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 アセンブラ、リンカ、ユーティリティ ユーザガイド

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

商標

ISBN: 978-1-5224-0615-0

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。

MPLAB® XC32ア センブラ、リンカ、ユーティリティ ユーザガイド

目次

序章

NOTICE TO CUSTOMERS

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。お客様のニーズを満たすため、Microchip 社の製品は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内容とは異なる場合があります。最新の文書は弊社ウェブサイト (www.microchip.com) でご覧になれます。

文書は「DS」番号によって識別します。この識別番号は各ページのフッタのページ番号の前に表記しています。DS 番号「DSXXXXXXXXA」の「XXXXXXXX」は文書番号、「A」はリビジョン レベルを表します。

開発ツールの最新情報は MPLAB® X IDE のオンラインヘルプでご覧になれます。[Help] メニューから [Topics] を選択すると、オンラインヘルプ ファイルのリストが表示されます。

はじめに

序章には、32 ビット言語ツールを使い始める前に知っておくと便利な一般情報を記載しています。主な内容は以下の通りです。

- 本書の構成
- 本書の表記規則
- 推奨参考資料
- Microchip 社のウェブサイト
- myMicrochip 変更通知サービス
- お客様サポート

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。

パート 1 - MPLAB XC32 アセンブラ

- **第 1 章 アセンブラの概要** - アセンブラ動作の概要
- **第2章 アセンブラ コマンドライン オプション** - アセンブラ向けコマンドライン オプションの詳細
- **第 3 章 MPLAB XC32 アセンブリ言語** - マクロアセンブラが使うソース言語の詳細
- **第 4 章 アセンブラ ディレクティブ** - ソースコード内のアセンブラ コマンドの説明
- **第 5 章 アセンブラのエラー / 警告 / メッセージ** - エラー、警告、メッセージの一覧

パート 2 - MPLAB XC32 オブジェクト リンカ

- **第 6 章 リンカの概要** - リンカ動作の概要
- **第 7 章 リンカのコマンドライン インターフェイス** - リンカ向けコマンドライン オプションの詳細
- **第 8 章 リンカスクリプト** - リンカスクリプトの生成方法と、リンカスクリプトを使ってリンカ動作を制御する方法
- **第9章 リンカ処理** - リンカによって入力ファイルからアプリケーションをビルドする方法
- **第 10 章 リンカの例** - 32 ビットリンカの例
- **第 11 章 リンカのエラーと警告** - エラー、警告、メッセージの一覧

パート 3 - 32 ビット ユーティリティ (アーカイバ / ライブラリアン等)

- **第 12 章 MPLAB XC32 オブジェクト アーカイバ / ライブラリアン** - アーカイバ / ライブラリアン向けコマンドライン オプションの詳細
- **第 13 章 その他のユーティリティ** - その他のユーティリティとそれらの動作の説明

パート 4 - 補遺

- **補遺 A. 非推奨の機能** - 非推奨の機能について
- **補遺 B. 便利なテーブル** - 便利なテーブル (ASCII キャラクタセット、16 進値から 10 進値への換算表)
- **補遺C. GNU フリー文書利用許諾契約書** - GNU言語ツールの使用に関するライセンス要件

本書の表記規則

本書には以下の表記規則を適用します。

本書の表記規則

概要	適用	例
Arial、MS ゴシックフォント		
二重かぎカッコ : 『』	参考資料	『MPLAB® IDE ユーザガイド』
太字	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 (>) を使い、角カッコで囲んだ斜体の下線付きテキスト	メニュー項目の選択	*File>Save*
太字	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 : 文字 :{ | }	いずれかの引数を選択する場合 (OR 選択)	`errorlevel {0|1}`
省略記号 : ...	繰り返されるテキスト	`var_name [, var_name...]`
	ユーザが定義するコード	`void main (void) { ... }`

推奨参考資料

本書には、32 ビット言語ツールの使い方を記載しています。参考資料として、Microchip 社が提供する以下の文書を推奨します。

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Readmeファイル(HTML ファイル) でご覧になれます。

MPLAB® XC32 C/C++ コンパイラ ユーザガイド (DS51686)

32 ビット C コンパイラの使い方を記載したガイド書です。32 ビットリンカは、このツールと組み合わせて使います。

32 ビット言語ツール ライブラリ (DS51685)

Microchip 社の 32 ビットデバイス向けに利用可能なライブラリを説明する一覧を記載しています。これには標準ライブラリ (算術ライブラリを含む) とコンパイラ ビルトイン関数を含みます。32 ビット周辺モジュール ライブラリについては、各タイプの周辺モジュール ライブラリに付属する HTML ファイルに記載しています。

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。

- 各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

Microchip社のウェブサイト

Microchip 社は、自社が運営するウェブサイト (www.microchip.com) を通してオンライン サポートを提供しています。当ウェブサイトでは、お客様に役立つ情報と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 ノートとサンプル プログラム、設計リソース、ユーザガイドとハードウェア サポート文書、最新のソフトウェアと過去のソフトウェア
- **技術サポート** - よく寄せられる質問 (FAQ)、技術サポートのご依頼、オンライン ディスカッション グループ、Microchip 社のコンサルタント プログラム メンバーの一覧
- **Microchip 社の事業** - プロダクト セレクタガイドとご注文案内、プレスリリース、セミナーとイベントの一覧、営業所の一覧

myMICROCHIP変 更通知サービス

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。

http://www.microchip.com/pcn 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がご覧になれます。ツールの主なカテゴリは下記の通りです。

- **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 MPLAB® REAL ICE™ インサーキット エミュレータの最新情報です。
- **インサーキット デバッガ** - Microchip 社製 MPLAB ICD 3 インサーキット デバッガと PICkit™ 3 インサーキット デバッガ / プログラマの最新情報です。
- **MPLAB® X IDE** - Microchip 社の MPLAB X IDE(開発システムツール向け統合開発環境) の最新情報です。これには MPLAB X IDE、MPLAB X IDE プロジェクト マネージャ、MPLAB X エディタ、MPLAB X 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 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 デバイス プログラマ等の量産プログラマが含まれます。また、PIC-kit 3 等、量産向けではない開発用プログラマも含まれます。
- **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、その他の各種評価用ボードが含まれます。

お客様サポート

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。

- 代理店または販売担当者
- 弊社営業所
- 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

サポートは販売代理店までお問い合わせください。各地の営業所もご利用になれます。本書の最後のページには各国の営業所の一覧を記載しています。

技術サポートは以下のウェブページからもご利用になれます。
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。

パート 1 - MPLAB XC32 アセンブラ

NOTE:

第 1 章　アセンブラの概要

1.1　はじめに

MPLAB® XC32 アセンブラ (xc32-as) は、dsPIC32 MCU ファミリ向けのシンボリック アセンブリ言語から再配置可能なマシンコードを生成します。本アセンブラは、アセンブリコードを開発するためのプラットフォームを提供するWindows®オペレーティング システム コンソール アプリケーションです。本アセンブラは Free Software Foundation が提供する GNU アセンブラの移植版です。

第 1 章の主な内容は以下の通りです。

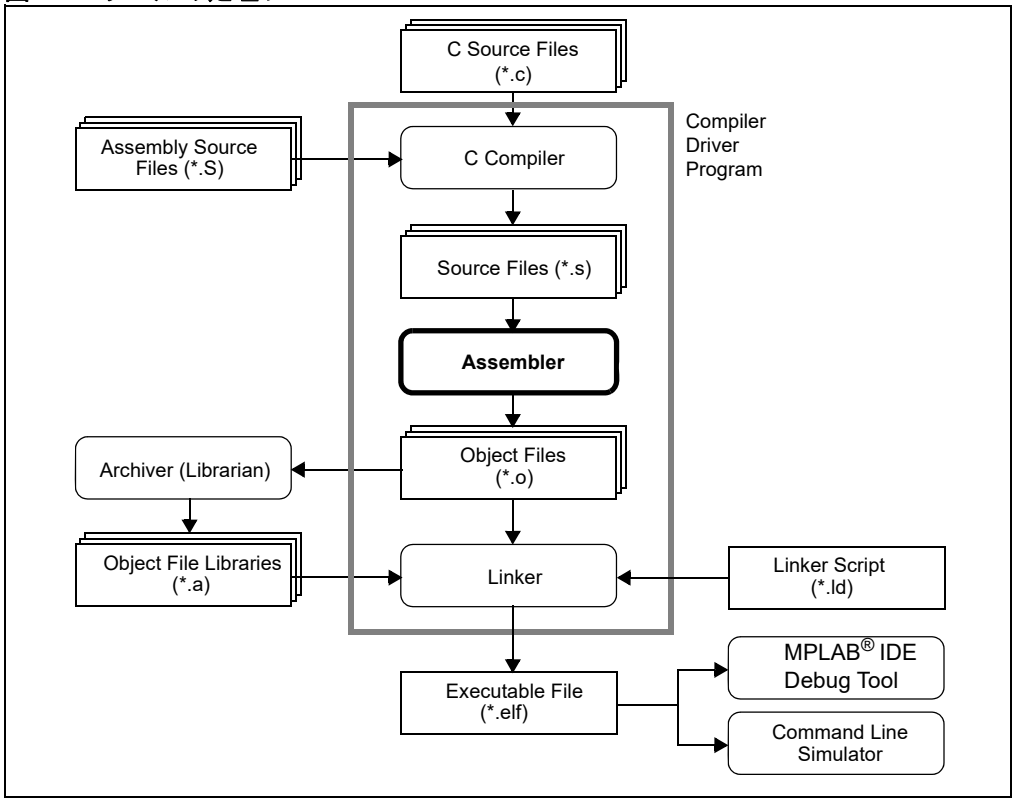
- アセンブラとその他の開発ツール
- 特長
- 入出力ファイル

1.2　アセンブラとその他の開発ツール

MPLAB XC32 アセンブラは、ユーザ アセンブリ ソースファイルを変換します。また、MPLAB XC32 C/C++ コンパイラは、本アセンブラを使ってオブジェクト ファイルを生成します。

C プリプロセッサがアセンブリ ソースファイル (*.S) を処理した後に、本アセンブラが再配置可能オブジェクト ファイルを生成します。このファイルをアーカイブに含めるか他の再配置可能オブジェクト ファイルおよびアーカイブとリンクする事で、実行可能ファイルを生成できます。ツールの処理フローを図 1-1 に示します。

図 1-1: ツールの処理フロー



1.3 特長

本アセンブラの主な特長は以下の通りです。

- MIPS32、MIPS16e、microMIPS 命令セットをサポート
- ELF オブジェクト フォーマットをサポート
- Linux®、OS X®、Windows オペレーティング システムに対応
- 豊富なディレクティブ セット
- 柔軟なマクロ言語
- コマンドライン インターフェイス
- MPLAB X IDE への統合

1.4 入出力ファイル

標準アセンブラ入出力ファイルを以下に示します。

拡張子	概要
入力	
.S	前処理されるアセンブリ ソースファイル (推奨)
.s	ソースファイル
出力	
.o	オブジェクト ファイル
.1st	リスティング ファイル

8 ビット PIC® MCU 向けの MPASM™ アセンブラとは異なり、MPLAB XC32 アセンブラはエラーファイル、HEX ファイル、シンボルおよびデバッグファイルを生成しません。XC32 アセンブラは、リスティング ファイルと再配置可能オブジェクト ファイル (デバッグ情報を含んでも含まなくても構わない) を生成する事ができます。本アセンブラと一緒に PIC32 MCU 向け MPLAB リンカを使う事で、最終的なオブジェクト ファイルおよびマップファイルと、MPLAB X IDE を使ってデバッグするための最終的な実行ファイルを生成します (図 1-1 参照)。

1.4.1 ソースファイル

本アセンブラは PIC32 命令、アセンブラ ディレクティブ、コメントを含むソースファイルを入力として受け入れます。ソースファイルの例を例 1-1 に示します。

> **Note:** Microchip 社は、アセンブリ ソースファイルの拡張子として「.S」を強く推奨します。そうする事で、C コンパイラ ドライバが容易に使えます (そのファイルをアセンブリ ファイルとして扱うようドライバに指示するためのオプションを指定する必要はありません)。また、大文字の「S」は、そのソースファイルはCプリプロセッサで前処理した後にアセンブラへ渡す必要があるという事を示します。C コンパイラ ドライバの詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

例 1-1: アセンブリコードの例

```
Updated example code:

#include <xc.h>
#define IOPORT_BIT_7 (1 << 7)
    .global main    /* define all global symbols here */
    .text
    /* 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.
     */
    .set noreorder
    /* This is important for an assembly programmer.This
     * directive tells the assembler not to optimize
     * the order of the instructions as well as not to insert
     * 'nop' instructions after jumps and branches.
     */
/*****************************************************************
     * main()
     * This is where the PIC32 start-up code will jump to after initial
     * set-up.
     *****************************************************************/
```

```
   .ent main /* directive that marks symbol 'main' as function in the ELF
             * output
             */
main:
   /* Call function to clear bit relevant to pin 7 of port A.
    * The 'jal' instruction places the return address in the $ra
    * register.
    */
   ori      a0, $0, IOPORT_BIT_7
   jal      mPORTAClearBits
   nop
   /* endless loop */
endless:
   j endless
   nop
   .end main /* directive that marks end of 'main' function and its
             * size in the ELF output
             */
/*********************************************************************
 * mPORTAClearBits(int bits)
 * This function clears the specified bits of IOPORT A.
 *
 * pre-condition:$ra contains return address
 * Input:Bit mask in $a0
 * Output: none
 * Side effect: clears bits in IOPORT A
 ********************************************************************/
   .ent mPORTAClearBits
mPORTAClearBits:
   /* 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 */
   /* function epilogue - restore registers used in this function
    * from stack and adjust stack-pointer
    */
   lw      s0, 0(sp)
   addiu   sp, sp, 4
   /* return to caller */
   jr      ra
   nop
   .end mPORTAClearBits
```

1.4.2 オブジェクト ファイル

本アセンブラは再配置可能 ELF オブジェクト ファイルを生成します。オブジェクト ファイル内のアドレスは未解決であり、実行ファイルとして使えるようにするにはリンクする必要があります。

既定値では、生成されるオブジェクト ファイルの名前は a.out です。コマンドラインで -o オプション (**第 2 章「アセンブラ コマンドライン オプション」**参照) を指定する事により、既定値名とは異なる名前を指定できます。

1.4.3 リスティング ファイル

本アセンブラは、リスティング ファイルを生成する事ができます。このリスティング ファイルは絶対リスティング ファイルでありません (リスティング内のアドレスは、そのセクションの開始位置に対する相対アドレスです)。

既定値では、リスティング ファイルは標準出力に出力されます。コマンドラインで -a=<file>オプション(**第2章「アセンブラ コマンドライン オプション」**参照)を指定すると、リスティング ファイルは指定されたファイル (file) へ出力されます。

本アセンブラが生成するリスティング ファイルは、以下で説明する要素を含みます。例 1-2 に、リスティング ファイルの例を示します。

1.4.3.1 ヘッダ

ヘッダはアセンブラの名前、アセンブルするファイルの名前、ページ番号で構成されます。-an オプションを指定するとヘッダは表示されません。

1.4.3.2 タイトル

タイトル行は .title ディレクティブによって指定されたタイトルを格納します。-an オプションを指定するとタイトルは表示されません。

1.4.3.3 サブタイトル

サブタイトル行は .sbttl ディレクティブによって指定されたサブタイトルを格納します。-an オプションを指定するとサブタイトルは表示されません。

1.4.3.4 高級言語ソース

高級言語ソースは、アセンブラに対して -ah オプションを指定した場合に表示されます。高級言語ソースの書式を以下に示します。

```
<line #>:<filename>       **** <source>
```

例 :

```
1:hello.c       **** #include <stdio.h>
```

1.4.3.5 アセンブラソース

アセンブラソースは、アセンブラに対して -al オプションを指定した場合に表示されます。アセンブラソースの書式を以下に示します。

```
<line #> <addr> <encoded bytes> <source>
```

例 :

```
35 0000 80000434    ori      $a0, $zero, IOPORT_BIT_7
```

Note 1: 行番号は繰り返し可能です。

2: アドレスは絶対アドレスではなく、このモジュール内のセクションに対する相対アドレスです。

3: 命令は、リトルエンディアン形式でエンコードされます。

1.4.3.6 シンボルテーブル

シンボルテーブルは、アセンブラに対して -as オプションを指定した場合に表示されます。定義済みシンボルのリストと未定義シンボルのリストが提供されます。

定義済みシンボルの書式を以下に示します。

```
DEFINED SYMBOLS
<filename>:<line #> <section>:<addr> <symbol>
```

例 :

```
DEFINED SYMBOLS
foo.S:79     .text:00000000 main
foo.S:107    .text:00000014 mPORTAClearBits
```

未定義シンボルの書式を以下に示します。

```
UNDEFINED SYMBOLS
<symbol>
```

例 :

```
UNDEFINED SYMBOLS
WDTCON
WDTCONCLR
```

例 1-2: アセンブラ リスティング ファイルの例

```
GAS LISTING foo.s      page 1

   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                 .nolist
   0
   0
   3                 .list
   4
   5              #define IOPORT_BIT_7 (1 << 7)
   6
   8                 .global baz /* define all global symbols here */
   9
  10                 /* define which section (for example "text")
  11                  * does this portion of code resides in.
  12                  * Typically, all of your code will reside in
                       * the .text section as shown.
  14                  */
  15                 .text
  16
  17                 /* This is important for an assembly programmer.
  18                  * This directive tells the assembler not to
  19                  * optimize the order of the instructions
  20                  * as well as not to insert 'nop' instructions
  21                  * after jumps and branches.
  22                  */
  23                 .set noreorder
  24
  25              .ent baz /* directive that marks symbol 'baz' as
  26                        * a function in ELF output
  27                        */
  28
  29              baz:
  30
  31                 /* Call function to clear bit relevant to pin
  32                  * 7 in port A. The 'jal' instruction places
```

```
  33                  * the return address in $ra.
  34                  */
  35 0000 80000434    ori      $a0, $zero, IOPORT_BIT_7
  36 0004 0500000C    jal      mPORTAClearBits
  37 0008 00000000    nop
  38
  39                /* endless loop */
  40              endless:
  41 000c 03000008    j      endless
  42 0010 00000000    nop
  43
  44              .end baz /* directive that marks end of 'baz'
  45                       * function and registers size in ELF
  46                       * output
  47                       */
DEFINED SYMBOLS
                           *ABS*:00000000 foo.S
                           *ABS*:00000001 __DEBUG
               foo.S:56    .text:00000014 mPORTAClearBits
               foo.S:38    .text:0000000c endless
```

NOTE:

第 2 章　アセンブラ コマンドライン オプション

2.1　はじめに

MPLAB XC32 アセンブラ (xc32-as) は、ホスト PC のコマンドライン インターフェイス (例 : cmd.exe) と MPLAB X IDE プロジェクト マネージャで使う事ができます。

MPLAB X IDE プロジェクト マネージャは、プロジェクトをビルドする際に、xc32-gcc コンパイル ドライバを介して自動的にアセンブラを呼び出します。一般的によく使われるオプションの多くは、MPLAB X IDE プロジェクト ビルドオプション ダイアログ内のチェックボックスで選択できます。しかし、より詳細なオプションは、このダイアログの [Alternate Settings] フィールドで指定する必要があります。MPLAB X IDE 内でプロジェクトをビルドすると、アセンブラに渡されるオプションが出力ウィンドウに表示されます。

第 2 章の主な内容は以下の通りです。

- アセンブラ インターフェイスの構文
- コンパイル ドライバ インターフェイスの構文
- リスティング出力を変更するためのオプション
- 情報出力を制御するためのオプション
- 出力ファイルの生成を制御するためのオプション
- アセンブラ シンボル定義および検索パス オプション
- コンパイル ドライバおよびプロセッサ オプション

2.2　アセンブラ インターフェイスの構文

アセンブラ コマンドラインでは各種オプションとファイル名を指定できます。オプションは任意の順番で指定でき、その位置はファイル名の前でも後でも構わず、ファイル名とファイル名の間であっても構いません。ファイル名の指定順はアセンブリの順番に影響します。

```
xc32-as [options|sourcefiles]...
```

「`--`」(2 つのハイフンのみ) は、アセンブラが変換するファイルの 1 つとして標準入力ファイルを明示的に指定します。「`--`」を除き、「`-`」で始まる全てのコマンドライン引数はオプションです。各オプションはアセンブラの挙動を変更しますが、他のオプションの動作には影響を与えません。

一部のオプションでは、オプションの後でファイル名を指定する必要があります。このファイル名はオプション文字に続けて指定する事も、次のコマンドライン引数として指定する事もできます。例えば、`test.o` という名前の出力ファイルを指定する場合、以下のどちらでも構いません。

- `-o test.o`
- `-otest.o`

> **Note:**　コマンドライン オプションは大文字と小文字を区別します。

2.3 コンパイル ドライバ インターフェイスの構文

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 アーカイブを前処理 / コンパイル / アセンブル / リンクします。このドライバはビルドプロセスを統合するため、多くの場合、ユーザはどのプログラム (ツール) が前処理 / コンパイル / アセンブル / リンクを実行するのか知る必要はありません。ドライバは、要求されたビルド処理に必要なツールを適切に呼び出します。

実際には、アセンブラは通常 xc32-gcc を介して呼び出されます。これにより、アセンブラはファイル名拡張子が *.S または *.s の入力ファイルをアセンブルするよう指定されます。ファイル名が *.S (拡張子が大文字の S) である場合、コンパイル ドライバはそのファイルを CPP スタイルのプリプロセッサで前処理してからアセンブラに渡します。ファイル名が *.s (拡張子が小文字の s) である場合、コンパイル ドライバはそのファイルを直接アセンブラに渡します。

コンパイル ドライバに対するコマンドラインの基本形態は以下の通りです。

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

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

コンパイル ドライバからアセンブラへアセンブラ オプションを渡すには、-Wa オプションを使います。オプションの引数には空白類文字を含めない事が必要です。

例 2-1: コンパイル ドライバ向けコマンドラインの例

```
xc32-gcc -mprocessor=32MX360F512L -I"./include" ASMfile.S
  -o"ASMfile.o" -DMYMACRO=1 -Wa,-ah="ASMfile.lst"
```

コンパイル ドライバの詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

> **Note:** MPLAB X IDE から xc32-gcc コンパイル ドライバを使う場合、ユーザのプロジェクト向けに XC32 コンパイラ ツールチェーンを選択する必要があります。

2.4 リスティング出力を変更するためのオプション

以下のオプションは、リスティング出力を制御するために使います。リスティング ファイルは、コード動作のデバッグと解析用に役立ちます。以下のオプションは、必要な情報を含むリスティング ファイルを構成するために使います。

- -a[suboption] [=file]
- --listing-lhs-width num
- --listing-lhs-width2 num
- --listing-rhs-width num
- --listing-cont-lines num

2.4.1 -a[*suboption*] [=*file*]

-a オプションはリスティング出力を有効にします。-a オプションには以下のサブオプションが使えます。これらはアセンブリ リスティングに含める内容をさらに詳細に制御します。

-ac	false 条件を省略します。
-ad	デバッグ ディレクティブを省略します。
-ah	高級言語ソースを含めます。
-al	アセンブリを含めます。
-am	マクロ展開を含めます。
-an	形態処理を省略します。
-as	シンボルを含めます。
-a=file	指定されたファイルにリスティングを出力します (ファイルはカレント ディレクトリ内にある事が必要です)

サブオプションを何も指定しない場合、既定値サブオプション hls が使われます。サブオプションなしの -a オプションは高級言語ソース、アセンブリ、シンボルのリスティングを要求します。

-aの後に付けるサブオプション文字が複数ある場合、それらの文字をまとめて指定できます。つまり、-al -an と指定する代わりに -aln と指定する事ができます。

2.4.1.1 -ac

-ac は、リスティングから false 条件を省略します。偽の .if または .ifdef (もしくは真の .if または .ifdef の .else) のためにアセンブルされない全ての行はリスティングから省略されます。例 2-2 に、-ac オプションを使わなかった場合のリスティングを示します。例 2-3 に、同じソースに対して -ac オプションを使った場合のリスティングを示します。

例 2-2: -alコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING asm.s                       page 1

   1              # 1 "asm.S"
   2              # 1 "<built-in>"
   1                    .data
   0
   2                    .if 0
   3                      .if 1
   4                      .endif
   5                      .long 0
   6                      .if 0
   7                        .long 0
   8                      .endif
   9                    .else
  10                      .if 1
  11                      .endif
  12 0000 02000000         .long 2
  13                      .if 0
  14                        .long 3
  15                      .else
  16 0004 04000000           .long 4
  17                      .endif
  18                    .endif
  19                    .if 0
  20                      .long 5
  21                    .elseif 1
  22                      .if 0
  23                        .long 6
  24                      .elseif 1
  25 0008 07000000           .long 7
  26                      .endif
  27                    .elseif 1
  28                      .long 8
  29                    .else
  30                      .long 9
  31                    .endif
```

例 2-3: -alcコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING asm.s page 1

  1              # 1 "asm.S"
  2              # 1 "<built-in>"
  1                    .data
  0
  0
  2                    .if 0
  9                    .else
 10                      .if 1
 11                      .endif
 12 0000 02000000         .long 2
 13                      .if 0
 15                      .else
 16 0004 04000000           .long 4
 17                      .endif
 18                   .endif
 19                   .if 0
 21                   .elseif 1
 22                      .if 0
 24                      .elseif 1
 25 0008 07000000           .long 7
 26                      .endif
 27                   .elseif 1
 29                   .else
 31                   .endif
```

Note: -ac オプションにより、一部の行が省略されます。

2.4.1.2 -ad

-ad は、リスティングからデバッグ ディレクティブを省略します。このオプションは、コンパイラが生成したデバッグ情報を含むアセンブリコードを処理する場合に便利です。このオプションを指定すると、コンパイラが生成したデバッグ ディレクティブが省略されるため、リスティングが煩雑になる事を防げます。例 2-4 に、d および hサブオプションの両方を使った場合のリスティングを示します。hサブオプションだけを使った場合 (次の項目参照) に比べると、リスティングが大幅に明解になります。

例 2-4: -alhdコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING test.s page 1


   1              .section .mdebug.abi32
   2              .previous
  10              .Ltext0:
  11              .align2
  12              .globlmain
  13              .LFB0:
  14              .file 1 "src\\test.c"
   1:src/test.c    **** #include <xc.h>
   2:src/test.c    **** volatile unsigned int testval;
   3:src/test.c    ****
   4:src/test.c    **** int
   5:src/test.c    **** main (void)
   6:src/test.c    **** {
  15              .loc 1 6 0
  16              .setnomips16
  17              .entmain
  18             main:
  19         .frame$fp,8,$31# vars= 0, regs= 1/0, args= 0, gp= 0
  20         .mask0x40000000,-8
  21         .fmask0x00000000,0
  22              .setnoreorder
  23              .setnomacro
  24
  25 0000 F8FFBD27 addiu$sp,$sp,-8
  26              .LCFI0:
  27 0004 0000BEAF sw$fp,0($sp)
  28              .LCFI1:
  29 0008 21F0A003 move$fp,$sp
  30              .LCFI2:
   7:src/test.c    ****   testval += 1;
  31              .loc 1 7 0
  32 000c 0000828F lw$2,%gp_rel(testval)($28)
  33 0010 01004224 addiu$2,$2,1
  34 0014 000082AF sw$2,%gp_rel(testval)($28)
   8:src/test.c    ****   return 0;
  35              .loc 1 8 0
  36 0018 21100000 move$2,$0
   9:src/test.c    **** }
  37              .loc 1 9 0
  38 001c 21E8C003 move$sp,$fp
  39 0020 0000BE8F lw$fp,0($sp)
  40 0024 0800BD27 addiu$sp,$sp,8
  41 0028 0800E003 j$31
  42 002c 00000000 nop
```

```
43
44              .setmacro
45              .setreorder
46              .endmain
47              .LFE0:
49
50              .commtestval,4,4
88              .Letext0:
```

2.4.1.3 -ah

-ah は、高級言語のリスティングを要求します。高級言語リスティングには、アセンブリソースがコンパイラによって生成され、かつデバッグ オプション (-g 等) がコンパイラに対して指定され、かつアセンブリ リスティング (-al) が要求される事が必要です。-al は、出力プログラム アセンブリ リスティングを要求します。例 2-5 に、コマンドライン オプション -alh を使って生成したリスティングを示します。

例 2-5: -alhコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING tempfile.s page 1


   1              .section .mdebug.abi32
   2              .previous
   3              .section.debug_abbrev,"",@progbits
   4              .Ldebug_abbrev0:
   5              .section.debug_info,"",@progbits
   6              .Ldebug_info0:
   7              .section.debug_line,"",@progbits
   8              .Ldebug_line0:
   9 0000 34000000 .text
  11              .align2
  12              .globlmain
  13              .LFB0:
  14              .file 1 "src/test.c"
   1:src/test.c    **** #include <xc.h>
   2:src/test.c    **** volatile unsigned int testval;
   3:src/test.c    ****
   4:src/test.c    **** int
   5:src/test.c    **** main (void)
   6:src/test.c    **** {
  15              .loc 1 6 0
  16              .setnomips16
  17              .entmain
  18             main:
  19        .frame$sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0
  20              .mask0x00000000,0
  21              .fmask0x00000000,0
  22              .setnoreorder
  23              .setnomacro
  24
   7:src/test.c    ****   testval += 1;
  25              .loc 1 7 0
  26 0000 0000848F lw$4,%gp_rel(testval)($28)
   8:src/test.c    ****   return 0;
   9:src/test.c    **** }
  27              .loc 1 9 0
  28 0004 21100000 move$2,$0
  29              .loc 1 7 0
  30 0008 01008324 addiu$3,$4,1
  31 000c 000083AF sw$3,%gp_rel(testval)($28)
  32              .loc 1 9 0
  33 0010 0800E003 j$31
  34 0014 00000000 nop
```

```
35
36              .setmacro
37              .setreorder
38              .endmain
39              .LFE0:
40              .sizemain, .-main
41
42              .commtestval,4,4
```

2.4.1.4 -al

-al はアセンブリ リスティングを要求します。このサブオプションは他のサブオプションと一緒に使う事ができます。他のサブオプションの例を参照してください。

2.4.1.5 -am

-am はリスティング内でマクロを展開します。例 2-6 に、-am オプションを使わなかった場合のリスティングを示します。例 2-7 に、同じソースに対して -am オプションを使った場合のリスティングを示します。

例 2-6: -alコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING foo.s                       page 1


   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                      .macro  sum from=0, to=5
   0
   0
   2                      .long   \from
   3                      .if     \to-\from
   4                      sum     "(\from+1)",\to
   5                      .endif
   6                      .endm
   7
   8                      .data
   9 0000 00000000         .long 0
  10 0004 0A000000         sum 10, 14
  10      0B000000
  10      0C000000
  10      0D000000
  10      0E000000
  11 0018 00000000         .long 0
```

例 2-7: -almコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING foo.s                       page 1

  1              # 1 "foo.S"
  2              # 1 "<built-in>"
  1                      .macro  sum from=0, to=5
  0
  0
  2                      .long   \from
  3                      .if     \to-\from
  4                      sum     "(\from+1)",\to
  5                      .endif
  6                      .endm
  7
  8                      .data
  9 0000 00000000         .long 0
 10                      sum 10, 14
 10 0004 0A000000 > .long 10
 10              > .if 14-10
 10              > sum "(10+1)",14
 10 0008 0B000000 >> .long (10+1)
 10              >> .if 14-(10+1)
 10              >> sum "((10+1)+1)",14
 10 000c 0C000000 >>> .long ((10+1)+1)
 10              >>> .if 14-((10+1)+1)
 10              >>> sum "(((10+1)+1)+1)",14
 10 0010 0D000000 >>>> .long (((10+1)+1)+1)
 10              >>>> .if 14-(((10+1)+1)+1)
 10              >>>> sum "((((10+1)+1)+1)+1)",14
 10 0014 0E000000 >>>>> .long ((((10+1)+1)+1)+1)
 10              >>>>> .if 14-((((10+1)+1)+1)+1)
 10              >>>>> sum "(((((10+1)+1)+1)+1)+1)",14
 10              >>>>> .endif
 10              >>>> .endif
 10              >>> .endif
 10              >> .endif
 10              > .endif
 11 0018 00000000         .long 0
```

Note: 「>」は展開されたマクロ命令を示します。

2.4.1.6 -an

-an はリスティング 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。例 2-8 に、-an オプションを使わなかった場合のリスティングを示します。例 2-9 に、同じソースに対して -an オプションを使った場合のリスティングを示します。

例 2-8: -alコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING foo.s                       page 1
User's Guide Example
Listing Options
   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                 .text
   0
   0
   2                 .title "User's Guide Example"
   3                 .sbttl "Listing Options"
GAS LISTING foo.s                       page 2
User's Guide Example
Listing Options
   4                 .psize 10
   5
   6 0000 01001A3C    lui $k0, 1
   7 0004 02001A3C    lui $k0, 2
   8 0008 03001A3C    lui $k0, 3
   9                 .eject
GAS LISTING foo.s                       page 3
User's Guide Example
Listing Options
  10 000c 04001A3C    lui $k0, 4
  11 0010 05001A3C    lui $k0, 5
```

例 2-9: -alnコ マンドライン オプションを使って生成したリスティング ファイル

```
   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                 .text
   0
   0
   2                 .title "User's Guide Example"
   3                 .sbttl "Listing Options"
   4                 .psize 10
   5
   6 0000 01001A3C    lui $k0, 1
   7 0004 02001A3C    lui $k0, 2
   8 0008 03001A3C    lui $k0, 3
   9                 .eject
  10 000c 04001A3C    lui $k0, 4
  11 0010 05001A3C    lui $k0, 5
```

2.4.1.7 -as

-as はシンボル テーブルのリスティングを要求します。例 2-10 に、コマンドライン オプション -as を使って生成したリスティングを示します。定義済みシンボルと未定義シンボルの両方がリスティングに含まれます。

例 2-10: -asコ マンドライン オプションを使って生成したリスティング ファイル

```
GAS LISTING example.s page 1

DEFINED SYMBOLS
                           *ABS*:00000000 src\example.c
           example.s:18    .text:00000000 main
                           *COM*:00000004 testval

UNDEFINED SYMBOLS
bar
```

2.4.1.8 -a=*file*

=fileは出力ファイルの名前を定義します。このファイルはカレント ディレクトリ内に存在する必要があります。

2.4.2 --listing-lhs-width *num*

--listing-lhs-width オプションは、リスティング ファイルの出力データ列の幅を設定するために使います。既定値では、これは 1 ワードに設定されます。以下の行がリスティングから抽出されます。出力データ列は太字テキストで示しています。

```
2 0000 54686973  .ascii "This is an example"
2      20697320
2      616E2065
2      78616D70
2      6C650000
```

--listing-lhs-width 2 を使うと、同じ行が以下のようにリスティング内に表示されます。

```
2 0000 54686973 20697320  .ascii "This is an example"
2      616E2065 78616D70
2      6C650000
```

2.4.3 --listing-lhs-width2 *num*

--listing-lhs-width2オプションは、リスティング ファイルの出力データ列の連続する複数行の幅を設定するために使います。既定値では、これは 1 に設定されます。指定された幅が最初の行より小さい場合、このオプションは無視されます。以下の行がリスティングから抽出されます。出力データ列は太字テキストで示しています。

```
2 0000 54686973  .ascii "This is an example"
2      20697320
2      616E2065
2      78616D70
2      6C650000
```

--listing-lhs-width2 3 を使うと、同じ行が以下のようにリスティング内に表示されます。

```
2 0000 54686973  .ascii "This is an example"
2      20697320 616E2065 78616D70
2      6C650000
```

2.4.4 --listing-rhs-width *num*

--listing-rhs-widthオプションは、ソースファイルからの行の最大幅(文字数)を設定するために使います。既定値では、これは 100 に設定されます。
--listing-rhs-width オプションを使わずに生成したリスティングからは以下の行が抽出されます。太字のテキストはソースファイルからの行です。

```
2 0000 54686973  .ascii "This line is long"
2      206C696E
2      65206973
2      206C6F6E
2      67000000
```

--listing-rhs-width 22 を使うと、同じ行が以下のようにリスティング内に表示されます。

```
2 0000 54686973  .ascii "This line is
2      206C696E
2      65206973
2      206C6F6E
2      67000000
```

行はリスティング内で切り詰められます (折り返されません) が、データはそこに存在します。

2.4.5 --listing-cont-lines *num*

--listing-cont-lines オプションは、リスティングの出力データ列向けに使われる連続した行の最大数を設定するために使います。既定値では、これは 8 に設定されます。--listing-cont-lines オプションを使わずに生成したリスティングからは以下の行が抽出されます。太字のテキストは、リスティングの出力データ列向けに使われる連続行です。

```
2 0000 54686973  .ascii "This is a long character
                         sequence"
2      20697320
2      61206C6F
2      6E672063
2      68617261
```

表示されるバイトの数は ASCII 文字列内のバイト数と一致します。しかし --listing-cont-lines 2 オプションが使われた場合、以下のように、出力データは 2 つの連続した行の後で切り詰められるという事に注意が必要です。

```
2 0000 54686973  .ascii "This is a long character
                         sequence"
2      20697320
2      61206C6F
```

2.5 情報出力を制御するためのオプション

以下で説明するオプションは情報の出力方法を制御します。コードの変換と実行に関するエラー、警告、メッセージコードはこれらのオプションによって制御されます。

丸カッコ内はオプション指定の短縮形です。例えば `--no-warn` は `-W` として指定する事もできます。

2.5.1 `--fatal-warnings`

警告をエラーであるかのように扱います。

2.5.2 `--no-warn (-W)`

警告を抑止します。このオプションを使うと警告は出力されません。このオプションは警告メッセージにだけ影響します。ファイルのアセンブリングには影響しません。このオプションを指定してもエラーは報告されます。

2.5.3 `--warn`

適切な場合に警告を出力します。これが既定値の挙動です。

2.5.4 `-J`

符号付きオーバーフローに関する警告を出力しません。

2.5.5 `--help`

アセンブラは、コマンドラインの使い方とオプションに関するメッセージを表示します。その後アセンブラは動作を停止します。

2.5.6 `--target-help`

アセンブラは、PIC32 ターゲットに固有のコマンドライン オプションに関するメッセージを表示します。その後アセンブラは動作を停止します。

2.5.7 `--version`

アセンブラのバージョン番号を表示します。その後アセンブラは動作を停止します。

2.5.8 `--verbose (-v)`

アセンブラのバージョン番号を表示します。その後アセンブラは動作を継続します。このコマンドライン オプションだけ指定した場合、アセンブラはバージョンを表示した後に、標準入力からのアセンブリ ソースの入力を待機します。アセンブリを終了するには、`<CTRL>-D` を使って EOF 文字を送信します。

2.6 出力ファイルの生成を制御するためのオプション

以下で説明するオプションは、出力ファイルの生成方法を制御します。例えば、出力オブジェクト ファイルの名前を変更するには -o を使います。

丸カッコ内はオプション指定の短縮形です。例えば --keep-locals は -L として指定する事もできます。

2.6.1 -g

シンボリック デバッグ情報を生成します。

2.6.2 --keep-locals (-L)

ローカルシンボル (すなわち .L(大文字) で始まるラベル) を保持します。通常、デバッグ時にそのようなラベルは現れません。なぜなら、それらはアセンブラ プログラムを生成するためのプログラム (コンパイラ等) の使用を意図しているからです。通常、アセンブラもリンカもそのようなシンボルを破棄します。このオプションは、オブジェクト ファイル内でそれらのシンボルを保持するようアセンブラに指示します。

2.6.3 -o *objfile*

オブジェクトファイル出力に *objfile* という名前を付けます。アセンブラを実行すると、エラーが発生しない限り常に 1 つのオブジェクト ファイル出力が生成されます。既定値では、このファイルの名前は a.out です。このオプション (ファイル名を 1 つだけ指定) は、オブジェクト ファイルに別の名前を付けます。オブジェクト ファイルが呼び出されると、アセンブラは同名の既存ファイルを上書きします。

2.6.4 -Z

エラーが発生してもオブジェクト ファイルを生成します。通常では、アセンブラはエラーメッセージを出力した後に出力を生成しません。何らかの理由により、アセンブラがエラーメッセージを出力した後のオブジェクト ファイル出力に興味がある場合、-Z オプションを使います。エラーが発生してもアセンブラは動作を継続し、「n errors, m warnings, generating bad object file」という書式の最終警告メッセージの後にオブジェクト ファイルを書き込みます。

2.6.5 -MD *file*

file に依存情報を書き込みます。アセンブラは依存性ファイルを生成できます。このファイルは、メイン ソースファイルの依存性を記述するのに適した 1 つの規則で構成されます。この規則は、引数で指定された名前のファイルに書き込まれます。この機能は、make ファイルの自動更新で使えます。

2.7 アセンブラ シンボル定義および検索パス オプション

以下で説明するオプションは、これまでの説明では定義されなかった関数を実行します。

2.7.1 --defsym *sym=value*

シンボル *sym* を指定された値 *value* に定義します。

2.7.2 -I *dir*

このオプションは、ディレクトリ *dir* 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。複数のパスをリストに追加するために、-I は何度でも使えます。常に現在作業中のディレクトリが最初に検索されます。その後に、アセンブラは -I によって指定されたディレクトリをコマンドライン上で指定された順番に (左から右へ) 検索します。

アセンブラにディレクトリが渡されると、このオプションはアセンブラの .include ディレクティブによって使われる検索パスに影響します。#include ディレクティブ向けに C プロセッサによって使われる検索パスに影響を与えるには、対応するオプションを xc32-gcc コンパイル ドライバに渡します。

2.8 コンパイル ドライバおよびプロセッサ オプション

以下で説明するコンパイル ドライバ (xc32-gcc) および C プリプロセッサ オプションは、アセンブリコード プロジェクトに役立つ場合があります。コンパイル ドライバは、要求に応じてオプションをプリプロセッサに渡します。コンパイル ドライバの詳細とドライバ オプションのより包括的な一覧については、『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

2.8.1 -mprocessor=*device*

コンパイルのターゲット デバイスを選択します
(例 : -mprocessor=32MX360F512L)。

2.8.2 -Wa,*option*

option をオプションとしてアセンブラに渡します。*option* は、コンマで区切る事で複数のアセンブラ オプションを指定できます。*option* の引数には空白類文字を含めない事が必要です。

2.8.3 -Dmacro=*defn*

マクロ (macro) 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。

2.8.4 -Dmacro

マクロ (macro) を 1 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。

2.8.5 -Umacro

マクロ (macro) を未定義にします。-U オプションは全ての -D オプションの後で評価され、その後で全ての -include および -imacros オプションが評価されます。

2.8.6 -I *dir*

#include プロセッサ ヘッダファイルの検索ディレクトリ リストの先頭にディレクトリ *dir* 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 ヘッダファイルをオーバーライド (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。

このオプションがコンパイル ドライバに渡されると、プロセッサの #include ディレクティブによって使われる検索パスに影響します。アセンブラの .include ディレクティブによって使われる検索パスに影響を与えるには、-Wa オプションを使って、対応するオプションをアセンブラに渡します。

2.8.7 -save-temps

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に基づく名前が付けられます。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)

2.8.8 -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。

2.8.9 --help

コマンドライン オプションの説明を表示します。

第 3 章　MPLAB XC32 アセンブリ言語

3.1　はじめに

以下では、マクロアセンブラ向けのソース言語について説明します。全てのオペコード ニーモニックとオペランド構文は、ターゲット デバイスに固有の物です。コンパイラからの中間生成アセンブリ ソースコードと手書きのアセンブリ ソースコードに対して同じアセンブラ アプリケーションを使います。

第 3 章の主な内容は以下の通りです。

- 内部プリプロセッサ
- ソースコードの書式
- 特殊文字
- シンボル
- シンボルに値を与える
- 特殊 DOT シンボル
- 式
- 演算子
- 特殊な演算子

3.2 内部プリプロセッサ

本アセンブラは、以下を実行する内部プリプロセッサを備えています。

1. 余分な空白類文字を調整 / 削除します。行内のキーワードの前に 1 つのスペースまたはタブを残し、その他の空白類文字は全て 1 つのスペースに置き換えます。
2. コメントを削除します。
   全てのコメントを 1 つのスペースまたは適切な数の復帰改行文字に置き換えます。
3. 文字定数を適切な数値に変換します。

   ソースコード内の単一文字 (例 : 'b') を適切な数値に置き換えます。単一文字で構文エラーが発生する場合、アセンブラは 'b' と表示せずに、等価な 10 進数の最初の桁を表示します。

   例えばソースコード内に `.global mybuf, 'b'` というコードがある場合、「Error: Rest of line ignored.First ignored character is '9'.」というエラーメッセージが表示されます。エラーメッセージは '9' と表示します。なぜなら、'b' は等価な 10 進数である 98 に変換されたからです。アセンブラは実際には `.global mybuf,98` を構文解析します。

内部プロセッサは以下を**実行しません**。

1. マクロの前処理
2. ファイルのインクルード処理
3. C コンパイラのプリプロセッサが提供しない全ての処理

ファイル インクルード前処理は `.include` ディレクティブを使って実行できます。**第 4 章「アセンブラ ディレクティブ」**を参照してください。

入力ファイルに拡張子「`.S`」を付ける事により、C コンパイラ ドライバを使って C スタイルの前処理が利用できます。詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

入力ファイルの最初の行が `#NO_APP` である場合、あるいは `-f` オプションを指定した場合、空白類文字とコメントは入力ファイルから削除されます。入力ファイル内の特定部分でのみ空白類文字とコメントを削除するよう指定する事ができます。これを行うには、その部分の前に `#APP` の行を挿入し、その部分の後に `#NO_APP` の行を挿入します。この機能は、コンパイラ出力にコメントと空白類文字がないアセンブリ命令文をサポートする事を主な目的とします。

> **Note:** 入力テキスト部分は前処理されないため、余分な空白類文字、コメント、文字定数を使う事はできません。

3.3 ソースコードの書式

アセンブリ ソースコードは命令文と空白類文字で構成されます。

空白類文字は 1 つまたは複数のスペースまたはタブです。空白類文字は、ソース行を複数の部分に区切るために使います。空白類文字は、コードが読みやすくなるように使う必要があります。文字定数内で使われる場合を除き、全ての空白類文字は 1 つのスペースと厳密に同じ意味を持ちます。

各命令文は以下の一般的書式を持ち、後に復帰改行が続きます。

```
[label:][mnemonic[operands] ][; comment]
```

または

```
[label:][directive[arguments] ][; comment]
```

- ラベル
- ニーモニック
- ディレクティブ
- オペランド
- 引数
- コメント

3.3.1 ラベル

ラベルは全ての文字、数字、アンダーライン文字 (_)、ピリオド (.) を含むキャラクタセットから選ばれた 1 つまたは複数の文字で構成されます。ラベルの頭文字に 10 進数字を使う事はできません (ローカルシンボルの特殊なケースを除く)。詳細は **3.5.1「ローカルシンボル」**を参照してください。ラベルでは大文字と小文字を区別します。長さに制限はなく、全ての文字が有意です。

ラベル定義の直後にコロンが必要です。コロンの後にはスペース、タブ、行末文字、アセンブラ ニーモニックまたはディレクティブを続ける事ができます。

ラベル定義は、それだけを 1 行に書く事ができ、その次に書かれたアドレスを参照します。

リンク後のラベルの値は、メモリ内の特定位置を指す絶対アドレスです。

3.3.2 ニーモニック

ニーモニックは、どのマシン命令をアセンブルするのかアセンブラに伝えます。例えば、加算 (ADD)、ジャンプ (J)、ロード (LUI) 等の命令があります。ユーザが作成するラベルとは異なり、ニーモニックは PIC32 MCU アセンブリ言語によって提供されます。ニーモニックでは大文字と小文字を区別しません。

ターゲットの PIC32 MCU で利用可能な CPU 命令セットの詳細は、そのデバイスのデータシートを参照してください。

本アセンブラは、アセンブリコードの記述を容易にする事を目的とする各種の合成 (synthesized)/ マクロ命令もサポートします。LI (load immediate) 命令は、合成マクロ命令の 1 例です。アセンブラは、32 ビット定数値をレジスタにロードするために、この 1 つの合成命令から 2 つのマシン命令を生成します。

> **Note:** 入力テキスト部分は前処理されないため、余分な空白類文字、コメント、文字定数を使う事はできません。

```
[label:][mnemonic [operands] ] [; comment]
```

OR

```
[label:][directive [arguments] ] [; comment]
```

3.3.3 アセンブリ構文

本アセンブラは以下のために命令を合成します。

• 32 ビット LI (Load Immediate)

• メモリ位置からのロード

• 拡張条件分岐

• 一部の 3 オペランド命令の 2 オペランド形態

• アラインメントされないロード / ストア命令

通常これらの機能は便利ですが、生成されるコードをユーザが完全に制御する必要がある場合、アセンブリ ディレクティブ (.set noat、.set nomacro、.set noreorder 等) を使ってこれらの機能を無効にできます。**4.13「コード生成を制御するディレクティブ」**を参照してください。

3.3.4 ディレクティブ

ソースコード内で使われるアセンブラ ディレクティブはマシンコードに直接変換されません。ディレクティブは入力、出力、データ割り当て等、アセンブラを制御するために使います。ディレクティブの先頭文字は (.) です。利用可能なディレクティブの詳細は**第 4 章「アセンブラ ディレクティブ」**に記載しています。

3.3.5 オペランド

各マシン命令は 0 ～ 4 個のオペランドを使います。全てのマシン命令を記載した一覧は、ターゲット PIC32 MCU のデータシートを参照してください。これらのオペランドは、使用するデータと保存位置に関する情報を命令に提供します。オペランドは、1 つまたは複数のスペースまたはタブによってニーモニックから区切る必要があります。

複数のオペランドはコンマを使って区切ります。オペランドがコンマで区切られていない場合、アセンブラは警告を表示し、最も妥当と推測されるオペランドの区切りを採用します。大部分の PIC32 MCU 命令のオペランドはコア汎用レジスタ、ラベル、リテラル、ベースレジスタ (basereg) + オフセットのいずれかで構成されます。

3.3.5.1 汎用レジスタ オペランド

PIC32 MCU コアは、整数演算とアドレス計算用に 32 個の 32 ビット汎用レジスタ (GPR) を備えています。大部分の PIC32 MCU 命令は、ソースまたはディスティネーション (もしくはその両方) 向けに 1 つまたは複数の GPR オペランドを使います。

レジスタ オペランドは、前にドル記号「$」を付ける事で区別します。「$」の直後にレジスタ番号が続きます。例 3-1 に、レジスタ番号オペランドを使うアセンブリ ソースコードを示します。

しかし、アセンブルする前にコンパイル ドライバ (xc32-gcc) を使って CPP スタイルのプリプロセッサでソースコードを前処理する場合、C コンパイラに付属する xc.h ヘッダファイル内で提供されるマクロが利用できます。これらのマクロは、通常のレジスタ名を対応するレジスタ番号へ割り当てます。例 3-2 に、オペランドに通常のレジスタ名を使ったアセンブリ コードを示します。PIC32 MCU レジスタの用法とコンパイラのランタイム環境に関する詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』'(DS51686) を参照してください。

例 3-1: レジスタ番号オペランドを使ったアセンブリ ソースコード

```
.text
# Add Word
li   $2, 123
li   $3, 456
add  $4, $2, $3
```

例 3-2: 通常のレジスタ名を使ったアセンブリ ソースコード

```
#include <xc.h>
  .text
  /* Add Word */
  li   v0, 123      /* v0 is a return-value register */
  li   v1, 456      /* v1 is a return-value register */
  add  a0, v0, v1   /* a0 is an argument register */
```

3.3.5.2 リテラル値オペランド

リテラル値は 16 進、8 進、2 進、10 進形式のいずれかで表現できます。16 進値は先頭の「0x」により区別します。8 進値は先頭の「0」により区別します。2 進値は先頭の「0B」または「0b」により区別します。10 進値の前後には特別な文字を付ける必要はありません。

例 :

`0xe`、`016`、`0b1110`、`14` は全てリテラル値 14 を表します。

`-5` はリテラル値 -5 を表します。

`symbol` は `symbol` の値を表します。

3.3.5.3 ベースレジスタ (BaseReg) + オフセット オペランド

ロード / ストア動作は、ベースレジスタ (BaseReg) + オフセット オペランドを使ってメモリ位置を選択します。このタイプのオペランドでは、32 ビット符号付きオフセット値をベースレジスタの内容に加算する事により、実効アドレスを生成します。PIC32 MCU のデータシートには、このタイプのオペランドを `Mem[R+offset]` として記載しています。

例 3-3: ベースレジスタ (BaseReg) + オフセット オペランドを使ったアセンブリ ソースコード

```
#include <xc.h>
        .data
        .align 4
MY_WORD_DATA:
        .word 0x10203040, 0x8090a0b0
        .text
        .global example
        /* Store Word */
example:
        la   v0, MY_WORD_DATA
        lui         v1,0x1111
        ori         v1,v1,0x4432
        lui         a0,0x5555
        ori         a0,a0,0x1123
        sw v1, 0(v0)     /* Mem[GPR[v0]+0] <- GPR[v1] */
        sw a0, 4(v0)     /* Mem[GPR[v0]+4] <- GPR[a0] */
        lw a1, 0(v0)     /* GPR[a1] <- Mem[GPR[v0]+0] */
        b .
```

C コンパイラはグローバル ポインタ相対 (gp-rel) 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 ±32 KB 内にあるデータ (合計 64 KB) 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。C コンパイラの `-Gnum` オプションは、2 命令ではなく 1 命令でアドレス指定可

能なグローバルおよびスタティック データアイテムの最大サイズを制御します。コンパイラの gnum の既定値は 8 バイトです。これは、全ての単純なスカラー変数を保持するには十分な大きさです。

> **Note:** GP 相対アドレス指定を使うには、コンパイラとアセンブラは全ての「小さな」(small) 変数と定数を 1 つの「小さな」(small) セクション内に集約する必要があります。グローバル ポインタと -G オプションの詳細は『MPLAB® XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

例 3-4: GP 相対アドレス指定を使ったアセンブリ ソースコード

```
    .align 2
    .globl foo
    .set   nomips32
    .ent   foo
foo:
    .set   noreorder
    .set   nomacro

    lw     $3,%gp_rel(testval)($28)
    addiu  $2,$3,1
    sw     $2,%gp_rel(testval)($28)
    j      $31
    nop

    .set   macro
    .set   reorder
    .end   foo
```

GP 相対アドレス指定を使う場合、以下の危険性があります。

- グローバルな (すなわちパブリックまたは外部の) データアイテムを宣言するアセンブラコードを書く際は、以下に特別な注意を払う必要があります。
  - gnum バイト以下の書き込み可能な初期化データは、下の例のように .sdata セクション内に明示的に配置する必要があります。

```
            .sdata
 small:.word  0x12345678
```

  - グローバルな共有データは、下の例のように適正サイズで宣言する必要があります。

```
            .comm small, 4
            .comm big, 100
```

  -「小さな」(small) 外部変数も、下の例のように正しく宣言する必要があります。

```
            .extern smallext, 4
```

- プログラムが非常に多数の小さな (small) データアイテムまたは定数を含む場合、C コンパイラの -G8 オプションはデータが 64 KB を超えても「small」領域内に配置しようと試みる場合があります。この兆候は、リンク時に「不明瞭な再配置エラー」("relocation truncated") が発生する事で分かります。これに対処するには、コンパイラの -G0 オプションを使って GP 相対アドレス指定を無効にするか、アセンブリコード内で small データセクション内の予約済み空間 (.sbss および .sdata) を減らします (あるいは両方の対策を講じます)。

3.3.6 引数

各ディレクティブは 0 ～ 3 個の引数を使います。これらの引数は、コマンドの実行方法に関する追加の情報をディレクティブに提供します。引数は 1 つまたは複数のスペースまたはタブによってディレクティブから区切る必要があります。複数の引数はコンマで区切る必要があります。利用可能なディレクティブの詳細は**第 4 章「アセンブラ ディレクティブ」**に記載しています。

3.3.7 コメント

アセンブラでは、コメントを以下の 2 通りの方法で表現できます。

3.3.7.1 1 行コメント

このタイプのコメントはコメント文字から始まり、行末まで終わります。1 行コメントにはハッシュ記号 (#) を使います。

> **Note:** このコメント文字は、MPASMアセンブラおよびMPLABアセンブラ(PIC24 MCU および dsPIC DSC 向け) によって認識される文字とは異なります。

3.3.7.2 複数行コメント

このタイプのコメントは複数行にまたがる事ができます。複数行コメントには「/* ... */」を使います。

これらのコメントはネスティングできません。

例 :

```
/* All
of these
lines
are
comments */
```

3.4 特殊文字

定数とは、文脈を知らなくても調べれば値が分かるように書かれた値です。
例 :

```
.byte 74, 0112, 0b01001010, 0x4A, 0x4a, 'J', '\J'#All the same value
.ascii "Ring the bell\7" #A string constant
.float 0f-3141592653589793238462643383279502884l971.693993751E-40
```

3.4.1 数値定数

アセンブラは、マシンに保存される方法に基づいて 2 種類の数値を識別します。整数は、C 言語の `long` に収まる数値です。浮動小数点数は IEEE 754 に従います。

3.4.1.1 整数

2 進整数は、「0b」または「0B」の後に 1 つまたは複数の 0 または非 0 の 2 進数字「01」が続きます。

8 進整数は、「0」の後に 1 つまたは複数の 0 または非 0 の 8 進数字「01234567」が続きます。

10 進整数は、非 0 の 10 進数字の後に 1 つまたは複数の 0 または非 0 の 10 進数字「0123456789」が続きます。

16 進整数は、「0x」または「0X」の後に 1 つまたは複数の 16 数字「0123456789abcdefABCDEF」が続きます。

負の整数には前置演算子「-」を使います。

3.4.1.2 浮動小数点数

浮動小数点数は IEEE 754 形式で表現します。浮動小数点数は先頭から順番に以下により構成されます。

- オプションの接頭辞
  数字「0」の後に「e」、「f」、「d」のいずれかが続きます (大文字でも小文字でも可)。浮動小数点数は `.float` および `.double` ディレクティブでのみ使われるため、2 進表現の精度は接頭辞とは無関係です。
- オプションの符号 (「+」または「-」)
- オプションの整数部 (0 または非 0 の 10 進数字)
- オプションの小数部「.」の後に続く 0 または非 0 の 10 進数字
- オプションの指数部
  - 「E」または「e」
  - オプションの符号 (「+」または「-」)
  - 1 つまたは複数の 10 進数字

少なくとも整数部または小数部のどちらかが必要です。浮動小数点数は通常の 10 を基数とする値を持ちます。

浮動小数点数は、アセンブラが動作するコンピュータ上の浮動小数点ハードウェアとは無関係に計算されます。

3.4.2 文字定数

文字定数には 2 種類あります。文字 (Character) は 1 つの文字 (1 バイト) を表し、その値は数式内で使えます。文字列 (String) は複数バイトを含む事ができ、それらのバイトの値は算術式内で使えません。

3.4.2.1 文字

1 つの文字は、一重引用符の直後にその文字を書くか、一重引用符の直後にその文字ともう 1 つの一重引用符を書く事によって表現できます。本アセンブラは、特殊な制御文字を表現するために以下のエスケープ文字をサポートします。

表 3-1: 特殊文字とその用法

エスケープ文字	概要	HEX 値
\a	ベル (警報音)	07
\b	バックスペース	08
f	改ページ	0C
\n	復帰改行	0A
\r	復帰	0D
\t	水平タブ	09
\v	垂直タブ	0B
\\	バックスラッシュ	5C
\?	クエスチョンマーク	3F
\"	二重引用符	22
\digit digit digit	8 進文字コード (数値コードは 3 桁の 8 進数字)	
\x hex-digits	16 進文字コード (後続の全ての 16 進数字は区切らずに続けます。大文字 (X) でも小文字 (x) でも使えます。)	

数式内の文字定数の値は、その文字に対応するマシンのバイト幅コードです。アセンブラは、文字コードは ASCII であると想定します。

3.4.2.2 文字列

文字列は二重引用符で囲みます。文字列には二重引用符と NULL 文字を含める事ができます。文字列に特殊文字を含めるには、それらの文字をエスケープします (文字の前にバックスラッシュ「\」を付けます)。文字列に適用されるのと同じエスケープシーケンスが文字にも適用されます。

3.4.2.3 一般構文規則

表 3-2 に、本アセンブラに適用される一般的構文規則をまとめて示します。

表 3-2: 構文規則

文字	概要	構文の用法
.	ピリオド	ディレクティブの先頭
#	ハッシュ	1 行コメントの先頭
/*	スラッシュとアスタリスク	複数行コメントの先頭
*/	アスタリスクとスラッシュ	複数行コメントの終端
:	コロン	ラベル定義の終端
	何もなし	リテラル値の先頭
'c'	一重引用符で囲んだ文字	単一文字値を指定
"string"	二重引用符で囲んだ文字列	文字列を指定

3.5 シンボル

シンボルは全ての文字、数字、アンダーライン文字 (_)、ピリオド (.) を含むキャラクタセットから選ばれた 1 つまたは複数の文字で構成されます。シンボルの先頭に数字は使えません。大文字と小文字は区別されます (例 : foo と Foo は異なるシンボルです)。長さに制限はなく、全ての文字が有意です。

各シンボルは厳密に 1 つの名前しか持ちません。アセンブリ プログラム内の各名前は、厳密に 1 つのシンボルを参照します。シンボルの名前はプログラム内で何度でも使えます。

3.5.1 ローカルシンボル

ローカルシンボルは、ラベル向けにテンポラリ スコープが必要な場合に使われます。プログラム全体で再使用可能なローカルシンボル名は 10 個あります。それらは、名前「0」、「1」、...、「9」を使って参照できます。ローカルシンボルを定義するには、「N」、「N」、...、「N」(N は 0 ～ 9 の任意の数字) の形態のラベルを書きます。そのシンボルの直近の定義を参照するには、ラベルを定義した時と同じ数字を使って「Nb」を書きます。ローカルラベルの次の定義を参照するには、「Nf」を書きます。「b」は「backwards」(後方) を意味し、「f」は「forwards」(前方) を意味します。

これらのラベルの使い方に関して制限はなく、再使用もできます。同じローカルラベルを (同じ番号「N」を使って) 繰り返し定義できます。しかし、参照できるのは、直近に定義されたその番号のローカルラベル (後方参照) または特定ローカルラベルの次の定義 (前方参照) だけです。

また、最初の 10 個のローカルラベル (「0:」...「9:」) は、他よりも少し効率の良い方法で実装されるという事に注意してください。

以下に例を示します。

例 3-5: シンボルの用法

```
1: b 1f
2: b 1b
1: b 2f
2: b 1b
```

上記は下記と等価です。

```
label_1: b label_3
label_2: b label_1
label_3: b label_4
label_4: b label_3
```

ローカルシンボル名は単なる表記上のデバイスに過ぎません。それらは即座により一般的なシンボル名に変換されてからアセンブラによって使われます。これらの一般的なシンボル名はシンボルテーブルに保存され、エラー メッセージ内で使われます。また、オプションにより、オブジェクト ファイルに出力されます。

3.6 シンボルに値を与える

シンボルの後に等号「=」と式を書く事により、そのシンボルに任意の値を与える事ができます。

例 :

```
VAR = 4
```

3.7 特殊 DOTシ ンボル

特殊シンボル「.」は、アセンブラが現在処理中のアドレスを参照します。従って、式「melvin: .long .」は、melvin がそれ自身のアドレスを格納するよう定義します。「.」への値の代入は、.org ディレクティブと同様に扱われます。従って、式「.=.+4」は「.space 4」と同じです。

実行可能セクション内で使った場合、「.」はプログラム カウンタ アドレスを参照します。PIC32 MCU の場合、プログラム カウンタは 32 ビット命令ワードごとに 4 つインクリメントします。ユーザコードは、DOT シンボルを変更した後に命令を正しくアラインメントするよう配慮する必要があります。

3.7.1 定義済みシンボル

本アセンブラは、ソースコード内で条件付きディレクティブによって評価可能な各種シンボルを定義済みです。

表 3-3: 定義済みシンボル

シンボル	定義
P32MX	PIC32MX ターゲット デバイスファミリ
P32MZ	PIC32MZ ターゲット デバイスファミリ
HAS_MIPS32R2	デバイスは MIPS32r2 命令セットをサポートする
HAS_MIPS16	デバイスは MIPS16e 命令セットをサポートする
HAS_MICROMIPS	デバイスは microMIPS 命令セットをサポートする
HAS_DSPR2	デバイスは DSPr2 エンジンをサポートする
HAS_MCU	デバイスは MIPS MCU 拡張をサポートする
HAS_L1CACHE	デバイスは L1 データ / プログラム キャッシュを有する
HAS_VECTOROFFSETS	デバイスはベクタテーブル向けに設定可能オフセットを使う

3.8 式

式はアドレスまたは数値を指定します。式の前および / または後に空白類文字を置く事ができます。式の結果は絶対的な値または特定セクションへのオフセットである事が必要です。式の結果が絶対的ではないにも関わらずセクションに関する十分な情報がアセンブラに提供されない場合、アセンブラは動作を停止してエラーメッセージを生成します。

3.8.1 空の式

空 (empty) の式は値を持ちません (空白または NULL)。絶対的な式が必要な場合、空の式は省略できます。これによりアセンブラは絶対的な値として 0 を想定します。

3.8.2 整数式

整数式はオペランドで区切られた 1 つまたは複数の引数です。引数はシンボル、数値、副次式のどれかです。副次式は丸カッコで囲まれた整数式または前置演算子が付いた引数です。

プログラムメモリ内のシンボルを含む整数式は、プログラム カウンタ (PC) 単位で評価されます。MIPS32 モードの場合、プログラム カウンタは命令ワードごとに 4 つインクリメントします。例えば、ラベル L の後で次の命令へ分岐する場合、デスティネーションとして L+4 を指定します。

例 :

```
b  L+4
```

3.9 演算子

演算子は算術関数 (+、% 等) です。前置演算子の後に引数が続きます。中置演算子は引数と引数の間で使います。演算子の前および / または後に空白類文字を置く事ができます。

前置演算子は中置演算子よりも高い優先度を有します。中置演算子の優先度はタイプに応じて異なります。

3.9.1 前置演算子

本アセンブラは以下の前置演算子をサポートします。各前置演算子は 1 つの引数を持ち、引数は絶対的である事が必要です。

表 3-4: 前置演算子

演算子	概要	例
-	否定 (2 の補数否定)	`-1`
~	ビットごとの反転 (1 の補数)	`~flags`

3.9.2 中置演算子

中置演算子は前後に 1 つずつ (合計 2 つ) の引数を持ちます。下表に示すように、これらの演算子はタイプに応じた優先度を持ちます。同じ優先度の演算子は左から右へ実行されます。「+」と「–」を除き、演算子の両側の引数は絶対的である事が必要であり、結果は絶対的です。

表 3-5: 中置演算子

演算子	概要	例
算術		
*	乗算	5 * 4 (=20)
/	除算 (切り詰めは C 演算子「/」と同様)	23 / 4 (=5)
%	剰余	30 % 4 (=2)
<<	左シフト (C 演算子の「<<」と同じ)	2 << 1 (=4)
>>	右シフト (C 演算子の「>>」と同じ)	2 >> 1 (=1)
ビット単位		
&	ビット単位の AND	4 & 6 (=4)
^	ビット単位の XOR	4 ^ 6 (=2)
!	ビット単位 NOT OR	0x1010 !0x5050 (=0xBFBF)
|	ビット単位の OR	2 | 4 (=6)
単純算術		
+	加算 一方の引数が絶対的である場合、結果は他方の引数のセクションを持ちます。異なるセクションからの引数同士を加算する事はできません。	4 + 10 (=14)
-	減算 右辺の引数が絶対的である場合、結果は左辺の引数のセクションを持ちます。両方の引数が同じセクション内である場合、結果は絶対的です。異なるセクションからの引数同士を減算する事はできません。	14 - 4 (=10)
関係		
==	等しい	.if (x == y)
!=	等しくない (<> も同じ意味)	.if (x != y)
<	より小さい	.if (x < 5)
<=	以下	.if (y <= 0)
>	より大きい	.if (x > a)
>=	以上	.if (x >= b)
論理		
&&	論理 AND	.if ((x > 1) && (x < 10))
||	論理 OR	.if ((y != x) || (y < 100))

3.10 特殊な演算子

本アセンブラは、以下の各アクション向けに特殊演算子のセットを提供します。

- 特定セクションのサイズの取得
- 特定セクションの開始アドレスの取得
- 特定セクションの終了アドレスの取得

DD

表 3-6: 特殊演算子

演算子	概要
.sizeof.(*name*)	*name* で指定されたセクションのサイズをアドレス単位で取得します。
.startof.(*name*)	*name* で指定されたセクションの開始アドレスを取得します。
.endof.(name)	*name* で指定されたセクションの終了アドレスを取得します。

3.10.1 特定セクションのサイズの取得

.sizeof.(section_name) 演算子は、リンク処理が発生した後の特定セクションのサイズ (バイト数) を取得するために使います。例えば、.data セクションの最終的なサイズは以下を使って特定できます。

```
.word .sizeof.(.data)
```

3.10.2 特定セクションの開始アドレスの取得

.startof.(section_name) 演算子は、リンク処理が発生した後の特定セクションの開始アドレスを取得するために使います。例えば、.data セクションの開始アドレスは以下を使って特定できます。

```
.word .sizeof.(.data)
```

3.10.3 特定セクションの終了アドレスの取得

.endof.(section_name) 演算子は、リンク処理が発生した後の特定セクションの終了アドレスを取得するために使います。例えば、.data セクションの終了アドレスは以下を使って特定できます。

```
.word .sizeof.(.data)
```

NOTE:

第 4 章　アセンブラ ディレクティブ

4.1　はじめに

ディレクティブはソースコード内に記述されるアセンブラ コマンドですが、通常はオペコードへ直接変換されません。これらは入力、出力、データ割り当て等、アセンブラの制御に使います。

> **Note:** アセンブラ ディレクティブはターゲット命令 (ADD、XOR、JAL 等) では**ありません**。命令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。

PIC24 MCU および dsPIC DSC 向け 16 ビット MPLAB アセンブラ (xc16-as) がサポートするディレクティブとは多くの点で非常に似ていますが、32 ビット MPLAB XC32 アセンブラ (xc32-as) がサポートするディレクティブとは多くの点で異なります。

第 4 章の主な内容は以下の通りです。

- セクションを定義するディレクティブ
- 定数を初期化するディレクティブ
- シンボルを宣言するディレクティブ
- シンボルを定義するディレクティブ
- セクション アラインメントを変更するディレクティブ
- 出力リスティングをフォーマットするディレクティブ
- 条件付きアセンブリを制御するディレクティブ
- 代入 / 展開用ディレクティブ
- ファイルをインクルードするディレクティブ
- 診断出力を制御するディレクティブ
- デバッグ情報用のディレクティブ
- コード生成を制御するディレクティブ

4.2 セクションを定義するディレクティブ

セクションは、配置可能なコードまたはデータのブロックです。これらのブロックは、32 ビット デバイスメモリ内の連続した領域を使います。3 つのセクションが定義済みです。.text は実行コード、.data は初期化データ、.bss は非初期化データ用のセクションです。これ以外のセクションは定義可能です。リンカは 32 ビットメモリの特定領域内にデータ配置するための各種セクションを定義します。

セクション ディレクティブには以下があります。

- .bss
- .data
- .pushsection *name*
- .popsection
- .section name [, “flags”] (deprecated)
- .section name [, attr1[,…,attrn]]
- .text

.bss

定義

後続の命令文を .bss (非初期化データ) セクションの最後へアセンブルします。

bss セクションは、ローカル共有変数を保存するために使います。bss セクション内にアドレス空間を割り当てる事は可能ですが、プログラムを実行する前に bss セクションにロードするデータを指定する事はできません。プログラムの実行が始まると、bss セクション内の全てのバイトは 0 に設定されます。

.bss ディレクティブを使って bss セクションへ切り換えてから通常のようにシンボルを定義します。bss セクションには 0 値だけがアセンブルできます。通常、このセクションはシンボル定義と .skip ディレクティブだけを含みます。

例

```
   # The following symbols (B1 and B2) will be placed in
   # the uninitialized data section.
.bss
B1:.space 4   # 4 bytes reserved for B1
B2:.space 1   # 1 byte reserved for B2
```

.data

定義

後続の命令文を .data (初期化データ) セクションの最後へアセンブルします。

例

```
   # The following symbols (D1 and D2) will be placed in
   # the initialized data section.
.data
D1:.long 0x12345678   # 4 bytes
D2:.byte 0xFF         # 1 byte
```

リンカは、.data セクション (および data 属性で定義されたその他のセクション) のための初期値を収集し、データ初期化テンプレートを生成します。アプリケーションのスタートアップ中にこのテンプレートを処理する事で、初期値をメモリへ転送できます。C アプリケーションの場合、この目的のためにライブラリ関数が自動的に呼び出されます。アセンブリ プロジェクトは、libpic32 ライブラリとリンクする事により、このライブラリを利用できます。詳細は **9.5.3「ランタイム ライブラリのサポート」** を参照してください。

.pushsection *name*

このディレクティブは､現在のセクションをセクション スタックのトップにプッシュした後に、現在のセクションを *name* という名前のセクションで置き換えます。全ての .pushsection には対応する .popsection が必要です。

.popsection

現在のセクションの記述をセクション スタックのトップ セクションで置き換えます。このセクションはスタックからポップオフされます。

.section name [, “flags”] (deprecated)
.section name [, attr1[,...,attrn]]

後続のコードを name という名前のセクションへアセンブルします。名前に文字「*」が指定された場合、アセンブラは入力ファイル名に基づいてそのセクション向けに一意の名前を filename.s.scnn という書式で生成します (n = 自動生成されるセクション名の番号)。

名前に「*」が指定されたセクションを使うとメモリを節約できます。なぜなら、アセンブラはこれらのセクションにアラインメントのためのパディングを追加しないからです。名前に「*」が指定されなかったセクションは、複数のファイルにまたがって結合される可能性があります。このためアセンブラは、要求されるアラインメントを保証するためにパディングを追加する必要があります。

オプションの引数が存在しない場合、セクション属性はセクション名によって決まります。暗黙の属性を持つ予約済みセクション名のテーブルは「Reserved Section Names with Implied Attributes」内で与えられます。セクション名が予約済みの名前と一致する場合、そのセクションには暗黙の属性が割り当てられます。セクション名が予約済みの名前として認識されない場合の既定値属性は data (データメモリ内で初期化される記憶域) です。

.text、.data、.bss 以外の予約済みセクション名向けの暗黙の属性は非推奨です。

これらの予約済みセクション向けの暗黙の属性が使われると、警告が発行されます。

最初のオプション引数を一重引用符で囲んだ場合、それはセクション属性を記述する 1 つまたは複数のフラグとして扱われます。引用符付きのセクションフラグは非推奨です (**補遺 A.「非推奨の機能」**参照)。引用符付きセクションフラグが使われると、警告が発行されます。

最初のオプション引数に引用符が付かない場合、その引数は属性リスト内の最初のエレメントとして扱われます。属性は任意の順番で指定でき、大文字と小文字は区別されます。セクション属性には 2 種類あります。1 つはセクションのタイプを表す属性であり、もう 1 つはセクションのタイプを変更する属性です。

4.2.1 セクションのタイプを表す属性

これらの属性は相互排他的です。1 つのセクションに対して以下の属性のいずれか 1 つだけを与える事ができます。

表 4-1: セクションのタイプを表す属性

属性	概要
code	プログラムメモリ内の実行可能コード
data	データメモリ内の初期化ストレージ
bss	データメモリ内の非初期化ストレージ
persist	データメモリ内の永続的ストレージ
ramfunc	データメモリ内の関数

4.2.2 セクションのタイプを変更する属性

表 4-2 に示す属性は、全てまたは一部のセクションタイプを変更します。

表 4-2: セクションのタイプを変更する属性

属性	概要	属性の適用先				
		code	data	bss	persist	ramfunc
address(a)	絶対アドレス a に配置	x	x	x	x	
near	メモリの先頭の 64k 内に配置		x	x	x	
reverse	終了アドレス +1 に整列		x	x	x	
align(n)	開始アドレスに整列	x	x	x	x	x
noload	割り当てのみ (ロードしない)	x	x	x	x	x
keep	ガベージ コレクションに対してセクションを保持	x	x	x	x	x

4.2.3 セクションのタイプを変更する属性の組み合わせ

表 4-3: セクションのタイプを変更する属性の組み合わせ

	address	near	reverse	align	noload	keep
address		x	x		x	x
near	x		x	x	x	x
reverse		x			x	x
align	x	x			x	x
noload	x	x	x	x		x
keep	x	x	x	x	x	

ユーザ アプリケーションには以下のセクション名が使えます。

表 4-4: 予約済みセクション名

セクション名	生成元	リンカスクリプト内のマップ先	暗黙の属性
.text	コンパイラまたはアセンブラによって生成された命令		code
.text.*	-ffunction-sections を使ってコンパイルした場合、関数はこの書式の一意の名前が付けられたセクションへ出力されます。		code
.startup	C スタートアップ コードは下位互換性のためにリンカスクリプト内に残されます。	kseg0_boot_mem	code
.app_excpt	一般例外ハンドラ	kseg0_boot_mem	code
.reset	リセットハンドラ	kseg0_boot_mem	code
.bev_excpt	BEV 例外ハンドラ	kseg0_boot_mem	code
.vector_n	割り込みベクタ n	kseg0_boot_mem	code
.rodata	const を宣言された文字列と C データ		code
.rodata.*	-fdata-sections を使ってコンパイルした場合、定数データはこの書式の一意の名前が付けられたセクションへ出力されます。		code
.data	初期値を持つ n バイトより大きな変数 (-Gn でコンパイル)		data
.data.*	大きな (large) 初期化変数 (-fdata-sections でコンパイル)		data

表 4-4: 予約済みセクション名 (続 き)

.ramfunc	RAM 関数		data
.bss	非初期化データ		bss
.lit4 / .lit8	アセンブラが命令ストリーム内ではなくメモリ内に保存すると決定した定数 (通常は浮動小数点数) GP 相対アドレス指定向けに使われます。		data
.sdata	初期値を持つサイズが n バイト以下の変数 (-Gn でコンパイル) GP 相対アドレス指定向けに使われます。		data
.sdata.*	小さな (small) 変数 (-fdata-sections でコンパイル) GP 相対アドレス指定向けに使われます。		data
.sbss	サイズが n バイト以下の非初期化変数 (-Gn でコンパイル) GP 相対アドレス指定向けに使われます。		data
.sbss.*	小さな (small) 非初期化変数 (-fdata-sections でコンパイル) GP 相対アドレス指定向けに使われます。		data
.bss	大きな (large) 非初期化変数		data
.bss.*	非初期化変数 (-fdata-sections でコンパイル)		data
.heap	動的メモリに使われるヒープ		data
.stack	スタック向けに予約される最小空間		data
.debug*	DWARF デバッグ情報		info
.line	DWARF デバッグ情報		info
.comment	#ident/.ident 文字列		info
.reginfo	情報セクション		info

セクション ディレクティブの例

```
.section foo;foo is initialized data memory.
.section bar,bss,align(256) ;bar is uninitialized data memory, aligned.
.section *,data,near ;section is near initialized data memory.
.section buf1,bss,address(0xa0000800);buf1 is uninitialized data memory at 0xa0000800.
.section *, code;section is in program memory
```

.text

定義

後続の命令文を .text (実行コード) セクションの最後へアセンブルします。

例

```
            .text
            .ent _main_entry
_main_entry:
            jal main
            nop
            jal exit
            nop
1:
            b  1b
            nop
            .end _main_entry
```

4.3 定数を初期化するディレクティブ

定数初期化ディレクティブには以下があります。

- .ascii “*string*$_1$” [, ..., “*string*$_n$”]
- .asciz “*string*$_1$” [, ..., “*string*$_n$”]
- .byte *expr*$_1$[, ..., *expr*$_n$]
- .double *value*$_1$[, ..., *value*$_n$]
- .float *value*$_1$[, ..., *value*$_n$]
- .single *value*$_1$[, ..., *value*$_n$]
- .hword *expr*$_1$[, ..., *expr*$_n$]
- .int *expr*$_1$[, ..., *expr*$_n$]
- .long *expr*$_1$[, ..., *expr*$_n$]
- .short *expr*$_1$[, ..., *expr*$_n$]
- .string “*str*”
- .word *expr*$_1$[, ..., *expr*$_n$]

.ascii “*string*$_1$” [, ..., “*string*$_n$”]

.ascii では、0 個または 1 個以上の文字列リテラル (複数個ある場合はコンマで区切る) を指定します。これは各文字列を (後にゼロバイトを自動的に付加せずに) 連続したアドレスへアセンブルします。

.asciz “*string*$_1$” [, ..., “*string*$_n$”]

.asciz は、各文字列の後に 1 つのゼロバイトが続くという点を除けば .ascii と同じです。.asciz の「z」は「ゼロ」を意味します。このディレクティブは .string と同義です。

.byte *expr*$_1$[, ..., *expr*$_n$]

.byte では、0 個または 1 個以上の式 (複数個ある場合はコンマで区切る) を指定します。各式は、現在のセクション内の次のバイトへアセンブルされます。

.double *value*$_1$[, ..., *value*$_n$]

1 つまたは複数の倍精度 (64 ビット) 浮動小数点定数をリトルエンディアン形式で連続したアドレスへアセンブルします。浮動小数点数は IEEE 形式です (**3.4.1.2「浮動小数点数」**参照)。

以下の式文は等価です。

```
.double 12345.67
.double 1.234567e4
.double 1.234567e04
.double 1.234567e+04
.double 1.234567E4
.double 1.234567E04
.double 1.234567E+04
```

浮動小数点定数の 16 進エンコーディングを指定する事もできます。以下の式文は等価であり、どの式文も値 12345.67 を 64 ビット倍精度数としてエンコードします。

```
.double 0e:40C81CD5C28F5C29
.double 0f:40C81CD5C28F5C29
.double 0d:40C81CD5C28F5C29
```

.float *value*$_1$[, ..., *value*$_n$]

1 つまたは複数の単精度 (32 ビット) 浮動小数点定数をリトルエンディアン形式で連続したアドレスへアセンブルします。これは .single と同じ働きをします。浮動小数点数は IEEE 形式です (**3.4.1.2「浮動小数点数」**参照)。

以下の式文は等価です。

```
.float 12345.67
.float 1.234567e4
.float 1.234567e04
.float 1.234567e+04
.float 1.234567E4
.float 1.234567E04
.float 1.234567E+04
```

浮動小数点定数の 16 進エンコーディングを指定する事もできます。以下の式文は等価であり、どの式文も値 12345.67 を 32 ビット倍精度数としてエンコードします。

```
.float 0e:4640E6AE
.float 0f:4640E6AE
.float 0d:4640E6AE
```

.single *value*$_1$[, ..., *value*$_n$]

1 つまたは複数の単精度 (32 ビット) 浮動小数点定数を、リトルエンディアン形式で連続したアドレスへアセンブルします。このディレクティブは .float と同義です。浮動小数点数は IEEE 形式です (**3.4.1.2「浮動小数点数」**参照)。

.hword *expr*$_1$[, ..., *expr*$_n$]

1 つまたは複数の 2 バイト値を、リトルエンディアン形式で連続したアドレスへアセンブルします。このディレクティブは .short と同義です。

.int *expr*$_1$[, ..., *expr*$_n$]

1 つまたは複数の 4 バイト値を、リトルエンディアン形式で連続したアドレスへアセンブルします。このディレクティブは .long と同義です。

.long *expr*$_1$[, ..., *expr*$_n$]

1 つまたは複数の 4 バイト値を、リトルエンディアン形式で連続したアドレスへアセンブルします。このディレクティブは .int と同義です。

.short *expr*$_1$[, ..., *expr*$_n$]

1 つまたは複数の 2 バイト値を、リトルエンディアン形式で連続したアドレスへアセンブルします。このディレクティブは .hword と同義です。

.string *“str”*

このディレクティブは .asciz と同義です。

.word *expr*$_1$[, ..., *expr*$_n$]

1 つまたは複数の 4 バイト値を、リトルエンディアン形式で連続したアドレスへアセンブルします。

4.4 シンボルを宣言するディレクティブ

シンボル宣言ディレクティブには以下があります。

- .comm *symbol*, *length* [, *algn*]
- .extern *symbol*
- .global *symbol* .globl *symbol*
- .lcomm *symbol*, *length*
- .weak *symbol*

.comm *symbol*, *length* [, *algn*]

.comm は、*symbol* によって指定された共有シンボルを宣言します。リンク時に、1 つのオブジェクトファイル内の共有シンボルは、別のオブジェクト ファイル内の同一名を持つ定義されたシンボルまたは共有シンボルとマージされる可能性があります。リンカがそのシンボルの定義を検出しなかった (1 つまたは複数の共有シンボルだけ検出した) 場合、リンカは非初期化メモリの *length* バイトを割り当てます。*length* は絶対式である事が必要です。リンカが同一名の共有シンボルを複数検出し、それらが全て同一サイズではない場合、リンカは最大のサイズを使って空間を割り当てます。

.comm ディレクティブにはオプションの第 3 の引数 (*algn*) があります。*algn* は、シンボルのアラインメント (何バイト境界に整列させるか) を指定します。例えば 16 を指定した場合、アドレスの最下位 4 ビットは 0 である事が必要です。アラインメント値は絶対式である事と 2 のべき乗値である事が必要です。リンカが共有シンボルに対して非初期化メモリを割り当てる場合、リンカはシンボルを配置する際にこのアラインメントを使います。この引数が指定されていない場合、アセンブラはアラインメントをシンボルのサイズ以下となる最大の 2 のべき乗値 (最大 1) に設定します。

.extern *symbol*

.extern ディレクティブは、現在のモジュール内で使用可能な (しかし他のモジュール内でグローバルとして定義されている) シンボル名を宣言します。全てのシンボルは既定値により extern であるため、このディレクティブは必ずしも指定する必要はありません。

.global *symbol*
.globl *symbol*

.global ディレクティブは、現在のモジュール内で定義されかつ他のモジュール内でも利用可能なシンボルを宣言します。.global により、そのシンボルはリンカから可視となります。プログラムの一部でシンボルを定義すると、その値は一緒にリンクされるプログラムの他の部分でも利用可能となります。シンボルを定義しない場合、そのシンボルの属性はそのプログラムにリンクされる別のファイル内の同名のシンボルから採用されます。

他のアセンブラとの互換性を維持するため、2 通りのスペル (.globl と .global) が使えます。

.lcomm *symbol*, *length*

symbol によって指定されたローカル共有シンボル向けに *length* バイトを予約します。シンボルのセクションと値は、新しいローカル共有シンボルのセクションと値です。アドレスは .bss セクション内に配置されるため、それらのバイトは実行時に 0 に初期設定されます。*symbol* はグローバルに宣言されないため、通常はリンカから不可視です。

.weak *symbol*

symbol によって指定されたシンボルを weak にします。weak 定義のシンボルが通常定義のシンボルとリンクされた場合、通常定義のシンボルがエラーなしで使われます。weak 定義シンボルがリンクされ、そのシンボルが未定義である場合、weak シンボルはエラーなしで 0 になります。

4.5 シンボルを定義するディレクティブ

シンボル定義ディレクティブには以下があります。

- .equ *symbol*, *expression*
- .equiv *symbol*, *expression*

.equ *symbol, expression*

このディレクティブは *symbol* の値を *expression* に設定します。シンボルはアセンブリ内で何度でも設定できます。グローバル シンボルを設定した場合、そのシンボルに対して最後に設定された値がオブジェクト ファイル内に保存されます。

.equiv *symbol, expression*

.equ と似ていますが、シンボルが既に定義済みである場合にアセンブラがエラーを出力するという点で異なります。参照されたシンボルが実際には定義されていない場合、そのシンボルは未定義であると見なされます。

エラー メッセージの内容は異なりますが、このディレクティブは以下とほぼ等価です。

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

4.6 セクション アラインメントを変更するディレクティブ

セクション アラインメントを明示的に変更するディレクティブには以下があります。

> **Note:** ユーザコードは、セクション アラインメントまたはロケーション カウンタを変更するディレクティブの後で命令を正しくアラインメントするよう配慮する必要があります。

- .align [*algn*[, *fill*]]
- .fill *repeat*[, *size*[, *value*]]
- .org *new-lc*[, *fill*]
- .skip *size*[, *fill*]
- .space *size*[, *fill*]
- .struct *expression*

.align [*algn*[, *fill*]]

.align ディレクティブは、現在のサブセクション内のロケーション カウンタを特定の記憶域境界へパディングします。最初の式 (絶対式である事が必要) は必要なアラインメントを指定します。これは、カウンタを進めた後にロケーション カウンタが持つ必要のある下位ゼロビットの数です。

アセンブラは 0 ～ 15 の *algn* 値を許容します。.align 0 は、擬似命令を生成するデータによって使われる自動的なアラインメントを OFF にします。ユーザは、データが確実に正しくアラインメントされる事を確認する必要があります。.align 擬似命令により、自動アラインメントを復帰させます。

2 番目の式 (これも絶対式である事) は、パディング バイトに格納する *fill* 値を指定します。この値 (およびコンマ) は省略しても構いません。省略した場合、パディング バイトは既定値により 0 となります。メモリのフラッシュ領域向けに 0xFF を使う事ができます。

.fill *repeat*[, *size*[, *value*]]

size をバイト数、繰り返し回数を *repeat* で指定してフィルします。*repeat* は 0 または 0 より大きな値です。*size* も 0 または 0 より大きな値ですが、8 より大きい場合は 8 であると見なされます。各 *repeat* のバイトの内容は 8 バイト値から取られます。最上位の 4 バイトは 0 です。最下位の 4 バイトはリトルエンディアンのバイト順で表された値です。　1 回の反復で各 *size* バイトがこの値の最下位 *size* バイトから取られます。

size は指定しなくても構いません。最初のコンマと後続のトークンが存在しない場合、*size* は 1 であると見なされます。

value は指定しなくても構いません。2 つ目のコンマと *value* が存在しない場合、*value* は 0 であると見なされます。

例 :

```
          .text
          .fill 0x3, 1, 0xFF
          .align 2
mylabel:b .
```

.org *new-lc*[, *fill*]

.orgディレクティブは、現在のセクションのロケーション カウンタを*new-lc*へ進めます。*new-lc* は絶対式または、現在のサブセクションと同じセクションを使った式のどちらかです。つまり、.org を使って複数のセクションを横切る事はできません。*new-lc* が不正なセクションを持つ場合、.org ディレクティブは無視されます。*new-lc* のセクションが絶対式である場合、xc32-as は警告を出力し、*new-lc* のセクションは現在のサブセクションと同じであるかのように振る舞います。

.org は、ロケーション カウンタを進めるか、そのまま変更しないかのどちらかしかできません。.org を使ってロケーション カウンタを後方へ動かす事はできません。

アセンブラは、プログラムを 1 つのパス内でアセンブルしようと試みるため、*new-lc* は未定義であってはなりません。

起点はセクションの開始位置 (サブセクションの開始位置ではない) に対して相対的であるという事に注意が必要です。

現在のサブセクションのロケーション カウンタが進められた時、中間にあるバイトには *fill* が書き込まれます (これは絶対式である事が必要です)。コンマと *fill* が省略された場合、fill は既定値により 0 となります。

.skip *size*[, *fill*]

.space *size*[, *fill*]

これらのディレクティブは *size* バイト (各バイトの値は *fill*) を出力します。*size* と *fill* はどちらも絶対式です。コンマと *fill* が省略された場合、*fill* は 0 であると見なされます。

.struct *expression*

絶対セクションへ切り換え、セクション オフセットを *expression* (絶対式である事が必要) に設定します。これは、以下のように使えます。

```
          .struct 0
field1:
          .struct field1 + 4
field2:
          .struct field2 + 4
field3:
```

これは、シンボル field1 が値 0、シンボル field2 が値 4、シンボル field3 が値 8 を持つよう定義します。アセンブリは絶対セクション内に残されます。ユーザは後続のアセンブリを行う前に何らかの .section ディレクティブを使って他のセクションへ変更する必要があります。

4.7 出力リスティングをフォーマットするディレクティブ

出力リスティングをフォーマットするディレクティブには以下があります。

- .eject
- .list
- .nolist
- .psize *lines*[, *columns*]
- .sbttl “*subheading*”
- .title “*heading*”

.eject

アセンブリ リスティングの生成中に、このディレクティブの位置でページをブレークします。

.list

.nolist との組み合わせにより、アセンブリ リスティングを生成するかどうか制御します。このディレクティブは内部カウンタ (初期値は 1) をインクリメントします。アセンブリ リスティングは、このカウンタが 0 より大きい場合に生成されます。

このディレクティブは、-a コマンドライン オプションによってリスティングが有効にされ、かつ、-an コマンドライン オプションによって形態処理が無効にされていない場合にのみ機能します。

.nolist

.list との組み合わせにより、アセンブリ リスティングを生成するかどうか制御します。このディレクティブは内部カウンタ (初期値は 1) をデクリメントします。アセンブリ リスティングは、このカウンタが 0 より大きい場合に生成されます。

このディレクティブは、-a コマンドライン オプションによってリスティングが有効にされ、かつ、-an コマンドライン オプションによって形態処理が無効にされていない場合にのみ機能します。

.psize *lines*[, *columns*]

リスティングを生成する際の各ページの行数と列数 (任意) を宣言します。

.psize を使わない場合、リスティングは既定値の行数 (60) を使います。コンマと *columns* の指定は省略できます。既定値幅は 200 です。

アセンブラは、指定された行数を超えるたびに (またはユーザが .eject を使って明示的に要求するたびに) フォームフィードを生成します。

lines に対して 0 を指定した場合、.eject による明示的な要求がない限りフォームフィードは生成されません。

.sbttl “*subheading*”

アセンブリ リスティングを生成する際に、*subheading* をサブタイトル (3 行目、タイトル行の直後) として使います。このディレクティブが先頭から 10 行目以内に現れた場合、後続ページと現在のページに影響します。

.title “*heading*”

アセンブリ リスティングを生成する際に、*heading* をタイトル (2 行目、ソースファイル名とページ番号の直後) として使います。

4.8 条件付きアセンブリを制御するディレクティブ

条件付きアセンブリ ディレクティブには以下があります。

- .else
- .elseif *expr*
- .endif
- .if *expr*

.else

.if ディレクティブと組み合わせて使う事で、.if が「偽」と評価された場合にアセンブリ コードの代替パスを提供します。

.elseif *expr*

.if ディレクティブと組み合わせて使う事で、.if が「偽」と評価された場合に、第 2 の条件によるアセンブリ コードの代替パスを提供します。

.endif

条件付きでのみアセンブルされるコードのブロックの終端を示します。

.if *expr*

引数 *expr* が非 0 の場合にのみアセンブルされるソースプログラム内のコードセクションの先頭を示します。コードの条件付きセクションの終端は .endif によって示す必要があります。必要に応じ、.else を使って代替条件向けのコードを含める事ができます。

アセンブラは、.if の以下の派生形もサポートします。

.ifdecl *symbol*

指定したシンボルが定義されていた場合にコードの後続セクションをアセンブルします。まだ定義されていないのに参照されたシンボルは未定義であると見なされます。

.ifc *string1,string2*

このディレクティブは、2 つの文字列が同じである場合にコードの後続セクションをアセンブルします。文字列は、一重引用符で囲んでも囲まなくても構いません。引用符で囲まない場合、最初の文字列は最初のコンマで終わり、2 番目の文字列は行末で終わります。空白類文字を含む文字列は引用符で囲む必要があります。この文字列比較は大文字と小文字を区別します。

.ifeq *absolute-expression*

このディレクティブは、引数が 0 である場合にコードの後続セクションをアセンブルします。

.ifeqs *string1,string2*

このディレクティブは .ifc の別形態です。文字列は二重引用符で囲む必要があります。

.ifge *absolute-expression*

このディレクティブは、引数が 0 以上である場合にコードの後続セクションをアセンブルします。

.ifgt *absolute-expression*

このディレクティブは、引数が 0 より大きい場合にコードの後続セクションをアセンブルします。

.ifle *absolute-expression*

このディレクティブは、引数が 0 以下である場合にコードの後続セクションをアセンブルします。

.iflt *absolute-expression*

このディレクティブは、引数が 0 より小さい場合にコードの後続セクションをアセンブルします。

.ifnc *string1,string2*

このディレクティブは .ifc に似ていますが、評価方法が逆です。つまり、2 つの文字列が同じではない場合にコードの後続セクションをアセンブルします。

.ifndef *symbol*

このディレクティブは、指定したシンボルが未定義である場合にコードの後続セクションをアセンブルします。まだ定義されていないのに参照されたシンボルは未定義であると見なされます。

.ifnotdef *symbol*

このディレクティブは .ifndef と同じです。

.ifne *absolute-expression*

このディレクティブは、引数が 0 ではない場合にコードの後続セクションをアセンブルします (.if と等価です)。

.ifnes *string1,string2*

このディレクティブは .ifeqs に似ていますが、評価方法が逆です。つまり、2 つの文字列が同じではない場合にコードの後続セクションをアセンブルします。

4.9 代入 /展 開用ディレクティブ

代入 / 展開ディレクティブには以下があります。

- .exitm
- .irp *symbol*, *value*$_1$ [, ..., *value*$_n$] ... .endr
- .irpc *symbol*, *value* ... .endr
- .macro

.exitm

現在のマクロ定義から早期退出します。.macro ディレクティブを参照してください。

.irp *symbol*, *value*$_1$ [, ..., *value*$_n$] ... .endr

symbol に一連の異なる値を代入する命令文のシーケンスを評価します。命令文のシーケンスは .irp ディレクティブで始まり、.endr ディレクティブで終わります。各値に対し、*symbol* が *value* に設定され、命令文のシーケンスがアセンブルされます。値が列記されない場合、命令文のシーケンスは1度だけアセンブルされ、*symbol* は NULL 文字列に設定されます。命令文のシーケンス内で *symbol* を参照するには、\symbol を使います。

以下に例を示します。

```
.irp reg,0,1,2,3
lw $a\reg, 1032+\reg($sp)
.endr
```

is equivalent to assembling

```
lw $a0,1032+0($sp)
lw $a1,1032+1($sp)
lw $a2,1032+2($sp)
lw $a3,1032+3($sp)
```

.irpc *symbol*, *value* ... .endr

symbol に一連の異なる値を代入する命令文のシーケンスを評価します。命令文のシーケンスは .irpc ディレクティブで始まり、.endr ディレクティブで終わります。*value* 内の各文字に対し、*symbol* はその文字に設定され、命令文のシーケンスがアセンブルされます。値が列記されない場合、命令文のシーケンスは 1 度だけアセンブルされ、*symbol* は NULL 文字列に設定されます。命令文のシーケンス内で *symbol* を参照するには、\symbol を使います。

以下に例を示します。

```
.irpc reg,0123
lw $a\reg, 1032+\reg($sp)
.endr
```

is equivalent to assembling

```
lw $a0,1032+0($sp)
lw $a1,1032+1($sp)
lw $a2,1032+2($sp)
lw $a3,1032+3($sp)
```

.macro

ディレクティブ .macro および .endm を使うと、アセンブリ出力を生成するためのマクロを定義できます。例えば以下の定義は、一連の値をメモリに格納する SUM という名前のマクロを指定します。

```
.macro SUM from=0, to=5
.long \from
.if    \+o-\from
SUM "(\from+1)",   \+o
.endif
.endm
```

この定義により、「SUM 0,5」は以下のアセンブリ入力と等価です。

```
.long 0
.long 1
.long 2
.long 3
.long 4
.long 5
```

.macro *macname*

.macro *macname macargs* …

macname という名前のマクロの定義を開始します。マクロ定義に引数が必要である場合、マクロ名の後でコンマまたはスペースで区切って引数の名前を指定します。名前の後に =deflt を付ける事により、任意のマクロ引数に既定値を与える事ができます。以下は全て有効な .macro 命令文です。

- .macro comm
  comm という名前のマクロ (引数なし) の定義を開始します。
- .macro plus1 p, p1
  .macro plus1 p p1
  どちらの命令文も 2 つの引数を取る plus1 という名前のマクロの定義を開始します。マクロ定義の中に \p または \p1 を書く事で引数を評価します。
- .macro reserve_str p1=0 p2
  2 つの引数を持つ reserve_str という名前のマクロの定義を開始します。最初の引数だけ既定値を持ちます。定義が完了した後に、「reserve_str a,b」(\p1 は a、\p2 は b と評価される) または「reserve_str ,b」(\p1 は既定値 (この場合は 0)、\p2 は b と評価される) としてマクロを呼び出す事ができます。

マクロを呼び出す際に、位置またはキーワードによって引数を指定できます。例えば、「SUM 9,17」は「sum to=9, from=17」と等価です。

.endm

マクロ定義の終端を示します。

.exitm

現在のマクロ定義から早期退出します。

\@

アセンブラは、実行したマクロの数を示すカウンタをこの擬似変数内で保持します。\@により、この数を出力にコピーできます (しかし、1 つのマクロ定義内でのみ可能)。以下の例は、再帰的マクロを使って、ラベル付きバッファの任意の番号を割り当てます。

```
       .macro make_buffers num,size
BUF\@:.space \size
       .if (\num - 1)
       make_buffers (\num - 1),\size
       .endif
       .endm

       .bss
       # create BUF0..BUF3, 16 bytes each
       make_buffers 4,16
```

この例のマクロは、以下リスティングに示すように展開します。

```
   6            make_buffers (\num - 1),\size
   7            .endif
   8            .endm
   9
  10            .bss
  11            # create BUF0..BUF3, 16 bytes each
  12            make_buffers 4,16
  12     > BUF0:.space 16
  12 0000 > .space 16
  12     > .if (4-1)
  12     > make_buffers (4-1),16
  12     >> BUF1:.space 16
  12 0010 >> .space 16
  12     >> .if ((4-1)-1)
  12     >> make_buffers ((4-1)-1),16
  12     >>> BUF2:.space 16
  12 0020 >>> .space 16
  12     >>> .if (((4-1)-1)-1)
  12     >>> make_buffers (((4-1)-1)-1),16
  12     >>>> BUF3:.space 16
  12 0030 >>>> .space 16
  12     >>>> .if ((((4-1)-1)-1)-1)
  12     >>>> make_buffers ((((4-1)-1)-1)-1),16
  12     >>>> .endif
  12     >>> .endif
  12     >> .endif
  12     > .endif
```

.purgem "*name*"

マクロ名を未定義にします。その後、その文字列の使用は展開されません。前記の .macro ディレクティブを参照してください。

.rept *count* ....endr

.rept ディレクティブと次の .endr ディレクティブの間の行のシーケンスを *count* 回繰り返します。

以下に例を示します。

```
.rept 3
.long 0
.endr
```

上記のアセンブルは下記のアセンブルと等価です。

```
.long 0
.long 0
.long 0
```

4.10 ファイルをインクルードするディレクティブ

他のファイルからデータをインクルードするディレクティブには以下があります。

- .incbin "*file*"[,*skip*[,*count*]]
- .include "*file*"

.incbin "*file*"[,*skip*[,*count*]]

.incbinディレクティブは、現在の位置で*file*を逐語的にインクルードします。ファイルにはバイナリデータが格納されていると想定します。検索パスは -I コマンドライン オプションを使って指定できます(**第2章「アセンブラ コマンドライン オプション」**参照)。*file* は二重引用符で囲む必要があります。

引数 *skip* は、ファイルの先頭からスキップするバイト数を指定します。引数 *count* は、読み出すバイトの最大数を指定します。データは一切アラインメントされません。従って、ユーザの責任により、.incbin ディレクティブの前後で正しいアライメントが提供されるようにする必要があります。

.include "*file*"

ソースコード内の指定位置でファイルをインクルードする方法を提供します。コードは、.include の位置の後に続くかのようにアセンブルされます。インクルードされるファイルの終端に達すると、元のファイルのアセンブルが .include の次の命令文で再開します。

4.11 診断出力を制御するディレクティブ

その他のディレクティブには以下があります。

- .abort
- .err
- .error "*string*"
- .fail *expression*
- .ident "*comment*"
- .print "*string*"
- .version "*string*"
- .warning "*string*"

.abort

メッセージ「.abort detected.Abandoning ship.」を出力し、プログラムを終了します。

.err

アセンブラは .err ディレクティブを検出するとエラーメッセージを出力し、-Z オプションが指定されていなければオブジェクト ファイルを生成しません。このディレクティブは、条件付きコンパイルされたコード内のエラーを示すために使えます。

.error "*string*"

.err と似ていますが、指定された文字列を出力するという点で異なります。

.fail *expression*

エラーまたは警告を生成します。*expression* の値が 500 以上である場合、xc32-as は警告メッセージを出力します。この値が 500 未満である場合、xc32-as はエラーメッセージを出力します。メッセージは *expression* の値を含みます。これは、複雑にネスティングされたマクロまたは条件付きアセンブリの内部で便利に使える場合があります。

.ident "*comment*"

.comment という名前のセクションにコメントを付加します。このセクションが存在しない場合、セクションが生成されます。リンカは、メモリを割り当てる際にこのセクションを無視しますが、全ての .comment セクションをリンク順に 1 つにまとめます。

.print "*string*"

アセンブル中に標準出力に *string* を出力します。

.version "*string*"

このディレクティブは .note セクションを生成し、その中に ELF 形式の note (タイプは NT_VERSION) を配置します。note の名前は string に設定されます。.version は、出力ファイル形式が ELF である場合にサポートされ、それ以外の場合は無視されます。

.warning "*string*"

ディレクティブ .error に似ていますが、エラーではなく警告を出力します。

4.12 デバッグ情報用のディレクティブ

デバッグ情報ディレクティブには以下があります。

- .ent *function*
- .end
- .file *fileno* "*filename*"
- .fmask *mask*, *offset*
- .frame *framereg*, *frameoffset*, *retreg*
- .loc *fileno*, *lineno* [*columnno*]
- .mask *mask*, *offset*
- .size *name*, *expression*
- .sleb128 $expr_1$ [, ..., $expr_n$]
- .type *name*, *description*
- .uleb128 *expr1*[,...,*exprn*]

.ent *function*

このディレクティブは、汎用的な .type ディレクティブと同様に、function シンボルを関数として指定します

.end

プログラムを終了します。

.file *fileno* "*filename*"

dwarf2 行番号情報を出力する時、.file はファイル名を .debug_line ファイル名テーブルへ割り当てます。*fileno* オペランドは、テーブル内のエントリのインデックスとして使う一意の正の整数である事が必要です。*filename* オペランドは C 文字列リテラルです。

filename インデックスの詳細はユーザに知らされます。なぜなら、ファイル名テーブルは dwarf2 デバッグ情報の .debug_info セクションと共有され、従ってユーザはテーブルエントリの正確なインデックスを知る必要があります。

.fmask *mask*, *offset*

現在の PIC32 MCU では使いません。mask 0x00000000 と offset 0 を維持します。

.frame *framereg*, *frameoffset*, *retreg*

このディレクティブは、スタックフレームの形状を記述します。使用する仮想フレームポインタは *framereg* です。通常、これは $fp または $sp です。フレームポインタは CFA (canonical frame address) から *frameoffset* バイト下にあります。これは、関数へのエントリに対するスタックポインタの値です。戻りアドレスは、.mask の指示に従って保存されるまで、*retreg* に初期配置されます。

.loc *fileno*, *lineno* [*columnno*]

オブジェクト ファイルのデバッグ情報は、アセンブリ命令をソースコードの行と列に関連付ける行番号マトリクスを含みます。.loc ディレクティブは、このディレクティブの直後のアセンブリ命令に対応するマトリクス行を追加します。行が追加される前に *fileno*、*lineno*、*columnno* がデバッグ ステートマシンへ適用されます。

.mask *mask, offset*

現在の関数のスタックフレームにどの整数レジスタが保存されるか示します。*mask* はビットマスクとして解釈されます (ビット n がセットされていればレジスタ n が保存される)。これらのレジスタは、CFA (canonical frame address) から *offset* バイト オフセットした位置にあるブロックに保存されます。これは関数へのエントリに対するスタックポインタの値です。

.size *name, expression*

このディレクティブは、シンボル *name* に割り当てるサイズを設定します。サイズ (バイト数) は、label arithmetic を使用可能な *expression* から計算されます。通常このディレクティブは関数シンボルのサイズを設定するために使います。

.sleb128 $expr_1$ [, ..., $expr_n$]

sleb128 は「signed little endian base 128」を意味します。これは DWARF シンボリック デバッグ フォーマットによって使われる値のコンパクトな可変長表現です。

.type *name, description*

これは、シンボル *name* のタイプを関数シンボルまたはオブジェクト シンボルのどちらかに設定します。他のアセンブラとの互換性を提供するため、タイプ *description* フィールドには 5 通りの構文が使えます。それらの構文は以下の通りです。

```
.type <name>,#function
.type <name>,#object
.type <name>,@function
.type <name>,@object
.type <name>,%function
.type <name>,%object
.type <name>,"function"
.type <name>,"object"
.type <name> STT_FUNCTION
.type <name> STT_OBJECT
```

.uleb128 $expr_1$[,...,$expr_n$]

uleb128 は「unsigned little endian base 128」を意味します。これは DWARF シンボリック デバッグ フォーマットによって使われる値のコンパクトな可変長表現です。

4.13 コード生成を制御するディレクティブ

アセンブラのコード生成を制御するディレクティブには以下があります。

- .set noat
- .set at
- .set noautoextend
- .set autoextend
- .set nomacro
- .set macro
- .set mips16e
- .set nomips16e
- .set noreorder
- .set reorder

.set noat

アドレス書式を合成する場合、アセンブラはスクラッチ レジスタを要求する可能性があります。既定値により、アセンブラは at ($1) レジスタを暗黙のうちに使います。このレジスタは、慣例によりアセンブラ テンポラリとして予約済みです。場合によっては、コンパイラはこのレジスタを使うべきではありません。.set noat ディレクティブは、アセンブラがこのレジスタを暗黙のうちに使う事を防ぎます。

.set at

アセンブラが暗黙のうちに at ($1) レジスタを使う事を許容します。

.set noautoextend

既定値により、MIPS16 命令は必要に応じて自動的に 32 ビットへ拡張されます。ディレクティブ .set noautoextend はこれを抑止します。.set noautoextend が機能すると、全ての 32 ビット命令は .e 修飾子を使って明示的に拡張する必要があります (例 : li.e $4,1000)。ディレクティブ .set autoextend を使う事で、必要な時に命令の自動拡張を再度有効にする事ができます。

.set autoextend

MIPS16 命令の 32 ビットへの自動的な拡張を有効にします。

.set nomacro

本アセンブラは合成された命令 (複数のマシン命令にアセンブルされる命令ニーモニック) をサポートします。例えば、sleu 命令は sltu 命令と xori 命令へアセンブルされます。.set nomacro ディレクティブを使うと、アセンブラは 1 つの命令が複数のマシン命令へ展開された時に警告メッセージを発行します。

.set macro

合成された命令に関する警告を抑止します。

.set mips16e

MIPS16e ISA 拡張を使ってアセンブルします。

.set nomips16e

MIPS16e ISA 拡張を使ってアセンブルしません。

.set noreorder

既定値により、本アセンブラは分岐または遅延スロットを、その前後の命令の順序を変更する事によって埋めようと試みます。この機能は非常に便利です。

しかし場合によっては、命令の順序をユーザが細かく制御する必要があります。そのような場合に .set noreorder ディレクティブを使うと、この機能を抑制するようアセンブラに指示する事ができます。この機能は .set reorder ディレクティブによって再度有効にできます。

.set reorder

分岐または遅延スロットを埋めるためにアセンブラが命令の順序を変更する事を許容します。

.set micromips

microMIPS ISA モードでアセンブルします。

.set nomicromips

microMIPS ISA モードでアセンブルしません。

第 5 章　アセンブラのエラー / 警告 / メッセージ

5.1　はじめに

PIC32 MCU 向け MPLAB アセンブラ (xc32-as) はエラー、警告、メッセージを生成します。本アセンブラが生成する最も一般的な診断メッセージの説明付きの一覧を以下に記載します。

第 5 章の主な内容は以下の通りです。

- 致命的エラー
- エラー
- 警告
- メッセージ

5.2 致命的エラー

以下のエラーは、アセンブラ内で内部エラーが発生した事を示します。以下の致命的エラーのいずれかをアセンブラが生成した場合、Microchip社(http://support.microchip.com)にサポートをお問い合わせください。その際、エラーが発生したソースコードとコマンドライン オプションの詳細をお知らせください。

- Bad char = '%c'
- Bad defsym; format is --defsym name=value
- Bad return from bfd_install_relocation:%x
- Broken assembler.No assembly attempted.
- Can't allocate elf private section data:%s
- Can't continue
- Can't create group:%s
- Can't extend frag %u chars
- Can't open a bfd on stdout %s
- Can't start writing .mdebug section:%s
- Cannot write to output file
- Could not write .mdebug section:%s
- Dwarf2 is not supported for this object file format
- Emulations not handled in this configuration
- Error constructing %s pseudo-op table:%s
- Expr.c(operand): bad atof_generic return val %d
- Failed sanity check
- Failed to read instruction table %s\n
- Failed to set up debugging information:%s
- Index into stored_fixups[] out of bounds
- Inserting into symbol table failed:%s
- Internal: bad mips opcode (bits 0x%lx undefined):%s %s.
- Internal: bad mips opcode (mask error):%s %s.
- Internal: bad mips opcode (unknown extension operand type `+%c'):%s %s.
- Internal: bad mips opcode (unknown operand type `%c'):%s %s.
- Internal error, line %d, %s
- Internal error: unknown dwarf2 format
- Internal: can't hash `%s':%s
- Invalid abi -mabi=%s
- Invalid listing option `%c'
- Macros nested too deeply
- Missing emulation mode name
- Multiple emulation names specified
- No compiled in support for 64 bit object file
- No object file generated
- Rva not supported
- Rva without symbol
- Too many fixups
- Unrecognized emulation name `%s'

5.3 エラー

以下に示すエラーは、通常アセンブリ ソースコード内またはアセンブラに渡されたコマンドライン オプション内のエラーを示します。

記号

.abort detected.Abandoning ship.

.abort ディレクティブによりユーザエラーが呼び出されました。

.else without matching .if

.else ディレクティブの前に対応する .if ディレクティブがありません。

.elseif after .else

.else ディレクティブの後で .elseif ディレクティブが指定されました。.elseif ディレクティブが .else ディレクティブより前に来るようにコードを変更する必要があります。

.elseif without matching .if

.elseif ディレクティブの前に対応する .if ディレクティブがありません。

.endfunc missing for previous .func

前の .func に対応する .endfunc ディレクティブがありません。

.endif without .if

.endif ディレクティブの前に対応する .if ディレクティブがありません。

.err encountered.

.err ディレクティブによりユーザエラーが呼び出されました。

.Ifeqs syntax error

コンマで区切られ二重引用符で囲まれた 2 つの文字列は .ifeqs ディレクティブへの引数として渡されませんでした。

.Set pop with no .set push

空のオプション スタックのオプションのポップオフが試みられました。.set pop の前に .set push を使う必要があります。

.Size expression too complicated to fix up

.size 式は定数にできます。あるいはラベル除算が使えます。

A

A bignum with underscores may not have more than 8 hex digits in any word.

bignum 定数は、1 ワード内で 8 桁を超える 16 進数を持つ事はできません。

A bignum with underscores must have exactly 4 words.

アンダースコア表記を使った bignum 定数は 4 つの 8 桁 16 進数部分を持つ必要があります。

Absolute sections are not supported.

本アセンブラは絶対セクション コマンドをサポートしません。

Alignment not a power of 2.

アラインメント値は 2 のべき乗値である事が必要です。

Alignment too large: 15.Assumed.

15 より大きなアラインメント値が要求されました。本アセンブラはアラインメント値を自動的に 15 として動作を続けます。

Arg/static registers overlap.

MIPS32 モードの退避 / 復元は、arg と static 向けにオーバーラップしたレジスタを使います。

Argument must be a string.

`.error` または `.warning` ディレクティブへの引数は二重引用符で囲んだ文字列である事が必要です。

Attempt to allocate data in common section.

このディレクティブは、割り当て不可能なセクションへのデータの割り当てを試みます。データは別のセクションに割り当てる必要があります。

Attempt to get value of unresolved symbol *name*

アセンブラは未解決のシンボルの値を取得できませんでした。

Attempt to set value of section symbol.

セクション シンボルへの代入は違反です。

B

Backward ref to unknown label *label*:

参照されたラベルは見つからないか、ここでは定義されていません。

Bad .common segment name

`.comm` シンボルの適切なアラインメント値を特定できませんでした。先に検出された `.comm` シンボルが正しくないかもしれません。

Bad escaped character in string.

文字列は非標準のバックスラッシュ エスケープ文字を使っています。

Bad expression.

式の型が特定できないか、オペランドの型が式の型に対して正しくありません。

Bad floating literal: %s.

トークンは浮動小数点値に変換できませんでした。

Bad floating-point constant: exponent overflow.

指数部オーバーフローのため、トークンは浮動小数点値に変換できませんでした。

Bad floating-point constant: unknown error code=%d.

トークンは浮動小数点値に変換できませんでした。

Bad format for ifc or ifnc.

itc または ifnc ディレクティブに対する引数が正しくありません。それらはコンマで区切られ二重引用符で囲まれた 2 つの文字列である事が必要です。

Bad or irreducible absolute expression.

絶対式が予期せぬ演算子型を持っています。

Bad register expression.

DWARF デバッグ ディレクティブが無効なレジスタ式を持っています。

Bignum invalid.

式内で指定されている bignum 値が無効です。

C

Can't parse register list.

MIPS32 モードにおいてレジスタリストが無効です。

Can't resolve value for symbol `%s'.

アセンブラはシンボルの正しい値を取得できませんでした。

Constant too large.

ベースレジスタからオフセットした定数を符号拡張する際に、定数が大きすぎました。

Could not skip to num in file filename

`.incbin` ディレクティブに対するスキップ パラメータは、指定されたファイルに対して無効でした。

D

Duplicate else.

各 `.if` ディレクティブは、対応する `.else` ディレクティブを 1 つしか持つ事ができません。

E

End of file inside conditional.

条件付き終了ディレクティブが見つかりません。ファイル終端の前に条件付きアセンブリを終了する必要があります。

End of macro inside conditional.

マクロ終了ディレクティブが見つかりません。ファイル終端の前にマクロを終了する必要があります。

Expected address expression.

式が違反しているか、欠落しているか、bignum です。式は定数アドレスである事が必要です。

Expected comma after %s.

このディレクティブの引数はコンマで区切る必要があります。

Expected comma after name `%s' in .size directive.

このディレクティブの引数はコンマで区切る必要があります。

Expected quoted string.

引数は引用符で囲んだ文字列である事が必要です。

Expected simple number.

この引数は単純な番号である事が必要です。

Expected symbol name.

この引数はシンボル名である事が必要です。

Expression out of range.

式はディレクティブまたは命令のレンジ外です (例 : 32 ビット値が期待される時は 32 ビット値が必要です)。

Expression too complex.

式はシンボルまたは定数である事が必要です。

F

File not found: %s.

ディレクティブに対して指定されたファイル (`.incbin` 等) を指定された通りに開く事ができませんでした。

File number %ld already allocated.

`.file` ディレクティブへ渡されたファイル番号は既に使用中です。

File number less than one.

`.file` ディレクティブへ渡すファイル番号は 1 より大きい事が必要です。

Floating point number invalid.

浮動小数点数が無効です。

G

Global symbols not supported in common sections.

MRI common セクションでは外部シンボルはサポートされません。

I

Ignoring attempt to redefine symbol name

`.weakext` によって再定義されているシンボルは既に定義済みです。

Improper insert size

INS 命令に対して指定されたフィールドの幅は、シフト位置に対して無効でした。

Improper extract size

EXT 命令に対して指定されたフィールドの幅は、シフト位置に対して無効でした。

Instruction insn requires absolute expression.

この命令には定数式が必要です。

Invalid astatic register list

MIPS32e 拡張された SAVE/RESTORE 命令の aregs フィールドが無効な astatic レジスタリストを指定しました。

Invalid arg register list.

MIPS32e 拡張された SAVE/RESTORE 命令の aregs フィールドが無効な arg レジスタリストを指定しました。

Invalid coprocessor 0 register number.

この命令に無効なコプロセッサ 0 レジスタ番号が渡されました。

Invalid coprocessor sub-selection value (%ld), not in range 0-7.

コプロセッサ サブセレクション値は 0 ～ 7 のレンジ内である事が必要です。

Invalid frame size

フレームサイズが無効でありエンコードできませんでした。

Invalid identifier for .ifdef.

指定された ID は無効な名前です。名前は適切な文字で始まる必要があります。

Invalid register list.

MIPS32 モードにおいてレジスタリストが無効なレジスタを含んでいました。

Invalid segment %s.

無効なセグメント内でロケーション カウンタの変更が試みられました。

Invalid static register list.

static レジスタリストは $s2 ～ $s8 のみを含む事が必要です。

J

Jump to misaligned address (0x%lx).

ジャンプ先のアドレスがアラインメントされていません。

Junk at end of line, first unrecognized character is char

期待された入力の後に余計な文字があります。

Junk at end of line, first unrecognized character valued 0xval

期待された入力の後に余計な文字があります。

L

Load/store address overflow (max 32 bits).

ロード / ストア アドレスの幅が 32 ビットを超えています。ラベルが正しいかどうか確認してください。

Local label is not defined.

参照されたローカルラベルは未定義です。

Lui expression not in range 0..65535.

LUI (Load Upper Immediate) 式は 32 ビットレンジ内である事が必要です。

N

New line in title.

タイトル ヘッディング文字列は二重引用符で囲む必要があります。

No such section.

`.global`ディレクティブ内で指定されたセクション名は存在しません(例: `.global foo .myscn`)。

Non-constant expression in .elseif statement

`.elseif`命令文は定数expr式を要求します。`.elseif`ディレクティブの引数は、ディレクティブの最初のパスで解決可能な定数値である事が必要です。この引数内で使われているシンボルの .equ がディレクティブより前に置かれているかどうか確認してください。詳細は **4.8「条件付きアセンブリを制御するディレクティブ」**を参照してください。

Non-constant expression in .if statement.

.if 命令文には定数 expr 式が必要です。.if ディレクティブの引数は、ディレクティブの最初のパスで解決可能な定数値である事が必要です。この引数内で使われているシンボルの .equ がディレクティブより前に置かれているかどうか確認してください。詳細は**4.8「条件付きアセンブリを制御するディレクティブ」**を参照してください。

`Noreorder' must be set before `nomacro'.

nomacro を設定する前に noreorder を設定する必要があります。

Number (0x%lx) larger than 32 bits.

32 ビット幅より大きな値のレジスタへのロード

Number larger than 64 bits.

64 ビット幅より大きな値の HI/LO レジスタへのロード

O

Offset too large.

オフセットは符号拡張した 32 ビットレンジ内である事が必要です。

Opcode not supported on this processor.

その命令オペコードは PIC32 MCU でサポートされません。

Operand overflow.

オペランドがこの命令の許容レンジ内ではありません。

Operation combines symbols in different segments.

式の左辺と右辺が異なるセクション内に配置されています。本アセンブラはこの式を処理できません。

R

Register value used as expression.

式の演算子が有効な演算子ではなくレジスタです。

Relocation reloc isn't supported by the current ABI.

この再配置は PIC32 リトルエンディアン ELF 出力フォーマットでサポートされません。

S

Seek to end of .incbin file failed `%s'.

.incbin によって指定されたファイルの終端が見つかりません。

Skip (%ld) + count (%ld) larger than file size (%ld).

.incbin スキップ値 + カウント値がファイルのサイズを超えています。

Store insn found in delay slot of noreorder code.

store 命令を分岐の前に移動し、遅延中に NOP を使ってください。

Symbol `%s' can not be both weak and common.

.weak ディレクティブと .comm ディレクティブの両方が同じソースファイル内の同じシンボルに対して使われました。

Symbol name is already defined.

このシンボルは再定義できません。

Symbol definition loop encountered at `%s'.

自己参照ループに遭遇したため、そのシンボルは定義できませんでした。シンボルの定義はシンボル自身の値に依存できません。

Syntax error in .startof.Or .sizeof.

アセンブラは `.startof.` または `.sizeof.` を見つけましたが、始まりカッコ「(」または終わりカッコ「)」が見つかりません。

T

This string may not contain '\0'.

この文字列は有効な C 文字列である事が必要です。文字列に「`\0`」を含める事はできません。

Treating warnings as errors.

`--fatal-warnings` コマンドライン オプションにより、アセンブラは全ての警告をエラーとして扱うよう指示されています。

U

Unassigned file number num

`.loc` ディレクティブは、未使用のファイル番号を指定しています。

Unclosed '('.

始まりカッコ「(」に対応する終わりカッコ「)」がありません。終わりカッコ「)」を追加してください。

Unexpected register in list.

MIPS32 モードで無効なレジスタが使われました。この命令のオペランドを確認してください。

5.4 警告

本アセンブラは、プログラムに欠陥はあるもののアセンブルは続行できると判断した場合に警告を生成します。警告は無視すべきではありません。各警告を注意深く調べ、アセンブラがプログラムの意図を確実に理解できるよう修正する必要があります。警告メッセージによってユーザプログラム内のバグが見つかる場合もあります。

記号

.end directive missing or unknown symbol

.end 関数デバッグ情報ディレクティブが見つからないか、対応するシンボルが未定義です。.end ディレクティブは .ent ディレクティブの後に適切に配置する必要があります。

.end directive without a preceding .ent directive.

.end 関数デバッグ情報ディレクティブに対し、シンボルを関数として指定するための .ent ディレクティブがありません。.end ディレクティブは .ent ディレクティブの後に適切に配置する必要があります。

.end not in text section

.end 関数デバッグ情報ディレクティブは、実行可能コードと一緒のセクション内にある事が必要です。

.end symbol does not match .ent symbol.

.end 関数デバッグ情報ディレクティブの関数引数は、前にある .ent ディレクティブの関数引数と合致しません。.end ディレクティブは . 対応する .ent ディレクティブの後に適切に配置する必要があります。

.endr encountered without preceding .rept, .irc, or .irp

.endr ディレクティブは .rept、.irc、.irp シーケンスを終了させますが、この .endr ディレクティブの前に .rept、.irc、.irp ディレクティブはありません。.endr ディレクティブはコード内の正しい位置に書く必要があります。

.ent or .aent not in text section.

.ent 関数デバッグ情報ディレクティブは、実行可能コードをと一緒のセクション内にある事が必要です。

.fail *expr* encountered

.fail 式の値が 500 以上である場合、本アセンブラは警告メッセージを出力します。メッセージには式の値が示されます。

.fill size clamped to 8

.fillサイズ値は0または0より大きな値に設定できますが、8より大きい場合は8であると見なされます。

.frame outside of .ent

.frame ディレクティブはスタックフレームを記述するため、関数の中で使う必要があります。

.incbin count zero, ignoring filename

.incbin カウントは 0 より大きい事が必要です。ファイルから 0 バイトを読み出しても何の効果も得られません。

.mask/.fmask outside of .ent

.mask/.fmask スタックフレーム情報は .ent 関数の中で定義する必要があります。.mask/.fmask ディレクティブがソースコード内の正しい位置にあるか確認してください。

.popsection without corresponding .pushsection; ignored

最初にスタックにセクションをプッシュしないと、セクション スタックのセクションをポップする事はできません。

.previous without corresponding .section; ignored

現在のセクションとスワップするセクションが現在のセクションより前にありません。.previous ディレクティブがソースコード内の正しい位置にあるか確認してください。

.space repeat count is negative, ignored.

.space のサイズ引数は 0 より大きい事が必要です。

.space repeat count is zero, ignored.

.space のサイズ引数は 0 より大きい事が必要です。

A

Alignment negative: 0 assumed.

アラインメント値 .align は非負の 2 のべき乗値である事が必要です。.align 0 は pseudo-ops を生成するデータによって使われる自動的なアラインメントを OFF にします。

Alignment too large: 15 assumed.

アラインメント値 .align が 15 を超えています。有効レンジは 0 ～ 15 です。

D

Divide by zero.

$zero レジスタを RT とする除算命令

Division by zero.

この式は 0 による除算を試みます。オペランドを確認してください。

E

Extended instruction in delay slot.

MIPS32e 拡張された命令をジャンプ遅延スロット内に置く事はできません (未確定の挙動が発生します)。この命令を遅延スロットの外に移動させてください。

F

Floating point constant too large.

浮動小数点定数の 16 進エンコーディングが大きすぎます。この浮動小数点値が 32 ビットまたは 64 ビットの IEEE 形式で正しくエンコードされる事を確認してください。

I

Ignoring changed section attributes for *name*

1 つのセクションに対してセクション属性が 2 度指定された場合、2 度目の属性は最初の属性と同じである事が必要です。異なる場合、アセンブラは最初のセクション属性が正しかったと見なします。

Ignoring changed section entity size for *name*

アセンブラが同じセクションを 2 度見つけた場合、それらのエンティティ サイズは同じである事が必要です。アセンブラは最初に見つけたセクションのエンティティ サイズが正しかったと見なします。

Ignoring changed section type for *name*

アセンブラが同じセクションを 2 度見つけた場合、それらのセクションタイプは同じである事が必要です。アセンブラは最初に見つけたセクションのセクションタイプが正しかったと見なします。

Ignoring incorrect section type for *name*

名前によって特別な定義済みセクションへ切り換える場合、セクションのタイプは定義済みのタイプと一致する必要があります。アセンブラはそのセクションに対して定義済みのタイプを使います。

Immediate for %s not in range 0..1023 (%lu).

デバッガの Break コードが有効レンジ内ではありません。通常のユーザコードは、デバッグ用に予約されたこの命令を使うべきではありません。

Improper shift amount (%lu).

シフト命令のシフト値 (例 : SLL、SRA、SRL) がレンジ外です。

Instruction sne: Instruction %s: result is always false.

SNE 命令によって評価される条件の結果は常に「偽」です (例 : オペランド s がゼロレジスタで、オペランド t が非ゼロ定数式)。

Instruction seq: result is always true.

SEQ 命令によって評価される条件の結果は常に「偽」です (例 : オペランド s がゼロレジスタで、オペランド t が 0 の定数)。

Invalid merge entity size.

セクションマージ エンティティ サイズは非負である事が必要です。

Invalid number.

その定数の書式は認識されませんでした。定数の接頭辞と基数を確認してください。

J

Jump address range overflow (0x%lx).

ジャンプ命令のターゲット アドレスが 228 バイト「ページ」の外側です。

L

Left operand is a bignum; integer 0 assumed.

式の左辺のオペランドが整数ではなく bignum です。アセンブラは整数に対してのみ式の評価を実行します。従って、アセンブラはこのオペランドを整数の 0 であると見なします。

Left operand is a float; integer 0 assumed.

式の左辺のオペランドが整数ではなく浮動小数点数です。アセンブラは整数に対してのみ式の評価を実行します。従って、アセンブラはこのオペランドを整数の 0 であると見なします。

Line numbers must be positive; line number %d rejected.

このディレクティブは、行番号に対して正の整数だけを受け入れます。

M

Missing close quote; (assumed).

一重文字引用符が正しく閉じられていません。

Missing operand; zero assumed.

式のオペランド (恐らく右辺) がありません。アセンブラは整数の 0 と見なして続行します。

O

Operand overflow.

ベースレジスタ (basereg) + オフセット オペランドとして使われる定数式は 32 ビット符号付き定数だけを受け入れます。

R

Repeat < 0; .fill ignored.

`.fill` ディレクティブに対する `repeat` 引数は非負である事が必要です。

Right operand is a bignum; integer 0 assumed.

式の右辺のオペランドが整数ではなく bignum です。アセンブラは整数に対してのみ式の評価を実行します。従って、アセンブラはこのオペランドを整数の 0 であると見なします。

Right operand is a float; integer 0 assumed.

式の右辺のオペランドが整数ではなく浮動小数点数です。アセンブラは整数に対してのみ式の評価を実行します。従って、アセンブラはこのオペランドを整数の 0 であると見なします。

S

Setting incorrect section attributes for *name*

特定セクションに対してセクション属性を設定する際は、セクションの属性が定義済みタイプの属性と合致する必要があります。アセンブラはセクションに対して定義済みタイプを使います。

Setting incorrect section type for *name*

特定セクションに対してセクション属性を設定する際は、セクションの属性が定義済みタイプの属性と合致する必要があります。アセンブラはセクションに対して定義済みタイプを使います。

Size negative; .fill ignored.

`.fill` ディレクティブに対する size 引数は非負である事が必要です。

T

Tried to set unrecognized symbol: *name*

`.set` ディレクティブ内のシンボルは PIC32 MCU アセンブラ シンボルとして認識されません。

Truncated file *filename*, *num1* of *num2* bytes read.

.incbin ファイルからの読み出しバイト数が引数で指定された数に対して不足しています。

U

Unknown escape *escape* in string; ignored.

文字列は認識されないバックスラッシュ エスケープ文字を含んでいます。バックスラッシュの後の文字が正しいか確認してください。

Used $at without .set noat.

このコードは $at (アセンブラ テンポラリ) レジスタを使っていますが、アセンブラは合成されたマクロ命令を生成する際にこのレジスタを使う可能性があります。このレジスタを暗黙のうちに使わないようアセンブラに指示するため、.set noat ディレクティブを使う必要があります。

5.5 メッセージ

本アセンブラは、プログラムに欠陥はあるもののアセンブルは問題を生じずに続行できると判断した場合にメッセージを生成します。このメッセージは無視してかまいません。しかし、メッセージによってユーザプログラム内のバグが見つかる場合もあります。

パート 2 - MPLAB XC32 オブジェクト リンカ

NOTE:

第 6 章　リンカの概要

6.1　はじめに

MPLAB XC32 オブジェクト リンカ (xc32-ld) は、再配置可能オブジェクトコードとアーカイブから PIC32 MCU ファミリデバイス向けのバイナリコードを生成します。この 32 ビットリンカは、実行可能コードの開発用プラットフォームを提供する Windows コンソール アプリケーションです。本リンカは Free Software Foundation が提供する GNU アセンブラの移植版です。

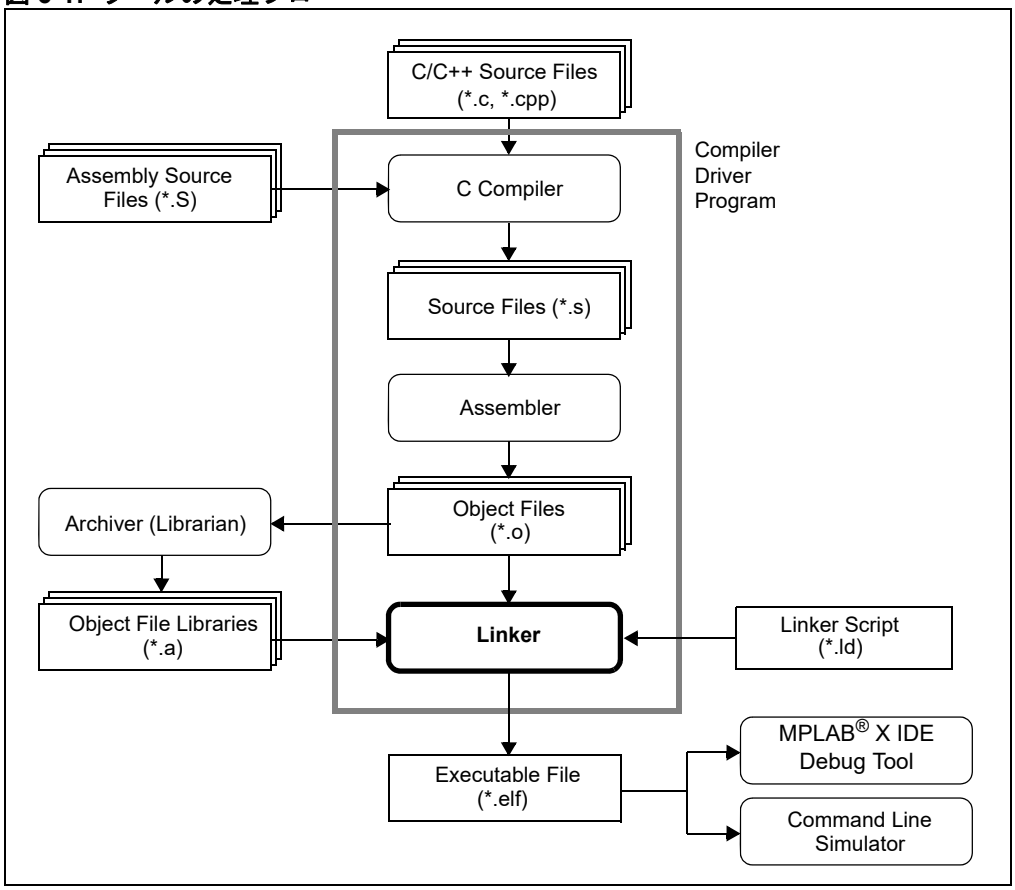
第 6 章の主な内容は以下の通りです。

- リンカとその他の開発ツール
- 特長
- 入出力ファイル

6.2　リンカとその他の開発ツール

PIC32 リンカは、PIC32 アセンブラからのオブジェクト ファイルと PIC32 アーカイバ / ライブラリアンからのアーカイブ ファイルを実行可能ファイルに変換します。ツールの処理フローの概要を図 6-1 に示します。

図 6-1: ツールの処理フロー



6.3 特長

本リンカの主な特長は以下の通りです。

- ユーザ定義による最小スタックの割り当て
- ユーザ定義によるヒープの割り当て
- Windows 対応
- 全ての現行 PIC32 デバイスに対応するリンカスクリプト
- コマンドライン インターフェイス
- MPLAB X IDE への統合

6.4 入出力ファイル

リンカ入出力ファイルを以下に示します。

表 6-1: リンカファイル

拡張子	概要
入力	
.o	オブジェクト ファイル
.a	ライブラリ ファイル
.ld	リンカスクリプト ファイル
出力	
.elf, .out	リンカ出力ファイル
.map	マップファイル

Microchip 社の MPLINK™ リンカとは異なり、本 32 ビットリンカは絶対的なリスティング ファイルを生成しません。本 32 ビットリンカは、マップファイルとバイナリ ELFファイル(デバッグ情報は必要に応じて含める事が可能)を生成できます。MPLINK のリスティング ファイルに類似したテキスト出力を生成するには、xc32-objdump バイナリ ユーティリティを使って ELF ファイルを実行します。

6.4.1 オブジェクト ファイル

オブジェクトファイルは、アセンブラが生成する再配置可能コードです。本リンカは ELF オブジェクト ファイル フォーマットに対応します。

6.4.2 ライブラリ ファイル

ライブラリ ファイルは、使いやすく構成されたオブジェクト ファイルの集まりです。

6.4.3 リンカスクリプト ファイル

リンカスクリプト (コマンドファイル) の用途は以下の通りです。

- リンカにセクションの配置を指示する
- 各部のメモリレンジを指定する

リンカスクリプトをカスタマイズする事で、ユーザ定義セクションを指定したアドレスに配置できます。リンカスクリプト ファイルの詳細は**第 8 章「リンカスクリプト」**を参照してください。

例 6-1: リンカスクリプト

> **Note:** 以下の簡潔なリンカスクリプトのサンプルは説明用に記載しています。完全な物ではないため、このままでは使えません。

```
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH(pic32mx)
ENTRY(_reset)

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  (w!x):ORIGIN=0xA0000000, LENGTH=0x2000
 sfrs                 :ORIGIN=0xBF800000, LENGTH=0x100000
}

SECTIONS
{
  .text ORIGIN(kseg0_program_mem)  :
  {
    _text_begin = .;
    *(.text .stub .text.*)
    *(.mips16.fn.*)
    *(.mips16.call.*)
    _text_end = .;
  } >kseg0_program_mem =0
  .data   :
  {
    _data_begin = .;
    *(.data .data.*.gnu.linkonce.d.*)
    KEEP (*(.gnu.linkonce.d.*personality*))
    *(.data1)
  } >kseg1_data_mem AT>kseg0_program_mem
  .bss   :
  {
    *(.dynbss)
    *(.bss .bss.*)
    *(COMMON)
    .= ALIGN(32 / 8) ;
  } >kseg1_data_mem
  .stack ALIGN(4) :
  {
    .+= _min_stack_size ;
  } >kseg1_data_mem
}
```

6.4.4 リンカ出力ファイル

リンカ出力バイナリファイルの既定値名は `a.out` です。コマンドラインで `-o` オプションを指定する事により、別の名前を指定できます。MPLAB X IDE プロジェクトマネージャは `-o`オプションを使って出力ファイルに*`projectname`*`.elf` という名前を指定します (*`projectname`*: ユーザの MPLAB X IDE プロジェクト名)。

バイナリファイルのフォーマットは ELF (Executable and Linking Format) です。ELF は元々 UNIX System Laboratories (USL) により ABI (Application Binary Interface) の一部として開発されました。ELF は、Tool Interface Standards (TIS) コミッティーが定めた仕様です (TIS とは、開発ツールに対して可視のソフトウェア インターフェイスを標準化するためにマイクロコンピュータ業界メンバーによって構成された協会です)。

ELF ファイル内のデバッグ情報は、DWARF デバッグ情報フォーマットに従います。DWARF フォーマットも TIS コミッティーにより定められました。このフォーマットは、一連のデバッグエントリを使ってソースプログラムのローレベル表現を定義します。DWARF 対応のツール (MPLAB X IDE 等) は、この表現方式を使う事で、オリジナル ソースプログラムの様相を正確に表現できます。

6.4.5 マップファイル

本リンカが生成するマップファイルは以下の内容を含みます。

- Archive Member Table - リンクに含めるアーカイブ ファイルからの全てのメンバーのリストです。
- Memory Usage Report - プログラムメモリとデータメモリ内の全ての出力セクションの開始アドレスと長さを示します。領域内のメモリ使用率 (%) も示します。
- Memory Configuration - リンク用に定義された全てのメモリ領域のリストを示します。
- Linker Script and Memory Map - リンカスクリプト内の指定に従ってリンクにインクルードされるモジュール、セクション、シンボルを示します。
- Outside Cross Reference Table (オプション) - シンボルを名前順に示します。各シンボルに対応するファイル名のリストを提供します。シンボルが定義されている場合、その定義はリストに示される最初のファイルの中にあります。リスト内の残りのファイルは、そのシンボルへの参照を含んでいます。

例 6-2: マップファイル

```
Archive member included because of file (symbol)

size\libc.a(general-exception.o)
         size/crt0.o (_general_exception_context)
size\libc.a(default-general-exception-handler.o)
         size\libc.a(general-exception.o) (_general_exception_handler)
size\libc.a(default-bootstrap-exception-handler.o)
         size/crt0.o (_bootstrap_exception_handler)
size\libc.a(default-on-reset.o)
         size/crt0.o (_on_reset)
size\libc.a(default-on-bootstrap.o)
         size/crt0.o (_on_bootstrap)
size\libc.a(default-nmi-handler.o)
         size/crt0.o (_nmi_handler)

Microchip PIC32 Memory-Usage Report

kseg0 Program-Memory Usage
section    address  length   (dec)  Description
------- ----------  --------------  -----------
.text   0x9d000000   0x678    1656  Application's executable code
.rodata 0x9d000678    0x14      20  Read-only constant data
.data   0x9d00068c     0xf     244  Data-initialization template
.sdata  0x9d000780     0x4       4  Small data-initialization template
Total kseg0_program_mem used:
                      0x784    1924  0.4% of 0x80000

kseg0 Boot-Memory Usage
section  address    length   (dec)  Description
-------  ---------- --------------  -----------
.startup 0x9fc00490  0x1e0     480  C startup code
Total kseg0_boot_mem used:
                      0x1e0     480  19.9% of 0x970

Exception-Memory Usage
section    address    length   (dec)  Description
-------    ---------- --------------  -----------
.app_excpt 0x9fc01180   0x10      16  General-Exception handler
.vector_1  0x9fc01220    0x8       8  Interrupt Vector 1
Total exception_mem used  :
                        0x18      24  0.6% of 0x1000

kseg1 Boot-Memory Usage
section    address     length   (dec)  Description
-------    ----------  --------------  -----------
.reset     0xbfc00000    0x10      16  Reset handler
.bev_excpt 0xbfc00380    0x10      16  BEV-Exception handler
Total kseg1_boot_mem used  :
                        0x20      32  2.7% of 0x490
        ---------------------------------------------
         Total Program Memory used  :
                        0x99c    2460  0.5% of 0x81e00
        ---------------------------------------------
```

```
kseg1 Data-Memory Usage
section    address     length   (dec)  Description
-------    ----------  -------------  -----------
.data      0xa0000000    0xf4     244  Initialized data
.sdata     0xa00000f4     0x4       4  Small initialized data
.sbss      0xa00000f8     0x4       4  Small uninitialized data
.bss       0xa00000fc   0x10c     268  Uninitialized data
.heap      0xa0000208   0x800    2048  Dynamic Memory heap
.stack     0xa0000a08   0x400    1024  Min space reserved for stack
         Total kseg1_data_mem used  :
                      0xe08    3592  11.0% of 0x8000
        ------------------------------------------------------------
            Total Data Memory used  :
                      0xe08     3592  11.0% of 0x8000
        ------------------------------------------------------------
Memory Configuration

Name             Origin             Length             Attributes
kseg0_program_mem 0x9d000000         0x00080000         xr
kseg0_boot_mem   0x9fc00490         0x00000970
exception_mem    0x9fc01000         0x00001000
kseg1_boot_mem   0xbfc00000         0x00000490
config0          0xbfc02ffc         0x00000004
kseg1_data_mem   0xa0000000         0x00008000         w !x
sfrs             0xbf800000         0x00100000
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD size/crt0.o
                0x00000800       _min_heap_size = 0x800
START GROUP
LOAD size\libc.a
LOAD size\libm.a
LOAD size\libmchp_peripheral_32MX360F512L.a
END GROUP
LOAD C:/xc32-Tools/bin/../lib/gcc/pic32mx/3.4.4/size\libgcc.a
              0x00000400    PROVIDE (_min_stack_size, 0x400)
              0x00000000    PROVIDE (_min_heap_size, 0x0)
LOAD ./proc/32MX360F512L\processor.o
              0x00000001    PROVIDE (_vector_spacing, 0x1)
              0x9fc01000    _ebase_address = 0x9fc01000
              0xbfc00000    _RESET_ADDR = 0xbfc00000
              0xbfc00380    _BEV_EXCPT_ADDR = 0xbfc00380
              0x9fc01180    _GEN_EXCPT_ADDR = (_ebase_address + 0x180)

.reset        0xbfc00000    0x10
 *(.reset)
 .reset       0xbfc00000    0x10 size/crt0.o
              0xbfc00000        _reset
.bev_excpt    0xbfc00380    0x10
 *(.bev_handler)
 .bev_handler 0xbfc00380    0x10 size/crt0.o
.vector_0     0x9fc01200    0x0
 *(.vector_0)

.startup      0x9fc00490    0x1e0
 *(.startup)
 .startup     0x9fc00490    0x1e0 size/crt0.o
```

```
.text          0x9d000000    0x678
               0x9d000000          _text_begin = .
 *(.text .stub .text.*.gnu.linkonce.t.*)
 .text         0x9d000000    0x18 size/crt0.o
 .text         0x9d000018    0x110 intermediate\object.o
               0x9d000089          testfunct
               0x9d0000a0          main
               0x9d000018          foo
 .text         0x9d000128    0xc intermediate   est.o
               0x9d000128          mylabel
 .text.general_exception
               0x9d000134    0xd0 size\libc.a(general-exception.o)
               0x9d000134          _general_exception_context
 .text._general_exception_handler
       0x9d0005bc 0x8 size\libc.a(default-general-exception-handler.o)
            0x9d0005bc          _general_exception_handler
 .text._bootstrap_exception_handler
    0x9d0005c4  0x8 size\libc.a(default-bootstrap-exception-handler.o)
               0x9d0005c4          _bootstrap_exception_handler
 .text._on_reset
               0x9d0005cc    0x8 size\libc.a(default-on-reset.o)
               0x9d0005cc          _on_reset
 .text._on_bootstrap
               0x9d0005d4    0x8 size\libc.a(default-on-bootstrap.o)
               0x9d0005d4          _on_bootstrap
 .text         0x9d0005dc    0x18 size\libc.a(default-nmi-handler.o)
               0x9d0005dc          _nmi_handler
.sdata         0xa00000f4    0x4 load address 0x9d000780
               0xa00000f4          _sdata_begin = .
.heap          0xa0000208    0x800
               0xa0000208          _heap = .
               0xa0000a08          .= (.+ _min_heap_size)
 *fill*        0xa0000208    0x800 00

.stack         0xa0000a08    0x400
               0xa0000e08          .= (.+ _min_stack_size)
 *fill*        0xa0000a08    0x400 00

.ramfunc       0xa0001000    0x0 load address 0x9d000784
               0xa0001000          _ramfunc_begin = .
 *(.ramfunc .ramfunc.*)
               0xa0001000          .= ALIGN (0x4)
               0xa0008000          _stack =
                                  (_ramfunc_length >0x0)?
                                  (_ramfunc_begin - 0x4):0xa0008000

OUTPUT(test-2.elf elf32-tradlittlemips)

Cross Reference Table

Symbol          File
PORTE           ./proc/32MX360F512L\processor.o
                size\libc.a(default-nmi-handler.o)
                size\libc.a(general-exception.o)
                intermediate/test.o
                size/crt0.o
foo             intermediate\cobject.o
main            intermediate\cobject.o
                size/crt0.o
mylabel         intermediate\asmobject.o
funct           intermediate\cobject.o
```

NOTE:

第 7 章　リンカのコマンドライン インターフェイス

7.1　はじめに

MPLAB XC32 オブジェクト リンカ (xc32-ld) は、コマンドライン インターフェイスまたは MPLAB X IDE を介して使う事ができます。

第 7 章の主な内容は以下の通りです。

- リンカ インターフェイスの構文
- コンパイル ドライバ リンカ インターフェイスの構文
- 出力ファイルの生成を制御するオプション
- ランタイム初期化を制御するオプション
- 情報出力を制御するオプション
- リンクマップ出力を変更するオプション

7.2 リンカ インターフェイスの構文

本リンカは多数のコマンドライン オプションをサポートしますが、どのような状況でも実際に使うのはその中の一部だけです。

```
xc32-ld [options] file...
```

Note: コマンドライン オプションは大文字と小文字を区別します。

例えば xc32-ld は、オブジェクト ファイルとアーカイブをリンクしてバイナリファイルを生成するために頻繁に使います。以下はファイル hello.o をリンクする場合の例です。

```
xc32-ld -o output hello.o -lpic32
```

これは、ファイル hello.o を libpic32.a とリンクして output という名前のファイルを生成するよう xc32-ld に指示します。

C アプリケーションをリンクする場合、通常は各種のアーカイブ (ライブラリとも呼ぶ) をリンクコマンド内でインクルードします。アーカイブのリストを --start-group オプションと --end-group オプションの間で指定すると、循環参照を解決するために役立ちます。

```
xc32-ld -o output hello.o --start-group -lpic32 -lm -lc --end-group
```

xc32-ld に対するコマンドライン オプションは任意の順番で指定でき、いくらでも繰り返す事ができます。引数の異なる同じオプションを繰り返し指定した場合、何の効果もない場合と、先に現れる (コマンドラインの左側にある) 同じオプションを上書きする場合があります。複数回指定しても意味のあるオプションについては後で説明します。

引数 (オプションではない) として、互いにリンクするオブジェクト ファイルを指定する必要があります。これらの引数を指定する位置はコマンドライン オプションの前でも後でも、オプションとオプションの間でも構いません。しかし、オプションとその引数の間に挿入する事はできません。

通常リンカは 1 つ以上のオブジェクト ファイルを指定して呼び出します。しかし、-l とスクリプト コマンド言語を使って、他の形態のバイナリ入力ファイルを指定する事ができます。バイナリ入力ファイルが何も指定されないと、リンカは出力を生成せずにメッセージ「No input files」を出力します。

オブジェクト ファイルのフォーマットを認識できない場合、リンカはそのファイルをリンカスクリプトであると見なします。この方法で指定されたスクリプトは、リンクに使われるメインのリンカスクリプト (既定値リンカスクリプトまたは -T を使って指定されたリンカスクリプト) を増補します。この機能は、オブジェクトまたはアーカイブのように見えるファイルに対してリンカがリンクする事を許容しますが、実際には単に何らかのシンボル値を定義するだけか、INPUT または GROUP を使って他のオブジェクトをロードします。

名前が 1 文字のオプションの場合、オプションの引数はオプション文字に続けて (空白で区切らずに) 指定するか、空白で区切ってオプションの直後で指定します。

名前が複数文字のオプションの場合、オプション名の前に 1 つまたは 2 つのダッシュ (-) を付けます (例 : -trace-symbol と --trace-symbol は等価です)。この規則には例外が 1 つあります。文字 o で始まる複数文字名のオプションの前にはダッシュを必ず 2 個付ける必要があります。

複数文字オプションの引数は、等号 (=) を使ってオプション名から区切るか、空白で区切ってオプションの直後で指定します。例えば --trace-symbol srec と --trace-symbol=srec は等価です。複数文字オプションの名前には一意の省略形が使えます。

7.3 コンパイル ドライバ リンカ インターフェイスの構文

通常、本リンカはコンパイル ドライバ xc32-gcc から呼び出されます。コンパイル ドライバに対するコマンドラインの基本形態は以下の通りです。

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

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

コンパイル ドライバからリンカにリンカオプションを渡すには、`-Wl,option` オプションを使います。

例 7-1: コンパイル ドライバ コマンドライン

```
xc32-gcc -mprocessor=32MX360F512L "input.o" -o"output.elf"
   -Os -Wl,--defsym=_min_heap_size=2048,-Map="mapfile.map",
   --cref,--report-mem
```

リンカを直接呼び出さずにコンパイル ドライバから呼び出す事により、いくつかの利点が得られます。

- ドライバの `-mprocessor` オプションを使うと、デバイスに固有のインクルードファイルとライブラリ検索パスを正しくリンカに渡す事ができます。例えば、`-mprocessor=32MX360F512L`と指定すると、ドライバはそのデバイスに固有のライブラリ検索パス (`pic32mx/lib/proc/32MX360F512L`) をリンカに渡します。リンカはこのパスを使って、ターゲット デバイスに応じた正しい既定値リンカスクリプトとプロセッサ ライブラリを見つける事ができます。
- 適切な複数ライブラリの並べ換えを選択するために必要なCコンパイラの最適化オプション、ISA モード、浮動小数点サポート オプションをドライバに対して指定する事ができます。例えば、サイズ最適化オプション `-Os` を指定すると、ドライバはライブラリ検索パスとして `pic32mx/lib/size` をリンカに渡します。これに従い、リンカはサイズに対して最適化されたプリコンパイル済みライブラリを使います。C コンパイラ ドライバの multilib 機能の詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

7.4 出力ファイルの生成を制御するオプション

以下は出力ファイル生成オプションです。

- -(archives -), --start-group archives, --end-group
- -d, -dc, -dp
- --defsym sym=expr
- --discard-all (-x)
- --discard-locals (-X)
- --fill=option
- --gc-sections
- --library name (-l name)
- --library-path <dir> (-L <dir>)
- -nodefaultlibs
- -nostartfiles
- -nostdlib
- --output file (-o file)
- --p PROC
- --relocatable (-r, -i, -Ur)
- --retain-symbols-file file
- --section-start sectionname=org
- --script file (-T file)
- --strip-all (-s)
- --strip-debug (-S)
- -Tbss address
- -Tdata address
- -Ttext address
- --undefined symbol (-u symbol)
- --no-undefined
- --wrap symbol

7.4.1 -(*archives* -), --start-group *archives*, --end-group

アーカイブのグループを指定します。

archives はアーカイブ ファイルのリストです。それらはファイル名を明示的に指定するか、-l オプションで指定します。指定された *archives* は、新たな未定義参照が生成されなくなるまで繰り返し検索されます。通常アーカイブは、コマンドラインで指定された順に一度だけ検索されます。そのアーカイブの中のシンボルが、コマンドラインの後方 (右側) に現れるアーカイブの中のオブジェクトによって参照される未定義シンボルを解決する必要がある場合、リンカはその参照を解決できません。アーカイブをグループ化すると、全ての参照が解決するまで、それら全てのアーカイブが繰り返し検索されます。このオプションは非常に大きな処理能力を要求します。2 つ以上のアーカイブの間の循環参照がどうしても避けられない場合にのみ、このオプションを使う事を推奨します。

7.4.2 -d, -dc, -dp

共有シンボルを強制的に定義します。

再配置可能出力ファイルがオプション -r によって指定された場合でも、共有シンボルに空間を割り当てます。これは、スクリプト コマンド FORCE_COMMON_ALLOCATION と同じ効果を持ちます。

7.4.3 --defsym *sym=expr*

シンボルを定義します。

出力ファイル内でグローバル シンボルを生成します。シンボルは *expr* によって与えられた絶対アドレスを格納します。複数のシンボルをコマンドラインで定義する場合、必要に応じてこのオプションを何度でも使えます。この状況では、*expr* に一部の算術演算が使えます。すなわち、16 進定数または既存シンボルの名前を指定する他に、「+」または「-」を使って 16 進定数またはシンボルを加減算できます。

> **Note:** *sym*、等号 (=)、*expr* の間に空白を挿入しない事が必要です。

7.4.4 --discard-all (-x)

全てのローカルシンボルを破棄します。

7.4.5 --discard-locals (-X)

テンポラリ ローカルシンボルを破棄します。

7.4.6 --fill=option

--fill=option

未使用プログラムメモリ位置に値を書き込みます。書式は以下の通りです。

--fill=[wn:]expression[@address[:end_address] | unused]

address と end_address は値を書き込むプログラムメモリの範囲を指定します。end_address を指定しない場合、expression で指定された値はメモリ内のアドレス address にだけ書き込まれます。必要に応じてリテラル値 unused を指定する事で、全ての未使用メモリに値を書き込むよう指定できます。アドレス パラメータを何も指定しないと、全ての未使用メモリが書き込まれます。expression は指定されたメモリに書き込む内容を指定します。以下のオプションが利用できます。

単一値 :

```
xc32-ld --fill=0x12345678@unused
```

値のレンジ :

```
xc32-ld --fill=1,2,3,4,097@0x9d000650:0x9d000750
```

インクリメント値 :

```
xc32-ld --fill=7+=711@unused
```

既定値では、リンカは命令ワードと同じ長さ (幅) のデータをメモリに書き込みます。32 ビットデバイスの場合、書き込まれる値の既定値幅は 32 ビットです。しかし、[wn:] を使って書き込み値の幅を指定する事ができます (w は値の幅、n は 1/2/4/8 のいずれか)。コマンドラインで複数の -fill オプションを指定できます。リンカは、常に指定されたメモリ位置を先頭にして -fill オプションを処理します。

7.4.7 --gc-sections

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す。このオプションは -r とは非互換です。既定値動作 (ガベージ コレクションは無効) に戻すには、コマンドラインで *--no-gc-sections* を指定します。

リンク時ガベージ コレクションを使う場合、削除すべきではないセクションをマーキングすると便利です。セクションをマーキングするには、入力セクションのワイルドカード エントリを KEEP() で囲みます (KEEP(*(.init)) または KEEP(SORT_BY_NAME(*)(.ctors)))。

7.4.8 --library *name* (-l *name*)

ライブラリ *name* を検索します。
アーカイブ ファイル *name* をリンクするファイルのリストに追加します。このオプションは何度でも使えます。xc32-ld は、指定された各 *name* に対して lib*name*.a をパスリスト内で検索します。リンカは、コマンドラインで指定された場所でアーカイブを一度だけ検索します。そのアーカイブが、コマンドラインの前方 (左側) に現れるオブジェクト内で未定義であったシンボルを定義した場合、リンカはそのアーカイブから適切なファイルをインクルードします。しかし、コマンドラインの後方 (右側) に現れるオブジェクト内に未定義シンボルが存在しても、リンカはそのアーカイブを再度検索しません。リンカにアーカイブを繰り返し検索させるための方法については、-(オプションを参照してください。コマンドライン上で同じアーカイブを複数回指定しても構いません。

アーカイブ ファイルのフォーマットが認識できない場合、リンカはそのファイルを無視します。従って、ライブラリとリンカの間でバージョンが整合しない場合、「undefined symbol」エラーが発生する可能性があります。

7.4.9 --library-path <*dir*> (-L <*dir*>)

<*dir*> をライブラリ検索パスに追加します。
xc32-ldがアーカイブ ライブラリとxc32-ld制御スクリプトを検索するパスのリストに <*dir*> を追加します。このオプションは何度でも指定できます。ディレクトリ (dir) は、コマンドラインで指定された順番に検索されます。全ての -L オプションは、指定された順番に関係なく、全ての -l オプションに対して適用されます。ライブラリ パスは、リンカスクリプト内で SEARCH_DIR コマンドを使って指定する事もできます。この方法で指定したディレクトリは、コマンドラインにそのリンカスクリプトが現れた時点で検索されます。

7.4.10 -nodefaultlibs

リンク時に標準システム 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、別の方法でこれらのエントリポイント提供する必要があります。

7.4.11 -nostartfiles

既定値のプリビルド済み C スタートアップ ファイル (pic32mx/lib/crt0.o) をリンカに渡しません。ユーザが独自バージョンのスタートアップ コードをアプリケーションに提供します。

7.4.12 -nostdlib

リンク時に標準システム スタートアップ ファイルまたはライブラリを使いません。リンカにはスタートアップ ファイルは渡されず、ユーザが指定したライブラリだけが渡されます。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、別の方法でこれらのエントリポイント提供する必要があります。

7.4.13 --output *file* (-o *file*)

出力 ELF ファイルの名前を設定します。

xc32-ld が生成したプログラムの名前として *file* を指定します。このオプションを指定しない場合、a.out という既定値名が使われます。

7.4.14 --p PROC

ターゲット プロセッサ (例 : 32MX795F512L) を指定します。

リンクのためにターゲット プロセッサを指定します。

7.4.15 --relocatable (-r, -i, -Ur)

再配置可能出力を生成します。
すなわち、再び xc32-ld への入力として使える出力ファイルを生成します。これはしばしばパーシャルリンクと呼ばれます。このオプションを指定しない場合、絶対的ファイルが生成されます。

7.4.16 --retain-symbols-file *file*

file 内のリストで指定されたシンボルだけを保持します。
file 内のリストで指定されたシンボルだけを保持し、その他は全て破棄します。*file* は、シンボル名を 1 行に 1 つずつ記述した単純なフラットファイルです。このオプションは、ランタイム メモリを節約するために大きなグローバル シンボルテーブルを徐々に蓄積する環境において特に便利です。--retain-symbols-file は未定義のシンボルまたは再配置のために必要なシンボルを破棄しません。--retain-symbols-file はコマンドラインで一度しか指定できません。このオプションは -s と -S を上書きします。

7.4.17 --section-start sectionname=*org*

org で指定された絶対アドレスで出力ファイル内にセクションを配置します。このオプションは必要に応じて何度でも使えます (コマンドラインで複数のシンボルを配置できます)。*org* は 1 つの 16 進整数である事が必要です。他のリンカとの互換性を維持するため、16 進値の前に付ける「0x」は省略可能です。

Note: sectionname、等号 (=)、*org* の間に空白を挿入しない事が必要です。

7.4.18 --script *file* (-T *file*)

リンカスクリプトを読み出します。
ファイル *file* からリンクコマンドを読み出します。これらのコマンドは xc32-ld の既定値リンカスクリプトを置き換えます (既定値スクリプトに追加されるのではありません)。従って、*file* はターゲット フォーマットを記述するために必要な全ての事を指定する必要があります。*file* が見つからない場合、xc32-ld はコマンドラインの前方 (左側) に現れる全ての -L オプションが指定するディレクトリを検索します。-T オプションを複数回指定した場合、読み出したコマンドは蓄積されます。

7.4.19 --strip-all (-s)

全てのシンボルをストリップします。出力ファイルから全てのシンボル情報を取り除きます。

7.4.20 --strip-debug (-S)

デバッグ シンボルをストリップします。出力ファイルからデバッガ シンボル情報 (全てのシンボル情報ではない) を取り除きます。

7.4.21 -Tbss *address*

.bss セクションのアドレスを設定します。
address を出力ファイルの bss セグメントの開始アドレスとして使います。*address* は 1 つの 16 進整数である事が必要です。他のリンカとの互換性を維持するため、16 進値の前に付ける「0x」は省略可能です。

通常、このセクションのアドレスはリンカスクリプト内で指定されます。

7.4.22 -Tdata *address*

.data セクションのアドレスを設定します。
address を出力ファイルのデータセグメントの開始アドレスとして使います。*address* は1つの16進整数である事が必要です。他のリンカとの互換性を維持するため、16 進値の前に付ける「0x」は省略可能です。

通常、このセクションのアドレスはリンカスクリプト内で指定されます。

7.4.23 -Ttext *address*

.text セクションのアドレスを設定します。

address を出力ファイルのテキスト セグメントの開始アドレスとして使います。*address*は1つの16進整数である事が必要です。他のリンカとの互換性を維持するため、16 進値の前に付ける「0x」は省略可能です。

通常、このセクションのアドレスはリンカスクリプト内で指定されます。

7.4.24 --undefined *symbol* (-u *symbol*)

symbol への未定義参照で開始します。

symbol を未定義シンボルとして出力ファイルに強制的に含めます。これにより、例えば、標準ライブラリから追加のモジュールがリンクされます。異なるオプション引数を使って -u を複数回指定する事により、複数の未定義シンボルを追加できます。

7.4.25 --no-undefined

未定義シンボルを許容しません。

7.4.26 --wrap *symbol*

symbol に対してラッパー関数を使います。

symbol に対してラッパー関数を使います。*symbol* に対する全ての未定義参照は __wrap_symbol へと解決されます。__real_symbol に対する全ての未定義参照は *symbol* へと解決されます。これはシステム関数にラッパーを提供するために使えます。ラッパー関数は __wrap_symbol として呼び出します。システム関数を呼び出す必要がある場合、ラッパー関数は __real_symbol を呼び出します。

以下に例を示します。

```
void *
__wrap_malloc (int c)
{
  printf ("malloc called with %ld\n", c);
  return __real_malloc (c);
}
```

--wrap malloc を使ってこのファイルに他のコードをリンクすると、malloc に対する全ての呼び出しは関数 __wrap_malloc を代わりに呼び出します。
__wrap_malloc 内の __real_malloc への呼び出しは、本当の malloc 関数を呼び出します。
--wrap オプションを使わなくてもリンクが成功するよう、__real_malloc 関数を提供したいと考えるかもしれません。その場合、__real_malloc の定義は __wrap_malloc と同じファイル内に置かない事が必要です。さもないと、先にアセンブラが呼び出しを解決してしまい、リンカが呼び出しを malloc へラップする機会を失う可能性があります。

7.5 ランタイム初期化を制御するオプション

ランタイム初期化オプションには以下があります。

- `--data-init`
- `--no-data-init`
- `--defsym=_min_stack_size=size`
- `--defsym=_min_heap_size=size`

7.5.1 `--data-init`

初期化データをサポートします (これが既定値です)。

データのランタイム初期化のためのテンプレートとして .dinit という名前を持つ特別な出力セクションを生成します。libpic32.a 内の C スタートアップ モジュールは、このテンプレートに従って初期データ値を初期化データセクションにコピーします。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。このオプションは永続的データセクション (.pbss) に影響を与えないという事に注意が必要です。

7.5.2 `--no-data-init`

初期化データをサポートしません。

データのランタイム初期化をサポートするために通常生成されるテンプレートをサポートしません。このオプションを指定すると、リンカは libpic32.a 内の短縮された形態の C スタートアップ モジュールを選択します。アプリケーションが初期化の必要なデータセクションを含む場合、警告メッセージが生成され、初期データ値は破棄されます。データセクションの記憶域は通常通りに配置されます。

7.5.3 `--defsym=_min_stack_size=`*`size`*

既定値リンカスクリプトは、最小限のスタックサイズ (1024 バイト) を提供します。--defsym オプションを使って _min_stack_size シンボルを定義する事で、この既定値 *size* 値を変更します。実効スタックサイズは、最小サイズより大きくなる場合があるという事に注意が必要です。

```
xc32-gcc foo.c -Wl,--defsym=_min_stack_size=1536
```

7.5.4 `--defsym=_min_heap_size=`*`size`*

既定値リンカスクリプトは 0 バイトのヒープサイズを提供します。--defsym オプションを使って _min_heap_size シンボルを定義する事で、この既定値 *size* 値を変更します。リンカは、この値によって定義されたサイズのヒープを生成します。

```
xc32-gcc foo.c -Wl,--defsym=_min_heap_size=2048
```

7.6 multilib ライブラリの選択を制御するオプション

multilib はプリビルド済みターゲット ライブラリのセットです。multilib 内の各ターゲット ライブラリは、コンパイラ オプションの各種組み合わせを使ってビルドされます。multilib は、アプリケーションのビルド用に使われるコンパイラ オプションにターゲット ライブラリを適合させる能力をリンカに提供します。プリビルド済みターゲット ライブラリは、コンパイラ オプションの最も一般的な組み合わせです。

アプリケーションをリンクするためにコンパイル ドライバを呼び出すと、ドライバはアプリケーション オプションに対応するターゲット ライブラリのバージョンを選択します。これらのオプションはコンパイル ドライバに (リンカにではない) 正しく渡す必要があります。コンパイル ドライバは、リンカを呼び出す際に、これらのオプションを適切な -L ライブラリ検索パスに変換します。

サイズと速度の最適化 (-Os と -O3)

サイズを最適化するには -Os を選択し、速度を最適化するには -O0 ～ -O3 を選択します。

-O0

未最適化の複数ライブラリ、ターゲット ライブラリ並べ換えを選択します (これはコマンドライン インターフェイスの既定値です。しかし、MPLAB X IDE のプロジェクトは、これ以外の最適化オプションを既定値として渡す場合があります)。

-O1

最適化レベル 1 でビルドされた複数ライブラリ、ターゲット ライブラリ並べ換えを選択します。

-O2

最適化レベル 2 でビルドされた複数ライブラリ、ターゲット ライブラリ並べ換えを選択します。この最適化レベルは、実行速度とコードサイズを良好にバランスさせた最適化を提供します。この multilib 最適化レベルは大部分のアプリケーションに適しています。

-O3

最適化レベル 3 でビルドされた複数ライブラリ、ターゲット ライブラリ並べ換えを選択します。この最適化は実行速度を最大化します。

-Os

コードサイズを最適化するようビルドされた複数ライブラリ、ターゲット ライブラリ並べ換えを選択します。以下に例を示します。

```
xc32-gcc foo.o -Os -o project.elf
```

7.6.1 命令セットモード (MIPS32/MIPS16E/microMIPS)

-mips16、-mmicromips、-mips32r2 のいずれかに基づく複数ライブラリの並べ換えを選択します。以下に例を示します。

```
xc32-gcc foo.o -O3 -mips16 -o project.elf
```

7.6.2 ソフトウェアによる浮動小数点数のサポート / 非サポート

浮動小数点数が非サポートのライブラリの並べ換えを使うと、浮動小数点数をサポートするライブラリの並べ換えを使うよりも大幅にオーバーヘッドを削減できます。アプリケーションが浮動小数点数のサポートを必要としない場合、このオプションを使います。

-mno-float

ソフトウェアによる浮動小数点数演算をサポートしない複数ライブラリ、ターゲット ライブラリ並べ換えを選択します。

例 :

```
xc32-gcc foo.o -Os -mno-mips16 -mno-float -o project.elf
```

-msoft-float

ソフトウェアによる浮動小数点数演算をサポートする複数ライブラリ、ターゲット ライブラリ並べ換えを選択します。

7.7 情報出力を制御するオプション

情報出力オプションには以下があります。

- --check-sections
- --no-check-sections
- --help
- --no-warn-mismatch
- --report-mem
- --trace (-t)
- --trace-symbol symbol (-y symbol)
- -V
- --verbose
- --version (-v)
- --warn-common
- --warn-once
- --warn-section-align

7.7.1 --check-sections

セクション アドレスのオーバーラップをチェックします **(これが既定値です)。**通常、リンカはこのチェックを実行し、オーバーラップを検出すると適切なエラーメッセージを生成します。

7.7.2 --no-check-sections

セクション アドレスのオーバーラップをチェックしません。このオプションは、メモリ割り当ての問題を診断する場合にのみ使います。

7.7.3 --help

オプションのヘルプを出力します。

標準出力にコマンドライン オプションの要約を出力した後に終了します。

7.7.4 --no-warn-mismatch

不整合な入力ファイルについて警告しません。

通常 xc32-ld は、ユーザが何らかの理由で互いに整合しない (例えば、異なるプロセッサまたは異なるエンディアン向けにコンパイルされた) 入力ファイル同士のリンクを試みるとエラーを出力します。このオプションを指定すると、xc32-ld はそのような不整合を暗黙の内に許容します。このオプションは、リンカエラーが明らかに不適切であるとユーザが確信する何らかの特別なアクションを取る場合にのみ使います。

7.7.5 --report-mem

メモリ使用レポートを出力します。

リンク中にメモリ使用の概要を標準出力に出力します。このレポートはリンクマップにも表示されます。

7.7.6 --trace (-t)

ファイルをトレースします。

xc32-ld が処理中の入力ファイルの名前を出力します。

7.7.7 --trace-symbol *symbol* (-y *symbol*)

symbol の記述をトレースします。

リンクするファイルの内、*symbol* が使われている各ファイルの名前を出力します。このオプションは何度でも使えます。このオプションは、リンクの中に未定義のシンボルが含まれるにも関わらず、それらの参照元が不明である場合に役立ちます。

7.7.8 -V

バージョン等の情報を出力します。

7.7.9 --verbose

リンク中に各種の情報を出力します。

xc32-ldのバージョン番号と、開く事が可能な入力ファイルおよび開く事ができない入力ファイルを表示します。既定値ビルトイン スクリプトを使う場合、リンカスクリプトを表示します。

7.7.10 --version (-v)

バージョン情報を出力します。

7.7.11 --warn-common

共有シンボルの重複を警告します。

ある共有シンボルが別の共有シンボルまたはシンボル定義と重複した場合に警告します。このオプションを使うと、グローバル シンボルの重複によって生じる潜在的問題を見つける事ができます。グローバル シンボルには 3 種類あり、以下にそれらの C コード例を示します。

```
int i = 1;
```

これは定義であり、出力ファイルの初期化データセクション内に配置されます。

```
extern int i;
```

これは未定義の参照であり、空間を割り当てません。この変数に対応する定義または共有シンボルがどこかに存在する必要があります。

```
int i;
```

これは共有シンボルです。1 つの変数に対して 1 つまたは複数の共有シンボルだけが存在する場合、その変数は出力ファイルの非初期化データ領域内に配置されます。

リンカは、同じ変数に対する複数の共有シンボルを 1 つのシンボルへとマージします。それらのサイズが互いに異なる場合、最大サイズを採用します。同じ変数の定義が存在する場合、リンカは共有シンボルを宣言に変換します。

--warn-commonオプションは5種類の警告を生成できます。各警告は2行で構成されます。第 1 行はその時に遭遇したシンボルを示し、第 2 行は同じ名前で以前に遭遇したシンボルを示します。これら 2 つのシンボルの片方または両方が共有シンボルになります。

以下の場合、共有シンボルは参照に変換されます。なぜなら、そのシンボルの定義が既に存在するからです。

```
file(section): warning: common of ‘symbol’ overridden by definition
file(section): warning: defined here
```

以下の場合、共有シンボルは参照に変換されます。なぜなら、そのシンボルの定義に後で遭遇したからです。これは、シンボルに遭遇する順番が異なるという点を除けば前のケースと同じです。

```
file(section): warning: definition of ‘symbol’ overriding common
file(section): warning: common is here
```

以下の場合、共有シンボルは以前に遭遇した同じサイズの共有シンボルとマージされます。

```
file(section): warning: multiple common of ‘symbol’
file(section): warning: previous common is here
```

以下の場合、共有シンボルは以前に遭遇したサイズがより大きな共有シンボルとマージされます。

```
file(section): warning: common of ‘symbol’ overridden by larger common
file(section): warning: larger common is here
```

以下の場合、共有シンボルは以前に遭遇したサイズがより小さな共有シンボルとマージされます。これは、シンボルに遭遇する順番が異なるという点を除けば前のケースと同じです。

```
file(section): warning: common of ‘symbol’ overriding smaller common
file(section): warning: smaller common is here
```

7.7.12 --warn-once

各未定義シンボルに対して一度だけ警告します (そのシンボルを参照するモジュールごとに警告するのではありません)。

7.7.13 --warn-section-align

セクション アラインメント ギャップが正常である事を示します。このオプションは、ギャップを最小化する方法を見つけるために役立ちます。

アライメントのためにセクションの開始位置が変更されると警告します。これは、(通常は連続的である) メモリの配置にギャップが導入されたという事を意味します。通常、入力セクションがアラインメントを設定します。アドレスは、明示的に指定されていない場合 (すなわち SECTIONS コマンドによってセクションの開始アドレスが指定されていない場合) にのみ変更されます。

7.8 リンクマップ出力を変更するオプション

リンクマップ出力を変更するためのオプションには以下があります。

- `--cref`
- `--print-map (-M)`
- `-Map file`

7.8.1 `--cref`

出力相互参照テーブル

リンクマップ ファイルが生成される場合、相互参照テーブルはマップファイルに出力されます。そうではない場合、相互参照テーブルは標準出力に出力されます。必要な場合にスクリプトが容易にテーブルを処理できるよう、テーブルの書式は意図的に簡素化されています。シンボルは名前順に出力されます。各シンボルに対してファイル名のリストが提供されます。シンボルが定義されている場合、最初に示されるファイルの中で定義されています。残りのファイルはシンボルへの参照を含みます。

7.8.2 `--print-map (-M)`

マップファイルを標準出力に出力します。リンクマップは、以下を含むリンクの情報を提供します。

- オブジェクト ファイルとシンボルがメモリ内のどの位置に配置されたか
- 共有シンボルはどのように配置されたか
- リンクにインクルードされる全てのアーカイブ メンバー (そのアーカイブ メンバーのインクルードを必要としたシンボルの情報を含む)

7.8.3 `-Map` *`file`*

マップファイルに書き込みます。

リンクマップをファイル *`file`* に出力します。`--print-map (-M)` オプションの説明を参照してください。

第 8 章　リンカスクリプト

8.1　はじめに

リンカスクリプトは、MPLAB XC32 オブジェクト リンカ (xc32-ld) の機能を制御するために使います。既定値により、本リンカはビルトイン リンカスクリプトをデバイス固有インクルード ファイルと組み合わせて使います。しかし、リンカスクリプトをカスタマイズする事で、ユーザ アプリケーションに固有の方法でリンカを制御できます。

第 8 章の主な内容は以下の通りです。

- リンカスクリプトの概要
- コマンドライン情報
- 既定値リンカスクリプト
- MPLAB X IDE プロジェクトにカスタム リンカスクリプトを追加する
- リンカスクリプトのコマンド言語
- リンカスクリプト内の式

8.2 リンカスクリプトの概要

リンカスクリプトは、以下を含むリンク処理の全てを制御します。

- データメモリとプログラムメモリの割り当て
- 入力ファイルから出力ファイルへのセクションのマッピング
- 特別なデータ構造体 (割り込みベクタテーブル等) の構成

リンカスクリプトは、一連のコマンドが書き込まれたテキストファイルです。各コマンドはキーワード (多くの場合、その後に引数が続く) かシンボルへの代入のどちらかです。

8.3 コマンドライン情報

リンカスクリプトは、コマンドライン上で -T オプションまたは --script オプションを使って指定します (**7.4「出力ファイルの生成を制御するオプション」**参照)。

```
xc32-ld -o output.elf input.o --script mylinkerscript.ld
```

リンカを xc32-gcc から呼び出す場合、-Wl, 接頭辞を追加する事でリンカにオプションを渡す事ができます。

```
xc32-gcc -o output.elf input.o -Wl,--script,mylinkerscript.ld
```

8.4 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が指定されないと、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) オプションで指定するデバイス値です)。

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。

> **Note:** 『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) では、既定値リンカスクリプトの内容を詳細に説明しています。これはアセンブリコードと C コードの両方のプロジェクトを対象とします。

既定値リンカスクリプトは、各標準入力セクションを 1 つまたは複数の特定メモリ領域内にマッピングします。さらに、各メモリ領域は PIC32MCU 上のアドレスセグメント (例 : kseg0、kseg1) にマッピングされます。ユーザ / カーネル アドレス セグメントの詳細は『PIC32MX ファミリ リファレンス マニュアル、セクション 03. メモリ構成』(DS61115) を参照してください。

以下のテーブルに、既定値リンカスクリプトによる標準セクションのメモリ領域へのマッピングを示します。

表 8-1: 既定値リンカスクリプトにおける PIC32の 予約済み標準セクション名

セクション名	生成元	最終位置	既定値リンカスクリプト メモリ領域
.reset	リセットハンドラ	実行可能ブートコード セグメント	kseg0_boot_mem
.bev_excpt	BEV 例外ハンドラ	実行可能ブートコード セグメント	kseg0_boot_mem
.app_excpt	一般例外ハンドラ	実行可能ブートコード セグメント	kseg0_boot_mem
.vector_n	割り込みベクタ n	実行可能ブートコード セグメント	kseg0_boot_mem
.startup	C スタートアップ コード	実行可能ブートコード セグメント	kseg0_boot_mem
.text	コンパイラまたはアセンブラによって生成された命令	実行可能コードセグメント	kseg0_program_mem
.rodata	const を宣言された文字列とCデータ	読み出し専用データセグメント	kseg0_program_mem
.sdata2	小さな(Small)初期化定数のグローバル / 静的データ	読み出し専用データセグメント	kseg0_program_mem
.sbss2	非初期化定数のグローバル / 静的データ (常に 0 となる変数)	読み出し専用データセグメント	kseg0_program_mem
.data	初期値を持つサイズが n バイトより大きな変数 (-Gn でコンパイル) C スタートアップ時にプログラムメモリからデータメモリにコピーされる値	初期化データセグメント	kseg1_data_mem & kseg0_program_mem
.sdata	初期値を持つサイズがnバイト以下の変数 (-Gn でコンパイル) GP 相対アドレス指定向けに使用	小さな (small) 初期化データセグメント	kseg1_data_mem & kseg0_program_mem
.lit4 / .lit8	アセンブラが命令ストリーム内ではなくメモリ内に保存すると決定した定数 (通常は浮動小数点数) GP 相対アドレス指定向けに使用	小さな(small)初期化データセグメント	kseg1_data_mem & kseg0_program_mem
.sbss	サイズが n バイト以下の非初期化変数 (-Gn でコンパイル) GP 相対アドレス指定向けに使用	0 で埋められた小さな (small) セグメント	kseg1_data_mem
.bss	大きな (large) 非初期化変数	0 で埋められたセグメント	kseg1_data_mem
.heap	動的メモリ用に使われるヒープ	リンカスクリプトにより予約済み	kseg1_data_mem
.stack	スタック用に予約される最小空間	リンカスクリプトにより予約済み	kseg1_data_mem
.ramfunc	C スタートアップ時にプログラムメモリからデータメモリにコピーされる RAM 関数	初期化データセグメント	kseg1_data_mem & kseg0_program_mem
.reginfo .stab* .debug*	デバッグ情報	ロード用メモリセグメントに物理的なイメージデータが存在しない	n/a
.line	DWARF デバッグ情報	ロード用メモリセグメントに物理的なイメージデータが存在しない	n/a
.comment	#ident/.ident 文字列	ロード用メモリセグメントに物理的なイメージデータが存在しない	n/a

Note: 上記のテーブルは、リンカスクリプト内でマッピングされなくなったセクションも含んでいます。XC32 v2.00 以上のバージョンでは、ベストフィット アロケータがそれらを割り当てます。

8.5 MPLAB X IDE プロジェクトにカスタム リンカスクリプトを追加する

標準の既定値 32 ビット リンカスクリプトは大部分のアプリケーションの要求を満たす汎用的なスクリプトです。しかし、カスタマイズしたリンカスクリプトが必要になる場合もあります。

その場合、既定値リンカスクリプト ファイル(例:pic32mx/lib/proc/32MZ2048ECH100/p32MZ2048ECH100.ld) をアプリケーションのプロジェクト ディレクトリにコピーします。そして、この新しい *.ld ファイルをプロジェクトに追加します。追加したファイルは「Linker Files」の下のプロジェクトツリーの中に表示されます。

新しい *.ld ファイルに加えたカスタマイズは、プロジェクトに影響を与えます。

カスタマイズしたリンカスクリプト内の未使用セクションはアプリケーションのメモリ使用量に影響しないため、それらはスクリプト内に残しておいても構いません。カスタマイズしたスクリプトから削除する必要があるセクションは、C スタイルのコメントを使ってを無効にできます。

8.6 リンカスクリプトのコマンド言語

リンカスクリプトは、一連のコマンドが書き込まれたテキストファイルです。各コマンドはキーワード (多くの場合、その後に引数が続く) かシンボルへの代入のどちらかです。複数のコマンドはセミコロンを使って区切ります。一般的に空白類文字は無視されますが、空白類文字が有意となるケースがいくつかあります。例えば、空白類文字は演算子の前後に必要です。

ファイル名やフォーマット名等の文字列は直接入力できます。コンマ (通常コンマはファイル名の区切り文字として使われる) 等の文字を含むファイル名は、二重引用符で囲む事で指定できます。ファイル名の中で二重引用符文字を使う事はできません。

コメントは C コードと同じ方法 (/* と */ で囲む) で挿入できます。C コードと同様に、コメントは構文的に空白類と同じです。

8.6.1 リンカスクリプトの基本概念

リンカは複数の入力ファイルから 1 つの出力ファイルを生成します。出力ファイルと各入力ファイルは特別なデータ フォーマットを持ちます (ELF オブジェクト ファイル フォーマットと呼ぶ)。各ファイルはオブジェクト ファイルと呼びます。各オブジェクト ファイルの主な内容はセクションのリストです。入力ファイル内のセクションは入力セクションと呼び、出力ファイル内のセクションは出力セクションと呼びます。

オブジェクト ファイル内の各セクションは名前とサイズを持ちます。大部分のセクションにはデータのブロック (セクション コンテンツと呼ぶ) が関連付けられます。セクションはロード可能 (loadable) として指定できます。これは、出力ファイルの実行時にコンテンツがメモリに書き込まれるという事を意味します。コンテンツを持たないセクションは割り当て可能 (allocatable) にできます。これは、メモリ内に領域は予約されるものの、そこには何も書き込まれないという事を意味します (場合によっては、このメモリ領域は 0 で埋める必要があります)。

ロード可能および割り当て可能な出力セクションは 2 つのアドレスを持ちます。1 つは VMA (Virtual Memory Address) です。出力ファイルの実行時にセクションはこのアドレスを持ちます。もう 1 つは LMA (Load Memory Address) です。これは、セクションが書き込まれる位置のアドレスです。ほとんどの場合、これら 2 つのアドレスは同じです。これらのアドレスが異なる例として、セクションに RAM 配置関数を格納する場合があります (例 : 既定値の .ramfunc セクション)。この場合、プログラムメモリ アドレスは LMA であり、データメモリ アドレスは VMA です。

> **Note:** VMA と LMA はどちらも PIC32 MCU の仮想アドレスを使います。PIC32MX の仮想 - 物理メモリ間の固定マッピングについては『PIC32MX ファミリ リファレンス マニュアル、セクション 03. メモリ構成』(DS61115) を参照してください。このファミリ リファレンス マニュアルには PIC32 のメモリレイアウトも記載しています。

オブジェクト ファイル内にあるセクションは、xc32-objdump プログラムでオプション -h を使って見る事ができます。

全てのオブジェクト ファイルはシンボルのリスト (シンボルテーブル) も格納します。シンボルは定義済みであっても未定義であっても構いません。各シンボルは名前を持ち、定義済みの各シンボルはアドレス等の情報も持ちます。C プログラムをオブジェクト ファイルへコンパイルすると、全ての定義済み関数とグローバルまたは静的変数に対して定義済みシンボルが生成されます。入力ファイル内で参照される全ての未定義関数または未定義グローバル変数は未定義シンボルになります。

オブジェクト ファイル内のシンボルは、xc32-nm プログラムを使うか、xc32-objdump プログラムでオプション -t を使う事によって閲覧できます。

8.6.2 ファイルを扱うコマンド

各種のリンカスクリプト コマンドはファイルを扱います。

INCLUDE filename

リンカスクリプト (filename) をその時点でインクルードします。このファイルは、現在作業中のディレクトリ内と -L オプションで指定された全てのディレクトリ内で検索されます。INCLUDE の呼び出しは 10 段までネストできます。

INPUT(file, file, ...)
INPUT(file file ...)

INPUTコマンドは、指定したファイル(file)をリンクに含めるようリンカに指示します。これは、コマンドラインでそれらのファイルを指定したのと同じ効果を持ちます。リンカは、最初に現在作業中のディレクトリ内でこれらのファイルを検索します。ファイルが見つからない場合、リンカはアーカイブ ライブラリの検索パス内でファイルを検索します。-L オプションの説明 (**7.4.9「--library-path <dir> (-L <dir>)」**) を参照してください。

INPUT (-lfile) を使うと、xc32-ld はファイル名を libfile.a に変換します。これはコマンドライン引数 -l を指定した場合と同じです。

暗黙的にインクルードされるリンカスクリプト内に INPUT コマンドが現れる場合、ファイルはそのリンカスクリプト ファイルがインクルードされた時点でリンクにインクルードされます。これはアーカイブの検索に影響します。

GROUP(file, file, ...)
GROUP(file file ...)

GROUPコマンドはINPUTに似ていますが、指定するファイル(file)が全てアーカイブであるという点で異なります。それらのファイルは、新たな未定義参照が生成されなくなるまで繰り返し検索されます。**7.4.1「-(archives -), --start-group archives, --end-group」**内のアーカイブの説明を参照してください。

OPTIONAL(file, file, ...)
OPTIONAL(file file ...)

OPTIONALコマンドもINPUTコマンドに似ていますが、指定されたファイル(file)が見つからなくてもリンク処理は継続するという点で異なります。このコマンドは、コンパイラと一緒にインストールされたかどうか不確かなアーカイブ (またはライブラリ) を指定する場合に特に便利です。XC32 コンパイラと一緒に提供される既定値リンカスクリプトは、この OPTIONAL ディレクティブを使ってデ、バイスに固有の周辺モジュール ライブラリをリンクします。

OUTPUT(filename)

OUTPUT コマンドは出力ファイルに名前を付けます。リンカスクリプト内の OUTPUT(filename) コマンドは、コマンドライン上の -o filenameと全く同じ効果を持ちます (**7.4.13「--output file (-o file)」**参照)。両方を使った場合、コマンドライン オプションの方が優先されます。

SEARCH_DIR(path)

SEARCH_DIRコマンドは、リンカがアーカイブ ライブラリを検索するために使うパスのリストにpathを追加します。SEARCH_DIR(path) はコマンドライン上の -L path と全く同じ効果を持ちます (**7.4.9「--library-path <dir> (-L <dir>)」**参照)。両方を使った場合、リンカは両方のパスを検索しますが、コマンドライン オプションで指定されたパスを最初に検索します。

STARTUP(filename)

STARTUP コマンドは INPUT コマンドとほとんど同じですが、指定されたファイル (filename) が入力ファイルの中で最初にリンクされるという点で異なります。これは、コマンドラインでそのファイルを最初に指定したのと同じ効果を持ちます。

8.6.3 シンボルへの値の代入

リンカスクリプト内でシンボルに値を代入する事ができます。これにより、そのシンボルはグローバル シンボルとして定義されます。

8.6.3.1 単純な代入

シンボルに対する値の代入には、以下の C 代入演算子が使えます。

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

最初のケースは、シンボルを expression の値に定義します。その他のケースでは、シンボルが定義済みである事が必要です。シンボルの値は定義済みの値に応じて決まります。

特殊なシンボル名「.」は、そのシンボルがロケーション カウンタである事を示します。このシンボルは SECTIONS コマンドの中でだけ使えます。

式 (expression) の後にセミコロンが必要です。

式の定義は **8.7「リンカスクリプト内の式」**に記載しています。

シンボルの代入は、シンボルの右側に書かれたコマンドとして、または SECTIONS コマンドの中の命令文として、あるいは SECTIONS コマンドの中の出力セクション記述の一部として現れます。

シンボルのセクションは式のセクションから設定されます。詳細は **8.7.6「式のセクション」**を参照してください。

以下のサンプルコードに、シンボルの代入が使える 3 通りの位置を示します。

```
floating_point = 0;
SECTIONS
{
  .text ORIGIN(kseg0_program_mem)  :
  {
    _text_begin = .;
    *(.text .stub .text.*)
    _text_end = .;
  } >kseg0_program_mem =0
   _bdata = (.+ 3) & ~ 3;
  .data :{ *(.data) }
}
```

この例では、シンボル floating_point は 0 として定義されます。シンボル _text_end は、最後の .text 入力セクションの直後のアドレスとして定義されます。シンボル _bdata は、4 バイト境界に対して上向きにアラインメントした .text 出力セクションの直後のアドレスとして定義されます。

8.6.3.2 PROVIDE

場合によっては、参照されている未定義のシンボル (リンクにインクルードされるどのオブジェクトによっても定義されていないシンボル) だけをリンカスクリプト内で定義する事が望まれます。例えば、従来のリンカはシンボル etext を定義しました。しかし ANSI C の要件によると、etext は関数名としてのみ使う事ができます (その他に使った場合、エラーが発生します)。PROVIDE キーワードを使うと、参照されているにもかかわらず未定義であるシンボル (etext 等) だけを定義する事ができます。その構文は PROVIDE(symbol = expression) です。

以下に、PROVIDE を使って etext を定義する場合の例を示します。

```
   SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

PIC32 の既定値リンカスクリプトは PROVIDE コマンドを使って既定値の _min_stack_size、_min_heap_size、_vector_spacing シンボル値を定義します。

```
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
PROVIDE(_vector_spacing = 0x00000001);
```

8.6.4 MEMORY コマンド

既定値コンフィグレーションでは、本リンカは全ての利用可能メモリを割り当てる事ができます。これは MEMORY コマンドを使って変更できます。

MEMORY コマンドは、ターゲット デバイス内のメモリブロックの位置とサイズを指定します。このコマンドにより、リンカが使っても良いメモリ領域とリンカが使ってはならないメモリ領域を指定する事ができます。これにより、セクションは特定のメモリ領域内に割り当てられます。リンカは、指定されたメモリ領域に基づいてセクションのアドレスを設定し、領域が不足する場合は警告します。リンカは、セクションの順番を変更して利用可能領域内にそれらを収めようとは試みません。

MEMORY コマンドの構文は以下の通りです。

```
  MEMORY
{
  name [(attr)] :ORIGIN = origin, LENGTH = len
  ...
}
```

name は、リンカスクリプト内でその領域を参照するために使う名前です。この領域名は、リンカスクリプトの外では意味を持ちません。領域名は専用の名前空間に保存されるためシンボル名、ファイル名、セクション名と競合しません。各メモリ領域は一意の名前を持つ必要があります。

attr 文字列には以下の文字だけが使えます。

R	読み出し専用セクション
W	読み書き可能セクション
X	実行可能セクション
A	割り当て可能セクション
I	初期化セクション
L	I と同じ
!	後続の全ての属性の意味を反転します。

未マッピングのセクションが *attr* 内の！を除くいずれかの属性に適合する場合、そのセクションはそのメモリ領域内に配置されます。! 属性は、この照合を反転します。すなわち、未マッピングのセクションは、*attr* 内のどの属性にも適合しない場合にのみメモリ領域内に配置されます。

origin はメモリ領域の開始アドレスを表す式です。この式は、メモリ割り当ての実行前に定数に対して評価する必要があります。従ってセクション相対シンボルは使えません。キーワード ORIGIN には短縮形 (org または o) が使えます (しかし ORG は使えません)。

len は、メモリ領域のサイズ (バイト数) を表す式です。origin と同様に、この式もメモリ割り当ての実行前に定数に対して評価する必要があります。キーワード LENGTH には短縮形 (len または l) が使えます。

以下の例では、2 つのメモリ領域 (0 から始まる 48K バイトと、0x800 から始まる 2K バイト) が割り当て可能であると指定します。リンカは、メモリ領域に明示的にマッピングされていない「読み出し専用 (r)」または「実行可能 (x)」属性を持つ各セクションを rom メモリ領域内に配置します。リンカは、明示的にマッピングされていない他のセクションを ram メモリ領域内に配置します。

```
  MEMORY
{
  rom (rx)  :ORIGIN = 0, LENGTH = 48K
  ram (!rx) : org = 0x800, l = 2K
}
```

メモリ領域が定義されると、>region 出力セクション属性を使って特定の出力セクションをそのメモリ領域内に配置するようリンカに指示する事ができます。例えば、memという名前のメモリ領域を指定するには、出力セクションの定義内で>memを使います。出力セクションに対してアドレスが指定されなかった場合、リンカはそのアドレスをメモリ領域内で次に利用可能なアドレスに設定します。結合された出力セクション (出力セクションの合計したサイズ) が割り当て先のメモリ領域に対して大きすぎる場合、リンカはエラーメッセージを出力します。

8.6.5 SECTIONS コマンド

SECTIONS コマンドは、入力セクションを出力セクションにマッピングする方法と、出力セクションをメモリ内に配置する方法をリンカに指示します。

SECTIONS コマンドの書式は以下の通りです。

```
   SECTIONS
{
  sections-command
  sections-command
  ...
}
```

各 SECTIONS コマンドは以下の中のどれかです。

- ENTRY コマンド (**8.6.6「その他のリンカスクリプト コマンド」**参照)
- シンボル代入 (**8.6.3「シンボルへの値の代入」**参照)
- 出力セクション記述
- オーバーレイ記述

ENTRY コマンドとシンボル代入は SECTIONS コマンドの中で使えます。これにより、それらのコマンド内でロケーション カウンタが使いやすくなります。また、それらのコマンドは出力ファイルのレイアウト内の意味のある位置で使われるため、リンカスクリプトが読みやすくなります。

次に、出力セクション記述とオーバーレイ記述について説明します。

リンカスクリプト内に SECTIONS コマンドが現れない場合、リンカは各入力セクションを同じ名前の出力セクションに配置します。配置する順番は、入力ファイルの中でセクションが最初に現れる順番に基づきます。例えば、全ての入力セクションが最初のファイル内に現れる場合、出力ファイル内のセクションの順番は、最初の入力ファイル内での順番と同じです。最初のセクションはアドレス 0 に配置されます。

8.6.5.1 入力セクション記述

最も一般的に使われる出力セクション コマンドは入力セクション記述です。

入力セクション記述は、最も基本的なリンカスクリプト動作です。出力セクションは、メモリ内にプログラムをレイアウトする方法をリンカに指示します。入力セクション記述は、メモリレイアウトに対して入力ファイルをマッピングする方法をリンカに指示します。

入力セクション記述には 1 つのファイル名が含まれます。オプションとして、ファイル名の後にカッコで囲まれたセクション名のリストが続きます。

ファイル名とセクション名にはワイルドカード パターンが使えます。これについては後で説明します。

最も一般的な入力セクション記述は、特定の名前を持つ全ての入力セクションを出力セクションにインクルードします。例えば、全ての入力 .text セクションをインクルードする場合、以下のように書けます。

```
   *(.text)
```

* は、全てのファイル名に一致するワイルドカードです。ワイルドカードに一致するファイルから特定ファイルを除外するには EXCLUDE_FILE を使います。EXCLUDE_FILE リスト内で指定されたファイルを除く全てのファイルがワイルドカードに一致します。
例 :

```
   *(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```

この場合、crtend.o と otherfile.o を除く全てのファイルから全ての .ctors セクションがインクルードされます。

複数のセクションをインクルードするには、以下の 2 通りの方法があります。

```
   *(.text .rodata)
   *(.text) *(.rodata)
```

これら 2 つの方法では、.text および .rodata 入力セクションが出力セクション内に現れる順番が異なります。最初の例の場合、.text 入力セクションと .rodata 入力セクションは入り乱れて現れます。2 番目の例の場合、最初に全ての .text 入力セクションが現れた後に、全ての .rodata 入力セクションが現れます。

特定の 1 つのファイルからセクションをインクルードする場合、ファイル名を 1 つだけ指定します。これは、メモリ内の特定位置に配置する必要がある特殊なデータをファイルが含んでいる場合に便利です。
例 : data.o(.data)

セクションのリストを指定せずにファイル名を指定した場合、その入力ファイル内の全てのセクションが出力セクションにインクルードされます。この方法は通常使いませんが、便利な場合もあります。
例 : data.o

ワイルドカード文字を含まないファイル名が指定された場合、リンカは最初にコマンドラインまたは INPUT コマンド内でそのファイル名が指定されたどうか調べます。指定されていなかった場合、リンカは、それがコマンドラインで指定されたかのように、そのファイルを入力ファイルとして開こうと試みます。これは INPUT コマンドとは異なります。なぜなら、リンカはアーカイブ検索パス内でそのファイルを検索するのではないからです。

8.6.5.2 入力セクションのワイルドカード パターン

入力セクション記述内では、ファイル名またはセクション名 (もしくはその両方) にワイルドカード パターンが使えます。

多くの例でファイル名に使われる「*」は、単純なワイルドカード パターンです。

ワイルドカード パターンは、UNIX シェルで使われる物と同様です。

*	任意の文字列 (文字数は問わない) と一致
?	任意の 1 文字と一致
[*chars*]	*chars* に含まれている文字の中のどれか 1 文字と一致 (ハイフン「-」を使って文字の範囲を指定できます。例えば、[a-z] は小文字アルファベットの全てと一致します)
\	直後の文字と一致

ファイル名をワイルドカードを使って照合する時、ワイルドカード文字は「/」(UNIX でディレクトリ名の区切り用に使われる) に一致しません。1 個の * 文字だけを含むワイルドカード パターンは例外です。これは全てのファイル名 (ファイル名が「/」を含んでいても) に常に一致します。セクション名の場合、ワイルドカード文字は「/」文字に一致します。

ファイル名のワイルドカード パターンは、コマンドラインまたは INPUT コマンドで明示的に指定されたファイルにのみ一致します。本リンカは、ワイルドカードを拡張してディレクトリを検索しません。

あるファイル名が複数のワイルドカード パターンに一致する場合、または明示的に指定されているファイル名がワイルドカード パターンにも一致する場合、リンカはリンカスクリプト内の最初の一致を使います。例えば、以下の一連の入力セクション記述は、data.o ルールが使われないため、恐らくエラーが発生します。

```
.data  :{ *(.data) }
.data1 :{ data.o(.data) }
```

通常、リンカは、ワイルドカードに一致したファイルとセクションを、リンク中にそれらが現れた順番に配置します。この順番は、カッコで囲まれたワイルドカード パターンの前で SORT キーワードを使う事によって変更できます (例 : SORT(.text*))。SORT キーワードを使った場合、リンカはファイルまたはセクションを名前によって昇順に並び換えてから出力ファイル内に配置ます。

入力セクションの配置先を確認するには、リンカ オプション -M を使ってマップファイルを生成します。マップファイルは、入力セクションがどのように出力ファイルにマッピングされるのかを詳細に示します。

以下の例に、ワイルドカード パターンを使ってファイルを分配する方法を示します。このリンカスクリプトは、全ての .text セクションを .text 内に配置し、全ての .bss セクションを .bss 内に配置するようリンカに指示します。リンカは、大文字で始まるファイル名を持つ全てのファイルからの .data セクションを .DATA 内に配置し、その他のファイルからの .data セクションを .data 内に配置します。

```
   SECTIONS {
  .text :{ *(.text) }
  .DATA :{ [A-Z]*(.data) }
  .data :{ *(.data) }
  .bss :{ *(.bss) }
}
```

8.6.5.3 共有シンボルの入力セクション

共有シンボルは特定の入力セクションを持たないため、特別な注意が必要です。本リンカは、共有シンボルを、それらが COMMON という名前の入力セクション内にあるかのように扱います。

他の入力セクションと同様に、COMMON セクションでもファイル名が使えます。これにより、特定入力ファイルからの共有シンボルは 1 つのセクション内に配置され、他の入力ファイルからの共有シンボルは別のセクション内に配置されます。

ほとんどの場合、入力ファイル内の共有シンボルは、出力ファイル内の .bss セクション内に配置されます。
例：`.bss { *(.bss) *(COMMON) }`

特に指定がない限り、共有シンボルは .bss セクションに割り当てられます。

8.6.5.4 入力セクションの例

以下の例に、完全なリンカスクリプトを示します。このスクリプトは、ファイル all.o から全てのセクションを読み出し、それらをアドレス 0x10000 から始まる出力セクション outputa の先頭に配置するようリンカに指示します。同じ出力セクション内で、その直後にファイル foo.o からの全てのセクション .input1 が配置されます。foo.o からの全てのセクション .input2 は出力セクション outputb 内に配置され、その後に foo1.o からのセクション .input1 が続きます。全てのファイル内の残りの .input1 および .input2 セクションは、出力セクション outputc に書き込まれます。

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

8.6.5.5　出力セクション記述

出力セクションの完全な記述を以下に示します。

```
   name [address] [(type)] :[AT(lma)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [=fillexp]
```

大部分の出力セクションは、オプションのセクション属性をほとんど使いません。

name と *address* の前後には空白類が必要です。コロンと波カッコ (中カッコ) も必要です。改行とその他の空白類は任意に使えます。

セクション名には任意の文字の並びが使えますが、コンマ等の記号を含む名前は引用符で囲む必要があります。

各出力セクション コマンドは以下の中のどれかです。

- シンボルの代入 (**8.6.3「シンボルへの値の代入」**参照)
- 入力セクション記述 (**8.6.5.1「入力セクション記述」**参照)
- ディレクトリを含めるためのデータ値 (**8.6.5.7「出力セクション データ」**参照)

8.6.5.6　出力セクションのアドレス

address は、出力セクションの VMA (仮想メモリアドレス) を表す式です。アドレスが指定されない場合、リンカは *region* に基づいてアドレスを設定し、*region* が指定されない場合はロケーション カウンタのその時点の値に基づいてアドレスを設定します。

address が指定された場合、出力セクションのアドレスはその通りに設定されます。*address* も *region* も指定されない場合、出力セクションのアドレスは、出力セクションのアラインメント要件に合わせたロケーション カウンタのその時点の値に設定されます。出力セクションのアラインメント要件は、その出力セクションに格納される全ての入力セクションの最も厳格なアラインメント要件です。

以下に例を示します。

```
   .text .:{ *(.text) }
   .text :{ *(.text) }
```

上記の 2 つは微妙に異なります。最初の例は、.text 出力セクションのアドレスをロケーション カウンのその時点の値に設定します。2 番目の例は、このアドレスを .text 入力セクションの最も厳格なアラインメントに合わせたロケーション カウンタのその時点の値に設定します。

address には任意の式が使えます (**8.7「リンカスクリプト内の式」**参照)。例えば、セクションを 0x10 バイト境界に配置する (セクション アドレスの最下位 4 ビットは 0) 場合、以下のコマンドが使えます。

```
   .text ALIGN(0x10) :{ *(.text) }
```

これは正しく機能します。なぜなら、ALIGN は、指定された値に対して上向きにアラインメントされたロケーション カウンタのその時点の値を返すからです。

セクションに対して *address* を指定すると、ロケーション カウンタの値は変更されます。

8.6.5.7 出力セクション データ

BYTE、SHORT、LONG、QUAD を出力セクション コマンドとして使う事により、決められたバイト数のデータを出力セクションに挿入できます。各キーワードの後にカッコで囲んだ式を書く事で、保存する値を指定します。式の値は、その時点のロケーション カウンタ値が指す位置に保存されます。

BYTE、SHORT、LONG、QUAD コマンドはそれぞれ 1、2、4、8 バイトを保存します。例えば、以下のコマンドはシンボル addr の 4 バイト値を保存します。

```
LONG(addr)
```

これらのバイトを保存した後に、ロケーション カウンタは保存したバイト数分だけインクリメントします。プログラムメモリ セクション内でデータコマンドを使う場合、リンカはプログラムメモリが 32 ビット幅であると見なすという事に注意が必要です (プログラムメモリは物理的には 24 ビット幅で実装されます)。従って、LONG データ値の最上位 8 ビットはデバイスメモリに書き込まれません。

データコマンドは 1 つのセクション記述内でのみ機能し、異なるセクション記述との間では機能しません。このため、以下のコードに対してリンカはエラー出力します。

```
SECTIONS { .text :{ *(.text) } LONG(1) .data :{ *(.data) } }
```

以下のコードは機能します。

```
SECTIONS { .text :{ *(.text) ; LONG(1) } .data :{ *(.data) } }
```

FILL コマンドを使うと、現在作業中のセクションに対するフィルパターン (ギャップ等を埋めるために書き込むデータのパターン) を設定できます。このコマンドの後にカッコで囲んだ式を書きます。セクション内の未指定の領域 (例 : 入力セクションのアラインメント要件のために残されたギャップ) には、式の最下位 2 バイトが必要なだけ繰り返して書き込まれます。FILL 命令文は、セクション定義内でこの命令文が現れた位置から後のメモリ位置に適用されます。FILL 命令文を複数回使う事で、出力セクションの部分ごとに異なるフィルパターンを適用できます。

以下の例に、メモリの未指定領域を値 0x9090 で埋める方法を示します。

```
FILL(0x9090)
```

FILL コマンドは =fillexp 出力セクション属性 (**8.6.5.9「出力セクションの属性」**参照) に似ていますが、セクションの全体ではなく FILL コマンドから後のセクション部分にのみ影響するという点で異なります。両方が使われた場合、FILL コマンドが優先されます。

8.6.5.8 出力セクションの破棄

本リンカは、内容を全く含まない出力セクションを生成しません。これは、いずれかの入力ファイルの中に存在するかどうか不確かな入力セクションを参照する際に便利です。

例 : `.foo { *(.foo) }`

これは、少なくとも 1 つの入力ファイル内に .foo セクションが存在すれば、出力ファイル内に .foo セクションを生成します。

入力セクション記述以外の何か (シンボル代入等) が出力セクション コマンドとして使われた場合、出力セクションは常に (該当する入力セクションが存在しなくても) 生成されます。

特別な出力セクション名 /DISCARD/ を使うと、入力セクションを破棄できます。出力セクション名 /DISCARD/ に割り当てられた全ての入力セクションは、出力ファイルにインクルードされません。

8.6.5.9　出力セクションの属性

出力セクションの完全な記述は以下により調べる事ができます。

```
name [address] [(type)] :[AT(lma)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...][=fillexp]
```

name、*address*、output-section-command については既に説明しました。以下では、その他のセクション属性について説明します。

8.6.5.10　出力セクションのタイプ

各出力セクションにはタイプ (*type*) を指定できます。タイプはカッコで囲んだキーワードです。以下のタイプが定義されています。

NOLOAD

セクションをロード不可 (not loadable) として指定します。そのセクションはプログラム実行時にメモリに書き込まれません。

DSECT、COPY、INFO、OVERLAY

これらのタイプ名は、旧式の MIPS および GNU アセンブラとの下位互換性を維持するためにサポートされますが、実際にはほとんど使いません。これらは全て同じ効果を持ち、セクションを割り当て不可 (not allocatable) として指定します。これを指定されたセクションには、プログラム実行時にメモリが割り当てられません。

通常、本リンカは、出力セクションにマッピングされた入力セクションに基づいて出力セクションの属性を設定します。この属性はセクションタイプを使って上書きできます。例として以下のサンプル スクリプトでは、ROM セクションはメモリ位置 0 にアドレス設定され、プログラム実行時にロードされる必要はありません。ROM セクションの内容は、通常通りにリンカ出力ファイル内に現れます。

```
SECTIONS {
  ROM 0 (NOLOAD) :{ ...}
  ...
}
```

8.6.5.11 出力セクション LMA

各セクションは仮想アドレス (VMA) とロードアドレス (LMA) を持ちます。出力セクション記述内に現れるアドレス式が VMA を設定します。

通常本リンカは、LMA を VMA と同じアドレスに設定します。これは AT キーワードを使って変更できます。AT キーワードに続く式 lma により、そのセクションのロードアドレスを指定します。あるいは式 AT>lma_region により、そのセクションのロードアドレスに対してメモリ領域を指定します。**8.6.4「MEMORY コマンド」**を参照してください。

この機能は、ROM イメージのビルドを容易にする事を目的とします。例として以下のリンカスクリプトは 3 つの出力セクションを生成します。1 つは .text と呼ぶセクションです。このセクションは 0xBFC00000 から始まります。2 つ目は .mdata と呼ぶセクションです。このセクションは、VMA が 0xA0000000 であるにもかかわらず .text セクションの終端に書き込まれます。3 つ目は .bss と呼ぶセクションです。このセクションはアドレス 0xA0001000 で非初期化データを保持します。シンボル _data は値 0xA0000000 により定義されます。これは、ロケーション カウンタが LMA 値ではなく VMA 値を保持するという事を示します。

```
SECTIONS
  {
  .text 0xBFC00000:{ *(.text) _etext = .; }
  .mdata 0xA0000000:
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = .; *(.data); _edata = .;  }
  .bss 0xA0001000:
    { _bstart = .;  *(.bss) *(COMMON) ; _bend = .;}
}
```

このリンカスクリプトを使って生成したプログラム向けに使用するランタイム初期化コードは、初期化データを ROM イメージからランタイム アドレスにコピーするための関数を含みます。この初期化関数は、リンカスクリプトにって定義されたシンボルを利用できます。

しかし、そのような関数を書く必要はほとんどありません。これらの関数は、C コンパラのスタートアップおよび初期化コードによって提供されます。コンパイラが提供するスタートアップ コードの詳細は『MPLAB® C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。スタートアップ ルーチンのアセンブリ ソースコードは \pic32-libs\c\startup\crt0.S にあります。

8.6.5.12 出力セクションの領域

セクションは、>region を使って先に定義済みのメモリ領域に割り当てる事ができます。**8.6.4「MEMORY コマンド」**を参照してください。

以下に簡単な例を示します。

```
MEMORY { rom :ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM :{ *(.text) } >rom }
```

8.6.5.13 出力セクションのフィル

フィルパターンは、=fillexp を使ってセクション全体に適用できます。fillexp は式として使います。出力セクション内の未指定の領域 (例 : 入力セクションのアラインメント要件のために残されたギャップ) には、値の最下位 2 バイトが必要なだけ繰り返して書き込まれます。

書き込む値は、出力セクション コマンド内で FILL コマンドを使って変更する事もできます (**8.6.5.7「出力セクション データ」**参照)。

以下に簡単な例を示します。

```
   SECTIONS { .text :{ *(.text) } =0x9090 }
```

8.6.5.14　オーバーレイ記述

オーバーレイ記述は、単一メモリイメージの一部としてロードされるものの全て同じメモリアドレスで実行される複数のセクションを記述するための容易な方法を提供します。実行時に、ある種のオーバーレイ マネージャは、要求に応じて、オーバーレイされたセクションをランタイム メモリアドレスにコピー (出し入れ) します。多くの場合、これは単純にアドレス指定ビットを操作する事により行います。

オーバーレイは OVERLAY コマンドを使って記述します。OVERLAY コマンドは、出力セクション記述と同様に、SECTIONS コマンドの中で使います。OVERLAY の完全な構文を以下に示します。

```
   OVERLAY [start] :[NOCROSSREFS] [AT ( ldaddr )]
  {
    secname1
      {
        output-section-command
        output-section-command
        ...
      } [:phdr...][=fill]
    secname2
      {
        output-section-command
        output-section-command
        ...
      } [:phdr...][=fill]
    ...
  } [>region] [:phdr...][=fill]
```

OVERLAY(キーワード) 以外は全てオプションです。各セクションは名前 (secname1、secname2、...) を持つ必要があります。OVERLAY 構造体の中のセクション定義は、一般的な SECTIONS 構造体の中のセクション定義と基本的に同じですが、OVERLAY 内ではセクションに対してアドレスもメモリ領域も定義されないという点で異なります。

全てのセクションは同じ開始アドレスで定義されます。セクションのロードアドレスは、OVERLAY 向けに使われるロードアドレスから始まるメモリ内で全体として連続するよう配置されます (通常のセクション定義と同様に、ロードアドレスはオプションであり、既定値では開始アドレスに設定されます。開始アドレスもオプションであり、既定値ではロケーション カウンタのその時点の値に設定されます)。

セクション間に参照が存在しない場合、NOCROSSREFS キーワードが使われるとリンカはエラーを出力します。全てのセクションは同じアドレスで実行されるため、あるセクションが別のセクションを直接参照しても通常は意味を成しません。

OVERLAY 内の各セクションに対し、リンカは自動的に 2 つのシンボルを定義します。シンボル __load_start_secname は、そのセクションの開始ロードアドレスとして定義されます。シンボル __load_stop_secname は、そのセクションの最後のロードアドレスとして定義されます。C 識別子用に使えない文字が secname に含まれている場合、それらの文字は削除されます。C (またはアセンブラ) コードは、必要に応じてこれらのシンボルを使う事で、オーバーレイされたセクションの間を移動できます。

オーバーレイの最後で、ロケーション カウンタの値は [オーバーレイの開始アドレス + 最大セクションのサイズ] に設定されます。

以下に例を示します。これは SECTIONS 構造体の中に現れるという事に注意が必要です。

```
  OVERLAY 0x9D001000 :AT (0xA0004000)
   {
     .text0 { o1/*.o(.text) }
     .text1 { o2/*.o(.text) }
   }
```

これは .text0 と .text1 の両方が 0x9D001000 で始まるよう定義します。.text0 はアドレス 0x9D001000 でロードされ、.text1 は .text0 の直後にロードされます。以下のシンボルが定義されます。
__load_start_text0、__load_stop_text0、__load_start_text1、__load_stop_text1

オーバーレイ .text1 をオーバーレイ領域にコピーする C コードの例を以下に示します。

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x9D001000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

OVERLAYコマンドは便宜を図るために用意されています(このコマンドが実行する全ての機能は、より基本的なコマンドを使って実行可能です)。上の例は以下のように書き換える事ができます。

```
.text0 0x9D001000:AT (0x9D004000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x9D001000:AT(0x9D004000+SIZEOF(.text0))
{o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
.= 0x9D001000+ MAX (SIZEOF (.text0), SIZEOF (.text1));
```

8.6.6 その他のリンカスクリプト コマンド

以下では、その他のリンカスクリプト コマンドについて簡潔に説明します。

ASSERT(*exp*, *message*)

*exp*が非0である事を確認します。これが0である場合、エラーコードと*message*を出力してリンカを終了します。

ENTRY(*symbol*)

symbol を、プログラム実行の最初の命令として指定します。リンカは、このシンボルのアドレスを出力オブジェクト ファイルのヘッダに記録します。これはアドレス 0 に配置されたリセット命令には影響しません。リセット命令は、別の方法で生成する必要があります。慣例により、32 ビット リンカスクリプトは GOTO __reset 命令をアドレス 0 に配置します。

EXTERN(*symbol symbol ...*)

symbol を未定義シンボルとして強制的に出力ファイルに含めます。これにより、例えば、標準ライブラリから追加のモジュールをリンクさせる事ができます。各 EXTERN に対して複数のシンボルを指定できます。EXTERN は何度でも使えます。このコマンドはコマンドライン オプション -u と同じ効果を持ちます。

FORCE_COMMON_ALLOCATION

このコマンドは、コマンドライン オプション -d と同じ効果を持ちます。このコマンドを使うと、たとえ再配置可能出力ファイルが -r によって指定されていても、32 ビットリンカは空間を共有シンボルに割り当てます。

NOCROSSREFS(*section section ...*)

このコマンドは、特定出力セクションの間の参照に関してエラーを発行するよう 32 ビットリンカに指示します。特定タイプのプログラムでは、あるセクションがメモリにロードされる時に別のセクションはロードされません。2 つのセクション間の直接参照はエラーになります。

NOCROSSREFS コマンドは出力セクションの名前のリストを指定します。リンカは、それらのセクション間の相互参照を検出すると、エラーを報告して非 0 の終了ステータスを返します。NOCROSSREFS コマンドは入力セクションではなく出力セクションの名前を使います。

OUTPUT_ARCH(*bfdarch*)

特定の出力マシン アーキテクチャを指定します。Microchip PIC32 MCU の場合、bfdarch 値は常に pic32mx です。

OUTPUT_FORMAT(*format_name*)

OUTPUT_FORMAT コマンドは、出力ファイル用に使うオブジェクト ファイルのフォーマットを指定します。Microchip PIC32 MCU の場合、*format_name* 値は常に elf32-tradlittlemips です。

TARGET(*format_name*)

TARGET コマンドは、入力ファイルの読み出し時に使うオブジェクト ファイルのフォーマットを指定します。これは後続の INPUT および GROUP コマンドに影響します。Microchip PIC32 MCU の場合、*format_name* 値は常に elf32-tradlittlemips である事が必要です。

8.7 リンカスクリプト内の式

リンカスクリプト言語の式の構文は C 式の構文と同じです。全ての式は 32 ビット整数として評価されます。

式の中でシンボル値を使用および設定できます。

本リンカは、式の中で使える各種の特殊用途ビルトイン関数を定義します。

8.7.1 定数

全ての定数は整数です。

C 言語と同様に、本リンカは `0` で始まる整数を 8 進数と見なし、`0x` または `0X` で始まる整数を 16 進数と見なします。その他の整数は 10 進数と見なします。

加えて、定数をスケーリングするための接尾辞 `K` (1024 倍) と `M` (1024x1024 倍) が使えます。例として、以下は全て同じ値を参照します。

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

8.7.2 シンボル名

引用符で囲まないシンボル名は文字 / アンダースコア / ピリオドのいずれかで始まり、文字 / 数字 / アンダースコア / ピリオド / ハイフンを含む事ができます。引用符で囲まないシンボル名はキーワードと競合しない事が必要です。特殊文字を含むシンボルまたはキーワードと同名のシンボルは、シンボル名を二重引用符で囲む事によって指定できます。

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

シンボルはアルファベット以外の文字を含む場合があるため、シンボルとシンボルの間はスペースで区切るのが最も安全です。例えば `A-B` は 1 つのシンボルですが、`A - B` は減算式です。

8.7.3 ロケーション カウンタ

特別なリンカ変数である「.」(ドット) は、常にロケーション カウンタのその時点の値を格納します。「.」は常に出力セクション内の位置を参照するため、SECTIONS コマンド内の式の中だけで使う事ができます。「.」シンボルは、通常のシンボルが使える位置であれば式内のどこででも使えます。

「.」に値を代入すると、ロケーション カウンタの位置が移動します。これにより、出力セクション内にギャップを生成できます。ロケーション カウンタは、絶対に後方へ移動させない事が必要です。

```
SECTIONS
{
  output :
    {
      file1(.text)
      .= .+ 1000;
      file2(.text)
      .+= 1000;
      file3(.text)
    } = 0x1234;
}
```

上の例では、file1 からの .text セクションは出力セクション output の先頭に配置され、その後に 1000 バイトのギャップが続きます。さらにその後に file2 からの .text セクション、1000 バイトのギャップ、file3 からの .text セクションが続いて配置されます。= 0x1234 は、ギャップに書き込むデータを指定します。

「.」は、現在格納中のオブジェクトの先頭からのバイトオフセットを参照します。通常これは開始アドレスが 0 の SECTIONS 命令文です。従って「.」は絶対アドレスとして使えます。しかし、「.」をセクション記述の中で使う場合、これはセクションの先頭からのバイトオフセット (絶対アドレスではない) を参照します。スクリプトを以下に示します。

```
SECTIONS
{
    .= 0x100
    .text:{
      *(.text)
      .= 0x200
    }
    .= 0x500
    .data:{
      *(.data)
      .+= 0x600
    }
}
```

.text セクションには開始アドレス 0x100 と 0x200 バイトのサイズが割り当てられます。.text 入力セクション内にこの領域を埋めるのに十分なデータが存在しなくても、このサイズが割り当てられます。データが多すぎる場合、「.」は後方に移動する事になるため、エラーが生成されます。.data セクションは 0x500 から始まり、余分な 0x600 バイト (.data 入力セクションからの値の終端から .data 出力セクション自体の終端までの空間) を含みます。

8.7.4 演算子

本リンカは、標準 C の算術演算子を標準の結合性と優先度で認識します。

表 8-2: 演算子の優先順位

優先順位	結合性	演算子	概要
1 (最高優先度)	左	!- ~	前置演算子
2	左	* / %	乗算、除算、剰余
3	左	+ -	加算、減算
4	左	>> <<	右ビットシフト、左ビットシフト
5	左	== != > < <= >=	大小比較
6	左	&	ビット単位の AND
7	左	|	ビット単位の OR
8	左	&&	論理 AND
9	左	||	論理 OR
10	右	?:	条件式
11 (最低優先度)	右	&= += -= *= /=	シンボル代入

8.7.5 評価

本リンカは式を lazily に評価します。すなわち、絶対に必要な時にだけ式の値を計算します。

どのようなリンクを実行する場合も、リンカは各種の情報 (最初のセクションの開始アドレスの値、メモリ領域の開始位置と長さ等) を必要とします。リンカは、リンカスクリプトを読み込んだ時に、可能な限り速やかにこれらの値を計算します。

しかし、他の値 (シンボル値等) は、記憶域の割り当てが済むまで未知または不要です。そのような値は、他の情報 (出力セクションのサイズ等) がシンボル代入式内で使えるようになった時点で評価されます。

セクションのサイズは割り当てが済むまで未知です。このため、これらに依存する代入は割り当て後に実行されます。

ロケーション カウンタ「.」に依存する式等は、セクション割り当て中に評価する必要があります。

式の結果が必要な時に値が利用できない場合、エラーとなります。
例 :

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

このスクリプトにより、エラーメッセージ「non-constant expression for initial address」が出力されます。

8.7.6 式のセクション

リンカが式を評価した時、その結果は何らかのセクションに対して絶対的か相対的です。相対的な式は、セクションのベースアドレスからの固定されたオフセットとして表現されます。

式が絶対的か相対的かは、リンカスクリプト内の式の位置によって決まります。出力セクションの定義内に現れる式は、その出力セクションのベースアドレスに対して相対的です。その他の場所に現れる式は絶対的です。

-rオプションを使って再配置可能出力を要求した場合､相対的な式に設定されたシンボルは再配置可能です。つまり、後続のリンク動作はそのシンボルの値を変更できます。そのシンボルのセクションは、相対的な式のセクションになります。

絶対的な式に設定されたシンボルの値は、後続のリンク動作によって変更されません (同じ値を保持します)。そのシンボルは絶対的となり、特定のどのセクションにも関連付けされません。

ビルトイン関数 ABSOLUTE を使うと、相対的な式を絶対的にできます。例えば、以下のスクリプトにより、出力セクション .data の終端アドレスに設定された絶対的シンボルを生成する事ができます。

```
SECTIONS
  {
    .data :{ *(.data) _edata = ABSOLUTE(.); }
  }
```

ABSOLUTE を使わない場合、_edata は .data セクションに対して相対的です。

8.7.7 ビルトイン関数

リンカスクリプト言語は、リンカスクリプトの式内で使う各種のビルトイン関数を含んでいます。

8.7.7.1 ABSOLUTE(*exp*)

式 *exp* の絶対的な (再配置不可という意味、非負の絶対値という意味ではない) 値を返します。この関数は、主にセクション定義内のシンボル (通常シンボル値はセクションに対して相対的) に絶対的な値を代入する場合に便利です。

8.7.7.2 ADDR(*section*)

指定されたセクションの絶対アドレス (VMA) を返します。そのセクションのアドレスはユーザのスクリプト内で先に定義済みである事が必要です。以下の例では、symbol_1 と symbol_2 に同じ値を代入します。

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
...
}
```

8.7.7.3 ALIGN(*exp*)

次の *exp* バイト境界にアラインメントされたロケーション カウンタ (.) を返します。*exp* は、値が 2 のべき乗値の式である事が必要です。これは以下と等価です。

```
(.+ exp - 1) & ~(exp - 1)
```

ALIGNはロケーション カウンタの値を変更しません(ロケーション カウンタ値を使って算術演算を行うだけです)。以下の例では、出力 .data セクションを前のセクションの次の 0x2000 バイト境界に配置し、そのセクション内の変数を入力セクションの次の 0x8000 境界に設定します。

```
SECTIONS { ...
  .data ALIGN(0x2000):{
    *(.data)
    variable = ALIGN(0x8000);
  }
...
}
```

この例の中で最初に現れる ALIGN はセクションの位置を指定します。なぜなら、これはセクション定義のオプションのアドレス属性として使われるからです (**8.6.5「SECTIONS コマンド」**参照)。次に現れる ALIGN は、シンボルの値を定義するために使います。

ビルトイン関数 NEXT は ALIGN と密接に関連します。

8.7.7.4 BLOCK(*exp*)

これは ALIGN と同義です (旧式のリンカスクリプトとの互換性のために用意されています)。これは、出力セクションのアドレスを設定する際に頻繁に使われます。

8.7.7.5 DEFINED(*symbol*)

symbol がリンカのグローバル シンボルテーブルに含まれかつ定義済みであれば「1」を返します。それ以外の場合、「0」を返します。この関数により、シンボルに既定値を与える事ができます。例として、グローバル シンボル begin を .text セクション内の先頭位置に設定する方法を以下のスクリプトに示します。begin という名前のシンボルが既に存在する場合、その値は維持されます。

```
SECTIONS { ...
  .text :{
    begin = DEFINED(begin) ? begin :.;
    ...
  }
  ...
}
```

8.7.7.6 KEEP(*section*)

リンク時ガベージ コレクションを使う場合 (--gc-sections)、削除すべきではないセクションをマーキングすると便利な事がよくあります。これを行うには、入力セクションのワイルドカード エントリを KEEP() で囲みます (KEEP(*(.init)) または KEEP(SORT_BY_NAME(*)(.ctors)))。

8.7.7.7 LOADADDR(*section*)

指定されたセクション (*section*) の絶対 LMA を返します。通常これは ADDR と同じですが、AT 属性が出力セクション定義内で使われている場合は異なる可能性があります (**8.6.5「SECTIONS コマンド」**参照)。

8.7.7.8 MAX(*exp1, exp2*)

exp1 と *exp2* の最大値を返します。

8.7.7.9 MIN(*exp1, exp2*)

exp1 と *exp2* の最小値を返します。

8.7.7.10 NEXT(*exp*)

exp の整数倍となる次の未割り当てアドレスを返します。この関数は ALIGN(*exp*) と等価です。

8.7.7.11 SIZEOF(*section*)

指定されたセクション (*section*) が割り当て済みであれば、そのサイズ (バイト数) を返します。これが評価された時にそのセクションが未割り当てであった場合、リンカはエラーを報告します。以下の例では、symbol_1 と symbol_2 に同じ値を代入します。

```
SECTIONS{ ...
  .output {
    .start = .;
    ...
    .end = .;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
...
}
```

NOTE:

第 9 章　リンカ処理

9.1　はじめに

MPLAB XC32 オブジェクト リンカ (xc32-ld) がどのように入力ファイルからアプリケーションをビルドするのか説明します。

第 9 章の主な内容は以下の通りです。

- リンカ処理の概要
- リンカ割り当て
- グローバル シンボルと weak シンボル
- 初期化データ
- スタックの割り当て
- ヒープの割り当て
- PIC32MX の割り込みベクタテーブル
- 専用プログラマブル変数オフセットを備えた PIC32　MCU 向けの割り込みベクタテーブル

9.2 リンカ処理の概要

リンカは 1 つまたは複数のオブジェクト ファイルとオプションのアーカイブ ファイルから 1 つの実行可能出力ファイルを生成します。オブジェクト ファイルはコードとデータの再配置可能セクションを格納します。リンカは、それらのコードとデータをターゲットメモリに割り当てます。リンカ処理の全てはリンカスクリプト (リンクコマンド ファイルとも呼ぶ) により制御されます。リンカスクリプトはどのリンク処理に必要です。リンク処理は以下の 5 ステップで構成されます。

1. 入力ファイルのロード
2. メモリ割り当て
3. シンボルの解決
4. 絶対アドレスの計算
5. 出力ファイルのビルド

9.2.1 入力ファイルのロード

リンカの最初のタスクは、リンクコマンド オプションを解釈して入力ファイルをロードする事です。リンカスクリプトが指定された場合、リンカはそのファイルを開いて内容を解釈します。リンカスクリプトが指定されなかった場合、リンカは既定値リンカスクリプトを使います。どちらの場合も、リンカスクリプトはターゲット デバイスの情報 (デバイスに固有のメモリ領域情報を含む) を提供します。詳細は**第 8 章「リンカスクリプト」**を参照してください。

次にリンカは全ての入力オブジェクト ファイルを開きます。リンカは、各入力ファイルのオブジェクト フォーマットの互換性を確認します。オブジェクト フォーマットが非互換である場合、リンカはエラーを生成します。互換性が確認された各入力ファイルの内容は内部データ構造にロードされます。通常、各入力ファイルは複数のコードまたはデータ セクションを含みます。各セクションは、セクションの生データ内のアドレスを再配置可能シンボルに関連付ける再配置エントリのリストを格納します。

9.2.2 メモリ割り当て

全ての入力ファイルをロードした後に、リンカは各入力セクションを出力セクションに割り当てる事によりメモリを割り当てます。入力セクションと出力セクションの関係は、リンカスクリプト内のセクションマップにより定義されます。出力セクションの名前は入力セクションの名前と同じであっても異なっていても構いません。各出力セクションは、ターゲット デバイス内のメモリ領域に割り当てられます。

> **Note:** 入力セクションは、コンパイラまたはアセンブラによってソースコードから生成されます。出力セクションはリンカによって生成されます。

入力セクションが明示的に出力セクションに割り当てられていない場合、リンカはセクション属性に従って未割り当てセクションを割り当てます。リンカによる割り当ての詳細は **9.3「リンカ割り当て」**を参照してください。

9.2.3 シンボルの解決

メモリの割り当てが済むと、リンカはシンボルを解決するための処理を開始します。各入力セクション内で定義されたシンボルは、そのセクションの先頭位置に対して相対的なオフセットを持ちます。リンカは、これらの値を出力セクションのオフセットに変換します。

次にリンカは、全ての外部シンボル参照をシンボル定義と照合します。同じ外部シンボルに複数の定義が存在する場合、エラーが発生します。外部シンボルの定義が見つからない場合、リンカはアーカイブ ファイル内でシンボル定義を見つけようと試みます。アーカイブ ファイル内にシンボル定義が見つかった場合、対応するアーカイブモジュールがロードされます。

アーカイブからロードされたモジュールは追加のシンボル参照を含んでいる場合があります。その場合、全ての外部シンボル参照に対応する定義が見つかるまで処理が続きます。「weak」として定義された外部シンボルは特別な方法で処理されます。**9.4「グローバル シンボルと weak シンボル」**を参照してください。外部シンボル参照が 1 つでも未定義のまま残ると、エラーが生成されます。

9.2.4 特殊セクションの生成

シンボルが解決すると、リンカは特殊な入力または出力セクションを生成します。例えば、初期化データをサポートするため、リンカは .dinit という名前の特別な入力セクションを生成します。セクション .dinit は、C ランタイム ライブラリによって解釈される初期化テンプレートです。初期化データの詳細は **9.5「初期化データ」**を参照してください。

9.2.5 絶対アドレスの計算

特殊セクションの生成後に、全ての出力セクションの最終的なサイズが明らかになります。この時点で、リンカは全ての出力セクションと外部シンボルの絶対アドレスを計算します。リンカは、割り当てたメモリ領域内に各出力セクションが収まるかどうか確認します。いずれかのセクションがメモリ領域からはみ出す場合、エラーが生成されます。リンカスクリプト内で定義された全てのシンボルも計算されます。

9.2.6 出力ファイルのビルド

最後にリンカは出力ファイルをビルドします。各セクション内の再配置エントリは絶対アドレスを使ってパッチされます。あるシンボル向けに計算されたアドレスが再配置エントリ内に収まらない場合、リンクエラーが発生します。これは、あるモジュールがある変数をそれが「small data」セクション内にあると想定して参照しているのに、他のモジュールがその変数を非 small セクション内で定義している場合等に発生します。

オプションが指定された場合、リンクマップも生成されます。リンクマップはメモリ使用のレポートを含みます。これは、データメモリとプログラムメモリ内の全てのセクションの開始アドレスと長さを示します。リンクマップの詳細は **6.4.5「マップファイル」**を参照してください。

9.3 リンカ割り当て

リンカ割り当てはリンカスクリプトによって制御され、以下の 3 ステップで処理されます。

1. 入力セクションの出力セクションへのマッピング
2. 出力セクションのメモリ領域への割り当て
3. 未マッピング セクションの割り当て

ステップ 1 と 2 はシーケンシャル メモリアロケータが実行します。リンカスクリプト内に現れる入力セクションは、ターゲット デバイス内の特定のメモリ領域に割り当てられます。メモリ領域内のアドレスは、最低アドレスから高位のアドレスに向かって連続的に割り当てられます。

ステップ 3 はベストフィット メモリアロケータが実行します。リンカスクリプト内に現れない入力セクションは、それらの属性に従ってメモリ領域に割り当てられます。ベストフィット アロケータにより、残されたメモリ領域 (シーケンシャル アロケータによって残された出力セクション間のギャップ等) の全てが効率的に使われます。

9.3.1 入力セクションの出力セクションへのマッピング

セクションマップに従い、入力セクションはグループ化されて出力セクションにマッピングされます。1 つの出力セクションに複数の異なる入力セクションを格納する場合、入力セクションの順序付けが重要です。例えば、以下の出力セクション定義について考えます。

```
/* Code Sections */
.text ORIGIN(kseg0_program_mem)  :
{
  *(.text .stub .text.*.gnu.linkonce.t.*)
  *(.mips16.fn.*)
  *(.mips16.call.*)
} >kseg0_program_mem =0
```

ここでは .text という名前の出力セクションを定義しています。このセクションの内容は波カッコ ({}) の中で指定されているという事に注意が必要です。波カッコを閉じた後の >kseg0_program_memは、この出力セクションをメモリ領域kseg0_program_memに割り当てるよう指定します。

出力セクション .text の内容は以下のように解釈されます。

- .text および .stub という名前の入力セクションと、ワイルドカード パターン .text.* および .gnu.linkonce.t.* に一致する入力セクションは、1 つのグループとしてこの出力セクションにマッピングされます。これらのセクションをグループ化する事で参照のローカル性が確保されます。
- ワイルドカード パターン .mips16.fn.* に一致する入力セクションは、2 つ目のグループとしてこの出力セクションにマッピングされます。
- ワイルドカード パターン .mips16.call.* に一致する入力セクションは、3 つ目のグループとしてこの出力セクションにマッピングされます。

9.3.2 出力セクションのメモリ領域への割り当て

全ての出力セクションのサイズが明らかになると、それらはメモリ領域に割り当てられます。通常、領域は出力セクション定義内で指定されます。領域が指定されなかった場合、最初に定義されたメモリ領域が使われます。

メモリ領域は、低位のアドレスから高位のアドレスに向かって、セクションマップ内でセクションが現れる順番通りに、連続的に書き込まれます。領域ごとに独立したロケーション カウンタは、その領域で次に利用可能なメモリ位置を指します。以下の 2 つの条件のいずれかにより、領域内のメモリ割り当てにギャップが生じます。

1. セクションマップが出力セクションに対して絶対アドレスを指定する
   または
2. 出力セクションが特定のアラインメントを要求する

どちらの場合も、その時点のロケーション カウンタと絶対アドレス (またはアラインメントされたアドレス) の間のメモリ位置はスキップされます。メモリに割り当てられた全てのアイテムの正確なアドレスは、リンクマップ ファイルから特定できます。アラインメントされた (C の `aligned` 属性またはアセンブリの `.align` ディレクティブを持つ) メモリブロックを含むセクションは、それらと同じ (またはそれらより大きな) アラインメント値に対してアラインメントする必要があります。複数の入力セクションが異なるアラインメント要件を持つ場合、最大のアラインメント要件が出力ファイルに適用されます。

9.3.3 未マッピング セクションの割り当て

セクションマップ内の全てのセクションの割り当てが済んだ後に残ったセクションは、未マッピングであると見なされます。未マッピング セクションは、セクション属性に従って割り当てられます。本リンカはベストフィット メモリアロケータを使って、最も効率的なメモリの使い方を決定します。ベストフィット アロケータの主目的は、アドレス アラインメントの制約によって生じたメモリギャップを削減または排除する事です。慣例により、ほとんどの標準的セクション (`.text`、`.data`、`.bss`、`.ramfunc` セクション等) はリンカスクリプト内で明示的にマッピングされません。このため、ベストフィット メモリ アロケータに最大限の柔軟性が与えられます。例外は、GP 相対アドレス指定に使われる「small」データセクションです。これらのセクションは互いにグループ化する必要があるため、リンカスクリプト内でマッピングされます。ツールチェーンの将来のリリースでは、「small」データセクションもベストフィット アロケータによって割り当て可能になるかもしれません。

セクション属性は以下のようにメモリ割り当てに影響します。セクション属性の一般的な説明は **A.2「セクションを定義するためのアセンブラ ディレクティブ」**を参照してください。

code

code 属性は、リンカスクリプト内の領域 kseg0_program_mem による定義に従って、セクションをプログラムメモリに割り当てるよう指定します。以下の属性を code と一緒に使う事で、さらに詳細な割り当てが指定できます。

- address(): 絶対アドレスを指定します。
- align(): セクション開始アドレスのアラインメントを指定します。

data

data 属性は、リンカスクリプト内の領域 kseg0_data_mem および kseg1_data_mem による定義に従って、セクションを初期化記憶域としてデータメモリに割り当てるよう指定します。以下の属性を data と一緒に使う事で、さらに詳細な割り当てが指定できます。

- address(): 絶対アドレスを指定します。
- near: データメモリの先頭の 64K 領域を指定します。
- align(): セクション開始アドレスのアラインメントを指定します。
- reverse(): [セクション終了アドレス + 1] のアラインメントを指定します。

bss

bss 属性は、リンカスクリプト内の領域 kseg0_data_mem および kseg1_data_mem による定義に従って、セクションを非初期化記憶域としてデータメモリに割り当てるよう指定します。以下の属性を bss と一緒に使う事で、さらに詳細な割り当てが指定できます。

- address(): 絶対アドレスを指定します。
- near: データメモリの先頭の 64K 領域を指定します。
- align(): セクション開始アドレスのアラインメントを指定します。
- reverse(): [セクション終了アドレス + 1] のアラインメントを指定します。

persist

persist 属性は、リンカスクリプト内の領域 kseg0_data_mem および kseg1_data_mem による定義に従って、セクションを永続的記憶域としてデータメモリに割り当てるよう指定します。persistent 記憶域は C ランタイム ライブラリによってクリアも初期化もされません。以下の属性を persist と一緒に使う事で、さらに詳細な割り当てが指定できます。

- address(): 絶対アドレスを指定します。
- near: データメモリの最初の 64K 領域を指定します。
- align(): セクション開始アドレスのアラインメントを指定します。
- reverse(): [セクション終了アドレス + 1] のアラインメントを指定します。

9.4 グローバル シンボルと weak シンボル

定義が存在しないシンボル参照が出力ファイル内に現れると、そのシンボルは外部シンボルとして宣言されます。既定値により、外部シンボルはグローバル結合を有し、グローバル シンボルとして参照されます。外部シンボルは weak 結合を使って明示的に宣言できます。これには C 言語の __weak__ 属性またはアセンブリ言語の .weak ディレクティブを使います。

グローバル シンボルは、その名が示す通り、リンクに含まれる全ての入力ファイルから可視です。参照される各グローバル シンボルには定義が 1 つだけ存在している事が必要です。グローバル定義がどの入力ファイルの中にも見つからない場合、アーカイブ ファイルが検索されます。そして、最初に定義が見つかったアーカイブ モジュールがロードされます。グローバル シンボルが最終的に見つからなかった場合、リンクエラーが報告されます。

weak シンボルはグローバル シンボルと同名の空間を共有しますが、扱いが異なります。weak シンボルの定義は複数存在しても構いません。weak シンボルの定義がどの入力ファイルの中にも見つからない場合、アーカイブは検索されず、その weak シンボルに対する全ての参照の値は 0 であると見なされます。同名のグローバル シンボル定義は weak シンボル定義よりも優先されます (または weak シンボルは無効にされる)。基本的に weak シンボルはオプションであると見なされ、グローバル シンボルによって置き換えられるか、完全に無視されます。

9.5 初期化データ

本リンカは、データメモリ内の初期化変数に対する自動的なサポートを提供します。変数は各種セクションに割り当てられます。各データセクションは、初期化か非初期化かを示すフラグを使って宣言されます。

各種データセクションの初期化を制御するため、本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プログラムメモリ内に割り当てられ、起動時にランタイム ライブラリによって処理されます。アプリケーションの main 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は初期化済みです。

- 標準データセクション名
- データ初期化テンプレート
- ラインタイム ライブラリのサポート

9.5.1 標準データセクション名

伝統的に、GNU テクノロジに基づくリンカは、リンクされるバイナリファイル内で以下の 3 種類のセクションをサポートします。

表 9-1: 伝統的セクション名

セクション名	概要	属性
.text	実行可能コード	code
.data	初期値を受け取るデータメモリ	data
.bss	初期化されないデータメモリ	bss

「bss」という名前の由来は数 10 年遡り、「Block Started by Symbol」を意味します。慣例により、プログラム起動時に bss メモリは値 0 で埋められます。伝統的セクション名は、表 9-1 に示す暗黙的な属性を持つと見なされます。code 属性は、そのセクションが実行可能コードを格納し、プログラムメモリにロードされるという事を示します。bss 属性は、そのセクションが初期化されない (しかしプログラム起動時に値 0 で埋められる) データ記憶域を格納するという事を示します。data 属性は、そのセクションがプログラム起動時に初期値を受け取るデータ記憶域を格納するという事を示します。

アセンブリ アプリケーションは、「A.2 セクションを定義するためのディレクティブ」に記載したセクション ディレクティブを使う事で、明示的な属性を持つ追加のセクションを定義できます。C アプリケーションの場合、32 ビットコンパイラは変数と関数を格納するためのセクションを自動的に定義します。自動的なセクション定義を生じさせる変数と関数の属性については『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

> **Note:** セクション ディレクティブが使われた場合、後続の全ての宣言は指定されたセクション内へアセンブルされます。これは、次のセクション ディレクティブが現れるか、ファイルの終端に達するまで続きます。セクションの定義とセクション属性の詳細は**「A.2 セクションを定義するためのディレクティブ」**を参照してください。

9.5.2 データ初期化テンプレート

9.5.1「標準データセクション名」で説明したように、本 32 ビット言語ツールは bss 型のセクション (初期化されないメモリ) と data 型のセクション (初期値を受け取るメモリ) をサポートします。スタートアップ時に、data 型セクションは初期値を受け取り、bss 型セクションは値 0 で埋められます。任意の bss 型セクションまたは data 型セクションを数に制限なくサポートする汎用的なデータ初期化テンプレートが使われます。データ初期化テンプレートはリンカによって生成され、プログラムメモリ内の .dinit という名前の出力セクション内にロードされます。ラインタイム ライブラリ内のスタートアップ コードは、このテンプレートに従ってデータメモリを初期化します。

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。このテンプレートは 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:7; /* 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。

フォーマット コードの値は 0 または 1 です。

表 9-2: フォーマット コードの値

フォーマット コード	概要
0	出力セクションを 0 で埋める
1	データ配列内の各命令ワードから 4 バイトのデータをコピーする

9.5.3 ランタイム ライブラリのサポート

データメモリ内の変数を初期化するため、起動時に (アプリケーションの main 関数が制御を引き継ぐ前に)、データ初期化テンプレートを処理する必要があります。C プログラムの場合、これはランタイム ライブラリ内の C スタートアップ モジュールによって実行されます。アセンブリ言語プログラムも、libpic32.a とリンクする事により、C スタットアップ モジュールを使う事ができます。

スタートアップ モジュールを使うには、デバイスリセット時にランタイム ライブラリが制御を行う事をアプリケーションが許容する必要があります。C プログラムの場合、自動的にそうなります。アプリケーションの main() 関数は、スタートアップ モジュールの実行が完了した後に呼び出されます。アセンブリ言語プログラムの場合、以下の命名規則に従って、デバイスリセット時に制御を行うルーチンを指定する必要があります。

表 9-3: main エントリポイント

main エントリポイント	概要
_reset	デバイスリセット後直ちに制御を引き継ぐ
main	スタートアップ モジュールの実行完了後に制御を引き継ぐ

エントリ名 _reset の頭文字はアンダースコアである事に注意が必要です。エントリ名 main にはアンダースコアを付けません。デバイスリセット時に、スタートアップ モジュールが呼び出されて以下を実行します。

1. スタックポインタを初期化します。
2. .dinit セクション内のデータ初期化テンプレートを読み出します。このテンプレートに基づいて、全ての非初期化セクションをクリアすると共に、プログラム メモリから読み出した値を使って全ての初期化セクションを初期化します。
3. プログラム フラッシュからデータメモリ (バスマトリクス初期化レジスタ) に RAM 関数をコピーします。
4. 関数 main を引数なしで呼び出します。
5. main がリターンすると、プロセッサはリセットします。

--no-data-init オプションが指定された場合、代替スタートアップ モジュールがリンクされます。

代替スタートアップ モジュールの動作は、上記のステップ (2) が省略されるという点を除けば、主スタートアップ モジュールと同じです。代替スタートアップ モジュールは主スタートアップ モジュールよりも小さく、データの初期化が不要な場合にプログラムメモリを節約するために使えます。

どちらのモジュールも、ソースコードは MPLAB XC32 C コンパイラ インストール ディレクトリ内の src サブディレクトリ内にあります。これらのスタートアップ モジュールは必要に応じて変更できます。例えば、アプリケーションが引数付きで main 関数を呼び出す必要がある場合、条件付きアセンブリ ディレクティブを切り換える事で対応できます。

9.6 スタックの割り当て

PIC32 MCU 向け MPLAB C 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。

既定値により、32 ビットリンカは未使用データメモリから可能な限り大きなスタックを動的に割り当てます。以前のリリースでは、リンカスクリプト内で指定された出力セクションを使ってスタックを割り当てました。

スタックの位置とサイズはリンクマップ出力ファイルとメモリ使用レポート (表題「Dyanmic Memory Usage」の下) で示されます。リンカ コマンドライン上で `--defsym=_min_stack_size=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する事により、アプリケーションは最小サイズ以上のスタックを確保する事ができます。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.
The linker script a default _min_stack_size of 1024.
```

> **Note:** コンパイラのスタックの使用に関する詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

リンカが報告する `.stack` セクションのサイズは、リンクエラーを防ぐために必要な最小サイズです。実効スタックサイズは、通常、報告された `.stack` セクションサイズよりも大きくなります。

9.7 ヒープの割り当て

C ランタイムヒープはデータメモリの非初期化領域であり、標準 C ライブラリの動的メモリ割り当て関数 (`calloc`、`malloc`、`realloc`) を使った動的メモリ割り当てのために使います。これらの関数を (直接 / 間接を問わず) 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。**既定値のヒープサイズは 0 です。**

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。例として、512 バイトのヒープを割り当てる場合のコマンドラインを以下に示します。

```
xc32-gcc foo.c -Wl,--defsym=_min_heap_size=512
```

リンカは、スタックの直前にヒープを割り当てます。ヒープの位置とサイズはリンクマップ出力ファイルとメモリ使用レポート (表題「Dyanmic Memory Usage」の下) で示されます。要求されたサイズが確保できない場合、リンカはエラーを報告します。

現在のリリースでは、ヒープはリンカによって動的に割り当てられます。以前のリリースでは、リンカスクリプト内で指定された出力セクションを使ってヒープを割り当てました。

9.8 PIC32MXの 割り込みベクタテーブル

各割り込みのベクタアドレスは、例外ベースレジスタ (EBASE<31:12>) を使って計算します。このレジスタは、カーネル セグメント (kseg) アドレス空間内で 4 KB ページ境界に配置されたベースアドレス値を提供します (EBASE は CPU レジスタです)。アドレスは、EBASE と VS ビット (INTCTL<9:5>) の値を使って計算されます。VS ビットは、隣り合うベクタアドレス間のベクタ間隔を提供します。

リンカスクリプトは、対応する割り込みベクタテーブルを以下のように生成します。

```
PROVIDE(_vector_spacing = 0x00000001);
_ebase_address  = 0x9FC01000;

SECTIONS
{
  .app_excpt _GEN_EXCPT_ADDR :
  {
    KEEP(*(.gen_handler))
  } > exception_mem
  .vector_0 _ebase_address + 0x200 :
  {
    KEEP(*(.vector_0))
  } > exception_mem
  ASSERT (_vector_spacing == 0 || SIZEOF(.vector_0) <=
     (_vector_spacing << 5),
     "function at exception vector 0 too large")
  .vector_1 _ebase_address + 0x200 +
     (_vector_spacing << 5) * 1 :
  {
    KEEP(*(.vector_1))
  } > exception_mem
  ASSERT (_vector_spacing == 0 || SIZEOF(.vector_1) <=
     (_vector_spacing << 5),
     "function at exception vector 1 too large")
  .vector_2 _ebase_address + 0x200 +
     (_vector_spacing << 5) * 2 :
  {
    KEEP(*(.vector_2))
  } > exception_mem
/* … */
  .vector_63 _ebase_address + 0x200 +
     (_vector_spacing << 5) * 63 :
  {
    KEEP(*(.vector_63))
  } > exception_mem
  ASSERT (_vector_spacing == 0 || SIZEOF(.vector_63) <=
     (_vector_spacing << 5),
     "function at exception vector 63 too large")
}
```

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 つの出力セクションが存在します。

9.9 専用プログラマブル変数オフセットを備えた PIC32 MCU向 けの割り込みベクタテーブル

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。

XC32 ツールチェーンは、デバイスに固有の既定値リンカスクリプトと、対応するオブジェクト ファイルを提供します。これらは既定値ラインタイム スタートアップ コードを使って機能します。
以下のファイルは、ベクタテーブル オフセットレジスタを初期化するために使います。これらのファイルは /pic32mx/lib/proc/<devicename> にあります。

デバイス リンカスクリプト　　*<devicename>*.ld

ベクタオフセット初期化　　vector_offset_init.o

デバイス ランタイム スタートアップ コード　crt0_*<boot_isa>*.o

9.9.1 デバイス固有のリンカスクリプト

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。アセンブリコードの場合、.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))
 }
}
```

9.9.2 ベクタオフセット初期化モジュール

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ベクタのアドレスまでのオフセットです。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 (対応する OFFxxx 特殊機能レジスタのアドレスを使用) を生成します。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。

```
  .section
.data.__vector_offset_BF810540,data,keep,address(0xBF810540)
  .word __vector_offset_0
  .word __vector_offset_1
  .word __vector_offset_2
  .word __vector_offset_3
  .word __vector_offset_4
  .word __vector_offset_5
  .word __vector_offset_6
```

9.9.3 ランタイム スタートアップ コードによるデータの初期化

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が生成したデータ初期化テンプレートにより、標準ランタイム スタートアップ コードは OFFxxx 特殊機能レジスタを既定値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。

9.9.4 ベクタテーブルのアセンブリ ソースコードの例

以下のサンプルコードに、割り込みベクタ 0 に対してベクタ ディスパッチを生成する方法を示します。ベクタ ディスパッチは、ベクタテーブルから実際の割り込みサービスルーチン (ISR) へのジャンプです。

```
/*  Input section .vector_0 is mapped to the .vectors output
       *  section in the linker script.
       */
   .globl _vector_dispatch_0
   .section .vector_0,code
   .align 2
   .set nomips16
   .ent __vector_dispatch_0
__vector_dispatch_0:
      /* Jump to the actual ISR code */
   j  isrvector0
   nop
   .end __vector_dispatch_0
   .size __vector_dispatch_0, .-__vector_dispatch_0
```

以下のサンプルコードに、割り込みサービスルーチンを直接ベクタテーブル内に配置する方法を示します。リンカスクリプト内のマッピングは、関数のコードを格納するためにベクタ間隔を自動的に調整します。

```
/*  Input section .vector_0 is mapped to the .vectors output
       *  section in the linker script.
       */
      .section .vector_0,code
   .align 2
   .globl isrvector0
   .set nomips16
   .set nomicromips
   .ent isrvector0
isrvector0:
   .set noat

     /*  Interrupt Service Routine code directly in the vector table.
      *  Be sure to preserve registers as appropriate for an ISR.
      */

   eret
   .set at
   .end isrvector0
   .size isrvector0, .-isrvector0
```

XC32 C コードから、標準の `vector(n)` および `at_vector(n)` 関数属性をユーザの ISR 関数に対して使います。これらの関数属性の詳細は『MPLAB XC32 C/C++ コンパイラ ユーザガイド』(DS51686) を参照してください。

NOTE:

第 10 章　リンカの例

10.1　はじめに

32 ビット コンパイラおよびアセンブラは、アプリケーションの特定エレメントを特殊な処理向けに指定するための構文をそれぞれ提供します。C 言語には、変数および関数定義を変更するための属性が豊富に用意されています (『MPLAB/XC32 C/C++ コンパイラ ユーザガイド』(DS51686) 参照)。アセンブリ言語の場合、変数と関数はメモリセクションへと抽象化され、それらがリンカへの入力となります。アセンブラは、セクション定義を変更するための一連の属性をリンカとは別に提供します (**4.6「セクション アラインメントを変更するディレクティブ」**参照)。

第 10 章では、32 ビットの各種リンカ例と、C 言語とアセンブリ言語の等価な構文を示します。

10.2　ハイライト

第 10 章の主な内容は以下の通りです。

- メモリアドレスと再配置可能コード
- 指定アドレスへの変数の配置
- 指定アドレスへの関数の配置
- プログラムメモリのアドレス指定と予約

10.3 メモリアドレスと再配置可能コード

大部分のアプリケーションでは、完全に再配置が可能なソースコードを書く事が望まれます。そうする事で、リンカは、関数と変数が配置されるメモリ内の厳密なアドレスを決定する事ができます。外部シンボルのメモリ内の最終的アドレスは、リンクマップ出力から特定できます。以下にその抜粋を示します。

```
...
.text0x9d0000f00x64
 .text0x9d0000f00x64 test.o
   0x9d0000f0myfunc
   0x9d000110main

.text._DefaultInterrupt0x9d0001540x48
 .text._DefaultInterrupt0x9d0001540x48 c:/program files/
       microchip/xc32/v1.20/bin/
       ../lib/gcc/pic32mx/4.5.2/
       ../../../../pic32mx/
       lib\libpic32.a
       defaultinterrupt.o)
   0x9d000154_DefaultInterrupt...
```

場合によっては、特定の変数または関数を配置するアドレスをユーザが指定する必要があります。伝統的に、これはユーザ定義セクションを生成する事と、カスタム リンカスクリプトを書く事により行います。32 ビット アセンブラおよびコンパイラは、絶対アドレスとメモリ空間をソースコード内で直接指定するための一連の属性を提供します。これらの属性使う場合、カスタム リンカスクリプトは不要です。

Note: 絶対アドレスを指定する場合、そのアドレスが妥当かつ利用可能かどうかはユーザの責任となります。指定したアドレスがレンジ外であるか静的に割り当てられたリソースと競合する場合、リンカエラーが発生します。

10.4 指定アドレスへの変数の配置

以下の例では、配列 buf1 をデータメモリ内の指定アドレスに配置します。buf1 のアドレスは、シミュレータ内のプログラムを実行するか、リンクマップを調べる事で確認できます。

```
#include "stdio.h"
int __attribute__((address(0xa0000200))) buf1[128];
```

アセンブリ言語による等価な配列定義を以下に示します。.align ディレクティブはオプションであり、データメモリ内の既定値アラインメントを表します。セクション名として「*」を使うと、アセンブラはソースファイル名に基づいて一意の名前を生成します。

```
.globlbuf1
    .section*,address(0xa0000200),bss
    .align2
    .typebuf1, @object
    .sizebuf1, 512
buf1:
    .space512
```

10.5 指定アドレスへの関数の配置

以下の例では、関数 func を指定アドレスに配置します。

```
#include "stdio.h"
void __attribute__((address(0x9d002000))) func()
{}
```

アセンブリ言語による等価な関数定義を以下に示します。.align ディレクティブはオプションであり、プログラムメモリ内の既定値アラインメントを表します。セクション名として「*」を使うと、アセンブラはソースファイル名に基づいて一意の名前を生成します。

```
.section*,address(0x9d002000),code
    .align2
    .globlfunc
func:
    ......
```

10.6 プログラムメモリのアドレス指定と予約

以下の例では、プログラムメモリの 1 ブロックをブートローダ等の特別な用途向けに予約します。このブロックには任意サイズの関数を割り当てる事ができます。残った空間は、拡張または他の目的のために予約されます。

以下の出力セクション定義はカスタム リンカスクリプトに追加されます。

```
BOOT_START = 0x9d00A200;
BOOT_LEN = 0x400;
my_boot BOOT_START :
{
*(my_boot);
.= BOOT_LEN; /* advance dot to the maximum length */
} > kseg0_program_mem
```

「ドットの代入」(.=) は入力セクションの後のセクション定義の中に現れるという事に注意が必要です。ドットは、現在作業中のセクション内のロケーション カウンタ (または次の書き込み位置) を表す特殊な変数です。これは、そのセクションの開始位置に対する相対的なオフセットです。この命令文は実質的に、「入力セクションの大きさに関係なく指定されたサイズの出力セクションを確保する」よう指示します。

以下の C 関数は予約されたブロック内に配置されます。

```
void __attribute__((section("my_boot"))) func1()
{
  /* etc. */
}
```

等価なアセンブリ言語は以下の通りです。

```
 .section        my_boot,code
        .align  2
        .globl  func1
func1:
    ......
```

第 11 章　リンカのエラーと警告

11.1　はじめに

MPLAB XC32 オブジェクト リンカ (xc32-ld) はエラーと警告を生成します。これらの出力について説明します。以下には、リンカが生成する最も一般的な診断メッセージだけを記載しています。

第 11 章の主な内容は以下の通りです。

- 致命的エラー
- エラー
- 警告

11.2 致命的エラー

以下のエラーは、リンカで内部エラーが発生した事を示します。カスタム リンカスクリプトを使っている時に、リンカが以下の致命的エラーのいずれかを生成した場合、スクリプトが OUTPUT_FORMAT (elf32-tradlittlemips) と OUTPUT_ARCH (pic32mx) を正しく指定している事を確認してください。他の値を指定した場合、リンカは未サポートのモードで動作します。また、サポートされているオプションだけをリンカのコマンドラインで渡している事を確認します。最後に、他のアプリケーションがリンカの入力または出力ファイルをロックしていない事を確認します。

OUTPUT_FORMATとOUTPUT_ARCHおよびコマンドライン オプションを正しく指定しても致命的エラーが発生する場合、Microchip 社のエンジニアリング サポート (http://support.microchip.com) にお問い合わせください。その際、エラーが発生したソースコードとコマンドライン オプションの詳細をお知らせください。

- Bfd backend error: bfd_reloc_ctor unsupported
- Bfd_hash_allocate failed creating symbol %s
- Bfd_hash_lookup failed:%e
- Bfd_hash_lookup for insertion failed:%e
- Bfd_hash_table_init failed:%e
- Bfd_hash_table_init of cref table failed:%e
- Bfd_link_hash_lookup failed:%e
- Bfd_new_link_order failed
- Bfd_record_phdr failed:%e
- Can't set bfd default target to `%s':%e
- Can not create link hash table:%e
- Can not make object file:%e
- Cannot represent machine `%s'
- Could not read relocs:%e
- Could not read symbols
- Cref_hash_lookup failed:%e
- Error closing file `%s'
- Error writing file `%s'
- Failed to create hash table
- Failed to merge target specific data
- File not recognized:%e
- Final close failed:%e
- Final link failed:%e
- Hash creation failed
- Out of memory during initialization
- Symbol `%t' missing from main hash table
- Target %s not found
- Target architecture respecified
- Unknown architecture:%s
- Unknown demangling style `%s'
- Unknown language `%s' in version information

11.3 エラー

以下に示すリンカエラーは、通常リンカスクリプト内のエラーまたはリンカに渡されたコマンドライン オプション内のエラーを示します。エラーは、1 つまたは複数の入力オブジェクト ファイルまたはアーカイブの問題を示す場合もあります。

記号

--gc-sections and -r may not be used together

ガベージ コレクション セクションオプションと再配置可能出力オプションが不整合です。`--gc-sections` オプションまたは `--relocatable` オプションのどちらかを削除する必要があります。

--relax and -r may not be used together

緩和オプションと再配置可能出力オプションが不整合です。どちらかのオプションを削除する必要があります。

A

A heap is required, but has not been specified.

標準 C 入出力関数を使う場合、ヒープを指定する必要があります。

Assignment to location counter invalid outside of SECTION

特殊なドットシンボルへの代入は、`SECTION` 内の割り当て中にのみ可能です。リンカスクリプト内の代入命令文の位置を確認してください。

B

Bad --unresolved-symbols option:*option*

`--unresolved-symbols` メソッド オプションは無効です。このオプションは非サポートです。代わりに既定値の `--unresolved-symbols=report-all` を試してください。

C

Can not PROVIDE assignment to location counter

特殊なドットシンボルへの代入は割り当て中にのみ可能です。*`PROVIDE`* コマンドは、ロケーション カウンタに対する代入を使えません。リンカスクリプトから誤りのある命令文を削除する必要があります。

Can not set architecture:*arch_name*

カスタム リンカスクリプトを使っている場合、リンカスクリプト内に `OUTPUT_ARCH (pic32mx)` コマンドが書かれている事を確認します。PIC32 MCU リンカは、現在「pic32mx」アーキテクチャだけをサポートします。

Cannot move location counter backwards (from *addr1* to *addr2*)

次のドットシンボル値は、現在のドットシンボル値より大きい事が必要です。

Could not allocate data memory.

リンカは、領域「kseg0_data_memory/kseg1_data_memory」に割り当てられた全てのセクションを配置する方法を見つける事ができませんでした。

Could not allocate program memory.

リンカは、領域「kseg0_program_memory」に割り当てられた全てのセクションを配置する方法を見つける事ができませんでした。

D

Dangerous relocation: *relocation_type*

シンボルは解決しましたが、使うのは危険です。このエラーは、例えばコードが GP 相対アドレス指定を使っているのに_gp初期化シンボルが定義されなかった場合等に発生します。_gp シンボルは、通常リンカスクリプト内で定義されます。

--data-init and --no-data-init options can not be used together.

--data-initは､データのランタイム初期化用テンプレートとして.dinitという名前の特殊な出力セクションを生成します。--no-data-init は、そのような出力セクションを生成しません。どちらか 1 つのオプションだけ指定する必要があります。

F

File format not recognized; treating as linker script

入力ファイルの 1 つが ELF オブジェクトとしてもアーカイブとしても認識されませんでした。リンカは、そのファイルをリンカスクリプトであると見なします。

G

Group ended before it began (--help for usage)

コマンドライン上で -) オプションが -(オプションより前に現れました。リンカ コマンドライン上でグループが正しく指定されている事を確認してください。

I

Illegal use of *name* section

そのセクション名は予約済みです。例えば、特別な出力セクション名 /DISCARD/ を使うと、入力セクションを破棄できます。/DISCARD/ という名前の出力セクションに割り当てられた全ての入力セクションは、出力ファイルにインクルードされません｡ユーザ独自の出力セクションに /DISCARD/ という名前を付けない事が必要です。

Includes nested too deeply

インクルードの最大ネスト深さは 10 です。

Invalid argument to option --section-start

--section-startに対する引数はsectionname=*org*である事が必要です｡*org*は 1 つの 16 進整数である事が必要です。sectionname、等号 (=)、*org* の間に空白類を挿入しない事が必要です。

Invalid assignment to location counter

特殊なドットシンボルに対する代入は無効です。

Invalid syntax in flags

セクションフラグが無効です。以下のフラグだけが使えます。

```
a r w x l
```

M

Macros nested too deeply

マクロの最大ネスト深さは 10 です。

May not nest groups (--help for usage)

アーカイブ グループは既に開始しています。-) オプションを使って現在のグループを閉じてから -(オプションを使って次のグループを開始する必要があります。

Member %b in archive is not an object

アーカイブ メンバーは有効なオブジェクトではありません。ライブラリ アーカイブが Microchip MPLAB XC32 C/C++ コンパイラに対して適正である事を確認してください。

Missing argument(s) to option --section-start

`--section-start` に対する引数は `sectionname=`*`org`* である事が必要です。

Multiple definition of *name*

リンカが、複数回定義されているシンボルを見つけました。余分な定義を削除する必要があります。

Multiple startup files

リンカスクリプトはスタートアップ ファイルの設定を試みましたが､スタートアップ ファイルは既に設定済みです。スタートアップ ファイルは、リンカスクリプト内で 1 つだけ指定されている事が必要です。

N

No input files

リンカはコマンドラインで指定された入力ファイルを見つける事ができませんでした。リンカは何もできませんでした。リンカに正しいオブジェクト ファイル名が渡される事を確認してください。

Nonconstant expression for *name*

name は非定数式である事が必要です。

Not enough memory for stack (num bytes available).

最小サイズのスタックを配置するための空きメモリ容量が不足しています。

R

region *region* is full (*filename* section *secname*).

メモリ領域 *region* がフルであるにも関わらずセクション *secname* がそこに割り当てられています。

Reloc refers to symbol *name* which is not being output

出力されていないシンボルを命令が参照しています。

Relocation truncated to fit *relocation_type name*.

このエラーは、*name* の再配置された値が用途に対して大きすぎるという事を示します。これは、問題の命令に対してアドレスがレンジ外である場合に発生します。意図するセクション内でそのシンボルが宣言かつ定義されている事を確認してください。例えば、変数の宣言と定義の両方が const または非 const のどちらかである事が必要です。

Relocation truncated to fit:*relocation_type name* against undefined symbol *name*

このエラーは、シンボルが存在しない場合に発生します。例えば、未定義の関数をコードが呼び出した場合に発生します。

U

Undefined MEMORY region *region* referenced in expression

リンカスクリプト内に存在しない MEMORY 領域を式が参照しました。

11.4 警告

本リンカは、プログラムに欠陥はあるもののリンクは続行できると判断した場合に警告を生成します。警告は無視すべきではありません。各警告を注意深く調べ、リンカがプログラムの意図を確実に理解できるよう修正する必要があります。警告メッセージによってユーザプログラム内のバグが見つかる場合もあります。

C

Cannot find entry symbol *name*

If the linker cannot find the specified entry symbol and it is not a number. テキスト セクション内の最初のアドレスを使う必要があります。

Changing start of section *name* by *num* bytes

アラインメントのために、リンカはセクション (*name*) の開始位置を変更します。

D

data initialization has been turned off, therefore section secname will not be initialized.

指定されたセクションは初期化を要求していますが、データの初期化は無効にされています。従って初期データ値は破棄されます。データセクションの記憶域は通常通りに配置されます。

I

initial values were specified for a non-loadable data section (name).These values will be ignored.

既定値により、永続的データセクションはデータが初期化されないという事を意味します。従って値は破棄されます。そのセクションの記憶域は通常通りに配置されます。

R

Redeclaration of memory region *name*

リンカスクリプト内で MEMORY 領域が複数回宣言されています。

U

Undefined reference to *name*

そのシンボルは未定義です。

パート 3 - 32 ビット ユーティリティ (アーカイバ / ライブラリアン等)

NOTE:

第 12 章　MPLAB XC32 オブジェクト アーカイバ / ライブラリアン

12.1　概要

MPLAB XC32 オブジェクト アーカイバ / ライブラリアン (xc32-ar) は、アーカイブからファイルを生成、変更、抜き出します。このツールはユーティリティの中の 1 つです。「アーカイブ」とは複数のファイルを 1 つにまとめたファイルの事です。アーカイブから個々のファイル (アーカイブのメンバーと呼ぶ) を元通りに取り出す事ができます。

32 ビット アーカイバ / ライブラリアンは、メンバーの名前の長さを制限しません。しかし、f 修飾子を使うと、ファイル名は 15 文字に切り詰められます。

ほとんどの場合、この種のアーカイブは、よく使われるサブルーチンを収めた「ライブラリ」として使われます。このため、アーカイバはバイナリ ユーティリティと見なされます。

修飾子 s を指定した場合、アーカイバはアーカイブ内の再配置可能オブジェクト モジュール内で定義されているシンボルに対するインデックスを生成します。このインデックスは、アーカイバがアーカイブの内容を変更した時にアーカイブ内で更新されます (q 更新オペレーション向けに保存)。アーカイブにインデックスを保存する事により、ライブラリへのリンクが高速になります。また、ライブラリ内の各ルーチンは、それらがアーカイブ内でどのように配置されていても、互いに呼び出せるようになります。

インデックス テーブルは、`xc32-nm -s` または `xc32-nm --print-armap` を使って表示できます。アーカイブがインデックス　テーブルを持っていない場合、`xc32-ranlib` (32 ビット アーカイバ / ライブラリアンの別のユーティリティ) を使ってテーブルだけ追加する事がきます。

32 ビット アーカイバ / ライブラリアンの動作は 2 通りの方法で制御できます。すなわち、コマンドライン オプションを使って制御するか、単純にコマンドライン オプション `-M` を指定して標準入力経由でスクリプトを使って制御できます。

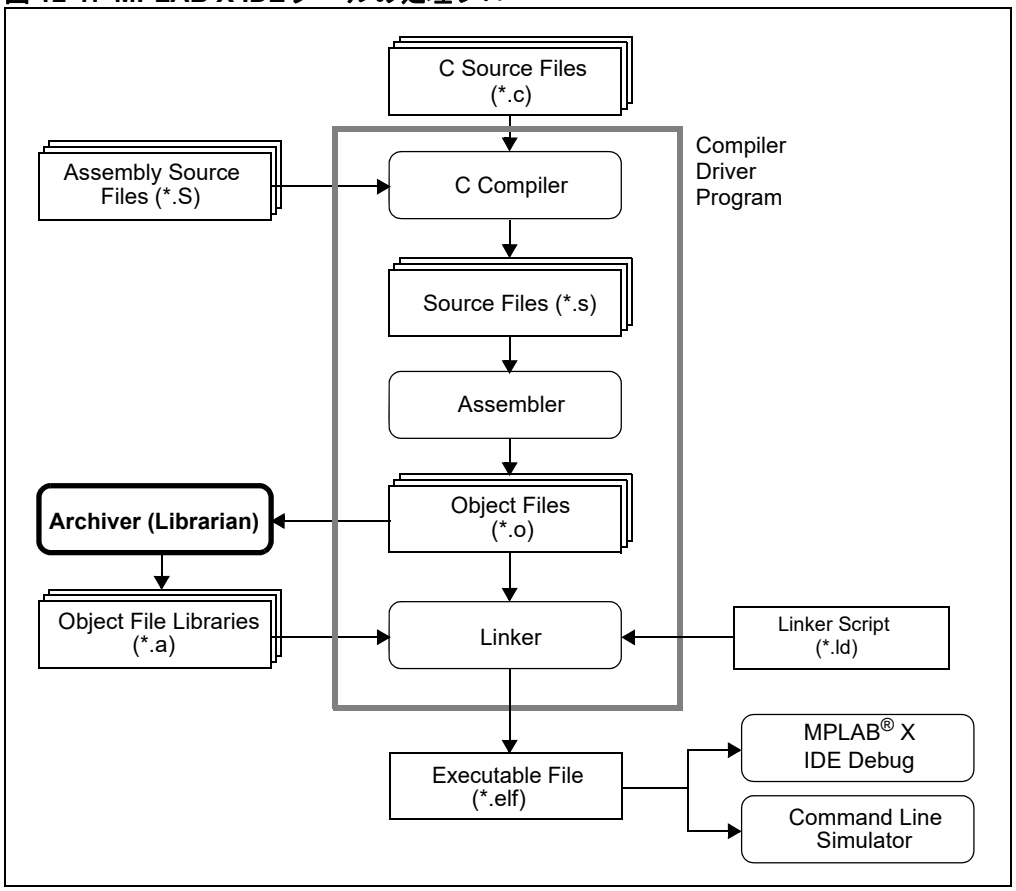
第 12 章の主な内容は以下の通りです。

- アーカイバ / ライブラリアンと、その他の開発ツール
- 特長
- 入出力ファイル
- 構文
- オプション
- スクリプト

12.2 アーカイバ/ラ イブラリアンと、その他の開発ツール

32 ビット ライブラリアンは、32 ビットアセンブラが生成したオブジェクト ファイルからアーカイブ ファイルを生成します。アーカイブ ファイルを 32 ビットリンカによって他の再配置可能オブジェクトファイルとリンクする事で、実行可能ファイルを生成できます。ツールの処理フローの概要を図 12-1 に示します。

図 12-1: MPLAB X IDEツ ールの処理フロー



12.3 特長

本アセンブラの主な特長は以下の通りです。

- Linux x86、Mac OS X、Windows に対応
- コマンドライン インターフェイス

12.4 入出力ファイル

32 ビット アーカイバ / ライブラリアンは、アーカイブ ファイル (.a) を生成します。アーカイブ ファイルとは複数のファイルを 1 つにまとめたファイルの事です。アーカイブから個々のファイル (アーカイブのメンバーと呼ぶ) を取り出す事ができます。全てのオブジェクトは ELF オブジェクト ファイル形式で処理されます。

ほとんどの場合、アーカイブ ファイルはよく使われるサブルーチンを収めた「ライブラリ」として使われるため、xc32-ar はバイナリ ユーティリティと見なされます。

12.5 構文

```
xc32-ar [-]P[MOD [RELPOS] [COUNT]] ARCHIVE [MEMBER...]
xc32-ar -M [ <mri-script ]
```

12.6 オプション

コマンドライン オプションを使って 32 ビット アーカイバ / ライブラリアンを制御する場合、アーカイバは実行のために 2 つ以上の引数を要求します。すなわち、オペレーションを指定するための 1 つのキー文字 (必要に応じて修飾子を指定するためのキー文字を追加) と、アーカイブ名を引数として指定する必要があります。

```
xc32-ar [-]P[MOD [RELPOS][COUNT]] ARCHIVE [MEMBER...]
```

Note: コマンドライン オプションは大文字と小文字を区別します。

ほとんどのオペレーションには、アーカイブ メンバーを指定するための *MEMBER* 引数も指定できます。メンバーを指定しない場合、アーカイブの全体が使われます。

32 ビット アーカイバ / ライブラリアンに対する最初のコマンドライン引数では、オペレーション コード P に修飾子フラグ *MOD* (修飾子の順番は任意) を組み合わせる事ができます。最初のコマンドライン引数の前にダッシュ文字「-」を付けても構いません。

キー文字 P は、実行するオペレーションを指定します。これには、下表のオプションの中のどれか 1 つだけが指定できます。

表 12-1: 実行するオペレーション

オペレーション	機能
d	アーカイブからモジュールを削除します。削除するモジュールの名前を「MEMBER...」として指定します。何も指定しなかった場合、アーカイブはそのままにされます。v 修飾子を指定した場合、32 ビット アーカイバ / ライブラリアンは削除した各モジュールを表示します。
m	アーカイブ内でメンバーを移動します。同一シンボルが複数のメンバー内で定義されている場合、ライブラリを使ってプログラムをリンクする時に、アーカイブ内のメンバーの並び順によってリンクの仕方が異なる可能性があります。m と一緒に修飾子を何も使わない場合、MEMBER 引数で指定されたメンバーはアーカイブの末尾に移動します。修飾子 a、b、i のいずれかを使う事で、メンバーを指定位置へ移動させる事ができます。
p	アーカイブ内の指定されたメンバーを標準出力ファイルへ出力します。v 修飾子が指定されると、メンバーの内容を標準出力にコピーする前にそのメンバーの名前を表示します。MEMBER 引数が何も指定されなかった場合、アーカイブ内の全てのファイルを出力します。
q	MEMBER... で指定されたファイルを ARCHIVE に追加します。
r	MEMBER... で指定されたファイルを ARCHIVE に挿入します (既存メンバーを置換します)。MEMBER... で指定されたファイルが存在しない場合、アーカイバはエラーメッセージを表示します。その名前に一致するアーカイブ内のメンバーは、そのまま残されます。既定値により、新しいメンバーはファイルの最後に追加されます。しかし、修飾子 a、b、i のいずれかを使う事で、既存メンバーに対する相対位置を指定する事ができます。r オプションと一緒に修飾子 v を使うと、挿入されたファイルごとに 1 行が出力されます。また、既存メンバーを削除せずにファイルを追加したのか、それとも既存メンバーをファイルで置換したのかが、文字 a または r によって示されます。
t	ARCHIVEの内容(またはMEMBER..., で指定されたファイルの内容)を示すテーブルを表示します。通常はメンバーの名前だけを表示します。v 修飾子を指定する事で、モード (パーミッション)、タイムスタンプ、オーナー、グループ、サイズも表示できます。MEMBER を指定しなかった場合、アーカイブ内の全てのファイルが表示されます。例えば、アーカイブ (b.a) の中に fie という名前のファイルが複数存在する場合、xc32-ar t b.a fie は最初に見つかったインスタンスだけを表示します。全ての file を表示するには、xc32-ar t b.a を使って全てのファイルを表示させる必要があります。
x	アーカイブからメンバー (MEMBER で指定) を抜き出します。このオプションと一緒に v 修飾子を指定する事で、抜き出した各メンバーの名前のリストを表示させる事ができます。MEMBER を指定しなかった場合、アーカイブ内の全てのファイルが抜き出されます。

Pキー文字の直後で下表に示す各種の修飾子(MOD)を指定する事により、オペレーションの挙動を変更する事ができます。

表 12-2: 修飾子

修飾子	機能
a	アーカイブ内の指定した既存メンバーの後に新しいファイルを追加します。修飾子 a を使う場合、ARCHIVE より前で、既存アーカイブ メンバーの名前を RELPOS 引数として指定する必要があります。
b	アーカイブ内の指定した既存メンバーの前に新しいファイルを追加します。修飾子 b を使う場合、ARCHIVE より前で、既存アーカイブ メンバーの名前を RELPOS 引数として指定する必要があります。これは修飾子 i と等価です。
c	アーカイブを新規作成します。アーカイブの更新を要求する際、ARCHIVE で指定されたアーカイブが既に存在していなければ、そのアーカイブは新しく作成されます。しかし、この修飾子を先に指定しなかった場合、警告が出力されます。
f	アーカイブ内の名前を切り詰めます。通常、32 ビット アーカイバ / ライブラリアンはファイル名の長さを制限しません。このため、一部のシステムにおいて、ネイティブのアーカイバ プログラムとは非互換のアーカイブを生成してしまいます。そのような場合、f 修飾子を使う事で、ファイルをアーカイブに挿入する際にファイル名を切り詰める事ができます。
i	アーカイブ内の指定した既存メンバーの前に新しいファイルを挿入します。修飾子 i を使う場合、ARCHIVE より前で、既存アーカイブ メンバーの名前を RELPOS 引数として指定する必要があります。これは修飾子 b と等価です。
l	この修飾子は使いません (使っても効果はありません)。
N	COUNT パラメータを使います。この修飾子は、アーカイブ内に複数の同一名エントリが複数存在する場合に使います。これを使うと、指定された名前のインスタンス COUNT が抜き出されます (または削除されます)。
o	アーカイブから抜き出したメンバーのタイムスタンプを元のまま保持します。この修飾子を指定しないと、アーカイブから抜き出したメンバーには抜き出した時点のタイムスタンプが付けられます。
P	フルパス名を使ってアーカイブ内の名前を照合します。32 ビット アーカイバ / ライブラリアンは、フルパス名を使ってアーカイブを生成できません (そのようなアーカイブは POSIX 非互換です)。しかし、そのようなアーカイブを生成できるアーカイバも存在します。このオプションを使うと、本アーカイバはフルパス名を使ってファイルを照合します。これは、別のツールを使って生成されたアーカイブからファイルを抜き出す場合に便利です。
s	アーカイブにオブジェクト ファイル インデックスを新しく書き込むか、既存のインデックスを更新します(アーカイブに対して他の変更を一切加えない場合でもインデックスを更新できます)。この修飾子フラグは、任意のオペレーション コードと一緒に使う事も、単独で使う事もできます。xc32-ar s は ranlib と等価です。
S	アーカイブ シンボルテーブルを生成しません。これにより、複数ステップによる大きなライブラリのビルドを高速化できます。生成されたアーカイブはリンカ向けに使えません。シンボルテーブルを生成するには、アーカイバの最後の実行で S 修飾子を外すか、ranlib をアーカイブに対して実行する必要があります。
u	通常、xc32-ar r... は、指定された全てのファイルをアーカイブに挿入します。既存メンバーより新しいファイルだけ挿入したい場合、この u 修飾子を使います。u 修飾子は、オペレーション r (置換) 向けにのみ使います。例えばオペレーション q と組み合わせた場合 (qu)、u はタイムスタンプを確認するため、q による高速化の効果は失われます。
v	この修飾子は、オペレーションの詳細 (verbose) バージョンを要求します。修飾子 v を使うと、多くのオペレーションは追加の情報 (処理されたファイルの名前等) を表示します。
V	この修飾子は、32 ビット アーカイバ / ライブラリアンのバージョン番号を表示します。

12.7 スクリプト

アーカイバでコマンドライン オプション -M を使うと、基本的なコマンドライン言語を使ってアーカイバの動作を制御する事ができます。

```
xc32-ar -M [ <SCRIPT ]
```

標準入力をターミナルから直接入力する場合、32 ビット アーカイバ / ライブラリアンはインタラクティブに動作します。アーカイバは入力プロンプト「AR >」を表示し、エラー発生後も動作を継続します。

標準入力をスクリプト ファイルへリダイレクトする場合､プロンプトは表示されません。エラーが発生すると 32 ビット アーカイバ / ライブラリアンは実行を中止します (非ゼロの終了コードを表示)。

アーカイバ コマンド言語は、コマンドライン オプションと等価となるよう設計されていません (アーカイブを制御する機能がある程度制限されます)。このコマンド言語は、MRI「ライブラリアン」プログラム向けに書かれたスクリプトを使っていたユーザが容易に 32 ビット アーカイバ / ライブラリアンに移行できるようにする事だけを目的としています。

32 ビット アーカイバ / ライブラリアン コマンド言語の構文は明解です。

- これらのコマンドは大文字と小文字を区別しません。例えば LIST と list は同じです。以下の説明では、読みやすくするために大文字でコマンドを表記します。
- コマンドは各行に 1 つだけ書く事ができます。行の最初のワードがコマンドです。
- 空白の行は許容されます (効果はなし)。
- コメントを書く事ができます (文字「*」または「;」より後のテキストは無視される)。
- xc32-ar コマンドの引数として複数の名前を指定する場合、それらの名前はコンマまたはブランクを使って区切る事ができます。以下の説明では、読みやすくするためにコンマで区切ります。
- 「+」は継続行文字として使います。行末に「+」があると、次の行は現在のコマンドの一部であると見なされます。

下の表に、アーカイバ スクリプトの中で使えるコマンドを示します。これらのコマンドは、アーカイバをインタラクティブに制御する場合にも使えます。

アーカイバ スクリプト コマンド

コマンド	機能
OPEN または CREATE	「カレント アーカイブ」を指定します。これは、大部分のコマンドに必要な一時ファイルです。
SAVE	それまでにスクリプトによって指定された変更を実際に適用します。SAVE より前のコマンドは、カレント アーカイブ内の一時的な複製にしか反映されません。
ADDLIB ARCHIVE ADDLIB ARCHIVE (MODULE, MODULE,...MODULE)	ARCHIVE の全ての内容 (または指定された各 MODULE) をカレント アーカイブに追加します。このコマンドを使う前に OPEN または CREATE を使う必要があります。
ADDMOD MEMBER, MEMBER, ...MEMBER	指定された各 MEMBER をモジュールとしてカレント アーカイブに追加します。このコマンドを使う前に OPEN または CREATE を使う必要があります。
CLEAR	カレント アーカイブの内容を破棄します。直近の SAVE から後の全てのオペレーションの効果はキャンセルされます。カレント アーカイブが指定されていなくても実行されます (効果はありません)。

アーカイバ スクリプト コマンド (続き)

コマンド	機能
CREATE ARCHIVE	アーカイブを生成し、それをカレント アーカイブにします。多くのコマンドはカレント アーカイブを必要とします。新しいアーカイブは一時名を使って生成されます。SAVE が使われるまで、このアーカイブは実際に ARCHIVE として保存されません。既存アーカイブを上書きする事もできます。同様に、SAVE が使われるまで、ARCHIVE で指定された既存ファイルの内容は上書きされません。
DELETE MODULE, MODULE, ...MODULE	指定された各MODULEをカレント アーカイブから削除します。これは xc32-ar -d ARCHIVE MODULE …MODULE と等価です。このコマンドを使う前に OPEN または CREATE を使う必要があります。
DIRECTORY ARCHIVE (MODULE, ...MODULE) [OUTPUTFILE]	ARCHIVE の中にある指定された各 MODULE を表示します。これとは別のコマンド VERBOSE によって出力の形態 (詳細かどうか) が決まります。詳細出力が OFF の場合、出力は xc32-ar -t ARCHIVE MODULE.... と同様です。詳細出力が ON の場合、出力は xc32-ar -tv ARCHIVE MODULE.... と同様です。通常、この出力は標準出力ストリームに向けて出力されます。しかし、最後の引数として OUTPUTFILE を指定すると、ファイルに出力する事ができます。
END	アーカイバの実行を終了します (終了コードは「0」(正常に終了) です)。このコマンドは出力ファイルを保存しません。直近の SAVE コマンドの後でカレント アーカイブを変更した場合、それらの変更は失われます。
EXTRACT MODULE, MODULE, ...MODULE	指定された各MODULEをカレント アーカイブから抜き出し、それらを別々のファイルとしてカレント ディレクトリに書き込みます。これは xc32-ar -x ARCHIVE MODULE.... と等価です。このコマンドを使う前に OPEN または CREATE を使う必要があります。
LIST	カレント アーカイブの全ての内容を表示します (VERBOSE の状態に関係なく、詳細形式で表示)。これは xc32-ar tv ARCHIVEと等価です(このコマンドは32ビット アーカイバ / ライブラリの拡張機能として提供されます (MRI 互換性のためではありません))。このコマンドを使う前に OPEN または CREATE を使う必要があります。
OPEN ARCHIVE	既存のアーカイブを開き、それをカレント アーカイブとして使います。後続のコマンドによる変更は、SAVE が使われるまで ARCHIVE に適用されません。
REPLACE MODULE, MODULE, ...MODULE	カレント アーカイブの中の各既存 MODULE (REPLACE の引数で指定) を、現在の作業中ディレクトリ内にあるファイルで置き換えます。このコマンドを正常に実行するには、ファイルとカレント アーカイブ内のモジュールが両方とも存在している必要があります。このコマンドを使う前に OPEN または CREATE を使う必要があります。
VERBOSE	DIRECTORYコマンドからの出力の形態(詳細にするかどうか) をトグルします。フラグが ON の場合の DIRECTORY 出力は xc32-ar -tv .... による出力と同じです。
SAVE	それまでにカレント アーカイブに適用された変更を、直近の CREATE または OPEN コマンドで指定された名前を持つファイルとして実際に保存します。このコマンドを使う前に OPEN または CREATE を使う必要があります。

NOTE:

第 13 章　その他のユーティリティ

13.1　はじめに

MPLAB XC32 オブジェクト アーカイバ / ライブラリアン (xc32-ar) の他にも各種のライブラリ ユーティリティが MPLAB XC32 アセンブラ / リンカ向けに利用できます。

第 13 章では、以下のユーティリティについて説明します。

ユーティリティ	概要
xc32-bin2hex	リンク済みのオブジェクト ファイルを Intel® HEX ファイルに変換します。
xc32-nm	オブジェクト ファイル内のシンボルの一覧を表示します。
xc32-objdump	オブジェクト ファイルに関する情報を表示します。
xc32-ranlib	アーカイブの内容からインデックスを生成し、それをアーカイブに保存します。
xc32-size	ファイルのセクション サイズと総サイズを表示します。
xc32-strings	印字可能文字の並びを出力します。
xc32-strip	オブジェクト ファイル内の全てのシンボルを破棄します。

13.2 xc32-bin2hexユ ーティリティ

xc32-bin2hex ユーティリティは、32 ビットリンカからのバイナリファイルを、デバイス プログラマ向けの Intel HEX 形式ファイルに変換します。

13.2.1 入出力ファイル

- 入力 : ELF 形式バイナリ オブジェクト ファイル
- 出力 : Intel HEX 形式ファイル

13.2.2 構文

コマンドラインの構文は以下の通りです。

```
xc32-bin2hex [options] file
```

例 13.1: hello.elf

絶対 ELF 実行ファイル hello.elf を hello.hex に変換します。

```
xc32-bin2hex hello.elf
```

13.2.3 オプション

以下のオプションをサポートします。

表 13-1: xc32-bin2hexの オプション

オプション	機能
-a、--sort	セクションをアドレス順に並べ換えます。
-i、--virtual	仮想アドレスを使います。
-p、--physical	物理アドレスを使います **(既定値)**。
-v、--verbose	メッセージを詳細 (verbose) 形式で出力します。
-?、--help	ヘルプ画面を出力します。

Note: PIC32MX の仮想 - 物理メモリ間の固定マッピングについては『PIC32MX ファミリ リファレンス マニュアル、セクション 03. メモリ構成』(DS61115) を参照してください。

例 13.2: -v オプションの出力

```
writing hello.hex

section  PC address   byte address   length (w/pad)   actual length  (dec)
-------   ----------   ------------   --------------   --------------------
.reset             0              0              0x8            0x6  (6)
.text          0x100          0x200           0x6a28         0x4f9e  (20382)
.dinit        0x3614         0x6c28            0xda4          0xa3b  (2619)
.const        0x3ce6         0x79cc             0x40           0x30  (48)
.ivt             0x4            0x8             0xf8           0xba  (186)
.aivt           0x84          0x108             0xf8           0xba  (186)

Total program memory used (bytes):0x5b83  (23427)
```

13.3 xc32-nmユ ーティリティ

xc32-nm ユーティリティは、オブジェクト ファイルからシンボルのリストを生成します。リスト内の各アイテムはシンボルの値、型、名前で構成されます。

13.3.1 入力ファイル

• 入力 : ELF オブジェクト ファイル

引数としてオブジェクト ファイルが何も指定されない場合、xc32-nm はファイル a.out が指定されたと見なします。

13.3.2 構文

コマンドラインの構文は以下の通りです。

```
xc32-nm [ -A | -o | --print-file-name ]
      [ -a | --debug-syms ] [ -B ]
      [ --defined-only ] [ -u | --undefined-only ]
      [ -f format | --format=format ] [ -g | --extern-only ]
      [ --help ] [-l | --line-numbers ]
      [ -n | -v | --numeric-sort ]  [-omf=format]
      [ -p | --no-sort ]
      [ -P | --portability ] [ -r | --reverse-sort ]
      [ -s  --print-armap ] [ --size-sort ]
      [ -t radix | --radix=radix ] [ -V | --version ]
      [ OBJFILE...]
```

13.3.3 オプション

以下の表 13-2 には、各オプションを長い形式と短い形式で記載しています。どちらを指定しても効果は同じです。

表 13-2: xc32-nmオ プション

オプション	機能
-A -o --print-file-name	各シンボルの前に、そのシンボルが見つかった入力ファイル (またはアーカイブ メンバー) の名前を表示します (入力ファイルを 1 度表示した後にその中の全てのシンボルをまとめて示すのではなく、シンボルごとに入力ファイルを示します)。
-a --debug-syms	全てのシンボルを表示します。通常は表示されないデバッガ専用シンボルも表示します。
-B	--format=bsd と等価です。
--defined-only	各オブジェクト ファイル向けに定義されているシンボルだけを表示します。
-u --undefined-only	未定義のシンボル (各オブジェクト ファイルに対して外部のシンボル) だけを表示します。
-f *format* --format=*format*	出力フォーマットとして *format* (bsd、sysv、posix のいずれか) を使います。既定値は bsd です。*format* の最初の文字だけが有意です (大文字と小文字は区別しません)。
-g --extern-only	外部シンボルのみ表示します。
--help	xc32-nm 向けオプションの要約を表示した後に終了します。
-l --line-numbers	デバッグ情報を使って各シンボルのファイル名と行番号を検索します。定義済みシンボルの場合、シンボルのアドレスの行番号を検索します。未定義シンボルの場合、シンボルを参照している再配置エントリの行番号を検索します。行番号情報が見つかった場合、他のシンボル情報の後に行番号を出力します。
-n -v --numeric-sort	シンボルを名前 (アルファベット) 順ではなくアドレスによる値順に並べ換えます。
-p --no-sort	シンボルを遭遇した順番に (並べ換えずにそのまま) 出力します。
-P --portability	既定値フォーマットではなく POSIX.2 標準出力フォーマットを使います。これは -f posix と等価です。
-r --reverse-sort	アドレスの値順またはアルファベット順の並べ換えを逆順にします。
-s --print-armap	アーカイブ メンバーからのシンボルの表示にインデックスを含めます (インデックスとは xc32-ar または xc32-ranlib によってアーカイブに保存されたマッピングであり、どのモジュールがどの名前の定義を格納しているのか示します)。
--size-sort	シンボルをシンボルの変数のサイズ順に並べ換えます。サイズは、あるシンボルの値と次に高い値を持つシンボルの値の差として計算されます。シンボルの値ではなくシンボルの変数のサイズが出力されます。
-t *radix* --radix=*radix*	*radix* を基数としてシンボル値を出力します。*radix* には d (10 進値)、o (8 進値)、x (16 進値) のいずれかを指定する必要があります。
-V --version	xc32-nm のバージョン番号を表示した後に終了します。

13.3.4 出力書式

シンボル値の基数はオプションにより選択します (既定値は 16 進)。

シンボルの型が小文字の場合、そのシンボルはローカルです。大文字場合、そのシンボルはグローバル (外部) です。表 13-3 にシンボルの型を示します。

表 13-3: シンボルの型

型	概要
A	この型のシンボルの値は絶対的であり、後続のリンクによって変更されません。
B	この型のシンボルは非初期化データセクション (BSS) 内に配置されます。
C	この型のシンボルは共有シンボルです。共有シンボルは非初期化データです。リンク時に、同一名を持つ複数の共有シンボルが存在しても構いません。そのシンボルがどこかで定義されている場合、共有シンボルは未定義参照として扱われます。
D	この型のシンボルは初期化データセクション内に配置されます。
N	この型のシンボルはデバッグシンボルです。
R	この型のシンボルは読み出し専用データセクション内に配置されます。
T	この型のシンボルはテキスト (コード) セクション内に配置されます。
U	この型のシンボルは未定義です。
V	この型のシンボルは weak オブジェクトです。weak 定義のシンボルが通常定義のシンボルとリンクされた場合、通常定義のシンボルがエラーなしで使われます。非 weak 定義のシンボルがリンクされ、そのシンボルが未定義である場合、weak シンボルの値はエラーなしで 0 になります。
W	この型のシンボルは、weak オブジェクト シンボルとして特別にタグ付けされていない weak シンボルです。weak 定義のシンボルが通常定義のシンボルとリンクされた場合、通常定義のシンボルがエラーなしで使われます。非 weak 定義のシンボルがリンクされ、そのシンボルが未定義である場合、weak シンボルの値はエラーなしで 0 になります。
?	この型のシンボルは未知のシンボルまたはオブジェクト ファイルのフォーマットに固有のシンボルです。

13.4 xc32-objdumpユ ーティリティ

xc32-objdump ユーティリティは、1 つまたは複数のオブジェクト ファイルに関する情報を表示します。オプションにより、特定の情報を表示するかどうか制御します。このユーティリティの出力は、逆アセンブリ リスティングに類似した情報を提供できます。

13.4.1 入力ファイル

• 入力 : オブジェクト アーカイブ ファイル

引数としてオブジェクト ファイルが何も指定されない場合、xc32-objdump はファイル a.out が指定されたと見なします。

13.4.2 構文

コマンドラインの構文は以下の通りです。

```
xc32-objdump [ -a | --archive-headers ]
      [ -d | --disassemble ]
      [ -D | --disassemble-all ]
      [ -f | --file-headers ]
      [ --file-start-context ]
      [ -g | --debugging ]
      [ -h | --section-headers | --headers ]
      [ -H | --help ]
      [ -j name | --section=name ]
      [ -l | --line-numbers ]
      [ -M options | --disassembler-options=options]
      [ --prefix-addresses]
      [ -r | --reloc ]
      [ -s | --full-contents ]
      [ -S | --source ]
      [ --[no-]show-raw-insn ]
      [ --start-address=address ]
      [ --stop-address=address ]
      [ -t | --syms ]
      [ -V | --version ]
      [ -w | --wide ]
      [ -x | --all-headers ]
      [ -z | --disassemble-zeros ]
      OBJFILE...
```

OBJFILE… は、調査対象のオブジェクト ファイルです。アーカイブを指定した場合、xc32-objdump は各メンバー オブジェクト ファイルの情報を表示します。

13.4.3 オプション

以下の表 13-4 には、各オプションを長い形式と短い形式で記載しています。どちらを指定しても効果は同じです。
以下のオプションの少なくとも 1 つを指定する必要があります :
-a、-d、-D、-f、-g、-G、-h、-H、-p、-r、-R、-S、-t、-T、-V、-x

表 13-4: xc32-objdumpオ プション

オプション	機能
-a --archive-header	OBJFILE ファイルの中のいずれかがアーカイブである場合、アーカイブ ヘッダ情報を表示します (表示形式は ls -l に類似)。xc32-ar tv, xc32-objdump -a により、ヘッダ情報に加えて各アーカイブ メンバーのオブジェクト ファイル フォーマットを表示できます。
-d --disassemble	OBJFILE から、マシン命令に対応するアセンブラ ニーモニックを表示します。このオプションは、命令を格納していると期待されるセクションだけを逆アセンブルします。
-D --disassemble-all	全てのセクションの内容を逆アセンブルします(これとは異なり、-d は命令を格納していると期待されるセクションだけを逆アセンブルします)。
-f --file-header	各OBJFILEファイルのヘッダ全体からの情報の要約を表示します。
--file-start-context	まだ表示されていないファイルからソースコードと逆アセンブル コードを一緒に表示する場合に (「-S」を想定)、コンテクストをファイルの先頭へ展開するよう指定します。
-g --debugging	デバッグ情報を表示します。ファイルに保存されているデバッグ情報を解析し、その結果を C 言語に似た構文を使って出力します。特定タイプのデバッグ情報だけが実装済みです。
-h --section-header --header	オブジェクト ファイルのセクションヘッダからの情報の要約を表示します。ファイル セグメントは非標準アドレスに再配置される場合があります (例 : ld に対して -Ttext、-Tdata、-Tbss オプションのいずれかを使った場合)。しかし、一部のオブジェクト ファイル フォーマット (a.out 等) は、ファイル セグメントの開始アドレスを保存しません。そのような場合、ld はセクションを正しく再配置しますが、xc32-objdump -h を使ってファイル セクション ヘッダを表示しても正しいアドレスは表示されません。代わりに、ターゲットに対して暗黙の通常アドレスが表示されます。
-H --help	xc32-objdump 向けオプションの要約を表示して終了します。
-j *name* --section=*name*	*name* で指定されたセクションの情報のみ表示します。
-l --line-numbers	デバッグ情報を使った表示に、オブジェクト コードまたは relocs に対応するファイル名とソース行番号を付けます。このオプションは -d、-D、-r とのみ組み合わせる事ができます。
-M *options* --disassembler- options=*options*	ターゲットに固有の情報を逆アセンブラに渡します。PIC32 デバイスは以下のターゲット固有オプションをサポートします。 symbolic - シンボリック逆アセンブリを実行します。
--prefix-addresses	逆アセンブルする時に、各行で完全なアドレスを出力します。これは旧式の逆アセンブリ フォーマットです。
-r --reloc	ファイルの再配置エントリを出力します。このオプションを -d または -D と一緒に使った場合、再配置エントリは逆アセンブル コードと混じり合って出力されます。
-s --full-contents	要求されたセクションの完全な内容を表示します。

表 13-4: xc32-objdumpオ プション (続 き)

オプション	機能
-S --source	可能であれば、ソースコードを逆アセンブルコードと一緒に表示します。このオプションは暗黙的に-dを指定します。
--show-raw-insn	命令を逆アセンブルする時に、16 進形式とシンボリック形式の両方で命令を出力します。--prefix-addresses を使う場合を除き、これが既定値です。
--no-show-raw-insn	命令を逆アセンブリする時に、命令バイトを出力しません。--prefix-addresses を使う場合、これが既定値です。
--start-address=*address*	指定されたアドレスでデータの表示を開始します。これは-d、-r、-s オプションの出力に影響します。
--stop-address=*address*	指定されたアドレスでデータの表示を停止します。これは-d、-r、-s オプションの出力に影響します。
-t --syms	ファイルのシンボルテーブル エントリを出力します。これは、xc32-nmユーティリティが提供する情報に類似します。
-V --version	xc32-objdump のバージョン番号を表示した後に終了します。
-w --wide	列数が80 を超える行を出力デバイス向けにフォーマットします。
-x --all-header	入手可能な全てのヘッダ情報(シンボルテーブルと再配置エントリを含む) を表示します。-x は、-a -f -h -r -t の全てを指定したのと等価です。
-z --disassemble-zeroes	通常、逆アセンブル出力は 0 のブロックをスキップします。このオプションを指定すると、逆アセンブラは 0 のブロックを他のデータと同様に逆アセンブルします。

13.5 xc32-ranlibユ ーティリティ

xc32-ranlib ユーティリティは、アーカイブの内容に対してインデックスを生成し、それをアーカイブ内に保存します。インデックスは、アーカイブのメンバー (再配置可能オブジェクト ファイル) によって定義されている各シンボルを示します。このインデックスは、xc32-nm -s または xc32-nm --print-armap を使って表示できます。アーカイブにインデックスを保存する事により、ライブラリへのリンクが高速になります。また、ライブラリ内の各ルーチンは、それらがアーカイブ内でどのように配置されていても、互いに呼び出せるようになります。

xc32-ranlibは、xc32-ar -s (32ビット アーカイバ/ライブラリアンで -sオプションを指定) と完全に等価です。

13.5.1 入出力ファイル

- 入力 : アーカイブ ファイル
- 出力 : アーカイブ ファイル

13.5.2 構文

コマンドラインの構文は以下の通りです。

```
xc32-ranlib [-v | -V | --version] ARCHIVE
```

xc32-ranlib [-h | --help]

13.5.3 オプション

以下の表には、各オプションを長い形式と短い形式で記載しています。どちらを指定しても効果は同じです。

表 13-5: xc32-ranlibオ プション

オプション	機能
-v -V --version	xc32-ranlib のバージョン番号を表示して終了します。
-h --help	ヘルプメッセージを出力します。

13.6 xc32-sizeユ ーティリティ

xc32-size ユーティリティは、引数リスト内の各オブジェクト ファイルまたはアーカイブ ファイルのセクション サイズと総サイズを表示します。既定値により、各オブジェクト ファイルまたはアーカイブ内の各モジュールに対して 1 行の出力が生成されます。

Note: リンカの --report-mem は、メモリ使用量に関する追加の情報を提供します。

13.6.1 入出力ファイル

- 入力 : オブジェクト ファイルまたはアーカイブ ファイル (複数可)
- 出力 : 標準出力

13.6.2 構文

xc32-size のコマンドライン構文は以下の通りです。

```
xc32-size [ -A | -B | --format=compatibility ]
[ --help ]
[ -d | -o | -x | --radix= number]
[ -t | --totals ]
[ -V | --version ]
[objfile...]
```

13.6.3 オプション

xc32-size のオプションは以下の通りです。

表 13-6: xc32-sizeオ プション

オプション	機能
-A -B --format=*compatibility*	これらのオプションの 1 つを使う事で、gnu size からの出力フォーマットを選択できます -A または --format=sysv: System V size からの出力に類似のフォーマット、 -B または --format=berkeley: Berkeley size からの出力に類似のフォーマット)。 既定値は、Berkeley に類似の 1 行フォーマットです。
--help	使用可能な引数とオプションの要約を表示します。
-d -o -x --radix=*number*	これらのオプションの 1 つを使う事で、各セクションのサイズの表記方法を選択できます -d または --radix=10: 10 進値 -o または --radix=8: 8 進値 -x または --radix=16: 16 進値 *number* には 8、10、16 のいずれかのみ指定できます。 総サイズは、-d または -x の場合に 10 進値と 16 進値で示され、-o の場合に 8 進値と 16 進値で示されます。
-t --totals	指定された全てのオブジェクトの総計を表示します (Berkeley 形式のリスティング モードのみ)
-V --version	xc32-size のバージョン番号を表示します。

13.6.4 例

以下は、Berkeley 形式 (既定値) での xc32-size からの出力例です。

```
xc32-size --format=Berkeley ranlib size
text   data  bss   dec    hex   filename
294880 81920 11592 388392 5ed28 ranlib
294880 81920 11888 388688 5ee50 size
```

以下は、同じデータの System V 形式での出力例です。

```
xc32-size --format=SysV ranlib size
ranlib :
section   size   addr
.text   294880   8192
.data    81920 303104
.bss     11592 385024

Total 388392
size :
section   size   addr
.text   294880   8192
.data    81920 303104
.bss     11888 385024
Total 388688
```

13.7 xc32-stringsユ ーティリティ

xc32-strings ユーティリティは、指定された各ファイルの中で印字可能文字が 4 文字以上 (またはオプションで指定された文字数以上) 連続する部分を見つけて出力します。既定値により、オブジェクト ファイルの初期化およびロード済みセクション内の文字列だけが出力されます。他のタイプのファイルの場合、ファイル全体から文字列が出力されます。

xc32-strings は、主に非テキストファイルの内容を確認するために役立ちます。

- 入出力ファイル
- 構文
- オプション

13.7.1 入出力ファイル

- 入力 :ELF オブジェクト ファイル
- 出力 : 標準出力

13.7.2 構文

コマンドラインの構文は以下の通りです。

```
xc32-strings [-a | --all | -] [-f | --print-file-name]
      [--help] [-min-len | -n min-len | --bytes=min-len]
      [-t radix | --radix=radix] [-v | --version] FILE...
```

13.7.3 オプション

以下の表 13-7 には、各オプションを長い形式と短い形式で記載しています。どちらを指定しても効果は同じです。

表 13-7: xc32-stringsオ プション

オプション	機能
-a --all -	オブジェクト ファイルの初期化およびロード済みセクションだけでなく、ファイル全体をスキャンします。
-f --print-file-name	各文字列の前にファイル名を出力します。
--help	標準出力にこのユーティリティの使い方を出力した後に終了します。
-min-len -n *min-len* --bytes=*min-len*	既定値の 4 文字ではなく、*-min-len* 文字以上の文字の並びを出力します。
-t *radix* --radix=*radix*	各文字列の前にファイル内のオフセットを出力します。1 文字の引数によりオフセットの基数を指定します (-o: 8 進値、-x: 16 進値、-d: 10 進値)。
-v --version	標準出力にユーティリティのバージョン番号を出力した後に終了します。

13.8 xc32-stripユ ーティリティ

xc32-stripユーティリティは､指定されたオブジェクト ファイルまたはアーカイブ ファイルから全てのシンボルを破棄します｡少なくとも 1 つのファイルを指定する必要があります。xc32-strip は、引数で指定されたファイルそのものを変更します (変更した複製を別の名前で書き込むのではありません)。

13.8.1 入出力ファイル

- 入力 : オブジェクト ファイルまたはアーカイブ ファイル
- 出力 : オブジェクト ファイルまたはアーカイブ ファイル

引数としてオブジェクト ファイルもアーカイブ ファイルも指定されない場合、xc32-strip はファイル a.out が指定されたと見なします。

13.8.2 構文

コマンドラインの構文は以下の通りです。

```
xc32-strip [ -g | -S | --strip-debug ] [ --help ]
      [ -K symbolname | --keep-symbol=symbolname ]
      [ -N symbolname | --strip-symbol=symbolname ]
      [ -o file ]
      [ -p | --preserve-dates ]
      [ -R sectionname | --remove-section=sectionname ]
      [ -s | --strip-all ] [--strip-unneeded]
      [ -v | --verbose ]  [ -V | --version ]
      [ -x | --discard-all ] [ -X | --discard-locals ]
      OBJFILE...
```

13.8.3 オプション

以下の表 13-8 には、各オプションを長い形式と短い形式で記載しています。どちらを指定しても効果は同じです。

表 13-8: xc32-stripオ プション

オプション	機能
-g -S --strip-debug	デバッグシンボルのみ削除します。
--help	xc32-strip 向けオプションの要約を表示した後に終了します。
-K *symbolname* --keep-symbol=*symbolname*	ソースファイルから、*symbolname* で指定されたシンボルのみ保持します。このオプションは複数回指定できます。
-N *symbolname* --strip-symbol=*symbolname*	ソースファイルから、*symbolname* で指定されたシンボルのみ削除します。このオプションは複数回指定できます。また、-K を除く xc32-strip オプションと組み合わせて使う事ができます。
-o *file*	ストリップした出力を *file* で指定されたファイルに書き込みます(既存のファイルを上書きするのではありません)。この引数を使う場合、OBJFILE 引数は 1 つだけ指定できます。
-p --preserve-dates	ファイルのアクセスおよび変更の日付を保持します。
-R *sectionname* --remove-section=*sectionname*	*sectionname* で指定されたセクションを出力ファイルから削除します。このオプションは複数回指定できます。このオプションを不適切に使うと、出力ファイルが使用不能になる可能性があります。
-s --strip-all	全てのシンボルを削除します。
--strip-unneeded	再配置処理にとって不要な全てのシンボルを削除します。
-v --verbose	変更された全てのオブジェクト ファイルを示す詳細 (Verbose) 出力を表示します。アーカイブの場合、xc32-strip -v はアーカイブの全てのメンバーを表示します。
-V --version	xc32-strip のバージョン番号を表示した後に終了します。
-x --discard-all	非グローバル シンボルを削除します。
-X --discard-locals	コンパイラが生成したローカルシンボルを削除します (通常それらのシンボルは「L」または「.」で始まります)。

パート 4 - 補遺

NOTE:

補遺 A. 非推奨の機能

A.1 はじめに

以下では、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。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。

A.2 セクションを定義するためのアセンブラ ディレクティブ

以下の .section ディレクティブ フォーマットは XC32 v2.00 で非推奨となりました。新しいディレクティブ フォーマットは **4.2「セクションを定義するディレクティブ」**に記載しています。

.section *name* [, *flags*] [, @*type*]

.sectionディレクティブは､後続のコードをnameで指定されたセクション内へアセンブルするために使います。オプションの *flags* 引数は、以下の文字を任意に組み合わせた文字列です。この引数は二重引用符で囲む必要があります。

a セクションは割り当て可能

w セクションは書き込み可能

x セクションは実行可能

@*type* 引数には以下のどちらかが使えます。

@progbits 内容を含んだ通常のセクション

@nobits データを含まないセクション (空間を占有するだけのセクション)

以下のセクション名が認識されます。

表 A-1: セクション名

セクション名	既定値フラグ
.text	x
.data	d
.bss	b

Note: *flags* は二重引用符で囲む必要があります。.section ディレクティブに対するオプション引数を引用符で囲まなかった場合、それはサブセクション 番号と見なされます。一重引用符で囲んだ 1 文字 (例 : 'b') は、プリプロセッサによって数字に変換されるという事に注意が必要です。

.section ディレクティブの例

```
.section foo,"aw",@progbits #foo is initialized
#data memory.
.section fob,"aw",@nobits #fob is uninitialized
#(but also not zeroed)
#data memory.
.section bar,"ax",@progbits #bar is in program memory
```

補遺 B. 便利なテーブル

B.1 はじめに

資料として役に立つ以下のテーブルを記載します。

- ASCII キャラクタセット
- 16 進値から 10 進値への変換

B.2 ASCIIキ ャラクタセット

以下のテーブルに ASCII 標準キャラクタセットを示します。

	上位ワード								
	HEX	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
下位コードニブル	**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

B.3 16進 値から 10進 値への変換

以下に 16 進値と 10 進値の対応表を示します。表からは、16 進値の各桁に対応する 10 進値が読み取れます。各桁の値は合計します。

上位バイト				下位バイト			
Hex 1000	10 進数	Hex 100	10 進数	Hex 10	10 進数	Hex 1	10 進数
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

例えば、hex A38F は以下の手順で 41871 に変換できます。

16 進値の 1000 の桁	16 進値の 100 の桁	16 進値の 10 の桁	16 進値の 1 の桁	結果
40960	768	128	15	41871 (10 進値)

補遺 C. GNU フリー文書利用許諾契約書

NOTE:

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を提供する。

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 (0,1, ･･･ ,9) の数字の総称。

AND 条件ブレークポイント (ANDed Breakpoints)
プログラムの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 / ライブラリ (Archive/Library)

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。

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 Bit)
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 Directive)
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。残りのファイルはシンボルへの参照を含む。

D

データ ディレクティブ (Data Directive)

アセンブラが行う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) で参照する手段として使う。

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: Data Monitor and Control Interface)

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 Feature)

後方互換性確保のためにサポートしているだけで現在は使っておらず、いずれ廃止にな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 (Directive)

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命令文。

ダウンロード (Download)

ホストから別のデバイス (例 : エミュレータ、プログラマ、ターゲットボード) 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。

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。関数のユーザコードの後、関数リターンの直前にエピローグを実行する。

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。

外部リンケージ (External Linkage)
関数または変数が、それを定義し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)
デバイス外部にある、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。エラーの発生後はメッセージも出力されない。

ファイルレジスタ (File Register)
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。

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われるメモリ領域。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われる。

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 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。

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 (Instruction)
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、それを定義したモジュールの外部から参照できない場合、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始めとする、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
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を再開する。

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 (IRQ: Interrupt Service Request)
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: Interrupt Service Routine)
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 / ライブラリ」参照。

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 File)

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 Directive)

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、電圧レベルでなく差動入力の電位差が正か負かでのみデータの意味を区別する。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) からなる。ある特定のプロセッサで使える全ての命令の集合を「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 Directive)
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。「µC」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、多くの機能を統合したチップ。

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使えない。

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を外部にマッピングする。

ニーモニック (Mnemonic)
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。

MPASM™ 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う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、「MPLAB C Compiler for PIC18 MCU」を使う。

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ミングに対応。MPLAB IDE/MPLAB X IDE と一緒に使う事も、単体で使う事も可能。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。

MPLIBTM オブジェクト ライブラリアン

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 モジュールに使用するオブジェクト 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 と一緒に使う事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、[Watches] ウィンドウで使う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) である事が必要。

入れ子の深さ (Nesting Depth)

マクロに他のマクロを入れ込める階層の数。

ノード (Node)

MPLAB IDE/MPLAB X IDE のプロジェクト 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 (例 : ライブラリ) 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 Directive)

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 (Operator)

定義可能な式を構成する際に使う「+」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を評価する。

OTP (One-Time-Programmable)

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パーソナル コンピュータ。

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。

PIC MCU

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-in)

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、各種ソフトウェアおよびハードウェア 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の短縮が特に重要である。

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。

Program Memory

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 (例 : ソースコード、リンカスクリプトファイル) 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 Signal)

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 (KB)

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。

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)

アドレスがメモリの固定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)

アプリケーションの実行につれて変数の値が変化する [Watch] ウィンドウ。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 シミュレータで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。

SQTP (Serialized Quick Turn Programming)

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ル化するソフトウェア 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表れる。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、並びに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、ブレークポイント以降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する命令の数をスキッドと呼ぶ。

ソースコード (Source Code)
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で実行される。

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。

スタック、ソフトウェア (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。

コンパイルド スタック (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。書き込み済みアプリケーションの動作を確認し、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える。

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、サブルーチン内をステップ実行せずにコードをデバッグできる。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するデータ。通常、テキストファイルにアクションのリストとしてこのデータを記述する。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)

実行サイクルを計測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に対応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 ファイルに保存する。

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)
エミュレータが内蔵する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 (Trigraph)
「??」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。

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 は最大 12 Mbps のデータレートをサポートしている。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 [Watches] ウィンドウで観察できる変数。

[Watches] 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: Watchdog Timer)

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 の有効化または無効化、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:

索引

記号

NOTE:

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