライン展開します。コンパイラは、そのようにインライン展開する価値のある十分に単純な関数を、発見的方法を使って決定します。static 宣言された関数に対する呼び出しが全て呼び出し元にインライン展開された場合、通常その関数自体はアセンブラコードとして出力されません。
-finline-limit=n	既定値では、コンパイラはインライン展開可能な関数のサイズを制限します。このフラグにより、inline キーワードによって明示的にインライン展開するよう指定されている関数に対するサイズの制限を制御できます。 *n* は、インライン展開可能な関数のサイズを擬似的な命令数 (パラメータ処理部を含まず) で指定します。n の既定値は 10000 です。この値を大きくするとインライン展開されるコードの量は増えますが、コンパイル時間とメモリ消費量が増加します。この値を小さくするとインライン展開されるコードの量が減り、コンパイル時間は短縮されますが、多くの場合プログラムの実行速度は低下します。このオプションは、主にインライン展開を多用するプログラムで役立ちます。 **Note:** 上記において、「擬似的な命令数」は関数のサイズを表す抽象的な尺度です。この値はアセンブリ命令の数を表す物では全くなく、その厳密な意味はコンパイラのリリースごとに変化する可能性があります。
-fkeep-inline-functions	static 宣言された関数に対する呼び出しが全てインライン展開された場合でも、実行時に他から呼び出し可能な独立した関数を出力します。このスイッチは extern 宣言されたインライン関数には影響しません。
-fkeep-static-consts	最適化を実行しない場合、static const 宣言された変数を、たとえその変数が参照されていなくても出力します。コンパイラは、既定値でこのオプションを有効にします。変数が参照されているかどうかをコンパイラにチェックさせる場合、最適化を実行するかどうかに関係なく、-fno-keep-static-consts オプションを使う必要があります。
-fno-function-cse	関数アドレスをレジスタに格納しません。常に同じ関数を呼び出す各命令に、関数アドレスを明示的に持たせます。このオプションを使うとコードの効率は低下します。しかし、このオプションを使わずに最適化を実行した場合、アセンブラ出力を後で変更するといった特殊な手順を踏む場合に混乱を招きます。

 © 2015 Microchip Technology Inc.

表 5-13: デバイスに依存しない最適化オプション (続き)

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、コンパイラに関数のインライン展開を一切させないようにするために使います。最適化を実行しなければ、関数は一切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に対して、フレームポインタをレジスタ内で保持しません。これにより、フレームポインタを退避/設定/復元するための命令を生成しなくて済みます。また、多くの関数では、余ったレジスタを他に利用できるようになります。
-foptimize-sibling-calls	末尾再帰呼び出し (sibling and tail recursive call) を最適化します。

5.9.8 プリプロセッサを制御するためのオプション

以下のオプションは、コンパイラのプリプロセッサを制御します。

表 5-14: プリプロセッサ オプション

オプション	定義
-C	コメントを破棄しないようプリプロセッサに指示します。このオプションは -E オプションと一緒に使います。
-dD	マクロ定義を削除せずに、それらを正しい順番で出力に渡すようプリプロセッサに指示します。
-Dmacro	macro で指定したマクロを文字列「1」として定義します。
-Dmacro=defn	macro で指定したマクロを defn として定義します。コマンドライン上の全ての -D オプションは、どの -U オプションよりも前に処理されます。
-dM	前処理の最後に、有効なマクロ定義のリストだけを出力するようプリプロセッサに指示します。このオプションは -E オプションと一緒に使います。
-dN	-dD と似ていますが、マクロの引数と内容が省略されるという点で異なります。出力には #define name の情報だけが含まれます。
-fno-show-column	診断結果に列番号を出力しません。このオプションは、列番号を認識しないプログラム(DejaGnu等)を使って診断結果を調べる場合に必要です。
-H	通常の動作に加えて、使われている各ヘッダファイルの名前を出力します。
-I-	-I- オプションの前で -I オプションを使って指定した全てのディレクトリは、#include "file" に対してのみ検索され、#include <file>に対しては検索されません。-I-の後で-Iオプションを使って追加したディレクトリは、全ての #include ディレクティブに対して検索されます (通常、全ての -I ディレクトリはこの方式で指定します)。加えて、-I- オプションを指定すると、現在作業中のディレクトリ (現在使用中の入力ファイルが保存されているディレクトリ) は #include "file" に対する最初の検索ディレクトリではなくなります。-I- のこの効果を上書きする方法はありません。-I. を使うと、コンパイラの起動時に「現在使用中」であったディレクトリを検索するよう指定できます。これは、プリプロセッサの既定値動作と厳密に同じではありませんが、多くの場合これで満足な結果が得られます。-I- は、ヘッダファイルに対する標準システム ディレクトリの使用を禁止しません。従って、-I- と -nostdinc の効果は互いに独立しています。 **NOTE:** プロジェクト プロパティでは、MPLAB XC32 のシステム インクルード ディレクトリ (例 : /pic32mx/include/) を指定しない事が必要です。xc32-gcc および xc32-g++ コンパイル ドライバは既定値の C libc または C++ libc と、それらに対応するインクルード ファイル ディレクトリを自動的に選択します。システム インクルード ファイルのパスを手動で追加すると、この自動処理が阻害され、不適切な libc インクルード ファイルがプロジェクト向けにコンパイルされる恐れがあります。その場合、インクルード ファイルとライブラリの間で不整合が生じます。プロジェクト プロパティへのシステム インクルードパスの追加は、以前から非推奨の行為である事に注意してください。

表 5-14: プリプロセッサ オプション (続き)

オプション	定義
-Idir	ヘッダファイルの検索ディレクトリのリストの先頭に、dir で指定されたディレクトリを追加します。これらのディレクトリは、システム ヘッダファイル ディレクトリの前に検索されるため、このオプションを使うとシステム ヘッダファイルを上書き (ユーザ独自のバージョンと置換) できます。-I オプションを複数回使った場合、ディレクトリは指定した順番 (コマンドラインの左から右) に検索されます。標準のシステム ディレクトリは、それらのディレクトリの後で検索されます。
-idirafter dir	dir で指定されたディレクトリを副インクルード パスに追加します。主インクルード パス (-I オプションはここに追加する) のどのディレクトリでもヘッダファイルが見付からなかった場合、副インクルード パスのディレクトリが検索されます。
-imacros file	通常の入力ファイルを処理する前に、file で指定されたファイルを入力として処理し、その出力は破棄します。file から生成された出力は破棄されるため、-imacros *file* は、file 内で定義されているマクロを主入力で使えるようにする働きしかしません。コマンドライン上の -D および -U オプションは、指定順に関係なく、必ず -imacros file の前に処理されます。全ての -include および -imacros オプションは、指定さ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、file で指定されたファイルを入力として処理します。これにより、file の内容が最初にコンパイルされます。コマンドライン上の -D および -U オプションは、指定順に関係なく、必ず -include file の前に処理されます。 全ての -include および -imacros オプションは、指定された順番に処理されます。
-M	makeに適した規則(各オブジェクト ファイルの依存関係を記述)を出力するよう、プリプロセッサに指示します。プリプロセッサは、各ソースファイルに対して 1 つの make 規則を出力します。この規則のターゲットはそのソースファイルに対応するオブジェクト ファイルの名前であり、そのソースファイルの依存性は全てそのソースファイルが使う #include ヘッダファイルに基づきます。この規則は 1 行で書く事もできますが、長い場合は \+ 改行を使って複数行に分ける事もできます。規則のリストは、前処理済み C プログラム内に出力されるのではなく、標準出力に表示されます。-M を指定すると、暗黙的に -E が有効になります (**5.9.2「出力の種類を制御するためのオプション」**参照)。
-MD	-M に似ていますが、依存情報はファイルに書き込まれ、コンパイルは継続されるという点で異なります。依存情報が書き込まれるファイルには、対応するソースファイル名の拡張子を .d に変更した名前が付けられます。
-MF file	このオプションを -M または -MM と一緒に使う事で、依存情報の書き込み先ファイルを指定します。-MF スイッチを指定しない場合、プリプロセッサは前処理出力と同じ場所に規則を出力します。-MF をドライバ オプション -MD または -MMD と一緒に使った場合、既定値の依存情報出力ファイルは上書きされます。
-MG	見付からないヘッダファイルは生成されるファイルとして扱い、それらはソースファイルと同じディレクトリ内にあると想定します。-MG を指定する場合、-M または -MM も指定する必要があります。-MG と一緒に -MD または -MMD を指定する事はできません。
-MM	-Mと似ていますが、出力には#include "file"によってインクルードされたユーザ ヘッダファイルだけが記述されます。#include <file> によってインクルードされたシステム ヘッダファイルは省略されます。
-MMD	-MD と似ていますが、ユーザ ヘッダファイルだけを記述し、システム ヘッダファイルは省略するという点で異なります。

© 2015 Microchip Technology Inc.

表 5-14: プリプロセッサ オプション (続き)

オプション	定義
-MP	このオプションは、メインファイル以外の各依存情報にダミーのターゲットを追加するよう CPP に指示する事で、それらが何にも依存しないようにします。これらのダミー規則は、ヘッダファイルを削除したのに make ファイルをそれに適合するよう更新しなかった場合の make に起因するエラーに対処します。以下は代表的な出力です。 test.o: test.c test.h test.h:
-MQ	-MT と似ていますが、make 専用の文字を引用するという点で異なります。-MQ '$(objpfx)foo.o' は $$(objpfx)foo.o: foo.c を与えます。既定値のターゲットは、それが -MQ によって与えられたかのように、自動的に引用されます。
-MT target	依存情報の生成によって出力される規則のターゲットを変更します。既定値では、CPP はメイン入力ファイルの名前 (パスを含む) から .c 等の拡張子を削除し、プラットフォームの通常のオブジェクト拡張子を追加します。結果はターゲットです。-MT オプションは、ターゲットをユーザが指定した通りの文字列に設定します。複数のターゲットを指定する場合、1 つの -MT に複数個の引数を指定する事も、-MT を複数回使って 1 つずつ指定する事もできます。 -MT '$(objpfx)foo.o'は$(objpfx)foo.o: foo.cを与えます。
-nostdinc	ヘッダファイルを標準システム ディレクトリで検索しません。-I オプションによって指定されたディレクトリ (および、適切であれば現在作業中のディレクトリ) だけを検索します。-I オプションの詳細は **5.9.11「ディレクトリを検索するためのオプション」**を参照してください。-nostdinc と -I- の両方を使う事で、インクルード ファイルの検索パスを明示的に指定したディレクトリだけに限定できます。
-P	#line ディレクティブを生成しないようプリプロセッサに指示します。このオプションは -E オプションと一緒に使います (**5.9.2「出力の種類を制御するためのオプション」**参照)。
-trigraphs	ANSI C の 3 文字表記をサポートします。-ansi オプションも同じ効果を持ちます。
-Umacro	macro で指定したマクロを定義しません。-U オプションは、全ての -D オプションの後で評価され、その後に -include および -imacros オプションが評価されます。
-undef	非標準のマクロ (アーキテクチャ フラグを含む) を事前定義しません。

5.9.9 アセンブリングのためのオプション

以下のオプションはアセンブラの動作を制御します。

表 5-15: アセンブリ オプション

オプション	定義
-Wa,option	option で指定したオプションをアセンブラに渡します。option は、コンマで区切る事で複数のオプションを指定できます。

5.9.10 リンクのためのオプション

-c、-S、-E のいずれかを指定した場合、リンカは実行されません。また、オブジェクト ファイル名を引数として使わない事が必要です。

表 5-16: リンク オプション

オプション	定義
-fill=<options	リンカにメモリフィル オプションを渡します。
-Ldir	コマンドライン オプション -l によって指定されたライブラリの検索ディレクトリのリストに、dir で指定されたディレクトリを追加します。
-llibrary	リンク時に、library で指定された名前を持つライブラリを検索します。リンカは標準のディレクトリ　リストを使ってライブラリ (「liblibrary.a」という名前を持つファイル) を検索します。リンカは、このファイルをフルネームで指定されたかのように扱います。このオプションは、コマンドラインでの指定順に影響されます。リンカは、ライブラリおよびオブジェクト ファイルを、それらが指定された順番に処理します。従って、foo.o -lz bar.o と指定した場合、最初に foo.o、次にライブラリ libz.a、最後に bar.o が検索されます。bar.o が libz.a 内の関数を参照している場合、それらの関数はロードされません。標準のシステム ディレクトリに加えて、-L で指定されたディレクトリが検索されます。通常、この方法で見付かるのはライブラリ ファイル (オブジェクト ファイルをメンバーとするアーカイブ ファイル) です。リンカはアーカイブ ファイルをスキャンし、参照されているが未定義であるシンボルを定義しているメンバーを探します。しかし、見付かったファイルが通常のオブジェクト ファイルである場合、そのファイルは通常の方法でリンクされます。-l オプション (例 : -lmylib) とファイル名の直接指定 (例 : libmylib.a) の唯一の違いは、-l は指定された複数のディレクトリを検索するという事です。既定値では、リンカは -l オプションで指定されたライブラリを以下のパスで検索します。 <install-path>\lib コンパイラを既定値のパスにインストールした場合のパスは以下の通りです。 Program Files\Microchip\mplab32\<version>\lib これは環境変数を使って変更できます。INPUT および OPTIONAL リンカスクリプト ディレクティブも参照してください。
-mips16	ライブラリの MIPS16 ISA バージョンをリンクします。
-mmicromips	ライブラリの microMIPS 圧縮 ISA バージョンと起動コードをリンクします。起動コードの ISA は BOOTISA コンフィグレーション ビットの設定に一致している必要があります。従って、microMIPS 起動コードをリンクするための -mmicromips リンクオプションを使う場合、ソースコード内で #pragma config BOOTISA=MICROMIPS を指定する必要があります。
-nodefaultlibs	リンク時に標準のシステム ライブラリを使いません。ユーザが指定したライブラリだけをリンカに渡します。コンパイラは memcmp、memset、memcpy に対する呼び出しを生成する場合があります。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によって解決されます。このオプションを指定した場合、別の何らかの機構によってこれらのエントリポイント提供する必要があります。
-nostdlib	リンク時に標準のシステム起動ファイルまたはライブラリを使いません。起動ファイルは含めず、ユーザが指定したライブラリだけをリンカに渡します。コンパイラは memcmp、memset、memcpy に対する呼び出しを生成する場合があります。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によって解決されます。このオプションを指定した場合、何らかの別の機構によってこれらのエントリポイントを提供する必要があります。
-s	実行ファイルから全てのシンボルテーブルと再配置情報を削除します。

© 2015 Microchip Technology Inc.

表 5-16: リンク オプション (続き)

オプション	定義
-u symbol	symbol が未定義であるかのようにふるまう事で、ライブラリ モジュールを強制的にリンクさせてシンボルを定義します。symbol の異なる複数の -u オプションを使って複数のライブラリ モジュールをロードさせる事ができます。
-Wl,option	option で指定したオプションをリンカに渡します。option は、コンマで区切る事で複数のオプションを指定できます。
-Xlinker option	option で指定したオプションをリンカに渡します。これを使うと、コンパイラが認識しないシステム固有のリンカオプションを指定できます。

5.9.11 ディレクトリを検索するためのオプション

以下のオプションは、ファイルを検索するディレクトリをコンパイラに対して指定します。

表 5-17: ディレクトリ検索オプション

オプション	定義
-Bprefix	このオプションは、コンパイラ自身の実行ファイル、ライブラリ、インクルード ファイル、データファイルが置かれている場所を指定します。コンパイラドライバ プログラムは、1 つまたは複数のサブプログラム (xc32-cpp、xc32-as、xc32-ld) を実行します。prefix は、実行を試みる各プログラムの接頭辞を指定します。コンパイラは最初に -B オプションで指定された接頭辞を使って、実行すべき各サブプログラムを探します。見付からない場合、ドライバは現在の PATH環境変数に基づいてサブプログラムを探します。コンパイラは、これらのオプションをリンカ用の -L オプションに反映するため、ディレクトリ名を指定する -B 接頭辞は、リンカにおいてライブラリにも適用されます。コンパイラはこれらのオプションをプリプロセッサ用の -isystem オプションに反映するため、それらはプリプロセッサにおけるインクルード ファイルにも適用されます。この場合、コンパイラは接頭辞に include を追加します。
-specs=file	コンパイラは標準の specs ファイルを読み込んだ後に、指定されたファイル(file)を処理します。これにより、どのスイッチをxc32-as または xc32-ld 等に渡すべきか決定する際に、xc32-gcc ドライバ プログラムが使う既定値を上書きします。-specs=file はコマンドラインで複数回指定でき、それらは指定された順番に (コマンドラインの左から右へ) 処理されます。

5.9.12 コード生成規則のオプション

-fflag の形態のオプションは、デバイスに依存しないフラグを指定します。大部分のフラグには肯定形と否定形の両方があります。-ffoo の否定形は -fno-foo です。以下の表には、既定値ではない方だけを記載しています。

表 5-18: コード生成規則オプション

オプション	定義
-fargument-alias -fargument-noalias -fargument-noalias-global	パラメータ同士およびパラメータとグローバルデータ間の関係を指定します。-fargument-alias を指定した場合、引数 (パラメータ) は他の引数とグローバル 記憶域のエイリアスになる事ができます。-fargument-noalias を指定した場合、引数は他の引数のエイリアスにはなれませんが、グローバル記憶域のエイリアスになる事ができます。-fargument-noalias-global を指定した場合、引数は他の引数のエイリアスにもグローバル記憶域のエイリアスにもなる事はできません。各言語は、規格が要求するオプションを自動的に選択します。ユーザがこのオプションを指定する必要はありません。
-fcall-saved-reg	regで指定された名前を持つレジスタを、関数によって退避される割り当て可能レジスタとして扱います。このレジスタは、1 つの関数呼び出しを越えて保持される一時的オブジェクトまたは変数に対しても割り当てる事ができます。この方法でコンパイルされた関数は、reg で指定されたレジスタを使用時に退避 / 復元します。このフラグと一緒にフレームポインタまたはスタックポインタを使うとエラーが発生します。デバイスの実行モデル内で固定的かつ広範な役割を持つ他のレジスタに対してこのフラグを使うと、破滅的な結果を招きます。関数の値が返されるレジスタに対してこのフラグを使うと、前記とは異なる種類の破滅的な結果が生じます。このフラグは、全てのモジュールで一貫して使う必要があります。
-fcall-used-reg	regで指定された名前を持つレジスタを、関数呼び出しによって上書きされない割り当て可能レジスタとして扱います。このレジスタは、1 つの関数呼び出しを越えて保持されない一時的オブジェクトまたは変数に対して割り当てる事ができます。この方法でコンパイルされた関数は、レジスタ reg を退避 / 復元しません。このフラグと一緒にフレームポインタまたはスタックポインタを使うとエラーが発生します。デバイスの実行モデル内で固定的かつ広範な役割を持つ他のレジスタに対してこのフラグを使うと、破滅的な結果を招きます。このフラグは、全てのモジュールで一貫して使う必要があります。
-ffixed-reg	reg で指定された名前を持つレジスタを固定されたレジスタとして扱います。生成されたコードは、このレジスタを (おそらくスタック ポインタ、フレーム ポインタ、その他の固定的な役割を持つレジスタとして以外には) 決して参照すべきではありません。 reg はレジスタの名前である事が必要です (例 : -ffixed-$0)。
-fno-ident	#ident ディレクティブを無視します。
-fpack-struct	全ての構造体メンバーを、間隔を空ける事なく格納します。このオプションを使うと、生成されるコードは最適な物ではなくなり、構造体メンバーのオフセットはシステム ライブラリと一致しなくなるので、通常はこのオプションが必要になる事はありません。
-fpcc-struct-return	サイズの小さい struct および union 型の値を、レジスタに返すのではなく、よりサイズの大きな値と同様にメモリ内に返します。これは効率的ではありませんが、利点として、32 ビットでコンパイルされたファイルと他のコンパイラでコンパイルされたファイルの間での互換性が得られます。「サイズの小さい」とは、構造体 /共用体のサイズとアラインメントが整数型と一致するという意味です。

 © 2015 Microchip Technology Inc.

表 5-18: コード生成規則オプション (続き)

オプション	定義
-fno-short-double	既定値では、コンパイラは double 型を float 型と等価として扱います。このオプションは、double 型を long double 型と等価にします。複数のモジュールが double 型のデータを引数を介して直接共有するかバッファ空間を介して間接的に共有している場合、それらのモジュール間でこのオプションが一致しないと、予期せぬ結果が生じます。ライブラリは、どちらかのスイッチ設定を持つプロダクト関数と一緒に提供されます。
-fshort-enums	enum 型に対して、その宣言において示された可能な値の範囲にとってちょうど必要となるだけのバイト数を割り当てます。これにより、enum 型は、十分な領域を持つ整数型のうち最もサイズの小さな物と等しくなります。
-fverbose-asm -fno-verbose-asm	生成するアセンブリコードにコメントを追加して読みやすくします。既定値 (-fno-verbose-asm) では、コメントは追加されません。これは 2 つのアセンブリ ファイルを比較する場合に便利です。
-fvolatile	ポインタを介するメモリ参照は全て volatile 指定されていると見なします。
-fvolatile-global	外部およびグローバル データアイテムに対するメモリ参照は全て volatile指定されていると見なします。このスイッチは静的データに影響しません。
-fvolatile-static	静的データに対するメモリ参照は全て volatile 指定されていると見なします。

NOTE:

© 2015 Microchip Technology Inc.

第 6 章　ANSI C 規格の問題

6.1　はじめに

本コンパイラは ANS X3.159-1989 プログラミング言語規格に準拠します。これは一般的に C89 規格と呼ばれます。本書では、C89 規格を ANSI C 規格と呼びます。本コンパイラは、より新しい C99 規格の一部の機能もサポートします。

- ANSI C 規格からの逸脱
- ANSI C 規格に対する拡張
- 処理系定義のふるまい

6.2　ANSI C 規格からの逸脱

ANSI C 規格からの逸脱はありません。

6.3　ANSI C 規格に対する拡張

MPLAB XC32 C/C++ コンパイラの C/C++ コードはキーワード、命令文、式の領域で ANSI C 規格から異なります。

6.3.1　キーワードの相違

新しいキーワードはベース GCC 処理系の一部です。以下の参照先の説明は標準 GCC 文書に基づき、GCC の 32 ビット コンパイラパートの特定の構文と意味に合わせて調整されています。

- **8.12「変数属性」**
- **12.2「関数属性と指定子」**
- **12.9「関数のインライン展開」**
- **8.12「変数属性」** - 指定レジスタ内の変数
- **8.8「複素数データ型」**
- **8.10「標準型修飾子」** - `typeof` による型への参照

6.3.2　命令文の相違

命令文の相違はベース GCC 処理系の一部です。以下の参照先の説明は標準 GCC 文書に基づき、GCC の 32 ビット コンパイラパートの特定の構文と意味に合わせて調整されています。

- **10.4「値としてのラベル」**
- **10.5「条件演算子のオペランド」** - オペランドを省略した条件式
- **10.6「case のレンジ指定」**

6.3.3　式の相違

- **8.9「定数の型と書式」** - バイナリ定数

6.4 処理系定義のふるまい

ANSI C 規格の一部の機能は処理系定義のふるまいを持ちます。これは、一部の C コードの厳密な挙動はコンパイラによって異なる可能性があるという事を意味します。MPLAB XC32 C/C++ コンパイラの厳密な挙動については本書全体を通して詳細に説明しており、**補遺 B.「処理系定義のふるまい」**に全てをまとめて記載しています。

© 2015 Microchip Technology Inc.

第 7 章　デバイスに固有の機能

7.1 はじめに

MPLAB XC32 C/C++ コンパイラは、ROM ベース アプリケーションの作成を容易にする各種の特殊な機能と C/C++ 言語に対する拡張をサポートします。本章では、これらのデバイスに固有の特殊な言語機能について説明します。

- サポートするデバイス
- デバイス ヘッダファイル
- スタック
- ID の格納位置
- C コードから SFR を使う

7.2 サポートするデバイス

MPLAB XC32 C/C++ コンパイラは、全ての PIC32 デバイスのサポートを目標としています。しかし、これらのファミリには新しいデバイスが頻繁にリリースされます。

7.3 デバイス ヘッダファイル

ユーザの各ソースファイルには、ヘッダファイル `<xc.h>` をインクルードする事を推奨します。このファイルは、プロジェクトのビルド時に他のデバイス固有ヘッダファイルをインクルードするための汎用ヘッダファイルです。

このファイルをインクルードする事で、専用の変数を介して SFR にアクセスできるようになります。また、このファイル内の `#define` により、アセンブリ言語ファイル内から通常のレジスタ名が使えるようになります。

7.3.1 CP0 レジスタ定義ヘッダファイル

CP0 レジスタ定義ヘッダファイル (`cp0defs.h`) は、CP0 レジスタとそれらのフィールドに関する定義と、CP0 レジスタにアクセスするためのマクロを含みます。

CP0 レジスタ定義ヘッダファイルは、コンパイラのインストール ディレクトリ構造の中の `pic32mx/include` ディレクトリ内にあります。汎用デバイス ヘッダファイル `xc.h` をインクルードすると、CP0 レジスタ定義ヘッダファイルは自動的にインクルードされます。

CP0 レジスタ定義ヘッダファイルは、アセンブリおよび C/C++ ファイルのどちらとでも動作するよう設計されています。CP0 レジスタ定義ヘッダファイルは、プロセッサ汎用ヘッダファイル内で定義されるマクロに依存します。

© 2015 Microchip Technology Inc.

7.4 スタック

本書では、PIC32 が使うスタックを「ソフトウェア スタック」と呼びます。これは、ほとんどのコンピュータが採用している標準的なスタック編成であり、データメモリにはプッシュ / ポップ型命令とスタックポインタ レジスタによってアクセスします。「ハードウェア スタック」という用語は、Microchip 社製 8 ビットデバイスが採用しているスタックを指す場合に使います。これは、関数のリターンアドレスを保存するためにだけ使います。

PIC32 は、専用のスタックポインタ レジスタ `sp` (レジスタ 29) をソフトウェア スタックポインタとして使います。関数呼び出し、割り込み、例外処理を含む全てのプロセッサ スタック動作にはソフトウェア スタックを使います。これは、スタック上で次に空いている位置を指します。スタックは、メモリアドレスの低い方へ向かって進みます。

既定値によるスタックのサイズは 1024 バイトです。スタックのサイズは、リンカ向けコマンドライン オプション `--defsym_min_stack_size` でサイズを指定する事により変更できます。コマンドラインを使って 2048 バイトのスタックを割り当てる場合の例を以下に示します。

```
xc32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

実行時にスタックはアドレスの高い方から低い方へ向かって進みます。スタックの管理には、以下の 2 つのワーキング レジスタを使います。

- レジスタ 29 (`sp`) - これはスタックポインタであり、スタック上で次に空いている位置を指します。
- レジスタ30 (`sp`) - これはフレームポインタであり、現在の関数のフレームを指します。

スタック オーバーフローの検出機能は提供されません。

C/C++ スタートアップ モジュールは、起動および初期化シーケンス中にスタックポインタを初期化します (**14.3.2「スタックポインタとヒープの初期化」** 参照)。

7.4.1 コンフィグレーション ビットへのアクセス

PIC32 は、コンフィグレーション ビットまたはヒューズを特定のアドレスに格納します。これらのビットは、基本的なデバイスの動作 (オシレータモード、ウォッチドッグ タイマ、プログラミング モード、コード保護作等) を指定します。これらのビットを正しく設定しないと、コードエラーが発生したり、デバイスが動作しなくなる可能性があります。

`#pragma config` ディレクティブは、アプリケーションが使うプロセッサに固有のコンフィグレーション設定 (コンフィグレーション ビット) を指定します。詳細は、「PIC32 Configuration Settings」オンラインヘルプ (*MPLAB X IDE>Help>Help Contents>XC32 Toolchain* から開く) を参照してください。コマンドラインからコンパイラを使う場合、このヘルプファイルは以下の既定値パスで開く事ができます。

```
Program Files/Microchip/ <install-dir>/<version>/docs/PIC32ConfigSet.html
```

コンフィグレーション設定は、複数の `#pragma config` ディレクティブを使って指定します。コンパイラは、指定されたコンフィグレーション設定がターゲット デバイスに対して有効かどうか確認します。コンフィグレーション ワード内の特定の設定がどの `#pragma config` でも指定されていない場合、その設定に対応するビットは既定値 (未プログラム状態の値) に設定されます。コンフィグレーション設定は、単一のトランザクション ユニット (関連する全てのヘッダファイルをインクルードした前処理後の 1 つの C/C++ ファイル) 内で指定する必要があります。

`#pragma config` ディレクティブによって設定を指定された各コンフィグレーション ワードに対して、コンパイラは `.config_`*`address`* という名前の読み出し専用データセクションを生成します (*`address`* はコンフィグレーション ワードの 16 進数表記アドレス)。例えば、アドレス `0xBFC02FFC` に置かれたコンフィグレーション ワードに対してコンフィグレーション設定を指定した場合、`.config_BFC02FFC` という名前の読み出し専用データセクションが生成されます。

- 構文
- 例

© 2015 Microchip Technology Inc.

7.4.1.1 構文

以下に、プラグマが取れる各種形態の構文 (メタ表記) を示します。

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

setting-name と *value-name* はデバイスに固有であり、PIC32ConfigSet.html で調べる事ができます (この文書はインストール ディレクトリ内の「docs」フォルダ内にあります)。

全ての #pragma config ディレクティブは、実行コードを定義しないため、関数定義の外に置く必要があります。

7.4.1.2 例

以下に #pragma config ディレクティブの使い方の例を示します。この例は以下を実行します。

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

```
#pragma config FWDTEN = ON, WDTPS = PS128
#pragma config POSCMOD = HS
...
int main (void)
{
...
}
```

7.5 ID の格納位置

ユーザ定義 ID の格納位置は、1 つのコンフィグレーション ワードに実装されます。これらの格納位置は、#pragma config ディレクティブを使って設定する必要があります。**7.4.1「コンフィグレーション ビットへのアクセス」**を参照してください。

例 :#pragma config USERID=0x1234.

7.6 C コードから SFR を使う

特殊機能レジスタ (SFR) は、MCU の動作またはデバイス上の周辺モジュールの動作を制御するためのレジスタです。これらのレジスタはメモリに配置されます (つまり、デバイス メモリマップ上の特定のアドレスに割り当てられます)。一部のレジスタは複数のビットフィールドを格納し、それらを使って別々の機能を制御します。

メモリに配置された SFR には、そのレジスタのアドレスに割り当てられた専用の C 変数を使ってアクセスします (専用の属性を使用)。これらの変数には通常の C 変数と同様にアクセスできるため、SFR へのアクセスのために特別な構文を使う必要はありません。

SFR 変数は対応するヘッダファイル内であらかじめ定義されており、`<xc.h>` ヘッダファイル (**7.3「デバイス ヘッダファイル」**参照) をソースコードにインクルードする事でアクセス可能になります。これらのヘッダファイルでは、SFR 内のビットにアクセスするための構造体も定義されます。

SFR およびその中のビット変数 (ビットフィールド) に割り当てられる C 変数の名前は、そのデバイスのデータシート内で指定されている名前に基づきます。通常、ビットフィールドを保持する構造体には、対応するレジスタの名前の後に `bits` を追加した名前が付けられます。例えば下のコードは、汎用ヘッダファイルをインクルードし、PORTB レジスタ全体をクリアした後に、構造体 / ビットフィールド定義を使って PORTB の bit 2 をセットします。

Note:	シンボル `PORTB` と `PORTBbits` は同じレジスタ (従って同じアドレス) を参照します。一方のレジスタに書き込むと、両方の保持値が変更されます。

```
#include <xc.h>
int main(void)
{
  PORTBCLR = 0xFFFFu;
  PORTBbits.RB2 = 1;
  PORTBSET= _PORTB_RB2_MASK;
}
```

アセンブリ言語を使う場合、`PORTB`レジスタは`.extern PORTB`として宣言されます。

使用するデバイスにおけるこれらの名前は、そのデバイスに固有のヘッダファイル (各変数の定義のために `<xc.h>` によってインクルードされるヘッダファイル) を調べる事で確認できます。これらのファイルはコンパイラの `pic32mx/include/proc` ディレクトリ内にあり、デバイスを表す名前が付けられています。`<xc.h>` によってインクルードされるヘッダファイルの名前とデバイス名は一対一に対応します。例えば PIC32MX360F512L 向けにコンパイルする場合、`<xc.h>` ヘッダファイルは `<proc/p32mx360f512l.h>` をインクルードします。このデバイス固有ファイルは、`<xc.h>` によって自動的にソースコードにインクルードされるため、ユーザが個別にインクルードする必要はありません。

PIC32 の一部の SFR には、その SFR 内のビットを 1 動作でセット / クリア / トグルするためのレジスタが割り当てられています。例えば `PORTB` SFR には、書き込み専用の `PORTBSET`、`PORTBCLR`、`PORTBINV` レジスタが割り当てられています。これらのレジスタ内のビットに「`1`」を書き込むと、`PORTB` SFR 内の対応するビットが、それぞれセット / クリア / トグルされます。例えば以下のコードは `PORTB` の bit 1 をセットします。

```
PORTBSET = 0x2;
```

あるいは、以下のように、デバイス固有ヘッダファイル内で定義されているマクロを使う事もできます。

```
PORTBSET = _PORTB_RB1_MASK;
```


© 2015 Microchip Technology Inc.

7.6.1 CP0 レジスタの定義

CP0 レジスタ定義ヘッダファイルをアセンブリ ファイルからインクルードする場合、CP0 レジスタは以下のように定義されます。

```
#define _CP0_register_name $register_number, select_number
```

例えば IntCtl レジスタは以下のように定義されます。

```
#define _CP0_INTCTL $12, 1
```

CP0 レジスタ定義ヘッダファイルを C ファイルからインクルードする場合、CP0 レジスタと select_number は以下のように定義されます。

```
#define _CP0_register_name register_number
#define _CP0_register_name_SELECT select_number
```

例えば IntCtl レジスタは以下のように定義されます。

```
#define _CP0_INTCTL 12
#define _CP0_INTCTL_SELECT 1
```

7.6.2 CP0 レジスタ フィールドの定義

アセンブリまたは C/C++ ファイルから CP0 レジスタ定義ヘッダファイルをインクルードする場合、各 CP0 レジスタ フィールドに対して以下の 3 つの #defines が存在します。

_CP0_register_name_field_name_POSITION - 開始ビットの位置

_CP0_register_name_field_name_MASK - このフィールドに含まれるビットがセットされる

_CP0_register_name_field_name_LENGTH - このフィールドに含まれるビットの個数

例えば、IntCtl レジスタのベクタ間隔フィールドは以下の定義を持ちます。

```
#define _CP0_INTCTL_VS_POSITION 0x00000005
#define _CP0_INTCTL_VS_MASK     0x000003E0
#define _CP0_INTCTL_VS_LENGTH   0x00000005
```

7.6.3 CP0 アクセスマクロ

CP0 レジスタ定義ヘッダファイルを C ファイルからインクルードする場合、CP0 アクセスマクロが定義されます。各 CP0 レジスタは、以下に示す最大で 6 つの異なるアクセスマクロ定義を持つ事ができます。

_CP0_GET_register_name ()	レジスタ (register_name) の値を返します。
_CP0_SET_register_name (val)	レジスタ (register_name) の値を val に設定し、返り値は void です。これは書き込み可能フィールドを格納したレジスタに対してのみ定義されます。
_CP0_XCH_register_name (val)	レジスタ (register_name) の値を val に設定し、以前のレジスタ値を返します。これは書き込み可能フィールドを格納したレジスタに対してのみ定義されます。
_CP0_BIS_register_name (set)	レジスタ (register_name) を (reg |= set) に設定し、以前のレジスタ値を返します。これは書き込み可能フィールドを格納したレジスタに対してのみ定義されます。
_CP0_BIC_register_name (clr)	レジスタ (register_name) を (reg &= ~clr) に設定し、以前のレジスタ値を返します。これは書き込み可能フィールドを格納したレジスタに対してのみ定義されます。
_CP0_BCS_register_name (clr, set)	レジスタ(register_name)を(reg = (reg & ~clr) | set) に設定し、以前のレジスタ値を返します。これは書き込み可能フィールドを格納したレジスタに対してのみ定義されます。

7.6.4 アドレス変換マクロ

システムコードは、仮想アドレスと物理アドレスの間の変換と、カーネル セグメント アドレス同士の間の変換を必要とする場合があります。これらの変換を容易にするため、また、あるアドレスがどのセグメント内にあるのか特定するために、以下のマクロが提供されます。

KVA_TO_PA(v)	カーネル仮想アドレスを物理アドレスに変換します。
PA_TO_KVA0(pa)	物理アドレスを KSEG0 仮想アドレスに変換します。
PA_TO_KVA1(pa)	物理アドレスを KSEG1 仮想アドレスに変換します。
KVA0_TO_KVA1(v)	KSEG0 仮想アドレスを KSEG1 仮想アドレスに変換します。
KVA1_TO_KVA0(v)	KSEG1 仮想アドレスを KSEG0 仮想アドレスに変換します。
IS_KVA(v)	アドレスがカーネル セグメント仮想アドレスである場合に「1」と評価し、それ以外の場合に「0」と評価します。
IS_KVA0(v)	アドレスが KSEG0 仮想アドレスである場合に「1」と評価し、それ以外の場合に「0」と評価します。
IS_KVA1(v)	アドレスが KSEG1 仮想アドレスである場合に「1」と評価し、それ以外の場合に「0」と評価します。
IS_KVA01(v)	アドレスが KSEG0 または KSEG1 仮想アドレスである場合に「1」と評価し、それ以外の場合に「0」と評価します。

© 2015 Microchip Technology Inc.

第 8 章　サポートするデータ型と変数

8.1 はじめに

MPLAB XC32 C/C++ コンパイラは各種のデータ型と属性をサポートします。本章では、これらのデータ型と変数について説明します。メモリ内の変数の格納位置については、**第 9 章「メモリの割り当てとアクセス」**を参照してください。

- データの表現
- 整数データの型
- 浮動小数点データの型
- 構造体と共用体
- ポインタの型
- 複素数データ型
- 定数の型と書式
- 標準型修飾子
- コンパイラに固有の修飾子
- 変数属性

8.2 識別子

C/C++ 変数の識別子は一連の文字と数字で表され、アンダースコア「_」は文字として数えられます (以下は関数識別子にも適用されます)。識別子の先頭に数字を使う事はできません。アンダースコアは識別子の先頭に使えますが、そのような識別子はコンパイラ用に予約されているため､ユーザ プログラムの中で定義すべきではありません。これはアセンブリ ドメインの識別子には当てはまりません (アンダースコアで始まる識別子はよく使われる)。

識別子では大文字と小文字を区別するため、`main` と `Main` は同じではありません。

識別子内の全ての文字は有意ですが、31 文字を越える長さの識別子を使うと移植性が低下します。

8.3 データの表現

本コンパイラは、マルチバイト値をリトルエンディアン形式で格納します。つまり、最下位バイト (LSB) は最も低いアドレスに格納されます。

例えば 32 ビット値 `0x12345678` は、以下のようにアドレス `0x100` に格納されます。

アドレス	0x100	0x101	0x102	0x103
データ	0x78	0x56	0x34	0x12

8.4 整数データの型

コンパイラ内では、整数値は 2 の補数で表現され、サイズは 8 ～ 64 ビットです。以下の値は、limits.h によって、コンパイル済みコード内で利用可能となります。

型	ビット数	Min	Max
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、long、signed long	32	-2^{31}	$2^{31}-1$
unsigned int、unsigned long	32	0	$2^{32}-1$
long long、signed long long	64	-2^{63}	$2^{63}-1$
unsigned long long	64	0	$2^{64}-1$

8.4.1 符号付き / 符号なしの文字型

既知値では、通常の char 型は符号付き値です。C 規格によると、このふるまいは処理系定義であり、環境[1] によっては、通常の C/C++ char 型が符号なし値として扱われます。コマンドライン オプション -funsigned-char を使うと、与えられた変換ユニットの既定値型を unsigned に設定できます。

8.4.2 limits.h

limits.h ヘッダファイルは、整数型で表現可能な値のレンジを定義します。

マクロ名	値	概要
CHAR_BIT	8	ビットフィールドではない最小オブジェクトのサイズ (ビット数)
SCHAR_MIN	-128	signed char型オブジェクトの最小可能値
SCHAR_MAX	127	signed char型オブジェクトの最大可能値
UCHAR_MAX	255	unsigned char 型オブジェクトの最大可能値
CHAR_MIN	-128 (または 0、**8.4.1「符号付き / 符号なしの文字型」**参照)	char 型オブジェクトの最小可能値
CHAR_MAX	127 (または 255、**8.4.1「符号付き / 符号なしの文字型」**参照)	char 型オブジェクトの最大可能値
MB_LEN_MAX	16	任意のロケールでのマルチバイト文字の最大長
SHRT_MIN	-32768	short int 型オブジェクトの最小可能値
SHRT_MAX	32767	short int 型オブジェクトの最大可能値
USHRT_MAX	65535	unsigned short int 型オブジェクトの最大可能値

1. 代表例 : PowerPC、ARM

 © 2015 Microchip Technology Inc.

マクロ名	値	概要
INT_MIN	-2^{31}	int 型オブジェクトの最小可能値
INT_MAX	$2^{31}-1$	int 型オブジェクトの最大可能値
UINT_MAX	$2^{32}-1$	unsigned int 型オブジェクトの最大可能値
LONG_MIN	-2^{31}	long 型オブジェクトの最小可能値
LONG_MAX	$2^{31}-1$	long 型オブジェクトの最大可能値
ULONG_MAX	$2^{32}-1$	unsigned long 型オブジェクトの最大可能値
LLONG_MIN	-2^{63}	long long 型オブジェクトの最小可能値
LLONG_MAX	$2^{63}-1$	long long 型オブジェクトの最大可能値
ULLONG_MAX	$2^{64}-1$	unsigned long long 型オブジェクトの最大可能値

8.5 浮動小数点データの型

本コンパイラは IEEE-754 方式の浮動小数点フォーマットを採用しています。処理系の制限に関する情報は、float.h によって、変換ユニット内で利用可能となります。

型	ビット数
float	32
double	32
long double	64

変数の型は float、double、long double キーワードを使って宣言できます。浮動小数点型は常に signed として扱われ、浮動小数点型の指定時に unsigned キーワードを使う事は禁じられています。全ての浮動小数点値はリトルエンディアン形式で表現されます (最下位バイト (LSB) は最も低いアドレスに格納されます)。

このフォーマットを表 8-1 に示します。

- 値の正 / 負は符号ビットにより表されます。
- 32 ビット浮動小数点型の指数部は 8 ビットであり、127 のゲタを履かせたバイアス形式で保存されます (ゼロの指数は 127 として保存)。
- 64 ビット浮動小数点型の指数部は 11 ビットであり、1023 のゲタを履かせたバイアス形式で保存されます (ゼロの指数は 1023 として保存)。
- 仮数部 (mantissa) は小数点の右側 (小数点以下) を表します。小数点の左側には暗黙の 1 ビットが置かれます。この暗黙ビットの値は、浮動小数点値がゼロの場合は「0」、それ以外の場合は常に「1」です。ゼロ値は、ゼロの指数部によって表されます。

32 ビット浮動小数点型の値は下式で表せます。

$(-1)^{sign} \times 2^{(exponent-127)} \times 1.\text{mantissa}$

64 ビット浮動小数点型の値は下式で表せます。

$(-1)^{sign} \times 2^{(exponent-1023)} \times 1.\text{mantissa}$

IEEE 754 形式の 32 ビット フォーマットを表 8-1 に示します。仮数部の最上位ビットは暗黙ビット (小数点の左側のビット) です。このビットは、指数部がゼロ (浮動小数点値がゼロ) の場合に「0」、それ以外の場合に「1」であると見なされます。

表 8-1: IEEE 754 浮動小数点フォーマットの例

フォーマット	値	バイアスした指数部	1.[仮数部]	10 進数
32 ビット	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		(251)	(1.302447676659)	—

表 8-1 の例は、以下のように手計算できます。

符号ビットは「0」で、バイアスした指数部は 251 であるため、指数部の値は 251-127=124 です。仮数部 (小数点の右側) の 2 進値を 10 進値に変換し、2^{23} (23 は仮数部のビット数) で除算する事により、値 0.302447676659 が得られます。この小数値に1を加算します。以上の2つの値から、浮動小数点値は以下のように求まります。

$-1^{0} \times 2^{124} \times 1.302447676659$

上式から以下の値が得られます。

$1 \times 2.126764793256e+37 \times 1.302447676659$

この値は約 2.77000*e*+37 です。

© 2015 Microchip Technology Inc.

2 進表記の浮動小数点値は時に誤解を招きます。「有限サイズの浮動小数点型は全ての浮動小数点値を厳密に表現できるわけではない」という事に注意が必要です。浮動小数値の指数部のサイズ (ビット数) によって保持可能な値のレンジが決まり、仮数部のサイズ (ビット数) は厳密に表現可能な値の間隔に関係します。従って、64 ビット浮動小数点フォーマットの方が、より幅広いレンジの値をより精度よく表現できます。

例えば 32 ビットの浮動小数点型を使うと、95000.0 という値は厳密に表現できます。しかし、厳密に表現可能な次に大きな値は (約) 95000.00781 であり、2 つの値の中間の値は丸め処理されます (厳密に表現できません)。このため、C/C++ コードにおける浮動小数点値の比較では、期待通りの結果が得られない場合があります。以下に例を示します。

```
volatile float myFloat;
myFloat = 95000.006;
if(myFloat == 95000.007)     // value will be rounded
   LATA++;                  // this line will be executed!
```

この場合、比較した 2 つの値は異なるにも関わらず、`if()` 式の結果は「真」と評価されます。

浮動小数点型の特性をまとめて表 8-2 に示します。この表内の「シンボル」はプロセッサマクロを表し、これらはソースコードに `<float.h>` をインクルードする事で利用可能となります。`float` 型と `double` 型用に 2 セットのマクロが提供されます。シンボル内の *XXX* は、 `FLT` (float 型用) または `DBL` (double 型用) を表します。例えば `FLT_MAX` は `float` 型の最大値を表し、`DBL_MAX` は `double` 型の最大値を表します。ANSI 規格は浮動小数点データ型のサイズとフォーマットを完全には規定していません。このため、これらのマクロを使って、その処理系における浮動小数点型の保持値のレンジを確認する事により、コードの移植性を高める事ができます。

表 8-2: 浮動小数点型の値のレンジ

シンボル	意味	32 ビット値	64 ビット値
*XXX*_RADIX	指数表現の底	2	2
*XXX*_ROUNDS	加算の丸め処理モード	1	
*XXX*_MIN_EXP	FLT_RADIX^{n-1} が正規化 float 値となる最小の *n*	-125	-1021
*XXX*_MIN_10_EXP	10^{n}が正規化float値となる最小の*n*	-37	-307
*XXX*_MAX_EXP	FLT_RADIX^{n-1} が正規化 float 値となる最大の *n*	128	1024
*XXX*_MAX_10_EXP	10^{n}が正規化float値となる最大の*n*	38	308
*XXX*_MANT_DIG	FLT_RADIX を底とする仮数部の桁数	24	53
*XXX*_EPSILON	1.0 に加算した時に結果が 1.0 とならない最小の数	1.1920929e-07	2.2204460492503131e-16

8.6 構造体と共用体

MPLAB XC32 C/C++ コンパイラは struct 型と union 型をサポートします。構造体と共用体では、各メンバーに適用されるメモリオフセットだけが異なります。

これらの型は少なくとも 1 バイト幅を持ちます。ビットフィールドは完全にサポートされます。

構造体と共用体は、関数の引数および戻り値として自由に渡す事ができます。構造体と共用体を指すポインタは完全にサポートされます。

8.6.1 構造体および共用体の修飾子

MPLAB XC32 C/C++ コンパイラは、構造体に対する型修飾子の使用をサポートします。修飾子を構造体に対して適用すると、構造体の全てのメンバーがその修飾子を継承します。下の例では、構造体に対して const 修飾子を適用しています。

```
const struct {
        int number;
        int *ptr;
} record = { 0x55, &i };
```

この場合、構造体の全体がプログラムメモリ内に配置され、各メンバーは読み出し専用です。通常、構造体を const 修飾する場合、全てのメンバーを忘れずに初期化する必要があります (実行時に初期化できないため)。

構造体は修飾せずに、構造体のメンバーを個々に const 修飾した場合、構造体は RAM 内に配置されますが、各メンバーは読み出し専用です。構造体のメンバーを修飾する場合の例を以下に示します。

```
struct {
        const int number;
        int * const ptr;
} record = { 0x55, &i};
```

8.6.2 構造体内のビットフィールド

MPLAB XC32 C/C++ コンパイラは、構造体の中のビットフィールドを完全にサポートします。

ビットフィールドは、通常 unsigned int 型を使って定義されますが、常に 8 ビットの記憶域ユニットに割り当てられます。記憶域ユニットは 32 ビット境界に配置されます (これは packed 属性を使って変更可能です)。

最初に定義されたビットは、格納先のワードの最下位ビット (LSb) に格納されます。ビットフィールドが宣言された時、使用中の 8 ビット記憶域ユニット内にそのビットフィールドが収まる場合、ビットフィールドはそこに割り当てられます。ビットフィールドがそこに収まり切らない場合は、新しいバイトが構造体内に割り当てられます。ビットフィールドが 8 ビット記憶域ユニットの境界を越える事はできません。以下に例を示します。

```
struct {
        unsigned        lo :1;
        unsigned        dummy :6;
        unsigned        hi :1;
} foo;
```

上のように宣言された構造体には 1 バイトだけが割り当てられます。

© 2015 Microchip Technology Inc.

制御レジスタ内のアクティブなビットの間の未使用領域を埋めるために、無名のビットフィールドを宣言する事ができます。上の例で dummy が決して参照されない場合、以下のように宣言する事ができます。

```
struct {
        unsigned        lo :1;
        unsigned           :6;
        unsigned        hi :1;
} foo;
```

ビットフィールドを含む構造体は、コンマで区切った各フィールドの初期値のリストを与える事によって初期化できます。以下に例を示します。

```
struct {
        unsigned        lo  :1;
        unsigned        mid :6;
        unsigned        hi  :1;
} foo = {1, 8, 0};
```

無名ビットフィールドを含む構造体を初期化する場合、無名メンバーには初期値を与えません。以下に例を示します。

```
struct {
        unsigned        lo  :1;
        unsigned            :6;
        unsigned        hi  :1;
} foo = {1, 0};
```

この場合、メンバー lo および hi は正しく初期化されます。

MPLAB XC コンパイラは無名の共用体をサポートします。これらは識別子を持たない共用体であり、そのメンバーには共用体を参照せずにアクセスできます。これらの共用体は構造体の中で使えます。以下に例を示します。

```
struct {
       union {
       int x;
       double y;
    };
} aaa;

int main(void)
{
    aaa.x = 99;
    // ...}
```

この共用体は無名であり、そのメンバーは、それらが構造体の一部であるかのうようにしてアクセスされます。無名共用体はどの C 規格にも含まれていません。このため、無名共用体を使うと、コードの移植性が制限されます。

© 2015 Microchip Technology Inc.

8.7 ポインタの型

MPLAB XC32 C/C++ コンパイラは、2 種類の基本的ポインタ型 (データポインタと関数ポインタ) をサポートします。データポインタは、プログラムによって間接的に読み書き可能な変数のアドレスを保持します。関数ポインタは、ポインタを介して間接的に呼び出す事ができる実行可能関数のアドレスを保持します。

8.7.1 型修飾子とポインタの組み合わせ

ポインタ型の定義に関しては、最初に ANSI C/C++ 規格の規定を調べる事を推奨します。

ポインタは他の C/C++ オブジェクトと同様に修飾できますが、ポインタには 2 つの量が関係するため、修飾するには注意が必要です。2 つ量の 1 つはポインタ自身です。ポインタは通常の C/C++ 変数と同様に扱われ、ポインタにはメモリが予約されます。もう 1 つはターゲットです (複数の場合あり)。ターゲットはポインタの参照先 (ポインタが指す対象) です。ポインタ定義の一般的な形態は以下の通りです。

```
target_type_&_qualifiers * pointer's_qualifiers pointer's_name;
```

*の右側にある(つまり*pointer's_ name*の前の)修飾子は、ポインタ変数自体に関係します。* の左側にある型と全ての修飾子は、ポインタのターゲットに関係します。それはポインタを逆参照する * 演算子でもあるため、これは理に適っています。これにより、ポインタ変数からその現在のターゲットを取得できます。

volatile 修飾子を使ったポインタ定義の 3 つの例を以下に示します。定義内のフィールドは、スペースを挿入して見やすくしています。

```
volatile int *            vip ;
int          * volatile   ivp ;
volatile int * volatile   vivp ;
```

最初の例は、名前が vip のポインタです。これは volatile で修飾された int 型オブジェクトのアドレスを格納します。ポインタ自体 (アドレスを保持する変数) は volatile ではありません。しかし、ポインタが逆参照された時にアクセスされるオブジェクトは volatile として扱われます。言い換えると、このポインタを介してアクセス可能なターゲット オブジェクトは外部で変更される可能性があります。

2 番目の例は、名前が ivp のポインタです。これも int 型オブジェクトのアドレスを格納します。この例では、ポインタ自体が volatile であり、ポインタが格納するアドレスは外部で変更される可能性があります。しかし、ポインタが逆参照された時にアクセスされるオブジェクトは volatile ではありません。

最後の例は、名前が vivp のポインタです。ポインタ自体が volatile で修飾され、ポインタが指すオブジェクトも volatile です。

1 つのポインタに複数のオブジェクトのアドレスが割り当てられる可能性があるという事に注意が必要です。例えば、関数へのパラメータとして使われるポインタには、その関数が呼び出されるたびに異なるオブジェクト アドレスが割り当てられます。ポインタの定義は、割り当てられる全てのターゲット アドレスに対して有効である事が必要です。

> **Note:** ポインタに関する表現には注意が必要です。「const ポインタ」という表現は、const オブジェクトを指すポインタなのか、ポインタ自体が const なのか曖昧です。定義を明確にするために「const を指すポインタ」と「const ポインタ」を使い分ける事はできますが、これらの表現が普遍的に理解されるとは限りません。

© 2015 Microchip Technology Inc.

8.7.2 データポインタ

本コンパイラで使うポインタのサイズは全て 32 ビットです。これらが保持するアドレスを使って全てのメモリ位置にアクセスできます。

8.7.3 関数ポインタ

MPLAB XC コンパイラは、関数を指すポインタを完全にサポートします。これらを使う事により、関数を間接的に呼び出す事ができます。これらのポインタは、ユーザ定義 C/C++ 配列に格納された複数の関数アドレスの中の 1 つを呼び出す場合によく使われます (配列をルックアップ テーブルのように使用)。

関数ポインタのサイズは常に 32 ビットであり、呼び出す関数のアドレスを保持します。

NULL を格納した関数ポインタを使って関数の呼び出しが試みられると、ifetch バスエラーが発生します。

8.7.3.1 特殊なポインタ ターゲット

ポインタと整数は互換ではありません。ポインタに整数定数を割り当てると警告が出力されます。以下に例を示します。

```
const char * cp = 0x123;  // the compiler will flag this as bad code
```

整数定数 (0x123) には、ターゲットの型またはサイズに関する情報は何もありません。また、このコードは移植性に欠けます。さらに、ポインタに整数アドレスが割り当てられて逆参照されると、特に複数のメモリ空間を備えた PIC® デバイスにおいて、コードエラーが非常に高い確度で発生します。

ポインタには必ず C/C++ オブジェクトのアドレスを割り当てる必要があります。デスティネーション アドレスで定義された C/C++ オブジェクトが存在しない場合、目的に適ったオブジェクトをこのアドレスで定義または宣言する必要があります。オブジェクトのサイズは、アクセス可能なメモリ位置のレンジに適合している必要があります。

例として、アドレス `0xA0001000` から始まる 1000 個のメモリ位置のチェックサムを生成する場合を想定します。このデータを読み出すためにポインタを使います。以下のようなコードを思い付くかもしれません。

```
int * cp;
cp = 0xA0001000;  // what resides at 0xA0001000???
```

そして、ポインタをデータごとにインクリメントします。もっと良い方法として、以下のコードが使えます。

```
int * cp;
int __attribute__((address(0xA0001000))) inputData [1000];
cp = &inputData;
// cp is incremented over inputData and used to read values there
```

この場合、配列のサイズと型が指定されるため、コンパイラはターゲットとメモリ空間のサイズを特定できます。

ポインタの比較 (減算) には注意が必要です。以下に例を示します。

```
if(cp1 == cp2)
  ; take appropriate action
```

ANSI C 規格では、2 つのポインタのターゲットが同じオブジェクトである場合にのみ、ポインタの比較が許容されます。アドレスは、配列の終端を越えて 1 エレメントまで拡張できます。

整数定数に対するポインタの比較はさらに危険です。以下に例を示します。

```
if(cp1 == 0xA0000100)
  ; take appropriate action
```

NULL ポインタは、ポインタに定数を割り当てる事ができる 1 つの事例であり、コンパイラはこれを正しく処理します。NULL ポインタは数値的に 0 と等価ですが、これは ANSI C 規格が定める特殊なケースです。`NULL` マクロとの比較も許容されます。

8.8 複素数データ型

現在、MPLAB XC32 C/C++ コンパイラは複素数データ型を実装していません。

8.9 定数の型と書式

定数はソースコード内で数値を表します。例えば 123 は定数です。他の値と同様に、定数にも C/C++ の型が必要です。定数の型に加えて、実際の値には複数ある書式の中の 1 つを指定できます。整数定数の書式は基数を指定します。MPLAB XC32 C は、ANSI 規格の基数指定子に加えて、C コードで 2 進定数を指定するための基数指定子をサポートします。

基数を指定するために使う書式を表 8-3 に示します。2 進または 16 進基数を指定するための文字は、16 進数の各桁に使う文字と同様に、大文字と小文字を区別しません。

表 8-3: 基数の書式

基数	書式	例
2 進数	`0b` *number* (または `0B` *number*)	0b10011010
8 進数	`0` *number*	0763
10 進数	*number*	129
16 進数	`0x` *number* (または `0X` *number*)	0x2F

全ての整数定数には、オーバーフローせずに値を保持できるように `int`、`long int`、`long long int` 型のいずれかが割り当てられます。8 進または 16 進定数においては、符号付きの型では保持可能な値のレンジが小さすぎる場合に、符号なし型 (`unsigned int`、`unsigned long int`、`unsigned long long int`) が割り当てられます。

定数の既定値型は、値の後に添え字を追加する事によって変更できます (例 : `23U` の `U` が添え字です)。表 8-4 に、添え字と型を割り当てる際に考慮される型の関係を示します。例えば、10 進数の定数に添え字 `l` を指定した場合、コンパイラは値を保持可能であれば `long int` 型を割り当てます。しかし、この型ではその定数を保持できない場合、`long long int` 型を割り当てます。8 進または 16 進定数に対しては、符号なし型も考慮されます。

表 8-4: 添え字に対する型の割り当て

添え字	10 進数	8 または 16 進数

`u` または `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` または `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` または `U` と `l` または `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` または `LL`	`long long int`	`long long int` `unsigned long long int`
`u` または `U` と `ll` または `LL`	`unsigned long long int`	`unsigned long long int`

© 2015 Microchip Technology Inc.

以下に、定数に割り当てられる既定値型が不適当であるために問題が生じる可能性のあるコードの例を示します。

```
unsigned long int result;
unsigned char shifter;

int main(void)
{
    shifter = 40;
    result = 1 << shifter;
    // code that uses result
}
```

定数 1 には int 型が割り当てられるため、シフト操作の結果は int 型です。従って、この定数をいくらシフトしても、long 型変数である result の上位ビットは絶対にセットされません。この例では、値 1 は左へ 40 ビットシフトされ、その結果は 0x10000000000 ではなく 0 です。

下の例では、添え字を使って定数の型を変更する事で、シフト結果に unsigned long 型を持たせています。

```
    result = 1UL << shifter;
```

浮動小数点定数の型は、添え字を付けないと double 型であり、添え字 f または F を付けると float 型です。添え字 l または L を付けた場合は long double 型です。

文字定数は、シングル クォーテーション「'」で囲みます (例 : 'a')。文字定数は int 型ですが、コンパイル時に char 型へと最適化される場合があります。

マルチバイト文字定数は、本コンパイラでは許容されますが、標準ライブラリはサポートしません。

文字列定数 (または文字列リテラル) は、ダブルクォーテーション「"」で囲みます (例 : "hello world")。文字列定数の型は const char * です。文字列を構成する文字は、const 修飾された全てのオブジェクトと同様に、プログラムメモリ内に保存されます。

ANSI C 規格に準拠するため、コンパイラは文字または文字配列における拡張キャラクタセットをサポートしません。その代わりとして、バックスラッシュ文字によるエスケープシーケンスを使います。以下に例を示します。

```
const char name[] = "Bj\370rk";
printf("%s's Resum\351", name); \\ prints "Bjørk's Resumé"
```

const なしの char 型を指すポインタに文字列リテラルを割り当てると、コンパイラは警告を出力します。このコードは違反ではありませんが、ポインタが文字列への書き込みを試みた場合の動作は失敗します。以下に例を示します。

```
char * cp= "one";          // "one" in ROM, produces warning
const char * ccp= "two";   // "two"  in ROM, correct
```

以下は、文字列を使って非 const の配列 (ポインタ定義ではない) を定義および初期化します。

```
char ca[]= "two";          // "two"  different to the above
```

これは特殊なケースであり、起動時に文字列 "two" (プログラム空間からコピーされる) を使って初期化される配列がデータ空間内に配置されます。これに対し、他の文脈で使われる文字列定数は無名の const 修飾付き配列を表し、プログラム空間内で直接アクセスされます。

全く同じ文字の並びを持つ文字列が複数存在する場合、MPLAB XC32 C/C++ コンパイラはそれらに対して同じ保存位置とラベルを使います。以下に例を示します。

```
if(strncmp(scp, "hello world", 6) == 0)
    fred = 0;
if(strcmp(scp, "hello world") == 0)
    fred++;
```

2 つの同じ文字列は同じメモリ位置を共有します。同じ文字列が複数の異なるモジュール内にある場合、この最適化を可能にするには、リンク時最適化を有効にする必要があります。

2 つの文字列定数がスペースのみによって区切られている場合、コンパイラはそれらを連結します。以下に例を示します。

```
const char * cp = "hello"  "world";
```

この場合、ポインタには文字列 "hello world" のアドレスが代入されます。

© 2015 Microchip Technology Inc.

8.10 標準型修飾子

型修飾子は、オブジェクトの使われ方に関する追加の情報を提供します。MPLAB XC32 C/C++ コンパイラは、ANSI C の修飾子に加えて、PIC MCU アーキテクチャを利用した組み込みアプリケーション向けの特殊な修飾子もサポートします。

8.10.1 const 型修飾子

MPLAB XC32 C/C++ コンパイラは、ANSI 型修飾子 const および volatile の使用をサポートします。

const 型修飾子は、そのオブジェクトが読み出し専用である (変更されない) という事をコンパイラに伝えるために使います。const 宣言されたオブジェクトに対して変更が試みられた場合、コンパイラは警告またはエラーを出力します。

実行中の任意時点で値を代入する事ができないため、通常、const オブジェクトは宣言時に初期化する必要があります。以下に例を示します。

```
const int  version = 3;
```

この場合、version は int 型変数としてプログラムメモリ内に配置され、常に値 3 を保持します。この値をプログラムによって変更する事はできません。

const 修飾されたオブジェクトは、-mno-embedded-data オプションを指定しない限り、プログラムメモリ内に配置されます。

8.10.2 volatile 型修飾子

volatile 型修飾子は、そのオブジェクトの値が一連のアクセスの間で保持されるとは保証できないという事をコンパイラに伝えるために使います。volatile 宣言は、そのオブジェクトに対する明らかに重複した参照が最適化によって削除されてしまう (それによってプログラムの動作が変わってしまう) 事を防ぎます。

ハードウェアによって変更可能な SFR またはハードウェアを駆動する SFR は全て volatile として修飾されます。また、割り込みルーチンによって変更される可能性のある全ての変数にも、この修飾子を使う必要があります。以下に例を示します。

```
extern volatile unsigned int WDTCON __attribute__((section("sfrs")));
```

volatile 修飾子は、1 動作によるアクセスを保証しませんが、コンパイラはこれを実現しようと試みます。

volatile オブジェクトにアクセスするためにコンパイラが生成するコードは、通常の変数にアクセスするためのコードと異なる場合があります。一般的に、volatile オブジェクト向けのコードではサイズが増加し、実行速度が低下します。従って、この修飾子は必要な場合にのみ使うべきです。しかし、この修飾子を必要な時に使わないと、コードエラーが生じる可能性があります。

volatile キーワードは、C/C++ ソース内で使われていない変数が削除されてしまう事を防ぐためにも使います。volatile ではない変数が一切使われていない場合や、使われていてもプログラムの動作に一切影響しない場合、それらの変数はコンパイラがコードを生成する前に削除される可能性があります。

volatile 変数の名前だけを含む C/C++ 命令文は、その変数のメモリ位置を読み出してその結果を破棄するコードを生成します。以下は、そのような命令文の例です。

```
PORTB;
```

この場合、PORTB を読み出すだけで、この値を使って何もしないアセンブリコードが生成されます。これは、割り込みフラグをリセットするために読み出す必要がある一部の周辺モジュール レジスタ向けに便利に使えます。通常、このような命令文は、何の効果も持たないためコード化されません。

© 2015 Microchip Technology Inc.

8.11 コンパイラに固有の修飾子

現在、MPLAB XC32 C/C++ コンパイラは規格外の修飾子を実装していません。変数と関数の制御には属性を使います。

8.12 変数属性

コンパイラ キーワード `__attribute__` を使うと、変数または構造体フィールドの特殊な属性を指定できます。このキーワードに続く 2 重丸カッコの囲みの中で属性を指定します。

複数の属性は、2 重カッコの中でコンマで区切って指定します。以下に例を示します。

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

> **Note:** 変数属性は、プロジェクト全体で一貫して使う必要があります。例えば、ファイル A で `aligned` 属性を使って定義された変数が、ファイル B で `aligned` 属性を使わずに `extern` 宣言されると、リンクエラーが発生する可能性があります。

`address (addr)`

変数の絶対仮想アドレスを指定します。この属性は、セクション属性と組み合わせて使う事ができます。

> **Note:** ターゲット デバイスが L1 キャッシュを備えていない場合、データ変数のアドレスレンジは通常 [0xA0000000,0xA00FFFFC] です (リンカスクリプト内で「kseg1_data_mem」領域として定義)。ターゲット デバイスが L1 データキャッシュを備えている場合、データ変数のアドレスレンジは通常 [0x80000000,0x800FFFFC] です (リンカスクリプト内で「kseg0_data_mem」領域として定義)。ターゲット デバイスに応じて適切な kseg 領域を使うよう特別な注意が必要です。そうしないと、複数の変数が同じ物理アドレスに割り当てられてしまう可能性があります。

この属性を使うと、以下のように、変数のグループの開始位置を特定のアドレスに指定できます。

```
int foo __attribute__((section("mysection"),address(0xA0001000)));
int bar __attribute__((section("mysection")));
int baz __attribute__((section("mysection")));
```

コンパイラは、指定されたアドレスに対してエラーチェックを実行しません。リンカスクリプトで定義されているメモリレンジまたはターゲット デバイス上の実際のメモリレンジに関係なく、セクションは指定されたアドレスに配置されます。ターゲット デバイスおよびアプリケーションに対して有効なアドレスを指定する事は、アプリケーション コード側の責任です。

また、絶対アドレス属性を指定した変数に対して GP 相対アドレス指定でアクセスしないよう注意が必要です。なぜならば、そのような属性の変数に対するアクセスは、アドレス属性なしの変数へのアクセスに比べて効率が悪いからです。

加えて、絶対アドレス セクションと新しいベストフィット アロケータを効果的に使うには、プログラムメモリおよびデータメモリ セクションをリンカスクリプト内でマッピングしない事が必要です。ビルトイン リンカスクリプトは、ほとんどの標準セクション (`.text`、`.data`、`.bss`、`.ramfunc` セクション等) をマッピングしません。これらのセクションをリンカスクリプト内でマッピングしない事により、シーケンシャル アロケータの代わりにベストフィット アロケータを使って、これらのセクションを割り当てる事ができます。リンカスクリプト内でマッピングされていないセクションは、絶対アドレス セクションの前後に流し込めますが、リンカスクリプトでマッピングされたセクションは互いにグループ化されて連続的に割り当てられるため、絶対アドレス セクションと競合する可能性があります。

© 2015 Microchip Technology Inc.

「小さな」データおよび bss (.sdata, .sbss 等) セクションは、既定値のビルトイン リンカスクリプト内でマッピングされます。なぜならば、「小さな」データ変数は、より効率的な GP 相対アドレス指定モードのアドレスレンジ内に収まるよう、互いにグループ化する必要があるためです。**リンカスクリプト内でマッピングされたこれらのセクションとの競合を防ぐため、絶対アドレス変数には高いアドレスを割り当てる必要があります。**

> **Note:** ほとんどの場合、address属性はspace属性(コードの場合はspace(prog)、データの場合はspace(data))と組み合わせて使います。
> **space(memory-space)**属性に関する説明も参照してください。

aligned (n)

この属性を持つ変数は、次の n バイト境界に配置されます。

aligned 属性は構造体メンバーに対して使う事もできます。そのようなメンバーは、構造体内で、指定された境界に配置されます。

アラインメント値nの指定を省略した場合、変数のアラインメント値は8 (基本のデータ型の最大アラインメント値) に設定されます。

aligned 属性を使うと、変数のアラインメント サイズは減少するのではなく増加するという事に注意が必要です。変数のアラインメント サイズを削減するには、packed 属性を使います。

cleanup (function)

この属性を持つ関数スコープの自動変数がスコープから出ると、指定された関数 (function) を呼び出します。

指定された関数 (function) は、1 つの引数 (この属性を付けた変数と型が互換のターゲットを指すポインタ) を取り、リターン型は void である事が必要です。

coherent

コンパイラ / リンカは、coherent 属性を持つ変数を、kseg0 領域 (L1 キャッシュを備えたデバイスにおける既定値) ではなく kseg1 領域に割り当てられた一意のセクション内に配置します。これにより、その変数はキャッシュ非適用アドレスを介してアクセスされます。

deprecated
deprecated (*msg*)

deprecated として指定された変数が使われた時に警告を出力します。オプションの引数msg (文字列である事が必要)を指定した場合、その文字列が警告に表示されます。

persistent

persistent 属性を持つ変数は、起動時に初期化またはクリアされません。persistent 属性付きの変数を使うと、デバイスリセット時にステート情報を保持する事ができます。コンパイラは、persistent 属性を持つ変数を、既定値の起動コードによってクリアされない特殊な (bss に類似する) セクションに配置します。このセクションは常にデータ空間内に置かれるため、この属性は space() 属性と互換ではありません。

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

persistent 属性は暗黙的に coherent 属性を指定します。つまり、persistent 属性を持つ変数はキャッシュ非適用アドレスを介してアクセスされます。

packed

この属性を持つ変数または構造体メンバーは、サイズが最小となるようにアラインメントされます。つまり、宣言に対してアラインメントのためのパディング保存領域は割り当てられません。aligned 属性と packed 属性を組み合わせて使う事により、アラインメントの制約を任意に設定できます (変数または構造体メンバーの型に対する既定値アラインメントよりも大きくする事も小さくする事もできます)。

section ("section-name")

この属性を持つ変数は、指定された名前 (section-name) を持つセクションに配置されます。以下に例を示します。

```
unsigned int dan __attribute__ ((section (".quixote")))
```

変数 dan はセクション .quixote に配置されます。

-fdata-sections コマンドライン オプションは、section 属性付きで定義された変数に対して効果を持ちません。この効果を得るには、unique_section も指定する必要があります。

space(memory-space)

コンパイラは、space 属性を持つ変数を、指定されたメモリ空間 (memory-space) に割り当てます。有効なメモリ空間として prog (プログラムメモリ) または data (データメモリ) を指定できます。data 空間は、非 const 変数向けの既定値です。この属性は、初期化済みデータの扱い方も制御します。既定値 (space(data)) の場合、リンカはデータ初期化テンプレート内にエントリを生成します。しかし、space(prog) の場合、変数は不揮発性メモリ内に配置されるため、リンカはエントリを生成しません。

以下に例を示します。

```
const unsigned int __attribute__((space(prog)) jack = 10;
signed int __attribute__((space(data))) oz = 5;
```

unique_section

この属性を持つ変数は、-fdata-sections が指定されたかのように、一意の名前を持つセクション内に配置されます。その変数が section ("section-name") 属性も持つ場合、指定された section-name は、生成される一意セクション名の接頭辞として使われます。

以下に例を示します。

```
int tin __attribute__ ((section (".ofcatfood"), unique_section)
```

変数 tin はセクション .ofcatfood 内に配置されます。

unused

この属性は、その変数がプログラム内で使われないという事をコンパイラに知らせます。コンパイラは、その変数が使われていなくても警告を出力しません。

weak

weak 属性により、その宣言は弱いシンボルとして発行されます。弱いシンボルとは、同じシンボルのグローバルなバージョンが存在する場合はグローバルな方を使うという事を意味します。

外部シンボルへの参照に weak 属性を適用した場合、シンボルはリンクには不要です。以下に例を示します。

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


© 2015 Microchip Technology Inc.

このプログラムにおいて、s が他のモジュールによって定義されていない場合、プログラムはリンクしますが、s にはアドレスが与えられません。プログラム内の条件式は、s が定義されているかどうかを確認します。そして、定義されていればその値を返し、未定義であれば「0」を返します。この機能には各種の用途がありますが、主にオプション ライブラリとのリンクが可能な汎用コードを提供するために使います。

NOTE:

© 2015 Microchip Technology Inc.

第 9 章　メモリの割り当てとアクセス

9.1 はじめに

RAM ベースの変数は、大きく 2 つのグループに分類されます。1 つのグループは何らかの形態のスタックに割り当てられる自動 / パラメータ変数であり、もう 1 つのグループはデータメモリ空間全体に自由に配置されるグローバル / 静的変数です。本章では、これら 2 つのグループのメモリ割り当てについて詳しく説明します。

- アドレス空間
- データメモリ内の変数
- auto 変数のメモリ割り当てとアクセス
- プログラムメモリ内の変数
- レジスタ内の変数
- 動的なメモリ割り当て
- メモリモデル

9.2 アドレス空間

8/16 ビット PIC デバイスとは異なり、PIC32 は統一プログラミング モデルを採用しています。PIC32 はコード、データ、周辺モジュール、コンフィグレーション ビットの全てに対して単一の 32 ビット幅アドレス空間を提供します。

この単一アドレス空間内のメモリ領域は、各種の用途向け (命令コード向け、データ向け等) に指定されます。デバイスは、これらの領域内の命令とデータに対して、別々の内部バスを使ってアクセスします[1]。従って並列アクセスが可能です。8/16 ビット PIC デバイスで使われる用語「プログラムメモリ」と「データメモリ」は PIC32 でも使いますが、8/16 ビット PIC はこれらを別々のアドレス空間内に実装するのに対し、PIC32 は単一のアドレス空間内に実装します。

CPU によって使われるデバイス内の全てのアドレスは仮想アドレスです。これらは、システム制御レジスタ (CP0) によって物理アドレスにマッピングされます。

1. 内部バスの構成という観点から、このデバイスはハーバード アーキテクチャであると見なせます。

9.3 データメモリ内の変数

ほとんどの変数は、最終的にデータメモリ内に配置されます。例外として、const 修飾された非 auto 変数は、プログラムメモリ空間内に配置されます (**8.10.1「const 型修飾子」**参照)。

auto 変数と非 auto 変数は根本的に異なる方法でメモリに割り当てられるため、以下では、これらの変数について別々に説明します。C/C++ 言語規格の用語に従うと、これら 2 種類の変数は、値の保持期間が自動的に設定される変数と、値が恒久的に保持される変数です。

> **Note:** 「ローカル」および「グローバル」という用語は変数の説明でよく使われますが、これらは言語規格によって定義されている用語ではありません。多くの場合、「ローカル変数」は「関数の内部をスコープとする変数」という意味で使われ、「グローバル変数」は「プログラム全体をスコープとする変数」という意味で使われます。しかし、C/C++ 言語には 3 種類の一般的スコープ (ブロック、ファイル (すなわち内部結合)、プログラム (すなわち外部結合)) があるため、2 つの用語だけを使ってこれら 3 つのスコープを表現すると、混乱を招く可能性があります。例えば、関数の外で定義された static 変数のスコープはそのファイル内のみに限られます。従って、その変数は「グローバル」にアクセス可能ではありません。一方、その変数にはそのファイル内の複数の関数からアクセス可能であるため、1 つの関数に対して「ローカル」でもありません。メモリ割り当てにおいては、変数は auto かどうかに基づいてメモリ空間に割り当てられます。従って、以下の説明はこの分類方法に従います。

9.3.1 非 auto 変数のメモリ割り当て

コンパイラは、非 auto 変数 (恒久的に保持される変数) を、利用可能な任意のデータバンクに配置します。これは 2 段階で処理されます。すなわち、各変数を適切なセクションに配置した後に、そのセクションをデータメモリ内にリンクします。

コンパイラは、非 auto 変数の 3 つのカテゴリを考慮します。このカテゴリ分けは、プログラムの実行開始までに変数に格納すべき値に基づきます。これらのカテゴリには以下のセクションが使われます。

.pbss　これらのセクションは、persistent 属性を持つ (つまり、スタートアップ コードによって値が変更されない) 変数を格納するために使います。それらの変数は、起動時にクリアも変更もされません。

.bss　これらのセクション(.sbssも含む)は、非初期化変数(定義時に値が代入されない変数) またはスタートアップ コードによってクリアされる変数を格納します。

.data　これらのセクション(.sdataも含む)は、初期化変数(定義時に非ゼロの初期値が代入され、スタートアップ コードによってコピーされた値を保持する変数) の RAM イメージを格納します。

初期化変数を保持するために使われる .data セクションは、RAM 変数自体を保持するセクションです。スタートアップ コードによって RAM 変数にコピーされる初期値を保持するために、対応するセクション (.dinit) がプログラムメモリ内に配置されます (従ってこのセクションは不揮発性です)。

© 2015 Microchip Technology Inc.

9.3.2 static 変数

全ての static 変数は、たとえそれらが関数の内部で宣言されたローカルな static 変数であっても、恒久的に保持されます。ローカルな static 変数のスコープは、それらが宣言された関数内またはブロック内に制限されますが、auto 変数とは異なり、割り当てられたメモリの内容はプログラムの全期間を通して保持されます。従って、それらの変数は、他の非 auto 変数と同様の方法でメモリに割り当てられます。static 変数は恒久的に保持されるため、それらの変数には他の関数からポインタを介してアクセスできます。

static 変数は、ポインタを介して明示的に変更されない限り、関数を何度呼び出しても同じ値を保持します。

static 変数は、一度代入された初期値をプログラムの実行中に保持します。auto オブジェクトを初期化するには、そのオブジェクトが定義されているブロックが実行を開始するたびに値を代入する必要があるため、static 変数を使った方が好ましい場合があります。static 変数は、他の非 auto オブジェクトと同様の方法で、スタートアップ コードによって初期化されます (**5.5.2「周辺モジュール ライブラリの関数」**参照)。static 変数は、その変数を非 static にした場合と同じセクション内に配置されます。

9.3.3 非 auto 変数のサイズの制限

本コンパイラは、どのような型の配列も (複数の型が混在する配列も含めて) 完全にサポートします。構造体および共用体でも型の混在は可能です (**8.6「構造体と共用体」**参照)。これらのオブジェクトのサイズに関する理論的な制限はありません。

9.3.4 非 auto 変数の既定値のメモリ割り当てを変更する

非 auto 変数は、各種の方法で既定値以外の位置に配置できます。

変数は、修飾子を使う事によって、異なるデバイスメモリ空間に配置できます。例えば、変数をプログラムメモリ空間に配置するには、const 指定子を使います (**8.10.1「const 型修飾子」**参照)。

1 つまたは複数のデータメモリ位置を他の目的に使えるよう確保し、それらのメモリ位置を避けて全ての変数を配置する必要がある場合、address 属性を使って、それらのメモリ位置を占有するダミーの変数 (または配列) を定義するのが最善の方法です (**8.12「変数属性」**参照)。

少数の非 auto 変数だけをデータメモリ空間内の特定アドレスに配置する必要がある場合は、それらの変数に対して address 属性を使います。この属性に関する説明は **8.12「変数属性」**に記載しています。

9.3.5 データメモリ割り当てマクロ

sys/attribs.h ヘッダファイルは、コードの可読性を向上させるためによく使われる属性を適用するための各種マクロを提供します。

__section__(s)	セクション名 s を持つ section 属性を適用します。
__unique_section__	unique_section 属性を適用します。
__ramfunc__	関数を RAM 関数コードセクション内に配置します。
__longramfunc__	関数を RAM 関数コードセクション内に配置し、longcall 属性を適用します。
__longcall__	longcall 属性を適用します。
__ISR(v,ipl)	優先度 ipl を持つ interrupt 属性と、ベクタ番号 v を持つ vector 属性を適用します。
__ISR_AT_VECTOR(v,ipl)	優先度 ipl を持つ interrupt 属性と、ベクタ番号 v を持つ at_vector 属性を適用します。このマクロは、特に可変ベクタオフセットを備える PIC32 デバイスで役立ちます。
__ISR_SINGLE__	関数をシングルベクタ モードの割り込みサービスルーチンとして指定します。これは、割り込みハンドラへの分岐を常に同じ 1 つのベクタアドレスに配置します。
__ISR_SINGLE_AT_VECTOR__	シングルベクタ割り込みハンドラ全体をベクタ 0 位置に配置します。これを使う場合、ベクタの間隔はハンドラのサイズに適合するよう設定されている事が必要です。

© 2015 Microchip Technology Inc.

9.4 auto 変数のメモリ割り当てとアクセス

以下では、auto 変数 (値の保持期間が自動的に決まる変数) のメモリ割り当てについて説明します。これには、auto 変数のように動作する関数パラメータ変数と、コンパイラによって定義される一時変数も含みます。

auto (autoは*automatic*の略) 変数は、ローカル変数の既定値型です。staticとして明示的に宣言しない限り、ローカル変数は auto 変数として扱われます。auto キーワードは、指定してもしなくても構いません。

auto 変数は、その名が示すように、関数が実行される時に自動的に存在し、関数がリターンする際に自動的に消滅します。これらの変数はプログラム実行期間の全体を通して存在するのではないため、それらの変数が存在していない時にそれらが使用するメモリが再要求されると、そのメモリはプログラム内の他の変数用に割り当てられる可能性があります。

全ての auto 変数を保存するために、PIC32 のソフトウェア スタックが使われます。関数は再入可能であり、関数の各インスタンスは各自の自動およびパラメータ変数を退避するために、スタック上に独自の領域を持ちます。スタックの詳細は、**7.4「スタック」**と **14.3.2「スタックポインタとヒープの初期化」**を参照してください。

コンパイラは、汎用レジスタ 29 をソフトウェア スタックポインタ専用に使います。関数呼び出し、割り込み、例外処理を含む全てのプロセッサ スタック動作にはソフトウェア スタックを使います。スタックはアドレスの高い方から低い方へ進みます。

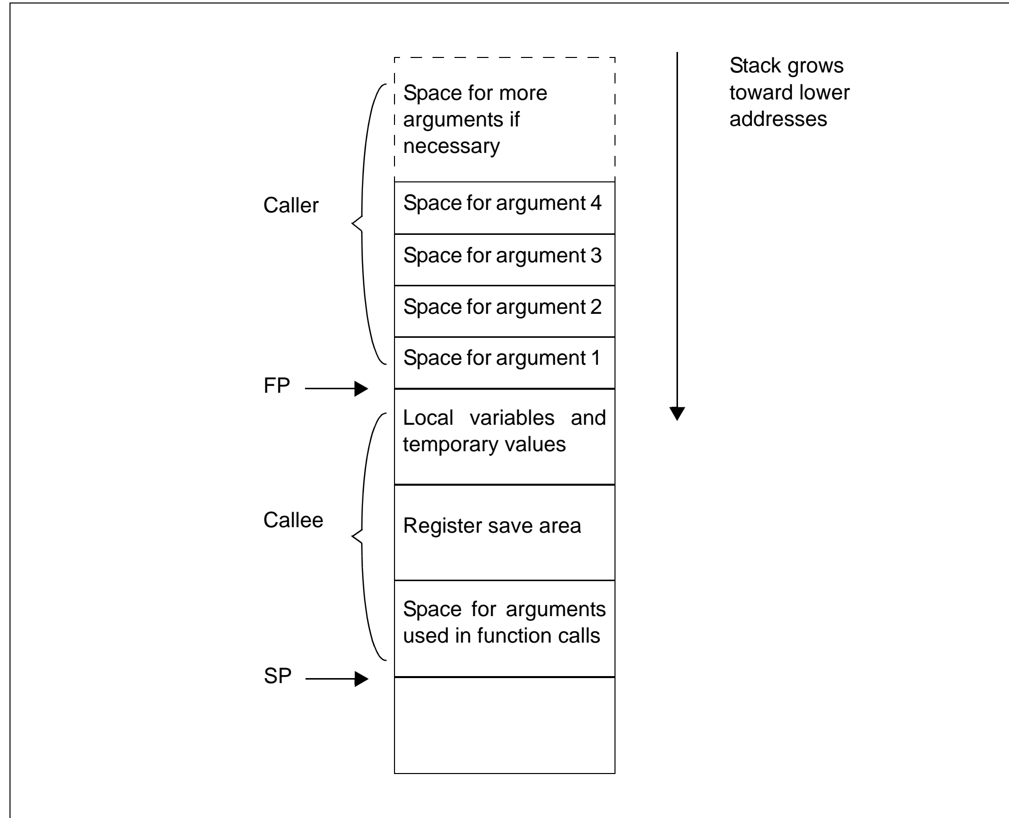
既定値によるスタックのサイズは 1024 バイトです。このサイズは、リンカのコマンドライン オプション --defsym_min_stack_size を使ってサイズを指定する事により変更できます。コマンドラインで 2048 バイトのスタックを割り当てる場合の例を以下に示します。

```
xc32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

実行時にスタックはアドレスの高い方から低い方へ進みます (図 9-1 参照)。コンパイラは、スタックを管理するために、以下の 2 つのレジスタを使います。

- レジスタ (sp) - これはスタックポインタです。スタック上で次に空いている位置を指します。
- レジスタ 30 (fp) - これはフレームポインタです。現在の関数のフレームを指します。各関数は必要に応じて新しいフレームを生成し、そこから自動変数または一時変数を割り当てます。コンパイラによる最適化は、フレームポインタを介するスタックポインタ参照を排除し、スタックポインタを介する等価な参照に置き換える場合があります。これにより、フレームポインタが汎用レジスタとして使えるようになります。

図 9-1: スタックフレーム



標準修飾子 const および volatile は、どちらも auto 変数に対して使えます。これらを使っても、メモリ内の変数の配置には影響しません。つまり、ローカルな const 修飾子付きオブジェクトは auto オブジェクトのままであり、データメモリ内でスタック上のメモリに割り当てられます (非 auto の const オブジェクトのようにプログラムメモリ内に割り当てられるのではありません)。

9.4.1 ローカル変数のサイズの制限

auto 変数のサイズに関する理論的な制限はありません。

© 2015 Microchip Technology Inc.

9.5 プログラムメモリ内の変数

プログラムメモリ内に配置される変数は、const で修飾された非 auto 変数だけです。-mno-embedded-data オプションを使うと、const オブジェクトであってもプログラムメモリではなく RAM 内に配置されます。const 修飾された auto 変数は、他の auto 変数と一緒にスタック上に配置されます。

> **Note:** デバイスがフラッシュメモリを内蔵している場合、const 修飾された変数に既定値の -membedded-data オプションを組み合わせる事で、定数を不揮発性フラッシュメモリ内に配置できます。

全ての const 修飾された変数は (auto であっても非 auto であっても) 常に読み出し専用であり、ソースコード内でそれらの変数への書き込みが試みられると、コンパイラはエラーを出力します。

プログラムの実行時にこれらのオブジェクトに書き込む事はできないため、通常 const オブジェクトは初期値と一緒に定義します。しかし、初期化は必須ではありません。未初期化の const オブジェクトは、他の未初期化 RAM 変数と一緒に bss セクション内の空間に割り当てられ、コンパイラはそれらの const オブジェクトをやはり読み出し専用として扱います。

```
const char IOtype = 'A';  // initialized const object
const char buffer[10];    // I just reserve memory in RAM
```

9.5.1 const 変数のサイズの制限

const 変数のサイズに関する理論的な制限はありません。

9.5.2 既定値のメモリ割り当てを変更する

1 つまたは複数のプログラムメモリ アドレスを他の目的に使うために、どの変数もそれらのアドレスを使わないようにする必要がある場合、メモリ調整オプションを使ってそれらのアドレスを予約するのが最善の方法です。

少数の非 auto 変数だけをデータメモリ空間内の特定アドレスに配置する必要がある場合は、address 属性を使う事で、それらの変数を目的のアドレスに配置します。この属性に関する説明は **8.12「変数属性」** に記載しています。

9.6 レジスタ内の変数

変数をメモリ位置ではなくレジスタに割り当てる事で、コードの効率を向上させる事ができます。MPLAB XC32 C/C++ コンパイラは、コード最適化の一部として、変数をレジスタに割り当てる場合があります。レベル 1 以上の最適化を行った場合、変数に代入される値がレジスタにキャッシュされる場合があります。その間は、その変数に割り当てられたメモリ位置は有効な値を保持しない可能性があります。

register キーワードを使うと、優先的にレジスタに割り当てて欲しい変数をコンパイラに対して指定できます。しかし、これはあくまでもコンパイラに対して推奨するだけであり、尊重されない場合があります。特定のレジスタを指定する事もできますが、これは推奨しません (指定したレジスタがコンパイラのニーズと競合する可能性があるため)。コード内で特定のレジスタを使うと、コンパイラが生成するコードの効率が低下する可能性があります。

例 :

```
volatile unsigned int special;
unsigned int example (void)
{
  register unsigned int my_reg __asm__("$4");
  my_reg += special;
  return my_reg;
}
```

12.6「関数のパラメータ」 で説明するように、レジスタを介して関数にパラメータを渡す事ができます。

9.7 動的なメモリ割り当て

実行時ヒープは、データメモリ内の初期化されない領域であり、標準 C ライブラリの動的メモリ割り当て関数 (calloc、malloc、realloc) と C++ の新しい演算子を使った動的なメモリ割り当てのために使います。ほとんどの C++ アプリケーションではヒープが必要です。

これらの関数をどれも使わない場合、ヒープを割り当てる必要はありません。既定値ではヒープは生成されません。

MPLAB X では、プロジェクト プロパティ内で xc32-ld リンカ向けにヒープサイズを指定できます。MPLAB X は、プロジェクトのビルド時に、このオプションを自動的にリンカに渡します。

動的メモリ割り当て関数を使う場合、その方法が直接的 (メモリ割り当て関数の 1 つを直接呼び出す) であれ間接的 (メモリ割り当て関数の 1 つを使う標準 C ライブラリ関数を呼び出す) であれ、ヒープを生成する必要があります。ヒープを生成するには、コマンドラインで--defsym_min_heap_size リンカ コマンドライン オプションを使ってサイズを指定します。512 バイトのヒープを割り当てる場合のコマンドラインの例を以下に示します。

```
xc32-gcc foo.c -Wl,--defsym,_min_heap_size=512
```

以下は、xc32-g++ ドライバを使って 0xF000 バイトのヒープを割り当てる場合の例です。

```
xc32-g++ vector.cpp -Wl,--defsym,_min_heap_size=0xF000
```

リンカは、スタックの直前にヒープを割り当てます。

© 2015 Microchip Technology Inc.

9.8 メモリモデル

MPLAB XC32 C/C++ コンパイラは、変数のメモリ割り当てを変更するために、固定されたメモリモデルを使いません。

GP 相対アドレス指定のしきい値を制御する -G オプション (**5.9.1「PIC32 専用オプション」**参照) の機能は、8/16 ビット アーキテクチャ向けに Microchip 社製コンパイラが提供する small-data/large-data/scalar-data メモリモデルに似ています。-G オプションの値は、「小さな」データ セクション (sbss、sdata 等) に割り当てられるオブジェクトの最大サイズを指定します。「小さな」データセクションに割り当てられた変数は、他のデータ セクションに割り当てられた変数よりも少ない命令数でアクセスできます。

一般的に、-G オプションの値を大きくすると、コードの効率は向上します。しかし、GP 相対アドレス指定は、64KB 以下の小さなデータに制限されます。

NOTE:

© 2015 Microchip Technology Inc.

第 10 章　演算子と命令文

10.1　はじめに

MPLAB XC32 C/C++ コンパイラは全ての ANSI 演算子をサポートします。一部の演算子の厳密な結果は処理系定義です。処理系定義のふるまいについては、**補遺 B.「処理系定義のふるまい」**で詳細に説明します。本章では、しばしば誤解される事のあるコード動作と、コンパイラが実行可能なその他の動作について説明します。

- 整数拡張
- 型の参照
- 値としてのラベル
- 条件演算子のオペランド
- case のレンジ指定

10.2　整数拡張

演算子に対して複数のオペランドが存在する場合、通常それらは厳密に同じ型である事が必要です。必要に応じてコンパイラはそれらが同じ型となるよう自動的にオペランドを変換します。この変換はサイズの大きい方の型に揃えるため、情報の損失はありません。しかし、型の変化によって期待とは異なるコードのふるまいが生じる場合があります。これらは標準型変換に分類されます。

演算子の両側のオペランドが同じ型であっても、標準型変換の前に一部のオペランドがより大きな型に無条件に変換される場合があります。これは「整数拡張」と呼び、標準 C のふるまいに含まれます。MPLAB XC32 C/C++ コンパイラは、整数拡張を必要に応じて実行します。この動作を制御する (または無効にする) ためのオプションはありません。型が変更された事に気付かないと、一部の式で予測外の結果が生じる可能性があります。

整数拡張は列挙型、`signed char` および `unsigned char` 型、`short int` 型、ビットフィールド型を `signed int` 型または `unsigned int` 型のどちらかへ暗黙的に変換します。変換した結果が `signed int` によって表現可能であれば `signed int` に変換し、そうでなければ `unsigned int` に変換します。

以下に例を示します。

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

`a - b` の `unsigned char` 型の結果は 206 であり、10 未満ではありません。しかし、`a` と `b` は減算の前に整数拡張によって `signed int` 型に変換されます。従って減算結果は -50 (10 未満) であり、`if()` 文の本体は実行されます。

減算結果を `unsigned` にするには、キャスト (型変換) を適用します。以下に例を示します。

```
if((unsigned int)(a - b) < 10)
  count++;
```

この場合、比較は `unsigned int` で行われ、`if()` 文の本体は実行されません。

他に、ビット単位の補数演算子「~」でもよく問題が起こります。この演算子は値の各ビットをトグルします。以下に例を示します。

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

c の値が 0x55 である場合、~c は 0xAA になるとよく勘違いされます。しかし、実際の結果は 0xFFFFFFAA であり、この例の比較結果は「偽」です。状況によっては、このような問題に対してコンパイラが不適合比較エラーを発行します。この問題にもキャストを使って対処できます。

以上のように、整数拡張によって演算は char 型オペランドではなく int 型オペランドを使って実行されます。しかし、オペランドの型が char であっても int であっても演算結果が異ならない場合があります。その場合、MPLAB XC32 C/C++ コンパイラはコード効率の向上のために整数拡張を適用しません。以下に例を示します。

```
unsigned char a, b, c;
a = b + c;
```

厳密に言うと、この命令文では b と c を unsigned int に拡張した後に加算を実行し、加算結果を a の型にキャストした後に代入します。b と c を型拡張した unsigned int 加算の結果が型拡張なしの unsigned char 加算の結果と異なったとしても、unsigned int 型の加算結果を unsigned char 型に戻した最終的な結果は同じです。8 ビット加算の方が 32 ビット加算よりも効率的である場合、コンパイラは前者を採用します。

上の例で a が unsigned int 型であった場合、ANSI C 規格に準拠するために整数拡張が必要です。

 © 2015 Microchip Technology Inc.

10.3 型の参照

式の型は typeof キーワードを使って参照できます。これは言語に対する規格外拡張です。この機能を使うとコードの移植性が低下します。

このキーワードの構文は sizeof と似ていますが、意味的には typedef で定義された型名であるかのように機能します。

typeof の引数は式または型で指定します。以下は、引数を式で指定する場合の例です。

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

これは x が関数の配列である事を想定しています。この場合、関数の値の型を参照します。

以下は、引数を型 (typename) で指定する場合の例です。

```
typeof (int *)
```

この場合、int へのポインタの型を参照します。

ANSI C プログラムにインクルードするヘッダファイルを書く場合、typeof の代わりに __typeof__ を使う必要があります。

typeof は、typedef 名が使える場所であればどこでも使えます。例えば宣言内、キャスト内、sizeof または typeof の内部で使えます。

- 以下は、x が指す対象の型を使って y 宣言します。
  ```
  typeof (*x) y;
  ```
- 以下は、x が指す対象の型を持つ値の配列として y を宣言します。
  ```
  typeof (*x) y[4];
  ```
- 以下は、char を指すポインタの配列として y を宣言します。
  ```
  typeof (typeof (char *)[4]) y;
  ```
  これは以下の伝統的な C 宣言と等価です。
  ```
  char *y[4];
  ```

typeof を使って宣言する事の意味と利点を理解するために、以下のマクロを書き換えます。

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

上の宣言は以下のように書き換え可能です。

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

従って、array (pointer (char), 4) は char を指す 4 つのポインタの配列の型です。

10.4 値としてのラベル

単項演算子「&&」を使うと、現在の関数 (またはそれを包含する関数) 内で定義されているラベルのアドレスを取得できます。これは言語に対する規格外拡張です。この機能を使うとコードの移植性が低下します。

戻り値の型は void * です。この値は定数であり、その型の定数が有効であれば使う事ができます。以下に例を示します。

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

これらの値を使うには、そこにジャンプできる必要があります。これには計算型ジャンプ文 goto *exp を使います。以下に例を示します。

```
goto *ptr;
```

void * 型の全ての式が許容されます。

例えばこれらの定数は、以下のようにジャンプテーブルとして機能する静的配列の初期化に使えます。

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

この場合、以下のようにインデックスを使って 1 つのラベルを選択できます。

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

> **Note:** これはインデックスが有効範囲内であるかどうかチェックしません (C 言語は配列のインデックスをチェックしない)。

このようなラベル値の配列は switch 文と同様の用途で使えます。しかし、よりクリーンな switch 文の方が配列には適しています。

ラベル値の他の用途として、スレッド化されたコードのインタープリタ内での使用があります。インタープリタ関数内のラベルは、高速な送信のためにスレッド化されたコード内に保存できます。

この方式の使用を誤ると、異なる関数内のコードへジャンプする可能性があります。コンパイラはこれを防ぐ事ができないため、現在の関数に対して有効なターゲット アドレスを指定するよう注意が必要です。

© 2015 Microchip Technology Inc.

10.5 条件演算子のオペランド

条件式内の中間にあるオペランドは省略できます。その場合、第 1 オペランドが非ゼロであれば、その値が条件式の値となります。これは言語に対する規格外拡張です。この機能を使うとコードの移植性が低下します。

以下に例を示します。

```
x ?: y
```

x が非ゼロであれば上式は x の値を持ち、x がゼロであれば y の値を持ちます。

この式は以下の式と全く等価です。

```
x ? x : y
```

この単純なケースでは、中間オペランドの省略は特に有用ではありません。しかし、第 1 オペランドが副作用を含むかその可能性がある場合 (マクロの引数である場合) は役立ちます。この場合、中間オペランドを 2 回使うと副作用も 2 回発生します。中間オペランドを省略すると、望ましくない再計算の影響を受けずに計算済みの値が使われます。

10.6 case のレンジ指定

以下に示すように、1 つの case ラベル内で連続する値のレンジを指定できます。

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

これは、*low* から *high* までの整数値 (*low* と *high* を含む) を 1 つずつ収めた複数の case ラベルを使った場合と効果は同じです。これは言語に対する規格外拡張です。この機能を使うとコードの移植性が低下します。

この機能は、ある範囲内の一連の ASCII 文字コードを指定する場合に特に便利です。

```
case 'A' ...'Z':
```

注意 : 整数値を指定する場合、「...」の前後にスペースが必要です。スペースを省略すると誤って構文解析されます。以下に例を示します。

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

上記は正しく、下記は誤りです。

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

NOTE:

© 2015 Microchip Technology Inc.

第 11 章　レジスタの使用

11.1　はじめに

本章では、C/C++ ソースコードからアセンブリコードを生成するためにコンパイラが使うレジスタについて説明します。

- レジスタの使用
- レジスタの用法

11.2　レジスタの使用

コンパイラは、C/C++ コードからアセンブリコードを生成するために PIC MCU 上の特定のレジスタを使います。コンパイラは、自身が生成するコード以外によってこれらのレジスタの内容が変更される事はないと想定します。しかし、コンパイラがコードを適切に調整できるよう、拡張アセンブリ言語フォーマットを使ってアセンブリコード内で使われるレジスタをコンパイラに知らせる事ができます。

11.3　レジスタの用法

PIC32 が備える 32 個の汎用レジスタを ?11-1 に示します。コンパイラはこれらの一部に特定のタスクを割り当てます。この表には、アセンブリコード内で使われる名前と用途を記載しています。

表 11-1:　レジスタの用法

レジスタ番号	ソフトウェア内の名称	用途
$0	`zero`	常に「0」として読み出されます。
$1	`at`	アセンブラー時変数を格納します。完全に理解せずにソースコードから $at レジスタを使わないでください。
$2-$3	`v0-v1`	関数からの戻り値を格納します。
$4-$7	`a0-a3`	関数に引数を渡すために使います。
$8-$15	`t0-t7`	式を評価するためにコンパイラが使う一時レジスタです。値は関数呼び出しを越えて保持されません。
$16-$23	`s0-s7`	関数呼び出しを越えて値を保持する一時レジスタ
$24-$25	`t8-t9`	式を評価するためにコンパイラが使う一時レジスタです。値は関数呼び出しを越えて保持されません。
$26-$27	`k0-k1`	割り込み / トラップハンドラ向けに予約済みです。
$28	`gp`	グローバル ポインタとして使います。
$29	`sp`	スタックポインタとして使います。
$30	`fp` または `s8`	必要に応じてフレームポインタとして使います。それ以外の場合は追加の一時保存レジスタとして使います。
$31	`ra`	関数の戻りアドレスを格納します。

NOTE:

© 2015 Microchip Technology Inc.

第 12 章　関数

本章では、関数定義の書き方と、アプリケーションに応じたカスタマイズの方法について説明します。パラメータと戻り値の用法と、アセンブリ呼び出しシーケンスについても説明します。

- 関数の書き方
- 関数属性と指定子
- 関数コードのメモリ割り当て
- 既定値の関数メモリ割り当てを変更する
- 関数のサイズ制限
- 関数のパラメータ
- 関数の戻り値
- 関数の呼び出し
- 関数のインライン展開

12.1　関数の書き方

関数は C/C++ 言語規格に従った通常の方法で書く事ができます。

`static`は関数に影響する唯一の指定子です。割り込み関数は`interrupt`属性を使って定義します (**12.2「関数属性と指定子」**参照)。

`static` 指定子を使って定義した関数は、その関数のスコープにのみ影響します。つまり、ソースコード内でその関数を呼び出せる箇所が制限されます。`static` 関数は、その関数が定義されているファイル内のコードからのみ直接呼び出せます。関数が `static` である場合、アセンブリコード内でその関数を表すために使う等価シンボルは変化する場合があります (**9.3.2「`static` 変数」**参照)。この指定子は、生成される関数のコードに影響しません。

© 2015 Microchip Technology Inc.

12.2 関数属性と指定子

12.2.1 関数属性

address(addr)

address 属性は関数の絶対仮想アドレスを指定します。address 属性を指定する場合、ターゲット デバイス上の適切な仮想アドレスを使う必要があります。一般的にアドレスの範囲は [0x9D000000,0x9D0FFFFC](リンカスクリプトで「kseg0_program_mem」として定義されているメモリ領域) です。以下に例を示します。

```
__attribute__((address(0x9D008000))) void bar (void);
```

コンパイラはアドレスに関するエラーチェックを実行しません。関数を格納するセクションは、リンカスクリプトで定義されているメモリ領域またはターゲット デバイス上の実際のメモリレンジに関係なく、指定されたアドレスに配置されます。アプリケーション コードはターゲット デバイス上の有効なアドレスを指定する必要があります。

絶対アドレス セクションと新しいベストフィット アロケータを効果的に使うには、プログラムメモリおよびデータメモリ セクションをリンカスクリプト内でマッピングしない事が必要です。ビルトイン リンカスクリプトは、ほとんどの標準セクション (.text、.data、.bss、.ramfunc セクション等) をマッピングしません。これらのセクションをリンカスクリプト内でマッピングしない事により、シーケンシャル アロケータの代わりにベストフィット アロケータを使ってこれらのセクションを割り当てる事ができます。リンカスクリプト内でマッピングされていないセクションは、絶対アドレス セクションの前後に流し込めますが、リンカスクリプトでマッピングされたセクションは互いにグループ化されて連続的に割り当てられるため、絶対アドレス セクションと競合する可能性があります。

alias ("symbol")

その関数が別のシンボルのエイリアスである事を示します。以下に例を示します。

```
void foo (void) { /* stuff */ }
__attribute__ ((alias("foo"))) void bar (void);
```

シンボル bar はシンボル foo のエイリアスであると見なされます。

always_inline

関数が inline 宣言されている場合、最適化レベルが指定されていなくても常にその関数をインライン展開します。

at_vector

関数の本体を指定された例外ベクタアドレスに配置します。
第 13 章「割り込み」と **13.5「例外ハンドラ」**を参照してください。

const

pure属性を持つ関数がその戻り値を自身のパラメータだけを使って(すなわちグローバル変数を一切使わずに) 決定する場合、const を宣言する事でより積極的な最適化が可能になります。ポインタ引数を逆参照する関数は const ではありません。なぜならば、ポインタ逆参照は (ポインタ自体はパラメータであっても) パラメータではない値を使うからです。

deprecated
deprecated (msg)

deprecated として指定された関数が使われた時に警告を出力します。オプションの引数 msg (文字列) を指定した場合、その文字列が警告に表示されます。deprecated 属性は変数と型に対しても使えます。

 © 2015 Microchip Technology Inc.

far

最初に関数のアドレスをレジスタに転送した後に、そのレジスタの内容を使って関数を呼び出します。これにより、CALL 命令で直接呼び出し可能な 28 ビット アドレスレンジを越えた位置にある関数を呼び出す事ができます。

format (type, format_index, first_to_check)

format 属性は、関数が printf、scanf、strftime、strfmon のいずれかのスタイルの書式文字列と引数を扱い、コンパイラは (標準ライブラリ関数の場合と全く同様に) 書式文字列に対してそれらの引数の型をチェックする必要があるという事を示します。

type パラメータは printf、scanf、strftime、strfmon のいずれかです (パラメータの前後に任意にアンダースコアを付ける事ができます (例 : __printf__))。これにより、書式文字列の解釈方法を指定します。

format_index パラメータは、どの関数パラメータが書式文字列なのか指定します。関数パラメータには左から順番に 1 から始まる連番が付けられます。

first_to_check パラメータは、書式文字列に対して最初にチェックするパラメータを指定します。first_to_check が 0 の場合、コンパイラは型をチェックせずに書式文字列の整合性のみチェックします (例 : vfprintf)。

format_arg (index)

format_arg 属性は、関数が printf スタイルの書式文字列を扱い、コンパイラは書式文字列の整合性をチェックする必要があるという事を指定します。index は、どの関数パラメータが書式文字列なのか指定します。

interrupt (priority)

その関数に対して割り込みハンドラ関数としてのプロローグおよびエピローグ コードを生成します。**第 13 章「割り込み」**を参照してください。引数には IPLnSOFT、IPLnSRS、IPLnAUTO のいずれかを使って割り込み優先度を指定します (n は 7 レベルの優先度を指定し、SOFT/SRS/AUTO はコンテクスト退避モードを指定)。

keep

__attribute__((keep)) は関数に対して適用できます。keep 属性は、リンカの --gc-sections オプションによって関数が削除される事を (たとえその関数が未使用であっても) 防ぎます。

longcall

far と機能的に等価です。

malloc

この属性を持つ関数からの非 NULL のポインタ戻り値は、関数の戻り時点でアクティブである他のポインタのエイリアスになりません。これにより、コンパイラは最適化を改善できます。

micromips

コードを圧縮する microMIPS 命令セットで関数のコードを生成します。

mips16

MIPS16 命令セットで関数のコードを生成します。

naked

その関数向けにプロローグ コードもエピローグ コードも生成しません。

near

-mlong-calls コマンドライン オプションが指定されていても、常に絶対アドレス CALL 命令を使って関数を呼び出します。

noinline

その関数をインライン展開の対象外にします。

nomips16

たとえ -mips16 コマンドライン オプションが指定された変換ユニットであっても、コンパイルの際に関数のコードを常に MIPS32® 命令セットで生成します。

nonnull (index, ...)

その関数に対する 1 つまたは複数のポインタ引数は非 NULL でなければならないという事をコンパイラに指示します。-Wnonnull コマンドライン オプションが指定されている場合、非 NULL 引数に NULL ポインタが渡されると警告が出力されます。

nonnull 属性に引数を何も指定しない場合、その関数の全てのポインタ引数は非 NULL である事が必要です。

noreturn

その関数は決して戻らないという事をコンパイラに知らせます。これにより、関数が戻った場合の挙動を考慮せずに最適化できるため、状況によってはコンパイラが生成する関数呼び出しコードの効率が向上します。noreturn として宣言された関数の戻り値の型は常に void である事が必要です。

optimize

optimize 属性を使う事で、ソースファイル内の個々の関数に異なる最適化オプションを指定する事が可能となります。引数には数字または文字列が使えます。数字は最適化レベルとして解釈されます。O で始まる文字列は最適化オプションとして解釈されます。例えば、ソースファイル内の頻繁に実行する関数にだけ実行速度を優先した (コードサイズは大きくなる) より積極的な最適化オプションを適用する事ができます。

```
int __attribute__((optimize("-O3"))) pandora (void)
{
  if (maya > axton) return 1;
  return 0;
}
```

pure

関数が戻り値以外に副作用を持たず、その戻り値がパラメータおよび / または (不揮発性の) グローバル変数にのみ依存する場合、コンパイラはその関数の呼び出しに関連するコードをより積極的に最適化できます。そのような関数は pure 属性を使って指定できます。

ramfunc

関数を、それがデータメモリ内にあったかのように扱います。実行コードでは、その関数を適切に整列された最も高いアドレスに割り当てます。ramfunc のアラインメント (配置) 要件により、address 属性を ramfunc 属性と一緒に使う事はできません。ramfunc セクションが存在すると、データメモリからのコードを実行するために、リンカはバスマトリクスを適切に初期化する crt0.S スタートアップ コード向けに必要なシンボルを生成します。

 © 2015 Microchip Technology Inc.

この属性は far/longcall および section 属性と一緒に使います。以下に例を示します。

```
__attribute__((ramfunc,section(".ramfunc"),far,unique_section))
unsigned int myramfunct (void_
  { /* code */ }
```

sys/attribs.h ヘッダファイル内のマクロは、ramfunc 属性の使用を容易にします。

```
#include <sys/attribs.h>
__longramfunc__ unsigned int  myramfunct (void)
  { /* code */ }
```

section("name")

この属性を持つ関数は指定された名前を持つセクションに配置されます。

以下に例を示します。

```
void __attribute__ ((section (".wilma"))) baz () {return;}
```

関数 baz はセクション .wilma 内に配置されます。

-ffunction-sections コマンドライン オプションは、section 属性付きで定義された関数に対して効力を持ちません。

unique_section

この属性を持つ関数は、-ffunction-sections が指定されたかのように、一意に命名されたセクション内に配置されます。関数が section 属性も持つ場合、そのセクション名は生成される一意セクション名の接頭辞として使われます。

以下に例を示します。

```
void __attribute__ ((section (".fred"), unique_section) foo (void)
{return;}
```

関数 foo はセクション .fred.foo 内に配置されます。

unused

この属性は、その関数がプログラム内で使われないという事をコンパイラに知らせます。コンパイラは、その関数が使われていなくても警告を出力しません。

used

たとえコンパイラがその関数への参照を検出できなくても、常にその関数のコードを生成するようコンパイラに指示します。この属性は、インライン アセンブリだけが静的関数を参照している場合等に使います。

vector (num)

指定した例外ベクタ (num) に、その関数をターゲットとする分岐命令を配置します。**第 13 章「割り込み」**と **13.5「例外ハンドラ」**を参照してください。

warn_unused_result

この属性を持つ関数の戻り値が呼び出し元で使われていない場合、警告を出力します。

weak

弱い (weak) シンボルとは、同じシンボルの別のバージョンが存在する場合に別のバージョンの方が使われるという事を意味します。例えば、ライブラリ関数と同名のユーザ関数が存在すれば、ユーザ関数を優先して使う事ができます。

12.3 関数コードのメモリ割り当て

C/C++ 関数のコードは、通常ターゲット デバイスのプログラム フラッシュメモリ内に配置されます。

__ramfunc__ および __longramfunc__ マクロを使う事で、関数をフラッシュではなくは RAM 内に配置して RAM から実行できます。

RAM 内関数として指定された関数はスタートアップ コードによって RAM にコピーされ、それらの関数に対する呼び出しは全て RAM 内のアドレスを参照します。RAM 内関数は、プログラムメモリ内に配置された関数とは異なる 512MB メモリセグメント内に格納されます。従って、RAM 外の関数から呼び出される RAM 内関数には longcall属性を適用する必要があります。__longramfunc__マクロはRAM内に関数を配置して longcall 属性を適用します[1]。

```
#include <sys/attribs.h>
/* function 'foo' will be placed in RAM */
void __ramfunc__ foo (void)
{
}

/* function 'bar' will be placed in RAM and will be invoked
   using the full 32 bit address */
void __longramfunc__ bar (void)
{
}
```

12.4 既定値の関数メモリ割り当てを変更する

C/C++ 関数のアセンブリコードは、絶対アドレスに配置できます。これには address 属性を使って関数の仮想アドレスを指定します (**8.12「変数属性」** 参照)。

関数をユーザ定義セクション内に配置する事によって特定位置に配置し、このセクションを適切なアドレスでリンクする事もできます (**8.12「変数属性」** 参照)。

12.5 関数のサイズ制限

関数のサイズに関する理論的な制限はありません。

1. __longramfunc__ は、__ramfunc__ および __longcall__ の両方を指定した場合と機能的に等価です。

 © 2015 Microchip Technology Inc.

12.6 関数のパラメータ

MPLAB XC は、決められた方法で引数を関数に渡します。この方法は引数のサイズと数によって異なります。

> **Note:** しばしば「引数」と「パラメータ」は同じ意味で使われますが、一般的に「引数」は関数に渡される実際の値を指し、「パラメータ」は引数を保持するために関数によって定義される変数を指します。

スタックポインタは常に 8 バイト境界に配置されます。

- 32 ビット整数よりも小さい全ての整数型は最初に 32 ビット値に変換されます。引数の先頭の 4x32 ビットはレジスタ `a0` ～ `a3` を介して渡されます (各データ型に必要なレジスタの数については ?12-1 参照)。
- 一部の引数はレジスタで渡されますが、全ての引数に対してスタック上の空間が割り当てられます (?12-1 参照)。アプリケーション コードでは、現在の引数にスタック上の空間が割り当てられていても、その引数が必ずスタック上にあると想定すべきではありません。
- 関数の呼び出し動作は以下の通りです。
  - レジスタ `a0` ～ `a3` を使って引数を関数に渡します。これらのレジスタ内の値は関数呼び出しを越えて保存されません。
  - レジスタ `t0` ～ `t7` および `t8` ～ `t9` は呼び出し元の退避レジスタです。関数呼び出しは、これらの値をスタックにプッシュする事でレジスタ値を退避させる必要があります。
  - レジスタ `s0` ～ `s7` は呼び出される側の退避レジスタです。呼び出される関数は、関数内で変更するこれらのレジスタ値を退避させる必要があります。
  - レジスタ `s8` は、最適化によってフレームポインタとして使われない場合、退避レジスタとして使われます。そうではない場合、`s8` は予約済みレジスタとなります。
  - レジスタ `ra` は関数呼び出しの戻りアドレスを格納します。

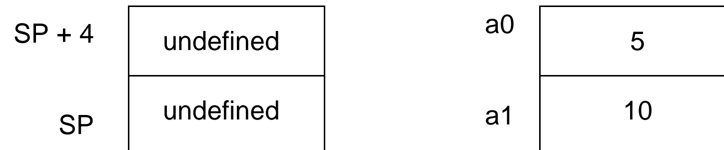
表 12-1: 必要なレジスタ

データ型	必要レジスタ数
`char`	1
`short`	1
`int`	1
`long`	1
`long long`	2
`float`	1
`double`	1
`long double`	2
`structure`	構造体のサイズに応じて最大 4

図 12-1: 引数の渡し方

Example 1:

```
int add (int, int)
a= add (5, 10);
```

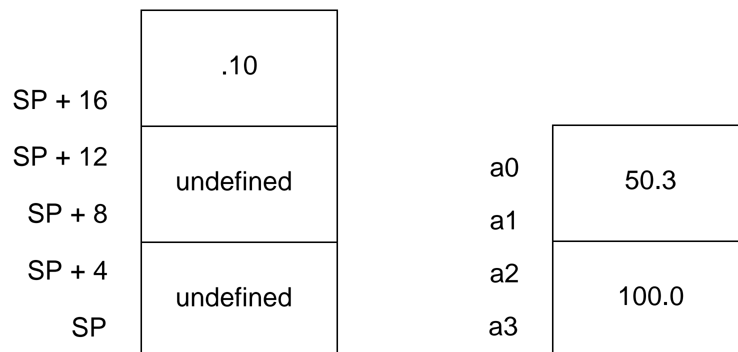


Example 2:

```
void foo (long double, long double)
call= foo (10.5, 20.1);
```

Example 3:

```
void calculate (long double, long double, int)
calculate (50.3, 100.0, .10);
```



© 2015 Microchip Technology Inc.

12.7 関数の戻り値

関数の戻り値はレジスタを介して返されます。

整数値またはポインタ値はレジスタ v0 に格納されます。

構造体または共用体そのもの (それらのオブジェクトを指すポインタではない) を返す関数は、このオブジェクトを呼び出し元が予約したメモリ領域にコピーします。関数呼び出し時に、呼び出し元はこのメモリ領域のアドレスをレジスタ $4 で渡します。関数はレジスタ v0 を介して戻りオブジェクトへのポインタを返します。呼び出し元が戻りオブジェクトの空間を提供する事により、再入が可能となります。

12.8 関数の呼び出し

既定値では、呼び出し (jal) 命令を単純に使って関数を呼び出します。この場合、256MB セグメント内のデスティネーションを呼び出す事ができます。関数に属性またはコマンドライン オプションを適用する事により、無制限の長さの呼び出しが可能になります。

-mlong-calls オプション (**5.9.1「PIC32 専用オプション」**参照) を指定すると、レジスタ形式の呼び出しが既定値により採用されます。生成されるコードは長くなりますが、呼び出しのデスティネーション アドレスは制限されません。

関数定義で属性 longcall または far を使う事で、その関数に対して常に長い呼び出しシーケンスを適用できます。関数定義で near 属性を使うと、たとえ -mlong-calls オプションが指定されていても、その関数に対して短い直接呼び出しを適用できます。

12.9 関数のインライン展開

関数を inline 宣言すると、コンパイラはその関数のコードを呼び出し元のコード内に統合します。通常これによって関数呼び出しのオーバーヘッドが除去され、実行速度が向上します。加えて、実際の引数の中に定数がある場合、コンパイル時にコードを単純化できます (インライン展開する関数のコードを一部省略できます)。コードサイズに対する影響を予測する事は比較的困難です。関数をインライン展開する事によってマシンコードのサイズが増加する場合もあれば、減少する場合もあります。

> **Note:** 関数のインライン展開は、その関数の定義 (プロトタイプだけではない) が可視である場合にのみ適用されます。1 つの関数を複数のソースにインライン展開するには、それらのソースファイルがインクルードするヘッダファイル内にその関数の定義を含める必要があります。

関数のインライン展開を宣言するには、以下のようにinlineキーワードを使います。

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

-traditional または -ansi オプションを使っている場合、inline の代わりに __inline__ を使います。コマンドライン オプション -finline-functions を使うと、全ての「十分に単純な」関数をインライン展開する事もできます。コンパイラは、そのようにインライン展開する価値のある「十分に単純な」関数を、関数サイズの予測に基づく発見的方法で判別します。

> **Note:** inline キーワードは、-finline を指定した場合または最適化を有効にした場合にのみ認識されます。

関数定義内で varargs、alloca、可変サイズデータ、計算型 goto、非ローカル goto 等を使っている関数は、インライン展開に適さない場合があります。コマンドライン オプション -Winline を指定すると、inline 宣言された関数がインライン展開できなかった場合に警告が出力され、その原因が示されます。

コンパイラ構文では、inline キーワードは関数のリンクに影響しません。

inline と static の両方が宣言された関数への呼び出しが全て呼び出し元に統合され、関数のアドレスが全く使われない場合、その関数本体のアセンブリコードは一切参照されません。この場合、-fkeep-inline-functions コマンドライン オプションを指定しない限り、コンパイラは関数本体のアセンブリコードを生成しません。各種の理由により、一部の呼び出し (例 : 関数定義よりも前の呼び出し、関数定義内の再帰呼び出し等) は統合できません。未統合の呼び出しが存在する場合、関数は通常通りにアセンブリコードへコンパイルされます。プログラムが関数のアドレスを参照する場合も、通常通りにコンパイルされます (そのような関数はインライン展開できないため)。コンパイラは、全ての呼び出しの前に定義されている static 宣言された inline 関数だけをコンパイルから除外します。

inline関数がstaticではない場合、コンパイラはその関数が他のソースファイルから呼び出される可能性があると見なします。グローバル シンボルはプログラム全体で一度しか定義できない (同じ関数が他のソースファイル内で定義されていてはならない) ため、他のソースファイルにおける呼び出しは統合できません。従って非 static のインライン関数は常にコンパイルされます。

関数定義で inline と extern の両方を指定した場合、その定義はインライン展開向けにのみ使われます。たとえ関数のアドレスを明示的に参照していても、関数本体がコンパイルされる事は決してありません。そのようなアドレスは、関数を宣言だけして未定義である場合と同様に、外部参照となります。

© 2015 Microchip Technology Inc.

`inline` と `extern` の組み合わせはマクロに似た効果を持ちます。例えば、これらのキーワードを指定した関数定義をヘッダファイルに含め、`inline` と `extern` を指定しないコピーの定義をライブラリ ファイルに含めます。ヘッダファイル内の定義により、その関数に対するほとんどの呼び出しはインライン展開されます。インライン展開されなかった呼び出しがある場合、それらはライブラリ内のコピーを参照します。

NOTE:

© 2015 Microchip Technology Inc.

第 13 章　割り込み

13.1　はじめに

割り込み処理は、大部分のマイクロコントローラ アプリケーションにおいて重要です。割り込みを使うと、ソフトウェア動作をイベントにリアルタイムに同期させる事ができます。割り込みイベントが発生するとソフトウェア実行の通常のフローが保留され、そのイベントを処理するための関数が呼び出されます。割り込み処理が完了すると、割り込み前のコンテクスト情報が復元され、通常の実行フローが再開します。

PIC32 は、内部および外部要因からの複数の割り込みをサポートします。デバイスは、割り込みの処理中に発生したより高優先度の割り込みを優先して処理する事を可能にします。

本コンパイラは、C/C++ またはインライン アセンブリコード内の割り込み処理をフルサポートします。本章には割り込み処理の概要を記載しています。

- 割り込み動作
- 割り込みサービスルーチンの書き方
- ハンドラ関数に例外ベクタを割り当てる
- 例外ハンドラ
- 割り込みサービスルーチンのコンテクスト スイッチング
- レイテンシ
- 割り込みのネスト
- 割り込みの有効化 / 無効化
- ISR に関する注意点

13.2　割り込み動作

本コンパイラは、C/C++ コードによる割り込み処理をフルサポートします。多くの場合、割り込み関数は「割り込みハンドラ」または「割り込みサービスルーチン (ISR)」と呼ばれます。

各割り込み要因には SFR 内の制御ビットが割り当てられており、このビットを使って割り込み要因を有効または無効にできます。各デバイスの割り込み処理に関する詳細は、データシートに記載されています。

「割り込みコード」とは、割り込みが発生した結果として実行される全てのコードを指します。割り込みコードは、割り込み命令からの戻りが実行された時点で完了します。これはメインライン コードと大きく異なります。メインライン コードとは、独立したアプリケーションにおいて通常リセット後に実行される主幹プログラムです。

© 2015 Microchip Technology Inc.

13.3 割り込みサービスルーチンの書き方

割り込みハンドラ関数は、割り込みからの戻り時にプログラムのコンテクストを維持するためにコンテクストを退避 / 復元するという点で、通常の関数とは異なります。これらの関数からの戻りに使われるコード シーケンスも、通常の関数とは異なります。

各種の属性を使う事で、コンパイラに適正な ISR コードを生成させる事ができます。これを容易にするためのマクロも提供されます。これらのマクロについては後述します。

割り込みサービスルーチンを生成するために、コンパイラは各種の動作を実行する必要があります。コンパイラには、通常の関数とは異なる形態の戻りコードを使うよう指示する必要があります。関数を割り込みベクタにリンクする必要もあります。全ての ISR は MIPS32®r2 または microMIPS™ ISA モードを使う必要があります。各割り込み関数には `nomips16` 関数属性を適用します。

> **Note:** 複数の命令セット アーキテクチャ (ISA) モードをサポートするデバイスは、例外 / 割り込みに使うモードを指定するためのコンフィグレーション ビットを備えている場合があります。割り込み時に microMIPS ISA を使うようターゲット デバイスが設定されている場合、割り込み関数には `micromips` 関数属性を適用する必要があります。ターゲット デバイスが代替の ISA モードによる例外 / 割り込み処理をサポートするかどうかについては、データシートを参照してください。

割り込み関数は `void` 型として宣言する必要があり、パラメータを持つ事はできません。これが割り込み関数向けに有効な唯一の関数プロトタイプです。なぜならば、割り込み関数はソースコード内で直接呼び出される事が一切ないためです。

割り込み関数を C/C++ コードから直接呼び出さない事が必要です (使われる戻り命令が異なるため)。しかし、割り込み関数から他の関数 (ユーザ関数とライブラリ関数) を呼び出す事はできます。その場合、追加のレジスタが使われ、それらはコンテクスト スイッチコードによって退避 / 復元する必要があるという事に注意が必要です。

関数は、`interrupt` 属性または `# pragma interrupt` によって割り込みハンドラ関数 (割り込みサービスルーチン (ISR) とも呼ぶ) として識別されます[1]。どちらを使っても機能的には等価ですが、一般的に用いられている `interrupt` 属性を推奨します。割り込みは、特定優先度のハンドリング割り込みとして指定するか、シングルベクタ モードでの動作向けに指定します。

特定のハンドラを持たない全ての割り込みベクタ向けに、既定値割り込みハンドラが実装されます。これは libpic32.a ライブラリによって提供され、デバッグ ブレークポイントを生成してデバイスをリセットします。アプリケーションは、`_DefaultInterrupt` の名前を持つ割り込み関数を宣言する事により、この既定値ハンドラに代わるアプリケーション専用の既定値割り込みハンドラを提供できます。

13.3.1 interrupt 属性

```
__attribute__((interrupt([IPLn[SRS|SOFT|AUTO]])))
```

n は 0 ～ 7 の値です。

この `interrupt` 属性により、その関数が割り込みハンドラである事を示します。この属性を指定すると、コンパイラは割り込みハンドラに適した関数開始 / 終了シーケンスを生成します。生成されたコードは、シャドー レジスタセット (`SRS`) または生成されたソフトウェア命令 (`SOFT`) を使ってコンテクストをスタックにプッシュする事により、コンテクストを保存します。`interrupt` 属性に関しては例 13-1 を参照してください。

1. プリプロセッサ マクロはプラグマ ディレクティブ内で展開されないという事に注意してください。

© 2015 Microchip Technology Inc.

> **Note:** 一部の PIC32 では、デバイス コンフィグレーション ビット (BOOTISA) を使って、例外 / 割り込みコードを MIPS32® モードまたは microMIPS™ ISA モードのどちらかに指定できます。これらのデバイスで BOOTISA ビットを microMIPS モードに設定した場合、割り込み関数に `micromips` 属性を追加する必要があります。これらのデバイスで BOOTISA ビットを MIPS32 モードに設定した場合、割り込み関数に `nomicromips` 属性を追加する必要があります。このコンフィグレーション ビットの詳細はデバイス データシートを参照してください。

例 13-1: `interrupt` 属性

```
void __attribute__((interrupt(IPL7SRS))) bambam (void);
```

多くの PIC32 デバイスでは、コンフィグレーション ビットの設定により、どの割り込み優先度にシャドー レジスタセットを使うのか指定できます (例 : `#pragma config` FSRSSEL=PRIORITY_7)。PIC32 ターゲット デバイスがこの機能をサポートしているかどうかは、データシートを参照してください。これは、各割り込みハンドラにどちらのコンテクスト退避方式を適用するのか指定する必要がある事を意味します。割り込み優先度 (IPL) 指定子として `IPLnSRS` を指定した場合、コンパイラはシャドーレジスタによるコンテクスト退避を使って割り込み関数プロローグおよびエピローグ コードを生成します。`IPLnSOFT` を指定した場合、ソフトウェアによるコンテクスト退避を使います。

上記のデバイス以外に、PIC32 には 8 個のレジスタセット (1x 標準レジスタセット + 7x シャドー レジスタセット) を備えた製品もあります。この場合、全ての割り込み優先度に対して十分な数のシャドー レジスタが存在します。従って、全ての割り込みサービス ルーチンに対して `IPLnSRS` IPL 指定子を使います。

> **Note:** アプリケーション コードは、ハンドラ ルーチンに適合する正しい値を適用する必要があります。

コンパイラは `IPLnAUTO` IPL 指定子もサポートします。この指定子を使った場合、コンパイラは `SRSCTL` 内のランタイム値に基づいて、どちらのコンテクスト退避コード (`SOFT` または `SRS`) を使うのか決定します。`interrupt()` 属性で IPL 指定子を省略した場合、コンパイラは既定値の `IPLnAUTO` を使います。

シャドー レジスタセットによる割り込みコンテクストの退避をサポートしないデバイスでは、全ての割り込みハンドラに対して `IPLnSOFT` を適用する必要があります。

> **Note:** `SOFT` ではレジスタをスタックに退避するため、`SRS` に比べてレイテンシが長くなります。`AUTO` の場合、`SRS` と `SOFT` のどちらを使うべきか評価するために、レイテンシが数サイクル増加します。

13.3.2 # pragma interrupt

> **Note:** # pragma interrupt は、他のコンパイラからコードを移植する際の互換性維持のためだけに使います。割り込みサービスルーチンを新たに書く場合、より一般的な interrupt 関数属性の使用を推奨します。

 # pragma interrupt *function-name* IPL*n*[AUTO|SOFT|SRS] [**vector** [**@**]*vector-number* [, *vector-number-list*]]
 # pragma interrupt *function-name* **single** [**vector** [**@**] **0**]

n は 0 ～ 7 の値です。

IPL*n* [AUTO|SOFT|SRS] IPL 指定子は、小文字のみまたは大文字のみで書く必要があります。

この後で、# pragma interrupt で指定した割り込みハンドラ関数を定義する必要があります (プラグマ自体と同じ変換ユニット内で定義)。

interrupt 属性も、その関数定義が割り込みハンドラである事を示します。これは # pragma interrupt と機能的に等価です。

下の 2 つ例は、どちらも関数 foo をソフトウェアによる コンテクスト退避を使う優先度 4 の割り込みハンドラ関数として定義します。

```
#pragma interrupt foo IPL4SOFT
void foo (void)
```

上記は下記と機能的に等価です。

```
void __attribute__ ((interrupt(IPL4SOFT))) foo (void)
```

13.3.3 __ISR マクロ

<sys/attribs.h> ヘッダファイルは、割り込み関数に対する属性の適用を容易にするためのマクロを提供します。プロセッサ ヘッダファイルはベクタマクロ定義も含みます (コンパイラの /pic32mx/include/proc ディレクトリ内の対応するヘッダファイル参照)。

- __ISR(V, IPL)
- __ISR_AT_VECTOR(v, IPL)
- 割り込みベクタマクロ

> **Note:** 一部の PIC32 では、デバイス コンフィグレーション ビット (BOOTISA) を使って、例外 / 割り込みコードを MIPS32® モードまたは microMIPS™ ISAモードのどちらかに指定できます。これらのデバイスでBOOTISAビットを microMIPS モードに設定した場合、割り込み関数に micromips 属性を追加する必要があります。BOOTISA ビットを MIPS32 モードに設定した場合、割り込み関数に nomicromips 属性を追加する必要があります。このコンフィグレーション ビットの詳細はデバイス データシートを参照してください。

13.3.3.1 __ISR(V, IPL)

__ISR(v, IPL) マクロは、ベクタ番号 v を割り込み優先度 IPL に割り当てます。これにより、指定したベクタ位置に割り込みハンドラへのジャンプを配置します。PIC32 の場合、割り込みハンドラは MIPS32 命令セットを使う必要があるため、このマクロは nomips16 属性も適用します。

例 13-2: コアタイマベクタ、IPL2SOFT

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_TIMER_VECTOR, IPL2SOFT) CoreTimerHandler(void);
```

 © 2015 Microchip Technology Inc.

例 13-2 は、優先度 2 のコアタイマ割り込み向けに割り込みハンドラ関数を作成します。コンパイラは対応するベクタ位置にディスパッチ関数を配置します。この関数が実行されるには、コアタイマ割り込みフラグおよびイネーブルビットがセットされ、割り込み優先度が 2 に設定されている必要があります。コンパイラは、このハンドラ関数向けにソフトウェアによるコンテクスト退避コードを生成します。

例 13-3: コア ソフトウェア 0 ベクタ、IPL3SRS

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_SOFTWARE_0_VECTOR,IPL3SRS)
CoreSoftwareInt0Handler(void);
```

例 13-3 は、優先度 3 のコア ソフトウェア割り込み 0 向けに割り込みハンドラ関数を作成します。コンパイラは、対応するベクタ位置にディスパッチ関数を配置します。この関数が実行されるには、コア ソフトウェア割り込みフラグおよびイネーブルビットがセットされ、割り込み優先度が 3 に設定されている必要があります。デバイス コンフィグレーション ヒューズは、シャドー レジスタセット 1 を割り込み優先度 3 に割り当てる必要があります。コンパイラは、レジスタ コンテクストが SRS1 に退避されると想定してコードを生成します。

例 13-4: コア ソフトウェア 1 ベクタ、IPL0AUTO

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_SOFTWARE_1_VECTOR, IPL0AUTO)
CoreSoftwareInt1Handler(void);
```

例 13-4 は、優先度 0 のコア ソフトウェア割り込み 1 向けに割り込みハンドラ関数を作成します。コンパイラは、対応するベクタ位置にディスパッチ関数を配置します。この関数が実行されるには、コア ソフトウェア割り込み 1 フラグおよびイネーブルビットがセットされ、割り込み優先度が 0 に設定されている必要があります。コンパイラは、実行時にソフトウェアによるコンテクスト退避が必要かどうか判断するコードを生成します。

例 13-5: コア ソフトウェア 1 ベクタ、既定値

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_SOFTWARE_1_VECTOR) _CoreSoftwareInt1Handler(void);
```

例 13-5 は機能的に例 13-3 と等価です。IPL 指定子が省略されているため、コンパイラは `IPL0AUTO` と見なします。

13.3.3.2 __ISR_AT_VECTOR(v, IPL)

`__ISR_AT_VECTOR(v, IPL)` は割り込みハンドラ全体をベクタ位置 `v` に配置し、それをソフトウェア割り当て割り込み優先度 `IPL` に割り当てます。アプリケーションコードは、ハンドラのサイズに適合するようにベクタ間隔を設定する必要があります。ISR 関数は MIPS32 命令セットを使う必要があるため、このマクロは `nomips16` 属性も適用します。

例 13-6: コアタイマベクタ、IPL2SOFT

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR_AT_VECTOR(_CORE_TIMER_VECTOR, IPL2SOFT)
CoreTimerHandler(void);
```

例 13-6 は、優先度 2 のコアタイマ割り込み向けに割り込みハンドラ関数を作成します。コンパイラは、割り込みハンドラ全体をベクタ位置に配置します。これはディスパッチ関数を使いません。この関数が実行されるには、コアタイマ割り込みフラグおよびイネーブルビットがセットされ、割り込み優先度が 2 に設定されている必要があります。コンパイラは、このハンドラ関数向けにソフトウェアによるコンテクスト退避コードを生成します。

13.3.3.3　割り込みベクタマクロ

各プロセッサ サポート ヘッダファイルは、各割り込みベクタ番号に対して 1 つのマクロを提供します (例：`/pic32mx/include/proc/p32mx360f512l.h`)、コンパイラ インストール ディレクトリ内の対応するヘッダファイル参照)。これらのマクロを `sys\attribs.h` ヘッダファイルが提供する `__ISR()` マクロと一緒に使う事で、割り込みサービスルーチンの作成と管理がより容易になります。

例 13-7: ヘッダによる割り込みベクタ

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR (_TIMER_1_VECTOR, IPL7SRS) Timer1Handler (void);
```

例 13-7 は、優先度 7 の Timer1 割り込み向けに割り込みハンドラ関数を作成します。コンパイラは、デバイス専用ヘッダファイル内の定義に従い、`_TIMER_1_VECTOR` に割り当てられたベクタ位置にディスパッチ関数を配置します。この関数が実行されるには、Timer1 割り込みフラグおよびイネーブルビットがセットされ、割り込み優先度が 7 に設定されている必要があります。シャドーレジスタを特定の IPL 値に割り当てる事ができるデバイスでは、デバイス コンフィグレーション ビットでシャドー レジスタセット 1 を割り込み優先度 7 に割り当てる必要があります。コンパイラは、レジスタ コンテクストが SRS1 に退避されると想定してコードを生成します。

© 2015 Microchip Technology Inc.

13.4 ハンドラ関数に例外ベクタを割り当てる

PIC32 デバイスでは、デバイス データシートの定義に従って、各割り込み要因が例外ベクタにマッピングされます。固定ベクタ間隔のデバイスでは、例外要因に対応するハンドラ関数へのディスパッチ関数を配置するために、各ベクタアドレスに対して既定値 (4 ワード) の間隔が確保されます。可変ベクタ間隔のデバイスでは、既定値リンカスクリプトが割り込み関数のサイズに合わせて各ベクタの間隔を調整します。

割り込みハンドラ関数は、例外ベクタアドレスに配置されたディスパッチ関数のターゲットとして割り込みベクタに対応付ける事も、例外ベクタアドレスに直接配置する事もできます。また、1 つのハンドラ関数を複数のディスパッチ関数のターゲットにする事ができます。

これには、関数宣言で 1 つの `vector` 属性を使って、そのハンドラ関数を 1 つまたは複数の例外ベクタアドレスに対応付けます。互換性のために、`# pragma interrupt` の `vector` 節、独立した `# pragma vector`、関数宣言での `vector` 属性のいずれかを使ってハンドラ関数をベクタアドレスに対応付ける事もできます。

13.4.1 vector 属性

1 つのハンドラ関数は、1 つの属性を使って 1 つまたは複数の例外ベクタアドレスに対応付ける事ができます。`at_vector` 属性は、ハンドラ関数自体を例外ベクタアドレスに配置します。`vector` 属性は、制御をハンドラ関数に渡すためのディスパッチ関数を例外ベクタアドレスに配置します。

以下の宣言は、関数 `foo` を優先度 4 の割り込みハンドラ関数として指定します。`foo` は、例外ベクタアドレス 54 に配置されます。

```
void __attribute__ ((interrupt(IPL4SOFT))) __attribute__
((at_vector(54))) foo (void)
```

下の宣言も、関数 `foo` を優先度 4 の割り込みハンドラ関数として指定します。この例では、`foo` をターゲットとするディスパッチ関数を例外ベクタアドレス 52 および 53 に配置します。

```
void __attribute__ ((interrupt(IPL4SOFT))) __attribute__
((vector(53, 52))) foo (void)
```

ベクタアドレスに直接配置したハンドラ関数の方が実行は高速です。しかし、ベクタ間隔の調整には制限があるため、大きなハンドラ関数を直接配置した場合、後続のベクタ位置にオーバーラップしてそれらが使えなくなる可能性があります。そのような場合、ディスパッチ関数を使った方が安全です。

13.4.2 # pragma interrupt の vector 節

> **Note:** # pragma interrupt とその vector 節は、他のコンパイラからコードを移植する際の互換性のためだけに使います。新たにハンドラ関数を例外ベクタアドレスに割り当てる場合、vector 関数属性を使う事を推奨します。

pragma interrupt では、優先度指定子の後で vector 節を任意に指定できます。

 # pragma interrupt *function-name IPL-specifier* [**vector**
[**@**]*vector-number* [, *vector-number-list*]]

指定されたハンドラ関数をターゲットとするディスパッチ関数は、指定されたベクタ番号の例外ベクタアドレスに配置されます。最初のベクタ番号の前に「@」を付けた場合、ハンドラ関数がそこに直接配置されます。

以下のプラグマは、関数 foo を優先度 4 の割り込みハンドラ関数として指定します。この場合、foo 自体が例外ベクタアドレス 54 に配置され、foo をターゲットとするディスパッチ関数が例外ベクタアドレス 34 に配置されます。

```
#pragma interrupt foo IPL4AUTO vector @54, 34
```

以下のプラグマは、関数 bar を優先度 5 の割り込みハンドラ関数として指定します。barは汎用プログラムメモリ(.textセクション)に配置されます。barをターゲットとするディスパッチ関数は例外ベクタアドレス 23 に配置されます。

```
#pragma interrupt bar IPL5SOFT vector 23
```

13.4.3 # pragma vector

> **Note:** # pragma vector は、他のコンパイラからコードを移植する際の互換性のためだけに使います。新たにハンドラ関数を例外ベクタアドレスに割り当てる場合、vector 関数属性を使う事を推奨します。

pragma vector は、指定された関数をターゲットとする 1 つまたは複数のディスパッチ関数を作成します。# pragma interrupt で指定されたターゲット関数の場合、これは vector 節が使われたかのように機能します。どの関数でも # pragma vector のターゲット関数になれます (アセンブリコードやその他によって実装された外部関数を含む)。

 # pragma vector *function-name* **vector** *vector-number* [,
vector-number-list]

以下のプラグマは、foo をターゲットとするディスパッチ関数を例外ベクタアドレス 54 で定義します。

```
#pragma vector foo 54
```


© 2015 Microchip Technology Inc.

13.5 例外ハンドラ

PIC32 は、非割り込み型の例外向けに例外ベクタも備えています。これらの例外はブートストラップ例外と一般的例外に分類されます。

13.5.1 ブートストラップ例外

ブートストラップ コードの実行中 ($Status_{BEV}$=1) に生じる全ての例外はリセット例外です。全てのリセット例外は 0xBFC00380 へジャンプします。

32 ビット ツールチェーンは、この位置に関数 _bootstrap_exception_handler() をターゲットとする分岐命令を配置します。標準ライブラリでは、この関数の既定値である「弱い」バージョンが提供され、これは単にソフトウェア リセットを生成します。インサーキット デバッグまたはエミュレーション向けにコンパイルする場合、_bootstrap_exception_handler の既定値実装では、最初にソフトウェア ブレークポイントが発生した後にソフトウェア リセットが発生します。ユーザ アプリケーションが _bootstrap_exception_handler() を実装する場合、そちらが代わりに使われます。

13.5.2 一般的例外

一般的例外は、ブートスラップ コード以外のプログラム実行中 ($Status_{BEV}$=0) に発生する全ての非割り込み型例外です。一般的例外はオフセット 0x180 ～ EBase へジャンプします。

32 ビット ツールチェーンは、この位置に関数 _general_exception_context() をターゲットとする分岐命令を配置します。この関数の実装によりコンテクストの退避、アプリケーション ハンドラ関数の呼び出し、コンテクストの復元、例外命令からの戻りを実行します。コンテクストの退避には hi および lo レジスタと s0 ～ s8 を除く全ての汎用レジスタが使われ、それらは全ての呼び出された関数によって保存されるよう定義されるため、ここで再度退避する必要はありません。Cause および Status レジスタの値はアプリケーション ハンドラ関数 (_general_exception_handler()) に渡されます。ユーザ アプリケーションが _general_exception_context() を実装する場合、そちらが代わりに使われます。

```
void _general_exception_handler (unsigned cause, unsigned status);
```

標準ライブラリでは _general_exception_handler() の既定値である「弱い」バージョンが提供され、これは単にソフトウェア リセットを生成します。インサーキット デバッグまたはエミュレーション向けにコンパイルする場合、_general_exception_handler の既定値実装では、最初にソフトウェア ブレークポイントが発生した後にソフトウェア リセットが発生します。ユーザ アプリケーションが _general_exception_handler() を実装する場合、そちらが代わりに使われます。

13.5.3 単純な TLB リフィル例外

命令フェッチまたはデータアクセス中に、マッピングされたアドレス空間への参照に一致する TLB エントリが存在せず、ステータス レジスタ内の EXL ビットが 0 である場合、TLB リフィル例外が発生します。これは、エントリは一致したが有効ビットが OFF である場合とは異なるという事に注意が必要です。その場合、TLB 無効例外が発生します。

```
void _simple_tlb_refill_exception_handler(void);
```

_simple_tlb_refill_exception_handler() の既定値である「弱い」バージョンが提供され、これは単にソフトウェア リセットを生成します。インサーキット デバッグまたはエミュレーション向けにコンパイルする場合、_simple_tlb_refill_exception_handlerの既定値実装では、最初にソフトウェア ブレークポイントが発生した後にソフトウェア リセットが発生します。

© 2015 Microchip Technology Inc.

13.5.4 キャッシュエラー例外

キャッシュエラー例外は、命令またはデータ参照でキャッシュタグまたはデータエラーが検出された時に発生します。この例外はノンマスカブルです。キャッシュアレイ内のエラーを妨げないようにするため、例外ベクタはマッピングもキャッシュもされないアドレスを指します。この例外は正確 (precise) です。

```
void _cache_err_exception_handler(void);
```

`_cache_err_exception_handler()` の既定値である「弱い」バージョンが提供され、これは単にソフトウェア リセットを生成します。インサーキット デバッグまたはエミュレーション向けにコンパイルする場合、`_cache_err_exception_handler` の既定値実装では、最初にソフトウェア ブレークポイントが発生した後にソフトウェア リセットが発生します。

© 2015 Microchip Technology Inc.

13.6 割り込みサービスルーチンのコンテクスト スイッチング

C/C++ 関数の標準呼び出しでは zero、s0 ～ s7、gp、sp、fp レジスタが保存されます。コンパイラは k0 と k1 を使って非 GPR コンテクストを保存しますが、これらは常に自動的に (グローバル割り込み無効状態で続けて) アクセスされるため、これらを保存する必要はありません。ハンドラ関数は、標準レジスタに加えて a0 ～ a3、t0 ～ t9、v0、v1、ra レジスタを保存します。

割り込みハンドラ関数も、その関数で使うプロセッサ ステータス レジスタを退避 / 復元します。特に EPC、SR、hi、lo レジスタはコンテクストとして保存されます。利用可能な全ての DSP アキュムレータは、必要に応じて保存されます。

また、DSP アキュムレータ レジスタが保存される場合、DSP 制御レジスタも保存されます。

ハンドラ関数はシャドー レジスタセットを使って汎用レジスタを保存する事ができます。これにより、ハンドラ関数のアプリケーション コードを実行する際のレイテンシを低減できます。一部のデバイスでは、デバイス コンフィグレーション ビットの設定 (例 : #pragma config FSRSSEL=PRIORITY_6) により、シャドー レジスタセットをいずれかの割り込み優先度 (IPL) に割り当てます。その他のデバイスでは、シャドー レジスタセットは IPL7 にハードウェアで固定的に割り当てられます。シャドー レジスタセットの詳細は、ターゲット デバイスのデータシートを参照してください。

13.6.1 コンテクストの復元

ソフトウェアによって退避された全てのオブジェクトは、割り込み関数が戻る前にソフトウェアによって自動的に復元されます。コンテクストは退避時とは逆順に復元されます。

13.7 レイテンシ

割り込み要因が発生してからユーザISRコード内の最初の命令が実行されるまでのサイクル数には、以下の 2 つの要因が影響します。

- **割り込みのプロセッサ サービス** - プロセッサが割り込みを認識して割り込みベクタの最初のアドレスに分岐するのに要する時間です。この時間は、プロセッサと割り込み要因のデータ シートを参照する事で特定できます。
- **ISR コード** - コンパイラは、ISR が使用するレジスタを退避させます。さらに、ISR が通常の関数を呼び出す場合、コンパイラはワーキング レジスタを、ISR 内で明示的に使われるかどうかに関係なく全て退避させます。なぜならば、呼び出された関数がどのリソースを使うのかコンパイラには知る事ができないからです。

13.8 割り込みのネスト

割り込みはネストできます。PIC32 アーキテクチャが実装する割り込み優先度スキームにより、ユーザはどの割り込み要因が他のどの割り込み要因に割り込めるのか指定できます。割り込み動作の詳細についてはデバイス データシートを参照してください。

コンパイラが生成する割り込みサービス ルーチンのプロローグコードは、既定値により自動的に割り込みを再度有効にします。

13.9 割り込みの有効化 / 無効化

以下のビルトイン関数は、CPU 割り込みの現在の状態を調査または操作します。

```
unsigned int __builtin_get_isr_state(void)
void __builtin_set_isr_state(unsigned int)
void __builtin_disable_interrupts(void)
void __builtin_enable_interrupts(void)
```

13.10 ISR に関する注意点

割り込みサービスルーチンを書く際に注意すべき事項が何点かあります。

どのようなコンパイラでもそうですが、割り込み関数または割り込み関数から呼び出される関数が使うレジスタの数を制限する事により、コンパイラが生成および実行するコンテクスト スイッチコードを削減できます (**13.7「レイテンシ」**参照)。これには、割り込み関数を小さく単純に書く事が効果的です。

割り込みレイテンシが問題となる場合、ISR から他の関数を呼び出さないようにします。そのような関数呼び出しは、アプリケーションのメイン制御ループによって処理される不揮発性フラグに置き換える事ができます。

© 2015 Microchip Technology Inc.

第 14 章　メイン、スタートアップ、リセット

14.1　はじめに

C/C++ コードが正しく動作するには、main 関数、スタートアップ コード (変数の初期化 / クリアとレジスタおよびプロセッサのセットアップ)、リセット条件処理が必要です。

- main 関数
- スタートアップ コード
- On Reset ルーチン

14.2　main 関数

識別子 main は特殊です。これはプログラム内で最初に実行する関数の名前として使う必要があります。プログラム内には main という名前の関数が 1 つだけ存在する事が必要です。しかし、リセット後に最初に実行されるのは main 関数のコードではなく、コンパイラが提供する追加のコード (スタートアップ コードと呼ぶ) です。このコードは制御を main() 関数に渡す働きをします。

© 2015 Microchip Technology Inc.

14.3 スタートアップ コード

C/C++ プログラムが main() 関数の実行を開始する前に、特定のオブジェクトを初期化し、プロセッサを特定の状態にしておく必要があります。これを行うのがスタートアップ コードの役割です。スタートアップ コードは main() の前に実行されますが、リセット直後に特殊な初期化が必要な場合は「On Reset」機能を使う必要があります (**14.4「On Reset ルーチン」**参照)。

PIC32 のスタートアップ コードは以下を実行します。

1. NMI (ノンマスカブル割り込み) が発生した場合、NMI ハンドラ (_nmi_handler) にジャンプする。
2. スタックポインタを初期化する。
3. ターゲット デバイス上の全てのレジスタセットでグローバル ポインタを初期化する。
4. 「On Reset」プロシージャ (_on_reset) を呼び出す。
5. ターゲット デバイスが L1 キャッシュを備えている場合、L1 キャッシュを初期化するために ___pic32_init_cache プロシージャを呼び出す。
6. ターゲット デバイスがあらかじめマッピングされた EBI および SQI 外部メモリ領域を使う場合、TLB を初期化するために __pic32_tlb_init_ebi_sqi プロシージャを呼び出す。
7. 初期化されていない bss セクションをクリアする。
8. リンカが生成したデータ初期化テンプレートを使ってデータを初期化する。
9. バスマトリクスを備えたターゲット デバイス上でアプリケーションが RAM 関数を使う場合、データメモリからコードを実行するためにバスマトリクスを初期化する。
10. CP0 レジスタを初期化する。
11. ターゲット デバイスが DSPr2 エンジンを備えている場合、これを有効にする。
12. 「On Bootstrap」プロシージャ (_on_bootstrap) を呼び出す。
13. 例外ベクタの位置を変更する。
14. C++ の場合、ファイルスコープの静的ストレージ オブジェクト向けの全てのコンストラクタを呼び出すために C++ 初期化コードを呼び出す。
15. main() を呼び出す。

ランタイムモデルには以下のように対応します。

- カーネルモードのみ
- KSEG1 のみ
- RAM 関数には __ramfunc__ または __longramfunc__ 属性が与えられます (sys/attribs.h 内で定義)。つまり、全ての RAM 関数は .ramfunc セクションに割り当てられ、関数には ramfunc 属性が与えられます。

14.3.1 NMI 発生時の NMI ハンドラへのジャンプ

NMI によってリセットベクタへの移行が発生すると、NMI ハンドラ プロシージャ (_nmi_handler) へのジャンプが発生します。ERET を実行する NMI ハンドラ プロシージャの「弱い」バージョンが提供されます。スタートアップ コードは _nmi_handler 関数にジャンプするため、この関数には nomips16 属性を与える必要があります (例 : __attribute__((nomips16)))。

© 2015 Microchip Technology Inc.

14.3.2 スタックポインタとヒープの初期化

スタックポインタ (sp) レジスタは、スタートアップ コード内で初期化する必要があります。スタートアップ コードによる sp レジスタの初期化を可能にするために、リンカは KSEG0/KSEG1 データメモリの終端を指す変数を初期化する必要があります[1]。

ターゲット デバイスが L1 データキャッシュを備えている場合、リンカはスタックを KSEG0 に割り当て、そうではない場合は KSEG1 に割り当てます。

この変数の名前は _stack です。ユーザは、リンカに対してコマンドライン オプション --defsym _min_stack_size=*N* を指定する事で、スタック空間の最小割り当てサイズを変更できます。リンカスクリプトは既定値 (1024) の _min_stack_size を提供します。

同様に、アプリケーションでヒープを使う場合があります。スタートアップ コードでヒープを初期化する必要はありませんが、標準 C ライブラリ (sbrk) にヒープの位置とサイズを知らせる必要があります。リンカは、ヒープの開始位置を指定する変数を生成します。ヒープは、使用するKSEG0/KSEG1データメモリの終端に配置されます。

ターゲット デバイスが L1 データキャッシュを備えている場合、リンカはヒープを KSEG0 に割り当て、そうではない場合は KSEG1 に割り当てます。

この変数の名前は _heap です。ユーザは、リンカに対してコマンドライン オプション --defsym _min_heap_size=*M* を指定する事で、ヒープ空間の最小割り当てサイズを変更できます。ヒープのサイズをゼロに設定した状態でヒープが使われた場合の挙動は、ヒープの使用量が最小ヒープサイズを超えた場合と同じです。すなわち、ヒープはスタックに割り当てられた空間へオーバーフローします。

ヒープとスタックは、未割り当ての KSEG0/KSEG1 データメモリを使います。ヒープは KSEG0/KSEG1 データメモリの低アドレスから高アドレスに向かって進み、スタックは KSEG1 データメモリの高アドレスから低アドレス (ヒープのある側) に向かって進みます。リンカは、KSEG0/KSEG1 データメモリ領域内で最大のメモリギャップにヒープとスタックの両方を割り当てようと試みます。ヒープとスタックの要求最小サイズに対して十分な空間が得られない場合、リンカはエラーを出力します。

1. RAM 関数が存在する場合、DRM の一部をカーネル プログラム向けに構成して RAM 関数を格納する必要があります。その場合、スタックポインタは DRM カーネル プログラムの境界 (開始) アドレスの 1 ワード前に置かれます。RAM 関数が存在しない場合、スタックポインタは DRM の本来の終端に置かれます。

図 14-1: スタックとヒープのレイアウト

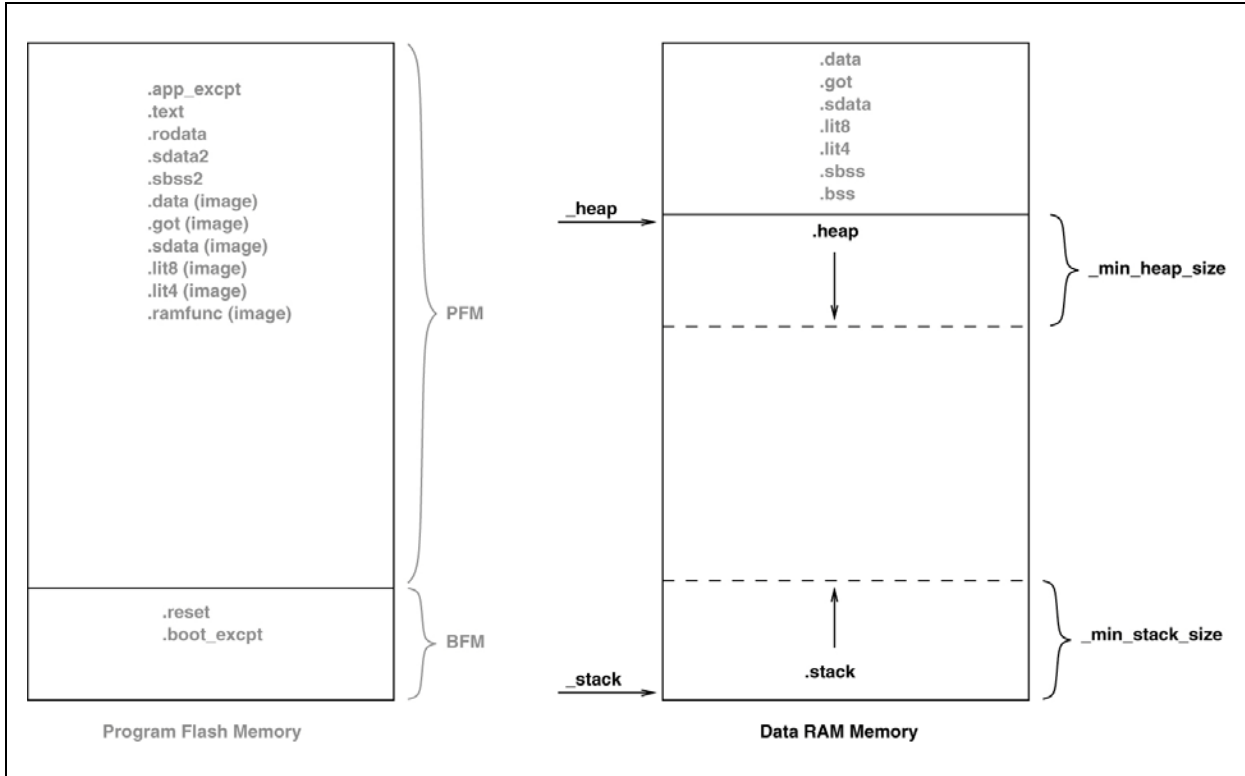
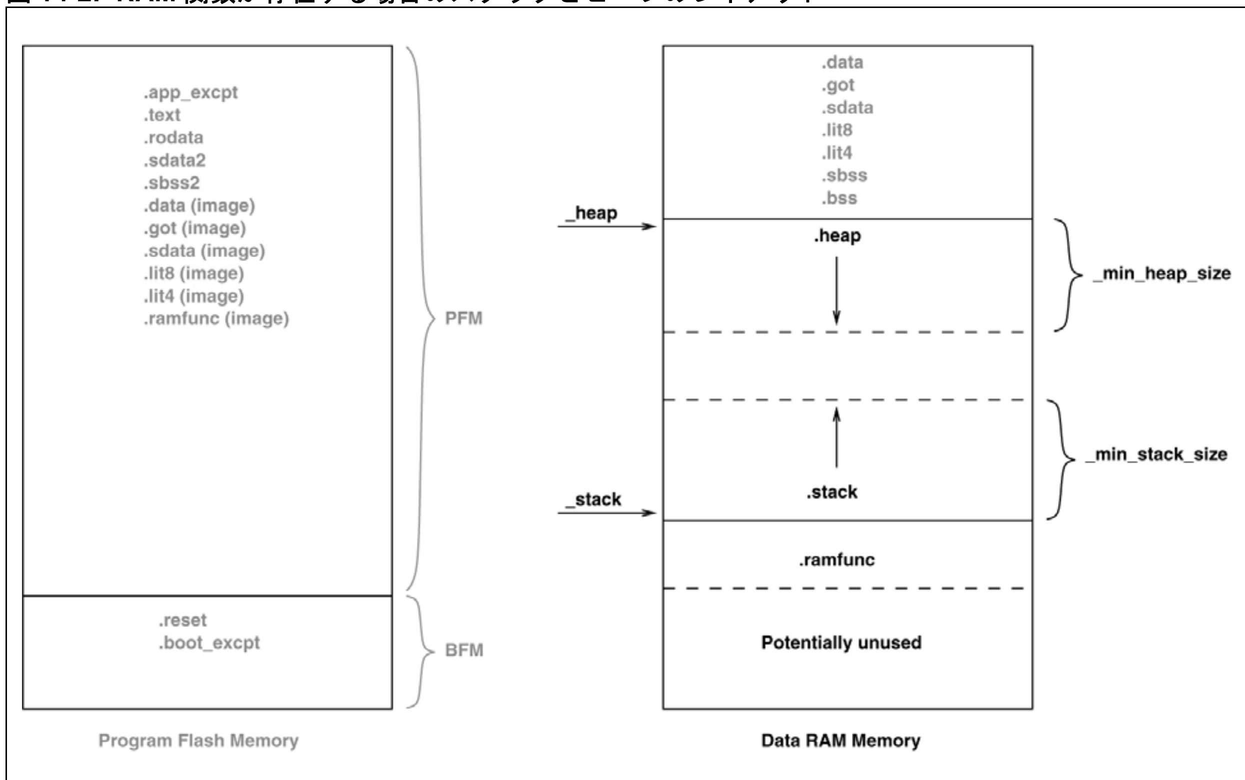


図 14-2: RAM 関数が存在する場合のスタックとヒープのレイアウト



© 2015 Microchip Technology Inc.

14.3.3 グローバル ポインタの初期化

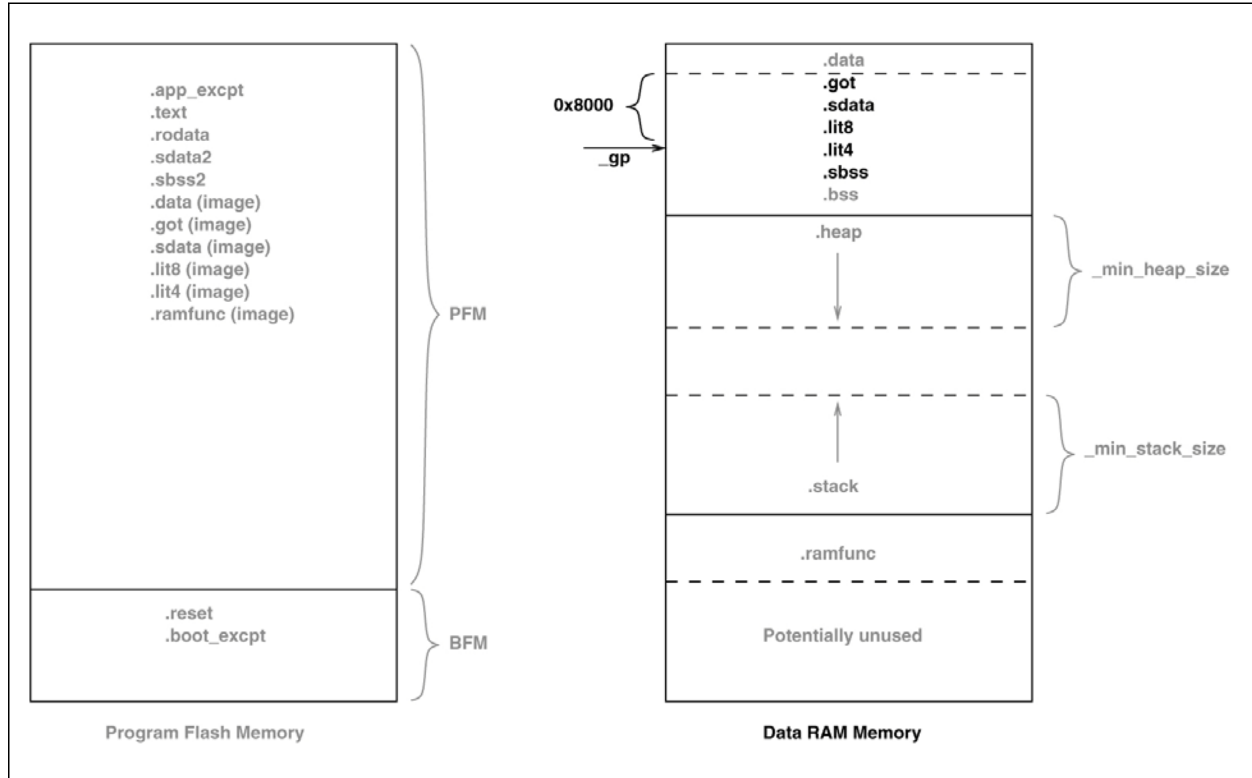
本コンパイラ ツールチェーンはグローバル ポインタ (gp) 相対アドレス指定をサポートします。gp レジスタに保存されているアドレスから ±32KB 内にあるデータに対するロード / ストアは、gp レジスタをベースレジスタとして使って 1 命令で実行できます。グローバル ポインタを使わずにスタティック メモリ領域からデータをロードするには 2 命令 (コンパイラ / リンカが計算した 32 ビット固定アドレスの最上位ビット (MSb) をロードするための命令と、データをロードするための命令) が必要です。

gp 相対アドレス指定を使うために、コンパイラとアセンブラは全ての「小さな」変数と定数を以下のいずれか 1 つのセクション内に集約する必要があります。

- .lit4.
- .sdata.
- .sdata.*
- .gnu.linkonce.s.*
- lit8
- sbss
- sbss.*
- .gnu.linkonce.sb.*

さらに、リンカは上記の入力セクションの全てを 1 つの出力セクションに集約する必要があります。この集約化は既定値のリンカスクリプトによって処理されます。スタートアップ コードは、gp レジスタがこの出力セクションの「中央」を指すように初期化する必要があります。スタートアップ コードによる gp レジスタの初期化を可能にするために、リンカスクリプトは 1 つの変数を初期化する必要があります (「小さな」変数と定数を格納した出力セクションの開始アドレスから32KB進んだアドレスを指すよう初期化)。この変数の名前は _gp です (コア リンカスクリプトと適合させるため)。グローバル ポインタは標準 GPR セットでの初期化に加えて、シャドー レジスタセットでも初期化する必要があります。

図 14-3: グローバル ポインタの位置



14.3.4 データ初期化テンプレートを使った変数と RAM 関数の初期化 / クリア

初期化されない非 auto オブジェクトは、プログラムの実行を開始する前にクリアする必要があります。これもスタートアップ コードによって実行されます。

非初期化変数は、定義内で値が代入されない非 auto オブジェクトです (下の例の output)。

```
int output;
int main(void) { ...
```

このような非初期化オブジェクトには、プログラム実行 (ランタイム) 中にそれらを保存してアクセスするための空間を RAM 内に予約しておく事だけが必要です。

非初期化データセクションには .sbss と .bss があります。.sbss セクションは、*n* バイト以下 (*n* は -G*n* コマンドライン オプションで指定) の非初期化変数を格納するデータセグメントです。.bss セクションは .sbss に含めない非初期化変数を格納します。

スタートアップ コードのもう 1 つの役割は、プログラムの実行を開始する前に初期化が必要な変数に初期値を格納する事です。初期化変数は、定義内で初期値が代入される非 auto オブジェクトです (下の例の input)。

```
int input = 88;
int main(void) { ...
```

初期化が必要なオブジェクトは 2 つの部分 (プログラムメモリ内に保存された初期値 (上の例では 0x0088) と、その変数の保存 / アクセス用に予約された RAM 内の空間) で構成されます。

スタートアップ コードは、プログラムメモリ内の初期値のブロックを全て RAM にコピーする事により、main の実行前に変数に正しい値を格納します。

auto オブジェクトは動的に生成されるため、変数を定義して初期化するためのコードは関数内に置く必要があります。auto オブジェクトの初期値は関数呼び出しのたびに変化する可能性があるため、初期値をプログラムメモリに保存しておいて RAM にコピーする事はできません。結果として、初期化 auto オブジェクトはスタートアップ コードでは考慮されず、各関数に対して生成されるアセンブリコードによって初期化されます。

> **Note:** 初期化 auto 変数は、特にオブジェクトのサイズが大きい場合にコードの性能に影響するため、代わりにグローバルまたは static オブジェクトの使用を検討すべきです。

リセット後 (さらには電源 OFF 後) も値を保持する必要がある変数は、persistent 属性で修飾する必要があります (**8.10「標準型修飾子」**参照)。このような変数はメモリの別の領域でリンクされ、スタートアップ コードによって変更される事は決してありません。

初期化データ セクションには次の 4 つがあります : .sdata、.data、.lit4、.lit8。.sdata セクションは、*n* バイト以下 (*n* は -G*n* コマンドライン オプションで指定) の初期化が必要な変数を格納するデータセグメントです。.data セクションは、.sdata に格納されない初期化変数を格納します。.lit4 および .lit8 セクションは、アセンブラによってメモリ内 (命令ストリーム内ではない) に保存される定数を格納します。

© 2015 Microchip Technology Inc.

これらのセクションをクリアまたは初期化するために、リンカはデータ初期化テンプレートを生成します。このテンプレートは、.dinit という名前の出力セクションに書き込まれます。リンカは、実行時にデータを初期化するためのテンプレートを保持するために、プログラムメモリ内に割り当てられたこの特殊なセクション (.dinit) を生成します。C/C++ スタートアップ モジュール crt0.o は、このテンプレートに従って初期データ値を初期データセクションにコピーします。これには ramfunc 属性付きの関数を格納するセクションを含みます。他のデータセクション (.bss 等) は、main() 関数が呼び出される前にクリアされます。persistent 属性のデータセクション (.pbss) は影響を受けません。アプリケーションのメインプログラムが制御を引き継ぐ時点で、データメモリ内の全ての変数と RAM 関数は既に初期化済みです。

データ初期化テンプレートは、データメモリ内の出力セクションごとに 1 つのレコードを格納します。テンプレートは NULL 命令ワードで終了します。データ初期化レコードの書式を以下に示します。

```
/* data init record */
struct data_record {
  char *dst;            /* destination address  */
  unsigned int len;     /* length in bytes      */
  unsigned int format;  /* format code          */
  char dat[0];          /* variable-length data */
};
```

レコードの最初のエレメントは、データメモリ内のセクションを指すポインタです。2 番目のエレメントはセクションの長さ、3 番目のエレメントは書式コードです。最後のエレメントはデータバイトの配列です (任意に指定)。bss 型セクションにはデータバイトは不要です。

現在サポートしている書式コードは以下の通りです。

- 0 – 出力セクションに全て 0 を書き込む
- 1 – データ配列からデータの各バイトをコピーする

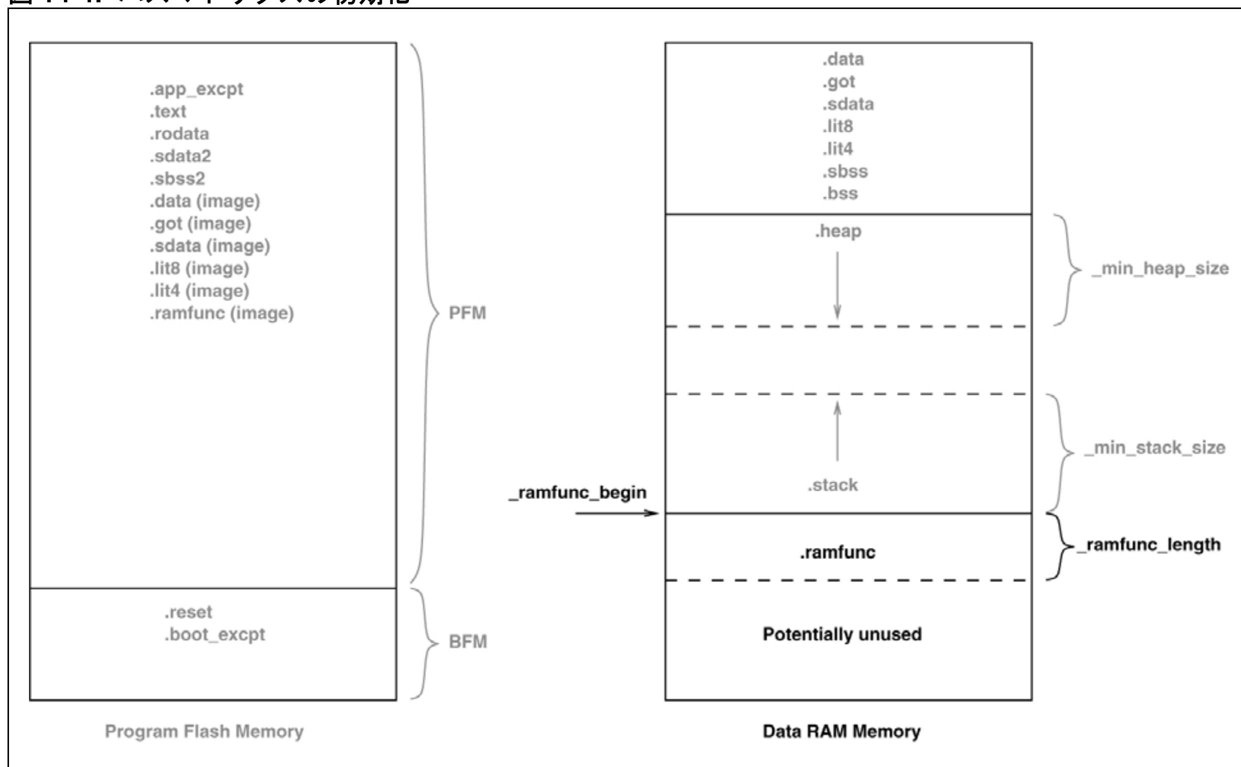
14.3.5 バスマトリクス レジスタの初期化

バスマトリクスを備えたデバイスで RAM 関数が存在する場合、スタートアップ コードはバスマトリクス レジスタ (BMXDKPBA、BMXDUDBA、BMXDUPBA) を初期化する必要があります。RAM 関数が存在しない場合、これらのレジスタは変更しません。リンカは RAM 関数の全てをデータメモリ内の 1 つのセクション (2K アラインメント境界に配置) に割り当てます。アプリケーション内に RAM 関数が存在するかどうかを判別するために、リンカはこのセクションの先頭アドレスを格納した変数を提供します。この変数の名前は _ramfunc_begin です。

加えて、リンカは境界レジスタ (BMXDKPBA) に必要な 2K アラインメント変数を提供します。この変数の名前は _bmxdkpba_address です。リンカは、バスマトリクス レジスタのアドレスを格納する3つの変数(_bmxdkpba_address、_bmxdudba_address、_bmxdupba_address) も提供します。

リンカは、BMXDKPBA レジスタの要求に従って、RAM 関数を 2K アラインメント境界に配置します。

図 14-4: バスマトリクスの初期化



14.3.5.1 CP0 レジスタの初期化

CP0 レジスタは以下の順番で初期化されます。

1. Count レジスタ
2. Compare レジスタ
3. EBase レジスタ
4. IntCtl レジスタ
5. Cause レジスタ
6. Status レジスタ

14.3.5.2 ハードウェア イネーブル レジスタ (HWREna – CP0 レジスタ 7、SELECT 0)

このレジスタは、どのハードウェア レジスタに対して RDHWR 命令によるアクセスを許可するのかを指定するビットマスクを格納します。特権ソフトウェアは、RDHWR 命令がどのハードウェア レジスタにアクセス可能であるかを決定できます。そうする事で、予約済み命令例外 (Reserved Instruction Exception) を処理し、命令を解釈し、仮想値を返すという代償を払ってレジスタを仮想化できます。例えば、Count レジスタへの直接アクセスが望ましくない場合、そのレジスタへのアクセスを個別に無効にし、オペレーティング システムによって戻り値を仮想化できます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.3 不正仮想アドレスレジスタ (BadVAddr – CP0 レジスタ 8、SELECT 0)

この読み出し専用レジスタは、直近のアドレスエラー例外 (AdEL または AdES) の原因となった仮想アドレスを保持します。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

 © 2015 Microchip Technology Inc.

14.3.5.4　カウントレジスタ (Count – CP0 レジスタ 9、SELECT 0)

このレジスタはタイマとして機能し、一定レートでインクリメントします。このレジスタは、命令が実行されても破棄されても、あるいはパイプラインで処理が進行してもしなくても一切関係なくインクリメントします。Cause レジスタの DC ビットが「0」の場合、カウンタは 1 クロックおきにインクリメントします。Count レジスタには、機能上の目的あるいは診断目的で書き込む事ができます (リセット時またはプロセッサを同期させるための書き込みを含む)。Debug レジスタの $Count_{DM}$ ビットに書き込む事により、プロセッサがデバッグモード中である時に Count レジスタのインクリメントを継続するかどうかを制御できます。

このレジスタは、PIC32 スタートアップ コード内でクリアされます。

14.3.5.5　コンペアレジスタ (Compare – CP0 レジスタ 11、SELECT 0)

このレジスタと Count レジスタの組み合わせにより、タイマおよびタイマ割り込み機能を実装します。タイマ割り込みはコアの出力の 1 つです。Compare レジスタは一定の値を保持します (自らは変化しません)。Count レジスタの値が Compare レジスタの値に一致すると、SI_TimerInt ピンがアサートされます。このピンは Compare レジスタが書き込まれるまでアサートされたままです。いずれかの割り込みピンを使って SI_TimerInt ピンをコアにフィードバックする事で、割り込みを生成できます。診断目的では、Compare レジスタは読み / 書き可能レジスタです。しかし通常の使い方では、Compare レジスタは書き込み専用です。Compare レジスタに値を書き込むと、副作用としてタイマ割り込みがクリアされます。

PIC32 スタートアップ コードは、このレジスタを 0xFFFFFFFF に設定します。

14.3.5.6　ステータス レジスタ (Status – CP0 レジスタ 12、SELECT 0)

この読み書き可能レジスタは、プロセッサの動作モード、割り込みの有効化、診断ステートを格納します。このレジスタ内のフィールドの組み合わせによってプロセッサの動作モードが決まります。

PIC32 スタートアップ コードは以下の設定を初期化します (レジスタを 0b000000000x0xx0?0000000000000000 に初期化)。

- ユーザモード中にコプロセッサ 0 へのアクセスを許可しない (CU0 = 0)
- ユーザモードは設定されたエンディアンを使う (RE = 0)
- 例外ベクタの位置を変更しない (BEV = no change)
- リセット例外ベクタへの移行の原因を示すフラグビットを変更しない (SR, NMI = no change)
- 割り込みマスクをクリアして全ての保留中割り込み要求を無効にする (IM7..IM2 = 0, IM1..IM0 = 0)
- 割り込み優先度を 0 に設定する (IPL = 0)
- ベースモードをカーネルモードにする (UM = 0)
- エラーレベルをノーマルにする (ERL = 0)
- 例外レベルをノーマルにする (EXL = 0)
- 割り込みを無効にする (IE = 0)

ターゲット デバイスが DSPr2 エンジンを備えている場合、DSPr2 エンジンを有効 (MX = 1) にします。

14.3.5.7　割り込み制御レジスタ (IntCtl – CP0 レジスタ 12、SELECT 1)

このレジスタは、アーキテクチャのリリース 2 で追加された拡張割り込み機能 (ベクタ割り込み、外部割り込みコントローラのサポート等) を制御します。

このレジスタは、割り込み処理のためのベクタ間隔を格納します。PIC32 スタートアップ コードは、このレジスタ内のベクタ間隔に対応するフィールド (bit 9-5) を _vector_spacing シンボルの値に初期化し、その他のビットを全て「1」にセットします。

14.3.5.8　シャドーレジスタ制御レジスタ (SRSCtl – CP0 レジスタ 12、SELECT 2)

このレジスタは、プロセッサ内の GPR シャドーセットの動作を制御します。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.9　シャドーレジスタ マップレジスタ (SRSMap – CP0 レジスタ 12、SELECT 3)

このレジスタは、ベクタ番号からシャドーセット番号へのマッピングを提供する 4 ビットのフィールドを 8 個格納します。これらは割り込みを処理する際に使います。このレジスタの値は、非割り込み型の例外または非ベクタ型の例外 ($\texttt{Cause}_{IV}$ = 0 または $\texttt{IntCtl}_{VS}$ = 0) には使いません。そのような場合、シャドーセット番号は SRSCtlESS によって決まります。SRSCtlHSS が「0」の場合、このレジスタに対するソフトウェア読み書きの結果は予測不可能です。このレジスタ内のいずれかのフィールドに SRSCtlHSS の値よりも大きな値が書き込まれた場合、プロセッサの動作は不確定です。SRSMap レジスタは、ベクタ番号 7 ～ 0 に対応するシャドーレジスタ セット番号を格納します。複数の割り込みベクタに対して同一のシャドーセット番号を割り当てる事ができます (ベクタからシャドーレジスタ セットへの多対一対応割り当てが可能)。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.10　原因レジスタ (Cause – CP0 レジスタ 13、SELECT 0)

このレジスタは、主に直前に発生した例外の原因に関する情報を提供します。加えて、割り込みを生成するためのソフトウェア割り込み要求およびベクタを制御します。Cause レジスタ内の DC、IV、IP1..IP0 以外のフィールドは全て読み出し専用です。アーキテクチャのリリース 2 には、外部割り込みコントローラ (EIC) 割り込みモードに対するサポートがオプションとして追加されています。このモードでは、IP7..IP2 は要求された割り込み優先度 (RIPL) として解釈されます。

PIC32 スタートアップ コードは以下の設定を初期化します。

- Count レジスタのカウントを有効にする (DC = no change)
- 特殊例外ベクタ (16#200) を使う (IV = 1)
- ソフトウェア割り込み要求を無効にする (IP1..IP0 = 0)

14.3.5.11　例外プログラム カウンタ (EPC – CP0 レジスタ 14、SELECT 0)

この読み書き可能レジスタは、例外をサービスした後に処理を再開するアドレスを格納します。EPC レジスタの全てのビットは効果を持ち、書き込み可能です。同期した (識別可能な) 例外の場合、EPC レジスタは以下のどれかを格納します。

- 例外の直接原因となった命令の仮想アドレス
- 直前の分岐またはジャンプ命令の仮想アドレス - 例外の原因となった命令が分岐遅延スロットにあり、かつ、Cause レジスタの Branch Delay ビットがセットされていた場合

新しい例外が発生した時に Status レジスタの EXL ビットがセットされていると、プロセッサは EPC レジスタに書き込みません。しかし、このレジスタには MTC0 命令を使って書き込む事ができます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

© 2015 Microchip Technology Inc.

14.3.5.12 プロセッサ識別レジスタ (PRId – CP0 レジスタ 15、SELECT 0)

この読み出し専用の 32 ビットレジスタは、プロセッサに関する情報 (製造者、製造者オプション、ID、リビジョンレベル) を格納します。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.13 例外ベースレジスタ (EBase – CP0 レジスタ 15、SELECT 1)

この読み / 書き可能レジスタは、$Status_{BEV}$ が「0」の時に使う例外ベクタのベースアドレスと、読み出し専用の CPU 番号を格納します。CPU 番号は、マルチプロセッサ システムでソフトウェアがプロセッサを識別するために使います。ソフトウェアは EBase レジスタを使ってマルチプロセッサ システム内の特定のプロセッサを識別できます。特に全く同じプロセッサを複数個使うシステムでは、各プロセッサの例外ベクタを識別するためにこのレジスタを使います。$Status_{BEV}$ が「0」の場合、EBase レジスタの bit 31-12 にゼロを連結する事で例外ベクタのベースアドレスを形成します。$Status_{BEV}$ が「1」の場合、または EJTAG デバッグ例外が発生した場合、例外ベクタのベースアドレスには固定された既定値を使います。リリース 1 処理系との下位互換性を提供するために、EBase レジスタの bit 31-12 のリセットによって例外ベースレジスタは 16#80000000 に初期化されます。例外ベースアドレスを KSEG0 または KSEG1 (マッピングされない仮想アドレス セグメント) 内に配置するために、EBase レジスタの bit 31-30 の値は 2#10 に固定されています。

例外ベースレジスタの値の変更は、$Status_{BEV}$ が「1」の時に行う必要があります。$Status_{BEV}$ が「0」の時に例外ベースフィールドに異なる値を書き込んだ場合、プロセッサの動作は不確定です。

例外ベースフィールドと bit 31-30 の組み合わせにより、例外ベクタのベースアドレスを任意の 4K バイトページ境界に配置できます。ベクタ割り込みを使う場合、4K バイトよりも大きなベクタオフセットを生成できます。この場合、例外ベースフィールドの bit 12 は「0」である事が必要です。ソフトウェアが例外ベースフィールドの bit 12 に「1」を書き込み、かつ、例外ベースアドレスからのオフセットが 4K バイトを越えるベクタ割り込みの使用を有効にした場合、プロセッサの動作は不確定です。

PIC32 スタートアップ コードは、_ebase_address シンボルの値を使ってこのレジスタを初期化します。リンカスクリプトは、_ebase_address を既定値 (KSEG1 プログラムメモリの開始アドレス) に設定します。ユーザは、リンカに対してコマンドライン オプション --defsym _ebase_address=A を指定する事で、この値を変更できます。

14.3.5.13.1 CONFIG レジスタ (Config – CP0 レジスタ 16、SELECT 0)

このレジスタは、各種のコンフィグレーションと機能に関する情報を定義します。Config レジスタ内の大部分のフィールドは、リセット例外処理中にハードウェアによって初期化されます (一部のフィールドだけ値を保持)。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.13.2 CONFIG1 レジスタ (Config1 – CP0 レジスタ 16、SELECT 1)

このレジスタは、Config レジスタの補助として、コアが実装する機能に関する追加の情報を格納します。Config1 レジスタ内の全てのフィールドは読み出し専用です。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.13.3 CONFIG2 レジスタ (Config2 – CP0 レジスタ 16、SELECT 2)

このレジスタは、Config レジスタの補助として、追加機能に関する情報を格納するために予約されています。Config2 レジスタは、レベル 2/3 キャッシュの設定を示します。コアは L2/L3 キャッシュをサポートしないため、これらのフィールドは「0」にリセットされます。Config2 レジスタ内の全てのフィールドは読み出し専用です。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.13.4 CONFIG3 レジスタ (Config3 – CP0 レジスタ 16、SELECT 3)

このレジスタは、追加機能に関する情報を格納します。Config3 レジスタ内の全てのフィールドは読み出し専用です。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.14 デバッグレジスタ (Debug – CP0 レジスタ 23、SELECT 0)

このレジスタは、デバッグ例外を制御すると共に、デバッグ例外の原因に関する情報を提供します。また、デバッグモード中に通常の例外が発生したためにデバッグ例外ベクタ位置で再入した時の情報も提供します。読み出し専用の情報ビットは、デバッグ例外が発生するたびに、または、既にデバッグモード中である場合に通常の例外が発生した時に更新されます。デバッグモード以外のモードからこのレジスタを読み出した場合、DM ビットと $EJTAG_{ver}$ フィールドだけが有効であり、その他のビットとフィールドは全て予測不可能です。デバッグモード以外のモードから Debug レジスタに書き込んだ場合、プロセッサの動作は未確定です。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.15 トレース制御レジスタ (TraceControl – CP0 レジスタ 23、SELECT 1)

このレジスタは制御とステータス情報を提供します。TraceControl レジスタは、EJTAG トレース機能が利用可能な場合にのみ実装されます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.16 トレース制御 2 レジスタ (TraceControl2 – CP0 レジスタ 23、SELECT 2)

このレジスタは、追加の制御とステータス情報を提供します。TraceControl2 レジスタは、EJTAG トレース機能が利用可能な場合にのみ実装されます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.17 ユーザトレース データレジスタ (UserTraceData – CP0 レジスタ 23、SELECT 3)

このレジスタには、タイプ 1 またはタイプ 2 のユーザ書式を示すトレースレコードが書き込まれます。このタイプは、TraceControl レジスタ内の UT ビットによって決まります。このレジスタには連続したサイクルで書き込む事はできません。このレジスタに連続したサイクルで書き込んだ場合、トレース出力データは予測不可能です。UserTraceData レジスタは、EJTAG トレース機能が利用可能な場合にのみ実装されます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.18 TRACEBPC レジスタ (TraceBPC – CP0 レジスタ 23、SELECT 4)

このレジスタは、EJTAG ハードウェア ブレークポイントを使ってトレースの開始 / 終了を制御するために使います。ハードウェア ブレークポイントはトリガ源として設定されます。オプションにより、デバッグ例外ブレークポイントとして設定する事もできます。TraceBPC レジスタは、ハードウェア ブレークポイント機能と EJTAG トレース機能の両方が利用可能な場合にのみ実装されます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

© 2015 Microchip Technology Inc.

14.3.5.19 デバッグ 2 レジスタ (Debug2 – CP0 レジスタ 23、SELECT 5)

このレジスタは、複雑なブレークポイント例外に関する追加の情報を保持します。Debug2 レジスタは、複雑なハードウェア ブレークポイント機能が利用可能な場合にのみ実装されます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.20 デバッグ例外プログラム カウンタ (DEPC – CP0 レジスタ 24、SELECT 0)

この読み書き可能レジスタは、「デバッグ例外」または「デバッグモード中の例外」をサービスした後に処理を再開するアドレスを格納します。同期した (識別可能な)「デバッグ例外」および「デバッグモード中の通常例外」の場合、DEPC レジスタは以下のいずれかを格納します。

- デバッグ例外の直接原因となった命令の仮想アドレス
- 直前の分岐またはジャンプ命令の仮想アドレス - デバッグ例外の原因となった命令が分岐遅延スロットにあり、かつ、Debug レジスタの Debug Branch Delay (DBD) ビットがセットされていた場合

非同期のデバッグ例外 (デバッグ割り込み、複雑なブレーク) の場合、DEPC はデバッグハンドラ コードを処理した後に実行を再開する命令の仮想アドレスを格納します。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.21 エラー例外プログラム カウンタ (ErrorEPC – CP0 レジスタ 30、SELECT 0)

この読み書き可能レジスタは、エラー例外で使われるという点を除けば、EPC レジスタと同様です。ErrorEPC の全てのビットは効果を持ち、書き込み可能です。このレジスタはリセット、ソフトリセット、ノンマスカブル割り込み (NMI) 例外時にプログラム カウンタを保存するためにも使います。ErrorEPC レジスタは、エラーを処理した後に命令の処理を再開可能な位置を指す仮想アドレスを格納します。このアドレスは以下のいずれかです。

- エラー例外の原因となった命令の仮想アドレス
- 直前の分岐またはジャンプ命令 - 仮想アドレスエラーの原因となった命令が分岐遅延スロットにある場合

EPC レジスタとは異なり、ErrorEPC レジスタは対応する分岐遅延スロットを示しません。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.5.22 デバッグ例外保存レジスタ (DeSave – CP0 レジスタ 31、SELECT 0)

この読み書き可能レジスタは、単純なメモリアドレスとして機能します。デバッグ例外ハンドラは、このレジスタを使って GPR の 1 つを保存します。この GPR は、残りのコンテクストをあらかじめ決められたメモリ領域 (EJTAG プローブ等) に保存するために使われます。このレジスタを使う事で、例外ハンドラとその他のコード (コンテクストの保存用に有効なスタックを確保できそうにないコード) を安全にデバッグできます。

PIC32 のスタートアップ コードは、このレジスタを初期化しません。

14.3.6 _on_bootstrap プロシージャの呼び出し

CP0 レジスタの初期化後に 1 つのプロシージャが呼び出されます。このプロシージャは、メインルーチンを開始する前のブートストラップ中 (StatusBEV がセットされている間) のユーザによるアクションを可能にします。このプロシージャ (_on_bootstrap) の空白の弱いバージョンがスタートアップ コードで提供されます。このプロシージャは、ハードウェアの初期化または RTOS が要求する環境の初期化 (もしくはその両方) に使えます。

14.3.7 例外ベクタ位置の変更

アプリケーション コードを実行する直前に、例外ベクタの位置をブートストラップ位置から通常位置へ変更するために、StatusBEV がクリアされます。

14.3.8 C++ 初期化コードの呼び出し

C++ ファイルスコープ静的ストレージ オブジェクト向けの全てのコンストラクタを呼び出します。アプリケーション コードを実行する前に低レベルの初期化が必要であるため、スタートアップ コードは最後にコンストラクタを呼び出す必要があります。

14.3.9 メインルーチンの呼び出し

スタートアップ コードは最後にメインルーチンを呼び出します。ユーザがメインルーチンから戻ると、スタートアップ コードは無限ループに入ります。

14.3.10 スタートアップ コードと C/C++ ライブラリに必要なシンボル

以下では、スタートアップ コードと C/C++ ライブラリに必要なシンボルの詳細について説明します。現在、これらのシンボルは既定値リンカスクリプトによって定義されます。アプリケーションが独自のリンカスクリプトを提供する場合、スタートアップ コードと C ライブラリが正しく機能するように、ユーザは以下のシンボルの全てを提供する必要があります。

シンボル名	概要
_bmxdkpba_address	_ramfunc_length が 0 よりも大きい場合に BMXDKPBA レジスタに格納するアドレス
_bmxdudba_address	_ramfunc_length が 0 よりも大きい場合に BMXDUDBA レジスタに格納するアドレス
_bmxdupba_address	_ramfunc_length が 0 よりも大きい場合に BMXDUPBA レジスタに格納するアドレス
_ebase_address	EBASE のアドレス
_end	データ割り当ての終端
_gp	「小さな」変数領域の中央のアドレスを指すポインタ (慣例により、このポインタは「小さな」変数用に使われる領域の先頭アドレスから 0x8000 バイトの位置を指します)
_heap	DRM 内のヒープの開始アドレス
_ramfunc_begin	RAM 関数の開始アドレス (これは BMXDKPBA レジスタの初期化に使われるため、2K バイト境界に配置する必要があります)
_ramfunc_length	.ramfunc セクションの長さ
_stack	DRM内のスタックの開始アドレス(スタックはデータメモリの高アドレスから低アドレスへ向かって進みます。従って、このシンボルは、スタックが割り当てられるセクションの終端位置を指す必要があります)
_vector_spacing	IntCtl レジスタ内のベクタ間隔フィールドの初期値

 © 2015 Microchip Technology Inc.

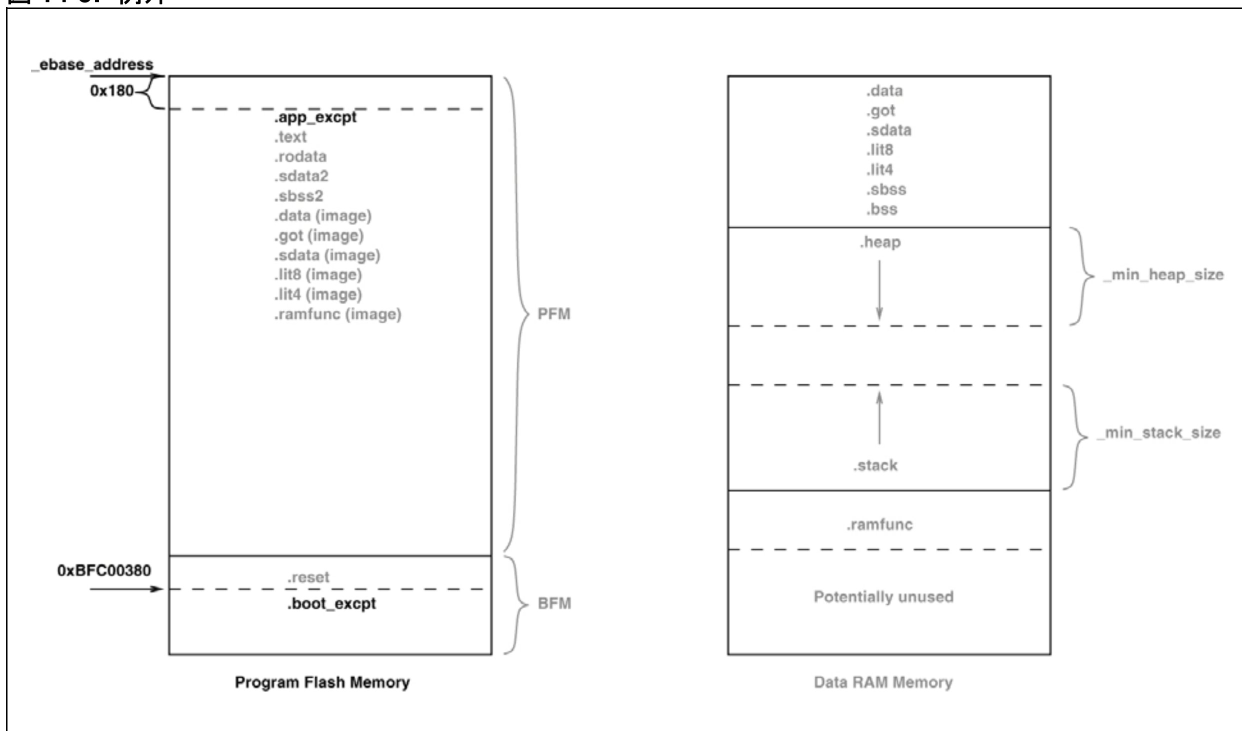
14.3.11 例外

2 つの「弱い」(すなわちアプリケーションによってオーバーライド可能な) 汎用例外ハンドラも提供されます (StatusBEV が「1」の場合に例外を処理する _bootstrap_exception_handler と、StatusBEV が「0」の時に例外を処理する _general_exception_handler)。これらの「弱い」リセット例外ハンドラと「弱い」一般例外ハンドラは、どちらもスタートアップ コードで提供され、ソフトウェア リセットを発生させます。スタートアップ コードは、ブートストラップ例外ハンドラへのジャンプを 0xBFC00380 に配置し、一般例外ハンドラへのジャンプを EBASE + 0x180 に配置します。

スタートアップ コードはこれらの関数にジャンプするため、どちらのハンドラにも nomips16 属性を与える必要があります (例 : __attribute__ ((nomips16)))。

BOOTISA コンフィグレーション ビットで例外を microMIPS モードに設定した場合、どちらのハンドラにも micromips 属性を与える必要があります (例 : __attribute__((micromips)))。

図 14-5: 例外



14.4 On Reset ルーチン

ハードウェア コンフィグレーションによっては、リセット後の最初の数命令サイクル中に特殊な初期化が必要になる事がしばしばあります。これに対応するため、On Reset ルーチンによってフックが提供されます。

このルーチンは、最小限の C コンテクストを初期化した後に呼び出されます。このプロシージャ (`_on_reset`) の空白の弱いバージョンがスタートアップ コードで提供されます。このルーチンのスタブは、コンパイラ インストール ディレクトリ内の `pic32-libs/libpic32/stubs` にあります。

このプロシージャを C 言語で書く場合、特別な配慮が必要です。最も重要なのは、静的に割り当てられた変数は初期化されないという事です (非初期化変数と同様に、指定された初期化子またはゼロによって初期化されません)。スタックポインタは、このルーチンが呼び出された時点で初期化済みです。

14.4.1 オブジェクトのクリア

スタートアップ コードは、非初期化変数によって使われる全てのメモリ位置をクリアします。従って、`main()` 実行前のそれらの値は 0 です。

リセットの前後で値を保持する必要がある変数には `persistent` 属性を使います。詳細は **8.10「標準型修飾子」** を参照してください。このような変数はメモリの別の領域でリンクされ、いかなる場合もスタートアップ コードによって変更されません。

© 2015 Microchip Technology Inc.

第 15 章　ライブラリ ルーチン

15.1　ライブラリ ルーチンの使用

ソースコード内で参照している全てのライブラリ関数またはルーチン(および関連する変数) は、自動的にプログラムにリンクされます。あるライブラリ ファイルから 1 つの関数だけを使う場合、そのライブラリ内の他の関数はインクルードされません。使用するライブラリ関数だけがプログラム出力にリンクされてメモリを消費します。

プログラムには、ライブラリから使用する全ての関数またはシンボルに対する宣言が必要です。これらは標準 C ヘッダ (.h) ファイルに含まれています。ヘッダファイルはライブラリ ファイルではありません。これら 2 種類のファイルを混同しないよう注意が必要です。ライブラリ ファイルにはプリコンパイル済みコード (通常は関数と変数の定義) が書かれています。ヘッダファイルはライブラリ ファイル内の関数、変数、型に関する宣言 （ ライブラリ ファイル内の定義に対応 ） と、その他のプリプロセッサ マクロを提供します。

```
#include <math.h>    // declare function prototype for sqrt

int main(void)
{
  double i;

  // sqrt referenced; sqrt will be linked in from library file
  i = sqrt(23.5);
}
```

NOTE:

© 2015 Microchip Technology Inc.

第 16 章　C/C++ とアセンブリ言語の併用

16.1　はじめに

アセンブリコードと C/C++ コードは 2 通りの方法で併用できます。すなわち、アセンブリコードを書いて別のアセンブラ モジュールに収める方法と、インライン アセンブリとして C/C++ モジュール内に含める方法があります。以下では、これらの方法について説明すると共に、C/C++ 変数 / 関数をアセンブリコード内で使う例と、アセンブリ変数 / 関数を C/C++ 内で使う例を示します。

プロジェクトに含めるアセンブリコードが多ければ多いほど、管理はより困難となり、手間がかかります。一部の最適化はプログラムの全体を見るため、プロジェクトの開発が進むにつれてコンパイラの挙動が変化する場合があります。また、コンパイラを新しいバージョンに更新すると、コンパイラの挙動が変化してアセンブリコードで問題が生じやすくなります。これらは C/C++ で書かれたコードには影響しません。

> **Note:** アセンブリコードを追加する必要がある場合、インライン アセンブリとして C コード内に含めるのではなく、別のアセンブリ モジュール内で独立したルーチンとして書く事を推奨します。

- アセンブリ言語と C 変数 / 関数の併用
- インライン アセンブリ言語の使い方
- 定義済みマクロ

16.2　アセンブリ言語と C 変数 / 関数の併用

別のアセンブリ モジュールを C モジュールと連携させるためのガイドラインを以下に示します。

- レジスタの表記法は **11.3「レジスタの用法」**に従います。パラメータの引き渡しにはレジスタ $4 ～ $7 を使います。これらのレジスタを介して、アセンブリ関数はパラメータを受け取ると共に、呼び出し元の関数に引数を渡します。
- 表 11-1「レジスタの用法」には、レジスタの値が非割り込み型関数呼び出しを越えて保存されるかどうかを記載しています。
- 割り込み型関数は、全てのレジスタを保存する必要があります。通常の関数呼び出しと異なり、割り込みはプログラムの実行中にいつでも発生する可能性があります。通常のプログラムに戻る際は、全てのレジスタを割り込み発生前の状態に戻す必要があります。
- C ソースファイルが別のアセンブリ ファイルを参照する場合、そのアセンブリ ファイル内で宣言する変数または関数は、アセンブラ ディレクティブ `.global` を使ってグローバルとして宣言する必要があります。アセンブリ ファイル内で使われる未宣言のシンボルは、外部で定義されるものとして扱われます。

以下の例では、アセンブリコードと C コードの両方で、変数と関数をそれらがどちらで定義されたかに関係なく使う方法を示します。

ファイル `ex2.c` は、アセンブリ ファイル内で使われる `foo` と `cVariable` を定義します。この C ファイルは、アセンブリ関数 (`asmFunction`) を呼び出す方法と、アセンブリ ファイルで定義された変数 (`asmVariable`) にアクセスする方法も示しています。

例 16-1: C 言語とアセンブリ言語の併用

```
/*
** file: ex1.S
*/
#include <xc.h>

    /* define which section (for example "text")
     * does this portion of code resides in. Typically,
     * all your code will reside in .text section as
     * shown below.
     */
   .text

    /*  This is important for an assembly programmer.This
     * directive tells the assembler that don't optimize
     * the order of the instructions as well as don't insert
     * 'nop' instructions after jumps and branches.
     */
   .set noreorder

/*********************************************************************
* asmFunction(int bits)
* This function clears the specified bites in IOPORT A.
********************************************************************/
.global asmFunction
.ent asmFunction
asmFunction:
   /* function prologue - save registers used in this function
     * on stack and adjust stack-pointer
     */
   addiu   sp, sp, -4
   sw      s0, 0(sp)

   la      s0, LATACLR
   sw      a0, 0(s0)      /* clear specified bits */

   la      s0, PORTA
   lw      s1, 0(s0)
   la      s0, cVariable
   sw      s1, 0(s0)      /* keep a copy */

   /* function epilogue - restore registers used in this function
     * from stack and adjust stack-pointer
     */
   lw      s0, 0(sp)
   addiu   sp, sp,

   addu    s1, ra, zero
   jal     foo
   nop
   addu    ra, s1, zero
   nop
   /* return to caller */
   jr      ra
   nop
.end asmFunction

  .bss
  .global asmVariable
  .align 2
asmVariable:.space 4
```


© 2015 Microchip Technology Inc.

ファイル ex1.S は、リンクされたアプリケーションに必要な asmFunction と asmVariable を定義します。このアセンブリ ファイルは、C 関数 foo を呼び出す方法と、C ファイルで定義された変数 cVariable にアクセスする方法も示しています。

```
;
;  file: ex2.c
;
#include <xc.h>
#include <plib.h>

extern void asmFunction(int bits);
extern unsigned int asmVariable;
volatile unsigned int cVariable = 0;
volatile unsigned int jak = 0;

int main(void) {
    SYSTEMConfigPerformance(80000000ull);
    TRISA = 0;
    LATA = 0xC6FFul;

    asmFunction(0xA55Au);
    while (1)
    {
        asmVariable++;
    }
}

void foo (void)
{
    jak++;
}
```

C ファイル ex2.c では、標準の extern キーワードを使って、アセンブリ ファイル内で宣言されているシンボルへの外部参照を宣言しています (asmFunction は void 関数として宣言している事に注意)。

アセンブリ ファイル ex1.S では、アセンブラ ディレクティブ .global を使って、シンボル asmFunction および asmVariable をグローバルに可視 (他の全てのソースファイルからアクセス可能) としています。

16.3 インライン アセンブリ言語の使い方

C/C++ 関数内で asm ステートメントを使う事により、コンパイラが生成するアセンブリ コードに 1 行のアセンブリ コードを挿入できます。インライン アセンブリには「シンプル」と「拡張」の 2 形態があります。

シンプル型の場合、以下の構文を使ってアセンブラ命令を書きます。

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

instruction は、アセンブリ言語のコンストラクタです。ANSI C プログラム内でインライン アセンブリを書く場合、asm の代わりに __asm__ を使います。

> **Note:** シンプル型のインライン アセンブリには 1 行だけ渡す事ができます。

asm を使う**拡張**型アセンブラ命令の場合、命令のオペランドは C/C++ 表現を使って指定します。

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

各オペランドには、アセンブラ命令 template に加えてオペランド constraint 文字列を指定する必要があります。template は命令ニーモニックと、オプションとしてオペランドのプレースホルダを指定します。constraint 文字列はオペランドの制約を指定します (例 : オペランドをレジスタに格納するか (通常のケース)、即値にするか)。

本コンパイラがサポートする制約文字と修飾子を表 16-1 ～表 16-4 に示します。

表 16-1: コンパイラがサポートするレジスタの制約文字

文字	制約
c	間接ジャンプでの使用に適したレジスタ
d	アドレスレジスタ (MIPS16 コードを生成しない場合、これは @code{r} と等価です)
ka	乗累算命令のターゲットとして使えるレジスタ
l	@code{lo} レジスタ (このレジスタは、1 ワード以下の値の保存用に使います)
x	@code{hi} および @code{lo} レジスタの連結 (このレジスタは、2 ワード値の保存用に使います)

© 2015 Microchip Technology Inc.

表 16-2: コンパイラがサポートする整数の制約文字

文字	制約
I	符号付き 32 ビット定数 (算術命令用)
J	整数のゼロ
K	符号なし 32 ビット定数 (論理命令用)
L	下位32ビットがゼロの符号付き32ビット定数(このような定数は@code{lui}を使ってロードできます)
M	@code{lui}、@code{addiu}、@code{ori}を使ってロードできない定数
N	-65535 ～ -1 の定数 (-65535 と -1 を含む)
O	符号付き 15 ビット定数
P	1 ～ 65535 の定数 (1 と 65535 を含む)

表 16-3: コンパイラがサポートする一般的制約文字

文字	制約
R	非マクロ型ロード / ストアで使えるアドレス

表 16-4: コンパイラがサポートする制約修飾子

コード	制約
=	このオペランドがこの命令に対して書き込み専用である(以前の値は出力データによって上書き (破棄) される) 事を意味します。
+	このオペランドが命令によって読み書き可能である事を意味します。
&	このオペランドが earlyclobber オペランドである (命令が終了する前に入力オペランドを使って変更される) 事を意味します。従ってこのオペランドは、入力オペランドまたはメモリアドレスの一部として使われるレジスタに格納する事はできません。
d	オペランド番号 *n* のための第 2 のレジスタ (すなわち %d*n*..)
q	オペランド番号 *n* のための第 4 のレジスタ (すなわち %q*n*..)
t	オペランド番号 *n* のための第 3 のレジスタ (すなわち %t*n*..)

© 2015 Microchip Technology Inc.

例

- ビットフィールドの挿入
- 複数のアセンブリ命令

ビットフィールドの挿入

この例では、INS 命令を使ってビットフィールドを 32 ビット幅の変数に挿入する方法を示します。この関数的なマクロは、インライン アセンブリを使って INS 命令を発行します (通常この命令は C/C++ コードから生成されません)。

```
/* MIPS32r2 insert bits */
#define _ins(tgt,val,pos,sz) __extension__({                     \
    unsigned int __t = (tgt), __v = (val);                       \
    __asm__ ("ins %0,%z1,%2,%3"                   /* template */ \
             :"+d" (__t)                          /* output   */ \
             :"dJ" (__v), "I" (pos), "I" (sz)); /* input    */ \
    __t;                                                         \
})
```

__v、pos、sz は入力オペランドです。__v オペランドには制約「d」(アドレスレジスタ) および「J」(整数ゼロ) を適用します。pos および sz オペランドには制約「I」(符号付き 32 ビット定数) を適用します。

__t 出力オペランドには制約「d」(アドレスレジスタ) を適用します。「+」修飾子は、このオペランドが命令によって読み書き可能である事 (つまり入出力が可能である事) を意味します。

下の例に、このマクロの使い方を示します。

```
unsigned int result;
void example (void)
{
    unsigned int insertval = 0x12;
    result = 0xAAAAAAAAu;
    result = _ins(result, insertval, 4, 8);
    /* result is now 0xAAAAA12A */
}
```

この例に対し、コンパイラは以下に類似するアセンブリ コードを生成します。

```
   li     $2,-1431699456      # 0xaaaa0000
   ori    $2,$2,0xaaaa        # 0xaaaa0000 | 0xaaaa

   li     $3,18               # 0x12
   ins  $2,$3,4,8             # inline assembly

   lui    $3,%hi(result)      # assign the result back
   j      $31                 # return
   sw     $2,%lo(result)($3)
```


© 2015 Microchip Technology Inc.

複数のアセンブリ命令

この例では、バイトスワップのために WSBH 命令と ROTR 命令を一緒に使う方法を示します。WSBH 命令は、2 つのハーフワードのそれぞれの中での 32 ビット バイトスワップです。ROTR 命令は即時の右ローテートです。この関数的なマクロは、インライン アセンブリを使って「バイトスワップ ワード」を生成します (通常これらの命令は C/C++ コードから生成されません)。

以下に、関数的マクロ _bswapw の定義を示します。

```
/* MIPS32r2 byte-swap word */
#define _bswapw(x) __extension__({        \
    unsigned int __x = (x), __v;          \
    __asm__ ("wsbh %0,%1;\n\t"            \
              "rotr %0,16" /* template */ \
             :"=d" (__v)  /* output */   \
             :"d" (__x))  /* input*/ ;   \
    __v;                                  \
})
```

__x は、入力オペランドの C 表現です。このオペランドには制約「d」(アドレスレジスタ) を適用します。

C 表現の __v は出力オペランドです。このオペランドにも制約「d」を適用します。「=」は、このオペランドがこの命令に対して書き込み専用である (以前の値は出力データによって上書き (破棄) される) 事を意味します。

下の例に示す関数的マクロは、value の内容をスワップして result に代入します。

```
unsigned int result;
int example (void)
{
   unsigned int value = 0x12345678u;
   result = _bswapw(value);
   /* result == 0x78563412 */
}
```

この例に対し、コンパイラは以下に類似するアセンブリ コードを生成します。

```
  li$2,305397760          # 0x12340000
  addiu$2,$2,22136        # 0x12340000 + 0x5678
  wsbh $2,$2;             # From inline asm
  rotr $2,16              # From inline asm
  lui$2,%hi(result)       # assign back to result
  j $31                   # return
  sw$3,%lo(result)($2)
```

16.3.1 等価アセンブリ シンボル

拡張アセンブリコードでは、変更なしで C/C++ シンボルに直接アクセスできます。

16.4 定義済みマクロ

<xc.h> をインクルードする事で、各種の定義済みマクロが利用できます。これらのマクロの厳密な動作は、使用する命令セットによって異なります。表 16-5 に、汎用定義済みマクロとその動作を示します。

表 16-5: 定義済みマクロ

マクロ	概要
_nop()	NOP 命令を挿入します。
_ehb()	EHB (Execution Hazard Barrier) 命令を挿入します。
_cache(op,addr)	キャッシュラインに対する操作を行います。利用可能な操作についてはデバイスの文書を参照してください。
_synci(addr)	I キャッシュを D キャッシュに同期させます (命令を書き込んだ後、それらを実行する前に、各キャッシュライン サイズのブロックに対して命令を実行します)。
_prefetch(hint,x)	メモリ参照を最適化するために命令をプリフェッチします。データが必要になる事があらかじめ予測できるアプリケーションでは、データをキャッシュに転送するよう準備できます。「hint」はプリフェッチの種類を定義します。
_sync()	共有メモリ同期 (Synchronize Shared Memory) 命令を挿入します。
_wait()	スタンバイモードへ移行するための命令を挿入します。
_mfc0(rn, sel)	<xc.h> ファイル参照。値を CP0 レジスタから移動します。
_mtc0(rn, sel, v)	<xc.h> ファイル参照。値を CP0 レジスタへ移動します。
_mxc0(rn, sel, v)	<xc.h> ファイル参照。値を CP0 レジスタ内の値と交換します。
_bcc0(rn, sel, clr)	rnとselで指定したCP0レジスタにおいて、clrの非ゼロのビットに対応するビットをクリアします。
_bsc0(rn, sel, set)	rnとselで指定したCP0レジスタにおいて、setの非ゼロのビットに対応するビットをセットします。
_bcsc0(rn, sel, clr, set)	rnとselで指定したCP0レジスタにおいて、clrの非ゼロのビットに対応するビットをクリアし、setの非ゼロのビットに対応するビットをセットします。
_clz(x)	x 内の先頭の「0」の数をカウントします。
_ctz(x)	x 内の末尾の「0」の数をカウントします。
_clo(x)	x 内の先頭の「1」の数をカウントします。
_dclz(x)	x 内の先頭の「0」の数の 64 ビット値をシミュレートします。
_dclo(x)	x 内の先頭の「1」の数の 64 ビット値をシミュレートします。
_dctz(x)	x 内の末尾の「0」の数の 64 ビット値をシミュレートします。
_wsbh(x)	<xc.h>ファイル参照。2つのハーフワードのそれぞれの中で32 ビット バイトスワップを行います。
_bswapw(x)	<xc.h> ファイル参照。ワードをバイトスワップします。
_ins(tgt,val,pos,sz)	<xc.h> ファイル参照。ビットを挿入します。
_ext(x,pos,sz)	<xc.h> ファイル参照。32 ビット変数からビットフィールドを抽出します。
_jr_hb()	<xc.h> ファイル参照。ハザードバリア付きのレジスタをジャンプします。

© 2015 Microchip Technology Inc.

表 16-5: 定義済みマクロ (続き)

マクロ	概要
_wrpgpr(regno, val)	<xc.h> ファイル参照。以前のレジスタセット内のレジスタに書き込みます。
_rdpgpr(regno)	<xc.h> ファイル参照。以前のレジスタセットからレジスタを読み出します。
_get_byte(addr, errp)	addr の最下位バイトを返します。
_get_half(addr, errp)	addr の最下位 16 ビットワードを返します。
_get_word(addr, errp)	addr の最下位 32 ビットワードを返します。
_get_dword(addr, errp)	addr の最下位 64 ビットを返します。
_put_byte(addr, v)	addr の最下位バイトに v を書き込みます。
_put_half(addr, v)	addr の最下位 16 ビットワードに v を書き込みます。
_put_word(addr, v)	addr の最下位 32 ビットワードに v を書き込みます。
_put_dword(addr, v)	addr の最下位 64 ビットワードに v を書き込みます。

NOTE:

© 2015 Microchip Technology Inc.

第 17 章　最適化

17.1　はじめに

MPLAB XC32 C/C++ コンパイラは、エディションごとにサポートする最適化レベルが異なります。一部のエディションは無償でダウンロードできます。C および C++ ライセンスの詳細は http://www.microchip.com/MPLABXCcompilers を参照してください。

以下のコンパイラ エディションがあります。

エディション	ライセンス	概要
プロフェッショナル (PRO)	有償	最高の最適化レベルと性能レベルを提供します。
標準 (STD)	有償	十分な最適化レベルと性能レベルを提供します。
無償 (FREE)	無償	コード最適化機能の大部分が制限されます。
評価用 (EVAL)	無償	PRO エディションの機能が 60 日間使えます。期限が過ぎると FREE エディションに戻ります。

最適化レベルの設定

コンパイラのエディションに応じて、「最適化なし」から「フル最適化」まで各種の最適化レベルが設定できます。コードをデバッグする際は、最適化を行わずにプログラムの動作を確認できます。

最適化を設定するためのコンパイラ オプションの詳細は **5.9.7「最適化を制御するためのオプション」**を参照してください。

NOTE:

© 2015 Microchip Technology Inc.

第 18 章　前処理

18.1　はじめに

全ての C/C++ ソースファイルは、コンパイルの前に前処理されます。アセンブリ ソースファイル (*.S: S は大文字) も前処理されます。プリプロセッサの動作と、前処理で生成されるコードを制御するために、多数のオプションが存在します (**5.9.8「プリプロセッサを制御するためのオプション」**参照)。

- C/C++ コードのコメント
- プリプロセッサ ディレクティブ
- プラグマ ディレクティブ
- 定義済みマクロ

18.2　C/C++ コードのコメント

コンパイラは C/C++ コード内のコメントを無視します。コメントには、ソースコードを読みやすくするためのテキストを自由に書き込めます。

コメントのテキストは「`/*`」と「`*/`」で囲みます。コメントは複数行に分割できますが、ネストはできません。コメントは C/C++ コード内のどこにでも挿入できます。式文の中に挿入する事も可能ですが、文字定数または文字列リテラルの中に挿入する事はできません。

コメントはネストできないため、コメントを既に含んでいるコードは、下の例のように `#if` プリプロセッサ ディレクティブを使ってコメント化する事を推奨します。

```
#if 0
   result = read();  /* TODO:Jim, check this function is right */
#endif
```

下の例に示す 1 行の C++ スタイル コメントも指定できます。この場合、`//` から行末までの全ての文字はコメントとして扱われ、コンパイラによって無視されます。

```
   result = read();  // TODO:Jim, check this function is right
```

18.3 プリプロセッサ ディレクティブ

MPLAB XC32 C/C++ コンパイラは、表 18-1 に示す全ての標準プリプロセッサ ディレクティブをサポートします。

表 18-1: プリプロセッサ ディレクティブ

ディレクティブ	意味	例
#	プリプロセッサ NULL ディレクティブ (動作なし)	#
#assert	条件が「偽」の場合にエラーを生成します。	#assert SIZE > 10
#define	プリプロセッサ マクロを定義します。	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	#else #if の短縮形	#ifdef 参照
#else	条件付きでソース行をインクルードします。	#if 参照
#endif	条件付きソース インクルードの終端	#if 参照
#error	エラーメッセージを生成します。	#error Size too big

#if	定数式が「真」であればソース行をインクルードします。	#if SIZE < 10
		  c = process(10)
		#else
		  skip();
		#endif

#ifdef	プロセッサ シンボルが定義されている場合にソース行をインクルードします。	#ifdef FLAG
		  do_loop();
		#elif SIZE == 5
		  skip_loop();
		#endif

#ifndef	プロセッサ シンボルが未定義の場合にソース行をインクルードします。	#ifndef FLAG
		  jump();
		#endif

#include	テキストファイルをソースにインクルードします。	#include <stdio.h> #include "project.h"
#line	リスティングの行番号とファイル名を指定します。	#line 3 final
#*nn*	(#line *nn* の短縮形 (*nn* は行数))	#20
#pragma	コンパイラに固有のオプション	(**18.4「プラグマ ディレクティブ」**参照)
#undef	プロセッサ シンボルを未定義にします。	#undef FLAG
#warning	警告メッセージを生成します。	#warning Length not set

引数を使ったマクロ展開は、「#」を使って引数を文字列に変換でき、「##」を使って引数を連結できます。2 つの式を連結する場合、どちらかの式がそれ自身の代入を必要とするケースでは、以下の例のように 2 つのマクロを使う事を考慮します。

```
#define   paste1(a,b)  a##b
#define   paste(a,b)   paste1(a,b)
```

paste マクロを使って、それら自身がさらなる展開を必要とする可能性のある 2 つの式を連結します。置換トークンはさらなるマクロ識別子に対して再スキャンされます。しかし、一度展開されたマクロ識別子が連結後に現れた場合、それは再度展開されないという事に注意が必要です。

 © 2015 Microchip Technology Inc.

プリプロセッサ ドメインにおける数値の型と用法は C ドメインと同じです。プリプロセッサ値は型を持ちませんが、プリプロセッサによって変換されると型が与えられます。C の式と同様に、式は割り当てられた型に対してオーバーフローします。

定数に添え字を付ける事により、オーバーフローを防ぐ事ができます。例えば、後に添え字「L」が付いた数値の変換後の型は Long として解釈されます。

以下に例を示します。

```
#define MAX 100000*100000
```

と

`#define MAX 100000*100000L` (添え字 L に注意)
は、それぞれ値 0x540be400 と 0x2540be400 を定義します。

18.4 プラグマ ディレクティブ

特定のコンパイル時ディレクティブを使う事により、コンパイラの挙動を変更できます。これらは、ANSI 規格の `#pragma` ファシリティを使って実装されます。コンパイラが理解しないプラグマは全て無視されます。

プラグマの書式は以下の通りです。

```
#pragma keyword options
```

keyword は、キーワードセットの中の1つを指定します。一部のキーワードには *options* を指定します。以下では、各キーワードについて説明します。

#pragma interrupt

関数を割り込みハンドラとして指定します。その関数のプロローグおよびエピローグコードは、通常よりも拡張されたコンテクスト保存を行います。関数を割り込みハンドラとして指定する場合、このプラグマよりも `interrupt` 属性を使う事を推奨します。このプラグマは、他のコンパイラとの互換性維持のために用意されています。**第 13 章「割り込み」**と **13.5「例外ハンドラ」**を参照してください。

#pragma vector

指定した例外ベクタに、その関数をターゲットとする分岐命令を配置します。例外 / 割り込みベクタの生成には、このプラグマよりも `vector` 属性を使う事を推奨します。**第 13 章「割り込み」**と **13.5「例外ハンドラ」**を参照してください。

#pragma config

`#pragma config` ディレクティブは、アプリケーションが使うプロセッサに固有のコンフィグレーション設定 (コンフィグレーション ビット) を指定します。**13.3.2「# pragma interrupt」**を参照してください。

© 2015 Microchip Technology Inc.

18.5 定義済みマクロ

本コンパイラでは、以下の定義済みマクロが使えます。

- 32 ビット C/C++ コンパイラ マクロ
- SDE 互換マクロ

18.5.1 32 ビット C/C++ コンパイラ マクロ

コンパイラは各種のマクロ (接頭辞「_MCHP_」付きのマクロを含む) を定義します。これらはターゲットに固有のオプション、ターゲット プロセッサ、その他のホスト環境を指定します。

_MCHP_SZINT	32 または 64 (整数のサイズを指定するコマンドライン オプション(-mint32、-mint64) に基づく)
_MCHP_SZLONG	32 または 64 (整数のサイズを指定するコマンドライン オプション(-mlong32、-mlong64) に基づく)
_MCHP_SZPTR	常に 32 (全てのポインタは 32 ビットであるため)
__mchp_no_float	-mno-float が指定されている場合に定義されます。
__NO_FLOAT	-mno-float が指定されている場合に定義されます。
__PIC__ __pic__	変換ユニットを位置非依存コード向けにコンパイルします。
__PIC32MX __PIC32MX__	-mprocessor オプションで PIC32MX を指定した場合に定義されます。
__PIC32MZ	-mprocessor オプションで PIC32MZ を指定した場合に定義されます。
__PIC32_FEATURE_SET__	コンパイラは、選択されたデバイス上で利用可能な機能に基づいてマクロを定義します。これらのマクロを使うと、従来デバイスとの互換性を維持したまま新しいデバイスの機能が使えるようにコードを書く事ができます。例 : PIC32MX795F512L は __PIC32_FEATURE_SET__ == 795 を使用し、PIC32MX340F128H は __PIC32_FEATURE_SET__ == 340 を使用します。
PIC32MX	-ansi が指定されていない場合に定義されます。
__LANGUAGE_ASSEMBLY __LANGUAGE_ASSEMBLY__ _LANGUAGE_ASSEMBLY	前処理済みアセンブリ ファイル (.S ファイル) をコンパイルする場合に定義されます。
LANGUAGE_ASSEMBLY	-ansi が指定されていない状態で前処理済みアセンブリ ファイルをコンパイルする場合に定義されます。
__LANGUAGE_C __LANGUAGE_C__ _LANGUAGE_C	C ファイルをコンパイルする場合に定義されます。
LANGUAGE_C	-ansi が指定されていない状態で C ファイルをコンパイルする場合に定義されます。
__LANGUAGE_C_PLUS_PLUS __cplusplus _LANGUAGE_C_PLUS_PLUS__	C++ ファイルをコンパイルする場合に定義されます。
__EXCEPTIONS	X++ 例外が有効な場合に定義されます。
__GXX_RTTI	ランタイム型情報が有効な場合に定義されます。

__processor__	「processor」は -mprocessor オプションの引数を大文字にした変換した値です (例 : -mprocessor=32mx12f3456 は __32MX12F3456__ を定義します)。
__XC	コンパイラがMicrochip XC である事を示すために常に定義されます。
__XC32	コンパイラがMicrochip XC32である事を示すために常に定義されます。
__VERSION__	__VERSION__ マクロは、使用中のコンパイラを記述する文字列定数へ展開します。内容の書式は自由ですが、少なくともリリース番号を含む必要があります。数値のバージョン番号には __XC32_VERSION マクロを使います。
__XC32_VERSION??? __C32_VERSION__	C コンパイラは、定数 __XC32_VERSION を定義する事で、バージョン識別子に数値を提供します。このマクロを使う事で、コンパイラの旧バージョンとの下位互換性を維持したまま新バージョンの機能を利用できるようにアプリケーションを作成できます。この値は、最新リリースの主バージョン番号と副バージョン番号に基づきます。例えば、リリース バージョン 1.03 の場合、__XC32_VERSION の定義値は 1030 です。このマクロを標準のプリプロセッサ比較命令文と組み合わせて使う事で、バージョン コード コンストラクタを条件付きでインクルードできます。
__mips_dsp 1 __mips_dspr2 1 __mips_dsp_rev 2	選択されたターゲット デバイスが DSPr2 エンジンをサポートしている場合、C コンパイラはこれらの定数を定義します。

選択されたデバイスで利用可能な機能を特定するために使えるその他のマクロに関しては、デバイスに固有のインクルード ファイル (pic32mx/include/proc/p32*.h) も参照してください。これらのマクロは、ヘッダファイルの最後の方に記載されています。

© 2015 Microchip Technology Inc.

18.5.2 SDE 互換マクロ

MIPS® SDE (Software Development Environment) は各種のマクロを定義します。それらのほとんどには接頭辞「_MIPS_」が付き、各種のターゲット固有オプションを特性化します (その一部はコマンドライン オプションによって決まります (例 : -mint64))。SDEからコンパイラへのアプリケーションとミドルウェアの移植を容易にするため、コンパイラは適用可能な場合にこれらのマクロを定義します。

_MIPS_SZINT	32 または 64 (整数のサイズを指定するコマンドライン オプション (-mint32、-mint64) に基づく)
_MIPS_SZLONG	32 または 64 (整数のサイズを指定するコマンドライン オプション (-mlong32、-mlong64) に基づく)
_MIPS_SZPTR	常に 32 (全てのポインタは 32 ビットであるため)
__mips_no_float	-mno-float が指定されている場合に定義されます。
__mips__ _mips _MIPS_ARCH_PIC32MX _MIPS_TUNE_PIC32MX _R3000 __R3000 __R3000__ __mips_soft_float __MIPSEL __MIPSEL__ _MIPSEL	常に定義されます。
R3000 MIPSEL	-ansi が指定されていない場合に定義されます。
_mips_fpr	32 として定義されます。
__mips16	-mips16 が指定されている場合に定義されます。
__mips	32 として定義されます。
__mips_isa_rev	2 として定義されます。
_MIPS_ISA	_MIPS_ISA_MIPS32 として定義されます。

NOTE:

 © 2015 Microchip Technology Inc.

第 19 章　プログラムのリンク

19.1　はじめに

リンカスクリプトの詳細は『**MPLAB® XC32 Assembler, Linker and Utilities User's Guide**』(DS50002186) を参照してください。

本コンパイラは、中間ファイルの生成後に停止するよう要求されない限り、自動的にリンカを起動します。

リンカスクリプトは利用可能なメモリ領域と、それらの領域内でのセクションの配置を指定するために使います。

リンカは、セクションに割り当てられたメモリの詳細を示すマップファイルを生成します。マップファイルはメモリに関する最良の情報源です。

- ライブラリ シンボルの置換
- リンカによって定義されるシンボル
- 既定値リンカスクリプト

19.2　ライブラリ シンボルの置換

Microchip 社の MPLAB XC8 コンパイラとは異なり、MPLAB XC32 C/C++ コンパイラでは一部のライブラリ関数 (「弱い」ライブラリ関数 (**8.12「変数属性」**参照)) だけがユーザ定義ルーチンと置換可能です。「弱い」ライブラリ関数は、ユーザコード内に同じ名前のユーザ定義関数が存在すると置換されます。

19.3　リンカによって定義されるシンボル

32 ビットリンカは、C コードの開発に使える各種のシンボルを定義します。詳細は『**MPLAB® XC32 Assembler, Linker and Utilities User's Guide**』(DS50002186) を参照してください。

リンカはシンボル `_ramfunc_begin` および `_bmxdkpba_address` を定義します。これらは、RAM 内の RAM 関数の先頭アドレスと、プログラムメモリ内の対応するアドレス (そこから関数が RAM にコピーされる) を表します。プロジェクトに RAM 関数が存在する場合、既定値スタートアップ コードはこれらのシンボルを使ってバスマトリクスを初期化します (**12.3「関数コードのメモリ割り当て」**参照)。

リンカはシンボル `_stack` も定義します。スタートアップ コードは、このシンボルを使ってスタックポインタを初期化します。このシンボルは、ソフトウェア スタックの先頭アドレスを表します。

上記の全てのシンボルはスタートアップ コード以外で必要になる事はほとんどありませんが、ユーザが独自のスタートアップ コードを書く場合に役立ちます。

19.4 既定値リンカスクリプト

PIC32MX デバイスの場合

コマンドラインでリンカスクリプトを指定しないと、リンカは内部バージョン (ビルトイン既定値リンカスクリプト) を使います。既定値リンカスクリプトは、全ての PIC32 MCU に適したセクション マッピングを提供します。このリンカスクリプトは、INCLUDE ディレクティブを使ってデバイスに固有のメモリ領域をインクルードします。

既定値リンカスクリプトは、ほとんどの PIC32 MCU アプリケーションに適します。アプリケーションが特殊なメモリ割り当てを行う場合にのみ、そのアプリケーションに固有のリンカスクリプトが必要です。既定値リンカスクリプトの内容を調べるには、以下のように --verbose オプションを指定してリンカを起動します。

```
xc32-ld --verbose
```

ツールスイートを通常の方法でインストールした場合、既定値リンカスクリプトは \pic32mx\lib\ldscripts\elf32pic32mx.x にコピーされます。このファイルは既定値リンカスクリプトの単なるコピーである事に注意が必要です。リンカが実際に使うスクリプトはリンカの内部にあります。リンカスクリプトのデバイスに固有の部分は \pic32mx\lib\proc\device\procdefs.ld にあります (device は、-mprocessor コンパイラ ドライバ (xc32-gcc) オプションで指定されたデバイス値です)。

PIC32MZ およびそれ以後のデバイスの場合

PIC32MZ とそれ以後のデバイス向けのリンカスクリプトは 1 つのファイルに統合されています (例: pic32mx/lib/proc/32MZ2048ECH100/p32MZ2048ECH100.ld)。これに対し、従来のリンカスクリプト モデルは 2 つのファイル (elf32pic32mx.x と procdefs.ld) を使います。-mprocessor=device オプションを指定した場合、xc32-gcc コンパイル ドライバは、従来と同様にビルド時にデバイス固有リンカスクリプトをリンカに渡します。

既定値リンカスクリプトは以下のカテゴリの情報を含みます。

- 出力フォーマットとエントリポイント
- 最小スタック / ヒープサイズの既定値
- プロセッサ定義インクルード ファイル
  - プロセッサ固有オブジェクト ファイルのインクルード
  - オプションによるプロセッサ固有周辺モジュールライブラリのインクルード
  - 例外ベクタ ベースアドレスとベクタ間隔シンボル
  - メモリアドレスの定義
  - メモリ領域
  - コンフィグレーション ワードの入出力セクションマップ
- 入出力セクションマップ

Note:	リンカスクリプト内で指定された全てのアドレスは仮想アドレスです (物理アドレスではありません)。

© 2015 Microchip Technology Inc.

19.4.1 出力フォーマットとエントリポイント

既定値リンカスクリプトの最初の数行は、出力フォーマットとアプリケーションのエントリポイントを定義します。

```
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH(pic32mx)
ENTRY(_reset)
```

OUTPUT_FORMAT 行は、出力ファイル向けにオブジェクト ファイル フォーマットを選択します。32 ビット言語ツールが生成する出力オブジェクト ファイルのフォーマットは「traditional/ リトルエンディアン /MIPS/ELF32」です。

OUTPUT_ARCH 行は、出力ファイル向けに特定のデバイス アーキテクチャを選択します。32 ビット言語ツールが生成する出力ファイルは、そのファイルが PIC32 アーキテクチャ向けに生成された事を示す情報を格納します。

ENTRY 行は、アプリケーションのエントリポイントを選択します。これは最初に実行する命令の位置を指定するシンボルです。32 ビット言語ツールは、_reset ラベルが示す命令から実行を開始します。

19.4.2 最小スタック / ヒープサイズの既定値

PIC32MX ファミリだけが旧式の 2 ファイル式リンカスクリプトを使います。既定値リンカスクリプトの次のセクションは、最小スタック / ヒープサイズの既定値を提供します。

```
/*
 * Provide for a minimum stack and heap size
 * - _min_stack_size - represents the minimum space that must
 *                     be made available for the stack.Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 * - _min_heap_size  - represents the minimum space that must
 *                     be made available for the heap.Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 */
EXTERN (_min_stack_size _min_heap_size)
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
```

EXTERN行により、以後のリンカスクリプトは_min_stack_sizeと_min_heap_sizeの既定値を使います(ユーザがリンカの--defsymコマンドライン オプションを使ってこれらの値を上書きしないと想定した場合)。

PROVIDEの2行により、_min_stack_sizeと_min_heap_sizeの両方に既定値が設定されます。最小スタックサイズの既定値は 1024 バイト (0x400) です。最小ヒープサイズの既定値は 0 バイトです。

19.4.3 プロセッサ定義インクルード ファイル

既定値リンカスクリプト内の次の行は、プロセッサに固有の情報を取り込みます。

```
INCLUDE procdefs.ld
```

procdefs.ld ファイルは、この時点でリンカスクリプトにインクルードされます。このファイルは、現在作業中のディレクトリと -L コマンドライン オプションで指定されたディレクトリ内で検索されます。コンパイラシェルにより、-mprocessor コマンドライン オプションで選択されたプロセッサに基づく適切なディレクトリが -L コマンドライン オプションと一緒にリンカに渡されます。

プロセッサ定義リンカスクリプトは以下の情報を含みます。

- プロセッサ固有オブジェクト ファイルのインクルード
- 例外ベクタ ベースアドレスとベクタ間隔シンボル
- メモリアドレスの定義
- メモリ領域
- コンフィグレーション ワードの入出力セクションマップ

19.4.3.1 プロセッサ固有オブジェクト ファイルのインクルード

プロセッサ定義リンカスクリプトのこのセクションにより、プロセッサに固有のオブジェクト ファイルがリンクにインクルードされます。

```
/*************************************************************
 * Processor-specific object file.Contains SFR definitions.
 *************************************************************/
INPUT("processor.o")
```

INPUT 行により、processor.o ファイルは、コマンドラインでファイル名が指定されたかのようにリンクにインクルードされます。リンカは、現在作業中のディレクトリでこのファイルを検索します。ファイルが見付からない場合、リンカはライブラリ検索パス (-L コマンドライン オプションで指定) を検索します。

19.4.3.2 オプションによるプロセッサ固有周辺モジュールライブラリのインクルード

> **Note:** レガシーの周辺モジュール ライブラリは廃止され、その代わりに MPLAB Harmony ライブラリが別にインストールされます。

プロセッサ定義リンカスクリプトのこのセクションにより、プロセッサ固有の周辺モジュール ライブラリがリンクにインクルードされます。

```
/*******************************************************************
 * Processor-specific peripheral libraries are optional

*******************************************************************/
OPTIONAL("libmchp_peripheral.a")
OPTIONAL("libmchp_peripheral_32MX795F512L.a")
```

OPTIONAL 行により、libmchp_peripheral.a と libmchp_peripheral_32MX795F512L.a は、コマンドラインでそれらのファイル名が指定されたかのようにリンクにインクルードされます。リンカは、現在作業中のディレクトリでこれらのファイルを検索します。そこでファイルが見付からない場合、リンカはライブラリ検索パスを検索します。そこでもファイルが見付からない場合、リンクプロセスはエラーを生成せずに処理を続けます。リンカは、要求された周辺モジュール ライブラリからのシンボルがどこにも見付からない場合にのみ、エラーを生成します。

© 2015 Microchip Technology Inc.

19.4.3.3　例外ベクタ ベースアドレスとベクタ間隔シンボル

プロセッサ定義リンカスクリプトのこのセクションは、例外ベクタのベースアドレスとベクタ間隔の値を定義します。

```
/*************************************************************
 * For interrupt vector handling
 *************************************************************/
_vector_spacing= 0x00000001;
_ebase_address= 0x9FC01000;
```

最初の行は、_vector_spacing に対して 1 を定義します。例外向けに利用可能なメモリは、1 のベクタ間隔しかサポートしません。2 行目は、例外ベクタのベースアドレス (EBASE) を定義します。

一部のデバイスでは、例外ベクタのベースアドレスは KSEG0 ブートセグメント内に配置されます。それ以外のデバイスでは、KSEG0 ブートセグメントのサイズがベクタテーブルを格納するには十分ではないため、例外ベクタのベースアドレスは KSEG0 プログラム セグメント内に配置されます。一般的に、12KB 以上の KSEB0 ブートセグメントを備えたデバイスは、例外ベクタテーブル向けにブートフラッシュを使います。KSEB0 ブートセグメントが 12KB 未満のデバイスは、例外ベクタテーブル向けに KSEG0 プログラム セグメントを使います。使用するターゲット デバイスの既定値アドレスは、procdefs.ld インクルード ファイルで確認する必要があります。

19.4.3.4　メモリアドレスの定義

プロセッサ定義リンカスクリプトのこのセクションは、既定値リンカスクリプトに必要な特定メモリアドレスに関する情報を提供します。

```
/*************************************************************
 * Memory Address Equates
 *************************************************************/
_RESET_ADDR      = 0xBFC00000;
_BEV_EXCPT_ADDR  = 0xBFC00380;
_DBG_EXCPT_ADDR  = 0xBFC00480;
_DBG_CODE_ADDR   = 0xBFC02000;
_GEN_EXCPT_ADDR  = _ebase_address + 0x180;
```

_RESET_ADDR はプロセッサのリセットアドレスを定義します。これはカーネルモードでの IFM ブートセクションの仮想開始アドレスです。

_BEV_EXCPT_ADDR は、例外 ($Status_{BEV}$ = 1) が発生した時のプロセッサのジャンプ先アドレスを定義します。

_DBG_EXCPT_ADDR は、デバッグ例外が発生した時のプロセッサのジャンプ先アドレスを定義します。

_DBG_CODE_ADDR は、デバッグ エグゼクティブの開始アドレスを定義します。このアドレスは、ターゲット デバイスの KSEG0 ブートセグメントのサイズに応じて異なる場合があるので注意が必要です。

_GEN_EXCPT_ADDR は、例外 ($Status_{BEV}$ = 0) が発生した時のプロセッサのジャンプ先アドレスを定義します。

19.4.3.5　メモリ領域

プロセッサ定義リンカスクリプトのこのセクションは、デバイス上で利用可能なメモリ領域に関する情報を提供します。

```
/*************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for
 * orphaned sections.Only sections specifically assigned to
 * these regions can be allocated into these regions.
 *************************************************************/
MEMORY
{
  kseg0_program_mem (rx) :ORIGIN = 0x9D000000, LENGTH = 0x8000
  kseg0_boot_mem         :ORIGIN = 0x9FC00490, LENGTH = 0x970
  exception_mem          :ORIGIN = 0x9FC01000, LENGTH = 0x1000
  kseg1_boot_mem         :ORIGIN = 0xBFC00000, LENGTH = 0x490
  debug_exec_mem         :ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  config3                :ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                :ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                :ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                :ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  kseg1_data_mem         :ORIGIN = 0xA0000000, LENGTH = 0x2000
  sfrs                   :ORIGIN = 0xBF800000, LENGTH = 0x10000
}
```

Note: L1 キャッシュを備えたデバイスは `kseg1_data_mem` を使います。

以下の 11 個のメモリ領域の開始アドレスと長さを定義します。

1. アプリケーション コード向けのプログラムメモリ領域 (`kseg0_program_mem`)
2. ブートメモリ領域 (`kseg0_boot_mem` と `kseg1_boot_mem`)
3. 例外メモリ領域 (`exception_mem`)
4. デバッグ エグゼクティブ メモリ領域 (`debug_exec_mem`)
5. コンフィグレーション メモリ領域 (`config3`、`config2`、`config1`、`config0`)
6. データメモリ領域 (`kseg1_data_mem`)
7. SFR メモリ領域 (`sfrs`)

既定値リンカスクリプトは、これらの名前を使って各セクションを正しい領域に配置します。非標準セクションは不明セクションになります。これらの非標準セクションを配置するために、メモリ領域の属性を使います。属性 `rx` は、読み出し専用セクションまたは実行可能セクションがプログラムメモリ領域内に配置可能である事を指定します。同様に、属性 `w!x` は、読み出し専用でも実行可能でもないセクションがデータメモリ領域内に配置可能である事を指定します。ブートメモリ領域、コンフィグレーション メモリ領域、SFR メモリ領域には属性が指定されないため、これらの領域には特定のセクションだけが配置できます (すなわち、非標準セクションはブートメモリ領域、例外メモリ領域、コンフィグレーション メモリ領域、デバッグ エグゼクティブ メモリ領域、SFR メモリ領域に配置できません)。

© 2015 Microchip Technology Inc.

19.4.3.6　コンフィグレーション ワードの入出力セクションマップ

プロセッサ定義リンカスクリプトの最後のセクションは、コンフィグレーション ワード向けの入出力セクションマップです。このセクションマップは、既定値リンカスクリプト内の入出力セクションマップ (**19.4.4「入出力セクションマップ」**) を補足します。このセクションマップは、コンフィグレーション ワードの出力セクションに対する入力セクションの割り当てを定義します。入力セクションはソースコード内で定義されるアプリケーションの一部ですが、出力セクションはリンカによって生成されます。一般的に、複数の入力セクションを 1 つの出力セクションに割り当てる事ができます。リンカスクリプト内で 1 つの SECTIONS コマンドを使って全ての出力セクションをまとめて指定します。

プロセッサ上の各コンフィグレーション ワードごとに異なる出力セクションが存在します (セクション名 : .config_*address*、*address* はメモリ内のコンフィグレーション ワードのアドレス)。これらの各セクションは、#pragma config ディレクティブ (**18.4「プラグマ ディレクティブ」**参照) によってそのコンフィグレーション ワード向けに生成されたデータを格納します。各セクションはそれぞれのメモリ領域 (config*n*) に割り当てられます。

```
SECTIONS
{
  .config_BFC02FF0 :{
    *(.config_BFC02FF0)
  } > config3
  .config_BFC02FF4 :{
    *(.config_BFC02FF4)
  } > config2
  .config_BFC02FF8 :{
    *(.config_BFC02FF8)
  } > config1
  .config_BFC02FFC :{
    *(.config_BFC02FFC)
  } > config0
}
```

19.4.4 入出力セクションマップ

既定値リンカスクリプト内の最後のセクションは、入出力セクションマップです。セクションマップはリンカスクリプトの中心的部分であり、出力セクションに対する入力セクションの割り当てを定義します。入力セクションはソースコード内で定義されるアプリケーションの一部ですが、出力セクションはリンカによって生成されます。一般的に、複数の入力セクションを 1 つの出力セクションに割り当てる事ができます。全ての出力セクションは、リンカスクリプト内で 1 つの `SECTIONS` コマンドを使って指定します。

リンカは以下の出力セクションを生成します。

- .config_<address> セクション
- .reset セクション
- .bev_excpt セクション
- .dbg_excpt セクション
- .dbg_code セクション
- .app_excpt セクション
- .vector_0 ...vector_63 セクション (PIC32MX 割り込みベクタテーブル)
- .vectors セクション
- .text セクション
- C++ 初期化セクション
- .rodata セクション
- .sdata2 セクション
- .sbss2 セクション
- .dbg_data セクション
- .data セクション
- .got セクション
- .sdata セクション
- .lit8 セクション
- .lit4 セクション
- .sbss セクション
- .bss セクション
- .heap セクション
- .stack セクション
- .ramfunc セクション
- スタックの位置
- デバッグ セクション
- L1 キャッシュ適用メモリに割り当てられる変数
- .tlb_init_values セクション

© 2015 Microchip Technology Inc.

19.4.4.1 .config_<address> セクション

これらのセクションは、コンフィグレーション ワードをターゲット デバイス上の対応する絶対アドレスにマッピングします。コンパイラの config プラグマは、この命名規則に従って入力セクションを生成します。リンカスクリプトは生成された入力セクションを、対応する絶対アドレスにマッピングされた出力セクションに割り当てます。

19.4.4.2 .reset セクション

このセクションは、プロセッサのリセット時に実行するコードを格納します。このセクションはプロセッサ定義リンカスクリプトで指定されたリセットアドレス (_RESET_ADDR) に配置され、ブートメモリ領域 (kseg1_boot_mem) に割り当てられます。.reset 出力セクションは、.reset.startup 入力セクションからの C スタートアップ コードも格納します。

```
.reset _RESET_ADDR :
{
  KEEP(*(.reset))
  KEEP(*(.reset.startup))
 } > kseg1_boot_mem
```

19.4.4.3 .bev_excpt セクション

このセクションは、$Status_{BEV}$ = 1 の時に発生する例外のハンドラを格納します。このセクションはプロセッサ定義リンカスクリプトで指定された BEV 例外アドレス (_BEV_EXCPT_ADDR) に配置され、ブートメモリ領域 (kseg1_boot_mem) に割り当てられます。

```
.bev_excpt _BEV_EXCPT_ADDR :
{
  (*(.bev_handler))
} > kseg1_boot_mem
```

19.4.4.4 .dbg_excpt セクション

このセクションは、デバッグ例外ベクタ向けの空間を予約します。このセクションは、シンボル _DEBUGGER が定義されている場合にのみ割り当てられます (このシンボルは、シェルに対して -mdebugger コマンドライン オプションが指定されている場合に定義されます)。このセクションは、プロセッサ定義リンカスクリプトの指定に従ってデバッグ例外アドレス (_DBG_EXCPT_ADDR) に配置され、ブートメモリ領域 (kseg1_boot_mem) に割り当てられます。このセクションは、デバッグ エグゼクティブ向けに予約されているアドレスにアプリケーション コードが配置されないようにする事だけを目的とするため、NOLOAD としてマークされます。

```
.dbg_excpt _DBG_EXCPT_ADDR (NOLOAD) :
{
  .+= (DEFINED (_DEBUGGER) ?0x8 :0x0);
} > kseg1_boot_mem
```

19.4.4.5 .dbg_code セクション

このセクションは、デバッグ例外ハンドラ向けの空間を予約します。このセクションは、シンボル _DEBUGGER が定義されている場合にのみ割り当てられます (このシンボルは、シェルに対して -mdebugger コマンドライン オプションが指定されている場合に定義されます)。このセクションは、プロセッサ定義リンカスクリプトの指定に従ってデバッグコード アドレス (_DBG_CODE_ADDR) に配置され、デバッグ エグゼクティブ領域 (debug_exec_mem) に割り当てられます。このセクションは、デバッグ エグゼクティブ向けに予約されているアドレスにアプリケーション コードが配置されないようにする事だけを目的とするため、NOLOAD としてマークされます。

```
.dbg_code _DBG_CODE_ADDR (NOLOAD) :
{
  .+= (DEFINED (_DEBUGGER) ?0xFF0 :0x0);
} > debug_exec_mem
```

19.4.4.6 .app_excpt セクション

このセクションは、$Status_{BEV}$ = 0 の時に発生する例外のハンドラを格納します。このセクションは、プロセッサ定義リンカスクリプトの指定に従って一般例外アドレス (_GEN_EXCPT_ADDR) に配置され、例外メモリ領域 (exception_mem) に割り当てられます。

```
.app_excpt _GEN_EXCPT_ADDR :
{
  KEEP(*(.gen_handler))
} > exception_mem
```

19.4.4.7 .vector_0 ...vector_63 セクション (PIC32MX 割り込みベクタテーブル)

PIC32MX デバイスが使う割り込みベクタ コントローラは、等間隔 (間隔はユーザが定義) に配置された 64 個の割り込みベクタを提供します。

これらのデバイスでは、テーブル内の各ベクタは出力セクションとして生成され、_ebase_address および _vector_spacing シンボルの値に基づいて絶対アドレスに配置されます。テーブル内の 64 個のベクタのそれぞれに対して 1 つの .vector_n 出力セクションが存在します。

これらのセクションは、各割り込みベクタに対応するハンドラを格納します。これらのセクションは、下式に従って正しいベクタアドレスに配置されます。

```
_ebase_address + 0x200 + (_vector_spacing << 5) * n
```

n は各ベクタの番号です。

各セクションの後の ASSERT により、ベクタに配置するコードが指定されたベクタ間隔を超えないようにします。

```
.vector_n _ebase_address + 0x200 + (_vector_spacing << 5) * n :
  {
    KEEP(*(.vector_n))
  } > exception_mem
ASSERT (SIZEOF(.vector_n) < (_vector_spacing << 5), "function at
exception vector n too large")
```


© 2015 Microchip Technology Inc.

19.4.4.8 .vectors セクション

一部の PIC32 ファミリは、ベクタ間隔の可変オフセット機能を備えています。この機能を使うと、アプリケーションの要件に合わせて割り込みベクタ間隔を設定できます。これを行うには、対応する OFFxxx レジスタを使って、各ベクタに固有の割り込みベクタオフセットを設定します。割り込みベクタテーブルの可変オフセット機能の詳細は『**PIC32 ファミリ リファレンス マニュアル、セクション 08. 割り込み**』(DS61108) と各 PIC32 MCU のデータシートを参照してください。

各割り込みベクタの入力セクション (.vector_n) は、アプリケーション コードで生成する必要があります。C/C++ コンパイラは、割り込みサービスルーチンに vector(n) または at_vector(n) 属性が適用されている場合に、このセクションを生成します。アセンブリコードでは、.section ディレクティブを使って新しい名前のセクションを生成します。

デバイス固有リンカスクリプトは、.vectors という名前の 1 つの出力セクションを生成し、そこにプロジェクトからの全ての .vector_n 入力セクションを割り当てます。割り込みベクタテーブルの開始アドレスは _ebase_address + 0x200 に設定されます。_ebase_address シンボルの既定値も、リンカスクリプト内で提供されます。

各ベクタに対してリンカスクリプトは __vector_offset_n という名前のシンボルも生成します。このシンボルの値は、_ebase_address アドレスからのベクタアドレスのオフセットです。

```
PROVIDE(_ebase_address = 0x9D000000);
SECTIONS {
 /* Interrupt vector table with vector offsets */
 .vectors _ebase_address + 0x200 :
 {
 /* Symbol __vector_offset_n points to .vector_n if it exists,
  * otherwise points to  the default handler.The
  * vector_offset_init.o module then provides a .data section
  * containing values used to initialize the vector-offset SFRs
  * in the crt0 startup code.
  */
 __vector_offset_0 = (DEFINED(__vector_dispatch_0) ?(.-
_ebase_address) :__vector_offset_default); KEEP(*(.vector_0))
 __vector_offset_1 = (DEFINED(__vector_dispatch_1) ?(.-
_ebase_address) :__vector_offset_default); KEEP(*(.vector_1))
 __vector_offset_2 = (DEFINED(__vector_dispatch_2) ?(.-
_ebase_address) :__vector_offset_default); KEEP(*(.vector_2))
 /* ...*/
 __vector_offset_190 = (DEFINED(__vector_dispatch_190) ?(.-
_ebase_address) :__vector_offset_default); KEEP(*(.vector_190))
 }
}
```

ベクタオフセット初期化モジュール (vector_offset_init.o) は、既定値リンカスクリプトで定義された __vector_offset_n シンボルを使います。各シンボルの値は、ebase レジスタのアドレスからのベクタアドレスのオフセットです。ベクタオフセット初期化モジュールは、このシンボル値を使って .data セクションを生成します (対応する OFFxxx 特殊機能レジスタのアドレスを使用)。つまり、標準リンカによって生成されたデータ初期化テンプレートは、OFFxxx レジスタの初期化に使う値を格納します。

プロジェクトに追加されたこれらの .data セクションと、リンカによって生成されたデータ初期化テンプレートにより、標準スタートアップ コードは OFFxxx 特殊機能レジスタを標準の初期化データとして初期化します。スタートアップ コードには OFFxxx レジスタを初期化するための特別なコードは不要です。

19.4.4.9 .text セクション

標準の実行可能コードセクションは、.text 出力セクションに割り当てられなくなりました。しかし、以下に示すように、いくつかの特殊な実行可能セクションは、現在でもこの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領域 (kseg0_program_mem) に割り当てられ、値は NOP (0) で埋められます。

ビルトイン リンカスクリプトは標準の .text 実行可能コード入力セクションをマッピングしなくなりました。これらのセクションをリンカスクリプト内でマッピングしない事により、シーケンシャル アロケータの代わりにベストフィット アロケータを使ってこれらのセクションを割り当てる事ができます。リンカスクリプト内でマッピングされていないセクションは、コード内で指定された絶対アドレス セクションの前後に流し込めますが、リンカスクリプトでマッピングされたセクションは互いにグループ化されて連続的に割り当てられるため、絶対アドレス セクション (address 関数属性を使用) と競合する可能性があります。

```
.text ORIGIN(kseg0_program_mem) :
{
*(.stub .gnu.linkonce.t.*)
  KEEP (*(.text.*personality*))
  *(.gnu.warning)
  *(.mips16.fn.*)
  *(.mips16.call.*)
} > kseg0_program_mem =0
```

19.4.4.10 C++ 初期化セクション

.init、.preinit_array、.init_array、.fini_array、.ctors、.dtors セクションは、全てファイルスコープ静的ストレージ C++ オブジェクトのコンストラクションとデストラクションに使われます。

```
  /* Global-namespace object initialization */
  .init   :
  {
    KEEP (*crti.o(.init))
    KEEP (*crtbegin.o(.init))
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o *crtn.o ).init))
    KEEP (*crtend.o(.init))
    KEEP (*crtn.o(.init))
    .= ALIGN(4) ;
  } >kseg0_program_mem
  .fini   :
  {
    KEEP (*(.fini))
    .= ALIGN(4) ;
  } >kseg0_program_mem
  .preinit_array   :
  {
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);
    .= ALIGN(4) ;
  } >kseg0_program_mem
  .init_array   :
  {
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array))
    PROVIDE_HIDDEN (__init_array_end = .);
    .= ALIGN(4) ;
  } >kseg0_program_mem
  .fini_array   :
```

 © 2015 Microchip Technology Inc.

```
  {
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(SORT(.fini_array.*)))
    KEEP (*(.fini_array))
    PROVIDE_HIDDEN (__fini_array_end = .);
    .= ALIGN(4) ;
  } >kseg0_program_mem
  .ctors   :
  {
    /* XC32 uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*crtbegin?.o(.ctors))
    /* We don't want to include the .ctor section from
       the crtend.o file until after the sorted ctors.
       The .ctor section from the crtend file contains the
       end of ctors marker and it must be last */
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
    .= ALIGN(4) ;
  } >kseg0_program_mem
  .dtors   :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*crtbegin?.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
    .= ALIGN(4) ;
  } >kseg0_program_mem
```

Note: 各出力セクション内の入力セクションの順序には意味があります。

19.4.4.11 .rodata セクション

標準の読み出し専用セクションは、リンカスクリプト内でマッピングされません。いくつかの特殊な読み出し専用セクションはリンカスクリプト内でマッピングされますが、ほとんどのセクションはマッピングされません (ベストフィット アロケータによる処理が可能です)。このセクションはプログラムメモリ領域 (kseg0_program_mem) に割り当てられます。

```
.rodata   :
{
  *(.gnu.linkonce.r.*)
  *(.rodata1)
} > kseg0_program_mem
```

19.4.4.12 .sdata2 セクション

このセクションは、アプリケーションの全ての入力ファイルからの「小さな」初期化定数 (グローバルで静的なデータ) を集約します。これらのデータは定数であるため、このセクションも読み出し専用セクションです。このセクションはプログラムメモリ領域 (kseg0_program_mem) に割り当てられます。

```
/*
 * Small initialized constant global and static data can be
 * placed in the .sdata2 section.This is different from
 * .sdata, which contains small initialized non-constant
 * global and static data.
 */
.sdata2   :
{
  *(.sdata2 .sdata2.*.gnu.linkonce.s2.*)
} > kseg0_program_mem
```

19.4.4.13 .sbss2 セクション

このセクションは、アプリケーションの全ての入力ファイルからの「小さな」非初期化定数 (グローバルで静的なデータ) を集約します。データは定数であるため、このセクションも読み出し専用セクションです。このセクションはプログラムメモリ領域 (kseg0_program_mem) に割り当てられます。

```
/*
 * Uninitialized constant global and static data (i.e.,
 * variables which will always be zero).Again, this is
 * different from .sbss, which contains small non-initialized,
 * non-constant global and static data.
 */
.sbss2   :
{
  *(.sbss2 .sbss2.*.gnu.linkonce.sb2.*)
} > kseg0_program_mem
```

19.4.4.14 .dbg_data セクション

このセクションは、デバッグ例外ハンドラが要求するデータ向けに空間を予約します。このセクションは、シンボル _DEBUGGER が定義されている場合にのみ割り当てられます (このシンボルは、シェルに対して -mdebugger コマンドライン オプションが指定されている場合に定義されます)。このセクションはデータメモリ領域 (kseg1_data_mem) に割り当てられます。このセクションは、デバッグ エグゼクティブ向けに予約されているアドレスにアプリケーション データが配置されないようにする事だけを目的とするため、NOLOAD としてマークされます。

```
.dbg_data (NOLOAD) :
{
  .+= (DEFINED (_DEBUGGER) ?0x200 :0x0);
} > kseg1_data_mem
```

19.4.4.15 .data セクション

リンカは、C スタートアップ コードが変数を初期化するために使うデータ初期化テンプレートを生成します。

© 2015 Microchip Technology Inc.

19.4.4.16 .got セクション

このセクションは、アプリケーションの全ての入力ファイルからグローバル オフセット テーブルを集約します。このセクションは、プログラムメモリ領域 (kseg0_program_mem) 内に配置されたロードアドレスを使ってデータメモリ領域 (kseg1_data_mem) に割り当てられます。グローバル ポインタ (_gp) の位置を表すためにシンボルが定義されます。

```
  _gp = ALIGN(16) + 0x7FF0 ;
  .got   :
  {
     *(.got.plt) *(.got)
  } > kseg1_data_mem AT> kseg0_program_mem
```

19.4.4.17 .sdata セクション

このセクションは、アプリケーションの全ての入力ファイルから「小さな」初期化データを集約します。このセクションは、プログラムメモリ領域 (kseg0_program_mem) 内に配置されたロードアドレスを使ってデータメモリ領域 (kseg1_data_mem) に割り当てられます。このセクションの仮想開始アドレス (_sdata_begin) と仮想終了アドレス (_sdata_end) を表すためにシンボルが定義されます。

```
/*
 * We want the small data sections together, so
 * single-instruction offsets can access them all, and
 * initialized data all before uninitialized, so
 * we can shorten the on-disk segment size.
 */
.sdata   :
{
  _sdata_begin = .;
  *(.sdata .sdata.*.gnu.linkonce.s.*)
  _sdata_end = .;
} > kseg1_data_mem AT> kseg0_program_mem
```

19.4.4.18 .lit8 セクション

このセクションは、アセンブラが命令ストリームではなくメモリ内に保存すると決定した 8 バイト定数を、アプリケーションの全ての入力ファイルから集約します。このセクションは、プログラムメモリ領域 (kseg0_program_mem) 内に配置されたロードアドレスを使ってデータメモリ領域 (kseg1_data_mem) に割り当てられます。

```
.lit8           :
{
  *(.lit8)
} > kseg1_data_mem AT> kseg0_program_mem
```

19.4.4.19 .lit4 セクション

このセクションは、アセンブラが命令ストリームではなくメモリに保存すると決定した 4 バイト定数を、アプリケーションの全ての入力ファイルから集約します。このセクションは、プログラムメモリ領域 (kseg0_program_mem) 内に配置されたロードアドレスを使ってデータメモリ領域 (kseg1_data_mem) に割り当てられます。初期化が必要なデータの仮想終了アドレス (_data_end) を表すためにシンボルが定義されます。

```
  .lit4           :
  {
    *(.lit4)
  } > kseg1_data_mem AT> kseg0_program_mem
  _data_end = .;
```

© 2015 Microchip Technology Inc.

19.4.4.20 .sbss セクション

このセクションは、アプリケーションの全ての入力ファイルから「小さな」非初期化データを集約します。このセクションはデータメモリ領域 (kseg1_data_mem) に割り当てられます。非初期化データの仮想開始アドレス (_bss_begin) を表すためにシンボルが定義されます。このセクションの仮想開始アドレス (_sbss_begin) と仮想終了アドレス (_sbss_end) を表すためのシンボルも定義されます。

```
  _bss_begin = .;
  .sbss   :
  {
    _sbss_begin = .;
    *(.dynsbss)
    *(.sbss .sbss.*.gnu.linkonce.sb.*)
    *(.scommon)
    _sbss_end = .;
  } > kseg1_data_mem
```

19.4.4.21 .bss セクション

このセクションは、アプリケーションの全ての入力ファイルから非初期化データを集約します。このセクションはデータメモリ領域 (kseg1_data_mem) に割り当てられます。非初期化データの仮想終了アドレス (_bss_end) を表すためにシンボルが定義されます。データメモリの仮想終了アドレス (_end) を表すためのシンボルも定義されます。

```
.bss   :
{
  *(.dynbss)
  *(.bss .bss.*.gnu.linkonce.b.*)
  *(COMMON)
  /*
   * Align here to ensure that the .bss section occupies
   * space up to _end.Align after .bss to ensure correct
   * alignment even if the .bss section disappears because
   * there are no input sections.
   */
  .= ALIGN(32 / 8) ;
} > kseg1_data_mem
.= ALIGN(32 / 8) ;
_end = .;
_bss_end = .;
```

19.4.4.22 .heap セクション

リンカは、ヒープ向けのメモリ領域を動的に確保するようになりました。.heap セクションは、リンカスクリプト内でマッピングされません。リンカは、他の全てのセクションを割り当てた後に、メモリ内で最大の未使用ギャップをヒープとスタックの両方に使います。ヒープ用に予約する空間の最小サイズはシンボル _min_heap_size で定義します。

19.4.4.23 .stack セクション

リンカは、スタック向けのメモリ領域を動的に確保するようになりました。.stack セクションは、リンカスクリプト内でマッピングされません。リンカは、他の全てのセクションを割り当てた後に、メモリ内で最大の未使用ギャップをヒープとスタックの両方に使います。スタック用に予約する空間の最小サイズはシンボル _min_stack_size で定義します。

© 2015 Microchip Technology Inc.

19.4.4.24 .ramfunc セクション

リンカは、ramfunc 属性を持つ「.ramfunc」という名前が付いたセクションを集め、それらをメモリの適切なレンジに順番に割り当てます。最初の ramfunc 属性付き関数は、適切にアラインメントされた最も高いアドレスに配置されます。

ramfunc セクションが存在すると、リンカはPIC32 バスマトリクスを適切に初期化するために、crt0.S スタートアップ コードに必要なシンボルを生成します。

```
/*
 * RAM functions go at the end of our stack and heap allocation.
 * Alignment of 2K required by the boundary register (BMXDKPBA).
 *
 * RAM functions are now allocated by the linker.The linker generates
 * _ramfunc_begin and _bmxdkpba_address symbols depending on the
 * location of RAM functions.
 */

_bmxdudba_address = LENGTH(kseg1_data_mem) ;
_bmxdupba_address = LENGTH(kseg1_data_mem) ;
```

19.4.4.25 スタックの位置

スタックポインタ (_stack) の位置を表すためにシンボルが定義されます。既に説明したように、ヒープとスタックは他のセクションが割り当てられた後に、メモリ内の最大ギャップに割り当てられます。

64K より大きなデータメモリを備えた PIC32 デバイスでは、GP 相対アドレス指定モードを使うべきではありません。リンカが生成したシンボルに対して GP 相対アドレス指定の使用が衝突する事を防ぐために、それらのシンボルは _linkergenerated セクションに割り当てます。

```
extern unsigned int
__attribute__((section("_linkergenerated"))) _splim;
```

19.4.4.26 デバッグ セクション

デバッグ セクションは DWARF2 デバッグ情報を格納します。それらはプログラム フラッシュに書き込まれません。

```
  /* Stabs debugging sections.*/
  .stab          0 :{ *(.stab) }
  .stabstr       0 :{ *(.stabstr) }
  .stab.excl     0 :{ *(.stab.excl) }
  .stab.exclstr  0 :{ *(.stab.exclstr) }
  .stab.index    0 :{ *(.stab.index) }
  .stab.indexstr 0 :{ *(.stab.indexstr) }
  .comment       0 :{ *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the
beginning
     of the section so we begin them at 0.*/
  /* DWARF 1 */
  .debug          0 :{ *(.debug) }
  .line           0 :{ *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 :{ *(.debug_srcinfo) }
  .debug_sfnames  0 :{ *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 :{ *(.debug_aranges) }
  .debug_pubnames 0 :{ *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 :{ *(.debug_info .gnu.linkonce.wi.*)}
  .debug_abbrev   0 :{ *(.debug_abbrev) }
  .debug_line     0 :{ *(.debug_line) }
  .debug_frame    0 :{ *(.debug_frame) }
  .debug_str      0 :{ *(.debug_str) }
  .debug_loc      0 :{ *(.debug_loc) }
  .debug_macinfo  0 :{ *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 :{ *(.debug_weaknames) }
  .debug_funcnames 0 :{ *(.debug_funcnames) }
  .debug_typenames 0 :{ *(.debug_typenames) }
  .debug_varnames  0 :{ *(.debug_varnames) }
  .debug_pubtypes 0 :{ *(.debug_pubtypes) }
  .debug_ranges   0 :{ *(.debug_ranges) }
  /DISCARD/ :{ *(.rel.dyn) }
  .gnu.attributes 0 :{ KEEP (*(.gnu.attributes)) }
  /DISCARD/ :{ *(.note.GNU-stack) }
  /DISCARD/ :{ *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*)
*(.discard) }
```

19.4.4.27 L1 キャッシュ適用メモリに割り当てられる変数

L1 データキャッシュを備えたデバイスでは、データ変数は KSEG0 データメモリ領域 (kseg0_data_mem) に割り当てられ、L1 キャッシュ経由のアクセスが可能です。同様に、リンカが割り当てるヒープとスタックは KSEG0 領域に割り当てられます。スタートアップ コードは、リンカスクリプト内でベースアドレスを定義しているシンボルを使って L1 キャッシュを初期化します。

例 :

```
EXTERN  (__pic32_init_cache_program_base_addr)
PROVIDE (__pic32_init_cache_program_base_addr = 0x9D000000) ;
EXTERN  (__pic32_init_cache_data_base_addr)
PROVIDE (__pic32_init_cache_data_base_addr = 0x80000000) ;
```


© 2015 Microchip Technology Inc.

19.4.4.28 .tlb_init_values セクション

PIC32 には、TLB (Translation Lookaside Buffer) 付きのメモリ管理ユニット (MMU) を備えたデバイスがあります。それらのデバイスの一部では、KSEG2/KSEG3 領域の特定レンジが SQI (Serial Quad Interface) および / または EBI (External Bus Interface) 専用としてデータシートに記載されています。

これらのデバイスでは、既定値スタートアップ コードは、TLB をこの専用メモリ マッピング向けに初期化するモジュールを呼び出します。TLB　初期化モジュール (`pic32_init_tlb_ebi_sqi.o`) は、このセクションによって生成されたテーブルを使って TLB を初期化します。

この書式の詳細については、`pic32-libs/libpic32/stubs` ディレクトリに保存されている `pic32_init_tlb_ebi_sqi.S` ソースファイルのコピーを参照してください。

NOTE:

© 2015 Microchip Technology Inc.

補遺 A. 組み込みコンパイラ互換モード

A.1 はじめに

MPLAB XC 8/16/32 C コンパイラは全て互換モードに設定できます。このモードでは、他社製組み込みコンパイラの規格外 C 言語拡張との構文的な互換性が得られます。この互換性により、他のコンパイラ向けに書かれた C ソースコードは、最小限の変更で MPLAB XC コンパイラを使ってコンパイルできます。

他のコンパイラがターゲットとするデバイスのアーキテクチャは大きく異なため、規格外拡張の意味は MPLAB XC コンパイラと異なる可能性があります。補遺 A では、どのような場合にオリジナルの C コードを見直す必要があるのか説明します。

また、MPLAB C コンパイラが提供する互換機能について、以下の項目を挙げて説明します。

- 互換モードでのコンパイル
- 構文互換性
- データ型
- 演算子
- 拡張キーワード
- 組み込み関数
- プラグマ

A.2 互換モードでのコンパイル

オプションにより、どのコンパイラとの構文互換性を有効にするのか指定します。このオプションは、`--ext=`*`vendor`* (MPLAB XC8 の場合) または `-mext=`*`vendor`* (MPLAB XC16/32 の場合) です。引数 *`vendor`* は構文を表すキーです。表 A-1 に、MPLAB XC コンパイラで使える全てのキーを示します。

表 A-1: vendor キー

vendor キー	構文	XC8 による サポート	XC16 による サポート	XC32 による サポート
`cci`	CCI (Common C Interface)	サポート	サポート	サポート
`iar`	ARM 向け IAR C/C++ Compiler™	サポート	サポート	サポート

Common C Interface は、全ての Microchip MPLAB XC コンパイラに共通の言語規格です。この構文に関連する規格外拡張については、既に第 2 章「**CCI (Common C Interface)**」で説明したので、ここでは繰り返し説明しません。

A.3 構文互換性

この構文互換機能の目標は､他のCコンパイラ向けに書かれたソースコードをMPLAB XC コンパイラに固有の構文に移植する際の移行プロセスを容易にする事です。

Microchip 社製デバイス向けにコンパイルする場合、多くの規格外拡張は不要であるため、MPLAB XC コンパイラはこれらに等価な拡張を提供しません。MPLAB XC コンパイラはこれらの拡張を単純に無視しますが、通常は警告メッセージを生成してユーザにコンパイラの挙動が異なる事を知らせます。ユーザは、これらの機能が無効であってもプロジェクトが正常に動作する事を確認する必要があります。

その他の規格外拡張は、Microchip 社製デバイスに対して非互換です。これらの拡張をソースコードから削除しないと、MPLAB XC コンパイラはエラーを生成します。ユーザは、これらの拡張を削除した事で予期しない結果が生じないか調査すると共に、プロジェクト内の他のソースコードを変更する必要があるかどうか判断する必要があります。

表 A-2 に、本補遺内の各表に適用する互換レベルの一覧を示します。

表 A-2: サポートする互換レベルの指標

レベル	説明
サポート	その構文は指定された互換モードでサポートされ、元のコンパイラでの意味が再現されます。
サポート (引数無視)	プラグマの場合、ベースプラグマは指定された互換モードでサポートされますが、引数は無視されます。
ネイティブ サポート	その構文は MPLAB XC コンパイラが既にサポートしている構文と等価であり、意味は互換です。この機能はベンダー互換モードを有効にしなくても使えます。
無視	その構文は指定された互換モードでサポートされますが、対応する動作は要求または実行されません。コンパイラは拡張機能を無視し、警告を出力します。
エラー	その構文は指定された互換モードでサポートされません。エラーが出力され、コンパイルは終了します。

たとえ MPLAB XC コンパイラがサポートしている C 機能であっても、Microchip 社製デバイス向けにコンパイルした時点でアドレス、レジスタ名、アセンブリ命令、その他のデバイスに固有の引数が有効であるとは見込めません。これらを使うコードは、Microchip 社製ターゲット デバイスのデータシートと照合して見直す必要があります。

© 2015 Microchip Technology Inc.

A.4 データ型

一部のコンパイラは、C99 ANSI 規格の仕様に従う論理型 `bool` とその値 `true` および `false` をサポートします。表 A-3 に示すように、全ての MPLAB XC コンパイラは互換モードでこの型と値をサポートします[1]。

この機能を MPLAB XC コンパイラで使う場合、ANSI 規格に従って `<stdbool.h>` ヘッダをこの機能向けにインクルードする必要があります。

表 A-3: C99 bool 型のサポート

IAR 互換モード			
型	**XC8**	**XC16**	**XC32**
`bool`	サポート	サポート	サポート

論理型 `bool` と MPLAB XC8 が実装している整数型 `bit` を混同しないよう注意が必要です。

A.5 演算子

他のコンパイラでは、`@` 演算子を使ってオブジェクトのメモリ位置を指定できます。表 A-4 に示すように、MPLAB XC8 だけがこの構文をサポートします。

別のデバイスで指定したアドレスが新しいアーキテクチャでも正しいとは見込めません。Microchip 社製ターゲット デバイスのデータシートと照合してアドレスを見直す必要があります。

MPLAB XC8 の互換モードで「`@`」使った場合、正しく動作しますが警告が生成されます。この警告が再び生成されないようにするには、見直したアドレスに対して「`@`」の代わりに MPLAB C の `__at()` 指定子を使います。

MPLAB XC16/32 コンパイラでは、`address` 属性の使用を検討する必要があります。

表 A-4: 規格外演算子のサポート

IAR 互換モード			
演算子	**XC8**	**XC16**	**XC32**
`@`	ネイティブ サポート	エラー	エラー

1. 全ての MPLAB XC コンパイラが全ての C99 機能を採用しているわけではありません。

A.6 拡張キーワード

多くの場合、規格外拡張はオブジェクトの定義方法またはアクセス方法を指定します。通常、これにはキーワードを使います。他のコンパイラに対応する規格外 C キーワードと MPLAB XC コンパイラが提供する互換レベルの一覧を表 A-5 に示します。表の下の Note に、一部の拡張に関する追加の情報を記載しています。

表 A-5: 規格外キーワードのサポート

IAR 互換モード			
キーワード	XC8	XC16	XC32
__section_begin	無視	サポート	サポート
__section_end	無視	サポート	サポート
__section_size	無視	サポート	サポート
__segment_begin	無視	サポート	サポート
__segment_end	無視	サポート	サポート
__segment_size	無視	サポート	サポート
__sfb	無視	サポート	サポート
__sfe	無視	サポート	サポート
__sfs	無視	サポート	サポート
__asm または asm[1]	サポート [2]	ネイティブ サポート	ネイティブ サポート
__arm	無視	無視	無視
__big_endian	エラー	エラー	エラー
__fiq	サポート	エラー	エラー
__intrinsic	無視	無視	無視
__interwork	無視	無視	無視
__irq	サポート	エラー	エラー
__little_endian[3]	無視	無視	無視
__nested	無視	無視	無視
__no_init	サポート	サポート	サポート
__noreturn	無視	サポート	サポート
__ramfunc	無視	無視	サポート [4]
__packed	無視 [5]	サポート	サポート
__root	無視	サポート	サポート
__swi	無視	無視	無視
__task	無視	サポート	サポート
__weak	無視	サポート	サポート
__thumb	無視	無視	無視
__farfunc	無視	無視	無視
__huge	無視	無視	無視
__nearfunc	無視	無視	無視
__inline	サポート	ネイティブ サポート	ネイティブ サポート

Note 1: この構文によって指定された全てのアセンブリコードはデバイスに固有であり、Microchip 社製デバイスに移植する際は常に見直しが必要です。

2: MPLAB XC8 の場合、キーワード asm の互換レベルは「ネイティブ サポート」ですが、IAR 互換モードでは __asm キーワードだけをサポートします。

3: これは全ての MPLAB XC コンパイラが使う既定値の (そして唯一の) エンディアンです。

4: これを MPLAB XC32 で使う場合、完全な互換性を得るには __longcall__ マクロを一緒に使う必要があります。

5: MPLAB XC8 の場合、このキーワードの互換レベルは「無視」ですが、既定値によって全ての構造体がパッキングされるため、機能的な問題はありません。

© 2015 Microchip Technology Inc.

A.7 組み込み関数

組み込み関数を使うと、ソースコード内で共通タスクを実行できます。他のコンパイラの組み込み関数に対する MPLAB XC コンパイラのサポートを表 A-6 に示します。

表 A-6: 規格外組み込み関数のサポート

IAR 互換モード			
関数	**XC8**	**XC16**	**XC32**
__disable_fiq[1]	サポート	無視	無視
__disable_interrupt	サポート	サポート	サポート
__disable_irq[1]	サポート	無視	無視
__enable_fiq[1]	サポート	無視	無視
__enable_interrupt	サポート	サポート	サポート
__enable_irq[1]	サポート	無視	無視
__get_interrupt_state	無視	サポート	サポート
__set_interrupt_state	無視	サポート	サポート

Note 1: これらの組み込み関数は、8 ビット PIC® デバイス上のグローバル割り込みイネーブルビットを無効または有効にするためのマクロに対応します。

サポートする関数が正常に動作するには、ヘッダファイル `<xc.h>` をインクリードする必要があります。

A.8 プラグマ

コンパイラは、コード生成を制御するためにプラグマを使います。どのコンパイラも未知のプラグマを無視しますが、他のコンパイラが実装しているプラグマの多くは互換モードの MPLAB XC コンパイラも実装しています。表 A-7 に、各種のプラグマに対する MPLAB XC コンパイラのサポートレベルを示します。

これらのプラグマの多くは引数を使います。MPLAB XC コンパイラがあるプラグマをサポートしていても、そのプラグマの全ての引数をサポートするとは限りません。表には引数をサポートするかどうかも記載しています。

表 A-7: 規格外プラグマのサポート

IAR 互換モード			
プラグマ	XC8	XC16	XC32
bitfields	無視	無視	無視
data_alignment	無視	サポート	サポート
diag_default	無視	無視	無視
diag_error	無視	無視	無視
diag_remark	無視	無視	無視
diag_suppress	無視	無視	無視
diag_warning	無視	無視	無視
include_alias	無視	無視	無視
inline	サポート (引数無視)	サポート (引数無視)	サポート (引数無視)
language	無視	無視	無視
location	無視	サポート	サポート
message	サポート	ネイティブ サポート	ネイティブ サポート
object_attribute	無視	無視	無視
optimize	無視	ネイティブ サポート	ネイティブ サポート
pack	無視	ネイティブ サポート	ネイティブ サポート
__printf_args	サポート	サポート	サポート
required	無視	サポート	サポート
rtmodel	無視	無視	無視
__scanf__args	無視	サポート	サポート
section	無視	サポート	サポート
segment	無視	サポート	サポート
swi_number	無視	無視	無視
type_attribute	無視	無視	無視
weak	無視	ネイティブ サポート	ネイティブ サポート

© 2015 Microchip Technology Inc.

補遺 B. 処理系定義のふるまい

B.1 はじめに

補遺 B では、コンパイラにおける処理系定義のふるまいの選択について説明します。

B.2 ハイライト

主な内容は以下の通りです。

- 概要
- 翻訳
- 環境
- 識別子
- 文字
- 整数
- 浮動小数点数
- 配列とポインタ
- ヒント
- 構造体、共用体、列挙体、ビットフィールド
- 修飾子
- 宣言子
- 式文
- 前処理ディレクティブ
- ライブラリ関数
- アーキテクチャ

B.3 概要

ISO C に準拠する処理系は、規格内で「処理系定義」(implementation-defined) として定義されているふるまいをどのように選択したか明示する必要があります。補遺 B では、これに該当する全ての領域を挙げ、本コンパイラの処理系定義ふるまいと、ISO/IEC 9899:1999 規格内の対応する項目番号を示します。

© 2015 Microchip Technology Inc.

B.4 翻訳

ISO 規格 : 「診断の識別方法 (3.10、5.1.1.3)」

処理系定義 : stderr に対する全ての出力は診断です。

ISO 規格 : 「連続する空白類文字 (改行文字を除く) の各並びを 1 つの空白文字に置き換えるかどうか (5.1.1.2 翻訳フェイズの項目 3)」

処理系定義 : 連続する空白類文字の各並びを 1 文字に置き換えます。

B.5 環境

ISO 規格 : 「自立環境においてプログラム開始時に呼び出す関数の名前と型 (5.1.2.1)」

処理系定義 : int main (void);

ISO 規格 : 「自立環境におけるプログラム終了の影響 (5.1.2.1)」

処理系定義 : 無限ループ (自身への分岐) 命令を実行します。

ISO 規格 : 「main 関数を定義可能な代替の方法 (5.1.2.2.1)」

処理系定義 : int main (void);

ISO 規格 : 「main への引数 argv が指す文字列に与えられる値 (5.1.2.2.1)」

処理系定義 : main には引数を渡しません。argc または argv への参照は未定義です。

ISO 規格 : 「対話型装置を構成する物 (5.1.2.3)」

処理系定義 : アプリケーション定義

ISO 規格 : 「プログラム開始時に signal(sig, SIG_IGN) と同等の事が実行される信号 (7.14.1.1)」

処理系定義 : 信号はアプリケーション定義です。

ISO 規格 : 「生成された SIGABRT 信号が捕捉されない時に、プログラムの終了が失敗した事をホスト環境に知らせるために返されるステータスの形態 (7.20.4.1)」

処理系定義 : ホスト環境はアプリケーション定義です。

ISO 規格 : 「プログラム終了の成功 / 失敗を報告するために exit 関数がホスト環境に返すステータスの形態 (7.20.4.3)」

処理系定義 : ホスト環境はアプリケーション定義です。

ISO 規格 : 「引数が 0、EXIT_SUCCESS、EXIT_FAILURE のいずれでもない場合に exit 関数がホスト環境に返すステータス (7.20.4.3)」

処理系定義 : ホスト環境はアプリケーション定義です。

ISO 規格 : 「getenv 関数が使う環境名のセットと環境リストを変更するための方法 (7.20.4.4)」

処理系定義 : ホスト環境はアプリケーション定義です。

ISO 規格 : 「system 関数による文字列の実行方法 (7.20.4.5)」

処理系定義 : ホスト環境はアプリケーション定義です。

© 2015 Microchip Technology Inc.

B.6 識別子

ISO 規格 : 「識別子に使える追加のマルチバイト文字と、それらに対応する国際文字名 (6.4.2)」

処理系定義 : なし

ISO 規格 : 「識別子の有意先頭文字数 (5.2.4.1, 6.4.2)」

処理系定義 : 全ての文字が有意です。

B.7 文字

ISO 規格 : 「1 バイトのビット数 (C90 3.4、C99 3.6)」

処理系定義 : 8

ISO 規格 : 「実行キャラクタセットのメンバーの値 (C90/C99 5.2.1)」

ISO 規格 : 「標準アルファベット エスケープ シーケンスのそれぞれに対して生成される実行キャラクタセットのメンバーの一意値 (C90/C99 5.2.2)」

処理系定義 : 実行キャラクタセットは ASCII です。

ISO 規格 : 「基本実行キャラクタセットに含まれない文字を格納した `char` 型オブジェクトの値 (C90 6.1.2.5、C99 6.2.5)」

処理系定義 : `char` 型オブジェクトの値は、ソース キャラクタセット内のその文字の 8 ビットバイナリ表現です。つまり、翻訳しません。

ISO 規格 : 「`signed char` 型と `unsigned char` 型のどちらが単なる `char` 型と同じレンジ、表現、挙動を持つか (C90 6.1.2.5、C90 6.2.1.1、C99 6.2.5、C99 6.3.1.1)」

処理系定義 : 既定値により、単なる `char` 型と機能的に等価なのは `signed char` 型です。この既定値は、`-funsigned-char` と `-fsigned-char` を使って変更できます。

ISO 規格 : 「実行キャラクタセットのメンバーに対するソース キャラクタセットのメンバー (文字定数および文字列リテラル内) の対応付け (C90 6.1.3.4、C99 6.4.4.4、C90/C99 5.1.1.2)」

処理系定義 : ソース キャラクタセットのバイナリ表現は、実行キャラクタセットへと維持されます。

ISO 規格 : 「複数の文字または 1 バイト実行キャラクタに対応しない文字 / エスケープ シーケンスを格納する整数型文字定数の値 (C90 6.1.3.4、C99 6.4.4.4)」

処理系定義 : コンパイラは、多文字定数の値を 1 文字ずつ特定します。前の値を 8 ビット左にシフトしてから次の文字のビットパターンを抽出します。最終的な結果は `int` 型です。結果が `int` 型で表現できなくなると警告を出力し、値を `int` 型に切り詰めます。

ISO 規格 : 「複数のマルチバイト文字または拡張実行キャラクタセットに含まれない 1 つのマルチバイト文字 / エスケープ シーケンスを格納するワイド文字定数の値 (C90 6.1.3.4、C99 6.4.4.4)」

処理系定義 : 前の項目を参照してください。

ISO 規格 : 「拡張実行キャラクタセットのメンバーに対応付けられた 1 つのマルチバイト文字から成るワイド文字定数をワイド文字コードに変換するために使うその時点のロケール (C90 6.1.3.4、C99 6.4.4.4)」

処理系定義 : LC_ALL

ISO 規格 : 「ワイド文字列リテラルをワイド文字コードに変換するために使うその時点のロケール (C90 6.1.4、C99 6.4.5)」

処理系定義 : LC_ALL

ISO 規格 : 「実行キャラクタセットで表現されないマルチバイト文字またはエスケープ シーケンスを格納した文字列リテラルの値 (C90 6.1.4、C99 6.4.5)」

処理系定義 : 文字のバイナリ表現はソース キャラクタセットから維持します。

B.8 整数

ISO 規格 : 「処理系内に存在する全ての拡張整数型 (C99 6.2.5)」

処理系定義 : 拡張整数型は存在しません。

ISO 規格 : 「符号付き整数型の表現に符号と絶対値、2 の補数、1 の補数のどれを使うか。また、異常値はトラップ表現か通常値か (C99 6.2.6.2)」

処理系定義 : 全ての整数型を 2 の補数として表現し、全てのビットパターンは通常値です。

ISO 規格 : 「同精度の拡張整数型同士間の相対的なランク (C99 6.3.1.1)」

処理系定義 : 拡張整数型はサポートしていません。

ISO 規格 : 「整数を符号付き整数型に変換してオブジェクトの値が表現できなかった場合の結果または生成される信号 (C90 6.2.1.2、C99 6.3.1.3)」

処理系定義 : 値 X を幅 N の型に変換すると、X の 2 の補数の最下位 N ビットが結果として得られます。つまり、X は N ビットに切り詰められます。信号は生成しません。

ISO 規格 : 「符号付き整数に対する一部のビット単位演算の結果 (C90 6.3、C99 6.5)」

処理系定義 : 符号付き値に対するビット単位演算は 2 の補数表現 (符号ビットを含む) に基づきます。符号付き右シフト式の結果は符号拡張されます。
C99 は、符号付きの「<<」が未定義の側面を持つとしています。本コンパイラでは、未定義ではありません。

© 2015 Microchip Technology Inc.

B.9 浮動小数点数

ISO 規格 : 「浮動小数点型演算の精度と、浮動小数点型結果を返す <math.h> および <complex.h> 内のライブラリ関数の精度 (C90/C99 5.2.4.2.2)」

処理系定義 : 精度は未知です。

ISO 規格 : 「浮動小数点型内部表現と <stdio.h>、<stdlib.h>、<wchar.h> 内のライブラリ関数によって実行される文字列表現の間の変換の精度 (C90/C99 5.2.4.2.2)」

処理系定義 : 精度は未知です。

ISO 規格 : 「FLT_ROUNDS の規格外値によって指定される丸め処理のふるまい (C90/C99 5.2.4.2.2)」

処理系定義 : そのような値は使いません。

ISO 規格 : 「FLT_EVAL_METHOD の規格外の負値によって指定される評価手法 (C90/C99 5.2.4.2.2)」

処理系定義 : そのような値は使いません。

ISO 規格 : 「整数を浮動小数点型に変換して元の値を厳密に表現できない場合の丸め処理の方向 (C90 6.2.1.3、C99 6.3.1.4)」

処理系定義 : C99 の付属文書 F に従います。

ISO 規格 : 「浮動小数点値をより低精度の浮動小数点値に変換する際の丸め処理の方向 (C90 6.2.1.4、6.3.1.5)」

処理系定義 : C99 の付属文書 F に従います。

ISO 規格 : 「浮動小数点型定数に対して、最も近い表現可能な値を選択するか、最も近い表現可能な値のすぐ隣のより大きなまたはより小さな表現可能値を選択するか (C90 6.1.3.1、C99 6.4.4.2)」

処理系定義 : C99 の付属文書 F に従います。

ISO 規格 : 「FP_CONTRAC プラグマによって禁止されない場合に、浮動小数点型の式を短縮するかどうかと、どのように短縮するか (C99 6.5)」

処理系定義 : このプラグマは実装していません。

ISO 規格 : 「FENV_ACCESS プラグマの既定値ステート (C99 7.6.1)」

処理系定義 : このプラグマは実装していません。

ISO 規格 : 「追加の浮動小数点例外、丸め処理モード、環境、分類と、それらのマクロ名 (C99 7.6、7.12)」

処理系定義 : どれもサポートしません。

ISO 規格 : 「FP_CONTRACT プラグマの既定値ステート (C99 7.12.2)」

処理系定義 : このプラグマは実装していません。

ISO 規格 : 「丸め処理結果がIEC 60559準拠の処理系における算術結果と実質的に等しい場合に「不正確」浮動小数点例外が発生するかどうか (C99 F.9)」

処理系定義 : 未知

ISO 規格 : 「結果がわずかに不正確であるものの、IEC 60559 準拠の処理系では不正確とはならない場合、「アンダーフロー」(および「不正確結果」) 浮動小数点例外が発生するかどうか (C99 F.9)」

処理系定義 : 未知

B.10 配列とポインタ

ISO 規格 : 「ポインタから整数への (またはその逆の) 変換の結果 (C90 6.3.4、C99 6.3.2.3)」

処理系定義 : 整数からポインタへの (またはその逆) のキャストの結果は、デスティネーションの型に適するように再解釈したソースの型のバイナリ表現を使います。
ソースの型がデスティネーションの型よりも長い場合、最上位ビットは破棄されます。ポインタから整数へのキャストにおいてソースの型がデスティネーションの型よりも短い場合、結果は符号拡張されます。整数からポインタへのキャストにおいてソースの型がデスティネーションの型よりも短い場合、結果はソースの型の符号指定に基づいて拡張されます。

ISO 規格 : 「同じ配列のエレメントを指す 2 つのポインタの減算結果のサイズ (C90 6.3.6、C99 6.5.6)」

処理系定義 : 32 ビット符号付き整数

B.11 ヒント

ISO 規格 : 「`register` 記憶域クラス指定子の有効性 (C90 6.5.1、C99 6.7.1)」

処理系定義 : `register` 記憶域クラス指定子は一般的に効力を持ちません。

ISO 規格 : 「`inline` 関数指定子の有効性 (C99 6.7.4)」

処理系定義 : `-fno-inline` または `-O0` が指定されている場合、`inline` 指定子が指定されても関数は一切インライン展開されません。その他の場合、関数がインライン展開可能かどうかは、コンパイラの最適化ヒューリスティックスによって決まります。

B.12 構造体、共用体、列挙体、ビットフィールド

ISO 規格 : 「共用体オブジェクトのメンバーは異なる型のメンバーを使ってアクセスされる (C90 6.3.2.3)」

処理系定義 : アラインメント等の条件が無効であろうとなかろうと、共用体オブジェクトの対応するバイトは、アクセス先のメンバーと同じ型のオブジェクトとして解釈されます。

ISO 規格 : 「単なる `int` 型のビットフィールドが `signed int` 型または `unsigned int` 型のどちらとして扱われるか (C90 6.5.2、C90 6.5.2.1、C99 6.7.2、C99 6.7.2.1)」

処理系定義 : 既定値では、単なる `int` 型のビットフィールドは `signed integer` 型として扱います。このふるまいは、`-funsigned-bitfields` コマンドライン オプションを使って変更できます。

ISO 規格 : 「`_Bool`、`signed int`、`unsigned int` の他に許容されるビットフィールドの型 (C99 6.7.2.1)」

処理系定義 : 他の型はサポートしていません。

ISO 規格 : 「ビットフィールドが記憶域ユニットの境界をまたぐ事でできるかどうか (C90 6.5.2.1、C99 6.7.2.1)」

処理系定義 : できません。

ISO 規格 : 「ユニット内のビットフィールドの割り当て順 (C90 6.5.2.1、C99 6.7.2.1)」

処理系定義 : ビットフィールドは左から右へ割り当てます。

ISO 規格 : 「構造体の非ビットフィールド メンバーの割り当て順 (C90 6.5.2.1、C99 6.7.2.1)」

処理系定義 : 各メンバーは、そのメンバー型のアラインメントの制約に従って、可能な限り小さなオフセットで配置します。

© 2015 Microchip Technology Inc.

ISO 規格 : 「各列挙型と互換性のある整数型 (C90 6.5.2.2、C99 6.7.2.2)」

処理系定義 : 列挙値が全て非負の場合は `unsigned int` 型、その他の場合は `int` 型です。これは `-fshort-enums` コマンドライン オプションを使って変更できます。

B.13 修飾子

ISO 規格 : 「`volatile` 修飾された型を持つオブジェクトへのアクセスは何によって構成するか (C90 6.5.3、C99 6.7.3)」

処理系定義 : `volatile` オブジェクトの値を使う式または `volatile` オブジェクトに値を保存する式は、そのオブジェクトへのアクセスであると見なします。そのようなアクセスがアトミックである事は保証しません。

`volatile` オブジェクトへの参照を含むものの、その値を使う事もそこに値を保存する事もしない式は、オブジェクトの型に応じて `volatile` 型へのアクセスであると見なすかどうかが決まります。オブジェクトがスカラー型、スカラー型のメンバーを 1 つ持つ集合体型、スカラー型のメンバー (だけ) を持つ共用体のいずれかである場合、その式は `volatile` オブジェクトへのアクセスであると見なします。その他の場合、その式の副作用だけを評価し、その式は `volatile` オブジェクトへのアクセスであるとは見なしません。

以下に例を示します。

```
volatile int a;
a; /* access to ‘a’ since ‘a’ is scalar */
```

B.14 宣言子

ISO 規格 : 「算術型、構造体型、共用体型を変更可能な宣言子の最大数 (C90 6.5.4)」

処理系定義 : 制限はありません。

B.15 式文

ISO 規格 : 「`switch` 文の中の `case` 値の最大数 (C90 6.6.4.2)」

処理系定義 : 制限はありません。

B.16　前処理ディレクティブ

ISO 規格 : 「ヘッダ名の両方の書式において、文字の並びをヘッダまたは外部ソースファイルの名前に対応付ける方法 (C90 6.1.7、C99 6.4.7)」

処理系定義 : 区切り文字の間の文字の並びは、ホスト環境向けのファイル名を表す文字列であると見なします。

ISO 規格 : 「条件付きインクルードを制御する定数式の中の文字定数の値が実行キャラクタセット内の同じ文字定数に一致するかどうか (C90 6.8.1, C99 6.10.1)」

処理系定義 : 一致します。

ISO 規格 : 「条件付きインクルードを制御する定数式の中の `character` 定数 (1 文字) は負値を持つ事ができるかどうか (C90 6.8.1、C99 6.10.1)」

処理系定義 : 負値を持つ事ができます。

ISO 規格 : 「`# include` 文の中の < > で囲まれたヘッダを検索する場所と、その場所の指定方法またはヘッダの識別方法 (C90 6.8.2、C99 6.10.2)」

処理系定義 : `<install directory>/lib/gcc/pic32mx/3.4.4/include`

`<install directory>/pic32mx/include`

ISO 規格 : 「`# include` 文の中の "" で囲まれたソースファイルを検索する方法 (C90 6.8.2、C99 6.10.2)」

処理系定義 : コンパイラは、最初に `-iquote` コマンドライン オプションによってディレクトリが指定されていればそこを検索し、次に < > で囲まれたヘッダ向けのディレクトリを検索します。

ISO 規格 : 「前処理トークンをヘッダ名に組み込む方法 (C90 6.8.2、C99 6.10.2)」

処理系定義 : 全てのトークン (空白類文字を含む) はヘッダファイル名の一部であると見なします。区切り文字の囲みの中のトークンに対してマクロ展開は行いません

ISO 規格 : 「`#include` 処理のネストの制限 (C90 6.8.2、C99 6.10.2)」

処理系定義 : 制限はありません。

ISO 規格 : 「認識された各非 `STDC #pragma` ディレクティブに対するふるまい (C90 6.8.6、C99 6.10.6)」

処理系定義 : **8.12「変数属性」**を参照してください。

ISO 規格 : 「翻訳の日付が得られない場合の `__DATE_ _` の定義と、翻訳の時刻が得られない場合の `__TIME_ _` の定義 (C90 6.8.8、C99 6.10.8)」

処理系定義 : 翻訳の日付と時刻は常に得られます。

B.17　ライブラリ関数

ISO 規格 : 「`NULL` マクロが展開する先の NULL ポインタ定数 (C90 7.1.6、C99 7.17)」

処理系定義 : `(void *)0`

ISO 規格 : 「Clause 4. が要求する最小限のライブラリ機能の他に自立環境プログラムで利用できるライブラリ機能 (5.1.2.1)」

処理系定義 : 『32-Bit Language Tools Libraries』(DS51685) を参照してください。

ISO 規格 : 「`assert` マクロが出力する診断情報の書式 (7.2.1.1)」

処理系定義 : 「Failed assertion '*message*' at line *line* of '*filename*'.\n」

ISO 規格 : `FENV_ACCESS` プラグマの既定値状態 (7.6.1)」

処理系定義 : 未実装

ISO 規格 : 「`fegetexceptflag` 関数が保存する浮動小数点例外フラグの表現 (7.6.2.2)」

© 2015 Microchip Technology Inc.

処理系定義 : 未実装

ISO 規格 : 「feraiseexcept 関数がオーバーフロー例外またはアンダーフロー例外の他に不正確結果例外を生成するかどうか (7.6.2.3)」

処理系定義 : 未実装

ISO 規格 : 「fesetenv または feupdateenv 関数への引数として使える FE_DFL_ENV 以外の浮動小数点環境マクロ (7.6.4.3、7.6.4.4)」

処理系定義 : 未実装

ISO 規格 : 「"C" と "" の他に setlocale 関数への引数として使える文字列 (7.11.1.1)」

処理系定義 : なし

ISO 規格 : 「FLT_EVAL_METHOD マクロの値が 0 未満か 2 よりも大きい場合に float_t と double_t に定義される型 (7.12)」

処理系定義 : 未実装

ISO 規格 : 「可能な場合に INFINITY マクロが展開する無限大 (7.12)」

処理系定義 : 未実装

ISO 規格 : 「定義されている場合に NAN マクロが展開する quiet NaN (7.12)」

処理系定義 : 未実装

ISO 規格 : 「この国際規格が要求する数学関数以外の数学関数の定義域エラー (7.12.1)」

処理系定義 : なし

ISO 規格 : 「定義域エラーが発生した場合に数学関数が返す値と、errno に EDOM マクロの値を設定するかどうか (7.12.1)」

処理系定義 : 定義域エラーが発生すると、errno に EDOM を設定します。

ISO 規格 : 「オーバーフローおよび / またはアンダーフロー レンジエラーが発生した時に、数学関数が errno を ERANGE マクロの値に設定するかどうか (7.12.1)」

処理系定義 : 設定します。

ISO 規格 : 「FP_CONTRACT プラグマの既定値ステート (7.12.2)」

処理系定義 : 未実装

ISO 規格 : 「fmod 関数の第 2 引数が 0 である場合に定義域エラーが発生するか、それとも 0 が返されるか (7.12.10.1)」

処理系定義 : NaN を返します。

ISO 規格 : 「remquo 関数が商を削減する際に使う法の 2 を底とする対数 (7.12.10.3)」

処理系定義 : 未実装

ISO 規格 : 「信号のセットと、それらの意味および既定値処理 (7.14)」

処理系定義 : 信号の既定値処理は常に「偽」を返します。実際の信号処理はアプリケーション定義です。

ISO 規格 : 「信号ハンドラの呼び出し前に signal(sig, SIG_DFL) と同等の事が実行されない場合に実行される信号のブロッキング (7.14.1.1)」

処理系定義 : アプリケーション定義

ISO 規格 : 「信号 SIGILL 向けの信号ハンドラの呼び出し前に signal(sig, SIG_DFL) と同等の事が実行されるかどうか (7.14.1.1)」

処理系定義 : アプリケーション定義

ISO 規格 : 「計算例外に対応する SIGFPE、SIGILL、SIGSEGV 以外の信号値 (7.14.1.1)」

処理系定義 : アプリケーション定義

ISO 規格 : 「テキストストリームの最終行に終端改行文字が必要かどうか (7.19.2)」

処理系定義 : 必要です。

ISO 規格 : 「改行文字の直前でテキスト ストリームに書き出された空白文字は読み出し時に現れるかどうか (7.19.2)」

処理系定義 : 現れます。

ISO 規格 : 「バイナリ ストリームに書き込まれるデータに付加可能なNULL文字の数 (7.19.2)」

処理系定義 : バイナリ ストリームに NULL 文字は付加しません。

ISO 規格 : 「アペンドモード ストリームのファイル位置インジケータがファイルの先頭または末尾位置に初期設定されるかどうか (7.19.3)」

処理系定義 : アプリケーション定義。システムレベル関数 `open` は `O_APPEND` フラグ付きで呼び出されます。

ISO 規格 : 「テキスト ストリームへの書き込み時に、関連するファイルがその位置を越えたところで切り詰められるかどうか (7.19.3)」

処理系定義 : アプリケーション定義

ISO 規格 : 「ファイル バッファリングの特性 (7.19.3)」

ISO 規格 : 「長さが 0 のファイルが実際に存在するかどうか (7.19.3)」

処理系定義 : アプリケーション定義

ISO 規格 : 「ファイルの命名規則 (7.19.3)」

処理系定義 : アプリケーション定義

ISO 規格 : 「同じファイルを複数回開く事ができるかどうか (7.19.3)」

処理系定義 : アプリケーション定義

ISO 規格 : 「ファイル内のマルチバイト文字に使うエンコーディングの種類と選択 (7.19.3)」

処理系定義 : 各ファイルでエンコーディングは同じです。

ISO 規格 : 「ファイルを開く際の `remove` 関数の影響 (7.19.4.1)」

処理系定義 : アプリケーション定義。システム関数 `unlink` を呼び出します。

ISO 規格 : 「`rename` 関数の呼び出し前に新しい名前のファイルが存在する場合の影響 (7.19.4.2)」

処理系定義 : アプリケーション定義。システム関数 `link` を呼び出して新しいファイル名を生成した後に、`unlink` を呼び出して古いファイル名を削除します。通常、新しいファイル名が既に存在していると、`link` は失敗します。

ISO 規格 : 「プログラムが異常終了した時に、開いていた一時ファイルが削除されるかどうか (7.19.4.3)」

処理系定義 : 削除されません。

ISO 規格 : 「`tmpnam`関数の呼び出し回数が`TMP_MAX`を超えた場合の動作(7.19.4.4)」

処理系定義 : 一時名はロールオーバーして再使用されます。

ISO 規格 : 「どのモード変更がどのような状況で許容されるか (7.19.5.4)」

処理系定義 : システムレベルの `close` 関数によってファイルを閉じた後に、`open` 関数によって新しいモードで再度開きます。アプリケーション定義の `open` および `close` 関数の制約の他に制約はありません。

ISO 規格 : 「無限大または NaN を出力するために使うスタイルと、そのスタイルを NaN のために出力する場合の n 文字シーケンスの意味 (7.19.6.1、7.24.2.1)」

処理系定義 : 文字シーケンスは出力しません。
NaN は「NaN」として出力します。
無限大は「[-/+]Inf」として出力します。

ISO 規格 : 「`fprintf` または `fwprintf` 関数での `%p` 変換のための出力 (7.19.6.1, 7.24.2.1)」

処理系定義 : 機能的に `%x` と等価です。

© 2015 Microchip Technology Inc.

ISO 規格 : 「fscanf または fwscanf 関数での %[変換のスキャンリストの中で、「-」文字が先頭文字でも末尾文字でもなく、先頭の「^」文字に続く第 2 文字でもない場合の「-」文字の解釈 (7.19.6.2、7.24.2.1)」

処理系定義 : 未知

ISO 規格 : 「fscanf または fwscanf 関数での %p 変換によって照合する文字の並びのセット (7.19.6.2、7.24.2.2)」

処理系定義 : %x によって照合する文字の並びのセットと同じです。

ISO 規格 : 「fscanf または fwscanf 関数での %p 変換に対応する入力アイテムの解釈 (7.19.6.2、7.24.2.2)」

処理系定義 : 結果が有効ポインタではない場合のふるまいは未定義です。

ISO 規格 : 「障害時に fgetpos、fsetpos、ftell 関数が errno マクロに設定する値 (7.19.9.1、7.19.9.3、7.19.9.4)」

処理系定義 : 結果が LONG_MAX を超えると、errno は ERANGE に設定されます。その他のエラーはアプリケーション定義です (lseek 関数のアプリケーション定義に従います)。

ISO 規格 : 「strtod、strtof、strtold、wcstod、wcstof、wcstold 関数によって変換された文字列内の n 文字シーケンスの意味 (7.20.1.3, 7.24.4.1.1)」

処理系定義 : n 文字シーケンスに意味は与えません。

ISO 規格 : 「アンダーフロー発生時に strtod、strtof、strtold、wcstod、wcstof、wcstold関数がerrnoをERANGEに設定するかどうか(7.20.1.3、7.24.4.1.1)」

処理系定義 : 設定します。

ISO 規格 : 「要求されたサイズが 0 である場合に calloc、malloc、realloc 関数は NULL ポインタを返すか、それとも割り当てられたオブジェクトへのポインタを返すか (7.20.3)」

処理系定義 : 静的に割り当てられたオブジェクトへのポインタを返します。

ISO 規格 : 「abort 関数が呼び出された時に、開いていた出力ストリームをフラッシュするのか閉じるのか、一時ファイルを削除するかどうか (7.20.4.1)」

処理系定義 : どれも行いません。

ISO 規格 : 「abort 関数がホスト環境に返す終了ステータス (7.20.4.1)」

処理系定義 : 既定値により、ホスト環境は存在しません。

ISO 規格 : 「引数が NULL ポインタではない場合に system 関数が返す値 (7.20.4.5)」

処理系定義 : アプリケーション定義

ISO 規格 : 「ローカル タイムゾーンと夏時間 (7.23.1)」

処理系定義 : アプリケーション定義

ISO 規格 : 「clock 関数の計時期間 (7.23.2.1)」

処理系定義 : アプリケーション定義

ISO 規格 : 「正規化された tmx 構造体内の tm_isdst の正の値 (7.23.2.6)」

処理系定義 : 1

ISO 規格 : 「C ロケールでの strftime、strfxtime、wcsftime、wcsfxtime 関数に対する%Z指定子の置換文字列(7.23.3.5、7.23.3.6、7.24.5.1、7.24.5.2)」

処理系定義 : 未実装

ISO 規格 : 「IEC 60559 準拠の処理系において三角関数、双曲線関数、*e* を底とする指数関数、*e* を底とする対数関数、エラー関数、対数ガンマ関数によって不正確結果例外が発生するかどうか。発生するならば、いつ発生するか (F.9)」

処理系定義 : 不正確結果例外は発生しません。

ISO 規格 : 「丸め処理結果がIEC 60559準拠の処理系における算術結果と事実上等しい場合に不正確結果例外が発生するかどうか (F.9)」

処理系定義 : 不正結果例外は発生しません。

ISO 規格 : 「結果がわずかに不正確であるものの、IEC 60559 準拠の処理系では不正確とはならない場合、「アンダーフロー」(および「不正確結果」) 例外が発生するかどうか (C99 F.9)」

処理系定義 : 不正結果例外は発生しません。

ISO 規格 : 「関数が丸め処理方向モードを尊重するかどうか (F.9)」

処理系定義 : 丸め処理モードは強制されません。

B.18 アーキテクチャ

ISO 規格 : 「ヘッダ `<float.h>`、`<limits.h>`、`<stdint.h>` 内で指定されたマクロに割り当てられる値または式 (C90/C99 5.2.4.2、C99 7.18.2、7.18.3)」

処理系定義 : **8.4.2「`limits.h`」**を参照してください。

ISO 規格 : 「オブジェクト内のバイトの数、順序、エンコーディング (規格が明示的に指定していない場合) (C99 6.2.6.1)」

処理系定義 : リトルエンディアン (最下位バイトから先に格納)**8.3「データの表現」**を参照してください。

ISO 規格 : 「演算子のサイズの結果の値 (C90 6.3.3.4、C99 6.5.3.4)」

処理系定義 : **8.3「データの表現」**を参照してください。

 © 2015 Microchip Technology Inc.

補遺 C. 推奨しない機能

C.1 はじめに

補遺 C では、より進んだ機能によって置き換えられた非推奨の機能について説明します。非推奨機能を使っているプロジェクトは、本書に記載した言語ツールのバージョンで正しく機能します。非推奨機能を使うと警告が出力されます。プロジェクトを修正して、これらの機能を使わなくする事を推奨します。言語ツールの将来のバージョンでは、これらの機能のサポートを完全に廃止する可能性があります。

非推奨機能には以下があります。

- 指定レジスタへの変数の格納

C.2 指定レジスタへの変数の格納

本コンパイラでは、少数のグローバル変数を指定したハードウェア レジスタに格納する事ができます。

> **Note:** 使用するレジスタの数が多すぎると、32 ビットコンパイラのコンパイル能力が低下する可能性があります。グローバル変数を特定のレジスタに固定的に配置する事は推奨しません。

通常のレジスタ変数を指定したレジスタに割り当てる事もできます。

- グローバル レジスタ変数は、プログラムの全体を通してレジスタを予約します。これは、2 ～ 3 個のグローバル変数に頻繁にアクセスするプログラム (プログラミング言語インタープリタ等) にとって便利です。
- 指定レジスタにローカルレジスタ変数を割り当てた場合、そのレジスタは予約されません。コンパイラのデータフロー解析は、指定されたレジスタがどの時点で必要な値を格納し、どの時点で他の目的に使えるようになるのか判断する事ができます。ローカルレジスタ変数への保存は、使われないと判断された時点で削除されます。ローカルレジスタ変数への参照は、削除または移動または単純化される場合があります。

アセンブリ命令の 1 つの出力を特定のレジスタに直接書き込む事が望ましい場合、拡張インライン アセンブリでこれらのローカル変数を使うと便利な場合があります (**第 16 章「C/C++ とアセンブリ言語の併用」**参照)。これは、指定したレジスタが、インライン アセンブリ式文内のそのオペランドに指定されている制約に適合する場合に正しく機能します。

C.2.1 グローバル レジスタ変数の定義

グローバル レジスタ変数の定義例を以下に示します。

```
register int *foo asm ("t0");
```

`t0` は、使用するレジスタの名前です。ライブラリ ルーチンがこのレジスタを上書きしないよう、関数呼び出しによって正常に退避 / 復元されるレジスタを選択します。

特定のレジスタでグローバル レジスタ変数を定義すると、そのレジスタはその用途向けに完全に (少なくとも現在のコンパイル中は) 予約されます。現在のコンパイルにおいて、そのレジスタは関数内の他の目的に一切割り当てられません。このレジスタは、それらの関数によって退避 / 復元されません。このレジスタに格納した値は、明らかに使われなくなったとしても決して削除されません。しかし、参照は削除または移動または単純化される場合があります。

© 2015 Microchip Technology Inc.

信号ハンドラまたは制御の複数のスレッドからのグローバル レジスタ変数へのアクセスは安全ではありません。なぜなら、システム ライブラリ ルーチンがそのレジスタを他の目的に一時的に使う可能性があるからです (ユーザが、それらを目前のタスクのために再コンパイルしない場合)。

ある関数からグローバル レジスタ変数を使って別の関数 (例えば foo) を、この変数の事を関知せずにコンパイルした (つまりソースファイル内でその変数を宣言しなかった) 第 3 の関数 (例えば lose) を経由して呼び出す事は安全ではありません。なぜなら、lose はレジスタを退避させて他の値をそこに書き込む可能性があるからです。例えば、qsort に渡す比較関数内でグローバル レジスタ変数が利用できると期待する事はできません。なぜなら、qsort は他の何かをそのレジスタに格納したかもしれないからです。この問題は、同じグローバル レジスタ変数定義を使って qsort を再コンパイルする事により防げます。

グローバル レジスタ変数を実際には使わない qsort または他のソースファイルを再コンパイルして、それらがそのグローバル レジスタ変数を他の目的に使わないようにするには、コンパイラ コマンドライン オプション -ffixed-*reg* を指定するだけで十分です。それらのソースファイルにグローバル レジスタ変数の宣言を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可能性のある関数は、この変数を関知せずにコンパイルした関数から安全に呼び出す事はできません。なぜなら、呼び出し元が期待する戻り値が上書きされる可能性があるからです。従って、グローバル レジスタ変数を使うプログラム部分へのエントリポイントとなる関数は、呼び出し元に属する値を明示的に退避 / 復元する必要があります。

ライブラリ関数 longjmp は、各グローバル レジスタ変数を setjmp の時点の値に復元します。

全てのグローバル レジスタ変数は、全ての関数定義よりも前で宣言する必要があります。関数定義より後で宣言した場合、そのレジスタは先行する関数内で他の目的に使われる可能性があります。

グローバル レジスタ変数に初期値を持たせる事はできません。なぜなら、実行可能ファイルがレジスタに初期値を提供する手段が存在しないからです。

C.2.2 ローカル変数向けにレジスタを指定する

以下の例のように、レジスタを指定してローカル レジスタ変数を定義できます。

```
register int *foo asm ("t0");
```

t0 は、使用するレジスタの名前です。これは、グローバル レジスタ変数の定義に使った構文と同じです。しかし、ローカル変数は関数内で定義するという事に注意が必要です。

ローカル レジスタ変数を定義した場合、そのレジスタは予約されません。つまり、フロー制御がその変数の値は使われないと判断した箇所では、そのレジスタを他の目的に使う事ができます。この機能を使うと、コンパイラが利用できるレジスタが非常に少ないままで特定の関数をコンパイルできます。

この場合、指定したレジスタが常にこの変数を格納しているとは限りません。つまり、asm 文の中でこのレジスタを明示的に参照していても、この変数が常に参照されるとは限らないという事です。

ローカルレジスタ変数への代入は、それらが使われないと判断された時点で削除される可能性があります。ローカルレジスタ変数への参照は、削除または移動または単純化される場合があります。

 © 2015 Microchip Technology Inc.

補遺 D. ビルトイン関数

D.1 はじめに

補遺Dでは、MPLAB XC32 Cコンパイラに固有のビルトイン関数について説明します。

ビルトイン関数は、C プログラマーに、現時点ではインライン関数を使ってアクセスする事しかできないアセンブリ オペレーションまたはマシン命令へのアクセスを提供します。これらの関数は幅広いアプリケーションに十分に役立てる事ができます。ビルトイン関数は、C ソースファイル内で関数に似た構文を使いますが、関数読み出またはライブラリ ルーチンを伴わずに関数を直接実装するアセンブリコードへとコンパイルされます。

ビルトイン関数は、インライン アセンブリを使う必要があるユーザに以下の利点を提供します。

1. 特定の目的向けにビルトイン関数を提供する事でコーディングが簡潔になります。
2. インライン アセンブリを使うと特定の最適化が無効になりますが、ビルトイン関数を使うとそうなりません。
3. 専用のレジスタを使うマシン命令の場合、レジスタ割り当てエラーが生じないようにインライン アセンブリコードを書くには、かなりの注意が必要です。ビルトイン関数を使った場合、各マシン命令に固有のレジスタ要件に気を配る必要がなくなるため、この手順を簡潔にできます。

ビルトイン関数の一覧

- unsigned long__builtin_section_begin(quoted-section-name)
- unsigned long __builtin_section_end(quoted-section-name)
- unsigned long __builtin_section_size(quoted-section-name)
- unsigned int __builtin_get_isr_state(void)
- __builtin_set_isr_state(unsigned int)
- void __builtin_disable_interrupts(void)
- void __builtin_enable_interrupts(void)

D.2 ビルトイン関数の説明

以下では、コンパイラのビルトイン関数へのプログラマ インターフェイスについて説明します。ビルトイン関数は「組み込み済み」であるため、対応するヘッダファイルはありません。同様に、コマンドライン スイッチもありません。ビルトイン関数は常時利用可能です。ビルトイン関数は、コンパイラの名前空間内で命名されます (全てのビルトイン関数の名前には接頭辞 `__builtin_` が付きます)。従って、ビルトイン関数はユーザの名前空間内の関数または変数名と競合しません。

以下のビルトイン関数は、セクション アドレスとサイズに関する実行時情報を取得します。

unsigned long __builtin_section_begin(quoted-section-name)

概要 :	`quoted-section-name` の開始アドレスを返します。
プロトタイプ :	`unsigned long __builtin_section_begin(quoted-section-name);`
引数 :	`quoted-section-name`: セクション名
戻り値 :	セクションのアドレス
アセンブリ オペレータ / マシン命令 :	`.startof.`
エラーメッセージ	`quoted-section-name` がリンク内に存在しない場合、「undefined reference」エラーメッセージが出力されます。

unsigned long __builtin_section_end(quoted-section-name)

概要 :	`quoted-section-name` の終了アドレス + 1 を返します。
プロトタイプ :	`unsigned long __builtin_section_end(quoted-section-name);`
引数 :	`quoted-section-name`: セクション名
戻り値 :	セクションの終了アドレス + 1
アセンブリ オペレータ / マシン命令 :	`.endof.`
エラーメッセージ	`quoted-section-name` がリンク内に存在しない場合、「undefined reference」エラーメッセージが出力されます。

unsigned long __builtin_section_size(quoted-section-name)

概要 :	`quoted-section-name` のサイズ (バイト数) を返します。
プロトタイプ :	`unsigned long __builtin_section_size(quoted-section-name);`
引数 :	`quoted-section-name`: セクション名
戻り値 :	`quoted-section-name` のサイズ (バイト数)
アセンブリ オペレータ / マシン命令 :	`.sizeof.`
エラーメッセージ	`quoted-section-name` がリンク内に存在しない場合、「undefined reference」エラーメッセージが出力されます。

以下のビルトイン関数は、CPU 割り込みの現在の状態を調査または操作します。

unsigned int __builtin_get_isr_state(void)

概要 :	現在の割り込み優先度と割り込みイネーブルビットを取得します。
プロトタイプ :	`unsigned int __builtin_get_isr_state(void);`
引数 :	なし
戻り値 :	その時点の IPL と割り込みイネーブルビットを 1 つにパックした値 この値は `__builtin_set_isr_state()` 関数で使います。

© 2015 Microchip Technology Inc.

unsigned int __builtin_get_isr_state(void) (続き)

アセンブリ オペレータ / マシン命令 :	`mfc0 $3, $12, 0`
	`srl $2,$3,10`
	`ins $2,$3,3,1`
	`andi $2,$2,0xf`
	`sw $2,0($fp)`

エラーメッセージ	なし

__builtin_set_isr_state(unsigned int)

概要 :	`__builtin_get_isr_state()` から取得した値を使って割り込み優先度と割り込みイネーブルビットを設定します。
プロトタイプ :	`void __builtin_set_isr_state(unsigned int);`
引数 :	`__builtin_get_isr_state()` から取得した 1 つの符号なし整数値
戻り値 :	なし

アセンブリ オペレータ / マシン命令 :	`di`
	`ehb`
	`mfc0 $2, $12, 0`
	`ins $2,$3,10,3`
	`srl $3,$3,3`
	`ins $2,$3,0,1`
	`mtc0 $2, $12, 0`
	`ehb`

エラーメッセージ	なし

void __builtin_disable_interrupts(void)

概要 :	割り込みを無効にします。
プロトタイプ :	`void __builtin_disable_interrupts(void);`
引数 :	なし
戻り値 :	なし
アセンブリ オペレータ / マシン命令 :	`di $2` `ehb`
エラーメッセージ	なし

void __builtin_enable_interrupts(void)

概要 :	割り込みを有効にします。
プロトタイプ :	`void __builtin_enable_interrupts(void);`
引数 :	なし
戻り値 :	なし
アセンブリ オペレータ / マシン命令 :	`ei $2`
エラーメッセージ	なし

D.3 ビルトイン DSP 関数

多くの PIC32 MCU は DSP エンジン (DSP およびメディア アプリケーションの性能を向上させるための命令を含む) をサポートします。DSPr2 エンジンは、パックされた 8/16 ビット整数データと Q7/Q15/Q31 小数データに対して動作する命令を提供します。

XC32 C コンパイラは、一般的ベクタ拡張と一群のビルトイン関数を使って、これらの DSP 動作をサポートします。どちらのサポートも、`-mprocessor` オプションで DSP デバイスを選択すると自動的に有効になります。

DSP 制御レジスタの SCOUNT および POS ビットはグローバルです。WRDSP、EXTPDP、EXTPDPV、MTHLIP 命令は、SCOUNT および POS ビットを変更します。最適化中に、コンパイラはこれらの命令と、これらの命令を含む関数への呼び出しを削除しません。

現行の XC32 C コンパイラは、32 ビットベクタでの動作だけをサポートします。通常、8 ビット整数データに対応するベクタ型は v4i8 と呼びます。同様に、Q7 に対しては v4q7、16 ビット整数データに対しては v2i16、Q15 に対しては v2q15 と呼びます。これらは C コード内で以下のように定義できます。

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef signed char v4q7 __attribute__ ((vector_size(4)));
typedef short v2i16 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
```

v4i8、v4q7、v2i16、v2q15 値は、集合体と同様の方法で初期化します。以下に例を示します。

```
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};

v2q15 c = {0x0fcb, 0x3a75};
v2q15 d;
d = (v2q15) {0.1234 * 0x1.0p15, 0.4567 * 0x1.0p15};
```

Note 1: 最初の値は最下位で、最後の値が最上位です。上記のコードでは、`a` の最下位バイトを 1 に設定しています。

2: Q7、Q15、Q31 値は、それらの整数表現で初期化する必要があります。この例に示すように、Q7 値の整数表現は小数値に 0x1.0p7 を乗算する事で得られます。同様に、Q15 値には 0x1.0p15 を乗算し、Q31 値には 0x1.0p31 を乗算します。

下の表に、ハードウェアがサポートする v4i8 および v2q15 演算を示します。`a` と `b` は v4i8 値であり、`c` と `d` は v2q15 値です。

C コード	命令
`a + b`	`addu.qb`
`c + d`	`addq.ph`
`a - b`	`subu.qb`
`c - d`	`subq.ph`

© 2015 Microchip Technology Inc.

下の表に、ハードウェアがサポートする v2i16 演算を示します。e と f は v2i16 値です。

C コード	命令
e * f	mul.ph

型をあらかじめ定義しておくと、DSP ビルトイン関数はより容易に記述できます。

```
typedef int q31;
typedef int i32;
typedef unsigned int ui32;
typedef long long a64;
```

q31 および i32 演算は実質的に int と同じですが、q31 は Q31 小数値を示すために使い、i32 は 32 ビット整数値を示すために使います。同様に、a64 は long long と同じですが、4 個の DSP アキュムレータ ($ac0、$ac1、$ac2、$ac3) の中の 1 つに格納する値を示すために使います。

また、一部のビルトイン関数にはパラメータとして即値を使う事が望まれます (または必要です)。なぜなら、対応する DSP 命令が即値とレジスタ オペランドの両方を許容する場合と、即値だけを許容する場合があるからです。即値パラメータは以下の通りです。

```
imm0_3:0 to 3.
imm0_7:0 to 7.
imm0_15:0 to 15.
imm0_31:0 to 31.
imm0_63:0 to 63.
imm0_255:0 to 255.
imm_n32_31:-32 to 31.
imm_n512_511:-512 to 511.
```

以下のビルトイン関数は、特定の DSP 命令に直接対応します。各命令の詳細は PIC32 DSP 関連文書を参照してください。下表の各関数を使うと、コード内で DSP 命令を生成できます。例えば、`v2q15 __builtin_mips_addq_ph (v2q15, v2q15)` に対応する実際の DSP 命令は **`addq_ph`** です。

表 D-1: DSP 命令に直接対応するビルトイン関数

v2q15 __builtin_mips_addq_ph (v2q15, v2q15)
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15)
q31 __builtin_mips_addq_s_w (q31, q31)
v4i8 __builtin_mips_addu_qb (v4i8, v4i8)
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8)
v2q15 __builtin_mips_subq_ph (v2q15, v2q15)
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15)
q31 __builtin_mips_subq_s_w (q31, q31)
v4i8 __builtin_mips_subu_qb (v4i8, v4i8)
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8)
i32 __builtin_mips_addsc (i32, i32)
i32 __builtin_mips_addwc (i32, i32)
i32 __builtin_mips_modsub (i32, i32)
i32 __builtin_mips_raddu_w_qb (v4i8)
v2q15 __builtin_mips_absq_s_ph (v2q15)
q31 __builtin_mips_absq_s_w (q31)
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15)
v2q15 __builtin_mips_precrq_ph_w (q31, q31)
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31)
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15)
q31 __builtin_mips_preceq_w_phl (v2q15)
q31 __builtin_mips_preceq_w_phr (v2q15)
v2q15 __builtin_mips_precequ_ph_qbl (v4i8)
v2q15 __builtin_mips_precequ_ph_qbr (v4i8)
v2q15 __builtin_mips_precequ_ph_qbla (v4i8)
v2q15 __builtin_mips_precequ_ph_qbra (v4i8)
v2q15 __builtin_mips_preceu_ph_qbl (v4i8)
v2q15 __builtin_mips_preceu_ph_qbr (v4i8)
v2q15 __builtin_mips_preceu_ph_qbla (v4i8)
v2q15 __builtin_mips_preceu_ph_qbra (v4i8)
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7)
v4i8 __builtin_mips_shll_qb (v4i8, i32)
v2q15 __builtin_mips_shll_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shll_ph (v2q15, i32)
v2q15 __builtin_mips_shll_s_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shll_s_ph (v2q15, i32)

© 2015 Microchip Technology Inc.

表 D-1: DSP 命令に直接対応するビルトイン関数 (続き)

q31 __builtin_mips_shll_s_w (q31, imm0_31)
q31 __builtin_mips_shll_s_w (q31, i32)
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7)
v4i8 __builtin_mips_shrl_qb (v4i8, i32)
v2q15 __builtin_mips_shra_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shra_ph (v2q15, i32)
v2q15 __builtin_mips_shra_r_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shra_r_ph (v2q15, i32)
q31 __builtin_mips_shra_r_w (q31, imm0_31)
q31 __builtin_mips_shra_r_w (q31, i32)
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15)
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15)
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15)
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15)
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15)
a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8)
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8)
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8)
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8)
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15)
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31)
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15)
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31)
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15)
a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15)
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15)
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15)
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15)
i32 __builtin_mips_bitrev (i32)
i32 __builtin_mips_insv (i32, i32)
v4i8 __builtin_mips_repl_qb (imm0_255)
v4i8 __builtin_mips_repl_qb (i32)
v2q15 __builtin_mips_repl_ph (imm_n512_511)
v2q15 __builtin_mips_repl_ph (i32)
void __builtin_mips_cmpu_eq_qb (v4i8, v4i8)
void __builtin_mips_cmpu_lt_qb (v4i8, v4i8)
void __builtin_mips_cmpu_le_qb (v4i8, v4i8)
i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8)
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8)
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8)
void __builtin_mips_cmp_eq_ph (v2q15, v2q15)

© 2015 Microchip Technology Inc.

表 D-1: DSP 命令に直接対応するビルトイン関数 (続き)

void __builtin_mips_cmp_lt_ph (v2q15, v2q15)
void __builtin_mips_cmp_le_ph (v2q15, v2q15)
v4i8 __builtin_mips_pick_qb (v4i8, v4i8)
v2q15 __builtin_mips_pick_ph (v2q15, v2q15)
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15)
i32 __builtin_mips_extr_w (a64, imm0_31)
i32 __builtin_mips_extr_w (a64, i32)
i32 __builtin_mips_extr_r_w (a64, imm0_31)
i32 __builtin_mips_extr_s_h (a64, i32)
i32 __builtin_mips_extr_rs_w (a64, imm0_31)
i32 __builtin_mips_extr_rs_w (a64, i32)
i32 __builtin_mips_extr_s_h (a64, imm0_31)
i32 __builtin_mips_extr_r_w (a64, i32)
i32 __builtin_mips_extp (a64, imm0_31)
i32 __builtin_mips_extp (a64, i32)
i32 __builtin_mips_extpdp (a64, imm0_31)
i32 __builtin_mips_extpdp (a64, i32)
a64 __builtin_mips_shilo (a64, imm_n32_31)
a64 __builtin_mips_shilo (a64, i32)
a64 __builtin_mips_mthlip (a64, i32)
void __builtin_mips_wrdsp (i32, imm0_63)
i32 __builtin_mips_rddsp (imm0_63)
i32 __builtin_mips_lbux (void *, i32)
i32 __builtin_mips_lhx (void *, i32)
i32 __builtin_mips_lwx (void *, i32)
i32 __builtin_mips_bposge32 (void)

© 2015 Microchip Technology Inc.

以下のビルトイン関数は、特定の MIPS DSP REV 2 命令に直接対応付けられています。各命令の詳細は PIC32 DSP 関連文書を参照してください。

表 D-2: MIPS DSP 命令に直接対応するビルトイン関数

`v4q7 __builtin_mips_absq_s_qb (v4q7);`
`v2i16 __builtin_mips_addu_ph (v2i16, v2i16);`
`v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16);`
`v4i8 __builtin_mips_adduh_qb (v4i8, v4i8);`
`v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8);`
`i32 __builtin_mips_append (i32, i32, imm0_31);`
`i32 __builtin_mips_balign (i32, i32, imm0_3);`
`i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8);`
`i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8);`
`i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8);`
`a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16);`
`a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16);`
`a64 __builtin_mips_madd (a64, i32, i32);`
`a64 __builtin_mips_maddu (a64, ui32, ui32);`
`a64 __builtin_mips_msub (a64, i32, i32);`
`a64 __builtin_mips_msubu (a64, ui32, ui32);`
`v2i16 __builtin_mips_mul_ph (v2i16, v2i16);`
`v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16);`
`q31 __builtin_mips_mulq_rs_w (q31, q31);`
`v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15);`
`q31 __builtin_mips_mulq_s_w (q31, q31);`
`a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16);`
`a64 __builtin_mips_mult (i32, i32);`
`a64 __builtin_mips_multu (ui32, ui32);`
`v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16);`
`v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31);`
`v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31);`
`i32 __builtin_mips_prepend (i32, i32, imm0_31);`
`v4i8 __builtin_mips_shra_qb (v4i8, imm0_7);`
`v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7);`
`v4i8 __builtin_mips_shra_qb (v4i8, i32);`
`v4i8 __builtin_mips_shra_r_qb (v4i8, i32);`
`v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15);`
`v2i16 __builtin_mips_shrl_ph (v2i16, i32);`
`v2i16 __builtin_mips_subu_ph (v2i16, v2i16);`
`v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16);`
`v4i8 __builtin_mips_subuh_qb (v4i8, v4i8);`
`v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8);`

表 D-2: MIPS DSP 命令に直接対応するビルトイン関数 (続き)

v2q15 __builtin_mips_addqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_addqh_w (q31, q31);
q31 __builtin_mips_addqh_r_w (q31, q31);
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_subqh_w (q31, q31);
q31 __builtin_mips_subqh_r_w (q31, q31);
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15);

© 2015 Microchip Technology Inc.

補遺 E. ASCII キャラクタセット

表 E-1: ASCII キャラクタセット

上位コード

下位コード

Hex	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

NOTE:

© 2015 Microchip Technology Inc.

補遺 F. 改訂履歴

改訂履歴

リビジョン D (2012 年 1 月)

- 製品名を MPLAB C32 C コンパイラから MPLAB XC32 C/C++ コンパイラに変更しました。他の Microchip 社製コンパイラの文書に合わせて本書全体を再構成しました。

リビジョン E (2012 年 7 月)

- 本書全体を通して C++ に関連する情報を追加しました。
- CCI (Common Compiler Interface) 規格に関する新しい章を追加しました。

リビジョン F (2012 年 12 月)

- 「表 3-11: 一般的な最適化オプション」に「エディション」列を追加しました。
- 「第 10 章 関数」に `keep` および `optimize` 関数属性を追加しました。
- 「14.2 アセンブリ言語と C 変数 / 関数の併用」を追加しました。
- 「補遺 D. ビルトイン関数」を追加しました。
- 「補遺 E. 組み込みコンパイラ互換モード」を追加しました。
- 「補遺 F. 改訂履歴」を追加しました。以前は、この情報を「はじめに」に記載していました。
- 「サポート」を追加しました。以前は、この情報を「はじめに」に記載していました。

リビジョン G (2013 年 11 月)

- MPLAB XC32 アセンブラ、リンカ、ユーティリティ ユーザガイドの新しい名称に対する参照を本書全体を通して更新しました。
- 「2.5.2.1 絶対アドレス指定の例」を変更しました。
- 「第 3 章 プロジェクトをビルドするための手引き」を追加しました。
- 「第 4 章 XC32 ツールチェーンと MPLAB X IDE」を追加しました。
- 「5.5 ランタイム ファイル」を変更し、BOOTISA ビットに関する Note を追加しました。
- 「5.5.1 ライブラリ ファイル」を変更し、ターゲット ライブラリと対応するコマンドライン オプションに関する情報を追加しました。
- 「5.6 起動と初期化」を変更し、BOOTISA ビットに関する Note を追加しました。
- 「表 5-4: PIC32 デバイスに固有のオプション」と「表 5-16: リンク オプション」を更新しました。
- 「7.4.1 コンフィグレーション ビットのアドレス」内の PIC32 コンフィグレーション設定へのファイルパスを更新しました。
- 「7.6 C コードから FSR を使う」内のサンプルコードを更新しました。
- 「8.12 変数属性」: ターゲット デバイス上のデータ変数に関する Note を追加しました。変数のグループで属性を使うためのサンプルコードを追加しました。`coherent` および `persistent` 属性を追加しました。
- 「9.3.5 データメモリ割り当てマクロ」内の表を変更しました。

- 「9.5 プログラムメモリ内の変数」に内部フラッシュに関する Note を追加しました。
- 「9.6 レジスタ内の変数」内のサンプルコードを更新しました。
- 「9.8 メモリモデル」を変更しました。
- 「表 11-1: レジスタの用法」を更新しました。
- 「12.2.1 関数属性」内の `interrupt (priority)` の説明を変更し、`micromips` を追加しました。
- 「13.3 割り込みサービスルーチンの書き方」に ISA モードに関する Note と、exception.interrupt コードに関する Note を追加し、変更しました。
- 「13.4 ハンドラ関数に例外ベクタを割り当てる」に新しい第 1 段落を追加して変更しました。
- 「13.5 例外ハンドラ」内のテキストを変更し、「13.5.3 単純な TLB リフィル例外」と「13.5.4 キャッシュエラー例外」を追加しました。
- 「13.9 割り込みの有効化 / 無効化」内のコードを変更しました。
- 「13.10 ISR に関する注意点」内のテキストを変更しました。
- 「14.3 スタートアップ コード」内の起動コードの番号付きリストを置換しました。
- 「14.3.2 スタックポインタとヒープの初期化」内のテキストを変更しました。
- 「14.3.5 バスマトリクス レジスタの初期化」内のテキストを改訂しました。
- 「表 16-5: 定義済みマクロ」を変更しました。
- 「18.5.1 32 ビット C/C++ コンパイラマクロ」内の表に `PIC32MX`、`PIC32MZ`、`mips` のエントリを追加して更新しました。
- 「19.4 既定値リンカスクリプト」に、PIC32 MX と PIC32MZ に関する新しいテキストを追加しました。
- 「19.4.3.2　プロセッサ固有周辺モジュール　ライブラリのオプションによるインクルード」に、レガシー周辺モジュール ライブラリに関する Note を追加しました。
- 「19.4.3.5 メモリ領域」に、kseg1 のコードと L1 キャッシュ デバイスに関する Note を追加しました。
- 「19.4.4 入出力セクションマップ」に内容を追加しました。
- 補遺 D に「D.3 ビルトイン DSP 関数」を追加しました。

© 2015 Microchip Technology Inc.

サポート

はじめに

以下ではサポートについて説明します。

- myMicrochip 変更通知サービス
- Microchip 社のウェブサイト
- Microchip フォーラム
- カスタマサポート
- Microchip Technology 社への問い合わせ

myMICROCHIP 変更通知サービス

myMicrochip: http://www.microchip.com/pcn

Microchip 社の変更通知サービスは、お客様に Microchip 社製品の最新情報をお届けする配信サービスです。ご興味のある製品ファミリまたは開発ツールに関する変更、更新、エラッタ情報をいち早くメールにてお知らせします。

myMicrochip からサービスに登録し、変更通知の配信をご希望になる製品カテゴリをお選びください。よく寄せられる質問 (FAQ) と登録方法の詳細も、上記のリンク先ページからご覧になれます。

変更通知の配信をご希望される製品カテゴリをお選びになる際に、「Development Systems」を選択すると、開発ツールのリストをご覧になれます。ツールの主なカテゴリは以下の通りです。

- **コンパイラ** - Microchip 社の C コンパイラ、アセンブラ、リンカ、その他の言語ツールの最新情報です。これには MPLAB C コンパイラ全製品、MPLAB アセンブラ全製品 (MPASM™ アセンブラを含む)、MPLAB リンカ全製品 (MPLINK™ オブジェクト リンカを含む)、MPLAB ライブラリアン全製品 (MPLIB™ オブジェクト ライブラリアンを含む) が含まれます。
- **エミュレータ** - Microchip 社製インサーキット エミュレータの最新情報です。これには MPLAB REAL ICE™ インサーキット エミュレータが含まれます。
- **インサーキット デバッガ** - Microchip 社のインサーキット デバッガの最新情報です。これには PICkit™ 2、PICkit 3、MPLAB ICD 3 インサーキット デバッガが含まれます。
- **MPLAB® IDE** ― Microchip 社の MPLAB IDE(開発システムツール向け Windows® 統合開発環境) の最新情報です。これには MPLAB IDE、MPLAB IDE プロジェクト マネージャ、MPLAB エディタ、MPLAB SIM シミュレータ、一般的な編集およびデバッグ機能が含まれます。
- **プログラマ** - Microchip 社製プログラマの最新情報です。これには、デバイス (量産品) プログラマの MPLAB REAL ICE インサーキット エミュレータ、MPLAB ICD 3 インサーキット デバッガ、MPLAB PM3、および開発用プログラマの PICkit 2 および 3 が含まれます。
- **スタータ / デモボード** - これには MPLAB スタータキット ボード、PICDEM デモボード、その他の評価用ボードが含まれます。

Microchip 社のウェブサイト

ウェブサイト :http://www.microchip.com

Microchip 社は、自社が運営するウェブサイトを通してオンライン サポートを提供しています。当ウェブサイトでは、お客様に役立つ情報やファイルを簡単に見つけ出せます。一般的なインターネット ブラウザから以下の内容がご覧になれます。

- **製品サポート** - データシートとエラッタ、アプリケーション ノート、サンプルコード、設計リソース、ユーザガイドとハードウェア サポート文書、最新ソフトウェア リリース、ソフトウェア アーカイブ
- **一般的技術サポート** - よく寄せられる質問 (FAQ)、技術サポートのご依頼、オンライン ディスカッション グループ、Microchip 社のコンサルタント プログラムおよびメンバーリスト
- **ご注文とお問い合わせ** - 製品セレクタと注文ガイド、最新プレスリリース、セミナー / イベントの一覧、お問い合わせ先 (営業所 / 販売代理店) の一覧

Microchip フォーラム

フォーラム : http://www.microchip.com/forums

Microchip 社のウェブフォーラムからもオンライン サポートをご利用になれます。現在、以下のフォーラムを開設しています。

- 開発ツール
- 8 ビット PIC MCU
- 16 ビット PIC MCU
- 32 ビット PIC MCU

カスタマサポート

Microchip 社製品をお使いのお客様は、以下のチャンネルからサポートをご利用になれます。

- 販売代理店
- 各地の営業所
- フィールド アプリケーション エンジニア (FAE)
- 技術サポート

サポートについては販売代理店、販売担当者、フィールド アプリケーション エンジニア (FAE) までお問い合わせください。もしくは弊社までご連絡ください。本書の最後のページには各国の営業所の一覧を記載しています。最新の営業所一覧は、弊社ウェブサイトでご確認ください。

技術サポート : http://support.microchip.com

本書の内容に関して、誤りやご意見がございましたら、docerrors@microchip.com までメールでお寄せください。

Microchip Technology 社への問い合わせ

Microchip Technology 社まで電話またはファックスでお問い合せください。

電話 : (480) 792-7200

Fax: (480) 792-7277

© 2015 Microchip Technology Inc.

用語集

A

絶対セクション (Absolute Section)

リンカで変更されない固定 (絶対) アドレスを持つ GCC コンパイラのセクション。

絶対変数 / 関数 (Absolute Variable/Function)

OCG コンパイラの `@ address` 構文を使って絶対アドレスに配置される変数または関数。

アクセスメモリ (Access Memory)

PIC18 のみ - PIC18 でバンクセレクト レジスタ (BSR) の設定にかかわらずアクセスできる特殊なレジスタ。

アクセス エントリポイント (Access Entry Points)

リンク時に定義されていない可能性のある関数に、セグメントの境界を越えて制御を渡すための手段。ブートセグメントとセキュア アプリケーション セグメントを別々にリンクする方法も提供する。

アドレス (Address)

メモリ内の位置を一意に特定する値。

アルファベット文字 (Alphabetic Character)

アルファベットの小文字と大文字の総称 (a, b, …, z, A, B, …, Z)。

英数字 (Alphanumeric)

アルファベット文字と 0 ～ 9 の 10 進数の数字の総称 (0,1, …, 9)。

AND 条件ブレークポイント (ANDed Breakpoints)

プログラム実行を停止するために設定する AND 条件 (ブレークポイント 1 とブレークポイント 2 が同時に発生した場合のみプログラム実行を停止する)。AND 条件で実行が停止するのは、データメモリのブレークポイントとプログラムメモリのブレークポイントが同時に発生した場合のみ。

匿名構造体 (Anonymous Structure)

16 ビット C コンパイラ **–** 無名の構造体。

PIC18 C コンパイラ **-** C 共用体のメンバーである無名の構造体。匿名構造体のメンバーは、その構造体を包含している共用体のメンバーと同じようにアクセスできる。例えば以下のサンプルコードでは、`hi` と `lo` は共用体 `caster` に含まれる匿名構造体のメンバーである。

```
union castaway
 int intval;
 struct {
  char lo; //accessible as caster.lo
  char hi; //accessible as caster.hi
 };
} caster;
```

ANSI

American National Standards Institute (米国規格協会) の略。米国における標準規格の策定と承認を行う団体。

アプリケーション (Application)

PIC® マイクロコントローラで制御されるソフトウェアとハードウェアを組み合わせたもの。

アーカイブ / アーカイバ (Archive/Archiver)

アーカイブ / ライブラリは、再配置可能なオブジェクト モジュールの集まり。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た後、アーカイバ / ライブラリアンを使ってこれらオブジェクト ファイルを 1 つのアーカイブ / ライブラリ ファイルにまとめると生成される。アーカイブ / ライブラリをオブジェクト モジュールや他のアーカイブ / ライブラリとリンクすると、実行コードが生成される。

ASCII

American Standard Code for Information Interchange の略。7 桁の 2 進数で 1 つの文字を表現する文字セットエンコード方式。大文字、小文字、数字、記号、制御文字等を含む。

アセンブリ / アセンブラ (Assembly/Assembler)

アセンブリとは、2 進数のマシンコードをシンボル表現で記述したプログラミング言語。アセンブラとは、アセンブリ言語のソースコードをマシンコードに変換する言語ツール。

割り当てセクション (Assigned Section)

リンカのコマンドファイルで特定のターゲット メモリブロックに割り当てられた GCC コンパイラのセクション。

非同期 (Asynchronously)

複数のイベントが同時には発生しない事。一般に、プロセッサ実行中の任意の時点で発生する割り込みに言及する際に使う。

非同期スティミュラス (Asynchronous Stimulus)

シミュレータ デバイスへの外部入力をシミュレートするために生成されるデータ。

属性 (Attribute)

GCC の C プログラムの変数または関数の特徴を表す情報で、マシン固有の特性を記述する目的で使う。

属性 (セクション属性) (Attribute、Section)

「executable」、「readonly」、「data」等、GCC のセクションの特徴を表す情報。アセンブラの `.section` ディレクティブでフラグとして指定できる。

B

2 進数 (Binary)

0 と 1 の数字を使う、2 を底とした記数法。一番右の桁が 1 の位、次の桁が 2 の位、その次の桁が $2^2 = 4$ の位を表す。

ブックマーク (Bookmark)

ファイル内の特定の行に簡単な操作でアクセスできるようにする機能。

[Editor] ツールバーの [Toggle Bookmarks] を選択してブックマークを追加または削除する。このツールバーの他のアイコンをクリックすると、次または前のブックマークに移動する。

ブレークポイント (Breakpoint)

ハードウェア ブレークポイント : 実行するとファームウェアの実行が停止するイベント。

ソフトウェア ブレークポイント : ファームウェアの実行が停止するアドレス。通常、特別な Break 命令で実行が停止される。

© 2015 Microchip Technology Inc.

ビルド (Build)

全てのソースファイルのコンパイルとリンクを行ってアプリケーションを作成する事。

C

C\C++

C 言語は、簡潔な表現、現代的な制御フローとデータ構造、豊富に用意された演算子等を特長とする汎用プログラミング言語。C++ とは、C 言語のオブジェクト指向バージョン。

校正メモリ (Calibration Memory)

PIC マイクロコントローラの内蔵 RC オシレータやその他の周辺モジュールの校正値を格納するための特殊機能レジスタまたはレジスタ。

中央演算処理装置 (Central Processing Unit)

デバイス内で、実行する正しい命令をフェッチし、デコードして実行する装置。必要に応じて、算術論理演算装置 (ALU) と組み合わせて命令実行を完了する。プログラムメモリのアドレスバス、データメモリのアドレスバス、スタックへのアクセスを制御する。

クリーン (Clean)

クリーンする事により、アクティブなプロジェクトのオブジェクト ファイル、Hex ファイル、デバッグファイル等、全ての中間ファイルが削除される。これらのファイルは、プロジェクトのビルド時に他のファイルから再構築される。

COFF

Common Object File Format の略。このフォーマットのオブジェクト ファイルは、マシンコードの他、デバッグ等に関する情報を含む。

コマンドライン インターフェイス (Command Line Interface)

プログラムとユーザのやり取りをテキストの入出力だけで行う方法。

コンパイルド スタック (Compiled Stack)

コンパイラが管理するメモリの領域で、この領域内で変数に静的に空間を割り当てる。ターゲット デバイス上にソフトウェア スタックまたはハードウェア スタックのメカニズムを効率的に実装できない場合、コンパイルド スタックがソフトウェア スタックまたはハードウェア スタックに置き換わる。

コンパイラ (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。

条件付きアセンブリ (Conditional Assembly)

アセンブリ言語で、ある特定の式のアセンブル時の値に基づいて含まれたり除外されたりするコード。

条件付きコンパイル (Conditional Compilation)

プログラムの一部を、プリプロセッサ ディレクティブで指定した特定の定数式が真の場合のみコンパイルする事。

コンフィグレーション ビット (Configuration Bits)

PIC MCU と dsPIC DSC の動作モードを設定するために書き込む専用ビット。コンフィグレーション ビットは事前プログラミングされている場合とされていない場合がある。

制御ディレクティブ (Control Directives)

アセンブリ言語コード内で使うディレクティブで、指定した式のアセンブル時の値に基づいてコードを含めるか除外するかを決定する。

CPU

「中央演算処理装置」参照。

相互参照ファイル (Cross Reference File)

シンボルテーブルとそのシンボルを参照するファイルリストを参照するファイル。シンボルが定義されている場合、リストの最初のファイルがシンボル定義の位置となる。残りのファイルにはシンボルへの参照が含まれる。

D

データ ディレクティブ (Data Directives)

アセンブラによって行われるプログラムメモリまたはデータメモリの割り当てを制御するディレクティブ。データ項目をシンボル (意味のある名前) を使って参照する手段としても使われる。

データメモリ (Data Memory)

Microchip 社の MCU と DSC では、データメモリ (RAM) は汎用レジスタ (GPR) と特殊機能レジスタ (SFR) で構成される。EEPROM データメモリを内蔵したデバイスもある。

データ監視および制御インターフェイス (DMCI)

MPLAB X IDE 内のツール。このインターフェイスは、プロジェクト内のアプリケーション変数の動的な入力制御を提供する。4 つの動的に割り当て可能なグラフィックウィンドウを使って、アプリケーションが生成するデータをグラフィカルに表示できる。

デバッグ / デバッガ (Debug/Debugger)

ICE/ICD 参照。

デバッグ情報 (Debugging Information)

コンパイラとアセンブラでこのオプションを選択すると、アプリケーション コードのデバッグに使える各種レベルの情報を出力できる。デバッグ オプションの選択の詳細はコンパイラまたはアセンブラのマニュアル参照。

推奨しない機能 (Deprecated Features)

後方互換性のためにサポートされているだけで現在は使用されておらず、いずれ廃止される事が決まっている機能。

デバイス プログラマ (Device Programmer)

マイクロ コントローラ等、電気的に書き込み可能な半導体デバイスをプログラミングするためのツール。

デジタルシグナル コントローラ (Digital Signal Controller)

デジタル信号処理機能をサポートしたマイクロコントローラ。Microchip 社の dsPIC DSC 等。

デジタル信号処理 / デジタルシグナル プロセッサ (Digital Signal Processing/Digital Signal Processor)

デジタル信号処理 (DSP) とは、デジタル信号をコンピュータで処理する事。通常は、アナログ信号 (音声または画像) をデジタル形式に変換 (サンプリング) して処理する事をいう。デジタルシグナル プロセッサとは、信号処理用に設計されたマイクロプロセッサの事。

ディレクティブ (Directives)

言語ツールの動作を制御するためにソースコードに記述するステートメント。

ダウンロード (Download)

ホストから別のデバイス (エミュレータ、プログラマ、ターゲットボード等) にデータを送信する事。

DWARF

Debug With Arbitrary Record Formatの略。ELFファイルのデバッグ情報フォーマット。

© 2015 Microchip Technology Inc.

E

EEPROM

Electrically Erasable Programmable Read Only Memory の略。電気的に消去可能なタイプの PROM。データの書き込みと消去をバイト単位で行う。EEPROM は電源を OFF にしても内容を保持する。

ELF

Executable and Linking Format の略。この形式のオブジェクト ファイルはマシンコードを含む。デバッグその他の情報は DWARF で指定する。ELF/DWARF の方が COFF よりも最適化したコードのデバッグに適している。

エミュレーション / エミュレータ (Emulation/Emulator)

ICE/ICD 参照。

エンディアン (Endianness)

マルチバイト オブジェクトにおけるバイトの並び順。

環境 (Environment)

MPLAB PM3 – デバイスのプログラミングに関する設定ファイルを保存したフォルダ。このフォルダを SD/MMC カードに転送できる。

エピローグ (Epilogue)

コンパイラで生成したコードのうち、スタック領域の割り当て解除、レジスタの復帰、ランタイムモデルで指定したその他のマシン固有の要件を実行するコード部分。エピローグは関数のユーザコードの後、関数リターンの直前に実行される。

EPROM

Erasable Programmable Read Only Memory の略。再書き込みが行えるタイプの ROM で、消去は紫外線照射で行うものが主流。

エラー / エラーファイル (Error/Error File)

プログラムの処理を継続できない問題が発生するとエラーとして報告される。可能な場合、エラーは問題が発生したソースファイル名と行番号を特定する。エラーファイルは、言語ツールから出力されたエラーメッセージと診断結果を格納する。

イベント (Event)

アドレス、データ、パスカウント、外部入力、サイクルタイプ (フェッチ、R/W)、タイムスタンプ等、バスサイクルを記述したもの。トリガ、ブレークポイント、割り込みを記述するために使う。

実行可能コード (Executable Code)

読み込んで実行できる形式のソフトウェア。

エクスポート (Export)

MPLAB IDE/MPLAB X IDE のデータを標準フォーマットで外部に出力する事。

式 (Expressions)

算術演算子または論理演算子で区切った定数または記号の組み合わせ。

拡張マイクロコントローラ モード (Extended Microcontroller Mode)

拡張マイクロコントローラ モードでは、内蔵プログラムメモリと外部メモリの両方が利用できる。プログラムメモリのアドレスが PIC18 の内部メモリ空間より大きい場合、自動的に外部メモリの実行に切り換わる。

拡張モード (Extended Mode) (PIC18 MCU)

コンパイラの動作モードの 1 つ。拡張命令 (`ADDFSR`、`ADDULNK`、`CALLW`、`MOVSF`、`MOVSS`、`PUSHL`、`SUBFSR`、`SUBULNK`) とリテラル オフセットによるインデックス アドレス指定を利用できる。

外部ラベル (External Label)

外部リンケージを持つラベル。

© 2015 Microchip Technology Inc.

外部リンケージ (External Linkage)

関数や変数が、それ自身が定義されたモジュールの外部から参照できる場合、外部リンケージを持つという。

外部シンボル (External Symbol)

外部リンケージを持つ識別子のシンボル。参照の場合と定義の場合がある。

外部シンボル解決 (External Symbol Resolution)

リンカが全ての入力モジュールの外部シンボル定義を 1 つにまとめ、全ての外部シンボル参照を解決しようとするプロセス。外部シンボル参照に対応する定義が存在しない場合、リンカエラーとなる。

外部入力ライン (External Input Line)

外部信号に基づいてイベントを設定するための外部入力信号ロジックプローブ ライン (TRIGIN)。

外部 RAM (External RAM)

オフチップの読み書き可能なメモリ。

F

致命的エラー (Fatal Error)

コンパイルがただちに停止するようなエラー。メッセージも出力されない。

ファイル レジスタ (File Registers)

汎用レジスタ (GPR) と特殊機能レジスタ (SFR) で構成される内蔵のデータメモリ。

フィルタ (Filter)

トレース ディスプレイまたはデータファイルにどのデータを含めるか / 除外するかを選択するもの。

フィックスアップ (Fixup)

リンカによる再配置後にオブジェクト ファイルのシンボル参照を絶対アドレスに置き換える処理。

フラッシュ (Flash)

データの書き込みと消去をバイト単位ではなくブロック単位で行えるタイプの EEPROM。

FNOP

Forced No Operation の略。Forced NOP サイクルは、2 サイクル命令の 2 サイクル目で発生する。PIC 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構造となっており、現在の命令を実行中に物理アドレス空間の次の命令をプリフェッチする。しかし、現在の命令によってプログラム カウンタが変化した場合、プリフェッチした命令は明示的に無視され、Forced NOP サイクルが発生する。

フレームポインタ (Frame Pointer)

スタックベースの引数とスタックベースのローカル変数の境界となるスタック番地を指し示すポインタ。ここを基準にすると、現在の関数のローカル変数やその他の値に容易にアクセスできる。

© 2015 Microchip Technology Inc.

フリースタンディング (Free-Standing)

複素数型を使っておらず、ライブラリ (ANSI C89 規格第 7 節) で規定する機能の使用が標準ヘッダ (`<float.h>`、`<iso646.h>`、`<limits.h>`、`<stdarg.h>`、`<stdbool.h>`、`<stddef.h>`、`<stdint.h>`) の内容のみに限定されている厳密な規格合致プログラムを受理する処理系。

G

GPR

General Purpose Register (汎用レジスタ) の略。デバイスのデータメモリ (RAM) のうち、汎用目的に使える部分。

H

Halt

プログラム実行を停止する事。Halt を実行する事は、ブレークポイントで停止する事と同じ。

ヒープ (Heap)

動的メモリ割り当てに使うメモリ空間。メモリブロックの割り当てと解放は実行時に任意の順序で行う。

HEX コード /HEX ファイル (Hex Code/Hex File)

HEX コードは、実行可能な命令を 16 進数形式のコードで保存したもの。HEX ファイルは、HEX コードを格納したファイル。

16 進数 (Hexadecimal)

0 ～ 9 の数字と A ～ F(または a ～ f) のアルファベットを使った、16 を底とした記数法。16 進数の A ～ F は、10 進数の 10 ～ 15 を表す。一番右の桁が 1 の位、次の桁が 16 の位、その次の桁が $16^2 = 256$ の位を表す。

高級言語 (High Level Language)

プログラムを記述するための言語で、プロセッサから見てアセンブリよりも遠い位置関係にあるもの。

I

ICE/ICD

インサーキット エミュレータ / インサーキット デバッガの略。ターゲット デバイスのデバッグとプログラミングを行うためのハードウェア ツール。エミュレータは、デバッガよりも多くの機能 (トレース等) を備える。

インサーキット エミュレーション / インサーキット デバッグとは、インサーキット エミュレータまたはデバッガを使った作業の事を指す。

-ICE/-ICD: インサーキット エミュレーション / デバッグ用の回路を内蔵したデバイス (MCU または DSC)。このデバイスは必ずヘッダ基板にマウントし、インサーキット エミュレータまたはデバッガによるデバッグ用に使う。

ICSP

In-Circuit Serial Programming (インサーキット シリアル プログラミング) の略。Microchip 社製の組み込みデバイスをシリアル通信を利用して最小限のデバイスピンでプログラミングする方法。

IDE

Integrated Development Environment の略。MPLAB IDE/MPLAB X IDE の IDE と同じ意味。

識別子 (Identifier)

関数または変数の名前。

IEEE

Institute of Electrical and Electronics Engineers の略。

インポート (Import)

Hex ファイル等の外部ソースから MPLAB IDE/MPLAB X IDE にデータを取り込む事。

初期化済みデータ (Initialized Data)

初期値を指定して定義されたデータ。C では、

```
int myVar=5;
```

として定義した変数は初期化済みデータセクションに格納する。

命令セット (Instruction Set)

特定のプロセッサが理解できるマシン語命令の集合。

命令 (Instructions)

CPU に対して特定の演算を実行するように指示するビット列。演算の対象となるデータを含める事もできる。

内部リンケージ (Internal Linkage)

関数または変数が、それ自身を定義したモジュールの外部からアクセスできない場合、内部リンケージを持つという。

国際標準化機構 (International Organization for Standardization)

コンピューティングや通信をはじめ、多くのテクノロジとビジネス関連の標準規格の策定を行っている団体。一般的に ISO と呼ぶ。

割り込み (Interrupt)

CPU に対する信号の一種。この信号が発生すると、現在動作中のアプリケーションの実行を一時停止し、制御を割り込みサービスルーチン (ISR) に渡してイベントを処理する。ISR の実行が完了すると、通常のアプリケーションの実行が再開される。

割り込みハンドラ (Interrupt Handler)

割り込み発生時に専用のコードを実行するルーチン。

割り込みサービス要求 (Interrupt Service Request (IRQ))

プロセッサの通常の命令実行を一時的に停止し、割り込みハンドラルーチンの実行開始を要求するイベント。プロセッサによっては複数の割り込み要求イベントを持ち、優先度の異なる割り込みを処理できるものもある。

割り込みサービスルーチン (ISR)

言語ツールの場合、割り込みを処理する関数。

MPLAB IDE/MPLAB X IDE の場合、割り込みが発生すると実行されるユーザ作成コード。通常、発生した割り込みの種類によってプログラムメモリ内の異なる位置のコードを実行する。

割り込みベクタ (Interrupt Vector)

割り込みサービスルーチンまたは割り込みハンドラのアドレス。

L

左辺値 (L-value)

検査または変更が可能なオブジェクトを指し示す式。左辺値は代入演算子の左側で使う。

レイテンシ (Latency)

イベントが発生してからその応答までの時間の長さ。

ライブラリ / ライブラリアン (Library/Librarian)

「アーカイブ / アーカイバ」参照。

© 2015 Microchip Technology Inc.

リンカ (Linker)

オブジェクト ファイルとライブラリを結合し、モジュール間の参照を解決して実行可能コードを生成する言語ツール。

リンカ スクリプト ファイル (Linker Script Files)

リンカのコマンド ファイル。リンカのオプションを定義し、ターゲット プラットフォームで利用可能なメモリを記述する。

リスティング ディレクティブ (Listing Directives)

アセンブラのリスティング ファイルのフォーマットを制御するディレクティブ。タイトルや改ページ指示等、リスティング ファイルに関する各種の設定を行う。

リスティング ファイル (Listing File)

ソースファイルにある各 C ソース 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、マクロに対して生成されたマシンコードを記述した ASCII テキストファイル。

リトル エンディアン (Little Endian)

マルチバイト データで最下位バイト (LSB) を最下位アドレスに格納するデータ並び順方式。

ローカルラベル (Local Label)

マクロ内で LOCAL ディレクティブを使って定義されたラベル。ローカルラベルは、マクロの同一インスタンス内でのみ有効。すなわち、LOCAL として宣言されたシンボルとラベルには、ENDM マクロ以降はアクセスできない。

ロジックプローブ (Logic Probes)

Microchip 社製エミュレータには、最大 14 のロジックプローブを接続できるものがある。ロジックプローブは、外部トレース入力、トリガ出力信号、+5 V、共通グランドを提供する。

ループバック テストボード (Loop-Back Test Board)

MPLAB REAL ICE インサーキット エミュレータの動作をテストするために用いる。

LVDS

Low Voltage Differential Signaling の略。銅線を使って低ノイズ、低消費電力、低振幅でデータを高速伝送 (Gbps) する方法。

標準の I/O シグナリングでは、データストレージは実際の電圧レベルに依存する。電圧レベルは信号線の長さに影響を受ける (信号線が長いと抵抗が増え電圧が下がる)。これに対し LVDS では、データストレージは実際の電圧レベルでなく正と負の電圧値によってのみ区別する。従って、長い信号線でもクリアで安定したデータストリームを維持した伝送が可能。

出典 :http://www.webopedia.com/TERM/L/LVDS.html.

M

マシンコード (Machine Code)

コンピュータ プログラムをプロセッサが実際に読み出して解釈できる形式で表現したもの。2 進数のマシンコードで記述されたプログラムは、マシン命令のシーケンス (命令間にデータを挟む事もある) から成る。ある特定のプロセッサで使用できる命令の集合を「命令セット」という。

マシン語 (Machine Language)

ある CPU が翻訳を必要とせず実行できる命令の集合。

マクロ (Macro)

マクロ命令。一連の命令シーケンスを短い名前で表現した命令。

マクロ ディレクティブ (Macro Directives)

マクロ定義の中で実行とデータ割り当てを制御するディレクティブ。

make ファイル (Makefile)

プロジェクトの Make に関する指示をファイルにエクスポートしたもの。このファイルは、MPLAB IDE/MPLAB X IDE 以外の環境で `make` コマンドを実行してプロジェクトをビルドする際に使う。

Make Project

アプリケーションを再ビルドするコマンド。前回の完全なコンパイル後に変更されたソースファイルのみを再コンパイルする。

MCU

Microcontroller Unit の略。マイクロコントローラの事。「uC」と表記する事もある。

メモリモデル (Memory Model)

C コンパイラの場合、アプリケーションで利用可能なメモリを表現したもの。PIC18 C コンパイラの場合、プログラムメモリを指し示すポインタのサイズに関する規定を記述したもの。

メッセージ (Message)

言語ツールの動作に問題が発生した事を知らせる文字列。メッセージが表示されても処理は停止しない。

マイクロコントローラ (Microcontroller)

CPU、RAM、プログラムメモリ、I/O ポート、タイマ等、多くの機能が統合されたチップ。

マイクロコントローラ モード (Microcontroller Mode)

PIC18 マイクロコントローラで設定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、内部実行のみを許可する。つまり、マイクロコントローラ モードでは内蔵プログラムメモリしか利用できない。

マイクロプロセッサ モード (Microprocessor Mode)

PIC18 マイクロコントローラで設定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、内蔵プログラムメモリは使わない。プログラムメモリ全体が外部にマッピングされる。

ニーモニック (Mnemonics)

マシンコードと 1 対 1 で対応したテキスト命令。オペコードとも呼ぶ。

モジュール (Module)

プリプロセッサ ディレクティブ実行後の前処理済みのソースファイル出力。翻訳単位とも呼ぶ。

MPASMª アセンブラ (MPASM Assembler)

PIC マイクロコントローラ、KeeLoq®、Microchip 社のメモリデバイスに対応した Microchip 社の再配置可能なマクロアセンブラ。

MPLAB *(言語ツール名)* for *(デバイス名)* (MPLAB Language Tool for Device)

特定のデバイスに対応した Microchip 社の C コンパイラ、アセンブラ、リンカ。言語ツールは、アプリケーションで使用するデバイスに対応したものを選択する必要がある。例えば PIC18 MCU 用の C コードを作成する場合は「MPLAB C Compiler for PIC18 MCU」を使用する。

 © 2015 Microchip Technology Inc.

MPLAB ICD

MPLAB IDE/MPLAB X IDE と連携する Microchip 社のインサーキット デバッガ。「ICE/ICD」参照。

MPLAB IDE/MPLAB X IDE

Microchip 社の統合開発環境。エディタ、プロジェクト マネージャ、シミュレータが付属する。

MPLAB PM3

Microchip 社のデバイス プログラマ。PIC18 マイクロ コントローラと dsPIC デジタルシグナル コントローラのプログラムに対応。MPLAB IDE/MPLAB X IDE との併用も、単体での使用も可能。PROMATE II の後継製品。

MPLAB REAL ICE™ インサーキット エミュレータ

MPLAB IDE/MPLAB X IDE と組み合わせて使う Microchip 社の次世代インサーキット エミュレータ。「ICE/ICD」参照。

MPLAB SIM

MPLAB IDE/MPLAB X IDE と組み合わせて使う Microchip 社のシミュレータで、PIC MCU と dsPIC DSC に対応する。

MPLIB™ オブジェクト ライブラリアン (MPLIB Object Librarian)

MPLAB IDE/MPLAB X IDE と組み合わせて使う Microchip 社のライブラリアン。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 C18 C コンパイラで作成した COFF オブジェクト モジュールに使うオブジェクト ライブラリアン。

MPLINK™ オブジェクト リンカ (MPLINK Object Linker)

Microchip 社の MPASM アセンブラと C18 C コンパイラに対応したオブジェクト リンカ。Microchip 社の MPLIB ライブラリアンと一緒に使う事も可能。MPLAB IDE/MPLAB X IDE との併用を前提に設計されているが、必須ではない。

MRU

Most Recently Used の略。最近使ったファイルとウィンドウの事。MPLAB IDE/MPLAB X IDE のメインメニューで選択できる。

N

ネイティブ データサイズ (Native Data Size)

ネイティブ トレースの場合、[Watch] ウィンドウで使用する変数のサイズは選択したデバイスのデータメモリと同じサイズ (PIC18 の場合は同じバイトサイズ、16 ビット デバイスの場合は同じワードサイズ) である必要がある。

入れ子の深さ (Nesting Depth)

マクロに他のマクロを含める事のできる階層の数。

ノード (Node)

MPLAB IDE/MPLAB X IDE のプロジェクトを構成するコンポーネント。

非拡張モード (Non-Extended Mode) (PIC18 MCU)

コンパイラの動作モードの 1 つ。拡張命令もリテラル オフセットによるインデックス アドレス指定も使わない。

非リアルタイム (Non Real Time)

ブレークポイントで停止中、またはシングルステップ実行中のプロセッサ、あるいはシミュレータ モードで動作中の MPLAB IDE/MPLAB X IDE を指す。

不揮発性ストレージ (Non-Volatile Storage)

電源を OFF にしても内容が失われないストレージ デバイス。

NOP

No Operation の略。実行してもプログラム カウンタが進むだけで何も動作を行わない命令。

O

オブジェクト コード / オブジェクト ファイル (Object Code/Object File)

オブジェクト コードとは、アセンブラまたはコンパイラで生成されるマシンコードの事。オブジェクト ファイルとは、マシンコードを格納したファイル。デバッグ情報を含む事もある。そのまま実行できるものと、他のオブジェクト ファイル (ライブラリ等) とリンクしてから完全な実行プログラムを生成する再配置可能形式のものがある。

オブジェクト ファイル ディレクティブ (Object File Directives)

オブジェクト ファイル作成時にのみ使うディレクティブ。

8 進数 (Octal)

0 ～ 7 の数字のみを使う、8 を底とした記数法。一番右の桁が 1 の位、次の桁が 8 の位、その次の桁が $8^2 = 64$ の位を表す。

オフチップメモリ (Off-Chip Memory)

PIC18 で選択できるメモリオプション。ターゲットボードのメモリを使うか、または全てのプログラムメモリをエミュレータから供給する。***[Options]>[Development Mode]*** の順にクリックして [Memory] タブでオフチップメモリを選択する。

オペコード (Opcodes)

Operational Code の略。「ニーモニック」参照。

演算子 (Operators)

定義可能な式を構成する際に使用される「+」や「-」等の記号。各演算子に割り当てられた優先順位に基づいて式が評価される。

OTP

One Time Programmable の略。パッケージに窓のない EPROM デバイス。EPROM を消去するには紫外線照射が必要なため、パッケージに窓のあるデバイスしか消去できない。

P

パスカウンタ (Pass Counter)

イベント (特定のアドレスの命令を実行する等) が発生するたびに値をデクリメントするカウンタ。パスカウンタの値がゼロになると、イベントの条件を満たす。パスカウンタはブレークロジック、トレースロジック、複合トリガダイアログの任意のシーケンシャル イベントに割り当てられる。

PC

パーソナル コンピュータまたはプログラム カウンタの略。

ホスト PC (PC Host)

サポートされた Windows オペレーティング システムが動作する PC。

永続データ (Persistent Data)

クリアも初期化もされないデータ。デバイスをリセットしてもアプリケーションがデータを保持できるようにするために使う。

ファントムバイト (Phantom Byte)

dsPIC アーキテクチャで、24 ビット命令ワードを 32 ビット命令ワードと見なして扱う場合に使う未実装バイト。dsPIC の hex ファイルに見られる。

© 2015 Microchip Technology Inc.

PIC MCU (PIC MCUs)

Microchip 社の全てのマイクロ コントローラ ファミリの総称。

PICkit 2/3

Microchip 社の開発用デバイス プログラマで、Debug Express によるデバッグ機能を備える。サポートしているデバイスの種類は、各ツールの Readme ファイル参照。

プラグイン (Plug-ins)

MPLAB IDE/MPLAB X IDE では、標準コンポーネントにプラグイン モジュールを追加する事で、各種ソフトウェア / ハードウェア ツールに対応する。一部のプラグイン ツールは、[Tools] メニューから利用できる。

ポッド (Pod)

インサーキット エミュレータまたはデバッガの筐体。丸型の場合「パック」(Puck) と呼ぶ事もある。あるいは「プローブ」(Probe) とも呼ぶが、「論理プローブ」と混同せぬよう注意が必要。

パワーオン リセット エミュレーション (Power-on-Reset Emulation)

データ RAM 領域にランダムな値を書き込んで、初回電源投入時の RAM の非初期化値をシミュレートするソフトウェア ランダム化処理。

プラグマ (Pragma)

特定のコンパイラにとって意味を持つディレクティブ。一般に、実装で定義した情報をコンパイラに伝達するために使う。MPLAB C30 は属性を利用してこの情報を伝達する。

優先順位 (Precedence)

式の評価順を定義した規則。

量産プログラマ (Production Programmer)

デバイスを高速にプログラミングできるようにリソースを強化したプログラマ。各種電圧レベルでのプログラミングに対応し、プログラミング仕様に完全に準拠している。量産環境では応用回路が組み立てラインにとどまる時間をなるべく短くする必要があるため、デバイスへの書き込み時間が特に重要である。

プロファイル (Profile)

MPLAB SIM シミュレータにおいて、実行したスティミュラスをレジスタ別に一覧表示したもの。

プログラム カウンタ (Program Counter)

現在実行中の命令のアドレスを格納した場所。

プログラム カウンタ ユニット (Program Counter Unit)

16 ビットアセンブラ - プログラムメモリのレイアウトを概念的に表現したもの。プログラム カウンタは 1 命令ワードで 2 つインクリメントする。実行可能セクションでは、2 プログラム カウンタ ユニットは 3 バイトに相当する。読み出し専用セクションでは、2 プログラム カウンタ ユニットは 2 バイトに相当する。

プログラムメモリ

MPLAB IDE/MPLAB X IDE - デバイス内で命令を保存するメモリ空間。また、エミュレータまたはシミュレータにダウンロードしたターゲット アプリケーションのファームウェアを格納するメモリ空間もプログラムメモリと呼ぶ。

16 ビット アセンブラ / コンパイラ – デバイス内で命令が保存されるメモリ領域。

プロジェクト (Project)

アプリケーションのビルドに必要なファイル (ソースコードやリンカスクリプト ファイル等) 一式と、各種ビルドツールやビルドオプションとの関連付けをまとめたもの。

プロローグ (Prologue)

コンパイラで生成したコードのうち、スタック領域の割り当て、レジスタの退避、ランタイムモデルで指定したその他のマシン固有の要件を実行するコード部分。プロローグは、関数のユーザコードの前に実行する。

プロトタイプ システム (Prototype System)

ユーザのターゲット アプリケーションまたはターゲットボードの事。

Psect

GCC のセクションに相当する OCG の用語。プログラム セクション (program section) の略語。リンカが 1 つのまとまりとして処理するコードまたはデータのブロック。

PWM 信号 (PWM Signals)

パルス幅変調 (Pulse Width Modulation) 信号。一部の PIC MCU は周辺モジュールとして PWM を内蔵している。

Q

修飾子 (Qualifier)

パスカウンタで使ったり、複合トリガにおける次の動作前のイベントとして使ったりするアドレスまたはアドレスレンジ。

R

基数 (Radix)

アドレスを指定する際の記数法 (16 進法、10 進法) の底。

RAM

Random Access Memory の略。データメモリ。任意の順にメモリ内の情報にアクセスできる。

生データ (Raw Data)

あるセクションに関連付けられたコードまたはデータを 2 進数で表現したもの。

読み出し専用メモリ (Read Only Memory)

恒久的に保存されているデータへの高速アクセスが可能なメモリ ハードウェア。ただし、データの追加や変更は不可。

リアルタイム (Real Time)

インサーキット エミュレータまたはデバッガが Halt 状態から解放されると、プロセッサの実行はリアルタイム モードとなり、通常のチップと同じ挙動をする。リアルタイム モードでは、エミュレータのリアルタイム トレースバッファが有効になり、選択した全てのサイクルを常時キャプチャする。また、全てのブレークロジックが有効になる。インサーキット エミュレータまたはデバッガでは、有効なブレークポイントで停止するか、またはユーザが実行を停止するまでプロセッサはリアルタイムで動作する。

シミュレータでは、ホストCPUでシミュレート可能な最大速度でマイクロコントローラの命令を実行する事をリアルタイムと呼ぶ。

再帰呼び出し (Recursive Calls)

直接または間接的に自分自身を呼び出す関数。

再帰 (Recursion)

定義した関数またはマクロがそれ自身を呼び出す事。再帰マクロを作成する際は、再帰から抜けずに無限ループとなりやすいため注意が必要。

再入可能 (Reentrant)

1 つの関数を複数呼び出して同時に実行できる事。直接または間接再帰、あるいは割り込み処理中の実行によって起こる事がある。

© 2015 Microchip Technology Inc.

緩和 (Relaxation)

ある命令を、機能が同じでよりサイズの小さい命令に変換する事。コードサイズを抑えるために便利である。最新の MPLAB XC32 には、`CALL` 命令を `RCALL` 命令に緩和する `relax` 機能がある。この変換は、現在の命令から +/-32k 命令ワード以内にあるシンボルを呼び出す場合に行われる。

再配置可能 (Relocatable)

アドレスが固定されたメモリ番地に割り当てられていないオブジェクト。

再配置可能セクション (Relocatable Section)

16 ビットアセンブラ - アドレスが固定されていない (絶対アドレスでない) セクション。再配置可能セクションには、再配置と呼ばれるプロセスによって、リンカがアドレスを割り当てる。

再配置 (Relocation)

リンカが絶対アドレスを再配置可能セクションに割り当てる事。再配置可能セクション内の全てのシンボルを新しいアドレスに更新する。

ROM

Read Only Memory の略。プログラムメモリ。メモリの内容を変更できない。

Run

エミュレータを Halt から解放するコマンド。エミュレータはアプリケーション コードを実行し、I/O に対してリアルタイムに変更、応答を行う。

ランタイムモデル (Run-time Model)

ターゲット アーキテクチャのリソースの使用を記述したもの。

ランタイム ウォッチ (Runtime Watch)

アプリケーション実行中に [Watches] ウィンドウで変数の値がリアルタイムに変化する事。ランタイム ウォッチの設定方法は各ツールの関連文書参照。ランタイム ウォッチをサポートしていないツールもある。

S

シナリオ (Scenario)

MPLAB SIM シミュレータにおいて、スティミュラス制御を具体的に設定したもの。

セクション (Section)

OCG の psect に相当する GCC の用語。リンカが 1 つのまとまりとして処理するコードまたはデータのブロック。

セクション属性 (Section Attribute)

GCC のセクションの特徴を表す情報 (例 : `access` セクション)。

シーケンス ブレークポイント (Sequenced Breakpoints)

シーケンスで発生するブレークポイント。ブレークポイントのシーケンス実行はボトムアップ方式で行われる。つまり、シーケンスの最後のブレークポイントが最初に発生する。

Serialized Quick Turn Programming (SQTP)

デバイス プログラマでマイクロコントローラをプログラムする際に、各デバイスに異なるシリアル番号を書き込めるようにする機能。エントリコード、パスワード、ID 番号等を書き込む目的で使う。

シェル (Shell)

MPASM アセンブラにおいて、マクロアセンブラへの入力を行うためのプロンプト インターフェイス。MPASM アセンブラには DOS 用シェルと Windows 用シェルの 2 種類がある。

シミュレータ (Simulator)

デバイスの動作をモデリングするソフトウェア プログラム。

シングルステップ (Single Step)

コードを 1 命令ずつ実行するコマンド。1 命令を実行するたびに、MPLAB IDE/MPLAB X IDE のレジスタ ウィンドウ、ウォッチ変数、ステータス ディスプレイの表示が更新されるため、命令実行を解析してデバッグできる。C コンパイラのソースコードもシングルステップ実行できるが、その場合は 1 命令ずつ実行されるのではなく、高級言語の C で記述されたコードの 1 行から生成される全てのアセンブリレベル命令がシングルステップで実行される。

スキュー (Skew)

命令実行に対応する情報は、異なる複数のタイミングでプロセッサバスに現れる。例えば、実行されるオペコードは直前の命令の実行時にフェッチとしてバスに表れる。ソースデータのアドレスと値、デスティネーション データのアドレスは、オペコードが実際に実行される時にバスに表れる。デスティネーション データの値は次の命令の実行時にバスに表れる。トレースバッファは、1 インスタンスでバス上に存在する情報をキャプチャする。従って、トレースバッファの 1 エントリには 3 つの命令の実行情報が含まれる。1 つの命令実行で、ある情報から次の情報までにキャプチャされるサイクル数をスキューと呼ぶ。

スキッド (Skid)

ハードウェア ブレークポイントを使ってプロセッサを停止する場合、ブレークポイントからさらに 1 つ以上の命令を実行してプロセッサが停止する事がある。ブレークポイントの後に実行される命令の数をスキッドと呼ぶ。

ソースコード (Source Code)

プログラマ (人間) が記述したコンピュータ プログラム。プログラミング言語で記述されたソースコードは、マシンコードに変換して実行するか、またはインタプリタによって実行される。

ソースファイル (Source File)

ソースコードを記述した ASCII テキストファイル。

特殊機能レジスタ (Special Function Registers: SFR)

I/O プロセッサ機能、I/O ステータス、タイマ等の各種モードや周辺モジュールを制御するレジスタ専用に使うデータメモリ (RAM) 領域。

SQTP

「Serialized Quick Turn Programming」参照。

スタック、ハードウェア (Stack, Hardware)

PIC マイクロコントローラで関数呼び出しを行う時に戻りアドレスを格納する場所。

ソフトウェア スタック (Stack, Software)

アプリケーションが戻りアドレス、関数パラメータ、ローカル変数を保存するのに使うメモリ。このメモリはプログラムでの命令の実行時に動的に割り当てられる。これによって、再入可能な関数の呼び出しが可能になる。

スタック、コンパイルド (Stack, Compiled)

コンパイラが管理し割り当てるメモリの領域で、この領域内で変数に静的に空間を割り当てる。ターゲット デバイス上にソフトウェア スタックのメカニズムを効率的に実装できない場合、ソフトウェア スタックがコンパイルド スタックに置き換わる。このメカニズムでは、関数は再入可能ではなくなる。

MPLAB Starter Kit for *(デバイス名)* (MPLAB Starter Kit for Device)

特定のデバイスでの作業を開始する上で必要となるものを全てセットにした Microchip 社のスタータキット。既製のアプリケーションの動作を確認した後で、一部を変更してカスタム アプリケーションとしてデバッグとプログラムを行う。

© 2015 Microchip Technology Inc.

スタティック RAM (SRAM) (Static RAM、SRAM)

Static Random Access Memory の略。ターゲットボード上の読み書き可能なプログラムメモリ。頻繁に書き換える必要のないプログラムを書き込む。

ステータスバー (Status Bar)

MPLAB IDE/MPLAB X IDE ウィンドウの一番下にあるバーで、カーソル位置、開発モードとデバイス、アクティブなツールバー等に関する情報が表示される。

Step Into

Single Step と同じコマンド。Step Over とは異なり、Step Into では CALL 命令が呼び出すサブルーチン内もステップ実行する。

Step Over

Step Over を実行すると、サブルーチン内はステップ実行せずにデバッグできる。Step Over では、`CALL` 命令があると `CALL` の次の命令にブレークポイントが設定される。何らかの理由により、サブルーチンが無限ループになる等、正しくリターンしない場合、次のブレークポイントには到達しない。`CALL` 命令の処理以外は、Step Over コマンドと Single Step コマンドは同じ。

Step Out

現在ステップ実行中のサブルーチンから抜け出すためのコマンド。このコマンドを実行すると、サブルーチンの残りのコードを全て実行し、サブルーチンの戻りアドレスで実行が停止する。

スティミュラス (Stimulus)

シミュレータへの入力。すなわち、外部信号に対する応答をシミュレートするために生成されるデータ。通常、このデータはテキストファイルにアクションのリストとして記述される。スティミュラスの種類には、非同期、同期 (ピン)、クロック動作、レジスタがある。

ストップウォッチ (Stopwatch)

実行サイクルを測定するためのカウンタ。

記憶域クラス (Storage Class)

指定されたオブジェクトを格納する記憶場所の持続期間を決定する。

記憶域修飾子 (Storage Qualifier)

宣言されるオブジェクトの特別な属性を示す (例 : `const`)。

シンボル (Symbol)

プログラムを構成する各種の要素を記述する汎用のメカニズム。関数名、変数名、セクション名、ファイル名、struct/enum/union タグ名等がある。MPLAB IDE/MPLAB X IDE では、主に変数名、関数名、アセンブリラベルをシンボルと呼ぶ。リンク実行後は、シンボルの値はメモリ内の値となる。

絶対シンボル (Symbol, Absolute)

アセンブリの `.equ` ディレクティブによる定義等、即値を表す。

システム ウィンドウ コントロール (System Window Control)

ウィンドウと一部のダイアログの左上隅にあるコントロール。通常、このコントロールをクリックすると、[最小化]、[最大化]、[閉じる] 等のメニュー項目がポップアップ表示される。

T

ターゲット (Target)

ユーザ ハードウェアの事。

ターゲット アプリケーション (Target Application)

ターゲットボードに読み込んだソフトウェア。

ターゲットボード (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。

ターゲット プロセッサ (Target Processor)

ターゲット アプリケーションの基板で使われているマイクロコントローラ。

テンプレート (Template)

後でファイルに挿入するために作成するテキスト行。MPLAB エディタでは、テンプレートはテンプレート ファイルに保存される。

ツールバー (Tool Bar)

MPLAB IDE/MPLAB X IDE の機能を実行するためのボタン (アイコン) を縦または横に並べたもの。

トレース (Trace)

プログラム実行のログ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のログをトレースバッファに記録し、これを MPLAB IDE/MPLAB X IDE のトレース ウィンドウにアップロードする。

トレースメモリ (Trace Memory)

エミュレータに内蔵されたトレース用のメモリ。トレースバッファとも呼ばれる。

トレースマクロ (Trace Macro)

エミュレータ データからのトレース情報を提供するマクロ。これはソフトウェア トレースのため、トレースを利用するには、マクロをコードに追加し、コードを再コンパイルまたは再アセンブルし､ターゲット デバイスにこのコードをプログラムする必要がある。

トリガ出力 (Trigger Output)

任意のアドレスまたはアドレス範囲で生成でき､トレースとブレークポイントの設定から独立したエミュレータ出力信号の事。トリガ出力ポイントはいくつでも設定できる。

トライグラフ (Trigraphs)

｢??｣で始まる3文字のシーケンス。ISO C で定義されており､1つの文字に置換される。

U

未割り当てセクション (Unassigned Section)

リンカのコマンドファイルで特定のターゲット メモリブロックに割り当てられていないセクション。リンカは､未割り当てセクションを割り当てるターゲット メモリブロックを検出する必要がある。

非初期化データ (Uninitialized Data)

初期値なしで定義されたデータ。C では、

```
int myVar;
```

は、非初期化済みデータセクションに格納される変数を定義する。

アップロード (Upload)

エミュレータやプログラマ等のツールからホスト PC へ、またはターゲットボードからエミュレータへデータを転送する事。

© 2015 Microchip Technology Inc.

USB

Universal Serial Bus の略。2 本のシリアル伝送線で PC と外部周辺機器の通信を行う外部周辺インターフェイス規格。USB 1.0/1.1 でサポートされるデータ転送レートは 12 Mbps。USB 2.0(ハイスピード USB) は最大 480 Mbps のデータ レートをサポートしている。

V

ベクタ (Vector)

リセットまたは割り込みが発生した時にアプリケーションのジャンプ先となるメモリ番地。

Volatile

メモリ内の変数へのアクセス方法に影響を与えるコンパイラの最適化を抑制する変数修飾子。

W

警告 (Warning)

MPLAB IDE/MPLAB X IDE - デバイス、ソフトウェア ファイル、装置に物理的な損傷を与える可能性のある状況で、ユーザに注意を促すために表示されるメッセージ。

16 ビットアセンブラ / コンパイラ - 問題となる可能性のある状態を警告として報告するが、処理は停止されない。MPLAB C30 の警告メッセージではソースファイル名と行番号が報告されるが、エラーメッセージと区別するために「`warning:`」の文字列も付加される。

ウォッチ変数 (Watch Variable)

デバッグ セッション中に [Watch] ウィンドウで観察できる変数。

[Watch] ウィンドウ (Watch Window)

ウォッチ変数の一覧が表示され、ブレークポイントで毎回表示が更新されるウィンドウ。

ウォッチドッグ タイマ (WDT)

PIC マイクロコントローラに内蔵されたタイマの 1 つで、ユーザが設定した期間が経過するとプロセッサをリセットする。WDT の有効化 / 無効化、設定はコンフィグレーション ビットで行う。

ワークブック (Workbook)

MPLAB SIM シミュレータにおいて、SCL スティミュラスの生成に関する設定を保存したもの。

NOTE:

© 2015 Microchip Technology Inc.

索引

記号

.app_excpt セクション 246
.bev_excpt セクション 245
.bss 204
.bss セクション 252
.c 78
.config_address 243
.data 204
.data セクション 250
.dbg_data セクション 250
.dbg_excpt セクション 245
.gld 78
.got セクション 251
.heap 252
.lit4 204
.lit4 セクション 251
.lit8 204
.lit8 セクション 251
.ramfunc セクション 253
.reset セクション 245
.rodata セクション 249
.s 78
.sbss 204
.sbss2 セクション 250
.sbss セクション 252
.sdata 204
.sdata2 セクション 250
.sdata セクション 251
.stack セクション 252
.startup セクション 248
.text セクション 248
.vector_n セクション 246
#define 123
#ident 128
#if 117
#include 123, 124
#line 125
#pragma 114
#pragma config 232
#pragma interrupt 232
#pragma vector 232
プロセッサ演算子 230
プロセッサ演算子 230

数字

0b 2 進基数指定子 148
16 進定数
  C コード 148
2 進定数
  C コード 148
32 ビット C コンパイラマクロ 233
3 文字表記 113
3 文字表記 125

A

a0-a3 197
address 属性 176
alias (symbol) 176
aligned (n) 152, 153
__align 修飾子 36
always_inline 176
-ansi 109, 110, 125, 184
ANSI C 規格
  処理系定義のふるまい 132
  準拠 131
ANSI C 規格 20
ANSI 標準ライブラリのサポート 16
ASCII キャラクタセット 287
ASCII 文字
  拡張 149
asm 220
asm C ステートメント 45
at_vector Attribute 176
auto 変数
  メモリ割り当て 161–162
  初期化 204
auto 変数 158, 161
-aux-info 109, 110

B

-B 93, 127
__bank 修飾子 35
_BEV_EXCPT_ADDR 241, 245
BMXDKPBA 205
_bmxdkpba_address 212
BMXDUDBA 205
_bmxdudba_address 212
BMXDUPBA 205
_bmxdupba_address 212
_bootstrap_exception_handler 213
_bootstrap_exception_handler() 195
_bss_begin 252
_bss_end 252

C

-C 123
-c 108, 126
C++ 方言の制御オプション 110
__C32_VERSION__ 234
calloc 164
case のレンジ 171
Cause 208
Cause レジスタ 207

CCI ... 21
char ... 110, 111, 140, 181
CHAR_BIT ... 140
CHAR_MAX ... 140
CHAR_MIN ... 140
char データ型 ... 26
cleanup (function) ... 153
coherent ... 153
common compiler interface ... 21
Compare ... 207
Compare レジスタ ... 207
Config ... 209
Config1 ... 209
Config1 レジスタ ... 209
Config2 ... 210
Config2 レジスタ ... 210
Config3 ... 210
Config3 レジスタ ... 210
confign ... 243
Config レジスタ ... 209
const ... 176
const オブジェクト
  初期化 ... 151
const オブジェクト
  格納位置 ... 163
const 修飾子 ... 151
Count ... 206, 207
$Count_{DM}$ ... 207
Count レジスタ ... 207
CP0 ... 206
CP0 アクセスマクロ ... 137
CP0 レジスタ ... 206
C スタック
  使い方 ... 161
C スタックの使い方 ... 161
C 方言の制御オプション ... 109
C 方言制御オプション
  -ansi ... 109, 110
  -aux-info ... 109, 110
  -ffreestanding ... 109, 110
  -fno-asm ... 109, 110
  -fno-builtin ... 109, 110
  -fno-signed-bitfields ... 110
  -fno-unsigned-bitfields ... 110
  -fsigned-bitfields ... 110
  -fsigned-char ... 110
  -funsigned-bitfields ... 110
  -funsigned-char ... 110
  -fwritable-strings ... 110
  -traditional ... 184
C 標準ライブラリ ... 102

D

-D ... 123, 124, 125
_data_end ... 251
_DBG_CODE_ADDR ... 241, 245
_DBG_EXCPT_ADDR ... 241, 245
-dD ... 123
Debug ... 210
Debug2 ... 211
Debug2 レジスタ ... 211
debug_exec_mem ... 245
_DEBUGGER ... 245, 250
Debug セクション ... 254
Debug レジスタ ... 210
Debug レジスタ ... 207
--defsym ... 239
--defsym _ebase_address=A ... 209
--defsym _min_heap_size=M ... 201
--defsym _min_stack_size=N ... 201
--defsym, _min_heap_size ... 164
--defsym_min_stack_size ... 161
deprecated 属性 ... 116
deprecated 属性 ... 153, 176
__deprecate 修飾子 ... 41
DeSave ... 211
-dM ... 123
-dN ... 123
double ... 129, 142, 181
DSPr2 エンジン ... 280

E

-E ... 108, 123, 124, 125, 126
EBase ... 209
_ebase_address ... 209, 212
EBase レジスタ ... 209
__eeprom 修飾子 ... 37
$EJTAG_{ver}$ ... 210
_end ... 212, 252
endianism ... 142
ENTRY ... 239
EPC ... 208
EPC レジスタ ... 197, 208, 211
ERET ... 200
exception_mem ... 246
EXL ビット ... 208
EXTERN ... 239
extern ... 116, 122, 184
EXTPDP ... 280
EXTPDPV ... 280

F

-falign-functions ... 119
-falign-labels ... 119
-falign-loops ... 119
far ... 177
-fargument-alias ... 128
-fargument-noalias ... 128
-fargument-noalias-global ... 128
-fcaller-saves ... 119
-fcall-saved ... 128
-fcall-used ... 128
-fcse-follow-jumps ... 119
-fcse-skip-blocks ... 120
-fdata-sections ... 120, 154
-fexceptions ... 108
-fexpensive-optimizations ... 120
-ffixed ... 128, 276
-fforce-mem ... 119, 122
-ffreestanding ... 109, 110
-ffunction-sections ... 120, 179
-fgcse ... 120

© 2015 Microchip Technology Inc.

-fgcse-lm 120
-fgcse-sm 120
file.c 93
file.cpp 93
file.h 93
file.i 93
file.o 93
file.S 93
file.s 93
-finline-functions 116, 119, 122, 184
-finline-limit=n 122
-fkeep-inline-functions 122, 184
-fkeep-static-consts 122
float 129, 142, 181
float.h 142
-fmove-all-movables 120
-fno 122, 128
-fno-asm 109, 110
-fno-builtin 109, 110
-fno-defer-pop 120
-fno-function-cse 122
-fno-ident 128
-fno-inline 123
-fno-keep-static-consts 122
-fno-peephole 120
-fno-peephole2 120
-fno-short-double 129
-fno-show-column 123
-fno-signed-bitfields 110
-fno-unsigned-bitfields 110
-fno-verbose-asm 129
-fomit-frame-pointer 118, 119, 123
-foptimize-register-move 120
-foptimize-sibling-calls 123
format (type, format_index, first_to_check) 177
format_arg (index) 177
fp 197
-fpack-struct 128
-fpcc-struct-return 128
-freduce-all-givs 120
-fregmove 120
-frename-registers 120
-frerun-cse-after-loop 120, 121
-frerun-loop-opt 120
-fschedule-insns 121
-fschedule-insns2 121
-fshort-enums 129, 269
-fsigned-bitfields 110
-fsigned-char 110
-fstrength-reduce 121
-fstrict-aliasing 119, 121
-fsyntax-only 111
-fthread-jumps 118, 121
-funroll-all-loops 119, 121
-funroll-loops 119, 121
-funsigned-bitfields 110, 268
-funsigned-char 110, 140
-fverbose-asm 129
-fvolatile 129
-fvolatile-global 129
-fvolatile-static 129
-fwritable-strings 110
F 定数添え字 149

G

-g 118
-G num 106
_general_exception_context() 195
_general_exception_handler 213
_GEN_EXCPT_ADDR 241, 246
GLOBAL ディレクティブ 59
-Gn 204
_gp 203, 212, 251
gp 197, 203

H

-H 123
_heap 201, 212
--help 108
HEX ファイル
  マージ 61
HEX ファイル 96
hi 197
HWREna 206

I

-I 124, 125
-I- 123, 125
-idirafter 124
-imacros 124, 125
-include 124, 125
INPUT 240
int 140, 181
IntCtl 207
interrupt 177
__interrupt 修飾子 38
INT_MAX 141
INT_MIN 141
-iquote 270
__ISR(v, ipl) 190
__ISR_AT_VECTOR(v, ipl) 191
-isystem 127

K

k0 197
k1 197
keep 属性 177
KSEG0/KSEG1 データメモリ 201
kseg0_program_mem 248, 249, 250, 251
kseg1_boot_mem 245
kseg1_data_mem 250, 251, 252

L

-L 126, 127, 240
-l 126
__LANGUAGE_ASSEMBLY 233
__LANGUAGE_ASSEMBLY__ 233
_LANGUAGE_ASSEMBLY 233
LANGUAGE_ASSEMBLY 233
__LANGUAGE_C 233
__LANGUAGE_C__ 233

_LANGUAGE_C 233
LANGUAGE_C 233
LED の点滅.. 73
limits.h 140
 CHAR_BIT 140
 CHAR_MAX 140
 CHAR_MIN 140
 INT_MAX 141
 INT_MIN 141
 LLONG_MAX 141
 LLONG_MIN 141
 LONG_MAX 141
 LONG_MIN 141
 MB_LEN_MAX 140
 SCHAR_MAX 140
 SCHAR_MIN 140
 SHRT_MAX 140
 SHRT_MIN 140
 UCHAR_MAX 140
 UINT_MAX 141
 ULLONG_MAX 141
 ULONG_MAX 141
 USHRT_MAX 140
link 272
little endian format 142
LLONG_MAX 141
LLONG_MIN 141
lo 197
long 140, 181
Long double 181
long double 129, 142
long long 116, 140, 181
longcall 177
longcall 属性 180
LONG_MAX 141
LONG_MIN 141
__longramfunc__ 180
L 定数添え字 148

M

-M 124
main 199
main 関数.. 23, 199
malloc 164, 177
-mappio-debug 106
MB_LEN_MAX 140
-mcci 106
-mcheck-zero-division 106
MCHP 233
_mchp_no_float 233
_MCHP_SZINT 233
_MCHP_SZLONG 233
_MCHP_SZPTR 233
-MD 124
-mdebugger 245, 250
-membedded-data 106
-MF 124
-mframe-header-opt 106
-MG 124
Microchip Technology 社への問い合わせ............... 292
Microchip 社のインターネット アドレ.................. 292
Microchip 社のウェブサイト 292
micromips 177
_min_heap_size 239
_min_stack_size 201, 239
-minterlink-compressed 107
MIPS 235
__mips 235
__mips__ 235
_mips 235
__mips16 235
-mips16 .... 107, 126, 178
mips16 177
_MIPS_ARCH_PIC32MX 235
__MIPSEL 235
__MIPSEL__ 235
_MIPSEL 235
MIPSEL 235
_mips_fpr 235
_MIPS_ISA 235
__mips_isa_rev 235
_mips_no_float 235
__mips_soft_float 235
_MIPS_SZINT 235
_MIPS_SZLONG 235
_MIPS_SZPTR 235
_MIPS_TUNE_PIC32MX 235
-mjals 107
-mlong-calls 107, 178
-MM 124
-MMD 124
-mmemcpy 107
-mmicromips 107, 126
-mno-check-zero-division 106
-mno-embedded-data 106
-mno-float 107
-mno-jals 107
-mno-long-calls 107
-mno-memcpy 107
-mno-micromips 107
-mno-mips16 107
-mno-peripheral-libs 107
-mno-uninit-const-in-rodata 107
-MP 125
MPLAB IDE
 コンパイラ動作モード 50
MPLAB IDE Build Options ダイアログ
 MPLAB ASM30 タブ.. 79
 MPLAB LINK30 タブ 83
MPLAB X IDE 75
 プロジェクト プロパティ オプション 51
-mprocessor 107, 240, 280
-MQ 125
-MT 125
MTC0 命令 208
MTHLIP 280
-muninit-const-in-rodata 107
myMicrochip 変更通知サービス.............................. 291

N

naked 177
near 178

 © 2015 Microchip Technology Inc.

__near 修飾子 33
_nmi_handler 200
-nodefaultlibs 126
__NO_FLOAT 233
noinline 178
NOLOAD 245, 250
nomips16 .... 178, 200, 213
nonnull (index, ...) 178
NOP 248
noreturn 178
noreturn 属性 116
-nostdinc 123, 125
-nostdlib 126
NULL ポインタ 147
NULL マクロ 30

O

-O 118
-o 96, 108
-o ex1.out 96
-O0 118
-O1 118
-O2 .... 119, 122
-O3 119
On Bootstrap プロシージャ 211
_on_reset 214
-Os 119
OUTPUT_ARCH 239
OUTPUT_FORMAT 239

P

-P 125
packed 154
__pack 修飾子 40
-pedantic 111, 116
-pedantic-errors 111
persistent 153
persistent 修飾子 204
persistent 修飾子 214
__persisten 修飾子 34
__PIC__ 233
__pic__ 233
pic30-ar 75
pic30-as 75
pic30-ld 75
PIC32MX デバイス専用オプション
    -msmart-io= 107
PIC32_C_INCLUDE_PATH 92
PIC32_C_INCLUDE_PATH 92
PIC32_COMPILER_PATH 92
PIC32_EXEC_PREFIX 93
__PIC32_FEATURE_SET__ 233
PIC32_LIBRARY_ PATH 93
__PIC32MX 233
__PIC32MX__ 233
PIC32MX 233
PIC32MX スタートアップ コード 200
PIC32MX デバイス専用オプション
    -Gnum 106
    -mappio-debug 106
    -mcci 106
    -mcheck-zero-division 106
    -membedded-data 106
    -mframe-header-opt 106
    -minterlink-compressed 107
    -mips16 107
    -mjals 107
    -mlong-calls 107
    -mmemcpy 107
    -mmicromips 107
    -mno-check-zero-division 106
    -mno-embedded-data 106
    -mno-float 107
    -mno-jals 107
    -mno-long-calls 107
    -mno-memcpy 107
    -mno-micromips 107
    -mno-mips16 107
    -mno-peripheral-libs 107
    -mno-uninit-const-in-rodata 107
    -mprocessor 107
    -muninit-const-in-rodata 107
POS 280
PRId 209
printf 関数 63
__processor__ 234
processor.o 240
Project Properties ダイアログ 79
PROVIDE 239
Provisions 103
pure 178

Q

-Q 118
Q15 小数データ 280
Q31 小数データ 280
Q7 小数データ 280

R

__R3000 235
__R3000__ 235
_R3000 235
R3000 235
ra 197
__ramfunc__ 180
ramfunc 178
_ramfunc_begin 212
_ramfunc_length 212
RAW 依存性 121
RDHWR 206
realloc 164
register 275
_reset 239
_RESET_ADDR 241, 245
rx 242

S

-S 108, 126
-s 126
s0-s7 197
-save-temps 118
sbrk 201

_sbss_begin 252
_sbss_end 252
SCHAR_MAX 140
SCHAR_MIN 140
Scheduling 121
SCOUNT 280
_sdata_begin 251
_sdata_end 251
SDE 互換マクロ
  __mips__ 235
  __mips 235
  _mips 235
  __mips16 235
  _MIPS_ARCH_PIC32MX 235
  __MIPSEL__ 235
  __MIPSEL 235
  _MIPSEL 235
  MIPSEL 235
  _mips_fpr 235
  _MIPS_ISA 235
  __mips_isa_rev 235
  _mips_no_float 235
  __mips_soft_float 235
  _MIPS_SZINT 235
  _MIPS_SZLONG 235
  _MIPS_SZPTR 235
  _MIPS_TUNE_PIC32MX 235
  __R3000__ 235
  __R3000 235
  _R3000 235
  R3000 235
section (“name”) 154
section (name) 179
SECTIONS コマンド 243, 244
__section 修飾子 42
SFR 136
SFR メモリ領域
  sfrs 242
short 140, 181
SHRT_MAX 140
SHRT_MIN 140
signed char 140
signed int 140
signed long 140
signed long long 140
signed short 140
SI_TimerInt 207
sp 197, 201
-specs= 127
SR 197
SRSCtl 208
SRSMap 208
_stack 201, 212, 253
static 129
static 変数 159
static 変数 204
static 関数 175
Status 207
Status Register 207
Status$_{BEV}$ 209, 213
Structure 181
switch 113

T

t0-t9 197
TMPDIR 93
TraceBPC 210
TraceBPC レジスタ 210
TraceControl 210
TraceControl2 210
-traditional 110, 184
-trigraphs 125

U

-U 123, 124, 125
-u 127
UCHAR_MAX 140
UINT_MAX 141
ULLONG_MAX 141
ULONG_MAX 141
-undef 125
unique_section 179
unlink 272
Unroll Loop 82, 121
unsigned char 140
unsigned int 140
unsigned long 140
unsigned long long 140
unsigned short 140
unused 属性 154, 179
USB 311
used 属性 179
UserTraceData 210
USHRT_MAX 140
U 定数添え字 148

V

-v 108
v0 197
v1 197
v2i16 280
v2q15 280
v4i8 280
v4q7 280
vector (num) 179
vector_size 280
_vector_spacing 207, 212, 241
__VERSION__ 234
volatile 129
volatile 修飾子 61, 63, 151

W

-W 111, 114, 115, 117
-w 111
w!x 242
-Wa 125
-Waggregate-return 115
-Wall 111, 114, 115, 117
warn_unused_result 179
-Wbad-function-cast 115
-Wcast-align 116

 © 2015 Microchip Technology Inc.

-Wcast-qual ... 116
-Wchar-subscripts ... 111
-Wcomment ... 111
-Wconversion ... 116
-Wdiv-by-zero ... 111
weak ... 154, 179
-Werror ... 116
-Wformat ... 111, 116
-Wimplicit ... 112
-Wimplicit-function-declaration ... 112
-Wimplicit-int ... 112
-Winline ... 116, 184
-Wl ... 127
-Wlarger-than- ... 116
-Wlong-long ... 116
-Wmain ... 112
-Wmissing-braces ... 112
-Wmissing-declarations ... 116
-Wmissing-format-attribute ... 116
-Wmissing-noreturn ... 116
-Wmissing-prototypes ... 116
-Wmultichar ... 112
-Wnested-externs ... 116
-Wno- ... 111
-Wno-deprecated-declarations ... 116
-Wno-div-by-zero ... 111
-Wno-long-long ... 116
-Wno-multichar ... 112
-Wnonnull ... 178
-Wno-sign-compare ... 115, 117
-Wpadded ... 116
-Wparentheses ... 112
-Wpointer-arith ... 117
WRDSP ... 280
-Wredundant-decls ... 117
-Wreturn-type ... 112
-Wsequence-point ... 113
-Wshadow ... 117
-Wsign-compare ... 117
-Wstrict-prototypes ... 117
-Wswitch ... 113
-Wsystem-headers ... 113
-Wtraditional ... 117
-Wtrigraphs ... 113
-Wundef ... 117
-Wuninitialized ... 114
-Wunknown-pragmas ... 113, 114
-Wunreachable-code ... 117
-Wunused ... 114, 115
-Wunused-function ... 114
-Wunused-label ... 114
-Wunused-parameter ... 114
-Wunused-value ... 114
-Wunused-variable ... 114
-Wwrite-strings ... 117

X

-x ... 108
xc.h ヘッダファイル ... 133
XC32_C_INCLUDE_PATH ... 92
XC32_COMPILER_PATH ... 92
XC32_EXEC_PREFIX ... 93
xc32-gcc ... 91
XC32_LIBRARY_ PATH ... 93
__xdata 修飾子 ... 35
-Xlinker ... 127

Y

__ydata 修飾子 ... 35

あ

アセンブリ オプション
  -Wa ... 125
アセンブリ オプション ... 125
アセンブリコード
  C 言語と併用 ... 59, 217
  書き方 ... 59–60
アセンブリ言語
  よくあるエラー ... 60
  レジスタ ... 60
アセンブリ リストファイル ... 104

い

一時変数 ... 161
一般的例外 ... 195
インクリメンタル ビルド ... 97
インクルード ファイル ... 127
インライン展開 ... 123, 184

う

ウォッチドッグ タイマ ... 73, 311

え

エラー制御オプション
  -pedantic-errors ... 111
  -Werror ... 116
エラー例外プログラム カウンタ ... 211
エラー メッセージ
  位置 ... 72

お

オブジェクト ファイル ... 81
オブジェクト ファイル ... 120
オブジェクト ファイル ... 124, 126
オプション
  C 言語の方言の制御 ... 109, 110
  アセンブリ ... 125
  コード生成規則 ... 128
  ディレクトリ検索 ... 127
  デバッグ ... 118
  プリプロセッサの制御 ... 123
  リンク ... 126
  最適化の制御 ... 118
  警告とエラーの制御 ... 111
オペランドの省略 ... 171
オペランドを省略した条件式 ... 171

か

外部割り込みコントローラ ... 208
カウントレジスタ ... 207

拡張キャラクタセット 149
拡張子 124
仮数部 142
カスタマサポート 292
型変換 116, 167
環境変数
PIC32_C_INCLUDE_PATH 92
PIC32_C_INCLUDE_PATH 92
PIC32_COMPILER_PATH 92
PIC32_EXEC_PREFIX 93
PIC32_LIBRARY_ PATH 93
TMPDIR 93
XC32_C_INCLUDE_PATH 92
XC32_COMPILER_PATH 92
XC32_EXEC_PREFIX 93
XC32_LIBRARY_ PATH 93
関数
static 175
アセンブリ言語 217
サイズ 57, 69
パラメータ 161, 181
ポインタ 147
位置 69
指定子 175
関数属性
address 176
alias (symbol) 176
always_inline 176
at_vector 176
const 176
deprecated 176
far 177
format (type, format_index, first_to_check) 177
format_arg (index) 177
interrupt 177
keep 177
longcall 177
malloc 177
micromips 177
mips16 177
naked 177
near 178
noinline 178
nomips16 178
nonnull (index, ...) 178
noreturn 116, 178
pure 178
ramfunc 178
section (name) 179
unique_section 179
unused 179
used 179
vector (num) 179
warn_unused_result 179
weak 179

き

基数指定子
C コード 148
起動と初期化 103
起動と初期化
C 103
C++ 103
キャスト 52, 114, 115, 116, 167
キャスト属性 114
キャスト演算子 52
共通部分式 122
共通部分式除去 119, 120, 121
共用体
修飾子 144
無名 145

く

グローバル レジスタ変数 275
グローバル レジスタ変数の定義 275

け

警告とエラーの制御オプション 111
警告とエラーの制御オプション
-fsyntax-only 111
-pedantic 111
-pedantic-errors 111
-w 111, 115
-Waggregate-return 115
-Wall 111
-Wbad-function-cast 115
-Wcast-align 116
-Wcast-qual 116
-Wchar-subscripts 111
-Wcomment 111
-Wconversion 116
-Wdiv-by-zero 111
-Werror 116
-Wformat 111
-Wimplicit 112
-Wimplicit-function-declaration.... 112
-Wimplicit-int 112
-Winline 116
-Wlarger-than- 116
-Wlong-long 116
-Wmain 112
-Wmissing-braces 112
-Wmissing-declarations 116
-Wmissing-format-attribute 116
-Wmissing-noreturn 116
-Wmissing-prototypes 116
-Wmultichar 112
-Wnested-externs 116
-Wno-long-long 116
-Wno-multichar 112
-Wno-sign-compare 117
-Wpadded 116
-Wparentheses 112
-Wpointer-arith 117
-Wredundant-decls 117
-Wreturn-type 112
-Wsequence-point 113
-Wshadow 117
-Wsign-compare 117
-Wstrict-prototypes 117

© 2015 Microchip Technology Inc.

-Wswitch ... 113
-Wsystem-headers ... 113
-Wtraditional ... 117
-Wtrigraphs ... 113
-Wundef ... 117
-Wuninitialized ... 114
-Wunknown-pragmas ... 114
-Wunreachable-code ... 117
-Wunused ... 114
-Wunused-function ... 114
-Wunused-label ... 114
-Wunused-parameter ... 114
-Wunused-value ... 114
-Wunused-variable ... 114
-Wwrite-strings ... 117
警告の抑止 ... 111
警告メッセージ ... 105
警告メッセージ
不要 ... 73
位置 ... 72
原因レジスタ ... 208
厳密な ANSI C 規格 ... 111

こ

コードサイズの削減 ... 118, 119
コード生成規則オプション ... 128
コード生成規則オプション
-fargument-alias ... 128
-fargument-noalias ... 128
-fargument-noalias-global ... 128
-fcall-saved ... 128
-fcall-used ... 128
-ffixed ... 128
-fno-ident ... 128
-fno-short-double ... 129
-fno-verbose-asm ... 129
-fpack-struct ... 128
-fpcc-struct-return ... 128
-fshort-enums ... 129
-fverbose-asm ... 129
-fvolatile ... 129
-fvolatile-global ... 129
-fvolatile-static ... 129
コードの保護 ... 63
構造体 ... 144
構造体
ビットフィールド ... 56, 144
構造体修飾子 ... 144
構造体ビットフィールド ... 144
構文チェック ... 111
高優先度割り込み ... 187
コマンドライン オプション、コンパイラ
-fdate-sections ... 154
-ffunction-sections ... 179
-fshort-enums ... 269
-funsigned-bitfields ...] 268
-funsigned-char ... 140
-I ... 126
-iquote ... 270
-mdebugger ... 245, 250
-mips16 ... 178
-mlong-calls ... 178
-mprocessor ... 240
-o ex1.out ... 96
-Wall ... 114
-Wnonnull ... 178
コマンドライン オプション、リンカ
--defsym ... 239
--defsym_min_stack_sizeI ... 161
-L ... 240
コマンドライン シミュレータ ... 16
コメント ... 111, 123
コンテキスト スイッチコード ... 58, 68
コンパイラ
ドライバ ... 91, 127
コンパイラのインストールと有効化 ...] 47
コンパイラの動作モード ... 15
コンパイラの動作モード ... 65
コンパイラの選択 ... 50
コンパイラ生成コード ... 69
コンフィグレーション プラグマ ... 134, 135
コンフィグレーション メモリ領域
config3, config2, config1, config0 ... 242
コンフィグレーション ワード ... 134, 135
コンペアレジスタ ... 207

さ

最適化
コードサイズ ... 66
コードの高速化 ... 67
変数の破壊 ... 63
ループ ... 120
割り込み関数 ... 68
最適化オプション ... 118
最適化オプション
-fgcse ... 120
-fgcse-lm ... 120
-fgcse-sm ... 120
-O ... 118
-O0 ... 118
-O1 ... 118
-O2 ... 119
-O3 ... 119
-Os ... 119
最適化制御オプション
-falign-functions ... 119
-falign-labels ... 119
-falign-loops[... 119
-fcaller-saves ... 119
-fcse-follow-jumps ... 119
-fcse-skip-blocks ... 120
-fdata-sections ... 120
-fexpensive-optimizations ... 120
-fforce-mem ... 122
-ffunction-sections ... 120
-finline-functions ... 122
-finline-limit=n ... 122
-fkeep-inline-functions ... 122
-fkeep-static-consts ... 122
-fmove-all-movables ... 120

-fno-defer-pop ... 120
-fno-function-cse ... 122
-fno-inline ... 123
-fno-peephole ... 120
-fno-peephole2 ... 120
-fomit-frame-pointer ... 123
-foptimize-register-move ... 120
-foptimize-sibling-calls ... 123
-freduce-all-givs ... 120
-fregmove ... 120
-frename-registers ... 120
-frerun-loop-opt ... 120
-fschedule-insns ... 121
-fschedule-insns2 ... 121
-fstrength-reduce ... 121
-fstrict-aliasing ... 121
-fthread-jumps ... 121
-funroll-all-loops ... 121
-funroll-loops ... 121
最適化制御レジスタ
-frerun-cse-after-loop ... 120
サイズの制限 ... 68
サポートするデバイス ... 133

し
識別子
有意文字数 ... 25, 56
指数部 ... 142
システム関数
link ... 272
unlink ... 272
システム ヘッダファイル ... 113, 124
自動変数 ... 114, 115
シミュレータ
コマンドライン ... 16
シャドーレジスタ マップレジスタ ... 208
シャドーレジスタ制御レジスタ ... 208
修飾子 ... 151–152
修飾子
__align ... 36
auto ... 161
auto 変数 ... 161
__bank ... 35
const ... 151
__deprecate ... 41
__eeprom ... 37
__interrupt ... 38
__near ... 33
__pack ... 40
__persisten ... 34
persistent ... 204, 214
__section ... 42
volatile ... 61, 63, 151
__xdata ... 35
__ydata ... 35
構造体 ... 144
出力
出力制御 ... 108
出力制御オプション ... 108
出力制御オプション
-c ... 108
-E ... 108
-fexceptions ... 108
--help ... 108
-o ... 108
-S ... 108
-v ... 108
-x ... 108
出力ファイル
ファイル名 ... 104
条件式 ... 171
定数
C 指定子 ... 148
文字 ... 149
文字列 ... 149
初期化変数 ... 204
処理系定義のふるまい ... 132
診断ファイル ... 104
シンボル ... 127

す
推奨参考資料 ... 12
スタートアップ コード
変数の初期化 ... 204
スタックの位置 ... 253
スタックポインタ ... 201
スタックポインタ (W15) ... 128, 134

せ
整数値
char ... 140
int ... 140
long long ... 140
long ... 140
short ... 140
signed char ... 140
signed int ... 140
signed long long ... 140
signed long ... 140
signed short ... 140
unsigned char ... 140
unsigned int ... 140
unsigned long long ... 140
unsigned long ... 140
unsigned short ... 140
整数定数 ... 148
整数の添え字 ... 148
セクション ... 81, 120
セクション
コンフィグレーション ワード ... 134
接頭辞 ... 127

そ
ソースコード ... 78

ち
遅延ルーチン ... 64
致命的エラー ... 105

© 2015 Microchip Technology Inc.

て

データ型
- サイズ 25, 143
- 浮動小数点 143

データ型のサイズ 143
データメモリ 158
データメモリ空間 164
データメモリ領域
- kseg1_data_mem 242

定義済みマクロ 152
低優先度割り込み 187
ディレクトリ 124, 125
ディレクトリ検索オプション 127
ディレクトリ検索オプション
- -B 93, 127
- -specs= 127

デバッガ 62
デバッグ エグゼクティブ メモリ領域
- debug_exec_mem 242

デバッグ オプション 118
デバッグオプション
- -g 118
- -Q 118
- -save-temps 118

デバッグ情報 118
デバッグ分岐遅延 211
デバッグ例外プログラム カウンタ 211
デバッグ例外保存レジスタ 211
電源投入ルーチン 214
伝統的 C 117

と

特殊機能レジスタ 96
ドライバ
- 入力ファイル 92

ドライバ オプション 51, 92, 106–128
ドライバ オプション
- CCI 46
- EXT 257

トレース制御レジスタ 210

に

入力ファイル 92

は

ハードウェア イネーブル レジスタ 206
バイアスした指数部 142
配列 159
配列
- 初期化 149
- ダミーポインタ ターゲット 147

バスマトリクス レジスタ 205
バスマトリクス レジスタ
- BMXDKPBA 205
- BMXDUDBA 205
- BMXDUPBA 205

ひ

ヒープ 201
ピープホール最適化 120
非初期化変数 204
ビット単位の補数演算子 168
ビットフィールド 28, 29, 56, 110, 144–145
標準 I/O 関数 16

ふ

ブートストラップ例外 195
ブートメモリ領域 242
- kseg1_boot_mem 242

ブートローダ 61
ファイル拡張子 93
ファイル拡張子
- file.c 93
- file.cpp ....]93
- file.h 93
- file.i 93
- file.ii 93
- file.o 93
- file.s 93
- file.S 93

ファイルタイプ
- 入力 92

不揮発性 RAM 151
符号ビット 142
不正仮想アドレスレジスタ 206
浮動小数点型 143
浮動小数点型
- バイアスした指数部 142
- 丸め処理 143
- 指数部 142

浮動小数点型の丸め処理 56
浮動小数点定数の添え字 149
浮動小数点フォーマット
- double 142
- float 142
- long double 142

フラグ
- 肯定形と否定形 122, 128

プラグマ
- #pragma config 134, 135, 232
- #pragma interrupt 232
- #pragma vector 232

プラグマ ディレクティブ 32
フラッシュメモリ 163
プリプロセッサ
- 型の用法 231

プリプロセッサ制御オプション 123
プリプロセッサ制御オプション
- -C 123
- -D 123
- -dD 123
- -dM 123
- -dN 123
- -fno-show-column 123
- -H 123
- -I- 123
- -I 124
- -idirafter 124
- -imacros 124
- -include 124

-M 124
-MD 124
-MF 124
-MG 124
-MM 124
-MMD 124
-MQ 125
-MT 125
-nostdinc 125
-P 125
-trigraphs 125
-U 125
-undef 125
プリプロセッサ ディレクティブ 230–231
プリプロセッサ マクロ
定義済み 44
フレームポインタ (W14) 82, 84, 123
フレームポインタ (W14) 128
プログラムメモリ 163
プログラムメモリ領域
kseg0_program_mem 242
プロジェクト 77
プロジェクト名 104
プロセッサ識別レジスタ 209
分岐遅延 208

へ

ベクタプラグマ 194
ヘッダファイル 23, 92, 93, 123, 124, 125, 215
ヘッダファイル
デバイス 133, 136
検索パス 24
変数
auto 161
static 159
位置 69
初期化 204
サイズ 143
最大サイズ 68
プログラムメモリ内 55, 163
保持期間 158
レジスタ内 164
変数のクリア 204
変数の属性
aligned (n) 152, 153
cleanup (function) 153
coherent 153
deprecated 153
packed 154
persistent 153
section (“name”) 154
unused 154
weak 154
変数のビットアクセス 56

ほ

ポートのグリッチ 61
ポインタ
関数 147
修飾子 146
スタック 128
整数割り当て 147
タイプ 146
ダミーターゲットの割り当て 147
定義 146
比較 147
フレーム 82, 84, 123, 128
ポインタ 117, 146–147
保持期間 158
本書について
表記規則 11
本書の構成
レイアウト 10

ま

前処理 229
マクロ 123, 125, 185
マクロ
__C32_VERSION_ 234
__LANGUAGE_ASSEMBLY__ 233
__LANGUAGE_ASSEMBLY 233
_LANGUAGE_ASSEMBLY 233
LANGUAGE_ASSEMBLY 233
__LANGUAGE_C__ 233
__LANGUAGE_C 233
_LANGUAGE_C 233
LANGUAGE_C 233
_mchp_no_float 233
_MCHP_SZINT 233
_MCHP_SZLONG 233
_MCHP_SZPTR 233
__NO_FLOAT 233
__PIC__ 233
__pic__ 233
__PIC32_FEATURE_SET__ 233
__PIC32MX__ 233
__PIC32MX 233
PIC32MX 233
__processor__ 234
__VERSION__ 234
マップファイル 104

み

未使用関数パラメータ 114
未使用変数 114
未使用変数
削除 151
未使用メモリ 70
未使用メモリの充填 63

む

無名共用体 145
無名の構造体メンバー 145
無名のビットフィールド 145

め

メインライン コード 187
メインルーチンを呼び出す 212
メッセージ
タイプ 105

© 2015 Microchip Technology Inc.

致命的エラー 105
メモリ
概要.. 70
残量.. 70
メモリモデル.. 165
メモリ割り当て 157
メモリ割り当て
static 変数.. 159
関数コード.. 180
データメモリ 158
非 auto 変数... 158
プログラムメモリ 163

も
文字定数
C 言語 149
文字列 110
文字列リテラル 149
文字列リテラル
型 149
保存位置 149
連結... 150
戻り値の型 112

ゆ
ユーザトレース データレジスタ 210

よ
要求された割り込み優先度.................................. 208
読み出し専用変数.. 151

ら
ライブラリ 101, 126
ライブラリ
ANSI 標準 16
検索順 92
ユーザ定義.. 102

り
リセット
コード実行後 214
リトル エンディアン 139
リンカ 126
リンカスクリプト 78, 237
リンクオプション 126
リンク オプション
-L 126, 127
-l 126
-nodefaultlibs 126
-nostdlib 126
-s 126
-u 127
-Wl 127
-Xlinkerl 127

る
ループ最適化 120
ループ展開 82, 84, 121

れ
例外プログラム カウンタ...................................... 208
例外ベースレジスタ 209
例外ベクタ 193
例外メモリ領域
exception_mem... 242
レジスタ 276
レジスタ
関数が使用 69
変数の割り当て 164
レジスタの用法 173

ろ
ローカル変数向けにレジスタを指定する 276
ローカルレジスタ変数............................... 275, 276
ローテート演算子 64

わ
割り込み 58
割り込み
コンテキスト スイッチ 58, 68
高優先度... 187
低優先度... 187
無効化 63
割り込み関数
コンテクスト スイッチ.................................... 197
最適化 68
割り込み制御レジスタ 207
割り込み属性... 188
割り込みの無効化 63
割り込みハンドラ関数.. 188
割り込みプラグマ... 190
割り込みプラグマ節... 193

MICROCHIP

各国の営業所とサービス

北米

本社
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel:480-792-7200
Fax:480-792-7277
技術サポート :
http://www.microchip.com/support
URL:
www.microchip.com

アトランタ
Duluth, GA
Tel:678-957-9614
Fax:678-957-1455

オースティン (TX)
Tel:512-257-3370

ボストン
Westborough, MA
Tel:774-760-0087
Fax:774-760-0088

シカゴ
Itasca, IL
Tel:630-285-0071
Fax:630-285-0075

クリーブランド
Independence, OH
Tel:216-447-0464
Fax:216-447-0643

ダラス
Addison, TX
Tel:972-818-7423
Fax:972-818-2924

デトロイト
Novi, MI
Tel:248-848-4000

ヒューストン (TX)
Tel:281-894-5983

インディアナポリス
Noblesville, IN
Tel:317-773-8323
Fax:317-773-5453

ロサンゼルス
Mission Viejo, CA
Tel:949-462-9523
Fax:949-462-9608

ニューヨーク (NY)
Tel:631-435-6000

サンノゼ (CA)
Tel:408-735-9110

カナダ - トロント
Tel:905-673-0699
Fax:905-673-6509

アジア / 太平洋

アジア太平洋支社
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel:852-2943-5100
Fax:852-2401-3431

オーストラリア - シドニー
Tel:61-2-9868-6733
Fax:61-2-9868-6755

中国 - 北京
Tel:86-10-8569-7000
Fax:86-10-8528-2104

中国 - 成都
Tel:86-28-8665-5511
Fax:86-28-8665-7889

中国 - 重慶
Tel:86-23-8980-9588
Fax:86-23-8980-9500

中国 - 東莞
Tel:86-769-8702-9880

中国 - 杭州
Tel:86-571-8792-8115
Fax:86-571-8792-8116

中国 - 香港 SAR
Tel:852-2943-5100
Fax:852-2401-3431

中国 - 南京
Tel:86-25-8473-2460
Fax:86-25-8473-2470

中国 - 青島
Tel:86-532-8502-7355
Fax:86-532-8502-7205

中国 - 上海
Tel:86-21-5407-5533
Fax:86-21-5407-5066

中国 - 瀋陽
Tel:86-24-2334-2829
Fax:86-24-2334-2393

中国 - 深圳
Tel:86-755-8864-2200
Fax:86-755-8203-1760

中国 - 武漢
Tel:86-27-5980-5300
Fax:86-27-5980-5118

中国 - 西安
Tel:86-29-8833-7252
Fax:86-29-8833-7256

アジア / 太平洋

中国 - 厦門
Tel:86-592-2388138
Fax:86-592-2388130

中国 - 珠海
Tel:86-756-3210040
Fax:86-756-3210049

インド - バンガロール
Tel:91-80-3090-4444
Fax:91-80-3090-4123

インド - ニューデリー
Tel:91-11-4160-8631
Fax:91-11-4160-8632

インド - プネ
Tel:91-20-3019-1500

日本 - 大阪
Tel:81-6-6152-7160
Fax:81-6-6152-9310

日本 - 東京
Tel:81-3-6880- 3770
Fax:81-3-6880-3771

韓国 - 大邱
Tel:82-53-744-4301
Fax:82-53-744-4302

韓国 - ソウル
Tel:82-2-554-7200
Fax:82-2-558-5932 または
82-2-558-5934

マレーシア - クアラルンプール
Tel:60-3-6201-9857
Fax:60-3-6201-9859

マレーシア - ペナン
Tel:60-4-227-8870
Fax:60-4-227-4068

フィリピン - マニラ
Tel:63-2-634-9065
Fax:63-2-634-9069

シンガポール
Tel:65-6334-8870
Fax:65-6334-8850

台湾 - 新竹
Tel:886-3-5778-366
Fax:886-3-5770-955

台湾 - 高雄
Tel:886-7-213-7828

台湾 - 台北
Tel:886-2-2508-8600
Fax:886-2-2508-0102

タイ - バンコク
Tel:66-2-694-1351
Fax:66-2-694-1350

ヨーロッパ

オーストリア - ヴェルス
Tel:43-7242-2244-39
Fax:43-7242-2244-393

デンマーク - コペンハーゲン
Tel:45-4450-2828
Fax:45-4485-2829

フランス - パリ
Tel:33-1-69-53-63-20
Fax:33-1-69-30-90-79

ドイツ - デュッセルドルフ
Tel:49-2129-3766400

ドイツ - ミュンヘン
Tel:49-89-627-144-0
Fax:49-89-627-144-44

ドイツ - プフォルツハイム
Tel:49-7231-424750

イタリア - ミラノ
Tel:39-0331-742611
Fax:39-0331-466781

イタリア - ベニス
Tel:39-049-7625286

オランダ - ドリューネン
Tel:31-416-690399
Fax:31-416-690340

ポーランド - ワルシャワ
Tel:48-22-3325737

スペイン - マドリッド
Tel:34-91-708-08-90
Fax:34-91-708-08-91

スウェーデン - ストックホルム
Tel:46-8-5090-4654

イギリス - ウォーキンガム
Tel:44-118-921-5800
Fax:44-118-921-5820

01/27/15

© 2015 Microchip Technology Inc.