IAR Embedded Workbench®

IAR アセンブラ リファレンスガイド

Advanced RISC Machines Ltd
ARM® コア

AARM-9-J

IAR SYSTEMS

改版情報

第 9 版 : 2012 年 5 月

部品番号 : AARM-9-J

本ガイドは、ARM 用 IAR Embedded Workbench® のバージョン 6.x に適用する。

内部参照 : M12, asrct2010.3, V_111012, asrcarm6.40, IMAE.

目次

表

はじめに

ARM 用 IAR アセンブラ リファレンスガイドへようこそ。このガイドは、ARM 用 IAR アセンブラを使用して要件に合ったアプリケーションを開発する際に役立つ、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 コア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、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。また、以下について十分な知識があるユーザを対象としています。

- ARM コアのアーキテクチャおよび命令セット。ARM コアについては、Advanced RISC Machines Ltd のドキュメントを参照してください
- アセンブラ言語でのプログラミングに関する基礎知識
- 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

IAR アセンブラを初めて使用する場合は、このリファレンスガイドの *ARM 用 IAR アセンブラの概要*をお読みになることをお勧めします。

中級者や上級者の場合、概要の後に続くリファレンス情報を中心にお読みいただけます。

IAR システムズのツールキットを初めて使用する場合は、まず *『ARM 用 IDE プロジェクト管理およびビルドガイド』* の前半を確認することをお勧めします。これらの章には、製品についての概要や、初心者向けのチュートリアルが用意されています。

このガイドの概要

本ガイドの構成および各章の概要を以下に示します。

- 「*ARM 用 IAR アセンブラの概要*」では、プログラミング情報を提供します。また、ソースコードのフォーマットや、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アセンブラオプションを設定する方法と、環境変数の使用方法について説明します。続いて、アセンブラオプションについてアルファベット順に簡単に説明し、各オプションの詳細なリファレンス情報を提供します。
- 「*アセンブラ演算子*」では、アセンブラ演算子の概要を優先順に説明し、各演算子の詳細なリファレンス情報を提供します。
- *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 用 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 用 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。

その他のドキュメント

ユーザドキュメンテーションは、ハイパーテキスト PDF 形式、およびコンテキスト依存のオンラインヘルプシステム（HTML フォーマット）があります。ドキュメンテーションには、インフォメーションセンタあるいは IAR Embedded Workbench IDE の［**ヘルプ**］メニューからアクセスできます。オンラインヘルプシステムは、F1 キーを押しても使用できます。

ユーザガイドおよびリファレンスガイド

IAR システムズの各開発ツールについては、一連のガイドで説明しています。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 システムズの製品のインストールおよび登録の要件と詳細については、同梱されている*クイックレファレンス*のブックレットおよび*『インストールとライセンス登録ガイド』*をご覧ください。
- IAR Embedded Workbench および提供されるツールの利用にあたっては、*『IAR Embedded Workbench® の使用開始の手順』*を参照してください。

- プロジェクト管理とビルドでの IDE の使用については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』*を参照してください。
- IAR C-SPY® デバッガの使用については、*『ARM 用 C-SPY® デバッガガイド』*を参照してください。
- ARM 用 IAR C/C++ コンパイラのプログラミングおよび IAR ILINK リンカを使用したリンクについては、*『ARM 用 IAR C/C++ 開発ガイド』*を参照してください。
- IAR DLIB ライブラリの使用については、オンラインヘルプで利用できる *DLIB ライブラリリファレンス情報*を参照してください。
- ARM用IAR Embedded Workbenchの旧バージョンで開発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 用 IAR Embedded Workbench® 移行ガイド』*を参照してください。
- MISRA-C ガイドラインを使用して、安全性を最重要視したアプリケーションを開発する方法については、*『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』*または *『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』*を参照してください。

注： 製品のインストール内容によっては、他のドキュメントも提供される場合があります。

オンラインヘルプシステムを参照

コンテキスト依存のオンラインヘルプの内容は以下のとおりです。

- IAR C-SPY® デバッガを使用したデバッグについての包括的な情報
- IDE のメニューやウィンドウ、ダイアログボックスに関するリファレンス情報
- コンパイラのリファレンス情報
- DLIB ライブラリ関数のキーワードリファレンス情報 関数のリファレンス情報を確認するには、エディタウィンドウで関数名を選択し、F1 キーを押します。

WEB サイト

推奨 Web サイト：

- Advanced RISC Machines Ltd の Web サイト (**www.arm.com**) には、ARM コアに関する情報とニュースが記載されています。
- IAR システムズの Web サイト (**www.iar.com/jp**) では、アプリケーションノートおよびその他の製品情報を公開しています。

表記規則

IAR システムズのドキュメントでプログラミング言語 C と記述されている場合、特に記述がない限り C++ も含まれます。

たとえば arm¥doc など製品インストール内のディレクトリについて言及する場合、その場所のフルパスを前提とします。つまり、この場合は c:¥Program Files¥IAR Systems¥Embedded Workbench 6.*n*¥arm¥doc です。

表記規則

IAR システムズのドキュメント セットでは、次の表記規則を使用します :

スタイル	用途
コンピュータ	• ソースコードの例、ファイルパス。 • コマンドライン上のテキスト。 • 2 進数、16 進数、8 進数。
パラメータ	パラメータとして使用される実際の値を表すプレースホルダ。たとえば、*filename*.h の場合、*filename* はファイルの名前を表します。
[オプション]	コマンドのオプション部分。
[a|b|c]	代替の選択肢を持つコマンドのオプション部分。
{a|b|c}	コマンドの必須部分に選択肢があることを示します。
太字	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。
斜体	• 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。 • 強調。
…	3 点リーダは、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® IDE 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。
	開発やプログラミングについてのヒントを示します。
	ワーニングを示します。

表 1: このガイドの表記規則

命名規約

以下の命名規約は、このドキュメントに記述されている IAR システムズの製品およびツールで使用されています。

ブランド名	一般名称
ARM 用 IAR Embedded Workbench®	IAR Embedded Workbench®
ARM 用 IAR Embedded Workbench® IDE	IDE
ARM 用 IAR C-SPY® デバッガ	C-SPY、デバッガ
IAR C-SPY® シミュレータ	シミュレータ
ARM 用 IAR C/C++ コンパイラ	コンパイラ
ARM 用 IAR アセンブラ	アセンブラ
IAR ILINK リンカ	ILINK、リンカ
IAR DLIB ライブラリ	DLIB ライブラリ

表 2: このガイドで使用されている命名規約

ARM 用 IAR アセンブラの概要

この章には、以下のセクションがあります。

- 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
- リストファイルのフォーマット
- プログラミングのヒント

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

アプリケーション全体をアセンブラ言語で記述するのではない場合でも、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 コア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ためには、ARM コアのアーキテクチャと命令セットを理解しておく必要があります。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。

イントロダクション

アセンブラアプリケーションの開発を始めるにあたって、以下の情報が参考になります。

- インフォメーションセンタ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混在に関するチュートリアルを実行しておく。
- *『ARM 用 IAR C/C++ 開発ガイド』*で、アセンブラ言語インタフェースについての説明を参照する。C 言語とアセンブラモジュールを結合する場合に役に立ちます。
- IAR Embedded Workbench IDE では、アセンブラプロジェクトの*テンプレート*をベースに新しいプロジェクトを作成できます。

モジュール方式のプログラミング

優れたソフトウェア設計においてモジュール方式プログラミングが大きな役割を果たすということは広く知られています。単体構造にするのではなく、複数の小型モジュールを集めてコードを構成すると、アプリケーションコードを論理的な構造に体系化できます。これによりコードがわかりやすくなるうえ、次のような効果があります。

- プログラム開発の効率化
- モジュールの再利用
- 保守の容易さ

IAR の開発ツールでは、ソフトウェアをモジュール構造にするためのさまざまな機能をご用意しています。

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場合、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。各モジュールを異なる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。セクションにコードとデータを配置するには、セクション制御ディレクティブを使用します。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。セクションにより、コードやデータをメモリ内でどのように配置するか制御できます。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カ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、さまざまなアプリケーションで使用される複数の便利なルーチンがすぐに蓄積されます。小さなオブジェクトファイルが大量に蓄積されるのを回避するためには、このようなルーチンが含まれるモジュールを*ライブラリ*オブジェクトに集めます。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、1 つのライブラリに多くのオブジェクトファイルを集めるためにライブラリプロジェクトを設定できます。この例については、インフォメーションセンタのチュートリアルを参照してください。

まとめると、ソフトウェアの設計にはモジュール方式のプログラミングが役に立ちます。また、モジュール構造は以下の方法で作成できます。

- 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（C レベルでの*関数*に相当）
- アセンブラソースコードを*セクション*に分割し、最終的にメモリ内でコードやデータをどのように配置するか正確に制御できるようにする
- ルーチンをライブラリに集める。つまり、オブジェクトファイルの数を減らし、モジュールが条件付きでリンクされるようにする

外部インタフェースの詳細

このセクションでは、アセンブラが環境とどのようにやりとりするかについて説明します。

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 用 IDE プロジェクト管理およびビルドガイド』*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、以下のコマンドを使用して、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 用 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。アセンブラ出力のデフォルトのファイル名拡張子は、`o` です。

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。

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受け渡すには、次の 3 つの方法があります。

- 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（*19 ページの 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、各アセンブリで必要なオプションを指定する便利な方法として、環境変数の値を各コマンドラインに自動的に付加します（*20 ページの 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（*39 ページの -f* を参照）。

オプションの構文の一般的なガイドライン、オプションの概要、各オプションの詳しい情報については、*アセンブラオプション*を参照してください。

環境変数

IAR アセンブラでは、以下の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	インフ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 3: 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、常に temp.lst という名称のリストファイルが生成されます。

```
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 用 IAR C/C++ 開発ガイド』*を参照してください。

エラーリターンコード

IAR アセンブラをバッチファイル内から使用する場合、次に行うステップを決定するために、アセンブリが成功したかどうかを判断しなければならない場合があります。このため、アセンブラはこれらのエラーリターンコードを返します。

リターンコード	説明
0	アセンブリは成功しましたが、ワーニングが発生している場合があります。
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です）。
2	エラーが発生しました。

表 4: アセンブラのエラーリターンコード

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]
```

ここで、コンポーネントは次のとおりです。

label	ラベルの定義。アドレスを表現するシンボルです。ラベルの開始位置を最初の列にする場合（つまり、行の左端から開始する場合）、（コロン）はオプションです。
operation	アセンブラ命令またはディレクティブ。開始位置は、最初の列にしないでください。左側に空白を含める必要があります。
operands	アセンブラ命令またはディレクティブには、オペランドを含めないか、1 つまたは複数のオペランドを使用することができます。オペランドはコンマで区切ります。
comment	コメント。前に ;（セミコロン）を付けます。 C または C++ のコメントも許可されます。

コンポーネントは空白またはタブで区切ります。

ソース行は 1 行あたり 2047 文字以内にします。

タブ文字 (ASCII `09H`) は、最も一般的な慣行に従って、8、16、24 カラムなどに設定できます。これにより、リストファイルでのソースコード出力およびデバッグ情報に影響があります。タブはエディタによって設定が異なる可能性があるため、ソースファイルでタブを使用しないでください。

アセンブラ命令

ARM 用 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、式オペランドと演算子から構成されています。

アセンブラでは、算術演算や論理演算などさまざまな式を使用できます。すべての演算子は、32 ビットの 2 の補数整数を使用します。コードの生成のために値が使用される場合、範囲チェックが行われます。

式は左から右へと評価されます。ただし、演算子の優先度によってこの順番が上書きされた場合を除きます。*51 ページの アセンブラ演算子*、も参照してください。

式で有効なオペランドは以下のとおりです。

- データまたはアドレスの定数。浮動小数点定数を除きます。
- シンボルのシンボル名。データとアドレスのどちらを表すこともできます。アドレスの場合、*ラベル*とも呼ばれます。
- 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、後ほど詳しく説明します。

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。または、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。解決されない場合、アセンブラはエラーを生成します。

整数定数

すべての IAR システムズのアセンブラは、32 ビットの 2 の補数内部演算を使用しているため、整数の（符号付き）範囲は -2147483648 ～ 2147483647 となります。

定数は一連の数字で記述し、オプションで先頭に –（マイナス）符号を付けて負の数を示します。

コンマと小数点は許可されません。

以下のような数値表現がサポートされます。

整数のタイプ	例
2 進数	1010b、b'1010
8 進数	1234q、q'1234
10 進数	1234、-1、d'1234
16 進数	0FFFFh、0xFFFF、h'FFFF

表 5: 整数定数のフォーマット

注：プレフィックスとサフィックスはいずれも、大文字または小文字で記述できます。

ASCII 文字定数

ASCII 定数は、任意の数の文字を半角または全角の引用符で囲みます。ASCII 文字列には、出力可能な文字と空白のみを使用できます。引用符の文字自体にアクセスする場合、引用符を 2 つ並べて使用する必要があります。

フォーマット	値
'ABCD'	ABCD（4 文字）
"ABCD"	ABCD'¥0'（5 文字、最後は ASCII の null）
'A''B'	A'B
'A'''	A'
''''（4 つの引用符）	'
''（2 つの引用符）	空白文字列（値なし）
""（2 つの二重引用符）	空白文字列（1 つの ASCII null 文字）
¥'	' — 文字列内で引用符を使用する場合（「I¥'d love to」）
¥¥	¥ — 文字列内で ¥ を使用する場合
¥"	" — 文字列内で二重引用符を使用する場合

表 6: ASCII 文字定数のフォーマット

浮動小数点定数

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（32 ビット）浮動小数点形式、倍精度形式（64 ビット）または小数形式に変換します。

浮動小数点値は次のフォーマットで記述できます。

```
[+|-][digits].[digits][{E|e}[+|-]digits]
```

次の表は、有効な例の一部を示します。

フォーマット	値
10.23	1.023×10^{1}
1.23456E-24	1.23456×10^{-24}
1.0E3	1.0×10^{3}

表 7: 浮動小数点定数

空白とタブは、浮動小数点定数では使用できません。

注記 : 浮動小数点定数を式で使用しても、有用な結果とはなりません。

TRUE および FALSE

式では、ゼロ値は FALSE と見なされ、ゼロ以外の値は TRUE と見なされます。

条件付きの式では、FALSE の場合は値 0、TRUE の場合は 1 が返されます。

シンボル

ユーザ定義シンボルの長さは最大 255 文字であり、すべての文字は有意です。シンボルの後に続く演算の種類に応じて、シンボルはデータシンボルまたはアドレスシンボルです（アドレスシンボルはラベルと呼びます）。命令の前のシンボルはラベルであり、EQU ディレクティブなどの前のシンボルはデータシンボルです。次のシンボルがあります。

- 絶対 — 値はアセンブラに既知です。
- 再配置可能 — リンク時に解決されます。

シンボルの先頭は、文字 a ～ z または A ～ Z、?（疑問符）、または _（下線）にする必要があります。シンボルには数字 0 ～ 9 と $（ドル）を使用できます。

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

```
`strange#label`
```

命令、レジスタ、演算子、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、アセンブラのユーザシンボルの大文字 / 小文字の区別オプション (-s) で切り換えることができます。詳細については、*46 ページの -s* を参照してください。

モジュール間でシンボルをどのように共有するか制御するには、シンボル制御ディレクティブを使用します。たとえば、1 つ以上のシンボルを他のモジュールで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。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。

シンボルとラベルはバイトアドレスです。

ラベル

メモリロケーションに使用されるシンボルをラベルと呼びます。

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トレースします。これを、*プログラムロケーションカウンタ*と呼びます。

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

```
section MYCODE:CODE(2)
arm
b       .               ; 永久ループ
end
```

レジスタシンボル

以下の表に、既存の定義済レジスタシンボルを示します。

名称	サイズ	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-D31	64 ビット	倍精度の浮動小数点コプロセッサレジスタ
Q0-Q15	128 ビット	高度な SIMD レジスタ
FPEXC	32 ビット	浮動小数点コプロセッサ、例外レジスタ
FPSCR	32 ビット	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	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

表 8: 定義済レジスタシンボル

名称	サイズ	説明
S0S31	32 ビット	単精度の浮動小数点コプロセッサレジスタ
SPSR	32 ビット	保存されたプログラムステータスレジスタ

表 8: 定義済レジスタシンボル（続き）

また、コアによっては、命令構文で使用可能な場合、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

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ます。シンボルは現在のアセンブリについての情報を提供するため、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ることができます。アセンブラから返された文字列は二重引用符に囲まれています。

以下の定義済シンボルがあります。

シンボル	値
__ARM_ADVANCED_SIMD__	--cpu オプションに基づいて設定される整数。選択されたプロセッサのアーキテクチャが Advanced SIMD アーキテクチャの拡張の場合、このシンボルは 1 に設定されます。このシンボルは、他のコアについては未定義です。
__ARM_MEDIA__	--cpu オプションに基づいて設定される整数。選択されたプロセッサのアーキテクチャがマルチメディア用の ARMv6 SIMD 拡張である場合、このシンボルは 1 に設定されます。このシンボルは、他のコアについては未定義です。
__ARM_MPCORE__	--cpu オプションに基づいて設定される整数。選択されたプロセッサのアーキテクチャが Multiprocessing Extensions の場合、このシンボルは 1 に設定されます。このシンボルは、他のコアについては未定義です。
__ARM_PROFILE_M__	--cpu オプションに基づいて設定される整数。選択されたプロセッサがプロファイル M コアの場合、このシンボルは 1 に設定されます。このシンボルは、他のコアについては未定義です。

表 9: 定義済シンボル

シンボル	値
__ARMVFP__	--fpu オプションに基づいて設定される整数で、ベクタ浮動小数点コプロセッサ用の浮動小数点命令が有効になっているかどうかを識別します。このシンボルは __ARMVFPV2__、__ARMVFPV3__、または __ARMVFPV4__ に定義されます。これらのシンボル名は、__ARMVFP__ シンボルの評価に使用できます。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。__BUILD_NUMBER__ は、後でリリースされたアセンブラのほうが遅い番号になるとは限りません。
__DATE__	dd/Mmm/yyyy フォーマットで示す現在の日付（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（数字）。将来のバージョンでは、番号が大きくなる可能性があります。このシンボルを #ifdef で評価し、コードが IAR システムズのアセンブラでアセンブルされたものかどうかを検出できます。
__IASMARM__	コードが ARM 用 IAR アセンブラでアセンブルされている場合は 1 に設定される整数です。
__LINE__	現在のソースの行番号（数字）。
__LITTLE_ENDIAN__	使用中のバイトオーダを識別します。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 フォーマットで示す現在の時刻（文字列）。
__VER__	整数フォーマットのバージョン番号。たとえば、バージョン 6.21.2 は、6021002（数字）として返されます。

表 9: 定義済シンボル（続き）

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 用 IAR C/C++ 開発ガイド』*を参照してください。

シンボル値をコードに含める

複数のデータ定義ディレクティブで、コードにシンボル値を含めることができます。これらのディレクティブは、値を定義するか、メモリを予約します。コードにシンボル値を含めるには、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、プログラムに表示されるようにアセンブリ時間を文字列として含めるには、次のように指定します。

```
            name    timeOfAssembly
            extern  printStr
            section MYCODE:CODE(2)

            adr     r0,time         ; 時間の文字列データの
                                    ; アドレスを R0 にストアする
            bl      printStr        ; 文字列を出力するルーチンを呼ぶ
            bx      lr              ; リターンする

            data                    ; data モード
time        dc8     __TIME__        ; アセンブリ時間を表す文字列
            end
```

条件付きアセンブリ用のシンボルをテストする

アセンブリ時にシンボルをテストするには、いずれかの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、アセンブリ時にアセンブリプロセスを制御できます。

たとえば、古いアセンブラバージョンと新しいアセンブラバージョンのどちらを使用しているかに応じて別々のコードをアセンブルするには、次のようにします。

```
#if (__VER__ > 6021000)             ; 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
```

詳細については、*84 ページの 条件付きアセンブリディレクティブ*を参照してください。

絶対式および再配置可能式

式を構成しているオペランドに応じて、式は*絶対*または*再配置可能*のいずれかになります。絶対式とは、絶対シンボルまたは再配置可能シンボルの差分のみを含む式のことです。

再配置可能セクションにシンボルが含まれている式は、セクションのロケーションに依存しているため、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式と呼ばれます。

このような式は、リンク時に IAR ILINK リンカによって評価され、解決されます。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 DATA と CODE セクションを以下のように定義できます。

```
            name    simpleExpressions
            section MYCONST:CONST(2)
first       dc8     5                ; リロケータブルラベル
second      equ     10 + 5           ; 絶対式

            dc8     first            ; 使用可能なリロケータブル式の例
            dc8     first + 1
            dc8     first + second
            end
```

注： アセンブリ時に、範囲チェックは行われません。範囲チェックはリンク時に行われ、値が長すぎる場合にはリンカエラーが発生します。

式の制限

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。一例としては、IF などの条件文で使用される式です。このような条件文では、アセンブリ時に式を評価する必要があるため、外部シンボルを含めることはできません。

次の式制限は、適用される各ディレクトリの説明で参照されています。

前方参照

式で参照されるすべてのシンボルは既知である必要があり、前方参照は許可されません。

外部参照

式では外部参照は許可されません。

絶対

式は絶対値に対して評価する必要があります。再配置可能値（セクションオフセット）は許可されません。

固定

式は固定である必要があります。つまり、可変サイズの命令に依存させることはできません。可変サイズの命令とは、オペランドの数値に応じてサイズが変動する可能性がある命令のことです。

リストファイルのフォーマット

アセンブラリストファイルのフォーマットは次のとおりです。

ヘッダ

ヘッダセクションには、製品のバージョン情報、ファイルの作成日時、使用されたオプションが含まれています。

ボディ

リストのボディは、以下の情報フィールドで構成されています。

- ソースファイル内の行番号。マクロで生成された行がリストされている場合、ソース行番号フィールドには .（ピリオド）が含まれています。
- アドレスフィールドは、メモリ内のロケーションを示します。これはセクションの種類に応じて絶対にも相対にもできます。表記法は 16 進法です。
- データフィールドは、ソース行によって生成されたデータを示します。表記法は 16 進法です。解決されなかった値は .....（ピリオド）として表現されます。ここで、2 つのピリオドが 1 バイトを示します。これらの未解決値はリンクプロセス中に解決されます。
- アセンブラソース行。

概要

ファイルの *フッタ* には、生成されたエラーと警告の概要が記述されています。

シンボルとクロスリファレンスの表

[クロスリファレンスを含む] 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、シンボルとクロスリファレンスの表が生成されます。

表の各シンボルに対して、次の情報が記述されています。

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（相対）。
セクション	このシンボルが相対的に定義されているセクションの名前。
値 / オフセット	現在のセクションの開始点に相対的な、現在のモジュール内でのシンボルの値（アドレス）。

表 10: シンボルとクロスリファレンスの表

プログラミングのヒント

このセクションでは 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++ ソースファイルの両方が含まれるプロジェクトについての情報は、*『ARM 用 IAR C/C++ 開発ガイド』*を参照してください。

特殊機能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。

例

デバイスの UART リードアドレス 0x40050000 は、ionuc100.h ファイルで以下のように定義されます :

```
__IO_REG32_BIT(UA0_RBR,0x40050000,__READ_WRITE ,__uart_rbr_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。

```
UA0_RBR DEFINE 0x40050000
```

C 形式のプリプロセッサディレクティブを使用する

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。そのため、マクロでプロプロセッサディレクティブを使用しないでください。また、これらをアセンブラ形式のコメントと混在させないでください。コメントの詳細については、*107 ページの アセンブラ制御ディレクティブ*を参照してください。

#define のような C 形式のプリプロセッサディレクティブは、ソースコードファイルの残り部分で有効ですが、EQU などのアセンブラディレクティブは現在のモジュール内でのみ有効です。

アセンブラオプション

この章では、まずコマンドラインでオプションを設定する方法と、アセンブラオプションの概要についてアルファベット順に説明します。続いて、各アセンブラオプションに関する詳細なリファレンス情報を提供します。

『ARM 用 IDE プロジェクト管理およびビルドガイド』 では、IAR Embedded Workbench® IDE でのアセンブラオプションの設定方法を説明し、使用可能なオプションについてのリファレンス情報を提供しています。

コマンドラインアセンブラオプションの使用

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、`iasmarm` コマンドの後にそれらをインクルードします。

```
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、ソースファイル `power2.s` をアセンブルする際は、このコマンドを使用してデフォルトのリストファイル (`power2.lst`) にリストを生成します。

```
iasmarm power2.s -L
```

一部のオプションではファイル名を指定できます。オプション文字の後に、スペースで区切って指定してください。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.s -l list.lst
```

ほかに、ファイル名以外の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.s -Llist¥
```

注： すでに存在しているサブディレクトリを指定しなくてはなりません。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必要があります。

コマンドライン拡張 (XCL) 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

以下の表に、コマンドラインで使用できるアセンブラオプションを示します。

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	プリプロセッサシンボルを定義。
-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を指定します。
-f	コマンドラインを拡張。
--fpu	浮動小数点コプロセッサアーキテクチャの構成。
-G	ソースファイルを標準入力から読み込みます。
-g	システムインクルードファイルの自動検索を無効化します。
-I	ヘッダファイルの検索パスを追加します。
-i	インクルードされたテキストを一覧表示します。
-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。

表 11: 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
-L	パスへのリストファイルを生成します。
-l	リストファイルを生成します。
--legacy	古いツールチェーンとリンク可能なコードを生成します。
-M	マクロの引用符。
-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します。
-O	パスへのオブジェクトファイル名を設定。
-o	オブジェクトファイル名を設定。
-p	リストファイルのページ行数を設定。
-r	デバッガ情報を生成。
-S	サイレント処理を設定。
-s	ユーザシンボルの大文字 / 小文字を区別します。
--system_include_dir	システムインクルードファイルのパスを指定。
-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	ワーニングを無効にします。
-x	クロスリファレンスをインクルードする。

表 11: 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概要

以下のセクションでは、各アセンブラオプションに関する詳細なリファレンス情報を提供します。

[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

構文

```
-B
```

説明

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。

関連項目

42 ページの -L。

[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

構文

```
-c{D|M|E|A|O}
```

パラメータ

D	リストファイルを無効化
M	マクロ定義を含める
E	マクロ拡張を除外
A	アセンブルされた行のみを含める
O	複数行のコードを含める

説明

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。

関連項目

42 ページの -L。

関連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

構文

```
--cpu target_core
```

パラメータ

target_core	ARM7TDMI のような値や、4T といったアーキテクチャのバージョンを使用できます。ARM7TDMI がデフォルト値です。

説明

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、このオプションを使用します。

関連項目

コプロセッサアーキテクチャの派生形の詳細なリストは、*『ARM 用 IAR C/C++ 開発ガイド』*を参照。

[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

構文

```
-Dsymbol[=value]
```

パラメータ

symbol	定義するシンボル名。
value	シンボルの値。値を指定しない場合、1 が使用されます。

説明

このオプションを使用して、プリプロセッサで使用するシンボルを定義します。

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、アプリケーション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
...     ; テストバージョンのみの追加コード行
#endif
```

次に、コマンドラインで必要となるバージョンを次のように選択します。

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

また、頻繁に変更する必要のある変数をソースで使用するとします。この場合、ソースではこの変数を定義せず、以下のように -D を使用してコマンドラインで値を指定することができます。

```
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] > [シンボル定義]

-E

構文

```
-Enumber
```

パラメータ

number	アセンブラがアセンブルを中止するエラー発生回数（*number* は正の整数）。0 は制限なしを示します。

説明

このオプションを使用して、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、最大値は 100 です。

[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

構文

```
-e
```

説明

このオプションを使用して、コードとデータをビッグエンディアンのバイトオーダで生成します。デフォルトのバイトオーダ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

構文

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

パラメータ

little、l（デフォルト）	リトルエンディアンのバイトオーダを指定します。
big、b	ビッグエンディアンのバイトオーダを指定します。

説明

このオプションは、生成されるコートおよびデータのバイトオーダを指定するときに使用します。

[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

構文

`-f filename`

パラメータ

filename	**コマンドラインを拡張するコマンドが、指定ファイルから読み込まれます。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。**

ファイル名の指定については、*33 ページの コマンドラインアセンブラオプションの使用*を参照してください。

説明

このオプションを使用して、指定されたファイルから読み込まれたテキストでコマンドラインを拡張します。

`-f` オプションは、オプションの数が多く、コマンドラインに指定するよりファイルに配置する方が簡単である場合に特に便利です。

例

ファイル `extend.xcl` からオプションを取得してアセンブラを実行するには、以下のように指定します。

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

このオプションを設定するには、以下のように指定します。

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

--fpu

構文

`--fpu fpu_variant`

パラメータ

fpu_variant	**浮動小数点コプロセッサアーキテクチャの派生形、たとえば `VFPv3` や `none`（デフォルト）などです。**

説明

このオプションを使用して、浮動小数点コプロセッサのアーキテクチャ派生形を指定し、正しい命令セットとレジスタを取得します。

関連項目

コプロセッサアーキテクチャの派生形の詳細なリストは、*『ARM 用 IAR C/C++ 開発ガイド』* を参照。

［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

構文

```
-G
```

説明

このオプションを使用して、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 を使用すると、ソースファイル名は指定できません。

このオプションは、IDE では使用できません。

-g

構文

```
-g
```

説明

デフォルトでは、アセンブラは自動的にシステムインクルードファイルを検索します。このオプションを使用して、システムインクルードファイルの自動検索を無効にします。この場合は、-I アセンブラオプションを使用して検索パスを設定しなければならないこともあります。

［プロジェクト］>［オプション］>［アセンブラ］>［プリプロセッサ］>［標準のインクルードディレクトリを無視］

-I

構文

```
-Ipath
```

パラメータ

path	#include **ファイルの検索パス。**

説明

このオプションを使用して、プリプロセッサで使用するパスを指定します。このオプションは、1 つのコマンドラインで複数個使用できます。

デフォルトでは、アセンブラは現在の作業ディレクトリ、システムヘッダディレクトリ、および IASMARM_INC 環境変数で指定されたパスにある #include ファイルを検索します。-I オプションは、現在の作業ディレクトリでファイルが見つからない場合に検索するディレクトリの名前をアセンブラに指定します。

例

以下に例を示します。

```
-Ic:¥global¥ -Ic:¥thisproj¥headers¥
```

というオプションを使用し、

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

とソースコードに記述すると、アセンブラはまず現在のディレクトリ内を検索してから、ディレクトリ `c:¥global¥` を検索し、続いてディレクトリ `C:¥thisproj¥headers¥` を検索します。最後にアセンブラは、`IASMARM_INC` 環境変数で指定されたディレクトリを検索します。ただし、この変数が設定され、システムヘッダディレクトリにある必要があります。

[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

構文

```
-i
```

説明

このオプションを使用して、リストファイル内の `#include` ファイルの一覧を表示します。

`#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] > [#included テキスト]

-j

構文

```
-j
```

説明

このオプションを使用して、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。

関連項目

*53 ページの 演算子の同義語*および *ARM 用 IAR アセンブラへの移行*の章。

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

-L

構文

```
-L[path]
```

パラメータ

パラメータなし	ソースファイルと同じ名前で、ファイル拡張子が lst のリストを生成します。
path	リストファイルの出力先のパス。ファイル名の前にスペースを含めることはできません。

説明

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 [*path*]*sourcename*.lst に送ります。

-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¥
```

関連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

構文

```
-l filename
```

パラメータ

filename	出力は指定ファイルに格納されます。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場合には、lst が使用されます。

*ファイル名の指定については、33 ページの コマンドラインアセンブラオプションの使用*を参照してください。

説明

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。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。

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。

関連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]

--legacy

構文

```
--legacy={RVCT3.0}
```

パラメータ

`RVCT3.0`	RVCT3.0 でリンカを指定します。このパラメータを `--aeabi` オプションとともに使用して、RVCT3.0 でリンカにリンクするコードを生成します。

説明

このオプションを使用して、指定したツールチェーンと互換性のあるオブジェクトコードを生成します。

このオプションを設定するには、**[プロジェクト] > [オプション] > [アセンブラ] > [追加オプション]** を選択します。

-M

構文

```
-Mab
```

パラメータ

ab	各マクロ引数の左右の引用符にそれぞれ使用する文字。

説明

このオプションを使用して、各マクロ引数の左側と右側の引用符として使用する文字（それぞれ *a* と *b*）を設定します。

デフォルトでは、これらの引用符は < と > です。`-M` オプションを使用して、他の表記法に合せて引用符を変更したり、マクロ引数に < や > 自体を使用したりできます。

例

以下のオプションを使用するとします。

```
-M[]
```

ソースには以下のように記述します。

```
print [>]
```

これにより、> を引数として使用してマクロ `print` を呼び出すことができます。

注：ホスト環境によっては、以下のようにマクロの引用符付きで引用符を使用する必要がある場合もあります。

```
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

構文

```
-N
```

説明

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 -l オプションと同時に使用すると便利です。

関連項目

42 ページの -L。

[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

構文

```
-n
```

説明

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。このオプションを使用して、ソースコード内のマルチバイト文字を、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します。

マルチバイト文字は、C/C++ スタイルのコメント、文字列定数、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] > [マルチバイト文字サポートを有効にする]

-O

構文

```
-O[path]
```

パラメータ

path	**オブジェクトファイルの出力先のパス。ファイル名の前にスペースを含めることはできません。**

説明

オブジェクトファイル名に使用するパスを設定します。

デフォルトでは、パスは null であるため、オブジェクトのファイル名はソースファイル名と一致します。-O オプションを使用するとパスを指定でき、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

構文

-o {*filename*|*directory*}

パラメータ

filename	**オブジェクトコードは指定ファイルに格納されます。**
directory	**オブジェクトコードはファイル（ファイル拡張子 o）に格納され、このファイルは指定のディレクトリに格納されます。**

ファイル名やディレクトリの指定については、33 ページの コマンドラインアセンブラオプションの使用を参照してください。

説明

デフォルトでは、アセンブラで生成されたオブジェクトコードは、ソースファイルと同じ名前で、拡張子が o のファイルに配置されます。このオプションは、オブジェクトコードに別の出力ファイル名を指定する場合に使用します。

-o オプションを -O オプションと同時に使用することはできません。

[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

構文

-p*lines*

パラメータ

lines	**ページあたりの行数 (10 ～ 150)。**

説明

このオプションを使用して、ページあたりの行数を明示的に設定します。

このオプションは、-L オプションまたは -l オプションとともに使用します。

関連項目

42 ページの -L。

［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

構文

```
-r
```

説明

このオプションを使用して、アセンブラでデバッグ情報を生成するようにします。つまり、生成された出力を IAR C-SPY® デバッガなどのシンボリックデバッガで使用できます。

［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

構文

```
-S
```

説明

デフォルトでは、さまざまな重要ではないメッセージが標準出力ストリームから送信されます。このオプションは、標準出力ストリームにメッセージ送信せずにアセンブラで処理を実行するときに使用します。

エラーおよび警告メッセージはエラー出力ストリームに送信されるため、この設定にかかわらず表示されます。

このオプションは、IDE では使用できません。

-s

構文

```
-s{+|-}
```

パラメータ

+	ユーザシンボルの大文字 / 小文字を区別します。
-	大文字 / 小文字が区別されないユーザシンボルです。

説明

このオプションを使用して、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どうかを設定します。デフォルトでは、大文字と小文字が区別されます。

例

たとえば、デフォルトでは LABEL と label は異なるシンボルを示します。-s を使用すると、LABEL と label は同じシンボルを示すようになります。

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

--system_include_dir

構文

```
--system_include_dir path
```

パラメータ

path	**システムインクルードファイルのパス。**

ファイル名やディレクトリの指定については、33 ページの コマンドラインアセンブラオプションの使用を参照してください。

説明

デフォルトでは、アセンブラは自動的にシステムインクルードファイルを検索します。このオプションを使用して、システムインクルードファイルの異なるパスを明示的に指定します。これは、デフォルトの位置に IAR Embedded Workbench をインストールしていない場合に便利です。

このオプションは、IDE では使用できません。

-t

構文

```
-tn
```

パラメータ

n	**タブの間隔** (2 ～ 9)。

説明

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。このオプションは、異なるタブの間隔を指定するときに使用します。

このオプションは、-L オプションまたは -l オプションとともに使用すると便利です。

関連項目

42 ページの -L。

［プロジェクト］>［オプション］>［アセンブラ］>［リスト］>［タブ間隔］

-U

構文

```
-Usymbol
```

パラメータ

symbol	定義を取り消す定義済シンボル。

説明

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。

このオプションを使用すると、こうした定義済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__
```

も参照してください

26 ページの 定義済シンボル。

このオプションは、IDE では使用できません。

-w

構文

```
-w[+|-|+n|-n|+m-n|-m-n][s]
```

パラメータ

パラメータはありません	すべての警告を無効にします。
+	すべてのワーニングを有効にします。
-	すべてのワーニングを無効にします。
+n	ワーニング *n* のみを有効にします。
-n	ワーニング *n* のみを無効にします。
+m-n	*m* から *n* のワーニングを有効にします。
-m-n	*m* から *n* のワーニングを無効にします。
`s`	ワーニングメッセージが生成された場合に、終了コード `1` を生成します。デフォルトでは、ワーニングにより終了コード `0` が生成されます。

説明

デフォルトでは、構文上は正しくてもプログラムエラーを含む可能性のあるエレメントをアセンブラがソースコード中に発見すると、ワーニングメッセージが表示されます。

このオプションを使用して、すべてのワーニングや 1 つのワーニング、特定範囲のワーニングを無効にします。

-w オプションは、コマンドライン上で 1 度しか使用できない点に注意してください。

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、次のコマンドを使用します。

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

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、次のコマンドを使用します。

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

関連項目

137 ページの アセンブラの診断。

関連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

構文

```
-x{D|I|2}
```

パラメータ

D	プリプロセッサ #defines をインクルードします。
I	内部シンボルをインクルードします。
2	2 行間隔をインクルードします。

説明

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 -l オプションとともに使用すると便利です。

関連項目

42 ページの -L。

[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、最初にアセンブラ演算子の優先順位について説明してから、各演算子について優先順に簡単に説明します。最後に、各演算子のリファレンス情報をアルファベット順に提供します。

アセンブラ演算子の優先順位

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。優先順位の範囲は 1（最高の優先順位であり、最初に評価される）から 7（最低の優先順位であり、最後に評価される）までです。

以下の規則により、式がどのように評価されるかが決まります。

- 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。
- 括弧「(」と「)」を、演算子およびオペランドのグループ化と、式の評価順序の変更のために使用できます。たとえば、以下の式の評価結果は 1 です。

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

アセンブラ演算子の概要

以下の表は、演算子を優先順にまとめたものです。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。

括弧演算子 – 1

()	括弧

単項演算子 – 1

+	単項プラス
–	単項マイナス
!	論理否定
~	ビット単位の否定

LOW	下位バイト
HIGH	上位バイト
BYTE1	1 バイト目
BYTE2	2 バイト目
BYTE3	3 バイト目
BYTE4	4 番目のバイト
LWRD	下位ワード
HWRD	上位ワード
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
SIZEOF	セクションのサイズ

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
>	より大きい
<	より小さい
UGT	符号なしの「より大きい」
ULT	符号なしの「より小さい」
>=	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	優先順位	演算子	優先順位	関数
:AND:	3	&	5	ビットごとの論理積
:EOR:	3	^	6	ビットごとの排他的論理和
:LAND:	8	&&	5	論理積
:LEOR:	8	XOR	6	排他的論理和
:LNOT:	1	!	1	論理否定
:LOR:	6	||	6	論理和
:MOD:	2	%	2	剰余
:NOT:	1	~	1	ビット単位の否定
:OR:	3	|	6	ビットごとの論理和

表 12: 演算子の同義語

演算子の同義語	優先順位	演算子	優先順位	関数
:SHL:	2.5	<<	4	論理左シフト
:SHR:	2.5	>>	4	論理右シフト

表 12: 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。場合によっては、ARM 演算子と演算子の同義語で優先順位が異なることがあります。また *ARM 用 IAR アセンブラへの移行*も参照してください。

アセンブラ演算子の説明

ここでは、それぞれのアセンブラ演算子について詳しく説明します。括弧内の数字は、演算子の優先順位を示します。

関連情報については、*22 ページの式*、*オペランド*、*演算子*を参照してください。

() 括弧 (1)

説明 「(」と「)」は、独立して評価する式をグループ化し、デフォルトの優先順位より優先されます。

例
```
1+2*3 → 7
(1+2)*3 → 9
```

* 乗算 (2)

説明 * は 2 つのオペランドによる積を計算します。オペランドは符号付きの 32 ビット整数として処理され、結果も符号付きの 32 ビット整数となります。

例
```
2*2 → 4
-2*2 → -4
```

+ 単項プラス (1)

説明 単項プラス演算子。

例
```
+3 → 3
3*+2 → 6
```

+ 加算 (3)

説明

+ 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。オペランドは符号付きの 32 ビット整数として処理され、結果も符号付きの 32 ビット整数となります。

例

```
92+19 → 111
-2+2 → 0
-2+-2 → -4
```

– 単項マイナス (I)

説明

単項マイナス演算子は、オペランドを算術的に論理否定します。

オペランドは符号付きの 32 ビット整数として解釈され、演算子の結果はその整数の 2 の補数の論理否定となります。

例

```
-3 → -3
3*-2 → -6
4--5 → 9
```

– 減算 (3)

説明

減算演算子は、左のオペランドから右のオペランドを引いた差異を計算します。オペランドは符号付きの 32 ビット整数として処理され、結果も符号付きの 32 ビット整数となります。

例

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

/ 除算 (2)

説明

/ 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。オペランドは符号付きの 32 ビット整数として処理され、結果も符号付きの 32 ビット整数となります。

例

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

< 小なり (7)

説明

< は、左のオペランドの数値が右のオペランドより小さい場合に 1（真）となり、それ以外の場合は 0（偽）となります。

例

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

<= 以下 (7)

説明

<= は、左のオペランドの数値が右のオペランドより小さいか等しい場合に 1（真）となり、それ以外の場合は 0（偽）となります。

例

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

<>, != 等しくない (7)

説明

<> は、2 つの演算子の値が等しい場合に 1（偽）となり、等しくない場合は 0（真）となります。

例

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

=、== 等しい (7)

説明

= は、2 つの演算子の値が等しい場合に 1（真）となり、等しくない場合は 0（偽）となります。

例

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

> 大なり (7)

説明

> は、左のオペランドの数値が右のオペランドより大きい場合に 1（真）となり、それ以外の場合は 0（偽）となります。

例

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

>= 以上 (7)

説明

>= は、左のオペランドの数値が右のオペランドと等しいか、それより大きい場合に 1（真）となり、それ以外の場合は 0（偽）となります。

例

```
1 >= 2 → 0
2 >= 1 → 1
1 >= 1 → 1
```

&& 論理積 (5)

説明

&& または同義語の :LAND: を使用して、2 つの整数オペランドの論理積を計算します。両方のオペランドがゼロ以外である場合、計算結果は 1（真）となり、それ以外の場合は 0（偽）となります。

注： :LAND: の優先順位は 8 です。

例

```
1010B && 0011B → 1
1010B && 0101B → 1
1010B && 0000B → 0
```

& ビット単位の論理積 (5)

説明

& または同義語の :AND: を使用して、整数オペランドのビット単位の論理積を計算します。32 ビットの結果の各ビットは、オペランドの該当するビットの論理積です。

注： :AND: の優先順位は 3 です。

例

```
1010B & 0011B → 0010B
1010B & 0101B → 0000B
1010B & 0000B → 0000B
```

~ ビット単位の否定 (1)

説明

~ または同義語の :NOT: を使用して、オペランドでビット単位の否定を計算します。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

```
~ 1010B → 11111111111111111111111111110101B
```

| ビット単位の論理和 (6)

説明

| または同義語の :OR: を使用して、オペランドでビット単位の論理和を計算します。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

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

^ ビット単位の排他的論理和 (6)

説明

^ または同義語の :EOR: を使用して、オペランドでビット単位の排他的論理和を計算します。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

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

% 剰余 (2)

説明

% または同義語の :MOD: により、左のオペランドを右のオペランドで整数除算した余りを計算します。オペランドは符号付きの 32 ビット整数として処理され、結果も符号付きの 32 ビット整数となります。

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
```

! 論理否定 (1)

説明

! または同義語の :LNOT: を使用して、論理引数を否定します。

例

```
! 0101B → 0
! 0000B → 1
```

|| 論理和 (6)

説明

|| または同義語の :LOR: を使用して、2 つの整数オペランドの論理和を計算します。

例

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

<< 論理左シフト (4)

説明

<< または同義語の :SHL: を使用して、常に符号なしとして扱われる左オペランドを左側にシフトします。シフト対象のビット数は、右オペランドで指定し、0 ～ 32 の整数値として解釈されます。

注： :SHL: の優先順位は 2.5 です。

例

```
00011100B << 3 → 11100000B
0000011111111111B << 5 → 1111111111100000B
14 << 1 → 28
```

>> 論理右シフト (4)

説明

>> または同義語の :SHR: を使用して、常に符号なしとして扱われる左オペランドを右側にシフトします。シフト対象のビット数は、右オペランドで指定し、0 ～ 32 の整数値として解釈されます。

注： :SHR: の優先順位は 2.5 です。

例

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

BYTE1 1 バイト目 (1)

説明 BYTE1 は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。結果は、そのオペランドの下位オーダバイトの符号なし 8 ビット整数値です。

例 `BYTE1 0xABCD → 0xCD`

BYTE2 2 バイト目 (1)

説明 BYTE2 は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。この結果は、オペランドの中下位バイト（ビット 15 ～ 8）です。

例 `BYTE2 0x12345678 → 0x56`

BYTE3 3 バイト目 (1)

説明 BYTE3 は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。この結果は、オペランドの中上位バイト（ビット 23 ～ 16）です。

例 `BYTE3 0x12345678 → 0x34`

BYTE4 4 バイト目 (1)

説明 BYTE4 は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。この結果は、オペランドの上位バイト（ビット 31 ～ 24）です。

例 `BYTE4 0x12345678 → 0x12`

DATE 現在の日時 (1)

説明 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 演算子は絶対引数（式）をとり、以下を返します。

`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`	現在の西暦年の下 2 桁 (1998 →98、2000 →00、2002 →02)

例

アセンブリ日時は以下のようにアセンブルします。

```
today: DC8 DATE 5, DATE 4, DATE 3
```

HIGH 上位バイト (I)

説明

`HIGH` は、符号なし 16 ビット整数値として解釈される、右側にある単一のオペランドを取得します。結果は、そのオペランドの上位オーダバイトの符号なし 8 ビット整数値です。

例

`HIGH 0xABCD → 0xAB`

HWRD 上位ワード (I)

説明

`HWRD` は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。この結果は、オペランドの上位ワード（ビット 31 ～ 16）です。

例

`HWRD 0x12345678 → 0x1234`

LOW 下位バイト (I)

説明

`LOW` は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。結果は、そのオペランドの下位オーダバイトの符号なし 8 ビット整数値です。

例

`LOW 0xABCD → 0xCD`

LWRD 下位ワード (I)

説明

`LWRD` は、符号なし 32 ビット整数値として解釈される単一のオペランドを取得します。この結果は、オペランドの下位ワード（ビット 15 ～ 0）です。

例

`LWRD 0x12345678 → 0x5678`

SFB セクション 開始 (I)

構文

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

パラメータ

section	セクションの名前。SFB の使用前に定義する必要があります。
offset	開始アドレスからの任意指定オフセット。*offset* を省略するときは丸括弧はオプションです。

説明

SFB は、右側にある単一のオペランドを受け入れます。この演算子の評価結果は、セクションの最初のバイトのアドレスです。この評価はリンク時に行われます。

例

```
            name    sectionBegin
            section MYCODE:CODE(2)  ; MYCODE の前方に宣言
            section MYCONST:CONST(2)
            data
start       dc32    sfb(MYCODE)
            end
```

このコードがその他多くのモジュールとリンクされている場合でも、start はセクション MYCODE の最初のバイトのアドレスに設定されます。

SFE セクション終了 (I)

構文

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

パラメータ

section	セクションの名前。SFE の使用前に定義する必要があります。
offset	開始アドレスからの任意指定オフセット。offset を省略するときは丸括弧はオプションです。

説明

SFE は、右側にある単一のオペランドを受け入れます。この演算子の評価結果は、セクションの最後にくるにくる最初のバイトのアドレスです。この評価はリンク時に行われます。

例

```
                name    sectionEnd
                section MYCODE:CODE(2)  ; MYCODE の前方に宣言
                section MYCONST:CONST(2)
                data
end             dc32    sfe(MYCODE)
                end
```

このコードがその他多くのモジュールとリンクしている場合でも、end はセクション MYCODE の後にくる最初のバイトに設定されます。

セクション MYCODE のサイズは、SIZEOF 演算子。

SIZEOF セクションサイズ (I)

構文

SIZEOF *section*

パラメータ

section	再配置可能セクションの名前。SIZEOF の使用前に定義する必要があります。

説明

SIZEOF は、引数として SFE-SFB を生成します。つまり、セクションのバイト数でサイズを計算します。この計算は、モジュール同士がリンクされると行われます。

例

これら 2 つのファイルは、size の値を、セクション MYCODE のサイズに設定します。

Table.s:

```
                module  table
                section MYCODE:CODE  ; MYCODE の前方に宣言
                section SEGTAB:CONST(2)
                data
size            dc32    sizeof(MYCODE)
                end
```

Application.s:

```
                module  application
                section MYCODE:CODE(2)
                nop                 ; application のための
                                    ; プレースフォルダ
                end
```

UGT 符号なし大なり (7)

説明

UGT は、左のオペランドの数値が右のオペランドより大きい場合に 1（真）となり、それ以外の場合は 0（偽）となります。この演算では、オペランドを符号なしの値として取り扱います。

例

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT 符号なし小なり (7)

説明

ULT は、左のオペランドの数値が右のオペランドより小さい場合に 1（真）となり、それ以外の場合は 0（偽）となります。この演算では、オペランドを符号なしの値として取り扱います。

例

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR 排他的論理和 (6)

説明

XOR または同義語の :LEOR: は、左右いずれかのオペランドがゼロ以外である場合に 1（真）となり、両方のオペランドがゼロまたはゼロ以外のときに 0（偽）となります。XOR は 2 つのオペランドの排他的論理和を計算します。

注： :LEOR: の優先順位は 8 です。

例

```
0101B XOR 1010B → 0
0101B XOR 0000B → 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についてアルファベット順に簡単に説明し、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提供します。

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、機能に応じて以下のようにグループ分けされます。

- *70 ページの モジュール制御ディレクティブ*
- *73 ページの シンボル制御ディレクティブ*
- *75 ページの モード制御のディレクティブ*
- *78 ページの セクションの制御ディレクティブ*
- *81 ページの 値割当てディレクティブ*
- *84 ページの 条件付きアセンブリディレクティブ*
- *86 ページの マクロ処理ディレクティブ*
- *94 ページの リスト制御ディレクティブ*
- *99 ページの C 形式のプリプロセッサディレクティブ*
- *104 ページの データ定義ディレクティブまたは割当てディレクティブ*
- *107 ページの アセンブラ制御ディレクティブ*
- *111 ページの 呼出しフレーム情報ディレクティブ*

以下の表に、すべてのアセンブラディレクティブの概要を示します。

ディレクティブ	説明	セクション
`_args`	マクロに受け渡される引数の数に設定されます。	マクロ処理
`$`	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` ブロックに新しい条件を実装します。	C 形式のプリプロセッサ
`#else`	条件が偽の場合に命令をアセンブルします。	C 形式のプリプロセッサ

表 13: 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
#endif	#if、#ifdef、または #ifndef ブロックを終了させます。	C 形式のプリプロセッサ
#error	エラーを生成します。	C 形式のプリプロセッサ
#if	条件が真の場合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場合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場合に命令をアセンブルします。	C 形式のプリプロセッサ
#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	行番号を変更します。	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	ラベルの定義を取り消します。	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。	アセンブラ制御
//	C++形式のコメント区切り文字。	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。	値の割当て
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。	値の割当て
ALIGN	ゼロが埋め込まれたバイトを挿入して、プログラムロケーションカウンタをアラインメントします。	セクションの制御
ALIGNRAM	プログラムロケーションカウンタをアラインメントします。	セクションの制御
ALIGNROM	ゼロが埋め込まれたバイトを挿入して、プログラムロケーションカウンタをアラインメントします。	セクション制御

表 13: 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
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	一時値を割り当てます。	値の割当て
CASEOFF	大文字 / 小文字の区別を無効にします。	アセンブラ制御
CASEON	大文字 / 小文字の区別を有効にします。	アセンブラ制御
CFI	呼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。THUMB に置き換わりました。	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。ARM に置き換わりました。	モード制御
COL	ページあたりのカラム数を設定します。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
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	文字列を含め 8 ビットの定数を生成します。	データ定義または割当て
DC16	16 ビットの定数を生成します。	データ定義または割当て
DC24	24 ビットの定数を生成します。	データ定義または割当て
DC32	32 ビットの定数を生成します。	データ定義または割当て
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当て
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当て
DEFINE	ファイル全体で有効な値を定義します。	値の割当て
DF32	32 ビットの浮動小数点定数を生成します。	データ定義または割当て
DF64	64 ビットの浮動小数点定数を生成します。	データ定義または割当て

表 13: 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
DS8	8 ビット整数に空間を割り当てます。	データ定義または割当て
DS16	16 ビット整数に空間を割り当てます。	データ定義または割当て
DS24	24 ビット整数に空間を割り当てます。	データ定義または割当て
DS32	32 ビット整数に空間を割り当てます。	データ定義または割当て
ELSE	条件が偽の場合に命令をアセンブルします。	条件付きアセンブリ
ELSEIF	IFENDIF ブロックに新しい条件を指定します。	条件付きアセンブリ
END	ファイル内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。	値の割当て
EVEN	偶数アドレスにプログラムカウンタをアラインメントします。	セクションの制御
EXITM	マクロが終了する前に抜け出します。	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
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	条件が真の場合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	マクロに対してローカルなシンボルを作成します。	マクロ処理
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	複数行からなるコードのリストを制御します。	リスト制御

表 13: 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
LSTEXP	マクロで生成された行のリストを制御します。	リスト制御
LSTMAC	マクロ定義のリストを制御します。	リスト制御
LSTOUT	アセンブラリスト出力を制御します。	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	繰返しディレクティブで生成された行のリストを制御します。	リスト制御
LSTSAS	構造化アセンブリリストを制御します。	リスト制御
LSTXRF	クロスリファレンス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を開始します。	モジュール制御
ODD	奇数アドレスにプログラムロケーションカウンタをアラインメントします。	セクションの制御
OVERLAY	認識されますが、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。複数の定義が許可されます。	シンボル制御
RADIX	デフォルトベースを設定します。	アセンブラ制御
REPT	指定回数だけ命令を繰り返します。	マクロ処理
REPTC	文字を繰り返し、置換します。	マクロ処理
REPTI	文字列を繰り返し、置換します。	マクロ処理
REQUIRE	シンボルを強制参照させます。	シンボル制御
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

表 13: 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
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	一時値を割り当てます。	値の割当て
SETA	一時値を割り当てます。	値の割り当て
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	一時値を割り当てます。	値の割当て

表 13: 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

モジュール制御ディレクティブは、ソースプログラムモジュールの開始と終了をマーキングし、それらのモジュールに名前を割り当てるために使用されます。式でディレクティブを使用する際に適用される制限については、*29 ページの式の制限*を参照してください。

ディレクティブ	説明	式の制限
AAPCS	エクスポートされたモジュール内のすべての関数が、AAPCS(Procedure Call Standard for the ARM Architecture) に従うことをリンカに指示するモジュール属性を設定します。	アセンブラでは、要求が満たされているかどうか検証されません。
END	ファイル内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	モジュールを開始します。PROGRAM のエイリアスです。	外部参照禁止 絶対
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カに通知するモジュール属性を設定します。	アセンブラでは、要求が満たされているかどうか検証されません。
PROGRAM	モジュールを開始します。	外部参照禁止 絶対

表 14: モジュール制御ディレクティブ

ディレクティブ	説明	式の制限
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カ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。	なし

表 14: モジュール制御ディレクティブ（続き）

構文

```
AAPCS [modifier [...]]
END
NAME symbol
PRESERVE8
PROGRAM symbol
REQUIRE8
RTMODEL key, value
```

パラメータ

key	キーを指定するテキスト文字列。
modifier	AAPCS の拡張。使用可能な値は INTERWORK、VFP、VFP_COMPATIBLE、ROPI、RWPI、RWPI_COMPATIBLE です。修飾子を組み合わせて AAPCS の派生形を指定できます。
symbol	モジュールに対して割り当てられ。
value	値を指定するテキスト文字列。

説明

モジュールの開始

ELF モジュールを開始して名前を割り当てるには、ディレクティブ NAME か PROGRAM を使用します。

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 用 IAR C/C++ 開発ガイド』*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

ソースファイルの最後を指定するには、END を使用します。END ディレクティブの後の行はすべて無視されます。ENDMOD ディレクティブで明示的に指定されていない場合でも、END ディレクティブを使用してを終了できます。

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 と修飾子（オプション）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ので、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。一緒にリンクされ、同一のランタイムモジュール属性のキーを定義するすべてのモジュールは、そのキー値に対応する値が同一であるか、特殊な * という値を持つ必要があります。特殊値 * を使用すると、属性が未定義である場合と等価になります。ただし、この値を使用することで、モジュールがランタイムモデルに対応していることを明示できます。

1 つのモジュールで複数のランタイムモデルを定義できます。

注： コンパイラランタイムモデル属性は、最初がダブルアンダースコアになります。混乱を避けるため、ユーザ定義アセンブラ属性ではこのスタイルを使用しないでください。

C/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する場合は、*『ARM 用 IAR C/C++ 開発ガイド』* を参照してください。

例

以下の例は、1 つのソース行に 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 CAN の値が異なるため、一緒にリンク*できません*。

- MOD_1 と MOD_3 は、ランタイムモデル RTOS の定義が同じであり、CAN の定義に矛盾がないため、一緒にリンク できます。
- MOD_2 と MOD_3 は、ランタイムモデルの矛盾がないため、一緒にリンク できます。値 * は、任意のランタイムモデル値に一致します。

アセンブラソースファイル f1.s:

```
module  mod_1
rtmodel "CAN",      "ISO11519"
rtmodel "Platform", "M7"
; ...
end
```

アセンブラソースファイル f2.s:

```
module  mod_2
rtmodel "CAN",      "ISO11898"
rtmodel "Platform", "*"
; ...
end
```

アセンブラソースファイル f3.s:

```
module  mod_3
rtmodel "Platform", "M7"
; ...
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、モジュール間でシンボルがどのように共有されるかを制御します。

ディレクティブ	説明
EXTERN, 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。
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。複数の定義が許可されます。
REQUIRE	シンボルを強制参照させます。

表 15: シンボル制御ディレクティブ

構文

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

パラメータ

symbol	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

1 つ以上のシンボルを他のモジュールで使用できるようにするには、PUBLIC を使用します。PUBLIC として定義されたシンボルは、再配置可能または絶対であり、（他のシンボルと同様の規則に従って）式の中で使用することもできます。

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 ビットプロセッサおよび 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、>>、<< 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 ビットのレジスタまたはワードに読み込むことができます。

1 つのモジュールには任意の数の PUBLIC 定義シンボルを使用できます。

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、複数のモジュールで同じシンボルを定義できます。これらの定義のいずれか 1 つのみが ILINK に使用されます。シンボルの PUBLIC 定義が含まれるモジュールが、同じシンボルの PUBWEAK 定義が含まれる 1 つ以上のモジュールにリンクされている場合、ILINK は PUBLIC 定義を使用します。

注：ライブラリモジュールへのリンクは、そのモジュール内のシンボルへの参照が行われ、シンボルがまだリンクされていない場合にのみ行われます。モジュール選択フェーズでは、PUBLIC 定義と PUBWEAK 定義は区別されません。つまり、PUBLIC 定義の含まれるモジュールが選択されていることを確認するためには、これを他のモジュールより前にリンクするか、そのモジュール内で他の PUBLIC シンボルへの参照が行われていることを確認する必要があります。

シンボルのインポート

型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 ディレクティブにより、シンボルが参照済としてマーキングされます。コードが参照されなくても、シンボルを含むセクションをロードしなければならないときに、これは便利です。

例

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

メッセージは二重引用符に囲まれているため、文字列の後にはゼロバイトが挿入されます。

print は外部ルーチンとして定義されており、アドレスはリンク時に解決されます。

```
            name    errorMessage
            extern  print
            public  err

            section MYCODE:CODE(2)
            arm

err         adr     r0,msg
            bl      print
            bx      lr

            data
msg         dc8     "** Error **"

            end
```

モード制御のディレクティブ

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。

表 16: モード制御のディレクティブ

ディレクティブ	説明
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. で規定されているように Unified Assembler 構文に従っています。

表 16: モード制御のディレクティブ（続き）

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、BX（Branch and Exchange）命令で CODE16/THUMB および CODE32/ARM ディレクティブを使用するか、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。

モードディレクティブ CODE32 と CODE16 の使用は廃止予定です。代わりに、ARM と THUMB をそれぞれ使用してください。

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 用 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 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。

```
            name    modeChange
            section MYCODE:CODE(2)
            thumb
thumbEntry
            bx      pc              ; armEntry にブランチし
                                    ; 実行モードを切り替える
            nop                     ; アラインメントの目的のみ
            arm
armEntry
            ; ...

            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 THUMB セクション内で使用できます。

```
            name    dataDirective
            section MYCODE:CODE(2)
            thumb
thumbLabel  ldr     r0,dataLabel
            bx      lr

            data                    ; data モードに切り替え、
                                    ; label のビット 0 がセットされな
                                    ; いようにする。
dataLabel   dc32    0x12345678
            dc32    0x12345678

            end
```

セクションの制御ディレクティブ

セクションディレクティブは、コードと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際に適用される制限については、*29 ページの式の制限*を参照してください。

ディレクティブ	説明	式の制限
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。	外部参照禁止 絶対
ALIGNROM	ゼロが埋め込まれたバイトを挿入して、プログラムロケーションカウンタをアラインメントします。	外部参照禁止 絶対
EVEN	偶数アドレスにプログラムカウンタをアラインメントします。	外部参照禁止 絶対
ODD	奇数アドレスにプログラムカウンタをアラインメントします。	外部参照禁止 絶対
RSEG	ELF セクションを開始します。これは SECTION のエイリアスです。	外部参照禁止 絶対
SECTION	ELF セクションを開始します。	外部参照禁止 絶対
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 17: 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 segment :type [:flag] [(align)]
SECTION_TYPE type-expr {,flags-expr}
```

パラメータ

align アドレスをアラインメントする 2 の累乗。有効な範囲は 0 ～ 8 です。
整列のデフォルト値は 0 で、コードセクションの場合はデフォルト値は 1 です。

flag

ROOT、NOROOT

ROOT（デフォルトモード）は、セクション フラグメントを破棄してはならないことを示します。

NOROOT は、このセクション フラグメント内のシンボルが参照されない場合、セクション フラグメントがリンカにより破棄されることを意味します。通常、起動コードと割込みベクタを除くすべてのセクション フラグメントで、このフラグを設定する必要があります。

REORDER、NOREORDER

NOREORDER（デフォルトモード）は、指定の名前のセクションで新しいフラグメントを開始します。該当するセクションがなければ、新しいセクションでフラグメントが開始されます。

REORDER は、指定した名前で新しいセクションを開始します。

section

セクションの名前。セクション名は、*24 ページの シンボル* で説明する規則に従うユーザ定義のシンボルです。

type

CODE、CONST、DATA のメモリタイプ。

value

パディングに使用されるバイト値。デフォルトはゼロです。

type-expr

セクションの ELF タイプを識別する定数式。

flags-expr

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。アセンブラは別々のロケーションカウンタ（開始時の設定はゼロ）をすべてのセクションに対して管理しています。これにより、セクションやモードをいつでも自由に切り替えることができ、現在のプログラムロケーションカウンタを保存する必要はありません。

注： 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、DC8 や DS8 など、ディレクティブを生成するコード、あるいはアセンブラ命令を付けないでください。

ELF タイプ、また該当する場合は新しく作成されるセクションの 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。

セクションのアラインメント

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。プログラムカウンタを整列する 2 の累乗に式を指定することにより、これを行います。つまり、値を 1 にすると偶数アドレスに、2 の場合は 4 で均等に分割可能なアドレスに整列されます。

アラインメントは、セクション先頭に対して相対的に行われます。つまり、通常、必要な結果を得るためには、セクションアラインメントは少なくともアラインメントディレクティブと同じ大きさでなければなりません。

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。埋め込みのバイト値は 0 ～ 255 の範囲内である必要があります。

指定されたアドレス境界に対してプログラムロケーションカウンタをアラインメントするには、ALIGNRAM を使用します。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。

RAM と ROM のどちらの場合でも、パラメータ align の有効な範囲は 0 ～ 30 です。

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クションに配置されます。

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。

```
            name    calculate
            extern  subrtn,divrtn

            section MYDATA:DATA (2)
            data
funcTable   dc32    subrtn
            dc32    divrtn

            section MYCODE:CODE(2)
            arm
```

```
main        ldr     r0,=funcTable   ; アドレスを取り出して
            ldr     pc,[r0]         ; そこにジャンプする。
            end
```

セクションのアラインメント

この例は、セクションを開始して、何らかのデータを追加します。続いて、64 バイト境界へのアラインメントを行ってから、64 バイトテーブルを作成します。このセクションでは、テーブルの 64 バイトのアラインメントを確実にするため、アラインメントは 64 バイトです。

```
            name    alignment
            section MYDATA:DATA(6)  ; 64 バイト境界へアラインメント
                                    ; された再配置可能
                                    ; データセクションの開始
            data
target1     ds16    1               ; 2 バイトのデータ
            alignram 6              ; 64 バイト境界にアラインメント
results     ds8     64              ; 64 バイトのテーブルを作り、
target2     ds16    1               ; さらに 2 バイト加える
            alignram 3              ; 8 バイト境界にアラインメントして
ages        ds8     64              ; 別の 64 バイトテーブルを作成
            end
```

値割当てディレクティブ

これらのディレクティブは、シンボルへの値の割当てに使用します。

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 18: 値割当てディレクティブ

構文

label = *expr*

label ALIAS *expr*

label ASSIGN *expr*

label DEFINE *const_expr*

label EQU *expr*

```
label SET expr
label SETA expr
label VAR expr
```

パラメータ

const_expr	シンボルに割り当てられる定数値。
expr	シンボルに割り当てられる値またはテスト対象の値。
label	定義されるシンボル。

説明

一時値の定義

マクロ変数で使用するなどの目的で再定義が必要になる可能性があるシンボルを定義するには、ASSIGN、SET、または VAR を使用します。ASSIGN、SET、または VAR で定義されたシンボルを PUBLIC として宣言することはできません。

ローカルな永久値の定義

数値またはオフセットを指定するローカルシンボルを作成するには、EQU または = を使用します。シンボルはそれが定義されたモジュール内のみで有効ですが、PUBLIC ディレクティブを使用すれば（PUBWEAK ディレクティブは不可）、他のモジュールでも使用できるようになります。

他のモジュールからシンボルをインポートするには、EXTERN 使用します。

グローバルな永久値の定義

ディレクティブが含まれるモジュールや、同じソースファイル内でそのモジュールの後に続くすべてのモジュールで認識されるシンボルは、DEFINE ディレクティブの後で認識されます。

DEFINE によって値が提供されるシンボルは、PUBLIC ディレクティブによって他のファイル内のモジュールでも使用可能にすることができます。

DEFINE によって定義されたシンボルは、同じファイル内に再定義できます。また、定義されたシンボルに割り当てられた式は定数値でなければなりません。

例

シンボルの再定義

以下の例では、SET を使用して cons というシンボルをループに再定義し、3 の累乗値を順に 8 つ含むテーブルを生成します。

```
            name    table
cons        set     1

; Generate table of powers of 3.
cr_tabl     macro   times
            dc32    cons
cons        set     cons * 3
            if      times > 1
            cr_tabl times - 1
            endif
            endm

            section .text:CODE(2)
table       cr_tabl 4
            end
```

これにより、以下のコードが生成されます。

```
    9                                       name    table
   10                           cons        set     1
   11
   12                                ; Generate table of powers of 3.
   20
   21                                       section .text:CODE(2)
   22                           table       cr_tabl 4
   22    00000000 01000000      table       cr_tabl 4
   22.1                         cons        set     cons * 3
   22.2                                     if      4 > 1
   22                                       cr_tabl 4 - 1
   22    00000004 03000000      table       cr_tabl 4
   22.1                         cons        set     cons * 3
   22.2                                     if      4 - 1 > 1
   22                                       cr_tabl 4 - 1 - 1
   22    00000008 09000000      table       cr_tabl 4
   22.1                         cons        set     cons * 3
   22.2                                     if      4 - 1 - 1 > 1
   22                                       cr_tabl 4 - 1 - 1 - 1
   22    0000000C 1B000000      table       cr_tabl 4
   22.1                         cons        set     cons * 3
   22.2                                     if      4 - 1 - 1 - 1 > 1
   22.3                                     cr_tabl 4 - 1 - 1 - 1 - 1
   22.4                                     endif
```

```
22.5                                    endm
22.6                                    endif
22.7                                    endm
22.8                                    endif
22.9                                    endm
22.10                                   endif
22.11                                   endm
23                                      end
```

条件付きアセンブリディレクティブ

これらのディレクティブにより、ソースコードの選択的なアセンブリを論理的に制御することができます。式でディレクティブを使用する際に適用される制限については、*29 ページの 式の制限*を参照してください。

ディレクティブ	説明	式の制限
ELSE	条件が偽の場合に命令をアセンブルします。	
ELSEIF	IFENDIF ブロックに新しい条件を指定します。	前方参照禁止 外部参照禁止 絶対 固定
ENDIF	IF ブロックを終了します。	
IF	条件が真の場合に命令をアセンブルします。	前方参照禁止 外部参照禁止 絶対 固定

表 19: 条件付きアセンブリディレクティブ

構文

```
ELSE
ELSEIF condition
ENDIF
IF condition
```

パラメータ

condition	以下のいずれかです	
	絶対式	式に、前方参照または外部参照を含めることはできず、ゼロ以外の値はすべて真と見なされます。

string1=string2	*string1* と *string2* の長さと内容が同じである場合に、この条件は真となります。
string1<>string2	*string1* と *string2* の長さまたは内容が異なる場合に、この条件は真となります。

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 ディレクティブの後の条件が真ではない場合、ELSE または ENDIF ディレクティブが検出されるまで、後続の命令はコードを一切生成しません（つまり、アセンブルも構文チェックも行われません）。

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内の任意の場所で使用できますが、マクロ処理と一緒に使用すると最も有用です。

（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、条件付きディレクティブで無効にできます。各 IF ディレクティブは ENDIF ディレクティブで終了する必要があります。ELSE ディレクティブは任意指定であり、使用する場合は、IF...ENDIF ブロック内に指定する必要があります。IF...ENDIF ブロックと IF...ELSE...ENDIF ブロックは、任意のレベルまでネストできます。

例

この例ではマクロを使用して、レジスタに定数を追加します。

```
?add        macro   a,b,c
            if      _args == 2
            adds    a,a,#b
            elseif  _args == 3
            adds    a,b,#c
            endif
            endm

            name    addWithMacro
            section MYCODE:CODE(2)
            arm

main        ?add    r1,0xFF         ; この行と、
            ?add    r1,r1,0xFF      ; この行は、
            adds    r1,r1,#0xFF     ; この行と同じ。

            end
```

マクロ処理ディレクティブ

これらのディレクティブを使用してマクロを定義できます。式でディレクティブを使用する際に適用される制限については、*29 ページの 式の制限*を参照してください。

ディレクティブ	説明	式の制限
_args	マクロに受け渡される引数の数に設定されます。	
ENDM	マクロ定義を終了します。	
ENDR	繰返し構造を終了します。	
EXITM	マクロが終了する前に抜け出します。	
LOCAL	マクロに対して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。	
REPT	指定回数だけ命令を繰り返します。	前方参照禁止 外部参照禁止 絶対 固定
REPTC	文字を繰り返し、置換します。	
REPTI	テキストを繰り返し、置換します。	

表 20: マクロ処理ディレクティブ

構文

```
_args
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	置換される文字列。
argument	シンボル引数名。
expr	式

formal	*actual* (REPTC) または各 *actual* (REPTI) 文字列で置換される引数。
name	マクロの名前。
symbol	マクロに対してローカルにするシンボル。

説明

マクロとは、1 行以上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シンボルです。マクロを定義すると、アセンブラディレクティブやアセンブラニーモニックのようにプログラム内でこのマクロを使用できるようになります。

アセンブラがマクロを検出すると、マクロの定義が検索され、ソースファイルの当該位置にそのマクロが含まれているかのように、マクロに記述されている行が挿入されます。

マクロは単純なテキスト置換を効率的に行います。マクロにパラメータを指定することで、置換対象を制御することができます。

マクロの定義

マクロの定義には以下の文を使用します。

name MACRO [*argument*] [,*argument*]

ここで、*name* はマクロに対して使用する名前、*argument* はマクロの展開時にマクロに受け渡す値の引数です。

たとえばマクロ errMacro を次のように定義できます。

```
            name    errMacro
            extern  abort
errMac      macro   text
            bl      abort
            data
            dc8     text,0
            endm
```

このマクロでは、パラメータ text（LR で渡されます）を使用して、abort というルーチンに対してエラーメッセージを設定しています。このマクロは、たとえば以下のような文で呼出します。

```
            section MYCODE:CODE(2)
            arm
            errMac  'Disk not ready'
```

アセンブラはこれを以下のように展開します。

```
            section MYCODE:CODE(2)
            arm
            bl      abort
            data
            dc8    'Disk not ready',0

            end
```

1 つ以上の引数から成るリストを省略すると、マクロを呼び出すときにユーザが指定する引数は ¥1 ～ ¥9 および ¥A ～ ¥Z と呼ばれます。

そのため、前の例は以下のように記述できます。

```
            name    errMacro
            extern  abort
errMac      macro
            bl      abort
            data
            dc8     ¥1,0
            endm
```

マクロが終了する前にマクロから抜け出すには EXITM ディレクティブを使用します。

EXITM は、REPT...ENDR、REPTC...ENDR、REPTI...ENDR の各ブロックの内部で使用できません。

マクロに対して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、シンボルの使用前に使用する必要があります。

マクロを展開するたびに、ローカルシンボルの新しいインスタンスが LOCAL ディレクティブによって作成されます。したがって、繰返しマクロ内でローカルシンボルを使用することができます。

注：マクロの*再定義*は不正です。

特殊文字の受渡し

マクロ呼出し内で引用符 < と > をペアで使用することにより、コンマや空間が含まれるマクロ引数を強制的に 1 つの引数として解釈させることができます。

次に例を示します。

```
            name    cmpMacro
cmp_reg     macro   op
            CMP     op
            endm
```

マクロは、マクロ引用符を使用して呼び出すことができます。

```
section MYCODE:CODE(2)
cmp_reg <r3,r4>
end
```

マクロ引用符は、コマンドラインオプション -M を使用して再定義できます。*43* ページの*-M* 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、マクロに引き渡される引数の数を設定します。以下の例は、_args の使用方法を示します。

```
fill        macro
            if      _args == 2
            rept    ¥2
            dc8     ¥1
            endr
            else
            dc8     ¥1
            endif
            endm

            module  filler
            section .text:CODE(2)
            fill    3
            fill    4, 3
            end
```

これにより、以下のコードが生成されます。

```
  19                                    module  filler
  20                                    section .text:CODE(2)
  21                                    fill    3
  21.1                                  if      _args == 2
  21.2                                  rept
  21.3                                  dc8     3
  21.4                                  endr
  21.5                                  else
  21    00000000 03                     fill    3
  21.1                                  endif
  21.2                                  endm
  22                                    fill    4, 3
  22.1                                  if      _args == 2
  22.2                                  rept    3
  22.3                                  dc8     4
  22.4                                  endr
  22    00000001 04                     dc8     4
```

```
22    00000002 04                    dc8     4
22    00000003 04                    dc8     4
22.1                                 else
22.2                                 dc8     4
22.3                                 endif
22.4                                 endm
23                                   end
```

マクロの処理方法

マクロプロセスは、以下の 3 つのフェーズで構成されます。

1 アセンブラはマクロ定義をスキャンし、保存します。MACRO と ENDM の間のテキストは保存されますが、構文はチェックされません。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 マクロ呼出しによりアセンブラはマクロプロセッサ（エクスパンダ）を起動します。マクロエクスパンダは、（マクロ内に存在しない場合）ソースファイルからのアセンブラ入力ストリームをマクロエクスパンダからの出力に切り替えます。マクロエクスパンダは、要求されたマクロ定義からの入力を取得します。

   マクロエクスパンダは、ソースレベルでのテキスト置換のみを処理するため、アセンブラシンボルを認識できません。呼び出されたマクロ定義からの行がアセンブラに受け渡される前に、エクスパンダはシンボルマクロ引数のすべてのオカレンスの行をスキャンし、展開引数に置換します。

3 その後、展開された行は、その他すべて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すべての行が読み込まれるまで、マクロプロセッサからの出力となります。

繰返し文

同じ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* の評価結果が 0 である場合、何も生成されません。

文字列の各文字に対して 1 回だけ命令ブロックをアセンブルするには、REPTC を使用します。文字列にコンマが含まれる場合、引用符で囲む必要があります。

特別な意味があるのは二重引用符のみであり、繰返し使用される文字を囲むためだけに使用されます。単一引用符には特別な意味はなく、通常の文字として処理されます。

一連の文字列内の各文字列に対して 1 回だけ命令ブロックをアセンブルするには、REPTI を使用します。文字列にコンマが含まれる場合、引用符で囲む必要があります。

例

ここでは、マクロによってアセンブラプログラミングを簡便化する方法の例をいくつか示します。

インラインコーディングによる効率化

時間が重要なコードでは、ルーチンをインラインコーディングすることによりサブルーチンの呼出しとリターンのオーバヘッドを避けることで、効率化を図ることができます。これはマクロを使用すると便利です。

以下の例では、バッファからポートへバイトが出力されます。

```
            name    ioBufferSubroutine
            section MYCODE:CODE(2)
            arm
play        ldr     r1,=buffer      ; buffer へのポインタ
            ldr     r2,=ioPort      ; ioPort へのポインタ
            ldr     r3,=512         ; buffer の大きさ
            add     r3,r3,r1        ; buffer の後の最初の
                                    ; バイトのアドレス
loop        ldrb    r4,[r1],#1      ; 1 バイトのデータを読み
            strb    r4,[r2]         ; ioPort に書く
            cmp     r1, r3          ; 1 バイト先に到着したか？
            bne     loop            ; No: 繰り返し
            bx      lr              ; リターン

ioPort      equ     0x0100

            section MYDATA:DATA (2)
            data
buffer      ds8     512             ; 512 バイト確保

            section MYCODE:CODE(2)
            arm
main        bl      play
done        b       done

            end
```

効率化のために、マクロを使用した再コーディングできます。

```
            name    ioBufferInline
play        macro   buf,size,port
            local   loop
            ldr     r1,=buf         ; buffer へのポインタ
            ldr     r2,=port        ; ioPort へのポインタ
            ldr     r3,=size        ; buffer の大きさ
```

```
            add     r3,r3,r1        ; buffer の後の最初の
                                    ; バイトのアドレス
loop        ldrb    r4,[r1],#1      ; 1 バイトのデータを読み
            strb    r4,[r2]         ; ioPort に書く。
            cmp     r1, r3          ; 1 バイト先に到着したか ?
            bne     loop            ; No: 繰り返し
            endm

ioPort      equ     0x0100

            section MYDATA:DATA (2)
            data
buffer      ds8     512             ; 512 バイト確保

            section MYCODE:CODE(2)
            arm
main        play    buffer,512,ioPort
done        b       done

            end
```

loop ラベルをマクロに対してローカルにするために、LOCAL ディレクティブが使用されています。さもなければ、loop ラベルは既に存在しているため、マクロが 2 回使用されるとエラーが生成されます。

REPTC および REPTI の使用

以下の例は、文字列内の各文字をプロットするために、サブルーチン plotc への一連の呼出しをアセンブルしています。

```
            name    reptc
            extern  plotc
            section MYCODE:CODE(2)

banner      reptc   chr,"Welcome"
            movs    r0,#'chr'       ; char をパラメータとして渡す
            bl      plotc
            endr

            end
```

これにより、以下のコードが生成されます。

```
  9                         name    reptc
 10                         extern  plotc
 11                         section MYCODE:CODE(2)
 12
 13              banner     reptc   chr,"Welcome"
```

```
   14                          movs    r0,#'chr'     ; char をパラメータとして
渡す
   15                          bl      plotc
   16                          endr
   16.1  00000000 5700B0E3     movs    r0,#'W'       ; char をパラメータとして
渡す
   16.2  00000004 ........     bl      plotc
   16.3  00000008 6500B0E3     movs    r0,#'e'       ; char をパラメータとして
渡す
   16.4  0000000C ........     bl      plotc
   16.5  00000010 6C00B0E3     movs    r0,#'l'       ; char をパラメータとして
渡す
   16.6  00000014 ........     bl      plotc
   16.7  00000018 6300B0E3     movs    r0,#'c'       ; char をパラメータとして
渡す
   16.8  0000001C ........     bl      plotc
   16.9  00000020 6F00B0E3     movs    r0,#'o'       ; char をパラメータとして
渡す
   16.10 00000024 ........     bl      plotc
   16.11 00000028 6D00B0E3     movs    r0,#'m'       ; char をパラメータとして
渡す
   16.12 0000002C ........     bl      plotc
   16.13 00000030 6500B0E3     movs    r0,#'e'       ; char をパラメータとして
渡す
   16.14 00000034 ........     bl      plotc
   17
   18                                      end
```

以下の例では、REPTI を使用して複数のメモリロケーションをクリアしています。

```
            name    repti
            extern  a,b,c
            section MYCODE:CODE(2)

clearABC    movs    r0,#0
            repti   location,a,b,c
            ldr     r1,=location
            str     r0,[r1]
            endr

            end
```

これにより、以下のコードが生成されます。

```
    9                           name    repti
   10                           extern  a,b,c
   11                           section MYCODE:CODE(2)
   12
```

```
13    00000000 0000B0E3  clearABC    movs    r0,#0
14               loop        repti   location,a,b,c
15                           ldr     r1,=location
16                           str     r0,[r1]
17                           endr
17.1  00000004 10109FE5     ldr     r1,=a
17.2  00000008 000081E5     str     r0,[r1]
17.3  0000000C 0C109FE5     ldr     r1,=b
17.4  00000010 000081E5     str     r0,[r1]
17.5  00000014 08109FE5     ldr     r1,=c
17.6  00000018 000081E5     str     r0,[r1]
18
19                           end
```

リスト制御ディレクティブ

これらのディレクティブは、アセンブラリストファイルを制御します。

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	条件付きアセンブラリストを制御します。
LSTCOD	複数行からなるコードのリストを制御します。
LSTEXP	マクロで生成された行のリストを制御します。
LSTMAC	マクロ定義のリストを制御します。
LSTOUT	アセンブリリスト出力を制御します。
LSTPAG	ページへの出力形式を制御します。
LSTREP	繰返しディレクティブで生成された行のリストを制御します。
LSTXRF	クロスリファレンス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21: リスト制御ディレクティブ

構文

COL *columns*

LSTCND{+|-}

LSTCOD{+|-}

LSTEXP{+|-}

LSTMAC{+|-}

LSTOUT{+|-}

```
LSTPAG{+ | -}
LSTREP{+|-}
LSTXRF{+|-}
PAGE
PAGSIZ 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

リストのオン / オフ切り替え

エラーメッセージを除くすべてのリスト出力を無効にするには、LSTOUT- を使用します。このディレクティブは、他のどのリスト制御ディレクティブよりも優先されます。

デフォルトは LSTOUT+ です。この場合、出力がリストされます（リストファイルが指定されていない場合）。

条件付きコードと文字列のリスト

前の条件付き文 IF によって無効にされていないアセンブリ部分のみのために、アセンブラにソースコードを強制的にリストさせるには、LSTCND+ を使用します。

デフォルト設定は LSTCND- であり、すべてのソース行がリストされます。

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*。

マクロのリストの制御

マクロで生成された行のリストを無効にするには、LSTEXP- を使用します。デフォルトは LSTEXP+ であり、マクロで生成されたすべての行がリストされます。

マクロ定義をリストするには、LSTMAC+ を使用します。デフォルトは LSTMAC- であり、マクロ定義のリストが無効になります。

生成された行のリストを制御します。

ディレクティブ REPT、REPTC、REPTI によって生成された行のリストをオフにするには LSTREP- を使用します。

デフォルトは LSTREP+ であり、生成された行がリストされます。

クロスリファレンステーブルの生成

現在のモジュールのアセンブラリストの最後にクロスリファレンステーブルを生成するには、LSTXRF+ を使用します。このテーブルは、値と行番号、およびシンボルの型を示します。

デフォルトは LSTXRF- であり、クロスリファレンス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 PAGSIZ 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 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

リストのオン / オフ切り替え

プログラム内のデバッグされた部分のリストを無効にするには、以下のように指定します。

```
lstout-
; This section has already been debugged.
lstout+
; This section is currently being debugged.
end
```

条件付きコードと文字列のリスト

以下の例は、IF ディレクティブによって無効にされたサブルーチンの呼出しを、LSTCND+ がどのように非表示にするのかを示します。

```
            name    lstcndTest
            extern  print
            section FLASH:CODE(2)

debug       set     0
begin       if      debug
            bl      print
            endif

            lstcnd+
begin2      if      debug
            bl      print
            endif

            end
```

これにより、以下のリストが生成されます。

```
    9                       name   lstcndTest
   10                       extern  print
   11                       section FLASH:CODE(2)
   12
   13             debug     set     0
   14             begin     if      debug
   15                       bl      print
   16                       endif
   17
   18                       lstcnd+
   19             begin2    if      debug
   21                       endif
   22
   23                       end
```

マクロのリストの制御

以下の例は、LSTMAC と LSTEXP の効果を示します。

```
            name    lstmacTest
            extern  memLoc
            section FLASH:CODE(2)
```

```
dec2        macro   arg
            subs    r1,r1,#arg
            subs    r1,r1,#arg
            endm

            lstmac+
inc2        macro   arg
            adds    r1,r1,#arg
            adds    r1,r1,#arg
            endm

begin       dec2    memLoc
            lstexp-
            inc2    memLoc
            bx      lr

; Restore default values for
; listing control directives.

            lstmac-
            lstexp+

            end
```

これにより、以下の出力が生成されます。

```
  13                        name    lstmacTest
  14                        extern  memLoc
  15                        section FLASH:CODE(2)
  16
  21
  22                        lstmac+
  23             inc2       macro   arg
  24                        adds    r1,r1,#arg
  25                        adds    r1,r1,#arg
  26                        endm
  27
  28             begin      dec2    memLoc
  28.1  00000000 ........    subs    r1,r1,#memLoc
  28.2  00000004 ........    subs    r1,r1,#memLoc
  28.3                      endm
  29                        lstexp-
  30                        inc2    memLoc
  31    00000010 1EFF2FE1    bx      lr
  32
  33             ; Restore default values for
  34             ; listing control directives.
  35
```

```
36                              lstmac-
37                              lstexp+
38
39                              end
```

C 形式のプリプロセッサディレクティブ

アセンブラには、C89 規格に似た C 形式のプリプロセッサがあります。

以下の C 言語プリプロセッサディレクティブを使用できます。

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 ブロックに新しい条件を導入します。
#else	条件が偽の場合に命令をアセンブルします。
#endif	#if、#ifdef、または #ifndef ブロックを終了させます。
#error	エラーを生成します。
#if	条件が真の場合に命令をアセンブルします。
#ifdef	プリプロセッサシンボルが定義されている場合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場合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	デバッグ情報内のソース参照を変更します。
#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取り消します。

表 22: C 形式のプリプロセッサディレクティブ

構文

```
#define symbol text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef symbol
#ifndef symbol
```

```
#include {"filename" | <filename>}
#line line-no {"filename"}
#message "message"
#undef symbol
```

パラメータ

condition	絶対式	式に、アセンブララベルまたはシンボルを含めることは一切できず、ゼロ以外の値はすべて真と見なされます。
filename	インクルードされるか参照されるファイルの名前。	
line-no	ソース行番号。	
message	表示されるテキスト。	
symbol	定義、定義取消し、またはテストされるプリプロセッサシンボル。	
text	割り当てられる値。	

説明

アセンブラ言語と C 形式のプリプロセッサディレクティブを混在させないでください。これらは概念的に異なる言語です。アセンブラディレクティブは C プリプロセッサ言語の一部として受け入れられない場合があるため、これらを混在させると、予期しない動作の原因となる可能性があります。

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ます。たとえば、以下のような矛盾を避けてください。

```
redef       macro                   ; Avoid the following!
#define ¥1 ¥2
            endm
```

これは、¥1 と ¥2 というマクロ引数は前処理フェーズで使用できないためです。

プリプロセッサシンボルの定義と定義取消し

プリプロセッサシンボルを定義するには、#define を使用します。

#define *symbol value*

シンボルの定義を取り消すには #undef を使用します。その結果、定義されていないようになります。

条件付き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プロセスを制御するには、#if...#else...#endif ディレクティブを使用します。#if ディレクティブの後の条件が真ではない場合、#endif または #else ディレクティブが検出されるまで、後続の命令はコードを一切生成しません（つまり、アセンブルも構文チェックも行われません）。

（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、条件付きディレクティブで無効にできます。各 #if ディレクティブは #endif ディレクティブで終了する必要があります。#else ディレクティブは任意指定であり、使用する場合は、#if...#endif ブロック内に指定する必要があります。

#if...#endif ブロックと #if...#else...#endif ブロックは、任意のレベルまでネストできます。

シンボルが定義されている場合に限り、次の #else または #endif ディレクティブまで命令をアセンブルするには、#ifdef を使用します。

シンボルが定義されていない場合に限り、次の #else または #endif ディレクティブまで命令をアセンブルするには、#ifndef を使用します。

ソースファイルのインクルード

#include を使用して、ヘッダファイルの内容を、ソースファイル中の指定した箇所に挿入します。

#include "*filename*" と #include <*filename*> は、以下のディレクトリを指定の順に検索します。

1 ソースファイルディレクトリ（この手順は、#include "*filename*" でのみ有効です）。

2 -I オプションで指定されたディレクトリ ディレクトリは、コマンドラインで指定したものと同じ順序で検索され、続いて環境変数で指定したものが検索されます。

3 現在のディレクトリ。アセンブラの実行可能ファイルがあるディレクトリと同じです。

4 自動的に設定されたライブラリシステムには、ディレクトリが含まれます。*40 ページの -g* を参照。

エラー表示

ユーザ定義テストなどでアセンブラに強制的にエラーを生成させるには、`#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でのコメント

定義文でコメントを記述するには、以下の形式を使用します。

- 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。
- 残りの行をコメントとしてマーキングするには、C++ コメント区切り文字 `//` を使用します。

定義された文の中でアセンブラコメントを使用すると、予期しない動作をする可能性があるため、使用しないでください。

以下の式では、コメント文字が `#define` によって保護されているため、評価結果は 3 となります。

```
#define x 3    ; 置く場所の違うコメント

            module  misplacedComment1
expression  equ     x * 8 + 5
            ;...
            end
```

以下の例は、C 形式のプリプロセッサでアセンブラコメントを使用すると発生する可能性のある問題の一部を示します。

```
#define five  5     ; このコメントは無効
#define six   6    // このコメントは有効
#define seven 7    /* このコメントは有効 */

            module  misplacedComment2
            section MYCONST:CONST(2)

            DC32    five, 11, 12
; 上の行は次のように展開される
;          "DC32    5    ; このコメントは無効 , 11, 12"
```

```
            DC32    six + seven, 11, 12
; 上の行は次のように展開される
;           "DC32    6 + 7, 11, 12"

            end
```

ソース行番号の変更

デバッグ情報で使用されるソース行番号およびソースファイル名を変更するには、#line ディレクティブを使用します。#line は、#line ディレクティブに続く行で処理されます。

例

条件付きプリプロセッサディレクティブの使用

以下の例は、tweak と adjust というラベルを定義します。tweak が定義されている場合、レジスタ r0 は adjust に応じた数値だけデクリメントされます（adjust が 3 であれば 30）。

```
            name    calibrate
            extern  calibrationConstant
            section MYCODE:CODE(2)
            arm

#define     tweak  1
#define     adjust 3

calibrate   ldr     r0,calibrationConstant
#ifdef      tweak
#if         adjust==1
            subs    r0,r0,#4
#elif       adjust==2
            subs    r0,r0,#20
#elif       adjust==3
            subs    r0,r0,#30
#endif
#endif      /* ifdef tweak */
            str     r0,calibrationConstant
            bx      lr

            end
```

ソースファイルのインクルード

以下の例では、ファイル定義マクロをソースファイルにインクルードするために #include が使用されます。たとえば、次のようなマクロを Macros.inc に定義できます。

```
; Exchange registers a and b.
; Use register c for temporary storage.

xch         macro   a,b,c
            movs    c,a
            movs    a,b
            movs    b,c
            endm
```

続いて、次の例のように、#include を使用してマクロ定義をインクルードできます。

```
            name    includeFile
            section MYCODE:CODE(2)
            arm

; Standard macro definitions.
#include "Macros.inc"

xchRegs     xch     r0,r1,r2
            bx      lr

            end
```

データ定義ディレクティブまたは割当てディレクティブ

これらのディレクティブは、値を定義するか、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際に適用される制限については、*29 ページの式の制限*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	文字列を含め 8 ビットの定数を生成します。
DC16	DCW	16 ビットの定数を生成します。
DC24		24 ビットの定数を生成します。
DC32	DCD	32 ビットの定数を生成します。
DF32		浮動小数点（32 ビット）定数を生成します。

表 23: データ定義ディレクティブまたは割当てディレクティブ

ディレクティブ	エイリアス	説明
DF64		浮動小数点（64 ビット）定数を生成します。
DS8	DS	8 ビット整数に空間を割り当てます。
DS16		16 ビット整数に空間を割り当てます。
DS24		24 ビット整数に空間を割り当てます。
DS32		32 ビット整数に空間を割り当てます。

表 23: データ定義ディレクティブまたは割当てディレクティブ（続き）

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] ...
DS count
DS8 count
DS16 count
DS24 count
DS32 count
```

パラメータ

count 予約する要素の数を指定する有効な絶対式。

expr 有効な絶対式、再配置可能式、外部式、または ASCII 文字列。ASCII 文字列は、ディレクティブで示唆されるデータサイズの倍数までゼロが埋め込まれます。二重引用符で囲まれた文字列はゼロで終了します。

value 有効な絶対式または浮上小数点定数。

説明

DC8、DC16、DC24、DC32、DCB、DCD、DCW、DF32、または DF64 を使用して、定数を作成します。つまり、バイトのある領域が定数に対して十分に予約されます。

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

例

ルックアップテーブルの生成

この例では、8 ビットデータの定数テーブルのエントリを合計します。

```
            module  sumTableAndIndex
            section MYDATA:CONST
            data

table       dc8     12
            dc8     15
            dc8     17
            dc8     16
            dc8     14
            dc8     11
            dc8     9

            section MYCODE:CODE(2)
            arm
count       set     0

addTable    movs    r0,#0
            ldr     r1,=table

            rept    7
            if      count == 7
            exitm
            endif
            ldrb    r2,[r1,#count]
            adds    r0,r0,r2
count       set     count + 1
            endr

            bx      lr

            end
```

文字列の定義

文字列を定義するには、以下のように指定します。

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

最後のゼロを含む文字列を定義するには、次のように指定します。

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

文字列で単一引用符を使用するには、次のように 2 つ入力します。

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

空間の予約

10 バイト用に空間を予約するには、次のように指定します。

```
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際に適用される制限については、*29 ページの 式の制限*を参照してください。

ディレクティブ	説明	式の制限
$	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。	
//	C++形式のコメント区切り文字。	
CASEOFF	大文字 / 小文字の区別を無効にします。	
CASEON	大文字 / 小文字の区別を有効にします。	
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	すべての数値にデフォルトの基数を設定します。	前方参照禁止 外部参照禁止 絶対 固定

表 24: アセンブラ制御ディレクティブ

構文

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

パラメータ

comment	アセンブラに無視されるコメント。
expr	デフォルトの基数。デフォルトは 10（10 進数）
filename	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。これは #include のエイリアスです（*101 ページの ソースファイルのインクルード*を参照）。

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。

残りの行をコメントとしてマーキングするには、// を使用します。

デフォルトの定数用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。

現在のリテラルプールがどこでアセンブルされるか指示するには、LTORG を使用します。デフォルトでは、END および RSEG ディレクティブごとに、これが行われます。例については、*131 ページの LDR (ARM)* を参照してください。

大文字 / 小文字の区別の制御

ユーザ定義シンボルで大文字と小文字を区別するかどうかを切り替えるには、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 を有効にすると、すべてのシンボルは大文字で格納され、ILINK によって使用されるすべてのシンボルは ILINK 定義ファイルに大文字で記述する必要があります。

例

ソースファイルのインクルード

この例では、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、次のようなマクロを Macros.inc に定義できます。

```
; レジスタ a と b を交換する
; レジスタ c を一時メモリに使用する

xch         macro   a,b,c
            movs    c,a
            movs    a,b
            movs    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    includeFile
            section MYCODE:CODE(2)
            arm

; 標準マクロ定義
$Macros.inc

xchRegs     xch     r0,r1,r2
            bx      lr

            end
```

コメントの定義

以下の例は、複数行から成るコメントでの /*...*/ の使用方法を示します。

```
/*
シリアル入力から読み込むプログラム
バージョン 1: 19.2.11
Author: mjp
*/
```

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

基数の変更

デフォルトの基数を 16 に設定するには、次のように指定します。

```
            module  radix
            section MYCODE:CODE(2)

            radix   16              ; デフォルトの基数を 16 に
            movs    r0,#12          ; セットしたので movs 命令の
            ;...                    ; イミディエイト値は 0x12 であると
                                    ; 解釈される。

; To reset the base from 16 to 10 again, the argument must be
; written in hexadecimal format.

            radix   0x0a            ; デフォルトの基数を 10 に
            movs    r0,#12          ; リセットしたので movs 命令の
            ;...                    ; イミディエイト値は 0x0c であると
                                    ; 解釈される。
            end
```

大文字 / 小文字の区別の制御

CASEOFF を設定すると、以下の例では、label と LABEL が同じになります。

```
            module  caseSensitivity1
            section MYCODE:CODE(2)

            caseoff
label       nop                 ; "LABEL" と同じ。
            b       LABEL
            end
```

以下の例では、重複ラベルエラーが生成されます。

```
            module  caseSensitivity2
            section MYCODE:CODE(2)

            caseoff
label       nop                 ; "LABEL" と同じ。
LABEL       nop                 ; エラー , "LABEL" は定義済み
            end
```

呼出しフレーム情報ディレクティブ

C-SPY を使用してアプリケーションをデバッグする場合は、*呼出しスタック*、すなわち現在の関数を呼出した関数のチェーンを表示できます。コンパイラは、呼出しフレームのレイアウトを説明するデバッグ情報、特にリターンアドレスの格納されている場所を提供することで、C ソースコードをコンパイルする際にこれを可能にします。

アセンブラ言語で記述したルーチンのデバッグ時に呼出しスタックを使用できるようにするには、アセンブラディレクティブ `CFI` を使用して、同等のデバッグ情報をアセンブラソースコードで提供する必要があります。

このディレクティブを使用すると、アセンブラソースコードにバックトレース情報を定義できます。

ディレクティブ	説明
`CFI BASEADDRESS`	ベースアドレス CFA(Canonical Frame Address) を宣言します。
`CFI BLOCK`	データブロックを開始します。
`CFI CODEALIGN`	コードアラインメントを宣言します。
`CFI COMMON`	共通ブロックを開始または拡張します。
`CFI CONDITIONAL`	データブロックを条件付きスレッドとして宣言します。
`CFI DATAALIGN`	データアラインメントを宣言します。
`CFI DEFAULT`	すべてのリソースのデフォルトの状態を宣言します。
`CFI ENDBLOCK`	データブロックを終了します。
`CFI ENDCOMMON`	共通ブロックを終了します。
`CFI ENDNAMES`	名前ブロックを終了します。
`CFI FRAMECELL`	呼出し元のフレームに参照情報を作成します。
`CFI FUNCALL`	スタック使用量解析のために関数呼出しを宣言します。
`CFI FUNCTION`	データブロックに関連する関数を宣言します。
`CFI INDIRECTCALL`	スタック使用量解析のために間接的な呼出しを宣言します。
`CFI INVALID`	無効なバックトレース情報の範囲を開始します。
`CFI NAMES`	名前ブロックを開始します。
`CFI NOCALLS`	スタック使用量解析のために呼出しの欠如を宣言します。
`CFI NOFUNCTION`	関数に関連しないものとしてデータブロックを宣言します。

表 25: 呼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 PICKER	データブロックをピッカースレッドとして宣言します。関数内または関数同士でコードが共有される場合に、コンパイラで実行パスを追跡するために使用されます。
CFI REMEMBERSTATE	バックトレース情報の状態を記憶します。
CFI RESOURCE	リソースを宣言します。
CFI RESTORESTATE	保存されたバックトレース情報の状態を復元します。
CFI RETURNADDRESS	リターンアドレス列を宣言します。
CFI STACKFRAME	スタックフレーム CFA を宣言します。
CFI VALID	無効なバックトレース情報の範囲を終了します。
CFI *cfa*	CFA の値を宣言します。
CFI *resource*	リソースの値を宣言します。

表 25: 呼出しフレーム情報ディレクティブ（続き）

構文

以下の構文定義は、各ディレクティブの構文を示します。ディレクティブは用途に応じてグループ化されています。

名前ブロックディレクティブ

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource : bits [, resource : bits]
CFI STACKFRAME cfa resource type [, cfa resource type]
CFI BASEADDRESS cfa type [, cfa type]
```

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 DEFAULT { UNDEFINED | SAMEVALUE }
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
```

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
```

スタック使用量解析ディレクティブ

```
CFI FUNCALL { caller } callee
CFI INDIRECTCALL { caller }
CFI NOCALLS { caller }
```

パラメータ

bits	リソースのサイズ（ビット単位）。
callee	呼び出される関数のラベル。
caller	呼び出す関数のラベル。
cfa	CFA(Canonical Frame Address) の名前。
cfiexpr	CFI 式（*120 ページの 複雑なケースでの式の使用*を参照）。
codealignfactor	すべての命令サイズで最小の係数。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。デフォルトは 1 で、いつでも使用できますが、値を大きくすると、生成されるバックトレース情報のサイズが小さくなります。可能な範囲 1 〜 256 です。
commonblock	以前に定義された共通ブロックの名前。

constant	定数値、または定数値を計算するアセンブラ式。
dataalignfactor	すべてのフレームサイズで最小の係数。スタックのアドレスが大きくなると、係数はマイナスになります。小さくなると、係数はプラスになります。デフォルトは 1 ですが、値を大きくすると、生成されるバックトレース情報のサイズが小さくなります。可能な範囲は -256 ～ -1 と 1 ～ 256 です。
label	関数ラベル。
name	ブロックの名前。
namesblock	以前に定義された名前ブロックの名前。
offset	CFA に相対的なオフセット。任意指定の符号が付く整数です。
part	複合リソースのパート。以前に宣言されたリソースの名前。
resource	リソースの名前。
size	フレームセルのサイズ（ビット単位）。
type	メモリタイプ。CODE、CONST、DATA など。また、IAR ILINK リンカでサポートされるすべてのメモリタイプを使用できます。アドレス空間の指定のみに使用されます。

説明

CFI ディレクティブは、呼出し元関数のステータス情報を C-SPY に提供します。この*バックトレース*情報は、アセンブラコード内でレジスタやメモリセルなど*リソース*の内容を追跡するために使用されます。この中で最も重要な情報はリターンアドレスと、関数やアセンブラルーチンのエントリ時点でのスタックポインタの値です。

この情報を使用して、C-SPY は呼出し元関数の状態を復元し、スタックを巻き戻して、関数を入力する前にレジスタの正しい値や他のリソースを表示することができます。これによって、デバッガはブレークポイントまでフルスピードで実行し、ブレークポイントで停止して、その時点でのアプリケーションのバックトレース情報を取得できます。続いて、任意の呼出し関数でこの情報を使用してリソース内容を計算できます（呼出しフレーム情報も存在することが前提となります）。スタック使用量解析ディレクティブは、呼出しフレーム情報の一部ではありません。スタック使用量解析ディレクティブは、コンパイラおよびシステムライブラリが呼出しグラフの情報をリンカに渡すための便利な方法です。

呼出し規約に関する詳細記述では、広範な呼出しフレーム情報を必要とする場合があります。多くの場合は、より限定的なアプローチで十分です。呼出しフレーム情報を記述するには、以下の 3 つのコンポーネントが必要です。

- 追跡可能なリソースを示す*名前ブロック*
- 呼出し規約に対応する*共通ブロック*
- 呼出しフレームで実行された変更を示す*データブロック*。通常、これには、スタックポインタが変更された時点、保護レジスタがスタックで待避、復帰した時点についての情報が含まれます。

呼出しフレーム情報を正しく処理するアセンブラ言語ルーチンを作成する良い方法は、アセンブラ出力を生成するためにコンパイルする C スケルトン関数から開始することです。例については、*『ARM 用 IAR C/C++ 開発ガイド』*を参照してください。

バックトレース行と列

デバッガが実行を停止できるプログラム内の各ロケーションには、*バックトレース行*があります。各バックトレース行は、*列*のセットから構成されています。それぞれの列は追跡対象の各アイテムを表します。以下の 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。

リターンアドレス列は常に 1 つあります。CFA 列は通常 1 つありますが、複数ある場合もあります。

名前ブロックの定義

プロセッサで使用可能なリソースを宣言するには、*名前ブロック*を使用します。名前ブロックの内部に、追跡可能なすべてのリソースが定義されています。

名前ブロックの開始と終了には、以下のディレクティブを使用します。

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

ここで、*name* はブロックの名前です。

一度に開ける名前ブロックは 1 つだけです。

名前ブロックの内部では、リソース宣言、スタックフレーム宣言、静的オーバーレイフレーム宣言、またはベースアドレス宣言という 4 種類の宣言を使用できます。

- リソースを宣言するには、次のディレクティブを使用します。

  ```
  CFI RESOURCE resource : bits
  ```

  パラメータは、リソースの名前とリソースのサイズ（ビット単位）です。名前は AEABI ドキュメント「*DWARF for the ARM architecture*」で定義されたレジスタ名のいずれかにする必要があります。

  複数のリソースを宣言する場合、リソース間をコンマで区切ります。
- スタックフレーム 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 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）、メモリタイプ（アドレス空間の取得用）です。複数のスタックフレーム CFA を宣言する場合、コンマで区切ります。

  呼出しスタックに戻る場合、前の関数フレームの正しい値を取得するために、スタックフレーム CFA の値が対応するスタックポインタリソースにコピーされます。
- 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 の名前とメモリタイプです。複数のベースアドレス CFA を宣言する場合、コンマで区切ります。

  ベースアドレス CFA を使用すると、CFA の取り扱いが簡単になります。スタックフレーム CFA と違い、復元する関連スタックポインタリソースはありません。

共通ブロックの定義

すべての追跡対象リソースの初期内容を宣言するには、*共通ブロック*を使用します。通常、使用される各呼出し規約に共通ブロックが 1 つずつあります。

共通ブロックを開始するには、以下のディレクティブを使用します。

```
CFI COMMON name USING namesblock
```

ここで、*name* は新しいブロックの名前であり、*namesblock* は以前に定義された名前ブロックの名前です。

リターンアドレス列を宣言するには、以下のディレクティブを使用します。

```
CFI RETURNADDRESS resource type
```

ここで *resource* は *namesblock* に定義されたリソースであり、*type* はメモリタイプです。共通ブロックに対してリターンアドレス列を宣言する必要があります。

共通ブロックを終了するには、以下のディレクティブを使用します。

```
CFI ENDCOMMON name
```

ここで、*name* は共通ブロックを開始するために使用される名前です。

共通ブロックの内部には、*112 ページの 共通ブロックディレクティブ*に記載の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できます。これらのディレクティブについては、*118 ページの 単純なケースの規則*、*120 ページの 複雑なケースでの式の使用*を参照してください。

データブロックの定義

*データブロック*には、1 つの連続したコードの実際の追跡情報が含まれます。

データブロックを開始するには、以下のディレクティブを使用します。

```
CFI BLOCK name USING commonblock
```

ここで、*name* は新しいブロックの名前であり、*commonblock* は以前に定義された共通ブロックの名前です。

当該コードが、定義された関数の一部である場合、次のディレクティブを使用して関数名を指定します。

```
CFI FUNCTION label
```

ここで、*label* は関数を開始するコードラベルです。

当該コードが関数の一部でない場合、次のディレクティブを使用してこのことを指定します。

```
CFI NOFUNCTION
```

データブロックを終了するには、以下のディレクティブを使用します。

```
CFI ENDBLOCK name
```

ここで、*name* はデータブロックを開始するために使用される名前です。

データブロックの内部では、*113 ページの データブロックディレクティブ*に記載のディレクティブを使用して列の値を操作できます。これらのディレクティブについては、*118 ページの 単純なケースの規則*、*120 ページの 複雑なケースでの式の使用*を参照してください。

単純なケースの規則

個々の列の追跡情報を記述するために、特別な構文の簡易規則の数々が用意されています。

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

これらの簡易規則は、共通ブロック内で使用してリソースと CFA の初期情報を記述したり、データブロック内で使用してリソースと CFA の情報への変更を記述したりすることができます。

万が一、簡易規則で十分に記述できない場合には、完全な CFI 式を使用して情報を記述できます（*120 ページの 複雑なケースでの式の使用*を参照）。ただし、可能な限り、CFI 式ではなく簡易規則を使用してください。

簡易規則には、リソース用と CFA 用という 2 種類のセットがあります。

リソース用簡易規則

呼出しフレームを 1 つ戻る場合にリソースがどこにあるのかを概念的に記述するリソース用規則です。このため、CFI ディレクティブでリソース名の後にあるアイテムを、リソースの*ロケーション*と呼びます。

追跡対象のリソースが復元されている、つまりこのリソースの場所が既に正しく認識されていることを宣言するには、ロケーションに SAMEVALUE を使用します。リソースには既に正しい値が含まれているため、概念的には、これによってリソースの復元が不要であることが宣言されます。たとえば、レジスタ REG が同じ値に復元されることを宣言するには、以下のディレクティブを使用します。

```
CFI REG SAMEVALUE
```

リソースが追跡対象では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。リソースは追跡されないため、概念的には、これによって（呼出しフレームを 1 つ戻る場合に）リソースの復元が不要であることが宣言されます。これを使用して意味があるのは、リソースの初期ロケーションを宣言する場合のみです。たとえば、REG がスクラッチレジスタであり復元不要であることを宣言するには、以下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、ロケーションとしてリソース名を使用します。たとえば、レジスタ REG1 が一時的にレジスタ REG2 に格納されており、そのレジスタから復元する必要があることを宣言するには、以下のディレクティブを使用します。

```
CFI REG1 REG2
```

リソースが現在、スタック内のどこかに存在することを宣言するには、FRAME(*cfa*、*offset*) をリソースのロケーションとして使用します。ここで、*cfa* は「フレームポインタ」として使用される CFA 識別子であり、*offset* は CFA に対して相対的なオフセットです。たとえば、レジスタ REG がフレームポインタ CFA_SP から数えてオフセット -4 に存在することを宣言するには、以下のディレクティブを使用します。

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

複合リソースには、追加ロケーションがもう 1 つあります。これは CONCAT で、複合リソースのリソースパートを結合するとリソースのロケーションを検出できることを宣言します。たとえば、リソースパート RETLO と RETHI から成る複合リソース RET を考えてみます。リソースパートを検証して連結すると RET の値を検出できることを宣言するには、以下のディレクティブを使用します。

```
CFI RET CONCAT
```

このためには、リソースパーツの少なくとも 1 つに、前述の規則を使用する定義が必要です。

CFA 用簡易規則

リソース用の規則と違い、CFA 用の規則には呼出しフレームの先頭のアドレスを記述します。呼出しフレームには、サブルーチン呼出し命令によってプッシュされるリターンアドレスが含まれる場合があります。CFA 規則は、このアドレスから現在の呼出しフレームの先頭を計算する方法を記述します。CFA には、スタックフレームと静的オーバーレイフレームという 2 種類の形式があり、それぞれ対応する名前ブロックに宣言されています。*112 ページの名前ブロックディレクティブ*を参照。

各スタックフレーム CFA には、スタックポインタなどのリソースが関連付けられています。呼出しフレームを 1 つ戻ると、関連リソースは現在の CFA で復元されます。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 を使用せず、関連するリソースは通常のリソースとして追跡する必要があ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使用しないことを宣言するには、以下のディレクティブを使用します。

```
CFI CFA_SP NOTUSED
```

CFA のアドレスが、リソースの値に相対的なオフセットであ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、SP というリソースの値に 4 を足すと CFA_SP という名前の CFA を取得できることを宣言するには、以下のディレクティブを使用します。

```
CFI CFA_SP SP + 4
```

複雑なケースでの式の使用

リソースと CFA 用の簡易規則では十分に記述できない場合には、呼出しフレーム情報式（CFI 式）を使用できます。ただし、可能な限り、簡易規則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 式で許可されている演算子は以下に挙げるもののみです。ほぼ、通常のアセンブラ式と同じ演算子を使用できます。

オペランドの記述では、*cfiexpr* は以下のいずれかを示します。

- オペランド付きの CFI 演算子
- 数値の定数
- CFA 名
- リソース名

単項演算子

全体的な構文：*OPERATOR*(*operand*)

演算子	オペランド	説明
COMPLEMENT	*cfiexpr*	CFI 式のビット単位の NOT を実行します。
LITERAL	*expr*	アセンブラ式の値を取得します。これにより、通常のアセンブラ式の値を CFI 式に挿入できます。
NOT	*cfiexpr*	論理 CFI 式を否定します。
UMINUS	*cfiexpr*	CFI 式を算術的に論理否定します。

表 26: CFI 式の単項演算子

2 項演算子

全体的な構文：*OPERATOR*(*operand1*,*operand2*)

演算子	オペランド	説明
ADD	*cfiexpr*、*cfiexpr*	加算
AND	*cfiexpr*、*cfiexpr*	ビット単位の AND
DIV	*cfiexpr*、*cfiexpr*	除算
EQ	*cfiexpr*、*cfiexpr*	等しい
GE	*cfiexpr*、*cfiexpr*	以上
GT	*cfiexpr*、*cfiexpr*	より大きい
LE	*cfiexpr*、*cfiexpr*	以下
LSHIFT	*cfiexpr*、*cfiexpr*	左オペランドの論理左シフト。シフト対象のビット数は、右オペランドで指定します。シフト時に符号ビットは保護されません。
LT	*cfiexpr*、*cfiexpr*	より小さい
MOD	*cfiexpr*、*cfiexpr*	剰余
MUL	*cfiexpr*、*cfiexpr*	乗算
NE	*cfiexpr*、*cfiexpr*	等しくない
OR	*cfiexpr*、*cfiexpr*	ビット単位の OR
RSHIFTA	*cfiexpr*、*cfiexpr*	左オペランドの算術右シフト。シフト対象のビット数は、右オペランドで指定します。RSHIFTL と違い、符号ビットはシフト時に保護されます。
RSHIFTL	*cfiexpr*、*cfiexpr*	左オペランドの論理右シフト。シフト対象のビット数は、右オペランドで指定します。シフト時に符号ビットは保護されません。
SUB	*cfiexpr*、*cfiexpr*	減算
XOR	*cfiexpr*、*cfiexpr*	ビット単位の XOR

表 27: CFI 式の 2 項演算子

3 項演算子

全体的な構文：*OPERATOR*(*operand1*,*operand2*,*operand3*)

演算子	オペランド	説明
FRAME	*cfa*、*size*、*offset*	スタックフレームから値を取得します。オペランドを以下に示します。 *cfa* 以前に宣言された CFA を指定する識別子。 *size* サイズをバイト単位で指定する定数式。 *offset* オフセットをバイト単位で指定する定数式。 サイズが *size* のアドレス *cfa+offset* から値を取得します。
IF	*cond*、*true*、*false*	条件演算子。オペランドを以下に示します。 *cond* 条件を示す CFA 式。 *true* 任意の CFA 式。 *false* 任意の CFA 式。 条件式がゼロ以外である場合、結果は *true* 式の値となりますが、それ以外の場合は *false* 式の値となります。
LOAD	*size*、*type*、*addr*	メモリから値を取得します。オペランドを以下に示します。 *size* サイズをバイト単位で指定する定数式。 *type* メモリタイプ。 *addr* メモリアドレスを示す CFA 式。 メモリタイプ *type* のアドレス *addr* における値（サイズ *size*）を取得します。

表 28: CFI 式の 3 項演算子

スタック使用量解析ディレクティブ

スタック使用量解析ディレクティブ (CFI FUNCALL、CFI INDIRECTCALL、CFI NOCALLS) は、呼出しグラフのビルドに使用されます。これらは、データブロック内でのみ使用できます。データブロックが関数ブロックの場合（つまり、CFI FUNCTION ディレクティブがデータブロック内で使用されている場合）、*caller* パラメータを指定しないでください。スタック使用量解析ディレクティブが関数同士で共有されているコード内で使用されている場合、情報が適用される可能性のある関数を指定するときには *caller* パラメータを使用する必要があります。

CFI FUNCALL と CFI INDIRECTCALL ディレクティブは、スタック使用量の情報が正しいところに配置する必要があります。これを行う最も簡単な方法は、呼出しを実行する命令の直前にそれらを配置することです。CFI NOCALLS ディレクティブは、データブロックのどこにでも配置できます。

例

以下は ARM コアに固有の例です。その他の例は、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入手できます。

スタックポインタ R13 を持つ Cortex-M3 デバイス、リンクレジスタ R14、および汎用目的のレジスタ R0-R12 について考えてください。レジスタ R0、R2、R3、R12 はスクラッチレジスタ（これらのレジスタは関数の呼出しによって破棄されることがあります）として使用されるのに対して、レジスタ R1 は関数呼出しの後に復元する必要があります。

以下の短いサンプルコードと、対応するバックトレース行および列を考えてみましょう。開始時点で、レジスタ R14 に 32 ビットのリターンアドレスが含まれているとします。スタックは上位アドレスからゼロに向かって大きくなります。CFA は呼出しフレームのトップを指定します。つまり、関数から戻ったときのスタックポインタの値です。

アドレス	CFA	R0	R1	R2	R3	R4-R11	R12	R13	R14	アセンブラコード
00000000	R13 + 0	—	SAME	—	—	SAME	—	—	SAME	PUSH {r1,lr}
00000002	R13 + 8		CFA - 8						CFA - 4	MOVS r1,#4
00000004										BL func2
00000008										POP {r0,lr}
0000000C	R13 + 0		R0						SAME	MOV r1,r0
0000000E			SAME							BX lr

表 29: バックトレース行と列付きのサンプルコード

各バックトレース行は、命令を実行する*前*の追跡対象リソースの状態を示します。たとえば、MOV R1,R0 命令では、R1 レジスタの元の値は R0 レジスタにあり、関数フレーム（CFA 列）のトップは R13 + 0 です。アドレス 0000 のバックトレース行は最初の行であり、関数に使用された呼出し規約の結果です。

R13 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。R14 列はリターンアドレスの列です。つまり、リターンアドレスのロケーションです。R0 列の最初の行は「—」です。これは、r0 の値が未定義であり、関数の終了時に復元する必要がないことを示します。R1 列の最初の行は SAME です。これは、r1 レジスタの値が、既知の値と同じ値に復元されることを示します。

ネームブロックの定義

上の例で指定する名前ブロックは次のようになります。

```
cfi     names ArmCore
cfi     stackframe cfa r13 DATA
cfi     resource  r0:32,  r1:32,  r2:32,  r3:32
cfi     resource  r4:32,  r5:32,  r6:32,  r7:32
cfi     resource  r8:32,  r9:32, r10:32, r11:32
cfi     resource r12:32, r13:32, r14:32
cfi     endnames ArmCore
```

共通ブロックの定義

```
cfi     common trivialCommon using ArmCore
cfi     codealign 2
cfi     dataalign 4
cfi     returnaddress r14 CODE
cfi     cfa     r13+0
cfi     default samevalue
cfi     r0      undefined
cfi     r2      undefined
cfi     r3      undefined
cfi     r12     undefined
cfi     endcommon trivialCommon
```

注： r13 は、CFA に関連付けられたリソースであるため、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

```
            section MYCODE:CODE(2)

            cfi     block trivialBlock using trivialCommon
            cfi     function func1

            thumb

func1       push    {r1,lr}

            cfi     r1  frame(cfa, -8)
            cfi     r14 frame(cfa, -4)
            cfi     cfa r13+8

            movs    r1,#4

            cfi     funcall func2
```

```
bl      func2
pop     {r0,lr}

cfi     r1  r0
cfi     r14 samevalue
cfi     cfa r13

mov     r1,r0

cfi     r1 samevalue

bx      lr

cfi     endblock trivialBlock

end
```

注： CFI ディレクティブは、バックトレース情報が変更された地点に配置してください。つまり、バックトレース情報を変更した命令の直*後*ということです。

アセンブラ擬似命令

ARM 用 IAR アセンブラではさまざまな擬似命令を使用でき、これらはすべて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 ディレクティブの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の表に示します。

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。
MOV	CODE16*	ADD	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。

表 30: 擬似命令

擬似命令	ディレクティブ	変換結果	説明
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 30: 擬似命令（続き）

*** 廃止予定。代わりに THUMB を使用してください。**

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)

構文

```
ADR{condition} register,expression
```

パラメータ

{condition}	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。
register	ロードするレジスタです。
expression	-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。

```
            name    armAdr
            section MYCODE:CODE(2)
            arm
            adr     r0,thumbLabel   ; "add r0,pc,#1"  になる
            bx      r0

            thumb
thumbLabel  ; ...

            end
```

ADR (CODE16)

構文

```
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

この Thumb-1 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）。

ADR (THUMB)

構文

```
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。

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できます。

例

```
name    thumbAdr
section MYCODE:CODE(2)
thumb
adr     r0,dataLabel    ; "add r0,pc,#4" になる
add     r0,r0,r1
bx      lr
```

```
            data
            alignrom 2
dataLabel   dc32    0xABCD19

            end
```

関連項目

16 ビット Thumb 命令のみが使用可能な場合、*129 ページの ADR (CODE16)* を参照してください。

ADRL (ARM)

構文

```
ADRL{condition} register,expression
```

パラメータ

{condition}	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。
register	ロードするレジスタです。
expression	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。

例

```
            name    armAdrL
            section MYCODE:CODE(2)
            arm
            adrl    r1,label+0x2345 ; "add r1,pc,#0x45"
                                    ; と "add r1,r1,#0x2300" になる
            data
label       dc32    0

            end
```

ADRL (THUMB)

構文

```
ADRL{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	± 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)

構文

```
LDR{condition} register,=expression1
```

または

```
LDR{condition} register,expression2
```

パラメータ

condition	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。
expression1	任意の 32 ビット式です。
expression2	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。

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 命令の範囲内にない場合または 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。

例

```
            name    armLdr
            section MYCODE:CODE(2)
            arm
            ldr     r1,=0x12345678  ; "ldr r1,[pc,#4]" になる
                                    ; リテラルプールから
                                    ; 0x12345678 をロードする
            ldr     r2,label        ; "ldr r2,[pc,#-4]" になる
                                    ; r2 に 0xFFEEDDCC をロードする
            data
label       dc32    0xFFEEDDCC
            ltorg                   ; リテラルプールはここに配置される
            end
```

関連項目

*107 ページの アセンブラ制御ディレクティブ*のセクションの LTORG ディレクティブ。

LDR (CODE16)

構文

LDR *register*, =*expression1*

または

LDR *register*, *expression2*

パラメータ

register	ロードするレジスタです。LDR は下位のレジスタ (R0-R7) にのみアクセス可能です。
expression1	任意の 32 ビット式です。
expression2	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。

LDR (THUMB)

構文

LDR{*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	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。

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通常の命令の一部です。

例

```
            name    thumbLdr
            extern  extLabel
            section MYCODE:CODE(2)
            thumb
            ldr     r1,=extLabel    ; "ldr r1,[pc,#8]" になる
            nop                     ; extLabel をリテラルプール
                                    ; からロードする
            ldr     r2,label        ; "ldr r2,[pc,#0]" になる
            nop                     ; r2 に 0xFFEEDDCC をロードする
            data
label       dc32    0xFFEEDDCC
            ltorg                   ; リテラルプールはここに配置される
            end
```

関連項目

16 ビット Thumb 命令のみが使用可能な場合、*132 ページの LDR (CODE16)* を参照してください。

MOV (CODE16)

構文

```
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 : 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 ディレクティブを使用する場合にのみ適用されます。THUMB ディレクティブの後、命令構文の解釈は、Advanced RISC Machines Ltd. の Unified Assembler 構文で定義されます。

例

```
MOV r2,r3  ; ADD r2,r3,#0 のためのオペコードを生成する
```

MOV32 (THUMB)

構文

```
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	ロードするレジスタです。
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)

構文

```
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
```

注： NOP は、NOP 命令を含むアーキテクチャのバージョン (ARMv6K、ARMv6T2、ARMv7) では擬似命令ではありません。

NOP (CODE16)

構文

```
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
```

注： NOP は、NOP 命令を含むアーキテクチャのバージョン (ARMv6T2、ARMv7) では擬似命令ではありません。

アセンブラの診断

ここでは、診断メッセージのフォーマットと診断メッセージの重要度について説明します。

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
```

重要度

ARM 用 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。

- すべての警告、警告の一部の範囲、個々の警告を無効または有効にします（*48 ページの -w* 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（*38 ページの -E* を参照してください）。

アセンブリ時の警告メッセージ

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。

アセンブリ時には複数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。こうしたエラーは通常は発生することはありません。ただし、このタイプのエラーが起きた場合、ソフトウェア販売代理店か IAR システムズの技術サポートまでご報告ください。その際、問題を再現するための情報をお知らせください。次のような情報が考えられます。

- 製品名
- アセンブラのバージョン番号（アセンブラの生成するリストファイルのヘッダで確認できます）
- 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 用 IAR アセンブラへの移行

他のメーカ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 用 IAR アセンブラに使用できます。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 用 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。

はじめに

ARM 用 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、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、エラーや警告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 用 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われるものには、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 `T` 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)
            arm
```

以下の 2 つの命令は、ARMASM と IASMARM とでは解釈が異なります。ICCARM は T への参照を奇数アドレス（Thumb モードビット設定済み）として解釈しますが、ARMASM では偶数です（Thumb モードビットは設定されません）。

```
            adr     r0,T+1
            mov     r1,#T-.
```

ARMASM と ICCARM で解釈を同じにするには、:OR: を使用して Thumb モードビットを設定するか、:AND: を使用してそれを消去します。

```
            add     r0,pc,#(T-.-8) :OR: 1
            mov     r1,#(T-.) :AND: ~1

            thumb
T           nop
            end
```

代替レジスタ名

オプション -j が選択されているとき、ARM 用 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。代替レジスタ名とアセンブラのレジスタ名を次の表に示します。

代替レジスタ	アセンブラレジスタ名
A1	R0
A2	R1
A3	R2
A4	R3
V1	R4
V2	R5
V3	R6
V4	R7
V5	R8
V6	R9
V7	R10

表 31: 代替レジスタ名一覧

代替レジスタ	アセンブラレジスタ名
SB	R9
SL	R10
FP	R11
IP	R12

表 31: 代替レジスタ名一覧（続き）

レジスタの詳細については、*25 ページの レジスタシンボル*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	アセンブラニーモニック
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
BNNE	BEQ
BNPL	BMI

表 32: 代替ニーモニック

代替ニーモニック	アセンブラニーモニック
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 32: 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、他のアセンブラが使用する演算子の多くが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	アセンブラ演算子
:AND:	&
:EOR:	^
:LAND:	&&
:LEOR:	XOR
:LNOT:	!
:LOR:	||

表 33: 演算子の同義語

演算子の同義語	アセンブラ演算子
:MOD:	%
:NOT:	~
:OR:	|
:SHL:	<<
:SHR:	>>

表 33: 演算子の同義語 （続き）

注： 場合によっては、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、*51 ページの アセンブラ演算子*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、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、宛先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

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

あ

い

う

お

く

こ

し

せ

そ

た

つ

て

と

の

は

ひ

れ

ろ

わ

記号

数字