

CodeWarrior®

SH-4 Assembler

Reference



CodeWarrior は出荷直前でも改良されることがあるので、このマニュアルに記載されている内容の一部が本物のソフトウェアの動きと異なることがあるかもしれません。最新の情報については CodeWarrior の「Release Notes」フォルダをご覧ください。

Revised: 990224 rw-JP990909

Metrowerks CodeWarrior © Copyright 1993-2000 by Metrowerks Inc. and its Licensors. All rights reserved.

お客様は、本 CD に記録されている文書を個人使用目的に限り、プリントすることができます。この場合を除いて、Metrowerks Inc. からの書面による承諾なしに、本 CD に記録されている文書の全部、または、一部をいかなる形態、方法（電子的、物理的な複製、または、写真複写、録音録画、その他すべての情報記録、再生システムを含む）により、複製または伝達することを禁じます。

Metrowerks の名称、ロゴ、CodeWarrior、Software at Work は、Metrowerks Inc. の登録商標です。

PowerPlant、PowerPlant Constructor は、Metrowerks Inc. の商標です。

記載の商標および登録商標は、各社が保有します。

CD に記録されているすべてのソフトウェアおよび文書は、CodeWarrior QuickStart の巻末に記述されているライセンス契約が適用されます。

連絡先：

Japan	メトロワークス株式会社 150-0042 東京都渋谷区宇田川町 36-6 ワールド宇田川ビル 8F TEL : (03) 3780-6091 FAX : (03) 3780-6092
U.S.A.	Metrowerks Corporation 9801 Metric Boulevard, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
WWW サーバ	http://www.metrowerks.co.jp/ http://www.metrowerks.com/
ユーザー登録	j-register@metrowerks.com
テクニカルサポート	j-emb-sup@metrowerks.com
購入 / 契約更新	j-sales@metrowerks.com
インフォメーション	j-info@metrowerks.com

目次

第 1 章 はじめに	5
アセンブラマニュアルの概要	5
Metrowerks アセンブラとは何か	5
このマニュアルで使われる約束事	6
もっと学ぶには	6

第 2 章 アセンブラの文法	7
アセンブラの文法の概要	7
ステートメントの文法	7
シンボルの文法	8
シンボルのスコープ	9
局所ラベル	9
グローバルな記号定数	10
再配置可能なラベル	11
定数の文法	12
整数定数	12
浮動小数点定数	12
文字定数	13
数式の文法	13
前方参照の文法	15
データのアラインメント	16

第 3 章 マクロの使い方	17
マクロの使い方の概要	17
マクロの定義	17
マクロ定義の文法	17
マクロ引数の使い方	18
ユニークなラベルの生成	19
引数の数の参照	20
マクロの呼び出し	20

第 4 章 疑似命令の使い方	21
疑似命令の使い方の概要	21
マクロ疑似命令	21
条件プロセッサ疑似命令	22
セクション制御疑似命令	25
スコープ制御疑似命令	28
シンボル定義疑似命令	29
データ定義疑似命令	30

整数型の宣言	30
文字列型の宣言	31
浮動小数点型の宣言	32
アセンブラ制御疑似命令	33
デバッグ疑似命令	35
<hr/>	
第 5 章 SH アセンブラの設定	37
アセンブラの設定の概要	37
アセンブラの設定パネル	37
<hr/>	
索引	41

第 1 章 はじめに

このマニュアルでは Metrowerks アセンブラの文法、および SH アセンブラの使い方について説明します。

アセンブラマニュアルの概要

このマニュアルでは、いくつもの異なるプロセッサに対応したアセンブラの集まりである Metrowerks アセンブラについて説明します。ステートメント、マクロ、疑似命令の文法について解説します。

このマニュアルでは、アセンブラとプログラムを書こうとしているプロセッサについて、読者が既に知識があると仮定しています。

このマニュアルでは以下の内容について説明します。

[アセンブラの文法の概要](#)

[マクロの使い方の概要](#)

[疑似命令の使い方の概要](#)

[アセンブラの設定の概要](#)

各プロセッサの命令のニーモニックやレジスタ名についての詳しい情報は、[「もっと学ぶには」\(p. 6\)](#)で紹介されている本を参照してください。

Metrowerks アセンブラとは何か

Metrowerks アセンブラとはいくつもの異なるプロセッサに対応したアセンブラの集まりです。それらはすべて同じステートメント、疑似命令とマクロを持ちます。違うのはそれぞれのプロセッサで使われる命令のニーモニックとレジスタ名だけです。Metrowerks アセンブラのリストを挙げます。

SH アセンブラは SH-4 プロセッサをサポートします。

注意： このマニュアルで説明するアセンブラコマンドは Metrowerks アセンブラのみに適用されるので、メトロワークス C/C++ コンパイラに含まれるインラインアセンブラと混乱しないようにしてください。C/C++ インラインアセンブラの使い方についての詳しい情報は「Targeting Dreamcast」および「C Compilers Reference」を参照してください。

このマニュアルで使われる約束事

このマニュアルではステートメントの使い方を説明する例文を紹介しています。[表 1.1](#) はこれらのステートメントをどう解釈するのかを示しています。

表 1.1 例文の理解の仕方

テキストのスタイル	説明
<code>literal</code>	リテラル：ステートメントを印刷された通りに正確に入力してください。
<code>metasymbol</code>	メタシンボル：シンボルを適切な値で置き換えてください。例文の後の文章で適切な値とは何かを説明します。
<code>a b c</code>	文中にはシンボルのどれか一つだけを使います。a, b, c のどれかです。
<code>[a]</code>	必要なときのみシンボルを使います。例文の後の文章で、いつ使うべきかを説明します。

もっと学ぶには

アセンブラは次に示す文書で定義された標準的なニーモニックとレジスタ名を使用します。

SH アセンブラ：「SH-4 ハードウェアマニュアル」(日立制作所)

第 2 章 アセンブラの文法

この章では Metrowerks アセンブラのソースファイルを書くために必要な文法規則を説明します。

アセンブラの文法の概要

この章では Metrowerks アセンブラのソースファイルを書く方法について解説します。読者がアセンブラと書きたいプロセッサの機械語について、すでに知識があることを前提としてこの章は書かれています。この章ではプロセッサの命令については説明しません。詳しくは、「[もっと学ぶには](#)」(p.6) を参照してください。

この章には以下の項目が含まれています。

[ステートメントの文法](#)

[シンボルの文法](#)

[シンボルのスコープ](#)

[定数の文法](#)

[数式の文法](#)

[前方参照の文法](#)

[データのアラインメント](#)

ステートメントの文法

アセンブリ言語の文法は以下のどれかのタイプになります。

命令ステートメント

疑似命令ステートメント

マクロステートメント

ステートメントは 1000 文字を越える事はできません。ステートメントを複数の行に分けることも、一つ以上のステートメントを 1 行に含めることもできません。

ステートメントの文法は次のようになります。

例 2.1

ステートメントの文法

```
[label] operation [operands] [comment]
```

アセンブラのステートメントの構成要素を示します。

ラベル (label) : デフォルトではラベルはコロン (:) で終わり、どの桁から始める事もできます。もしこの規則に沿わないプログラムを移植するときは、『Labels must end with ':'』オプションをオフにしてください。このオプションをオフにすると、1 桁目から始まるシンボルとコロン (:) で終わるシンボルがラベルになります。

ラベルについての詳しい説明は、『[シンボルの文法](#)」(p. 8) を参照してください。

オペレーション (operation) : オペレーションとは次のものに対する名前です。

機械語命令 : あるプロセッサでどの機械語命令が使えるのかについては『[もっと学ぶには](#)」(p. 6) を参照してください。

マクロ呼び出し : マクロについての詳しい説明は、『[マクロの使い方の概要](#)」(p. 17) を参照してください。

アセンブラ疑似命令 : すべてのアセンブラ疑似命令のリストは、『[疑似命令の使い方の概要](#)」(p. 21) を参照してください。

命令、疑似命令、マクロ名の大文字小文字は区別されません。例えば、MOV、Mov、mov はすべて同じ命令です。

オペランド (operand) : オペランドはオペレーションが使用するデータを指定します。オペレーションの種類によって、いくつのオペランドが必要であるのかが決まります。オペランドはコンマ (,) で区切ります。

コメント (comment) : コメントは Metrowerks アセンブラが無視する文で、プログラムに説明を加えるのに有効です。Metrowerks アセンブラはセミコロン (;) と行の終わりで挟まれた文を無視します。存在するプログラムの移植を助けるために、Metrowerks アセンブラの特定のアセンブラは次の文をコメントとして扱います。

現在の全てのアセンブラは行の始めのアスタリスク (*) と行の終わりで挟まれた文を無視します。アスタリスクは行の最初の文字でなければならないことに注意してください。行の別の場所にある時は別の意味を持ちます。また、将来のアセンブラでは行の最初のアスタリスクを別の目的で使う可能性があることも覚えておいてください。

現在の全てのアセンブラでは、『Allow space in operand field』オプションをオフにすると、オペランドフィールド内の空白文字と行の終わりで挟まれた文を無視します。しかし、将来のアセンブラではこれは別の目的で使われる可能性があります。

シンボルの文法

シンボルは文字の組合せで、アドレスや数値定数、文字列定数、文字定数などの値を表します。シンボルにはラベルと記号定数の 2 種類があります。ラベルはアドレスを表すシンボルです。記号定数は値を表すシンボルで、.equ や .set 疑似命令で作ることができます。

シンボルの長さには制限がなく、次のものを含むことができます。

1 文字目は次のどれかでなければなりません。

- ・ 局所ラベルでなければ、a-z、A-Z、ピリオド (.)、疑問符 (?)、または下線 (_)。
- ・ 局所ラベルなら、アットマーク (@)。

残りの文字は、a-z、A-Z、数字 0-9、下線(_)、疑問符(?)、ドルマーク(\$)、ピリオド(.) のどれかです。

『Case sensitive identifiers』 オプションで、シンボルの大文字小文字を区別するか否かを選ぶことができます。このオプションがオンの場合、シンボルの大文字小文字は区別され、例えば SYM1、sym1、Sym1 は 3 つの異なるシンボルとなります。このオプションがオフの場合、シンボルの大文字小文字は同じと見なされ、SYM1、sym1、Sym1 は同じシンボルになります。このオプションはデフォルトではオンです。

プログラムカウンタを参照するには次の文字のどれかを使います。ピリオド(.)、ドルマーク(\$)、アスタリスク(*)。

シンボルのスコープ

普通、シンボルはファイル内のスコープを持ちます。シンボルはそれが定義されたファイルのどこからでもアクセスでき、またそのファイル内でしかアクセスできません。

ラベルは局所スコープを持つこともできます。局所ラベルには、非局所ラベルに重ならない部分においてアクセスすることができます。局所ラベルを作るにはその名前をアットマーク(@) で始めます。

記号定数はグローバルスコープを持つこともでき、外のファイルからそれをアクセスすることができます。グローバルな記号定数を作るには、.global または .public 疑似命令を使います。

この節では以下の内容について説明します。

[局所ラベル](#)

[グローバルな記号定数](#)

注意： 記号定数は局所スコープを持つことはできません。局所ラベルは普通のラベルのようにグローバルスコープを持つことはできません。

局所ラベル

アットマーク(@) 文字で始まる名前を持つラベルは局所的で、局所ラベルのスコープは非局所ラベルと重複しない部分まで拡張されます。(「[前方参照の文法](#)」(p. 15) で説明されている) 記号定数の前方参照はスコープを終わらせません。

マクロ定義で生成される行の局所ラベルは、特有の名前のスコープを持ちます。

マクロで展開される非局所ラベルは展開されない部分の局所スコープを終わらせることはありません。

マクロ内で定義される局所ラベルのスコープはマクロの外側に拡張されることはありません。

次の例は、マクロ内の局所ラベルのスコープについての分かりやすい例です。

例 2.2 マクロ中の局所ラベルのスコープ

```
MAKEPOS    .MACRO
    tst     #1, r0
    bra     @SKIP
    neg     r0
@SKIP:     ;Scope of this label is within the macro
.ENDM
START:
    mov.l   COUNT, r1
    cmp/eq  #1, r1
    bra     @SKIP
    MAKEPOS
@SKIP:     ;Scope of this label is START to END
           ;excluding lines arising from
           ;macro expansion
    add     #1, r0
END:      rts
```

この例でマクロ内で定義されている @SKIP ラベルは、プログラムの本体で定義されている @SKIP と衝突しません。

マクロの中では Metrowerks アセンブラは \@ シンボルを、局所ラベルや前方参照として使用可能でユニークな名前置き換えます。 \@ についての詳しい情報は、[「ユニークなラベルの生成」\(p. 19\)](#) を、記号定数の前方参照については、[「前方参照の文法」\(p. 15\)](#) を参照してください。

注意： 局所ラベルをエクスポートしたり、デバッグテーブルに含めたりすることはできません。

グローバルな記号定数

記号定数はグローバルスコープを持つこともできます。それは他のファイルからアクセスすることができます。記号定数がグローバルスコープを持つように宣言するには、`.global` 疑似命令を使います。

例 2.3 グローバルな記号定数の宣言

```
CONST      .set      256
           .global   CONST
```

他のファイルから記号定数にアクセスするには `.extern` 疑似命令を使用します。

例 2.4 グローバルな記号定数のインポート

```
.extern     CONST
ADDQ.L     CONST, r1
```

代わりに、宣言とインポートの両方に `.public` 疑似命令を使うこともできます。与えられた記号定数が既に定義されていたら、それはグローバルとして宣言されます。定義されていないときは、インポートされます。

C/C++ からグローバルな記号定数をインポートするとき、シンボル名が変わることがあります。ソースファイルを逆アセンブルして、シンボルテーブルを見てみましょう。例えば、以下のような大域変数を定義したソースファイルを例とします。

```
unsigned long  this_long = 0x12345678;
```

ソースファイルを逆アセンブルする場合、名前が変更されていることに気付くでしょう。この例では、シンボル名にアンダースコア文字 `'_'` が添付されています。

*** SYMBOL TABLE (.symtab) ***							
no	value	size	bind	type	other	shndx	name
13	0x00000000	0x00000004	GLOBAL	OBJECT	0x00	.data	_this_long

再配置可能なラベル

Metrowerks アセンブラは 32 ビットのフラットなメモリ空間を想定しています。以下の表現によって 32 ビットラベルの再配置を指示することができます。

注意： いくつかの表現については、すべてのアセンブラで使えるわけではありません。

表 2.1 再配置可能なラベルの表現

表現	意味
<i>label</i>	ラベルからセクションのベースアドレスまでのオフセット。分岐や呼び出しの PC 相対ターゲットでもある。32 ビットアドレス。
<i>label@l</i>	シンボルの低いほうの 16 ビットアドレス
<i>label@h</i>	シンボルの高いほうの 16 ビットアドレス。フルの 32 ビットアドレスにするには、 <i>label@l</i> とこのアドレスの OR をとる。
<i>label@ha</i>	アドレスの上位 16 ビット。フルの 32 ビットアドレスにするには、 <i>label@l</i> とこのアドレスの和をとる。
<i>label@sdx</i>	スモールデータ (<code>.sdata</code>) セクション中にあるラベル用に使用する。スモールデータセクションからラベルまでのオフセット。セクションを超えることはできない。
<i>label@got</i>	グローバルオフセットテーブル付きのチップでは、グローバルオフセットテーブルからラベルに対しての 32 ビットエントリへのオフセット。

定数の文法

Metrowerks アセンブラはいくつかの定数を認識します。

[整数定数](#)

[浮動小数点定数](#)

[文字定数](#)

それぞれの定数の文法はすべてのアセンブラで同一です。

整数定数

この表は整数定数の推奨される記法を示します。この記法はすべての Metrowerks アセンブラで使うことができます。

表 2.2 推奨される整数定数の記法

数値の形式	記法
10 進数	10 進数字の並び。例：12345678
16 進数	ドル記号 (\$) の後に 16 進文字を並べたもの。例：\$deadbeef
2 進数	パーセント記号 (%) の後に 2 進数字を並べたもの。例： %01010001

既存のコードの移植を助けるために、現在のアセンブラでは次の表に示す表記もサポートしています。しかし、将来にはこれらの表記が他の目的に使用されるかもしれません。

表 2.3 もう一つの整数定数の記法

数値の形式	記法
16 進数	0x の後に 16 進文字を並べたもの。例：0xdeadbeef
16 進数	0 の後に 16 進文字を並べたもの。例：0deadbeef さらに h で終わるもの。例：0deadbeefh
10 進数	d で終わる 10 進数字の並び。例：12345678d
2 進数	b で終わる 2 進数字の並び。例：01010001b

Metrowerks アセンブラは整数定数を格納したり操作したりするときに 32 ビット符号付き演算を使用することに注意してください。

浮動小数点定数

浮動小数点定数は 16 進数もしくは 10 進数の形式で指定することができます。10 進数形式の浮動小数点定数は小数点が指数のどちらかを含まなければなりません。1E-10 や 1.0 です。

浮動小数点定数は `.float` や `.double` のようなデータ生成擬似命令か、浮動小数点命令の中でのみ使うことができます。式の中では使うことができません。

文字定数

文字定数は引用符で囲まれていなければならない、文脈によっては最大 4 バイトになります。例を挙げると `'A'`、`'ABC'`、`'TEXT'` などです。

引用符 (') を文字定数に含めるには、2 つの引用符を使います。たとえば、`'IT'S'` です。文字定数にはさらに以下のエスケープシーケンスを含めることができます。

表 2.4 エスケープシーケンス

シーケンス	記述
<code>\b</code>	バックスペース
<code>\n</code>	ラインフィード (ASCII 文字コード 10)
<code>\r</code>	リターン (ASCII 文字コード 13)
<code>\t</code>	タブ
<code>\"</code>	二重引用符
<code>\\</code>	バックスラッシュ
<code>\nnn</code>	8 進数の <code>\nnn</code>

文字定数は演算時には 32 ビットにゼロ拡張されます。文字定数は整数定数を使うことのできるすべての場所で使用できます。

数式の文法

Metrowerks アセンブラは数式を 32 ビット符号付き演算で扱います。数値演算のオーバーフローはチェックされません。

異なるプロセッサに対する既存のアセンブラには共通の演算子のセットがないため、Metrowerks アセンブラは C 言語のそれに似た文法を使用しています。数式は括弧や結合法則などに C 言語の計算規則を使います。演算子も同じ物を使います。

すべての Metrowerks アセンブラは以下の表に挙げる演算子をサポートします。

表 2.5 単項演算子

演算子	説明
<code>+</code>	単項演算子のプラス
<code>-</code>	単項演算子のマイナス
<code>~</code>	単項演算子の 1 ビット補数

表 2.6 二項演算子

演算子	説明
+	加算
-	減算
*	乗算
/	除算
%	余り
	論理 OR
&&	論理 AND
	ビット OR
&	ビット AND
^	ビット XOR
<<	左シフト
>>	右シフト (高位ビットには 0 が入る)
==	等しい
!=	等しくない
<=	より小さいか等しい
>=	より大きい等しい
<	より小さい
>	より大きい

現在の Metrowerks アセンブラはすべて [表 2.7](#) に挙げる演算子も許可しています。しかし、将来のアセンブラでは他の目的のためにこれらの演算子が予約されるかもしれません。

表 2.7 代用できる演算子

演算子	説明
<>	等しくない
//	余り
!	論理 OR
!!	論理 XOR

演算子は次のような優先順位を持ちます。優先順位の高い方が先です。

1. 単項演算子の + - ~
2. * / %

3. 二項演算子の + -
4. << >>
5. < <= > >=
6. == !=
7. &
8. ^
9. |
- 10.&&
- 11.||

前方参照の文法

Metrowerks アセンブラでは前方参照を扱うことができます。前方参照とはシンボルがファイル中で定義される前に参照できることをいいます。アセンブラが、値のわからないシンボルを参照しているために解決できない数式に出会ったときに、その数式を保存し未解決の印をつけます。アセンブラがファイル全体を読み終わったときは、未解決の数式を再評価し、必要な場合すべてを解決するか、それ以上解決できなくなるまで繰り返し再評価します。アセンブラが数式を解決できない場合はエラーが発生します。

ただし、アセンブラはロケーションカウンタに影響を与える値を持つ数式を直接解決しなければなりません。

注意： もしアセンブラがロケーションカウンタについて自然な仮定を置くことができるならば、その数式を書くことができることに注意してください。例えば、68K プロセッサの前方分岐命令では 8、16、32 ビットのデフォルト値を指定することができます。

従って、[例 2.5](#) のようなコードを書くことができます。

例 2.5 正しい前方参照

```
.long      alloc_size
alloc_size .set      rec_size + 4
           ;a valid forward equate on next line
rec_size   .set      table_start-table_end
           ; ...
table_start:
           ; ...
table_end:
```

しかし、次の例のコードは正しくありません。アセンブラはロケーションカウンタの影響がわからないため、`.space` 擬似命令中の数式を直接解決することができません。

例 2.6 正しくない前方参照

```
                                ;invalid forward equate on next line
rec_size      .set      table_start-table_end
               .space    rec_size
               ; ...
table_start:
               ; ...
table_end:
```

データのアラインメント

デフォルトで、すべてのデータはデータサイズと対象プロセッサファミリに適した自然な境界に配置されます。[「option」\(p. 34\)](#) の項目で説明されている `.option` 擬似命令の `alignment` 引数でアラインメントをオフにすることができます。

アセンブラは `.debug` セクション内のデータを自動的に配置しません。詳しくは [「デバッグ擬似命令」\(p. 35\)](#) の `.debug` の項目を参照してください。

第 3 章 マクロの使い方

この章ではマクロの定義と使い方について説明します。

マクロの使い方の概要

Metrowerks アセンブラでは、どのターゲットプロセッサに対しても同じマクロ言語を使うことができます。このマクロ言語は、日立アセンブラの文法に少々拡張を加えたものとはほぼ同じであることを覚えておいてください。

この章では以下の内容について説明します。

[マクロの定義](#)

[マクロの呼び出し](#)

マクロの定義

この節ではマクロの定義の仕方について説明します。以下の内容について説明します。

[マクロ定義の文法](#)

[マクロ引数の使い方](#)

[引数の数の参照](#)

[ユニークなラベルの生成](#)

マクロ定義の文法

マクロ定義とは、マクロの名前、呼び出しの書式、展開されたときに処理されるアセンブラのステートメントを定義するようなアセンブラのステートメントの並びで、次のようになります。

例 3.1

マクロの定義

```
name:      .macro    [ param1, ] [ param2 ]. . .  
           ; macro body  
           .endm
```

name はマクロを呼び出すときに使うラベルです。必要であれば *param1*、*param2* のようなパラメータのリストを含めることもできます。これらはオペランドとしてマクロに渡され、マクロの本体で使われます。*macro body* はマクロ展開時に置き換えられるアセンブラのステートメントから構成されます。

マクロ定義は `.endm` で終わらなければなりません。`.endm` に到達する前にマクロの処理を止めたいとき (例えばマクロが条件アセンブルを含むとき) は `.mexit` を使ってください。

マクロ引数の使い方

引数は直接名前で参照することができます。整数を `r0` に移動し、`_final_setup` に分岐する `setup` マクロを参照してください。

例 3.2 setup マクロ

```
setup:      .macro name
             mov.l  #name, r0
             bsr    _final_setup
             .endm
```

次のようにして呼び出すことができます。

例 3.3 setup の呼び出し

```
          setup      'VECT'
```

これは次のように展開されます。

例 3.4 展開された setup

```
          MOV.L     #'VECT', R0
          BSR       _final_setup
```

マクロの本体で名前の付いた引数を参照するとき、`&&` をマクロ引数の前や後ろに付けることができます。これによって文字列にマクロ引数を埋め込むことができます。例えば、文字列 `E-50` をマクロ引数に付け加えて小さな浮動小数を作り出す `smallnum` マクロを参照してください。

例 3.5 smallnum マクロ

```
smallnum:   .macro      mantissa
             .float      mantissa&&E-50
             .endm
```

次のようにして呼び出すことができます。

例 3.6 smallnum の呼び出し

smallnum	10
----------	----

これは次のように展開されます。

例 3.7 展開された smallnum

.float	10E-50
--------	--------

ユニークなラベルの生成

\@ というシンボルをマクロ中で使うと、ユニークなラベルを生成することができます。マクロが呼び出されるたびに、アセンブラは ??nnnn という形式のユニークなシンボルを生成します。マクロが呼ばれるたびに、??0001 や ??0002 のようになります。

@ で始まるラベルは局所ラベルと呼ばれますが、これは展開されたマクロの中だけに制限されたスコープを持ちます。詳しくは「[シンボルのスコープ](#)」(p. 9) を参照してください。

ユニークなラベルとシンボル (\@ を使うもの) は、通常のラベルやシンボルと同様にプログラム中で参照することができます。 \@ はマクロが呼び出されるたびに 1 増やされたユニークな文字列で置き換えられます。

例 3.8 ユニークなラベルのマクロ

my_mac:	.macro
	fred\@ = my_count
my_count	.set my_count + 1
	add fred\@, r1
	bra label\@
	add r1, r2
label\@:	
	nop
	.endm

もし例 3.8 のマクロが 2 度呼ばれたら (my_count が 0 に初期化されたとして)、例 3.9 のようにアセンブルされます。

例 3.9 ユニークなラベルのアセンブラの出力

0x00000000:	foo??0000	=	my_count
0x00000001:	my_count	.set	my_count + 1
0x00000008:		add	fred??0000, r1
0x0000000c:		bra	label??0000
0x00000010:		add	r1, r2
0x00000014:	label??0000		

```
0x00000014:      nop
0x00000000:      my_macro
0x00000000:      fred??0001 =      my_count
0x00000001:      my_count      .set      my_count + 1
0x00000008:      add      fred??0001, r1
0x0000000c:      bra      label??0001
0x00000010:      add      r1, r2
0x00000014:      label??0001
0x00000014:      nop
0x00000000:
```

引数の数の参照

マクロに渡された空ではない引数の数を参照するには、特別なシンボル `narg` を使ってください。これはマクロ展開の中でのみ使うことができます。

マクロの呼び出し

マクロを呼び出すにはその名前だけをアセンブラのプログラム中に書いてください。

マクロを呼び出すときは引数をコンマで区切ってください。コンマを含む引数を渡すときは `<>` で括ってください。 `mov.l` 命令を拡張する `moveit` マクロ ([例 3.10](#) で定義されています) を呼び出すステートメントの例を参照してください。

例 3.10 コンマを含む引数とともに `moveit` を呼び出す

```
moveit.l      <@(1020,pc)>, r15
```

第 4 章 疑似命令の使い方

この章では Metrowerks アセンブラで利用できる疑似命令について説明しています。

疑似命令の使い方の概要

この章では Metrowerks アセンブラの疑似命令の使い方について解説します。いくつかの疑似命令はすべてのアセンブラで利用できるわけではありません。疑似命令の説明ではどのアセンブラがその疑似命令をサポートしているかも示しています。

デフォルトでは、疑似命令はピリオド (.) で始まらなければなりません。しかし、アセンブラの設定パネルで『Directives begin with '!'』オプションをオフにすれば、ピリオドを外すことができます。

この章では、次のように分類された疑似命令を紹介します。

[マクロ疑似命令](#)

[条件プロセッサ疑似命令](#)

[セクション制御疑似命令](#)

[スコープ制御疑似命令](#)

[シンボル定義疑似命令](#)

[データ定義疑似命令](#)

[アセンブラ制御疑似命令](#)

[デバッグ疑似命令](#)

マクロ疑似命令

次の疑似命令によってマクロを作ることができます。詳しくは、[「マクロの使い方の概要」](#) (p. 17) 以下を読んでください。

[macro](#) : マクロ定義を開始します。

[endm](#) : マクロ定義を終了します。

[mexit](#) : endm に到達する前にマクロ展開を終了します。

macro

```
label .macro [ param1, param2 . . . ]
```

与えられた引数の、label という名前のマクロの定義を始めます。

```
endm  
    .endm
```

マクロ定義を終了します。

```
mexit  
    .mexit
```

.endm に到達する前にマクロ展開を終わらせます。

条件プロセッサ疑似命令

疑似命令は条件アセンブルブロックを作ります。`.ifdef` と `.endif` でコードを囲むと、コンパイルするか否かをコントロールできます。コードが少しだけ異なるようなターゲットをいくつかビルドしたいときに便利です。

条件疑似命令は完全なブロックを作るように、組みにして使わなければなりません。Metrowerks アセンブラはブロックの生成を簡単にするため、文字列の一致やシンボルが定義されているかなどを検査するための `.if` の派生版を持っています。その疑似命令を示します。

[`if`](#) : 条件アセンブルを開始します。ブール型の式を使うことができます。

[`ifdef`](#) : 条件アセンブルを開始します。シンボルが定義されているかどうかを検査します。

[`ifndef`](#) : 条件アセンブルを開始します。シンボルが定義されていないかどうかを検査します。

[`ifc`](#) : 条件アセンブルを開始します。2 つの文字列が一致するかどうかを検査します。

[`ifnc`](#) : 条件アセンブルを開始します。2 つの文字列が不一致かどうかを検査します。

[`endif`](#) : 条件アセンブルを終了させます。

[`elseif`](#) : 最初の条件が偽だったときに、別の検査を行います。

[`elif`](#) : 最初の条件が偽だったときに、別の検査を行います。`.elseif` と同じ機能です。

[`else`](#) : どの検査も偽だったときに実行されるステートメントを記述します。

[`ifeq`](#), [`ifne`](#), [`iflt`](#), [`ifle`](#), [`ifgt`](#), [`ifge`](#) : 下位互換性のための、条件アセンブルステートメントです。

```
if  
    .if bool-expr
```

条件アセンブルの開始を表わします。`bool-expr` はブール式です。`bool-expr` が真の場合、アセンブラは `.if` 疑似命令に属するステートメントを処理します。`bool-expr` が偽の場合、アセンブラは `.if` 疑似命令に属するステートメントを無視します。

各 `.if` 疑似命令は対応する `.endif` 疑似命令を持つ必要があります。

注意： ブール式とは数式の特別な形です。0 に評価されたブール式は偽、0 以外に評価されたブール式は真と解釈されます。詳しくは「[数式の文法](#)」(p. 13) を参照してください。

ifdef

```
.ifdef symbol
```

symbol が定義されているシンボルの名前なら、条件アセンブルの開始を表わします。名前が以前から定義されている場合、アセンブラは `.ifdef` 疑似命令に属するステートメントを処理します。名前が以前に定義されていない場合、アセンブラは `.ifdef` 疑似命令に属するステートメントを無視します。

各 `.ifdef` 疑似命令は対応する `.endif` 疑似命令を持つ必要があります。

ifndef

```
.ifndef symbol
```

symbol が定義されていないシンボルの名前なら、条件アセンブルの開始を表わします。名前が以前から定義されていない場合、アセンブラは `.ifndef` 疑似命令に属するステートメントを処理します。名前が以前に定義されている場合、アセンブラは `.ifndef` 疑似命令に属するステートメントを無視します。

各 `.ifndef` 疑似命令は対応する `.endif` 疑似命令を持つ必要があります。

ifc

```
.ifc string1, string2
```

string1 と *string2* が等しい文字列同士なら、条件アセンブルの開始を表わします。比較は大文字小文字を区別します。文字列が等しい場合、アセンブラは `.ifc` 疑似命令に属するステートメントを処理します。文字列が等しくない場合、アセンブラは `.ifc` 疑似命令に属するステートメントを無視します。

各 `.ifc` 疑似命令は対応する `.endif` 疑似命令を持つ必要があります。

ifnc

```
.ifnc string1, string2
```

string1 と *string2* が等しい文字列同士でないなら、条件アセンブルの開始を表わします。比較は大小文字を区別します。文字列が等しくない場合、アセンブラは `.ifnc` 疑似命令に属するステートメントを処理します。文字列が等しい場合、アセンブラは `.ifnc` 疑似命令に属するステートメントを無視します。

各 `.ifnc` 疑似命令は対応する `.endif` 疑似命令を持つ必要があります。

endif

`.endif`

条件アセンブルの終わりを示します。どのタイプの `.if` 疑似命令も対応する `.endif` 疑似命令を持たなければなりません。

elseif

`.elseif bool-expr`

`.if` 疑似命令と、前にある `.elseif` 疑似命令のブール式が偽で、この `.elseif` ステートメントの `bool-expr` が真だったときに処理されるべき条件アセンブルステートメントの開始を示します。`.if` 疑似命令は `.elseif` 疑似命令を必要としません。

`.if` 疑似命令のブール式が偽の場合、アセンブラは `.if` 疑似命令に属するステートメントを無視し、最初の `.elseif` 疑似命令のブール式を評価します。ブール式が真の場合、アセンブラは `.elseif` ステートメントに属するステートメントを処理します。そうではない場合、次の `.elseif` ステートメントのブール式を評価します。アセンブラは真と評価されるブール式に当たるまで、続く `.elseif` ステートメントのブール式を評価し続けます。`.if-endif` ブロックのどの `.elseif` 疑似命令も真と評価されるブール式を持たない場合、アセンブラは `.else` ステートメントのブロックに属するステートメントが存在するならば、それを評価します。

elif

`.elif bool-expr`

[elseif](#) と同じです。

else

`.else`

`.if` 疑似命令と、それに属する `.elseif` 疑似命令のブール式が偽だったときに処理されるべき条件アセンブルステートメントの開始を示します。`.if` 疑似命令は `.else` 疑似命令を必要としません。

ifeq, ifne, iflt, ifle, ifgt, ifge

<code>.ifeq</code>	; 等しい場合
<code>.ifne</code>	; 等しくない場合
<code>.iflt</code>	; 小さい場合
<code>.ifle</code>	; 小さいか等しい場合
<code>.ifgt</code>	; 大きい場合
<code>.ifge</code>	; 大きいか等しい場合

他のアセンブラとの互換性のために、これらの疑似命令もサポートされています。

セクション制御疑似命令

これらの疑似命令はアセンブラファイルの異なるセクションを示します。現在のすべての Metrowerks アセンブラで全部が利用可能ですが、将来のアセンブラではすべてはサポートされないかもしれません。

[text](#) : 実行可能コードセクションを指定します。

[data](#) : 初期化された読み書き用のデータセクションを指定します。

[rodata](#) : 初期化された読み込み専用のデータセクションを指定します。

[bss](#) : 初期化されない読み書き用のデータセクションを指定します。

[sdata](#) : 初期化された読み書き用の小さなデータセクションを指定します。

[sdata2](#) : 初期化された読み込み専用の小さなデータセクションを指定します。

[sbss](#) : 初期化されない読み書き用の小さなデータセクションを指定します。

[debug](#) : デバッグセクションを指定します。

[previous](#) : 前のセクションに戻します。

[offset](#) : レコードを定義します。

[section](#) : あらゆるタイプのセクションを指定します。

text

```
.text
```

実行可能コードセクションを指定します。これはファイル中の実際のコードの前になければなりません。

data

```
.data
```

初期化された読み書き用のデータセクションを指定します。

rodata

```
.rodata
```

初期化された読み込み専用のデータセクションを指定します。

bss

```
.bss
```

初期化されない読み書き用のデータセクションを指定します。

sdata

```
.sdata
```

初期化された読み書き用の小さなデータセクションを指定します。

sdata2

```
.sdata2
```

初期化された読み込み専用の小さなデータセクションを指定します。

sbss

```
.sbss
```

初期化されない読み書き用の小さなデータセクションを指定します。

debug

```
.debug
```

デバッグセクションを指定します。デバグを有効にしている場合、アセンブラは自動的にプロジェクトに対してデバッグ情報を生成します。しかし、デバッグセクション中で、デバグにより詳しい情報を与える特殊な疑似命令を使うことができます。デバッグ用疑似命令に関する詳しい情報は、[「デバッグ疑似命令」\(p. 35\)](#) を参照してください。

previous

```
.previous
```

前のセクションに戻します。これは現在のセクションと前のセクションを切替えます。

offset

```
.offset [expr]
```

レコードを定義します。省略可能な引数 `expr` はロケーションカウンタの初期値を指定します。レコード定義は次のセクションの開始の前までとなります。レコード中では次の疑似命令のみが使えます。

<code>.equ</code>	<code>.set</code>	<code>.textequ</code>
<code>.align</code>	<code>.org</code>	<code>.space</code>
<code>.byte</code>	<code>.short</code>	<code>.long</code>
<code>.space</code>	<code>.ascii</code>	<code>.asciz</code>
<code>.float</code>	<code>.double</code>	

データ定義疑似命令 (`.byte` や `.short` など) は記憶領域を確保しません。ただロケーションカウンタを更新するだけです。

レコード定義の例を挙げます。

例 4.1 offset 疑似命令を用いたレコード定義

```
                .offset  
top:            .short  0  
left:           .short  0
```

```
bottom:      .short    0
right:       .short    0
rectSize     .equ      *
```

section

```
.section name [,alignment], [type], [flags]
```

name という名前で *type* というタイプのセクションを指定します。この一般的な形式は任意の再配置可能なセクションを生成するために使います。これには絶対アドレスにロードされるセクションを含みます。`.section` は引数をとります。*name* 引数のみが必須であることに注意してください。

name はセクションの名前です。シンボルが使用できます。

type と *flags* は ELF セクションのタイプとフラグになり共に数値です。これらのフィールドのデフォルトはタイプとフラグのコードセクションです。次の例はセクション名 `vector` という 4 バイトアラインメントのセクションを定義しています。

```
.section vector,4
```

ELF セクションの *type* を[表 4.1](#) に示し、ELF セクションの *flag* を[表 4.2](#) に示します。

表 4.1 ELF セクションの Type

Type	Name
0	NULL
1	PROGBITS
2	SYMTAB
3	STRTAB
4	RELA
5	HASH
6	DYNAMIC
7	NOTE
8	NOBITS
9	REL
10	SHLIB
11	DYNSYM

表 4.2 ELF セクションの Flag

Flag	Name
0x00000001	WRITE
0x00000002	ALLOC
0x00000004	EXECINSTR
0xF0000000	MASKPROC
0x10000000	GPREL

スコープ制御疑似命令

Metrowerks アセンブラは、それが定義されたファイルの外でも記号定数を使用できるようにする疑似命令を提供します。記号定数とは `.set` か `.equ` で宣言されたシンボルで、[「シンボル定義疑似命令」\(p. 29\)](#) で解説されています。この疑似命令を示します。

[global](#) : 記号定数のエクスポートを宣言します。

[extern](#) : 記号定数のインポートを宣言します。

[public](#) : 記号定数がパブリックであると宣言します。

global

```
.global equate [,equate]...
```

列挙された記号定数のエクスポート、つまり他のファイルから参照可能であることを宣言します。記号定数とは `.set` か `.equ` で宣言されたシンボルで、[「シンボル定義疑似命令」\(p. 29\)](#) で解説しています。

別のファイルのシンボルを参照するには `.extern` か `.public` 疑似命令を使ってください。
ラベルをエクスポートすることはできません。

extern

```
.extern equate [,equate]...
```

列挙された記号定数のインポート、つまり現ファイルで参照できるが、別ファイルで定義されていることを宣言します。記号定数とは `.set` か `.equ` で宣言されたシンボルで、[「シンボル定義疑似命令」\(p. 29\)](#) で解説しています。

別のファイルへシンボルをエクスポートするには `.global` か `.public` 疑似命令を使ってください。

ラベルをインポートすることはできません。

```
public  
    .public equate [,equate]...
```

列挙された記号定数がパブリックであると宣言します。記号定数が既に定義されている場合、アセンブラはそれをエクスポート、つまり他のファイルから参照可能にします。記号定数がまだ定義されていない場合、アセンブラはそれをインポート、つまり現ファイルで参照可能だが別ファイルで定義されているものとしします。

記号定数とは `.set` か `.equ` で宣言されたシンボルで、[「シンボル定義疑似命令」\(p. 29\)](#) で解説しています。ラベルをインポートすることはできません。

シンボル定義疑似命令

次の疑似命令は記号定数を生成します。

[set](#) : 一時的に値をシンボルに代入します。

[等号 \(=\)](#) : 一時的に値をシンボルに代入します。他のアセンブラとの互換性のために用意されています。

[equ](#) : 恒久的に値をシンボルに代入します。

[textequ](#) : 任意のテキストに置換されるシンボルを定義します。

```
set  
    symbol .set expr
```

一時的に値 `expr` をシンボル `symbol` に代入します。`symbol` の値は後で変更することができます。シンボル `symbol` は行のラベルフィールドに、値 `expr` はオペランドフィールドに置かれます。

```
等号 (=)  
    symbol = expr
```

一時的に値 `expr` をシンボル `symbol` に代入します。`symbol` の値は後で変更することができます。シンボル `symbol` は行のラベルフィールドに、値 `expr` はオペランドフィールドに置かれます。

この疑似命令は `.set` と等価で、他のアセンブラとの互換性のためにのみ用意されています。将来のアセンブラではこの疑似命令はサポートされないかもしれません。

```
equ  
    symbol .equ expr
```

恒久的に値 `expr` をシンボル `symbol` に代入します。`symbol` の値は後で変更することができません。シンボル `symbol` は行のラベルフィールドに、値 `expr` はオペランドフィールドに置かれます。

textequ

```
symbol .textequ "string"
```

シンボル *symbol* が任意のテキスト *string* で置換されるように定義します。この疑似命令は、機械語命令、疑似命令、オペランドに新しい名前を付けることができるようにすることで、既存のコードを移植するのを助けます。

symbol を使うときはいつも、アセンブラはプログラム行に他の処理をほどこす前に、*string* で置き換えます。有益な例をいくつか挙げます。

例 4.2 textequ の例

```
dc.b      .textequ      ".byte"  
endc      .textequ      ".endif"
```

データ定義疑似命令

Metrowerks アセンブラはデータを初期化する疑似命令を持っています。これらは 3 つの項目に分かれます。

整数型の宣言

- [byte](#) : 初期化されたバイトブロックを宣言します。
- [short](#) : 初期化された 16 ビット整数のブロックを宣言します。
- [long](#) : 32 ビット整数のブロックを宣言します。
- [space](#) : 0 で初期化されたバイトブロックを宣言します。
- [fill](#) : 0 で初期化されたバイトブロックを宣言します。

文字列型の宣言

- [ascii](#) : 文字列のための記憶領域ブロックを宣言します。
- [asciz](#) : 文字列のための 0d で終わる記憶領域ブロックを宣言します。

浮動小数点型の宣言

- [float](#) : 初期化された 32 ビット浮動小数点数のブロックを宣言します。
- [double](#) : 初期化された 64 ビット浮動小数点数のブロックを宣言します。

整数型の宣言

この疑似命令は整数データのブロックを初期化します。

[byte](#) : 初期化されたバイトブロックを宣言します。

[short](#) : 初期化された 16 ビット整数のブロックを宣言します。

[long](#) : 32 ビット整数のブロックを宣言します。

[space](#) : 0 で初期化されたバイトブロックを宣言します。

[fill](#) : 0 で初期化されたバイトブロックを宣言します。

byte

```
[label] .byte expr[,expr]...
```

label という名の初期化されたバイトブロックを宣言します。アセンブラはそれぞれの式 *expr* に対して 8 ビットを 1 バイト確保します。それぞれの式は与えられたサイズに収まらなければなりません。

short

```
[label] .short expr[,expr]...
```

label という名の初期化された 16 ビット整数のブロックを宣言します。アセンブラはそれぞれの式 *expr* に対して 16 ビット確保します。それぞれの式は与えられたサイズに収まらなければなりません。

long

```
[label] .long expr[,expr]...
```

label という名の 32 ビット整数のブロックを宣言します。アセンブラはそれぞれの式 *expr* に対して 32 ビット確保します。それぞれの式は与えられたサイズに収まらなければなりません。

space

```
[label] .space expr
```

label という名の 0 で初期化されたバイトブロックを宣言します。アセンブラはブロックを *expr* バイト確保し、各バイトを 0 で初期化します。

fill

```
[label] .fill expr
```

label という名の 0 で初期化されたバイトブロックを宣言します。アセンブラはブロックを *expr* バイト確保し、各バイトを 0 で初期化します。

文字列型の宣言

この疑似命令は文字データのブロックを初期化します。

[ascii](#) : 文字列のための記憶領域ブロックを宣言します。

[asciz](#) : 文字列のための 0d で終わる記憶領域ブロックを宣言します。

文字列は次のエスケープシーケンスも含むことができることに注意してください。

表 4.3 エスケープシーケンス

シーケンス	説明
<code>\b</code>	バックスペース
<code>\n</code>	ラインフィード (ASCII 文字コード 10)
<code>\r</code>	リターン (ASCII 文字コード 13)
<code>\t</code>	タブ
<code>\"</code>	二重引用符
<code>\\</code>	バックスラッシュ
<code>\nnn</code>	8 進数の <code>\nnn</code>

ascii

```
[label] .ascii "string"
```

文字列のための記憶領域ブロックを宣言します。アセンブラは文字列中の各文字に対して 8 ビット 1 バイトを割り当てます。

asciz

```
[label] .asciz "string"
```

文字列のための 0d で終わる記憶領域ブロックを宣言します。アセンブラは文字列中の各文字に対して 8 ビット 1 バイトを割り当てた後、その終わりに 1 ブロック余計に割り当て、それを 0 で初期化します。

浮動小数点型の宣言

この疑似命令は浮動小数点データのブロックを初期化します。

[float](#) : 初期化された 32 ビット浮動小数点数のブロックを宣言します。

[double](#) : 初期化された 64 ビット浮動小数点数のブロックを宣言します。

float

```
[label] .float value[,value]...
```

`label` という名の初期化された 32 ビット浮動小数点数のブロックを宣言します。アセンブラはそれぞれの値 `value` に対して 32 ビット確保します。それぞれの値は与えられたサイズに収まらなければなりません。

double

```
[label] .double value[,value]...
```

`label` という名の初期化された 64 ビット浮動小数点数のブロックを宣言します。アセンブラはそれぞれの値 `value` に対して 64 ビット確保します。それぞれの値は与えられたサイズに収まらなければなりません。

アセンブラ制御疑似命令

この疑似命令はアセンブラがどう動作するかを制御します。

[align](#) : ロケーションカウンタを式の次の倍数にそろえます。

[endian](#) : 対象プロセッサのバイト順序を指定します。

[error](#) : エラーメッセージを表示します。

[include](#) : アセンブラに他のファイルへ入力を切替えさせます。

[pragma](#) : ある種のコードの生成を有効にしたり、無効にしたりします。

[org](#) : ロケーションカウンタを変更します。

[option](#) : いくつかのアセンブラオプションをセットします。

align

```
.align expr
```

ロケーションカウンタを式 *expr* の次の倍数にそろえます。式 *expr* は 2 の累乗、2, 4, 8, 16, 32 でなければなりません。

endian

```
.endian big | little
```

対象プロセッサのバイト順序を指定します。この疑似命令は MIPS や PowerPC のような、バイト順序が変更できるプロセッサでのみ使用できます。

error

```
.error "error"
```

エラーメッセージ *error* を CodeWarrior IDE の「エラーと警告」ウインドウに表示します。

include

```
.include filename
```

アセンブラに他のファイルへ入力を切替えさせます。アセンブラは、ファイルの終わりに到達するまで指定されたファイルから入力を行ないます。そして `.include` 疑似命令の次のアセンブラステートメント行から入力を続けます。

filename で指定されたファイルは他のファイルへの `.include` 疑似命令を持つことができます。

pragma

```
.pragma pragma-type setting
```

アセンブラに指定したプラグマの設定でアセンブルすることを伝えます。pragma ステートメントの詳細については『C Compilers Reference』をご覧ください。

org

`.org expr`

ロケーションカウンタを変更します。続くアセンブラステートメントのアドレスをロケーションカウンタの新しい値から始めます。`expr` の値は現在のロケーションカウンタの値より大きくなければなりません。

option

`.option keyword setting`

以下で説明するアセンブラオプションをセットします。`reset` を指定するとオプションを前の設定にセットします。`reset` を 2 回使うとオプションを現在の値の前の設定に再設定します。

表 4.4 オプションキーワード

キーワード	動作
<code>alignment off on reset</code>	データが自然な境界にそろえられるかどうかを制御します。これは設定パネルのどのオプションとも対応しません。
<code>branchsize 8 16 32</code>	前方分岐命令のディスプレースメントのサイズを指定します。これは x86 と 68K アセンブラでのみ使用できますが、設定パネルのどのオプションにも一致しません。
<code>colon off on reset</code>	ラベルがコロン (:) で終わらなければならないかどうかを指定します。 <code>on</code> の場合、ラベルにはコロンが必要です。 <code>off</code> の場合、ラベルが 1 桁目から始まる場合、コロンは不要です。これは「 Labels must end with ':' 」(p. 38) で説明されている『Labels must end with ':'』オプションに対応します。
<code>space off on reset</code>	オペランドフィールドに空白が必要かどうかを指定します。 <code>on</code> の場合、オペランドフィールドは空白を含むことができます。 <code>off</code> の場合、オペランドフィールド中の空白はコメントの開始を意味します。これは「 Allow space in operand field 」(p. 38) で説明されている『Allow space in operand field』オプションに対応します。
<code>period off on reset</code>	擬似命令の名前にピリオド (.) が必要かどうかを指定します。 <code>on</code> の場合、各擬似命令はピリオドで始まるなければなりません。 <code>off</code> の場合、擬似命令はピリオドで始まる必要はありません。これは「 Directives begin with '.' 」(p. 38) で説明されている『Directives begin with '.'』オプションに対応します。

キーワード	動作
<code>case off on reset</code>	識別子の大文字小文字を区別するか否かを指定します。 on の場合、識別子の大文字小文字は区別されます。 off の場合、識別子の大文字小文字は区別されません。 これは「 Case sensitive identifiers 」(p. 38) で説明されている『Case sensitive identifiers』オプションに対応します。
<code>no_at_macros off on</code>	on の場合、\$AT マクロを許可しません。off の場合、\$AT マクロを使うと警告を出します。

アセンブラがジャンプや分岐命令の後に NOP 命令 (何もしない命令) を挿入するのをやめさせ、別の命令を挿入することもできます。そのためには、スタンドアローンアセンブラで `.option reorder off` を定義してください。

スタンドアローンアセンブラはデフォルトでは NOP 命令を挿入します。しかし、インラインアセンブラは NOP 命令を挿入しません。

デバッグ疑似命令

これらの疑似命令はアセンブラファイルの `.debug` セクション内でのみ使うことができます。デバッガを有効にすると、アセンブラは自動的にプロジェクトのデバッグ情報を生成します。しかし、より詳しい情報をデバッガに与えるために、デバッグセクション内でこれらの疑似命令を使うことができます。

[file](#) : デバッグ情報を指定した出力ファイルに書き出します。

[function](#) : サブルーチンの情報を指定します。

[line](#) : 次のコードの行番号の絶対値を指定します。

[size](#) : シンボルの長さを指定します。

[type](#) : シンボルが関数かオブジェクトが指定します。

file

```
.file "filename"
```

デバッグ情報を *filename* というファイルに書き出します。このオプションを使わない場合、デバッグ情報はプロジェクトファイルに書き出されます。

function

```
.function "func", label, length
```

label で始まるサブルーチン *func* が *length* バイトであることを指定します。

line

`.line number`

コードやデータを生成する現在のソースファイルの行番号の絶対値を指定します。ファイルの最初の行の番号は 1 です。

size

`.size symbol, expr`

symbol が *expr* バイトであることを指定します。

type

`.type symbol, type`

symbol のタイプが *type* であることを指定します。タイプは `@function` か `@object` のどちらかです。

第 5 章 SH アセンブラの設定

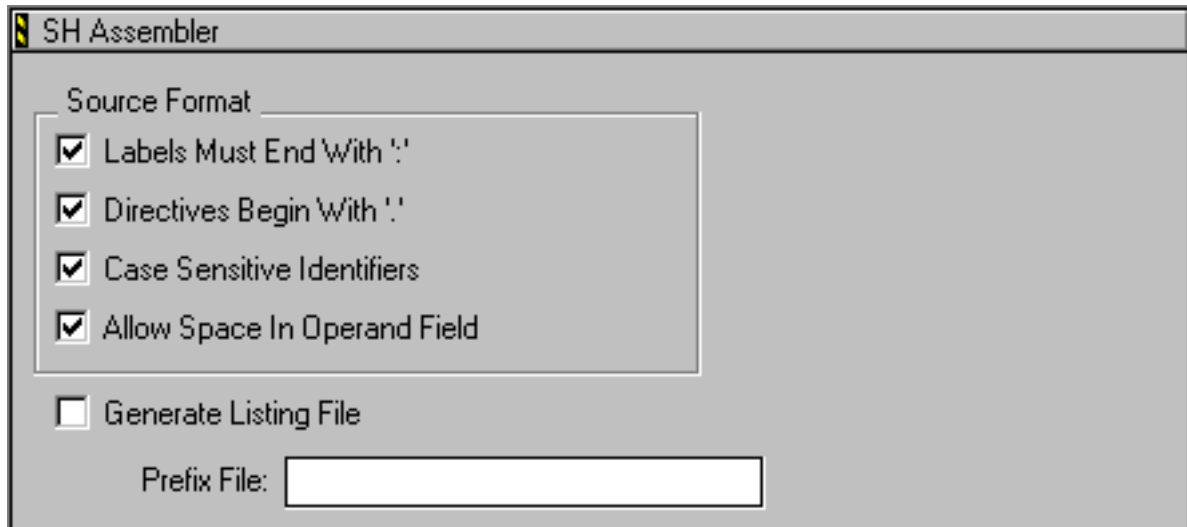
この章では Metrowerks アセンブラの設定可能なオプションについて説明します。

アセンブラの設定の概要

対象となるそれぞれのプロセッサファミリに対応して、異なるアセンブラが利用可能です。各アセンブラは設定パネルを通じて操作可能なオプションをいくつかもっています。アセンブラの設定を変更するには、編集メニューの『プロジェクト設定』を選びます。次に表示されるダイアログで、設定パネルを見たいアセンブラの名前を選んでください。

設定パネルはすべて [図 5.1](#) に示すものにとっても似ています。これは SH アセンブラのパネルです。ソースフォーマットの項目はすべてに共通しています。

図 5.1 SH アセンブラのアセンブラ設定パネル



アセンブラの設定パネル

設定可能なオプションは次の通りです。

[Labels must end with ':'](#)

[Directives begin with ':'](#)

[Case sensitive identifiers](#)

[Allow space in operand field](#)

[Generate listing file](#)

[Prefix file](#)

Labels must end with ':'

『Labels must end with ':'』 オプションはラベルが ':' で終わらなければならないかどうかを選択します。このオプションがオンなら、ラベルはコロン (:) で終わらなければなりません、どの桁から始めることもできます。このオプションがオフの場合、1 桁目から始まるシンボルとコロン (:) で終わるシンボルはラベルとなります。このオプションはこの約束に沿わない既存のコードを移植するときに特に便利です。ラベルについて詳しくは「[シンボルの文法](#)」(p. 8) を参照してください。

このオプションのデフォルトはオンです。これは「[option](#)」(p. 34) で説明されている、`.option` 疑似命令の `colon` 引数に対応します。

Directives begin with '.'

『Directives begin with '.'』 オプションは各疑似命令の名前の最初にピリオドを置かなければならないかどうかを選択します。このオプションがオンの場合、疑似命令はピリオドで始めなければなりません。このオプションがオフの場合、ピリオドを除くことができます。疑似命令についての詳しい情報は、「[疑似命令の使い方の概要](#)」(p. 21) を参照してください。

このオプションのデフォルトはオンです。これは「[option](#)」(p. 34) で説明されている、`.option` 疑似命令の `period` 引数に対応します。

Case sensitive identifiers

『Case sensitive identifiers』 オプションはシンボルの大文字小文字を区別するかどうかを選択します。このオプションがオンの場合、シンボルの大文字小文字は区別されるので、例えば `SYM1`, `sym1`, `Sym1` は 3 つの異なるシンボルとなります。このオプションがオフの場合、シンボルの大文字小文字は無視されるので、`SYM1`, `sym1`, `Sym1` は同じシンボルとなります。シンボルについての詳しい情報は、「[シンボルの文法](#)」(p. 8) を参照してください。

このオプションの設定の如何に関わらず、命令や疑似命令、マクロでは常に大文字小文字は区別されないことに注意してください。

このオプションのデフォルトはオンです。これは「[option](#)」(p. 34) で説明されている、`.option` 疑似命令の `case` 引数に対応します。

Allow space in operand field

『Allow space in operand field』 オプションはオペランドフィールド内のスペースでコメントを始めるかどうかを選択します。このオプションがオンの場合、オペランドフィールド内のスペースは許されます。このオプションがオフの場合、オペランドフィールド内のスペースと行の終わりで挟まれた文は無視されます。コメントについての詳しい情報は、「[ステータメントの文法](#)」(p. 7) を参照してください。

このオプションのデフォルトはオンです。これは「[option](#)」(p. 34) で説明されている、`.option` 疑似命令の `space` 引数に対応します。

Generate listing file

『Generate listing file』オプションはソースコードとアセンブラが出力した、機械語コードが比較できるテキストファイルを生成します。このオプションがオンの場合、ソースファイルの名前に `.list` をつけ加えたリスティングファイルを生成します。例えば、`test.asm` は `test.asm.list` になります。オプションがオフの場合、リスティングファイルは生成されません。

このオプションのデフォルトはオンです。

Prefix file

『Prefix file』フィールドはプロジェクト内のアセンブラファイルの前にアセンブラが処理するファイルを指定します。これはアセンブラファイルに `.include` 疑似命令を置くことと同じです。

このフィールドはデフォルトで空欄になっています。

索引

記号

= 29

@ 10, 19

A

align 33

alignment 34

Allow space in operand field 8, 34, 38

ascii 32

asciz 32

Assembler Control Directives

 pragma 33

B

branchsize 34

bss 25

byte 31

C

case 35

Case sensitive identifiers 9, 35, 38

colon 34

Conditional Directives

 elif 24

D

data 25

debug 26

Directives begin with '.' 21, 34, 38

double 32

E

ELF 27

elif conditional directive 24

else 24

elseif 24

endian 33

endif 24

endm 22

equ 29

error 33

extern 10, 28

F

file 35

fill 31

float 32

function 35

G

Generate listing file 39

global 9, 10, 28

I

if 22

ifc 23

ifdef 23

ifeq 24

ifge 24

ifgt 24

ifle 24

iflt 24

ifnc 23

ifndef 23

ifne 24

include 33

L

Labels must end with ':' 8, 34, 38

line 36

long 31

M

macro 21

macro body 17

mexit 22

N

name 17

no_at_macros 35

O

offset 26
option 34
org 34

P

period 34
pragma assembler control directive 33
Prefix file 39
previous 26
public 9, 11, 29

R

rodata 25

S

sbss 26
sdata 25
sdata2 26
section 27
set 29
short 31
SH アセンブラ 37
size 36
space 31, 34

T

text 25
textequ 30
type 36

ア行

アセンブラ制御疑似命令 33 ~ 35
 align 33
 endian 33
 error 33
 include 33
 option 34
 org 34
アットマ - ク (@) 9
アラインメント 16
エスケ - プシ - ケンス 13, 32
オプション

Allow space in operand field 34, 38, 8
Case sensitive identifiers 35, 38, 9
Directives begin with '.' 21, 34, 38
Generate listing file 39
Labels must end with ':' 34, 38, 8
Prefix file 39

オプションキ - ワ - ド

alignment キ - ワ - ド 34
branchsize キ - ワ - ド 34
case キ - ワ - ド 35
colon キ - ワ - ド 34
no_at_macros キ - ワ - ド 35
period キ - ワ - ド 34
space キ - ワ - ド 34

オペランド 8

オペレ - ション 8

カ行

記号定数 8
 グロ - バルな 9
局所ラベル 9
グロ - バルな記号定数 9
コメント 8

サ行

再配置可能なラベル 11
シンボル 8
 のスコ - プ 9 ~ 11
 の文法 8 ~ 9
シンボル定義疑似命令 29 ~ 30
 equ 29
 set 29
 textequ 30
 等号 (=) 29
条件疑似命令 22 ~ 24
 else 24
 elseif 24
 endif 24
 if 22
 ifc 23
 ifdef 23
 ifeq 24
 ifge 24
 ifgt 24
 ifle 24
 iflt 24
 ifnc 23

- ifndef 23
- ifne 24
- 数式
 - の文法 13 ~ 15
- スコ - プ
 - シンボルの 9 ~ 11
- スコ - プ制御疑似命令 28 ~ 29
 - extern 28
 - global 28
 - public 29
- ステ - トメント
 - の文法 7 ~ 8
- 整数定数 12
- セクション制御疑似命令 25 ~ 27
 - bss 25
 - data 25
 - debug 26
 - offset 26
 - previous 26
 - rodata 25
 - sbss 26
 - sdata 25
 - sdata2 26
 - section 27
 - text 25
- 設定
 - Allow space in operand field 8, 34, 38
 - Case sensitive identifiers 9, 35, 38
 - Directives begin with '!' 21, 34, 38
 - Generate listing file 39
 - Labels must end with ':' 8, 34, 38
 - Prefix file 39
- 前方参照 9, 15
 - の文法 15 ~ 16

タ行

- 単項演算子 13
- 定数
 - 整数 12
 - の文法 12 ~ 13
 - 文字 13
 - 浮動小数点 12
- デ - タ定義疑似命令 30 ~ 32
 - ascii 32
 - asciz 32
 - byte 31
 - double 32
 - fill 31

- float 32
- long 31
- short 31
- space 31
- デバッグ疑似命令 35 ~ 36
 - file 35
 - function 35
 - line 36
 - size 36
 - type 36
- 等号 (=) 29

ナ行

- 二項演算子 14

ハ行

- 引数
 - マクロ 18
- 浮動小数点定数 12
- 文法
 - シンボルの 8 ~ 9
 - 数式の 13 ~ 15
 - ステ - トメントの 7 ~ 8
 - 前方参照の 15 ~ 16
 - 定数の 12 ~ 13

マ行

- マクロ
 - の定義 17 ~ 20
 - の呼び出し 20
 - 引数 18
- マクロ疑似命令 21 ~ 22
 - endm 22
 - macro 21
 - mexit 22
- マクロ定義 17
- メタシンボル 6
- 文字定数 13

ラ行

- ラベル 8
 - 局所 9
 - 再配置可能な 11
 - ユニ - クな ~ の生成 19
- リテラル 6

CodeWarrior

SH-4 Assembler Reference

Credits

writing lead: Roger Wong

other writers: BitHead, John Roseborough, Jeff Mattson, L. Frank Turovich

engineering: Matt Cole

frontline warriors: Jim Trudeau, Eric Clapton

translation: Eiko Kambara

proofreader: Shoji Ueda, Chizu Kanbara

CodeWarrior 文書のガイド

CodeWarrior の文書はツールと同様にモジュールのように構成されています。ツール、言語、ライブラリ、ターゲットごとにマニュアルがあります。各 CodeWarrior 製品によって含まれるマニュアルが異なります。この表に記載されていないマニュアルが含まれることもあります。

コアマニュアル	
IDE User Guide	CodeWarrior IDE と CodeWarrior デバッガの使い方
言語 / コンパイラのマニュアル	
C Compilers Reference	C/C++ フロントエンドコンパイラの情報
Pascal Compilers Reference	Pascal フロントエンドコンパイラの情報
Error Reference	コンパイラ / リンカのエラーメッセージのリストおよび解説
Pascal Language Reference	Metrowerks における ANS Pascal の実装
Assembler Guide	スタンドアローンアセンブラのシンタックス
Command-Line Tools Reference	Mac OS MPW コンパイラのコマンドラインのオプション
Plugin API Manual	CodeWarrior のプラグインコンパイラ / リンカの API
ライブラリのマニュアル	
MSL C Reference	Metrowerks ANSI 標準 C ライブラリの関数のリファレンス
MSL C++ Reference	Metrowerks ANSI 標準 C++ ライブラリの関数のリファレンス
Pascal Library Reference	Metrowerks ANS Pascal ライブラリの関数のリファレンス
MFC Reference	Win32 用の Microsoft Foundation Classes リファレンス
Win32 SDK Reference	Win32 API の Microsoft リファレンス
The PowerPlant Book	Mac OS 用アプリケーションフレームワークのガイド
PowerPlant Advanced Topics	PowerPlant での Mac OS プログラミングの高度なテクニック
ターゲットマニュアル	
Targeting Java VM	Java 仮想マシンプログラミングでの CodeWarrior の使い方
Targeting Mac OS	Mac OS プログラミングでの CodeWarrior の使い方
Targeting MIPS	MIPS 組み込みプロセッサプログラミングでの CodeWarrior の使い方
Targeting NEC V800 series	NEC V810/830 プロセッサプログラミングでの CodeWarrior の使い方
Targeting Net Yaroze	Net Yaroze プログラミングでの CodeWarrior の使い方
Targeting Palm OS	Palm OS プログラミングでの CodeWarrior の使い方
Targeting PlayStation	PlayStation プログラミングでの CodeWarrior の使い方
Targeting PowerPC	PPC 組み込みプロセッサプログラミングでの CodeWarrior の使い方
Targeting Windows	Windows プログラミングでの CodeWarrior の使い方

印の付いているマニュアルは日本語訳が用意されています。