



**IAR Embedded
Workbench**

IAR アセンブラ リファ レンスガイド

Arm Limited

Arm コア

AARM-11-J

 **IAR**
SYSTEMS

著作権事項

© 1999–2017 IAR Systems AB.

本書のいかなる部分も、IAR システムズの書面による事前の同意なく複製することを禁止します。本書で解説するソフトウェアは使用許諾契約に基づき提供され、その条項に従う場合に限り使用または複製できるものとします。

免責事項

本書の内容は予告なく変更されることがあります。また、IAR システムズは、その内容についていかなる責任を負うものではありません。本書の内容については正確を期していますが、IAR システムズは誤りや記載漏れについて一切の責任を負わないものとします。

IAR システムズおよびその従業員、契約業者、本書の執筆者は、いかなる場合でも、特殊、直接、間接、または結果的な損害、損失、費用、負担、請求、要求、およびその性質を問わず利益損失、費用、支出の補填要求について、一切の責任を負わないものとします。

商標

IAR Systems、IAR Embedded Workbench、IAR Connect、C-SPY、C-RUN、C-STAT、IAR Visual State、visualSTATE、IAR KickStart Kit、I-jet、I-jet Trace、I-scope、IAR Academy、IAR、および IAR Systems のロゴタイプは、IAR Systems AB が所有権を有する商標または登録商標です。

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

Arm、Cortex、Thumb、and TrustZone は、Arm Limited の登録商標です。EmbeddedICE は Arm Limited の商標です。uC/OS-II および uC/OS-III は Micrium, Inc の商標です。CMX-RTX は CMX Systems, Inc の商標です。ThreadX は Express Logic の商標です。RTXC は、Quadros Systems の商標です。Fusion は、Unicoi Systems の商標です。

Adobe および Acrobat Reader は、Adobe Systems Incorporated の登録商標です。

その他のすべての製品名は、その所有者の商標または登録商標です。

改版情報

第 11 版：2017 年 10 月

部品番号：AARM-11-J

本ガイドは、Arm 用 IAR Embedded Workbench® のバージョン 8.2x に適用します。

内部参照：BB2、Mym8.0、asrct2010.3、V_110411、asrcarm7.80、IMAE。

目次

表	11
はじめに	13
本ガイドの対象者	13
本ガイドの使用方法	13
本ガイドの内容	14
表記規則	14
表記規則	15
命名規約	16
Arm 用 IAR アセンブラの概要	17
アセンブラプログラミングの概要	17
イントロダクション	17
モジュール方式のプログラミング	18
外部インタフェースの詳細	19
アセンブラ呼出し構文	19
オプションの受渡し	20
環境変数	20
エラーリターンコード	20
ソースフォーマット	21
アセンブラ命令	22
式、オペランド、演算子	22
整数定数	22
ASCII 文字定数	23
浮動小数点定数	23
True および false	24
シンボル	24
ラベル	25
レジスタシンボル	25
定義済シンボル	26
絶対式および再配置可能式	30
式の制限	31

リストファイルのフォーマット	32
ヘッダ	32
ボディ	32
概要	32
シンボルとクロスリファレンスの表	32
プログラミングのヒント	33
特殊機能レジスタへのアクセス	33
C 形式プリプロセッサディレクティブの使用	33
このセクションのコールフレームの使用の追跡では	33
コールフレーム情報概要を参照してください	34
コールフレーム情報の詳細	35
NAME ブロックの定義	36
COMMON ブロックの定義	37
データブロック内のソースコードに注釈をつける	38
リソースおよびスタックの深さを追跡するための	
規則を指定する	38
複雑なケースを追跡するための CFI 式の使用	40
スタック使用量解析ディレクティブ	41
CFI ディレクティブの使用例	42
アセンブラオプション	45
コマンドラインアセンブラオプションの使用	45
オプションとそのパラメータを指定	45
コマンドライン拡張 (XCL) ファイル	46
アセンブラオプションの概要	46
アセンブラオプションの概要	48
--arm	48
-B	48
-c	49
--cmse	49
--cpu	50
--cpu_mode	50
-D	50
-E	51

-e	52
--endian	52
-f	52
--fpu	53
-G	53
-g	54
-I	54
-i	55
-j	55
-L	55
-l	56
--legacy	56
-M	57
-N	57
--no_dwarf3_cfi	58
--no_it_verification	58
--no_literal_pool	58
--no_path_in_file_macros	59
-O	59
-o	60
-p	60
-r	61
-S	61
-s	61
--source_encoding	62
--suppress_vfe_header	62
--system_include_dir	63
-t	63
--thumb	63
-U	64
-w	64
-x	65

アセンブラ演算子	67
アセンブラ演算子の優先順位	67
アセンブラ演算子の概要	67
括弧演算子	67
単項演算子	68
乗算型算術演算子	68
加算型算術演算子	69
シフト演算子	69
AND 演算子	69
OR 演算子	69
比較演算子	70
アセンブラ演算子の説明	70
() 括弧	70
* 乗算	71
+ 単項プラス	71
+ 加算	71
単項マイナス	71
- 減算	72
/ 除算	72
< より小さい	72
<= 以下	73
<>, != 等しくない	73
=, == 等しい	73
> より大きい	73
>= 以上	74
&& 論理 AND	74
& ビット単位の AND	74
~ ビット単位の NOT	75
ビット単位の OR	75
^ ビットごとの排他 OR	75
% 剰余	75
! 論理否定	76
論理 OR	76

<< 論理左シフト	76
>> 論理右シフト	77
BYTE1 1 バイト目	77
BYTE2 2 バイト目	77
BYTE3 3 バイト目	77
BYTE4 4 バイト目	78
DATE 現在の日時	78
HIGH 上位バイト	78
HWRD 上位ワード	79
LOW 下位バイト	79
LWRD 下位ワード	79
SFB セクション 開始	79
SFE セクション 終了	80
SIZEOF セクション サイズ	81
UGT 符号なし大なり	81
ULT 符号なし小なり	82
XOR 論理排他 OR	82
アセンブラディレクティブ	83
アセンブラディレクティブの概要	83
アセンブラディレクティブの説明	88
モジュール制御ディレクティブ	88
シンボル制御ディレクティブ	91
モード制御のディレクティブ	93
セクション制御のディレクティブ	95
値割当てディレクティブ	99
条件付きアセンブリディレクティブ	101
マクロ処理ディレクティブ	102
リスト制御ディレクティブ	111
C 形式のプリプロセッサディレクティブ	116
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	121
アセンブラ制御ディレクティブ	124
関数ディレクティブ	127
NAME ブロックのコールフレーム情報ディレクティブ	128

COMMON ブロックのコールフレーム	
情報ディレクティブ	129
データブロックのコールフレーム情報ディレクティブ	130
リソースや CFA を追跡するためのコールフレーム情報ディレクティブ	132
スタック使用量分析のコールフレーム情報	135
アセンブラ擬似命令	137
要約	137
擬似命令の説明	138
ADR (ARM)	138
ADR (CODE16)	139
ADR (THUMB)	139
ADRL (ARM)	140
ADRL (THUMB)	141
LDR (ARM)	141
LDR (CODE16)	142
LDR (THUMB)	143
MOV (CODE16)	144
MOV32 (THUMB)	145
NOP (ARM)	145
NOP (CODE16)	145
アセンブラの診断	147
メッセージフォーマット	147
重要度	147
診断オプション	147
アセンブラの警告メッセージ	147
コマンドラインエラーのメッセージ	148
アセンブラのエラーメッセージ	148
アセンブラの致命的なエラーメッセージ	148
アセンブラの内部エラーメッセージ	148

Arm 用 IAR アセンブラへの移行	149
概要	149
Thumb コードのラベル	149
代替レジスタ名	150
代替ニーモニック	151
演算子の同義語	152
ワーニングメッセージ	153
The first register operand omitted	153
The first register operand duplicated	153
Immediate #0 omitted in Load/Store	153
索引	155

表

1: 本ガイドで使用されている表記規則	15
2: このガイドで使用されている命名規約	16
3: アセンブラの環境変数	20
4: アセンブラのエラーリターンコード	21
5: 整数定数のフォーマット	23
6: ASCII文字定数のフォーマット	23
7: 浮動小数点定数	24
8: 定義済レジスタシンボル	25
9: 定義済シンボル	26
10: シンボルとクロスリファレンスの表	32
11: バックトレース行と列付きのサンプルコード	42
12: アセンブラオプションの概要	46
13: アセンブラディレクティブの概要	83
14: モジュール制御ディレクティブ	89
15: シンボル制御ディレクティブ	91
16: モード制御のディレクティブ	93
17: セクション制御のディレクティブ	96
18: 値割当てディレクティブ	99
19: マクロ処理ディレクティブ	103
20: リスト制御ディレクティブ	112
21: C形式のプリプロセッサディレクティブ	117
22: 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	122
23: アセンブラ制御ディレクティブ	124
24: コールフレーム情報のディレクティブ	129
25: コールフレーム情報ディレクティブCOMMONブロック	130
26: データブロックのコールフレーム情報ディレクティブ	131
27: CFI式の単項演算子	132
28: CFI式の2項演算子	133
29: CFI式の3項演算子	134
30: リソースやCFAを追跡するためのコールフレーム 情報ディレクティブ	134

31: スタック使用量分析のコールフレーム情報	135
32: 擬似命令	137
33: 代替レジスタ名一覧	150
34: 代替ニーモニック	151
35: 演算子の同義語	152

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 Arm の使用方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、命令セット（チップメーカーのドキュメントを参照）
- アセンブラ言語でのプログラミングに関する基礎知識
- 組み込みシステム用アプリケーションの開発
- ホストコンピュータのオペレーティングシステム

本ガイドの使用方法

Arm 用 IAR アセンブラを使用し始めたら、最初に必ず『Arm 用 IAR アセンブラの概要』の章をお読みください。

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 Embedded Workbench を初めてお使いになられる場合は、IAR Embedded Workbench の使用方法の習得に役立つ IAR インフォメーションセンタにあるチュートリアルを一通り試していただくことをお勧めします。

本ガイドの内容

本ガイドの構成および各章の概要を以下に示します。

- 「*Arm 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*」Arm 用の IAR アセンブラで許可されていない擬似命令の一覧です。
- 「*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*」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。

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。

製品のインストールでディレクトリを参照するとき、たとえば arm¥doc、場所のフルパスを前提とします。例えば、c:¥Program Files¥IAR Systems¥Embedded Workbench N.n¥arm¥doc のようになります。ここで、バージョン番号の最初の数字は、IAR Embedded Workbench 共有コンポーネントのバージョン番号の最初の数字を反映しています。

表記規則

IAR システムズのドキュメントでは、以下の表記規則を使用します。





スタイル	用途
computer	<ul style="list-style-type: none"> 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。
parameter	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、 <code>filename.h</code> の場合、 <code>filename</code> はファイルの名前を表します。
[option]	ディレクティブのオプション部分。【と】は実際のディレクティブの一部ではなく、[、]、{、} はディレクティブの構文の一部です。
{option}	ディレクティブの必須部分。【と】は実際のディレクティブの一部ではなく、[、]、{、} はディレクティブの構文の一部です。
[option]	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	画面で表示されるメニュー、メニューコマンド、ボタン、ダイアログボックス の名前を示します。
斜体	<ul style="list-style-type: none"> 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 点リーダーは、その前の項目を任意の回数繰り返せることを示します。
	IAR Embedded Workbench® 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: 本ガイドで使用されている表記規則

命名規約

以下の命名規約は、このガイドに記述されている 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 ランタイム環境™	DLIB ランタイム環境

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

- 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
- 外部インターフェースの詳細
- 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
- このセクションのコールフレームの使用の追跡では

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あることがあります。

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については、Arm Limited ハードウェアのマニュアル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クで組み込むことができます。

大規模なプロジェクトに取り組んでいると、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。

- 19 ページの *アセンブラ呼出し構文*
- 20 ページの *オプションの受渡し*
- 20 ページの *環境変数*
- 20 ページの *エラーリターンコード*

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、*IAR Embedded Workbench® IDE User Guide for Arm* を参照してください。

アセンブラ呼出し構文

アセンブラの呼び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、`aiasmarm` 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、各アセンブリに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（52 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。 <code>set IASMARM=-L -ws</code>
IASMARM_INC	インクルードファイルを検索するディレクトリを指定します。例： <code>set IASMARM_INC=c:\myinc\</code>

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
```

コンパイラおよびリンカで使用する環境変数について詳しくは、『*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	警告が発生しました (-ws オプションを使用している場合のみ)。
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 は一般的な慣行に従って、列 8、16、24 などに拡張されています。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、エラーが発生します。

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。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

IAR システムのすべてのアセンブラでは 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。

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: 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。シンボルは以下のいずれかです。

- 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。

シンボルは文字 `a-z` または `A-Z`、`?`（クエスチョンマーク）、または `_`（アンダースコア）で始まる必要があります。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** (`-s`) オプションで切り換えることができます。詳細については、61 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。121 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を参照してください。

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。

名前	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
S0-S31	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

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

Arm 用 IAR アセンブラには、アセンブラソースファイルで使用するシンボルのセットが定義されています。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。

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_ARCH	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
__ARM_ARCH_ISA_ARM	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
__ARM_ARCH_ISA_THUMB	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
__ARM_ARCH_PROFILE	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
__ARM_BIG_ENDIAN	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
__ARM_FEATURE_CMSE	アセンブラオプション --cpu and --cmse に基づいて設定される整数。選択したプロセッサの構造に CMSE (Cortex-M セキュリティエクステンション) があり、アセンブラオプション --cmse が指定されている場合は、シンボルは 3 に設定されます。 選択したプロセッサの構造に CMSE (Cortex-M セキュリティエクステンション) があり、アセンブラオプション --cmse が指定されていない場合は、シンボルは 1 に設定されます。 シンボルは、CMSE なしのコアには未定義です。

表 9: 定義済シンボル

シンボル	値
<code>__ARM_FEATURE_CRC32</code>	CRC32 命令がサポートされる場合、このシンボルは 1 に設定されます (Armv8-A/R ではオプション)。このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FEATURE_CRYPTO</code>	CRC32 命令がサポートされる場合、このシンボルは 1 に設定されます (Neon の Armv8-A/R を指示)。このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FEATURE_DIRECTED_ROUNDING</code>	丸め方向および変換命令がサポートされている場合は、このシンボルは 1 に設定されます。このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FEATURE_DSP</code>	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FEATURE_FMA</code>	FPU が結合した浮動小数点積算 / 累積をサポートしている場合は、このシンボルは 1 に設定されます。このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FEATURE_IDIV</code>	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FEATURE_NUMERIC_MAXMIN</code>	浮動小数点の最大と最小命令がサポートされている場合は、このシンボルは 1 に設定されます。このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_FP</code>	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
<code>__ARM_MEDIA__</code>	--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。
<code>__ARM_MPCORE__</code>	--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。
<code>__ARM_NEON</code>	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。

表 9: 定義済シンボル (続き)

シンボル	値
__ARM_NEON_FP	このシンボルは、Arm C 言語拡張 (ACLE) に従って定義されます。
__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。
__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。
__ARMVFP_D16__	--fpu アセンブラオプションに基づいて設定される整数。選択された FPU が 16 D レジスタのみを持つ VFPv3 または VFPv4 ユニットの 경우、このシンボルは 1 に設定されます。それ以外の場合、シンボルは未定義です。
__ARMVFP_FP16__	--fpu アセンブラオプションに基づいて設定される整数。選択された FPU が 16 ビットの浮動小数点数のみをサポートする場合、このシンボルは 1 に設定されます。それ以外の場合、シンボルは未定義です。
__ARMVFP_SP__	--fpu アセンブラオプションに基づいて設定される整数。選択された FPU が 32 ビットの単精度のみをサポートする場合、このシンボルは 1 に設定されます。それ以外の場合、シンボルは未定義です。
__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。ビルド番号は、必ずしも後でリリースされたアセンブラの方が大きい番号になるとは限りません。
__CORE__	使用中のチップコアを示す整数です。アセンブラオプション --cpu の設定を反映した値です。使用可能な値については、『ARM 用 IAR C/C++ 開発ガイド』を参照してください。
__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）。

表 9: 定義済シンボル（続き）

シンボル	値
__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 <code>#ifdef</code> 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ーを識別します。コードがリトルエンディアンのバイトオーダーでアセンブルされる場合、番号 1 を返し、ビッグエンディアンコードが生成される場合は、番号 0 を返します。リトルエンディアンがデフォルトです。
__TID__	2 バイトからなるターゲットの識別子（数）。上位バイトはターゲットの識別を行い、Arm の IAR アセンブラでは 0x4F（= 79 進数）です。
__TIME__	hh:mm:ss フォーマットで示す現在の時刻（文字列）。
__VER__	整数形式のバージョン番号。たとえば、バージョン 6.21.2 は 6021002（数値）として返されます。

表 9: 定義済シンボル（続き）

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、アセンブリの時刻を文字列として表示させるには、次のようにして行います。

```
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、101 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、プログラムは以下のような絶対式と再配置可能式を定義することが可能です。

```

                                name      simpleExpressions
                                section MYCONST:CONST(2)
first      dc8      5              ; A relocatable label.
second     equ      10 + 5         ; An absolute expression.

                                dc8      first          ; Examples of some legal
                                dc8      first + 1       ; relocatable expressions.
                                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する式です。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。

前方参照なし

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。

外部参照禁止

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ワーニングの要約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 形式プリプロセッサディレクティブの使用

C 形式のプリプロセッサディレクティブは、他のアセンブラディレクティブの前に処理されます。そのため、マクロでプリ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、124 ページの *アセンブラ制御* 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。

このセクションのコールフレームの使用の追跡では

これらのトピックについて説明されます：

- 34 ページの *コールフレーム情報概要* を参照してください
- 35 ページの *コールフレーム情報の詳細*

以下のタスクについて説明します：

- 36 ページの *NAME* ブロックの定義
- 37 ページの *COMMON* ブロックの定義
- 38 ページのデータブロック内のソースコードに注釈をつける
- 38 ページのリソースおよびスタックの深さを追跡するための規則を指定する
- 40 ページの複雑なケースを追跡するための *CFI* 式の使用
- 41 ページのスタック使用量解析ディレクティブ
- 42 ページの *CFI* ディレクティブの使用例

リファレンス情報については、

- 128 ページの *NAME* ブロックのコールフレーム情報ディレクティブ
- 129 ページの *COMMON* ブロックのコールフレーム情報ディレクティブ
- 130 ページのデータブロックのコールフレーム情報ディレクティブ
- 132 ページのリソースや *CFA* を追跡するためのコールフレーム情報ディレクティブ
- 135 ページのスタック使用量分析のコールフレーム情報

コールフレーム情報概要を参照してください

コールフレーム情報 (*CFI*) はコールフレームの情報です。通常、コールフレームには、リターンアドレス、関数の引数、保存したレジスタ値、一時的なコンパイラ、ローカル変数が含まれます。コールフレーム情報には、2つの重要な機能をサポートするためのコールフレームに関する十分な情報があります。

- *C-SPY* は現在の *PC* (プログラム カウンタ) からコールチェーン全体を再構築して、コールチェーンの各関数のローカル変数値を表示するために、コールフレーム情報に使用できます。
- アプリケーションのスタック使用量の計算のために、可能な呼び出しの情報とコールフレームの情報を使用できます。この機能はお使いの製品ではサポートされていない場合があります。

コンパイラは、すべての *C* および *C++* ソースコードのために、自動的にコールフレーム情報を生成します。コールフレームの情報も通常、システム ライブラリの各アセンブラルーチンのために提供されます。ただし、ほかのアセンブラルーチンを持っていて、これらのルーチンを実行するときに *C-SPY* を有効にしてコールスタックを表示したい場合は、アセンブラの必要なコールフレーム情報のソースコードに注釈を追加する必要があります。スタック使用量もこの方法を使用できます (各関数呼び出しに必要な注釈を追加する方法) が、スタック使用量制御ファイルのルーチンのスタック使用量情報を指

定できます (『ARM 用 IAR C/C++ 開発ガイド』を参照)。この方法のほうが通常簡単です。

コールフレーム情報の詳細

`cfi` ディレクティブを使用して、アセンブラ ファイルにコールフレーム情報を追加できます。これらを使用して次のことを指定できます。

- コールフレームの開始アドレスは *CFA 列* (CFA) として参照されます。コールフレームには 2 つの種類があります。
 - スタックのスタックフレーム。スタックフレームには、ルーチンから返ってきたあと通常 CFA はスタックポインタの値です。
 - 静的メモリには、静的オーバーレイシステムの静的オーバーレイフレームとして使用されます。この種類のコールフレームは、Arm コアでは必要なく、またそのためサポートされていません。
- リターンアドレスを検索する方法。
- ルーチンから返す時にレジスタなどの様々なリソースを復元する方法。

各アセンブラモジュールのコールフレーム情報を追加するとき、次のことをする必要があります。

- 1 追跡されるリソースを説明している *names block* を提供します。
- 2 追跡されるリソースを定義する *common block* を提供して、そのデフォルト値を指定します。この情報はコンパイラが使用する呼び出し規約に関連している必要があります。
- 3 ソースコードで使用するリソースを注釈をつけることについては、コールフレームで実行された変更を説明します。通常、これには、いつスタックポインタが変更されたか、いつ保護レジスタがスタックに待避、復帰したかについての情報が含まれます。

これを行うには、追跡する各リソースの規則を指定する連続したソースコードの含む *data block* を定義する必要があります。簡易規則が十分でない場合は、代わりに *CFI* 式を使用できます。

呼び出し規約に関する詳細記述では、広範なコール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。コール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。

NAME ブロックの定義

NAME ブロックは、プロセッサで使用可能なリソースを宣言するために使用します。NAME 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。

NAME 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 NAME ブロックは 1 つだけです。

NAME ブロック内には、4 つの異なる宣言があります。リソース宣言、スタックフレーム宣言、静的オーバーレイ宣言、またはベースアドレス宣言。

- リソースを宣言するには、以下のいずれかのディレクティブを使用します。

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

パラメータは、リソースの名前とリソースのサイズ（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。仮想リソースは論理的な概念であり、プロセッサレジスタなどの「物理」リソースと対比されません。仮想リソースは通常、リターンアドレスに使用します。

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。

リソースは、2 つ以上のパーツから成る複合リソースを使用することもできます。複合リソースの構成を宣言するには、次のようにディレクティブを使用します。

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

パートはコンマで区切ります。リソースとそのパートは、上で説明したように、既にリソースとして宣言されている必要があります。

- スタックフレーム CFA を宣言するには、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）、メモリアイプ（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。

コール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。

COMMON 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、*COMMON* ブロックを使用します。通常、使用される各呼び出し規約に *COMMON* ブロックが 1 つずつあります。

COMMON 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 *NAME* 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* は呼び出し関数を格納するメモリです。*COMMON* 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。

COMMON ブロックの内部には、*COMMON* ブロックで利用できる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。129 ページの *COMMON* ブロックのコールフレーム情報ディレクティブを参照してください。これらのディレクティブの使用方法については、38 ページのリソースおよびスタックの深さを追跡するための規則を指定するおよび 40 ページの複雑なケースを追跡するための *CFI* 式の使用を参照してください。

COMMON 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 *COMMON* ブロックを開始するために使用される名前です。

データブロック内のソースコードに注釈をつける

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 **COMMON** ブロックの名前です。

現在のデータブロックの当該コードが、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。

現在のデータブロックの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、データブロックに利用できるディレクティブを使用して、リソースの値を操作できます。130 ページの *データブロックのコールフレーム情報ディレクティブ* を参照してください。これらのディレクティブの使用方法については、38 ページの *リソースおよびスタックの深さを追跡するための規則を指定する* および 40 ページの *複雑なケースを追跡するための CFI 式の使用* を参照してください。

リソースおよびスタックの深さを追跡するための規則を指定する

個々のリソースの追跡情報を記述するために、特別な構文の2セットの簡易規則が用意されています。

- リソース追跡のための規則

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

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

- スタックの深さを追跡するための規則 (CFA)

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

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

これらの規則は、COMMON 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、決定されたオペレータの完全な CFI 式を使用して情報を記述できます。40 ページの複雑なケースを追跡するための CFI 式の使用を参照してください。ただし、可能な限り、CFI 式ではなく規則を使用してください。

リソース追跡のための規則

コールフレームを 1 つ戻す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、言い換えれば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ので、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 R11 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 R11 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（コールフレームを 1 つ戻す場合に）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。例えば R11 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 R11 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 R11 が一時的にレジスタ R12 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 R11 R12
```

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 R11 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 R11 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。

スタックの深さを追跡するための規則 (CFA)

リソース用の規則と違い、CFA 用の規則にはコールフレームの先頭のアドレスを記述します。コールフレームには、アセンブラ呼び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、現在のスタックフレームの先頭のアドレスを計算する方法を記述します。

各スタックフレーム CFA には、スタックポインタが関連付けられています。コールフレームを 1 つ戻ると、関連スタックポインタは現在の CFA で復元されます。スタックフレーム CFA に対しては 2 つの規則があります。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、関連するスタックポインタ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、スタックポインタ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
```

複雑なケースを追跡するための CFI 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、*コールフレーム情報式* (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 式には 3 セットの演算子が使用できます。

- 単項演算子
- 2 項演算子
- 3 項演算子

ほぼ、通常のアセンブラ式と同じ演算子を使用できます。

この例には、R12 がその元の値に格納されます。ただし、それを保存する代わりに、2 つのインクリメント後の効果は、影響が減算命令により未実行となります。

AddTwo:

```
cfi block addTwoBlock using myCommon
cfi function addTwo
cfi nocalls
cfi r12 samevalue
add @r12+, r13
cfi r12 sub(r12, 2)
add @r12+, r13
cfi r12 sub(r12, 4)
sub #4, r12
cfi r12 samevalue
ret
cfi endblock addTwoBlock
```

CFI 式の演算子の使用のための構文については、132 ページの *リソース* や *CFA* を追跡するための *コールフレーム情報ディレクティブ* を参照してください。

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 TAILCALL、CFI INDIRECTCALL、および CFI NOCALLS) は、スタック使用量解析に必要なコールグラフを構築するために使用されます。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。

呼び出しを実行する命令の前に CFI FUNCALL、CFI TAILCALL、CFI INDIRECTCALL ディレクティブはすぐに配置する必要があります。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。

CFI ディレクティブの使用例

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び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 は関数呼び出しの後に復元する必要があります。

以下の短いソースコードと、対応するコールフレームの情報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 はコール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	R1	R4-R11	R14	R0、R2、R3、R12	アセンブラコード
00000000	R13 + 0	SAME	SAME	SAME	未定義	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 11: バックトレース行と列付きのサンプルコード

各行は、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 の行は最初の行であり、関数に使用された呼び出し規約の結果です。

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ーケーションです。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。関数からの終了に復元される必要がないので、定義されていないレジスタがいくつかあります。

NAME ブロックの定義

上の例で指定する NAME 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
```

COMMON 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：CFA とリソースが関連しているので、R13 は CFI ディレクティブを使用して変更できません。

データブロックの定義

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。

```
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
```

アセンブラオプション

- コマンドラインアセンブラオプションの使用
- アセンブラオプションの概要
- アセンブラオプションの概要

コマンドラインアセンブラオプションの使用

アセンブラオプションはアセンブラのデフォルトの動きの変更を指定できるパラメータです。コマンドラインからオプションを指定できます。詳細は IAR Embedded Workbench® IDE 内のこのセクションに記載されています。



IAR Embedded Workbench® IDE User Guide for Arm では、IDE でのアセンブラオプションの設定方法および利用可能なオプションのリファレンス情報が記載されています。

オプションとそのパラメータを指定

コマンドラインからアセンブラオプションを設定するには、`iasmarm` コマンドの後にそれらを含めます。

```
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（パスの先頭も可能）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
--arm	アセンブラディレクティブ CODE のデフォルトモードを Arm に設定します
-B	マクロ実行情報の出力
-c	条件リスト
--cmse	CMSE 安全部ジェクトの生成を有効にする
--cpu	コア設定
--cpu_mode	アセンブラディレクティブ CODE のデフォルトモードを設定します
-D	プリプロセッサシンボルを定義
-E	エラーの最大数
-e	ビッグエンディアンのバイト順でコードの生成
--endian	コードおよびデータのバイトオーダを指定
-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

表 12: 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
-g	システムインクルードファイルの自動検索を無効にする
-I	ヘッダファイルの検索パスを追加
-i	インクルードされたテキストを一覧表示
-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する
-L	パスへのリストファイルを生成
-l	リストファイルを生成
--legacy	古いツールチェーンとリンク可能なコードを生成
-M	マクロの引用符
-N	アセンブラリストからヘッダを除外
--no_dwarf3_cfi	DWARF 3 の呼出しフレーム命令の生成を無効にします
--no_it_verification	次の IT 命令の状態の検証を中止
--no_path_in_file_macros	シンボル <code>__FILE__</code> および <code>__BASE_FILE__</code> のリターン値からパスを削除
-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	デバッグ情報を生成
-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
--source_encoding	ソースファイルのエンコードを指定
--suppress_vfe_header	VFE ヘッダー情報の生成を無効にする
--system_include_dir	システムインクルードファイルのパスを指定
-t	タブ間隔
--thumb	アセンブラディレクティブ CODE のデフォルトモードを Thumb に設定
-U	シンボルの定義を解除
--use_unix_directory_separators	パス内で / をディレクトリの区切り文字として使用
-w	ワーニングを無効にする
-x	クロスリファレンスをインクルードする

表 12: 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、IDE では、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のインスタントチェックは実行しません。

--arm

構文

--arm

説明

このオプションを使用して、Arm をアセンブラディレクティブ CODE のデフォルトモードにします。

関連項目

50 ページの --cpu_mode。



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] を使用します。

-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

55 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

55 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]

--cmse

構文

`--cmse`


説明

このオプションを ARMv8-M の TrustZone のターゲットセキュアモードに使用します。このオプションを有効にすると、命令 MRS と MSR を使用したサフィックス `_NS` のシステムレジスタにアクセスできます。また、命令 SG、TTA、TTAT、BLXNS、および BXNS を使用できます。




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] を使用します。

--cpu

構文	<code>--cpu target_core</code>
パラメータ	<p><code>target_core</code> 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。デフォルト値は ARM7TDMI です。</p>
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	<p>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、『<i>ARM 用 IAR C/C++ 開発ガイド</i>』を参照。</p> <p> [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]</p>

--cpu_mode

構文	<code>--cpu_mode {arm thumb}</code>
パラメータ	<p><code>arm</code> (デフォルト) ARM モードを使用します。</p> <p><code>thumb</code> Thumb モードを使用します。</p>
説明	<p>このオプションを使用して、アセンブラオプション <code>CODE</code> にモードを選択します。</p> <p> 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] を使用します。</p>

-D

構文	<code>-Dsymbol [=value]</code>
パラメータ	<p><code>symbol</code> 定義するシンボル名。</p> <p><code>value</code> シンボルの値。値を指定しない場合、1 が使用されます。</p>

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が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 -DFRAME RATE=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、生成されるコード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、45 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
```

関連項目

46 ページの [コマンドライン拡張\(XCL\) ファイル](#)。



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) 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] > [#include されたテキスト]

-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

151 ページの *代替ニーモニック* 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] > [代替ニーモニック]、[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 `1st` 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.1st` 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、45 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] を使用します。

-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、他の表記法に合わ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

55 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]

--no_dwarf3_cfi

構文

`--no_dwarf3_cfi`

説明

このオプションを使用して、DWARF 3 の呼出しフレーム命令の生成を無効にします。これにより、デバッグエクスペリエンスが低下しますが、DWARF 3 をサポートしていないデバッガでのロードが可能になることがあります。



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] を使用します。

--no_it_verification

構文

`--no_it_verification`

説明

このオプションを使用して、次の IT 命令の状態の検証を中止します。



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] を使用します。

--no_literal_pool

構文

`--no_literal_pool`

説明

データバス経由でのリードアクセスが禁止されているメモリアドレス範囲から実行するコードに対しては、このオプションを使用してください。

オプション `--no_literal_pool` によって、アセンブラは、LDR のリテラルプールを使用するかわりに、MOV32 疑似命令を使用します。他の命令では、データバス経由のリードアクセスを発生させることに注意してください。

このオプションは、リンカが実行するライブラリの自動選択にも影響します。IAR- 固有の ELF 属性は、オプション `--no_literal_pool` によってコンパイルされたライブラリを使用すべきかどうか決定するのに使用されます。

オプション `--no_literal_pool` は、ARMv6-M、ARMv7-M、および ARMv8-M アーキテクチャを持つコアに対してのみ許可されます。

関連項目 『ARM 用 IAR C/C++ 開発ガイド』の同じ名前のコンパイラとリンカオプション。



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] を使用します。

--no_path_in_file_macros

構文 `--no_path_in_file_macros`

説明 このオプションは、定義済プリプロセッサシンボル `__FILE__` および `__BASE_FILE__` のリターン値からパスを除外する場合に使用します。



このオプションは、IDE では使用できません。

-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

<i>filename</i>	オブジェクトコードは指定ファイルに格納されます。
<i>directory</i>	オブジェクトコードはファイル（ファイル拡張子 <code>.o</code> ）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、45 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

<i>lines</i>	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

55 ページの *-L*56 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]


--source_encoding

構文	<code>--source_encoding {locale utf8}</code>	
パラメータ	locale	デフォルトのソースエンコードは、システムのロケールのエンコードです。
	utf8	デフォルトのソースエンコードは、UTF-8 エンコードです。
説明	<p>バイト オーダーマーク (BOM) がないソースファイルを読み込むとき、このオプションを使用してエンコードを指定します。</p> <p>このオプションが指定されていなくて、ソースファイルに BOM がない場合、Raw エンコードが使用されます。</p>	
関連項目	<p>エンコーディングの詳細は、『<i>ARM 用 IAR C/C++ 開発ガイド</i>』を参照してください。</p> <p> 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] を使用します。</p>	


--suppress_vfe_header

構文	<code>--suppress_vfe_header</code>	
説明	<p>このオプションを使用して、生成されたオブジェクトコードにある VFE（仮想関数除去）ヘッダー情報の自動生成を中止します。</p> <p>このオプションは、以下の 2 つの場合に使用できます。</p> <ul style="list-style-type: none"> ● リンカ VFE の最適化は自動的にオフになっているか確認してください。 ● アセンブラソースコードに VFE 情報を手動で提供します。 	
関連項目	<p>『<i>ARM 用 IAR C/C++ 開発ガイド</i>』のリンカオプション <code>--vfe</code>。</p> <p> 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] を使用します。</p>	

--system_include_dir

構文	<code>--system_include_dir path</code>
パラメータ	<code>path</code> システムインクルードファイルのパス。
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	<code>-tn</code>
パラメータ	<code>n</code> タブの間隔 (2 ～ 9)。
説明	<p>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。</p> <p>このオプションは、<code>-L</code> オプションまたは <code>-l</code> オプションとともに使用すると便利です。</p>
関連項目	55 ページの <code>-L</code> 56 ページの <code>-l</code>  [プロジェクト] > [オプション] > [アセンブラ] > [リスト] > [タブの間隔]

--thumb

構文	<code>--thumb</code>
説明	このオプションを使用して、Thumb をアセンブラディレクティブ CODE のデフォルトモードにします。
関連項目	50 ページの <code>--cpu_mode</code> 。



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] を使用します。

-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__
```

関連項目

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

パラメータなし	すべてのワーニングを無効にします。
+	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 <i>n</i> のみを有効にします。
-n	ワーニング <i>n</i> のみを無効にします。
+m-n	<i>m</i> から <i>n</i> のワーニングを有効にします。
-m-n	<i>m</i> から <i>n</i> 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		<p>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。</p> <p>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。</p> <p>-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。</p>
例		<p>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。</p> <pre>iasmarm prog -w-0</pre> <p>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。</p> <pre>iasmarm prog -w-0-8</pre>
関連項目		<p>147 ページの <i>アセンブラの診断</i>。</p> <p>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。</p> <p> [プロジェクト] > [オプション] > [アセンブラ] > [診断]</p>
-x		
構文		-x{D I 2}
パラメータ		<p>D プリプロセッサ #defines をインクルードします。</p> <p>I 内部シンボルをインクルードします。</p> <p>2 2 行間隔をインクルードします。</p>
説明		<p>リストの最後に相互参照リストを作成します。</p> <p>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。</p>

関連項目

55 ページの *-L*

56 ページの *-I*



[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める]

アセンブラ演算子

- アセンブラ演算子の優先順位
- アセンブラ演算子の概要
- アセンブラ演算子の説明

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「()」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。

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。また『*Arm 用 IAR アセンブラへの移行*』も参照してください。

括弧演算子

優先順位：1

() 括弧。

単項演算子

優先順位: 1

+	単項プラス。
-	単項マイナス。
!, :LNOT:	論理 NOT。
~, :NOT:	ビット単位の NOT。
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	セクション開始。
SFE	セクション終了。
SIZEOF	セクションサイズ。

乗算型算術演算子

優先順位: 2

*	乗算。
/	除算。
%, :MOD:	剰余。

加算型算術演算子

優先順位 : 3

+	加算。
-	減算。

シフト演算子

優先順位 : 2.5-4

>>	論理右シフト (4)。
:SHR:	論理右シフト (2.5)。
<<	論理左シフト (4)。
:SHL:	論理左シフト (2.5)。

AND演算子

優先順位 : 3-8

&&	論理 AND (5)。
:LAND:	論理 AND (8)。
&	ビット単位の AND (5)。
:AND:	ビット単位の AND (3)。

OR 演算子

優先順位 : 3-8

, :LOR:	論理 OR (6)。
	ビット単位の OR (6)。
:OR:	ビット単位の OR (3)。
XOR	論理排他 OR (6)。
:LEOR:	論理排他 OR (8)。

^	ビット単位の排他 OR (6)。
:EOR:	ビット単位の排他 OR (3)。

比較演算子

優先順位 : 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。

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。
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

優先順位	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

優先順位	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	<pre>2*2 -> 4 -2*2 -> -4</pre>

+ 単項プラス

優先順位	1
説明	単項プラス演算子 ; 何も実行しない。
例	<pre>+3 -> 3 3*+2 -> 6</pre>

+ 加算

優先順位	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	<pre>92+19 -> 111 -2+2 -> 0 -2+-2 -> -4</pre>

単項マイナス

優先順位	1
説明	<p>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。</p> <p>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。</p>

例	-3 -> -3 3*-2 -> -6 4--5 -> 9
---	-------------------------------------

- 減算

優先順位	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

優先順位	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
---	--

< より小さい

優先順位	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

優先順位	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	<pre>1 <= 2 -> 1 2 <= 1 -> 0 1 <= 1 -> 1</pre>

<>, != 等しくない

優先順位	7
説明	<> は、2 つのオペランドの値が等しい場合に 0（偽）となり、等しくない場合は 1（真）となります。
例	<pre>1 <> 2 -> 1 2 <> 2 -> 0 'A' <> 'B' -> 1</pre>

=, == 等しい

優先順位	7
説明	= は、2 つのオペランド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	<pre>1 = 2 -> 0 2 == 2 -> 1 'ABC' = 'ABCD' -> 0</pre>

> より大きい

優先順位	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	<pre>-1 > 1 -> 0 2 > 1 -> 1 1 > 1 -> 0</pre>
---	--

>= 以上

優先順位	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	<pre>1 >= 2 -> 0 2 >= 1 -> 1 1 >= 1 -> 1</pre>
---	--

&& 論理 AND

優先順位	5
------	---

:LAND: の優先順位は 8 です。

説明	&& または同義語 :LAND: 2 つの整数オペランドの間で論理 AND 演算を行います。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。
----	--

例	<pre>1010B && 0011B -> 1 1010B && 0101B -> 1 1010B && 0000B -> 0</pre>
---	---

& ビット単位の AND

優先順位	5
------	---

:AND: の優先順位は 3 です。

説明	& または同義語 :AND: は整数オペランドの間でビットごとの AND 演算を行います。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 AND です。
----	---

例	<pre>1010B & 0011B -> 0010B 1010B & 0101B -> 0000B 1010B & 0000B -> 0000B</pre>
---	--

～ ビット単位の NOT

優先順位	1
説明	<p>～または同義語 :NOT: オペランドのビット単位の NOT 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。</p>
例	<p>～ 1010B -> 111111111111111111111111111111110101B</p>

| ビット単位の OR

優先順位	6
説明	<p> または同義語 :OR: オペランドのビット単位の OR を行います。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。</p>
例	<pre>1010B 0101B -> 1111B 1010B 0000B -> 1010B</pre>

^ ビットごとの排他 OR

優先順位	6
	:EOR: の優先順位は 3 です。
説明	^ または同義語 :EOR: オペランドのビット単位の XOR を行います。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。
例	<pre> 1010B ^ 0101B -> 1111B 1010B ^ 0011B -> 1001B </pre>

% 剩余

優先順位	2
説明	<p>% または同義語 :MOD: は左側のオペランドを右側のオペランドで除算した余りを計算します。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。</p> <p>$x \% y$ 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。</p>

例	2 % 2 -> 0 12 % 7 -> 5 3 % 2 -> 1
---	---

! 論理否定

優先順位	1
説明	! または同義語 :LNOT: は引数を否定します。
例	! 0101B -> 0 ! 0000B -> 1

|| 論理 OR

優先順位	6
説明	または同義語 :LOR: は 2 つの整数オペランドの間で論理和演算を行います。
例	1010B 0000B -> 1 0000B 0000B -> 0

<< 論理左シフト

優先順位	4
説明	<< または同義語 :SHL: 左側のオペランドを、左側にシフトします（左側のオペランドは符号なしとして扱う）。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 -> 11111111111100000B 14 << 1 -> 28

>> 論理右シフト

優先順位	4
説明	>> または同義語 :SHR: 左側のオペランドを、右側にシフトします（左側のオペランドは符号なしとして扱う）。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	<pre>01110000B >> 3 -> 00001110B 111111111111111B >> 20 -> 0 14 >> 1 -> 7</pre>

BYTE1 1 バイト目

優先順位	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	<pre>BYTE1 0xABCD -> 0xCD</pre>

BYTE2 2 バイト目

優先順位	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	<pre>BYTE2 0x12345678 -> 0x56</pre>

BYTE3 3 バイト目

優先順位	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	<pre>BYTE3 0x12345678 -> 0x34</pre>

BYTE4 4 バイト目

優先順位	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

優先順位	1												
説明	<p>DATE は現在のアセンブリの開始した時の時間を取得します。</p> <p>DATE 演算子は絶対引数（式）をとり、以下を返します。</p> <table><tr><td>DATE 1</td><td>現在の秒 (0-59)</td></tr><tr><td>DATE 2</td><td>現在の分 (0-59)</td></tr><tr><td>DATE 3</td><td>現在の時 (0-23)</td></tr><tr><td>DATE 4</td><td>現在の日 (1-31)</td></tr><tr><td>DATE 5</td><td>現在の月 (1-12)</td></tr><tr><td>DATE 6</td><td>現在の年 MOD 100 (1998 ->98、2000 ->00、2002 ->02)。</td></tr></table>	DATE 1	現在の秒 (0-59)	DATE 2	現在の分 (0-59)	DATE 3	現在の時 (0-23)	DATE 4	現在の日 (1-31)	DATE 5	現在の月 (1-12)	DATE 6	現在の年 MOD 100 (1998 ->98、2000 ->00、2002 ->02)。
DATE 1	現在の秒 (0-59)												
DATE 2	現在の分 (0-59)												
DATE 3	現在の時 (0-23)												
DATE 4	現在の日 (1-31)												
DATE 5	現在の月 (1-12)												
DATE 6	現在の年 MOD 100 (1998 ->98、2000 ->00、2002 ->02)。												
例	<p>アセンブリ日時は以下のように指定します。</p> <p>today: DC8 DATE 5, DATE 4, DATE 3</p>												

HIGH 上位バイト

優先順位	1
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

優先順位	1
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

優先順位	1
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ドバイトの符号なし 8 ビット整数値です。
例	LOW 0xABCD -> 0xCD

LWRD 下位ワード

優先順位	1
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

構文	SFB(<i>section</i> [{+ -} <i>offset</i>])	
優先順位	1	
パラメータ	<i>section</i>	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。
	<i>offset</i>	開始アドレスからの任意指定オフセット。 <i>offset</i> を省略するときには丸括弧はオプションです。

```

例
name      sectionBegin
section MYCODE:CODE(2)    ; MYCODE の
                           ; 前方宣言
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

構文	SFE (section [{+ -} offset])
優先順位	1
パラメータ	<div><div><div>section</div><div>offset</div></div><div>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。 開始アドレスからの任意指定オフセット。offset を省略するときには丸括弧はオプションです。</div></div>
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、セクションが終わって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 演算子を使用して計算できます。

SIZEOF セクション サイズ

構文	SIZEOF <i>section</i>	
優先順位	1	
パラメータ	<i>section</i>	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	<p>これらの 2 つのファイルは、size の値をセクション MYCODE のサイズに指定します。</p> <pre>Table.s: module table section MYCODE:CODE ; Forward declaration of MYCODE. section SEGTAB:CONST(2) data size dc32 sizeof(MYCODE) end Application.s: module application section MYCODE:CODE(2) code nop ; application のためのブレースフォルダ end</pre>	

UGT 符号なし大なり

優先順位	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	<pre>2 UGT 1 -> 1 -1 UGT 1 -> 1</pre>	

ULT 符号なし小なり

優先順位

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 論理排他 OR

優先順位

6

説明

XOR または同義語 :LEOR: は、左右のオペランドどちらか片方がゼロでもう片方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両方のオペランドがゼロ以外である場合に 0（偽）となります。XOR は 2 つのオペランドに対して排他的論理和演算を行うときに使用します。

注: :LEOR: の優先順位は 8 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の概要を説明し、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。

- 88 ページのモジュール制御ディレクティブ
- 91 ページのシンボル制御ディレクティブ
- 93 ページのモード制御のディレクティブ
- 95 ページのセクション制御のディレクティブ
- 99 ページの値割当てディレクティブ
- 101 ページの条件付きアセンブリディレクティブ
- 102 ページのマクロ処理ディレクティブ
- 111 ページのリスト制御ディレクティブ
- 116 ページの C 形式のプリプロセッサディレクティブ
- 121 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
- 124 ページのアセンブラ制御
- 127 ページの関数ディレクティブ
- 128 ページの NAME ブロックのコールフレーム情報ディレクティブ
- 129 ページの COMMON ブロックのコールフレーム情報ディレクティブ
- 130 ページのデータブロックのコールフレーム情報ディレクティブ
- 132 ページのリソースや CFA を追跡するためのコールフレーム情報ディレクティブ
- 135 ページのスタック使用量分析のコールフレーム情報

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
<code>_args</code>	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
<code>\$</code>	ファイルをインクルードします。	アセンブラ制御

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
#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
#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
#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++ スタイル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
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
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	コールフレーム情報を指定します。	コールフレーム情報
CODE	関連のアセンブラオプションの設定に応じて、これ以降の命令は Arm または Thumb 命令として解釈されます。	モード制御
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。下位互換性のために保持されています。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

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
DEFINE	ファイル全体で有効な値を定義します。	値の割当て
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（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

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
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
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。 認識されますが、無視されます。	リスト制御
LSTREP	繰り返しディレクティブで生成された行の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
SETA	一時値を割り当てます。	値の割当て
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

表 13: アセンブラディレクティブの概要 (続き)

アセンブラディレクティブの説明

以下のページでは、アセンブラディレクティブについてのリファレンス情報を提供します。

モジュール制御ディレクティブ

構文	AAPCS [<i>modifier</i> [...]] END NAME <i>symbol</i> PRESERVE8 PROGRAM <i>symbol</i> REQUIRE8 RTMODEL <i>key</i> , <i>value</i>	
パラメータ	<i>key</i>	キーを指定するテキスト文字列。
	<i>modifier</i>	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。
	<i>symbol</i>	モジュールに割り当てられる名前。
	<i>value</i>	値を指定するテキスト文字列。
説明	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、31 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
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
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
PROGRAM	モジュールを開始します。	外部参照禁止 絶対
RTMODEL	ランタイムモデル属性を宣言します。	適用されません。

表 14: モジュール制御ディレクティブ

プログラムモジュールの開始

プログラムモジュールを開始したり、IAR XLINK リンカ、IAR XAR ライブラリビルダ、IAR XLIB ライブラリアンによって参照するために名前を割り当てるには、NAME または PROGRAM を使用します。

プログラムモジュールは、その他のモジュールがこれらを参照しない場合にも、XLINK によって無条件にリンクされます。

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。また END ディレクティブは、ファイル内のモジュールを終了させます。

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ため、ユーザが検証する必要があります。

ランタイムモデル属性の宣言

モジュール間の互換性を確保するには RTMODEL を使用します。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には、『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

次の例は 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

構文	EXTERN <i>symbol</i> [, <i>symbol</i>] ... EXTWEAK <i>symbol</i> [, <i>symbol</i>] ... IMPORT <i>symbol</i> [, <i>symbol</i>] ... PUBLIC <i>symbol</i> [, <i>symbol</i>] ... PUBWEAK <i>symbol</i> [, <i>symbol</i>] ... REQUIRE <i>symbol</i>											
パラメータ	<i>label</i> <i>symbol</i>	C/C++ シンボルのエイリアスとして使用するラベル。 インポートまたはエクスポートされるシンボル。										
説明	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。											
	<table><tr><th>ディレクティブ</th><th>説明</th></tr><tr><td>EXTERN, IMPORT</td><td>外部シンボルをインポートします。</td></tr><tr><td>EXTWEAK</td><td>外部シンボルをインポートします。シンボルは定義できません。</td></tr><tr><td>OVERLAY</td><td>認識されますが、無視されます。</td></tr><tr><td>PUBLIC</td><td>他のモジュールにシンボルをエクスポートします。</td></tr></table>	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。シンボルは定義できません。	OVERLAY	認識されますが、無視されます。	PUBLIC	他のモジュールにシンボルをエクスポートします。	
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。シンボルは定義できません。											
OVERLAY	認識されますが、無視されます。											
PUBLIC	他のモジュールにシンボルをエクスポートします。											

表 15: シンボル制御ディレクティブ

ディレクティブ	説明
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 (続き)

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。

ディレクティブ	説明
CODE	アセンブラオプション <code>--arm</code> 、 <code>--cpu_mode</code> 、または <code>--thumb</code> に応じて、これ以降の命令は Arm または Thumb 命令として解釈されます。
CODE16	これ以降の命令は、従来の CODE16 構文を使用して、16 ビット (Thumb) 命令としてアセンブルされます。CODE16 内のラベルは bit 0 が 1 に設定されます。2 バイトの境界整列が強制されます。
DATA	コードセクション内で領域を定義します。ラベルは CODE32 領域として扱われます。
THUMB	これ以降の命令は、16 ビット Thumb 命令または 32 ビット Thumb-2 命令（指定コアで Thumb-2 命令セットがサポートされている場合）のいずれかとしてアセンブルされます。アセンブラ構文は、Arm Limited 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

Thumb および Arm 間でプロセッサモードを変更するには、BX 命令 (Branch および Exchange) 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

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                ; Branch to armEntry, and
                                ; change execution mode.
    nop                                ; For alignment only.
    arm
armEntry
    ; ...

end

```

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                ; Change to data mode, so
                                ; that bit 0 is not set
                                ; on labels.

dataLabel dc32     0x12345678
            dc32     0x12345678

end

```

セクション制御のディレクティブ

構文

```

ALIGN align [,value]
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。 align 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 の場合、セクションフラグメント内のシンボルが参照されていないければ、このセクションフラグメントはリンカによって廃棄されることになります。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	メモリタイプで CODE、CONST、または DATA のいずれかです。
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

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、31 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
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。	外部参照禁止 絶対

表 17: セクション制御のディレクティブ

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。	
STACK	スタックセグメントを開始します。	

表 17: セクション制御のディレクティブ (続き)

名前付き絶対形式セグメントを開始します

ASEGN を使用して、アドレス *address* にある名前付き絶対形式セグメントを開始します。

このディレクティブを使用すると、セグメントのメモリタイプを指定できるというメリットがあります。

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。

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

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

```

                                name    calculate
                                extern  subrtn,divrtn

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

                                section MYCODE:CODE(2)
                                arm
main       ldr      r0,=funcTable   ; Get address, and
                                ldr      pc,[r0]           ; jump to it.
                                end

```

セクションのアライメント

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、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 で割り切れる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 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。埋め込みバイトの値は 0 ～ 255 の範囲である必要があります。

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

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。

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)    ; Start a relocatable data
                                                                ; section aligned to a
                                                                ; 64-byte boundary.
                                data
target1    ds16    1    ; Two bytes of data.
                                alignram 6    ; 64 バイト境界にアラインメント
results    ds8    64    ; Create a 64-byte table, and
target2    ds16    1    ; two more bytes of data.
                                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

```

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
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。

表 18: 値割当てディレクティブ

ディレクティブ	説明
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 (続き)

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。

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。

```
cons      name    table
          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
```

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

DEFINEを使用して、ディレクティブを含むモジュールに認識させるシンボルを定義します。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。

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

<i>condition</i>	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。
<i>string1</i> = <i>string2</i>	<i>string1</i> と <i>string2</i> の長さや内容が同じである場合に、この条件は真となります。
<i>string1</i> <> <i>string2</i>	<i>string1</i> と <i>string2</i> 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効です。

（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          ; This,
            ?add     r1,r1,0xFF      ; and this,
            adds     r1,r1,#0xFF     ; are the same as this.

            end
```

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

<i>actual</i>	置換される文字列。
<i>argument</i>	シンボル引数名。
<i>expr</i>	式。
<i>formal</i>	<i>actual</i> (REPTC) または各 <i>actual</i> (REPTI) 文字列で置換される引数。
<i>name</i>	マクロの名前。

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 マクロ呼び出しによりアセンブラはマクロプロセッサ（エキスパンダ）を起動します。マクロエキスパンダは、（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キスパンダからの出力に切り替えます。マクロエキスパンダは、要求されたマクロ定義からの入力を取得します。

マクロエキスパンダは、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、エキ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。

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。

```
errMac      name      errMacro
             extern   abort
             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び出します。

```
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    text
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 の各ブロック内では使用できません。

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

マクロ呼び出し内で引用符 < と > をペアで使用するにより、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 (57 ページの **-M** を参照)。

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	fill
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	

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。

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 `putc` への一連の呼び出しをアセンブルしています。

```

                                name    reptc
                                extern  putc
                                section MYCODE:CODE(2)

banner    reptc    chr, "Welcome"
           movs    r0, #'chr'        ; Pass char as parameter.
           bl      putc
           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  putc
    11                                section MYCODE:CODE(2)
    12
    13                                banner    reptc    chr, "Welcome"
    14                                movs    r0, #'chr'    ; Pass char as
parameter
    15                                bl      putc
    16                                endr
    16.1  00000000 5700B0E3    movs    r0, #'W'        ; Pass char as
parameter
    16.2  00000004 .....    bl      putc
    16.3  00000008 6500B0E3    movs    r0, #'e'        ; Pass char
as
    16.4  0000000C .....    bl      putc
    16.5  00000010 6C00B0E3    movs    r0, #'l'        ; Pass char
as parameter.
    16.6  00000014 .....    bl      putc
    16.7  00000018 6300B0E3    movs    r0, #'c'        ; Pass char
as parameter.
    16.8  0000001C .....    bl      putc
    16.9  00000020 6F00B0E3    movs    r0, #'o'        ; Pass char
as parameter.
    16.10 00000024 .....    bl      putc
    16.11 00000028 6D00B0E3    movs    r0, #'m'        ; Pass char
as parameter.
    16.12 0000002C .....    bl      putc
    16.13 00000030 6500B0E3    movs    r0, #'e'        ; Pass char
as parameter.
    16.14 00000034 .....    bl      putc
    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                                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

```

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び出しとリターンオーバー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      ; Pointer to buffer.
          ldr      r2,=ioPort      ; Pointer to ioPort.
          ldr      r3,=512        ; Size of buffer.
          add      r3,r3,r1        ; Address of first byte
                                      ; after buffer.
loop      ldrb     r4,[r1],#1      ; Read a byte of data, and
          strb     r4,[r2]        ; write it to the ioPort.
          cmp      r1,r3          ; Reached first byte after?
          bne      loop           ; No: repeat.
          bx       lr             ; Return.

ioPort    equ      0x0100

                                section MYDATA:DATA (2)
                                data
buffer    ds8      512            ; Reserve 512 bytes.

                                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           ; Pointer to buffer.
          ldr    r2,=port          ; Pointer to ioPort.
          ldr    r3,=size          ; Size of buffer.
          add    r3,r3,r1          ; Address of first byte
                                   ; after buffer.
          loop   ldrb    r4,[r1],#1 ; Read a byte of data, and
          strb   r4,[r2]           ; write it to the ioPort.
          cmp    r1, r3            ; Reached first byte after?
          bne    loop             ; No: repeat.
          endm

ioPort     equ      0x0100

          section MYDATA:DATA (2)
          data
buffer     ds8      512            ; Reserve 512 bytes.

          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。さもないければ、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。

リスト制御ディレクティブ

構文	COL <i>columns</i>
	LSTCND{+ -}
	LSTCOD{+ -}
	LSTEXP{+ -}
	LSTMAC{+ -}
	LSTOUT{+ -}
	LSTPAG{+ -}
	LSTREP{+ -}

$$\text{LSTXRF}\{+|- \}$$

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

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	1 ページあたりの行数を設定します。

表 20: リスト制御ディレクティブ

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）。

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

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。

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び出しを、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
                              if        debug
                              bl        print
                              endif

begin2                         lstcnd+
                              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

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。

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

```

以下のテキストが表示されます。

```

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

```

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。

C 形式のプリプロセッサディレクティブ

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

パラメータ	<code>condition</code>	絶対アセンブラ式は 22 ページの式、オペランド、演算子を参照してください。 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。C プリプロセッサ演算子 <code>defined</code> が使用されます。
	<code>filename</code>	インクルードされるか参照されるファイルの名前。
	<code>line-no</code>	ソース行番号。
	<code>message</code>	表示されるテキスト。

<code>symbol</code>	定義、定義取り消し、またはテストされるプリプロセッサシンボル。
<code>text</code>	割り当てられる値。

説明

アセンブラには、C89 標準にと似たような 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
<code>#define</code>	プリプロセッサシンボルに値を割り当てます。
<code>#elif</code>	<code>#if...#endif</code> ブロックに新しい条件を導入します。
<code>#else</code>	条件が偽の場合に命令をアセンブルします。
<code>#endif</code>	<code>#if</code> 、 <code>#ifdef</code> 、または <code>#ifndef</code> ブロックを終了させます。
<code>#error</code>	エラーを生成します。
<code>#if</code>	条件が真の場合に命令をアセンブルします。
<code>#ifdef</code>	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。
<code>#ifndef</code>	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。
<code>#include</code>	ファイルをインクルードします。
<code>#message</code>	標準出力上にメッセージを生成します。
<code>#pragma</code>	このディレクティブは、認識はされますが、無視されます。
<code>#undef</code>	プリプロセッサシンボルの定義を取り消します。

表 21: C 形式のプリプロセッサディレクティブ

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。

```
redefine      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` および `#if...#else...#endif` のブロックは、どのレベルにもネストすることができます。

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。

以下の例は、`tweak` と `adjust` というラベルを定義します。`adjust` が定義されている場合、レジスタ 16 は `adjust` に応じた数値だけデクリメントされず（`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

`#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一順序で検索され、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せん。54 ページの `-g` を参照してください。

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 the 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
```

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。

ソース行番号の変更

デバッグ情報で使用す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。

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      ; これは配置を間違えたコメントです。
```

```
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      ; This comment is not OK.
#define six 6       // This comment is OK.
#define seven 7     /* This comment is OK. */

module misplacedComment2
section MYCONST:CONST(2)

        DC32      five, 11, 12
; The previous line expands to:
;        "DC32      5      ; This comment is not OK., 11, 12"

        DC32      six + seven, 11, 12
; The previous line expands to:
;        "DC32      6 + 7, 11, 12"

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

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

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。

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

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

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、31 ページの式の制限を参照してください。

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 Arm Limited ディレクティブを示します。

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		32 ビットの浮動小数点定数を生成します。
DF64		64 ビットの浮動小数点定数を生成します。
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。

表 22: 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
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 22: 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)

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

構文

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

パラメータ

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

説明

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、31 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。	

表 23: アセンブラ制御ディレクティブ

ディレクティブ	説明	式の制限
//	C++ スタイル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ベースを設定します。	前方参照禁止 外部参照禁止 絶対 固定

表 23: アセンブラ制御ディレクティブ (続き)

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。124 ページの *アセンブラ制御*
章のソースファイルのインクルード項を参照してください。

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ベースを設定するには、RADIX を使用します。デフォル
トのベースは 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、141 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。

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                    ; Stored as "LABEL".
            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

                                caseoff
label      nop                    ; Stored as "LABEL".
LABEL     nop                    ; Error, "LABEL" already defined.
                                end

```

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。

```

; 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

```

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)

                                ; Standard macro definitions.
                                $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。

```
/*
Program to read serial input.
Version 1: 19.2.11
Author: mjp
*/
```

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

ベースの変更

デフォルトベースを 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                ; With the default base set
movs     r0,#12            ; to 16, the immediate value
;...                      ; of the mov instruction is
;                          ; interpreted as 0x12.
```

; ベースを 16 から 10 にリセットし直すには、引数を
; 16 進数フォーマットで記述する必要があります。

```
radix    0x0a              ; Reset the default base to 10.
movs     r0,#12            ; Now, the immediate value of
;...                      ; the mov instruction is
;                          ; interpreted as 0x0c.
end
```

関数ディレクティブ

構文

```
CALL_GRAPH_ROOT function [,category]
```

パラメータ

<i>function</i>	関数、シンボル。
<i>category</i>	コールグラフルートカテゴリ、文字列。

説明

このプラグマディレクティブを使用して、スタック使用量解析の目的で、関数 *function* がコールグラフルートであるように指定します。また、オプションのカテゴリ、引用符で囲まれた文字列も指定できます。

必要の場合は、コンパイラはアセンブラリストファイルにこのディレクティブを生成しす。

例 `CALL_GRAPH_ROOT my_interrupt, "interrupt"`

関連項目 スタック使用量分析に必要な CFI ディレクティブに情報は、135 ページの *スタック使用量分析のコールフレーム情報を参照してください。*
スタック使用量分析を有効にして使用する方法についての情報は『*ARM 用 IAR C/C++ 開発ガイド*』を参照してください。

NAME ブロックのコールフレーム情報ディレクティブ

構文 **NAME ブロックディレクティブ**
`CFI NAMES name`
`CFI ENDNAMES name`
`CFI RESOURCE resource : bits [, resource :bits] ...`
`CFI VIRTUALRESOURCE resource : bits [, resource :bits] ...`
`CFI RESOURCEPARTS resource part, part [, part] ...`
`CFI STACKFRAME cfa resource type [, cfa resource type] ...`
`CFI BASEADDRESS cfa type [, cfa type] ...`

パラメータ

<i>bits</i>	リソースのサイズ（ビット単位）。
<i>cfa</i>	CFA (Canonical Frame Address) の名前。
<i>name</i>	ブロックの名前。
<i>namesblock</i>	以前に定義された NAME ブロックの名前。
<i>offset</i>	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。
<i>part</i>	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。
<i>resource</i>	リソースの名前。
<i>size</i>	フレームセルのサイズ（ビット単位）。
<i>type</i>	CODE、CONST、または DATA などのセグメントメモリタイプです。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	これらのディレクティブを使用して、NAME ブロックを定義します。
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 ENDNAMES	NAME ブロックを終了します。
CFI FRAMECELL	呼び出し元のフレームに参照情報を作成します。
CFI NAMES	NAME ブロックを開始します。
CFI RESOURCE	リソースを宣言します。
CFI RESOURCEPARTS	複合リソースを宣言します。
CFI STACKFRAME	スタックフレーム CFA を宣言します。
CFI VIRTUALRESOURCE	仮想リソースを宣言します。

表 24: コールフレーム情報のディレクティブ

例	42 ページの CFI ディレクティブの使用例。
関連項目	33 ページの このセクションのコールフレームの使用の追跡では。

COMMON ブロックのコールフレーム情報ディレクティブ

構文	COMMON ブロックディレクティブ
	CFI COMMON <i>name</i> USING <i>namesblock</i>
	CFI ENDCOMMON <i>name</i>
	CFI CODEALIGN <i>codealignfactor</i>
	CFI DATAALIGN <i>dataalignfactor</i>
	CFI DEFAULT { UNDEFINED SAMEVALUE }
	CFI RETURNADDRESS <i>resource type</i>
パラメータ	
	<i>codealignfactor</i> すべての命令サイズで最小の共通係数。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、生成されるコールフレーム情報のサイズが小さくなります。可能な範囲 1 ~ 256 です。
	<i>commonblock</i> 以前に定義された COMMON ブロックの名前。

<code>dataalignfactor</code>	すべてのフレームサイズの最小の共通係数。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、生成されるコールフレーム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。
<code>name</code>	ブロックの名前。
<code>namesblock</code>	以前に定義された NAME ブロックの名前。
<code>resource</code>	リソースの名前。
<code>type</code>	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セグメント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

これらのディレクティブを使用して、COMMON ブロックを定義します。

ディレクティブ	説明
CFI CODEALIGN	コードアラインメントを宣言します。
CFI COMMON	COMMON ブロックを開始または拡張します。
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 ENDCOMMON	COMMON ブロックを終了します。
CFI RETURNADDRESS	リターンアドレス列を宣言します。

表 25: コールフレーム情報ディレクティブ *COMMON* ブロック

これらのディレクティブに加えて、リソースや CFA のための規則や CFI 式を指定するために、コールフレーム情報ディレクティブが必要になることがあります。132 ページのリソースや CFA を追跡するためのコールフレーム情報ディレクティブを参照してください。

例

42 ページの *CFI* ディレクティブの使用例。

関連項目

33 ページのこのセクションのコールフレームの使用の追跡では。

データブロックのコールフレーム情報ディレクティブ

構文	<code>CFI BLOCK name USING commonblock</code>
	<code>CFI ENDBLOCK name</code>

```
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] ...
```

パラメータ

<i>commonblock</i>	以前に定義された COMMON ブロックの名前。
<i>label</i>	関数ラベル。
<i>name</i>	ブロックの名前。

説明

これら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コールフレーム情報を定義できます。

ディレクティブ	説明
CFI BLOCK	データブロックを開始します。
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 ENDBLOCK	データブロックを終了します。
CFI FUNCTION	データブロックに関連する関数を宣言します。
CFI INVALID	無効なコールフレーム情報の範囲を開始します。
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。
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	コールフレーム情報の状態を記憶します。
CFI RESTORESTATE	保存されたコールフレーム情報の状態を復元します。
CFI VALID	無効なコールフレーム情報の範囲を終了します。

表 26: データブロックのコールフレーム情報ディレクティブ

これらのディレクティブに加えて、リソースや CFA のための規則や CFI 式を指定するために、コールフレーム情報ディレクティブが必要になることがあります。132 ページの *リソースや CFA を追跡するためのコールフレーム情報ディレクティブ* を参照してください。

例	42 ページの <i>CFI</i> ディレクティブの使用例。
関連項目	33 ページのこのセクションのコールフレームの使用の追跡では。

リソースや CFA を追跡するためのコールフレーム情報ディレクティブ

構文	<pre>CFI cfa { resource resource + constant resource - constant } CFI cfa cfiexpr CFI resource { UNDEFINED SAMEVALUE CONCAT } CFI resource { resource FRAME(cfa, offset) } CFI resource cfiexpr</pre>
----	---

パラメータ	
<i>cfa</i>	CFA (Canonical Frame Address) の名前。
<i>cfiexpr</i>	CFI 式は次のいずれかになります。 <ul style="list-style-type: none">● オペランド付きの CFI 演算子● 数値の定数● CFA 名● リソース名
<i>定数</i>	定数値、または定数値を計算するアセンブラ式。
<i>offset</i>	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。
<i>resource</i>	リソースの名前。

単項演算子	全体的な構文： <i>OPERATOR</i> (<i>operand</i>)
-------	--

CFI 演算子	Operand	説明
COMPLEMENT	<i>cfiexpr</i>	CFI 式のビット単位の NOT を実行します。
LITERAL	<i>expr</i>	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	<i>cfiexpr</i>	論理 CFI 式を否定します。
UMINUS	<i>cfiexpr</i>	CFI 式を算術的に論理否定します。

表 27: CFI 式の単項演算子

2 項演算子

全体的な構文：`OPERATOR(operand1, operand2)`

CFI 演算子	オペランド	説明
ADD	<i>cfiexpr</i> , <i>cfiexpr</i>	加算
AND	<i>cfiexpr</i> , <i>cfiexpr</i>	ビット単位の AND
DIV	<i>cfiexpr</i> , <i>cfiexpr</i>	除算
EQ	<i>cfiexpr</i> , <i>cfiexpr</i>	等しい
GE	<i>cfiexpr</i> , <i>cfiexpr</i>	以上
GT	<i>cfiexpr</i> , <i>cfiexpr</i>	より大きい
LE	<i>cfiexpr</i> , <i>cfiexpr</i>	以下
LSHIFT	<i>cfiexpr</i> , <i>cfiexpr</i>	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	<i>cfiexpr</i> , <i>cfiexpr</i>	より小さい
MOD	<i>cfiexpr</i> , <i>cfiexpr</i>	剰余
MUL	<i>cfiexpr</i> , <i>cfiexpr</i>	乗算
NE	<i>cfiexpr</i> , <i>cfiexpr</i>	等しくない
OR	<i>cfiexpr</i> , <i>cfiexpr</i>	ビット単位の OR
RSHIFTA	<i>cfiexpr</i> , <i>cfiexpr</i>	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	<i>cfiexpr</i> , <i>cfiexpr</i>	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	<i>cfiexpr</i> , <i>cfiexpr</i>	減算
XOR	<i>cfiexpr</i> , <i>cfiexpr</i>	ビット単位の XOR

表 28: 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	<i>cfa</i> , <i>size</i> , <i>offset</i>	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。 <i>cfa</i> 以前に宣言された CFA を指定する識別子。 <i>size</i> サイズをバイト単位で指定する定数式。 <i>offset</i> サイズをバイト単位で指定する定数式。 サイズが <i>size</i> のアドレス <i>cfa+offset</i> から値を取得します。
IF	<i>cond</i> , <i>true</i> , <i>false</i>	条件演算子。オペランドを以下に示します。 <i>cond</i> 条件を示す CFI 式。 <i>true</i> 任意の CFI 式。 <i>false</i> 任意の CFI 式。 条件式がゼロ以外である場合、結果は <i>true</i> 式の値となりますが、それ以外の場合は <i>false</i> 式の値となります。
LOAD	<i>size</i> , <i>type</i> , <i>addr</i>	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。 <i>size</i> サイズをバイト単位で指定する定数式。 <i>type</i> メモリタイプ。 <i>addr</i> メモリアドレスを示す CFI 式。 セグメントメモリタイプ <i>type</i> のアドレス <i>addr</i> における値サイズ <i>size</i> を取得します。

表 29: CFI 式の 3 項演算子

説明

これらのディレクティブを使用して、COMMON ブロックやデータブロックのリソースや CFA を追跡します。

ディレクティブ	説明
CFI <i>cfa</i>	CFA の値を宣言します。
CFI <i>resource</i>	リソースの値を宣言します。

表 30: リソースや CFA を追跡するためのコールフレーム情報ディレクティブ

例

42 ページの CFI ディレクティブの使用例。

関連項目

33 ページのこのセクションのコールフレームの使用の追跡では。

スタック使用量分析のコールフレーム情報

構文

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

パラメータ

callee 呼び出される関数のラベル。
caller 呼び出す関数のラベル。

説明

これら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コールフレーム情報を定義できます。

ディレクティブ	説明
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 INDIRECTCALL	スタック使用量解析のために間接的な呼び出しを宣言します。
CFI NOCALLS	スタック使用量解析のために呼び出しの欠如を宣言します。
CFI TAILCALL	スタック使用量解析のために末尾再帰の呼び出しを宣言します。

表 31: スタック使用量分析のコールフレーム情報

関連項目

33 ページのこのセクションのコールフレームの使用の追跡では。
スタック使用量解析の詳細は『ARM 用 IAR C/C++ 開発ガイド』。

アセンブラ擬似命令

Arm 用 IAR アセンブラは、さまざまな擬似命令を使用し、これらはすべて正しいソース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える擬似命令を示します。
- CODE16* は CODE16 ディレクティブの後で使える擬似命令を示します。
- THUMB は THUMB ディレクティブ後で使え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。

注：Thumb モード（および CODE16）では、構文 LDR レジスタ、= 式は 0 ～ 255 の値には MOVs 命令に変更されます。この命令はプログラムステータスレジスタを変更します。

使用可能な擬似命令の概要を次表に示します。

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

表 32: 擬似命令

擬似命令	ディレクティブ	変換結果	説明
LDR	THUMB	MOV、MOVS、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。
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 32: 擬似命令 (続き)

* 廃止予定。代わりに THUMB を使用します。

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、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することができます。

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、139 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	<pre> name armAdrL section MYCODE:CODE(2) arm adr1 r1,label+0x2345 ; "add r1,pc,#0x45" と ; "add r1,r1,#0x2300" になる data label dc32 0 end </pre>
---	---

ADRL (THUMB)

構文 `ADRL{condition} register,expression`

パラメータ

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

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

LDR (ARM)

構文 `LDR{condition} register,=expression1`

または

`LDR{condition} register,expression2`

パラメータ

<code>condition</code>	オプションの条件コードです。
<code>register</code>	ロードするレジスタです。
<code>expression1</code>	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。

說明

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 命令の範囲内でない場合または expression1 が未解決の場合には、アセンブラは定数をリテラルプールに入れ、その定数をリテラルプールから読み出すプログラム相対 LDR 命令を生成します。プログラマ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
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

セクション 124 ページの **アセンブラ制御**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) 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。最初の書式は MOVs 命令に変換され、これはプログラムステータスレジスタを変更します。

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。

LDR (CODE16) には、16 ビット派生型は MOVs 命令に変換され、プログラムステータスレジスタを変更します。

注: 構文 LDR{*condition*} *register*, *expression2* は、LDR (ARM) および LDR (CODE16) で説明されているように、擬似命令とはみなされません。これ

は、Advanced RISC Machines Ltd. の Unified Assembler 構文で指定されているように通常の命令の一部です。

例	<pre>name thumbLdr extern extLabel section MYCODE:CODE(2) thumb ldr r1,=extLabel ; "ldr r1,[pc,#8]" になる : nop ; リテラルプールから extLabel を ; ロード 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</pre>
---	--

関連項目	16 ビット Thumb 命令のみが使用可能な場合、139 ページの <i>ADR (CODE16)</i> 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	<p>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。</p> <p>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。</p> <p>MOV 擬似命令は即値ゼロで ADD 即値命令を使用します。</p> <p>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。</p>
----	---

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	<code>MOV32{condition} register, expression</code>
パラメータ	<p><i>condition</i> この命令が IT 命令の後にある場合は、オプションの条件コードです。</p> <p><i>register</i> ロードするレジスタです。</p> <p><i>expression</i> 任意の 32 ビット式です。</p>
説明	<p>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。</p> <p>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。</p>

NOP (ARM)

構文	<code>NOP</code>
説明	<p>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。</p> <pre>MOV r0, r0</pre> <p>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、Armv8-M) では擬似命令ではありません。</p>

NOP (CODE16)

構文	<code>NOP</code>
説明	<p>NOP 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。</p> <pre>MOV r8, r8</pre> <p>注: NOP は NOP 命令を含むアーキテクチャのバージョン (Armv6T2、Armv7、Armv8-M) では擬似命令ではありません。</p>

アセンブラの診断

次のページでは、診断メッセージの書式および診断メッセージが異なる重要度に分けられる方法について説明しています。

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

診断のオプションには、以下の 2 種類があります。たとえば、以下のようになります。

- 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 (64 ページの `-w` を参照してください)。
- アセンブリを停止する最大エラー数を設定します (51 ページの `-E` を参照してください)。

アセンブラの警告メッセージ

アセンブラの警告メッセージは、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。

アセンブラのエラーメッセージ

アセンブラ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。

アセンブラの致命的なエラーメッセージ

アセンブラ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。この情報には、主に以下のものが含まれます。

- 製品名
- アセンブラのバージョン番号（アセンブラ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
- 内部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
- 内部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

ほかのベンダーのアセンブラ用に書かれたアセンブラソースコードは、Arm 用 IAR アセンブラにも使用できます。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。

この章にはほかの製品から 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。

概要

Arm (IASMARM) 用 IAR アセンブラは、その他の IAR アセンブラと同じ見た目を使用して設計されているので、Arm Limited の ARMASM アセンブラ用に書かれたソースコードを変換しやすくします。

オプション `-j` (代替のレジスタ名、ニーモニック、およびオペランドを許可) が選択されていると、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 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 を同じ解釈にするには、:OR: を使用し、Thumb モードビットか :AND: を設定します。クリアにするには:

```
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、Arm 用 IAR アセンブラは、ほか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 33: 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 33: 代替レジスタ名一覧 (続き)

レジスタの詳細は 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 34: 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 34: 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 35: 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 35: 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。演算子の詳細説明については 67 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]
```

あ

アセンブラオブジェクトファイル、ファイル名指定	59
アセンブラオプション	
アセンブラへの受渡し	20
コマンドライン拡張ファイル、設定	46
コマンドライン、設定	45
パラメータの指定	45
概要	46
アセンブラシンボル	24
インポート	92
エクスポート	92
再配置可能式内	30
定義済	26
定義の解除	64
アセンブラソースファイル、インクルード	119, 126
アセンブラソースフォーマット	21
アセンブラディレクティブ	
アセンブラ制御	124
シンボル制御	91
スタック使用量分析の CFI	135
セグメント制御	95
データブロックの CFI ディレクティブ	130
データ定義または割当て	121
マクロ処理	102
モジュール制御	88
リストファイル制御	111
リソースや CFA を追跡するための CFI ディレクティブ	132
概要	83
条件付きアセンブリ	101
参照 C 形式プリプロセッサディレクティブ	
値割当て	99
C 形式のプリプロセッサ	116
COMMON ブロックの CFI ディレクティブ	129
function	127
NAME ブロックの CFI ディレクティブ	128
アセンブラのエラーメッセージ	148

アセンブラの環境変数	20
アセンブラの警告メッセージ	147
アセンブラの出力、デバッグ情報を含める	61
アセンブラマクロ	
ルーチンのインラインコーディング	109
引数、受け渡し	106
引用符、指定	57
生成された行、リストファイルでの制御	114
定義	104
定義済シンボル	106
特殊文字、使用	105
アセンブララベル	25
フォーマット	21
Thumb コードの	149
アセンブラリストファイル	
アドレスフィールド	32
#include ファイル、指定 (-i)	55
クロスリファレンス	
生成 (LSTXRF)	115
生成 (-x)	65
コメント	125
シンボルとクロスリファレンスの表	32
タブによる移動量、指定	63
ディレクティブ (フォーマット)	115
データフィールド	32
ファイル名、指定 (-l)	56
ヘッダセクション、無効 (-N)	57
ページあたりの行数、指定 (-p)	60
マクロで生成された行、制御	114
マクロの実行情報、含む (-B)	48
条件付きコードと文字列	113
生成 (-L)	55
生成された行、制御 (LSTREP)	115
有効化と無効化 (LSTOUT)	112
アセンブラ演算子	67
式内	22
優先順位	67
アセンブラ擬似命令	137
アセンブラ呼び出し構文	19

アセンブラ式	22
アセンブラ診断	147
アセンブラ制御ディレクティブ	124
アセンブラ命令	22
BX	94
アセンブラ、呼び出し構文	19
アセンブリ時のメッセージのフォーマット	147
アセンブリ時のワーニングメッセージ 無効	64
アドレスフィールド、アセンブラリストファイル	32
アドレス、レジスタへロード	138-141

い

#include ファイル	55
#include ファイル、指定	54
#include (アセンブラディレクティブ)	117
インクルードパス、指定	54
インクルードファイル、検索の無効化	54
インストール先ディレクトリ	14
インラインコーディング、マクロの使用	109

う

エラーメッセージ	
最大数、指定	51
format	147
#error、表示のために使用	120

お

オプションの概要	46
----------------	----

く

クロスリファレンス、アセンブラリストファイル内	
生成 (LSTXRF)	115
生成 (-x)	65
グローバル値、定義	100

こ

このガイドで使用されている規則	14
コマンドプロンプト アイコン、本ガイド	15
コマンドラインエラーのメッセージ、アセンブラ	148
コマンドラインオプション	45
呼び出し構文のパート	19
受渡し	20
表記規則	15
コマンドラインオプション、拡張	52
コメント	
アセンブラソースコード	21
アセンブラリストファイル	125
複数行、アセンブラディレクティブで使用	127
C 形式のプリプロセッサディレクティブでの	121
コンピュータスタイル、表記規則	15
コールフレーム情報ディレクティブ	128-130, 132, 135

し

システムインクルードファイル、検索の無効化	54
シンボル	
ユーザ定義、大文字 / 小文字を区別する	61
他のモジュールへのエクスポート	92
定義済、アセンブラマクロ	106
定義済、アセンブラ内	26
アセンブラシンボルも参照	
シンボルとクロスリファレンスの表、アセンブラリス トファイル内	32
クロスリファレンスのインクルードも参照	
シンボル制御ディレクティブ	91

す

スタック使用量分析の CFI	135
スタック使用量分析、CFI ディレクティブ	135

せ

セグメントサイズ (アセンブラ演算子).....	81
セグメント開始 (アセンブラ演算子).....	79
セグメント終了 (アセンブラ演算子).....	80
セグメント制御ディレクティブ.....	95

そ

ソースファイル	
インクルード.....	119
インクルードの例.....	126
ソースフォーマット、アセンブラ.....	21
ソース行番号、変更.....	120

た

タブによる移動量、アセンブラリストファイルに 指定.....	63
ターゲットコア、指定。→プロセッサの設定を参照	

っ

ツールアイコン、本ガイド.....	15
-------------------	----

て

ディレクティブ。アセンブラディレクティブを参照	
デバッグ情報、アセンブラ出力に含める.....	61
デフォルトベース、定数用.....	125
データフィールド、アセンブラリストファイル.....	32
データブロックの CFI ディレクティブ.....	130
データブロック、CFI ディレクティブ.....	130
データブロック (コールフレーム情報).....	35
データ割当てディレクティブ.....	121
データ定義ディレクティブ.....	121
データ、Thumb コードセクション中で定義.....	94

の

ノーオペレーションコード、生成.....	145
----------------------	-----

は

バイトオーダー.....	29
パラメータ	
指定.....	45
表記規則.....	15
バージョン	
本ガイド.....	2

ひ

ビット単位の排他 OR (アセンブラ演算子).....	75
ビット単位の AND (アセンブラ演算子).....	74
ビット単位の NOT (アセンブラ演算子).....	75
ビット単位の OR (アセンブラ演算子).....	75

ふ

ファイルタイプ	
コマンドライン拡張.....	46, 52
#include、パスを指定.....	54
ファイル拡張子。ファイル名拡張子を参照	
ファイル名拡張子	
xcl.....	46, 52
ファイル名、アセンブラオブジェクトファイルの 指定.....	59-60
フォーマット	
アセンブラソースコード.....	21
リストファイル.....	32
診断メッセージ.....	147
#pragma (アセンブラディレクティブ).....	117
プリプロセッサシンボル	
コマンドラインで定義.....	50
定義と定義取り消し.....	118
プログラミングのヒント.....	33

プログラミング経験、必須.....	13
プログラムモジュール、開始.....	89
プログラムロケーションカウンタ (PLC)	25

へ

ペア、レジスタの	26
ヘッダセクション、アセンブラリストファイルで 無効	57
ヘッダファイル、SFR	33
ページあたりの行数、アセンブラリストファイルの ..	60

ま

マクロの引用符	105
指定	57
マクロの実行情報、リストファイルに含める	48
マクロ処理ディレクティブ	102
マクロ。アセンブラマクロも参照	

め

メッセージ、標準出力ストリームから除外	61
メモリ、空間の予約	122

も

モジュールの互換性	90
モジュール制御ディレクティブ	88
モジュール、開始	89

ゆ

ユーザシンボルの大文字 / 小文字を区別する	61
ユーザシンボル、大文字 / 小文字を区別する	61

ら

ラベル。アセンブララベルを参照	
ランタイムモデル属性、宣言	90

り

リストファイル	
#include ファイル、指定 (-i)	55
クロスリファレンス、生成 (-x)	65
ファイル名、指定 (-I)	56
ヘッダセクション、無効 (-N)	57
制御ディレクティブ	111
生成 (-L)	55
内容の制御 (-c)	49
リストファイルのフォーマット	32
シンボルとクロスリファレンス	
ヘッダ	32
ボディ	32
CRC	32
リソースや CFA を追跡するための	
CFI ディレクティブ	132
リソース、追跡用 CFI ディレクティブ	132
リテラルプール	58, 142
リンカオプション	
表記規則	15

れ

レジスタ	25–26
代替名称	150

ろ

ローカル値、定義	100
----------------	-----

わ

ワーニング

無効.....	64
ワーニングアイコン、本ガイド.....	15

A

ADD (CFI 演算子).....	133
ADR (ARM) (擬似命令).....	138
ADR (CODE16) (擬似命令).....	139
ADR (THUMB) (擬似命令).....	139
ADRL (ARM) (擬似命令).....	140
ADRL (THUMB) (擬似命令).....	141
ALIAS (アセンブラディレクティブ).....	99
ALIGNRAM (アセンブラディレクティブ).....	97
ALIGNROM (アセンブラディレクティブ).....	97
ALIGN (アセンブラディレクティブ).....	96
AND (CFI 演算子).....	133
_args (定義済マクロシンボル).....	106
_args (アセンブラディレクティブ).....	103
--arm (アセンブラオプション).....	48
Arm 用 IAR アセンブラへの移行.....	149
演算子の同義語.....	152
警告メッセージ.....	153
代替ニーモニック.....	151
代替レジスタ名.....	150
ARMASM アセンブラ.....	149
_ARMVFP__ (定義済シンボル).....	28
_ARMVFP_D16__ (定義済シンボル).....	28
_ARMVFP_FP16__ (定義済シンボル).....	28
_ARMVFP_SP__ (定義済シンボル).....	28
_ARM_ADVANCED_SIMD__ (定義済シンボル).....	26
_ARM_ARCH (定義済シンボル).....	26
_ARM_ARCH_ISA_ARM (定義済シンボル).....	26
_ARM_ARCH_ISA_THUMB (定義済シンボル).....	26
_ARM_ARCH_PROFILE (定義済シンボル).....	26
_ARM_BIG_ENDIAN (定義済シンボル).....	26
_ARM_FEATURE_CMSE (定義済シンボル).....	26

_ARM_FEATURE_CRC32 (定義済シンボル).....	27
_ARM_FEATURE_CRYPTO (定義済シンボル).....	27
_ARM_FEATURE_DIRECTED_ROUNDING (定義済シンボル).....	27
_ARM_FEATURE_DSP (定義済シンボル).....	27
_ARM_FEATURE_FMA (定義済シンボル).....	27
_ARM_FEATURE_IDIV (定義済シンボル).....	27
_ARM_FEATURE_NUMERIC_MAXMIN (定義済シンボル).....	27
_ARM_FP (定義済シンボル).....	27
_ARM_MEDIA__ (定義済シンボル).....	27
_ARM_MPCORE__ (定義済シンボル).....	27
_ARM_NEON (定義済みのシンボル).....	27
_ARM_NEON_FP (定義済みのシンボル).....	28
_ARM_PROFILE_M__ (定義済シンボル).....	28
ASCII 文字定数.....	23
ASSIGN (アセンブラディレクティブ).....	100

B

-B (アセンブラオプション).....	48
_BUILD_NUMBER__ (定義済シンボル).....	28
BX (アセンブラ命令).....	94
BYTE1 (アセンブラ演算子).....	77
BYTE2 (アセンブラ演算子).....	77
BYTE3 (アセンブラ演算子).....	77
BYTE4 (アセンブラ演算子).....	78

C

-c (アセンブラオプション).....	49
C 形式のプリプロセッサディレクティブ.....	116
CALL_GRAPH_ROOT (アセンブラディレ クティブ).....	127
CASEOFF (アセンブラディレクティブ).....	125
CASEON (アセンブラディレクティブ).....	125
CFI BASEADDRESS (アセンブラディレ クティブ).....	129
CFI BLOCK (アセンブラディレクティブ).....	131

CFI cfa (アセンブラディレクティブ)	134
CFI CODEALIGN (アセンブラディレクティブ)	130
CFI COMMON (アセンブラディレクティブ)	130
CFI CONDITIONAL (アセンブラディレクティブ)	131
CFI DATAALIGN (アセンブラディレクティブ)	130
CFI DEFAULT (アセンブラディレクティブ)	130
CFI ENDBLOCK (アセンブラディレクティブ)	131
CFI ENDCOMMON (アセンブラディレクティブ)	130
CFI ENDNAMES (アセンブラディレクティブ)	129
CFI FRAMECELL (アセンブラディレクティブ)	129
CFI FUNCALL (アセンブラディレクティブ)	135
CFI FUNCTION (アセンブラディレクティブ)	131
CFI INDIRECTCALL (アセンブラディレクティブ)	135
CFI INVALID (アセンブラディレクティブ)	131
CFI NAMES (アセンブラディレクティブ)	129
CFI NOCALLS (アセンブラディレクティブ)	135
CFI NOFUNCTION (アセンブラディレクティブ)	131
CFI PICKER (アセンブラディレクティブ)	131
CFI REMEMBERSTATE (アセンブラディレクティブ)	131
CFI RESOURCE (アセンブラディレクティブ)	129
CFI resource (アセンブラディレクティブ)	134
CFI RESOURCEPARTS (アセンブラディレクティブ)	129
CFI RESTORESTATE (アセンブラディレクティブ)	131
CFI RETURNADDRESS (アセンブラディレクティブ)	130
CFI STACKFRAME (アセンブラディレクティブ)	129
CFI TAILCALL (アセンブラディレクティブ)	135
CFI VALID (アセンブラディレクティブ)	131
CFI VIRTUALRESOURCE (アセンブラディレクティブ)	129
CFI 式	40
--cmse (アセンブラオプション)	49
CODE (アセンブラディレクティブ)	94
CODE16 (アセンブラディレクティブ)	94
CODE32 (アセンブラディレクティブ)	93

COL (アセンブラディレクティブ)	112
COMMON ブロック (コールフレーム情報)	35
COMMON ブロックの定義	37
COMMON ブロックの CFI ディレクティブ	129
COMMON ブロック、定義	37
COMMON ブロック、CFI ディレクティブ	129
COMPLEMENT (CFI 演算子)	132
__CORE__ (定義済シンボル)	28
--cpu (アセンブラオプション)	50
--cpu_mode (アセンブラオプション)	50
CPU、アセンブラでの定義。→プロセッサの設定を参照	
CRC、アセンブラリストファイル内	32
C++ 用語	14

D

-D (アセンブラオプション)	50
DATA (アセンブラディレクティブ)	94
__DATE__ (定義済シンボル)	28
DATE (アセンブラ演算子)	78
DCB (アセンブラディレクティブ)	122
DCD (アセンブラディレクティブ)	122
DCW (アセンブラディレクティブ)	122
DC8 (アセンブラディレクティブ)	122
DC16 (アセンブラディレクティブ)	122
DC24 (アセンブラディレクティブ)	122
DC32 (アセンブラディレクティブ)	122
#define (アセンブラディレクティブ)	117
DEFINE (アセンブラディレクティブ)	100
DF32 (アセンブラディレクティブ)	122
DF64 (アセンブラディレクティブ)	122
DIV (CFI 演算子)	133
DLIB	
命名規約	16
DS (アセンブラディレクティブ)	122
DS8 (アセンブラディレクティブ)	122
DS16 (アセンブラディレクティブ)	122

DS24 (アセンブラディレクティブ)	123
DS32 (アセンブラディレクティブ)	123

E

-E (アセンブラオプション)	51
-e (アセンブラオプション)	52
#elif (アセンブラディレクティブ)	117
#else (アセンブラディレクティブ)	117
--endian (アセンブラオプション)	52
#endif (アセンブラディレクティブ)	117
ENDM (アセンブラディレクティブ)	103
ENDR (アセンブラディレクティブ)	103
END (アセンブラディレクティブ)	89
EQU (アセンブラディレクティブ)	99
EQ (CFI 演算子)	133
#error (アセンブラディレクティブ)	117
EVEN (アセンブラディレクティブ)	97
EXITM (アセンブラディレクティブ)	103
EXTERN (アセンブラディレクティブ)	91
EXTWEAK (アセンブラディレクティブ)	91

F

-f (アセンブラオプション)	46, 52
false 値、アセンブラ式内	24
__FILE__ (定義済シンボル)	28
--fpu (アセンブラオプション)	53
FRAME (CFI 演算子)	134

G

-G (アセンブラオプション)	53
-g (アセンブラオプション)	54
GE (CFI 演算子)	133
GT (CFI 演算子)	133

H

HIGH (アセンブラ演算子)	78
HWRD (アセンブラ演算子)	79

I

-I (アセンブラオプション)	54
__IAR_SYSTEMS_ASM__ (定義済シンボル)	29
__IASMARM__ (定義済シンボル)	29
#if (アセンブラディレクティブ)	117
#ifdef (アセンブラディレクティブ)	117
#ifndef (アセンブラディレクティブ)	117
IF (CFI 演算子)	134
IMPORT (アセンブラディレクティブ)	91
INCLUDE (アセンブラディレクティブ)	125

J

-j (アセンブラオプション)	55
-----------------------	----

L

-L (アセンブラオプション)	55
-l (アセンブラオプション)	56
LDR (ARM) (擬似命令)	141
LDR (CODE16) (擬似命令)	142
LDR (THUMB) (擬似命令)	143
--legacy (アセンブラオプション)	56
LE (CFI 演算子)	133
LIBRARY (アセンブラディレクティブ)	86
lightbulb アイコン、本ガイドの	15
__LINE__ (定義済シンボル)	29
LITERAL (CFI 演算子)	132
__LITTLE_ENDIAN__ (定義済シンボル)	29
LOAD (CFI 演算子)	134
LOCAL (アセンブラディレクティブ)	103
:LOR: (アセンブラ演算子)	76
LOW (アセンブラ演算子)	79

LSHIFT (CFI 演算子)	133
LSTCND (アセンブラディレクティブ)	112
LSTCOD (アセンブラディレクティブ)	112
LSTEXP (アセンブラディレクティブ)	112
LSTMAC (アセンブラディレクティブ)	112
LSTOUT (アセンブラディレクティブ)	112
LSTPAG (アセンブラディレクティブ)	112
LSTREP (アセンブラディレクティブ)	112
LSTXRF (アセンブラディレクティブ)	112
LTORG (アセンブラディレクティブ)	125
LT (CFI 演算子)	133
LWRD (アセンブラ演算子)	79

M

-M (アセンブラオプション)	57
MACRO (アセンブラディレクティブ)	103
#message (アセンブラディレクティブ)	117
MOD (CFI 演算子)	133
MOV (CODE16) (擬似命令)	144
MOV (THUMB) (擬似命令)	145
MUL (CFI 演算子)	133

N

-N (アセンブラオプション)	57
NAME ブロックの定義	36
NAME ブロックの CFI ディレクティブ	128
NAME ブロック、定義	36
NAME ブロック、CFI ディレクティブ	128
NAME ブロック (コールフレーム情報)	35
NAME (アセンブラディレクティブ)	89
NE (CFI 演算子)	133
NOP (ARM) (擬似命令)	145
NOP (CODE16) (擬似命令)	145
:NOT: (アセンブラ演算子)	75
NOT (CFI 演算子)	132
--no_dwarf3_cfi (アセンブラオプション)	58
--no_it_verification (アセンブラオプション)	58

--no_literal_pool (アセンブラオプション)	58
--no_path_in_file_macros (アセンブラオプション)	59

O

-O (アセンブラオプション)	59
-o (アセンブラオプション)	60
ODD (アセンブラディレクティブ)	97
operands	
アセンブラ式内	22
フォーマット	21
:OR: (アセンブラ演算子)	75
OR (CFI 演算子)	133
OVERLAY (アセンブラディレクティブ)	91

P

-p (アセンブラオプション)	60
PAGE (アセンブラディレクティブ)	112
PAGSIZ (アセンブラディレクティブ)	112
PROGRAM (アセンブラディレクティブ)	89
PUBLIC (アセンブラディレクティブ)	91
PUBWEAK (アセンブラディレクティブ)	92

R

-r (アセンブラオプション)	61
RADIX (アセンブラディレクティブ)	125
REPTC (アセンブラディレクティブ)	103
REPTI (アセンブラディレクティブ)	103
REPT (アセンブラディレクティブ)	103
REQUIRE (アセンブラディレクティブ)	92
RSEG (アセンブラディレクティブ)	97
RSHIFTA (CFI 演算子)	133
RSHIFTL (CFI 演算子)	133
RTMODEL (アセンブラディレクティブ)	89

S

-S (アセンブラオプション)	61
-s (アセンブラオプション)	61
SECTION (アセンブラディレクティブ)	97
sections	
アラインメント	98
開始	97
SECTION_TYPE (アセンブラディレクティブ)	97
SETA (アセンブラディレクティブ)	100
SET (アセンブラディレクティブ)	100
SFB (アセンブラ演算子)	79
SFE (アセンブラ演算子)	80
SFR. 特殊機能レジスタを参照	
SIZEOF (アセンブラ演算子)	81
--source_encoding (アセンブラオプション)	62
STACK (アセンブラディレクティブ)	97
SUB (CFI 演算子)	133
--suppress_vfe_header (アセンブラオプション)	62
--system_include_dir (アセンブラオプション)	63

T

-t (アセンブラオプション)	63
--thumb (アセンブラオプション)	63
THUMB (アセンブラディレクティブ)	94
__TID__ (定義済シンボル)	29
__TIME__ (定義済シンボル)	29
true 値、アセンブラ式内	24

U

-U (アセンブラオプション)	64
UGT (アセンブラ演算子)	81
ULT (アセンブラ演算子)	82
UMINUS (CFI 演算子)	132
#undef (アセンブラディレクティブ)	117

V

VAR (アセンブラディレクティブ)	100
__VER__ (定義済シンボル)	29

W

-w (アセンブラオプション)	64
-----------------------	----

X

-x (アセンブラオプション)	65
xcl (ファイル名拡張子)	46, 52
XOR (アセンブラ演算子)	82
XOR (CFI 演算子)	133

記号

__args (定義済マクロシンボル)	106
__args (アセンブラディレクティブ)	103
__ARMVFP_D16__ (定義済シンボル)	28
__ARMVFP_FP16__ (定義済シンボル)	28
__ARMVFP_SP__ (定義済シンボル)	28
__ARMVFP__ (定義済シンボル)	28
__ARM_ADVANCED_SIMD__ (定義済シンボル)	26
__ARM_ARCH (定義済シンボル)	26
__ARM_ARCH_ISA_ARM (定義済シンボル)	26
__ARM_ARCH_ISA_THUMB (定義済シンボル)	26
__ARM_ARCH_PROFILE (定義済シンボル)	26
__ARM_BIG_ENDIAN (定義済シンボル)	26
__ARM_FEATURE_CMSE (定義済シンボル)	26
__ARM_FEATURE_CRC32 (定義済シンボル)	27
__ARM_FEATURE_CRYPT (定義済シンボル)	27
__ARM_FEATURE_DIRECTED_ROUNDING (定義済シンボル)	27
__ARM_FEATURE_DSP (定義済シンボル)	27
__ARM_FEATURE_FMA (定義済シンボル)	27
__ARM_FEATURE_IDIV (定義済シンボル)	27

__ARM_FEATURE_NUMERIC_MAXMIN (定義済みシンボル)	27
__ARM_FP (定義済みシンボル)	27
__ARM_MEDIA__ (定義済みシンボル)	27
__ARM_MPCORE__ (定義済みシンボル)	27
__ARM_NEON (定義済みのシンボル)	27
__ARM_NEON_FP (定義済みのシンボル)	28
__ARM_PROFILE_M__ (定義済みシンボル)	28
__BUILD_NUMBER__ (定義済みシンボル)	28
__CORE__ (定義済みシンボル)	28
__DATE__ (定義済みシンボル)	28
__FILE__ (定義済みシンボル)	28
__IAR_SYSTEMS_ASM__ (定義済みシンボル)	29
__IASMARM__ (定義済みシンボル)	29
__LINE__ (定義済みシンボル)	29
__LITTLE_ENDIAN__ (定義済みシンボル)	29
__TID__ (定義済みシンボル)	29
__TIME__ (定義済みシンボル)	29
__VER__ (定義済みシンボル)	29
-B (アセンブラオプション)	48
-c (アセンブラオプション)	49
-D (アセンブラオプション)	50
-E (アセンブラオプション)	51
-e (アセンブラオプション)	52
-f (アセンブラオプション)	46, 52
-G (アセンブラオプション)	53
-g (アセンブラオプション)	54
-i (アセンブラオプション)	55
-I (アセンブラオプション)	54
-j (アセンブラオプション)	55, 149
-L (アセンブラオプション)	55
-l (アセンブラオプション)	56
-M (アセンブラオプション)	57
-N (アセンブラオプション)	57
-O (アセンブラオプション)	59
-o (アセンブラオプション)	60
-p (アセンブラオプション)	60
-r (アセンブラオプション)	61
-S (アセンブラオプション)	61
-s (アセンブラオプション)	61

-t (アセンブラオプション)	63
-U (アセンブラオプション)	64
-w (アセンブラオプション)	64
-x (アセンブラオプション)	65
--arm (アセンブラオプション)	48
--cmse (アセンブラオプション)	49
--cpu (アセンブラオプション)	50
--cpu_mode (アセンブラオプション)	50
--endian (アセンブラオプション)	52
--fpu (アセンブラオプション)	53
--legacy (アセンブラオプション)	56
--no_dwarf3_cfi (アセンブラオプション)	58
--no_it_verification (アセンブラオプション)	58
--no_literal_pool (アセンブラオプション)	58
--no_path_in_file_macros (アセンブラオプション)	59
--source_encoding (アセンブラオプション)	62
--suppress_vfe_header (アセンブラオプション)	62
--system_include_dir (アセンブラオプション)	63
--thumb (アセンブラオプション)	63
- (アセンブラ演算子)	71–72
:AND: (アセンブラ演算子)	74
:EOR: (アセンブラ演算子)	75
:LAND: (アセンブラ演算子)	74
:LEOR: (アセンブラ演算子)	82
:LNOT: (アセンブラ演算子)	76
:LOR: (アセンブラ演算子)	76
:MOD: (アセンブラ演算子)	75
:NOT: (アセンブラ演算子)	75
:OR: (アセンブラ演算子)	75
:SHL: (アセンブラ演算子)	76
:SHR: (アセンブラ演算子)	77
! (アセンブラ演算子)	76
!= (アセンブラ演算子)	73
() (アセンブラ演算子)	70
* (アセンブラ演算子)	71
/ (アセンブラ演算子)	72
/*...*/ (アセンブラディレクティブ)	124
// (アセンブラディレクティブ)	125
& (アセンブラ演算子)	74

&& (アセンブラ演算子)	74
#define (アセンブラディレクティブ)	117
#elif (アセンブラディレクティブ)	117
#else (アセンブラディレクティブ)	117
#endif (アセンブラディレクティブ)	117
#error (アセンブラディレクティブ)	117
#ifdef (アセンブラディレクティブ)	117
#ifndef (アセンブラディレクティブ)	117
#if (アセンブラディレクティブ)	117
#include ファイル	55
#include ファイル、指定	54
#include (アセンブラディレクティブ)	117
#message (アセンブラディレクティブ)	117
#pragma (アセンブラディレクティブ)	117
#undef (アセンブラディレクティブ)	117
^ (アセンブラ演算子)	75
+ (アセンブラ演算子)	71
< (アセンブラ演算子)	72
<< (アセンブラ演算子)	76
<= (アセンブラ演算子)	73
<> (アセンブラ演算子)	73
= (アセンブラディレクティブ)	99
= (アセンブラ演算子)	73
== (アセンブラ演算子)	73
> (アセンブラ演算子)	73
>= (アセンブラ演算子)	74
>> (アセンブラ演算子)	77
(アセンブラ演算子)	75
(アセンブラ演算子)	76
~ (アセンブラ演算子)	75
\$ (アセンブラディレクティブ)	124
\$ (プログラムロケーションカウンタ)	25

32 ビット式、レジスタへのロード	141
4 バイト目 (アセンブラ演算子)	78

数字

1 バイト目 (アセンブラ演算子)	77
2 バイト目 (アセンブラ演算子)	77
3 バイト目 (アセンブラ演算子)	77

