

# **ARM® IAR アセンブラ**

## リファレンスガイド

Advanced RISC Machines Ltd

**ARM** コア対応

## **版權事項**

© Copyright 1999–2007 IAR Systems. All rights reserved.

IAR Systems が事前に書面で同意した場合を除き、このドキュメントを複製することはできません。このドキュメントに記載するソフトウェアは、正当な権限の範囲内でインストール、使用、およびコピーすることができます。

## **免責事項**

このドキュメントの内容は、予告なく変更されることがあります。また、IAR Systems 社では、このドキュメントの内容に関して一切責任を負いません。記載内容には万全を期していますが、万一、誤りや不備がある場合でも IAR Systems 社はその責任を負いません。

IAR Systems 社、その従業員、その下請企業、またはこのドキュメントの作成者は、特殊な状況で、直接的、間接的、または結果的に発生した損害、損失、費用、課金、権利、請求、逸失利益、料金、またはその他の経費に対して一切責任を負いません。

## **商標**

IAR Systems、IAR Embedded Workbench、C-SPY、visualSTATE、From Idea to Target、IAR KickStart Kit、IAR PowerPac、IAR YellowSuite、IAR Advanced Development Kit、IAR、および IAR Systems のロゴタイプは、IAR Systems AB が所有権を有する商標または登録商標です。J-Link は IAR Systems AB にライセンス供与されている商標です。

Microsoft および Windows は、Microsoft Corporation の登録商標です。

ARM および Thumb は、Advanced RISC Machines Ltd の登録商標です。

その他の製品名はすべて、各製品名の所有者の商標または登録商標です。

## **改版情報**

第 8 版：2008 年 9 月

部品番号：AARM-8-J

本ガイドは、ARM® IAR Embedded Workbench® IDE のバージョン 5.x に適用する。

内部参照：ISUD.

# 目次

表 .....	ix
はじめに .....	xi
<b>本ガイドの対象者</b> .....	xi
<b>本ガイドの使用方法</b> .....	xi
<b>本ガイドの内容</b> .....	xii
<b>参考資料</b> .....	xii
<b>表記規則</b> .....	xiii
<b>ARM IAR アセンブラの概要</b> .....	1
<b>アセンブラプログラミングの概要</b> .....	1
開始 .....	2
<b>モジュール方式のプログラミング</b> .....	2
<b>外部インタフェースの詳細</b> .....	3
アセンブラ呼出し構文 .....	3
オプションの受渡し .....	4
環境変数 .....	4
戻り値（エラー） .....	5
<b>ソースフォーマット</b> .....	5
<b>アセンブラ命令</b> .....	6
<b>式、オペランド、演算子</b> .....	6
整数定数 .....	6
ASCII 文字定数 .....	7
浮動小数点定数 .....	7
TRUE と FALSE .....	8
シンボル .....	8
ラベル .....	9
レジスタシンボル .....	9
定義済シンボル .....	10
絶対式および再配置可能な式 .....	11
式に関する制限事項 .....	12

リストファイルフォーマット .....	13
ヘッダ .....	13
本体 .....	13
要約 .....	13
シンボルと相互参照テーブル .....	13
プログラミングヒント .....	14
特殊な関数レジスタへのアクセス .....	14
C 形式プリプロセッサディレクティブの使用 .....	14
アセンブラオプション .....	15
コマンドラインオプションの設定 .....	15
コマンドライン拡張ファイル .....	15
アセンブラオプションの概要 .....	16
アセンブラオプションの説明 .....	17
アセンブラ演算子 .....	29
演算子の優先順位 .....	29
アセンブラ演算子の概要 .....	29
単項演算子 -1 .....	29
乗算型算術演算子 -2 .....	30
加算型算術演算子 -3 .....	30
シフト演算子 -4 .....	30
AND 演算子 -5 .....	30
OR 演算子 -6 .....	30
比較演算子 -7 .....	31
演算子の同義語 .....	31
演算子の説明 .....	32
アセンブラディレクティブ .....	43
アセンブラディレクティブの概要 .....	43
モジュール制御ディレクティブ .....	47
構文 .....	48
パラメータ .....	48
説明 .....	48

<b>シンボル制御ディレクティブ</b>	50
構文	50
パラメータ	50
説明	51
例	51
<b>モード制御ディレクティブ</b>	52
構文	52
説明	53
例	53
<b>セクション制御ディレクティブ</b>	54
構文	54
パラメータ	55
説明	55
例	56
<b>値割り当てディレクティブ</b>	57
構文	57
パラメータ	57
説明	58
<b>条件付きアセンブリのディレクティブ</b>	58
構文	59
パラメータ	59
説明	59
例	60
<b>マクロ処理ディレクティブ</b>	60
構文	61
パラメータ	61
説明	61
例	65
<b>リスト制御ディレクティブ</b>	68
構文	68
パラメータ	69
説明	69
例	70

<b>C 形式のプリプロセッサディレクティブ</b> .....	72
構文 .....	73
パラメータ .....	73
説明 .....	74
例 .....	76
<b>データ定義または割り当てのディレクティブ</b> .....	77
構文 .....	77
パラメータ .....	78
説明 .....	78
例 .....	78
<b>アセンブラ制御ディレクティブ</b> .....	79
構文 .....	80
パラメータ .....	80
説明 .....	80
例 .....	81
<b>呼び出しフレーム情報ディレクティブ</b> .....	82
構文 .....	83
パラメータ .....	84
説明 .....	85
単純ルール .....	89
CFI 式 .....	91
例 .....	93
<b>アセンブラ擬似命令</b> .....	97
要約 .....	97
擬似命令の説明 .....	98
<b>アセンブラの診断</b> .....	107
メッセージフォーマット .....	107
重大度 .....	107
診断オプション .....	107
アセンブリ時のワーニングメッセージ .....	107
コマンドラインエラーのメッセージ .....	107
アセンブリ時のエラーメッセージ .....	108
アセンブリ時の致命的なエラーメッセージ .....	108
アセンブラの内部エラーメッセージ .....	108

ARM ARM IAR アセンブラへの移行 .....	109
概要 .....	109
Thumb コードのラベル .....	109
代替レジスタ名 .....	110
代替ニーモニック .....	111
演算子の同義語 .....	112
ワーニングメッセージ .....	113
索引 .....	115





# 表

1: このガイドの表記規則 .....	xiii
2: アセンブラの環境変数 .....	4
3: アセンブラの戻り値 (エラー) .....	5
4: 整数定数の形式 .....	6
5: ASCII 文字定数の形式 .....	7
6: 浮動小数点定数 .....	7
7: 定義済のレジスタシンボル .....	9
8: 定義済シンボル .....	10
9: シンボルと相互参照テーブル .....	13
10: アセンブラオプションの概要 .....	16
11: 条件リスト (-c) .....	17
12: ユーザシンボルの大文字/小文字制御 (-s) .....	25
13: アセンブラのワーニングを無効にする (-w) .....	26
14: アセンブラのリストファイルに相互参照リストを生成 (-x) .....	27
15: 演算子の同義語 .....	31
16: アセンブラディレクティブの概要 .....	43
17: モジュール制御ディレクティブ .....	47
18: シンボル制御のディレクティブ .....	50
19: モード制御のディレクティブ .....	52
20: セクションの制御ディレクティブ .....	54
21: 値割り当てのディレクティブ .....	57
22: 条件付きアセンブリのディレクティブ .....	58
23: マクロ処理のディレクティブ .....	60
24: リスト制御のディレクティブ .....	68
25: C 形式のプリプロセッサディレクティブ .....	72
26: データ定義または割り当てのディレクティブ .....	77
27: アセンブラ制御のディレクティブ .....	79
28: 呼び出しフレーム情報ディレクティブ .....	82
29: CFI 式の単項演算子 .....	92
30: CFI 式の 2 項演算子 .....	92
31: CFI 式の 3 項演算子 .....	93

32: バックトレース行と列およびコード例 .....	94
33: 擬似命令 .....	97
34: 代替レジスタ名一覧 .....	110
35: 代替ニーモニック .....	111
36: 演算子の同義語 .....	112

# はじめに

ARM® IAR アセンブラ リファレンスガイドをご利用頂きありがとうございます。このガイドは、ご要件に合わせてご使用のアプリケーションを開発するために ARM IAR アセンブラをご利用いただく際に役立つ、詳細なリファレンス情報を提供します。

---

## 本ガイドの対象者

このガイドは、ARM core 用のアセンブラ言語でアプリケーション、またはアプリケーションの一部を開発する予定で、ARM® IAR アセンブラの使用法について詳細なリファレンス情報を得る必要がある方を対象としています。さらに、このガイドを利用する方は、次の内容に関する実務的な知識が必要です。

- ARM core のアーキテクチャ、命令セット。（ARM core については、Advanced RISC Machines Ltd の提供するドキュメントを参照）
- アセンブラ言語の一般的プログラミング
- 組込みシステム用アプリケーションの開発
- ホストコンピュータのオペレーティングシステム

---

## 本ガイドの使用方法

ARM® IAR アセンブラをはじめてご使用になる場合は、このリファレンスガイドの *ARM IAR アセンブラの概要*をお読みになることをお勧めいたします。

中級または上級レベルの方は、概要に続くリファレンスの章を中心にお読みになることをお勧めいたします。

IAR システムズのツールキットのご利用経験がない場合には、『*ARM® IAR Embedded Workbench® IDE ユーザガイド*』を最初にお読みになることをお勧めします。このユーザガイドでは、製品概要から製品を使い始める際に役立つチュートリアルまで解説しています。『*ARM® IAR C/C++ 開発ガイド*』には、用語集も収録されています。

---

## 本ガイドの内容

本ガイドの構成および各章の概要を以下に示します。

- *ARM IAR アセンブラの概要*では、プログラミング情報について説明しています。また、ソースコードフォーマットやアセンブラリストのフォーマットについても説明しています。
- *アセンブラオプション*では、まずコマンドラインでのアセンブラオプションの使い方と、環境変数の使い方について説明しています。次にオプションの概要をアルファベット順に説明した後、各オプションの詳細なリファレンス情報を記載しています。
- *アセンブラ演算子*では、アセンブラ演算子の概要をその優先順に示し、次に各演算子のリファレンス情報を説明しています。
- *アセンブラディレクティブ*では、ディレクティブの概要をアルファベット順に示し、次に機能別に分類して、各ディレクティブのリファレンス情報を詳細に説明しています。
- *アセンブラ擬似命令*では、有効な擬似命令について説明し、その使用例を示しています。
- *アセンブラの診断*では、診断メッセージのフォーマットと重大度について説明しています。
- *ARM IAR アセンブラへの移行*では、他のアセンブラ用に開発されたソースコードを ARM IAR アセンブラで使用するときに便利な情報が含まれています。

---

## 参考資料

IAR システムズの ARM core 用開発ツールについては、それぞれのガイドとオンラインヘルプで説明しています。知りたい情報に対応するドキュメントを以下に示します。

- IAR Embedded Workbench IDE での IAR C-SPY® デバッガの使用については、『*ARM® IAR Embedded Workbench® IDE ユーザガイド*』を参照してください。
- ARM IAR C/C++ コンパイラを使用したプログラミングおよび IAR ILINK リンカについては、『*ARM® IAR C/C++ 開発ガイド*』を参照してください。
- IAR DLIB ライブラリの使用方法については、オンラインヘルプシステムを参照してください。
- 旧バージョンの ARM IAR Embedded Workbench で作成したアプリケーションコードやプロジェクトの移植については、『*ARM® IAR Embedded Workbench® マイグレーションガイド*』を参照してください。

これらのガイドはすべて、ハイパーテキスト PDF または HTML フォーマットでインストール用メディアに収録されています。印刷物として提供されるガイドもあります。

# 表記規則

このガイドでは、次の表記規則を使用します。



スタイル	用途
computer	入力するテキストまたは画面に表示されるテキストを示します。
<i>parameter</i>	コマンドの一部として入力すべき値を示すラベル
[option]	コマンドのオプション部分
{option}	コマンドの必須入力部分を示します。
a   b   c	コマンド内の選択可能な部分
<b>bold</b>	画面に表示されるメニュー名、メニューコマンド、ボタン、およびダイアログボックス
<i>参照</i>	本ガイドや他のガイドへのクロスリファレンスを示します。
...	3点リーダーは、その前の項目を任意の回数繰り返せることを示します。
	IAR Embedded Workbench インタフェース固有の内容を示します。
	コマンドラインインタフェース固有の内容を示します。

表 1: このガイドの表記規則



# ARM IAR アセンブラの概要

この章は次のセクションで構成されています。

- アセンブラプログラミングの概要
- モジュール方式のプログラミング
- 外部インタフェースの詳細
- ソースフォーマット
- アセンブラ命令
- 式、オペランド、演算子
- リストファイルフォーマット
- プログラミングヒント

命令ニーモニックの構文説明については、Advanced RISC Machines Ltd 社のハードウェアドキュメントを参照してください。

---

## アセンブラプログラミングの概要

アセンブラ言語で完全なアプリケーションを記述しようとしていない場合でも、たとえば、正確なタイミングや特殊な命令シーケンスを要求する ARM core のメカニズムを使用する場合など、コードの一部をアセンブラで記述する必要が生じることがあります。

効率的なアセンブラアプリケーションを記述するには、ARM core のアーキテクチャおよび命令セットに精通する必要があります。命令ニーモニックの構文説明については、Advanced RISC Machines Ltd 社のハードウェアドキュメントを参照してください。

## 開始

以下のことを実行しておくと、アセンブラアプリケーションの開発を簡単に始めることができます。

- 『ARM® IAR Embedded Workbench® IDE ユーザガイド』にあるチュートリアル、特に C およびアセンブラモジュールの結合に関するチュートリアルを実行しておく。
- 『ARM® IAR C/C++ 開発ガイド』でアセンブラ言語インタフェースに関する項目を読んでおく（これは C およびアセンブラモジュールが混在するときにも便利です）。
- IAR Embedded Workbench IDE では、アセンブラプロジェクトのテンプレートに基づいて新しいプロジェクトを作成してみる。

---

## モジュール方式のプログラミング

一般的に、モジュール方式のプログラミングは、優れたソフトウェア設計の卓越した特長であるとみなされています。1 つの大きなモジュールとは対照的に、小さなモジュールでコードを構築することで、アプリケーションコードを論理的な構造で編成できます。これにより、コードが分かりやすくなりますし、以下の点で役に立ちます。

- 効率的なプログラム開発
- モジュールの再利用
- メンテナンス

IAR 開発ツールは、ソフトウェアでモジュール構造を実現するためのさまざまな機能を提供します。

通常、アセンブラソースファイルでアセンブラコードを記述します。各ファイルは、モジュールと呼ばれます。ソースコードをいくつかの小さいソースファイルに分割することで、たくさんの小さいモジュールを使用することになります。各モジュールは、さらにいくつかのサブルーチンに分割できます。

セクションとは、メモリ内の物理位置にマッピングされるデータやコードを含む論理エンティティです。コードとデータは、セクション制御ディレクティブを使用して、セクションに配置されます。セクションは再配置可能です。再配置可能セクションのアドレスは、リンク時に解決されます。セクションを使用することで、コードおよびデータがメモリにどのように配置されるかを制御できます。セクションとは、リンク可能な最小ユニットです。これにより、参照されるユニットだけをリンクで組み込むことができます。

大規模なプロジェクトに取り組んでいると、多くのアプリケーションで使用する便利なルーチンが集まってきます。小さなオブジェクトファイルが大量にならないようにするため、このようなルーチンをライブラリオブジェクトファイルに含むモジュールを収集できます。ライブラリのモジュールは、常に条件付きでリンクされることに注意してください。IAR Embedded Workbench



IDE では、ライブラリプロジェクトを設定して、大量のオブジェクトファイルを 1 つのライブラリにまとめることができます。この例については、『*ARM® IAR Embedded Workbench® IDE ユーザガイド*』のチュートリアルを参照してください。

要約すると、ソフトウェア設計では、モジュール方式のプログラミングを利用するとうまくいきます。モジュール構造を実現するには、以下のことを実行します。

- ソースファイルごとに 1 つずつ、大量の小さなモジュールを作成する。
- 各モジュールで、アセンブラソースコードを小さなサブルーチンに分割する（サブルーチンは C レベルの関数に対応します）
- アセンブラソースコードをセクションに分割して、コードとデータが最終的にメモリでどのように配置されるかをより精密に制御する。
- ルーチンをライブラリにまとめる。これにより、オブジェクトファイルの数を減らし、モジュールを条件付きでリンクできます。

---

## 外部インタフェースの詳細

このセクションは、アセンブラが設定された環境により、どのように動作するか説明します。

アセンブラは、IAR Embedded Workbench IDE またはコマンドラインインタフェースから使用できます。IAR Embedded Workbench IDE からのアセンブラの使用については、『*ARM® IAR Embedded Workbench® IDE ユーザガイド*』を参照してください。

### アセンブラ呼出し構文

アセンブラの呼出し構文は次のとおりです。

```
iasmarm [options] [sourcefile] [options]
```

たとえば、prog.s というソースファイルをアセンブルする場合は、次のコマンドを使用し、デバッグ情報を含むオブジェクトファイルを生成します。

```
iasmarm prog -r
```

デフォルトでは、ARM IAR アセンブラは、ソースファイルのファイル拡張子として s、asm、msa を認識します。

通常、コマンドラインでのオプションの順序とソースファイル名の前後のどちらに入力するかは、重要ではありません。ただし、例外が 1 つあります。-I オプションを使用する場合には、ディレクトリの検索はコマンドラインに指定した順序で行われます。アセンブラ出力のデフォルトのファイル名。

コマンドラインから引数なしでアセンブラを実行する場合、アセンブラのバージョン番号と利用可能なすべてのオプション（簡単な説明を含む）が stdout に転送され、画面に表示されます。

オプションの受渡し

オプションをアセンブラに渡す方法は 3 とおりあります。

- コマンドラインから直接渡す方法  
コマンドラインで、iasmarm コマンドの後にオプションを指定します (ページ 3 のアセンブラ呼出し構文を参照)。
- 環境変数経由で渡す方法  
アセンブラ は、自動的に環境変数の値を各コマンドラインの後に付加します (ページ 4 の環境変数を参照)。
- -f オプションを使用してテキストファイル経由で渡す方法 (-f、ページ 19 を参照)。

オプションの構文、オプションの概要、各オプションの詳細な説明に関する一般的なガイドラインについては、アセンブラオプションを参照してください。

環境変数

アセンブラオプションは IASMARM 環境変数で指定することもできます。アセンブラはこの変数の値をすべてのコマンドラインに自動的に追加します。この機能はすべてのアセンブリが必要とするオプションを指定するときに便利です。

ARMIAR アセンブラでは次表に示す環境変数を使用できます。

環境変数	説明
IASMARM	コマンドラインのオプションを指定します。 set IASMARM=-L -ws
IASMARM_INC	インクルードファイルを検索するディレクトリを指定します。例： set IASMARM_INC=c:\myinc\

表 2: アセンブラの環境変数

たとえば、次の環境変数を設定すると、リストは常に temp.lst ファイルに生成されます。

```
set IASMARM=-l temp.lst
```

コンパイラとリンカが使用する環境変数については、『ARM® IAR C/C++ 開発ガイド』を参照してください。

戻り値（エラー）

バッチファイル内から ARM IAR アセンブラを使用するとき、次に何を行うかを判断するため、アセンブリが成功したかどうかを確認しなければならないことがあります。この目的のため、アセンブラは次表に示す戻り値（エラー）を戻します。

戻り値	説明
0	アセンブリに成功、ワーニングが表示される場合があります。
1	ワーニングが発生しました（-ws オプションが使用されている場合のみ）。
2	エラーが発生しました。

表 3: アセンブラの戻り値（エラー）

ソースフォーマット

アセンブラソース行のフォーマットは次のとおりです。

[*label* [:]] [*operation*] [*operands*] [*;* *comment*]

それぞれのコンポーネントは次のとおりです。

<i>label</i>	ラベル定義（アドレスを表すシンボル）。最初の列から始まるラベル、つまり行の左端にあるラベルには : (コロン) を付けても付けなくてもかまいません。
<i>operation</i>	アセンブラ命令またはディレクティブです。行の 2 列目以降から開始しなくてはなりません。
オペランド	カンマで区切った、オペランドのリスト。
<i>comment</i>	コメントはセミコロン (;) から開始します。 C/C++ コメントは使用できません。

コンポーネントはスペースまたタブで区切ります。

ソース行の 1 行は 2047 文字を超えることはできません。

タブ文字 (ASCII 09H) は、最も一般的な慣行に従い、8、16、24 カラムなどに設定できます。これは、リストファイルおよびデバッグ情報のソースコード出力に影響を与えます。タブの設定はエディタごとに異なるので、ソースファイルではタブを使用しないことをお勧めします。

# アセンブラ命令

ARM IAR アセンブラは、『*ARM Architecture Reference Manual*』で説明されているようにアセンブラ命令の構文をサポートします。また、ワードアラインメントに関する ARM アーキテクチャの要件に準拠しています。コードセクションの奇数アドレスに命令を置くと、ワードアラインメント制限に関するエラーがコアで発生します。

# 式、オペランド、演算子

式は、式オペランドおよび演算子で構成されます。

アセンブラでは算術および論理演算の両方を含め、様々な式を使用できます。演算子はすべて 32 ビットの 2 の補数を使用します。範囲チェックは、値がコード生成に使用される場合に実行されます。

式は左から右に評価されますが、演算子の優先順位に基づく場合はこの限りではありません。詳細はアセンブラ演算子、ページ 29 を参照してください。

式中の有効なオペランドには次のものがあります。

- データまたはアドレスの定数（浮動小数定数除く）。
- データまたはアドレスを表現できるシンボル（シンボル名）。アドレスは、ラベルともいいます。
- プログラムロケーションカウンタ（PLC）、.（ピリオド）。

オペランドについては、以降のページで詳しく説明しています。

注：1 つの式で 2 つのシンボルを使用することはできません。式がアセンブリ時に解決できない限り、複雑な式を使用することもできません。この場合は、エラーが表示されます。

## 整数定数

IAR Systems アセンブラはすべて 32 ビットの 2 の補数による内部演算を使用するため、整数の（符号付き）範囲は -2147483648 から 2147483647 までとなります。

定数は一連の数字で記され、負数の場合には前にマイナス符号（-）が付けられます。

カンマと小数点は許されません。

次のタイプの表記法がサポートされています。

整数のタイプ	例
バイナリ	1010b、b'1010

表 4: 整数定数の形式

整数のタイプ	例
8 進数	1234q、q'1234
10 進数	1234, -1, d'1234
16 進数	0FFFFh、0xFFFF、h'FFFF

表 4: 整数定数の形式

注記：プレフィックスとサフィックスは、大文字と小文字のどちらでも表記できます。

ASCII 文字定数

ASCII 文字定数は、引用符または二重引用符に囲まれた任意の数の文字から構成されます。ASCII 文字列には印刷可能な文字とスペースのみを使用できます。引用符自体を使用するときには、引用符を 2 個続けて使用します。

Format	値
'ABCD'	ABCD (4 文字)
"ABCD"	ABCD'\0' (5 文字で最後の文字は ASCII nul)
'A''B'	A'B
'A''''	A'
'''' (4 つの引用符)	'
'' (2 つの引用符)	空白の 文字列 (値なし)
"" (2 つの二重引用符)	空白の文字列 (ASCII nul文字)
\'	','、'I\'d love to' のような文字列内での引用符
\\	\\、文字列内の \
\"	\"、文字列内での二重引用符

表 5: ASCII 文字定数の形式

浮動小数点定数

ARM IAR アセンブラは、浮動小数点値を定数として扱い、IEEE の単精度 (符号付 32 ビット) 浮動小数点形式、倍精度形式 (符号付 64 ビット) または fractional 形式に変換します。

浮動小数点は次のような形式で表現できます。

[+|-] [ 数字 ] . [ 数字 ] [{E|e} [+|-] 数字 ]

次の表に例を示します。

形式	値
10.23	1.023 × 10 <sup>1</sup>

表 6: 浮動小数点定数

形式	値
I.23456E-24	1.23456 x 10 <sup>-24</sup>
I.0E3	1.0 x 10 <sup>3</sup>

表 6: 浮動小数点定数 ( 続き )

浮動小数点定数にはスペースとタブにはスペースまたはタブを含めることはできません。

注 : 浮動小数点定数は、式の中で使用しても意味がありません。

fractional 形式が使用される場合 (例えば、DQ15)、表現できる範囲は -1.0 ≤ x < 1.0 になります。その範囲外の値はすべて、表現できる最大値または最小値に変更されます。

fractional 形式データのワード長が n の場合、小数は 2 の補数、つまり x \* 2<sup>(n-1)</sup> で表されます。

TRUE と FALSE

式においてゼロは FALSE、ゼロ以外の値は TRUE と見なされます。

条件式は FALSE の場合に 0、TRUE の場合には 1 を返します。

シンボル

ユーザ定義のシンボルは 255 字までの長さで、またすべての文字が有効となります。シンボルの後に続く操作の種類により異なりますが、シンボルは、データシンボルまたはアドレスシンボルのいずれかになります。アドレスシンボルはラベルともいいます。命令の前のシンボルはラベルです。また、たとえば、EQU ディレクティブの前のシンボルはデータシンボルです。シンボルは以下のいずれかにできます。

- 絶対シンボル (値はアセンブラで認識されます)
- 再配置可能シンボル (値はリンク時に解決されます)

シンボルは a ~ z または A ~ Z の英字、疑問符 (?)、または下線 ( \_ ) で始めなければなりません。シンボルには、数字の 0 ~ 9 とダラー ( \$ ) が使用できます。

シンボルには、バッククオート ( ` ) で囲まれているかぎりどのような刷可能文字も使用できます。

``strange#label``

命令、レジスタ、演算子、およびディレクティブなどの組み込みシンボルでは、大文字 / 小文字は区別されません。ユーザ定義のシンボルに関しては、デフォルトで大文字 / 小文字が区別されますが、区別するかどうかは、アセンブラの **Case sensitive user symbols** オプション ( -s ) で切り換えることができます。詳細は -s、ページ 25 を参照してください。

シンボル制御のディレクティブは、モジュール間でどのようにシンボルが共有されるかを制御します。たとえば、他のモジュールで 1 つまたは複数のシンボルを使用できるようにするには PUBLIC ディレクティブを使用します。また、タイプが設定されていない外部シンボルをインポートするには、EXTERN ディレクティブを使用します。

シンボルおよびラベルはバイトアドレスなので注意してください。その他の情報については、*ルックアップテーブルの生成*、ページ 78 を参照してください。

ラベル

ラベルは、メモリロケーションを表すシンボルです。

プログラムロケーションカウンタ（PLC）

アセンブラは、現在の命令の開始アドレスを追跡します。これは、プログラムロケーションカウンタと呼ばれます。

プログラムロケーションカウンタをアセンブラソースコードで参照する必要がある場合、.（ピリオド）記号を使用できます。次に例を示します。

```
SECTION MYCODE :CODE (2)
CODE32
    B . ; 永久ループ
END
```

レジスタシンボル

定義済のレジスタシンボルを次の表に示します。

名称	Size	説明
CPSR	32 ビット	現在のプログラムステータスレジスタ
D0-D15	64 ビット	倍精度のベクタ浮動小数点コプロセッサレジスタ
FPEXC	32 ビット	ベクタ浮動小数点コプロセッサ、例外レジスタ
FPSCR	32 ビット	ベクタ浮動小数点コプロセッサ、ステータスおよび制御レジスタ
FPSID	32 ビット	ベクタ浮動小数点コプロセッサ、システム ID レジスタ
R0-R12	32 ビット	汎用レジスタ
R13 (SP)	32 ビット	スタックポインタ
R14 (LR)	32 ビット	リンクレジスタ
R15 (PC)	32 ビット	プログラムカウンタ
S0-S31	32 ビット	単精度のベクタ浮動小数点コプロセッサレジスタ
SPSR	32 ビット	保存されたプロセスステータスレジスタ

表 7: 定義済のレジスタシンボル

また、コアによっては、命令構文で必要な場合、たとえば、Cortex-M3 の APSR など、他のレジスタシンボルを使用することもできます。

定義済シンボル

ARM IAR アセンブラでは、アセンブラソースファイル内で使用できるシンボルセットを定義しています。これらのシンボルは現在のアセンブリに関する情報を示し、それらをプリプロセッサディレクティブでテストしたり、アセンブルされたコードに含めたりすることができます。アセンブラが返す文字列は、二重引用符で囲まれています。

次の定義済シンボルが使用できます。

シンボル	値
__ARMVFP__	ベクタ浮動小数点コプロセッサの浮動小数点命令が有効にされているかどうかを示します (--fpu)。VFPv1 では数値 1 に、VFPv2 では数値 2 に展開します。浮動小数点命令が無効な場合（デフォルト）、シンボルの定義は解除されます。
__BUILD_NUMBER__	使用中のアセンブラのビルド番号を示す固有の整数です。
__DATE__	dd/Mmm/yyyy 形式の現在の日付（文字列）
__FILE__	現在のソースファイルの名前（文字列）
__IAR_SYSTEMS_ASM__	IAR アセンブラの識別子（数）
__IASMARM__	コードが ARM IAR アセンブラでアセンブルされた場合に 1 に設定される整数です。
__LINE__	現在のソース行番号（数）
__LITTLE_ENDIAN__	使用中のバイトオーダーを識別します。コードがリトルエンディアンのバイトオーダーでアセンブルされる場合、番号 1 を返し、ビッグエンディアンコードが生成される場合は、番号 0 を返します。リトルエンディアンがデフォルトです。
__TID__	2 バイトからなるターゲットの識別子（数）。上位バイトはターゲットの識別を行い、ARM IAR アセンブラでは 0x4F（= 10 進数の 79）です。下位バイトは使用されません。
__TIME__	hh:mm:ss 形式での現在の時刻（文字列）。
__VER__	整数形式のバージョン番号。たとえば、バージョン 4.17.54017005（数値）として返されます。

表 8: 定義済シンボル

また、事前に定義されているシンボルにより、たとえば、\_\_ARM5\_\_ および \_\_CORE\_\_ など、アセンブルするコアを識別できます。詳細については、『ARM® IAR C/C++ 開発ガイド』を参照してください。



## コードにシンボルの値を含める

シンボル値をコードに含めることができる、いくつかのデータ定義ディレクティブが提供されています。これらのディレクティブは値を定義するか、またはメモリを予約します。シンボル値をコードに含めるには、適切なデータ定義ディレクティブでシンボルを使用します。

たとえば、アセンブリの時刻を文字列として表示させるには、次のようにして行います。

```

                                EXTERN printstr
                                SECTION MYDATA :DATA (2)
                                DATA
tim      DC8 __TIME__ ; time string
                                SECTION MYCODE :CODE (2)
                                CODE32
                                ADR R0, tim ; 文字列のアドレスをロードする
                                BL printstr ; routine to print string

```

## 条件付きアセンブリでのシンボルのテスト

アセンブリ時にシンボルをテストするには、条件付きアセンブリディレクティブを使用します。これらのディレクティブを使用すると、アセンブリ時のアセンブリプロセスを制御できます。

たとえば、使用しているアセンブラバージョンが古いか新しいかにより異なるコードセクションをアセンブルする場合、以下のようにできます。

```

#if ( __VER__ > 4016005); 新しいアセンブラのバージョン
...
...
#else                                ; 古いアセンブラのバージョン
...
...
#endif

```

条件付きアセンブリのディレクティブ、ページ 58 を参照してください。

## 絶対式および再配置可能な式

式を構成するオペランドにより異なりますが、式は、*絶対式*または*再配置可能な式*のいずれかになります。絶対式は、絶対シンボルまたは再配置可能シンボルのいずれかのみを含む式です（両方は含みません）。

再配置可能なセクション内のシンボルを含む式は、そのセクションの配置に依存するため、アセンブリ時に解決することはできません。これらは、再配置可能な式ともいいます。

このような式は、リンク時に IAR ILINK リンカが評価して解決します。これらは、アセンブラにより縮小された後で、最大で 1 つのシンボル参照およびオフセットで構築できます。

たとえば、プログラムでは次のようにセクション MYDATA および MYCODE を定義できます。

```

        EXTERN third
        SECTION MYDATA :DATA  (2)
first:  DC32 3
second: DC32 4

```

セクション MYCODE では次のような再配置可能な式が使用できます。

```

        SECTION MYCODE :CODE  (2)
CODE32
; MYDATA は、0-255 の範囲になくてもなりません。
; さもないと、#first, #second 等が
; 範囲外になってしまいます。
MOV R1,#first
MOV R2,#second
MOV R3,#third
LDR R1,=first+4
LDR R2,=second
LDR R3,=third

```

**注記：**アセンブリ時に範囲のチェックは行いません。範囲のチェックは、リンク時にのみ行われ、値が大きすぎる場合、リンカがエラーを出力します。

## 式に関する制限事項

式は、いくつかのアセンブラディレクティブに適用される制限事項に応じて分類できます。たとえば、IF のような条件文で 사용되는式です。この場合、式は、アセンブリ時に評価される必要があるため、外部シンボルを含むことはできません。

以下に示す式に関する制限事項は、適用される各ディレクティブの説明に記載されています。

## 前方参照なし

式で参照されるすべてのシンボルは、解決していなければならなりません。前方参照は使用できません。

## 外部参照なし

式で外部参照は使用できません。

## 絶対式

絶対式は、絶対値に評価される必要があります。再配置可能な値（セクションオフセット）は使用できません。

固定

式は固定でなければなりません。つまり、可変サイズの命令は使用できません。可変サイズの命令は、そのオペランドの数値に従って、サイズが変化します。

リストファイルフォーマット

アセンブラリストファイルのフォーマットは以下のとおりです。

ヘッダ

ヘッダセクションには、製品のバージョン、ファイルが作成された日時、使用されたオプションに関する情報が含まれます。

本体

リストの本体には、以下の情報フィールドが含まれます。

- ソースファイルの行番号。マクロにより生成される行がリストされる場合は、（ピリオド）がソース行番号フィールドに示されます。
- アドレスフィールドには、メモリの位置が示されます。これは、セクションのタイプにより異なり、絶対値または相対値のいずれかです。表記は 16 進数です。
- データフィールドには、ソース行により生成されたデータが示されます。表記は 16 進数です。未解決の値は、.....（ピリオド）で表されます。ピリオド 2 つが 1 バイトです。これらの未解決の値は、リンクプロセで決されます。
- アセンブラソース行

要約

リストファイルの最後には、生成されたエラーとワーニングの要約が記述されています。

シンボルと相互参照テーブル

**Include cross-reference** オプションを指定する場合、または LSTXRF+ ディレクティブがソースファイルに含まれている場合、シンボルおよび相互参照テーブルが生成されます。

以下の情報は、表の各シンボルに提供されます。

情報	説明
シンボル	シンボルのユーザ定義名。
モード	ABS（絶対）または REL（再配置可能）。

表 9: シンボルと相互参照テーブル

情報	説明
セクション	このシンボルの定義に関連するセクションの名前。
値 / オフセット	現在のモジュール内でのシンボルの値 ( アドレス ) のセクションの開始点からの相対値。

表 9: シンボルと相互参照テーブル ( 続き )

## プログラミングヒント

このセクションでは ARM IAR アセンブラで効率的なコードを記述するためのヒントを示します。アセンブラおよび C/C++ ソースファイルを使用するプロジェクトについての詳細は、『ARM® IAR C/C++ 開発ガイド』を参照してください。

### 特殊な関数レジスタへのアクセス

多数の ARM デバイス用の固有ヘッダファイルは IAR システムズの製品パッケージに同梱され、ディレクトリ \arm\inc にあります。これらのヘッダファイルは、プロセス固有の特殊関数レジスタ (SFR)、および場合によっては割込みベクタ番号を定義します。

ヘッダファイルは ARM IAR C/C++ コンパイラで使用されるので、SFR 宣言はマクロで作成されています。宣言をアセンブラやコンパイラの構文用に変換するマクロは、io\_macros.h ファイルで定義されています。

また、ヘッダファイルは、他の ARM デリバティブ用に新しいヘッダファイルを作成するとき、テンプレートとしての使用に適しています。

#### 例

デバイスの USART 書き込みアドレス 0xFFFFD0000 は、ioat91m40400.h ファイルで次のように定義されています。

```
__IO_REG32_BIT (__US_CR, 0xffffd0000, __WRITE, __usartcr_bits)
```

宣言は、io\_macros.h ファイルで定義されているマクロによって次のように変換されます。

```
__US_CR DEFINE 0xffffd0000
```

### C 形式プリプロセッサディレクティブの使用

C 形式のプリプロセッサディレクティブは、他のアセンブラディレクティブの前に処理されます。したがってプリプロセッサディレクティブは、マクロ内では使えません。またアセンブラ形式のコメントと混用できません。コメントの詳細については、アセンブラ制御のディレクティブ、ページ 79 を参照してください。

# アセンブラオプション

この章ではまずコマンドラインからオプションを設定する方法について説明し、アセンブラオプションの概要をアルファベット順に示します。次にそれぞれのアセンブラオプションについて詳細に説明します。



『ARM® IAR Embedded Workbench® IDE ユーザガイド』には、IAR Embedded Workbench 内でアセンブラオプションを設定する方法と、使用可能なオプションのリファレンス情報が記載されています。

---

## コマンドラインオプションの設定

コマンドラインからアセンブラオプションを設定するには、次のようにコマンドラインの `iasmarm` コマンドの後にオプションを指定します。

```
iasmarm [options] [sourcefile] [options]
```

これらの項目は 1 つ以上のスペースまたはタブで区切る必要があります。

オプションのパラメータをすべて省略すると、アセンブラは使用可能なオプションのリストを画面上に表示します。リストの続きを見るには **Enter** キーを押します。

たとえば、デフォルトのリストファイル (`power2.lst`) にリストを生成するには、次のコマンドを使用してソースファイル `power2.s` をアセンブルします。

```
iasmarm power2 -L
```

オプションによっては、オプションの後にスペースで区切ってファイル名を指定できるものがあります。たとえば、ファイル `list.lst` にリストを生成するには、次のように指定します。

```
iasmarm power2 -l list.lst
```

また、ファイル名ではない文字列を指定できるオプションもあります。これもオプションの後に指定しますが、ただし区切りのスペースは使用しません。たとえば、`list` というサブディレクトリにデフォルトのファイル名でリストを生成するときのコマンドは次のようになります。

```
iasmarm power2 -Llist\
```

**注：**すでに存在しているサブディレクトリを指定してはなりません。サブディレクトリの名前とデフォルトのファイル名を区別するため、バックスラッシュを続けて指定することが必要です。

### コマンドライン拡張ファイル

アセンブラにはオプションとソースファイル名をコマンドラインから入力する方法の他に、コマンドライン拡張ファイル経由で入力することもできます。

デフォルトではコマンドライン拡張ファイルには拡張子「.xcl」が付けられ、「-f」コマンドラインオプションを使用してそのファイルを指定します。たとえば `extend.xcl` からコマンドラインオプションを読み込むには、次のように入力します。

```
iasmarm -f extend.xcl
```

## アセンブラオプションの概要

コマンドラインから使用可能なアセンブラオプションの概要を次表に示します。

コマンドラインオプション	説明
-B	マクロ実行情報を出力します。
-c	条件リストです。
--cpu	コア設定です。
-D	シンボルを定義します。
-E	エラーの最大数です。
-e	ビッグエンディアンのバイト順でコードを生成します。
--endian	コードおよびデータのバイトオーダーを指定します。
-f	コマンドラインを拡張します。
--fpu	浮動小数点ユニのを選択します。
-G	ソースファイルを標準入力から読み込みます。
-I	インクルードパスを指定します。
-i	#include ディレクティブでインクルードしたテキストを表示します。
-j	代替のレジスタ名、ニーモニック、および演算子を使用可能にします。
-L	指定した [ プレフィックス ] + ソース名にリストを出力します。
-l	リストファイルを出力します。
-M	マクロ引数の引用符を指定します。
-N	リストにヘッダを含めません。
-n	マルチバイト文字サポートを有効化
-O	オブジェクトファイルのプレフィックスを設定します。
-o	オブジェクトファイルの名称を設定します。
-p	ページ当りの出力行数を設定します。
-r	デバッグ情報を生成します。
-S	サイレント処理を設定

表 10: アセンブラオプションの概要

コマンドラインオプション	説明
-s	ユーザシンボルの大文字 / 小文字を区別します。
-t	タブによるスペースを設定します。
-U	シンボルの定義を解除します。
-w	ワーニング無効にする、または終了コードを変更します。
-x	相互参照リストを生成します。

表 10: アセンブラオプションの概要 ( 続き )

## アセンブラオプションの説明

以降のセクションでは、それぞれのアセンブラオプションについて詳細に説明します。



**[追加オプション]** ページを使用して特定のコマンドラインオプションを指定する場合、オプションの競合、オプションの重複、不適切なオプションの使用などの整合性問題に関する整合性チェックは実行されません。

-B -B

マクロが呼び出されるたびに、そのマクロの実行情報が標準の出力ストリームに出力されるよう設定します。この情報には次のものが含まれます。

- マクロ名称
- マクロ定義
- マクロ引数
- マクロ展開されたテキスト

このオプションは、主に -L オプションまたは -l オプションと同時に使用します。詳細については 22 ページを参照してください。



**[プロジェクト]>[オプション]>[アセンブラ]>[リスト]>[マクロ実行情報]**

-c -c{DMEAO}

アセンブラリストファイルの内容を制御します。このオプションは、主に -L オプションまたは -l オプションと同時に使用します。詳細については 22 ページを参照してください。

次の表に有効なパラメータを示します。

コマンドラインオプション	説明
-cD	リストファイルを無効

表 11: 条件リスト (-c)

コマンドラインオプション	説明
-cM	マクロ定義
-cE	マクロ展開なし
-cA	アセンブルされた行のみ
-cO	複数行コード

表 II: 条件リスト (-c) ( 続き )



オプションを設定するには、以下のように選択します。

[ プロジェクト ]>[ オプション ]>[ アセンブラ ]>[ リスト ]

--cpu --cpu target\_core

ターゲットコアを指定して正しい命令セットを得るには、--cpu オプションを使用します。

target\_core の有効値は ARM7TDMI のようなターゲット値とアーキテクチャバージョン（たとえば、デフォルトの 4T. ARM7TDMI）です。



[プロジェクト]>[オプション]>[一般オプション]>[ターゲット]>[プロセッサ選択]>[コア]

-D -Dsymbol [=value]

symbol という名称の、value という値を持つプリプロセッサシンボルを定義します。値が指定されていないときには値は 1 となります。

-D オプションを使用すると、ソースファイルで指定しなければならない値または選択肢を、コマンドラインから指定できます。

例

たとえば、シンボル TESTVER が定義されているかどうかに応じてプログラムのテストバージョンと製品バージョンのいずれかを生成するように、ソースコードを記述するとします。これには次のようにセクションに組み込みます。

```
#ifdef TESTVER
... ; テストバージョンでの追加コード
#endif
```

次に、必要なバージョンをコマンドラインで指定します。

```
製品バージョン: iasmarm prog
テストバージョン: iasmarm prog -DTESTVER
```



あるいは、頻繁に変更しなければならない変数がソースに使用されている場合があります。この場合ソース中の変数の値は定義しないで、次のように `-D` を使用してコマンドラインから値を指定します。

```
iasmarm prog -DFRAMERATE=3
```



[プロジェクト]>[オプション]>[アセンブラ]>[プリプロセッサ]>[定義済みシンボル]

---

`-E` *-Enumber*

アセンブラがレポートするエラーの最大数を設定します。

デフォルトでは、この最大数は 100 に設定されています。`-E` オプションを使用することにより、1 つのアセンブリ中のエラーの数を増減できます。



[プロジェクト]>[オプション]>[アセンブラ]>[診断]>[最大エラー数]

---

`-e` *-e*

コードとデータをビッグエンディアンバイトオーダーで生成します。デフォルトのバイトオーダーはリトルエンディアンです。



[プロジェクト]>[オプション]>[一般オプション]>[ターゲット]>[エンディアンモード]

---

`--endian` *--endian={little|l|big|b}*

このオプションは、生成したコードおよびデータのバイトオーダーを指定します。



[プロジェクト]>[オプション]>[一般オプション]>[ターゲット]>[エンディアンモード]

---

`-f` *-f filename*

`extend.xcl` というファイルから読み出されたテキストをコマンドラインに追加します。オプション自体とファイル名の間にはスペースが必要です。

`-f` オプションは、オプションを数多く使用し、それらをコマンドライン自体よりもファイルに配置した方が便利なときに使用します。

### 例

たとえば、`extend.xcl` ファイルからオプションを読み込んでアセンブラを実行するには、次のようなコマンドを使用します。

```
iasmarm prog -f extend.xcl
```



オプションを設定するには、以下を使用します。

[プロジェクト]>[オプション]>[アセンブラ]>[追加オプション]

---

```
--fpu --fpu={VFPv1|VFPv2|VFP9-S|none}
```

ターゲット浮動小数点コプロセッサを指定して正しい命令セットを得るには、  
--fpu オプションを使用します。

このオプションには次のパラメータが有効です。

VFPv1	VFPv10 rev 0 などの VFPv1 アーキテクチャに準拠したベクトル浮動小数点ユニット。
VFPv2	VFPv10 rev 1 などの VFPv2 アーキテクチャに準拠した VFP ユニットを実装するシステム。
VFP9-S	CPU コアの ARM9E ファミリと使用できる VFPv2 アーキテクチャの実装である VFP9-S。そのため、VFP9-S コプロセッサを選択することは、VFPv2 アーキテクチャを選択することと同じです。
none (デフォルト)	ソフトウェア浮動小数点ライブラリが使用されます。

--cpu オプションを使用して浮動小数点ユニットを持つターゲットコアを選択した場合、--fpu オプションが自動的に設定されます。



[プロジェクト]>[オプション]>[一般オプション]>[ターゲット]>[FPU]

---

```
-G -G
```

アセンブラに、指定したソースファイルではなく、標準入力からソースを読み込ませます。

-G を使用する場合、ソースファイル名は指定しません。



このオプションは、IAR Embedded Workbench IDE では使用できません。

---

```
-I -Iprefix
```

#include ファイル検索用のプレフィックス *prefix* を追加して、プリプロセッサが使用するパスを指定します。

デフォルトでは、アセンブラは `#include` ファイルを現在の作業ディレクトリと、`IASMARM_INC` 環境変数で指定されたパス内でのみ検索します。現在の作業ディレクトリ内にない場合、`-I` オプションを使用することにより、次に検索すべきディレクトリをアセンブラに指示できます。

### 例

次のオプションを使用し、

```
-Ic:\global\ -Ic:\thisproj\headers\
```

ソース内に次のように記述した場合、

```
#include "asmlib.hdr"
```

アセンブラはまず現在のディレクトリ内を検索します。次に `c:\global\` ディレクトリを検索し、最後に `c:\thisproj\headers\` ディレクトリ内を検索します。

`IASMARM_INC` 環境変数でインクルードパスを指定することもできます。詳細については、*環境変数*、ページ 4 を参照してください。



[プロジェクト]>[オプション]>[アセンブラ]>[プリプロセッサ]>[追加インクルードディレクトリ]

---

`-i -i`

`#include` ファイルをリストファイルに一覧表示します。

`#include` ファイルは通常、よく使用されるファイルであるため、リストの無駄を排除する目的で、デフォルトではアセンブラはこれらの行をリストに含めません。`-i` オプションを使用すると、これらのファイル行をリストに含めることができます。



[プロジェクト]>[オプション]>[アセンブラ]>[リスト]>[`#include` ファイル]

---

`-j -j`

他のアセンブラとの互換性を向上させ、コード移植を可能にするため、代替のレジスタ名、ニーモニック、および演算子を使用可能にします。

詳細については*演算子の同義語*、ページ 31、および *ARM IAR アセンブラへの移行*を参照してください。



[プロジェクト]>[オプション]>アセンブラ>[言語]>[代替ニーモニック、オペランド、レジスタ名を許可]

---

**-L** **-L** [*prefix*]

デフォルトでは、アセンブラはリストファイルを生成しません。このオプションを使用すると、アセンブラがリストファイルを生成し、それを [*prefix*] *sourcename.lst* に送ります。

単にリストを生成するには、プレフィックスなしで **-L** オプションを使用します。このリストはソースと同じ名前に拡張子 *lst* を付けたファイルに送られます。

**-L** オプションでは、たとえばリストファイルをサブディレクトリに置くため、プレフィックスを指定できます。プレフィックスの前にスペースを入れることはできません。

**-L** と **-l** とは同時に使用できません。

**例**

リストファイルをデフォルトの *prog.lst* ではなく、*list\prog.lst* に送るには、次のようなコマンドを使用します。

```
iasmarm prog -Llist\
```



オプションを設定するには、以下のように選択します。

[プロジェクト]>[オプション]>[アセンブラ]>[リスト]

---

**-l** **-l** *filename*

アセンブラがリストを作成し、これを *filename* で指定したファイルに送ります。拡張子が指定されていない場合には、*lst* が使用されます。ファイル名の前にはスペースが必要です。

デフォルトでは、アセンブラはリストファイルを生成しません。**-l** オプションによりリスト作成機能が有効になり、指定したファイルに送られます。デフォルトのファイル名にリストファイルを作成するときは、**-L** オプションを使用します。



オプションを設定するには、以下のように選択します。

[プロジェクト]>[オプション]>[アセンブラ]>[リスト]

---

**-M** **-Mab**

マクロ引数の左右の引用符を、それぞれ *a* と *b* に設定します。

デフォルトでは、これらの引用符は *<* と *>* です。**-M** オプションを使用することにより、別の表記法を使用するか、あるいはマクロ引数自体に *<* または *>* を含められるようになります。

**例**

次のようにオプションを使用します。

```
-M[]
```

ソース中で次のように記述すると、

```
print [>]
```

引数として > を使用する print マクロを呼び出せます。

**注記：**ホストの環境により、たとえば次のようにマクロの引用符に引用符 (') を使用しなければならないことがあります。

```
iasmarm filename -M'<>'
```



[プロジェクト]>[オプション]>[アセンブラ]>[言語]>[マクロの引用符]

---

-N -N

リストファイルの最初に出力される、ヘッダセクションを無効にします。

このオプションは、-l オプションまたは -1 オプションと同時に使用すると便利です。詳細については 22 ページを参照してください。



[プロジェクト]>[オプション]>[アセンブラ]>[リスト]>[ヘッダを含む]

---

-n -n

デフォルトでは、マルチバイト文字をアセンブラのソースコードで使用することはできません。このオプションを有効にすると、ソースコード内のマルチバイト文字は、ホストコンピュータのデフォルトのマルチバイト文字サポート設定に従って解釈されます。

マルチバイト文字は、C/C++ スタイルのコメント、文字列リテラル、文字定数で使用できます。これらはそのまま生成コードに移動します。



[プロジェクト]>[オプション]>[アセンブラ]>[言語]>[マルチバイトサポートを有効にする]

---

-O -Oprefix

オブジェクトファイルが格納されるディレクトリを設定します。プレフィックスの前にスペースを入れることはできません。

デフォルトではこの prefix はヌルです。したがって (-o が使用されていないかぎり) オブジェクトファイル名はソースファイル名に対応します。-o オプションを使用するとディレクトリを指定でき、たとえばオブジェクトファイルをサブディレクトリに格納することができます。

-o を -o と同時に使用することはできません。

**例**

オブジェクトをデフォルトファイルの `prog.o` : ではなく、`obj\prog.o` に送るには、次のようなコマンドを使用します。

```
iasmarm prog -Oobj\
```



[プロジェクト]>[オプション]>[一般オプション]>[出力]>[出力ディレクトリ]>  
[オブジェクトファイル]

---

```
-o -o filename
```

オブジェクトファイルに使用するファイル名を設定します。ファイル名の前にはスペースが必要です。拡張子が指定されていないときは、`o` が使用されます。

`-o` を `-o` と同時に使用することはできません。

**例**

次のコマンドはオブジェクトコードをデフォルトの `prog.o` ではなく、`obj.o` ファイルに入れます。

```
iasmarm prog -o obj
```

オプション自体とファイル名の間にはスペースが必要です。



[プロジェクト]>[オプション]>[一般オプション]>[出力]>[出力ディレクトリ]>  
[オブジェクトファイル]

---

```
-p -p lines
```

ページあたりの行数を `-p` に設定します。`lines` の範囲は 10 から 150 までです。

このオプションは、`-L` オプションまたは `-l` オプションと同時に使用します。詳細については 22 ページを参照してください。



[プロジェクト]>[オプション]>[アセンブラ]>[リスト]>[行数/ページ]

---

```
-r -r
```

`-r` オプションを使用すると、アセンブラは IAR C-SPY デバッガなどのシンボリックデバッガが使用するデバッグ情報を生成します

オブジェクトファイルのサイズとリンク時間を削減するため、デフォルトではアセンブラはデバッグ情報を生成しません。



[プロジェクト]>[オプション]>[アセンブラ]>[出力]>[デバッグ情報の生成]

-S -S

-s オプションを指定すると、アセンブラは標準出力にメッセージを送りません。

デフォルトでは、アセンブラはさまざまなメッセージを標準出力に送ります。  
-s オプションを使用することにより出力に送らないようにします。

エラーおよびワーニングメッセージは標準エラー出力に送られるため、この設定にかかわらず表示されます。



このオプションは、IAR Embedded Workbench IDE では使用できません。

-s -s{+|-}

アセンブラがユーザシンボルについて、大文字 / 小文字を区別するか否かを設定します。

コマンドラインオプション	説明
-s+	ユーザシンボルの大文字 / 小文字を区別します。
-s-	ユーザシンボルの大文字 / 小文字を区別しません。

表 12: ユーザシンボルの大文字 / 小文字の区別 (-s)

デフォルトでは大文字 / 小文字の区別が行われます。たとえば、`LABEL` と `label` は別のシンボルを意味します。大文字 / 小文字を区別しないときは `-s-` を使用し、この場合には `LABEL` と `label` は同じシンボルを意味するようになります。



[プロジェクト]>[オプション]>[アセンブラ]>[言語]>[ユーザシンボルで大文字と小文字を区別する]

-t -tn

デフォルトでは 1 つのタブあたり 8 文字分のスペースが設定されています。このオプションを使用して、タブによる移動量を `-t` で設定します。`n` の範囲は 2 から 9 までです。

このオプションは、`-l` オプションまたは `-1` オプションと同時に使用すると便利です。詳細については 22 ページを参照してください。



[プロジェクト]>[オプション]>[アセンブラ]>[リスト]>[タブ間隔]

-U -Usymbol

シンボル `symbol` の定義を解除します。

デフォルトでは、アセンブラにはあらかじめシンボルがいくつか定義されています。**定義済シンボル**、ページ 10 を参照してください。-U オプションを使用することによりこれらのシンボルの定義を解除し、その名称を以降の -D オプションまたはソース定義により、ユーザ定義のシンボルとして使用できるようになります。

例

定義済のシンボル `__TIME__` の名称をユーザ定義のシンボルとして使用するには、次のように指定します。

```
iasmarm prog -U __TIME__
```



このオプションは、IAR Embedded Workbench IDE では使用できません。

```
-w [-w[string] [s]]
```

デフォルトでは、アセンブラがソース中に構文上は正しいが、プログラムエラーを含む可能性のあるエレメントを発見するとワーニングメッセージが表示されます。詳細については**アセンブラの診断**、ページ 107 を参照してください。

このオプションはワーニングを無効にします。The -w オプションを範囲なしに使用すると、すべてのワーニングが表示されなくなります。範囲を指定した -w オプションは次のように動作します。

コマンドラインオプション	説明
-w+	すべてのワーニングを有効にします。
-w-	すべてのワーニングを無効にします。
-w+n	ワーニング <i>n</i> のみを有効にします。
-w-n	ワーニング <i>n</i> のみを無効にします。
-w+m-n	<i>m</i> から <i>n</i> までのワーニングを有効にします。
-w-m-n	<i>m</i> から <i>n</i> までのワーニングを無効にします。

表 13: アセンブラのワーニングを無効にする (-w)

コマンドラインでは -w オプションは 1 つだけ使用できます。

デフォルトでは、アセンブラはワーニングが存在しても終了コード 0 を生成します。ワーニングメッセージが作成されているときに終了コード 1 を生成するには -ws オプションを使用します。

例

ワーニング 0（参照なしのラベル）のみを表示しないようにするには、次のように -w オプションを使用します。



```
iasmarm prog -w-0
```

あるいは0から8までのワーニングを表示しないようにするには、次のように指定します。

```
iasmarm prog -w-0-8
```



オプションを設定するには、以下のように選択します。  
[プロジェクト]>[オプション]>[アセンブラ]>[診断]

```
-x -x{DI2}
```

リストの最後に相互参照リストを作成します。  
このオプションは、-l オプションまたは -i オプションと同時に使用すると便利です。詳細については 22 ページを参照してください。  
これには次のオプションを使用できます。

コマンドラインオプション	説明
-xD	#define
-xI	内部シンボル
-x2	ダブルスペースによる改行

表 14: アセンブラのリストファイルに相互参照リストを生成 (-x)



[プロジェクト]>[オプション]>[アセンブラ]>[リスト]>[クロスリファレンスを含む]



# アセンブラ演算子

この章ではまずアセンブラ演算子の優先順位について説明し、次に演算子の概要を優先順位に従って説明します。最後にそれぞれの演算子について、アルファベット順に詳細に説明します。

## 演算子の優先順位

それぞれの演算子には優先順位が設定され、演算子とオペランドが評価される順番はそれによって決定されます。この優先順位は 1（最高の優先順位 - 最初に評価される）から 7（最低の優先順位 - 最後に評価される）までです。

式の評価には、次のルールが適用されます。

- 優先順位が最高の演算子が最初に評価され、次に 2 番目に高い演算子が評価され、以下同様にして優先順位が最低の演算子が評価されるまで続きます。
- 優先順位が同一の演算子は、式内で左から右に評価されます。
- 演算子とオペランドをグループ化し、式が評価される順番を制御するには括弧の「(」と「)」を使用します。たとえば、次の式を評価した結果は 1 になります。

7/ (1+ (2\*3))

## アセンブラ演算子の概要

演算子の概要を、優先順位に従って次表に示します。同義語が存在する場合には、演算子名の後に同義語を示しています。

### 単項演算子 -I

+	単項プラス
-	単項マイナス
!	論理否定
~	ビットごとの論理否定
LOW	下位バイト
HIGH	上位バイト

BYTE1	1 番目のバイト
BYTE2	2 番目のバイト
BYTE3	3 番目のバイト
BYTE4	4 番目のバイト
LWRD	下位のワード
HWRD	上位のワード
DATE	現在の時刻 / 日付
SFB	セクションの開始
SFE	セクションの終了

**乗算型算術演算子 -2**

*	乗算
/	除算
%	剰余

**加算型算術演算子 -3**

+	加算
-	減算

**シフト演算子 -4**

>>	論理右シフト
<<	論理左シフト

**AND 演算子 -5**

&&	論理積
&	ビットごとの論理積

**OR 演算子 -6**

	論理和
	ビットごとの論理和
XOR	排他的論理和
^	ビットごとの排他的論理和

比較演算子 -7

=, ==	等しい
<>, !=	等しくない
>	大なり
<	小なり
UGT	符号なし大なり
ULT	符号なし小なり
>=	以上
<=	以下

演算子の同義語

他のアセンブラとの互換性のため、いくつかの演算子には同義語が設定されています。.

演算子の同義語	演算子	機能
:AND:	&	ビットごとの論理積
:EOR:	^	ビットごとの排他的論理和
:LAND:	&&	論理積
:LEOR:	XOR	排他的論理和
:LNOT:	!	論理否定
:LOR:		論理和
:MOD:	%	剰余
:NOT:	~	ビットごとの論理否定
:OR:		ビットごとの論理和
:SHL:	<<	論理左シフト
:SHR:	>>	論理右シフト

表 15: 演算子の同義語

**注：** 演算子の同義語を有効にするには、-j オプションを使用します。ARM の演算子と、演算子の同義語とでは優先順位が異なる場合があります。演算子とその同義語の優先順位については、以降の詳細な説明のセクションを参照してください。また *ARM IAR* アセンブラへの移行も参照してください。

---

## 演算子の説明

以降のセクションでは、それぞれのアセンブラ演算子について詳細に説明しています。関連情報として式、オペランド、**演算子**、ページ 6 を参照してください。括弧内の数字は、演算子の優先順位を示します。

---

### \* 乗算 (2)

\* は 2 つのオペランドによる積を計算します。これらのオペランドは符号付き 32 ビット整数であり、結果も符号付き 32 ビット整数になります。

#### 例

2\*2 → 4  
-2\*2 → -4

---

### + 単項プラス (1)

単項加算演算子です。

#### 例

+3 → 3  
3\*+2 → 6

---

### + 加算 (3)

+ 加算演算子は、それを囲む 2 つのオペランドの和を計算します。これらのオペランドは符号付き 32 ビット整数であり、結果も符号付き 32 ビット整数になります。

#### 例

92+19 → 111  
-2+2 → 0  
-2+-2 → -4

---

### - 単項マイナス (1)

単項マイナス演算子はオペランドに対して符号反転を行います。

このオペランドは符号付き 32 ビット整数として解釈され、また結果はその整数の 2 の補数になります。

**例**

```
-3 → -3
3*-2 → -6
4--5 → 9
```

---

**- 減算 (3)**

減算演算子は、左側のオペランドから右側のオペランドを差し引いた差を計算します。これらのオペランドは符号付き 32 ビット整数であり、また結果も符号付き 32 ビット整数になります。

**例**

```
92-19 → 73
-2-2 → -4
-2--2 → 0
```

---

**/ 除算 (2)**

/ は左側のオペランドを右側のオペランドで除算した商を計算します。これらのオペランドは符号付き 32 ビット整数であり、結果も符号付き 32 ビット整数になります。

**例**

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

---

**< 小なり (7)**

< は左側のオペランドが右側のオペランドよりも小さい数値を持つとき 1 (真) と評価され、それ以外の場合は 0 (偽) と評価されます。

**例**

```
-1 < 2 → 1
2 < 1 → 0
2 < 2 → 0
```

---

<= 以下 (7)

<= は左側のオペランドが右側のオペランドよりも小さい、またはそれと等しい数値を持つとき 1 (真) と評価され、それ以外の場合は 0 (偽) と評価されます。

**例**

```
1 <= 2 → 1
2 <= 1 → 0
1 <= 1 → 1
```

---

<>, != 等しくない (7)

<> は 2 つのオペランドの値が同一のとき 0 (偽)、値が等しくないとき 1 (真) と評価されます。

**例**

```
1 <> 2 → 1
2 <> 2 → 0
'A' <> 'B' → 1
```

---

=, == 等しい (7)

= は 2 つのオペランドの値が同一のとき 1 (真)、値が等しくないとき 0 (偽) と評価されます。

**例**

```
1 = 2 → 0
2 == 2 → 1
'ABC' = 'ABCD' → 0
```

---

> 大なり (7)

> は左側のオペランドが右側のオペランドよりも大きい数値を持つとき 1 (真) と評価され、それ以外の場合は 0 (偽) と評価されます。

**例**

```
-1 > 1 → 0
2 > 1 → 1
1 > 1 → 0
```



$\geq$  以上 (7)

**>=** は左側のオペランドが右側のオペランドよりも大きい、または等しい数値を持つとき 1（真）と評価され、それ以外の場合は 0（偽）と評価されます。

**例**

$$\begin{array}{lcl} 1 & \geq & 2 \rightarrow 0 \\ 2 & \geq & 1 \rightarrow 1 \\ 1 & \geq & 1 \rightarrow 1 \end{array}$$

&& (:LAND:) 論理積 (5)

**&&** は 2 つの整数オペランドの間で論理 AND 演算を行うときに使用します。両方のオペランドがゼロ以外の場合結果は 1 になり、それ以外の場合はゼロになります。

注： :LAND: の優先順位は 8 です。

**例**

$$\begin{array}{lcl} B'1010 \ \&\& \ B'0011 &\rightarrow 1 \\ B'1010 \ \&\& \ B'0101 &\rightarrow 1 \\ B'1010 \ \&\& \ B'0000 &\rightarrow 0 \end{array}$$

& (:AND:) ビットごとの論理積 (5)

&は整数オペランドの間でビットごとの AND 演算を行うときに使用します。32 ビットの結果の各ビットは、オペランドの対応するビットの論理 AND です。

注： :AND: の優先順位は 3 です。

**例**

$$\begin{array}{l} B'1010 \ \& \ B'0011 \rightarrow B'0010 \\ B'1010 \ \& \ B'0101 \rightarrow B'0000 \\ B'1010 \ \& \ B'0000 \rightarrow B'0000 \end{array}$$

~ (:NOT:) ビットごとの否定 (1)

~はオペランドに対してビットごとの否定演算を行うときに使用します。32ビットの結果の各ビットは、オペランドの対応するビットの補数です。

**例**

[illegible]

---

| (:OR:) ビットごとの論理和 (6)

| はオペランドに対してビットごとの論理和演算を行うときに使用します。32 ビットの結果の各ビットは、オペランドの対応するビットの包含的 OR です。

注: :OR: の優先順位は 3 です。

**例**

```
B'1010 | B'0101 → B'1111
B'1010 | B'0000 → B'1010
```

---

^ (:EOR:) ビットごとの排他的論理和 (6)

^ はオペランドに対してビットごとの排他的論理和演算を行うときに使用します。32 ビットの結果の各ビットは、オペランドの対応するビットの排他的 OR です。

注: :EOR: の優先順位は 3 です。

**例**

```
B'1010 ^ B'0101 → B'1111
B'1010 ^ B'0011 → B'1001
```

---

% (:MOD:) 剰余 (2)

% は左側のオペランドを右側のオペランドで除算した余りを計算します。これらのオペランドは符号付き 32 ビット整数であり、結果も符号付き 32 ビット整数になります。

$X \% Y$  は整数による除算を行った場合の  $X - Y * (X/Y)$  と等価になります。

**例**

```
2 % 2 → 0
12 % 7 → 5
3 % 2 → 1
```

---

! (:LNOT:) 論理否定 (1)

! は引数の否定に使用します。

**例**

```
! B'0101 → 0
! B'0000 → 1
```

---

|| (:LOR:) 論理和 (6)

|| は 2 つの整数オペランドの間で論理和演算を行うときに使用します。

**例**

```
B'1010 || B'0000 → 1
B'0000 || B'0000 → 0
```

---

BYTE1 1 番目のバイト (1)

BYTE1 は符号なし 32 ビット整数値と解釈される 1 つのオペランドをもちます。結果はそのオペランドの下位バイトの符号なし 8 ビット整数値になります。

**例**

```
BYTE1 0xABCD → 0xCD
```

---

BYTE2 2 番目のバイト (1)

BYTE2 は符号なし 32 ビット整数値と解釈される 1 つのオペランドをもちます。結果はオペランドの中間の下位バイト (15 ~ 8 ビット) になります。

**例**

```
BYTE2 0x12345678 → 0x56
```

---

BYTE3 3 番目のバイト (1)

BYTE3 は符号なし 32 ビット整数値と解釈される 1 つのオペランドをもちます。結果はオペランドの上位バイト (31 ~ 24 ビット) になります。

**例**

```
BYTE3 0x12345678 → 0x34
```

---

BYTE4 4 番目のバイト (1)

BYTE4 は符号なし 32 ビット整数値と解釈される 1 つのオペランドをもちます。結果はそのオペランドの上位 (31 ~ 24 ビット) のバイトになります。

**例**

```
BYTE4 0x12345678 → 0x12
```

---

DATE 現在の時刻 / 日付 (1)

DATE 演算子はアセンブリを開始した時刻を取得するときに使用します。

DATE 演算子は絶対値引数 (式) をとり、次のものを戻します。

DATE 1                   現在の秒 (0-59)

DATE 2                   現在の分 (0-59)

DATE 3                   現在の時 (0-23)

DATE 4                   現在の日 (1-31)

DATE 5                   現在の月 (1-12)

DATE 6                   現在の年 MOD 100 (1998 →98, 2000 →00, 2002 →02)

**例**

アセンブリの日付を表します。

today: DC8 DATE 5, DATE 4, DATE 3

---

HIGH 上位バイト (1)

HIGH は右側に 1 つのオペランドをもち、それを符号なし 16 ビット整数値と解釈します。結果はこのオペランドの上位のバイトの 8 ビット整数値になります。

**例**

HIGH 0xABCD → 0xAB

---

HWRD 上位のワード (1)

HWRD は 1 つのオペランドをもち、それを符号なし 32 ビット整数値と解釈します。結果はそのオペランドの上位 (31 ~ 16 ビット) のワードになります。

**例**

HWRD 0x12345678 → 0x1234

---

**LOW** 下位バイト (1)

LOW は 1 つのオペランドをもち、それを符号なし 16 ビット整数値と解釈します。結果はそのオペランドの下位バイトの符号なし 8 ビット整数値になります。

**例**

LOW 0xABCD → 0xCD

---

**LWRD** 下位のワード (1)

LWRD は 1 つのオペランドをもち、それを符号なし 32 ビット整数値と解釈します。結果はそのオペランドの下位 (15 ～ 0 ビット) のワードになります。

**例**

LWRD 0x12345678 → 0x5678

---

**SFB** セクションの開始 (1)**構文**

SFB (*section* [{+|-}*offset*])

**パラメータ**

<i>セクション</i>	セクションの名称で、これは SFB を使用する前に定義しておく必要があります。
<i>offset</i>	終了アドレスからのオフセットで、オプションです。 <i>offse</i> を省略するときには丸括弧はオプションです。

**説明**

SFB は右側に 1 つのオペランドをもちます。このオペランドはセクションの名称でなければなりません。

この演算子による評価の結果は、そのセクションの最初のバイトの絶対アドレスになります。評価はリンク時に行われます。

**例**

```
NAME demo
SECTION MYCODE :CODE (2)
start: DC16 SFB (MYCODE)
```

このコードが他の多数のモジュールにリンクされている場合にも、start はこのセクションの最初のバイトのアドレスに設定されます。

---

SFE セクションの終了 (1)

## 構文

SFE (*section* [{+ | -} *offset*])

## パラメータ

<i>section</i>	セクションの名称で、これは SFE を使用する前に定義しておく必要があります。
<i>offset</i>	終了アドレスからのオフセットで、オプションです。 offset を省略するときには丸括弧はオプションです。

## 説明

SFE は右側に 1 つのオペランドをもちます。このオペランドはセクションの名称でなければなりません。この演算子による評価の結果は、セクションの開始アドレスにセクションのサイズを加えたものになります。評価はリンク時に行われます。

## 例

```
NAME demo
SECTION MYCODE :CODE (2)
...
SECTION MYCONST : CONST (2)
end: DC32 SFE (MYCODE)
END
```

このコードが他の多数のモジュールにリンクされている場合にも、end はこのセクションの最後のバイトのアドレスに設定されます。

セクションの MYCODE のサイズは次のように計算します。

SFE (MYCODE) - SFB (MYCODE)

---

<< (:SHL:) 論理左シフト (4)

左側のオペランドを、左側にシフトするときに << を使用します (左側のオペランドは符号なしとして扱う)。シフトするビット数は、右側のオペランドで指定します (0 から 32 までの整数値として解釈される)。

## 例

```
B'00011100 << 3 → B'11100000
B'0000011111111111 << 5 → B'11111111111100000
14 << 1 → 28
```

---

**>> (:SHR:) 論理右シフト (4)**

左側のオペランドを、右側にシフトするときに >> を使用します（左側のオペランドは符号なしとして扱う）。シフトするビット数は、右側のオペランドで指定します（0 から 32 までの整数値として解釈される）。

**注：** :SHR: の優先順位は 2.5 になります。

**例**

```
B'01110000 >> 3 → B'00001110
B'1111111111111111 >> 20 → 0
14 >> 1 → 7
```

---

**UGT 符号なし大なり (7)**

UGT は左側のオペランドが右側のオペランドよりも大きい値を持つとき 1（真）と評価され、それ以外の場合は 0（偽）と評価されます。このオペランドは符号なしの値として扱われます。

**例**

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

---

**ULT 符号なし小なり (7)**

ULT は左側のオペランドが右側のオペランドよりも小さい値を持つとき 1（真）と評価され、それ以外の場合は 0（偽）と評価されます。このオペランドは符号なしの値として扱われます。

**例**

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

---

**XOR (:LEOR:) 排他的論理和 (6)**

XOR は、左側のオペランドと右側のオペランドのいずれかがゼロでないとき 1（真）と評価され、両側のオペランドがゼロのとき、または両側のオペランドがゼロ以外のとき 0（偽）と評価されます。XOR は 2 つのオペランドに対して排他的論理和演算を行うときに使用します。

**注：** :LEOR: の優先順位は 8 になります。

**例**

```
B'0101 XOR B'1010 → 0
B'0101 XOR B'0000 → 1
```





# アセンブラディレクティブ

この章では、アセンブラディレクティブをアルファベット順に説明した後、ディレクティブの各カテゴリの詳細なリファレンス情報を記載しています。

## アセンブラディレクティブの概要

アセンブラディレクティブは、各機能に従い以下のグループに分類されます。

- モジュール制御ディレクティブ、ページ 47
- シンボル制御ディレクティブ、ページ 50
- モード制御ディレクティブ、ページ 52
- セクション制御ディレクティブ、ページ 54
- 値割り当てディレクティブ、ページ 57
- 条件付きアセンブリのディレクティブ、ページ 58
- マクロ処理ディレクティブ、ページ 60
- リスト制御ディレクティブ、ページ 68
- C 形式のプリプロセッサディレクティブ、ページ 72
- データ定義または割り当てのディレクティブ、ページ 77
- アセンブラ制御ディレクティブ、ページ 79
- 呼び出しフレーム情報ディレクティブ、ページ 82.

すべてのアセンブラディレクティブについて、次表にその概要を示します。

ディレクティブ	説明	セクション
\$	ファイルをインクルードします。	アセンブラ制御
#define	ラベルに値を割り当てます。	C 形式のプリプロセッサ
#elif	#if...#endif ブロック内に新しい条件を追加します。	C 形式のプリプロセッサ
#else	条件が偽のときに命令をアセンブルします。	C 形式のプリプロセッサ
#endif	#if、#ifdef、または #ifndef ブロックを終了します。	C 形式のプリプロセッサ
#error	エラーを発生させます。	C 形式のプリプロセッサ
#if	条件が真のときに命令をアセンブルします。	C 形式のプリプロセッサ
#ifdef	シンボルが定義されているときに命令をアセンブルします。	C 形式のプリプロセッサ
#ifndef	シンボルが定義されていないときに命令をアセンブルします。	C 形式のプリプロセッサ

表 16: アセンブラディレクティブの概要

ディレクティブ	説明	セクション
#include	ファイルをインクルードします。	C 形式のプリプロセッサ
#line	行番号の変更	C 形式のプリプロセッサ
#message	標準出力上にメッセージを生成します。	C 形式のプリプロセッサ
#pragma	認識はされますが、無視されます。	C 形式のプリプロセッサ
#undef	ラベルの定義を解除します。	C 形式のプリプロセッサ
/*comment*/	C 形式のコメント区切り文字です。	アセンブラ制御
//	C++形式のコメント区切り文字です。	アセンブラ制御
=	モジュールにローカルな恒久的な値を割り当てます。	値割り当て
AAPCS	モジュール属性を設定	モジュール制御
ALIAS	モジュールにローカルな恒久的な値を割り当てます。	値割り当て
ALIGNRAM	プログラムロケーションカウンタをインクリメントして境界整列します。	セクション制御
ALIGNROM	ゼロのバイト列を挿入してプログラムロケーションカウンタの境界整列を行います。	セクション制御
ARM	これ以降の命令は 32 ビット (ARM) 命令として解釈されます。	モード制御
ASSIGN	一時的な値を割り当てます。	値割り当て
CASEOFF	大文字 / 小文字を区別しないように設定します。	アセンブラ制御
CASEON	大文字 / 小文字を区別するように設定します。	アセンブラ制御
CFI	呼び出しフレーム情報を指定します。	呼出しフレーム情報
CODE16	これ以降の命令は 16 ビット (Thumb) 命令として解釈されます。	モード制御
CODE32	これ以降の命令は 32 ビット (ARM) 命令として解釈されます。	モード制御
COL	ページあたりのカラム数を設定します。	リスト制御
DATA	コードセクション内のデータ領域を定義します。	モード制御
DC8	文字列を含む、バイト定数を生成します。	データ定義または割り当て
DC16	16 ビット定数を生成します。	データ定義または割り当て
DC24	24 ビット定数を生成します。	データ定義または割り当て
DC32	32 ビット定数を生成します。	データ定義または割り当て
DCB	文字列を含む、バイト (8 ビット) 定数を生成します。	データ定義または割り当て

表 16: アセンブラディレクティブの概要 ( 続き )

ディレクティブ	説明	セクション
DCD	32 ビットのロングワード定数を生成します。	データ定義または割り当て
DCW	文字列を含む、ワード（16 ビット）定数を生成します。	データ定義または割り当て
DEFINE	ファイル全体で有効な値を定義します。	値割り当て
DF32	浮動小数点（32 ビット）定数を生成します。	データ定義または割り当て
DF64	浮動小数点（64 ビット）定数を生成します。	データ定義または割り当て
DS8	8 ビット整数で空間を割り当てます。	データ定義または割り当て
DS16	16 ビット整数で空間を割り当てます。	データ定義または割り当て
DS24	24 ビット整数で空間を割り当てます。	データ定義または割り当て
DS32	32 ビット整数で空間を割り当てます。	データ定義または割り当て
ELSE	条件が偽のときに命令をアセンブルします。	条件付きアセンブリ
ELSEIF	IF...ENDIF ブロック内で新しい条件を指定します。	条件付きアセンブリ
END	ファイル中の最後のモジュールのアセンブリを終了します。	モジュール制御
ENDIF	IF ブロックを終了します。	条件付きアセンブリ
ENDM	マクロ定義を終了します	マクロ処理
ENDR	繰り返し構造を終了します。	マクロ処理
EQU	モジュールにローカルな恒久的な値を割り当てます。	値割り当て
EVEN	プログラムカウンタを偶数アドレスに境界整列します。	セクション制御
EXITM	マクロ定義が終了する前にマクロから抜け出します。	マクロ処理
EXTERN	外部シンボルをインポートします。	シンボル制御
EXTRN	外部シンボルをインポートします。	シンボル制御
EXTWEAK	外部シンボルをインポートします。シンボルが未定義の場合もあります。	シンボル制御
IF	条件が真のときに命令をアセンブルします。	条件付きアセンブリ
IMPORT	外部シンボルをインポートします。	シンボル制御
INCLUDE	ファイルをインクルードします。	アセンブラ制御
LIBRARY	モジュールのアセンブリを開始します。これは、PROGRAM および NAME のエイリアスです。	モジュール制御
LOCAL	マクロにローカルなシンボルを作成します。	マクロ処理

表 16: アセンブラディレクティブの概要（続き）

ディレクティブ	説明	セクション
LSTCND	条件付きアセンブラのリスト出力を制御します。	リスト制御
LSTCOD	複数行のコードのリスト出力を制御します。	リスト制御
LSTEXP	マクロが生成した行のリスト出力を制御します。	リスト制御
LSTMAC	マクロ定義のリスト出力を制御します。	リスト制御
LSTOUT	アセンブラリストの出力を制御します。	リスト制御
LSTPAG	これは、旧バージョンとの互換性のためです。認識はされますが、無視されます。	リスト制御
LSTREP	繰り返しディレクティブによって生成される行のリスト出力を制御します。	リスト制御
LSTXRF	相互参照テーブルを生成します。	リスト制御
LTORG	現在のリテラルプールを、ディレクティブの直後にアセンブルするよう指示します。	アセンブラ制御
MACRO	マクロを定義します。	マクロ処理
MODULE	モジュールのアセンブリを開始します。これは、PROGRAM および NAME のエイリアスです。	モジュール制御
NAME	プログラムモジュールのアセンブリを開始します。	モジュール制御
ODD	プログラムロケーションカウンタを奇数アドレスに境界整列します。	セクション制御
OVERLAY	認識はされますが、無視されます。	シンボル制御
PAGE	これは、旧バージョンとの互換性のためです。	リスト制御
PAGSIZ	これは、旧バージョンとの互換性のためです。	リスト制御
PRESERVE8	モジュール属性を設定	モジュール制御
PROGRAM	モジュールのアセンブリを開始します。	モジュール制御
PUBLIC	他のモジュールにシンボルをエクスポートします。	シンボル制御
PUBWEAK	他のモジュールにシンボルをエクスポートします。複数の定義が可能です。	シンボル制御
RADIX	デフォルトのベースを設定します。	アセンブラ制御
REPT	指定された回数だけ命令をアセンブルします。	マクロ処理
REPTC	文字の置き換えを繰り返します。	マクロ処理
REPTI	文字列の置き換えを繰り返します。	マクロ処理
REQUIRE	シンボルへの参照を強制します。	シンボル制御

表 16: アセンブラディレクティブの概要 ( 続き )

ディレクティブ	説明	セクション
REQUIRE8	モジュール属性を設定	モジュール制御
RSEG	セクションを開始します。	セクションの制御
RTMODEL	ランタイムモデル属性を宣言します。	モジュール制御
SECTION	セクションを開始します。	セクション制御
SECTION_TYPE	セクションの ELF タイプおよびフラグを設定します。	セクション制御
SET	一時的な値を割り当てます。	値割り当て
SETA	一時的な値を割り当てます。	値割り当て
THUMB	これ以降の命令は Thumb 拡張モード命令として解釈されます。	モード制御
VAR	一時的な値を割り当てます。	値割り当て

表 16: アセンブラディレクティブの概要 ( 続き )

## モジュール制御ディレクティブ

モジュール制御に関するディレクティブは、ソースプログラムモジュールの最初と最後へのマーク付け、およびそれらへの名称設定に使用されます。式でディレクティブを使用するときに適用される制限事項については、式に関する制限事項、ページ 12 を参照してください。

ディレクティブ	説明	式に関する制限事項
AAPCS	エクスポートされるモジュールのすべての関数が AAPCS に従っていることをリンクに通知するモジュール属性を設定します。	アセンブラは、用件が満たされているかどうか検証しません。
END	ファイル中の最後のモジュールのアセンブリを終了します。	ローカルで定義されたシンボルおよびオフセットまたは整数定数です。
NAME	モジュールのアセンブリを開始します。	外部参照なし 絶対式
PRESERVE8	エクスポートされるモジュールのすべての関数がスタックを 8 バイトアラインメントに維持することをリンクに通知するモジュール属性を設定します。	アセンブラは、要求が満たされているかどうか検証しません。
PROGRAM	モジュールのアセンブリを開始します。	外部参照なし 絶対式
REQUIRE8	モジュールがスタックを 8 バイトアラインメントにする必要があることをリンクに通知するモジュール属性を設定します。	
RTMODEL	ランタイムモデル属性を宣言します。	該当なし

表 17: モジュール制御ディレクティブ

## 構文

```
AAPCS [modifier [...]]
```

```
END
```

```
NAME symbol
```

```
PRESERVE8
```

```
PROGRAM symbol
```

```
REQUIRE8
```

```
RTMODEL key, value
```

## パラメータ

*key* キーを指定する文字列

**修飾子** AAPCS 拡張。可能な値は、INTERWORK および VFP、あるいはこれらの組み合わせです。

*symbol* モジュールに割り当てられる名称で

*value* 値を指定するテキスト文字列

## 説明

### モジュールの開始

ディレクティブ `NAME` または `PROGRAM` を使用して、ELF モジュールを開始し、名称を割り当てます。

モジュールは、他のモジュールにより参照されない場合でも、リンク後のアプリケーションに含まれます。リンク後のアプリケーションにモジュールがどのように含まれるかの詳細については、『*ARM® IAR C/C++ Development Guide*』のリンクプロセスを参照してください。

**注：** ファイルに含めることができるモジュールは 1 つだけです。

### ソースファイルの終了

`END` を使用してソースファイルの最後を指示します。`END` ディレクティブより後の行はすべて無視されます。ファイルの最後のモジュールが `ENDMOD` ディレクティブで明示的にしていない場合でも、`END` ディレクティブを使用してこのモジュールを終了できます。

## AEABI への準拠のためのモジュール属性の設定

モジュールで特定の属性を設定することで、モジュールのエクスポートされる関数が AEABI 規格の特定の部分に準拠していることをリンカに通知できます。

AAPCS を使用すると、モジュールが AAPCS 仕様に準拠していることを通知できます。また、モジュールがスタックを 8 バイトアラインメントに保存している場合は PRESERVE8、スタックの 8 バイトアラインメントを要求する場合は REQUIRE8 を使用します。

モジュールが実際にこれらの部分に準拠しているかどうかは、アセンブラでは検証されないため、ユーザが検証する必要があります。

## ランタイムモデル属性の宣言

モジュール間の互換性を確保するには RTMODEL を使用します。互いにリンクされ、同一のランタイム属性キーを定義するモジュールにはすべて、対応するキー値に同一の値が設定されているか、または特殊な値 \* を設定する必要があります。特殊な値 \* を使用することは、属性をまったく定義しないことと同じ意味を持ちます。ただしこれは、モジュールがどのようなランタイムモデルも取り扱えるということを明示的に示すときには便利です。

1 つのモジュールには複数のランタイムモデルの定義を行えます。

**注：** コンパイラのランタイムモデル属性は、2 つの下線で始まります。混乱を避けるため、ユーザ定義アセンブラ属性内ではこの形式を使用しないでください。

C or C++ コードで使用するためのアセンブラルーチンを作成し、モジュール間の互換性をコントロールしたい場合は、『*ARM® IAR C/C++ Development Guide*』を参照してください。

### 例

以下は、1 つのソース行それぞれに 3 つのモジュールを定義します。これらのモジュールについて説明します。

- MOD\_1 と MOD\_2 はランタイムモデル「foo」の値が異なるため、互いにリンクすることはできません。
- MOD\_1 と MOD\_3 はランタイムモデル「bar」の定義が同じであり、また「foo」の定義が矛盾しないため、互いにリンクすることができます。
- MOD\_2 と MOD\_3 はランタイムモデルの矛盾が存在しないため、互いにリンクすることができます。値「\*」はどのランタイムモデルの値とも一致します。

アセンブラソースファイル fl.s:

```
MODULE MOD_1
    RTMODEL    "foo", "1"
    RTMODEL    "bar", "XXX"
    ...
END
```

アセンブラソースファイル f2.s:

```
MODULE MOD_2
    RTMODEL    "foo", "2"
    RTMODEL    "bar", "*"
    ...
END
```

アセンブラソースファイル f3.s:

```
MODULE MOD_3
    RTMODEL    "bar", "XXX"
    ...
END
```

# シンボル制御ディレクティブ

これらのディレクティブは、モジュール間でどのようにシンボルが共有されるかを制御します。

ディレクティブ	説明
EXTERN, EXTRN, IMPORT	外部シンボルをインポートします。
EXTWEAK	外部シンボルをインポートします。シンボルが未定義の場合もあります。
OVERLAY	認識はされますが、無視されます。
PUBLIC	他のモジュールにシンボルをエクスポートします。
PUBWEAK	他のモジュールにシンボルをエクスポートします。複数の定義が可能です。
REQUIRE	シンボルへの参照を強制します。

表 18: シンボル制御のディレクティブ

## 構文

```
EXTERN symbol [, symbol] ...
EXTWEAK symbol [, symbol] ...
IMPORT symbol [, symbol] ...
PUBLIC symbol [, symbol] ...
PUBWEAK symbol [, symbol] ...
REQUIRE symbol
```

## パラメータ

<i>label</i>	C/C++ シンボルのエイリアスとして使用されるラベル
<i>symbol</i>	インポートまたはエクスポートされるシンボル



## 説明

### 他のモジュールへのシンボルのエクスポート

他のモジュールで 1 つまたは複数のシンボルを使用できるようにするには `PUBLIC` を使用します。`PUBLIC` と定義されたシンボルには再配置可能または絶対形式のいずれもが可能で、式の中でも使用できます（他のシンボルと同じルールが適用されます）。

`PUBLIC` ディレクティブは常に完全な 32 ビット値をエクスポートするため、8 ビットおよび 16 ビットのプロセッサ用のアセンブラでも可能なグローバル 32 ビット定数にすることができます。`LOW`、`HIGH`、`>>`、および `<<` 演算子を使用することにより、このような定数の任意の部分を、8 ビットまたは 16 ビットレジスタまたはワードにロードすることができます。

モジュール中で `PUBLIC` と宣言するシンボルの数に制限はありません。

### 複数の定義があるシンボルの他のモジュールへのエクスポート

`PUBWEAK` は、`PUBLIC` と似ていますが、同じシンボルを複数回定義できる点が異なります。これらの定義のうち 1 つだけが `ILINK` により使用されます。シンボルの `PUBLIC` 定義を含むモジュールが同じシンボルの `PUBWEAK` 定義を含む 1 つ以上のモジュールとリンクされる場合、`ILINK` は、`PUBLIC` 定義を使用します。

セクションには、`PUBLIC` シンボルと `PUBWEAK` シンボルの両方を含めることはできません。

**注記：**ライブラリモジュールがリンクされるのは、そのモジュール内のシンボルへの参照が行われ、そのシンボルがリンクされていない場合だけです。モジュール選択フェーズ中は、`PUBLIC` および `PUBWEAK` 定義は区別されません。つまり、`PUBLIC` 定義を含むモジュールが選択されるようにするには、これを他のモジュールの前にリンクするか、そのモジュール内の他の `PUBLIC` シンボルへの参照を行うようにする必要があります。

### シンボルのインポート

タイプが設定されていない外部シンボルをインポートするには `EXTERN` または `IMPORT` を使用します。

`REQUIRE` ディレクティブは、参照されたというマークをシンボルに付けます。これは、シンボルへの参照を含むコードを動作させるため、このシンボルを含むセクションがロードされる必要があるが依存関係が明確でない場合に便利です。

### 例

次の例ではエラーメッセージを出力するサブルーチンを定義し、他のモジュールから呼び出せるようにエントリアドレス `err` をエクスポートしています。

メッセージは二重引用符で囲まれているため、文字列の後はヌルになります。

ここで `print` を外部ルーチンとして定義しています。アドレスはリンク時に解決されます。

```
MODULE error
EXTERN print
PUBLIC err
SECTION MYCODE :CODE (2)
CODE16
PUSH {LR}
err ADR R0,msg
BL print
POP {PC}
SECTION MYDATA :DATA (2)
DATA
msg DC8 "***Error ***"
END
```

## モード制御ディレクティブ

これらのディレクティブはプロセッサのモードを制御します。

ディレクティブ	説明
ARM、CODE32	これ以降の命令は 32 ビット（ARM）命令としてアセンブルされます。CODE32 内のラベルは bit 0 が 0 に設定されます。4 バイトの境界整列が強制されます。
CODE16	これ以降の命令は、従来の CODE16 構文を使用して、16 ビット（Thumb）命令としてアセンブルされます。CODE16 内のラベルは bit 0 が 1 に設定されます。2 バイトの境界整列が強制されます。
DATA	コードセクション内で領域を定義します。ラベルは CODE32 領域として扱われます。
THUMB	これ以降の命令は、16 ビット Thumb 命令または 32 ビット Thumb-2 命令（指定コアで Thumb-2 命令セットがサポートされている場合）のいずれかとしてアセンブルされます。アセンブラ構文は、Advanced RISC Machines Ltd. で規定されている Unified Assembler 構文に従います。

表 19: モード制御のディレクティブ

### 構文

```
ARM
CODE16
CODE32
DATA
THUMB
```

## 説明

Thumb および ARM 間でプロセッサモードを変更するには、CODE16/THUMB および CODE32/ARM ディレクティブと BX (Branch and Exchange) 命令または実行モードを変更するその他の命令を使用します。CODE16/THUMB と CODE32/ARM モードディレクティブはモードを変更する命令にアセンブルされることはなく、単にアセンブラにそれ以降の命令をどのように解釈するかを指示するだけです。

DC8、DC16 または DC32 を持つ Thumb コードセクション中でデータを定義するときは、必ず DATA ディレクティブを使用します。これを行わない場合、データのラベルには bit 0 がセットされます。

**注：** 他のアセンブラ用に作成されたアセンブラソースコードを移植するときは、慎重に作業を行ってください。IAR アセンブラは常に Thumb コードラベル (local、external、または public) の bit 0 をセットします。詳細については *ARM ARM IAR アセンブラへの移行* を参照してください。

指定したコアで ARM モードがサポートされていない場合を除き、アセンブラは、最初に CODE32/ARM モードになります。ARM モードがサポートされていない場合、アセンブラは、最初に THUMB になります。

## 例

### プロセッサモードの変更

以下は、ARM 関数に対する THUMB エントリがどのように実装されるかを示した例です。

```
MODULE example
SECTION MYCODE :CODE (2)

    THUMB
thumbEntryToFunction
    BX PC ;armEntryToFunction にブランチし、動作モードを変更する
    ;
    NOP ; 境界整列のため
    ARM
armEntryToFunction
    .
    .
END
```

### DATA ディレクティブの使用

次の例では、DATA ディレクティブの後の 32 ビットラベルをどのように初期化するかを示しています。このラベルは CODE16 セクション内で使用できます。

```
MODULE example
SECTION MYCODE :CODE (2)
```

```
CODE16 ; Thumb instruction set used
my_code_label1 ldr r0,my_data_label1
my_code_label2 nop
DATA ; Remember the data directive,
; so that bit 0 is not set
; on labels
my_data_label1 DC32 0x12345678
my_data_label2 DC32 0x12345678

END
```

## セクション制御ディレクティブ

セクションディレクティブは、コードおよびデータがどのように配置されるかを制御します。式でディレクティブを使用するときに適用される制限事項については、式に関する制限事項、ページ 12 を参照してください。

ディレクティブ	説明	式に関する制限事項
ALIGNRAM	プログラムロケーションカウンタをインクリメントして境界整列します。	外部参照なし 絶対式
ALIGNROM	ゼロのバイト列を挿入してプログラムロケーションカウンタの境界整列を行います。	外部参照なし 絶対式
EVEN	プログラムカウンタを偶数アドレスに境界整列します。	外部参照なし 絶対式
ODD	プログラムカウンタを奇数アドレスに境界整列します。	外部参照なし 絶対式
RSEG	セクションを開始します。これは、SECTION のエイリアスです。	外部参照なし 絶対式
SECTION	ELF セクションを開始します。	外部参照なし 絶対式
SECTION_TYPE	セクションの ELF タイプおよびフラグを設定します。	

表 20: セクションの制御ディレクティブ

### 構文

```
ALIGNRAM align
ALIGNROM align [,value]
EVEN [value]
ODD [value]
RSEG section :type [flag] [ (align) ]
SECTION section :type [flag] [ (align) ]
SECTION_TYPE type-expr {,flags-expr}
```

## パラメータ

<i>align</i>	アドレスを整列させる 2 の累乗です。通常の場合、この範囲は 0 ～ 30 です。 整列のデフォルト値が 1 であるコードセクションを除いてデフォルト値は 0 です。
<i>flag</i>	NOROOT, ROOT  NOROOT では、セクションフラグメント内のシンボルが参照されない場合、このセクションフラグメントがリンクによって破棄されます。開始コードと割り込みベクタを除くすべてのセクションフラグメントでは、通常このフラグを設定すべきです。デフォルトモードは ROOT であり、セクションフラグメントは決して破棄されません。  REORDER、NOREORDER REORDER は、指定した名前で新しいセクションを開始します。デフォルトモードは、NOREORDER です。このモードでは、指定した名前のセクションで、または指定した名前のセクションがない場合は新しいセクションで、新しいフラグメントを開始します。
<i>section</i>	セクション名。
<i>type</i>	CODE、CONST、DATA のメモリタイプ。
<i>value</i>	パディングに使用されるバイト値。
<i>type-expr</i>	セクションの ELF タイプを識別する定数式。
<i>flags-expr</i>	セクションの ELF フラグを識別する定数式。

## 説明

### 再配置可能セクションの開始

SECTION（または RSEG）を使用して、新しいセクションを開始します。アセンブラはすべてのセクションについて別々のロケーションカウンタ（当初はゼロに設定）を維持するため、現在のプログラムロケーションカウンタを保存することなく、いつでもセクションとモードを切り換えることができます。

**注：**1 つのモジュール中に、SECTION または RSEG ディレクティブの最初のインスタンスの前には、DC や DS など、コードを生成する任意のディレクティブまたは任意のアセンブラ命令を付けないでください。

ELF タイプやセクションを新しく生成する場合の ELF フラグを設定するには、SECTION\_TYPE を使用します。デフォルトでは、フラグの値はゼロです。有効な値については、ELF マニュアルを参照してください。

## セクションの整列

プログラムロケーションカウンタを指定したアドレス境界で整列させるには `ALIGNROM` を使用します。`ALIGNROM` の式には、プログラムロケーションカウンタを整列すべき 2 のべき乗を指定します。許容範囲は 0 ～ 8 です。

境界整列はセクションの開始点から相対的に行われます。これは通常、望む結果を得るには、セクションの整列は少なくとも境界整列ディレクティブで指定された境界整列と同じ大きさでなければならないことを意味します。

`ALIGNROM` は値ゼロのバイト列を挿入して整列を行います。最大値は 255 です。

`EVEN` ディレクティブはプログラムカウンタを偶数アドレスに整列し（これは `ALIGNROM 1` と等価です）、`ODD` ディレクティブはプログラムロケーションカウンタを奇数アドレスに整列します。パディングのバイト値は、0 ～ 255 の範囲内になければなりません。

`ALIGNRAM` を使用して、プログラムロケーションカウンタを指定したアドレス境界で整列します。この式には、プログラムロケーションカウンタを整列すべき 2 のべき乗を指定します。`ALIGNRAM` は、プログラムロケーションカウンタをインクリメントすることによってデータの整列を行います。データは生成しません。

## 例

### 再配置可能セクションの開始

以下の例では、最初の `SECTION` ディレクティブに続くデータが、`MYDATA` という再配置可能セグメントに配置されます。

次の `SECTION` ディレクティブに続くコードは、`MYCODE` という再配置可能セクションに配置されます。

```

        EXTERN subrtn, divrtn
        SECTION MYDATA :DATA    (2)

        DATA
funcTable:
f1:DC32 subrtn
      DC32 divrtn

        SECTION MYCODE :CODE    (2)
        CODE32

main:
        LDR R0,=f1 ; アドレスを得る
        LDR PC,[R0] ; そこにジャンプする

```

セクションの整列

以下は、ALIGNRAM がどのように使用されるかを示した例です。

```
NAME align
SECTION MYDATA : DATA (6) ; リロケートブルセクションをスタートし、
; 64 バイト境界に整列されているかどうか
; 検証する
DATA
target1 DS16 1 ; 2 バイトデータ
ALIGNRAM 6 ; 64 バイト境界に整列する
results DS8 64 ; 64 バイトのテーブルを作成する
target2 DS16 1 ; 2 バイトデータ
ALIGNRAM 3 ; 8 バイト境界に整列します
ages DS8 64 ; 別の 64 バイトテーブルを作成します
END
```

値割り当てディレクティブ

これらのディレクティブはシンボルに値を割り当てるのに使用されます。

ディレクティブ	説明
=, EQU	モジュールにローカルな恒久的な値を割り当てます。
ALIAS	モジュールにローカルな恒久的な値を割り当てます。
ASSIGN, SET, SETA, VAR	一時的な値を割り当てます。
DEFINE	ファイル全体で有効な値を定義します。

表 21: 値割り当てのディレクティブ

構文

```
label = expr
label ALIAS expr
label ASSIGN expr
label DEFINE expr
label EQU expr
label SET expr
label SETA expr
label VAR expr
```

パラメータ

expr	シンボルに割り当てる値、またはテストする値
label	定義するシンボル

説明

一時的な値の定義

マクロ変数に使用するような、再定義されるシンボルの定義には、`ASSIGN`、`SET`、`SETA`、`VAR` を使用します。`ASSIGN`、`SET`、`VAR` によって定義されたシンボルは、`PUBLIC` として宣言することはできません。

恒久的なローカル値の定義

シンボルに値を割り当てるには、`EQU` または `=` を使用します。

数値あるいはオフセットを指定するローカルシンボルには `EQU` または `=` を使用します。このシンボルはそれが定義されたモジュール内でのみ有効ですが、`PUBLIC` ディレクティブ（`PUBWEAK` ディレクティブではありません）により他のモジュールでも使用可能になります。

他のモジュールからシンボルをインポートするには `EXTERN` を使用します。

恒久的なグローバル値の定義

ディレクトリを含むモジュール シンボルは、`DEFINE` ディレクティブの後で認識されます。

`DEFINE` で値が与えられるシンボルは、`PUBLIC` ディレクティブを使用して、他のファイルのモジュールで使用可能にできます。

`DEFINE` によって定義したシンボルを、同一ファイル内で再定義することはできません。

条件付きアセンブリのディレクティブ

これらのディレクティブは、ソースコードを選択的にアセンブルするための、論理的な制御メカニズムを提供します。式でディレクティブを使用するときに適用される制限事項については、式に関する制限事項、ページ 12 を参照してください。

ディレクティブ	説明	式に関する制限事項
ELSE	条件が偽のときに命令をアセンブルします。	
ELSEIF	IF...ENDIF ブロック内で新しい条件を指定します。	前方参照なし 外部参照なし 絶対式 固定
ENDIF	IF ブロックを終了します。	

表 22: 条件付きアセンブリのディレクティブ



ディレクティブ	説明	式に関する制限事項
IF	条件が真のときに命令をアセンブルします。	前方参照なし 外部参照なし 絶対式 固定

表 22: 条件付きアセンブリのディレクティブ ( 続き )

構文

ELSE  
ELSEIF *condition*  
ENDIF  
IF *condition*

パラメータ

<i>condition</i>	次のいずれかになります。	
	絶対式	この式には前方参照や外部参照を含めることはできません。ゼロ以外の値はすべて真と見なされます。
	<i>string1=string2</i>	この条件は <i>string1</i> と <i>string2</i> の長さおよび内容が同じ場合に真となります。
	<i>string1&lt;&gt;string2</i>	この条件は <i>string1</i> と <i>string2</i> の長さまたは内容が異なる場合に真となります。

説明

アセンブリ時にアセンブリ処理を制御するためには IF、ELSE、および ENDIF ディレクティブを使用します。IF ディレクティブに続く条件が真でない場合、それ以降の命令には ELSE または ENDIF ディレクティブが出現するまでコードが生成されません（つまりアセンブルや構文チェックは行われません）。

IF ディレクティブの後に新しい条件を追加するには、ELSEIF を使用します。条件付きアセンブリディレクティブはアセンブリ中のどこにでも使用できますが、マクロ処理と組み合わせた場合に最も役立ちます。

(END を除く) すべてのアセンブラディレクティブおよびファイルのインクルードは、条件ディレクティブによって無効にできます。IF ディレクティブはそれぞれ ENDEF ディレクティブによって閉じる必要があります。ELSE ディレクティブは省略可能ですが、使用する場合には IF...ENDEF ブロック内になければなりません。IF...ENDEF および IF...ELSE...ENDEF ブロックは、何段階にでも入れ子構造にすることができます。

例

下記のマクロでは、マクロ内で使われる引数の数に応じて、2 つの異なる ADD 命令を定義しています。

```
SECTION MYCODE :CODE (2)
CODE32
?add MACRO a,b,c
    IF _args <3
        ADD a,a, #b
    ELSE
        ADD a,b, #c
    ENDEF
ENDM
```

2 つのマクロ引数が与えられた場合には、add 命令の最初と 2 番目の引数は同一と想定されます。

```
main:
    MOV R1, #0xF0
    ?add R1, 0xFF ; この行
    ?add R1, R1, 0xFF ; とこの行
    add R1, R1, #0xFF ; は、この行と同じ
```

マクロ処理ディレクティブ

これらのディレクティブはユーザマクロの定義に使用します。式でディレクティブを使用するときに適用される制限事項については、式に関する制限事項、ページ 12 を参照してください。

ディレクティブ	説明	式に関する制限事項
ENDM	マクロ定義を終了します	
ENDR	繰り返し構造を終了します。	
EXITM	マクロ定義が終了する前にマクロから抜け出します。	
LOCAL	マクロにローカルなシンボルを作成します。	
MACRO	マクロを定義します。	

表 23: マクロ処理のディレクティブ ( 続き )

ディレクティブ 説明		式に関する制限事項
REPT	指定された回数だけ命令をアセンブルします。	前方参照できません 外部参照できません 絶対式 固定
REPTC	文字の置き換えを繰り返します。	
REPTI	テキストの置き換えを繰り返します。	

表 23: マクロ処理のディレクティブ( 続き )( 続き )

構文

ENDM  
ENDR  
EXITM  
LOCAL *symbol* [, *symbol*] ...  
*name* MACRO [*argument*] [, *argument*] ...  
REPT *expr*  
REPTC *formal*, *actual*  
REPTI *formal*, *actual* [, *actual*] ...

パラメータ

*actual*            代入される文字列  
*argument*        引数名 (シンボル)  
*expr*             式  
*formal*            *actual* (REPTC) または *actual* (REPTI) の各文字列が代入される引数  
*name*             マクロの名前  
*symbol*           マクロにローカルなシンボル

説明

マクロとは、1 つまたは複数のアセンブラソース行のブロックを表現するユーザ定義のシンボルです。定義したマクロは、アセンブラディレクティブやアセンブラニーモニックと同様にプログラム内で使用できます。

アセンブラがマクロを検出すると、アセンブラはそのマクロの定義を検索し、マクロで表現されたアセンブラソース行をソースファイル中のその位置に挿入します。

マクロは単純なテキストの代入を効率的に行いますが、パラメータを指定することにより代入するものを制御することができます。

## マクロの定義

マクロの定義は次の文によって行います。

```
name MACRO [argument] [,argument] ...
```

ここで *name* はマクロの名前で、*argument* はマクロの展開時にマクロに渡す値の引数です。

たとえばマクロ `errmac` を次のように定義できます。

```
        EXTERN errfunc
errmac MACRO text
        BL errfunc
        DATA
        DC8 text,0
        ENDM
```

このマクロでは、`errfunc` ルーチンのためのエラーメッセージを設定するために、`text` パラメータを使用します。このマクロは次のような文によって呼び出します。

```
SECTION MYCODE :CODE (2)
CODE32
errmac 'Disk not ready'
```

アセンブラはこれを次のように展開します。

```
BL errfunc
DATA
DC8 'Disk not ready',0
```

1 つまたは複数の引数のリストを省略した場合、マクロを呼び出すときに指定した引数を `\1` ～ `\9` および `\A` ～ `\Z` で参照できます。

したがってこの例は次のように書くことができます。

```
        EXTERN errfunc
errmac MACRO
        BL errfunc
        DATA
        DC8 \1,0
        ENDM
```

マクロ定義が終了する前にマクロから抜け出すには `EXITM` を使用します。

`EXITM` は `REPTENDR`、`REPTCENDR`、および `REPTIENDR` ブロック内では使用できません。

マクロにローカルなシンボルを作成するには `LOCAL` を使用します。`LOCAL` ディレクティブはシンボルを使用する前に使用しなければなりません。

マクロが展開されるたびに、LOCAL ディレクティブによりローカルなシンボルの新しいインスタンスが作成されます。したがってローカルなシンボルは再帰的なマクロ内でも使用できます。

**注記：**マクロの再定義は許されません。

## 特殊文字の使用

カンマまたはスペースを含むマクロ引数は、マクロ呼び出し内で引用符(< および >) で囲むことにより、1 つの引数として扱うことができます。

例

```
cmp_reg MACRO op
    CMP    op
ENDM
```

このマクロは、引数を引用符で囲んで次のように呼び出すことができます。

```
SECTION MYCODE :CODE (2)
CODE32
cmp_reg <R3, R4>
```

このマクロの引用符文字は、-M コマンドラインオプションによって変更できます。-M、ページ 22 を参照してください。

## 定義済マクロシンボル

シンボル `_args` にはマクロに渡される引数の数が設定されます。どのように `_args` が使用されるか、下記の例は示しています。

```
FILL MACRO
    IF _args == 2
        REPT \1
        DC8 \2
    ENDR
    ELSE
        DC8 \1
    ENDIF
ENDM

SECTION MYCODE :CODE (2)
    FILL 3, 4
    FILL 3
```

これにより次のコードが生成されます。

```
12                                SECTION MYCODE :CODE (2)
13                                FILL 3, 4
13.1                             IF _args == 2
13.2                             REPT 3
13.3                             DC8 4
```

```

13.4                                ENDR
13      00000000 04                DC8 4
13      00000001 04                DC8 4
13      00000002 04                DC8 4
13.1                                ELSE
13.2                                DC8 3
13.3                                ENDIF
13.4                                ENDM
14                                FILL 3
14.1                                IF _args == 2
14.2                                REPT 3
14.3                                DC8
14.4                                ENDR
14.5                                ELSE
14      00000003 03                FILL 3
14.1                                ENDIF
14.2                                ENDM

```

## マクロの処理方法

マクロの処理には、3つの明確なフェーズが存在します。

- 1 アセンブラはマクロ定義の走査と保存を行います。MACRO と ENDM の間のテキストが保存されますが、構文チェックは行われません。インクルードファイルのリファレンス *\$file* が記録され、マクロの展開時にインクルードされます。
- 2 マクロ呼び出しによりアセンブラはマクロプロセッサ（エクспанダ）を起動します。マクロエクспанダは（まだマクロ処理中でないときは）アセンブラの入力ストリームを、ソースファイルからマクロエクспанダからの出力に切り換えます。マクロエクспанダは、要求されたマクロ定義から入力を取得します。

マクロエクспанダは、ソースレベルでのテキスト置き換えを処理するだけなので、アセンブラシンボルに関する情報は意識しません。呼び出されたマクロ定義からの1行がアセンブラに渡される前に、エクспанダはマクロ引数シンボルのすべての出現箇所を走査し、それを展開する引数で置き換えます。

- 3 展開された行は他のアセンブラソース行と同様に処理されます。アセンブラへの入力ストリームは、現在のマクロ定義の行がすべて読み出されるまで引き続きマクロプロセッサから得られます。

## 文の繰り返し

同一の命令ブロックを複数回アセンブルするには REPT...ENDR 構造を使用します。expr を評価した結果が 0 のときは、何も生成されません。

文字列中の各文字について、命令のブロックを文字ごとにアセンブルするときは、REPTC を使用します。文字列にカンマが含まれるときは、その文字列を引用符で囲む必要があります。

二重引用符だけが特別な意味があり、繰り返す文字を囲むためだけに使用します。一重引用符には特別な意味はなく、通常の文字として扱われます。

一連の文字列中の各文字列について、命令のブロックを文字列ごとにアセンブルするときは REPTI を使用します。カンマが含まれる文字列は引用符で囲む必要があります。

## 例

このセクションでは、マクロを使用することによりアセンブラのプログラミングが容易になる例を、いくつか示しています。

### インラインのコーディングによる効率改善

実行時間が重要なコードでは、サブルーチン呼び出しと戻りによるオーバーヘッドを避けるために、ルーチンをインラインでコーディングした方が好都合な場合がよくあります。マクロはこのための便利な手法となります。

次の例では、バッファからポートにバイト出力されます。

```
IO_PORT EQU 0x0100
DATA
SECTION MYDATA_A :DATA (2)
start: DC32 main ; リセットベクタ
SECTION MYDATA_B :DATA (2)
DATA
buffer DS8 512 ; 512 バイトのバッファを確保
SECTION MYCODE :CODE (2)
CODE32
main: BL play
done: B done

play:
LDR R1,=buffer ; R1 をバッファへのポインタとして使用
LDR R2,=IO_PORT ; R2 をポートへのポインタに使用
LDR R3,=512 ; バッファサイズを R3 に格納
ADD R3,R3,R1 ; バッファの最終アドレス +1 を計算して R3 に格納

loop:
LDRB R4,[R1],#1 ; バイトデータを取り出し R4 に格納
; バッファポインタを +1 する
STRB R4,[R2] ; R2 のアドレスに書き込む
CMP R1, R3 ; R1 がバッファの最後に到達したかどうか
; 調べる
BNE loop ; 等しくない場合は繰り返す
MOV PC,LR ; リターン
```

効率を高めるため、マクロを使用して次のようにコードを作り直せます。

```
play: MACRO buf, size, port
LOCAL loop
```

```

        LDR R1,=buf           ; R1 をバッファへのポインタとして使用
        LDR R2,=port         ; R2 をポートへのポインタに使用
        LDR R3,=size         ; バッファサイズを R3 に格納
        ADD R3,R3,R1         ; バッファの最終アドレス +1 を計算して R3 に格納
loop:
        LDRB R4,[R1],#1      ; バイトデータを取り出し R4 に格納
                               ; バッファポインタを +1 する
        STRB R4,[R2]         ; R2 で指し示されるアドレスに
                               ; 書き込む
        CMP R1, R3           ; R1 がバッファの最終アドレス +1 と
                               ; 等しくなったかどうか調べる
        BNE loop             ; 等しくない場合は繰り返す
        ENDM
IO_PORT EQU 0x0100
        SECTION RESET :CODE (2)
start:B main
        SECTION MYDATA :DATA (2)
        DATA
buffer: DS8 512              ; 512 バイトのバッファを確保
        SECTION MYCODE :CODE (2)
        CODE32
main:
        play buffer,512,IO_PORT
done:B  done

```

LOCAL ディレクティブにより、ラベル loop がこのマクロに対してローカルに設定されていることに注意してください。

## REPTC と REPTI の使用

次の例では、文字列中の各文字を描くために、サブルーチン plotc に対する一連の呼び出しをアセンブルしています。

```

        NAME signon
        EXTERN plotc
        SECTION MYCODE :CODE (2)
        CODE32

banner REPTC chr, "Welcome"
        MOV R0,#'chr'        ; 各文字を R0 にパラメータとして渡す
        BL plotc
        ENDR

```

これにより、以下のコードが生成されます。

```

1
2                               NAME signon
3                               EXTERN plotc

```



```

4          SECTION MYCODE :CODE (2)
5          CODE32
6          banner REPTC chr, "Welcome"
7          MOV R0,#'chr' ; 各文字を R0 にパラメータ
8          ; として渡す
9          BL plotc
10         ENDR
10.1       00000000 5700A0E3 MOV R0,#'W' ; 各文字を R0 にパラメータ
10.2       ; として渡す
10.3       00000004 ..... BL plotc
10.4       00000008 6500A0E3 MOV R0,#'e' ; 各文字を R0 にパラメータ
10.5       ; として渡す
10.6       0000000C ..... BL plotc
10.7       00000010 6C00A0E3 MOV R0,#'l' ; 各文字を R0 にパラメータ
10.8       ; として渡す
10.9       00000014 ..... BL plotc
10.10      00000018 6300A0E3 MOV R0,#'c' ; 各文字を R0 にパラメータ
10.11      ; として渡す
10.12      0000001C ..... BL plotc
10.13      00000020 6F00A0E3 MOV R0,#'o' ; 各文字を R0 にパラメータ
10.14      ; として渡す
10.15      00000024 ..... BL plotc
10.16      00000028 6D00A0E3 MOV R0,#'m' ; 各文字を R0 にパラメータ
10.17      ; として渡す
10.18      0000002C ..... BL plotc
10.19      00000030 6500A0E3 MOV R0,#'e' ; 各文字を R0 にパラメータ
10.20      ; として渡す
10.21      00000034 ..... BL plotc

```

次の例では、REPTI を使用してメモリロケーションをいくつかクリアしています。

```

NAME repti
EXTERN base, count, init
SECTION MYCODE :CODE (2)
CODE32
MOV R0,#0

banner REPTI addr, base, count, init
LDR R1,=addr
STR R0,[R1]
ENDR

```

これにより、以下のコードが生成されます。

```

1          NAME repti
2          EXTERN base, count, init
3          SECTION MYCODE :CODE (2)
4          CODE32
5          00000000 0000A0E3 MOV R0,#0

```

```
6 banner REPTI addr, base, count, init
7 LDR R1,=addr
8 STR R0,[R1]
9 ENDR
9.1 00000004 10109FE5 LDR R1,= base
9.2 00000008 000081E5 STR R0,[R1]
9.3 0000000C 0C109FE5 LDR R1,= count
9.4 00000010 000081E5 STR R0,[R1]
9.5 00000014 08109FE5 LDR R1,= init
9.6 00000018 000081E5 STR R0,[R1]
```

## リスト制御ディレクティブ

これらのディレクティブは、アセンブラによるリストファイル作成を制御します。

ディレクティブ 説明	
COL	ページあたりのカラム数を設定します。
LSTCND	条件付きアセンブリのリスト出力を制御します。
LSTCOD	複数行のコードのリスト出力を制御します。
LSTEXP	マクロによって生成される行のリスト出力を制御します。
LSTMAC	マクロ定義のリスト出力を制御します。
LSTOUT	アセンブリリストの出力を制御します。
LSTPAG	ページへの出力形式を制御します。
LSTREP	繰り返しディレクティブによって生成される行のリスト出力を制御します。
LSTXRF	相互参照テーブルを生成します。
PAGE	新しいページを生成します。
PAGSIZ	ページあたりの行数を設定します。

表 24: リスト制御のディレクティブ

### 構文

```
COL columns
LSTCND{+|-}
LSTCOD{+|-}
LSTEXP{+|-}
LSTMAC{+|-}
LSTOUT{+|-}
LSTPAG{+|-}
LSTREP{+|-}
LSTXRF{+|-}
```

PAGE  
PAGESIZ *lines*

## パラメータ

*columns*      80 から 132 までの範囲の絶対式で、デフォルトは 80 です。  
*lines*          10 から 150 までの範囲の絶対式で、デフォルトは 44 です。

## 説明

### リスト出力のオン/オフ

エラーメッセージを除くすべてのリスト出力を行わないようにするには LSTOUT- を使用します。このディレクティブは他のすべてのリスト制御ディレクティブより優先されます。

デフォルトは LSTOUT+ で、(リストファイルが指定されていれば) リスト出力が行われます。

### 条件付きコードと文字列のリスト

LSTCND+ を使用すると、IF 文によって無効にされていないアセンブル箇所ソースコードのみがリスト出力されます。

デフォルトの設定は LSTCND- で、すべてのソース行がリスト出力されます。

LSTCOD- を使用すると、出力コードのリストが、ソースコード 1 行につき最初の行だけに制限されます。

デフォルトの設定は LSTCOD+ で、ソースコード 1 行につき必要があれば複数行のコードがリスト出力されます。つまり長い ASCII 文字列からは、複数行が出力されます。コードの生成には影響はありません。

### マクロのリスト制御

LSTEXP- を使用すると、マクロで生成された行がリスト出力されなくなります。デフォルトは LSTEXP+ で、マクロで生成された行はすべてリスト出力されます。

マクロ定義をリスト出力するには LSTMAC+ を使用します。デフォルトは LSTMAC- で、マクロ定義のリストは出力されません。

### 生成された行のリスト制御

LSTREP- を使用すると、ディレクティブ REPT、REPTC、および REPTI によって生成された行はリスト出力されなくなります。

デフォルトは LSTREP+ で、生成された行がリスト出力されます。

## 相互参照テーブルの生成

LSTXRF+ を使用すると、現在のモジュールのアセンブラリストの最後に、相互参照テーブルが生成されます。このテーブルにはシンボルの値と行番号、およびタイプが示されます。

デフォルトは LSTXRF- で、相互参照テーブルは生成されません。

## リストファイル出力形式の指定

アセンブラリストのページあたりのカラム数を設定するには COL を使用します。デフォルトのカラム数は 80 です。

アセンブラリストのページあたりの行数を設定するには PAGESIZ を使用します。デフォルトの行数は 44 です。

アセンブラ出力リストをページ単位でフォーマットするには LSTPAG+ を使用します。

デフォルトは LSTPAG- で、連続したリストが出力されます。

ページ作成が有効なとき、アセンブラリスト中に新しいページを生成するには PAGE を使用します。

## 例

### リスト出力のオン/オフ

次の例では、デバッグされたプログラムセクションについては、リストが出力されないように設定しています。

```
LSTOUT-
; Debugged section
LSTOUT+
; Not yet debugged
```

### 条件付きコードと文字列のリスト

次の例では、IF ディレクティブによって無効とされたサブルーチン呼び出しを、LSTCND+ を使用して隠しています。

```
NAME lstcndtst
EXTERN print
SECTION PROM : CODE (2)
CODE32

debug SET 0
begin IF debug
BL print
ENDIF
LSTCND+
```

```
begin2  IF debug
        BL print
        NOP
        ENDIF
```

これにより以下のリストが生成されます。

```
1                                     NAME lstcndtst
2                                     EXTERN print
3                                     SECTION PROM :CODE (2)
4                                     CODE32
5                                     debug SET 0
6                                     begin IF debug
7                                     BL print
8                                     ENDIF
9                                     LSTCND+
10                                    begin2 IF debug
13                                    ENDIF
```

次の例では、DATA ディレクティブによって生成されたコードに対する LSTCOD- の効果を示しています。

```
        NAME lstcodtst
        SECTION MYDATA :DATA (2)
        DATA
table1:DC32 1,10,100,1000,10000
        LSTCOD-
table2:
        DC32 1,10,100,1000,10000
        END
```

これにより以下のリストが生成されます。

```
1                                     NAME lstcodtst
2                                     SECTION MYDATA :DATA (2)
3                                     DATA
4      00000000 010000000A00 table1:DC32 1,10,100,1000,10000
                                     000064000000
                                     E80300001027
                                     0000
5                                     LSTCOD-
6                                     table2:
7      00000014 010000000A00* DC32 1,10,100,1000,10000
8                                     END
```

## マクロのリスト制御

次の例では、LSTMAC と LSTEXP の効果を示しています。

```
        SECTION MYCODE :CODE (2)
        CODE32
```

```
dec2      MACRO arg
          ADD R1,R1,#-arg
          ADD R1,R1,#-arg
          ENDM
          LSTMAC-
inc2      MACRO arg
          ADD R1,R1,#arg
          ADD R1,R1,#arg
          ENDM

begin:
          dec2 127
          LSTEXP-
          inc2 0x20
```

これにより以下のリストが生成されます。

```
1
2
3          SECTION MYCODE :CODE (2)
4          CODE32
5          dec2 MACRO arg
6          ADD R1,R1,#-arg
7          ADD R1,R1,#-arg
8          ENDM
9          LSTMAC-
10
11         begin:
12         dec2 127
13.1 00000000 7F1041E2 ADD R1,R1,#-127
13.2 00000004 7F1041E2 ADD R1,R1,#-127
13.3
14         ENDM
15         LSTEXP-
16         inc2 0x20
```

## C 形式のプリプロセッサディレクティブ

以下の C 言語プリプロセッサディレクティブが利用できます。

ディレクティブ	説明
#define	プリプロセッサシンボルに値を割り当てます。
#elif	#if...#endif ブロック内に新しい条件を追加します。
#else	条件が偽のときに命令をアセンブルします。
#endif	#if、#ifdef、または #ifndef ブロックを終了します。
#error	エラーを発生させます。
#if	条件が真のときに命令をアセンブルします。

表 25: C 形式のプリプロセッサディレクティブ

ディレクティブ	説明
#ifdef	プリプロセッサシンボルが定義されているときに命令をアセンブルします。
#ifndef	プリプロセッサシンボルが定義されていないときに命令をアセンブルします。
#include	ファイルをインクルードします。
#line	#line ディレクティブの直後のソースコード行の行番号、またはコンパイルされるファイルのファイル名を変更します。
#message	標準出力上にメッセージを生成します。
#pragma	このディレクティブは、認識はされますが、無視されます。
#undef	プリプロセッサシンボルの定義を解除します。

表 25: C 形式のプリプロセッサディレクティブ ( 続き )

### 構文

```
#define symbol text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef symbol
#ifndef symbol
#include { "filename" | <filename> }
#message "message"
#undef symbol
```

### パラメータ

<i>condition</i>	絶対式	この式にはアセンブララベルまたはシンボルを含めることはできません。ゼロ以外の値はすべて真とみなされます。
<i>filename</i>	インクルードするファイルの名称です。	
<i>message</i>	表示されるテキストです。	
<i>symbol</i>	定義、定義解除、またはテストされるプロセッサシンボルです。	
<i>text</i>	割り当てられる値です。	

## 説明

アセンブラ言語には複数の C 形式プリプロセッサディレクティブを混在させることが重要です。概念的にはディレクティブは異なる言語であり、混在させると意外な動作を招くことがあります。アセンブラディレクティブは必ずしも C プリプロセッサ言語として扱われていないからです。

プリプロセッサディレクティブは、他のディレクティブの前に処理されるので注意してください。avoid の構成の例を以下に示します。

```
redef macro      ; avoid the following
#define \1 \2
    endm
```

\1 および \2 マクロ引数は、プリプロセッサフェーズ中には使用できなくなります。

## プリプロセッサシンボルの定義および定義解除

プリプロセッサシンボルの値を定義するには、`#define` を使用します。

```
#define symbol value
```

シンボルの定義を解除するには `#undef` を使用します。この結果、このラベルが定義されたことがないかのように取り扱われます。

## 条件プリプロセッサディレクティブ

アセンブリ時にアセンブリ処理を制御するには `#if...#else...#endif` ディレクティブを使用します。`#if` ディレクティブに続く条件が真でない場合、それ以降の命令は `#endif` または `#else` ディレクティブが出現するまでコードを生成しません（つまりアセンブルや構文チェックは行われません）。

END を除く）すべてのアセンブラディレクティブおよびファイルのインクルードは、条件ディレクティブによって無効にできます。`#if` ディレクティブはそれぞれ `#endif` ディレクティブによって閉じる必要があります。`#else` ディレクティブは省略可能で、使用する場合には `#if...#endif` ブロック内になければなりません。

`#if...#endif` およ `#if...#else...#endif` ブロックは、何段階にでも入れ子構造にすることができます。

`#ifdef` を使用すると、シンボルが定義されている場合のみ次の `#else` または `#endif` ディレクティブまでの命令がアセンブルされます。

`#ifndef` を使用すると、シンボルが定義されていない場合のみ次の `#else` または `#endif` ディレクティブまでの命令がアセンブルされます。

## ソースファイルのインクルード

あるファイルの内容を、ソースファイル中の指定した箇所に挿入するには `#include` を使用します。



`#include "filename "` は、以下のディレクトリをこの順番で検索します。

- 1 ソースファイルディレクトリ
- 2 `-I` オプションで指定されたディレクトリ
- 3 現在のディレクトリ

`#include <filename >` は、以下のディレクトリをこの順番で検索します。

- 1 `-I` オプションで指定されたディレクトリ
- 2 現在のディレクトリ

## エラーの表示

ユーザ定義のテストの場合などに、アセンブラにエラーを発生させるには `#error` を使用します。

## #pragma の無視

`#pragma` 行は、C およびアセンブラと共通するヘッダファイルを使用しやすくなるように、アセンブラにより無視されます。

## C 形式のプリプロセッサディレクティブのコメント

`define` 文内でコメントを作成するには、以下のようになります。

- C コメントデリミタ `/* ... */` を使用して、セクションをコメント化します。
- C++ コメントデリミタ `//` を使用し、行の残りの部分をコメント化します。

予期せぬ動作を起こすため、`define` 文内でアセンブラコメントを使用しないでください。

以下の式は、3 に評価されます。これは、コメント文字が `#define` により保持されるためです。

```
#define x 3      ; コメント
exp EQU x*8+5
```

下記の例は、アセンブラコメントを C 形式プリプロセッサで使用するときに発生することがある問題を示しています。

```
#define five 5      ; このコメントは NG
#define six 6       // このコメントは OK
#define seven 7     /* このコメントは OK */

DC32 five,11,12
; Expands to "DC32 5      ; このコメントは NG"

DC32 six + seven,11,12 ; OK
; Expanded to "DC32 6+7,11,12"
```

## 例

### 条件ディレクティブの使用

次の例では `#ifdef` を使用して特定のシンボルが定義されているかどうかをチェックし、されている場合には 2 つの内部定義シンボルを生成しています。そうでない場合には、同じシンボルが `EXTERN` と宣言され、`#message` によってメッセージが表示されます。`STAND_ALONE` シンボルは、たとえば、`-D` オプションを使用してコマンドライン上で定義できます。`-D`、ページ 18 を参照してください。

```

        PROGRAM target
        SECTION MYCODE : CODE   (2)
        CODE32
        PUBLIC main
#ifdef  STAND_ALONE
alpha   EQU  0x20
beta    EQU  0x22
#else

        EXTERN alpha, beta
#message "Program depends on additional link information"
#endif

main:
        MOV  R1,#alpha
        MOV  R2,#beta
        ADD  R2,R2,R1
        EOR  R1,R1,R2 ; R1 = (alpha XOR (alpha + beta))

```

### ソースファイルのインクルード

次の例では、`#include` を使用して、ソースファイルにマクロを定義しているファイルをインクルードしています。たとえば、次のマクロがファイル `macros.s` に定義されているとします。

```

; c を一時バッファに使用して a と b を入れ替える
xch    MACRO  a,b, c
        MOV  c,a
        MOV  a,b
        MOV  b,c
        ENDM

```

このマクロ定義は、`#include` を使用してインクルードできます。

```

        NAME   include
        SECTION MYCODE : CODE   (2)
        CODE32
; 標準マクロ定義
#include "macros.s"

```

```
; プログラム
main:
    xch R0,R1,R2
```

## データ定義または割り当てのディレクティブ

これらのディレクティブは値を定義するか、またはメモリを予約します。以下の表のエイリアス列は、IAR システムズのディレクティブに対応する Advanced RISC Machines Ltd ディレクティブを示しています。式でディレクティブを使用するときに適用される制限事項については、式に関する制限事項、ページ 12 を参照してください。

ディレクティブ エイリアス		説明
DC8	DCB	文字列を含む、バイト定数を生成します。
DC16	DCW	16 ビット定数を生成します。
DC24		24 ビット定数を生成します。
DC32	DCD	32 ビット定数を生成します。
DF32		浮動小数点（32 ビット）定数を生成します。
DF64		浮動小数点（64 ビット）定数を生成します。
DS8	DS	8 ビット整数の空間を割り当てます。
DS16		16 ビット整数の空間を割り当てます。
DS24		24 ビット整数の空間を割り当てます。
DS32		32 ビット整数の空間を割り当てます。

表 26: データ定義または割り当てのディレクティブ

### 構文

```
DC8 expr [, expr] ...
DC16 expr [, expr] ...
DC24 expr [, expr] ...
DC32 expr [, expr] ...
DCB expr [, expr] ...
DCD expr [, expr] ...
DCW expr [, expr] ...
DF32 value [, value] ...
DF64 value [, value] ...
DS8 count
DS16 count
DS24 count
DS32 count
```

## パラメータ

<i>count</i>	予約される要素数を指定する有効な絶対式です。
<i>expr</i>	有効な絶対式、再配置可能式、外部式、または ASCII 文字列です。ASCII 文字列は、ディレクティブで示されるデータサイズの倍数になるまでゼロで埋められ、二重引用符で囲まれた文字列は、ゼロ終端されます。*
<i>value</i>	有効な絶対式または浮動小数点定数です。

\*DC64 では、*expr* は再配置可能式または外部式にはできません。

## 説明

DC8、DC16、DC24、DC32、DCB、DCD、DCW、DF32、DF64 を使用して定数を作成すると、その定数に十分な大きさのバイト領域が予約されます。

初期化されていないバイト領域を予約するには、DS8、DS16、DS24、DS32 を使用します。

## 例

### ルックアップテーブルの生成

次の例は、ルーチンアドレスのルックアップテーブルを生成します。

```

ADD_SELECTOR DEFINE 0
SUB_SELECTOR DEFINE 4
DIV_SELECTOR DEFINE 8
MUL_SELECTOR DEFINE 12
    NAME table
    SECTION VECTORS :CODE (2)
    CODE32
    EXTERN add_f, sub_f, div_f, mul_f
start B main          ; RESET vector
    SECTION MYDATA :DATA (2)
    DATA
                                ; 浮動小数点ルーチンへのポインタ

table: DC32 add_f, sub_f, div_f, mul_f
PI:    DC32 3.1415927
radius: DC32 1.3e2
    SECTION MYCODE :CODE (2)
    CODE32
main:
    LDR R1, =PI      ; PI のアドレスを得る
    LDR R2, =radius  ; 同じく radius...
    LDR R0, =table   ; テーブルのアドレスを R0 に置く

```

```
LDR R0, [R0, #MUL_SELECTOR] ; テーブルから mul_f のアドレスを取り出す
MOV PC, R0 ; mul_f にジャンプする
```

文字列の定義

次の例では文字列を定義しています。

```
myMsg DC8 'Please enter your name'
```

次の例では、最後にゼロを含む文字列を定義しています。

```
myCstr DC8 "This is a string."
```

文字列中に一重引用符を含めるには、次の例のように引用符を 2 つ記述します。

```
errMsg DC8 'Don't understand!'
```

空間の予約

次の例では 10 バイトの空間を予約します。

```
table DS8 10
```

アセンブラ制御ディレクティブ

これらのディレクティブはアセンブラの動作を制御します。式でディレクティブを使用するときに適用される制限事項については、式に関する制限事項、ページ 12 を参照してください。

ディレクティブ	説明	式に関する制限事項
\$	ファイルをインクルードします。	
/*comment*/	C 形式のコメント区切り文字です。	
//	C++形式のコメント区切り文字です。	
CASEOFF	大文字 / 小文字を区別しないように設定します。	
CASEON	大文字 / 小文字を区別するように設定します。	
INCLUDE	ファイルをインクルードします。	
LTORG	リテラルプールをディレクティブの直後にアセンブルするように指示します。	
RADIX	すべての数値のデフォルトのベースを設定します。	前方参照なし 外部参照なし 絶対式 固定

表 27: アセンブラ制御のディレクティブ

## 構文

```
$filename  
/*comment*/  
//comment  
CASEOFF  
CASEON  
INCLUDE filename  
LTORG  
RADIX expr
```

## パラメータ

<i>comment</i>	アセンブラによって無視されるコメント
<i>expr</i>	デフォルトの基数です。デフォルトは 10（10 進数）です。
<i>filename</i>	インクルードするファイルの名称です。行の最初の文字は \$ でなければなりません。

## 説明

ファイルの内容を、ソースファイル中の指定した箇所に挿入するには \$ を使用します。

アセンブラリストのセクションにコメントするには */\*...\*/* を使用します。

行に *//* を使用すると、それ以降がコメントになります。

定数のデフォルトの基数を設定するには、RADIX を使用します。デフォルトの基数は 10 です。

リテラルプールがアセンブルされるよう指示するには LTORG を使用します。デフォルトでは、END と RSEG ディレクティブごとに行われます。

## 大文字 / 小文字の区別の制御

ユーザ定義のシンボルについて、大文字 / 小文字の区別をオン / オフするには、CASEON または CASEOFF を使用します。デフォルトでは、大文字と小文字が区別されません。

CASEOFF が有効なときは、すべてのシンボルは大文字で格納され、また ILINK が対象とするすべてのシンボルは ILINK 定義ファイル中において大文字で書かれている必要があります。

## 例

### ソースファイルのインクルード

次の例では、マクロを定義するファイルをソースファイルにインクルードするため、\$を使用しています。たとえば、次のマクロがファイル `macros.s` に定義されているとします。

```
; exchange a and b using c as temporary
xch    MACRO a,b, c
        MOV c, a
        MOV a, b
        MOV b, c
        ENDM
```

マクロ定義は次のように \$ ディレクティブによってインクルードできます。

```
NAME    include
SECTION MYCODE : CODE (2)
CODE32
; standard macro definitions
$macros.s

; program
main:
    xch R0,R1,R2
```

### コメントの定義

次の例では、`/*...*/` を複数行に渡るコメントに使用しています。

```
/*
Program to read serial input.
Version 5: 19.2.06
Author:mjp
*/
```

*C 形式のプリプロセッサディレクティブのコメント*、ページ 75 も参照してください。

### 基数の変更

次の例ではデフォルトの基数を 16 に変更しています。

```
SECTION MYCODE : CODE (2)
CODE32
RADIX 16D
MOV R0,#12
```

これにより直後の引数は `0x12` として解釈されます。

大文字 / 小文字の区別の制御

CASEOFF が設定されているときには、label と LABEL は次の例に示すように同じものとして扱われます。

```
label    NOP           ; Stored as "LABEL"
        BL      LABEL
        NOP
```

次の例ではラベルの重複エラーが発生します。

```
CASEOFF

label    NOP
LABEL    NOP           ; Error, "LABEL" already defined
```

呼び出しフレーム情報ディレクティブ

これらのディレクティブはバックトレース情報をアセンブラソースコードで定義するのに使用します。これには、アセンブラコードをデバッグするときに呼出しフレームスタックを表示できるというメリットがあります。

ディレクティブ	説明
CFI BASEADDRESS	ベースアドレス CFA （Canonical Frame Address）の宣言です。
CFI BLOCK	データブロックの開始です。
CFI CODEALIGN	コードの境界整列を宣言します。
CFI COMMON	共通ブロックの開始または拡張です。
CFI CONDITIONAL	条件付スレッドのデータブロックの宣言です。
CFI DATAALIGN	データの境界整列の宣言です。
CFI ENDBLOCK	データブロックの終了です。
CFI ENDCOMMON	共通ブロックの終了です。
CFI ENDNAMES	ネームブロックの終了です。
CFI FRAMECELL	呼び出し側フレームにリファレンスを作成します。
CFI FUNCTION	データブロックと結び付く関数の宣言です。
CFI INVALID	無効なバックトレース情報の範囲の開始です。
CFI NAMES	ネームブロックの開始です。
CFI NOFUNCTION	関数に結び付かないデータブロックの宣言です。
CFI PICKER	ピッカースレッドのデータブロックの宣言です。
CFI REMEMBERSTATE	バックトレース情報状態を記憶します。
CFI RESOURCE	リソースを宣言します。

表 28: 呼び出しフレーム情報ディレクティブ



ディレクティブ	説明
CFI RESOURCEPARTS	複合リソースを宣言します。
CFI RESTORESTATE	保存済みバックトレース情報状態を復元します。
CFI RETURNADDRESS	リターンアドレスの列の宣言です。
CFI STACKFRAME	スタックフレーム CFA の宣言です。
CFI STATICOVERLAYFRAME	静的オーバーレイフレーム CFA の宣言です。
CFI VALID	無効なバックトレース情報の範囲の終了です。
CFI VIRTUALRESOURCE	仮想リソースの宣言です。
CFI <i>cfa</i>	CFA の値の宣言です。
CFI <i>resource</i>	リソースの値の宣言です。

表 28: 呼び出しフレーム情報ディレクティブ ( 続き )

## 構文

次の構文定義はそれぞれのディレクティブの構文を表しています。ディレクティブは使用法に基づいてグループ化されています。

## ネームブロックディレクティブ

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource :bits [, resource : bits] ...
CFI VIRTUALRESOURCE resource :bits [, resource : bits] ...
CFI RESOURCEPARTS resource part, part [, part] ...
CFI STACKFRAME cfa resource type [, cfa resource type] ...
CFI STATICOVERLAYFRAME cfa section [, cfa section] ...
CFI BASEADDRESS cfa type [, cfa type] ...
```

## 拡張ネームブロックディレクティブ

```
CFI NAMES name EXTENDS namesblock
CFI ENDNAMES name
CFI FRAMECELL cell cfa ÅoffsetÅ:size [, cell cfa (offset):size] ...
```

## 共通ブロックディレクティブ

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
```

```
CFI RETURNADDRESS resource type
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME (cfa, offset) }
CFI resource cfiexpr
```

## 拡張共通ブロックディレクティブ

```
CFI COMMON name EXTENDS commonblock USING namesblock
CFI ENDCOMMON name
```

## データブロックディレクティブ

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] ...
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME (cfa, offset) }
CFI resource cfiexpr
```

## パラメータ

<i>bits</i>	リソースのビット単位でのサイズ
<i>cell</i>	フレームセルの名前
<i>cfa</i>	CFA（canonical frame address）の名前
<i>cfiexpr</i>	CFI 式（CFI 式、ページ 91 を参照）
<i>codealignfactor</i>	すべての命令サイズの最小単位。データブロックの各 CFI ディレクティブは、このアラインメントに従って配置する必要があります。1 はデフォルト値で、常に使用できます。1 より大きい値を使用すると、生成されるバックトレース情報のサイズが縮小します。可能な範囲は 1 ～ 256 です。

<i>commonblock</i>	定義済みの共通ブロックの名前
<i>constant</i>	定数値または評価結果が定数値になるアセンブラ式
<i>dataalignfactor</i>	すべてのフレームサイズの最小単位。スタックが上位アドレスに向かって成長する場合、負の値になり、下位アドレスに向かって成長する場合、正の値になります。1 はデフォルト値です。1 より大きい値を使用すると、生成されるバックトレース情報のサイズが縮小します。可能範囲は -256 ~ -1 および 1 ~ 256 です。
<i>label</i>	関数ラベル
<i>name</i>	ブロックの名前
<i>namesblock</i>	定義済みネームブロックの名前
<i>offset</i>	CFA からの相対オフセット。符号がオプションの整数
<i>part</i>	複合リソースの一部。宣言済みリソースの名前
<i>resource</i>	リソースの名前
<i>section</i>	セクションの名前
<i>size</i>	フレームセルのサイズ（単位はバイト）
<i>type</i>	CODE、CONST、DATA のようなメモリタイプ。また、IAR ILINK リンカによりサポートされている任意のメモリタイプを使用することもできます。アドレス空間を示すためにだけ使用されます。

## 説明

呼び出しフレーム情報ディレクティブ（CFIディレクティブ）は、IAR C-SPY® デバッガのデバッグ情報形式の拡張です。CFIディレクティブはプログラムの命令のバックトレース情報を定義するために使用されます。コンパイラは通常この情報を生成しますが、ライブラリ関数や純粋なアセンブラ言語で書かれた他のコードの場合、デバッガの中で呼び出しフレームスタックを使用するとき、バックトレース情報を追加する必要があります。

バックトレース情報はレジスタやメモリセルのようなアセンブラコードのリソースの内容を追跡するために使用されます。IAR C-SPY デバッガが呼び出しスタックで戻って行くためにこの情報を使い、関数を実行する前のレジスタや他のリソースの正しい値を示します。従来とは異なり、デバッガはブ

ブレークポイントに達するまで全速力で実行してブレークポイントで停止し、そこでバックトレース情報を検索することができます。この情報を使って任意の呼び出し関数のリソースの内容を確認することができます（呼び出しフレーム情報も持っているとは仮定）。

## バックトレースの行と列

デバッガがプログラムを中断できる各ロケーションで、バックトレース情報を参照できます。バックトレース情報は行と列で示されます。バックトレースの各行は1組の列から構成されています。それぞれの列は追跡される項目を表します。3種類の列があります。

- リソース列ではリソースの元の値を追跡できます。
- CFA（*canonical frame address*）列は関数フレームの先頭を追跡します。
- リターンアドレス列はリターンアドレスのロケーションを追跡します。

リターンアドレスが複数あっても、リターンアドレス列はただ1つで、CFA列も通常1つです。

## ネームブロックの定義

ネームブロックは、プロセッサが使用できるリソースを宣言するのに使用します。追跡できるリソースはすべてネームブロック内で定義されます。

次のディレクティブでネームブロックを開始および終了します。

```
CFI NAMES name
CFI ENDNAMES name
```

*name* はブロックの名前です。

一度に開くことができるネームブロックは1つのみです。

ネームブロック内で、リソース宣言、スタックフレーム宣言、静的オーバーレイ宣言、またはベースアドレス宣言の4つの異なる宣言ができます。

- リソースを宣言するには次のディレクティブの1つを使用します。

```
CFI RESOURCE resource :bits
CFI VIRTUALRESOURCE resource :bits
```

パラメータはリソースの名前とリソースのビットサイズです。プロセッサレジスタのような物理リソースとは異なり、仮想リソースは論理的な概念です。仮想リソースは通常、リターンアドレスに使用されます。

複数のリソースをコンマで区切って宣言することができます。

またリソースを最低 2 つの部分リソースから構成して複合リソースにすることができます。複合リソースの構成を宣言するには、次のようにディレクティブを使用します。

```
CFI RESOURCEPARTS resource part, part, ...
```

部分リソースをコンマで区切ります。リソースとその部分リソースは、上記の説明のようにリソースとしてあらかじめ宣言されている必要があります。

- スタックフレーム CFA を宣言するには、ディレクティブを次のように使用します。

```
CFI STACKFRAME cfa resource type
```

パラメータは、スタックフレーム CFA の名前、関連するリソースの名前（スタックポインタ）、セクションタイプ（アドレス空間の取得用）です。複数のスタックフレーム CFA はコンマで区切って宣言することができます。

呼び出しスタックで戻っていくときは、前の関数フレームの正しい値を得るために、スタックフレーム CFA の値が関連するスタックポインタリソースにコピーされます。

- 静的オーバーレイフレーム CFA を宣言するには、次のディレクティブを使用します。

```
CFI STATICOVERLAYFRAME cfa section
```

パラメータは、CFA の名前と関数の静的オーバーレイフレームがあるセクションの名前です。複数の静的オーバーレイフレーム CFA はコンマで区切って宣言することができます。

- ベースアドレス CFA を宣言するには、次のようにディレクティブを使用します。

```
CFI BASEADDRESS cfa type
```

パラメータは、CFA の名前とセクションタイプです。複数のベースアドレス CFA はコンマで区切って宣言することができます。

ベースアドレス CFA を使えば CFA を容易に扱うことができます。スタックフレーム CFA とは異なり、復元すべき関連スタックポインタリソースはありません。

## ネームブロックの拡張

既存のネームブロックを新しいリソースで拡張する必要が生ずる場合があります。常に拡張が必要になる場合としては、C or C++ 関数への出入りを処理するルーチンのように、それ自身以外の呼び出しフレームを操作するルーチンがある場合です。拡張ネームブロックは通常コンパイラ開発者が使用します。

既存のネームブロックは次のディレクティブを使用して拡張します。

```
CFI NAMES name EXTENDS namesblock
```

*namesblock* は既存のネームブロックの名前で、*name* は新規の拡張ブロックの名前です。拡張ブロックを次のディレクティブで終了する必要があります。

```
CFI ENDNAMES name
```

## 共通ブロックの定義

**共通ブロック**は、すべての追跡されるリソースの初期内容を宣言するのに使用します。通常は、使用される呼び出し規則ごとに 1 つの共通ブロックがあります。

次のディレクティブで共通ブロックを開始します。

```
CFI COMMON name USING namesblock
```

*name* は新しいブロックの名前で、*namesblock* は既に定義されているネームブロックの名前です。

次のディレクティブでリターンアドレス列を宣言します。

```
CFI RETURNADDRESS resource type
```

*resource* は *namesblock* で定義されたリソースで、*type* はセクションタイプです。共通ブロックでのリターンアドレス列を宣言する必要があります。

次のディレクティブで共通ブロックを終了します。

```
CFI ENDCOMMON name
```

*name* は共通ブロックを開始するために使用した名前です。

共通ブロック内で、**共通ブロックディレクティブ**、ページ 83 の最後に記載したディレクティブを使用して CFA またはリソースの初期値を宣言することができます。ディレクティブに関する詳細は**単純ルール**、ページ 89 と **CFI 式**、ページ 91 を参照してください。

## 共通ブロックの拡張

ネームブロックは新しいリソースで拡張できるので、それらの新しいリソースの初期値を記述する手段が必要になります。このため、別の共通ブロックの宣言を含みながら、これら新しいリソースの初期値を正式に宣言して共通ブロックを拡張することもできるようになっています。拡張ネームブロックと同様に、拡張共通ブロックは通常コンパイラ開発者が使用します。

次のディレクティブで既存の共通ブロックを拡張します。

```
CFI COMMON name EXTENDS commonblock USING namesblock
```

*name* は新しい拡張ブロックの名前、*commonblock* は既存の共通ブロックの名前、*namesblock* は既に定義されているネームブロックの名前です。拡張ブロックを次のディレクティブで終了する必要があります。

```
CFI ENDCOMMON name
```

## データブロックの定義

**データブロック**には、一連のコードに対する実際の追跡情報が含まれていません。セクション制御ディレクティブが、データブロック内に存在してはなりません。

次のディレクティブでデータブロックを開始します。

```
CFI BLOCK name USING commonblock
```

*name* は新しいブロックの名前、*commonblock* は既に定義されている共通ブロックの名前です。

一連のコードが定義済み関数の一部である場合、次のようにディレクティブを使用して関数の名前を指定します。

```
CFI FUNCTION label
```

*label* は関数を開始するコードラベルです。

一連のコードが関数の一部でない場合は、次のようにディレクティブを使用してこれを指定します。

```
CFI NOFUNCTION
```

次のようにディレクティブを使用してデータブロックを終了します。

```
CFI ENDBLOCK name
```

*name* はデータブロックを開始するのに使用した名前です。

データブロック内で、データブロックディレクティブ、ページ 84 の最後に記載されたディレクティブを使用して列の値を操作することができます。ディレクティブに関する詳細は単純ルール、ページ 89 と CFI 式、ページ 91 を参照してください。

## 単純ルール

個々の列の追跡情報を記述するために、次に示すように、特殊構文で記述された単純なルールがあります。

```
CFI cfa { NOTUSED | USED }
```

```
CFI cfa { resource | resource + constant | resource - constant }
```

```
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
```

```
CFI resource { resource | FRAME (cfa, offset) }
```

これらの単純ルールは、共通ブロック中でリソースと CFA の初期情報を記述するためにも、また、データブロック内でリソースまたは CFA の情報変更を記述するためにも使用できます。

単純ルールでは記述できない場合がまれにあります。その場合は、完全な CFI 式を使って情報を記述することができます (CFI 式、ページ 91 を参照)。ただし、可能な限り CFI 式の代わりに単純ルールを使用するようにしてください。

単純ルールにはリソース用と CFA 用の 2 種類があります。

## リソースの単純ルール

リソースの単純ルールでは、1つの呼び出しフレームから戻る時にリソースを検索する場所を論理的に記述します。このため、CFI ディレクティブのリソース名の次の項目はリソースのロケーションとみなされます。

追跡されるリソースが復元される（つまり既に正しく配置されている）ことを宣言するには、ロケーションとして SAMEVALUE を使用します。概念上では、既に正しい値が格納されているため、リソースを復元する必要がないことを宣言しています。たとえば、REG レジスタが同じ値に復元されていることを宣言するには次のディレクティブを使用します。

```
CFI REG SAMEVALUE
```

リソースが追跡されないことを宣言するには、ロケーションとして UNDEFINED を使用します。これは、追跡されないため（呼び出しフレームから戻る時）リソースを復元する必要がないことを宣言します。これを使用するのは通常、リソースの初期ロケーションを宣言するときだけです。たとえば、REG がスクラッチレジスタで、復元の必要がないことを宣言するには、次のディレクティブを使用します。

```
CFI REG UNDEFINED
```

リソースが一時的に他のリソースに格納されていることを宣言するには、そのロケーションとしてリソース名を使用します。たとえば、REG1 レジスタが一時的に REG2 レジスタに配置（したがって、復元はこのレジスタからする）されていることを宣言するには、次のディレクティブを使用します。

```
CFI REG1 REG2
```

リソースが現在スタック上に配置されていることを宣言するには、リソースのロケーションとして FRAME (*cfa*, *offset*) を使用します。ここで、*cfa* はフレームポインタとして使用するための CFA 識別子、*offset* は CFA からの相対オフセット値です。たとえば、REG レジスタが CFA\_SP フレームポインタから数えて -4 のオフセットに配置されていることを宣言するには、次のディレクティブを使用します。

```
CFI REG FRAME (CFA_SP, -4)
```

複合リソースでは、別にもう 1 つの CONCAT というロケーションがあります。これは、リソースのロケーションが、複合リソースの部分リソースを連結することで得られることを宣言しています。たとえば、部分リソースの RETLO と RETHI からなる複合リソースを RET とします。部分リソースを調べ、それらを連結することで RET の値が得られることを宣言するには次のディレクティブを使用します。

```
CFI RET CONCAT
```

この場合、少なくとも 1 つの部分リソースが、上記で説明したルールを使用して定義されている必要があります。



## CFA の単純ルール

リソースの単純ルールとは異なり、CFA のルールは呼び出しフレームの開始アドレスを記述します。呼び出しフレームには、サブルーチン呼出し命令によってプッシュされたリターンアドレスがしばしば含まれます。CFA ルールは、現在の呼び出しフレームの開始アドレスを計算する方法を記述します。CFA 形式にはスタックフレームと静的オーバーレイフレームの 2 種類があり、いずれも関連するネームブロックで宣言されます。ネームブロックディレクティブ、ページ 83 を参照してください。

各スタックフレーム CFA はスタックポインタのようなリソースに関連しています。1 つの呼び出しフレームから戻る時、その関連リソースが現在の CFA に復元されます。スタックフレーム CFA に対しては 2 つの単純ルールがあります。リソースからのオフセット（スタックフレーム CFA に関連していないリソースでもよい）、または NOTUSED の 2 つです。

CFA が使用されず、その関連リソースが通常のリソースとして追跡されることを宣言するには、CFA のアドレスとして NOTUSED を使用します。たとえば、CFA\_SP という名前の CFA をこのコードブロックでは使用しないことを宣言するには、次のディレクティブを使用します。

```
CFI CFA_SP NOTUSED
```

CFA がリソースの値に対して相対オフセットのアドレスを持っていることを宣言するには、リソースとオフセットを指定します。たとえば、CFA\_SP という名前の CFA が、SP リソースの値に 4 を加算して取得できることを宣言するには、次のディレクティブを使用します。

```
CFI CFA_SP SP + 4
```

静的オーバーレイフレーム CFA では、共通ブロックとデータブロック内で使用できる宣言は USED と NOTUSED の 2 つです。

## CFI 式

リソースと CFA の単純ルールでは十分記述できない場合は、CFI（Call Frame Information: 呼び出しフレーム情報）式を使用できます。ただし、可能な限り単純ルールを使用するようにしてください。

CFI 式はオペランドと演算子から構成されています。CFI 式で使用する演算子は、以下に示すものだけです。大部分は、通常のアセンブラ式で使われているものと同じです。

オペランド記述では、*cfiexpr* は次のいずれかを表します。

- オペランド付 CFI 演算子
- 数値定数
- CFA 名
- リソース名

単項演算子

全体構文：演算子（オペランド）

演算子	オペランド	説明
COMPLEMENT	<i>cfiexpr</i>	ビットごとの CFI 式否定
LITERAL	<i>expr</i>	アセンブラ式の値の取得。これを使えば CFI 式に通常アセンブラ式の値を代入できます。
:NOT:	<i>cfiexpr</i>	CFI 式の論理否定
UMINUS	<i>cfiexpr</i>	CFI 式の算術否定

表 29: CFI 式の単項演算子

2 項演算子

全体構文：演算子（オペランド 1、オペランド 2）

演算子	オペランド	説明
ADD	<i>cfiexpr, cfiexpr</i>	加算
AND	<i>cfiexpr, cfiexpr</i>	ビットごとの論理積
DIV	<i>cfiexpr, cfiexpr</i>	除算
EQ	<i>cfiexpr, cfiexpr</i>	等しい
GE	<i>cfiexpr, cfiexpr</i>	以上
GT	<i>cfiexpr, cfiexpr</i>	大なり
LE	<i>cfiexpr, cfiexpr</i>	以下
LSHIFT	<i>cfiexpr, cfiexpr</i>	左のオペランドを左に論理シフトします。シフトのビット数は右のオペランドで指定されます。符号ビットは、シフト時に維持されません。
LT	<i>cfiexpr, cfiexpr</i>	小なり
MOD	<i>cfiexpr, cfiexpr</i>	剰余
MUL	<i>cfiexpr, cfiexpr</i>	乗算
NE	<i>cfiexpr, cfiexpr</i>	等しくない
OR	<i>cfiexpr, cfiexpr</i>	ビットごとの論理和
RSHIFTA	<i>cfiexpr, cfiexpr</i>	左のオペランドを右に算術シフトします。シフトのビット数は右のオペランドで指定されます。RSHIFTL とは異なり、符号ビットはシフト時に維持されます。
RSHIFTL	<i>cfiexpr, cfiexpr</i>	左のオペランドを右に論理シフトします。シフトのビット数は右のオペランドで指定されます。符号ビットは、シフト時に維持されません。

表 30: CFI 式の 2 項演算子

演算子	オペランド	説明
SUB	<i>cfiexpr, cfiexpr</i>	減算
XOR	<i>cfiexpr, cfiexpr</i>	ビットごとの排他的論理和

表 30: CFI 式の 2 項演算子 ( 続き )

### 3 項演算子

全体構文: 演算子 (オペランド 1、オペランド 2、オペランド 3)

演算子	オペランド	説明
FRAME	<i>cfa, size, offset</i>	<p>スタックフレームから値を取得します。オペランドは次のとおりです。</p> <p><i>cfa</i> 宣言済み CFA を示す識別子  <i>size</i> バイト単位でサイズを示す定数式  <i>offset</i> バイト単位でオフセットを示す定数式</p> <p>アドレス <i>cfa+offset</i> の値 (サイズ <i>size</i>) を取得します。</p>
IF	<i>cond, true, false</i>	<p>条件付演算子です。オペランドは次のとおりです。</p> <p><i>cond</i> 条件を示す CFA 式  <i>true</i> 任意の CFA 式  <i>false</i> 任意の CFA 式</p> <p>条件式がゼロでない場合は、結果は <i>true</i> 式の値になります。そうでない場合は <i>false</i> 式の値になります。</p>
LOAD	<i>size, type, addr</i>	<p>メモリから値を取得します。オペランドは次のとおりです。</p> <p><i>size</i> バイト単位でサイズを示す定数式  <i>type</i> メモリタイプ  <i>addr</i> メモリアドレスを示す CFA 式</p> <p>セクションタイプ <i>type</i> のアドレス <i>addr</i> における値 (サイズ <i>size</i>) を取得します。</p>

表 31: CFI 式の 3 項演算子

### 例

次に示す例は一般的なもので、ARM core 固有の例ではありません。これは単純化した例で、CFI ディレクティブの使用方法を分かりやすく示したものです。ターゲットに固有な例は、C ソースファイルをコンパイルするときに、アセンブラ出力を生成すれば得られます。

スタックポインタ SP および 2 つのレジスタ R0 と R1 がある一般的なプロセッサを例に考えます。R0 レジスタはスクラッチレジスタ（関数呼び出しで破壊される）として使用されます。R1 レジスタは関数呼び出し後に復元する必要があります。単純化するために、命令、レジスタ、アドレスはすべて 16 ビット幅とします。

次のように小さいコードと関連するバックトレースの行と列を考えます。開始時点で、スタックには 16 ビットリターンアドレスが入っていると仮定します。スタックは高アドレスからゼロに向かって増えていきます。CFA は呼び出しフレームの先頭を示し、関数から戻った後のスタックポインタの値になります。

アドレス	CFA	SP	R0	R1	RET	アセンブラコード
0000	SP + 2		—	SAME	CFA - 2	func1: PUSH R1
0002	SP + 4			CFA - 4		MOV R1, #4
0004						CALL func2
0006						POP R0
0008	SP + 2			R0		MOV R1, R0
000A				SAME		RET

表 32: バックトレース行と列およびコード例

各バックトレース行は、命令を実行する *前の* 追跡されるリソースの状態を表しています。たとえば、MOV R1,R0 命令では、R1 レジスタの元の値は R0 レジスタに格納され、関数フレーム（CFA 列）の先頭は SP + 2 になっています。0000 アドレスのバックトレース行は初期行で、関数に適用された呼び出し規則の結果が現われています。

SP 列は、CFA がスタックポインタから定義されるため空になっています。RET 列はリターンアドレスの列で、リターンアドレスのロケーションになります。R0 列の最初の行に "—" がありますが、これは R0 の値が未定義なので関数から出る時に復元する必要がないことを示しています。R1 列の最初の行には SAME がありますが、これは R1 レジスタの値が既存の値と同じ値に復元されることを示しています。

ネームブロックの定義

上記の例に対するネームブロックは次のようになります。

```
CFI NAMES trivialNames
CFI RESOURCE SP:16, R0:16, R1:16
CFI STACKFRAME CFA SP DATA

;; The virtual resource for the return address column
CFI VIRTUALRESOURCE RET:16
CFI ENDNAMES trivialNames
```

## 共通ブロックの定義

上記の例に対する共通ブロックは次のようになります。

```
CFI COMMON trivialCommon USING trivialNames
CFI RETURNADDRESS RET DATA
CFI CFA SP + 2
CFI R0 UNDEFINED
CFI R1 SAMEVALUE
CFI RET FRAME (CFA, -2) ; FRAME の先頭から -2 のオフセット
CFI ENDCOMMON trivialCommon
```

**注記：**SP は CFA に関連するリソースなので、CFI ディレクティブを使用して変更することはできません。

## データブロックの定義

上記の例に対するデータブロックは次のようになります。

```

RSEG CODE:CODE
CFI BLOCK func1block USING trivialCommon
CFI FUNCTION func1
func1:
PUSH R1
CFI CFA SP + 4
CFI R1 FRAME (CFA, -4)
MOV R1, #4
CALL func2
POP R0
CFI R1 R0
CFI CFA SP + 2
MOV R1, R0
CFI R1 SAMEVALUE
RET
CFI ENDBLOCK func1block
```

CFI ディレクティブは、バックトレース情報に関連する命令の後に配置されるので注意してください。



# アセンブラ擬似命令

ARMIAR アセンブラではさまざまな擬似命令を使用することができ、正しいコードに変換されます。この章では擬似命令をリストアップし、使用方法の例を示します。

## 要約

以下の表および説明の意味を示します。

- ARM は、ARM ディレクティブ後で使用できる擬似命令を示します。
- CODE16 は、CODE16 ディレクティブ後で使用できる擬似命令を示します。
- THUMB は、THUMB ディレクティブ後で使用できる擬似命令を示します。

**注：** THUMB 擬似命令のプロパティは、使用するコアが Thumb-2 命令セットを持つかどうかにより異なります。

使用可能な擬似命令の概要を次表に示します。

擬似命令	ディレクティブ	変換結果	説明
ADR	ARM	ADD, SUB	プログラム相対アドレスをレジスタにロードします。
ADR	CODE16	ADD	プログラム相対アドレスをレジスタにロードします。
ADR	THUMB	ADD, SUB	プログラム相対アドレスをレジスタにロードします。
ADRL	ARM	ADD, SUB	プログラム相対アドレスをレジスタにロードします。
ADRL	THUMB	ADD, SUB	プログラム相対アドレスをレジスタにロードします。
LDR	ARM	MOV, MVN, LDR	32 ビット値をレジスタにロードします。
LDR	CODE16	MOV, LDR	32 ビット値をレジスタにロードします。
LDR	THUMB	MOV, MVN, LDR	32 ビット値をレジスタにロードします。

表 33: 擬似命令

擬似命令	ディレク ティブ	変換結果	説明
MOV	CODE16	ADD	R0-R7 のレジスタ値を他の R0-R7 のレジスタに移動します。
MOV32	THUMB	MOV, MOVT	32 ビット値をレジスタにロード します。
NOP	ARM	MOV	ARM の NOP コードを生成します。
NOP	CODE16	MOV	Thumb の NOP コードを生成し ます。

表 33: 擬似命令 ( 続き )

## 擬似命令の説明

このセクションは、それぞれの擬似命令についてのレファレンス情報です。

ADR (ARM) *ADR{condition} register, expression*

### パラメータ

<i>{condition}</i>	次のいずれかになります。EQ、NE、CS、CC、MI、PL、VS、VC、HI、LS、GE、LT、GT、LE、AL。
<i>register</i>	ロードするレジスタです。
<i>expression</i>	-247 ～ +263 バイトの範囲でワード整列されていないアドレス、または -1012 ～ +1028 バイトの範囲でワード整列されたアドレスとなる、プログラムロケーションカウンタ相対式です。未解決な式（たとえば外部ラベルまたは他のセクション中のラベルを含む式）は、-247 ～ +263 バイトの範囲になければなりません。

### 説明

ADR は常に 1 つの命令にアセンブルされます。アセンブラはアドレスをロードするため、1 つの ADD または SUB 命令を生成します。

```
SECTION MYCODE :CODE (2)
CODE32
ADR r0,thumb ; => ADD r0,pc,#1
BX r0
CODE16
thumb
```



---

ADR (CODE16)    ADR *register, expression*

### パラメータ

*register*            ロードするレジスタです。

*expression*        +4 ～ +1024 バイトの範囲でワード整列されたアドレスになる、プログラム相対式です。

### 説明

Thumb モードでは、ADR はワード整列されたアドレス（つまり 4 で割り切れるアドレス）のみを生成できます。アドレスが必ず整列されるようにするには ALIGNROM ディレクティブを使用します（ただし DC32 が使用された場合を除く。この場合は常にワード整列されます）。

```
SECTION MYCODE :CODE (2)
ADR r0,my_data ; => ADD r0,pc,#4
ADD r0,r0,r1
BX lr
DATA
ALIGNROM 2
my_data DC32 0xABCD19
END
```

---

ADR (THUMB)    ADR{*condition*} *register, expression*

### パラメータ

{*condition*}        この命令が IT 命令の後にある場合は、オプションの条件コードです。

*register*            ロードするレジスタです。

*expression*        -4095 ～ 4095 バイトの範囲でワード整列されたアドレスになる、プログラム相対式です。

### 説明

ADR (CODE16) と似ていますが、使用するアーキテクチャで 32 ビット Thumb-2 命令を使用できる場合、アドレス範囲は広がります。16 ビット Thumb 命令のみが使用可能な場合、ADR (CODE16)、ページ 99 を参照してください。

アドレスオフセットが正の数値で、アドレスがワード整列されている場合、デフォルトで、16 ビット ADR (CODE16) バージョンが生成されます。

16 ビットバージョンは、ADR.N 命令を使用して明示的に指定することができます。32 ビットバージョンは、ADR.W 命令を使用して明示的に指定することができます。

---

ADRL (ARM) *ADRL{condition} register, expression*

## パラメータ

<i>{condition}</i>	次のいずれかになります。EQ、NE、CS、CC、MI、PL、VS、VC、HI、LS、GE、LT、GT、LE、AL。
<i>register</i>	ロードするレジスタです。
<i>expression</i>	64 キロバイト以内のワード整列されていないアドレス、または 256 キロバイト以内のワード整列されたアドレスになる、レジスタ相対式です。未解決な式（たとえば外部ラベルまたは他のセクション中のラベルを含む式）は、64 キロバイト以内でなければなりません。このアドレスは命令アドレスの前後にすることができます。

## 説明

ADRL 擬似命令はプログラム相対アドレスをレジスタにロードします。これは ADR 擬似命令に似ています。ADRL は 2 つのデータ処理命令を生成するため、ADR よりも広い範囲のアドレスをロードします。ADRL は常に 2 つの命令にアセンブルします。1 つの命令によってアドレスに到達できる場合でも、もう 1 つの命令が重複して生成されます。2 つの命令によってもアセンブラがアドレスを構築できない場合、エラーメッセージが生成され、アセンブリは失敗します。

**注記：**ADRL は Thumb 命令をアセンブルするときには使用できません。これは ARM モードでのみ使用します。

## 例

```
SECTION MYCODE :CODE (2)
ADRL r1,my_data+0x2345 ; => ADD r1,pc,#0x45
                        ; => ADD r1,r1,#0x2300

DATA
my_data:DC32 0
END
```

---

ADRL (THUMB) `ADRL{condition} register, expression`

### パラメータ

<code>{condition}</code>	この命令が IT 命令の後にある場合は、オプションの条件コードです。
<code>register</code>	ロードするレジスタです。
<code>expression</code>	±1MB の範囲でワード整列されたアドレスになる、プログラム相対式です。

### 説明

ADRL (ARM) と似ていますが、アドレス範囲は広がります。この命令は、Thumb-2 命令セットをサポートするコアでのみ使用できます。

---

LDR (ARM) `LDR{condition} register, =expression1`

または

`LDR{condition} register, expression2`

### パラメータ

<code>condition</code>	オプションの条件コードです。
<code>register</code>	ロードされるレジスタです。
<code>expression1</code>	任意の 32 ビット式です。
<code>expression2</code>	プログラムロケーションカウンタから -4087 ~ +4103 の範囲内にあるプログラムロケーションカウンタ相対式です。

### 説明

最初の書式の LDR 擬似命令は、任意の 32 ビット式をレジスタにロードします。2 番目の書式の命令は、その式によって指定されたアドレスから 32 ビットの値を読み込みます。実際の LDR 命令と等しくなる場合もあることに注意してください。

`expression1` の値が MOV または MVN 命令の範囲内にある場合、アセンブラは適切な命令を生成します。`expression1` の値が MOV または MVN 命令の範囲内でない場合または `expression1` が未解決の場合には、アセンブラは定数をリテラルプールに入れ、その定数をリテラルプールから読み出すプログラム相対 LDR 命令を生成します。プログラムロケーションカウンタから定数へのオフセットは 4 キロバイト未満でなければなりません。詳細についてはアセンブラ制御ディレクティブ、ページ 79 のセクションにある、LTORG ディレクティブの項目も参照してください。

**例**

```

SECTION MYCODE :CODE (2)
LDR r1,=0x12345678 ; => LDR r1,[pc,#4]
; リテラルプールから 0x12345678 をロードし r1 に格納
LDR r2,my_data ; r2 に 0xFFEEDDCC をロード
; => LDR r2,[pc,#-4]
DATA
my_data DC32 0xFFEEDDCC
LTORG
END

```

---

LDR (CODE16) LDR *register*, =*expression1*

または

LDR *register*, *expression2*

**パラメータ**

<i>register</i>	ロードされるレジスタです。LDR は下位のレジスタ (R0-R7) にのみアクセス可能です。
<i>expression1</i>	任意の 32 ビット式です。
<i>expression2</i>	プログラムロケーションカウンタから +4 ~ +1024 の範囲内にあるプログラムロケーションカウンタ相対式です。

**説明**

ARM モードの場合と同様、Thumb モードにおける最初の書式の LDR 擬似命令は、任意の 32 ビット式をレジスタにロードします。2 番目の書式の命令は、その式によって指定されたアドレスから 32 ビットの値を読み込みます。ただし、プログラムロケーションカウンタから定数までのオフセットは 1 キロバイト未満の正の値でなければなりません。

**例**

```

EXTERN ext_label
SECTION MYCODE :CODE (2)
LDR r1,=ext_label ; => LDR r1,[pc,#8]
; リテラルプールから ext_label をロードし r1 に格納
NOP
LDR r2,my_data ; r2 に 0xFFEEDDCC をロード
NOP ; => LDR r2,[pc,#0]
DATA
my_data DC32 0xFFEEDDCC
LTORG
END

```

---

LDR (THUMB) LDR{*condition*} *register*,=*expression*

## パラメータ

<i>condition</i>	この命令が IT 命令の後にある場合は、オプションの条件コードです。
<i>register</i>	ロードされるレジスタです。
<i>expression</i>	任意の 32 ビット式です。

## 説明

LDR (CODE16) 命令と似ていますが、32 ビット命令を使用すると、定数をリテラルプールに入れずに、MOV または MVN 命令でより大きな値を直接ロードできます。

16 ビット Thumb 命令のみが使用可能な場合、LDR (CODE16)、ページ 102 を参照してください。

LDR.N 命令を使用して 16 ビットバージョンを明示的に指定することで、16 ビット命令が常に生成されます。この場合、32 ビット命令が MOV または MVN を使用して値を直接ロードできたとしても、定数がリテラルプールに入れられることがあります。

LDR.W 命令を使用して 32 ビットバージョンを明示的に指定することで、32 ビット命令が常に生成されます。

.N または .W のいずれも指定しない場合、Rd が R8 ～ R15 (この場合 32 ビット派生型が生成されます) でない限り、16 ビット LDR (CODE16) 命令が生成されます。

**注：** 構文 LDR{*condition*} *register*, *expression2* は、LDR (ARM) および LDR (CODE16) で説明されているように、擬似命令とはみなされません。これは、Advanced RISC Machines Ltd. の Unified Assembler 構文で指定されているように通常の命令の一部です。

MOV (CODE16) MOV Rd, Rs

パラメータ

Rd                    移動先のレジスタです。  
Rs                    移動元のレジスタです。

説明

Thumb MOV 擬似命令は下位レジスタの値を、別の下位レジスタ（R0-R7）に移動します。Thumb MOV 命令は、値を下位レジスタから別の下位レジスタへ移すことはできません。

**注記：**アセンブラによって生成された ADD 即値命令では、条件コードが更新される副作用があります。

MOV 擬似命令は即値ゼロで ADD 即値命令を使用します。

**注：**この説明は、CODE16 デイレクティブを使用する場合にのみ適用されます。THUMB デイレクティブの後、命令構文の解釈は、Advanced RISC Machines Ltd. の Undefined Assembler 構文で定義されます。

例

MOV r2,r3 ; ADD r2,r3,#0 を生成

MOV32 (THUMB) MOV32{condition} register,expression

パラメータ

condition            この命令が IT 命令の後にある場合は、オプションの条件コードです。  
register              ロードされるレジスタです。  
expression            任意の 32 ビット式です。

説明

LDR (THUMB) 命令と似ていますが、MOV (MOVW) と MOVT 命令のペアを生成することで定数をロードします。

この擬似命令は、常に 2 つの 32 ビット命令を生成し、Thumb-2 命令セットをサポートするコアでのみ使用できます。

---

NOP (ARM) NOP

### 説明

NOP は次のように ARM のノーオペレーションコードを生成します。

MOV r0,r0

---

NOP (CODE16) NOP

### 説明

NOP は次のように Thumb のノーオペレーションコードを生成します。

MOV r8,r8





# アセンブラの診断

この章では診断メッセージのフォーマット、および診断メッセージの重大度による分類方法を説明します。

---

## メッセージフォーマット

すべての診断メッセージは、オプションのリストファイルに出力されるとともに、画面に表示されます。

メッセージはすべて、説明を要しない完結型のメッセージとして出力されます。このメッセージでは、正しくないソース行が示されます。また、問題が検出された場所へのポインタ、その後にソース行番号と診断メッセージが続きます。インクルードファイルが使用されている場合、エラーメッセージの前には、ソース行番号と現在のファイルが示されます。

```
          ADS      B,C
-----^
"subfile.h",4  Error[40]: bad instruction
```

---

## 重大度

ARM IAR アセンブラが生成する診断メッセージには、ソースコード上に存在する、もしくはアセンブリ時に発生する問題やエラーが表示されます。

### 診断オプション

診断のためのアセンブラオプションには、以下の2種類があります。

- すべてのワーニング、ワーニングの一部の範囲、個々のワーニングを無効または有効にします (-w、ページ 26 を参照してください)。
- コンパイルを停止する最大エラー数を設定します (-E、ページ 19 を参照してください)。

### アセンブリ時のワーニングメッセージ

アセンブリ時のワーニングメッセージは、プログラミングのエラーや脱落によって生じたと思われる構文をアセンブラが検出したときに生成されます。

### コマンドラインエラーのメッセージ

コマンドラインのエラーは、アセンブラが不適切なパラメータで呼び出された場合に発生します。よくある状況としてはファイルを開けない、あるいはコマンドラインが重複している、スペルミスがある、またはコマンドラインオプションが見当たらないなどがあります。

## アセンブリ時のエラーメッセージ

アセンブリ時のエラーメッセージは、アセンブラが文法違反を構文中に見つけたときに生成されます。

## アセンブリ時の致命的なエラーメッセージ

アセンブリ時の致命的なエラーメッセージは、ソースをそれ以上処理することが無意味とみなされるほど重大なユーザーエラーがアセンブラで見つかったときに生成されます。診断メッセージが出力された後、アセンブリは直ちに中止されます。これらのエラーメッセージは、エラーメッセージリストで **Fatal** と示されます。

## アセンブラの内部エラーメッセージ

インターナルエラーは、アセンブラでの問題が原因で、重大かつ予期しない障害が発生したことを示す診断メッセージです。

アセンブリ時には一連の一貫性チェックが内部的に行われ、いずれかのチェックに不合格となった場合には簡単な説明が出力された後、アセンブラは終了します。このようなエラーは通常発生しないはずですが、もし発生した場合にはソフトウェア販売会社または **IAR Technical Support** までご連絡いただくようお願いします。問題の再現に十分な情報も、合わせてお知らせください。この情報には、主に以下のものが含まれます。

- 製品名
- アセンブラのバージョン番号（アセンブラが生成するリストファイルのヘッダ部分を参照）
- ライセンス番号
- インターナルエラーメッセージ本文
- インターナルエラーの原因となったプログラムのソースファイル
- インターナルエラー発生時に指定していたオプションの一覧

# ARM IAR アセンブラへの移行

他のアセンブラ向けに作成されたアセンブラソースコードも、ARM IAR アセンブラに使用することができます。アセンブラオプションの `-j` を指定すると、別のさまざまなレジスタ名、ニーモニック、および演算子を使用できるようになります。

この章には、他の製品から ARM IAR アセンブラへの移行に役立つ情報が記載されています。

---

## 概要

ARM IAR アセンブラ (IASMARM) は他の IAR アセンブラとの共通点を持ちながら、かつ Advanced RISC Machines Ltd の ARMASM 向けに作成されたソースコードを容易に変換できるよう設計されています。

オプション `-j` (別のレジスタ名、ニーモニック、およびオペランドの使用) が選択されているとき、IASMARM の命令の構文は ARMASM と同一になります。ただしディレクティブやマクロなど多数の機能には互換性がなく、構文エラーを引き起こします。また Thumb コードのラベルにも違いがあり、エラーやワーニングを生成しない障害が発生することがあります。ジャンプラベル以外の状況でこのようなラベルを使用するときには、特に慎重に行ってください。

**注：**新しいコードに関しては ARM IAR アセンブラのレジスタ名、ニーモニック、および演算子を使用してください。

## THUMB コードのラベル

Thumb コード中に配置されたラベル、すなわち CODE16 ディレクティブの後にあられるものには、常に IASMARM 内で bit 0 が 1 にセットされます (つまり奇数ラベル)。これに比べ ARMASM では、アセンブリ時に解決される式中のシンボルには bit 0 は 1 にセットされません。次の例では、シンボル `α` はローカルであり、Thumb コード内に配置されます。これには、IASMARM によってアセンブルされるときに bit 0 が 1 にセットされますが、ARMASM によってアセンブルされるときには 1 にセットされません (ただし DCD については再配置可能セクションがリンク時に解決されるため例外です)。したがってアセンブルされた結果の命令も異なります。

例

```
SECTION MYCODE :CODE (2)
CODE32
ADR R0,T+1
MOV R1,#T-.
DD DATA
DCD T
CODE16
T NOP
```

このような命令を移植可能とするため（すなわち IASMARM と ARMASM のどちらを使用してアセンブルされても同じ結果が得られるよう）命令を書き換えます。ADR は PC との ADD と同等であることに注意してください。

```
SECTION MYCODE :CODE (2)
CODE32
ADD R0,PC,#(T--8) :OR: 1
MOV R1,#(T-.):AND:0xFFFFFFFF
DD DATA
DCD T
CODE16
T NOP
```

代替レジスタ名

オプション -j が選択されているとき、ARM IAR アセンブラは他のアセンブラに使用されるレジスタ名を変換します。これらの代替レジスタ名は、ARM と Thumb の両方のモードで使用できます。代替レジスタ名と ARM IAR アセンブラのレジスタ名を次表に示します。

代替レジスタ	ARM IAR アセンブラのレジスタ名
A1	R0
A2	R1
A3	R2
A4	R3
V1	R4
V2	R5
V3	R6
V4	R7
V5	R8

表 34: 代替レジスタ名一覧

代替レジスタ	ARM IAR アセンブラのレジスタ名
V6	R9
V7	R10
SB	R9
SL	R10
FP	R11
IP	R12

表 34: 代替レジスタ名一覧 ( 続き )

レジスタの詳細については、レジスタシンボル、ページ 9 を参照してください。

## 代替ニーモニック

オプション -j が指定されているとき、他のアセンブラが使用するニーモニックの多くが ARM ARMIAR アセンブラによって変換されます。これらの代替ニーモニックは CODE16 モードでのみ使用できます。代替ニーモニックを次表に示します。

代替ニーモニック	ARM IAR アセンブラのニーモニック
ADCS	ADC
ADDS	ADD
ANDS	AND
ASLS	LSL
ASRS	ASR
BICS	BIC
BNCC	BCS
BNCS	BCC
BNEQ	BNE
BNGE	BLT
BNGT	BLE
BNHI	BLS
BNLE	BGT
BNLO	BCS
BNLS	BHI
BNLT	BGE
BNMI	BPL

表 35: 代替ニーモニック

代替ニーモニック	ARM IAR アセンブラのニーモニック
BNNE	BEQ
BNPL	BMI
BNVC	BVS
BNVS	BVC
CMN{cond}S	CMN{cond}
CMP{cond}S	CMP{cond}
EORS	EOR
LSLS	LSL
LSRS	LSR
MOVS	MOV
MULS	MUL
MVNS	MVN
NEGS	NEG
ORRS	ORR
RORS	ROR
SBCS	SBC
SUBS	SUB
TEQ{cond}S	TEQ{cond}
TST{cond}S	TST{cond}

表 35: 代替ニーモニック ( 続き )

ニーモニックの詳細については『*ARM Architecture Reference Manual* (Prentice-Hall)』を参照してください。

## 演算子の同義語

オプション -j が指定されているとき、他のアセンブラに使用される演算子の多くは ARM IAR アセンブラによって変換されます。次表に示す演算子の同義語は ARM と Thumb の両方のモードで使用できます。

演算子の同義語	ARM IAR アセンブラの演算
:AND:	&
:EOR:	^
:LAND:	&&
:LEOR:	XOR

表 36: 演算子の同義語

演算子の同義語	ARM IAR アセンブラの演算
:LNOT:	!
:LOR:	
:MOD:	%
:NOT:	~
:OR:	
:SHL:	<<
:SHR:	>>

表 36: 演算子の同義語 ( 続き )

**注記：**ARM ARM IAR アセンブラの演算子と演算子の同義語では、優先順位のレベルが異なります。演算子の詳細な説明は、アセンブラ演算子、ページ 29 を参照してください。

## ワーニングメッセージ

オプション -j が指定されていない場合、代替的な名称が使用されたとき、またはオペランドの不正な組み合わせを検出したとき、ARM IAR アセンブラはワーニングメッセージを出力します。以降のセクションにワーニングメッセージをリストアップします。

### The first register operand omitted

3 つのオペランドを必要とし、その最初の 2 つがインデックス付きでないレジスタとなる命令 (ADD、SUB、LSL、LSR、ASR) から、最初のレジスタオペランドが欠落しています。

### The first register operand duplicated

最初のレジスタオペランドは操作に含まれるレジスタで、デスティネーションレジスタでもあります。

不正なコードの例

```
MUL R0, R0, R1
```

正しいコードの例

```
MUL R0, R1
```

### Immediate #0 omitted in Load/Store

ロード / ストア命令から即値 #0 が欠落しています。

不正なコードの例

```
LDR R0, [R1]
```

正しいコードの例

```
LDR R0, [R1, #0]
```



# A

AAPCS (アセンブラディレクティブ)	47
ADD (CFI 演算子)	92
ADD (アセンブラ命令)	98
ADR (ARM) (擬似命令)	98
ADR (CODE16) (擬似命令)	99
ADR (THUMB) (擬似命令)	99
ADRL (ARM) (擬似命令)	100
ADRL (THUMB) (擬似命令)	101
ALIAS (アセンブラディレクティブ)	57
ALIGNRAM (アセンブラディレクティブ)	54
ALIGNROM (アセンブラディレクティブ)	54
AND (CFI 演算子)	92
_args (定義済マクロシンボル)	63
ARM IAR アセンブラへの移行	109
演算子の同義語	112
ワーニングメッセージ	113
代替ニーモニック	111
代替レジスタ名	110
ARM (アセンブラディレクティブ)	52
ARMASM アセンブラ	109
_ARMVFP_ (定義済シンボル)	10
ARM アーキテクチャと命令セット	xi
ASCII 文字定数	7
asm (ファイル名の拡張子)	3
ASSIGN (アセンブラディレクティブ)	57

# B

-B (アセンブラオプション)	17
__BUILD_NUMBER__ (定義済シンボル)	10
BX (アセンブラ命令)	53
BYTE1 (アセンブラ演算子)	37
BYTE2 (アセンブラ演算子)	37
BYTE3 (アセンブラ演算子)	37
BYTE4 (アセンブラ演算子)	37

# C

-c (アセンブラオプション)	17
CASEOFF (アセンブラディレクティブ)	79
CASEON (アセンブラディレクティブ)	79
CFI ディレクティブ	82
CFI 演算子	92
CFI 式	91
CODE16 (アセンブラディレクティブ)	52
CODE32 (アセンブラディレクティブ)	52
COL (アセンブラディレクティブ)	68
COMPLEMENT (CFI 演算子)	92
--cpu (アセンブラオプション)	18
CPU、アセンブラでの定義→プロセッサの設定を参照	
CRC、アセンブラリストファイル	13
C 形式のプリプロセッサディレクティブ	72

# D

-D (アセンブラオプション)	18
DATA (アセンブラディレクティブ)	52
__DATE__ (定義済シンボル)	10
DATE (アセンブラ演算子)	38
DCB (アセンブラディレクティブ)	77
DCD (アセンブラディレクティブ)	77
DCW (アセンブラディレクティブ)	77
DC8 (アセンブラディレクティブ)	77
DC16 (アセンブラディレクティブ)	77
DC24 (アセンブラディレクティブ)	77
DC32 (アセンブラディレクティブ)	77
DEFINE (アセンブラディレクティブ)	57
DF64 (アセンブラディレクティブ)	77
DIV (CFI 演算子)	92
DS8 (アセンブラディレクティブ)	77
DS16 (アセンブラディレクティブ)	77
DS24 (アセンブラディレクティブ)	77
DS32 (アセンブラディレクティブ)	77

## E

-E (アセンブラオプション) .....	19
-e (アセンブラオプション) .....	19
ELSE (アセンブラディレクティブ) .....	58
ELSEIF (アセンブラディレクティブ) .....	58
END (アセンブラディレクティブ) .....	47
ENDIF (アセンブラディレクティブ) .....	58
ENDM (アセンブラディレクティブ) .....	60
ENDR (アセンブラディレクティブ) .....	60
EQ (CFI 演算子) .....	92
EQU (アセンブラディレクティブ) .....	57
EVEN (アセンブラディレクティブ) .....	54
EXITM (アセンブラディレクティブ) .....	60
EXTERN (アセンブラディレクティブ) .....	50
EXTRN (アセンブラディレクティブ) .....	50
EXTWEAK (アセンブラディレクティブ) .....	50

## F

-f (アセンブラオプション) .....	15, 19
FALSE 値、アセンブラの式 .....	8
--fpu (アセンブラオプション) .....	20
FRAME (CFI 演算子) .....	93

## G

-G (アセンブラオプション) .....	20
GE (CFI 演算子) .....	92
GT (CFI 演算子) .....	92

## H

HIGH (アセンブラ演算子) .....	38
HWRD (アセンブラ演算子) .....	38

## I

-I (アセンブラオプション) .....	20
-----------------------	----

-i (アセンブラオプション) .....	21
__IAR_SYSTEMS_ASM__ (定義済シンボル) .....	10
__IASMARM__ (定義済シンボル) .....	10
IASMARM (環境変数) .....	4
IASMARM_INC (環境変数) .....	4
IF (CFI 演算子) .....	93
IF (アセンブラディレクティブ) .....	59
IMPORT (アセンブラディレクティブ) .....	50
#include (アセンブラディレクティブ) .....	73
#include ファイル .....	20–21
INCLUDE (アセンブラディレクティブ) .....	79
iomacros.h .....	14

## J

-j (アセンブラオプション) .....	21
-----------------------	----

## L

-L (アセンブラオプション) .....	21
-l (アセンブラオプション) .....	22
LDR (ARM) (擬似命令) .....	101
LDR (CODE16) (擬似命令) .....	102
LDR (THUMB) (擬似命令) .....	103
LDR (アセンブラ命令) .....	101
LE (CFI 演算子) .....	92
LIBRARY (アセンブラディレクティブ) .....	45
#line (アセンブラディレクティブ) .....	73
LITERAL (CFI 演算子) .....	92
LOAD (CFI 演算子) .....	93
LOCAL (アセンブラディレクティブ) .....	60
:LOR: (アセンブラ演算子) .....	37
LOW (アセンブラ演算子) .....	39
LSHIFT (CFI 演算子) .....	92
LSTCND (アセンブラディレクティブ) .....	68
LSTCOD (アセンブラディレクティブ) .....	68
LSTEXP (アセンブラディレクティブ) .....	68
LSTMAC (アセンブラディレクティブ) .....	68
LSTOUT (アセンブラディレクティブ) .....	68

LSTPAG (アセンブラディレクティブ) .....	68
LSTREP (アセンブラディレクティブ) .....	68
LSTXRF (アセンブラディレクティブ) .....	68
LT (CFI 演算子) .....	92
LTORG (アセンブラディレクティブ) .....	79
LWRD (アセンブラ演算子) .....	39

## M

-M (アセンブラオプション) .....	22
MACRO (アセンブラディレクティブ) .....	60
#message (アセンブラディレクティブ) .....	73
MOD (CFI 演算子) .....	92
MOV (CODE16) (擬似命令) .....	104
MOV (THUMB) (擬似命令) .....	104
MOV (アセンブラ命令) .....	101
msa (ファイル名の拡張子) .....	3
MUL (CFI 演算子) .....	92
MVN (アセンブラ命令) .....	101

## N

-N (アセンブラオプション) .....	23
-n (アセンブラオプション) .....	23
NAME (アセンブラディレクティブ) .....	47
NE (CFI 演算子) .....	92
NOP (ARM) (擬似命令) .....	105
NOP (CODE16) (擬似命令) .....	105
NOT (CFI 演算子) .....	92

## O

-O (アセンブラオプション) .....	23
-o (アセンブラオプション) .....	24
ODD (アセンブラディレクティブ) .....	54
OR (CFI 演算子) .....	92
OVERLAY (アセンブラディレクティブ) .....	50

## P

-p (アセンブラオプション) .....	24
PAGE (アセンブラディレクティブ) .....	68
PAGSIZ (アセンブラディレクティブ) .....	68
parameters, typographic convention .....	xiii
#pragma (アセンブラディレクティブ) .....	73
PRESERVE8 (アセンブラディレクティブ) .....	47
PROGRAM (アセンブラディレクティブ) .....	47
PUBLIC (アセンブラディレクティブ) .....	50
PUBWEAK (アセンブラディレクティブ) .....	50

## R

-r (アセンブラオプション) .....	24
RADIX (アセンブラディレクティブ) .....	79
REPT (アセンブラディレクティブ) .....	61
REPTC (アセンブラディレクティブ) .....	61
REPTI (アセンブラディレクティブ) .....	61
REQUIRE (アセンブラディレクティブ) .....	50
REQUIRE8 (アセンブラディレクティブ) .....	48
RSEG (アセンブラディレクティブ) .....	54
RSHIFTA (CFI 演算子) .....	92
RSHIFTL (CFI 演算子) .....	92
RTMODEL (アセンブラディレクティブ) .....	48

## S

-S (アセンブラオプション) .....	24
-s (アセンブラオプション) .....	25
s (ファイル名の拡張子) .....	3
SECTION (アセンブラディレクティブ) .....	54
SECTION_TYPE (アセンブラディレクティブ) .....	54
SET (アセンブラディレクティブ) .....	57
SETA (アセンブラディレクティブ) .....	57
SFB (アセンブラ演算子) .....	39
SFE (アセンブラ演算子) .....	40
SFR → 特殊な関数レジスタを参照	
SUB (CFI 演算子) .....	93
SUB (アセンブラ命令) .....	98

# T

-t (アセンブラオプション) .....	25
THUMB (アセンブラディレクティブ) .....	52
TRUE 値、アセンブラの式 .....	8

# U

-U (アセンブラオプション) .....	25
UGT (アセンブラ演算子) .....	41
ULT (アセンブラ演算子) .....	41
UMINUS (CFI 演算子) .....	92

# V

VAR (アセンブラディレクティブ) .....	57
--------------------------	----

# W

-w (アセンブラオプション) .....	26
-----------------------	----

# X

-x (アセンブラオプション) .....	26
xcl (ファイル名の拡張子) .....	15, 19
XOR (CFI 演算子) .....	93
XOR (アセンブラ演算子) .....	41

# あ

アーキテクチャ、ARM .....	xi
アセンブラ	
呼出し構文 .....	3
アセンブラ EXTRN (アセンブラディレクティブ) ..	50
アセンブラオブジェクトファイル、ファイル名指定	23
アセンブラオプション	
アセンブラへの受渡し .....	4
コマンドライン、設定 .....	15

コマンドライン拡張ファイル、設定 .....	15
概要 .....	16
表記規則 .....	xiii
アセンブラシンボル .....	8
インポート .....	51
エクスポート .....	51
再配置可能な式 .....	11
定義済 .....	10
定義の解除 .....	25
アセンブラソースコードを移植 .....	53
アセンブラソースファイルのインクルード .....	74, 81
アセンブラディレクティブ	
CFI ディレクティブ .....	82
C 形式のプリプロセッサ .....	72
アセンブラの制御 .....	79
シンボル制御 .....	50
セクション制御 .....	54
データ定義または割り当ての .....	77
マクロ処理 .....	60
モジュール制御 .....	47
リストファイルの制御 .....	68
概要 .....	43
呼び出しフレーム情報 .....	82
条件付きアセンブリ .....	58
値割り当て .....	57
アセンブラのソースフォーマット .....	5
アセンブラのラベル .....	9
Thumb コードの .....	109
フォーマット .....	5
アセンブラの環境変数 .....	4
アセンブラの式 .....	6
アセンブラの出力 デバッグ情報を含める .....	24
アセンブラの診断 .....	107
アセンブラマクロ	
インラインルーチン .....	65
マクロの引用符、指定 .....	22
引数、渡される .....	63
処理 .....	64
生成した行、リストファイルの制御 .....	69

定義	62
定義済シンボル	63
特殊文字の使用	63
アセンブラリストファイル	
アドレスフィールド	13
クロスリファレンス、生成	26, 70
コメント	80
シンボルと相互参照テーブル	13
タブによる移動量、指定	25
ディレクティブ (フォーマット)	70
データフィールド	13
ファイル名、指定	22
フォーマット、指定	70
ページあたりの行数、指定	24
ヘッダセクション、無効	23
マクロの実行情報、含む	17
マクロ生成された行、制御	69
条件、指定	17
条件付コードと文字列	69
生成	21
生成された行、制御	69
無効	69
有効	69
アセンブラ演算子	29
概要	29
式での	6
同義語	31
優先順位	29
アセンブラ擬似命令	97
アセンブラ制御のディレクティブ	79
アセンブラ命令	6
ADD	98
BX	53
LDR	101
MOV	101
MVN	101
SUB	98
アセンブリ時のエラーメッセージ	108
アセンブリ時のメッセージのフォーマット	107
アセンブリ時のワーニングメッセージ	107

無効	26
アドレスフィールド、アセンブラリストファイル	13
アドレス、レジスタへロード	98-101

## い

インクルードパス、指定	20
-------------	----

## え

エラーメッセージ	
フォーマット	107
最大数、指定	19

## お

オプションの概要	16
オペランド	
アセンブラの式の	6
フォーマット	5
オペらんど演算子 → アセンブラ演算子を参照	

## く

グローバル値、定義	58
-----------	----

## こ

コマンドラインエラーのメッセージ、アセンブラ	107
コマンドラインオプション	15
呼出し構文のパート	3
受渡し	4
コマンドライン、拡張	19
コメント	
アセンブラソースコード	5
アセンブラリストファイル	80
複数行に渡るコメントに使用する	81
コメント、C 形式のプリプロセッサディレクティブ	75
コンピュータスタイル、表記規則	xiii

# し

## シンボル

→ アセンブラのシンボルを参照	
ユーザ定義、大文字 / 小文字を区別.....	25
他のモジュールへのエクスポート .....	51
定義済、アセンブラ .....	10
定義済、アセンブラマクロ .....	63
シンボルと相互参照テーブル、アセンブラリストファイル .....	13
クロスリファレンスのインクルードも参照	
シンボル制御のディレクティブ.....	50

# せ

## セクション

再配置可能.....	55
整列.....	56
セクション開始 (アセンブラ演算子) .....	39
セクション終了 (アセンブラ演算子) .....	40
セクション制御のディレクティブ.....	54

# そ

ソースファイル、インクルード.....	74, 81
ソースフォーマット、アセンブラの.....	5

# た

ターゲットコア、指定→プロセッサの設定を参照	
タブによる移動量、アセンブラリストファイルに指定.....	25

# て

ディレクティブ→アセンブラディレクティブを参照	
データフィールド、アセンブラリストファイル.....	13
データ、Thumb コードセクション中で定義.....	53
データ割り当てのディレクティブ.....	77
データ定義のディレクティブ.....	77

デバッグ情報、アセンブラ出力での生成.....	24
デフォルトのベース、定数.....	80

# の

ノーオペレーションコード、生成.....	105
----------------------	-----

# は

バイトオーダー .....	10
指定.....	19
バックトレース情報、定義.....	82

# ひ

ビットごとの排他的論理和 (アセンブラ演算子) ....	36
ビットごとの否定 (アセンブラ演算子) .....	35
ビットごとの論理積 (アセンブラ演算子) .....	35
ビットごとの論理和 (アセンブラ演算子) .....	36

# ふ

ファイルタイプ	
アセンブラソース .....	3
コマンドライン拡張 .....	15, 19
ファイル名の拡張子	
asm .....	3
msa .....	3
s .....	3
xcl .....	15, 19
ファイル名、アセンブラオブジェクトファイルに対する指定 .....	23–24
フォーマット、アセンブラソースコード.....	5
プリプロセッサシンボル	
定義.....	18
定義と定義の解除.....	74
プログラミングヒント .....	14
プログラミング経験、必要.....	xi
プログラムロケーションカウンタ (PLC).....	9
プロセッサのモード、ディレクティブ.....	52

## へ

ページあたりの行数、 アセンブラリストファイルの.....	24
ヘッダセクション、 アセンブラリストファイルで無効.....	23
ヘッダファイル、SFR.....	14

## ま

マクロの実行情報、リストファイルに含める.....	17
マクロを使用したインラインのコード作成.....	65
マクロ→アセンブラマクロを参照	
マクロ引用符.....	63
指定.....	22
マクロ処理ディレクティブ.....	60
マルチバイト文字のサポート.....	23

## め

メッセージ、標準出力への抑止.....	24
メモリ、メモリ空間を予約する.....	77
メモリ空間、予約と初期化.....	78

## も

モード制御のディレクティブ.....	52
モジュールの互換性.....	49
モジュール、開始.....	48
モジュール制御のディレクティブ.....	47

## ゆ

ユーザシンボルの大文字/小文字を区別する.....	25
ユーザシンボル、大文字/小文字を区別.....	25

## ら

ラベル→アセンブララベルを参照	
ランタイムモデル属性、宣言.....	49

## り

リストファイルの条件.....	17
リストファイルフォーマット.....	13
CRC.....	13
シンボルと相互参照	
ヘッダ.....	13
本体.....	13
リスト制御ディレクティブ.....	68
リテラルプール.....	101

## れ

レジスタ.....	9
代替名称.....	110

## ろ

ローカル値、定義.....	58
---------------	----

## 記号

! (アセンブラ演算子).....	36
!= (アセンブラ演算子).....	34
#define (アセンブラディレクティブ).....	72
#elif (アセンブラディレクティブ).....	72
#else (アセンブラディレクティブ).....	72
#endif (アセンブラディレクティブ).....	72
#error (アセンブラディレクティブ).....	72
#if (アセンブラディレクティブ).....	72
#ifndef (アセンブラディレクティブ).....	73
#include (アセンブラディレクティブ).....	73
#include ファイル.....	20-21
#line (アセンブラディレクティブ).....	73
#message (アセンブラディレクティブ).....	73
#pragma (アセンブラディレクティブ).....	73
#undef (アセンブラディレクティブ).....	73
\$ (アセンブラディレクティブ).....	79
\$ (プログラムロケーションカウンタ).....	9
% (アセンブラ演算子).....	36

& (アセンブラ演算子) .....	35
&& (アセンブラ演算子) .....	35
* (アセンブラ演算子) .....	32
+ (アセンブラ演算子) .....	32
- (アセンブラ演算子) .....	32-33
-B (アセンブラオプション) .....	17
-c (アセンブラオプション) .....	17
-D (アセンブラオプション) .....	18
-E (アセンブラオプション) .....	19
-e (アセンブラオプション) .....	19
-f (アセンブラオプション) .....	15, 19
-G (アセンブラオプション) .....	20
-I (アセンブラオプション) .....	20
-i (アセンブラオプション) .....	21
-j (アセンブラオプション) .....	21, 109
-L (アセンブラオプション) .....	21
-l (アセンブラオプション) .....	22
-M (アセンブラオプション) .....	22
-N (アセンブラオプション) .....	23
-n (アセンブラオプション) .....	23
-O (アセンブラオプション) .....	23
-o (アセンブラオプション) .....	24
-p (アセンブラオプション) .....	24
-r (アセンブラオプション) .....	24
-S (アセンブラオプション) .....	24
-s (アセンブラオプション) .....	25
-t (アセンブラオプション) .....	25
-U (アセンブラオプション) .....	25
-w (アセンブラオプション) .....	26
-x (アセンブラオプション) .....	26
--cpu (アセンブラオプション) .....	18
--endian (アセンブラオプション) .....	19
--fpu (アセンブラオプション) .....	20
/ (アセンブラ演算子) .....	33
/**/ (アセンブラディレクティブ) .....	79
// (アセンブラディレクティブ) .....	79
:AND: (アセンブラ演算子) .....	35
:EOR: (アセンブラ演算子) .....	36
:LAND: (アセンブラ演算子) .....	35

:LEOR: (アセンブラ演算子) .....	41
:LNOT: (アセンブラ演算子) .....	36
:MOD: (アセンブラ演算子) .....	36
:NOT: (アセンブラ演算子) .....	35
:OR: (アセンブラ演算子) .....	36
:SHL: (アセンブラ演算子) .....	40
:SHR: (アセンブラ演算子) .....	41
= (アセンブラディレクティブ) .....	57
= (アセンブラ演算子) .....	34
== (アセンブラ演算子) .....	34
> (アセンブラ演算子) .....	34
>= (アセンブラ演算子) .....	35
>> (アセンブラ演算子) .....	41
^ (アセンブラ演算子) .....	36
_ARMVFP_ (定義済シンボル) .....	10
_BUILD_NUMBER_ (定義済シンボル) .....	10
_DATE_ (定義済シンボル) .....	10
_FILE_ (定義済シンボル) .....	10
_IAR_SYSTEMS_ASM_ (定義済シンボル) .....	10
_LINE_ (定義済シンボル) .....	10
_LITTLE_ENDIAN_ (定義済シンボル) .....	10
_TID_ (定義済シンボル) .....	10
_TIME_ (定義済シンボル) .....	10
_VER_ (定義済シンボル) .....	10
_args (定義済マクロシンボル) .....	63
_IASMARM_ (定義済シンボル) .....	10
(アセンブラ演算子) .....	36
(アセンブラ演算子) 論理和 (アセンブラ演算子) .....	37
~ (アセンブラ演算子) .....	35

## 数字

1 番目のバイト (アセンブラ演算子) .....	37
2 番目のバイト (アセンブラ演算子) .....	37
32 ビット式、レジスタへのロード .....	101
3 番目のバイト (アセンブラ演算子) .....	37
4 番目のバイト (アセンブラ演算子) .....	37