IAR Embedded Workbench®

IAR C/C++ 開発ガイド
コンパイルおよびリンク

Advanced RISC Machines Ltd

ARM® コア

DARM-14-J

IAR SYSTEMS

改版情報
第 14 版 : 2015 年 2 月

部品番号 : DARM-14-J

本ガイドは、ARM 用 IAR Embedded Workbench® のバージョン 7.40.x に適用する。

内部参照 : M18、csrct2010.1、V_110411、IMAE。

目次（章）

目次

表

はじめに

*ARM 用 IAR C/C++ 開発ガイド*へようこそ。このガイドは、開発中のアプリケーション要件に対し、最適な方法でビルドツールをご利用いただくのに役立つ、詳細なリファレンス情報を提供します。また、アプリケーションを効率的に開発するための推奨コーディングテクニックも説明しています。

本ガイドの対象者

本ガイドは、C/C++ 言語を使用して ARM コア用アプリケーションを開発する予定があり、ビルドツールの使用方法に関する詳細情報を必要とするユーザ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。

必要な知識

IAR Embedded Workbench のツールを使用するには、以下について十分な知識が必要です。

- ARM コアのアーキテクチャと命令セット（チップメーカーのドキュメントを参照してください）
- C/C++ プログラミング言語
- 組込みシステム用アプリケーションの開発
- ホストコンピュータのオペレーティングシステム

IDE に組み込まれている他の開発ツールについて詳しくは、「32 ページの*その他のドキュメント*」を参照してください。

本ガイドの使用方法

ARM 用 IAR C/C++ コンパイラコンパイラおよびリンカを使用する際には、本ガイドの「*パート 1. ビルドツールの使用*」を参照してください。

コンパイラとリンカの使用方法を確認し、プロジェクトの設定が完了したら、「*パート 2. リファレンス情報*」に進んでください。

本製品を初めて使用する場合、*『IAR Embedded Workbench® の使用開始の手順』*で IDE に備わっているツールと機能の概要に目を通すことをお勧めします。

IAR インフォメーションセンタのチュートリアルは、IAR Embedded Workbench を初めて使用する際に役に立ちます。

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。

パート 1. ビルドツールの使用

- 「*IAR ビルドツールの概要*」では、ツール、プログラミング言語、利用可能なデバイスサポート、ARM コアの特定の機能をサポートするために提供されている拡張機能など、IAR ビルドツールの概要について説明します。
- 「*組込みアプリケーションの開発*」では、IAR ビルドツールを使用する組込みソフトウェアの開発に必要な基礎について説明します。
- 「*データ記憶*」では、メモリへのデータの保存方法について説明します。
- 「*関数*」に関連した拡張（関数を制御するための仕組み）の概要を説明した後、これらの仕組みのいくつかを取り上げて詳しく説明します。
- 「*ILINK を使用したリンク*」では、IAR ILINK リンカを使用するリンクプロセスおよび関連する概念について説明します。
- 「*アプリケーションのリンク*」では、ILINK オプションの使用およびリンカ設定ファイルの調整など、アプリケーションをリンクするときに注意する必要がある多くの事項を示します。
- 「*DLIB ランタイム環境*」では、アプリケーションの実行環境である DLIB ランタイム環境について説明します。オプションの設定、デフォルトライブラリモジュールへのオーバライド、自作ライブラリのビルドにより、ランタイムライブラリを変更する方法を説明します。また、システムの初期化、`cstartup` ファイルの概要、ロケール用モジュールの使用方法、ファイル I/O についても説明します。
- 「*アセンブラ言語インタフェース*」では、アプリケーションの一部をアセンブラ言語で記述する場合に必要な情報を説明します。呼出し規約についても説明しています。
- 「*C の使用*」は、C 言語でサポートされている 2 つの派生型の概要と、C 規格の拡張などコンパイラ拡張の概要を説明します。
- 「*C++ の使用*」では、業界標準の EC++ と IAR 拡張 EC++ という 2 種類の C++ サポートの概要を説明します。
- 「*アプリケーションに関する考慮事項*」では、コンパイラおよびリンカの使用に関連する一部の範囲のアプリケーション問題について説明します。
- 「*組込みアプリケーション用の効率的なコーディング*」では、組込みアプリケーションに適した効率的なコーディング方法のヒントを提供します。

パート 2. リファレンス情報

- 「*外部インタフェースの詳細*」では、コンパイラおよびリンカがそれらの環境を操作する方法として、呼出し構文、コンパイラおよびリンカにオプションを渡すための手法、環境変数、インクルードファイル検索手順、さまざまな種類のコンパイラおよびリンカ出力について説明します。また、診断システムの機能についても説明します。
- 「*コンパイラオプション*」では、オプションの設定方法、オプションの要約、各コンパイラオプションの詳細なリファレンス情報について説明します。
- 「*リンカオプション*」では、オプションの要約について説明し、各リンカオプションの詳細なリファレンス情報について説明します。
- 「*データ表現*」では、使用可能なデータ型、ポインタ、構造体について説明します。また、型やオブジェクト属性についても説明します。
- 「*拡張キーワード*」では、標準 C/C++ 言語を拡張した ARM 固有のキーワードのリファレンス情報を提供します。
- 「*プラグマディレクティブ*」では、プラグマディレクティブのリファレンス情報を提供します。
- 「*組込み関数*」では、ARM 固有の低レベル機能にアクセスするための関数のリファレンス情報を提供します。
- 「*プリプロセッサ*」では、さまざまなプリプロセッサディレクティブ、シンボル、その他の関連情報など、プリプロセッサの概要を説明します。
- 「*ライブラリ関数*」では、C/C++ ライブラリ関数の概要と、ヘッダファイルの要約を説明します。
- 「*リンカ設定ファイル*」では、リンカ設定ファイルの目的およびその内容について説明します。
- 「*セクションリファレンス*」では、セクション使用に関するリファレンス情報を収録しています。
- 「*スタック使用解析制御ファイル*」は、スタック使用制御ファイルの構文と動作について説明します。
- 「*IAR ユーティリティ*」では、ELF および DWARF オブジェクトフォーマットを扱う IAR ユーティリティについて説明します。
- 「*C 規格の処理系定義の動作*」では、コンパイラが C 規格の処理系定義エリアをどのように扱うかについて説明します。
- 「*C89 の処理系定義の動作*」では、コンパイラが C89 言語標準の処理系定義エリアをどのように扱うかについて説明します。

その他のドキュメント

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

ユーザガイドおよびリファレンスガイド

IAR システムズの各開発ツールについては、一連のガイドで説明しています。以下はツールとガイドの一覧です。

- IAR システムズの製品のインストールおよび登録の要件と詳細については、同梱されているクイックレファンレンスのブックレットおよび *『インストールおよびライセンス ガイド』* を参照してください。
- IAR Embedded Workbench および同梱のツールを使用するにあたっては、*『IAR Embedded Workbench の使用開始の手順』* を参照してください。
- プロジェクト管理とビルドでの IDE の使用については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。
- IAR C-SPY® デバッガの使用については、*『ARM 用 C-SPY® デバッグガイド』* を参照してください。
- IAR ILINK リンカを使用した ARM 用 IAR C/C++ コンパイラのプログラミングとリンクについては、*『ARM 用 IAR C/C++ 開発ガイド』* を参照してください。
- ARM 用 IAR アセンブラのプログラミングについては、*『ARM 用 IAR アセンブラリファレンスガイド』* を参照してください。
- IAR DLIB ライブラリの使用については、オンラインヘルプシステムの *DLIB ライブラリリファレンス情報* を参照してください。
- C-STAT と必要なチェックを使用した静的解析の実行については、*C-STAT® Static Analysis Guide* を参照してください。
- MISRA-C ガイドラインを使用して、安全性を最重要視したアプリケーションを開発する方法については、*『IAR Embedded Workbench® MISRA-:2004 Reference Guide』* または *『IAR Embedded Workbench® MISRA-C:1998 Reference Guide』* を参照してください。
- I-jet の使用については、*『I-jet®、I-jet Trace、I-scope 用 IAR デバッグプローブガイド』* を参照してください。
- JTAGjet-Trace の使用については、*『ARM 用 JTAGjet-Trace ユーザガイド』* を参照してください。
- IAR J-Link および IAR J-Trace の使用については、*『ARM コア JTAG エミュレータ用 IAR J-Link および IAR J-Trace ユーザガイド』* を参照してください。

- ARM用IAR Embedded Workbenchの旧バージョンで開発したアプリケーションコードやプロジェクトの移植については、*『IAR Embedded Workbench® 移行ガイド』*を参照してください。

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

オンラインヘルプシステムを参照

コンテキスト依存のオンラインヘルプの内容は以下のとおりです。

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

参考資料

IAR システムズ開発ツールの使用時は、以下の資料が参考になります。

- Seal, David, and David Jagger. ARM Architecture Reference Manual. Addison-Wesley
- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. OReilly & Associates.
- Furber, Steve, ARM System-on-Chip Architecture. Addison-Wesley.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual.* Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language. Prentice Hall*.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. C für Mikrocontroller. Franzis-Verlag. ［ドイツ語］
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.

- Meyers, Scott. *Effective STL.* Addison-Wesley.
- Sloss, Andrew N. et al, ARM System Developer's Guide: Designing and Optimizing System Software. Morgan Kaufmann.
- Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++.* Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley.

WEB サイト

推奨 Web サイト :

- Advanced RISC Machines Ltd の Web サイト (**www.arm.com**) には、ARM コアに関する情報とニュースが記載されています。
- IAR システムズの Web サイト (**www.iar.com/jp**) では、アプリケーションノートおよびその他の製品情報を公開しています。
- C 標準化作業グループの Web サイト、**www.open-std.org/jtc1/sc22/wg14**。
- C++ Standards Committee の Web サイト、**www.open-std.org/jtc1/sc22/wg21**。
- Embedded C++ Technical CommitteeのWebサイト(**www.caravan.net/ec2plus**)には、Embedded C++ 規格についての情報が公開されています。

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のディレクトリ（arm¥doc など）の記述がある場合、その場所までのフルパス (c:¥Program Files¥IAR Systems¥Embedded Workbench 7.*n*¥arm¥doc) を示します。

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	• 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。

表 1: 本ガイドで使用されている表記規則

スタイル	用途
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、*filename*.h の場合、*filename* 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	• 本ガイドや他のガイドへのクロスリファレンスを示します。 • 強調。
…	3 点リーダは、その前の項目を任意の回数繰り返せることを示します。
	IAR Embedded Workbench® IDE 固有の内容を示します。
	コマンドライン インタフェース固有の内容を示します。
	開発やプログラミングについてのヒントを示します。
	ワーニングを示します。

表 1: 本ガイドで使用されている表記規則 （続き）

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

表 2: このガイドで使用されている命名規約

ブランド名	一般名称
ARM 用 IAR アセンブラ	アセンブラ
IAR ILINK リンカ	ILINK、リンカ
IAR DLIB ライブラリ	DLIB ライブラリ

表 2: このガイドで使用されている命名規約 （続き）

パート I. ビルドツールの使用

『ARM 用 IAR C/C++ 開発ガイド』 のこのパートは、以下の章で構成されています。

- IAR ビルドツールの概要
- 組込みアプリケーションの開発
- データ記憶
- 関数
- ILINK を使用したリンク
- アプリケーションのリンク
- DLIB ランタイム環境
- アセンブラ言語インタフェース
- C の使用
- C++ の使用
- アプリケーションに関する考慮事項
- 組込みアプリケーション用の効率的なコーディング

IAR ビルドツールの概要

- IAR ビルドツール — 概要
- IAR 言語の概要
- デバイスサポート
- 組込みシステム用の特殊サポート

IAR ビルドツール — 概要

IAR 製品インストールには、ARM ベースの組込みアプリケーションのソフトウェア開発に最適なツール、サンプルコード、ユーザマニュアルのセットがあります。これらを使用することで、C/C++ またはアセンブラ言語でアプリケーションを開発できます。

IAR Embedded Workbench® は、完全な組込みアプリケーションプロジェクトの開発および管理が可能な、非常に強力な統合開発環境 (IDE) です。本製品は、コードを最大限に再利用できることや、一般的な機能とターゲット固有の機能をサポートすることで、操作が分かりやすく、非常に効率的な開発環境を実現しています。IAR Embedded Workbench は実用的な作業手法を採用しており、開発時間を大幅に短縮することができます。

IDE については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。

ビルド済みプロジェクト環境で外部ツールとして利用したい場合は、コンパイラ、アセンブラ、リンカを、コマンドライン環境で実行することもできます。

IAR C/C++ コンパイラ

ARM 用 IAR C/C++ コンパイラは、C/C++ 言語の標準機能に加えて、ARM 固有の機能を利用するための拡張機能を装備した最新のコンパイラです。

IAR アセンブラ

ARM 用 IAR アセンブラは、柔軟なディレクティブや式演算子セットを備えた強力な再配置マクロアセンブラです。C 言語プリプロセッサを内蔵しており、条件アセンブリをサポートしています。

ARM 用 IAR アセンブラでは、Advanced RISC Machines Ltd ARM アセンブラと同じニーモニックとオペランド構文を使用するため、既存のコードを容易に移行できます。詳細については、*『ARM 用 IAR アセンブラリファレンスガイド』*を参照してください。

IAR ILINK リンカ

ARM 用 IAR ILINK リンカは、組込みコントローラアプリケーションの開発に適した、強力で柔軟性のあるソフトウェアツールです。リンカは、サイズの大きい再配置可能な入力の、マルチモジュールの C/C++ プログラムや C/C++ プログラムとアセンブラプログラムの混合リンクに適していますが、サイズの小さい単一ファイルの絶対アドレスを持つアセンブラプログラムのリンクにも同様に適しています。

専用 ELF ツール

ILINK は、業界標準の ELF と DWARF の両方をオブジェクトフォーマットとして使用および生成するので、これらのフォーマットを処理する追加の IAR ユーティリティが用意されています。

- IAR Archive Tool (`iarchive`) は、複数の ELF オブジェクトファイルで構成するライブラリ（アーカイブ）の作成および操作を行います。
- IAR ELF Tool (`ielftool`) は、ELF 実行可能イメージ上でさまざまな変換（フィル、チェックサム、フォーマット変換など）を実行します。
- IAR ELF Dumper for ARM (`ielfdumparm`) は、ELF 再配置可能イメージまたは実行可能イメージの内容のテキスト表示を作成します。
- IAR ELF Object Tool (`iobjmanip`) は、ELF オブジェクトファイルの下位レベルの操作を実行する際に使用します。
- IAR Absolute Symbol Exporter (`isymexport`) は、ROM イメージファイルから絶対シンボルをエクスポートします。これらのシンボルは、アドオンアプリケーションのリンク時に使用できます。

注 : これらの ELF ユーティリティは、IAR システムズのツールにより生成されるオブジェクトファイルに非常に適しています。このため、GNU バイナリユーティリティではなく、こちらの使用をお勧めします。

外部ツール

IDE のツールチェーンを拡張する方法については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』*を参照してください。

IAR 言語の概要

ARM 用 IAR C/C++ コンパイラは以下をサポートしています。

- C：組込みシステム業界で最も幅広く使用されている高級プログラミング言語です。以下の標準に準拠したフリースタンディングアプリケーションのビルドが可能です。
  - 標準 C － C99 とも呼ばれる標準の C。本ガイドでは、この規格を *C 規格*と呼びます。
  - C89 は C94、C90、C89、ANSI C とも呼ばれ、この規格は MISRA-C が有効なときに必須です。
- C++：最新のオブジェクト指向プログラミング言語です。モジュール方式のプログラミングに最適なフル機能のライブラリを備えています。以下のすべての標準が使用できます。
  - 標準 C++ — 例外およびランタイム型情報 (RTTI) で、異なるレベルのサポートを使用できます。
  - Embedded C++ (EC++) — C++ プログラミング標準のサブセットであり、組込みシステムのプログラミング用に設計されています。業界団体である Embedded C++ Technical Committee により定義されています。「*C++ の使用*」を参照してください。
  - IAR 拡張 Embedded C++ (EEC++) — テンプレート、多重継承、名前空間、新しいキャスト演算子、標準テンプレートライブラリ (STL) などの追加機能をサポートしています。

サポートされている各言語は、*厳密*、*標準*、標準（IAR 拡張あり）のいずれかのモードで使用できます。厳密モードは、規格に厳密に準拠します。標準、標準（IAR 拡張あり）モードでは、ある程度の一般的な規格非準拠を許容しています。

C の詳細は、「*C の使用*」を参照してください。

C++、Embedded C++、および IAR 拡張 Embedded C++ の詳細については、「*C++ の使用*」を参照してください。

コンパイラでの言語の処理系定義の処理方法については、「*C 規格の処理系定義の動作*」を参照してください。

また、アプリケーションの一部または全部をアセンブラ言語で実装することもできます。*『ARM 用 IAR アセンブラリファレンスガイド』*を参照してください。

デバイスサポート

製品開発を問題なく開始できるように、IAR 製品のインストールには、広範囲のデバイス固有のサポートが提供されています。

サポートされている ARM デバイス

ARM 用 IAR C/C++ コンパイラは、命令セットバージョン 4、5、6、6M、7 に基づいていくつかの異なる ARM コアおよびデバイスをサポートします。コンパイラが生成するオブジェクトコードは、コア間で準拠しているバイナリであるとは限りません。そのため、コンパイラのプロセッサオプションを指定する必要があります。デフォルトコアは ARM7TDMI です。

事前に定義されているサポートファイル

IAR 製品のインストールには、さまざまなデバイスをサポートするための定義済ファイルが含まれています。追加のファイルが必要な場合は、既存のファイルをテンプレートとして使用することにより、作成できます。

I/O ヘッダファイル

標準の周辺ユニットは、デバイス専用の I/O ヘッダファイル（拡張子は `h`）で定義されています。製品パッケージには、リリース時に入手可能なすべてのデバイス用の I/O ファイルが付属しています。これらのファイルは、`arm¥inc¥<vendor>` ディレクトリにあります。該当するインクルードファイルをアプリケーションソースファイルにインクルードしてください。追加の I/O ヘッダファイルが必要な場合は、既存のヘッダファイルをテンプレートとして使用することにより、作成できます。ヘッダファイルの形式について詳しくは、`arm¥doc` ディレクトリの `EWARM_HeaderFormat.pdf` を参照してください。

リンカ設定ファイル

`arm¥config` ディレクトリには、サポートされているすべてのデバイス用に規制のリンカ設定ファイルがあります。これらのファイルのファイル名拡張子は `icf` で、リンカで必要な情報が含まれています。リンカ設定ファイルの詳細は、「81 ページの *コードおよびデータの配置（リンカ設定ファイル）*」を、リファレンス情報については「*リンカ設定ファイル*」を参照してください。

デバイス記述ファイル

デバッガは、使用可能なメモリエリア、周辺レジスタおよびこれらのグループの定義など、いくつかのデバイス固有の要件を、デバイス記述ファイルを使用して処理します。これらのファイルは、`arm¥config` ディレクトリにあり、そのファイル名拡張子は `ddf` です。周辺レジスタおよびそのグループは、

個別のファイル（ファイル名拡張子 sfr）で定義できます。これは、ddf ファイルに含まれています。これらのファイルの詳細については、arm¥doc ディレクトリにある『*ARM 用 C-SPY® デバッグガイド*』および EWARM_DDFFORMAT.pdf を参照してください。

開発を開始するためのサンプルプロジェクト

arm¥examples ディレクトリには、開発を問題なく開始できるように、数百の動作アプリケーションのサンプルが提供されています。これらのサンプルは、LED を点滅させる単純なものから、USB のマスストレージコントーラまで、その範囲もさまざまです。これらのサンプルは、サポートされているほとんどのデバイスで提供されています。

組込みシステム用の特殊サポート

ここでは、コンパイラで ARM コア固有の機能をサポートするために提供されている拡張の概要を説明します。

拡張キーワード

コンパイラは、コード生成方法の設定に使用するキーワードセットを提供しています。たとえば、データオブジェクトへのアクセスと保存を制御するキーワードや、関数が内部的にどのように機能するか、およびどのように呼出し / リターンを行うかを制御するキーワードなどがあります。

IDE では、デフォルトで言語拡張が有効になっています。

コマンドラインオプション -e を指定すると、拡張キーワードが使用可能になり、変数名として使用できないように予約されます。詳細は、263 ページの *-e* を参照してください。

拡張キーワードの詳細は、「*拡張キーワード*」を参照してください。61 ページの *データ記憶*、65 ページの *関数*を参照してください。

プラグマディレクティブ

プラグマディレクティブは、コンパイラの動作（メモリの配置方法、拡張キーワードの許可 / 禁止、ワーニングメッセージの表示 / 非表示など）を制御します。

プラグマディレクティブは、コンパイラでは常に有効になっています。プラグマディレクティブは C 規格に準拠しており、ソースコードの移植性を確認する場合に非常に便利です。

プラグマディレクティブの詳細は、「*プラグマディレクティブ*」を参照してください。

定義済シンボル

定義済プリプロセッサシンボルを使用して、コンパイルの時間やコンパイラのビルド番号など、コンパイル時の環境を調べることができます。

定義済シンボルの詳細は、「*プリプロセッサ*」を参照してください。

低レベル機能へのアクセス

アプリケーションのハードウェア関連部分では、低レベル機能へのアクセスが必要不可欠です。このコンパイラは、組込み関数、C モジュールとアセンブラモジュールの混在、インラインアセンブラなどの方法でサポートしています。これらの方法については、151 ページの *C 言語とアセンブラの結合*を参照してください。

組込みアプリケーションの開発

- IAR ビルドツールを使用した組込みソフトウェアの開発
- ビルドプロセス — 概要
- アプリケーションの実行 — 概要
- アプリケーションのビルド — 概要
- 基本的なプロジェクト設定

IAR ビルドツールを使用した組込みソフトウェアの開発

通常、専用マイクロコントローラに記述される組込みソフトウェアは、何らかの外部イベントの発生を待機するエンドレスループとして設計されます。このソフトウェアは、ROM に置かれ、リセット時に実行されます。この種のソフトウェアを作成する際には、いくつかのハードウェア要因およびソフトウェア要因を考慮する必要があります。コンパイラオプション、拡張キーワード、プラグマディレクティブなどを利用することができます。

メモリのマッピング

組込みシステムには、通常、オンチップ RAM、外部 DRAM、外部 SRAM、外部 ROM、外部 EEPROM、フラッシュメモリなど、さまざまなタイプのメモリが含まれます。

組込みソフトウェア開発者は、これらのさまざまなメモリタイプの機能を理解する必要があります。たとえば、オンチップ RAM は、通常、他のメモリタイプより高速なので、実行時間重視のアプリケーションでは、頻繁にアクセスされる変数をこのメモリに配置することでメリットを得ることができます。逆に、アクセスは頻繁に行われないが、電源を切った後でもその値を保持する必要がある設定データは、EEPROM またはフラッシュメモリに保存する必要があります。

メモリを効率的に使用するため、コンパイラでは、関数とデータオブジェクトのメモリへの配置を制御するためのさまざまな仕組みを提供しています。詳細については、217 ページの*データと関数のメモリ配置制御*を参照してください。リンカは、リンカ設定ファイルで指定したディレクティブに従って、メモリにコードおよびデータのセクションを配置します（81 ページの*コード*

*およびデータの配置（リンカ設定ファイル）*を参照)。

周辺ユニットとの通信

外部デバイスがマイクロコントローラに接続される場合、信号伝達用インタフェースを初期化および制御し、たとえば、チップセレクトピンを使用して、その外部デバイスを選択し、外部割込みシグナルを検出および処理して行います。通常、初期化および制御はランタイムに実行する必要があります。通常、これはスペシャルファンクションレジスタ (SFR) を使用して行います。これらのレジスタは、通常、チップ設定を制御するビットを含む、専用アドレスで使用できます。

標準の周辺ユニットは、デバイス専用の I/O ヘッダファイル（拡張子は `h`）で定義されています。42 ページの*デバイスサポート*を参照してください。例については、230 ページの*特殊機能レジスタへのアクセス*を参照してください。

イベント処理

組込みシステムでは、ボタン押下の検出など、外部イベントを即座に処理するために*割込み*を使用します。通常、コード中で割込みが発生すると、コアはすぐにコードの実行を停止し、その代わりに割込みルーチンの実行を開始します。

コンパイラには、ハードウェアおよびソフトウェア割込みを管理するためのさまざまな基本関数が用意されています。つまり、C で割込みルーチンを記述できるということです（67 ページの *Cortex-M デバイスの割込み関数*と 68 ページの *ARM7/9/11、Cortex-A、およびCortex-R の割込み関数*を参照)。

システム起動

すべての組込みシステムでは、アプリケーションの `main` 関数が呼出される前に、システム起動コードが実行され、ハードウェアとソフトウェアの両方のシステムが初期化されます。CPU は、固定メモリアドレスから実行を開始することで、これを行います。

組込みソフトウェア開発者は、この起動コードを専用メモリアドレスに配置するか、ベクタテーブルからポインタを使用してアクセスできるようにしなければなりません。つまり、起動コードおよび初期ベクタテーブルは、ROM、EPROM、フラッシュなど、不揮発性メモリに配置する必要があります。

C/C++ アプリケーションでは、さらに、すべてのグローバル変数を初期化する必要があります。この初期化は、リンカおよびシステム起動コードで扱われます。詳細については、51 ページの*アプリケーションの実行 — 概要*を参照してください。

リアルタイムオペレーティングシステム

通常、組込みアプリケーションは、システムで実行する唯一のソフトウェアです。ただし、RTOS を使用した場合、いくつかのメリットがあります。

たとえば、優先順位の高いタスクのタイミングが、優先順位の低いタスクで実行されるプログラムの他の部分による影響を受けることはありません。これにより、一般的に、プログラムの順序をより簡単に制御できるようになります。また、CPU を効率的に使用し、待機時に CPU を低電力モードにすることで、消費電力が削減されます。

RTOS を使用すると、プログラムの判読および保守が簡単になり、多くの場合、サイズも小さくなります。アプリケーションコードは、それぞれが完全に独立したタスクに明確に分割できます。これにより、1 人の開発者または開発者のグループが担当できるように開発作業を個々のタスクに簡単に分割できるので、チームでの共同作業がより円滑になります。

さらに、RTOS を使用することで、ハードウェア依存関係が削減され、アプリケーションに明確なインタフェースが作成されるので、異なるターゲットハードウェアにプログラムを移植しやすくなります。

他のビルドツールとの相互運用

IAR コンパイラおよびリンカは、AEABI (Embedded Application Binary Interface for ARM) のサポートを提供します。このインタフェース仕様の詳細については、Web サイト `www.arm.com` を参照してください。

このインタフェースでは、これをサポートするベンダ間での相互運用性が提供されるというメリットがあります。アプリケーションは、AEABI 標準に準拠しているのであれば、別のベンダで生成されたオブジェクトファイルのライブラリで構築し、任意のベンダのリンカでリンクできます。

AEABI は、C/C++ オブジェクトコード、C ライブラリの完全な互換性を規定します。AEABI には、C++ ライブラリの仕様は含まれません。

IAR ビルドツールでの AEABI サポートの詳細については、207 ページの *AEABI への準拠*を参照してください。

ARM IAR ビルドツールのバージョン 6.xx 以降は、以前のバージョンの製品と完全に互換ではありません。詳細は *『ARM 用 IAR Embedded Workbench® 移行ガイド』* を参照してください。

詳細については、207 ページの *リンカの最適化*を参照してください。

ビルドプロセス — 概要

このセクションでは、ビルドプロセスの概要について説明します。つまり、コンパイラ、アセンブラ、リンカなどさまざまなビルドツールがどのように組み合わさり、ソースコードから実行可能イメージに移行するかについて説明します。

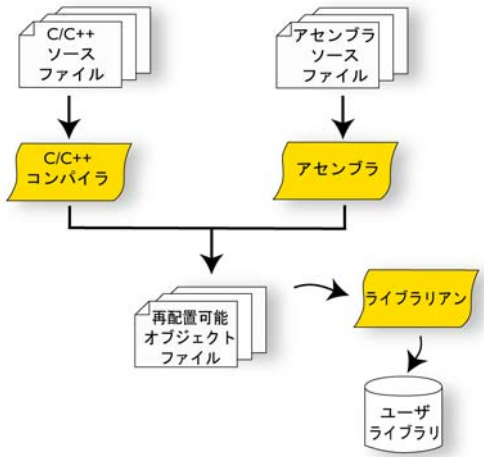
実際のプロセスをより理解できるように、IAR インフォメーションセンタで使用できるチュートリアルを 1 つ以上実行してみてください。

変換プロセス

アプリケーションソースファイルを中間オブジェクトファイルに変換するツールは IDE に 2 つあります。それは、IAR C/C++ コンパイラおよび IAR アセンブラです。これらのいずれも、デバッグ情報用の DWARF フォーマットを含む、業界標準のフォーマット ELF で再配置可能オブジェクトファイルを生成します。

注 : このコンパイラは、C/C++ ソースコードをアセンブラソースコードに変換するときにも使用できます。必要な場合、オブジェクトコードにアセンブルできるアセンブラソースコードを修正できます。IAR アセンブラの詳細については、*『ARM 用 IAR アセンブラリファレンスガイド』* を参照してください。

以下の図は、変換プロセスを示しています。



変換後は、任意の数のモジュールを 1 つのアーカイブ、つまりライブラリにパッキングすることができます。ライブラリを使用する重要な理由は、ライブラリの各モジュールが条件付きでアプリケーションにリンクされるという

ことです。すなわち、オブジェクトファイルとして提供されたモジュールによって直接的または間接的に使用されるモジュールのみアプリケーションに含まれることになります。また、ライブラリを作成してから、IAR ユーティリティ iarchive を使用することもできます。

リンク処理

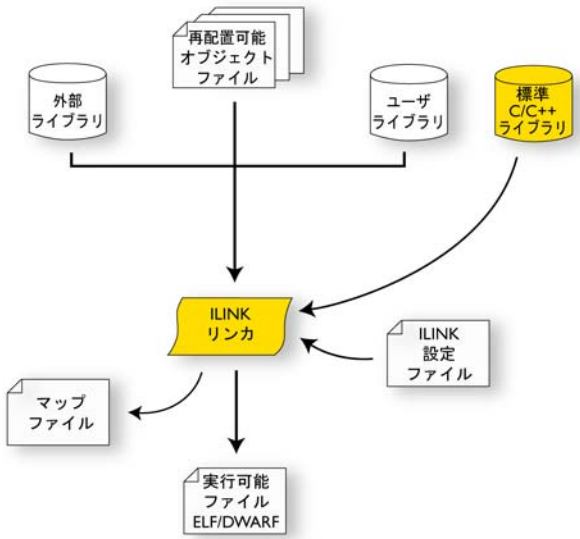
IAR コンパイラおよびアセンブラにより生成されるオブジェクトファイルおよびライブラリの再配置可能モジュールは、そのまま実行することはできません。これらが実行可能なアプリケーションとなるには、*リンク*される必要があります。

注 : 別のベンダのツールセットにより生成されたモジュールもビルドに含めることができます。ただし、AEABI 準拠でない場合、同じベンダのコンパイラユーティリティライブラリが必要なので注意してください。

最終的なアプリケーションのビルドには、IAR ILINK リンカ (ilinkarm.exe) が使用されます。通常は、リンカでは入力として以下の情報が必要になります。

- いくつかのオブジェクトファイル、場合によっては特定のライブラリ
- プログラムの開始ラベル（デフォルトで設定）
- ターゲットシステムのメモリ内でのコードおよびデータの配置を記述したリンカ設定ファイル

以下の図は、リンク処理を示しています。



注 : 標準の C/C++ ライブラリには、コンパイラのサポートルーチンと、C/C++ 標準ライブラリ関数の実装が含まれます。

リンク中、リンカはエラーメッセージおよびログメッセージを `stdout` および `stderr` に生成することがあります。このログメッセージは、アプリケーションがなぜリンクされたかを理解する場合、たとえば、モジュールが含まれた理由やセクションが削除された理由を理解するときに役に立ちます。

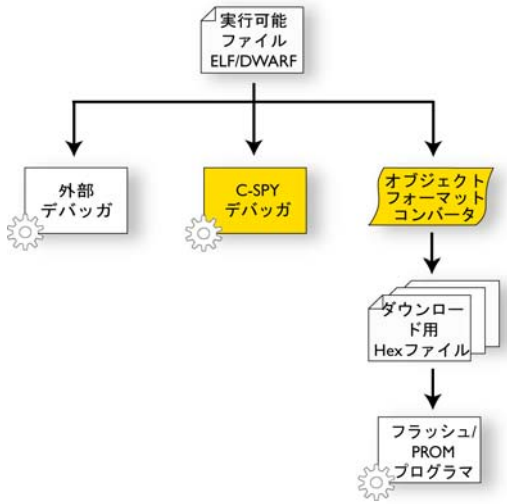
リンカにより実行される手順について詳しくは、79 ページの *リンクプロセスの詳細*を参照してください。

リンク後

IAR ILINK リンカは、実行可能イメージを含む ELF フォーマットの絶対オブジェクトファイルを生成します。リンク後、生成された絶対実行可能イメージは以下のことに使用できます。

- IAR C-SPY デバッガ、または ELF や DWARF を読み取るその他の互換性のある外部デバッガへのロード。
- フラッシュ /PROM プログラマを使用したフラッシュ /PROM へのプログラミング。これを実現するには、イメージの実際のバイトを標準の Motorola 32-bit S-record フォーマットまたは Intel Hex-32 フォーマットに変換する必要があります。この変換には、`ielftool` を使用します（483 ページの *IAR ELF ツール — ielftool* 参照）。

以下の図は、絶対出力 ELF/DWARF ファイルで可能な使用方法を示します。



アプリケーションの実行 — 概要

ここでは 3 つのフェーズに分かれた組込みアプリケーションの実行の概要を説明します。

- 初期化フェーズ
- 実行フェーズ
- 終了フェーズ

初期化フェーズ

初期化フェーズは、アプリケーションの起動時（CPU のリセット時）、`main` 関数が入力される前に実行されます。初期化フェーズは、簡単に以下のように分割できます。

- ハードウェア初期化。通常、少なくともスタックポインタが初期化されます

  ハードウェア初期化は、通常、システム起動コード `cstartup.s` で実行され、必要に応じて、ユーザが提供する最低レベルのルーチンで実行されます。また、ハードウェアの残りの部分のリセット / 起動や、ソフトウェア C/C++ システム初期化の準備のための CPU などの設定が行われる場合もあります。

- ソフトウェア C/C++ システム初期化

  一般的に、この初期化フェーズでは、`main` 関数が呼出される前に、すべてのグローバル（静的にリンカされた）C/C++ シンボルがその正しい初期化値を受け取っていることが前提です。

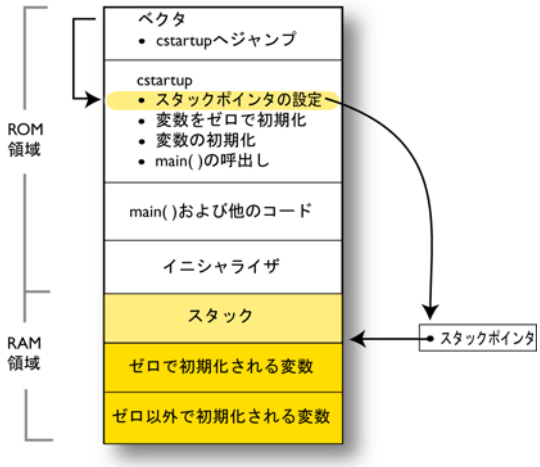
- アプリケーション初期化

  これは、使用しているアプリケーションにより異なります。RTOS カーネルの設定や、RTOS が実行するアプリケーションの初期タスクの開始が含まれます。ベアボーンアプリケーションでは、さまざまな割込みの設定、通信の初期化、デバイスの初期化などが含まれます。

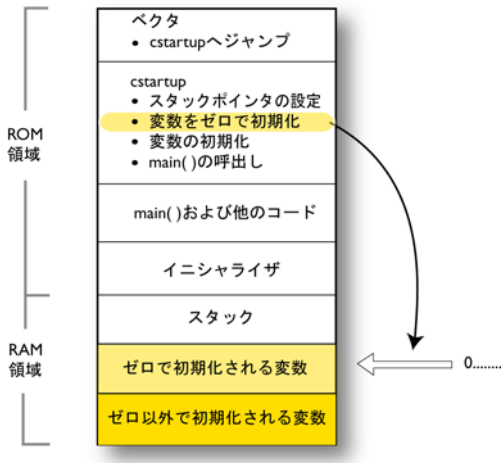
ROM/ フラッシュベースのシステムでは、定数や関数がすでに ROM に配置されています。RAM に配置されたすべてのシンボルは、`main` 関数が呼出される前に初期化される必要があります。また、リンカにより、利用可能な RAM は、すでに変数、スタック、ヒープなどの異なるエリアに分割されています。

以下の一連の図は、初期化の各種段階の概要を簡単に示します。

I アプリケーションが起動したら、システム起動コードは、まず、あらかじめ定義されたスタックエリアの最後を指すようにするスタックポインタの初期化など、ハードウェアの初期化を実行します。

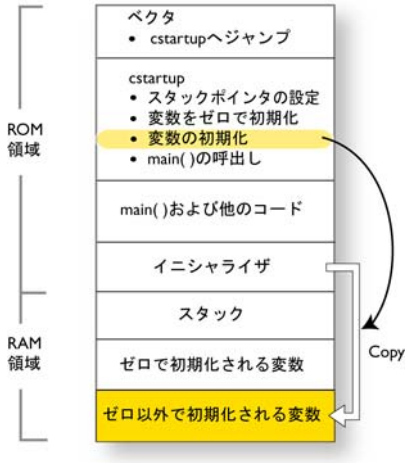


2 次に、ゼロ初期化されるメモリがクリアされます。すなわち、これらのメモリにゼロが埋め込まれます。

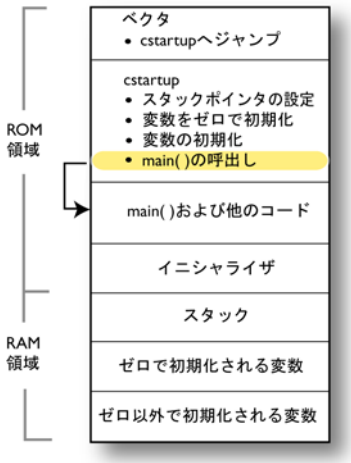


一般的に、これは*ゼロで初期化されるデータ*などのデータで、たとえば `int i = 0;` などで宣言される変数です。

3 *初期化されるデータ*、たとえば `int i = 6;` のように宣言されたデータでは、イニシャライザが ROM から RAM にコピーされます。



4 最後に、main 関数が呼出されます。



各段階の詳細については、125 ページの *システムの起動と終了*を参照してください。データ初期化の詳細については、84 ページの *システム起動時の初期化*を参照してください。

実行フェーズ

組込みアプリケーションのソフトウェアは、通常、割込み駆動型のループか、外部相互処理や内部イベントを制御するためのポーリングを使用するループのいずれかで実装されます。割込み駆動型システムの場合、割込みは、通常、main 関数の開始時に初期化されます。

リアルタイム動作システムで、応答性が重要な場合、マルチタスクシステムが必要になることがあります。つまり、アプリケーションソフトウェアは、リアルタイムオペレーティングシステムで補足する必要があります。この場合、RTOS およびさまざまなタスクは、main 関数の開始前に初期化される必要があります。

終了フェーズ

一般的に、組込み関数は終了しません。終了する場合、正しい終了動作を定義する必要があります。

アプリケーションを制御したまま終了するには、標準 C ライブラリ関数の exit、_Exit、abort のいずれかを呼出すか、main から戻ります。main から戻ると、exit 関数が実行されます。すなわち、静的およびグローバル変数の

C++ デストラクタが呼出され（C++ のみ）、開いているすべてのファイルが閉じます。

ただし、プログラムロジックが間違っている場合、アプリケーションを制御したまま終了できず、異常終了して、システムがクラッシュすることがあります。

これらの詳細は、128 ページの*システム終了*を参照してください。

アプリケーションのビルド — 概要

コマンドラインインタフェースで以下のコマンドを実行すると、デフォルト設定を使用して、ソースファイル `myfile.c` がオブジェクトファイル `myfile.o` にコンパイルされます。

```
iccarm myfile.c
```

また、重要なオプションもいくつか指定する必要があります（55 ページの*基本的なプロジェクト設定*を参照）。

コマンドラインで以下のコマンドを実行すると、リンカが起動します。

```
ilinkarm myfile.o myfile2.o -o a.out --config my_configfile.icf
```

この例では、`myfile.o` および `myfile2.o` はオブジェクトファイルであり、`my_configfile.icf` はリンカ設定ファイルです。オプション `-o` は、出力ファイルの名前を指定します。

注 : デフォルトではアプリケーションが起動するラベルは、`__iar_program_start` です。このラベルは、`--entry` コマンドラインオプションを使用して変更できます。

プロジェクトをビルドする際に、IAR Embedded Workbench IDE は［ビルド］メッセージウィンドウで詳細なビルド情報を生成することができます。この情報は、たとえばコマンドライン上でビルドするときにバッチファイルを生成する基礎として役立つことがあります。情報はコピーしてテキストファイルに貼り付けることができます。詳細なビルド情報の生成を有効にするには、**［ツール］> ［オプション］> ［メッセージ］**を選んで、オプション **［ビルドメッセージの表示 : すべて］**を選択します。

基本的なプロジェクト設定

ここでは、使用する ARM デバイスに最適なコードをコンパイラおよびリンカで生成するために必要なプロジェクトセットアップの基本設定の概要を説明します。オプションの指定は、コマンドラインインタフェースや IDE で行えます。

以下の設定が必要です。

- プロセッサ設定。すなわち、派生プロセッサ、CPU モード、インターワーク、VFP および浮動小数点演算、バイトオーダの設定です
- 最適化設定
- ランタイム環境
- ILINK 設定のカスタマイズ（「*アプリケーションのリンク*」を参照）

これらの設定に加えて、その他多数のオプションや設定により、結果をさらに詳細に調整できます。オプションの設定方法の詳細、および利用可能なすべてのオプションの一覧については、それぞれ「*コンパイラオプション*」、「*リンカオプション*」、および *『ARM 用 IDE プロジェクト管理およびビルドガイド』* をそれぞれ参照してください。

プロセッサ構成

コンパイラで最適なコードを生成するには、使用する ARM コアに合わせてコンパイラを設定する必要があります。

プロセッサ選択

ARM 用 IAR C/C++ コンパイラは、命令セットバージョン 4、5、6、7 に基づいていくつかの異なる ARM コアおよびデバイスをサポートします。サポートされているすべてのコアでは、Thumb 命令および 64 ビット乗算命令がサポートされます。コンパイラが生成するオブジェクトコードは、コア間で準拠しているバイナリであるとは限りません。そのため、コンパイラのプロセッサオプションを指定する必要があります。デフォルトコアは ARM7TDMI です。

IDE での［**プロセッサ選択**］オプションの設定については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。

ARM コアを指定するには、`--cpu` オプションを使用します。構文については、254 ページの *--arm* と 288 ページの *--thumb* を参照してください。

インターワーク

`--interwork` オプションを使用してコードをコンパイルすると、ARM および Thumb コードを自由に混在させることができ、インターワーク関数は、ARM と Thumb の両方のコードから呼出すことができます。インターワークは、命令セットバージョン 5、6、7 に基づいたデバイス、または `--aeabi` コンパイルオプションを使用した場合のデフォルトです。106 ページのベニアを参照してください。

IDE での［**インターワークコードを生成**］オプションの設定については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。


プロジェクトで相互作用機能を指定するには、`--interwork` オプションを使用します。構文については、269 ページの *--interwork* を参照してください。

VFP および浮動小数点演算

ベクタ浮動小数点 (VFP) コプロセッサを含む ARM コードを使用している場合、`--fpu` オプションを使用して、ソフトウェア浮動小数点ライブラリルーチンではなく、コプロセッサを使用して、浮動小数点演算を実行するコードを生成できます。


IDE での **[FPU]** オプションの設定については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。


ARM コアを指定するには、`--fpu` オプションを使用します。構文については、267 ページの *--fpu* を参照してください。

バイトオーダ

コンパイラはビッグエンディアンとリトルエンディアンのバイトオーダをサポートしています。アプリケーションのすべてのユーザおよびライブラリモジュールで、同じバイトオーダを使用する必要があります。


IDE での［**エンディアンモード**］オプションの設定については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。


プロジェクトでバイトオーダを指定するには、`--endian` オプションを使用します（265 ページの *--endian* を参照）。

速度とサイズの最適化

不要なコードの削除や定数の伝播、インライン化、共通部分式除去、静的クラスタ化、命令スケジューリング、精度調整などを実行するコンパイラのオプティマイザ。また、帰納変数の削除などのループ最適化も実行します。

最適化レベルを複数のレベルから選択することができ、最高レベルの場合は、最適化の目標として、*サイズ*、*速度*、*バランス*から選択できます。ほとんどの最適化で、アプリケーションのサイズ縮小と高速化の両方が実現されます。しかし、効果が得られない場合は、コンパイラはユーザが指定した最適化目標に準じて、最適化の実行方法を決定します。

最適化レベルと目標は、アプリケーション全体、ファイル単位、関数単位のいずれのレベルに対しても指定できます。また、関数インライン化などの一部の最適化を無効にすることもできます。

コンパイラの最適化と効率的なコーディングテクニックの詳細は、「*組込みアプリケーション用の効率的なコーディング*」を参照してください。

ランタイム環境

必要なランタイム環境を構築するには、ランタイムライブラリを選択し、ライブラリのオプションを設定する必要があります。場合によっては、特定のライブラリモジュールを、ユーザがカスタマイズしたモジュールでオーバライドすることも必要となります。提供されるランタイムライブラリは、IAR DLIB ライブラリです。

効率的なランタイム環境を設定するには、さまざまな機能を十分理解する必要があります（「*DLIB ランタイム環境*」を参照）。

IDE でのランタイム環境の設定

ライブラリは、**[ターゲット]**、**[ライブラリ構成]**、**[ライブラリオプション]** のページで **[プロジェクト] > [オプション] > [一般オプション]** の設定に基づいて自動的に選択されます。正しいインクルードパスは、システムヘッダファイルとデバイス固有のインクルードファイルに自動的に設定されます。

DLIB ライブラリには、異なる設定（ノーマルとフル）があり、これにはロケール、ファイル記述子、マルチバイト文字など異なるレベルのサポートが含まれます（130 ページの *ライブラリ構成*を参照）。

コマンドラインのランタイム環境の設定

ILINK により正しいライブラリファイルが自動的に使用されるので、ライブラリファイルを明示的に指定する必要はありません。

ライブラリオブジェクトファイルに一致するライブラリ設定ファイルが自動的に使用されます。ライブラリ設定を明示的に指定するには、`--dlib_config` オブジェクトを使用します。

これらのオプションのほかに、`-I` オプションなどを使用して、アプリケーション固有のリンカオプションやアプリケーション固有のヘッダフィルのパスを指定すると便利です。

```
-I MyApplication¥inc
```

ビルド済のライブラリオブジェクトファイルの詳細については、112 ページの *ビルド済ライブラリの使用*を参照してください。

ライブラリとランタイム環境オプションの設定

オプションの設定により、ライブラリやランタイム環境のサイズを削減できます。

- 関数 `printf`、`scanf`、およびこれらの派生関数で使用されるフォーマッタ。116 ページの *printf、scanf のフォーマッタの選択* を参照してください。

- スタックやヒープのサイズ。100 ページの*スタックメモリの設定*、100 ページの*ヒープメモリの設定*を参照してください。

データ記憶

- 概要
- 自動変数とパラメータの記領域憶
- ヒープ上の動的メモリ

概要

ARM コアは、4GB のシーケンシャルメモリ（範囲は `0x00000000` ～ `0xFFFFFFFF`）を持っています。メモリ範囲には、さまざまな種類の物理メモリを設置できます。一般的な用途では、リードオンリーメモリ (ROM) とリード / ライトメモリ (RAM) の両方を使用します。また、メモリ範囲の一部に、プロセッサが管理するレジスタや周辺ユニットが含まれます。

さまざまなデータ記憶方法

一般的な用途では、データを以下の 3 種類の方法でメモリに格納できます。

- 自動変数

  static として宣言された変数を除き、関数にローカルな変数はすべて、レジスタまたはスタックに格納されます。これらの変数は、関数の実行中にアクセスが可能です。関数が呼出し元に戻ると、このメモリ空間は無効になります。詳細については、62 ページの *自動変数とパラメータの記領域憶*を参照してください。
- グローバル変数、モジュール静的変数、`static` と宣言されたローカル変数

  この場合、メモリは 1 度だけ割り当てられます。ここでの「静的」とは、このタイプの変数に割り当てられたメモリ容量がアプリケーション実行中に変化しないことを意味します。ARM コアには、単一アドレス空間があり、コンパイラはフルメモリアドレッシングをサポートします。
- 動的に割り当てられたデータ

  アプリケーションは、データを *ヒープ*上に割り当てることができます。この場合、アプリケーションが明示的にヒープをシステムに解放するまでデータは有効な状態で保持されます。このタイプのメモリは、アプリケーションを実行するまで必要なオブジェクト量がわからない場合に便利です。動的に割り当てられたデータを、メモリ容量が限られているシステムや、長期間実行するシステムで使用すると、問題が生じる危険性があります。詳細については、63 ページの *ヒープ上の動的メモリ*を参照してください。

自動変数とパラメータの記領域憶

関数内で定義された（static 宣言ではない）変数は、C 言語規格では自動変数と呼ばれます。これらの変数のうち、いくつかはプロセッサのレジスタに配置され、残りはスタック上に配置されます。意味上は、これらは同一です。主な違いは、変数をスタックに配置するよりもレジスタに配置した方がアクセスが高速で、必要なメモリ容量も小さくなるということです。

自動変数は、関数の実行中にのみ有効になります。関数から戻るときに、スタックに配置されたメモリが解放されます。

スタック

スタックには、以下を格納できます。

- レジスタに格納されていないローカル変数、パラメータ
- 式の中間結果
- 関数のリターン値（レジスタで引き渡される場合を除く）
- 割込み時のプロセッサ状態
- 関数から戻る前に復元する必要のあるプロセッサレジスタ（呼出し先保存レジスタ）

スタックは、2 つのパートで構成される固定メモリブロックです。最初のパートは、現在の関数を呼出した関数やその関数を呼出した関数などに配置されたメモリを格納します。後のパートは、割当て可能な空きメモリを格納します。2 つのパートの境界をスタックの先頭と呼び、専用プロセッサレジスタであるスタックポインタで表します。スタック上のメモリは、スタックポインタを移動することで配置します。

関数が空きメモリを含むスタックエリアのメモリを参照しないようにする必要があります。これは、割込みが発生した場合に、呼出し先の割込み関数がスタック上のメモリの割当て、変更、割当て解除を行うことがあるためです。

198 ページの *スタックについて*および 100 ページの *スタックメモリの設定*を参照してください。

利点

スタックの主な利点は、プログラムの異なる部分にある関数が、同一のメモリ空間を使用してデータを格納できることです。ヒープとは異なり、スタックでは断片化やメモリリークが発生しません。

関数が自身を呼出すことができます（再帰関数）。また、呼出しごとに自身のデータをスタックに格納できます。

潜在的な問題

スタックの仕組み上、関数から戻った後も有効にすべきデータを格納することはできません。次の関数で、よくあるプログラミング上の誤りを説明します。この関数は、変数 x へのポインタを返します。この変数は、関数から戻るときに無効になります。

```
int *MyFunction()
{
  int x;
  /* 何らかの処理 */
  return &x; /* 誤り */
}
```

別の問題として、スタック容量が不足する危険性があります。この問題は、関数が別の関数を呼出し、その関数がさらに別の関数を呼出す場合など、各関数のスタック使用量の合計がスタックのサイズよりも大きくなるときに発生します。大きなデータオブジェクトがスタック上に格納されたり、再帰関数が使用されると、リスクは高くなります。

ヒープ上の動的メモリ

ヒープ上で配置されたオブジェクト用のメモリは、そのオブジェクトを明示的に解放するまで有効です。このタイプのメモリ記憶領域は、実行するまでデータ量がわからないアプリケーションの場合に非常に便利です。

C では、メモリは標準ライブラリ関数の `malloc` や、関連関数の `calloc`、`realloc` のいずれかを使用して配置します。メモリは、`free` を使用して解放します。

C++ では、`new` という特殊なキーワードによってメモリの割当てやコンストラクタの実行を行います。`new` を使用して割り当てたメモリは、キーワード `delete` を使用して解放する必要があります。

100 ページの *ヒープメモリの設定*を参照してください。

潜在的な問題

ヒープ上で割り当てたオブジェクトを使用するアプリケーションは、ヒープ上でオブジェクトを配置できない状況が発生しやすいため、慎重に設計する必要があります。

アプリケーションで使用するメモリ容量が大きすぎる場合、ヒープが不足することがあります。また、すでに使用されていないメモリが解放されていない場合にも、ヒープが不足することがあります。

配置されたメモリブロックごとに、管理用に数バイトのデータが必要になります。小さなブロックを多数配置するアプリケーションの場合は、管理用データが原因のオーバヘッドが問題になることがあります。

断片化の問題もあります。断片化とは、小さなセクションの空きメモリが、配置されたオブジェクトで使用されるメモリにより分断されることです。空きメモリの合計サイズがオブジェクトのサイズを超えている場合でも、そのオブジェクトに十分な大きさの連続した空きメモリがない場合は、新しいオブジェクトを配置することができません。

断片化は、メモリの割当てと解放を繰り返すほど増加する傾向があります。この理由から、長期間の実行を目的とするアプリケーションでは、ヒープ上に割り当てられたメモリの使用を回避するようにしてください。

関数

- 関数関連の拡張
- ARM および Thumb コード
- RAM での実行
- Cortex-M デバイスの割込み関数
- ARM7/9/11、Cortex-A、および Cortex-R の割込み関数
- インライン関数

関数関連の拡張

コンパイラでは、C 規格のサポートに加えて、関数を記述するための拡張が利用できます。

- 異なる CPU モードである ARM と Thumb 用にコードを生成します。これは `__arm` や `__thumb` などのターゲット固有のキーワードを参照します。こういうものがない場合は、項目を削除します
- RAM で関数を実行
- 異なるデバイスに割込み関数を記述
- 関数インライン化の制御
- 関数の最適化を円滑化
- ハードウェア機能にアクセス

コンパイラでは、これらの機能をコンパイラオプション、拡張キーワード、プラグマディレクティブ、組込み関数で実現します。

最適化の詳細は、213 ページの*組込みアプリケーション用の効率的なコーディング*を参照してください。ハードウェア操作のアクセスに使用可能な組込み関数の詳細については、「*組込み関数*」を参照してください。

ARM および Thumb コード

ARM 用 IAR C/C++ コンパイラは、32 ビットの ARM または 16 ビットの Thumb または Thumb2 命令セットのコードを生成できます。`--cpu_mode` オプション、あるいは `--arm` または `--thumb` オプションを使用して、プロジェク

トで使用する命令セットを指定します。個々の関数に対して、拡張キーワード __arm および __thumb を使用して、プロジェクト設定をオーバライドできます。コードが相互に作用する限り、ARM および Thumb コードが同じアプリケーションに混在しても問題はありません。

関数呼出しを実行する場合、コンパイラは、使用できる最も効率的なアセンブラ言語命令または命令シーケンスを生成しようとします。そのため、範囲 0x0 ～ 0xFFFFFFFF の連続する 4GB のメモリがコードの配置に使用されます。コードモジュールあたり 4MB という制限があります。

すべてのコードポインタのサイズは 4 バイトです。コードポインタからデータポインタや整数型、およびその逆の場合での暗黙的および明示的なキャスティングには制限があります。制限の詳細については、334 ページの *ポインタ型*を参照してください。

「*アセンブラ言語インタフェース*」では、アセンブラ言語からの C 関数の呼出し、およびその逆の方法に関する説明で、生成されるコードを詳しく説明しています。

RAM での実行

__ramfunc キーワードは、関数を RAM モードで実行します。つまり、リード / ライト属性を持つセクションに関数が配置されます。関数は、初期化された変数のように、システム起動時に ROM から RAM にコピーされます。詳細については、125 ページの *システムの起動と終了*を参照してください。

キーワードは、以下のようにリターン型の前に指定します。

```
__ramfunc void foo(void);
```

__ramfunc により宣言された関数が ROM にアクセスしようとすると、警告が発生します。

コードおよび定数に使用されるメモリエリア全体が無効な場合、たとえばフラッシュメモリ全体が消去された場合など、RAM に格納されている関数およびデータのみを使用できます。割込みベクタおよび割込みサービスルーチンが RAM に格納されていない限り、割込みを無効にする必要があります。

文字列リテラルおよびその他の定数は、初期化した変数を使用することで回避できます。以下に例を示します。

```
__ramfunc void test()
{
  /* myc: ROM 内のイニシャライザ */
  const int myc[] = { 10, 20 };

  /* ROM 内の文字列リテラル */
```

```
  msg("Hello");
}
```

これを次のように記述し直すことができます。

```
__ramfunc void test()
{
  /* myc: cstartup により初期化 */
  static int myc[] = { 10, 20 };

  /* hello: cstartup により初期化 */
  static char hello[] = "Hello";

  msg(hello);
}
```

詳細については、103 ページの *コードを初期化する（ROM から RAM にコピーする）* を参照してください。

Cortex-M デバイスの割込み関数

Cortex-M は、以前の ARM アーキテクチャとは割込みメカニズムが異なります。つまり、コンパイラにより提供されるプリミティブも異なります。

CORTEX-M の割込み

Cortex-M では、割込みサービスルーチンの入力とリターンは、通常の関数と同じです。つまり、特殊なキーワードは必要ありません。そのため、キーワード `__irq`、`__fiq` および `__nested` は、Cortex-M のコンパイルには使用できません。

これらの例外関数の名前は、`cstartup_M.c` および `cstartup_M.s` に定義されています。これらは、ライブラリ例外ベクタコードによって参照されます。

```
NMI_Handler
HardFault_Handler
MemManage_Handler
BusFault_Handler
UsageFault_Handler
SVC_Handler
DebugMon_Handler
PendSV_Handler
SysTick_Handler
```

ベクタテーブルは配列として実装されます。C-SPY デバッガはベクタテーブルを配置する場所を決めるときにシンボルを検索するので、`__vector_table` という名前は常にあるべきです。

定義済みの例外関数は、weak シンボルとして定義されます。weak シンボルは、重複するシンボルがない限り、リンカにより使用されます。別のシンボルが同じ名前で定義されている場合、これが優先されます。そのためアプリケーションは、上記の一覧から正しい名前を使用するだけで、独自の例外関数を定義できます。他の割込みまたは他の例外ハンドラが必要な場合には、`cstartup_M.c` または `cstartup_M.s` ファイルのコピーを作成し、ベクタテーブルに正しく追加する必要があります。

組込み関数 `__get_CPSR` および `__set_CPSR` は、Cortex-M のコンパイルには使用できません。レジスタまたは他のレジスタの値を取得または設定する必要がある場合、インラインアセンブラを使用できます。詳細については、232 ページの *C およびアセンブラオブジェクト間での値の受渡し*を参照してください。

ARM7/9/11、Cortex-A、および Cortex-R の割込み関数

ARM 用の IAR C/C++ コンパイラは、ARM7/9/11、Cortex-A、および Cortex-R デバイスの割込み関数に関連する以下の基本関数を提供します。

- 拡張キーワード : `__irq`、`__fiq`、`__swi` 、`__nested`、
- 組込み関数 : `__enable_interrupt`、`__disable_interrupt`、`__get_interrupt_state`、`__set_interrupt_state`。

注 : Cortex-M は、他の ARM デバイスとは割込みメカニズムが異なります。また、これらのデバイスとは使用できるプリミティブセットが異なります。詳細については、67 ページの *Cortex-M デバイスの割込み関数*を参照してください。

割込み関数

組込みシステムでは、ボタン押下の検出など、外部イベントを即座に処理するために割込みを使用します。

割込みサービスルーチン

通常、コード中で割込みが発生すると、コアはすぐにコードの実行を停止し、その代わりに割込みルーチンの実行を開始します。割込み処理の完了後、割込まれた関数の環境を復元することが重要です。これには、プロセッサレジスタの値やプロセッサステータスレジスタの値の復元も含まれます。これにより、割込み処理用コードの実行が終了したときに、元のコードの実行を続行できます。

コンパイラは、割込み、ソフトウェア割込み、高速割込みをサポートします。割込みタイプごとに、割込みルーチンを記述できます。

すべての割込み関数は、ARM モードでコンパイルする必要があります。Thumb モードを使用する場合、__arm 拡張キーワードまたは #pragma type_attribute=__arm ディレクティブを使用して、デフォルトの動作をオーバライドします。これは Cortex-M デバイスでは適用されません。

割込みベクタと割込みベクタテーブル

各割込みルーチンは、ARM コアのドキュメントで指定されている、例外ベクタテーブルのベクタアドレス / 命令に関連付けられます。割込みベクタは、例外ベクタテーブルへのアドレスです。ARM コアの場合は、例外ベクタテーブルはアドレス 0x0 から開始します。

デフォルトでは、ベクタテーブルは無期限でループする *デフォルトの割込みハンドラ*で定義されます。各割込みソースは割込みサービスルーチンがないので、デフォルトの割込みハンドラが呼び出されます。特定のベクタに自分のサービスルーチンを記述する場合、そのルーチンはデフォルトの割込みハンドラを上書きします。

割込み関数の定義 — 例

割込み関数を定義するには、__irq または __fiq キーワードを使用できます。次に例を示します。

```
__irq __arm void IRQ_Handler(void)
{
  /* 何らかの処理 */
}
```

割込みベクタテーブルの詳細については、ARM コアのドキュメントを参照してください。

注 : 割込み関数のリターン型は void でなければならず、パラメータの指定は一切できません。

割込みと C++ メンバ関数

static メンバ関数だけが割込み関数になれます。非静的メンバ関数を呼出す際には、オブジェクトに割当てる必要があります。割込みが発生し、割込み関数が呼出されるときは、メンバ関数の割り当てに使用可能なオブジェクトは存在しません。

例外関数のインストール

すべての割込み関数およびソフトウェア割込みハンドラは、ベクタテーブルにインストールする必要があります。これは、システム起動ファイル cstartup.s のアセンブラ言語で行われます。

標準ランタイムライブラリでの ARM 例外ベクタテーブルのデフォルトの実装は、無限ループを実装する事前定義関数にジャンプします。そのため、アプリケーションで扱われないイベントに対して発生する例外は、無限ループ (`B.`) になります。

事前定義関数は、weak シンボルとして定義されます。weak シンボルは、重複するシンボルがない限り、リンカにより使用されます。別のシンボルが同じ名前で定義されている場合、これが優先されます。そのためアプリケーションは、正しい名前を使用するだけで、独自の例外関数を定義できます。

以下の例外関数名は、`cstartup.s` で定義され、ライブラリ例外ベクタコードで参照されます。

```
Undefined_Handler
SWI_Handler
Prefetch_Handler
Abort_Handler
IRQ_Handler
FIQ_Handler
```

独自の例外ハンドラを実装するには、上記のリストから適切な実行関数名を使用して関数を定義します。

たとえば、C に割込み関数を追加するには、`IRQ_Handler` という名前の割込み関数を定義します。

```
__irq __arm void IRQ_Handler()
{
}
```

割込み関数は、C リンケージを持つ必要があります。詳細については、164 ページの*呼出し規約*を参照してください。

C++ を使用する場合の割込み関数の例を以下に示します。

```
extern "C"
{
  __irq __arm void IRQ_Handler(void);
}
__irq __arm void IRQ_Handler(void)
{
}
```

その他の変更は必要ありません。

割込みおよび高速割込み

割込みおよび高速割込み関数は、パラメータを受け取らず、値を返さないため、簡単に扱うことができます。これらのキーワードのどれかを使用します :

- 割込み関数を宣言するには、__irq 拡張キーワードまたは #pragma type_attribute=__irq ディレクティブを使用します。構文については、それぞれ 347 ページの *__irq* および 378 ページの *type_attribute* を参照してください。
- 高速割込み関数を宣言するには、__fiq 拡張キーワードまたは #pragma type_attribute=__fiq ディレクティブを使用します。構文については、それぞれ 346 ページの *__fiq* および 378 ページの *type_attribute* を参照してください。

注 : 割込み関数 (irq) および高速割込み関数 (fiq) には、リターン型 void を指定する必要があります。また、パラメータを指定することはできません。ソフトウェア割込み関数 (swi) にはパラメータを指定できます。指定した場合、値を返すことができます。デフォルトでは、4 つのレジスタ (R0-R3) のみをパラメータとして使用でき、R0-R1 のレジスタをリターン値に使用できます。

ネスト割込み

割込みは、割込みハンドラが開始される前に、ARM コアにより自動的に禁止されます。割込みハンドラが割込みを再び有効にし、関数を呼出して別の割込みが発生した場合、LR に格納されている割込み関数のリターンアドレスは、2 番目の IRQ が取得されるときにオーバライドされます。また、SPSR の内容は、2 番目の割込みが発生したときに破棄されます。__irq キーワード自体は、LR および SPSR を保存し復元しません。ネストされた割込みを処理するときに必要な必須ステップを割込みハンドラで実行するには、__irq のほかに、キーワード __nested を使用する必要があります。ネストされた割込みハンドラに対してコンパイラが生成する関数プロローグ（関数入口シーケンス）は、IRQ モードからシステムモードに切り替わります。IRQ スタックおよびシステムスタックの両方が設定されていることを確認してください。デフォルトの cstartup.s ファイルを使用する場合、両方のスタックは正しく設定されます。

ネストされた割込みを可能にするコンパイラにより生成された割込みハンドラは、IRQ 割込みのみでサポートされます。FIQ 割込みは、迅速な提供を目的としているので、通常、ネストされた割込みのオーバヘッドは非常に大きくなります。

以下の例は、ARM ベクタ割込みコントローラ (VIC) でネストされた割込みを使用する方法を示します。

```
__irq __nested __arm void interrupt_handler(void)
{
  void (*interrupt_task)();
  unsigned int vector;

  /* 割込みベクタを取得 */
  vector = VICVectAddr;

  interrupt_task = (void(*)())vector;

  /* 他の IRQ 割込みがサービスを受けることを許可 */
  __enable_interrupt();

  /* この割込みに関連するタスクを実行 */

  (*interrupt_task)();
}
```

注 : __nested キーワードでは、プロセッサモードがユーザモードまたはシステムモードのいずれかであることが必要です。

ソフトウェア割込み

ソフトウェア割込み関数は、ソフトウェア割込みハンドラ（ディスパッチャ）を必要とし、実行中のアプリケーションソフトウェアから起動され（呼出され）ます。また、引数を使用し、値を返します。このため、他の割込み関数より少し複雑になります。ここでは、ソフトウェア割込み関数を呼出すメカニズムと、ソフトウェア割込みハンドラが実際のソフトウェア割込み関数をディスパッチする方法について説明します。

ソフトウェア割込み関数の呼出し

ソフトウェア割込み関数をアプリケーションソースコードから呼出すには、アセンブラ命令 SVC #immed を使用します。ここで、immed はソフトウェア割込み番号と呼ばれる整数値です（このガイドでは、swi_number とも表記）。コンパイラでは、この命令を C/C++ ソースコードから暗黙的に生成するための簡単な方法を提供しています。関数を宣言する際に __swi キーワードおよび #pragma swi_number ディレクティブを使用することによって生成できます。

__swi 関数は、たとえば、以下のように宣言できます。

```
#pragma swi_number=0x23
__swi int swi_function(int a, int b);
```

この場合、アセンブラ命令 SVC 0x23 は、関数が呼出されるときに生成されます。

ソフトウェア割込み関数は、スタックの使用以外、パラメータおよびリターン値に関して通常の関数と同じ呼出し規則に従います（164 ページの*呼出し規約*を参照）。

詳細については、352 ページの *__swi*、377 ページの *swi_number* を参照してください。

ソフトウェア割込みハンドラと関数

割込みハンドラ（たとえば SWI_Handler）は、ソフトウェア割込み関数のディスパッチャとして機能します。割込みハンドラは、割込みベクタから呼出され、ソフトウェア割込み番号の取得と適切なソフトウェア割込み関数の呼出しを行う役割を持ちます。ソフトウェア割込み番号を C/C++ ソースコードから呼出す方法はないため、SWI_Handler はアセンブラ内に記述する必要があります。

ソフトウェア割込み関数

ソフトウェア割込み関数は、C/C++ で記述できます。__swi キーワードを関数定義で使用することにより、特定のソフトウェア割込み関数に対する正しいリターンシーケンスがコンパイラで生成されます。割込み関数定義に #pragma swi_number ディレクティブは必要ありません。

詳細については、352 ページの *__swi* を参照してください。

ソフトウェア割込みスタックポインタの設定

ソフトウェア割込みをアプリケーションで使用する場合には、ソフトウェア割込みスタックポインタ (SVC_STACK) を設定し、スタックにエリアを割り当てる必要があります。SVC_STACK ポインタは、他のスタックと一緒に cstartup.s ファイルで設定できます。例としては、割込みスタックポインタの設定を参照してください。SVC_STACK ポインタの関連エリアは、リンカ設定ファイルで設定します（100 ページの*スタックメモリの設定*を参照）。

割込み処理

割込み関数は、外部イベントが発生するときに呼出されます。通常、別の関数が実行するとすぐに呼出されます。割込み関数の実行が終了すると、元の関数に戻ります。割込まれた関数の環境の復元が必要になります。これには、プロセッサレジスタやプロセッサステータスレジスタの値の復元が含まれます。

割込みが発生すると、以下の処理が実行されます。

- 処理モードが、特定の例外に合わせて変化する
- 例外入口命令の後の命令のアドレスが、新しいモードの R14 に保存される
- CPSR の古い値が、新しいモードの SPSR に保存される
- 割込み要求は、ビット 7 の CPSR を設定して無効にされ、例外が高速割込みの場合、さらに高速割込みがビット 6 の CPSR を設定して無効にされる
- PC が、関連するベクタアドレスでの実行開始を強制される

たとえば、ベクタ 0x18 の割込みが発生した場合、プロセッサは、アドレス 0x18 でコードの実行を開始します。割込みの開始位置として使用されるメモリエリアは、割込みベクタテーブルと呼ばれます。割込みベクタの内容は、通常、割込みルーチンにジャンプする分岐命令です。

注 : 割込み関数により割込みが有効にされると、割込みルーチンから返される必要がある特殊なプロセッサレジスタは、破棄されたとみなされます。このため、返される前に復元できるように、これらを割込みルーチンで格納する必要があります。この処理は、__nested キーワードが使用されている場合は自動的に行われます。

インライン関数

関数インライン化とは、定義がコンパイル時に判明している関数を、その呼出し元関数の本体に統合し、関数呼出しによるオーバヘッドを解消することです。この最適化は、最適化レベルが［高］の場合に実行可能で、通常は実行時間が短縮されますが、コードサイズは増加する可能性があります。生成されるコードのデバッグが困難になる場合があります。インライン化が実際に行われるかどうかは、コンパイラのヒューリスティックに基づいて決定されます。

インライン化する関数は、コンパイラがヒューリスティックにより決定します。実行する最適化の内容（速度、サイズ、速度とサイズのバランス）に応じて、異なるテクニックが使用されます。通常はサイズを最適化する際はコードサイズは増加しません。

C と C++ の動作の比較

C++ では、個別のコンパイル単位における特定のインライン関数のすべての定義は、そのまま同じにする必要があります。関数がコンパイル単位のいずれかでインライン化されていない場合、これらのコンパイル単位からの定義のひとつが関数の実装として使用されます。

C では、インライン関数のインライン化されていないバージョンを含むコンパイル単位を手動で 1 つ選択する必要があります。これは、そのコンパイル

単位内で関数を extern として明示的に宣言することにより行います。複数のコンパイル単位で関数を extern として宣言すると、リンカは*複数定義*エラーを出力します。また、C ではインライン関数は静的変数や関数を参照できません。

次に例を示します。

```
// ヘッダファイル内
static int sX;
inline void F(void)
{
  //static int sY; // 静的を参照できない
  //sX;            // 静的を参照できない
}


// あるソースファイル内
// この F は使用する非インラインバージョンとして宣言
extern inline void F();
```

関数のインライン化を制御する機能

関数のインライン化を制御するしくみはいくつかあります。

- inline キーワードは、ディレクティブの直後に定義された関数をインライン化するようコンパイラに指示します。

  C または C++ モードで関数をコンパイルする場合、キーワードはそれぞれ標準の C または標準の C++ における定義に従って解釈されます。

  主な動作の違いは、標準の C では一般的にヘッダファイルでインライン定義を提供できません。コンパイル単位のひとつでインライン定義を外部と定義することにより、外部の定義を提供する必要があります。

- #pragma inline は、inline キーワードと似ていますが、コンパイラは常に C++ のインライン動作を使用する点が異なります。

  #pragma inline ディレクティブを使用することで、コンパイラのヒューリスティックを無効化して、インライン化を強制するか、完全に無効にすることができます。詳細については、367 ページの *inline* を参照してください。

- --use_c++_inline を使用すると、標準の C ソースコードファイルをコンパイルする際に C++ の動作を使用するようコンパイラに強制的に指示します。

- --no_inline、#pragma optimize=no_inline、#pragma inline=never は、いずれも関数のインライン化を無効にします。デフォルトでは、関数のインライン化は最適化レベル［高］で有効になっています。

コンパイラは、定義が分かっている場合にのみ関数をインライン化することができます。これは通常、現在の翻訳単位にのみ制限されています。ただし、複数ファイルのコンパイルの `--mfc` コンパイラオプションが使用される場合、コンパイラは複数ファイルコンパイル単位のすべてのコンパイル単位からの定義をインライン化できます。詳細については、222 ページの*複数ファイルのコンパイルユニット*を参照してください。

関数のインライン化の最適化の詳細については、225 ページの*関数インライン化*を参照してください。

ILINK を使用したリンク

- リンクの概要
- モジュールおよびセクション
- リンクプロセスの詳細
- コードおよびデータの配置（リンカ設定ファイル）
- システム起動時の初期化
- スタック使用量解析

リンクの概要

IAR ILINK リンカは、組込みアプリケーションの開発に適した、強力で柔軟性のあるソフトウェアツールです。IAR ILINK リンカは、サイズの大きい再配置可能なマルチモジュールの C/C++ プログラムや、C/C++ プログラムとアセンブラプログラムの混合リンクに適していますが、サイズの小さい単一ファイルの絶対アドレスを持つアセンブラプログラムのリンクにも同様に適しています。

リンカでは、再配置可能な 1 つまたは複数のオブジェクトファイル（IAR システムズのコンパイラまたはアセンブラで作成）を、1 つまたは複数のオブジェクトライブラリから選択した部品と組み合わせて、業界標準形式の *Executable and Linking Format* (ELF) で、実行可能なイメージを作成します。

リンカでは、リンクするアプリケーションが実際に必要なライブラリモジュール（ユーザライブラリおよび標準 C/C++ の派生ライブラリ）だけを自動的にロードします。さらに、重複セクションや必要のないセクションを削除します。

ILINK では、ARM と Thumb の両方のコード、およびこれらの組み合わせのリンクが可能です。自動的に追加命令（ベニア）を挿入することで、ILINK では、リンク先が呼出しや分岐に到達し、プロセッサの状態が必要に応じて切り変わることを確認します。ベニアの生成方法の詳細については、106 ページのベニアを参照してください。

リンカでは*設定ファイル*を使用します。このファイルでは、ターゲットシステムのメモリマップのコードやデータ領域に対して、別々の位置を指定できます。このファイルではアプリケーションの初期化フェーズの自動処理もサ

ポートしています。すなわち、イニシャライザのコピーや、場合によっては解凍も行って、グローバル変数領域とコード領域のイニシャライズを行います。

ILINK が作成する最終出力は、ELF（デバッグ情報の DWARF を含む）形式の実行可能なイメージを含む、絶対オブジェクトファイルです。このファイルは、C-SPY のほか ELF/DWARF をサポートする互換性のあるデバッガにダウンロードできます。あるいは、EPROM またはフラッシュに格納することができます。

ELF ファイルを使用するために、さまざまなツールが提供されています。付属のユーティリティについては、40 ページの *専用 ELF ツール*を参照してください。

モジュールおよびセクション

各再配置可能オブジェクトファイルには、以下の要素で構成される 1 つのモジュールが含まれます。

- コードまたはデータのいくつかのセクション
- ランタイム環境のバージョンなど、さまざまな情報を指定するランタイム属性
- DWRAF フォーマットのデバッグ情報（オプション）
- 使用されているすべてのグローバルシンボルおよびすべての外部シンボルのシンボルテーブル

*セクション*とは、メモリ内の物理位置に配置されるデータやコードを含む論理エンティティです。セクションは、いくつかの *セクションフラグメント*で構成できます。セクションフラグメントは、通常、各変数または関数（シンボル）に対して 1 つです。セクションは、RAM または ROM のいずれかに配置できます。通常の組込みアプリケーションでは、RAM に配置したセクションには内容がなく、エリアを占有するだけです。

各セクションには、名前とその内容を判別するための型属性が付けられています。この型属性は、ILINK 設定のセクションを選択するときに（名前とともに）使用されます。一般的に使用される属性を以下に示します。

code	実行可能コード
readonly	定数変数
readwrite	初期化される変数
zeroinit	変数のゼロ初期化

注 : これらのセクション型（アプリケーションの一部であるコードおよびデータを含むセクション）のほかに、最終オブジェクトファイルには、デバッグ情報や型の異なるメタ情報を含むセクションなど、その他多くの型のセクションが含まれます。

セクションは、最小のリンク可能ユニットです。ただし、可能な場合、ILINK は、最終アプリケーションからさらに小さいユニット（セクションフラグメント）を実行できます。詳細については、99 ページの *モジュールの保持*、99 ページの *シンボルおよびセクションの保持*を参照してください。

コンパイル時に、データおよび関数は、さまざまなセクションに配置されます。リンク時、リンカの最も重要な機能の 1 つは、アプリケーションで使用されるさまざまなセクションにアドレスを割り当てることです。

IAR ビルドツールには、多くのセクション名が事前に定義されています。各セクションの詳細については、「*セクションリファレンス*」を参照してください。

ブロックを使用して配置するためにセクションをいっしょにグループ化できます。446 ページの *define block ディレクティブ*を参照してください。

リンクプロセスの詳細

IAR コンパイラおよびアセンブラにより生成されるオブジェクトファイルおよびライブラリの再配置可能モジュールは、そのまま実行することはできません。これらが実行可能なアプリケーションとなるには、*リンク*される必要があります。

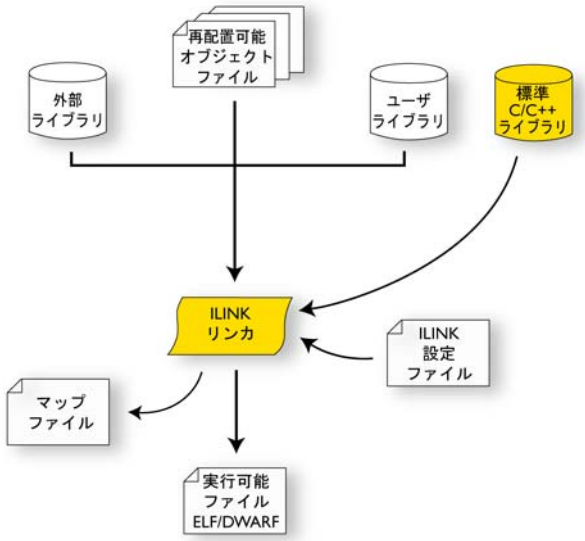
注 : 別のベンダのツールセットで生成されたモジュールも同様にビルドに含めることができます。ただし、モジュールが AEABI (ARM Embedded Application Binary Interface) 準拠の場合に限ります。ただし、AEABI 準拠でない場合、同じベンダのコンパイラユーティリティライブラリが必要なので注意してください。

リンカはリンクプロセスに使用されます。通常、以下の手順を実行します (一部の手順は、コマンドラインオプションやリンカ設定ファイルのディレクティブによって無効化できます)。

- アプリケーションに含めるモジュールを判別する。オブジェクトファイルで提供されるモジュールは、常に含まれます。ライブラリファイルのモジュールは、インクルードされるモジュールから参照されるグローバルシンボルの定義を与えるものだけが含まれます。
- 使用する標準ライブラリファイルを選択する。選択は、インクルードするモジュールの属性に基づいて行われます。これらのライブラリは、依然として解決されていないすべての未定義シンボルを充足するために使用されます。

- 複数の定義を持つシンボルを処理します。Weak ではない定義が複数あると、エラーが出力されます。それ以外の場合、いずれかひとつの定義が選択され（weak でない定義がある場合はそれが選択されます）、その他は無効化されます。Weak 定義は通常、インライン関数およびテンプレート関数に使用されます。ライブラリモジュールからの weak でない定義のいくつかをオーバライドする必要がある場合は、ライブラリモジュールがインクルードされていないことを確認してください（通常は、そのライブラリモジュールでアプリケーションが使用するすべてのシンボルについて、代替の定義を指定します）。
- 追加したモジュールのうちアプリケーションに含めるセクション/セクションフラグメントを判別する。アプリケーションで実際に必要なセクション / セクションフラグメントのみが含まれます。必要なセクション / セクションフラグメントを判別する方法は、いくつかあります。たとえば、`__root` オブジェクト属性、`#pragma required` ディレクティブ、`keep` リンカディレクティブなどが使用できます。セクションが重複する場合は、1 つのみ含まれます。
- 必要に応じて、RAM 内の初期化変数およびコードの初期化を実行する。`initialize` ディレクティブを使用すると、リンカによって追加のセクションが作成され、ROM から RAM へのコピーが可能になります。コピーによって初期化される各セクションは、2 つのセクションに分割され、1 つが ROM パート、もう 1 つが RAM パートに使用されます。手動の初期化を使用しない場合、初期化を実行するための起動コードもリンカで作成されます。
- *リンカ設定ファイル*のセクション配置ディレクティブに従って、各セクションの配置場所を判別する。コピーによって初期化されるセクションは、配置ディレクティブに照らして ROM パート用と RAM パート用の 2 か所あり、それぞれ異なる属性を持ちます。配置の過程において、リンカは、コード参照をその宛先まで進めるためや CPU モードを切り替えるために必要なベニアの追加も行います。
- 実行可能イメージおよび提供されたすべてのデバッグ情報を含む絶対ファイルを生成する。再配置可能な入力ファイルの必要な各セクションの内容は、そのファイルおよびセクションの配置時に決定されたアドレスで提供された再配置情報を使用して計算されます。このプロセスによって、特定セクションの要件の一部が満たされないと、1 つまたは複数の再配置エラーとなることがあります。たとえば、配置によって PC 関連のジャンプ命令の目的地のアドレスが、その範囲外となる場合などです。
- セクション配置の結果、各グローバルシンボルのアドレス、各モジュールおよびライブラリ用のメモリ使用量のサマリをリストするマップファイルを生成する（オプション）。

以下の図は、リンク処理を示しています。



リンク中、ILINK は、エラーメッセージおよびログメッセージを `stdout` および `stderr` に生成することがあります。ログメッセージは、アプリケーションがリンクされた理由を理解するときに役に立ちます。たとえば、モジュールまたはセクション（あるいはセクションフラグメント）が含まれた理由などです。

注 : ELF オブジェクトファイルの実際の内容を確認するには、`ielfdumparm` を使用します。484 ページの *IAR ELF Dumper — ielfdump* を参照してください。

コードおよびデータの配置（リンカ設定ファイル）

メモリへのセクションの配置は、IAR ILINK リンカが行います。これは、*リンカ設定ファイル*を使用します。このファイルでは、ユーザが、ILINK が各セクションをどのように扱うか、および使用可能メモリにセクションがどのように配置されるかを定義できます。

一般的なリンカ設定ファイルには、以下の定義が含まれます。

- 使用できるアクセス可能メモリ
- これらのメモリの使用領域
- 入力セクションの扱い方

- 作成されるセクション
- 使用できる領域にセクションを配置する方法

ファイルは、一連の宣言型ディレクティブで構成されます。つまり、リンクプロセスは、すべてのディレクティブで同時に制御されます。

該当設定ファイルを使用してコードをリビルドするだけで、同一ソースコードをさまざまな派生品で使用できます。

設定ファイルの簡単な例

以下のメモリ条件を持つ単純な 32 ビットのアーキテクチャを想定します。

- アドレス可能な 4GB のメモリがある。
- アドレス範囲 0x0000-0x10000 に ROM メモリがある。
- RAM メモリが 0x20000-0x30000 の範囲にある。
- スタックのアラインメントが 8 である。
- システム起動コードが固定アドレスにあること。

ここで想定するアーキテクチャの単純な構成ファイルは、以下のようになります。

```
/* 最大のアクセス可能なメモリ空間 */
define memory Mem with size = 4G;

/* アドレス空間のメモリ領域 */
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* スタックを定義 */
define block STACK with size = 0x1000, alignment = 8 { };

/* 初期化操作 */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* ゼロに初期化されるセクションを
                                     除き RW セクションを
                                     初期化する */

/* 固定アドレスにスタートアップコードを配置する */
place at start of ROM { readonly section .cstartup };

/* コードとデータを配置 */
place in ROM { readonly }; /* 定数 (.romdata) と
                              初期化：データ (.data_init) を ROM に
                              配置する */
place in RAM { readwrite,  /* .data、.bss、.noinit、STACK を */
               block STACK }; /* RAM に配置する */
```

この設定ファイルは、最大 4GB のアドレッシング可能なメモリ Mem を 1 つ定義しています。さらに、Mem において、それぞれ ROM および RAM という ROM 領域および RAM 領域を定義しています。各領域のサイズは 64KB です。

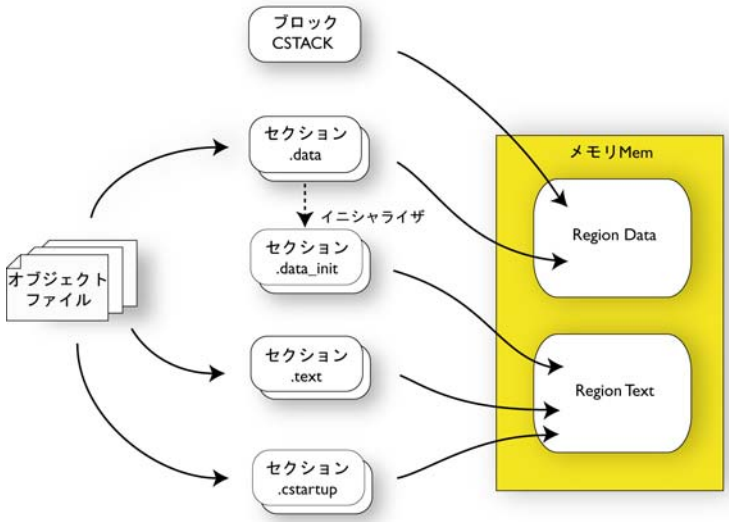
次に、このファイルは、アプリケーションスタックが常駐する、STACK という名前のサイズ 4KB の空のブロックを作成します。*ブロック*を作成するのが、配置やサイズなどを詳細に制御する基本的な方法です。この方法を使用してセクションをグループ化したり、この例にあるように、メモリエリアのサイズと配置を指定することもできます。

次に、設定ファイルは、変数、リード / ライト型 (readwrite) セクションの初期化方法を定義します。この例では、イニシャライザは ROM に配置され、アプリケーション起動時に RAM エリアにコピーされます。デフォルトでは、ILINK は、圧縮した方がいいと判断した場合はイニシャライザを圧縮します。

設定ファイルの最後のパートは、使用可能な領域に対してすべてのセクションを実際にどのように配置するかを定義しています。まず、起動コード（リードオンリー (readonly) セクション .cstartup にあるよう定義されているもの）が、ROM 領域、つまりアドレス 0x10000 に配置されます。{} 内は、*セクション選択*と呼ばれ、ディレクティブが適用されるセクションを選択します。次に、リードオンリーセクションの残りの部分が、ROM 領域に配置されます。選択セクション { readonly section .cstartup } は、さらに一般的なセクション選択 { readonly } より優先されます。

さらに、リード / ライト (readwrite) セクションおよび STACK ブロックは、RAM 領域に配置されます。

以下の図は、アプリケーションがメモリにどのように配置されるかを示します。



これらの標準ディレクティブのほか、設定ファイルは、以下の方法を定義するディレクティブを含むことができます。

- いくつかの方法でアドレスが可能なメモリのマッピング
- 条件ディレクティブの扱い
- 値がアプリケーションで使用できるシンボルの作成
- ディレクティブが適用されるセクションのさらに詳細な選択
- コードおよびデータのさらに詳細な初期化

リンカ設定ファイルのカスタマイズの詳細および例については、*アプリケーションのリンク*を参照してください。

リンカ設定ファイルの詳細は、*リンカ設定ファイル*を参照してください。

システム起動時の初期化

標準 C では、固定メモリアドレスに割り当てられるすべての静的変数は、アプリケーション起動時にランタイムシステムにより既知の値に初期化される必要があります。この値は、変数に明示的に割り当てられた値か、値が指定されていない場合はゼロにクリアされます。コンパイラには、この規則の例

外があります。たとえば、__no_init 宣言される変数です。これはまったく初期化されません。

コンパイラは、変数初期化の各型に対して、特定の型のセクションを生成します。

宣言データのカテゴリ	ソース	セクション型	セクション名	セクションの内容
ゼロで初期化されるデータ	`int i;`	リード / ライトデータ、ゼロ初期化	.bss	なし
ゼロで初期化されるデータ	`int i = 0;`	リード / ライトデータ、ゼロ初期化	.bss	なし
初期化されるデータ（ゼロ以外）	`int i = 6;`	リード / ライトデータ	.data	イニシャライザ
非初期化データ	`__no_init int i;`	リード / ライトデータ、ゼロ初期化	.noinit	なし
定数	`const int i = 6;`	リードオンリーデータ	.rodata	定数
コード	`__ramfunc void myfunc() {}`	リード / ライトコード	.textrw	コード

表 3: 初期化データを保持するセクション

注 : 静的変数をクラスタ化すると、ゼロで初期化される変数と初期化されるデータが一緒に .data にグループ化される可能性があります。定数テーブルから定数のアドレスをロードしなくていいように、コンパイラは定数を .text セクションに配置するよう決定できます。

サポートされているすべてのセクションについては、*セクションリファレンス*を参照してください。

初期化プロセス

データの初期化は、ILINK およびシステム起動コードで扱われます。

変数の初期化を設定するには、以下のことを考慮する必要があります。

- ゼロ初期化されるセクションは、ILINK により自動的に扱われる。これらは RAM にのみ配置されます

- 初期化されるセクションは、ゼロ初期化されるセクションを除き、initialize ディレクティブにリストされていなければならない

  通常、リンク時に、初期化されるセクションは、2 つのセクションに分割される。ここで、元の初期化されるセクションはその名前を保持します。その内容は新しいイニシャライザセクションに配置され、サフィックス _init が付いた元の名前を取得します。配置ディレクティブにより、イニシャライザは ROM に、初期化されるセクションは RAM に配置されます。この最も一般的な例は、.data セクションです。このセクションは、リンカにより .data および .data_init に分割されます。
- 定数を含むセクションは初期化されない。このようなセクションは、フラッシュ /ROM にのみ配置されます
- __no_init により宣言された変数を保持するセクションは、初期化してはならないため、do not initialize ディレクティブにリストしてください。このようなセクションは RAM に配置されます

リンカ設定ファイルでは、次のように定義されています。

```
/* 初期化操作 */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* ゼロに初期化されるセクションを
                                     除き RW セクションを
                                     初期化する */

/* 固定アドレスにスタートアップコードを配置する */
place at start of ROM { readonly section .cstartup };

/* コードとデータを配置 */
place in ROM { readonly }; /* 定数 (.romdata) と
                              初期化：データ (.data_init) を ROM に
                              配置する */
place in RAM { readwrite,  /* .data、.bss、.noinit、STACK を */
               block STACK }; /* RAM に配置する                */
```

注 : 圧縮されたイニシャライザが使用される場合（448 ページの *initialize ディレクティブ*を参照）、内容（イニシャライザ）のセクション（つまり、サフィックス _init の付いたセクション）はマップファイルで個別のセクションとしてはリストされません。その代わりに、これらは「イニシャライザバイト」の集合に組込まれます。内容のセクションを通常のようにリンカ設定ファイルに配置できますが、こうすることでイニシャライザバイトの集合の配置（およびその番号）に影響が出ます。

初期化の設定方法の詳細および例については、95 ページの *リンクについて*を参照してください。

C++ 動的初期化

コンパイラは、C++ の動的初期化を実行するためのサブルーチンポインタを、ELF セクションタイプ SHT_PREINIT_ARRAY および SHT_INIT_ARRAY のセクションに配置します。デフォルトでは、リンカはこれらをリンカが作成したブロックに配置し、セクションタイプ SHT_PREINIT_ARRAY のセクションがすべてタイプ SHT_INIT_ARRAY の前に配置されるようにします。このようなセクションが含まれる場合、ルーチンを呼出すコードも含まれることになります。

リンカが作成したブロックは、リンカ設定に preinit_array および init_array セクションタイプのセクションセレクタのパターンが含まれない場合にのみ生成されます。リンカにより作成されたブロックの効果は、リンカ設定ファイルに以下が含まれる場合と同じようになります。

```
define block SHT$$PREINIT_ARRAY { preinit_array };
define block SHT$$INIT_ARRAY { init_array };
define block CPP_INIT with fixed order { block
                                         SHT$$PREINIT_ARRAY,
                                         block SHT$$INIT_ARRAY };
```

これをリンカ設定ファイルに入れる場合、セクション配置ディレクティブのいずれかで CPP_INIT ブロックも記述する必要があります。リンカにより作成されたブロックをどこに配置するか選択する場合は、".init_array" という名前のセクションセレクタを使用できます。

455 ページの *セクションセレクタ*を参照してください。

スタック使用量解析

適切な状況下では、リンカはそれぞれのコールグラフルート（別の関数から呼出されない各関数）の最大スタック使用を正確に計算します。

スタック使用量解析を有効にすると、リンカマップファイルにスタック使用の項目が追加され、各コールグラフルートについてスタック深さの最大値になる特定のコールチェーンが一覧表示されます。304 ページの *--enable_stack_usage* を参照してください。

これは、アプリケーション内の各関数に正確なスタック使用情報がある場合にのみ正確となります。

一般的には、コンパイラは各 C 関数についてこの情報を生成しますが、間接的な呼出し（関数ポインタを使用した呼出し）がアプリケーション内にある場合、各呼出しコール関数から呼出しが可能な関数のリストを提供する必要があります。これを行うには、ソースファイル内でプラグマディレクティブを使用するか、リンク処理の際に別のスタック使用解析制御ファイルを使用します。

スタック使用解析制御ファイルを使用する場合、スタック使用情報を持たないモジュール内の関数のスタック使用情報も提供できます。321 ページの *--stack_usage_control* を参照してください。

リンカが計算したスタック使用量が割り当てたスタックエリアを超えないか確認するために、リンカ設定ファイルで `check that` ディレクティブを使用できます。459 ページの *check that ディレクティブ*を参照してください。

制限

スタック使用情報が不足していたり正しくないことのほか、解析が不正確となる他の原因もあります。

- リンカはスタック使用情報が欠けているオブジェクトモジュールの関数を、常にすべては識別できない場合があります。特に、アセンブラに記述されたオブジェクトモジュールまたは IAR ツールでないもので生成されたオブジェクトモジュールでこれが問題になる場合があります。
- フレームサイズの変更や関数呼出しの実行にインラインアセンブラを使用する場合、これは解析には反映されません。
- 他のソース（プロセッサ、オペレーティングシステムなど）で消費される余分なエリア。
- 例外を使用する C++ ソースコードがサポートされていない。
- ソフトウェア割込みのような他の形式の関数呼出しを使用する場合、それらはコールグラフには反映されません。
- 複数ファイルのコンパイル (`--mfc`) が、関係するファイルでモジュールローカルの関数のプロパティを指定するときに、スタック使用量制御ファイルの使用に干渉することがあります。

スタック使用量解析は一番悪い場合の結果を生成します。実際にはプログラムは設計でまたは偶然に最大コールチェーンで終わっていない場合があります。特に、C++ の呼び出す仮想関数呼び出しの宛先の設定には、関連した関数の処理が含まれることがあります。実際にコードのそのポイントから呼び出されます。

スタック使用量解析は、実際の測定に対する補足でしかありません。結果が重要であれば、解析の結果を個別に検証する必要があります。

スタック使用解析制御ファイル

スタック使用解析制御ファイルには、スタック使用解析制御ディレクティブが含まれます。

スタック使用解析制御ファイルを使用して、次のことが可能です。

- スタック使用解析制御ディレクティブ `function` を使用することで、完全なスタック使用情報（コールグラフルートカテゴリ、スタック使用、起こり得る呼出し）を指定します。
- スタック使用解析制御ディレクティブ `exclude` を使用して、スタック使用量解析から特定の関数を除外します。
- スタック使用解析制御ディレクティブ `possible calls` を使用して、関数内の間接的な呼出しの宛先を指定します。
- スタック使用解析制御ディレクティブ `call graph root` を使用して、オプションのコールグラフルートカテゴリなど、関数がコールグラフルートであるように指定します。
- 再帰ネストの再帰の最も深い値を指定します（少なくとも 1 つの共通ノードを持つコールグラフのサイクルのセット）。
- スタック使用量制御ファイルでスタック使用量情報を提供したモジュールにより参照されている、記述されていない関数についてのワーニングを選択して非表示にします。

コンパイラによって割込み関数がコールグラフルートとして指定されていない場合、手動でそのように指定する必要があります。これを行うには、ソースコードで `#pragma call_graph_root` ディレクティブを使用するか、次のような単純なスタック使用解析制御ファイルを使用します。

```
call graph root [interrupt]: Irq1Handler, Irq2Handler;
```

詳細については、361 ページの *call_graph_root* および*スタック使用解析制御ファイル*の章を参照してください。

ソースの注釈

スタック使用量制御ファイルで可能な呼出しを指定する代わりに、ソースコードを注釈を付けることができます。

C ファイルでは、間接的な呼出しの地点で、`#pragma calls` ディレクティブを使用して、その呼出しの宛先となり得るものをリストできます。

また、関数の定義で、`#pragma call_graph_root` ディレクティブを使用すると、それがコールグラフルートであることを指定できます。

ワーニングが発行される状況

スタック使用量解析がリンカで有効な場合、次の状況ではワーニングが生成されます。

- スタック使用情報を持たない関数が少なくとも 1 つある。
- アプリケーション内に少なくとも 1 つの間接的な呼出し元があり、呼出される可能性がある関数のリストが提供されていない。
- 既知の間接的な呼出しはないが、コールグラフルートで認識されていない呼出されていない関数が少なくとも 1 つある。
- アプリケーションには再帰（コールグラフ内のサイクル）が含まれ、再帰の最大深度が指定されていないか、またはリンカが信頼できるスタック使用量の見積もりを計算できない形式になっている。
- コールグラフルートとして制限された関数への呼出しがある。
- スタック使用量制御ファイルを使用して、スタック使用量情報を持たないモジュール内の関数に使用量情報を提供してる。また、そのモジュールが参照している関数で、スタック使用量制御ファイルで呼出されるように記述されていないものがある。

マップファイルの内容

スタック使用量の解析が有効な場合、リンカマップファイルにはスタック使用量の項目が含まれ、それに各コールグラフルートカテゴリにスタック使用量のサマリも含まれます。さらに、各コールグラフルートについて、最大スタック深度となるコールチェーンがリストされます。以下は、マップファイルにおけるスタック使用の章の一例です。

```
****************************************************************
  *** STACK USAGE
  ***

  Call Graph Root Category  Max Use  Total Use
  ------------------------  -------  ---------
  interrupt                     104        136
  Program entry                 168        168

Program entry
  "__iar_program_start": 0x000085ac
  Maximum call chain                    168 bytes
```

```
    "__iar_program_start"                  0
    "__cmain"                              0
    "main"                                 8
    "printf"                              24
    "_PrintfTiny"                         56
    "_Prout"                              16
    "putchar"                             16
    "__write"                              0
    "__dwrite"                             0
    "__iar_sh_stdout"                     24
    "__iar_get_ttio"                      24
    "__iar_lookup_ttioh"                   0

interrupt
  "FaultHandler": 0x00008434

  Maximum call chain                      32 bytes

    "FaultHandler"                        32

interrupt
  "IRQHandler": 0x00008424

  Maximum call chain                     104 bytes

    "IRQHandler"                          24
    "do_something" in suexample.o [1]     80
```

サマリには、各カテゴリで最も深いコールチェーンの深度と、そのカテゴリで最も深いコールチェーンの深さの合計が含まれます。

この場合、プログラムエントリ (__iar_program_start) のスタック深さの最大値は 168 バイトで、システムライブラリ printf 関数内で発生します。public 関数は名前順にリストされ、モジュールローカル関数にはモジュール名（前述の do_something など）も含まれます。

スタックに十分な大きさがあるかどうかの確認

リンカ設定ファイルで check that ディレクティブを使用して、スタックが十分大きいかどうかをチェックできます。

たとえば、MY_STACK というスタックブロックがあるとします。

```
check that size(block MY_STACK) >= maxstack("Program entry")
                                 + totalstack("interrupt") + 100;
```

リンクする際、式が偽（ゼロ）ならばリンカはエラーを出力します。この例では、168（program entry の最大スタック使用量)、136（"interrupt" カテゴリ

の最大スタック使用量合計)、100（安全マージン）の合計が MY_STACK ブロックのサイズを超えれば、エラーになります。

コールグラフログ

スタック使用量解析の結果を理解しやすくするため、コールグラフのテキストによる単純な表現を生成するログ出力オプションがあります (--log call_graph)。

出力の例：

```
Program entry:
0 __iar_program_start [168]
  0 __cmain [168]
    0 __iar_data_init3 [16]
      8 __iar_zero_init3 [8]
        16 - [0]
      8 __iar_copy_init3 [8]
        16 - [0]
    0 __low_level_init [0]
    0 main [168]
      8 printf [160]
        32 _PrintfTiny [136]
          88 _Prout [80]
            104 putchar [64]
              120 __write [48]
                120 __dwrite [48]
                  120 __iar_sh_stdout [48]
                    144 __iar_get_ttio [24]
                      168 __iar_lookup_ttioh [0]
                  120 __iar_sh_write [24]
                    144 - [0]
          88 __aeabi_uidiv [0]
            88 __aeabi_idiv0 [0]
          88 strlen [0]
    0 exit [8]
      0 _exit [8]
        0 __exit [8]
          0 __iar_close_ttio [8]
            8 __iar_lookup_ttioh [0] ***
    0 __exit [8] ***
```

各行には次の情報が含まれます。

- 関数の呼出しポイントにおけるスタック使用量
- 関数名、および関数呼出しのないポイントにおける関数（通常は leaf 関数）の使用量を示す、1 つの '-'

- そのポイントから最も深いコールチェーンのスタック使用量。そのような値が計算できないときは、代わりに "[---]" が出力されます。"***" は、すでに表示された関数を示します。

コールグラフ XML 出力

また、リンカは XML フォーマットでコールグラフファイルを生成します。このファイルには、アプリケーション内の各関数ごとに 1 つのノードと、その関数に特定のスタック使用および呼出し情報が含まれます。処理後に使用するツールのための入力を意図しており、特に可読というわけではありません。

使用される XML フォーマットの詳細については、製品のインストールに含まれる `callGraph.txt` ファイルを参照してください。

アプリケーションのリンク

- リンクについて
- トラブルシューティングについてのヒント

リンクについて

アプリケーションをリンクするには、ILINK で必要な構成を設定する必要があります。通常は以下の点を考慮する必要があります。

- リンカ設定ファイルの選択
- 独自のメモリエリアの定義
- セクションの配置
- RAM の空間の予約
- モジュールの保持
- シンボルおよびセクションの保持
- アプリケーションの起動
- スタックメモリの設定
- ヒープメモリの設定
- atexit 制限の設定
- デフォルト初期化の変更
- ILINK とアプリケーション間の相互処理
- 標準ライブラリの処理
- ELF/DWARF 以外の出力フォーマットの生成

リンカ設定ファイルの選択

`config` ディレクトリには、リンカ設定ファイル用の既成のテンプレートが 2 つあります。

- `generic.icf`: Cortex-M コア以外のすべてのコア向け
- `generic_cortex.icf`: すべての Cortex-M コア向け

これらのファイルには、ILINK で必要な情報が含まれています。この付属の設定ファイルは、ターゲットシステムメモリマップに合わせて各領域の開始および終了アドレスをカスタマイズするだけで簡単に使用できます。たとえば、アプリケーションが追加外部 RAM を使用する場合は、外部 RAM のメモリエリアについての情報を追加する必要もあります。

一部のデバイスでは、デバイス固有の設定ファイルが自動的に選択されます。

リンカ設定ファイルを編集するには、IDE のエディタ、またはその他の適切なエディタを使用します。また、**［プロジェクト］> ［オプション］> ［リンカ］**を選択し、**［設定］**ページの**［編集］**ボタンをクリックして、専用のリンカ設定ファイルエディタを開きます。

元のテンプレートファイルは変更しないでください。作業ディレクトリにコピーを作成し、そのコピーを修正することをお勧めします。IDE でリンカ設定ファイルエディタを使用する場合、IDE によりコピーが作成されます。

IDE 内の各プロジェクトは、リンカ設定ファイルへの参照を 1 つだけ持つ必要があります。このファイルは編集可能ですが、すべてのプロジェクトの大半では、**［プロジェクト］> ［オプション］> ［リンカ］> ［設定］**から重要パラメータを設定するだけで十分です。

独自のメモリエリアの定義

選択したデフォルトの設定ファイルには、ROM および RAM 領域が事前に定義されています。以下の例は、この章で示されるすべての詳細な例の基本例として使用されます。

```
/* アクセス可能な空間を定義する */
define memory Mem with size = 4G;

/* 0 番地から始まる 64kB の大きさの ROM という名前の領域を定義する */
define region ROM = Mem:[from 0 size 0x10000];

/* 0x20000 番地から始まる 64kB の大きさの ROM という名前の領域を定義する
*/
define region RAM = Mem:[from 0x20000 size 0x10000];
```

各領域定義は、実際のハードウェアに合わせて調整する必要があります。

リンク後のコードおよびデータがどのくらいのメモリを占有するかを確認するには、マップファイルのメモリ概要（コマンドラインオプション `--map`）を参照してください。

領域の追加

領域を追加するには、`define region` ディレクティブを使用します。以下に例を示します。

```
/* 0x80000 番地から始まる 128kB の大きさの、2 番めの領域を定義する */
define region ROM2 = Mem:[from 0x80000 size 0x20000];
```

異なるエリアを 1 つの領域にマージする

領域が複数のエリアで構成されている場合、領域式を使用して、異なるエリアを 1 つの領域にマージできます。以下に例を示します。

```
/* 2 番目の ROM 領域が 2 つの領域を持つように定義する。2 つのエリアから成る
2 番めの領域を定義する。1 つめは 0x80000 番地から始まり 128kB の大きさ、
2 つめは 0xc0000 番地から始まり 32kB の大きさ */
define region ROM2 = Mem:[from 0x80000 size 0x20000]
                   | Mem:[from 0xC0000 size 0x08000];
```

以下の例も同じです。

```
define region ROM2 = Mem:[from 0x80000 to 0xC7FFF]
                     -Mem:[from 0xA0000 to 0xBFFFF];
```

セクションの配置

選択したデフォルト設定ファイルでは、事前に定義されているすべてのセクションがメモリに配置されますが、場合によっては、これを修正する必要があります。たとえば、定数シンボルを保持するセクションをデフォルトの場所ではなく CONSTANT 領域に配置する場合です。この場合、place in ディレクティブを使用します。以下に例を示します。

```
/* ROM 領域に readonly の内容を配置する */
place in ROM {readonly};

/* constant 領域に定数シンボルを配置する */
place in CONSTANT {readonly section .rodata};
```

注 : IAR ビルドツールで使用されるセクションを、その内容を異なる方法で参照するメモリに配置しようとすると、エラーが発生します。

リンク後に配置ディレクティブを使用する場合、マップファイルで配置の概要（コマンドラインオプション --map）を確認してください。

セクションをメモリの特定のアドレスに配置する

セクションをメモリの特定のアドレスに配置するには、place at ディレクティブを使用します。以下に例を示します。

```
/* .vectors セクションを 0 番地に配置する */
place at address Mem:0x0 {readonly section .vectors};
```

セクションを領域の開始または終了位置に配置する

セクションを領域の開始または終了位置に配置する方法は、特定のアドレスに配置する方法と似ています。以下に例を示します。

```
/* .vecotors セクションを ROM の先頭に配置する */
place at start of ROM {readonly section .vectors};
```

独自のセクションの宣言および配置

IAR ビルドツールで使用されるセクションのほかに、コードまたはデータの固有な部分を保持する新しいセクションを宣言するには、コンパイラおよびアセンブラのメカニズムを使用します。以下に例を示します。

```
/* セクションを作る（アセンブラ）*/
const short MyVariable @ "MYOWNSECTION" = 0xF0F0;
```

以下はアセンブラ言語の場合の例です。

```
            name    createSection
            section MYOWNSECTION:CONST  ; セクションを作成して
                                        ; 定数バイトを
            dc16     0xF0F0             ; 入力
            end
```

新しいセクションを配置するには、オリジナルの `place in ROM {readonly};` ディレクティブをそのまま使用します。

ただし、セクション `MyOwnSection` を明示的に配置するには、`place in` ディレクティブでリンカ設定ファイルを更新します。以下に例を示します。

```
/* MyOwnSection セクションを ROM 領域に配置する */
place in ROM {readonly section MyOwnSection};
```

RAM の空間の予約

多くの場合、アプリケーションで、たとえばヒープやスタックなど、一時的な記憶領域として使用するために、空の初期化されていないメモリエリアが必要です。これは、リンク時に行うのが最も簡単です。このようなメモリエリアを作成するには、サイズを指定したブロックを作成し、これをメモリに配置する必要があります。

リンカ設定ファイルでは、次のように定義されています。

```
define block TempStorage with size = 0x1000, alignment = 4 { };
place in RAM { block TempStorage };
```

割り当てられるメモリの開始位置をアプリケーションから取得するには、ソースコードは以下のようになります。

```
/* 一時的な記憶領域としてセクションを定義 */
#pragma section = "TempStorage"
char *GetTempStorageStartAddress()
{
  /* セクション TempStorage の開始アドレスをリターン */
  return __section_begin("TempStorage");
}
```

モジュールの保持

モジュールがオブジェクトファイルとしてリンクされている場合、これは常に保持されます。つまり、モジュールは、リンクされたアプリケーションに含まれます。ただし、モジュールがライブラリの一部の場合、モジュールが含まれるのは、アプリケーションの他の部分からシンボルで参照されている場合のみです。これは、ライブラリモジュールにルートシンボルが含まれている場合でも同様です。このようなライブラリモジュールが常に確実に含まれるようにするには、iarchive を使用してライブラリからモジュールを抽出します（479 ページの *IAR アーカイブツール — iarchive* を参照）。

含まれるモジュールおよび除外されるモジュールについては、ログファイル（コマンドラインオプション `--log modules`）を確認してください。

モジュールの詳細については、78 ページの *モジュールおよびセクション*を参照してください。

シンボルおよびセクションの保持

デフォルトでは、ILINK は、アプリケーションで必要ない任意のセクション、セクションフラグメント、グローバルシンボルを削除します。必要がないと思われるシンボル、実際にはそのシンボルが定義されているセクションフラグメントを保持するには、C/C++ またはアセンブラソースコードでシンボルのルート属性を使用するか、ILINK オプション `--keep` を使用します。属性名またはオブジェクト名に基づいてセクションを保持するには、リンカ設定ファイルでディレクティブ `keep` を使用します。

ILINK がセクションおよびセクションフラグメントを除外しないようにするには、それぞれにコマンドラインオプション `--no_remove` または `--no_fragments` を使用します。

含まれるおよび除外されるシンボルとセクションについては、ログファイル（コマンドラインオプション `--log sections`）を確認してください。

シンボルとセクションを保持するリンク手順の詳細については、49 ページの *リンク処理*を参照してください。

アプリケーションの起動

デフォルトでは、アプリケーションが実行を開始する位置は __iar_program_start ラベルで定義されています。これは、cstartup.s ファイルの開始位置に定義されています。このラベルは、ELF を介して、使用される任意のデバッガにも送られます。

アプリケーションの開始位置を別のラベルに変更するには、ILINK オプション --entry を使用します（305 ページの *--entry* を参照）。

スタックメモリの設定

CSTACK ブロックのサイズは、リンカ設定ファイルで定義されています。割り当てられるメモリの容量を変更するには、CSTACK のブロック定義を変更します。

```
define block CSTACK with size = 0x2000, alignment = 8{ };
```

アプリケーションで必要なサイズを指定してください。

スタックの情報については、「198 ページの *スタックについて*」を参照してください。

ヒープメモリの設定

ヒープのサイズは、リンカ設定ファイルでブロックとして定義されます。

```
define block HEAP with size = 0x1000, alignment = 8{ };
place in RAM {block HEAP};
```

アプリケーションで必要なサイズを指定してください。ヒープを使用する場合は、少なくとも 50 バイトを割り当てる必要があります。

ATEXIT 制限の設定

デフォルトでは、atexit 関数は、アプリケーションから最大で 32 回呼出すことができます。この回数を増加または減少するには、設定ファイルに行を追加します。たとえば、10 回の呼出しを保持する空間を予約するには、以下のように記述します。

```
define symbol __iar_maximum_atexit_calls = 10;
```

デフォルト初期化の変更

デフォルトでは、メモリの初期化は、アプリケーション起動時に実行されます。ILINK は、初期化プロセスを設定し、最適なパッキング方法を選択します。デフォルトの初期化プロセスがアプリケーションに適していないため、初期化プロセスをより正確に制御する必要がある場合、以下の方法を使用できます。

- 初期化の無効化
- パッキングアルゴリズムを選択する
- 手動で初期化する
- コードを初期化する（ROM から RAM にコピーする）

実行された初期化については、ログファイル（コマンドラインオプション `--log initialization`）を確認してください。

初期化の無効化

一部またはすべてのセクションについて、コピーによる初期化をリンカに設定しない場合は、これらのセクションが `initialize by copy` ディレクティブのパターンに一致しないようにしてください（または、`except` 句を使用して、一致しないように除外します）。コピーによる初期化をまったく必要としない場合、`initialize by copy` ディレクティブを完全に省略できます。

これは、何らかのメカニズムによって、アプリケーションが起動する前に、アプリケーションまたは変数だけを RAM にロードする場合に役立ちます。

パッキングアルゴリズムの選択

デフォルトのパッキングアルゴリズムをオーバライドするには、たとえば、以下のように記述します。

```
initialize by copy with packing = lzw { readwrite };
```

使用可能なそのパッキングアルゴリズムの詳細については、448 ページの *initialize ディレクティブ*を参照してください。

手動で初期化する

通常、`initialize by copy` ディレクティブは、リンカに、アプリケーションの起動時にイニシャライザのあるセクションをコピーすることによる初期化を指示するときに使用します。リンカは、各セクションに対して初期化セクションを論理的に作成し、そのセクションの内容を保持して、元のセクションをイニシャライザを持たないセクションに変換することにより行います。続いて、リンカはアプリケーション起動時に初期化が実行されるように、初期化テーブルに テーブルの要素を追加します。`initialize manually` を使用すると、テーブルの要素の追加を無効にし、要素がいつどのようにコピーされるかを制御することができます。これはオーバレイのほかに、その他多くの場合にも便利です。

イニシャライザを持たないセクション（ゼロ初期化されたセクション）の場合、状況は逆になります。`do not initialize` ディレクティブで記述されたものを除いて、リンカはアプリケーション起動時にこうしたすべてのセクションのゼロ初期化を行います。通常は `.noinit` セクションのみが `do not`

initializeディレクティブで指定されますが、必要なだけゼロ初期化されたセクションを追加して、いつどのようにセクションが初期化されるかを制御することができます。

暗黙的なブロックを使用した単純なコピーの例

MYSECTIONに初期化された変数があるとします。リンカ設定ファイルに次のディレクティブを追加します。

```
initialize manually { section MYSECTION };
```

このソースコードサンプルを使用して、次のセクションを初期化できます。

```
#pragma section = "MYSECTION"
#pragma section = "MYSECTION_init"

void DoInit()
{
  char * from = __section_begin("MYSECTION_init");
  char * to   = __section_begin("MYSECTION");
  memcpy(to, from, __section_size("MYSECTION"));
}
```

このソースコードは、__section_begin（および関連の演算子）をセクション名に使用した場合に、これらのセクションに対して、リンカによって合成のブロックが作成されることを活用しています。

明示的なブロックの例

特定のセクションの変数を手動で初期化しなければならない場合と異なり、特定のライブラリからのすべての初期化済み変数を手動で初期化する場合を考えます。この場合、変数と内容の両方について明示的なブロックを作成する必要があります。以下のようにします。

```
initialize manually       { section .data      object mylib.a };
define block MYBLOCK      { section .data      object mylib.a };
define block MYBLOCK_init { section .data_init object mylib.a };
```

また、2つの新しいブロックをセクション配置ディレクティブを使用して配置する必要があります。MYBLOCKブロックをRAMに、MYBLOCK_initブロックをROMにそれぞれ配置します。

前述の例と同じソースコードを使用して、セクションを初期化できます。ただし、MYSECTIONの代わりにMYBLOCKを使用します。

オーバレイの例

これは、自動ブロック作成を活用した単純なオーバレイの例です。

```
initialize manually { section MYOVERLAY* };

define overlay MYOVERLAY { section MYOVERLAY1 };
define overlay MYOVERLAY { section MYOVERLAY2 };
```

また、RAM のどこかに overlay MYOVERLAY を配置する必要があります。コピーは以下のようになります。

```
#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1_init"
#pragma section = "MYOVERLAY2_init"

void SwitchToOverlay1()
{
  char * from = __section_begin("MYOVERLAY1_init");
  char * to   = __section_begin("MYOVERLAY");
  memcpy(to, from, __section_size("MYOVERLAY1_init"));
}

void SwitchToOverlay2()
{
  char * from = __section_begin("MYOVERLAY2_init");
  char * to   = __section_begin("MYOVERLAY");
  memcpy(to, from, __section_size("MYOVERLAY2_init"));
}
```

コードを初期化する（ROM から RAM にコピーする）

場合によっては、アプリケーションは、コードの一部をフラッシュ ROM から RAM にコピーします。アプリケーション起動時にこの処理が自動的に行われるようリンカに指示するか、101 ページの *手動で初期化する*の説明に従って後で自分ですることもできます。

initialize by copy ディレクティブでコピーされるコードセクションをリストする必要があります。最も簡単な方法は通常、適切な関数を特定のセクション（RAMCODE など）に配置して、section RAMCODE を initialize by copy ディレクティブに追加することです。次に例を示します。

```
initialize by copy { rw, section RAMCODE };
```

特定の場所に RAMCODE 関数を配置する必要がある場合は、配置ディレクティブでその関数を記述しなければなりません。そうしなければ、他のリード / ライトセクションとともに配置されます。

コピーの動作やタイミングを制御しなければならない場合は、代わりに initialize manually ディレクティブを使用してください。101 ページの *手動で初期化する*を参照してください。

フラッシュ /ROM にアクセスしないで関数を実行する必要がある場合、コンパイル時に __ramfunc キーワードを使用できます。66 ページの *RAM での実行*を参照してください。

すべてのコードを RAM から実行

プログラム起動時にアプリケーション全体を ROM から RAM にコピーする場合は、たとえば、initilize by copy ディレクティブを使用して、以下のように実現できます。

```
initialize by copy { readonly, readwrite };
```

readwrite パターンは、静的に初期化されるすべての変数に一致し、これらが起動時に初期化されるように準備します。readonly パターンは、初期化に必要なコードおよびデータを除くすべてのリードオンリーコードおよびデータに対して同様に機能します。

必要な ROM エリアを小さくするには、利用可能なパッキングアルゴリズムのいずれかでデータを圧縮する方法が有効です。以下に例を示します。

```
initialize by copy with packing = lzw { readonly, readwrite };
```

使用可能なその圧縮アルゴリズムの詳細については、448 ページの *initialize ディレクティブ*を参照してください。

関数 __low_level_init が存在する場合、この関数は初期化の前に呼出されるため、この関数およびこの関数を必要とするものはすべて、ROM から RAM にコピーされません。特定の状況（たとえば、起動後に ROM の内容がプログラムで使用できなくなる場合など）においては、起動中およびコードの残りの部分での同じ関数の使用を避ける必要があります。

コピーされる必要のないものがあれば、except 句に入れます。たとえば、割込みベクタテーブルなどに適用できます。

また、RAM へのコピーから C++ 動的初期化テーブルを除外することが推奨されます。通常、このテーブルは、1 度だけ読み込まれ、再度参照されることはないためです。たとえば、以下のように指定します。

```
initialize by copy { readonly, readwrite }
            except { section .intvec,       /* 割込みテーブルを
                                               コピーしない */
                     section .init_array }; /* C++ init テーブルを
                                               コピーしない */
```

ILINK とアプリケーション間の相互処理

ILINK は、アプリケーションの制御に使用できるシンボルを定義するコマンドラインオプション `--config_def` および `--define_symbol` を提供しています。また、リンカ設定ファイルで定義される連続するメモリエリアの開始および終了位置を表すシンボルを使用することもできます。詳細については、200 ページの *ツールとアプリケーション間の相互処理*を参照してください。

シンボルの参照を変更するには、ILINK コマンドラインオプション `--redirect` を使用してください。これは、たとえば、実装されていない関数からスタブ関数に参照を変更する場合や、標準ライブラリ関数 `printf` および `scanf` の DLIB フォーマッタを選択する方法など、特定の関数においていくつかの異なる実装からいずれか 1 つを選択する場合に便利です。

コンパイラは、マングル化された名前を生成して、複雑な C/C++ シンボルを表します。アセンブラソースコードからこれらのシンボルに参照する場合、マングル化された名前を使用する必要があります。

すべてのグローバル（静的にリンクされた）シンボルのアドレスおよびサイズについては、マップファイルで空のリスト（コマンドラインオプション `--map`）を確認してください。

詳細については、200 ページの *ツールとアプリケーション間の相互処理*を参照してください。

標準ライブラリの処理

デフォルトでは、ILINK は、リンク中に含める標準ライブラリのバリアントを自動的に判別します。これは、各オブジェクトファイルおよび ILINK に渡されたライブラリオプションで使用できるランタイム属性に基づいて決定されます。

ライブラリの自動追加を無効にするには、オプション `--no_library_search` を使用します。この場合、ライブラリに含めるすべてのライブラリファイルを明示的に指定する必要があります。使用可能なライブラリファイルについては、112 ページの *ビルド済ライブラリの使用*を参照してください。

ELF/DWARF 以外の出力フォーマットの生成

ILINK は、ELF/DWARF フォーマットでのみ出力ファイルを生成できます。このフォーマットを PROM/ フラッシュのプログラムに適したフォーマットに変換するには、483 ページの *IAR ELF ツール — ielftool* を参照してください。

ベニア

ARM コアは、以下の 2 つの状況でベニアを使用する必要があります。

- ARM関数をThumbモードから呼出す場合、またはThumb関数をARMモードから呼出す場合、ベニアにより、マイクロプロセッサの状態が変更されます。呼出される関数は、インターワーク関数である必要があります（56 ページの*インターワーク*を参照）。コアが `BLX` 命令をサポートしている場合、モード変更のためのベニアは必要ありません。
- 通常は到達できない関数を呼出す場合、ベニアは、呼出しを確実に宛先に到達させるコードを導入します。

ベニアのコードは、任意の呼出し元および呼出し先関数間に挿入できます。そのため、`R12` レジスタは、アセンブラで記述された関数を含み、関数呼出し時のスクラッチレジスタとして扱う必要があります。これは、ジャンプにも適用されます。

詳細については、316 ページの *--no_veneers* を参照してください。

トラブルシューティングについてのヒント

ILINK は、以下のような、コードおよびデータの配置を正しく管理するために役に立ついくつかの機能を提供しています。

- リンク時のメッセージ（たとえば、再配置エラーが発生した場合など）
- ILINK で情報を `stdout` に記録させる `--log` オプション。この情報は、実行可能イメージが現在の状態になった理由を理解するときに役に立ちます（310 ページの *--log* を参照）
- ILINK でメモリマップファイルを生成する `--map` オプション。このファイルには、リンカ設定ファイルの結果が含まれます（311 ページの *--map* を参照）

再配置エラー

命令を正しく再配置できない場合、ILINK により、*再配置エラー*が発生します。このエラーは、ターゲットが範囲外にある命令または型が一致しない命令などで発生します。

ILINK で発生する再配置エラーの例を以下に示します。

```
Error[Lp002]:  relocation failed: out of range or illegal value
    Kind     :   R_XXX_YYY[0x1]
    Location :   0x40000448
                 "myfunc" + 0x2c
                 Module:  somecode.o
                 Section: 7 (.text)
                 Offset:  0x2c
    destination: 0x9000000c
                 "read"
                 Module:  read.o(iolib.a)
                 Section: 6 (.text)
                 Offset:  0x0
```

このメッセージエントリについて、以下の表で説明します。

メッセージエントリ	説明
Kind	失敗した再配置ディレクティブ。ディレクティブは、使用される命令により異なります。
Location	問題が発生した場所。詳細については、以下を参照してください。 • 16 進数およびオフセットを持つラベルとして表される命令アドレス。この例の場合、0x40000448 および "myfunc" + 0x2c です。 • モジュールおよびファイル。この例の場合、モジュールは somecode.o です。 • セクション番号およびセクション名。この例の場合、セクション番号は 7 で、名前は .text です。 • バイト数で指定されるセクション内のオフセット。この例のオフセットを以下に示します。0x2c
Destination	命令のターゲット。詳細については、以下を参照してください。 • 16 進数およびオフセットを持つラベルとして表される命令アドレス。この例の場合、0x9000000c および "read" です（オフセットなし）。 • モジュール、およびライブラリ（適切な場合）。この例の場合、モジュールは read.o、ライブラリは iolib.a です。 • セクション番号およびセクション名。この例の場合、セクション番号は 6 で、名前は .text です。 • バイト数で指定されるセクション内のオフセット。この例のオフセットを以下に示します。0x0

表 4: 再配置エラーの説明

考えられる解決方法

このケースでは、myfunc にある命令から __read までの長さが、分岐命令の飛び先の範囲を超えています。

考えられる解決方法としては、2 つの .text セクションをそれぞれの近くに配置するか、必要な距離に到達できる他の呼出し方法を使用します。また、参照元の関数が不正な飛び先を参照したために範囲エラーが発生した可能性もあります。

範囲エラーに対しては、その内容に応じた解決方法があります。通常は、上記の方法をベースに多少変更を加えた方法、すなわち、コードやセクション配置を変更することによって解決できます。

DLIB ランタイム環境

*DLIB ランタイム環境*では、アプリケーションの実行環境である DLIB ランタイム環境について説明します。特に、DLIB ランタイムライブラリと、使用するアプリケーション向けにそれを最適化する方法について解説します。

ランタイム環境の概要

ランタイム環境は、アプリケーションを実行するための環境です。ランタイム環境は、ターゲットハードウェア、ソフトウェア環境、アプリケーションコードによって異なります。

ランタイム環境の機能

*ランタイム環境*は、標準の C と標準テンプレートライブラリを含む C++ をサポートしています。*ランタイム環境*は、C/C++ の規格で定義された関数、およびライブラリインタフェースを定義するインクルードファイル（システムヘッダファイル）から構成されます。

提供されるランタイムライブラリには、（製品パッケージに応じて）ビルド済ライブラリとソースファイルの両方があり、これらはそれぞれ、製品のサブディレクトリ `arm¥lib` と `arm¥src¥lib` に格納されています。

ランタイム環境には、以下のようなターゲットシステム固有のサポートも含まれています。

- ハードウェア機能のサポート
  - *組込み*関数（割込みマスク処理用関数など）による低レベルプロセッサ処理へ直接アクセス
  - インクルードファイルでの周辺ユニットレジスタと割込みの定義
  - ベクタ浮動小数点 (VFP) コプロセッサ
- ランタイム環境サポート（起動 / 終了コード、一部のライブラリ関数との低レベルインタフェース）。
- 浮動小数点演算のサポートを含む浮動小数点環境 (fenv)（436 ページの *fenv.h* を参照）。
- 特殊なコンパイラのサポート。たとえば、切替えの処理や整数演算のための関数など。

AEABI 準拠の詳細については、207 ページの *AEABI への準拠*を参照してください。

ライブラリの詳細は、「*ライブラリ関数*」を参照してください。

ランタイム環境の設定

IAR DLIB ランタイム環境は、デバッガでそのまま使用できます。ただし、ハードウェアでアプリケーションを実行するには、ランタイム環境を適応させる必要があります。また、最もコード効率の高いランタイム環境を設定するには、アプリケーションやハードウェアの要件を特定する必要があります。必要な機能が多いほど、コードのサイズも大きくなります。

以下は、使用するターゲットハードウェアに最も効率の良いランタイム環境を設定する手順の概要です。

- 使用するランタイム環境のオブジェクトファイルを選択します

  ILINK により正しいライブラリファイルが自動的に使用されるため、ライブラリファイルを明示的に指定する必要はありません。112 ページの *ビルド済ライブラリの使用*を参照してください。
- どの定義済ランタイムライブラリ設定を使用するかを選択します（ノーマルまたはフル）

  特定のライブラリ機能のサポートレベルを設定できます。たとえば、ロケールやファイル記述子、マルチバイト文字などがあります。何も指定しない場合、ライブラリオブジェクトファイルに一致するデフォルトのライブラリ設定ファイルが自動的に使用されます。ライブラリ設定を明示的に指定するには、`--dlib_config` コンパイラオブジェクトを使用します。130 ページの *ライブラリ構成*を参照してください。
- ランタイムライブラリのサイズの最適化

  関数 `printf`、`scanf`、およびこれらの派生関数で使用されるフォーマッタを指定できます（116 ページの *printf、scanf のフォーマッタの選択*を参照）。

  また、スタックとヒープのサイズおよび配置を指定できます（それぞれ 100 ページの *スタックメモリの設定*、100 ページの *ヒープメモリの設定*を参照）。
- ランタイムのデバッグサポートおよび I/O デバッグのインクルード

  ライブラリは、C-SPY の［ターミナル I/O］ウィンドウへの標準入力および出力のリダイレクトや、ホストコンピュータ上のファイルへのアクセスのサポートを提供します（118 ページの *アプリケーションデバッグサポート*を参照）。

- ターゲットハードウェアのライブラリの適合

  ライブラリは、ターゲットシステムへのアクセス処理に低レベルの関数のセットを使用します。これらのアクセスを機能させるには、これらの関数の自分のバージョンを実装する必要があります。たとえば、`printf` でボード上の LCD ディスプレイに書き込むようにするには、ターゲットに適合したバージョンの低レベルの関数 `__write` を実装して、文字をディスプレイに書き込めるようにする必要があります。こうした関数をカスタマイズするには、ライブラリ低レベルを十分に理解してください（122 ページの *ターゲットハードウェアのライブラリの適合*を参照）。

- ライブラリモジュールのオーバライド

  ライブラリの機能をカスタマイズしている場合、デフォルトのモジュールではなく、自分のライブラリモジュールのバージョンが使用されるようにしてください。オーバライドは、ライブラリ全体をリビルドせずに行うことができます（123 ページの *ライブラリモジュールのオーバライド*を参照）。

- システム初期化のカスタマイズ

  システム初期化のソースコードをカスタマイズしなければならないことがほとんどです。たとえば、使用するアプリケーションがメモリをマッピングされた特殊な関数レジスタを初期化したり、データセクションのデフォルトの初期化を省略しなければならないことがあります。この場合、ルーチン `__low_level_init` をカスタマイズします。これによって、データセクションが初期化される前に実行されるようにします。125 ページの *システムの起動と終了*および 129 ページの *システム初期化のカスタマイズ*を参照してください。

- 独自のライブラリ設定ファイルの設定

  ビルド済のライブラリ設定のほかに、独自のライブラリ設定を作成できますが、これにはライブラリの *リビルド*が必要です。この機能により、ランタイム環境を完全に管理できます。124 ページの *カスタマイズしたライブラリのビルドと使用*を参照してください。

- マルチスレッド環境の管理

  マルチスレッド環境では、すべてのライブラリオブジェクトがスレッドにとってグローバルかローカルに応じて処理されるように、ランタイムライブラリを適応させる必要があります。143 ページの *マルチスレッド環境の管理*を参照してください。

- モジュールの整合性チェック

  ランタイムモデル属性を使用して、モジュールが互換性のある設定を使用してビルドされるようにします（149 ページの *モジュールの整合性チェック*を参照）。

ビルド済ライブラリの使用

ビルド済のランタイムライブラリは、以下の機能のさまざまな組合せについて設定されています。

- アーキテクチャ
- CPU モード
- バイトオーダ
- ライブラリ構成（ノーマルまたはフル）
- 浮動小数点数実装。

リンカは、正しいライブラリオブジェクトファイルとライブラリ設定ファイルを自動的にインクルードします。ライブラリ設定を明示的に指定するには、`--dlib_config` オブジェクトを使用します。詳細については、58 ページの *ランタイム環境*を参照してください。

ライブラリファイル名構文

ライブラリの名前は、以下のように構成されます。

{*architecture*}	アーキテクチャの名前です。ARM アーキテクチャ v4T、v5TE、v6M、v7M、v7A/R に対して、`4t`、`5E`、`6M`、`7M`、`7S` をそれぞれ使用できます。v5TE アーキテクチャ用にビルドされたライブラリは、v6 アーキテクチャでも使用されます（v6M 用を除く）。 C-RUN サポートライブラリは、アーキテクチャ名に `4` または `7` を使用します。 アーキテクチャ名はサフィックスとして、`--no-literal_pool` でビルドされたライブラリの場合は `x` を、C-RUN の C++ サポートライブラリについては `as` を使用します。
{*cpu_mode*}	Thumb と ARM の場合、それぞれ `t` または `a` のどちらかです。
{*byte_order*}	リトルエンディアンとビッグエンディアンの場合、それぞれ `l` または `b` です。

{*fp_implementation*}	_ VFP のサポートなしにライブラリをコンパイルする場合。つまり、ソフトウェア実装の浮動小数点関数を使用する場合です。 s 単精度 (VFPv4_sp、VFPv5_sp) のみのサポートで VFP アーキテクチャに対してライブラリをコンパイルする場合。 v アーキテクチャ VFPv2 またはそれ以降について、VFP のサポートがある状態でライブラリをコンパイルする場合。VFP のサポートがある状態でコンパイルされたライブラリには、それぞれの関数に浮動小数点の署名を持つ 2 つのエントリがあります。1 つのエントリは AAPCS ベースの標準の派生 VFP に準拠します。リンカは VFP の呼出し規約を使用し、他のモジュールについてはベース標準のエントリを使用します。
{*language*}	標準の C++ サポートに対してライブラリがコンパイルされる場合は c、Embedded C++ サポートでコンパイルされる場合は e です。
{*lib_config*>	ノーマルとフルの場合は、それぞれ n か f です。
{*debug_interface*}	s、b、i（それぞれ SWI/SVC メカニズム、BKPT メカニズム、IAR 固有のブレークポイントメカニズムを示します）のいずれかです。詳細については、320 ページの *--semihosting* を参照してください。
{*rwpi*}	ライブラリにリード / ライトかつ位置に依存しないコードが含まれる場合、s になります（286 ページの *--rwpi* を参照）。

注 : ライブラリ設定ファイルには、DLib_Config_Normal.h と DLib_Config_Full.h の 2 つがあります。

ライブラリオブジェクトファイルやライブラリ設定ファイルは、arm¥lib¥ サブディレクトリにあります。

ライブラリファイルのグループ

ライブラリは、以下のグループのライブラリ関数で提供されます。

C ライブラリ関数のライブラリファイル

これらは標準の C により定義される関数で、たとえば printf や scanf などです。このライブラリには数学関数は含まれません。

ライブラリファイルの名前は、以下のように構成されます。

```
dl{architecture}_{cpu_mode}{byte_order}{lib-config}{rwpi}.a
```

これは、特に次のことを表します。

```
dl{4t|5E|6M|6Mx|7M|7Mx|7Sx}_{a|t}{l|b}{n|f}{s}.a
```

C++ ライブラリ関数および Embedded C++ ライブラリ関数のライブラリファイル

これらは C++ により定義される関数で、標準の C++ または Embedded C++ のサポートを使用してコンパイルされます。

ライブラリファイルの名前は、以下のように構成されます。

```
dlpp{architecture}_{cpu_mode}{byte_order}
_{lib-config}{language}{rwpi}.a
```

これは、特に次のことを表します。

```
dlpp{4t|5E|6M|6Mx|7M|7Mx|7Sx|4as|7as}_{a|t}{l|b}_{n|f}{c|e}{s].a
```

数学関数のライブラリファイル

これらは浮動小数点算術の関数および C 規格で定義された署名の浮動小数点型を持つ関数です。たとえば、`sqrt` のような関数です。

ライブラリファイルの名前は、以下のように構成されます。

```
m{architecture}_{cpu_mode}{byte_order}{fp_implementation}.a
```

これは、特に次のことを表します。

```
m{4t|5E|6M|6Mx|7M|7Mx|7Sx}_{a|t}{l|b}{|v|s}.a
```

ランタイムサポート関数のライブラリファイル

これらの関数は、システム起動、初期化、非浮動小数点 AEABI サポートルーチン、C/C++ 標準に含まれる一部の関数用です。

ライブラリファイルの名前は、以下のように構成されます。

```
rt{architecture}_{cpu_mode}{byte_order}.a
```

これは、特に次のことを表します。

```
rt{4t|5E|6M|6Mx|7M|7Mx|7Sx}_{a|t}{l|b}.a
```

デバッグサポート関数のライブラリファイル

これらの関数は、セミホストインタフェースのデバッグサポート用です。ライブラリファイルの名前は、以下のように構成されます。

```
sh{debug_interface}_{byte_order}.a
```

または

```
sh{architecture}_{byte_order}.a
```

これは、特に次のことを表します。

```
sh{s|b|i}_{l|b}.a
```

または

```
sh{6Mx|7Mx|7Sx}_{l|b}.a
```

C-RUN サポート関数のライブラリファイル

これらは、C-RUN を使用したランタイムチェックのサポート関数です。ライブラリファイルの名前は、以下のように構成されます。

```
as_wp{architecture}{byte_order}.a
```

または

```
as{architecture}{byte_order}.a
```

これは、特に次のことを表します。

```
as_wp{4|7}{l|b}.a
```

または

```
as{4|7}{l|b}.a
```

ビルド済ライブラリのカスタマイズ（リビルドなし）

IAR コンパイラに付属のビルド済ライブラリは、そのまま使用できます。ただし、リビルドせずにライブラリの一部をカスタマイズできます。

カスタマイズ可能な項目は、以下のとおりです。

カスタマイズ可能な項目	参照先
printf、scanf のフォーマッタ	116 ページの *printf、scanf のフォーマッタの選択*
起動 / 終了コード	125 ページの *システムの起動と終了*
低レベル I/O	131 ページの *標準 I/O ストリーム*
ファイル I/O	135 ページの *ファイル I/O*
低レベル環境関数	138 ページの *環境の操作*
低レベルシグナル関数	139 ページの *signal と raise*
低レベル時間関数	139 ページの *時間*

表 5: カスタマイズ可能な項目

カスタマイズ可能な項目	参照先
一部のライブラリ数学関数	140 ページの *数学関数*
ヒープ、スタック、セクションのサイズ	198 ページの *スタックについて* 199 ページの *ヒープについて* 81 ページの *コードおよびデータの配置（リンカ設定ファイル）*

表 5: カスタマイズ可能な項目 （続き）

ライブラリモジュールのオーバライドの詳細については、123 ページの *ライブラリモジュールのオーバライド*を参照してください。

printf、scanf のフォーマッタの選択

リンカは、コンパイラからの情報に基づいて、printf および scanf 関連の関数に適切なフォーマッタを自動的に選択します。printf が関数ポインタを介して使用されていたり、オブジェクトファイルが古いなどの理由で情報が入手できなかったり不十分な場合は、自動的にフルフォーマッタが選択されます。この場合は、フォーマッタを手動で選択した方がいいときもあります。

すべての printf- および scanf 関連の機能に対するデフォルトのフォーマッタをオーバライドするには（wprintf と wscanf の派生型を除く）、適切なライブラリオプションを設定します。ここでは、使用可能なオプションについて説明します。

注 : ライブラリをリビルドする場合は、これらの関数をさらに最適化できます（133 ページの *printf、scanf の構成シンボル*を参照）。

PRINTF フォーマッタの選択

printf 関数は、_Printf というフォーマッタを使用します。フルバージョンのフォーマッタはサイズが非常に大きく、多くの組込みアプリケーションで不要な機能が用意されています。メモリ消費量を削減するため、標準 C/EC++ ライブラリでは 3 つの小さい別バージョンも提供されています。

以下の表に、各種フォーマッタの機能の概要を示します。

フォーマット機能	極小	小 / マルチバイトなし	大 / マルチバイトなし	フル / マルチバイトなし
基本指定子 c、d、i、o、p、s、u、X、x、%	あり	あり	あり	あり
マルチバイト文字サポート	なし	あり / なし	あり / なし	あり / なし

表 6: printf のフォーマッタ

フォーマット機能	極小	小 / マルチバイトなし	大 / マルチバイトなし	フル / マルチバイトなし
浮動小数点数指定子 a、A	なし	なし	なし	あり
浮動小数点数指定子 e、E、f、F、g、G	なし	なし	あり	あり
変換指定子 n	なし	なし	あり	あり
フォーマットフラグ +、-、#、0、空白	なし	あり	あり	あり
サイズ修飾子 h、l、L、s、t、Z	なし	あり	あり	あり
フィールド幅、精度（* を含む）	なし	あり	あり	あり
long long のサポート	なし	なし	あり	あり

表 6: printf のフォーマッタ（続き）

フォーマット機能をさらに調整する方法については、133 ページの *printf、scanf の構成シンボル*を参照してください。

IDE での手動による printf のフォーマッタの指定

フォーマッタを手動で指定するには、[プロジェクト] > [オプション] を選択し、[一般オプション] カテゴリを選びます。[ライブラリオプション] ページで該当オプションを選択します。

コマンドラインからの手動による printf のフォーマッタの指定

フォーマッタを手動で指定するには、以下の ILINK コマンドラインオプションのいずれかを使用します。

```
--redirect _Printf=_PrintfFull
--redirect _Printf=_PrintfFullNoMb
--redirect _Printf=_PrintfLarge
--redirect _Printf=_PrintfLargeNoMb
--redirect _Printf=_PrintfSmall
--redirect _Printf=_PrintfSmallNoMb
--redirect _Printf=_PrintfTiny
--redirect _Printf=_PrintfTinyNoMb
```

SCANF フォーマッタの選択

printf 関数と同様に、scanf でも _Scanf という一般的なフォーマッタを使用します。フルバージョンのフォーマッタはサイズが非常に大きく、多くの組込みアプリケーションで不要な機能が用意されています。メモリ消費量を削減するため、標準 C/C++ ライブラリでは 2 つの別バージョンも提供されています。

以下の表に、各種フォーマッタの機能の概要を示します。

フォーマット機能	小 / マルチバイトなし	大 / マルチバイトなし	フル / マルチバイトなし
基本指定子 c、d、i、o、p、s、u、X、x、%	あり	あり	あり
マルチバイト文字サポート	あり / なし	あり / なし	あり / なし
浮動小数点数指定子 a、A	なし	なし	あり
浮動小数点数指定子 e、E、f、F、g、G	なし	なし	あり
変換指定子 n	なし	なし	あり
スキャンセット [、]	なし	あり	あり
代入抑制 *	なし	あり	あり
long long のサポート	なし	なし	あり

表 7: scanf のフォーマッタ

フォーマット機能をさらに調整する方法については、133 ページの *printf、scanf の構成シンボル*を参照してください。

IDE での手動による scanf のフォーマッタの指定

フォーマッタを手動で指定するには、[プロジェクト] > [オプション] を選択し、[一般オプション] カテゴリを選びます。[ライブラリオプション] ページで該当オプションを選択します。

コマンドラインからの手動による scanf のフォーマッタの指定

フォーマッタを手動で指定するには、以下の ILINK コマンドラインオプションのいずれかを使用します。

```
--redirect _Scanf=_ScanfFull
--redirect _Scanf=_ScanfFullNoMb
--redirect _Scanf=_ScanfLarge
--redirect _Scanf=_ScanfLargeNoMb
--redirect _Scanf=_ScanfSmall
--redirect _Scanf=_ScanfSmallNoMb
```

アプリケーションデバッグサポート

デバッグ情報を生成するツールのほかに、ライブラリ低レベルインタフェース（通常は I/O 処理と基本ランタイムサポート）のデバッグバージョンがあります。デバッグライブラリを使用すると、ホストコンピュータ上でファイ

ルを開いて stdout をデバッガの［ターミナル I/O］ウィンドウにリダイレクトするなどの処理を速く実行できます。

C-SPY デバッグサポートを含める

ライブラリで以下についてデバッグサポートを提供できます。

- プログラムの中止、終了、アサーションの処理
- I/O 処理。stdin と stdout が C-SPY の［ターミナル I/O］ウィンドウにリダイレクトされ、デバッグ中にホストコンピュータ上のファイルにアクセス可能になります。

ILINK オプションの［**セミホスティング**］(--semihosted) または［**IAR ブレークポイント**］(--semihosting=iar_breakpoint) を使用してアプリケーションをビルドした場合、ライブラリ内の特定の関数が、デバッガと通信する関数に置き換えられます。

IDE でデバッグサポートのリンカオプションを指定するには、［**プロジェクト**］>［**オプション**］>［**一般オプション**］を選択します。［**ライブラリ構成**］ページで、［セミホスティング］オプションまたは［**IAR ブレークポイント**］オプションを選択します。

一部の Cortex-M デバイスでは、SWO 経由で stdout/stderr を出力することも可能です。これによって、セミホスティングに比べて stdout/stderr のパフォーマンスが大幅に向上します。ハードウェア要件については *『ARM 用 C-SPY® デバッグガイド』* を参照してください。

コマンドライン上で SWO 経由で stdout を有効にするには、リンカオプション --redirect __iar_sh_stdout=__iar_sh_stdout_swo を使用します。

IDE で SWO 経由で stdout を有効にするには、［**プロジェクト**］>［**オプション**］>［**一般オプション**］を選択します。［**ライブラリ構成**］ページで、［**セミホスティング**］オプションおよび［**SWO 経由の stdout/stderr**］オプションを選択します。

ライブラリ機能のデバッグ

デバッグライブラリは、デバッグしているアプリケーションとデバッガ自体の通信に使用されます。デバッガは、低レベルの DLIB インタフェース経由で、ファイルやターミナル I/O などの機能をホストコンピュータ側で実行するためのランタイムサービスをアプリケーションに提供します。

これらの機能は、アプリケーション開発の初期段階、たとえばファイル I/O を使用するアプリケーションで、フラッシュファイルシステム I/O ドライバを実装する前などに非常に便利な場合があります。また、stdin や stdout を使用するアプリケーションで、実際の I/O 用ハードウェアデバイスがない状

態でデバッグする必要がある場合にも使用します。もう 1 つの使用法は、デバッグ出力の生成です。

DLIB により提供される低レベルデバッガランタイムインタフェースは、ARM Limited により提供されるセミホストインタフェースに準拠しています。アプリケーションがセミホスティングを呼出す際、デバッガのブレークポイントで実行が停止します。続いて、デバッガが呼出しを処理し、ホストコンピュータ上で必要なすべてのアクションを行って実行を再開します。

セミホスティングのメカニズム

使用可能なセミホスティングのメカニズムには、以下の 3 つがあります。

- Cortex-M の場合、インタフェースは BKPT 命令を使用してセミホスティングの呼出しを実行します
- 他の ARM コアの場合、セミホスティングの呼出しに SVC 命令が使用されます
- *IAR ブレークポイント*。これは SVC を使用するセミホスティングに対する IAR 固有の代替手段です

SVC 経由でセミホスティングをサポートするには、デバッガが Supervisor Call ベクタ上にセミホスティングのブレークポイントを設定して、SVC の呼出しを検出する必要があります。アプリケーションでセミホスティング以外の目的に SVC 呼出しを使用する場合、このブレークポイントの処理によって、こうした呼出しの度に重大なパフォーマンスへの影響が発生します。IAR ブレークポイントは、これを回避するひとつの方法です。セミホスティングの実行に SVC 命令ではなく特殊な関数呼出しを使用することにより、その特殊な関数上にセミホスティングのブレークポイントを設定できます。つまり、セミホスティングが Supervisor Call ベクタの他の用法に干渉しなくなります。

IAR ブレークポイントは、セミホスティング標準の IAR 固有の拡張です。IAR システム以外のベンダからのツールチェーンによってビルドしたライブラリにアプリケーションをリンクして、IAR ブレークポイントを使用する場合、これらのライブラリのコードからのセミホスティング呼出しは機能しません。

C-SPY の［ターミナル I/O］ウィンドウ

［ターミナル I/O］ウィンドウを使用可能にするには、I/O デバッグのサポートを使用してアプリケーションをリンクする必要があります。つまり、関数 `__read` や `__write` が呼出されて、`stdin`、`stdout`、`stderr` ストリーム上で I/O 操作を実行するときに、C-SPY の［ターミナル I/O］ウィンドウに対してデータ送信またはデータの読取りが行われます。

注： `__read` や `__write` が呼出されても、［ターミナル I/O］ウィンドウは自動的には表示されません。手動で表示する必要があります。

［ターミナル I/O］ウィンドウの詳細については、*『ARM 用 C-SPY® デバッグガイド』*を参照してください。

ターミナル出力の高速化

一部のシステムでは、ホストコンピュータとターゲットハードウェアが 1 文字ごとに通信する必要があるため、ターミナル出力が遅いことがあります。

このため、`__write` 関数の代替として、`__write_buffered` 関数が DLIB ライブラリに用意されています。このモジュールでは、出力をバッファし、一度に 1 ラインずつデバッガに送信するため、出力が高速化されます。この関数は、約 80 バイトの RAM メモリを使用する点に注意が必要です。

この機能を使用するには、IDE で［**プロジェクト**］>［**オプション**］>［**一般オプション**］>［**ライブラリオプション**］> を選択し、［**バッファターミナル出力**］を選択するか、以下の行をリンカコマンドラインに追加します。

```
--redirect __write=__write_buffered
```

デバッグライブラリの低レベル関数

デバッグライブラリには、以下の低レベル関数の実装が含まれています。

DLIB の低レベルインタフェースの関数	C-SPY の対応
`abort`	アプリケーションを終了します
`clock`	ホストコンピュータ上のクロックを返します
`__close`	ホストコンピュータ上の対応するファイルを閉じます
`__exit`	アプリケーションの最後に到達したことを通知します
`__lseek`	ホストコンピュータ上の対応するファイル内を検索します
`__open`	ホストコンピュータ上のファイルを開きます
`__read`	`stdin` が［ターミナル I/O］ウィンドウにダイレクトされ、それ以外のすべてのファイルはホスト上の対応するファイルを読み取ります
`remove`	ホストコンピュータ上のファイルを削除します
`rename`	ホストコンピュータ上のファイル名を変更します
`__iar_ReportAssert`	ターミナル I/O にアサートメッセージを出力します
`system`	［デバッグログ］ウィンドウにメッセージを出力し、`-1` を返します
`time`	ホストコンピュータ上の時刻を返します

表 8: デバッグライブラリ付きでリンクした場合に特殊な意味を持つ関数

DLIB の低レベルインタフェースの関数	C-SPY の対応
__write	stdout と stderr が［ターミナル I/O］ウィンドウにダイレクトされ、それ以外のすべてのファイルはホスト上の対応するファイルに書き込みます

表 8: デバッグライブラリ付きでリンクした場合に特殊な意味を持つ関数（続き）

注 : 先頭に _ または __ が付く低レベルのインタフェースは、直接アプリケーションで使用しないでください。代わりに、これらの関数を使用する高レベルの関数を使用してください。詳細については、122 ページの *ライブラリの低レベルインタフェース*を参照してください。

ターゲットハードウェアのライブラリの適合

ライブラリは、ターゲットシステムへのアクセス処理に低レベルの関数のセットを使用します。これらのアクセスを機能させるには、これらの関数の自分のバージョンを実装する必要があります。これらの低レベル関数は、*ライブラリ低レベルインタフェース*と呼ばれます。

低レベルインタフェースを実装したら、これらの関数のバージョンを自分のプロジェクトに追加する必要があります。これについては、123 ページの *ライブラリモジュールのオーバライド*を参照してください。

ライブラリの低レベルインタフェース

ライブラリは、ターゲットシステムとのやりとりに低レベルの関数のセットを使用します。たとえば、printf および他のすべての標準出力関数は、低レベル関数 __write を使用して、実際の文字を出力デバイスに送信します。低レベル関数のほとんどは、__write と同じように実装を持ちません。その代わりに、自分のハードウェアに合わせて自ら実装する必要があります。

ただし、デバッグバージョンのライブラリ低レベルインタフェースがライブラリに含まれており、ここでターゲットハードウェアではなくデバッガを介してホストコンピュータとやりとりするように低レベル関数が実装されます。デバッグライブラリを使用する場合、［ターミナル I/O］ウィンドウへの書込みや、ホストコンピュータ上のファイルへのアクセス、ホストコンピュータからの時間の取得といったタスクをアプリケーションが実行できます。詳しくは 119 ページの *ライブラリ機能のデバッグ*を参照してください。

アプリケーションで低レベル関数を直接使用しないでください。対応する標準ライブラリ関数を使用するようにしてください。たとえば、stdout に書き込むには、__write の代わりに printf や puts といった標準のライブラリ関数を使用してください。

自作モジュールでオーバライドできるライブラリファイルは、arm¥src¥lib ディレクトリにあります。

低レベルインタフェースについては、以下のセクションでさらに詳しく説明しています。

- 131 ページの *標準 I/O ストリーム*
- 135 ページの *ファイル I/O*
- 139 ページの *signal と raise*
- 139 ページの *時間*
- 142 ページの *Assert*

ライブラリモジュールのオーバライド

実装したライブラリ低レベルのインタフェースを使用するには、それをアプリケーションに追加します。122 ページの *ターゲットハードウェアのライブラリの適合*を参照してください。そうでなければ、カスタマイズしたバージョンでデフォルトのライブラリルーチンをオーバライドすると便利です。どちらの場合も、以下の手順に従ってください。

1. テンプレートソースファイル（ライブラリソースファイルまたは別のテンプレート）を使用して、それを自分のプロジェクトディレクトリにコピーします。
2. ファイルを修正します。
3. 他のソースファイルと同じように、カスタマイズしたファイルを自分のプロジェクトに追加します。

   注 : ライブラリ低レベルインタフェースを実装し、デバッグサポートによりビルドしたプロジェクトにそれを追加済の場合、低レベル関数は使用されますが、C-SPY のデバッグサポートモジュールは使用されません。たとえば、デバッグサポートモジュール __write を自作モジュールに置き換えた場合は、C-SPY の［ターミナル I/O］ウィンドウはサポートされません。

   モジュール内の関数をオーバライドするには、オーバライドするモジュール内のすべての必要なシンボルに代替の実装を用意する必要があります。これを行わないと、重複定義エラーが発生します。

   自作モジュールでオーバライドできるライブラリファイルは、arm¥src¥lib ディレクトリにあります。

カスタマイズしたライブラリのビルドと使用

カスタマイズされたライブラリのビルドは、複雑なプロセスです。そのため、本当に必要かどうかを慎重に検討する必要があります。以下の場合には、独自の C/C++ 標準ライブラリをビルドする必要があります。

- ロケール、ファイル記述子、マルチバイト文字などをサポートする、独自のライブラリ構成を定義したい。

このような場合は、以下を行う必要があります。

- ライブラリプロジェクトをセットアップする
- 必要なライブラリ修正をする
- カスタマイズしたライブラリをビルドする
- カスタマイズしたライブラリをアプリケーションプロジェクトで使用する

注 : IAR コマンドラインビルドユーティリティ (`iarbuild.exe`) を使用して、コマンドラインで IAR Embedded Workbench プロジェクトをビルドします。ただし、コマンドラインでライブラリをビルドするための make ファイルやバッチファイルは提供されていません。

ビルドプロセスと IAR コマンドラインユーティリティについては、*ARM 用 IDE プロジェクト管理およびビルドガイド*を参照してください。

ライブラリプロジェクトのセットアップ

IDE では、ランタイム環境構成のカスタマイズに使用できるライブラリプロジェクトテンプレートが提供されています。このライブラリテンプレートは、ライブラリ構成がフルに設定されています。表 9*「ライブラリ構成」*を参照してください。

IDE で、作成したライブラリプロジェクトの［一般オプション］をアプリケーションに応じて変更します（55 ページの*基本的なプロジェクト設定*を参照）。

注 : オプションの設定について、1 つ重要な制限があります。ファイルレベルでオプションを設定（ファイルレベルでオーバライド）すると、ファイルを適用対象とする上位レベルのオプションは、一切このファイルに適用されなくなります。

ライブラリ機能の修正

ロケール、ファイル記述子、マルチバイト文字などのサポートを修正する場合は、ライブラリ設定ファイルを修正し、自作のライブラリをビルドする必要があります。これには、ランタイム環境の一部の追加 / 削除が含まれます。

ライブラリの機能は、*構成シンボル*で決定されます。これらのシンボルのデフォルト値は、DLib_Defaults.h ファイルで定義されています。このファイルはリードオンリーで、設定可能な値が記述されています。さらに、ユーザのライブラリには独自のライブラリ設定ファイルがあり、これによって必要なライブラリ設定を使用して特定のライブラリを設定します。詳細については、115 ページの *ビルド済ライブラリのカスタマイズ（リビルドなし）*を参照してください。

ライブラリ設定ファイルは、ランタイムライブラリのビルドや、システムヘッダファイルの調整に使用します。

ライブラリ設定ファイルの修正

ライブラリプロジェクトで、ライブラリ設定ファイルを開き、アプリケーション要件に従って構成シンボルの値を設定してカスタマイズします。

終了したら、適切なプロジェクトオプションを設定してライブラリプロジェクトをビルドします。

カスタマイズしたライブラリの使用

ライブラリをビルドしたら、アプリケーションプロジェクトで使用できるように設定する必要があります。

IDE で以下の手順を実行する必要があります。

1. **［プロジェクト］>［オプション］**を選択し、**［一般オプション］**カテゴリで**［ライブラリ構成］**タブをクリックします。
2. **［ライブラリ］**ドロップダウンリストから、**［カスタム DLIB］**を選択します。
3. **［設定ファイル］**テキストボックスで、ライブラリ設定ファイルを指定します。
4. **［リンカ］**カテゴリで、**［ライブラリ］**タブをクリックします。**［追加ライブラリ］**テキストボックスで、ライブラリファイルを指定します。

システムの起動と終了

ここでは、アプリケーションの起動と終了時のランタイム環境の動作について説明します。

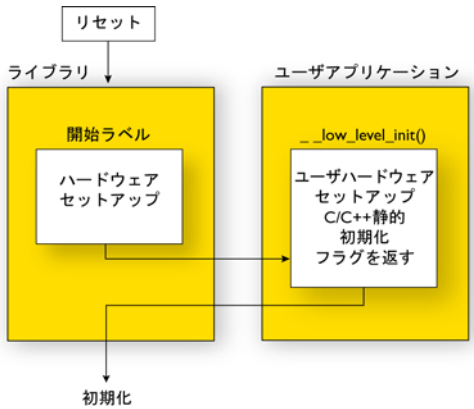
起動と終了を扱うコードは、ソースファイル cstartup.s、cmain.s、cexit.s、および arm¥src¥lib ディレクトリの low_level_init.c にあります。

システム起動コードのカスタマイズ方法については、「129 ページの *システム初期化のカスタマイズ*」を参照してください。

システム起動

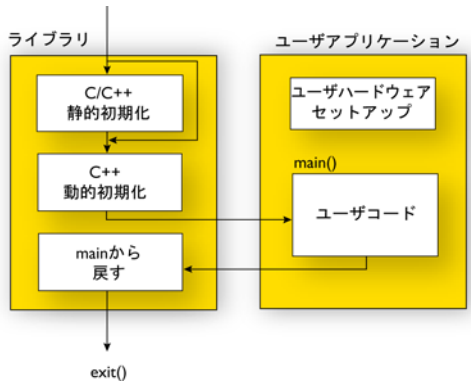
システムの起動時、`main` 関数が入力される前に初期化シーケンスが実行されます。このシーケンスでは、ターゲットハードウェアと C/C++ 環境で必要とされる初期化を実行します。

ハードウェアの初期化は、以下のように実行されます。



- CPU がリセットされると、システム起動コードのプログラムエントリラベル `__iar_program_start` で起動を開始します
- スタックポインタは、`CSTACK` ブロックの最後の位置に初期化されます
- ARM7/9/11、Cortex-A、Cortex-R のデバイスについては、例外スタックポインタは対応する各セクションの最後の位置に初期化されます
- 関数 `__low_level_init` を定義している場合、この関数が呼出され、アプリケーションで低レベルの初期化を実行できます

C/C++ の初期化は、以下のように実行されます。

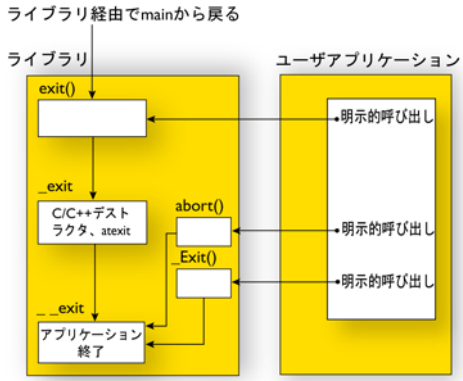


- 静的変数とグローバル変数が初期化されます。つまり、ゼロ初期化変数がクリアされ、他の初期化変数の値が ROM から RAM メモリにコピーされます。このステップは、`__low_level_init` がゼロを返す場合には、省略されます。詳細については、84 ページの *システム起動時の初期化*を参照してください
- 静的 C++ オブジェクトが生成されます
- `main` 関数が呼出され、アプリケーションが起動します

初期化フェーズについて詳しくは、51 ページの *アプリケーションの実行— 概要*を参照してください。

システム終了

以下の図には、組込みアプリケーションの終了を制御するための各種の方法を示します。



アプリケーションは、以下の 2 つの方法で正常終了できます。

- `main` 関数から戻る
- `exit` 関数を呼出す

C 規格では、2 つの方法は同等であると規定されているため、システム起動コードは `main` から戻る際に `exit` 関数を呼出します。`exit` 関数には、`main` のリターン値がパラメータとして引き渡されます。

デフォルトの `exit` 関数は、C で記述されています。この関数は、これらの動作を実行する小さなアセンブラ関数 `_exit` を呼出します。

- アプリケーション終了時に実行するように登録した関数を呼出します。これには、C++ の静的 / グローバル変数のデストラクタと、標準関数 `atexit` で登録された関数が含まれます
- 開かれているすべてのファイルを閉じます
- `__exit` を呼出します
- `__exit` の最後まで到達したら、システムを停止します

アプリケーションは、`abort` または `_Exit` 関数を呼出すことによっても終了できます。`abort` 関数は、単に `__exit` を呼出してシステムの停止を行うだけで、終了処理は実行しません。`_Exit` 関数も `abort` 関数とほとんど同じですが、`_Exit` では、終了ステータス情報を渡すための引数をとります。

終了時にアプリケーションで追加処理（システムのリセットなど）を実行する場合には、独自の `__exit(int)` 関数を記述することができます。

システム終了と C-SPY のインタフェース

プロジェクトがセミホストインタフェースにリンクされている場合、通常の __exit 関数は特殊なものに置き換えられます。これにより、C-SPY はこの関数が呼出されたことを認識し、適切な処理を実行して、プログラム終了のシミュレーションを行います。詳細については、118 ページの *アプリケーションデバッグサポート*を参照してください。

システム初期化のカスタマイズ

多くの場合、システム初期化用コードのカスタマイズが必要になります。たとえば、メモリマップされた特殊機能レジスタ (SFR) をアプリケーションで初期化することが必要となる場合や、cstartup によってデフォルトで実行されるデータセクションの初期化を省略することが必要となる場合などがあります。

これは、__low_level_init ルーチンをカスタマイズすることで実現できます。このルーチンは、データセクションの初期化前に cmain.s から呼出されます。cstartup.s ファイルは直接修正しないでください。

Cortex-M の場合、システム起動を処理するコードはソースファイル cstartup_M.s と low_level_init.c にあり、これらは arm¥src¥lib ディレクトリに格納されています。

その他の ARM デバイスの場合、システム起動を処理するコードはソースファイル cstartup.s と low_level_init.c にあります。これらは arm¥src¥lib ディレクトリに格納されています。

注 : 通常は、ファイル cmain.s または cexit.s をカスタマイズする必要はありません。

ライブラリをリビルドする場合は、テンプレートライブラリプロジェクトに含まれるソースファイルを使用できます（124 ページの *カスタマイズしたライブラリのビルドと使用*を参照）。

注 : __low_level_init ルーチンや cstartup.s ファイルを修正する場合も、ライブラリをリビルドする必要はありません。

__LOW_LEVEL_INIT

本製品には、low_level_init.c という低レベル初期化ファイルのスケルトンが付属しています。この時点では変数は初期化されていないため、初期化された静的変数はこのファイル内では使用できません。

__low_level_init から返される値によって、データセクションをシステム起動コードで初期化するかどうかが決定されます。関数が 0 を返す場合、データセクションは初期化されません。

CSTARTUP.S ファイルの修正

前述のように、__low_level_init のカスタマイズによって目的の動作を達成できる場合は、cstartup.s ファイルを修正する必要はありません。ただし、cstartup.s ファイルの修正が必要な場合は、一般的な手順に従いファイルをコピーして修正し、プロジェクトに追加することをお勧めします（123 ページの *ライブラリモジュールのオーバライド*を参照）。

使用している cstartup.s のバージョンで使用される開始ラベルが、確実にリンカで使用されるようにする必要があります。リンカで使用される開始ラベルの変更方法については、305 ページの *--entry* を参照してください。

Cortex-M の場合、割込みまたは他の例外ハンドラを使用するには、cstartup_M.s または cstartup_M.c の修正済みコピーを作成する必要があります。

ライブラリ構成

ロケール、ファイル記述子、マルチバイト文字などのサポートレベルの設定が可能です。

ランタイム環境の構成は、*ライブラリ設定ファイル*で定義します。このファイルでは、ランタイム環境に含まれる機能が定義されています。設定ファイルは、ランタイムライブラリのビルド構成や、アプリケーションのコンパイル時に使用するシステムヘッダファイルの設定に使用します。ランタイム環境で必要な機能が少ないほど、サイズも小さくなります。

ライブラリの機能は、*構成シンボル*で決定されます。これらのシンボルのデフォルト値は、DLib_Defaults.h ファイルで定義されています。このファイルはリードオンリーで、設定可能な値が記述されています。

以下のいずれかの定義済ライブラリ構成を使用できます。

ライブラリ構成	説明
ノーマル DLIB（デフォルト）	ロケールインタフェース、C ロケールなし、ファイル記述子のサポートなし、また printf および scanf でのマルチバイト文字もなし。
フル DLIB	フルロケールインタフェース、C ロケール、ファイル記述子のサポート、また printf および scanf でのマルチバイト文字もなし。

表 9: ライブラリ構成

ランタイム構成の選択

ランタイム構成を選択するには、以下のいずれかの方法を使用します。

- デフォルトのビルド済構成。ライブラリ構成を明示的に指定しない場合、デフォルト構成が使用されます。ランタイムライブラリのオブジェクトファイルに一致する構成ファイルが、自動的に使用されます。
- 選択したビルド済構成。ランタイム構成を明示的に指定するには、`--dlib_config` コンパイラオプションを使用します。262 ページの *--dlib_config* を参照してください。
- 独自の構成。自分用に構成を定義できます。つまり、構成ファイルを修正する必要があります。ライブラリ設定ファイルには、ライブラリがどのようにビルドされたかが記述されています。そのため、変更を有効にするには、ライブラリをリビルドする必要があります。詳細については、124 ページの *カスタマイズしたライブラリのビルドと使用*を参照してください。

ビルド済ライブラリは、デフォルト構成を使用してビルドされています（130 ページの *ライブラリ構成*を参照）。

標準 I/O ストリーム

標準通信チャンネル（ストリーム）は、`stdio.h` で定義されます。これらのストリームのいずれかをアプリケーション（関数 `printf` や `scanf` など）で使用している場合は、ハードウェアに合わせて低レベル機能をカスタマイズする必要があります。

C /C++ ですべてのキャラクタベース I/O を実行する際に使用される、低レベルの I/O 関数があります。キャラクタベース I/O を使用する場合は、ハードウェア環境で提供される機能を使用して、これらの関数を定義する必要があります。低レベル関数の実装について詳しくは、122 ページの *ターゲットハードウェアのライブラリの適合*を参照してください。

低レベルキャラクタ I/O の実装

ストリーム `stdin` および `stdout` の低レベル関数を実装するには、`__read` および `__write` をそれぞれ記述する必要があります。これらの関数のテンプレートソースコードは、`arm¥src¥lib` ディレクトリにあります。

ライブラリをリビルドする場合は、テンプレートライブラリプロジェクトに含まれるソースファイルを使用できます（124 ページの *カスタマイズしたライブラリのビルドと使用*を参照）。I/O 用の低レベルルーチンをカスタマイズする場合は、ライブラリのリビルドは必要ありません。

注 : __read や __write を自分で記述する場合は、C-SPY ランタイムインタフェースを考慮する必要があります（118 ページの *アプリケーションデバッグサポート*を参照）。

__write の使用例

この例のコードは、メモリマップド I/O を使用して LCD ディスプレイに書込み、そのポートがアドレス 0x1000 にあると想定しています。

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0x1000;

size_t __write(int handle,
               const unsigned char *buf,
               size_t bufSize)
{
  size_t nChars = 0;

  /* すべてのハンドルをフラッシュするコマンドをチェック */
  if (handle == -1)
  {
    return 0;
  }

  /* stdout と stderr をチェック
     (FILE 記述子が有効な場合にのみ必要です) */
  if (handle != 1 && handle != 2)
  {
    return -1;
  }

  for (/* empty */; bufSize > 0; --bufSize)
  {
    lcdIO = *buf;
    ++buf;
    ++nChars;
  }

  return nChars;
}
```

注 : DLIB が __write を呼出すとき、DLIB は次のインタフェースを想定します。buf の値が NULL のときに __write を呼出すコマンドは、stream をフラッシュするためのものです。ハンドルが -1 の場合、すべてのストリームがフラッシュされます。

__read の使用例

この例のコードは、メモリマップド I/O を使用してキーボードから読込み、そのポートがアドレス 0x1000 にあると想定しています。

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0x1000;

size_t __read(int handle,
              unsigned char *buf,
              size_t bufSize)
{
  size_t nChars = 0;

  /* stdin をチェック
     (FILE 記述子が有効の場合にのみ必要) */
  if (handle != 0)
  {
    return -1;
  }

  for (/* empty */; bufSize > 0; --bufSize)
  {
    unsigned char c = kbIO;
    if (c == 0)
      break;

    *buf++ = c;
    ++nChars;
  }

  return nChars;
}
```

@ 演算子の詳細は、217 ページの*データと関数のメモリ配置制御*を参照してください。

printf、scanf の構成シンボル

アプリケーションプロジェクトをセットアップする場合は、アプリケーションで必要な printf と scanf のフォーマット機能を考慮する必要があります(116 ページの *printf、scanf のフォーマッタの選択*を参照)。

通常のフォーマッタでは要求に対応できない場合は、フォーマッタをカスタマイズできます。ただし、ランタイムライブラリをリビルドする必要があります。

printf と scanf のフォーマッタのデフォルトの動作は、DLIB_Defaults.h ファイルで構成シンボルにより定義されています。

以下の構成シンボルで、printf 関数の機能を決定します。

printf の構成シンボル	サポートする機能
_DLIB_PRINTF_MULTIBYTE	マルチバイト文字
_DLIB_PRINTF_LONG_LONG	long long（ll 修飾子）
_DLIB_PRINTF_SPECIFIER_FLOAT	浮動小数点数
_DLIB_PRINTF_SPECIFIER_A	16 進数の浮動小数点数値
_DLIB_PRINTF_SPECIFIER_N	出力回数 (%n)
_DLIB_PRINTF_QUALIFIERS	修飾子 h、l、L、v、t、z
_DLIB_PRINTF_FLAGS	フラグ -、+、#、0
_DLIB_PRINTF_WIDTH_AND_PRECISION	幅 / 精度
_DLIB_PRINTF_CHAR_BY_CHAR	文字単位出力 / バッファ出力

表 10: printf の構成シンボルの詳細

ライブラリのビルド時には、以下の構成シンボルで scanf 関数の機能を決定します。

scanf の構成シンボル	サポートする機能
_DLIB_SCANF_MULTIBYTE	マルチバイト文字
_DLIB_SCANF_LONG_LONG	long long（ll 修飾子）
_DLIB_SCANF_SPECIFIER_FLOAT	浮動小数点数
_DLIB_SCANF_SPECIFIER_N	出力回数 (%n)
_DLIB_SCANF_QUALIFIERS	修飾子 h、j、l、t、z、L
_DLIB_SCANF_SCANSET	スキャンセット ([*])
_DLIB_SCANF_WIDTH	幅
_DLIB_SCANF_ASSIGNMENT_SUPPRESSING	代入抑止 ([*])

表 11: scanf の構成シンボルの詳細

フォーマット機能のカスタマイズ

フォーマット機能をカスタマイズするには、以下を行う必要があります。

1. ライブラリプロジェクトをセットアップする（124 ページの*カスタマイズしたライブラリのビルドと使用*を参照）。
2. アプリケーションの必要に応じて、構成シンボルを定義します。

ファイル I/O

このライブラリには、fopen や fclose、fprintf、fputs など、ファイルの I/O 処理のための数多くの強力な関数が含まれています。これらの関数はすべて、いくつかの低レベル関数を呼出します。これらの関数は、それぞれ特定のタスクを 1 つ実行するように設計されています。たとえば、__open はファイルを開き、__write は文字を出力します。アプリケーションでファイルの I/O 処理用のライブラリ関数を使用する前に、ターゲットハードウェアにあわせて、対応する低レベル関数を実装する必要があります。詳細については、122 ページの *ターゲットハードウェアのライブラリの適合*を参照してください。

ライブラリのファイル I/O 機能は、フルのライブラリ構成を持つライブラリのみによってサポートされます（130 ページの *ライブラリ構成*を参照）。すなわち、ファイル I/O は、構成シンボル __DLIB_FILE_DESCRIPTOR が有効な場合にのみサポートされます。有効でない場合は、関数に FILE * 引数を指定することはできません。

以下のファイル I/O 用テンプレートコードが付属しています。

I/O 関数	ファイル	説明
__close	close.c	ファイルを閉じます。
__lseek	lseek.c	ファイル位置インジケータを設定します。
__open	open.c	ファイルを開きます。
__read	read.c	文字バッファをリードします。
__write	write.c	文字バッファをライトします。
remove	remove.c	ファイルを削除します。
rename	rename.c	ファイルリネームします。

表 12: 低レベルファイル I/O

低レベル関数は、開かれたファイルなどの I/O ストリームを、ファイル識別子（固有の整数）を使用して識別します。通常、stdin、stdout、stderr に関連付けられている I/O ストリームは、それぞれ 0、1、2 のファイル識別子を持ちます。

注 : I/O デバッグサポートを使用してアプリケーションをリンクする場合は、C-SPY との通信用に C-SPY 版の低レベル I/O 関数がリンクされます。詳細については、118 ページの *アプリケーションデバッグサポート*を参照してください。

ロケール

*ロケール*とは C 言語の機能であり、通貨記号、日付 / 時刻、マルチバイト文字エンコーディングなど、多数の項目を言語や国ごとに設定することができます。

使用しているランタイムライブラリに応じて、ロケールサポートのレベルが異なります。ただし、ロケールサポートのレベルが高いほど、コードのサイズも大きくなります。そのため、アプリケーションで必要なサポートのレベルを考慮する必要があります。

DLIB ライブラリは、以下の 2 つのメインモードで使用できます。

- ロケールインタフェースを使用し、実行中にロケールの切替えを可能にする
- ロケールインタフェースを使用せず、1 つの選択したロケールをアプリケーションに組込む

ビルド済ライブラリでのロケールサポート

ビルド済ライブラリでのロケールサポートのレベルは、ライブラリ設定によって異なります。

- すべてのビルド済ライブラリは、C のロケールのみをサポートしています
- *フルライブラリ設定*のライブラリはすべて、ロケールインタフェースをサポートしています。ロケールインタフェースをサポートするビルド済ライブラリの場合は、デフォルトでは実行時のマルチバイト文字エンコーディング切替えのみがサポートされます
- *ライブラリ設定がノーマル*のライブラリは、ロケールインタフェースをサポートしません

アプリケーションで別のロケールサポートが必要な場合には、ライブラリをリビルドする必要があります。

ロケールサポートのカスタマイズ

ライブラリをリビルドする場合は、いずれかのロケールを選択できます。

- 標準 C ロケール
- POSIX ロケール
- 広範な European ロケール

ロケールの構成シンボル

構成シンボル `_DLIB_FULL_LOCALE_SUPPORT` は、ライブラリ設定ファイルで定義され、ライブラリでロケールインタフェースをサポートするかどうかを

決定します。ロケール構成シンボル _LOCALE_USE_*LANG_REGION* および _ENCODING_USE_*ENCODING* は、サポートされているすべてのロケールおよびエンコーディングを定義します。

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C         /* C ロケール */
#define _LOCALE_USE_EN_US     /* American English */
#define _LOCALE_USE_EN_GB     /* British English */
#define _LOCALE_USE_SV_SE     /* Swedish in Sweden */
```

サポートされるロケールおよびエンコーディングの設定の一覧については、DLib_Defaults.h を参照してください。

ロケールサポートをカスタマイズする場合は、アプリケーションで必要なロケール構成シンボルを定義します。詳細については、124 ページの*カスタマイズしたライブラリのビルドと使用*を参照してください。

注 : C やアセンブラソースコードでマルチバイト文字を使用する場合は、正しいロケールシンボル（ローカルホストのロケール）を選択してください。

ロケールインタフェースをサポートしないライブラリのビルド

構成シンボル _DLIB_FULL_LOCALE_SUPPORT を 0（ゼロ）に設定する場合、ロケールインタフェースは含まれません。すなわち、固定ロケール（デフォルトでは標準 C）が使用され、サポートされているロケール構成シンボルのいずれか 1 つを選択できます。setlocale 関数が使用できないため、ロケールを実行中に変更することはできません。

ロケールインタフェースをサポートするライブラリのビルド

構成シンボル _DLIB_FULL_LOCALE_SUPPORT を 1 に設定すると、ロケールインタフェースのサポートが有効になります。デフォルトでは標準 C ロケールが使用されますが、必要な個数の構成シンボルを定義できます。setlocale 関数をアプリケーションで使用できるため、実行中にロケールを切り替えることができます。

実行中のロケール変更

アプリケーションの実行中にアプリケーションの正しいロケールを選択するには、標準ライブラリ関数 setlocale を使用します。

setlocale 関数では、2 つの引数を指定します。最初の引数には、LC_*CATEGORY* というフォーマットでロケールカテゴリを指定します。2 番目の引数には、ロケールを示す文字列を指定します。setlocale が返した文字列か、以下のフォーマットの文字列を指定します。

lang_REGION

または

lang_REGION.encoding

lang は言語コード、*REGION* は地域を示す修飾子、*encoding* は使用するマルチバイト文字エンコーディングを示します。

lang_REGION の部分は、ライブラリ設定ファイルで指定可能な _LOCALE_USE_*LANG_REGION* プリプロセッサシンボルに一致します。

例

この例は、ロケール構成シンボルを、フィンランドで使用できるようにスウェーデン語に設定し、UTF8 マルチバイト文字エンコーディングに設定します。

```
setlocale (LC_ALL, "sv_FI.Utf8");
```

環境の操作

C 規格に従い、アプリケーションは関数 getenv および system を使用して環境を操作できます。

注 : 規格では putenv 関数は必須ではなく、ライブラリにこの関数の実行は含まれません。

GETENV 関数

getenv 関数は、グローバル変数 __environ が指す文字列で、引数として指定したキーを検索します。キーが見つかった場合は、その値が返されます。見つからなかった場合は、0（ゼロ）が返されます。デフォルトでは、文字列は空白です。

文字列内のキーを作成や編集するには、NULL 終端文字列のシーケンスを次のフォーマットで作成する必要があります。

```
key=value¥0
```

文字列の最後に null 文字を 1 つ付けます（C 文字列を使用する場合、これは自動的に追加されます）。作成した文字列のシーケンスを、__environ 変数に代入します。

次に例を示します。

```
const char MyEnv[] = "Key=Value¥0Key2=Value2¥0";
__environ = MyEnv;
```

より高度な環境変数の操作が必要な場合は、独自の getenv 関数や、場合によっては putenv 関数を実装する必要があります。この場合、ライブラリのリ

ビルドは必要ありません。ソーステンプレートは、getenv.c や environ.c の各ファイルに含まれています。これらのファイルは、arm¥src¥lib ディレクトリにあります。デフォルトのライブラリモジュールのオーバライドについては、123 ページの *ライブラリモジュールのオーバライド*を参照してください。

システム関数

system 関数を使用する必要がある場合は、独自に実装する必要があります。ライブラリが提供する system 関数は、単に -1 を返します。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、124 ページの *カスタマイズしたライブラリのビルドと使用*を参照してください。

注 : I/O デバッグのサポートを使用してアプリケーションをリンクする場合、関数 system は C-SPY 版の関数に置き換えられます。詳細については、118 ページの *アプリケーションデバッグサポート*を参照してください。

signal と raise

関数 signal、raise がデフォルトで実装されています。デフォルトの関数で必要な機能が提供されていない場合は、自分で実装できます。

この場合、ライブラリのリビルドは必要ありません。ソーステンプレートは、signal.c や raise.c の各ファイルに含まれています。これらのファイルは、arm¥src¥lib ディレクトリにあります。デフォルトのライブラリモジュールのオーバライドについては、123 ページの *ライブラリモジュールのオーバライド*を参照してください。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、124 ページの *カスタマイズしたライブラリのビルドと使用*を参照してください。

時間

関数 __time32、__time64、date が機能するためには、関数 clock、__time32、__time64、および __getzone を実装する必要があります。__time32 と __time64 のどちらを使用するかは、time_t でどのインタフェースを使用するかによって決まります（437 ページの *time.h* を参照）。

これらの関数を実装する場合、ライブラリのリビルドは必要ありません。ソーステンプレートは、arm¥src¥lib ディレクトリのファイル clock.c、time.c、time64.c、getzone.c にあります。デフォルトのライブラリモ

ジュールのオーバライドについては、123 ページの *ライブラリモジュールのオーバライド*を参照してください。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、124 ページの *カスタマイズしたライブラリのビルドと使用*を参照してください。

デフォルトで実装されている `__getzone` は、UTC（Coordinated Universal Time＝万国標準時）をタイムゾーンとして指定します。

注 : I/O デバッグのサポートを使用してアプリケーションをリンクする場合、関数 `clock` および `time` は、C-SPY 版の関数に置き換えられます。これらの代替関数は、ホストのクロックと時刻をそれぞれ返します。詳細については、118 ページの *アプリケーションデバッグサポート*を参照してください。

数学関数

一部のライブラリ数学関数は、サイズが最適化されたバージョンやより精度の高いバージョンでも使用可能です。

より小さいバージョン

関数 `cos`、`exp`、`log`、`log2`、`log10`、`__iar_Log`（`log`、`log2`、`log10` のヘルプ関数）、`pow`、`sin`、`tan`、`__iar_Sin`（`sin` および `cos` のヘルプ関数）のより小さいその他のバージョンがライブラリにあります。これらはデフォルトのバージョンより 20% 小さく、20% 高速です。これらの関数は INF および NaN の値を処理します。欠点は、常に精度が失われることと、デフォルトのバージョンと同じ入力範囲を持っていない点です。

関数の名前は以下のように構成されます。

```
__iar_xxx_small<f|l>
```

`f` は `float` の派生型に、`l` は `long double` の派生型に使用され、`double` の派生型にサフィックスは使用されません。

これらの関数を使用するには、リンクする際に次のオプションを使用して、デフォルトの関数名がこれらの名前にリダイレクトされる必要があります。

```
--redirect sin=__iar_sin_small
--redirect cos=__iar_cos_small
--redirect tan=__iar_tan_small
--redirect log=__iar_log_small
--redirect log2=__iar_log2_small
--redirect log10=__iar_log10_small
--redirect exp=__iar_exp_small
--redirect pow=__iar_pow_small
--redirect __iar_Sin=__iar_Sin_small
--redirect __iar_Log=__iar_Log_small

--redirect sinf=__iar_sin_smallf
--redirect cosf=__iar_cos_smallf
--redirect tanf=__iar_tan_smallf
--redirect logf=__iar_log_smallf
--redirect log2f=__iar_log2_smallf
--redirect log10f=__iar_log10_smallf
--redirect expf=__iar_exp_smallf
--redirect powf=__iar_pow_smallf
--redirect __iar_FSin=__iar_Sin_smallf
--redirect __iar_FLog=__iar_Log_smallf

--redirect sinl=__iar_sin_smalll
--redirect cosl=__iar_cos_smalll
--redirect tanl=__iar_tan_smalll
--redirect logl=__iar_log_smalll
--redirect log2l=__iar_log2_smalll
--redirect log10l=__iar_log10_smalll
--redirect expl=__iar_exp_smalll
--redirect powl=__iar_pow_smalll
--redirect __iar_LSin=__iar_Sin_smalll
--redirect __iar_LLog=__iar_Log_smalll
```

関数 sin、cos、__iar_Sin のいずれかをリダイレクトする場合は、3 つすべての関数をリダイレクトする必要があります。

関数 log、log2、log10、__iar_Log のいずれかをリダイレクトする場合には、4 つすべての関数をリダイレクトする必要があります。

より正確なバージョン

関数 cos、pow、sin、tan、およびヘルプ関数 __iar_Sin と __iar_Pow は、より正確なバージョンがライブラリにあり、これらはより大きい引数の範囲を処理できます。デフォルトのバージョンよりサイズが大きく、低速なのが欠点です。

関数の名前は以下のように構成されます。

```
__iar_xxx_accurate<f|l>
```

f は float の派生型に、l は long double の派生型に使用され、double の派生型にサフィックスは使用されません。

これらの関数を使用するには、リンクする際に次のオプションを使用して、デフォルトの関数名がこれらの名前にリダイレクトされる必要があります。

```
--redirect sin=__iar_sin_accurate
--redirect cos=__iar_cos_accurate
--redirect tan=__iar_tan_accurate
--redirect pow=__iar_pow_accurate
--redirect __iar_Sin=__iar_Sin_accurate
--redirect __iar_Pow=__iar_Pow_accurate

--redirect sinf=__iar_sin_accuratef
--redirect cosf=__iar_cos_accuratef
--redirect tanf=__iar_tan_accuratef
--redirect powf=__iar_pow_accuratef
--redirect __iar_FSin=__iar_Sin_accuratef
--redirect __iar_FPow=__iar_Pow_accuratef

--redirect sinl=__iar_sin_accuratel
--redirect cosl=__iar_cos_accuratel
--redirect tanl=__iar_tan_accuratel
--redirect powl=__iar_pow_accuratel
--redirect __iar_LSin=__iar_Sin_accuratel
--redirect __iar_LPow=__iar_Pow_accuratel
```

関数 sin、cos、__iar_Sin のいずれかをリダイレクトする場合は、3 つすべての関数をリダイレクトする必要があります。

関数 pow または __iar_Pow のいずれかをリダイレクトする場合は、両方の関数をリダイレクトする必要があります。

Assert

オプション［**セミホスティング**］または［**IAR ブレークポイント**］を使用してアプリケーションをリンクした場合、C-SPY は失敗したアサートについて通知を受け取ります。これが必要でない場合は、ソースファイル assert.c をアプリケーションプロジェクトに追加することができます。__aeabi_assert 関数は、assert 通知を生成します。arm¥src¥lib ディレクトリにあるテンプレートコードを使用できます。詳細については、123 ページの *ライブラリモジュールのオーバライド*を参照してください。assert を無効にするには、シンボル NDEBUG を定義する必要があります。

IDE では、このシンボル NDEBUG がリリースプロジェクトにデフォルトで定義されており、デバッグプロジェクトには*定義されていません*。コマンドラインでビルドする場合は、必要に応じてこのシンボルを明示的に定義する必要があります。426 ページの *NDEBUG* を参照してください。

Atexit

リンカは、atexit 関数呼出しの静的メモリエリアを割り当てます。デフォルトでは、atexit 関数の呼出し数は 32 バイトに制限されています（100 ページの *atexit 制限の設定*を参照）。

マルチスレッド環境の管理

マルチスレッド環境において、標準ライブラリは、スレッドにとってグローバルかローカルに応じてすべてのライブラリオブジェクトを処理する必要があります。あるオブジェクトが本当にグローバルなオブジェクトである場合、そのオブジェクトの状態の更新はすべてロックメカニズムによってガードし、どんなときにも 1 つのスレッドのみが更新できるようにする必要があります。オブジェクトがスレッドに対してローカルである場合、そのオブジェクトの状態を含む静的変数は、そのスレッドのローカルの変数領域に存在しなければなりません。この領域を一般的に*スレッドのローカル記憶* (TLS) といいます。

マルチスレッドをサポートするビルド済みのライブラリは、製品のインストールに付属しています。

ロックと TLS の下位レベルの実装はシステムに固有のものであり、DLIB ライブラリには含まれていません。RTOS を使用している場合は、必要な機能の一部またはすべてが備わっているか確認してください。備わっていない場合は、自分で用意する必要があります。

DLIB ライブラリでのマルチスレッドのサポート

DLIB ライブラリは、2 種類のロック（*システムロック*と*ファイルストリームロック*）を使用します。ファイルストリームロックは、ファイルストリームの状態が更新されたときにガードとして使用され、フルライブラリ設定でのみ必要になります。以下のオブジェクトは、システムロックによってガードされます。

- ヒープ。つまり malloc、new、free、delete、realloc、calloc が使用される場合。
- ファイルシステム（フルライブラリ設定でのみ使用可能）。ただし、ファイルストリーム自体を除きます。ストリームが開かれたり閉じられたとき

にファイルシステムが更新されます。つまり、fopen、fclose、fdopen、fflush、freopen が使用されるときです。

- シグナルシステム。つまり signal が使用されるときです。
- 一時ファイルシステム。つまり tmpnam が使用されるときです。
- 静的記憶寿命を持ち、動的に初期化された関数ローカルオブジェクト。

以下のライブラリオブジェクトは TLS を使用します。

TLS を使用するライブラリオブジェクト	以下の関数が使用される場合
エラー関数	errno, strerror
ロケール関数	localeconv, setlocale
時間関数	asctime, localtime, gmtime, mktime
マルチバイト関数	mbrlen, mbrtowc, mbsrtowc, mbtowc, wcrtomb, wcsrtomb, wctomb
ランダム関数	rand, srand
C++ 例外エンジン	なし
その他の関数	atexit, strtok

表 13: TLS を使用するライブラリオブジェクト

注 : printf/scanf（または任意の派生型）をフォーマッタとともに使用する場合、各フォーマッタはガードされますが、完全な printf/scanf の呼出しはガードされません。

C++ 派生型のいずれかをマルチスレッドをサポートする DLIB ライブラリと併用する場合、コンパイラオプション --guard_calls を使用して、動的イニシャライザを持つ関数 - 静的変数が、複数のスレッドによって同時に初期化されないように注意する必要があります。

マルチスレッドのサポートの有効化

スレッド化されたアプリケーションで使用するために、コマンドライン上でランタイム環境を設定するには、リンカオプション --threaded_lib を使用します。

スレッド化されたアプリケーションで使用するために IDE でランタイム環境を設定するには、**[プロジェクト] > [オプション] > [一般オプション] > [ライブラリ構成] > [ライブラリでスレッドのサポートを有効にする]** を選択します。リンカオプション --threaded_lib と、それに対応するスレッドをサポートするビルド済みライブラリが自動的に使用されます。C++ 派生形のいずれかが使用される場合、IDE は自動的にコンパイラオプション --guard_calls を使用します。

ライブラリ内でビルトインのマルチスレッドサポートを補完するには、以下も実行する必要があります。

- ライブラリのシステムロックインタフェースのコードを実装
- ファイルストリームを使用する場合、ライブラリのファイルストリームロックインタフェースのコードを実装するか、インタフェースをシステムロックインタフェースにリダイレクト（リンカオプション `--redirect` を使用）
- スレッドの作成と破棄、およびライブラリの TLS アクセス方法を処理するコードを実装
- リンカ設定ファイルを適切に修正

必要な関数の宣言およびマクロの定義は、`DLib_Threads.h` ファイルにあります。これは `yvals.h` によってインクルードされます。

注 : サードパーティの RTOS を使用している場合、IAR システムズのツールでマルチスレッドのサポートを有効にする方法については、その会社のガイドラインを確認してください。

システムロックインタフェース

システムロックが機能するためには、このインタフェースが完全に実装されている必要があります。

```
typedef void *__iar_Rmtx;                    /* ロック情報オブジェクト */

void __iar_system_Mtxinit(__iar_Rmtx *); /* システム解放 */
void __iar_system_Mtxdst(__iar_Rmtx *);/* システムロックの破壊 */
void __iar_system_Mtxlock(__iar_Rmtx *); /* システムロックのロック */
void __iar_system_Mtxunlock(__iar_Rmtx *); /* システムロックのロック
                                              解放 */
```

ロックおよびロック解放の実装は、ネストしている呼出しの後も有効でなければなりません。

ファイルストリームロックインタフェース

このインタフェースは、フルライブラリ設定でのみ必要です。ファイルストリームが使用される場合、完全に実装するか、またはシステムロックインタ

フェースにリダイレクトすることもできます。ファイルストリームロックが機能するためには、このインタフェースが実装されている必要があります。

```
typedef void *__iar_Rmtx;                    /* ロック情報オブジェクト */

void __iar_file_Mtxinit(__iar_Rmtx *);/* ファイルロックの初期化 */
void __iar_file_Mtxdst(__iar_Rmtx *);   /* ファイルロックの破壊 */
void __iar_file_Mtxlock(__iar_Rmtx *);  /* ファイルロックのロック */
void __iar_file_Mtxunlock(__iar_Rmtx *); /* ファイルロックのロック
                                            解放 */
```

ロックおよびロック解放の実装は、ネストしている呼出しの後も有効でなければなりません。

DLIB ロックの使用

DLIB ライブラリによって以下の数のロックが存在すると想定されます。

- _FOPEN_MAX — ファイルストリームロックの最大数。これらのロックはフルライブラリ設定でのみ使用されます。つまり、マクロシンボル _DLIB_FILE_DESCRIPTOR と _FILE_OP_LOCKS が両方とも真の場合のみです
- _MAX_LOCK — システムロックの最大数

アプリケーションがほとんどロックを使用しない場合でも、DLIB ライブラリは上記のロックをすべて初期化して破壊します。

初期化および破壊コードについて詳しくは、xsyslock.c を参照してください。

TLS 処理

DLIB ライブラリは、2 種類のスレッドについて TLS メモリエリアをサポートしています。*メインスレッド*（システム起動および終了コードを含む main 関数）と *セカンダリスレッド*です。

メインスレッドの TLS メモリエリアには以下の特徴があります。

- アプリケーションの起動シーケンスによって自動的に作成および初期化されます
- アプリケーションの破壊シーケンスによって自動的に破壊されます
- セクション __DLIB_PERTHREAD にあります
- スレッド化されていないアプリケーションにも存在します

各セカンダリスレッドの TLS メモリエリアには以下の特徴があります。

- 手動で作成および初期化する必要があります
- 手動で破壊しなければなりません
- 手動で割り当てられたメモリエリアに配置されます

ランタイムライブラリでセカンダリスレッドをサポートする必要がある場合、以下の関数をオーバライドする必要があります。

```
void *__iar_dlib_perthread_access(void *symbp);
```

パラメータはアクセスされる TLS 変数へのアドレス（メインスレッドの TLS エリア内）で、現在の TLS エリアにあるシンボルのアドレスを返します。

セカンダリスレッドの作成および破壊には、2 つのインタフェースを使用できます。ヒープ上のメモリエリアを割り当ててそれを初期化する、以下のインタフェースを使用できます。割当て解除の際に、そのエリアのオブジェクトが破壊されて、メモリが解放されます。

```
void *__iar_dlib_perthread_allocate(void);
void  __iar_dlib_perthread_deallocate(void *);
```

または、アプリケーションが TLS の割当てを処理する場合、このインタフェースをメモリエリアにあるオブジェクトの初期化と破壊に使用できます。

```
void __iar_dlib_perthread_initialize(void *);
void __iar_dlib_perthread_destroy(void *);
```

これらのマクロは、セカンダリスレッドを作成および破壊するインタフェースを実装する際に便利です。

マクロ	説明
`__IAR_DLIB_PERTHREAD_SIZE`	TLS メモリエリアに必要なサイズ。
`__IAR_DLIB_PERTHREAD_INIT_SIZE`	TLS メモリエリアのイニシャライザのサイズ。TLS メモリエリアの残り部分を `__IAR_DLIB_PERTHREAD_SIZE` までゼロに初期化する必要があります。
`__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(`*`symbolptr`*`)`	TLS メモリエリアのシンボルへのオフセット。

表 14: TLS 割当てを実装するためのマクロ

TLS 変数に必要なサイズは、アプリケーションでどの DLIB リソースを使用するかによって変わります。

以下はスレッド処理の一例です。

```
#include <yvals.h>

/* スレッドの TLS ポインタ */
void _DLIB_TLS_MEMORY *TLSp;

/* セカンダリスレッドにいますか？ */
int InSecondaryThread = 0;

/* スレッドに対してローカルの TLS メモリエリア
   を割り当て、そのポインタを TLSp に格納 */
void AllocateTLS()
{
  TLSp = __iar_dlib_perthread_allocate();
}

/* スレッドに対してローカルの TLS メモリエリアの割当解除 */
void DeallocateTLS()
{
  __iar_dlib_perthread_deallocate(TLSp);
}

/* スレッドに対してローカルの
   TLS メモリエリアにあるオブジェクトにアクセス */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
  void _DLIB_TLS_MEMORY *symbp)
{
  char _DLIB_TLS_MEMORY *p = 0;
  if (InSecondaryThread)
    p = (char _DLIB_TLS_MEMORY *) TLSp;
  else
    p = (char _DLIB_TLS_MEMORY *)
      __segment_begin("__DLIB_PERTHREAD");

  p += __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(symbp);
  return (void _DLIB_TLS_MEMORY *) p;
}
```

TLSp 変数は各スレッドに対して一意であり、スレッドの切替えが発生したときに必ず RTOS または手動で交換される必要があります。

リンカ設定ファイルにおける TLS

通常は、リンカは静的データの初期化方法を自動的に選択します。スレッドが使用される場合、メインスレッドの TLS メモリエリアは通常のコピーによって初期化される必要があります。これは、各セカンダリスレッドの TLS

メモリエリアについてもイニシャライザが使用されるためです。これは、リンカ設定ファイルの以下の文によって制御されます。

```
initialize by copy with packing = none {section __DLIB_PERTHREAD };
```

モジュールの整合性チェック

ここでは、ランタイムモデル属性の概念について概要を説明します。これは、IAR システムズが提供するツールで使用されるメカニズムで、アプリケーションにリンクされているモジュールに互換性があること、つまり互換性のある設定を使用してビルドされていることを確認します。ツールでは、定義済のランタイムモデル属性のセットを使用します。これらに加えて、互換性のないモジュールが一緒に使用されないようにするため、独自のものを定義することができます。

注 : 定義済の属性のほかに、AEABI ランタイム属性に対しても互換性がチェックされます。これらの属性は、主にオブジェクトコードの互換性を対象にします。コンパイル設定を反映して、ユーザ設定はできません。

ランタイムモデル属性

ランタイム属性は、名前付きのキーと対応する値のペアで構成されます。一般的に、2 つのモジュールの両方で定義されている各キーの値が同一の場合にのみ、これらのモジュールをリンクできます。

例外が 1 つあります。属性の値が * の場合は、その属性は任意の値に一致します。これをモジュールで指定して、整合性プロパティが考慮されていることを示すことができます。これにより、モジュールがその属性に依存しないことが保証されます。

注 : IAR の定義済ランタイムモデル属性の場合、リンカはいくつかの方法でそれらをチェックします。

例

以下の表では、オブジェクトファイルで `color` と `taste` の 2 つのランタイム属性を定義可能であること（ただし必須ではない）が示されています。

オブジェクトファイル	色	taste
`file1`	`blue`	**未定義**
`file2`	`red`	**未定義**
`file3`	`red`	*

表 15: ランタイムモデル属性の例

オブジェクトファイル	色	taste
file4	red	spicy
file5	red	lean

表 15: ランタイムモデル属性の例（続き）

この場合は、file1 は、ランタイム属性 color が他のファイルと一致しないため、他のファイルとはリンクできません。また、file4 と file5 は、taste ランタイム属性が一致しないため、一緒にリンクすることはできません。

一方で、file2 と file3 は互いにリンクできます。file4 または file5 のどちらかとはリンクできますが、両方とリンクすることはできません。

ランタイムモデル属性の使用

他のオブジェクトファイルとのモジュール整合性を保証するには、#pragma rtmodel ディレクティブを使用して、ランタイムモデル属性を C/C++ ソースコードに指定してください。たとえば、2 つのモードで実行できる UART がある場合は、uart などのランタイムモデル属性を指定できます。モードごとに、mode1、mode2 などのように値を指定します。UART が特定のモードであることを前提とする各モジュールで、これを宣言する必要があります。モジュールの 1 つは以下のようになります。

```
#pragma rtmodel="uart", "mode1"
```

または、rtmodel アセンブラディレクティブを使用して、ランタイムモデル属性をアセンブラソースコードに指定することもできます。次に例を示します。

```
        rtmodel "uart", "mode1"
```

先頭に 2 つの下線を持つ名前は、コンパイラで予約済です。構文について詳しくは、「374 ページの *rtmodel*」と *『ARM 用 IAR アセンブラリファレンスガイド』* をそれぞれ参照してください。

IAR ILINK リンカは、ランタイム属性が衝突するモジュールが同時に使用されないようにすることで、リンク時にモジュール整合性をチェックします。衝突が検出された場合は、エラーが発生します。

アセンブラ言語インタフェース

- C 言語とアセンブラの結合
- C からのアセンブラルーチンの呼出し
- C++ からのアセンブラルーチンの呼出し
- 呼出し規約
- コールフレーム情報

C 言語とアセンブラの結合

ARM 用 IAR C/C++ コンパイラには、低レベルのリソースにアクセスする方法がいくつか用意されています。

- アセンブラだけで記述したモジュール
- 組込み関数（C の代替関数）
- インラインアセンブラ

単純なインラインアセンブラが使用される傾向があります。ただし、どの方法を使用するかは慎重に選択する必要があります。

組込み関数

コンパイラは、アセンブラ言語を必要とせずに低レベルのプロセッサ処理に直接アクセスできる定義済関数をいくつか提供しています。これらの関数を、組込み関数と呼ます。組込み関数は、時間が重要なルーチンなどで非常に便利です。

組込み関数は、通常の関数呼出しと変わらないように見えますが、実際にはコンパイラが認識する組込み関数です。組込み関数は、単一の命令か短い命令シーケンスとして、インラインコードにコンパイルされます。

使用可能な組込み関数の詳細は、「*組込み関数*」を参照してください。

C 言語とアセンブラモジュールの結合

アプリケーションの一部をアセンブラで記述し、C/C++ モジュールと混在させることができます。

これは関数呼び出しの形式でオーバーヘッドになったり、命令シーケンスを返したりする原因になります。またコンパイラがスクラッチレジスタとしていくつかのレジスタをみなします。多くの場合、外部命令によるオーバヘッドは、オプティマイザにより削 除されます。

重要な利点は、コンパイラが生成したものとアセンブラに自分で書いたものの間でよく定義されたインタフェースを持つことができることです。インラインアセンブラを使用するときは、ご自分のインラインアセンブラがコンパイラが生成したコードに邪魔しないという保証は一切ありません。

アプリケーションで、一部をアセンブラ言語、一部を C/C++ で記述する場合、多くの疑問点に遭遇します。

- C から呼出せるようにアセンブラコードを記述する方法は ?
- アセンブラコードのパラメータの場所は ? また、呼出し元へ値を返す方法は ?
- C で記述した関数をアセンブラコードから呼出す方法は ?
- アセンブラ言語で記述したコードから、C のグローバル変数にアクセスする方法は ?
- アセンブラコードをデバッグする際に、デバッガでコールスタックが表示されない理由は ?

最初の疑問については、161 ページの *C からのアセンブラルーチンの呼出し*で説明します。2 番目と 3 番目の疑問については、164 ページの *呼出し規約*で説明します。

最後の疑問については、アセンブラコードをデバッガで実行する際、コールスタックを表示できるというのが答えです。ただし、デバッガではコールフレームについての情報が必要になります。この情報は、アセンブラソースファイルでコメントとして記述する必要があります。詳細については、171 ページの *コールフレーム情報*を参照してください。

C/C++ とアセンブラモジュールを混在させる望ましい方法は、161 ページの *C からのアセンブラルーチンの呼出し*と 163 ページの *C++ からのアセンブラルーチンの呼出し*でそれぞれ説明しています。

インラインアセンブラ

インラインアセンブラを使用して、アセンブラ命令を C/C++ 関数に直接挿入できます。通常これは以下の必要がある場合に便利です。

- C でアクセスできないハードウェアリソースにアクセスする（つまり、SFR の定義がなかったり、適切な組込み関数がない場合）。
- C で記述するとタイミングが適切でなくなる時間が重要なコードのシーケンスを手動で記述する。
- C で記述すると遅くなりすぎる速度が重要なコードのシーケンスを手動で記述する。

インラインアセンブラの文は、引数が入力可能（入力オペランド）でリターン値があり（出力オペランド）、C シンボルのリードやライトが可能（オペランドを介して）という点で、C 関数に似ています。インラインアセンブラ文は、*上書きされるリソース*（つまり、レジスタやメモリ内のオーバライドされた値）を宣言することもできます。

制限

通常のアセンブラ言語でできることの多くが、インラインアセンブラでも可能です。違う点は以下のとおりです。

- アラインメントは制御できません。つまり、`DC32` などのディレクティブの位置が誤ってアラインメントされることがあります。
- 認められているレジスタの同義語は、`SP` (`R13`)、`LR` (`R14`)、`PC` (`R15`) のみです。
- 一般的に、アセンブラディレクティブはエラーを発生させるか、何の意味も持ちません。ただし、データ定義ディレクティブは予期したとおりに機能します。
- 使用されているリソース（レジスタ、メモリなど）で、C コンパイラでも使用されているものは、オペランドまたは上書きされるリソースとして宣言する必要があります。
- インラインアセンブラの文がコンパイラによって最適化されすぎないようにするには、`volatile` として宣言してください。
- C シンボルにアクセスしたり、定数式を使用するにはオペランドを使用する必要があります。
- オペランドの式間の依存関係によって、エラーが発生することがあります。
- 擬似命令 `LDR Rd, =`*`expr`* は、インラインアセンブラでは使用できません。

インラインアセンブラに関するリスク

オペランドや上書きされるリソースがなければ、インラインアセンブラの文には周囲の C ソースコードとのインタフェースがありません。このため、インラインアセンブラコードが脆弱になります。また、将来コンパイラを更新した場合に保守面で問題が生じる可能性もあります。オペランドや上書きさ

れるリソースのないインラインアセンブラの使用には、いくつかの制約もあります。

- コンパイラによる最適化では、インライン文の効果を無視します。これらはまったく最適化されません。
- インラインアセンブラの文は volatile となり、*上書きされるメモリ*を暗黙的に示します。つまり、コンパイラはアセンブラ文を削除しません。単純にプログラムフローの指定位置に挿入されます。挿入による前後のコードへの影響や副作用は考慮されません。たとえば、レジスタやメモリ位置が変更される場合は、それ以降のコードが正常に動作するには、インラインアセンブラ命令のシーケンス内での復元が必要になることがあります。

次の例（ARM モードの場合）は、オペランドおよび上書きされない asm キーワードを使用する場合のリスクを示します。

```
int Add(int term1, int term2)
{
  asm("adds r0,r0,r1");
  return term1;
}
```

この例の場合：

- 関数 Add は、値がレジスタ内で渡されて戻されると想定します。たとえば、関数がインライン化されている場合は、常にそうはならない可能性があるというような方法です。
- adds 命令の s は、条件フラグが更新されていることを示します。これは cc の上書きされるオペランドを使用して指定します。それ以外の場合、コンパイラは条件フラグが変更されていないと想定します。

したがって、オペランドや上書きされるリソースを使用しないインラインアセンブラは、極力避けるようにしてください。

インラインアセンブラのリファレンス情報

キーワード asm および __asm は、いずれもインラインアセンブラ命令を挿入します。ただし、C ソースコードのコンパイル時に --strict オプションを使用している場合は、asm キーワードは使用できません。__asm キーワードはいつでも使用できます。

構文

インラインアセンブラ文の構文は以下のとおりです（GNU gcc で使用されるものに類似しています）。

```
asm [volatile]( string [assembler-interface])
```

string は、1 つ以上の有効なアセンブラ命令またはデータ定義アセンブラディレクティブを、¥n で区切って含めることができます。

次に例を示します。

```
asm("label:nop¥n"
    "b label");
```

インラインアセンブラ命令では、ローカルラベルを定義して使用できます。

assembler-interface は以下のようになります。

```
  : カンマで区切られた出力オペランドのリスト      /* オプション */
  : カンマで区切られた入力オペランドのリスト      /* オプション */
  : カンマで区切られた上書きリソースのリスト      /* オプション */
```

オペランド

インラインアセンブラ文は、1 つの入力と、1 つの出力のコンマ区切りのオペランドを持つことができます。それぞれのオペランドは、括弧内にオプションのシンボル名と、引用符で囲まれた制約、その後に丸括弧に入った C の式が続きます。

オペランドの構文

[[*シンボル名*]] "［*修飾子*］*制約*"（*C の式*）

次に例を示します。

```
int Add(int term1, int term2)
{
  int sum;

  asm("add %0,%1,%2"
      : "=r"(sum)
      : "r" (term1), "r" (term2));

  return sum;
}
```

この例では、アセンブラ命令は sum という 1 つの出力オペランドと、term1 および term2 という 2 つの入力オペランドを使用し、上書きされるリソースは使用しません。

すべてのリストは空にすれば省略可能です。次に例を示します。

```
int matrix[M][N];

void MatrixPreloadRow(int row)
{
  asm volatile ("pld [%0]" : : "r" (&matrix[row][0]));
}
```

オペランドの制約

制約	説明
r	式に汎用目的のレジスタを使用します。 R0-R12、R14（ARM および Thumb2 の場合） R0-R7（Thumb1 の場合）
l	R0-R7（Thumb1 の場合にのみ有効）。
Rp	R0、R1 のように汎用目的のレジスタの組合せを使用します。
i	定数値を持つイミディエイト整数オペランド。シンボル定数を使用できます。
j	MOVW 命令に適した 16 ビットの定数（ARM と Thumb2 で使用可）。
n	イミディエイトオペランドで、i のエイリアスです。
I	データ処理命令で有効なイミディエイト定数（ARM および Thumb2）、または 0 ～ 255 の定数 (Thumb1)。
J	-4095 ～ 4095 のイミディエイト定数（ARM および Thumb2）、または -255 ～ -1 の定数 (Thumb1)。
K	反転されている場合に I 定数を満たすイミディエイト定数（ARM および Thumb2）、または 2 の累乗の I 制約を満たす定数 (Thumb1)。
L	否定された場合に I 制約を満たすイミディエイト定数（ARM および Thumb2）、または -7 ～ 7 の定数 (Thumb1)。
M	4 の倍数で 0 ～ 1020 のイミディエイト定数（Thumb1 でのみ有効）。
N	0 ～ 31 のイミディエイト定数（Thumb1 でのみ有効）。
O	4 の倍数で -508 ～ 508 のイミディエイト定数（Thumb1 でのみ有効）。
t	S レジスタ。
w	D レジスタ。
q	Q レジスタ。
Dv	VMOV.F32 命令の 32 ビット浮動小数点イミディエイト定数。
Dy	VMOV.F64 命令の 64 ビット浮動小数点イミディエイト定数。
v2S ... v4Q	2、3、4 の連続した S、D、または Q のレジスタ。たとえば、v4Q は、4 つの Q レジスタのベクタです。ベクタは重複しないため、使用可能な v4Q レジスタのベクタは Q0-Q3、Q4-Q7、Q8-Q11、Q12-Q15 です。

表 16: インラインアセンブラのオペランドの制約

制約修飾子

制約修飾子を制約とともに使用して、意味を変更することができます。この表は、サポートされている制約修飾子の一覧です。

修飾子	説明
=	ライトのみのオペランド
+	リード / ライトのオペランド
&	命令がすべての入力オペランドを処理する前に書き込まれた、早い上書きされる出力オペランド

表 17: サポートされている制約修飾子

オペランドの参照

アセンブラ命令は、順序番号の先頭に % を付けることでオペランドを参照します。最初のオペランドは順序番号が 0 となり、%0 によって参照されます。

オペランドにシンボル名がある場合、構文 %[*operand.name*] を使用してそれを参照できます。シンボルオペランド名は C/C++ コードとは別の名前空間にあり、C/C++ 変数名と同じにすることも可能です。ただし、各オペランド名はそれぞれのアセンブラ文で一意でなければなりません。次に例を示します。

```
int Add(int term1, int term2)
{
  int sum;

  asm("add %[Rd],%[Rn],%[Rm]"
      : [Rd]"=r"(sum)
      : [Rn]"r" (term1), [Rm]"r" (term2));

  return sum;
}
```

オペランドの修飾子

オペランドの修飾子は、% からオペランド番号までの 1 文字で、オペランドを変換するときに使用されます。

次の例では、修飾子 L と H を使用して、イミディエイトオペランドの最下位と最上位の 16 ビットにそれぞれアクセスします。

```
int Mov32()
{
  int a;
  asm("movw %0,%L1 ¥n"
      "movt %0,%H1 ¥n" : "=r"(a) : "i"(0x12345678UL));
  return a;
}
```

一部のオペランド修飾子は組み合わせることができます。この場合、それぞれの文字によって前の修飾子の結果が変換されます。次の表は、それぞれの有効な修飾子によって実行される変換を示します。

修飾子	説明
L	レジスタの組合せの最も番号の小さいレジスタ、またはイミディエイト定数の最下位の 16 ビット。
H	レジスタの組合せの最も番号の大きいレジスタ、またはイミディエイト定数の最上位の 16 ビット。
c	イミディエイトオペランドの場合、前に # 符号のつかない整数またはシンボルのアドレス。その他のオペランド修飾子によって実行することはできません。
B	イミディエイトオペランドの場合、前に # 符号のつかない整数またはシンボルのビット単位の反転。その他のオペランド修飾子によって実行することはできません。
Q	レジスタの組合せで最下位のレジスタ。
R	レジスタの組合せで最上位のレジスタ。
M	1 つのレジスタまたはレジスタの組合せの場合、ldm または stm に適したレジスタのリスト。その他のオペランド修飾子によって実行することはできません。
a	レジスタ Rn を pld に適したメモリオペランド [Rn,#0] に変換します。
b	D レジスタの下位 S レジスタ部分。
p	D レジスタの上位の S レジスタ部分。
e	Q レジスタの下位 D レジスタ部分、または Neon レジスタのベクタにおける下位レジスタ。
f	Q レジスタの上位 D レジスタ部分、または Neon レジスタのベクタにおける上位レジスタ。
h	D レジスタ（の配列）または Q レジスタの場合、中カッコ内の対応する D レジスタのリスト。たとえば、Q0 は {D0,D1} となります。その他のオペランド修飾子によって実行することはできません。
y	D レジスタとしてインデックス化された S レジスタ。たとえば、S7 は D3[1] となります。その他のオペランド修飾子によって実行することはできません。

表 18: オペランド修飾子と変換

入力オペランド

入力オペランドは修飾子を持つことはできませんが、有効な C 式を持つことはできます（式の型がレジスタに合っていることが条件）。

C 式はインラインアセンブラ文でアセンブラ命令の直前に評価され、レジスタなどの制約に割り当てられます。

出力オペランド

出力オペランドは修飾子として = を持つ必要があり、C 式は l 値で、ライト可能な場所を指定する必要があります。たとえば、ライトのみの汎用レジスタの場合、=r とします。制約は、インラインアセンブラ文の最後のアセンブラ命令の直後に、評価済みの C 式に（l 値として）割り当てられます。入力オペランドは、出力が生成される前に消費されるものと想定され、コンパイラは入力と出力のオペランドに同じレジスタを使用してもかまいません。これを禁止するには、出力制約のプレフィックスを & として、=&r のように早い上書きされるリソースにします。こうすることで、出力オペランドが入力オペランドとは異なるレジスタに確実に割り当てられます。

入出力オペランド

入力と出力の両方に使用されるべきオペランドは、出力オペランドとしてリストし、修飾子 + を付ける必要があります。C 式は l 値でなければならず、ライト可能な場所を指定する必要があります。この位置はアセンブラ命令の直前に読み込まれ、最後のアセンブラ命令の直後に書き込まれます。

これはリード / ライトのオペランドを使用した例です。

```
int Double(int value)
{
  asm("add %0,%0,%0" : "+r"(value));

  return value;
}
```

この例では、value の入力値は汎用レジスタ内に配置されます。アセンブラ文の後、ADD 命令の結果が同じレジスタに配置されます。

上書きされるリソース

インラインアセンブラ文は、上書きされるリソースのリストを持つことができます。

"*resource1*", "*resource2*", ...

上書きされるリソースを指定して、インラインアセンブラ文によってどのリソースが上書きされるかをコンパイラに伝えます。上書きされるリソース内のすべての値で、インラインアセンブラ文の後に必要なものは再びロードされます。

上書きされるリソースは、入力または出力オペランドとしては使用されません。

これは、上書きされるリソースの使用方法の一例です。

```
int Add(int term1, int term2)
{
  int sum;

  asm("adds %0,%1,%2"
      : "=r"(sum)
      : "r" (term1), "r" (term2)
      : "cc");

  return sum;
}
```

この例では、条件コードが ADDS 命令によって変更されるため、"cc" を上書きされるリストに記載する必要があります。

この表は、有効な上書きされるリソースの一覧です。

上書き	説明
R0-R12、R14（ARM モードおよび Thumb2 の場合） R0-R7、R12、R14（Thumb1 の場合）	汎用レジスタ
S0-S31、D0-D31、Q0-Q15	浮動小数点レジスタ
cc	条件フラグ (N、Z、V、C)
メモリ	命令が何らかのメモリを変更する場合に使用します。これによって、インラインアセンブラ文でメモリ値がレジスタ内にキャッシュとして保存されないようにします。

表 19: 有効な上書きされるリソースの一覧

上書きされるメモリの使用方法の例

```
int StoreExclusive(unsigned long * location, unsigned long value)
{
  int failed;

  asm("strex %0,%2,[%1]"
      : "=&r"(failed)
      : "r"(location), "r"(value)
      : "memory");

  /* 注 : 'strex' には ARMv6 (ARM) または ARMv6T2 (THUMB) が必要 */

  return failed;
}
```

C からのアセンブラルーチンの呼出し

C から呼出すアセンブラルーチンは、以下を満たしている必要があります。

- 呼出し規約に準拠していること
- PUBLIC エントリポイントラベルがあること
- 型チェックやパラメータの型変換（オプション）を可能にするため、以下の例のようにすべての呼出しの前に external として宣言されていること

  ```
  extern int foo(void);
  ```

  または

  ```
  extern int foo(int i, int j);
  ```

これらの条件を満たすには、C でスケルトンコードを作成してコンパイルし、アセンブラリストファイルを調べるという方法があります。

スケルトンコードの作成

正しいインタフェースを持つアセンブラ言語ルーチンを作成するには、C コンパイラで作成されたアセンブラ言語ソースファイルから開始することをお勧めします。関数プロトタイプごとにスケルトンコードを作成する必要があります。

以下の例は、ルーチン本体を簡単に追加できるスケルトンードの作成方法を示します。スケルトンソースコードで必要な処理は、必要な変数を宣言し、それらにアクセスするだけです。この例では、アセンブラルーチンは int、char を引数に指定し、int を返します。

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
  int locInt = arg1;
  gInt = arg1;
  gChar = arg2;
  return locInt;
}

int main()
{
  int locInt = gInt;
  gInt = Func(locInt, gChar);
  return 0;
}
```

注 : この例では、ローカル変数とグローバル変数のアクセスを示すため、コードのコンパイル時の最適化レベルを低くしています。最適化レベルを高くすると、ローカル変数への必要な参照が最適化で削除される場合があります。最適化レベルによって実際の関数宣言が変更されることはありません。

スケルトンコードのコンパイル

IDE において、リストオプションをファイルレベルで指定します。［ワークスペース］ウィンドウでファイルを選択します。次に、**［プロジェクト］>［オプション］**を選択します。**［C/C++ コンパイラ］**カテゴリで、**［継承した設定をオーバライド］**を選択します。**［リスト］**ページで **［リストファイルの出力］**の選択を解除し、代わりに **［アセンブラファイルの出力］**オプションおよびそのサブオプション **［ソースのインクルード］**を選択します。また、低い最適化レベルを指定します。

スケルトンコードをコンパイルするには、以下のオプションを使用します。

```
iccarm skeleton.c -lA . -On -e
```

`-lA` オプションは、アセンブラ言語出力ファイルを作成します。このファイルでは、C/C++ ソース行がアセンブラのコメントとして記述されています。.（ピリオド）は、アセンブラファイル名を C/C++ モジュール (`skeleton`) と同様の方法で設定し、拡張子のみを `s` に変更するように指定します。`-On` オプションは、最適化が使用されないことを意味し、`-e` は言語拡張を有効なことを意味します。さらに、関連のコンパイラオプションを必ず使用してください。通常ご自分のプロジェクトの他の C/C++ ソースファイルで使用するものと同じです。

その結果、アセンブラソース出力ファイル `skeleton.s` が生成されます。

注 : `-lA` オプションは、呼出しフレーム情報 (`CFI`) ディレクティブを含むリストファイルを作成します。これは、これらのディレクティブと使用方法について調べる意図がある場合に便利です。呼出し規約のみを調べるのであれば、`CFI` ディレクティブをリストファイルから除外できます。

IDE で、**［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［リスト］**を選択し、サブオプションの **［コールフレーム情報のインクルード］**を選択解除します。

コマンドラインで、-lA ではなく `-lB` を使用します。C-SPY Call Stack ウィンドウを機能させるためには、ソースコードに `CFI` 情報が含まれている必要があります。

出力ファイル

出力ファイルには、以下の重要情報が含まれています。

- 呼出し規約

- リターン値
- グローバル変数
- 関数パラメータ
- スタック（自動変数）空間を作成する方法
- コールフレーム情報 (CFI)

CFI ディレクティブは、デバッガの［コールスタック］ウィンドウで必要なコールフレーム情報を記述します。詳細については、171 ページの *コールフレーム情報*を参照してください。

C++ からのアセンブラルーチンの呼出し

C の呼出し規約は、C++ 関数には適用されません。最も重要なことは、関数名だけでは C++ 関数を特定できない点です。型安全なリンケージを保証し、オーバロードを解決するために、関数のスコープと型も必要になります。

もう 1 つの違いとして、非静的メンバ関数が、別の隠し引数である this ポインタを取る点があります。

ただし、C リンケージを使用する場合は、呼出し規約は C の呼出し規約に準拠します。したがって、アセンブラルーチンは以下の方法で宣言した場合に C++ から呼出されます。

```
extern "C"
{
  int MyRoutine(int);
}
```

以下の例は、非静的メンバ関数と同等の処理を実現する方法を示します。すなわち、暗黙的な this ポインタを明示する必要があります。アセンブラルーチンの呼出しをメンバ関数内にラップできます。関数インライン化が有効に

なっていれば、インラインメンバ関数を使用することで、余分な呼出しによるオーバヘッドが解消されます。

```
class MyClass;

extern "C"
{
  void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
  inline void DoIt(int arg)
  {
    ::DoIt(this, arg);
  }
};
```

呼出し規約

呼出し規約とは、プログラム内の関数が別の関数を呼出す方法を規定したものです。コンパイラはこれを自動的に処理しますが、関数をアセンブラ言語で記述している場合は、そのパラメータの位置や特定方法、呼出されたプログラム位置に戻る方法、結果を返す方法がわかっている必要があります。

また、アセンブラレベルのルーチンがどのレジスタを保存する必要があるかを知ることも重要です。プログラムが保存するレジスタが多すぎると、効率が低下する場合があります。保存するレジスタが少なすぎると、不正なプログラムになる可能性があります。

ここでは、コンパイラで使用される呼出し規約について説明します。内容は以下のとおりです。

- 関数の宣言
- C/C++ のリンケージ
- 保護レジスタとスクラッチレジスタ
- 関数の入口
- 関数の終了
- リターンアドレスの処理

最後に、実際の呼出し規約の例を示します。

コンパイラで使用される呼出し規約は AEABI の一部である AAPCS (Procedure Call Standard for the ARM Architecture) に準拠します。詳細は 207 ページの

AEABI への*準拠*を参照してください。ここでは AAPCS については、概略のみ説明します。たとえば、VFP 呼出し規約を使用する際の浮動小数点コプロセッサレジスタの使用については、省略します。

関数の宣言

C では、コンパイラで関数の呼出し方法を特定できるように、関数は規則に沿って宣言する必要があります。宣言の例を次に示します。

```
int MyFunction(int first, char * second);
```

この宣言は、整数と文字へのポインタの 2 つのパラメータを関数で指定することを示します。この関数は、整数を返します。

通常は、コンパイラが関数について特定できるのはこれだけです。したがって、この情報から呼出し規約を推定できる必要があります。

C++ ソースコードでの C リンケージの使用

C++ では、関数は C または C++ のいずれかのリンケージを持つことができます。アセンブラルーチンを C++ から呼出すには、C++ 関数に C リンケージを持たせるのが最も簡単です。

C リンケージを持つ関数の宣言例を示します。

```
extern "C"
{
  int F(int);
}
```

多くの場合は、ヘッダファイルを C と C++ で共有するのが実用的です。C と C++ の両方で C リンケージを持つ関数を宣言する例を示します。

```
#ifdef __cplusplus
extern "C"
{
#endif

int F(int);

#ifdef __cplusplus
}
#endif
```

保護レジスタとスクラッチレジスタ

通常の ARM CPU のレジスタは、以下で説明する 3 種類に分類されます。

スクラッチレジスタ

スクラッチレジスタの内容は、任意の関数により破壊される可能性があります。関数が別の関数の呼出した後もレジスタの値を必要とする場合は、呼出し中はその値をスタックなどで保存する必要があります。

レジスタ R0 ～ R3 のすべてと R12 は、スクラッチレジスタとして関数で使用できます。ベニアのために自動的に挿入される命令のため、R12 は、アセンブラ関数間の呼出し時にもスクラッチレジスタとみなされるので注意してください。

保護レジスタ

一方、保護レジスタは、他の関数の呼出し後も保持されます。呼出された関数は保護レジスタを他の用途で使用できますが、使用前に値を保存し、関数終了時に値を復元する必要があります。

レジスタ R4 ～ R11 が保護レジスタです。これらは、呼出し先関数で保持されます。

専用レジスタ

レジスタによっては、考慮すべき特別な要件があります。

- スタックポインタレジスタ R13/SP は、常にスタック上の最後のエレメントかその下の位置を示します。割込みが発生すると、スタックポインタが示す位置より下のエレメントはすべて上書きされます。関数の入り口および終了位置では、スタックポインタは 8 バイトに整列されている必要があります。関数内では、スタックポインタは常にワード整列されていなければなりません。終了位置では、SP の値はエントリ位置の値と同じである必要があります。
- レジスタ R15/PC は、プログラムカウンタ専用です。
- リンクレジスタ R14/LR は、関数の入口にリターンアドレスを保持します。

関数の入口

これらの基本的な方法ひとつを使用して、パラメータを関数に渡すことができます。

- レジスタ内
- スタック上

メモリ経由で迂回して引き渡すよりもレジスタを使用する方が大幅に効率的です。そのため、呼出し規約では可能な限りレジスタを使用するように規定されています。パラメータの引き渡しには制限されたレジスタ数だけが使用

できます。利用できるレジスタがなくなった時は、残りのパラメータがスタックに引渡されます。これらの例外にはルールが適用されます。

- パラメータを受け入れて値を返すソフトウェア割込み関数を除いて、割込み関数にはどんなパラメータも指定できません。
- ソフトウェア割込み関数は、通常の関数と同じ様にはスタックを使用できません。SVC 命令が実行されると、プロセッサはスーパーバイザモードに切り替わり、スーパーバイザスタックが使用されるので、引数は、アプリケーションが割込み発生前にスーパーバイザモードで実行していない場合、スタックに渡すことはできません。

隠しパラメータ

関数の宣言や定義で明示されるパラメータに加えて、隠しパラメータが存在する場合があります。

- 関数より返された構造体が 32 ビットを超えている場合、その構造体が格納されるメモリ位置は、追加パラメータとして渡されます。これは、常に*最初のパラメータ*として扱われることに注意してください。
- 関数が非静的 C++ メンバ関数の場合、this ポインタが最初のパラメータとして渡されます（ただし、1 つのみの場合、配置されるのは構造体ポインタのリターン後）。詳細については、161 ページの *C からのアセンブラルーチンの呼出し*を参照してください。

レジスタパラメータ

パラメータの引渡しに利用可能なレジスタ

パラメータ	レジスタでの引渡し
32 ビット以下のスカラ値と浮動小数点値、および単精度（32 ビット）浮動小数点値	最初の空きレジスタを使用して渡されます : R0-R3
long long および倍精度（64 ビット）値	最初に使用できるレジスタペアで渡されます : R0:R1、R2:R3

表 20: パラメータの引渡しに使用されるレジスタ

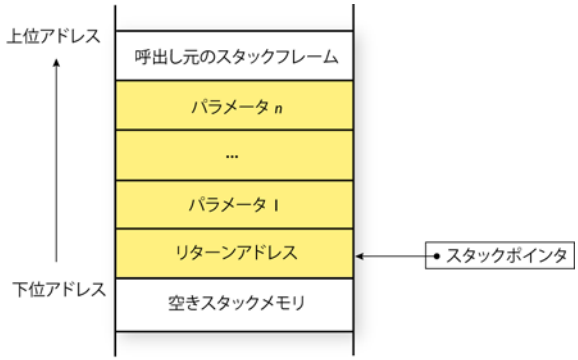
レジスタをパラメータに割り当てるプロセスは単純明快です。パラメータを左から右の順に調べ、最初のパラメータを空きレジスタに代入します。空きレジスタがない場合は、パラメータはスタックを逆方向で使用して引き渡されます。

32 ビット未満のパラメータを持つ関数が呼出される場合、未使用ビットの値が一貫するように、値は符号拡張またはゼロ拡張されます。値が符号拡張またはゼロ拡張されるかどうかは、そのタイプ signed または unsigned により異なります。

スタックパラメータとレイアウト

スタックパラメータは、メモリのスタックポインタが指す位置を開始位置として格納されています。スタックポインタ以下（下位メモリ方向）には、呼出し先関数が使用可能な空きエリアがあります。最初のスタックパラメータは、スタックポインタが指す位置に格納されています。それ以降のスタックパラメータは、スタック上の 4 で割り切れる次の位置に順に格納されます。呼出された関数がリターンした後スタックをクリーンするのは、呼出し元で行うべきです。

次の図は、パラメータがスタック上に格納される様子を示します。



関数の終了

関数は、呼出し元の関数やプログラムに値を返すことができます。または、関数のリターン型が `void` の場合もあります。

関数のリターン値がある場合は、スカラ（整数、ポインタなど）、浮動小数点数、構造体のいずれかになります。

リターン値に使用されるレジスタ

リターン値に使用できるレジスタは、`R0` と `R0:R1` です。

リターン値	レジスタでの引渡し
32 ビット以下のスカラ値と構造体リターン値、および単精度（32 ビット）浮動小数点リターン値	`R0`
32 ビット超の構造体リターン値のメモリアドレス	`R0`
`long long` および倍精度（64 ビット）リターン値	`R0:R1`

表 21: リターン値に使用されるレジスタ

リターン値が 32 ビット未満の場合、値は 32 ビットに符号拡張またはゼロ拡張されます。

関数終了時のスタックのレイアウト

呼出された関数がリターンした後スタックをクリーンするのは、呼出し元で行うべきです。

リターンアドレスの処理

アセンブラ言語で記述した関数は、レジスタ LR によりポイントされるアドレスまでジャンプすることで、終了時に呼出し元に戻ります。

関数の入口で、非スカラレジスタおよび LR レジスタは、1 つの命令でプッシュできます。関数の出口では、これらすべてのレジスタは 1 つの命令でポップできます。リターンアドレスは、PC に直接ポップできます。

以下の例は、この動作を示しています。

```
name    call
section .text:CODE
extern  func

push    {r4-r6,lr}  ; スタックのアラインメント 8 を保持
bl      func

; 何らかの処理

pop     {r4-r6,pc}  ; リターン

end
```

例

以下では、宣言の例や対応する呼出し規約を紹介します。後の例ほど複雑になっています。

例 1

以下の関数が宣言されているとします。

```
int add1(int);
```

この関数は、1 つのパラメータをレジスタ R0 を使用して引き渡し、リターン値をレジスタ R0 を使用して呼出し元に返します。

以下のアセンブラルーチンは、この宣言に適合します。このルーチンは、パラメータの値よりも 1 つ大きな値を返します。

```
name    return
section .text:CODE
adds    r0, r0, #1
bx      lr
end
```

例 2

この例は、構造体がスタックを使用して引き渡される方法を説明しています。以下が宣言されているとします。

```
struct MyStruct
{
  short a;
  short b;
  short c;
  short d;
  short e;
};

int MyFunction(struct MyStruct x, int y);
```

構造値メンバの値 a、b、c、d は、レジスタ R0-R3 で渡されます。最後の構造体メンバ e および整数パラメータ y はスタックで渡されます。呼出し元関数は、スタックの上位 8 バイトを確保し、2 つのスタックパラメータの内容をその位置にコピーします。リターン値は、R0 レジスタを使用して呼出し元に返されます。

例 3

次の関数は、struct MyStruct 型の構造体を返します。

```
struct MyStruct
{
  int mA[20];
};

struct MyStruct MyFunction(int x);
```

リターン値のメモリ位置を割り当てて、それにポインタを最初の隠しパラメータとして引き渡すのは、呼出し元の関数の役割です。リターン値が格納されるべき位置へのポインタは、R0 内で引き渡されます。パラメータ x は R1 で引き渡されます。

関数が構造体へのポインタを返すよう宣言されているとします。

```
struct MyStruct *MyFunction(int x);
```

この場合、リターン値はスカラであり、隠しパラメータはありません。パラメータ x は R0 で引き渡され、リターン値は R0 で返されます。

コールフレーム情報

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

アセンブラ言語で記述したルーチンのデバッグ時にコールスタックを使用できるようにするには、アセンブラディレクティブ CFI を使用して、同等のデバッグ情報をアセンブラソースで提供する必要があります。このディレクティブの詳細は、*『ARM 用 IAR アセンブラリファレンスガイド』*を参照してください。

CFI ディレクティブ

CFI ディレクティブは、呼出し元関数のステータス情報を C-SPY に提供します。この中で最も重要な情報は、リターンアドレスと、関数やアセンブラルーチンのエントリ時点でのスタックポインタの値です。C-SPY は、この情報を使用して、呼出し元関数の状態を復元し、スタックを巻き戻すことができます。

呼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記述するには、以下の 3 つのコンポーネントが必要です。

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

リソース	説明
CFA R13	スタックの呼出しフレーム
R0-R12	プロセッサ汎用 32 ビットレジスタ

表 22: 名前ブロックで定義されている呼出しフレーム情報リソース

リソース	説明
R13	スタックポインタ、SP
R14	リンクレジスタ、LR
D0-D31	ベクトル浮動小数点 (VFP) 64 ビットコプロセッサレジスタ
CPSR	現在のプログラムステータスレジスタ
SPSR	保存されたプログラムステータスレジスタ

表 22: 名前ブロックで定義されている呼出しフレーム情報リソース

CFI サポートを持つアセンブラソースの作成

呼出しフレーム情報を正しく処理するアセンブラ言語ルーチンを作成するには、コンパイラで作成されたアセンブラ言語ソースファイルから開始することをお勧めします。

1 適当な C ソースコードを使用して開始します。以下に例を示します。

```
int F(int);
int cfiExample(int i)
{
  return i + F(i);
}
```

2 C ソースコードをコンパイルします。呼出しフレーム情報（CFI ディレクティブ）を含むリストファイルを必ず作成してください。

コマンドラインでは、-lA オプションを使用します。

IDE で、［**プロジェクト**］>［**オプション**］>［**C/C++ コンパイラ**］>［**リスト**］を選択し、サブオプションの［**呼出しフレーム情報のインクルード**］が選択されていることを確認します。

この例のソースコードの場合、リストファイルは以下のようになります。

```
        NAME Cfi

        RTMODEL "__SystemLibrary", "DLib"

        EXTERN F

        PUBLIC cfiExample

        CFI Names cfiNames0
        CFI StackFrame CFA R13 DATA
        CFI Resource R0:32, R1:32, R2:32, R3:32, R4:32, R5:32,
R6:32, R7:32
        CFI Resource R8:32, R9:32, R10:32, R11:32, R12:32,
R13:32, R14:32
        CFI EndNames cfiNames0

        CFI Common cfiCommon0 Using cfiNames0
        CFI CodeAlign 4
        CFI DataAlign 4
        CFI ReturnAddress R14 CODE
        CFI CFA R13+0
        CFI R0 Undefined
        CFI R1 Undefined
        CFI R2 Undefined
        CFI R3 Undefined
        CFI R4 SameValue
        CFI R5 SameValue
        CFI R6 SameValue
        CFI R7 SameValue
        CFI R8 SameValue
        CFI R9 SameValue
        CFI R10 SameValue
        CFI R11 SameValue
        CFI R12 Undefined
        CFI R14 SameValue
        CFI EndCommon cfiCommon0


        SECTION `.text`:CODE:NOROOT(2)
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        ARM
cfiExample:
        PUSH     {R4,LR}
        CFI R14 Frame(CFA, -4)
        CFI R4 Frame(CFA, -8)
```

```
CFI CFA R13+8
MOVS     R4,R0
MOVS     R0,R4
BL       F
ADDS     R0,R0,R4
POP      {R4,PC}          ;; return
CFI EndBlock cfiBlock0

END
```

注 : ヘッダファイル Common.i は、マクロ CFI_NAMES_BLOCK、CFI_COMMON_ARM、CFI_COMMON_Thumb を含み、これらは一般的な名前ブロックおよび一般的な共通ブロックを各 1 つ宣言します。これらの 2 つのマクロは、仮想リソースと具体的なリソースの両方を宣言します。

C の使用

- C 言語の概要
- 拡張の概要
- IAR C 言語拡張

C 言語の概要

ARM 用 IAR C/C++ コンパイラは、ISO/IEC 9899:1999 規格（最新の技術的誤植 No.3 も含む）、通称 C99 をサポートしています。このガイドでは、この規格を*標準の C* と呼び、これがコンパイラで使用されるデフォルト標準です。この標準は C89 よりも厳密です。

また、コンパイラは ISO 9899:1990 規格（すべての技術的誤植と追加事項を含む）、通称 C94、C90、C89、ANSI C もサポートしています。本ガイドでは、この規格を *C89* といいます。この規格を有効にするには、`--c89` コンパイラオプションを使用します。

C99 規格は C89 から派生したものですが、以下のような特長が追加されています。

- `inline` キーワードは、キーワードの直後に定義された関数をインライン化するようコンパイラに指示します
- 宣言と文は、同じスコープ内で混在させることが可能です
- `for` ループの初期化式における宣言
- `bool` データ型
- `long long` データ型
- 複雑な浮動小数点型
- C++ スタイルのコメント
- 複合リテラル
- 構造体終端の不完全な配列
- 16 進数の浮動小数点定数
- 構造体と配列における指定イニシャライザ
- プリプロセッサ演算子 `_Pragma()`
- 可変数引数マクロは、`printf` スタイルの関数に相当するプリプロセッサマクロです

- VLA（可変長配列）は、コンパイラオプション --vla によって明示的に有効化する必要があります
- asm キーワードまたは __asm キーワードを使用したインラインアセンブラ（参照）

注 : たとえ C99 の機能であっても、ARM 用 IAR C/C++ コンパイラは UCN（universal character name ＝汎用文字名）をサポートしていません。

拡張の概要

コンパイラでは、C 標準の機能のほかに、組込み業界における効率的なプログラミング専用の機能から、規格上の軽微な問題の緩和にいたるまで、幅広い拡張を提供します。

以下は使用可能な拡張の概要です。

- IAR C 言語拡張

  使用可能な言語拡張については、「177 ページの *IAR C 言語拡張*」を参照してください。拡張キーワードの詳細は、「*拡張キーワード*」を参照してください。C++、言語の 2 レベルのサポート、C++ 言語拡張については、「*C++ の使用*」の章を参照してください。

- プラグマディレクティブ

  #pragma ディレクティブは、C 規格によって定義されたものであり、ベンダ固有の拡張の使用方法を規定することにより、ソースコードの移植性を維持するための仕組みです。

  コンパイラでは、コンパイラの動作（メモリの割当て方法、拡張キーワードの許可 / 禁止、ワーニングメッセージの表示 / 非表示など）の制御に使用可能な定義済プラグマディレクティブを提供します。ほとんどのプラグマディレクティブは前処理され、マクロに置換されます。プラグマディレクティブは、コンパイラでは常に有効になっています。これらのいくつかに対しては、対応する C/C++ 言語拡張も用意されています。使用可能なプラグマディレクティブについては、「*プラグマディレクティブ*」の章を参照してください。

- プリプロセッサ拡張

  コンパイラのプリプロセッサは、C 規格に準拠しています。また、コンパイラにより、いくつかのプリプロセッサ関連拡張も利用可能になります。詳細については、「*プリプロセッサ*」を参照してください。

- 組込み関数

  組込み関数は、低レベルのプロセッサ処理に直接アクセスするための関数であり、時間が重要なルーチンなどで非常に便利です。組込み関数は、単一の命令か短い命令シーケンスとして、インラインコードにコンパイルさ

れます。組込み関数の使用方法については、「151 ページの *C 言語とアセンブラの結合*」を参照してください。使用可能な関数については、「*組込み関数*」を参照してください。

- ライブラリ関数

  IAR DLIB ライブラリは、組込みシステムに利用される最も重要な C/C++ ライブラリ定義を提供します。詳細については、431 ページの *IAR DLIB ライブラリ*を参照してください。

注 : プラグマディレクティブ以外の拡張を使用する場合、アプリケーションは C 規格との整合性がなくなります。

言語拡張の有効化

プロジェクトオプションを使用して、さまざまな言語の適合レベルを選択できます。

コマンドライン	IDE*	説明
`--strict`	**厳密**	すべての IAR C 言語拡張が無効になり、C 規格外のすべてに対してエラーが発せられます。
なし	**標準**	*C 規格に対するすべての拡張*が有効ですが、組込みシステムのプログラミングの*拡張は一切有効になりません*。拡張については、177 ページの *IAR C 言語拡張*を参照してください。
`-e`	**標準（IAR 拡張あり）**	すべての *IAR C 言語拡張*が有効になります。

表 23: 言語拡張

* IDE では、**[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語] > [言語の適合]** を選択して、適切なオプションを選びます。言語拡張はデフォルトで有効になっています。

IAR C 言語拡張

コンパイラには、幅広い C 言語拡張セットが用意されています。アプリケーションで必要な拡張が簡単に見つかるように、このセクションでは拡張は以下のようにグループ化されています。

- *組込みシステムプログラミングの拡張* — 一般的にメモリの制限を満たすために、使用する特定のコアでの効率的な組込みプログラミングに特化した拡張。
- *C 規格に対する緩和* — つまり、C 規格の重要でない問題の緩和や、便利ではあるが重要性の低い構文の拡張。180 ページの *C 規格に対する緩和*を参照してください。

組込みシステムプログラミングの拡張

以下の言語拡張は、C/C++ 両方のプログラミング言語で使用可能なもので、組込みシステムのプログラミングに最適です。

- 型属性およびオブジェクト属性

  関連する概念、一般的な構文規則、リファレンス情報については、「*拡張キーワード*」を参照してください。

- 絶対アドレスへの配置、指定セクションへの配置

  `@` 演算子や `#pragma location` ディレクティブを使用して、グローバル変数や静的変数を絶対アドレスに配置することや、指定されたセクションに変数や関数を配置することができます。これらの機能の使用方法については、217 ページの*データと関数のメモリ配置制御*、369 ページの *location* を参照してください。

- アラインメントの制御

  それぞれのデータ型には独自のアラインメントがあります。詳細については、325 ページの*アラインメント*を参照してください。アラインメントを変更する場合、`__packed` データ型属性、`#pragma pack`、`#pragma data_alignment` ディレクティブが利用できます。オブジェクトのアラインメントをチェックする場合は、`__ALIGNOF__()` 演算子を使用します。

  `__ALIGNOF__` 演算子は、オブジェクトのアラインメントの取得に使用できます。以下の 2 つのフォーマットのいずれかで指定します。

  - `__ALIGNOF__(`*type*`)`
  - `__ALIGNOF__(`*expression*`)`

  2 番目のフォーマットの expression は評価されません。

- 匿名構造体と匿名共用体

  C++ には、匿名共用体という機能があります。コンパイラでは、C プログラミング言語において、構造体と共用体の両方に対する同様の機能を使用できます。詳細については、215 ページの*匿名構造体と匿名共用体*を参照してください。

- ビットフィールドと非標準型

  標準の C では、ビットフィールドの型は `int` か `unsigned int` でなければなりません。IAR C の言語拡張を使用することで、任意の整数型や列挙型を使用できます。これには、場合によって構造体のサイズが小さくなるという利点があります。詳細については、328 ページの*ビットフィールド*を参照してください。

- `static_assert()`

  構造 `static_assert(`*const-expression*`,"`*message*`");` は、C/C++ で使用できます。この構造はコンパイル時に評価され、*const-expression* が偽であれば、*message* 文字列を含むメッセージが出力されます。

- 可変数引数マクロのパラメータ

  可変数引数マクロは、printf スタイルの関数に相当するプリプロセッサマクロです。プリプロセッサは、引数なしで可変数引数マクロを受け入れます。つまり、... パラメータに一致するパラメータがない場合、", ##__VA_ARGS__" マクロ定義でコンマが削除されます。標準の C では、... パラメータは少なくとも 1 つの引数と一致する必要があります。

専用セクション演算子

コンパイラは、以下のビルトインセクション演算子を使用して、開始アドレスや終了アドレス、セクションの取得をサポートします。

__section_begin	指定のセクションまたはブロックの最初のバイトアドレスを返します。
__section_end	指定のセクションまたはブロックの*後にくる*最初のバイトアドレスを返します。
__section_size	指定のセクションまたはブロックのサイズ（バイト）を返します。

注 : エイリアス、__segment_begin/__sfb、__segment_end/__sfe、__segment_size/__sfs も使用できます。

これらの演算子は、リンカ設定ファイルで定義された指定のセクションまたは指定ブロック上で使用できます。

これらの演算子は構文的に以下のように宣言された場合と同じように動作します。

```
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t __section_size(char const * section)
```

@ 演算子または #pragma location ディレクティブを使用してデータオブジェクトや関数をユーザ定義のセクションに配置したり、指定ブロックをリンカ設定ファイルで使用する場合、セクション演算子を使用して、セクションまたはブロックが配置されたメモリ範囲の開始アドレスと終了アドレスを取得できます。

指定の*セクション*は文字列リテラルである必要があり、#pragma *section* ディレクティブで先に宣言されている必要があります。__section_begin 演算子のタイプは、void へのポインタです。この組込み演算子を使用するには、言語拡張を有効にしておく必要があります。

これらの演算子は、専用の名前を持つ*シンボル*として実装され、以下の名前でリンカマップファイルに表示されます。

演算子	シンボル
`__section_begin(sec)`	`sec$$Base`
`__section_end(sec)`	`sec$$Limit`
`__section_size(sec)`	`sec$$Length`

表 24: セクション演算子とそのシンボル

これらの演算子を使用しない場合、リンカは同じ名前を持つセクションを連続して配置するとは限らない点に注意してください。これらの演算子（または同等のシンボル）を使用すると、セクションが指定のブロック内にあるかのようにリンカが動作します。これは、演算子に意味のある値が割り当てられるように、セクションが連続して配置されるためです。このことで、リンカ設定ファイルで指定したセクションの配置と矛盾が生じる場合、リンカでエラーが出力されます。

例

以下の例では、`__section_begin` 演算子の型は `void *` です。

```
#pragma section="MYSECTION"
...
セクション _start_address = __section_begin("MYSECTION");
```

375 ページの *section*、369 ページの *location* を参照してください。

C 規格に対する緩和

このセクションでは、一部の C 規格の問題の一覧と、緩和について説明するとともに、重要度の低い構文の拡張についても解説します。

- 不完全型の配列

  配列では、不完全な `struct` 型、`union` 型、`enum` 型をエレメントの型として使用できます。型は、配列が使用される場合はその前に、使用されない場合はコンパイル単位の終了までに完全にする必要があります。

- `enum` 型の前方宣言

  拡張を使用して、`enum` の名前を先に宣言しておき、後で中括弧で囲んだリストを指定することでその名前を解決できます。

- `struct` または `union` 指定子の最後のセミコロンの欠損を容認する

  `struct` や `union` 指定子の末尾にセミコロンがないと、ワーニング（エラーの代わりとして）が出力されます。

- NULL と void

  ポインタの処理において、void へのポインタは必要に応じて別の型に暗黙的に変換されます。また、NULL ポインタ定数は、必要に応じて適切な型の NULL ポインタに暗黙的に変換されます。C 規格では、一部の演算子でこの動作が認められていますが、そうでないものもあります。

- 静的イニシャライザでのポインタから整数へのキャスト

  イニシャライザでは、ポインタ定数値を整数型にキャストできます（整数型のサイズが十分に大きい場合）。ポインタのキャストに関する詳細については、334 ページのキャストを参照してください。

- レジスタ変数のアドレスの取得

  C 規格では、レジスタ変数として指定した変数のアドレスを取得することは不正です。コンパイラではこれは可能ですが、ワーニングが出力されます。

- long float は double を意味します

  long float 型は、double 型の同義語として扱われます。

- typedef 宣言の繰返し

  同一スコープ内で typedef を繰り返し宣言することは可能ですが、ワーニングが出力されます。

- ポインタ型の混在

  交換可能だが同一ではない型へのポインタ間（unsigned char * と char * など）で代入、差分計算を行うことが可能です。これには、同一サイズの整数型へのポインタが含まれます。ワーニングが出力されます。

  文字列リテラルを任意の種類の文字へのポインタに代入することは可能であり、ワーニングは出力されません。

- トップレベルでない const

  ポインタの代入は、代入先の型に、トップレベルでない型修飾子が追加されている場合には可能です（int ** から int const ** への代入など）。また、このようなポインタの差分の比較および取得も可能です。

- -lvalue 以外の配列

  lvalue 以外の配列式は、使用時に配列の最初のエレメントへのポインタに変換されます。

- プリプロセッサディレクティブ終了後のコメント

  この拡張は、プリプロセッサディレクティブの後にテキストを配置できるようにするもので、厳密な C 規格モードを使用していない場合に有効になります。この言語拡張の目的は、レガシーコードのコンパイルをサポートすることであり、このフォーマットで新しいコードを記述することは推奨しません。

- enum リスト最後の余分なカンマ

  enum リストの最後に、余分なカンマを付けてもかまいません。厳密な C 規格モードでは、ワーニングが出力されます。

- } の前のラベル

  C 規格では、ラベルに続けて少なくとも 1 つの文を記述する必要があります。したがって、ラベルをブロックの最後に配置するのは不正になります。コンパイラはこれを許可しますが、ワーニングが出力されます。

  これは、switch 文のラベルについても同様です。

- 空白の宣言

  空白の宣言（セミコロンのみ）は可能ですが、リマークが出力されます（リマークが有効な場合）。

- 単一の値の初期化

  C 規格では、静的な配列、struct、union のイニシャライザ式は、すべて中括弧で囲む必要があります。

  単一の値のイニシャライザは、中括弧なしで記述できますが、ワーニングが出力されます。コンパイラは次の式を受け入れます。

```
struct str
{
  int a;
} x = 10;
```

- 他のスコープでの宣言

  他のスコープでの外部 / 静的宣言は可視になります。以下の例では、変数 y は if 文の本体でのみ可視になるべきですが、関数の最後で使用できます。ワーニングが出力されます。

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

- 関数をコンテキストとして持つ文字列への関数名の拡張

  シンボル __func__ または __FUNCTION__ を関数本体の中で使用すると、そのシンボルが現在の関数名を持つ文字列に拡張されます。また、シンボル __PRETTY_FUNCTION__ を使用すると、パラメータ型とリターン型も含

まれます。シンボル `__PRETTY_FUNCTION__` を使用した場合の結果は、以下の例のようになります。

```
"void func(char)"
```

これらのシンボルは、アサーションやその他のトレースユーティリティに便利です。これらは、言語拡張が有効化されていることが必要です（「263 ページの *-e*」を参照）。

- 関数およびブロックスコープ内の静的関数

  静的関数を関数およびブロックスコープ内で宣言可能。宣言はファイルスコープに移動します。

- 数値の構文に従って数値の走査が行われる

  数値は、`pp-number` 構文ではなく、数値の構文に従って走査されます。このため、`0x123e+1` は 1 つの有効なトークンではなく、3 つのトークンとして走査されます。（`--strict` オプションを使用する場合、代わりに `pp-number` 構文が使用されます）。

C++ の使用

- 概要 — EC++ および EEC++
- 概要 — 標準 C++
- C++ および派生言語のサポートを有効にする
- C++ および EC++ の機能の説明
- EEC++ の機能の説明
- EC++ および C++ の言語拡張

概要 — EC++ および EEC++

IAR システムズは C++ 言語をサポートしています。標準の C++、業界標準の Embedded C++、拡張 Embedded C++ の規格から選択できます。ここでは、*C++ 言語を使用する*際の注意事項について説明します。

Embedded C++ は、ISO/IEC C++ 規格の正当なサブセットで、組込みシステムのプログラミング用に設計されています。業界団体である Embedded C++ Technical Committee により定義されています。組込みシステム開発においてはパフォーマンスと移植性が特に重要であり、言語の定義時には、このことが考慮されています。EC++ には C++ と同じオブジェクト指向の利点がありますが、予測がつきにくいコードサイズや実行時間の増加につながる一部の機能はありません。

EMBEDDED C++

以下の C++ 機能がサポートされています。

- クラス。データ構造と動作の両方をまとめたユーザ定義の型のことです。本質的な機能である継承により、データ構造と動作を複数のクラスで共有できます
- ポリモフィズム。1 つの処理が異なるクラスで異なる動作を実現できる機能です。仮想関数によって提供されます
- 演算子と関数名のオーバロード。引数リストが明確に異なる場合に、名前が同一の演算子や関数を複数使用できます
- 型安全なメモリ管理。演算子 `new`、`delete` を使用します
- インライン関数。特にインライン展開に適しています

プログラマによる制御が不可能なオーバヘッドが、実行時間やコードサイズに生じるような C++ 機能は除外されています。他に除外されているのは、C++ 標準が定義される直前に追加された機能です。したがって、Embedded C++ は、効率性が高く、既存の開発ツールで完全にサポートされている C++ のサブセットを提供します。

Embedded C++ は、以下の C++ の機能が削除されています。

- テンプレート
- 多重継承と仮想継承
- 例外処理
- ランタイムの型情報
- 新しいキャスト構文（演算子 `dynamic_cast`、`static_cast`、`reinterpret_cast`、`const_cast`）
- 名前空間
- `mutable` 属性

これらの言語機能の除外により、ランタイムライブラリの効率性が大幅に向上しています。他にも、Embedded C++ ライブラリはフル C++ ライブラリと以下の点で異なります。

- 標準テンプレートライブラリ (STL) が除外されています
- テンプレートを使用せずにストリーム、文字列、複素数をサポートしています
- 例外処理、ランタイムの型情報（`except`、`stdexcept`、`typeinfo` の各ヘッダ）に関連するライブラリ機能が除外されています

注 : Embedded C++ は名前空間をサポートしていないため、ライブラリは `std` 名前空間中には存在しません。

拡張 EMBEDDED C++

IAR システムズの拡張 EC++ は、標準の EC++ に以下の機能を追加した C++ のサブセットです。

- フルテンプレートサポート
- 多重継承と仮想継承
- 名前空間のサポート
- `mutable` 属性
- キャスト演算子 `static_cast`、`const_cast`、`reinterpret_cast`

これらの追加機能は、C++ 規格に準拠しています。

拡張 EC++ をサポートするため、本製品には、標準テンプレートライブラリ (STL) が付属しています。STL には C++ 標準チャプタユーティリティ、コンテナ、イテレータ、アルゴリズム、数値が含まれています。この STL は、拡張 EC++ 言語の使用に合わせて変更されており、例外処理、ランタイムの型情報 (rtti) をサポートしていません。また、ライブラリは std 名前空間には存在しません。

注 : 拡張 EC++ を有効化してコンパイルされたモジュールは、拡張 EC++ を有効化せずにコンパイルされたモジュールと完全なリンク互換性を持ちます。

概要 — 標準 C++

IAR C++ 実装は、ISO/IEC 14882:2003 C++ 標準に完全に準拠しています。本ガイドでは、この規格を C++ といいます。

EC++ や EEC++ ではなく標準の C++ を使用する主な理由は、以下のいずれかの必要性がある場合です。

- 例外のサポート
- ランタイム型情報 (RTTI) のサポート
- 標準の C++ ライブラリ (EC++ ライブラリは C++ ライブラリの簡易版で、ストリームと文字列はテンプレートではありません)

コードサイズが重要で、アプリケーションにこれらの機能が必要なければ、EC++（または EEC++）の方が適しています。

例外および RTTI サポートのモード

例外とランタイム型の情報がアプリケーションにインクルードされることによって、コードサイズは増加します。サイズの増加を避けるために、以下のどちらか一方または両方を無効にした方がいい場合もあります。

- ランタイム型情報のコンストラクトのサポートは、コンパイラオプション --no_rtti を使用すれば無効にできます
- 例外のサポートは、コンパイラオプション --no_exceptions を使用して無効化できます

コンパイル中にサポートが有効な場合でも、リンカは余分なコードやテーブルの最終アプリケーションへのインクルードを避けることができます。アプリケーションで例外が発生しない場合、例外の使用をサポートするコードやテーブルは、アプリケーションイメージにインクルードされません。また、動的ランタイム型情報コンストラクト (dynamic_cast/typeid) がポリモフィズム型と併用されない場合、それらのサポートに必要なオブジェクトは、アプリケーションのコードイメージにインクルードされません。この動作を制

御するには、リンカオプション --no_exceptions、--force_exceptions、--no_dynamic_rtti_elimination を使用します。

例外サポートを無効にする

コンパイラオプション --no_exceptions を使用する場合、以下によってコンパイラエラーが出力されます。

- throw 式
- try-catch 文
- 関数定義上の例外仕様

さらに、例外が関数を介して伝播されるときにオブジェクトの破棄を処理するのに必要な、自動記憶寿命を持つ追加のコードやテーブルは、コンパイラオプション --no_exceptions を使用したときに生成されません。

例外に直接関係のないシステムヘッダのすべての機能は、コンパイラオプション --no_exceptions の使用時にサポートされています。

例外サポートを持たずにコンパイルされたモジュールと例外サポートありでコンパイルされた C++ モジュールをリンクしようとすると、リンカでエラーが出力されます。

詳細については、274 ページの *--no_exceptions* を参照してください。

RTTI サポートを無効にする

コンパイラオプション --no_rtti を使用する場合、以下によってコンパイラエラーが出力されます。

- typeid 演算子
- dynamic_cast 演算子

注 : --no_rtti を使用して、例外サポートが有効になっている場合、ほとんどの RTTI サポートは例外が機能する上で必要なため、コンパイラの出力オブジェクトファイルにインクルードされます。

詳細については、276 ページの *--no_rtti* を参照してください。

例外処理

例外処理は以下の 3 つの部分に分けることができます。

- 例外の発生メカニズム — C++ では throw 式および rethrow 式です。
- 例外のキャッチメカニズム — C++ では try-catch 文や関数の例外仕様、main から例外がリークするのを防ぐための暗黙的キャッチです。

- 現在アクティブな関数についての情報 — try-catch 文と自動オブジェクトのセットがあって、例外が関数を介して伝播されるときにそれらのデストラクタを実行する必要がある場合。

例外が引き起こされると、関数のコールスタックが関数およびブロックごとに巻き戻されます。それぞれの関数やブロックについて、破棄が必要な自動オブジェクトのデストラクタが実行され、例外のキャッチハンドラがあるかどうかチェックが行われます。ある場合は、そのキャッチハンドラから実行が続けられます。

C++ コードをアセンブラおよび C コードと併用するアプリケーション、およびアセンブラルーチンと C 関数を介して、ある C++ 関数から別の関数へ例外をスローするアプリケーションは、リンカオプション --exception_tables と引数 unwind を併用する必要があります。

例外の実装

例外はテーブル方式を使用して実装されます。それぞれの関数について、テーブルで以下を記述します。

- 関数の巻き戻し方法。つまり、スタック上で呼出し元を探して復元が必要なレジスタを復元する方法です
- 関数にどのキャッチハンドラがあるのか
- 関数に例外仕様があるかどうか、どの例外の伝播が許可されているか
- デストラクタを実行する必要がある自動オブジェクトのセット

例外が引き起こされるとき、ランタイムは 2 つのフェーズで進行します。最初のフェーズは例外テーブルを使用して、スタックの巻き戻しをその時点で停止させるキャッチハンドラまたは例外仕様を含む関数呼出しのスタックを検索します。このポイントが見つかれば、第 2 のフェーズに入り、実際の巻き戻しとそれが必要な自動オブジェクトのデストラクタの実行が行われます。

テーブル方式は、例外が実際にスローされない場合、実質的に実行時間や RAM 使用量でのオーバヘッドがありません。テーブルおよび追加コードに対して、リードオンリーメモリに非常に大きな影響が出るだけでなく、例外のスローやキャッチが比較的コストのかかる処理になります。

例外の結果によるスタックの巻き戻し中の自動オブジェクトの破棄は、通常の関数の処理を扱うコードとは別にコードに実装されます。このコードはキャッチハンドラのコードとともに、通常のコード（本来は .text に配置）とは別のセクション (.exc.text) に配置されます。場合によっては、たとえば高速と低速の ROM メモリがある場合、リンカ設定ファイルにセクションを配置する際に、この違いに基づいて選択すると有益なことがあります。

C++ および派生言語のサポートを有効にする

コンパイラでは、デフォルトの言語は C です。

標準の C++ で記述されたファイルをコンパイルするには、`--c++` コンパイラオプションを使用します。257 ページの *--c++* を参照してください。

Embedded C++ で記述されたファイルをコンパイルするには、`--ec++` コンパイラオプションを使用します。263 ページの *--ec++* を参照してください。

拡張 Embedded C++ の機能をソースコードで利用するには、`--eec++` コンパイラオプションを使用します。264 ページの *--eec++* を参照してください。

IDE で EC++、EEC++、C++ を有効にするには、**［プロジェクト］> ［オプション］> ［C/C++ コンパイラ］> ［言語］**を選び、適切な標準を選択します。

C++ および EC++ の機能の説明

ARM 用 IAR C/C++ コンパイラ用の C++ ソースコードを記述する際、C++ の機能（クラス、クラスメンバなど）と IAR 言語拡張（IAR 固有の属性など）を組み合わせる場合の利点や、特異な動作の可能性について認識しておく必要があります。

IAR 属性とクラスを使用する

C++ クラスの静的データメンバは、グローバル変数と同じように処理され、適切なすべての IAR 型とオブジェクト属性を持つことができます。

原則的にメンバ関数は解放された関数と同じように扱われ、適切なすべての IAR 型とオブジェクト属性を持つことができます。仮想メンバ関数はデフォルトの関数ポンタと互換性のある属性しか持つことができません。コンストラクタとデストラクタはこうした属性を持つことはできません。

場所演算子 `@` と `#pragma location` ディレクティブは、静的データメンバ上ですべてのメンバ関数とともに使用することができます。

属性とクラスの使用例

```
class MyClass
{
public:
  // 静的変数の場所を __memattr メモリ内のアドレス 60 に指定します
  static __no_init int mI @ 60;

  // 静的 Thumb 関数
  static __thumb void F();

  // A Thumb 関数
  __thumb void G();

  // インターワーク
  virtual __arm void ArmH();

  // 仮想関数の場所を SPECIAL に指定します
  virtual void M() const volatile @ "SPECIAL";
};
```

関数型

extern "C" リンケージを持つ関数型は、C++ リンケージを持つ関数と互換性があります。

例

```
extern "C"
{
  typedef void (*FpC)(void);     // C 関数 typedef
}

typedef void (*FpCpp)(void);     // C++ 関数 typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
  MyF(F1);                       // 常に機能する
  MyF(F2);                       // fpCpp は fpC と互換
}
```

割込みで静的クラスオブジェクトを使用する

割込み関数が、（コンストラクタを使用して）生成または（デストラクタを使用して）破棄しなければならない静的クラスオブジェクトを使用する場合、オブジクトの生成前、または破棄の後や途中に割込みが発生すると、アプリケーションが正しく機能しなくなります。

これを回避するために、静的オブジェクトが構築されるまで、これらの割込みが有効になっておらず、`main` から戻ったり、`exit` を呼出すときに無効になっていることを確認します。システムの起動について詳しくは、125 ページの *システムの起動と終了*を参照してください。

関数ローカルの静的クラスオブジェクトは、実行が最初に宣言を通過すると生成され、`main` から戻ったり、`exit` を呼出す際に破棄されます。

新しいハンドラを使用する

メモリの消耗に対応するには、`set_new_handler` 関数を使用します。

C++ 規格の新しいハンドラ

`set_new_handler` を呼出さないか、NULL 新規ハンドラを使用して呼出す場合、例外が有効になっていて `operator new` が十分なメモリの割当てに失敗すると、`operator new` が `std::bad_alloc` をスローします。例外が有効でない場合、`operator new` は代わりに `abort` を呼出します。

NULL 以外の新規ハンドラを用いて `set_new_handler` を呼出す場合、`operator new` が十分なメモリの割当てに失敗すると、`operator new` によって提供された新規ハンドラが呼出されます。新規ハンドラはより多くのメモリを使用できるようにして、何らかの形で実行を返すか、中止する必要があります。例外が有効な場合、新規ハンドラは `std::bad_alloc` 例外をスローすることもできます。`operator new` の派生型である `nothrow` は、例外が有効で新規ハンドラが `std::bad_alloc` をスローする場合に、新規ハンドラがある状態でのみ NULL を返します。

Embedded C++ の新しいハンドラ

`set_new_handler` を呼出さないか、NULL 新規ハンドラを使用して呼出す場合、`operator new` が十分なメモリの割当てに失敗すると、`abort` が呼出されます。new operator の派生型である `nothrow` は、代わりに NULL を返します。

NULL 以外の新規ハンドラを用いて `set_new_handler` を呼出す場合、`operator new` がメモリの割当てに失敗すると、`operator new` によって提供された新規ハンドラが呼出されます。新規ハンドラはより多くのメモリを使用できるようにして、何らかの形で実行を返すか、中止する必要があります。`operator new` の派生型 `nothrow` は、新規ハンドラがある状態で NULL を返すことはありません。

テンプレート

C++ および拡張 EC++ は、C++ 標準に基づいてテンプレートをサポートしますが、export キーワードはサポートしません。実装では、2 段階のルックアップを使用します。すなわち、必要なときには常に typename キーワードを挿入する必要があります。さらに、テンプレートを使用するたびに、使用可能なすべてのテンプレート定義が可視になる必要があります。すなわち、すべてのテンプレートの定義がインクルードファイルまたは実際のソースファイルに存在する必要があります。

C-SPY でのデバッグサポート

C-SPY® には、STL コンテナ用に組込みの表示サポートがあります。コンテナの論理構造は、わかりやすく追跡しやすい方法で包括的に［ウォッチ］ビューに表示されます。

C++ を使用すると、throw 文の位置や、引き起こされた例外に対応する catch 文がない場合に、C-SPY を停止させることができます。

これらの詳細は、*『ARM 用 C-SPY® デバッグガイド』* を参照してください。

EEC++ の機能の説明

以下のページでは、拡張 EC++ と EC++ で大きく異なる機能について説明します。

テンプレート

コンパイラは、標準 C++ で定義された構文および動作を持つテンプレートをサポートしています。ただし、製品に付属の STL（標準テンプレートライブラリ）は、拡張 EC++ 用に調整されています（186 ページの *拡張 Embedded C++* を参照）。

キャスト演算子の派生形

拡張 EC++ では、以下の C++ キャスト演算子の派生形を使用できます。

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

MUTABLE

拡張 EC++ では mutable 属性がサポートされています。クラスオブジェクト全体が const である場合でも、mutable シンボルを変更できます。

名前空間

名前空間機能は、*拡張* EC++ でのみサポートされています。すなわち、名前空間を使用してコードを分割できます。ただし、ライブラリそのものは std 名前空間には配置されません。

STD 名前空間

std 名前空間は、標準 EC++ と拡張 EC++ のいずれでも使用されません。std 名前空間内のシンボルを参照するコードがある場合は、以下の例のように std を無定義とします。

```
#define std
```

使用するアプリケーション内の識別子が、ランタイムライブラリの識別子を妨害しないように注意する必要があります。

EC++ および C++ の言語拡張

コンパイラをいずれかの C++ モードで使用し、IAR 言語拡張を有効にする場合、以下の C++ 言語拡張がコンパイラで使用できます。

- クラスの friend 宣言において、class キーワードを省略できます。以下に例を示します。

```
class B;
class A
{
  friend B;        // IAR 言語拡張を用いると
                   // 可能
  friend class B; // 標準規格に従った書き方
};
```

- スカラ型の定数は、クラス内で定義できます。以下に例を示します。

```
class A
{
  const int mSize = 10; // IAR 言語拡張を用いると
                        // 可能
  int mArr[mSize];
};
```

規格では、初期化した静的データメンバを代わりに使用することになっています。

- クラスメンバの宣言において、修飾名を使用できます。以下に例を示します。

```
struct A
{
  int A::F(); // IAR 言語拡張を用いると可能
  int G();    // 標準規格に従った書き方
};
```

- C リンケージ (extern "C") を持つ関数のポインタと、C++ リンケージ (extern "C++") を持つ関数のポインタとの間の暗黙的な型変換の使用が許可されています。以下に例を示します。

```
extern "C" void F(); // C リンケージを持つ関数
void (*PF)()         // pf は C++ リンケージを持つ関数を指す
           = &F;  // ポインタの暗黙の変換
```

  規格では、ポインタは明示的に変換する必要があります。

- ? 演算子を含む構造体の 2 番目または 3 番目のオペランドが文字列リテラルまたはワイド文字列リテラル（C++ の場合の定数）の場合、オペランドを暗黙的に char * または wchar_t * に変換できます。以下に例を示します。

```
bool X;

char *P1 = X ? "abc" : "def";          // IAR 言語拡張を用いると
                                       // 可能
char const *P2 = X ? "abc" : "def";    // 標準規格に従った書き方
```

- 関数パラメータに対するデフォルトの引数は、規格に従ったトップレベルの関数宣言の中ではなく、typedef 宣言の中、関数へのポインタの関数宣言の中、メンバへのポインタの関数宣言の中でも指定できます。
- 非静的ローカル変数を含む関数、評価されない式（sizeof 式など）を含むクラスにおいては、式から非静的ローカル変数を参照できます。ただし、ワーニングが出力されます。
- typedef 名によって、含有クラスに匿名共用体を導入できます。最初に共用体を宣言する必要はありません。次に例を示します。

```
typedef union
{
  int i,j;
} U;  // U は再利用可能な匿名共用体を識別

class A
{
public:
  U;  // OK -- A::i および A::j への参照が許可されています。
};
```

  また、この拡張は *anonymous classes* と *anonymous structs* も許可します。ただし、C++ の機能がなく（たとえば、静的データメンバやメンバ関数を持

たず、パブリックでないメンバがないなど)、他の匿名クラスや構造体、共用体以外のネスト型を持たないことが条件です。次に例を示します。

```
struct A
{
  struct
  {
    int i,j;
  }; // OK -- A::i および A::j への参照が許可されています。
};
```

- friend class の構文では、nonclass 型のほか、精密型名なしに typedef によって表現されたクラス型も使用可能です。次に例を示します。

```
typedef struct S ST;

class C
{
public:
  friend S;  // OK（S がスコープ内にある必要あり）
  friend ST; // OK（"friend S;" と同じ）
  // friend S const; // エラー、cv-qualifiers は直接
                     // 現れることはできません
};
```

注：最初に言語拡張を有効にせずに、これらの構造体のいずれかを使用すると、エラーが出力されます。

アプリケーションに関する考慮事項

- 出力形式に関する注意事項
- スタックについて
- ヒープについて
- ツールとアプリケーション間の相互処理
- チェックサムの計算
- リンカの最適化
- AEABI への準拠
- CMSIS の統合

出力形式に関する注意事項

リンカは、ELF/DWARF オブジェクトファイル形式で絶対実行可能イメージを生成します。

絶対 ELF イメージは、IAR ELF Tool (`ielftool`) を使用して、メモリへの直接ロードや PROM またはフラッシュメモリなどへの書込みに適したフォーマットに変換できます。

`ielftool` では、以下の出力形式を生成できます。

- バイナリ
- Motorola S-records
- Intel hex

注 : `ielftool` は、絶対イメージ内のチェックサムの埋め込みや計算など、別のタイプの変換にも使用できます。

`ielftool` のソースコードは、`arm¥src` ディレクトリにあります。`ielftool` の詳細は、483 ページの *IAR ELF ツール — ielftool* を参照してください。

スタックについて

お使いのアプリケーションでスタックメモリを効果的に使用するには、考慮することがいくつかあります。

スタックサイズについて

必要なスタックサイズは applications の動作によって大きく異なります。スタックサイズが大きすぎる場合は、RAM が無駄に消費されます。スタックサイズが小さすぎる場合には、2 つうちのどちらかが発生します。これは、スタックのメモリ上の位置により異なり、

- 変数記憶領域が上書きされ、未定義の動作を引き起こします
- スタックの位置がメモリエリアを超えアプリケーションが異常終了します

いずれの場合もアプリケーション障害が発生します。後者は発見が容易なため、メモリの最後に向かって大きくなるようにスタックを配置するのが得策です。

スタックサイズの詳細については、100 ページの*スタックメモリの設定*、228 ページの*スタックエリアと RAM メモリの節約*を参照してください。

スタックのアラインメント

デフォルトの `cstartup` コードは、すべてのスタックを自動的に 8 バイトにアラインメントされたアドレスに初期化します。

スタックのアラインメントの詳細については、164 ページの*呼出し規約*、166 ページの*専用レジスタ*、168 ページの*スタックパラメータとレイアウト*を参照してください。

例外スタック

Cortex-M には、個別の例外スタックがありません。デフォルトでは、すべての例外スタックは、`CSTACK` セクションに配置されています。

ARM7/9/11、Cortex-A、および Cortex-R デバイスでは、異なる例外が発生したときに入力される 5 つの例外モードをサポートしています。各例外モードには、システム / ユーザモードスタックの破損を回避するための独自のスタックがあります。

以下の表に、さまざまな例外スタックの推奨スタック名を示します。ただし、スタックには任意の名前を付けることができます。

プロセッサのモード	推奨スタックセクション名	説明
スーパバイザ	SVC_STACK	オペレーティングシステムスタック。
割込み	IRQ_STACK	汎用 (IRQ) 割込みハンドラのスタック。
高速割込み	FIQ_STACK	高速 (FIQ) 割込みハンドラのスタック。
未定義	UND_STACK	未定義命令割込みのスタック。ハードウェアコプロセッサおよび命令セット拡張のソフトウェアエミュレーションをサポートします。
アボート	ABT_STACK	命令フェッチおよびデータアクセスメモリ中断割込みハンドラのスタック。

表 25: ARM7/9/11、Cortex-A、および Cortex-R の例外スタック

スタックが必要な各プロセッサモードでは、個別のスタックポインタを起動コードで初期化し、セクション配置をリンカ設定ファイルで行う必要があります。IRQ および FIQ スタックは、提供されている `cstartup.s` および `lnkarm.icf` ファイルで事前に定義されている唯一の例外スタックです。ただし、他の例外スタックを簡単に追加することができます。

これらのスタックを IDE で使用できる［スタック］ウィンドウで表示するには、ユーザ定義セクション名ではなく、これらの事前定義セクション名を使用する必要があります。

ヒープについて

ヒープには、C 関数 `malloc`（あるいは関連関数）か C++ の演算子 `new` を使用して割り当てられた動的データが格納されます。

アプリケーションで動的メモリ割当てを使用する場合には、以下の内容に精通しておく必要があります。

- ヒープに使用されるリンカセクション
- ヒープサイズの割当て。詳細については、100 ページの *ヒープメモリの設定*を参照してください。

アドバンストヒープとベーシックヒープ

システムライブラリには 2 つの異なるヒープメモリハンドラ、*ベーシックヒープハンドラ*と *アドバンスト*ヒープハンドラが含まれます。アプリケーションでヒープの割当てを明示的に使用するようであれば、リンカは自動的

にアドバンストヒープを選択します。それ以外の場合はベーシックヒープが使用されます。

注 : 製品にコードサイズの制限がある場合、ベーシックヒープが自動的に選択されます。

リンカオプションを使用すると、使用するハンドラを明示的に指定できます。

- ベーシックヒープ (`--basic_heap`) はオーバヘッドが少なく、ヒープメモリのみを割り当てるアプリケーションの例に適していますが、`free` は呼出されません。ヒープの他の使用については、一般的にアドバンストヒープの方がより効率的です。297 ページの *--basic_heap* を参照してください。
- アドバンストヒープ (`--advanced_heap`) は、ヒープインタフェースを重点的に使用したり、メモリを繰り返し解放するアプリケーションの効率的なメモリ管理を可能にします。297 ページの *--advanced_heap* を参照してください。この定義については、436 ページの *iar_dlmalloc.h* を参照してください。

ヒープサイズと標準 I/O

通常の設定のように `FILE` 記述子を DLIB ランタイムライブラリから除外すると、I/O バッファが無効になります。詳細設定のように、それ以外の場合は、`stdio` ライブラリのヘッダファイルで I/O バッファが 512 バイトに設定されます。ヒープが小さすぎる場合は、I/O がバッファされず、I/O がバッファされた場合よりも大幅に低速になります。IAR C-SPY® デバッガのシミュレータドライバを使用してアプリケーションを実行する場合には、速度低下が現れない可能性がありますが、アプリケーションを ARM コアで実行すると、速度低下を明確に認識できます。標準 I/O ライブラリを使用する場合は、ヒープサイズを標準 I/O バッファの必要に応じたサイズに設定してください。

ツールとアプリケーション間の相互処理

リンクプロセスとアプリケーションでシンボルを相互処理する方法は以下の 4 種類があります。

- リンカコマンドラインオプション `--define_symbol` を使用してシンボルを作成する。リンカは、アプリケーションがラベル、サイズ、デバッガのセットアップなどとして使用できるパブリック絶対定数シンボルを作成します。
- コマンドラインオプション `--config_def` または設定ディレクティブ `define symbol` を使用し、`export symbol` ディレクティブを使用しシンボルをエクスポートして、エクスポート済み設定シンボルを作成する。LINK は、アプリケーションがラベル、サイズ、デバッガのセットアップなどとして使用できるパブリック絶対定数シンボルを作成します。

このシンボル定義の利点の 1 つは、このシンボルを設定ファイルで式として使用できる点です。たとえば、メモリ範囲へのセクションの配置を制御するときなどに使用します。

- コンパイラ演算子 __section_begin、__section_end または __section_size、またはアセンブラ演算子 SFB、SFE、または指定の SIZEOF セクションまたは block を使用します。これらの演算子は、開始アドレス、終了アドレス、セクションの連続シーケンスに、同じ名前、またはリンカ設定ファイルで指定されたリンカブロックのシーケンスを提供します。
- コマンドラインオプション --entry は、アプリケーションの開始ラベルをリンカに通知します。これは、リンカより、実行の開始位置をデバッガに通知するときのルートシンボルとして使用されます。

次のラインは -D を使用してシンボルを作成する方法を示します。この方法を使用する必要がある場合は、これらのオプションをこのようなコマンドラインに追加します:

```
--define_symbol NrOfElements=10
--config_def HEAP_SIZE=1024
```

リンカ設定ファイルでは、次のように定義されています。

```
define memory Mem with size = 4G;
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* シンボルをエクスポートする */
export symbol MY_HEAP_SIZE;

/* ILINK のオプションで定義された大きさのヒープエリアを設定 */
define block MyHEAP with size = MY_HEAP_SIZE, alignment = 8 {};

place in RAM { block MyHEAP };
```

以下の行をアプリケーションソースコードに追加します。

```
#include <stdlib.h>

/* ILINK オプションで定義したシンボルを使い、指定したサイズのエレメント配列
を動的に割り当てます。値はラベルの形式をとります。
 */
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
  return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* ILINK オプションで定義したシンボルを使う。
 * リンカ設定ファイル内のシンボルはアプリケーションで使用できるようになってい
る。
 */
extern char MY_HEAP_SIZE;

/* ヒープを含むセクションを宣言 */
#pragma section = "MYHEAP"

char *MyHeap()
{
  /* 最初に、静的に配置されたセクションの最初アドレスを得る */
  char *p = __section_begin("MYHEAP");

  /* インポートしたヒープサイズを使用し、0 で初期化する */
  for (int i = 0; i < (int) &MY_HEAP_SIZE; ++i)
  {
    p[i] = 0;
  }
  return p;
}
```

チェックサムの計算

IAR ELF Tool (`ielftool`) は、特定の範囲のメモリをパターンで埋め、これらの範囲のチェックサムを計算します。計算されるチェックサムは、入力 ELF イメージの既存シンボルの値を置換します。アプリケーションは、これらの範囲が変更されていないか検証できます。

チェックサムを使用してアプリケーションの整合性を検証する場合、以下のことを行う必要があります。

- ielftool により計算されるチェックサムについて、配置を関連する名前とサイズとともに予約する
- チェックサムアルゴリズムを選択し、その ielftool を設定して、アルゴリズムのソースコードをアプリケーションに含める
- アプリケーションソースコードでielftoolとそのソースコードの両方を検証および設定するメモリ範囲を決定する

IDE に ielftool を設定するには、**[プロジェクト] > [オプション] > [リンカ] > [チェックサム]** を選択します。

チェックサムの計算

この例では、0x8002 ～ 0x8FFF にある ROM メモリのチェックサムが計算されます。また、計算された 2 バイトチェックサムが 0x8000 に配置されます。

計算されるチェックサムの配置の作成

計算されるチェックサムの配置は、2 とおりの方法で作成できます。特定のセクション（この例では .checksum）に常駐する正しいサイズのグローバル C/C++ またはアセンブラ定数シンボルを作成する方法と、リンカオプション --place_holder を使用する方法です。

たとえば、シンボル __checksum の 2 バイトスペースをセクション .checksum にアラインメント 4 で作成するには、以下のようにします。

```
--place_holder __checksum,2,.checksum,4
```

注： .checksum セクションは、必要と思われる場合に、アプリケーションにのみインクルードされます。アプリケーション自体でチェックサムが必要でない場合、リンカオプション --keep=__checksum か、リンカディレクティブ keep を使用して、セクションを強制的にインクルードすることができます。

.checksum セクションを配置するには、リンカ設定ファイルを修正する必要があります。例えば、これを次に示します（ブロック CHECKSUM の扱いに注意してください）。

```
define memory Mem with size = 4G;

define region ROM_region = Mem:[from 0x8000 to 0x80000000 - 1];
define region RAM_region = Mem:[from 0x80000000 to 0x100000000 -2
];

initialize by copy { rw };
do not initialize { section .noinit };

define block HEAP      with alignment = 8, size = 16M {};
define block CSTACK    with alignment = 8, size = 16K {};
define block IRQ_STACK with alignment = 8, size = 16K {};
define block FIQ_STACK with alignment = 8, size = 16K {};

define block CHECKSUM    { ro section .checksum };
place at address Mem:0x0 { ro section .intvec};
place in ROM_region  { ro, first block CHECKSUM };
place in RAM_region { rw, block HEAP, block CSTACK, block
                      IRQ_STACK, block FIQ_STACK };
```

ielftool の実行

チェックサムを計算するには、ielftool を実行します。

```
ielftool --fill=0x00;0x8000-0x8FFF
--checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out
destinationFile.out
```

チェックサムを計算するには、フィル操作を定義する必要があります。この例では、フィルパターン 0x0 が使用されます。使用されるチェックサムアルゴリズムは crc16 です。

ielftool では、リンカオプションを使用していない ELF イメージが必要です。--strip リンカオプションを使用する場合、これを削除し、代わりに --strip ielftool オプションを使用します。

チェックサム関数をソースコードに追加する

ielftool により生成されるチェックサムの値をチェックするには、アプリケーションにより計算されたチェックサムと比較する必要があります。つまり、チェックサム計算用の関数（ielftool と同じアルゴリズムを使用）をアプリケーションソースコードに追加する必要があります。アプリケーションには、この関数へのコールを含める必要があります。

チェックサム計算用の関数

以下の関数（計算時間は遅いがメモリ使用量は少ない版）では、crc16 アルゴリズムが使用されます。

```
unsigned short SlowCrc16(unsigned short sum,
                         unsigned char *p,
                         unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

    for (i = 0; i < 8; ++i)
    {
      unsigned long oSum = sum;
      sum <<= 1;
      if (byte & 0x80)
        sum |= 1;
      if (oSum & 0x8000)
        sum ^= 0x1021;
      byte <<= 1;
    }
  }
  return sum;
}
```

チェックサムアルゴリズムのソースコードは、製品インストールの arm¥src¥linker ディレクトリにあります。

チェックサムの計算の例

以下のコードは、チェックサムがどのように計算されるかを示した例です。

```
/* The checksum calculated
 *（これはアドレス 0x8000 上にあります）
 */
extern unsigned short const __checksum;


void TestChecksum()
{
  unsigned short calc = 0;
  unsigned char zeros[2] = {0, 0};

  /* チェックサム計算の実行 */
  calc = SlowCrc16(0,
                   (unsigned char *) checksumStart,
                   (checksumEnd - checksumStart+1));

  /* 結果を格納 */
  calc = SlowCrc16(calc, zeros, 2);

  /* チェックサムのテスト */
  if (calc != __checksum)
  {
    abort();   /* 失敗 */
  }
}
```

注意事項

チェックサムを計算する場合、以下のことに注意してください。

- チェックサムは、すべてのメモリ範囲で最下位アドレスから最上位アドレスに計算する必要があります
- 各メモリ範囲は定義されている順序で検証する必要があります（ABC は ACB とは異なります）
- 1 つのチェックサムに対して複数の範囲が存在することは問題ありません
- 複数のチェックサムが使用される場合、セクションごとに一意のシンボル名を使用する必要があります
- 低速の関数バリアントが使用される場合、チェックサム計算の最後の呼出しは、チェックサム内のバイト数と同じバイト数（値 0x00）で行う必要があります
- チェックサムが含まれている場所のチェックサムは計算しません

詳細については、483 ページの *IAR ELF ツール— ielftool* を参照してください。

C-SPY に関する注意事項

デフォルトでは、リンカオプション `--place_holder` を使用してメモリ内に割り当てたシンボルは、C-SPY によって `int` 型であると見なされます。チェックサムのサイズが `int` のサイズとは異なる場合、そのサイズに合わせてチェックサムシンボルの表示フォーマットを変更できます。

［C-SPY ウォッチ］ウィンドウでシンボルを選択し、コンテキストメニューから［**表示フォーマット**］を選択します。チェックサムシンボルのサイズに合った表示フォーマットを選択します。

リンカの最適化

仮想関数の除去

仮想関数除去 (VFE) は、不要な仮想関数と動的ランタイム型情報を除去するリンカの最適化です。

仮想関数除去が機能するためには、仮想関数テーブルについての情報や、どの仮想関数が呼出されるか、どのクラスの動的ランタイム型情報が必要かをすべての適切なモジュールで指定する必要があります。1 つまたは複数のモジュールでこの情報が得られない場合、リンカでワーニングが生成され、仮想関数除去は実行されません。

このような情報を持たないモジュールが仮想関数の呼出しを実行せず、仮想関数テーブルを定義しないことが分かっている場合、`--vfe=forced` リンカオプションを使用して、仮想関数除去を有効にできます。

現在は IAR システムズおよび RealView のツールが、リンカで使用可能な形での仮想関数除去に必要な情報を提供しています。

仮想関数除去は、`--no_vfe` リンカオプションを使用して完全に無効にすることができます。この場合、VFE 情報を持たないモジュールについてワーニングは出力されません。

詳細については、322 ページの *--vfe*、316 ページの *--no_vfe* を参照してください。

AEABI への準拠

ARM 用 IAR ビルドツールは、ARM Limited が提唱する ARM Embedded Application Binary Interface (AEABI) をサポートしています。このインタフェースは、Intel IA64 ABI インタフェースに基づいています。AEABI に準拠する

と、他のベンダにより提供されるツールで生成されていても、モジュールを他の任意の AEABI 準拠モジュールとリンクできるというメリットがあります。

ARM 用 IAR ビルドツールは、AEABI の以下のパートをサポートしています。

AAPCS	ARM アーキテクチャのプロシージャ呼出し標準
CPPABI	ARM アーキテクチャの C++ ABI（EC++ パーツのみ）
AAELF	ARM アーキテクチャの ELF
AADWARF	ARM アーキテクチャの DWARF
RTABI	ARM アーキテクチャのランタイム ABI
CLIBABI	ARM アーキテクチャの C ライブラリ ABI

IAR ビルドツールは、明示的なオペレーティングシステムがない ROM ベースシステムのみをサポートします。

注 :

- AEABI は C89 専用です
- IAR ビルドツールは、デフォルトおよび C ロケールの使用のみをサポートします
- AEABI は、C++ ライブラリとの互換性を指定しません
- `enum` および `wchar_t` のいずれのサイズも、AEABI では一定ではありません

AEABI 準拠が有効な場合、システムヘッダファイルで実行されるほとんどすべての最適化が無効になり、特定のプロセッサ定数が、実定数の変数になります。

IAR ILINK リンカを使用して AEABI 準拠モジュールをリンクする

IAR ILINK リンカを使用してアプリケーションを構築する場合、以下のタイプのモジュールを組み合わせることができます。

- IAR ビルドツールを使用して生成されたモジュール（AEABI 準拠モジュールと AEABI 準拠でないモジュールの両方）
- 別のベンダのビルドツールを使用して生成された AEABI 準拠モジュール

注 : 別のベンダのコンパイラで生成されたモジュールをリンクするには、そのベンダの追加サポートライブラリが必要になることがあります。

IAR ILINK リンカでは、オブジェクトファイルに含まれる属性に基づき、使用する適切な標準 C/C++ ライブラリが自動的に選択されます。インポートされるオブジェクトファイルには、これらのすべての属性が含まれないことが

あります。そのため、このような場合、以下の 1 つ以上の項目を検証して、ILINK による標準ライブラリの選択をサポートする必要があります。

- 使用する CPU（--cpu リンカオプションを指定）
- フル I/O が必要な場合、フルライブラリ設定の標準ライブラリとリンクする必要がある
- --no_library_search リンカオプションとの組み合わせが可能な、ランタイムライブラリファイルを明示的に指定する

リンクする際は、仮想関数除去についても考慮してください（207 ページの *仮想関数の除去*を参照）。

サードパーティ製リンカを使用して AEABI 準拠のモジュールをリンクする

IAR C/C++ コンパイラを使用してモジュールを生成し、このモジュールを別のベンダのリンカを使用してリンクする場合、このモジュールは AEABI 準拠モジュールでなければなりません（209 ページの *AEABI 準拠をコンパイラで有効にする*を参照）。

また、このモジュールが任意の IAR 固有コンパイラ拡張を使用する場合、これらの機能が他のベンダのツールによりサポートされている必要があります。特に以下のことに注意してください。

- 以下の拡張のサポートを検証する必要があります。#pragma pack、__no_init、__root、__ramfunc
- 以下の拡張は使用しても問題ありません。#pragma location/@、__arm, __thumb、__swi、__irq、__fiq、and __nested

AEABI 準拠をコンパイラで有効にする

AEABI 準拠をコンパイラで有効にするには、--aeabi オプションを設定します。この場合、--guard_calls オプションも使用する必要があります。

IDE で、**[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション]** ページを使用して、--aeabi オプションおよび --guard_calls オプションを指定します。

コマンドラインで、オプション --aeabi と --guard_calls を使用して、コンパイラでの AEABI サポートを有効にします。

また、特定のシステムヘッダファイルで AEABI のサポートを有効にするには、システムヘッダを含める前にプロセッサシンボル

_AEABI_PORTABILITY_LEVEL を非ゼロに定義し、シンボル AEABI_PORTABLE がヘッダファイルの追加後に非ゼロに設定されるようにする必要があります。

```
#define _AEABI_PORTABILITY_LEVEL 1
#undef _AEABI_PORTABLE
#include <header.h>
#ifndef _AEABI_PORTABLE
 #error "header.h not AEABI compatible"
#endif
```

CMSIS の統合

arm¥CMSIS サブディレクトリには、CMSIS (Cortex Microcontroller Software Interface Standard) および CMSIS DSP ヘッダとライブラリファイル、ドキュメントが含まれます。CMSIS について詳しくは、http://www.arm.com/cmsis をご覧ください。

特殊なヘッダファイル inc¥c¥cmsis_iar.h が、現行バージョンの IAR C/C++ コンパイラの CMSIS 版として用意されています。

CMSIS DSP ライブラリ

IAR Embedded Workbench には、arm¥CMSIS¥Lib¥IAR ディレクトリにビルド済の CMSIS DSP ライブラリが用意されています。ライブラリファイルの名前は次のように作成されます :

```
iar_cortexM<0|3|4><l|b>[f]_math.a
```

<0|3|4> は Cortex–M 派生品、<l|b> はバイトオーダをそれぞれ選択し、[f] はライブラリが FPU（Cortex–M4 のみ）用にビルドされていることを示します。

Cortex-M4 のライブラリは Cortex-M7 にも適用可能です。

CMSIS DSP ライブラリのカスタマイズ

CMSIS DSP ライブラリのソースコードが、arm¥CMSIS¥DSP_Lib¥Source ディレクトリに用意されています。カスタマイズされた DSP ライブラリをビルドするために用意された IAR Embedded Workbench プロジェクトは、arm¥CMSIS¥DSP_Lib¥Source¥IAR ディレクトリにあります。

コマンドラインでの CMSIS を使用したビルド

ここでは、CMSIS 互換のアプリケーションをコマンドライン上でビルドする方法の例を示します。

CMSIS のみ（DSP ライブラリなし）

```
iccarm -I $EW_DIR$¥arm¥CMSIS¥Include
```

DSP ライブラリあり、Cortex-M4、リトルエンディアン、FPU あり

```
iccarm --endian=little --cpu=Cortex-M4 --fpu=VFPv4_sp -I
$EW_DIR$¥arm¥CMSIS¥Include -D ARM_MATH_CM4

ilinkarm $EW_DIR$¥arm¥CMSIS¥Lib¥IAR¥iar_cortexM3l_math.a
```

IDE での CMSIS を使用したビルド

［プロジェクト］>［オプション］>［一般オプション］>［ライブラリ構成］を選択して、CMSIS のサポートを有効にします。

有効にすると、CMSIS のインクルードパスと DSP ライブラリが自動的に使用されます。詳細については、*ARM 用 IDE プロジェクト管理およびビルドガイド*を参照してください。

組込みアプリケーション用の効率的なコーディング

- データ型の選択
- データと関数のメモリ配置制御
- コンパイラ最適化の設定
- 円滑なコードの生成

データ型の選択

データを効率的に処理するため、使用するデータ型や最も効率的な変数配置を検討する必要があります。

効率的なデータ型の使用

使用するデータ型は、コードのサイズ / 速度に大きく影響することがあるため、慎重に検討する必要があります。

- 可能な限り `char` や `short` の代わりに `int` または `long` を使用して、符号拡張やゼロ拡張を回避します。特に、ループインデックスは、コード生成を最小に抑えるため、必ず `int` または `long` にしてください。また、Thumb モードでは、スタックポインタ (`SP`) を介したアクセスは、32 ビットデータ型に制限されます。これにより、このようなデータ型を使用するメリットを利用できます。
- アプリケーションで符号付き値が必要でない限り、符号なしデータ型を使用してください。
- 64 ビットデータ型（`double`、`long long` など）を使用する場合は、短所を理解しておいてください。
- ビットフィールドやパック構造体を用いると、大きくて低速のコードを生成します。
- 数値演算コプロセッサのないマイクロプロセッサで浮動小数点数型を使用すると、コードサイズと実行速度の両面で非常に効率が低下します。
- `const` 型データへのポインタを宣言すると、参照先のデータが変化しないことが呼出し元関数に通知され、最適化の向上につながります。

サポートされているデータ型、ポインタ、構造体の表現の詳細は、「*データ表現*」を参照してください。

浮動小数点数型

数値演算コプロセッサのないマイクロプロセッサで浮動小数点数型を使用するのは、コードサイズと実行速度の両面で非常に非効率です。そのため、浮動小数点数演算を使用するコードを、整数演算を使用するコードに置き換えることを検討してください。これにより効率が向上します。

コンパイラは、2 種類（32 ビットと 64 ビット）の浮動小数点数フォーマットをサポートしています。32 ビット浮動小数点数型の `float` の方は、コードサイズと実行速度の両面において効率が優れます。一方、64 ビットフォーマットの `double` は、より高い精度とより大きな数値に対応します。

このコンパイラでは、`float` 浮動小数点数型は常に 32 ビットフォーマットを使用し、`double` 型は常に 64 ビットフォーマットを使用します。

64 ビット浮動小数点数で得られる超高精度がアプリケーションで必要な場合を除き、32 ビット浮動小数点数の使用をお勧めします。

デフォルトでは、ソースコード内の*浮動小数点定数*は、`double` 型として扱われます。このため、何でもないような式が倍精度で評価される可能性があります。下の例 `a` では `float` から `double` に変換しています。`double` 定数 `1.0` が追加され、その結果は `float:` に戻して変換されます。

```
double Test(float a)
{
   return a + 1.0;
}
```

浮動小数点定数を `double` ではなく `float` として扱うには、以下の例のように `f` を追加します。

```
double Test(float a)
{
   return a + 1.0f;
}
```

不動小数点型の詳細については、332 ページの*基本データ型浮動小数点数型*を参照してください。

構造体エレメントのアラインメント

ARM コアではメモリ内のデータにアクセスするときに、そのデータがアラインメントされている必要があります。構造体の各エレメントは、指定した型の要件に応じてアラインメントされている必要があります。つまり、コンパイ

ラは、正しいアラインメントを保守するため、*パッドバイト*を挿入しなければならないことがあります。

これが問題になりうる状況があります。

- 外部要件。たとえば、ネットワーク通信プロトコルは通常、間にパディングのないデータ型に関して指定されます。
- データメモリを節約する必要がある場合。

アラインメントの要件については、325 ページの *アラインメント*を参照してください。

構造体のレイアウトをより密にするために `#pragma pack` ディレクティブまたは `__packed` データ型属性を使用します。欠点は構造体のアラインメントされていないエレメントにアクセスするたびにコードが使用されることです。

または構造体の*パック / アンパック*用のユーザカスタム関数を記述する。こちらの方が移植性が高く、ユーザカスタム関数以外の追加コードは生成されません。欠点として、構造体のデータをパックとアンパックの 2 つの状態で確認する必要があります。

`#pragma pack` ディレクティブの詳細は、372 ページの *pack* を参照してください。

匿名構造体と匿名共用体

構造体や共用体を名前なしで宣言すると、それらは匿名になります。その結果、それらのメンバは前後のスコープでのみ認識されます。

匿名構造体は C++ 言語の機能の一部ですが、C にはこの機能はありません。ARM 用 IAR C/C++ コンパイラでは、言語拡張が有効になっていれば、これらを C で使用できます。

IDE では、デフォルトで言語拡張が有効になっています。

言語拡張を有効にするには、`-e` コンパイラオプションを使用します。詳細については、263 ページの *-e* を参照してください。

例

以下の例では、匿名の union のメンバに、union 名を明示的に指定せずに、関数 F でアクセスできます。

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;

void F(void)
{
  St.mL = 5;
}
```

メンバ名は、その前後のスコープ内で固有なものであることが必要です。匿名の struct/union を、ファイルスコープレベルで、グローバル変数、外部変数、静的変数のいずれかとして使用することもできます。これは、以下の例のように、I/O レジスタの宣言などに使用できます。

```
__no_init volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char way: 1;
    unsigned char out: 1;
  };
} @ 0x1000;



/* ここで変数を使用 */
void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

この例は、I/O レジスタバイト IOPORT をアドレス 0x1000 で宣言します。この I/O レジスタでは、Way と Out の 2 ビットが宣言されます。内部の構造体と外部の共用体のいずれも匿名になっています。

匿名の構造体や共用体はオブジェクトとして実装されます。このオブジェクトの名前は、最初のフィールドにプレフィックス _A_ を付けた名前となり、名前空間の実装部で配置されます。この例では、匿名共用体は _A_IOPORT というオブジェクトを使用して実装されます。

データと関数のメモリ配置制御

コンパイラでは、関数とデータオブジェクトのメモリへの配置を制御するためのさまざまな仕組みを提供しています。メモリを効率的に使用するためには、これらの仕組みに精通し、さまざまな状況に応じて最適な方法を判別できることが必要です。これらを以下に示します。

- @ 演算子および #pragma location ディレクティブによる絶対配置

  @ 演算子または #pragma location ディレクティブを使用して、個々のグローバル変数および静的変数を絶対アドレスに配置できます。ただし、この表記を個々の関数の絶対配置に使用することはできません。詳細については、218 ページの*絶対アドレスへのデータ配置*を参照してください。

- @ 演算子および #pragma location ディレクティブによるセクションの配置

  @ 演算子または #pragma location ディレクティブを使用して、セクションの名前で個々の関数、変数、および定数を配置できます。これらセクションの配置はリンカディレクティブによって制御されます。詳細については、219 ページの*データと関数のセクションへの配置*を参照してください。

- @ 演算子および #pragma location ディレクティブによるレジスタの配置

  @ 演算子または #pragma location ディレクティブを使用して、個々のグローバル変数および静的変数をレジスタに配置できます。変数は __no_init として宣言する必要があります。これは、特定のレジスタに配置しなければならない個々のデータオブジェクトに役立ちます。

- --section オプションを使用して、特定のモードの関数、変数、および定数にデフォルトのセグメントを設定できます。詳細については、286 ページの *--section* を参照してください。

絶対アドレスへのデータ配置

@ 演算子か #pragma location ディレクティブを使用して、グローバル変数や静的変数を絶対アドレスに配置できます。

変数を絶対アドレスに配置するには、@ 演算子や #pragma location ディレクティブの引数に、実際のアドレスを示す定数を指定します。配置する変数のアラインメント条件を満たしている絶対アドレスを指定する必要があります。

注 : 絶対アドレスに配置される __no_init 変数のすべての宣言は、*仮定義*です。仮定義の変数は、コンパイルするモジュールで必要な場合に、コンパイラからの出力にのみ保持されます。こうした変数は、使用されるすべてのモジュールで定義され、同じ方法で定義されている限りは機能します。こうした宣言は、変数を使用する全モジュールにインクルードされるすべてのヘッダファイルに配置することをお勧めします。

絶対アドレスに配置される他の変数は、通常の宣言と定義の区別を使用します。こうした変数については、1 つのモジュール（通常はイニシャライザ）でのみ定義を提供する必要があります。他のモジュールは、明示的アドレスの有無に関わらず、extern 宣言を使用すれば変数を参照できます。

例

この例では、__no_init で宣言した変数が絶対アドレスに配置されます。これは、複数のプロセス、アプリケーションなどの間でインタフェースする場合に便利です。

```
__no_init volatile char alpha @ 0xFF2000;/* OK */
```

次の例では、2 つの const 宣言オブジェクトが含まれます。1 つは初期化されず、もう 1 つは特定の値に初期化されます。両方のオブジェクトは ROM に配置されます。これは、外部インタフェースからアクセス可能な構成パラメータに便利です。2 番目においては、値が既知であるため、必ずコンパイラが変数から実際に読み出すとは限りません。

```
#pragma location=0xFF2002
__no_init const int beta;                /* OK */

const int gamma @ 0xFF2004 = 3;          /* OK */
```

1 番目においては、値はコンパイラで初期化されません。別の方法で値を設定する必要があります。代表的な用途は、値が別々に ROM にロードされる構成や、リードオンリーの特殊機能レジスタです。

```
__no_init int epsilon @ 0xFF2007;  /* Error, misaligned. */
```

C++ についての注意

C++ では、モジュールスコープの const 変数は静的（モジュールローカル）ですが、C ではこれらはグローバルです。つまり、特定の const 変数を宣言する各モジュールには、この名前で別の変数が含まれるということです。このようなモジュールの複数とアプリケーションをリンクする場合において、これらのモジュールがすべて、たとえば以下の宣言を（ヘッダファイル経由で）含む場合、

```
volatile const __no_init int x @ 0x100;          /* C++ では無効 */
```

リンカは、複数の変数がアドレス 0x100 に配置されていることを報告します。

この問題を回避し、プロセスを C と C++ で同じにするには、以下の例のように、これらの変数を extern として宣言します。

```
/* extern キーワードによって x がパブリックに */
extern volatile const __no_init int x @ 0x100;
```

注 : C++ の静的メンバ変数は、他の静的変数と同様に、絶対アドレスに配置できます。

データと関数のセクションへの配置

データまたは関数をデフォルト以外の指定セクションに配置する場合、以下の方法を使用できます。

- @ 演算子または #pragma location ディレクティブを使用して、個々の変数や関数を指定のセクションに配置できます。指定のセクション は定義済セクションまたはユーザ定義セクションです。
- --section オプションは、コンパイルユニット全体の一部である変数および関数を指定セクションに配置するときに使用できます。

C++ の静的メンバ変数は、他の静的変数と同様に、指定セクションに配置できます。

独自のセクションを使用する場合、定義済セクションに加えて、セクションをリンカ設定ファイルで定義する必要があります。

注 : デフォルトで使用している以外の定義済セクションの変数や関数を、明示的に配置する場合に注意してください。状況によっては有益なオプションですが、配置を間違えると、コンパイル時やリンク時のエラーメッセージからアプリケーションの誤動作までを発生することがあります。状況を慎重に考慮し、宣言および関数や変数の使用に関する要件に、厳密に従ってください。

セクションの位置は、リンカ設定ファイルから制御できます。

セクションの詳細は、「*セクションリファレンス*」を参照してください。

指定セクションへの変数の配置例

以下の例では、データオブジェクトがユーザ定義セクションに配置されます。リンクの際には、セクションが適切なメモリエリアに配置されていることを必ず確認してください。

```
__no_init int alpha @ "MY_NOINIT"; /* OK */

#pragma location="MY_CONSTANTS"
const int beta = 42;               /* OK */

const int gamma @ "MY_CONSTANTS" = 17;/* OK */
int theta @ "MY_ZEROS";            /* OK */
int phi @ "MY_INITED" = 4711;      /* OK */
```

ゼロまたは初期化されたケースのリンカは、変数について正しいタイプの初期化を行います。__no_init 変数をユーザ定義セクションに配置する際、そのセクションに一致するパターンをリンカ設定ファイルの do not initialize ディレクティブに追加する必要があります。初期化された変数の場合は、initialize manually ディレクティブを使用すれば自動初期化を無効にできます。

指定セクションへの関数の配置例

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_FUNCTIONS"
void h(void);
```

レジスタへのデータ配置

@ 演算子か #pragma location ディレクティブを使用して、グローバル変数や静的変数をレジスタに配置できます。

変数をレジスタに配置するには、@ 演算子および #pragma location ディレクティブの引数が、R4-R11 範囲の ARM コアレジスタに一致する識別子である必要があります（R9 は --rwpi コマンドラインオプションと組み合わせて指定することはできません）。

変数をレジスタに配置できるのは、変数が __no_init として宣言され、ファイルスコープがあって、サイズが 4 バイトの場合のみです。レジスタに配置される変数は、メモリアドレスを持たないため。アドレス演算子 & は使用できません。

変数がレジスタに配置されているモジュール内では、指定されたレジスタはその変数へのアクセスのみに使用されます。変数の値は他のモジュールへの関数呼出しで保持されます。その理由は、レジスタ R14-R11 が呼出し先で保存され、実行が返されるときに復元されるためです。ただし、レジスタに配置される変数の値は、常に予想通り保持されるとは限りません。

- 例外ハンドラやライブラリコールバックルーチン（qsort に引き渡されたコンパレータ関数など）では、値が保持されないことがあります。コマンドラインオプション --lock_regs が、ライブラリモジュールも含むアプリケーションの全モジュール内のレジスタのロックに使用される場合、値は保持されます。
- 高速割込みハンドラでは、R8-R11 の変数の値は、ハンドラ外部からは保持されません。これらのレジスタがバンクされているためです。
- longjmp 関数および C++ の例外によって、保持されない他の静的記憶寿命変数とは異なり、レジスタに配置された変数は古い値に復元されることがあります。

リンカは、モジュールが同じレジスタに異なる変数を配置するのを防止することはありません。異なるモジュールにある変数は同じレジスタ内に配置でき、別のモジュールはそのレジスタを他の目的で使用できます。

注： レジスタに配置された変数は、インクルードファイルで定義され、その変数を使用するすべてのモジュールにインクルードされる必要があります。モジュール内の未使用の定義によって、そのモジュールでレジスタが使用されなくなります。

コンパイラ最適化の設定

コンパイラは、可能な限り最良のコードを生成するために、アプリケーションで多くの変換を行います。変換の例としては、値をメモリではなくレジスタに格納する、余剰なコードを削除する、計算の順序をより効率的に変更する、数値演算をより安価な処理に置換するなどが挙げられます。

リンカによって実行される最適化もあるため、リンカもコンパイルシステムの構成要素の一部と考える必要があります。たとえば、すべての未使用の関数、変数は削除され、最終的な出力には含まれません。

最適化実行のスコープ

実行される最適化の対象を、アプリケーション全体とするか個々のファイルとするか指定できます。デフォルトでは、プロジェクト全体で同一の最適化タイプが使用されますが、個々のファイルに対して異なる最適化設定を使用することを検討してください。たとえば、非常に高速に実行する必要があるコードは別ファイルに記述して、実行時間が最小になるようにコンパイルし、

残りのコードはコードサイズを最小にするようにします。これにより、プログラムが小さくなり、重要部分での十分な高速性も実現できます。

また、個々の関数を最適化の実行から除外することも可能です。`#pragma optimize` ディレクティブでは、最適化レベルを下げることや、別のタイプの最適化の実行を指定できます。プラグマディレクティブについては、371 ページの *optimize* を参照してください。

複数ファイルのコンパイルユニット

さまざまな最適化を異なるソースファイルや関数に適用するだけでなく、コンパイルユニットに含まれるソースコードのファイル数（なし、または複数など）を指定することもできます。

デフォルトでは、コンパイルユニットは 1 つのソースファイルから構成されますが、複数ファイルのコンパイルを使用して、いくつかのソースファイルを 1 つのコンパイルユニットに作成することも可能です。この利点は、インライン化やクロスジャンプなど、プロシージャ間の最適化で対象のソースコードが多くなることです。アプリケーション全体を 1 つのコンパイルユニットとしてコンパイルするのが理想的です。ただし、大きなアプリケーションの場合はホストコンピュータにリソースの制限があるため、この方法は実用的ではありません。詳細については、272 ページの *--mfc* を参照してください。

注 : オブジェクトファイルが 1 つだけ生成されるため、すべてのシンボルはこのオブジェクトファイルの一部となります。

アプリケーション全体を 1 つのコンパイルユニットとしてコンパイルする場合、プロシージャ間の最適化を実行する前に、コンパイラで未使用のパブリック関数と変数を破棄するのも非常に役に立ちます。こうすることで、最適化のスコープが実際に使用される関数と変数に限定されます。詳細については、261 ページの *--discard_unused_publics* を参照してください。

最適化レベル

コンパイラでは、さまざまな最適化のレベルをサポートしています。以下の表は、各レベルで一般的に実行される最適化の一覧です。

最適化レベル	説明
なし（デバッグサポートに最適）	変数は、そのスコープ全体を通して有効です
低	上記と同じですが、変数は必要時のみ有効となり、スコープ全体を通して有効とは限りません
中	上記のほかに以下があります 生死解析と最適化 不要なコードの除去 冗長なラベルの除去 冗長な分岐の除去 コードホイスト ピープホール最適化 一部のレジスタ内容の解析と最適化 共通部分式除去 コード移動 静的クラスタ
高（バランス）	上記のほかに以下があります 命令スケジューリング クロスジャンプ 詳細なレジスタ内容の解析と最適化 ループ展開 関数インライン化 型ベースエイリアス解析

表 26: コンパイラ最適化レベル

注 : 一部の最適化は、個別に有効化 / 無効化が可能です。これらの詳細については、224 ページの *変換の微調整* を参照してください。

最適化レベルを高くするとコンパイル時間が長くなることがあり、また、生成されたコードとソースコードの関係がわかりにくくなるため、デバッグも困難になります。たとえば、最適化レベルの低、中、高の場合、変数がスコープ全体を通して有効とは限らないため、変数の格納に使用されたプロセッサレジスタは、最後に使用された状態のままで再使用される可能性があります。このため、C-SPY の［ウォッチ］ウィンドウには、スコープ全体を通した変数の値が表示できないことがあります。コードのデバッグが困難な場合は、最適化レベルを下げてください。

速度とサイズ

最適化レベルが高の場合、コンパイラはサイズの最適化と速度の最適化の間のバランスを取ります。ただし、サイズまたは速度に対して明示的に最適化を微調整することも可能です。それらは、使用するしきい値のみの違いです。速度の場合は、サイズを犠牲にして速度を上げ、サイズの場合は、速度を犠牲にしてサイズを小さくします。ある最適化により別の最適化が実行可能になり、サイズよりも速度を優先して最適化した場合でも、アプリケーションのサイズが小さくなることがあります。

最適化レベル高（速度）を使用する場合、`--no_size_constraints` コンパイラオプションは、コードサイズ拡張に対する通常の制限を緩和して、より積極的な最適化を可能にします。

変換の微調整

最適化レベルごとに、一部の変換を個別に無効にできます。変換を無効にするには、適当なオプション（コマンドラインオプション `--no_inline`、IDE での同等オプションの［**関数インライン化**］など）か、`#pragma optimize` ディレクティブを使用します。以下の変換は、個別に無効化することができます。

- 共通部分式除去
- ループ展開
- 関数インライン化
- コード移動
- 型ベースエイリアス解析
- 静的クラスタ

共通部分式除去

デフォルトでは［中］、［高］の最適化レベルにおいて、冗長な共通部分式が除去されます。この最適化により、コードサイズと実行時間の両方が削減されます。ただし、生成されるコードのデバッグが困難になる場合があります。

注： このオプションは、最適化レベルが［なし］、［低］の場合には動作しません。

コマンドラインオプションの詳細については、273 ページの *--no_cse* を参照してください。

ループ展開

ループ展開とは、コンパイル時に繰り返し回数を決定できるループのコード本体の複製を意味します。ループ展開によって、複数の繰り返しに分割することにより、ループのオーバヘッドが少なくなります。

この最適化は、ループのオーバヘッドがループ本体合計の大部分を占めるような、小さいループの場合に最も効率的です。

ループ展開は、最適化レベルが［高］の場合に実行可能で、通常は実行時間が短縮されますが、コードサイズは増加します。また、生成されるコードのデバッグも困難になる場合があります。

コンパイラは、ヒューリスティックにより、展開するループを決定します。ループのオーバヘッド減少が明確な、比較的小さいループのみが展開されます。実行する最適化の内容（速度、サイズ、速度とサイズのバランス）に応じて、異なるテクニックが使用されます。

注 : このオプションは、最適化レベルが［なし］、［低］、［中］の場合には動作しません。

ループの展開を無効にするには、コマンドラインオプション `--no_unroll` を使用します（280 ページの *--no_unroll* を参照）。

関数インライン化

関数インライン化とは、定義がコンパイル時に判明している関数を、その呼出し元関数の本体に統合し、呼出しによるオーバヘッドを解消することです。通常この最適化では実行時間は短縮されますが、コードサイズが大きくなることがあります。

詳細については、74 ページの*インライン関数*を参照してください。

コード移動

ループ不変式や共通部分式の評価式を移動し、冗長な再評価を回避します。この最適化は、最適化レベルが［中］またはそれ以上の場合に実行可能で、通常はコードサイズと実行時間が短縮されます。ただし、生成されるコードのデバッグは困難になる場合があります。

注 : このオプションは、最適化レベルが中以上の場合にのみ有効です。

コマンドラインオプションの詳細については、273 ページの *--no_code_motion* を参照してください。

型ベースエイリアス解析

複数のポインタが同一メモリ位置を参照する場合、これらのポインタをそれぞれの*エイリアス*といいます。エイリアスが存在すると、特定の値が変更されるかどうかがコンパイル時にわからない場合があるため、最適化が困難になります。

型ベースエイリアス解析による最適化では、同一オブジェクトへのすべてのアクセスは、そのオブジェクトの宣言型または `char` 型の使用を前提としてい

ます。これにより、コンパイラはポインタが同一のメモリ位置を参照しているかどうかを検出することができます。

型ベースエイリアス解析は、最適化レベルが［高］の場合のみ実行されます。標準の C/C++ アプリケーションコードに準拠するアプリケーションコードの場合、この最適化によってコードサイズが減少して実行時間が短縮されることがあります。ただし、非標準の C/C++ コードの場合は、予期せぬ動作の原因となるコードをコンパイラが生成することがあります。そのため、この最適化を無効にできるようになっています。

注 : このオプションは、最適化レベルが［なし］、［低］、［中］の場合には動作しません。

コマンドラインオプションの詳細については、279 ページの *--no_tbaa* を参照してください。

例

```
short F(short *p1, long *p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

型ベースエイリアス解析では、`p1` にポイントされる `short` へのライトアクセスは、`p2` がポイントする `long` の値に影響しないと見なされます。そのため、この関数が 0 を返すことをコンパイル時に判定できます。しかし、規格に準拠していない C/C++ コードでは、これらのポインタが同一の共用体に含まれ、相互に重複することがあります。明示的なキャストを使用する場合、異なるポインタ型のポインタが同一メモリ位置を参照するように強制することもできます。

静的クラスタ

静的クラスタが有効にされている場合、同じモジュール内で定義される静的およびグローバル変数は、同じ関数でアクセスされる変数がそれぞれ近くに格納されるように配置されます。これにより、コンパイラは、いくつかのアクセスに対して同じベースポインタを使用できるようになります。

注 : このオプションは、最適化レベルが［なし］、［低］の場合には動作しません。

コマンドラインオプションの詳細については、272 ページの *--no_clustering* を参照してください。

命令スケジューリング

コンパイラは、生成されるコードのパフォーマンスを改善する命令スケジューラとして機能します。スケジューラは、その目的を達成するため、命令を再配置して、マイクロプロセッサ内のリソース競合から広がるパイプラインストールの数を最小に抑えます。

コマンドラインオプションの詳細については、277 ページの *--no_scheduling* を参照してください。

円滑なコードの生成

ここでは、コンパイラで良いコードを生成するためのヒントについて説明します。たとえば、次のようなものがあります。

- 効率的なアドレッシングモードの使用
- コンパイラの最適化の促進
- より有意義なエラーメッセージの生成

最適化を容易にするソースコードの記述

以下に、コンパイラでのアプリケーション最適化を改善できるプログラミングテクニックを示します。

- 静的 / グローバル変数よりもローカル変数（自動変数とパラメータ）を使用することをお勧めします。これは、呼出し先関数がローカル以外の変数を変更する可能性などをオプティマイザが想定する必要があるためです。ローカル変数の使用期間が終了すると、占有されていたメモリを再利用できます。グローバルに宣言した変数は、プログラムの実行中はデータメモリを占有します。
- & 演算子を使用してローカル変数のアドレスを取ることは避けてください。これは、主に 2 つの理由で非効率です。まず、変数はメモリに配置する必要があり、プロセッサのレジスタに配置できません。そのため、コードのサイズが大きくなり、速度が低下します。次に、ローカル変数が関数呼出しの影響を受けないとオプティマイザが想定できなくなります。
- グローバル変数（非静的）よりも、モジュール内でローカルな変数（static として宣言された変数）を使用することをお勧めします。また、頻繁にアクセスされる静的変数のアドレスを取ることは避けてください。
- コンパイラによる関数のインライン化が可能です（225 ページの *関数インライン化* を参照）。インライン化の効果を最大限にするには、複数のモジュールから呼出される小さい関数の定義を、実装ファイルではなくヘッダファイルに配置することをお薦めします。または、複数ファイルコンパイルを使用します。詳細については、222 ページの *複数ファイルのコンパ*

*イルユニット*を参照してください。

- オペランドや上書きされるリソースを持たないインラインアセンブラは使用しないようにしてください。代わりに、可能であれば SFR や組込み関数を使用します。そうでなければ、オペランドや上書きされるリソースの*あ*るインラインアセンブラを使用するか、アセンブラ言語で別のモジュールを記述してください。詳細については、151 ページの *C 言語とアセンブラの結合*を参照してください。

スタックエリアと RAM メモリの節約

以下に、メモリやスタックエリアを節約できるプログラミングテクニックを示します。

- スタックエリアが少ない場合は、長い呼出しチェーンや再帰関数の使用は避けてください。
- 大きなサイズの非スカラ型（構造体など）をパラメータやリターン型として使用することは避けてください。スタックエリアを節約するため、代わりにポインタか、C++ の場合は参照として引き渡してください。

関数プロトタイプ

2 つのスタイルのいずれかを使用して、関数の宣言と定義を行うことができます。

- プロトタイプ
- カーニハン & リッチー C (K&R C)

どちらのスタイルも有効な C ですが、できる限りプロトタイプスタイルを使用して、関数を定義するコンパイルユニットおよびそれを使用するすべてのコンパイルユニットの両方に含まれるヘッダの public 関数ごとに、プロトタイプ宣言を指定するようお勧めします。

コンパイラは、K&R スタイルを使用して宣言された関数に引き渡されるパラメータに対してはチェックを実行しません。プロトタイプ宣言を使用すると、コードがより効率的になることがあります。これは、これらの関数に型変換が必要ないためです。

すべての関数定義が適切なプロトタイプスタイルを使用し、すべての public 関数が定義される前に宣言されていることをコンパイラで義務づけるには、**［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［プロトタイプの強制］**コンパイラオプション (`--require_prototypes`) を使用します。

プロトタイプスタイル

プロトタイプ関数の宣言では、各パラメータの型を指定する必要があります。

```
int Test(char, int); /* 宣言 */

int Test(char ch, int i) /* 定義 */
{
  return i + ch;
}
```

カーニハン & リッチースタイル

カーニハン & リッチースタイル（標準 C 以前）では、プロトタイプ化された関数を宣言することはできません。その代わりに、空のパラメータリストを関数宣言で使用します。また、定義の記述が異なります。

次に例を示します。

```
int Test();      /* 宣言 */

int Test(ch, i) /* 定義 */
char ch;
int i;
{
  return i + ch;
}
```

整数型とビット否定

場合によっては、整数型とその変換の規則が、混乱を招く動作の原因となることがあります。異なるサイズの型や論理演算（特にビット否定）が関係する代入文や条件文（評価式）に注意する必要があります。この場合、*types* 型には定数型も含まれます。

ワーニング（定数の条件文や無意味な比較など）が発生する場合と、予期した結果と異なるだけの場合があります。コンパイラが定数の条件文のインスタンスを特定するために最適化を使用する場合などは、より高水準の最適化でのみ、コンパイラがワーニングを生成することがあります。以下の例では、8 ビット文字、32 ビット整数、2 の補数を想定しています。

```
void F1(unsigned char c1)
{
  if (c1 == ~0x80)
    ;
}
```

この場合、評価結果は常に false になります。右辺の `0x80` は `0x00000080` であり、`~0x00000080` は `0xFFFFFF7F` になります。左辺の `c1` は 8 ビットの符号

なし文字であるため、255 を超えることはありません。また、負の値にもならないため、汎整数拡張された値の上位 24 ビットが設定されることもありません。

同時にアクセスされる変数の保護

非同期でアクセスされる変数（割込みルーチンからアクセスされる変数、独立したスレッドで実行しているコードからアクセスされる変数など）は、適切にマークして保護する必要があります。例外は、常に*リードオンリー*の変数のみです。

変数を適切にマークするには、`volatile` キーワードを使用します。このキーワードは、変数が他のスレッドから変更される可能性があることをコンパイラに示します。すると、コンパイラでは、変数の最適化を回避（レジスタ内の変数を追跡するなど）し、変数へのライトを遅延させず、ソースコードで指定された回数だけ変数にアクセスするように注意します。

`volatile` 型修飾子および volatile オプションのアクセス規則について詳しくは、337 ページの*オブジェクトの volatile 宣言*を参照してください。

特殊機能レジスタへのアクセス

IAR 製品のインストール内容には、ARM デバイス用の専用ヘッダファイルがいくつか付属しています。ヘッダファイルは、`io`*`device`*`.h` という形式で命名され、プロセッサ固有の特殊機能レジスタ (SFR) を定義します。

注 : 各ヘッダファイルには、コンパイラが使用するセクションが 1 つ、アセンブラが使用するセクションが 1 つ含まれています。

ビットフィールド付きの SFR が、ヘッダファイルで定義されています。以下に ioks32c5000a.h の例を示します。

```
__no_init volatile union
{
  unsigned short mwctl2;
  struct
  {
    unsigned short edr: 1;
    unsigned short edw: 1;
    unsigned short lee: 2;
    unsigned short lemd: 2;
    unsigned short lepl: 2;
  } mwctl2bit;
} @ 0x1000;


/* コードに適切なインクルードファイルを含めることによって、
 * 以下のようにレジスタ全体または個々のビット
 *（あるいはビットフィールド）に C コードからアクセスできます。
 */

void Test()
{
  /* レジスタ全体へのアクセス */
  mwctl2 = 0x1234;

  /* ビットフィールドアクセス */
  mwctl2bit.edw  = 1;
  mwctl2bit.lepl = 3;
}
```

他の ARM デバイス用に新しいヘッダファイルを作成する場合には、ヘッダファイルをテンプレートとして使用することもできます。

C およびアセンブラオブジェクト間での値の受渡し

以下の例は、C ソースコードでアセンブラをインライン化して、特殊な目的のレジスタから値を設定および取得する方法を示しています。

```
static unsigned long get_APSR( void )
{
  unsigned long value;
  asm volatile( "MRS %0, APSR" : "=r"(value) );
  return value;
}

static void set_APSR( unsigned long value)
{
  asm volatile( "MSR APSR, %0" :: "r"(value) );
}
```

汎用レジスタは、特殊な目的のレジスタ APSR の値の取得および設定に使用されます。他の特殊な目的のレジスタおよび特殊な命令へのアクセスにも同じ方法を使用できます。

インラインアセンブラの詳細については、152 ページの*インラインアセンブラ*を参照してください。

非初期化変数

通常は、アプリケーション起動時に、ランタイム環境がすべてのグローバル変数と静的変数を初期化します。

コンパイラは、__no_init 型修飾子により、初期化されない変数の宣言をサポートしています。このような変数は、キーワードとして指定することや、#pragma object_attribute ディレクティブを使用して指定することができます。コンパイラは、このような変数を個別のセグメントに配置します。

__no_init を使用した場合、const キーワードは、リードオンリーメモリにオブジェクトが格納されるのではなく、オブジェクトがリードオンリーであることを意味します。__no_init オブジェクトに初期値を指定することはできません。

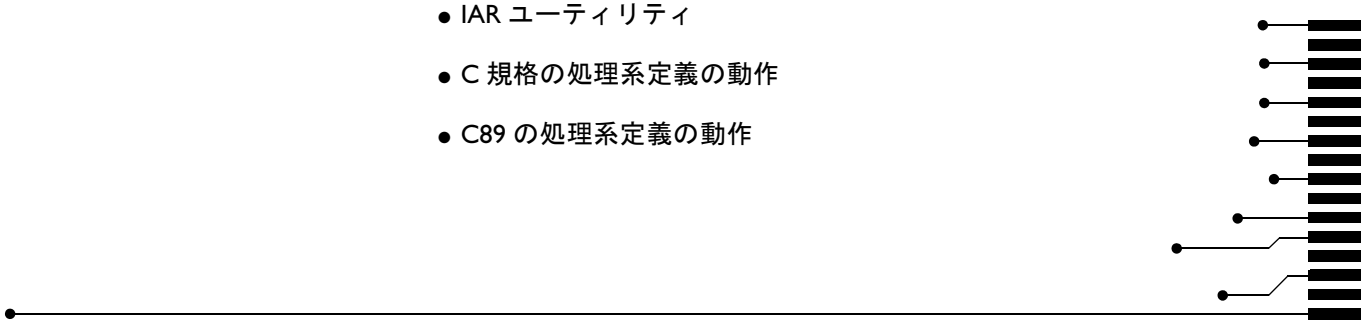
__no_init キーワードを使用して宣言した変数は、大きな入力バッファとして使用することや、アプリケーション終了後も内容を保持する特殊な RAM にマッピングすることができます。

詳細については、349 ページの *__no_init* を参照してください。このキーワードを使用するには、言語拡張を有効にする必要があります（263 ページの *-e* を参照）。詳細については、370 ページの *object_attribute* を参照してください。

パート 2. リファレンス情報

『ARM 用 IAR C/C++ 開発ガイド』 のこのパートは、以下の章で構成されています。

- 外部インタフェースの詳細
- コンパイラオプション
- リンカオプション
- データ表現
- 拡張キーワード
- プラグマディレクティブ
- 組込み関数
- プリプロセッサ
- ライブラリ関数
- リンカ設定ファイル
- セクションリファレンス
- スタック使用解析制御ファイル
- IAR ユーティリティ
- C 規格の処理系定義の動作
- C89 の処理系定義の動作



外部インタフェースの詳細

- 呼出し構文
- インクルードファイル検索手順
- コンパイラ出力
- ILINK 出力
- 診断

呼出し構文

コンパイラとリンカは、IDE またはコマンドラインインタフェースから使用できます。IDE からのビルドツールの使用については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。

コンパイラ呼出し構文

コンパイラの呼出し構文は次のとおりです。

```
iccarm [options] [sourcefile] [options]
```

たとえば、`prog.c` というソースファイルをコンパイルする場合は、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。

```
iccarm prog.c --debug
```

ソースファイルには、C/C++ ファイルを使用でき、それぞれ `c` または `cpp` のファイル名拡張子を指定します。ファイル名拡張子を指定しない場合、コンパイルするファイルの拡張子は `c` でなければなりません。

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コンパイラを実行する場合、コンパイ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。

ILINK 呼出し構文

ILINK の呼出し構文は次のとおりです。

```
ilinkarm［引数］
```

各引数は、コマンドラインオプション、オブジェクトファイル、ライブラリのいずれかです。

たとえば、オブジェクトファイル prog.o をリンクする場合、以下のコマンドを使用します。

```
ilinkarm prog.o --config configfile
```

リンカ設定ファイルの拡張子を指定しない場合、設定ファイルの拡張子は icf でなければなりません。

通常、コマンドラインの引数の順序は重要ではありません。ただし、例外が 1 つあります。複数のライブラリを適用する場合、ライブラ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。デフォルトライブラリは常に最後に検索されます。

出力実行可能イメージは、-o オプションが使用されない限り、a.out という名前のファイルに置かれます。

コマンドラインから引数なしで ILINK を実行する場合、ILINK 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

オプションをコンパイラおよび ILINK に渡す方法は 3 とおりあります。

- コマンドラインから直接渡す方法

  コマンドラインで、iccarm または ilinkarm コマンドの後にオプションを指定します（235 ページの *呼出し構文*を参照）。
- 環境変数経由で渡す方法

  コンパイラおよびリンカは、自動的に環境変数の値を各コマンドラインの後に付加します（237 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（266 ページの *-f* を参照）。

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

環境変数

以下の環境変数をコンパイラで使用できます。

環境変数	説明
C_INCLUDE	インフルードファイルを検索するディレクトリを指定します。例： C_INCLUDE=c:¥program files¥iar systems¥embedded workbench 7.*n*¥arm¥inc;c:¥headers
QCCARM	コマンドラインのオプションを指定します。QCCARM=-lA asm.lst

表 27: コンパイラの環境変数

以下の環境変数が ILINK で使用できます。

環境変数	説明
ILINKARM_CMD_LINE	コマンドラインのオプションを指定します。 ILINKARM_CMD_LINE=--config full.icf --silent

表 28: ILINK 環境変数

インクルードファイル検索手順

コンパイラの #include ファイル検索手順の詳細を以下に示します。

- #include ファイルの名前が角括弧または二重引用符で指定された絶対パスの場合は、そのファイルが開きます。
- 以下のように #include ファイルの名前が角括弧で囲まれている場合

  ```
  #include <stdio.h>
  ```

  以下のディレクトリでインクルード対象ファイルが検索されます。

  1 -I オプションで指定されるディレクトリ。指定順に検索されます（268 ページの *-I* を参照）。

  2 C_INCLUDE 環境変数で指定されるディレクトリ（設定されている場合）（237 ページの*環境変数*を参照）。

  3 自動的に設定されたライブラリシステムには、ディレクトリが含まれます。262 ページの *--dlib_config* を参照してください。

- 次のように #include ファイルの名前が二重引用符で囲まれている場合

  ```
  #include "vars.h"
  ```

  #include 文が記述されているソースファイルのあるディレクトリが検索され、その後、角括弧で囲まれたファイル名の場合と同じ手順が実行されます。

#include ファイルが入れ子になっている場合は、最後にインクルードされたファイルのあるディレクトリから検索が開始され、上位方向に各インクルードファイルの検索が繰り返され、最後にソースファイルのディレクトリが検索されます。次に例を示します。

```
src.c in directory dir¥src
  #include "src.h"
  ...
src.h in directory dir¥include
  #include "config.h"
  ...
```

dir¥exe がカレントディレクトリの場合は、以下のコマンドを使用してコンパイルします。

```
iccarm ..¥src¥src.c -I..¥include -I..¥debugconfig
```

すると、以下のディレクトリ（記載順）で config.h ファイルが検索されます。この例では、このファイルは dir¥debugconfig ディレクトリにあります。

dir¥include	現在のファイルは src.h です。
dir¥src	現在のファイルが含まれるファイル (src.c)。
dir¥include	最初の -I オプションで指定した通りになります。
dir¥debugconfig	2 番目の -I オプションで指定した通りになります。

stdio.h などの標準ヘッダファイルは角括弧、アプリケーション用のヘッダファイルは二重引用符で囲んでください。

注 : ¥ および / は両方ともディレクトリの区切り文字として使用できます。

ヘッダファイルをインクルードする構文については、419 ページの *プリプロセッサの概要* を参照してください。

コンパイラ出力

コンパイラでは、以下の出力を生成できます。

- リンク可能オブジェクトファイル

  コンパイラにより生成されるオブジェクトファイルは、業界標準フォーマットの ELF を使用します。デフォルトでは、オブジェクトファイルは o のファイル名拡張子を持ちます。

- リストファイル（オプション）

  コンパイラオプション `-l` を使用して、さまざまな種類のリストファイルを指定できます（269 ページの *-l* を参照）。デフォルトでは、これらのファイルのファイル名拡張子は `lst` です。

- プロセッサ出力ファイル（オプション）

  プロセッサ出力ファイルを作成するには、`--preprocess` オプションを使用します。デフォルトでは、このファイルのファイル名拡張子は `i` です。

- 診断メッセージ

  診断メッセージは、標準エラーストリームに転送されて画面上に表示されるほか、オプションのリストファイルにも出力されます。診断メッセージの情報については、「241 ページの*診断*」を参照してください。

- エラーリターンコード

  これらのコードは、バッチファイル内で評価可能なステータス情報をオペレーティングシステムに提供します（239 ページの*エラーリターンコード*を参照）。

- サイズ情報

  関数およびメモリごとのデータに対して生成されたバイト数に関する情報が標準出力ストリームに転送され、画面上に表示されます。それらのバイトの一部が *「共有」*として報告されることもあります。

  共有オブジェクトとは、モジュール間で共有される関数またはデータオブジェクトのことです。このような共有が 2 つ以上のモジュールで発生した場合、1 つの関数 / データオブジェクトのみが保持されます。たとえば、インライン関数がインライン化されない場合があります。これは、これらの関数が共有とマークされていて、各関数の 1 つのインスタンスしか最終的なアプリケーションにインクルードされないためです。この仕組みは、特定の関数や変数に直接的には関連しないコンパイラ生成コードやデータで、最終的なアプリケーションには 1 つのインスタンスしか必要とされない場合にも使用されることがあります。

エラーリターンコード

コンパイラおよびリンカは、バッチファイル内で評価可能なステータス情報をオペレーティングシステムに返します。

以下のコマンドラインエラーコードがサポートされています。

コード	説明
0	コンパイルまたはリンク処理は成功していますが、ワーニングが発生した可能性があります。

表 29: エラーリターンコード

コード	説明
1	オプション --warnings_affect_exit_code の使用中にワーニングが発生しました。
2	エラーが発生しました。
3	致命的なエラーが発生したためツールが停止しました。
4	インターナルなエラーが発生したためツールが停止しました。

表 29: エラーリターンコード（続き）

ILINK 出力

ILINK では、以下の出力を生成できます。

- 絶対実行可能イメージ

  IAR ILINK リンカは、最終出力として、実行可能イメージを含む絶対オブジェクトファイルを生成します。このファイルは、EPROM に格納したり、ハードウェアエミュレータにダウンロードしたり、IAR C-SPY デバッガシミュレータを使用して PC 上で実行できます。デフォルトでは、ファイルは out のファイル名拡張子を持ちます。出力フォーマットは、常に ELF です。これは、オプションで DWARF フォーマットのデバッグ情報を含みます。

- オプションのログイン情報

  操作中、ILINK は、その決定を stdout、およびオプションでファイルに記録します。たとえば、ライブラリが検索される場合、必要なシンボルがライブラリモジュールで見つかったかどうか、またはモジュールが出力の一部になるかどうかが記録されます。各 ILINK サブシステムのタイミング情報も記録されます。

- オプションのマップファイル

  リンカマップファイル（リンク、ランタイム属性、メモリ、配置のサマリ、エントリリストを含くむ）は、ILINK オプション --map を使用して生成できます（311 ページの *--map* を参照）。デフォルトでは、マップファイルのファイル名拡張子は map です。

- 診断メッセージ

  診断メッセージは、stderr に転送され、画面上に表示されるほか、オプションのマップファイルにも出力されます。診断メッセージの情報については、「241 ページの*診断*」を参照してください。

- エラーリターンコード

  ILINK は、バッチファイル内で評価可能なステータス情報をオペレーティングシステムに提供します（239 ページのエラー*リターンコード*を参照）。

- 使用メモリのサイズ情報および経過時間
  関数およびメモリごとのデータに対して生成されたバイト数に関する情報が stdout に転送され、画面上に表示されます。

診断

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

コンパイラのメッセージフォーマット

診断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。コンパイラの診断メッセージは、次のフォーマットで生成されます。

filename,*linenumber level*[*tag*]: *message*

各エレメントの意味は以下のとおりです。

filename	問題が発生したソースファイルの名前
linenumber	コンパイラが問題を検出した行の番号
level	問題の重要度のレベル
tag	診断メッセージを示す固有のタグ
message	説明（場合によっては複数行）

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。

オプション --diagnostics_tables を使用すると、すべてのコンパイラ診断メッセージが一覧表示されます。

リンカのメッセージフォーマット

診断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。ILINK が生成する診断メッセージは、通常次のような形式です。

level[*tag*]: *message*

各エレメントの意味は以下のとおりです。

level	問題の重要度のレベル
tag	診断メッセージを示す固有のタグ
message	説明（場合によっては複数行）

診断メッセージは、オプションのマップファイルに出力されるとともに、画面に表示されます。

オプション `--diagnostics_tables` を使用すると、すべてのリンカ診断メッセージが一覧表示されます。

重要度

診断メッセージは、以下の重要度に分類されます。

リマーク

生成したコードで誤った動作を引き起こす可能性がある構造をコンパイラまたはリンカが検出した場合に生成される診断メッセージ。リマークはデフォルトでは出力されません。有効にする方法については、285 ページの *--remarks* を参照してください。

ワーニング

コンパイラまたはリンカがコードの生成に支障のあるような潜在的な問題プログラミングのエラーや漏れはあるが、コンパイルやリンクの途中終了の原因にはならないものを示します。ワーニングは、`--no_warnings` コマンドラインオプションを使用して無効にできます（280 ページの *--no_warnings* を参照）。

エラー

コンパイラまたはリンカで重大なエラーが見つかったときに生成される診断メッセージ。エラーが発生した場合は、ゼロ以外の終了コードが生成されます。

致命的なエラー

コードが生成できなくなるだけでなく、それ以降の処理が無意味となるような状態をコンパイラが検出した場合に生成される診断メッセージ。このメッセージが出力された後、コンパイルが終了します。致命的なエラーが発生した場合は、ゼロ以外の終了コードが生成されます。

重要度の設定

致命的なエラーや一部の通常エラーを除くすべての診断メッセージに対し、診断メッセージの出力抑制や重要度の変更ができます。

重要度の設定に使用可能なコンパイラオプションについては、*コンパイラオプション*を参照してください。

コンパイラについては、重要度レベルの設定に使用可能なプラグマディレクティブについては、*プラグマディレクティブ*を参照してください。

内部エラー

内部エラーは、コンパイラまたはリンカでの問題が原因で、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。このメッセージは、以下の形式で生成されます。

```
Internal error: message
```

ここで、*message* はエラーの説明を示します。内部エラーが発生した場合は、ソフトウェアの配布元か 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、問題を再現できるように、以下の情報をお知らせください。

- 製品名
- コンパイラまたはILINKのバージョン番号(コンパイラまたはILINKが生成するリストまたはマップファイルのヘッダ部分にあります)
- 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アプリケーションの関連ファイル
- 内部エラー発生時に指定していたオプションの一覧

コンパイラオプション

- オプションの構文
- コンパイラオプションの概要
- コンパイラオプションの説明

オプションの構文

コンパイラオプションとは、コンパイラのデフォルトの動作を変更するためのパラメータです。オプションの指定は、コマンドラインまたは IDE 内から行えます。ここでは、コマンドラインからの指定について詳細に説明します。

IDE で使用可能なコンパイラオプションとそれらの設定方法については、オンラインヘルプシステムを参照してください。

オプションのタイプ

コマンドラインオプションには、*省略形*の名前と*完全形*の名前の 2 *種類*があります。一部のオプションは両方の名前を持ちます。

- オプションの省略形は 1 文字で構成され、パラメータが付くこともあります。このフォーマットで指定する場合は、`-e` のようにダッシュを付けて入力します。
- オプションの完全形は、複数の語をアンダースコアで連結した形で構成され、パラメータが付くこともあります。このフォーマットで指定する場合は、`--char_is_signed` のようにダッシュを 2 個付けて入力します。

さまざまなオプションの渡し方については、236 ページの*オプションの受渡し*を参照してください。

パラメータの指定に関する規則

オプションパラメータの指定に関する一般的な構文規則があります。最初に、パラメータが*任意指定*か*必須*かどうか、オプション名が省略形か完全形かどうかに分けて規則を説明します。次に、ファイル名およびディレクトリを指定するための規則を一覧で示します。最後に、残りの規則を一覧で示します。

任意指定パラメータの規則

省略形のオプションで任意指定パラメータを伴なう場合は、以下のように、パラメータの前にスペースを空けずに指定します。

-O または -Oh

完全形のオプションで任意指定パラメータを伴なう場合は、以下のように、パラメータの前に等号 (=) を付けて指定します。

--misrac2004=*n*

必須パラメータの規則

省略形のオプションで必須パラメータを伴なう場合は、以下のように、パラメータの前にスペースを空けても空けなくてもかまいません。

-I..¥src または -I ..¥src¥

完全形のオプションで必須パラメータを伴なう場合は、以下のように、パラメータの前に等号 (=) を付けるかスペースを空けて指定します。

--diagnostics_tables=MyDiagnostics.lst

または

--diagnostics_tables MyDiagnostics.lst

任意指定パラメータと必須パラメータの両方を伴なうオプションの規則

任意指定パラメータと必須パラメータの両方をとるオプションでのパラメータ指定の規則は以下のとおりです。

- 省略形のオプションでは、任意指定パラメータの前にスペースを空けずに指定します
- 完全形のオプションでは、任意指定パラメータの前に等号 (=) を付けて指定します
- 省略形および完全形のオプションでは、必須パラメータの前にスペースを空けて指定します

省略形のオプションで、任意指定パラメータの後に必須パラメータを指定する例を以下に示します。

-lA MyList.lst

完全形のオプションで、任意指定パラメータの後に必須パラメータを指定する例を以下に示します。

--preprocess=n PreprocOutput.lst

ファイル名またはディレクトリをパラメータとして指定する場合の規則

ファイル名またはディレクトリをパラメータとして指定するオプションの規則は以下のとおりです。

- ファイル名をパラメータとして指定するオプションは、ファイルパスの指定も可能です。パスは、相対パスと絶対パスのいずれでもかまいません。たとえば、..¥listings¥ ディレクトリのファイル List.lst のリストを生成するには、以下のように入力します。

  ```
  iccarm prog.c -l ..¥listings¥List.lst
  ```

- ファイル名を出力先として指定するオプションの場合、ファイル名のないパスとしてパラメータを指定できます。コンパイラは、このディレクトリ内のオプションに基づいた拡張子を持つファイルに出力を保存します。ファイル名は、コンパイルしたソースファイルの名前と同じになります。ただし、-o オプションで別の名前を指定した場合は、その名前が使用されます。次に例を示します。

  ```
  iccarm prog.c -l ..¥listings¥
  ```

  生成されるリストファイルには、デフォルト名の ..¥listings¥prog.lst が付けられます。

- *現在のディレクトリ*は、以下のように、ピリオド (.) で指定します。次に例を示します。

  ```
  iccarm prog.c -l .
  ```

- / をディレクトリの区切り文字として ¥ の代わりに使用できます。
- - を指定することにより、入力ファイルおよび出力ファイルがそれぞれ、標準の入力および出力ストリームにリダイレクトされます。次に例を示します。

  ```
  iccarm prog.c -l -
  ```

その他の規則

さらに、以下の規則も適用されます。

- オプションでパラメータを指定する場合は、パラメータの最初にダッシュ (-) を付け、その後に別の文字を続けることはできません。その代わりに、パラメータのプレフィックスとして 2 個のダッシュを指定します。以下の例は、-r という名前のリストファイルを作成します。

  ```
  iccarm prog.c -l ---r
  ```

- 同じ型の複数の引数を指定可能なオプションの場合、引数は、以下の例のようにカンマ区切り（スペースなし）のリストとして指定できます。

  ```
  --diag_warning=Be0001,Be0002
  ```

また、以下のように、引数ごとにオプションを繰り返して指定することもできます。

```
--diag_warning=Be0001
--diag_warning=Be0002
```

コンパイラオプションの概要

以下の表に、コンパイラのコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--aapcs	呼出し規約を指定します
--aeabi	AEABI 準拠のコード生成を有効にします
--align_sp_on_irq	__irq 関数への入り口で SP を整列されるコードを生成します
--arm	デフォルトの関数モードを ARM に設定します
--c89	C89 の派生言語を指定します
--char_is_signed	char を符号付として処理
--char_is_unsigned	char を符号なしとして処理
--cpu	プロセッサ選択を指定します
--cpu_mode	関数のデフォルト CPU モードを指定します
--c++	標準 C++ を指定します
-D	プリプロセッサシンボルを定義
--debug	デバッガ情報を生成
--dependencies	ファイル依存関係をリスト化
--diag_error	エラーとして処理
--diag_remark	リマークとして処理
--diag_suppress	診断を無効化
--diag_warning	ワーニングとして処理
--diagnostics_tables	すべての診断メッセージをリスト化
--discard_unused_publics	未使用のパブリックシンボルを破棄
--dlib_config	DLIB ライブラリのシステムインクルードファイルを使用して、どのライブラリの設定を使用するかを決定
-e	言語拡張を有効化
--ec++	Embedded C++ を指定

表 30: コンパイラオプションの一覧

コマンドラインオプション	説明
--eec++	Extended Embedded C++ を指定
--enable_hardware_workaround	個別のハードウェア対策を有効化します
--enable_multibytes	ソースファイルのマルチバイト文字のサポートを有効化
--enable_restrict	標準の C のキーワード制限を有効にします
--endian	生成されるコードおよびデータのバイトオーダを指定します
--enum_is_int	列挙型の最小サイズを設定します
--error_limit	コンパイルを停止するエラー数の上限を指定
-f	コマンドラインを拡張
--fpu	浮動小数点ユニット型を選択します
--generate_entries_without_bounds	C-RUN が有効でないコードから呼出し可能な関数を生成します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください
--guard_calls	関数の静的変数初期化のガードを有効にします
--header_context	すべての参照先ソースファイルとヘッダファイルをリスト化
-I	インクルードファイルのパスを指定
--ignore_uninstrumented_pointers	C-RUN が有効でないメモリからのポインタのチェックを無効化します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください
--interwork	インターワークコードを生成します
-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オブジェクトコードを生成します
--lock_regs	コンパイラが指定のレジスタを使用しないようにします
--macro_positions_in_diagnostics	診断メッセージでマクロ内の位置を取得
--make_all_definitions_weak	すべての変数と関数の定義を弱い定義にします
--mfc	複数ファイルのコンパイルを有効化

表 30: コンパイラオプションの一覧（続き）

コマンドラインオプション	説明
--misrac	MISRA-C:1998 固有のエラーメッセージを有効にします。このオプションは --misrac1998 と同義で、旧バージョンとの互換性のためだけに使用できます
--misrac1998	MISRA-C:1998 固有のエラーメッセージを有効にします。『IAR Embedded Workbench® MISRA C:1998 リファレンスガイド』を参照してください
--misrac2004	MISRA-C:2004 固有のエラーメッセージを有効にします。『IAR Embedded Workbench® MISRA C:2004 リファレンスガイド』を参照してください
--misrac_verbose	『IAR Embedded Workbench® MISRA C:1998 リファレンスガイド』または『IAR Embedded Workbench® MISRA C:2004 リファレンスガイド』を参照してください
--no_alignment_reduction	単純な thumb 関数のアラインメント減少を無効にします
--no_clustering	静的クラスタ最適化を無効にします
--no_code_motion	コード移動最適化を無効化
--no_const_align	定数のアラインメント最適化を無効にします
--no_cse	共通部分式除去を無効化
--no_exceptions	C++ 例外のサポートを無効化
--no_fragments	セクションフラグメント処理を無効化
--no_inline	関数インライン化を無効化
--no_literal_pool	データの読取りが許可されておらず、コードの実行のみが可能なメモリ領域から実行されるコードを生成します
--no_loop_align	ループ内のラベルのアラインメントを無効化
--no_mem_idioms	コンパイラで、特定のメモリアクセスのパターンを最適化しないようにします
--no_path_in_file_macros	シンボル __FILE__ および __BASE_FILE__ のリターン値からパスを削除
--no_rtti	C++ RTTI のサポートを無効化
--no_rw_dynamic_init	静的 C 変数のランタイムの初期化を無効化
--no_scheduling	命令スケジューラを無効にします

表 30: コンパイラオプションの一覧（続き）

コマンドラインオプション	説明
--no_size_constraints	速度を優先して最適化するときに、コードサイズ拡張の通常の制限を緩和します
--no_static_destruction	プログラム終了時に C++ 静的変数の破壊を無効化します
--no_system_include	システムインクルードファイルの自動検索を無効化します
--no_tbaa	型ベースエイリアス解析を無効化
--no_typedefs_in_diagnostics	診断での typedef 名の使用を無効化
--no_unaligned_access	アラインメントされないアクセスを回避します
--no_unroll	ループ展開を無効化
--no_warnings	すべての警告を無効化
--no_wrap_diagnostics	診断メッセージのラッピングを無効化
-O	最適化レベルを設定
-o	オブジェクトファイル名を設定。--output のエイリアス
--only_stdout	標準出力のみを使用
--output	オブジェクトファイル名を設定
--predef_macros	定義済シンボルの一覧を表示
--preinclude	ソースファイルを読み込む前にインクルードファイルをインクルード
--preprocess	プリプロセッサ出力を生成
--public_equ	グローバル名のアセンブララベルを定義
-r	デバッガ情報を生成。--debug のエイリアス
--relaxed_fp	浮動小数点式の最適化規則を緩和します
--remarks	リマークを有効化
--require_prototypes	関数が定義前に宣言されていることを検証
--ropi	アドレスコードおよびリードオンリーのデータへの PC 関連の参照を使用するコードを生成
--runtime_checking	ランタイムのエラー解析を有効にします。*ARM 用 C-SPY® デバッグガイド*で C-RUN のドキュメントを参照してください

表 30: コンパイラオプションの一覧（続き）

コマンドラインオプション	説明
--rwpi	静的ベースレジスタからアドレス書込み可能なデータへのオフセットを使用するコードを生成
--section	セクション名を変更
--separate_cluster_for_initialized_variables	初期化変数と非初期化変数を分離
--silent	サイレント処理を設定
--strict	標準 C/C++ への厳密な準拠を確認
--system_include_dir	システムインクルードファイルのパスを指定
--thumb	デフォルトの関数モードを Thumb に設定します
--use_c++_inline	C99 で C++ インライン動作を使用
--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
--vectorize	NEON ベクタ命令の生成を有効にする
--vla	C99 VLA のサポートを有効化
--warn_about_c_style_casts	C- スタイルキャストが C++ ソースコードで使用されるときにコンパイラを警告
--warnings_affect_exit_code	ワーニングが終了コードに影響
--warnings_are_errors	ワーニングをエラーとして処理

表 30: コンパイラオプションの一覧（続き）

コンパイラオプションの説明

次のセクションでは、それぞれのコンパイラオプションについて詳細に説明します。

[追加オプション] ページを使用して特定のコマンドラインオプションを指定する場合、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。

--aapcs

構文

```
aapcs={std|vfp}
```

パラメータ

`std`	AAPCS 規格に従って、関数呼出しでの浮動小数点パラメータおよびリターン値にプロセッサレジスタが使用されます。ソフトウェア FPU が選択されている場合は、`std` がデフォルトです。
`vfp`	浮動小数点パラメータおよびリターン値に VFP レジスタが使用されます。生成されたコードは AEAB コードに準拠していません。VFP ユニットが使用されている場合、`vfp` がデフォルトです。

説明

このオプションは、浮動小数点の呼出し規約を指定するときに使用します。

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] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--aeabi

構文

```
--aeabi
```

説明

このオプションは、AEABI 準拠のオブジェクトコードを生成するときに使用します。このオプションは、`--guard_calls` オプションとともに使用する必要があります。

関連項目

207 ページの *AEABI への準拠*、268 ページの *--guard_calls*。

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] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--align_sp_on_irq

構文

```
--align_sp_on_irq
```

説明

このオプションを使用して、`__irq` により宣言された関数への入り口でスタックポインタ (`SP`) を整列させます。

これは、割込みコードが割込みハンドラと同じ `SP` を使用する、ネストされた割込みに特に便利です。つまり、スタックは AEABI（および一部のコアでコ

ンパイラにより生成される特定の命令）で必要とされる 8 バイトのアラインメントではなく、4 バイトのアラインメントしか持たない可能性があります。

関連項目

347 ページの *__irq*。

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］>［C/C++ コンパイラ］>［追加オプション］**を選択します。

--arm

構文

```
--arm
```

説明

このオプションは、デフォルトの関数モードを ARM に設定するときに使用します。この設定は、相互作用していない限り、プログラムに含まれるすべてのファイルで同じでなければなりません。

注 : このオプションの効果は、`--cpu_mode=arm` オプションと同じです。

関連項目

269 ページの *--interwork*、346 ページの *__interwork*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［コード］>［プロセッサのモード］>［Arm］

--c89

構文

```
--c89
```

説明

このオプションを使用して、標準の C ではなく C89 C の派生言語を有効にします。

注 : このオプションは、MISRA-C のチェックが有効な場合に必須です。

関連項目

175 ページの *C 言語の概要*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C の派生言語］>［C89］

--char_is_signed

構文

```
--char_is_signed
```

説明

デフォルトでは、コンパイラは単純な `char` 型を符号なしとして解釈します。このオプションは、コンパイラで単純な `char` 型を符号付きと解釈する場合に

使用します。これは、他のコンパイラとの互換性を確保する場合などに便利です。

注 : ランタイムライブラリは --char_is_signed オプションを使用せずにコンパイルされ、このオプションによってコンパイルされたコードとは一緒に使用できません。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 2］>［'CHAR' の型］

--char_is_unsigned

構文

```
--char_is_unsigned
```

説明

このオプションは、コンパイラで単純な char 型を符号なしと解釈する場合に使用します。これは、単純な char 型のデフォルトの解釈です。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 2］>［'CHAR' の型］

--cpu

構文

```
--cpu=core
```

パラメータ

core	特定のプロセッサ選択を指定します

説明

このオプションは、コードの生成対象のプロセッサ選択をする場合に使用します。デフォルトは、ARM7TDMI です。以下のコアおよびプロセッサマクロセルが認識されます。

ARM7TDMI	ARM1020E	Cortex-M0
ARM7TDMI-S	ARM1022E	Cortex-M0+
ARM710T	ARM1026EJ-S	Cortex-M1
ARM720T	ARM1136J	Cortex-Ms1*
ARM740T	ARM1136J-S	Cortex-M3
ARM7EJ-S	ARM1136JF	Cortex-M4
ARM9TDMI	ARM1136JF-S	Cortex-M4F

ARM920T	ARM1176J	Cortex-M7
	(ARM1176JZ のエイリアス)	
ARM922T	ARM1176J-S	Cortex-R4
	(ARM1176JZ-S のエイリアス)	
ARM940T	ARM1176JF	Cortex-R4F
	(ARM1176JZF のエイリアス)	
ARM9E	ARM1176JF-S	Cortex-R5
	(ARM1176JZF-S のエイリアス)	
ARM9E-S	Cortex-A5	Cortex-R5F
ARM926EJ-S	Cortex-A5F	Cortex-R7
ARM966E-S	Cortex-A7	Cortex-R7F
ARM968E-S	Cortex-A8	XScale
ARM946E-S	Cortex-A9	XScale-IR7
ARM10E	Cortex-A15	

* オペレーティングシステム拡張を持つ Cortex-M1。

関連項目 56 ページの *プロセッサ選択*。

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

--cpu_mode

構文

```
--cpu_mode={arm|a|thumb|t}
```

パラメータ

arm, a (デフォルト)	ARM モードを関数のデフォルトモードとして選択します
thumb, t	Thumb モードを関数のデフォルトモードとして選択します

説明

このオプションは、関数のデフォルトモードを選択するときに使用します。この設定は、相互作用していない限り、プログラムに含まれるすべてのファイルで同じでなければなりません。

関連項目

269 ページの *--interwork*、346 ページの *__interwork*。

［プロジェクト］>［オプション］>［一般オプション］>［ターゲット］>［プロセッサモード］

--c++

構文

```
--c++
```

説明

デフォルトでは、コンパイラでサポートしている言語は C 言語です。標準の C++ を使用する場合は、このオプションを使用して C++ をコンパイラで使用する言語に設定する必要があります。

関連項目

263 ページの *--ec++*、264 ページの *--eec++*、185 ページの *C++ の使用*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

および

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++ の派生言語］>［C++］

-D

構文

```
-D symbol[=value]
```

パラメータ

symbol	プリプロセッサシンボルの名前
value	プリプロセッサシンボルの値

説明

このオプションは、プリプロセッサシンボルの定義に使用します。値を指定しない場合、1 が使用されます。このオプションは、コマンドラインで任意の回数使用できます。

-D オプションは、ソースファイルの最初に #define 文を記述した場合と同様に機能します。

```
-Dsymbol
```

は、以下の文と等価です。

```
#define symbol 1
```

以下の文と等価なコマンドを考えてみます。

```
#define FOO
```

この文と同じ結果を得るには、以下のように、= 記号を付け、その後には何も付けずに指定します。

```
-DFOO=
```

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [プリプロセッサ] > [シンボル定義]

--debug, -r

構文

```
--debug
-r
```

説明

--debug や -r オプションは、IAR C-SPY® デバッガまたは他のシンボリックデバッガで必要なコンパイラのインクルード情報をオブジェクトモジュールに含める場合に使用します。

注 : デバッグ情報を含めると、オブジェクトファイルのサイズが増加します。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [出力] > [デバッグ情報の生成]

--dependencies

構文

```
--dependencies[=[i|m|n][s]] {filename|directory|+}
```

パラメータ

i（デフォルト）	ファイルの名前のみをリスト化
m	makefile スタイルでリスト化（複数のルール）
n	makefile スタイルでリスト化（1 つのルール）
s	システムファイルの無効化
+	-o と同じ内容を出力しますが、ファイル名拡張子が d となります

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則* を参照してください。

説明

このオプションは、ファイルへの入力のために開かれたすべてのソースファイルおよびヘッダファイルの一覧を、デフォルトのファイル名拡張子 i を持つファイルに含める場合に使用します。

例

--dependencies や --dependencies=i を使用すると、開かれている各入力ファイルの名前とフルパス（ある場合）が独立した行に出力されます。次に例を示します。

```
c:¥iar¥product¥include¥stdio.h
d:¥myproject¥include¥foo.h
```

--dependencies=m を使用した場合は、makefile スタイルで出力されます。各入力ファイルについて、makefile の依存関係規則を含む行が生成されます。各行は、オブジェクトファイル名、コロン、空白文字、入力ファイル名で構成されます。次に例を示します。

```
foo.o: c:¥iar¥product¥include¥stdio.h
foo.o: d:¥myproject¥include¥foo.h
```

たとえば、gmake (GNU make) などの一般的な make ユーティリティで --dependencies を使用するには、以下のように操作します。

1 以下のように、ファイルのコンパイル規則を設定します。

```
%.o : %.c
      $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

すなわち、このコマンドを使用すると、オブジェクトファイルの生成に加えて、makefile スタイルで依存関係ファイルが生成されます（この例では、拡張子に .d を使用）。

2 以下のようにして、すべての依存関係ファイルを makefile に含めます。

```
-include $(sources:.c=.d)
```

ダッシュ (-) があるため、.d ファイルがまだ存在しない最初の時点でも機能します。

このオプションは、IDE では使用できません。

--diag_error

構文

```
--diag_error=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージをエラーとして再分類する場合に使用します。エラーは、オブジェクトコードが生成されなくなるような重大な C/C++ 言語規則の違反を示します。終了コードはゼロ以外になります。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。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［エラーとして処理］

--diag_remark

構文

```
--diag_remark=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe177` など）

説明

このオプションは、特定の診断メッセージをリマークとして再分類する場合に使用します。リマークは、最も軽微な診断メッセージです。生成されたコードに異常動作の原因となる可能性があるソースコード構造が存在することを示します。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。

注： デフォルトでは、リマークは表示されません。リマークを表示するには、`--remarks` オプションを使用します。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［リマークとして処理］

--diag_suppress

構文

```
--diag_suppress=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe117` など）

説明

このオプションは、特定の診断メッセージを無効にする場合に使用します。これらのメッセージは表示されなくなります。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。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［診断を無効化］

--diag_warning

構文

```
--diag_warning=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe826 など）

説明

このオプションは、特定の診断メッセージをワーニングとして再分類する場合に使用します。ワーニングは、問題はあるが、コンパイルの途中終了の原因にはならないエラーや脱落を示します。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。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［ワーニングとして処理］

--diagnostics_tables

構文

```
--diagnostics_tables {filename|directory}
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、すべての診断メッセージを指定ファイルに保存する場合に使用します。これは、プラグマディレクティブを使用して診断メッセージの重要度を無効化または変更したが、その理由を記述し忘れた場合などに便利です。

このオプションは、他のオプションと併用できません。

このオプションは、IDE では使用できません。

--discard_unused_publics

構文

```
--discard_unused_publics
```

説明

このオプションは、--mfc コンパイラオプションでコンパイルする際に未使用のパブリック関数と変数を破棄するときに使用します。

注 : このオプションをアプリケーションの一部のみで使用しないでください。生成された出力から必要なシンボルが削除されることがあります。オブジェクト属性 __root を使用して、割込みハンドラなどコンパイルユニット外から

使用されるシンボルを保持します。シンボルが __root 属性を持たず、ライブラリで定義される場合、代わりにそのライブラリ定義が使用されます。

関連項目

272 ページの *--mfc*、222 ページの *複数ファイルのコンパイルユニット*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［未使用のパブリックを破棄］

--dlib_config

構文

```
--dlib_config filename.h|config
```

パラメータ

filename	DLIB 設定ヘッダファイル、247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
config	指定された設定のデフォルト設定ファイルが使用されます。以下から選択します。 none。設定は使用されません。 normal。通常のライブラリ設定が使用されます（デフォルト）。 full。フルライブラリ設定が使用されます。

説明

このオプションを使用して、明示的なファイルを指定するか、ライブラリ設定を指定することによって、どのライブラリ設定を使用するかを指定します。ライブラリ設定を指定する場合は、ライブラリ設定のデフォルトファイルが使用されます。使用するライブラリに対応する設定を指定してください。このオプションを指定しない場合、デフォルトのライブラリ設定ファイルが使用されます。

すべてのビルド済ランタイムライブラリについて、対応する設定ファイルが提供されています。ライブラリオブジェクトファイルやライブラリ設定ファイルは、arm¥lib ディレクトリにあります。ビルド済ランタイムライブラリの例と詳細については、112 ページの *ビルド済ライブラリの使用*を参照してください。

自分でビルドしたランタイムライブラリの場合は、対応するカスタマイズしたライブラリ設定ファイルも作成し、コンパイラで指定する必要があります。詳細については、124 ページの *カスタマイズしたライブラリのビルドと使用*を参照してください。

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］>［一般オプション］>［ライブラリ構成］

-e

構文

```
-e
```

説明

コマンドラインバージョンのコンパイラのデフォルトでは、言語拡張が無効になっています。拡張キーワードや匿名構造体 / 共用体などの言語拡張をソースコードで使用する場合には、このオプションを使用して有効化する必要があります。

注 : `-e` オプションと `--strict` オプションは、同時に使用できません。

関連項目

177 ページの *言語拡張の有効化*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［標準（IAR 拡張あり）］

注 : IDE では、このオプションがデフォルトで選択されています。

--ec++

構文

```
--ec++
```

説明

コンパイラでは、デフォルトの言語は C 言語です。Embedded C++ を使用する場合は、このオプションを使用してコンパイラで使用する言語に Embedded C++ を設定する必要があります。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

および

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++ の派生言語］>［Embedded C++］

--eec++

構文

```
--eec++
```

説明

名前空間や標準テンプレートライブラリなどの拡張 Embedded C++ の機能をソースコードで利用する場合は、このオプションを使用して、コンパイラが使用する言語を拡張 Embedded C++ に設定する必要があります。

関連項目

186 ページの *拡張 Embedded C++*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

および

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++ の派生言語］>［拡張 Embedded C++］

--enable_hardware_workaround

構文

```
--enable_hardware_workaround=waid[,waid...]
```

パラメータ

waid	有効化対策の ID 番号です。使用可能な有効化対策の一覧については、リリースノートを参照してください。

説明

このオプションは、特定のハードウェア問題の対策をコンパイラで生成する場合に使用します。

関連項目

使用可能なパラメータの一覧については、コンパイラのリリースノートを参照してください。

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］>［C/C++ コンパイラ］>［追加オプション］**を選択します。

--enable_multibytes

構文

```
--enable_multibytes
```

説明

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

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

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 2］>［マルチバイト文字サポートを有効にする］

--enable_restrict

構文

```
--enable_restrict
```

説明

標準の C のキーワード制限を有効にします。このオプションは最適化中の分析の制度を向上するために役立ちます。

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］>［C/C++ コンパイラ］>［追加オプション］を選択します。

--endian

構文

```
--endian=
```

パラメータ

`big, b`	ビッグエンディアンをデフォルトのバイトオーダとして指定します
`little, l`（デフォルト）	リトルエンディアンをデフォルトのバイトオーダとして指定します

説明

このオプションは、生成されるコートおよびデータのバイトオーダを指定するときに使用します。デフォルトでは、コンパイラは、リトルエンディアンバイトオーダでコードを生成します。

関連項目

326 ページの*バイトオーダ*。

［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］

--enum_is_int

構文

```
--enum_is_int
```

説明

このオプションは、列挙型のサイズを少なくとも 4 バイトに強制的に指定するときに使用します。

注 : このオプションでは、整数型よりサイズの大きい enum 型を使用できるということは考慮されません。

関連項目 327 ページの *enum 型*。

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］> ［C/C++ コンパイラ］> ［追加オプション］**を選択します。

--error_limit

構文

```
--error_limit=n
```

パラメータ

n	コンパイラがコンパイルを中止するエラー発生回数（*n* は正の整数）。0 は制限なしを示します。

説明

`--error_limit` オプションは、コンパイラがコンパイルを停止するエラー数を指定するために使用します。デフォルトでは、エラー数の上限は 100 です。

このオプションは、IDE では使用できません。

-f

構文

```
-f filename
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、コンパイラで、指定ファイル（デフォルトのファイル名拡張子は `xcl`）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。

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］> ［C/C++ コンパイラ］> ［追加オプション］**を選択します。

--fpu

構文

```
--fpu={VFPv2|VFPv3|VFPv3_d16|VFPv4|VFPv4_sp|VFPv4_d16|VFPv5_sp|VFPv5_d16|VFP9-S|none}
```

パラメータ

VFPv2	VFPv2 アーキテクチャに準拠した VFP ユニットを実装するシステム。
VFPv3	VFPv3 アーキテクチャに準拠した VFP ユニットを実装するシステム。
VFPv3_d16	VFPv3 アーキテクチャの D16 派生品に準拠した VFP ユニットを実装するシステム。
VFPv4	VFPv4 アーキテクチャに準拠した VFP ユニットを実装するシステム。
VFPv4_d16	VFPv4 アーキテクチャの倍精度 (D16) 派生品に準拠した VFP ユニットを実装するシステム。
VFPv4_sp	VFPv4 アーキテクチャの単精度派生品に準拠した VFP ユニットを実装するシステム。
VFPv5_sp	VFPv5 アーキテクチャの単精度派生品に準拠した VFP ユニットを実装するシステム。
VFPv5_d16	VFPv5 アーキテクチャの倍精度 (D16) 派生品に準拠した VFP ユニットを実装するシステム。
VFP9-S	CPU コアの ARM9E ファミリと使用できる VFPv2 アーキテクチャの実装である VFP9-S。そのため、VFP9-S コプロセッサを選択することは、VFPv2 アーキテクチャを選択することと同じです。
none（デフォルト）	ソフトウェア浮動小数点ライブラリが使用されます。

説明

このオプションは、ベクタ浮動小数点 (VFP) コプロセッサを使用して浮動小数点演算を実行するコードを生成するときに使用します。VFP コプロセッサを選択することで、ソフトウェア浮動小数点ライブラリの使用を、サポートされたすべての浮動小数点演算にオーバライドします。

関連項目

57 ページの *VFP および浮動小数点演算*。

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

--guard_calls

構文

```
--guard_calls
```

説明

このオプションは、関数の静的変数初期化のガードを有効にするときに使用します。このオプションは、スレッド環境で使用してください。

関連項目

143 ページの*マルチスレッド環境の管理*。

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］>［C/C++ コンパイラ］>［追加オプション］**を選択します。

--header_context

構文

```
--header_context
```

説明

問題の原因を特定するために、どのソースファイルからどのヘッダファイルがインクルードされたかの確認が必要となる場合があります。このオプションは、診断メッセージごとに、ソースでの問題の位置に加えて、その時点でのインクルードスタック全体を表示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。

-I

構文

```
-I path
```

パラメータ

path	`#include` ファイルの検索パス

説明

このオプションは、`#include` ファイルの検索パスの指定に使用します。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。

関連項目

237 ページの*インクルードファイル検索手順*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［プリプロセッサ］>［追加インクルードディレクトリ］

--interwork

構文

```
--interwork
```

説明

このオプションは、相互作用コードを生成するときに使用します。

このオプションでコンパイルされたコードでは、関数は、デフォルトで相互作用型になります。すべてが `--interwork` オプションでコンパイルされている限り、`arm` および `thumb`（`--cpu_mode` オプションを使用）としてコンパイルされたファイルを混合できます。

注 : ARM アーキテクチャ v5 以降でコンパイルされたソースコード、または AEABI 準拠のコードは、デフォルトで相互作用します。

関連項目

346 ページの *__interwork*。

[プロジェクト] > [オプション] > [一般オプション] > [ターゲット] > [インターワークコードを生成]

-l

構文

```
-l[a|A|b|B|c|C|D][N][H] {filename|directory}
```

パラメータ

`a`（デフォルト）アセンブラリストファイル

`A` C/C++ ソースをコメントとして記述したアセンブラリストファイル

`b` 基本アセンブラリストファイル。このファイルは、`-la` を使用して生成したリストファイルと同様の内容になっていますが、コンパイラが生成する追加情報（ランタイムモデル属性、呼出しフレーム情報、フレームサイズ情報）は含まれていません *

`B` 基本アセンブラリストファイル。このファイルは、`-lA` を使用して生成したリストファイルと同様の内容になっていますが、コンパイラが生成する追加情報（ランタイムモデル属性、呼出しフレーム情報、フレームサイズ情報）は含まれていません *

`c` C/C++ リストファイル

`C`（デフォルト）アセンブラソースをコメントとして記述した C/C++ リストファイル

D	C/C++ コメントとしてアセンブラソースを持つリストファイル。ただし、命令オフセットと 16 進数バイト値はなし
N	ファイルに診断を含めません
H	ヘッダファイルのソース行を出力に含めます。このオプションを指定しない場合は、1 次ソースファイルのソース行のみ含まれます

* この場合、リストファイルはアセンブラへの入力としては使用しにくくなりますが、人には読みやすくなります。

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、アセンブラまたは C/C++ リストをファイルに出力する場合に使用します。このオプションは、コマンドラインで任意の回数使用できます。

オプションを設定するには、以下のように選択します。

［プロジェクト］> ［オプション］> ［C/C++ コンパイラ］> ［リスト］

--legacy

構文

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

パラメータ

RVCT3.0	RVCT3.0 でリンカとリンク可能なオブジェクトコードを生成します。RVCT3.0 でリンカとリンクされる必要のあるコードを生成する場合には、このモードとともに `--aeabi` オプションを使用してください。

説明

このオプションは、指定のツールチェーンと互換性のあるコードを生成するときに使用します。

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］> ［C/C++ コンパイラ］> ［追加オプション］**を選択します。

--lock_regs

構文

```
--lock_regs=register
```

パラメータ

レジスタ	コンマ区切りレジスタ名リスト、およびロックするレジスタ間隔（R4 ～ R11 の範囲）。

説明

このオプションは、指定されたレジスタを使用するコードをコンパイラで生成しないようにするときに使用します。

例

```
--lock_regs=R4
--lock_regs=R8-R11
--lock_regs=R4,R8-R11
```

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] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--macro_positions_in_diagnostics

構文

```
--macro_positions_in_diagnostics
```

説明

このオプションを使用して、診断メッセージのマクロ内にある位置の参照を取得します。これは、マクロ内の不正なソースコード構造を検出するときに便利です。

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] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--make_all_definitions_weak

構文

```
--make_all_definitions_weak
```

説明

コンパイルユニット内のすべての変数と関数の定義を弱い定義にします。

注 : 通常は拡張キーワードやプラグラマディレクティブを使用して、個々の変数と関数の定義を弱いものにする方が理想的です。

関連項目

354 ページの *__weak*。

このオプションは、IDE では使用できません。

--mfc

構文

```
--mfc
```

説明

このオプションは、*複数ファイルのコンパイル*を有効にするときに使用します。つまり、コンパイラはコマンドラインで指定された 1 つまたは複数のソースファイルを 1 単位としてコンパイルし、それによってプロシージャ間の最適化を強化します。

注 : コンパイラでは、入力ソースコードファイルごとに 1 つのオブジェクトファイルを生成します。その際、最初のオブジェクトファイルにすべての関連データが含まれ、他のオブジェクトファイルは空になります。最初のファイルのみを生成する場合は、-o コンパイラオプションを使用し、出力ファイルを含むように指定してください。

例

```
iccarm myfile1.c myfile2.c myfile3.c --mfc
```

関連項目

261 ページの *--discard_unused_publics*、282 ページの *--output, -o*、222 ページの *複数ファイルのコンパイルユニット*。

［プロジェクト］> ［オプション］> ［C/C++ コンパイラ］> ［複数ファイルのコンパイル］

--no_alignment_reduction

構文

```
--no_alignment_reduction
```

説明

一部の単純な Thumb/Thumb2 関数は 2 バイトでアラインメントすることができます。このオプションを使用して、これらの関数を 4 バイトでアラインメントします。

このオプションは、ARM モードでコンパイルする際は機能しません。

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］> ［C/C++ コンパイラ］> ［追加オプション］**を選択します。

--no_clustering

構文

```
--no_clustering
```

説明

このオプションを使用して静的クラスタ最適化を無効にします。

注 : このオプションは最適化には影響しません。

関連項目

226 ページの*静的クラスタ*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［静的クラスタ］

--no_code_motion

構文

```
--no_code_motion
```

説明

このオプションは、コード移動最適化を無効にする場合に使用します。

注 : このオプションは、最適化レベルが［中］以上の場合にのみ有効です。

関連項目

225 ページの*コード移動*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［コード移動］

--no_const_align

構文

```
--no_const_align
```

説明

デフォルトでは、サイズが 4 バイト以上のオブジェクトに対してアラインメント 4 がコンパイラで使用されます。このオプションは、コンパイラで const オブジェクトをそれらの型のアラインメントに基づいてアラインメントする場合に使用します。

たとえば、文字列リテラルはアラインメント 1 になります。文字列リテラルは、const char 型のエレメントを持つ配列であり、この型のアラインメントが 1 になるためです。このオプションを使用すると、ROM 空間が節約される可能性がありますが、場合によっては処理速度が犠牲になります。

関連項目

325 ページの*アラインメント*。

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］>［C/C++ コンパイラ］>［追加オプション］を選択します。

--no_cse

構文

```
--no_cse
```

説明

このオプションは、共通部分式除去を無効にする場合に使用します。

注 : このオプションは、最適化レベルが［中］以上の場合にのみ有効です。

関連項目

224 ページの *共通部分式除去*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［共通部分式除去］

--no_exceptions

構文

```
--no_exceptions
```

説明

このオプションを使用して、C++ 言語での例外のサポートを無効にします。throw や try-catch のような例外文、および関数定義の例外仕様によって、エラーメッセージが生成されます。関数宣言の例外仕様は無視されます。このオプションは、--c++ コンパイラオプションと併用した場合にのみ有効です。

アプリケーションで例外が使用されない場合、このオプションを使用して例外のサポートを無効化するよう推奨します。この理由は、例外によってコードサイズが大幅に増加するためです。

関連項目

188 ページの *例外処理*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

および

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++ の派生言語］>［C++］>［例外あり］

--no_fragments

構文

```
--no_fragments
```

説明

このオプションはセクションフラグメント処理を無効にします。通常、ツールセットは、セクションフラグメント情報をリンカに転送するために IAR 独自の情報を使用します。リンカは、この情報を使用して、未使用のコードおよびデータを削除し、実行可能イメージのサイズをさらに最小化します。このオプションを使用すると、情報はオブジェクトファイルに出力されません。

関連項目

99 ページの *シンボルおよびセクションの保持*。

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］>［C/C++ コンパイラ］>［追加オプション］**を選択します。

--no_inline

構文

```
--no_inline
```

説明

このオプションは、関数インライン化を無効にする場合に使用します。

関連項目

74 ページの*インライン関数*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［関数インライン化］

--no_literal_pool

構文

```
--no_literal_pool
```

説明

このオプションを使用して、データの読取りが許可されておらず、コードの実行のみが可能なメモリ領域から実行されるコードを生成します。

このオプションを使用すると、コンパイラはリテラルプールではなく、MOV32 擬似命令を使用してアドレスおよび大きな定数を生成します。switch 文はテーブルを使用して変換されなくなり、定数データが .rodata セクションに配置されます。

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 属性を使用して、このオプションによりコンパイルされたライブラリを使用するかどうかが決まります。

このオプションは ARMv6-M コアと ARMv7 コアでのみ使用でき、--ropi や --rwpi のオプションと併用できるのは ARMv7 コアの場合のみです。

関連項目

314 ページの *--no_literal_pool*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［コード］>［コードメモリ内のデータリードなし］

--no_loop_align

構文

```
--no_loop_align
```

説明

このオプションを使用して、ループ内にあるラベルの 4 バイトのアラインメントを無効にします。このオプションは、Thumb2 モードでのみ役に立ちます。

ARM/Thumb1 モードでは、このオプションは有効ですが何の処理も実行しません。

関連項目 325 ページの *アラインメント*。

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] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--no_mem_idioms

構文

```
--no_mem_idioms
```

説明

このオプションを使用して、メモリ領域を消去、設定またはコピーするコードシーケンスをコンパイラが最適化しないようにします。これを使用しなければ、これらのメモリアクセスのパターン（イディオム）は極端に最適化され、場合によってはランタイムライブラリの呼出しが使用されることもあります。原則的に、変換にはライブラリの呼出し以外のことが関係してきます。

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] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--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。

関連項目 420 ページの *定義済プリプロセッサシンボルの詳細*。

このオプションは、IDE では使用できません。

--no_rtti

構文

```
--no_rtti
```

説明

このオプションを使用して、C++ 言語でのランタイム型情報 (RTTI) のサポートを無効にします。`dynamic_cast` および `typeid` のような RTTI 文によって、エラーメッセージが生成されます。このオプションは、`--c++` コンパイラオプションと併用した場合にのみ有効です。

関連項目

185 ページの *C++ の使用*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++］

および

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C++ の派生言語］>［C++］>［RTTI あり］

--no_rw_dynamic_init

構文

```
--no_rw_dynamic_init
```

説明

このオプションを使用して、静的 C 変数のラインタイム初期化を無効にします。

`--ropi` または `--rwpi` を使用してコンパイルされた C ソースコードは、静的ポインタ変数および定数を、リンク時に既知のアドレスを持たないオブジェクトのアドレスに初期化することはできません。書込み可能な静的変数でこれを解決するために、コンパイラはプログラムの起動時に初期化を実行するコードを生成します（C++ での動的初期化と同じように）。

関連項目

285 ページの *--ropi*、286 ページの *--rwpi*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［コード］>［動的リード / ライト / 初期化なし］

--no_scheduling

構文

```
--no_scheduling
```

説明

このオプションは、命令スケジューラを無効にするときに使用します。

注 : このオプションは、最適化レベルが高の場合にのみ有効です。

関連項目

227 ページの *命令スケジューリング*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［命令スケジューリング］

--no_size_constraints

構文

```
--no_size_constraints
```

説明

このオプションを使用して、高速度の最適化の時にコードサイズの拡張に対する通常の制限を緩和します。

注 : このオプションは、`-Ohs` とともに使用したときだけ効果があります。

関連項目

224 ページの *速度とサイズ*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［サイズの制約なし］

--no_static_destruction

構文

```
--no_static_destruction
```

説明

通常は、コンパイラはコードを出力して、プログラム終了時に破壊が必要な C++ 静的変数を破壊します。こうした破壊が不要なこともあります。

このオプションを使用して、こうしたコードの出力を無効にします。

関連項目

100 ページの *atexit 制限の設定*。

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］>［C/C++ コンパイラ］>［追加オプション］を選択します。

--no_system_include

構文

```
--no_system_include
```

説明

デフォルトでは、コンパイ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` コンパイ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。

関連項目

262 ページの *--dlib_config*、288 ページの *--system_include_dir*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［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］

--no_tbaa

構文

```
--no_tbaa
```

説明

このオプションは、型ベースエイリアス解析を無効にする場合に使用します。

注 : このオプションは、最適化レベルが［高］の場合にのみ有効です。

関連項目

225 ページの*型ベースエイリアス解析*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［型ベースエイリアス解析］

--no_typedefs_in_diagnostics

構文

```
--no_typedefs_in_diagnostics
```

説明

このオプションは、診断で typedef 名の使用を無効化する場合に使用します。通常は、コンパイラからのメッセージ（ほとんどの場合は何らかの診断メッセージ）で型についての記述がある場合、元の宣言で使用されていた typedef 名の方のテキストが短くなるときは typedef 名で記述されます。

例

```
typedef int (*MyPtr)(char const *);
MyPtr p = "My text string";
```

この場合、以下のようなエラーメッセージが表示されます。

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

`--no_typedefs_in_diagnostics` オプションを使用した場合は、エラーメッセージは以下のようになります。

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

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］>［C/C++ コンパイラ］>［追加オプション］**を選択します。

--no_unaligned_access

構文

```
--no_unaligned_access
```

説明

このオプションは、コンパイラでアラインメントされないアクセスを回避するときに使用します。データアクセスは、通常、パフォーマンス上の理由で

アラインメントされて実行されます。ただし、アクセスによっては、特にパックデータ構造体に対する読込みまたは書込みの場合、アラインメントされない場合があります。このオプションを使用すると、このようなすべてのアクセスは、アラインメントされないアクセスを拒否するため、小さいデータサイズを使用して実行されます。このオプションは、ARMv6 アーキテクチャ以降で役に立ちます。

関連項目

325 ページの *アラインメント*。

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］> ［C/C++ コンパイラ］> ［追加オプション］**を選択します。

--no_unroll

構文

```
--no_unroll
```

説明

このオプションは、ループ展開を無効にする場合に使用します。

注 : このオプションは、最適化レベルが［高］の場合にのみ有効です。

関連項目

224 ページの *ループ展開*。

［プロジェクト］> ［オプション］> ［C/C++ コンパイラ］> ［最適化］> ［使用可能な変換］> ［ループ展開］

--no_warnings

構文

```
--no_warnings
```

説明

デフォルトでは、コンパイラは警告メッセージを出力します。このオプションは、すべてのワーニングを無効に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。

--no_wrap_diagnostics

構文

```
--no_wrap_diagnostics
```

説明

デフォルトでは、診断メッセージ中の長い行は、読みやすくするため複数行に分割されます。このオプションは、診断メッセージのラインラッピングを無効に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[n|l|m|h|hs|hz]
```

パラメータ

`n`	**なし*（デバッグサポートに最適）**
`l`（デフォルト）	**低***
`m`	**中**
`h`	**高（バランス）**
`hs`	**高（速度優先）**
`hz`	**高（サイズ優先）**

*「なし」と「低」の最も重要な違いは、「なし」ではすべての非静的変数がその変数のスコープ内全体で有効になることです。

説明

このオプションは、コンパイラでコードを最適する際に使用される最適化レベルを設定する場合に使用します。最適化オプションを指定していない場合は、デフォルトで最適化レベルが使用されます。`-O` のみを使用し、パラメータを指定しない場合は、高（バランス）の最適化レベルが使用されます。

最適化レベルを低くすると、デバッガでプログラムのフローを追跡するのが比較的容易になります。逆に、最適化レベルを高くすると、追跡が比較的難しくなります。

関連項目

221 ページの *コンパイラ最適化の設定*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］

--only_stdout

構文

```
--only_stdout
```

説明

コンパイラで、通常はエラー出力ストリーム (stderr) に転送されるメッセージにも標準出力ストリーム (stdout) を使用する場合に、このオプションを使用します。

このオプションは、IDE では使用できません。

--output, -o

構文

```
--output {filename|directory}
-o {filename|directory}
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

デフォルトでは、コンパイラで生成されたオブジェクトコード出力は、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出力用に別の出力ファイル名を明示的に指定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。

--predef_macros

構文

```
--predef_macros {filename|directory}
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、定義済シンボルを一覧表示するときに使用します。このオプションを使用する場合には、プロジェクトの他のファイルと同一のオプションを使用する必要もあります。

filename を指定した場合は、コンパイラは出力をそのファイルに保存します。ディレクトリを指定した場合、コンパイラは出力をそのディレクトリ内のファイル（拡張子 predef）に保存します。

このオブジェクトでは、コマンドラインでソースファイルを指定する必要がある点に注意してください。


このオプションは、IDE では使用できません。

--preinclude

構文

```
--preinclude includefile
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、コンパイラでソースファイルのリードを開始する前に、指定のインクルードファイルをリードする場合に使用します。これは、アプリケーション全体のソースコードで変更を行う場合（新しいシンボルを定義する場合など）に便利です。


［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［プリプロセッサ］>［プリインクルードファイル］

--preprocess

構文

```
--preprocess[=[c][n][l]] {filename|directory}
```

パラメータ

c	コメントの保持
n	プリプロセスのみ
l	#line ディレクティブを生成

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、プリプロセッサ出力を指定ファイルに生成する場合に使用します。


［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［プリプロセッサ］>［ファイルへのプリプロセッサ出力］

--public_equ

構文

```
--public_equ symbol[=value]
```

パラメータ

symbol	定義するアセンブラシンボルの名前
value	定義したアセンブラシンボルの値（任意指定）

説明

このオプションは、EQU ディレクティブを使用してアセンブラ言語でラベルを定義し、PUBLIC ディレクティブを使用してエクスポートする操作と等価です。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。

このオプションは、IDE では使用できません。

--relaxed_fp

構文

```
--relaxed_fp
```

説明

このオプションを使用して、コンパイラで言語規則を緩和させ、浮動小数点式の最適化をより積極的に実行します。このオプションは、以下の条件を満たす浮動小数点式のパフォーマンスを向上させます。

- 式に単精度および倍精度の値が両方含まれている
- 倍精度の値が精度を失わずに単精度に変換できる
- 式の結果は単精度に変換されます

倍制度の代わりに単精度で計算を実行すると、精度が失われることがあります。

例

```
float F(float a, float b)
{
  return a + b * 3.0;
}
```

C 標準では、この例における 3.0 の型が double であるため、式全体が倍精度で評価される必要があります。ただし、--relaxed_fp オプションを使用する場合、3.0 は float に変換され、式全体が float 精度で評価できるようになります。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 2］>［浮動小数点数動作］

--remarks

構文

```
--remarks
```

説明

最も軽度の診断メッセージを、リマークと呼びます。リマークは、ソースコード中で、生成したコードで異常な動作の原因となる可能性がある部分を示します。デフォルトでは、コンパイラはリマークを生成しません。このオプションは、コンパイラでリマークを生成する場合に使用します。

関連項目

242 ページの*重要度*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［リマークの有効化］

--require_prototypes

構文

```
--require_prototypes
```

説明

このオプションは、すべての関数が正しいプロトタイプを持つかどうかをコンパイラで強制的に検証する場合に使用します。このオプションを使用すると、以下のいずれかが含まれるコードではエラーが発生します。

- 宣言のない関数、またはカーニハン & リッチー C 形式で宣言された関数の呼出し
- 先にプロトタイプが宣言されていない public 関数の関数定義
- プロトタイプを含まない型の関数ポインタによる間接的な関数呼出し

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［プロトタイプの強制］

--ropi

構文

```
--ropi
```

説明

このオプションを使用して、アドレスコードおよびリードオンリーのデータへの PC 関連の参照を使用するコードをコンパイラで生成します。

このオプションを使用する場合、以下の制限が適用されます。

- C++ の構文は使用できません
- オブジェクト属性 `__ramfunc` は使用できません

- 別の定数、文字列リテラル、関数のアドレスを使用してポインタ定数は初期化できません。ただし、書込み可能な変数は実行時に定数アドレスに初期化することができます。

関連項目

277 ページの *--no_rw_dynamic_init*、420 ページの *定義済プリプロセッサシンボル*の詳細。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［コード］>［コードおよびリードオンリのデータ (ropi)］

--rwpi

構文

```
--rwpi
```

説明

このオプションを使用して、静的ベースレジスタ (R9) からアドレス書込みが可能なデータへのオフセットを使用するコードをコンパイラで生成します。

このオプションを使用する場合、以下の制限が適用されます。

- オブジェクト属性 `__ramfunc` は使用できません
- ポインタ定数は、書込み可能な変数のアドレスでは初期化できません。ただし、書込み可能な静的変数は、実行時に書込み可能なアドレスに初期化できます。

関連項目

277 ページの *--no_rw_dynamic_init*、420 ページの *定義済プリプロセッサシンボル*の詳細。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［コード］>［リード / ライトデータ (rwpi)］

--section

構文

```
--section OldName=NewName
```

説明

コンパイラは、関数およびデータオブジェクトを、IAR ILINK リンカが参照する指定セクションに配置します。このオプションは、セクションの名前を *OldName* から *NewName* に変更するときに使用します。

異なるアドレス範囲にコードやデータを配置し、`@` 表記または `#pragma location` ディレクティブでは不十分な場合に、このオプションが有益です。セクション名の変更時には、対応するリンカ設定ファイルも変更する必要があることに、注意してください。

例

セクション MyText に関数を配置するには、以下のコマンドを使用します。

```
--section .text=MyText
```

関連項目

217 ページの *データと関数のメモリ配置制御*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［出力］>［コードセクション名］

--separate_cluster_for_initialized_variables

構文

```
--separate_cluster_for_initialized_variables
```

説明

このオプションは、変数クラスタ使用時、初期化変数と非初期化変数を分離する場合に使用します。このオプションを使用すると、データの初期化に必要なバイト数を ROM エリアから削減できる可能性がありますが、コードが大きくなる可能性もあります。

このオプションは、ユーザ独自のデータ初期化ルーチンを使用する場合に便利ですが、ゼロ初期化変数の作成に IAR ツールが必要となります。

関連項目

101 ページの *手動で初期化する*、448 ページの *initialize ディレクティブ*。

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］>［C/C++ コンパイラ］>［追加オプション］を選択します。

--silent

構文

```
--silent
```

説明

デフォルトでは、コンパイラは開始メッセージや最終的な統計レポートを出力します。このオプションは、コンパイラがこれらのメッセージを標準出力ストリーム（通常は画面）に送信しないようにする場合に使用します。

このオプションは、エラー / ワーニングメッセージの表示には影響しません。

このオプションは、IDE では使用できません。

--strict

構文

```
--strict
```

説明

デフォルトでは、コンパイラで標準の C/C++ に緩く対応したスーパセットを使用できます。このオプションを使用して、アプリケーションのソースコードが厳密な標準の C/C++ に準拠するよう徹底します。

注 : `-e` オプションと `--strict` オプションは、同時に使用できません。

関連項目

177 ページの *言語拡張の有効化*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［言語の適合］>［厳密］

--system_include_dir

構文

```
--system_include_dir path
```

パラメータ

path	システムインクルードファイルのパス、247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

デフォルトでは、コンパイ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。

関連項目

262 ページの *--dlib_config*、278 ページの *--no_system_include*。

このオプションは、IDE では使用できません。

--thumb

構文

```
--thumb
```

説明

このオプションは、デフォルトの関数モードを Thumb に設定するときに使用します。この設定は、相互作用していない限り、プログラムに含まれるすべてのファイルで同じでなければなりません。

注 : このオプションの効果は、`--cpu_mode=thumb` オプションと同じです。

関連項目

269 ページの *--interwork*、346 ページの *__interwork*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［コード］>［プロセッサのモード］>［Thumb］

--use_c++_inline

構文

```
--use_c++_inline
```

説明

標準の C では、`inline` キーワードに対して C++ の場合とはわずかに異なる動作が使用されます。C を使用する際に C++ の動作が必要な場合は、このオプションを使用してください。

関連項目

74 ページの*インライン関数*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C の派生言語］>［C99］>［C++ インライン動作］

--use_unix_directory_separators

構文

```
--use_unix_directory_separators
```

説明

このオプションを使用して、DWARF デバッグ情報で / を（¥ の代わりに）ファイルパスのディレクトリ区切り文字として使用します。

このオプションは、UNIX 形式のディレクトリ区切り文字を必要とするデバッガを使用する場合に便利です。

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］>［C/C++ コンパイラ］>［追加オプション］を選択します。

--vectorize

構文

```
--vectorize
```

説明

このオプションを使用して、NEON ベクタ命令の生成を有効にします。

ターゲットプロセッサに NEON 機能があり、最適化レベルが `-Ohs` の場合にのみループがベクトル化されます。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［ベクトル化］

--vla

構文

```
--vla
```

説明

このオプションを使用して、C99 の可変長配列のサポートを有効にします。こうした配列は、ヒープに配置されます。このオプションには標準の C が必要であり、`--c89` コンパイラオプションとは併用できません。

注 : `--vla` と `longjmp` ライブラリ関数は併用できません。併用するとメモリリークとなることがあります。

関連項目

175 ページの *C 言語の概要*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C の派生言語］>［VLA の許可］

--warn_about_c_style_casts

構文

```
--warn_about_c_style_casts
```

説明

このオプションを使用して、C- スタイルキャストが C++ ソースコードで使用されるときにコンパイラを警告します。

このオプションは、IDE では使用できません。

--warnings_affect_exit_code

構文

```
--warnings_affect_exit_code
```

説明

デフォルトでは、ゼロ以外の終了コードが生成されるのはエラーが発生した場合のみであるため、ワーニングは終了コードには影響しません。このオプションを使用すると、ワーニングが発生した場合もゼロ以外の終了コードが生成されます。

このオプションは、IDE では使用できません。

--warnings_are_errors

構文

```
--warnings_are_errors
```

説明

このオプションは、コンパイラでワーニングをエラーとして処理する場合に使用します。コンパイラがエラーを検出した場合、オブジェクトコードは生成されません。リマークに変更されたワーニングは、エラーとして処理されません。

注 : オプション `--diag_warning` または `#pragma diag_warning` ディレクティブにより警告として再分類された診断メッセージも、`--warnings_are_errors` 使用時はエラーとして処理されます。

関連項目

261 ページの *--diag_warning*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［すべてのワーニングをエラーとして処理］

リンカオプション

- リンカオプションの概要
- リンカオプションの説明

一般的な構文規則については、245 ページの*オプションの構文*を参照してください。

リンカオプションの概要

以下の表は、リンカオプションの概要を示します。

コマンドラインオプション	説明
`--advanced_heap`	アドバンストヒープを使用
`--basic_heap`	ベーシックヒープを使用
`--BE8`	ビッグエンディアンフォーマット BE8 を使用します
`--BE32`	ビッグエンディアンフォーマット BE32 を使用します
`--bounds_table_size`	グローバル境界テーブルのサイズを指定します。*ARM 用 C-SPY® デバッグガイド*で C-RUN のドキュメントを参照してください
`--call_graph`	コールグラフファイルを XML フォーマットで生成
`--config`	リンカによって使用されるリンカ設定ファイルを指定します
`--config_def`	設定ファイルのシンボルを定義します
`--config_search`	リンカ設定ファイルの検索に使用する追加のディレクトリを指定します
`--cpp_init_routine`	ユーザ定義 C++ 動的初期化ルーチンを指定します
`--cpu`	プロセッサ選択を指定します
`--debug_heap`	ヒープチェックを有効にします。*ARM 用 C-SPY® デバッグガイド*で C-RUN のドキュメントを参照してください

表 31: リンカオプションの概要

コマンドラインオプション	説明
--define_symbol	アプリケーションが使用できるシンボルを定義します
--dependencies	ファイル依存関係をリスト化
--diag_error	メッセージタグをエラーとして処理
--diag_remark	メッセージタグをリマークとして処理
--diag_suppress	診断メッセージを無効にします
--diag_warning	メッセージタグをワーニングとして処理
--diagnostics_tables	すべての診断メッセージをリスト化
--enable_hardware_workaround	指定したハードウェアの回避方法を有効化
--enable_stack_usage	スタックの使用量解析を有効化
--entry	シンボルをルートシンボルおよびアプリケーションの開始として処理します
--error_limit	リンクを停止するエラー数の上限を指定
--exception_tables	C コードの例外テーブルを生成します
--export_builtin_config	デフォルト設定の icf ファイルを生成します
--extra_init	定義されていれば呼出される追加の初期化ルーチンを指定します
-f	コマンドラインを拡張
--force_exceptions	例外ランタイムコードを常にインクルードします
--force_output	エラーが発生した場合でも出力ファイルを生成します
--ignore_uninstrumented_pointers	C-RUN が有効でないメモリからのポインタのチェックを無効化します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください
--image_input	イメージファイルをセクションに配置します
--inline	小さいルーチンをインライン化します
--keep	シンボルをアプリケーションに強制的に追加します
--log	選択したトピックのログ出力を有効にします
--log_file	ログをファイルに記録します
--mangled_names_in_messages	マングル化名をメッセージに追加します
--map	マップファイルを生成します

表 31: リンカオプションの概要 （続き）

コマンドラインオプション	説明
--merge_duplicate_sections	同等のリードオンリーのセクションをマージ
--misrac	MISRA-C:1998 固有のエラーメッセージを有効にします。このオプションは --misrac1998 と同義で、旧バージョンとの互換性のためだけに使用できます
--misrac1998	MISRA-C:1998 固有のエラーメッセージを有効にします。『IAR Embedded Workbench® MISRA C:1998 リファレンスガイド』を参照してください
--misrac2004	MISRA-C:2004 固有のエラーメッセージを有効にします。『IAR Embedded Workbench® MISRA C:2004 リファレンスガイド』を参照してください
--misrac_verbose	MISRA-C チェックの冗長なロギングを有効にします。『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』および『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』を参照してください
--no_dynamic_rtti_elimination	不要な場合でも動的ランタイム型の情報をインクルードします
--no_exceptions	例外が使用された場合にエラーを出力します
--no_fragments	セクションフラグメント処理を無効化
--no_library_search	自動ランタイムライブラリ検索を無効にします
--no_literal_pool	データの読取りが許可されておらず、コードの実行のみが可能なメモリ領域から実行されるコードを生成します
--no_locals	ローカルシンボルを ELF 実行可能イメージから削除します
--no_range_reservations	絶対シンボルの範囲予約を無効化
--no_remove	未使用のセクションの削除を無効にします
--no_veneers	ベニアの生成を無効にします
--no_vfe	仮想関数の除去を無効化
--no_warnings	ワーニングの生成を無効にします
--no_wrap_diagnostics	診断メッセージの長い行をラップしません

表 31: リンカオプションの概要 （続き）

コマンドラインオプション	説明
-o	オブジェクトファイル名を設定。--output のエイリアス
--only_stdout	標準出力のみを使用
--output	オブジェクトファイル名を設定
--pi_veneers	位置独立コードのベニアを生成します
--place_holder	他のツールによってフィルされる ROM の部分を予約するときに使用します（icheckサムなど）
--redirect	シンボルへの参照を別のシンボルにリダイレクトします
--remarks	リマークを有効化
--search	オブジェクトとライブラリファイルを検索するディレクトリを追加して指定します
--semihosting	デバッグインタフェースでリンクします
--silent	サイレント処理を設定
--skip_dynamic_initialization	システム起動時に動的初期化を無効にします
--stack_usage_control	スタック使用解析制御ファイルを指定
--strip	デバッグ情報を実行可能イメージから削除します
--threaded_lib	スレッドで使用するためにランタイムライブラリを設定します
--vfe	仮想関数の除去を制御
--warnings_affect_exit_code	ワーニングが終了コードに影響
--warnings_are_errors	ワーニングをエラーとして処理
--whole_archive	アーカイブにあるすべてのオブジェクトファイルを、コマンドライン上で指定したときと同じように扱います

表 31: リンカオプションの概要 （続き）

リンカオプションの説明

次のセクションでは、それぞれのリンカオプションについて詳細に説明します。

［追加オプション］ページを使用して特定のコマンドラインオプションを指定する場合、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。

--advanced_heap

構文 `--advanced_heap`

説明 このオプションにより、アドバンストヒープを使用します。

関連項目 199 ページの *アドバンストヒープとベーシックヒープ*。

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] > [リンカ] > [追加オプション]** を使用します。

--basic_heap

構文 `--basic_heap`

説明 このオプションにより、ベーシックヒープを使用します。

関連項目 199 ページの *アドバンストヒープとベーシックヒープ*。

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] > [リンカ] > [追加オプション]** を使用します。

--BE8

構文 `--BE8`

説明 このオプションは、Byte Invariant Addressing モードを指定するときに使用します。

つまり、リンカは命令のバイトオーダを反転させ、リトルエンディアンコードおよびビッグエンディアンデータが生成されます。これは、ARMv6 ビッグエンディアンイメージでのデフォルトのバイトアドレッシングモードです。これは、ARM v6M および ARM v7 のビッグエンディアンイメージで使用可能な唯一のモードです。

Byte Invariant Addressing モードは、ARMv6、ARM v6M、ARM v7 をサポートする ARM プロセッサのみで使用できます。

関連項目 57 ページの *バイトオーダ*、326 ページの *バイトオーダ*、298 ページの *--BE32*、265 ページの *--endian*。

[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]

--BE32

構文

```
--BE32
```

説明

このオプションは、旧式のビッグエンディアンモードを指定するときに使用します。

このオプションは、ビッグエンディアンコードおよびデータを生成します。これは、ARMv6 以前のすべてのビッグエンディアンイメージでの唯一のバイトアドレッシングモードです。このモードは、ARM v6 のビッグエンディアンにも使用できますが、ARM v6M や ARM v7 では使用できません。

関連項目

57 ページの*バイトオーダ*、326 ページの*バイトオーダ*、297 ページの *--BE8*、265 ページの *--endian*。

[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]

--call_graph

構文

```
--call_graph {filename|directory}
```

パラメータ

247 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、コールグラフファイルを生成するときに使用します。拡張子を指定しない場合は、cgx が使用されます。このオプションは、1 つのコマンドラインで 1 回だけ使用できます。

このオプションを使用すると、リンカでスタック使用量解析が有効になります。

関連項目

87 ページの*スタック使用量解析*。

[プロジェクト] > [オプション] > [リンカ] > [アドバンスト] > [コールグラフ出力 (XML)]

--config

構文

```
--config filename
```

パラメータ

247 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、リンカにより使用される設定ファイルを指定するときに使用します（デフォルトのファイル名拡張子は `icf` です）。設定ファイルが指定されていない場合、デフォルトの設定が使用されます。このオプションは、1 つのコマンドラインで 1 回だけ使用できます。

関連項目

*リンカ設定ファイル*の章。

[プロジェクト] > [オプション] > [リンカ] > [設定] > [リンカ設定ファイル]

--config_def

構文

```
--config_def symbol=constant_value
```

パラメータ

symbol	設定ファイルで使用されるシンボルの名前。
constant_value	設定シンボルの定数値。

説明

このオプションは、設定ファイルで使用される定数設定シンボルを定義するときに使用します。このオプションの効果は、リンカ設定ファイルの `define symbol` ディレクティブと同じです。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。

関連項目

301 ページの *--define_symbol*、105 ページの *ILINK とアプリケーション間の相互処理*。

[プロジェクト] > [オプション] > [リンカ] > [設定] > [設定ファイルに定義されたシンボル]

--config_search

構文

```
--config_search path
```

パラメータ

path	リンカがリンカ設定インクルードファイルを検索するディレクトリのパス。

説明

このオプションを使用して、リンカ設定ファイルで `include` ディレクティブを処理する際に、ファイル検索で使用する追加のディレクトリを指定します。

デフォルトでは、リンカはシステム設定ディレクトリ内の設定インクルードファイルのみを検索します。複数の検索ディレクトリを指定するには、各パスについてこのオプションを使用します。

関連項目

464 ページの *include ディレクティブ*。

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] > [リンカ] > [追加オプション]** を使用します。

--cpp_init_routine

構文

```
--cpp_init_routine routine
```

パラメータ

routine	ユーザ定義 C++ 動的初期化ルーチン。

説明

IAR C/C++ コンパイラおよび標準ライブラリを使用する場合、C++ 動的初期化は自動的に処理されます。これ以外の場合、このオプションの使用が必要になることがあります。

セクション型が `INIT_ARRAY` または `PREINIT_ARRAY` のセクションがアプリケーションに含まれている場合、C++ 動的初期化ルーチンは必須です。デフォルトでは、このルーチンの名前は `__iar_cstart_call_ctors` で、標準ライブラリの起動コードで呼出されます。このオプションは、標準ライブラリを使用していないときに、別ルーチンでこれらのセクション型を処理する必要がある場合に使用します。

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] > [リンカ] > [追加オプション]** を使用します。

--cpu

構文

```
--cpu=core
```

パラメータ

core	特定のプロセッサ選択を指定します。

説明

このオプションを使用して、アプリケーションをリンクする派生プロセッサを選択します。デフォルトは、オブジェクトファイル属性と互換性のあるプロセッサまたはアーキテクチャを使用することです。

関連項目

255 ページの *--cpu*。

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

--define_symbol

構文

```
--define_symbol symbol=constant_value
```

パラメータ

symbol	アプリケーションで使用できる定数シンボルの名前。
constant_value	シンボルの定数値。

説明

このオプションは、アプリケーションで使用できる定数シンボル、すなわちラベルを定義するときに使用します。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。このオプションは、`define symbol` ディレクティブと異なる点に注意してください。

関連項目

299 ページの *--config_def*、105 ページの *ILINK とアプリケーション間の相互処理*。

[プロジェクト] > [オプション] > [リンカ] > [#define] > [シンボル定義]

--dependencies

構文

```
--dependencies[=[i|m]] {filename|directory}
```

パラメータ

`i`（デフォルト）	ファイルの名前のみをリスト化。
`m`	makefile スタイルでリスト化。

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、ファイル入力のために開かれたリンカ設定ファイル、オブジェクトファイル、ライブラリファイルのファイル名を、デフォルトのファイル名拡張子 `i` を持つファイルに含める場合に使用します。

例

--dependencies や --dependencies=i を使用すると、開かれている各入力ファイルの名前とフルパス（ある場合）が独立した行に出力されます。次に例を示します。

```
c:¥myproject¥foo.o
d:¥myproject¥bar.o
```

--dependencies=m を使用した場合は、makefile スタイルで出力されます。各入力ファイルについて、makefile の依存関係規則を含む行が生成されます。各行は出力ファイル名、コロン、空白文字、入力ファイル名で構成されます。次に例を示します。

```
a.out: c:¥myproject¥foo.o
a.out: d:¥myproject¥bar.o
```

このオプションは、IDE では使用できません。

--diag_error

構文

```
--diag_error=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージをエラーとして再分類する場合に使用します。エラーは、実行可能イメージが生成されなくなるような重要度の問題を示します。終了コードはゼロ以外になります。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。

[プロジェクト] > [オプション] > [リンカ] > [診断] > [エラーとして処理]

--diag_remark

構文

```
--diag_remark=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）

説明

このオプションは、特定の診断メッセージをリマークとして再分類する場合に使用します。リマークは、最も軽微な診断メッセージです。実行可能イ

メージでの異常な動作の原因となる可能性がある構造が存在することを示します。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。

注 : デフォルトでは、リマークは表示されません。リマークを表示するには、`--remarks` オプションを使用します。

[プロジェクト] > [オプション] > [リンカ] > [診断] > [リマークとして処理]

--diag_suppress

構文

```
--diag_suppress=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe117` など）

説明

このオプションは、特定の診断メッセージを無効にする場合に使用します。これらのメッセージは表示されなくなります。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。

[プロジェクト] > [オプション] > [リンカ] > [診断] > [診断を無効化]

--diag_warning

構文

```
--diag_warning=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe826` など）

説明

このオプションは、特定の診断メッセージをワーニングとして再分類する場合に使用します。ワーニングは、問題はあるが、リンク処理の終了前にリンカが終了する原因にはならないエラーや脱落を示します。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。

[プロジェクト] > [オプション] > [リンカ] > [診断] > [ワーニングとして処理]

--diagnostics_tables

構文

```
--diagnostics_tables {filename|directory}
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、すべての診断メッセージを指定ファイルに保存する場合に使用します。これは、プラグマディレクティブを使用して診断メッセージの重要度を無効化または変更したが、その理由を記述し忘れた場合などに便利です。

このオプションは、他のオプションと併用できません。

このオプションは、IDE では使用できません。

--enable_hardware_workaround

構文

```
--enable_hardware_workaround=waid[waid[...]]
```

パラメータ

waid	有効化する回避方法の ID 番号。使用可能な回避方法のリストは、インフォメーションセンタからリリースノートを参照してください。

説明

このオプションは、特定のハードウェア問題の対策をリンカで生成する場合に使用します。

関連項目

使用可能なパラメータの一覧については、リンカのリリースノートを参照してください。

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] > [リンカ] > [追加オプション]** を使用します。

--enable_stack_usage

構文

```
--enable_stack_usage
```

説明

このオプションは、スタック使用量解析を有効にするときに使用します。リンカマップファイルが生成される場合、スタック使用量の章がマップファイルに含まれます。

注 : --stack_usage_control か --call_graph オプションのどちらか最低 1 つを使用する場合、スタック使用量解析は自動的に有効になります。

関連項目 87 ページの*スタック使用量解析*。

[プロジェクト] > [オプション] > [リンカ] > [アドバンスト] > [スタックの使用量解析を有効にする]

--entry

構文

```
--entry symbol
```

パラメータ

symbol	ルートシンボルおよび開始ラベルとして処理されるシンボルの名前

説明

このオプションは、ルートシンボルおよびアプリケーションの開始ラベルとして処理されるシンボルを作成するときに使用します。これは、ローダを使用する場合に便利です。このオプションを使用しない場合、デフォルトの開始シンボルは __iar_program_start です。ルートシンボルは、そのモジュールが含まれる場合、アプリケーション内で参照されるかどうかに関わらず保持されます。オブジェクトファイルのモジュールは常に含まれますが、ライブラリのモジュールパートは必要な場合にのみ含まれます。

注 : 参照先のラベルは、アプリケーション内で使用可能でなければなりません。また、リセットベクタが必ず新しい開始ラベルを参照するようにしてください（たとえば、--redirect __iar_program_start=_myStartLabel など）。

[プロジェクト] > [オプション] > [リンカ] > [ライブラリ] > [デフォルトプログラムエントリのオーバライド]

--error_limit

構文

```
--error_limit=n
```

パラメータ

n	リンカがリンクを中止するエラー発生回数（*n* は正の整数）。0 は制限なしを示します。

説明

`--error_limit` オプションは、リンカがリンクを停止する前のエラー数を指定するために使用します。デフォルトでは、エラー数の上限は 100 です。

このオプションは、IDE では使用できません。

--exception_tables

構文

```
--exception_tables={nocreate|unwind|cantunwind}
```

パラメータ

`nocreate`（デフォルト）	エントリは生成されません。最少のメモリが使用されますが、関数を通して例外情報なしに例外が伝播されると、結果は定義されません。
`unwind`	例外情報がなくても関数を通して例外を正しく伝播する、巻き戻しエントリが生成されます。
`cantunwind`	巻き戻しのないエントリを生成し、関数を通じて例外を伝播しようとすると `terminate` が呼出されるようにします。

説明

このオプションを使用して、正しい呼出しフレーム情報を持っているが例外情報を持たない関数をリンカでどう処理するかを決定します。

コンパイラは、C 関数が正しい呼出しフレーム情報を取得するよう徹底します。アセンブラ言語で記述された関数の場合、アセンブラディレクティブを使用して呼出しフレーム情報を生成する必要があります。

関連項目

185 ページの *C++ の使用*。

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] > [リンカ] > [追加オプション]** を使用します。

--export_builtin_config

構文

```
--export_builtin_config filename
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

デフォルトで使用される設定をファイルにエクスポートします。

このオプションは、IDE では使用できません。

--extra_init

構文

```
--extra_init routine
```

パラメータ

routine	ユーザ定義の初期化ルーチン。

説明

このオプションは、リンカにより初期化テーブルの最後にある指定のルーチンにエントリを追加するときに使用します。このルーチンはシステム起動時、初期化ルーチンが呼出された後、main が呼出される前に呼出されます。このルーチンは、レジスタ R0 で引き渡された値を保持する必要があります。ルーチンが定義されていなければ、エントリは追加されません。

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] > [リンカ] > [追加オプション]** を使用します。

-f

構文

```
-f filename
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、リンカで、指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。

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] > [リンカ] > [追加オプション]** を使用します。

--force_exceptions

構文

```
--force_exceptions
```

説明

このオプションを使用して、リンカのヒューリスティックで例外を使用しないことが指示されていても、リンカが例外テーブルと例外コードをインクルードするようにします。

インクルードされるコードに rethrow ではない throw 式がある場合、リンカは使用する例外を考慮します。コードの残り部分にそのような throw 式がない場合、リンカは operator new、dynamic_cast、typeid を使用して、失敗時に例外をスローする代わりに abort を呼出します。コードに他のスローが含まれておらず、これらのコンストラクトからの例外を検出しなければならない場合、このオプションを使用しなければならないことがあります。

関連項目

185 ページの *C++ の使用*。

［プロジェクト］>［オプション］>［リンカ］>［最適化］>［C++ 例外］>［許可］>［常にインクルード］

--force_output

構文

```
--force_output
```

説明

このオプションは、リンクエラーに関係なく出力実行可能イメージを生成するときに使用します。

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］>［リンカ］>［追加オプション］**を使用します。

--image_input

構文

```
--image_input filename [,symbol,[section[,alignment]]]
```

パラメータ

filename	リンクする raw イメージを含むピュアバイナリファイル。
symbol	バイナリデータを参照できるシンボル。
section	バイナリデータを配置するセクション。デフォルトは .text です。
alignment	セクションのアラインメント。デフォルトは 1 です。

説明

このオプションは、通常の入力ファイルの他に、ピュアバイナリファイルをリンクします。ファイルの全体の内容がセクションに配置されます。つまり、ピュアバイナリデータのみを含むことができます。

注 : オブジェクトファイルからのセクションの場合のみ、--image_input オプションを使用して作成されたセクションは実際に必要でない限り含まれません。オプションでシンボルを指定してアプリケーションでこのシンボルを

参照（または --keep オプションを使用）するか、あるいはセクション名を指定してリンカ設定ファイルで keep ディレクティブを使用し、セクションが含まれていることを確認します。

例

```
--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4
```

ビュアバイナリファイル bootstrap.abs の内容は、セクション CSTARTUPCODE に配置されます。ファイルの内容が配置されるセクションは 4 バイト境界にアラインメントされ、アプリケーションがシンボル Bootstrap を参照している場合（またはコマンドラインオプション --keep）にのみ含まれます。

関連項目

309 ページの *--keep*。

[プロジェクト] > [オプション] > [リンカ] > [入力] > [ロウバイナリイメージ]

--inline

構文

```
--inline
```

説明

一部のルーチンは小さいため、ルーチンを呼出す命令のスペースに収まります。このオプションを使用して、可能な場合にはリンカがルーチンの呼出しをルーチン本体と置き換えるようにします。

[プロジェクト] > [オプション] > [リンカ] > [最適化] > [小さいルーチンのインライン化]

--keep

構文

```
--keep symbol
```

パラメータ

symbol	ルートシンボルとして処理されるシンボルの名前

説明

一般的にリンカでは、アプリケーションで必要なシンボルのみを保存します。このオプションは、シンボルを常に最終アプリケーションに含めるときに使用します。

[プロジェクト] > [オプション] > [リンカ] > [入力] > [シンボルをキープ]

--log

構文

```
--log topic[,topic,...]
```

パラメータ

ここで、*topic* は以下のいずれかです。

`call_graph`	スタック使用量解析で表示されたコールグラフをリストします。
`initialization`	各バッチに選択されたコピーバッチおよび圧縮をリストします。
`libraries`	自動ライブラリセレクタによって行われるすべての決定をリストします。これには、必要なその他のシンボル (`--keep`)、リダイレクト (`--redirect`)、どのランタイムライブラリが選択されたかが含まれることがあります。
`modules`	アプリケーションにインクルードするよう選択された各モジュールと、インクルードの要因となったシンボルをリストします。
`redirects`	リダイレクトされたシンボルをリストします。
`sections`	アプリケーションにインクルードするよう選択された各モジュールとセクションのフラグメント、インクルードの要因となった依存関係をリストします。
`unused_fragments`	アプリケーションにインクルードされなかったセクションフラグメントをリストします。
`veeners`	ベニアの作成および使用状況の統計をリストします。

説明

このオプションは、リンカログ情報を `stdout` に出力するときに使用します。ログ情報は、実行可能なイメージが現在の状態になった原因を把握するために利用できる場合があります。

関連項目

311 ページの *--log_file*。

[プロジェクト] > [オプション] > [リンカ] > [リンカ] > [ログの生成]

--log_file

構文

```
--log_file filename
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、ログを指定ファイルに出力するときに使用します。

関連項目

310 ページの *--log*。

［プロジェクト］>［オプション］>［リンカ］>［リンカ］>［ログの生成］

--mangled_names_in_messages

構文

```
--mangled_names_in_messages
```

説明

このオプションは、メッセージの C/C++ シンボルにマングル化した名前とデマングル化した名前の両方を生成するときに使用します。マングル化とは、複雑な C 名や C++ 名を簡単な名前にマッピングするときに使用するテクニックです（オーバロードなどに利用）。たとえば、`void h(int, char)` は、`_Z1hic` になります。

このオプションは、IDE では使用できません。

--map

構文

```
--map {filename|directory}
```

説明

このオプションは、リンカメモリマップファイルを生成するときに使用します。マップファイルのデフォルトのファイル名拡張子は `map` です。このマップファイルの内容は、以下のとおりです。

- リンカのバージョン、現在の日時、使用されたコマンドラインをリストするマップファイルヘッダのリンクの概要。
- ランタイム属性をリストするランタイム属性の概要。
- 配置ディレクティブでソートしアドレス順序で各セクション/ブロックをリストした配置の概要。
- データ範囲、パッキング方手法、圧縮率をリストする初期化テーブルのレイアウト。

- 各モジュールからイメージへのメモリ使用率を、ディレクトリおよびライブラリでソートしてリストするモジュールの概要。
- すべてのパブリックシンボルおよび一部のローカルシンボルをアルファベット順に表示し、そのシンボルを含むモジュールを一覧表示したエントリリスト。
- それらのバイトの一部が*「共有」*として報告されることもあります。

  共有オブジェクトとは、モジュール間で共有される関数またはデータオブジェクトのことです。このような共有が 2 つ以上のモジュールで発生した場合、1 つの関数 / データオブジェクトのみが保持されます。たとえば、インライン関数がインライン化されない場合があります。これは、これらの関数が共有とマークされていて、各関数の 1 つのインスタンスしか最終的なアプリケーションにインクルードされないためです。この仕組みは、特定の関数や変数に直接的には関連しないコンパイラ生成コードやデータで、最終的なアプリケーションには 1 つのインスタンスしか必要とされない場合にも使用されることがあります。

このオプションは、1 つのコマンドラインで 1 回だけ使用できます。

[プロジェクト] > [オプション] > [リンカ] > [リスト] > [リンカマップファイルの表示]

--merge_duplicate_sections

構文

```
--merge_duplicate_sections
```

説明

このオプションを使用して、リードオンリーのセクションのコピーを 1 つだけ保持します。これによって異なる関数や定数が同じアドレスを持つことがあるため、このオプションを有効にすると、異なるアドレスに依存するアプリケーションが正しく機能しなくなるため注意してください。

[プロジェクト] > [オプション] > [リンカ] > [最適化] > [重複セクションのマージ]

--no_dynamic_rtti_elimination

構文

```
--no_dynamic_rtti_elimination
```

説明

このオプションを使用して、リンカのヒューリスティックで動的（ポリモーフィック）ランタイム型情報 (RTTI) データが必要ないと指定されている場合でも、リンカがアプリケーションイメージにその情報をインクルードするようにします。

リンカは、インクルードされたコードにポリモーフィック型の typeid や dynamic_cast 式がある場合に、動的型情報を必要と判断します。デフォルトでは、こうした式が検出されない場合は、動的 RTTI リクエストを機能させるために RTTI データがインクルードされることはありません。

注 : *多形でない* 型の typeid 式は、特定の RTTI オブジェクトが直接参照されることになり、不要になりかねない一切のオブジェクトをリンカがインクルードしなくなります。

関連項目

185 ページの *C++ の使用*。

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] > [リンカ] > [追加オプション]** を使用します。

--no_exceptions

構文

```
--no_exceptions
```

説明

このオプションを使用して、インクルードされたコードにスローがある場合にリンカがエラーを出力するようにします。このオプションは、アプリケーションで例外を使用しないようにする場合に役立ちます。

関連項目

185 ページの *C++ の使用*。

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] > [リンカ] > [最適化] > [C++ 例外を許可]

--no_fragments

構文

```
--no_fragments
```

説明

このオプションはセクションフラグメント処理を無効にします。通常、ツールセットは、セクションフラグメント情報をリンカに転送するために IAR 独自の情報を使用します。リンカは、この情報を使用して、未使用のコードおよびデータを削除し、実行可能イメージのサイズをさらに最小化します。このオプションでは、セクションのフラグメントの除去を無効にして、各セクション全体をインクルードまたは除外します。通常、アプリケーションのサイズが大きくなります。

関連項目

99 ページの *シンボルおよびセクションの保持*。

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] > [リンカ] > [追加オプション]** を使用します。

--no_library_search

構文

```
--no_library_search
```

説明

このオプションは、自動ランタイムライブラリ検索を無効にするときに使用します。このオプションは、正しい標準ライブラリの自動追加を無効にします。これは、たとえば、ユーザが構築した標準ライブラリをアプリケーションで必要な場合などに役に立ちます。

このオプションは、自動ライブラリ選択のすべての手順を無効にする点に注意してください。標準ライブラリを使用する場合は、一部の手順を複製しなければならないことがあります。どの手順を複製するか判断するには、`--log libraries` リンカオプションと自動ライブラリ選択を有効にしてともに使用します。

[プロジェクト] > [オプション] > [リンカ] > [ライブラリ] > [自動ランタイムライブラリ選択]

--no_literal_pool

構文

```
--no_literal_pool
```

説明

このオプションは、データの読取りが許可されておらず、コードの実行のみが可能なメモリ領域から実行されるコードに使用します。

このオプションを使用すると、リンカは `MOV32` 擬似命令をモードを変更するベニアで使用し、データバスを使用して目的地のアドレスをロードしないようにします。またこのオプションは、これを用いてコンパイルされたライブラリが使用されることを意味します。

オプション `--no_literal_pool` は、ARMv7-M コアでのみ使用できます。

関連項目

275 ページの *--no_literal_pool*。

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] > [リンカ] > [追加オプション]** を使用します。

--no_locals

構文

```
--no_locals
```

説明

このオプションは、ローカルシンボルを ELF 実行可能イメージから削除するときに使用します。

注 : このオプションは、実行可能イメージの DWARF 情報からはローカルシンボルを削除しません。

［プロジェクト］>［オプション］>［リンカ］>［出力］

--no_range_reservations

構文

```
--no_range_reservations
```

説明

通常は、リンカは絶対シンボルが使用するすべての範囲をサイズゼロとして予約し、`place in` コマンドの対象から除外します。

このオプションを使用する場合、これらの予約は無効になり、リンカは絶対シンボルの範囲と重複する形でセクションを自由に配置することができます。

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］>［リンカ］>［追加オプション］を使用します。

--no_remove

構文

```
--no_remove
```

説明

このオプションが使用されている場合、未使用のセクションは削除されません。つまり、実行可能イメージに含まれる各モジュールには、その元のセクションがすべて含まれます。

関連項目

99 ページの *シンボルおよびセクションの保持*。

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］>［リンカ］>［追加オプション］を使用します。

--no_veneers

構文

```
--no_veneers
```

説明

このオプションは、実行可能イメージで必要とする場合でもベニアの挿入を無効にするときに使用します。ベニアの挿入を無効にすると、ベニアを必要とする各参照に対して再配置エラーが発生します。

関連項目

106 ページのベニア。

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] > [リンカ] > [追加オプション]** を使用します。

--no_vfe

構文

```
--no_vfe
```

説明

このオプションは、仮想関数除去の最適化を無効化するときに使用します。少なくとも 1 つのインスタンスを持つ全クラスのすべての仮想関数が保持され、ランタイム型情報データはすべてのポリモーフィッククラスで保持されます。また、VFE 情報を持たないモジュールについてワーニングメッセージは出力されません。

関連項目

322 ページの *--vfe*、207 ページの *仮想関数の除去*。

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] > [リンカ] > [最適化] > [C++ 仮想関数除去を実行]

--no_warnings

構文

```
--no_warnings
```

説明

デフォルトでは、リンカは警告メッセージを出力します。このオプションは、すべてのワーニングを無効に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。

--no_wrap_diagnostics

構文

```
--no_wrap_diagnostics
```

説明

デフォルトでは、診断メッセージ中の長い行は、読みやすくするため複数行に分割されます。このオプションは、診断メッセージのラインラッピングを無効に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。

--only_stdout

構文

```
--only_stdout
```

説明

リンカで、通常はエラー出力ストリーム (stderr) に転送されるメッセージにも標準出力ストリーム (stdout) を使用する場合に、このオプションを使用します。

このオプションは、IDE では使用できません。

--output, -o

構文

```
--output {filename|directory}
-o {filename|directory}
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

デフォルトでは、リンカで生成されたオブジェクト実行可能イメージは、a.out という名前のファイルに配置されます。このオプションは、別の出力ファイル名（デフォルトのファイル名拡張子は out）を明示的に指定する場合に使用します。

[プロジェクト] > [オプション] > [リンカ] > [出力] > [出力ファイル]

--pi_veneers

構文

```
--pi_veneers
```

説明

このオプションは、リンカで位置独立コードのベニアを生成する場合に使用します。このタイプのベニアは、通常のベニアよりも大きく、低速である点に注意してください。

関連項目

106 ページのベニア。

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] > [リンカ] > [追加オプション]** を使用します。

--place_holder

構文

```
--place_holder symbol[,size[,section[,alignment]]]
```

パラメータ

symbol	作成するシンボルの名前
size	ROM のサイズ、デフォルトは 4 バイト
section	使用するセクション名。デフォルトは `.text`
alignment	セクションのアラインメント。デフォルトは `1`

説明

このオプションは、たとえば、`ielftool` により計算されるチェックサムなど、他のツールによってフィルされる ROM の部分を予約するときに使用します。このリンカオプションを使用するたびに、指定した名前、サイズ、アラインメントのセクションが生成されます。シンボルは、そのセクションを参照するためにアプリケーションで使用できます。

注 : 他のセクションと同様、`--place_holder` オプションにより作成されるセクションは、そのセクションが必要だとみなされた場合のみアプリケーションに含まれます。`--keep` リンカオプション、または `keep` リンカディレクティブを使用すると、このようなセクションを強制的に含めることができます。

関連項目

479 ページの *IAR ユーティリティ*。

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] > [リンカ] > [追加オプション]** を使用します。

--redirect

構文

```
--redirect from_symbol=to_symbol
```

パラメータ

from_symbol	変更前のシンボルの名前
to_symbol	変更後のシンボルの名前

説明

このオプションは、シンボルの参照を変更するときに使用します。

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] > [リンカ] > [追加オプション]** を使用します。

--remarks

構文

```
--remarks
```

説明

最も軽度の診断メッセージを、リマークと呼びます。リマークは、ソースコード中で、生成したコードで異常な動作の原因となる可能性がある部分を示します。デフォルトでは、リンカはリマークを生成しません。このオプションは、リンカでリマークを生成する場合に使用します。

関連項目

242 ページの *重要度*。

[プロジェクト] > [オプション] > [リンカ] > [診断] > [リマークの有効化]

--search

構文

```
--search path
```

パラメータ

path	リンカがオブジェクトやライブラリファイルを検索するディレクトリのパス。

説明

このオプションを使用して、リンカがオブジェクトやライブラリファイルを検索するディレクトリを追加して指定します。

デフォルトでは、リンカは作業ディレクトリにあるオブジェクトおよびライブラリファイルのみを検索します。コマンドライン上でこのオプションを使用するたびに、検索ディレクトリが 1 つ追加されます。

関連項目　49 ページの *リンク処理*。


このオプションは、IDE では使用できません。

--semihosting

構文　`--semihosting[=iar_breakpoint]`

パラメータ

`iar_breakpoint`	IAR 固有のメカニズムは、SWI/SVC を広範囲に使用するアプリケーションのデバッグ時に使用できます。

説明　このオプションは、デバッグインタフェース（ブレークポイントメカニズム）を出力イメージに含めるときに使用します。パラメータを指定しない場合、SWI/SVC メカニズムが ARM7/9/11 用に含まれ、BKPT メカニズムが Cortex-M 用に含まれます。

関連項目　118 ページの *アプリケーションデバッグサポート*。


[プロジェクト] > [オプション] > [一般オプション] > [ライブラリ構成] > [セミホスティング]

--silent

構文　`--silent`

説明　デフォルトでは、リンカは開始メッセージや最終的な統計レポートを出力します。このオプションは、リンカがこれらのメッセージを標準出力ストリーム（通常は画面）に送信しないようにする場合に使用します。

このオプションは、エラー / ワーニングメッセージの表示には影響しません。


このオプションは、IDE では使用できません。

--skip_dynamic_initialization

構文　`--skip_dynamic_initialization`

説明　IAR C/C++ コンパイラおよび標準ライブラリを使用する場合、C++ 動的初期化は自動的に処理されます。

このオプションは、システム起動時に実行する動的初期化を無効にするときに使用します。これは通常、初期化の前に RTOS のヒープ管理を設定する場合などに役立ちます。

この場合、アプリケーションのソースコードに、ライブラリ関数`__iar_dynamic_initialization` の呼出しを追加する必要があります。この関数を呼出するときに、初期化が行われます。

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] > [リンカ] > [追加オプション]** を使用します。

--stack_usage_control

構文

```
--stack_usage_control filename
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、スタックの使用量制御ファイルを指定するときに使用します。このファイルは、スタック使用量解析を制御したり、モジュールや関数についてより詳細なスタック使用情報を提供します。このオプションを何度も使用して、複数のスタック使用量制御ファイルを指定できます。拡張子を指定しない場合は、`suc` が使用されます。

このオプションを使用すると、リンカでスタック使用量解析が有効になります。

関連項目

87 ページの *スタック使用量解析*。

[プロジェクト] > [オプション] > [リンカ] > [アドバンスト] > [制御ファイル]

--strip

構文

```
--strip
```

説明

デフォルトでは、リンカは、入力オブジェクトファイルのデバッグ情報を出力実行可能イメージに保持します。このオプションは、この情報を削除するときに使用します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [出力] > [出力ファイルにデバッグ情報を含める]

--threaded_lib

構文

```
--threaded_lib
```

説明

このオプションを使用して、スレッドで使用するランタイムライブラリを自動的に設定します。

［プロジェクト］>［オプション］>［一般オプション］>［ライブラリ設定］>［ライブラリでスレッドのサポートを有効にする］

--vfe

構文

```
--vfe=[forced]
```

パラメータ

`forced`	1 つまたは複数のモジュールで必要な仮想関数除去の情報が不足している場合でも、仮想関数除去を実行します。

説明

このオプションを使用して、1 つまたは複数のモジュールで必要な仮想関数除去の情報が不足している場合でも、仮想関数除去を実行します。パラメータ `forced` がないと、このオプションは効果がありません。

仮想関数除去を強制的に使用すると、必要な情報を持たないモジュールが仮想関数の呼出しを実行したり、動的ランタイム型情報を使用する場合に、安全でなくなる可能性があります。

関連項目

316 ページの *--no_vfe*、207 ページの *仮想関数の除去*。

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］>［リンカ］>［最適化］>［C++ 仮想関数除去を実行］

--warnings_affect_exit_code

構文

```
--warnings_affect_exit_code
```

説明

デフォルトでは、ゼロ以外の終了コードが生成されるのはエラーが発生した場合のみであるため、ワーニングは終了コードには影響しません。このオプションを使用すると、ワーニングが発生した場合もゼロ以外の終了コードが生成されます。

このオプションは、IDE では使用できません。

--warnings_are_errors

構文

```
--warnings_are_errors
```

説明

このオプションは、リンカでワーニングをエラーとして処理する場合に使用します。リンカがエラーを検出した場合は、実行可能イメージは生成されません。リマークに変更されたワーニングは、エラーとして処理されません。

注 : オプション `--diag_warning` オプションによりワーニングとして再分類された診断メッセージも、`--warnings_are_errors` 使用時はエラーとして処理されます。

関連項目

261 ページの *--diag_warning*、303 ページの *--diag_warning*。

［プロジェクト］>［オプション］>［リンカ］>［診断］>［すべてのワーニングをエラーとして処理］

--whole_archive

構文

```
--whole_archive filename
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションを使用して、アーカイブにあるすべてのオブジェクトファイルを、コマンドライン上で指定したときと同じように扱います。これは、常にオブジェクトファイル（ファイル名拡張子 o）から含まれるルートコンテンツがアーカイブに含まれているものの、モジュールから何らかのエントリが参照されている場合にのみアーカイブからインクルードされるときに役立ちます。

例

`archive.a` にオブジェクトファイル `file1.o`、`file2.o`、`file3.o` が含まれる場合、`--whole_archive archive.a` を使用すると、`file1.o file2.o file3.o` と指定したときと同じになります。

関連項目

99 ページの *モジュールの保持*。

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］>［リンカ］>［追加オプション］を使用します。

データ表現

- アラインメント
- バイトオーダ
- 基本データ型整数型
- 基本データ型浮動小数点数型
- ポインタ型
- 構造体型
- 型修飾子
- C++ のデータ型

特定のアプリケーションで最も効率的なコードを作成するためのデータ型やポインタについては、*組込みアプリケーション用の効率的なコーディング*章を参照してください。

アラインメント

すべての C データオブジェクトには、オブジェクトをメモリ内に記憶する方法を指定するためのアラインメントが設定されています。たとえば、オブジェクトのアラインメントが 4 の場合は、このオブジェクトは 4 で割り切れるアドレスに格納する必要があります。

一部のプロセッサではメモリのアクセス方法に制限があるため、アラインメントのコンセプトが採用されています。

4 で割り切れるアドレスにメモリ読取りが設定された場合にのみ、プロセッサが 1 回の命令で 4 バイトのメモリを読み取ることができるとします。この場合、`long` 型の整数など、4 バイトオブジェクトのアラインメントは 4 になります。

また、一度に 2 バイトしか読み取ることができないプロセッサの場合には、4 バイトの `long` 型整数のアラインメントは 2 になります。

構造体のアラインメントは、アラインメントが最も厳密な構造体メンバと同じです。構造体およびそのメンバのアラインメント要件を下げるには、`#pragma pack` または `__packed` データ型属性を使用します。

すべてのデータ型は、それらのアラインメントの倍数のサイズにする必要があります。これ以外については、アラインメントの最初のエレメントのみ配列の要件に従って配置されることが保証されます。つまり、コンパイラが構造体の最後にパッドバイトを追加しなければならないことがあります。パッドバイトについては、336 ページの*パック構造体型*を参照してください。

`#pragma data_alignment` ディレクティブを使用すると、特定の変数のアラインメント要件を上げることができます。

ARM コア のアラインメント

データオブジェクトのアラインメントは、データオブジェクトがメモリでどのように格納されるかを制御します。アラインメントを使用する理由は、4 バイトオブジェクトが 4 で割り切れるアドレスに格納されている場合、ARM コアがより効率的に 4 バイトオブジェクトにアクセスできるためです。

アラインメント 4 のオブジェクトは、4 で割り切れるアドレスに格納する必要があり、アラインメント 2 のオブジェクトは、2 で割り切れるアドレスに格納する必要があります。

コンパイラでは、すべてのデータ型のアラインメントを割り当てることで、このようなデータの配置を保証し、ARM コアが確実にデータを読み取ることができるようにしています。

関連情報については、253 ページの *--align_sp_on_irq*、273 ページの *--no_const_align* を参照してください。

バイトオーダ

注 : ビッグエンディアンモードには、BE8 および BE32 という 2 つの種類があり、これらはリンク時に指定します。BE8 の場合、データはビッグエンディアン、コードはリトルエンディアンになります。BE32 では、データとコードの両方がビッグエンディアンコードになります。v6 以前のアーキテクチャでは BE32 エンディアンモードが使用され、v6 以降は BE8 モードが使用されます。v6 (ARM11) アーキテクチャでは、両方のビッグエンディアンがサポートされます。

基本データ型整数型

コンパイラは、標準の C の基本データ型のすべてと追加のデータ型の両方をサポートします。

整数型概要

以下の表に、各整数型のサイズと範囲の一覧を示します。

データ型	サイズ	範囲	アラインメント
bool	8 ビット	0 ～ 1	1
char	8 ビット	0 ～ 255	1
signed char	8 ビット	-128 ～ 127	1
unsigned char	8 ビット	0 ～ 255	1
signed short	16 ビット	-32768 ～ 32767	2
unsigned short	16 ビット	0 ～ 65535	2
signed int	32 ビット	-2^{31} ～ $2^{31}-1$	4
unsigned int	32 ビット	0 ～ $2^{32}-1$	4
signed long	32 ビット	-2^{31} ～ $2^{31}-1$	4
unsigned long	32 ビット	0 ～ $2^{32}-1$	4
signed long long	64 ビット	-2^{63} ～ $2^{63}-1$	8
unsigned long long	64 ビット	0 ～ $2^{64}-1$	8

表 32: 整数型

符号付変数は、2 の補数フォーマットで表現されます。

BOOL 型

bool データ型は、C++ 言語のデフォルトでサポートされています。言語拡張を有効化し、stdbool.h ファイルをインクルードした場合は、bool 型を C ソースコードでも使用できます。この場合は、ブール値の false および true も使用可能になります。

ENUM 型

コンパイラでは、enum 定数の保持に必要な最小の型を使用し、unsigned よりも signed を優先します。

IAR システムズの言語拡張が有効化されている場合や、C++ においては、enum 手数および型を long、unsigned long、long long、unsigned long long 型にすることも可能です。

コンパイラで自動的に使用される型より大きな型を使用するには、enum 定数を十分に大きな値で定義します。次に例を示します。

```
/* enum での char 型の使用を無効化 */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

関連情報については、265 ページの *--enum_is_int* を参照してください。

CHAR 型

char 型は、コンパイラのデフォルトでは符号なしですが、--char_is_signed コンパイラオプションを使用して符号付にすることが可能です。ただし、ライブラリは char 型は符号なしでコンパイルされています。

WCHAR_T 型

wchar_t 型は、サポートされるロケール間で設定された最大の拡張文字集合のすべてのメンバに対して個別のコードを表現できる値の範囲を持つ整数型です。

wchar_t 型は、C++ 言語のデフォルトでサポートされています。wchar_t 型を C ソースコードでも使用するには、ランタイムライブラリから stddef.h ファイルをインクルードする必要があります。

ビットフィールド

標準の C では、int、signed int と unsigned int を整数ビットフィールドの基本型として使用できます。標準の C++ および C では、コンパイラで言語拡張が有効になっている場合、任意の整数または列挙型を基本型として使用できます。単純な整数型（char、short、int など）が符号つきまたは符号なしのビットフィールドになるかどうかは、実装によって定義されます。

ARM 用 IAR C/C++ コンパイラでは、単純な整数型は符号なしとして処理されます。

式のビットフィールドは、int がビットフィールドのすべての値を表せる場合、int として扱われます。それ以外の場合は、ビットフィールドの基本型として処理されます。

各ビットフィールドは、ビットフィールドを格納するために使用可能なビットを十分に持つ基本型の次のコンテナに配置されます。それぞれのコンテナの中では、バイトオーダを考慮して、最初に使用可能なバイト内にビットフィールドが配置されます。

また、異なる型のビットフィールドコンテナが重複できない場合、コンパイラは代替のビットフィールド割当て方式（分割された型）に対応します。こ

の割当て方式を使用すると、各ビットフィールドは型が前のビットフィールドのものと異なったり、前のビットフィールドと同じコンテナにビットフィールドが合わない場合、新しいコンテナに配置されます。各コンテナ内では、ビットフィールドは最下位のビットから最上位ビット（分割された型）へ、あるいは最上位ビットから最下位ビット（逆順の分割された型）へと配置されます。この割当て方式では、デフォルトの割当て方式（連結された型）より使用スペースが少なくなることは決してなく、ビットフィールドの型が混在する場合は格段に多くのスペースを使用することがあります。

`#pragma bitfield` ディレクティブを使用して、ビットフィールドの割当て方式を選択してください（360 ページの *bitfields* を参照）。

次の例を考えてみます。

```
struct BitfieldExample
{
  uint32_t a : 12;
  uint16_t b : 3;
  uint16_t c : 7;
  uint8_t  d;
};
```

連結された型のビットフィールド割当て方式の例

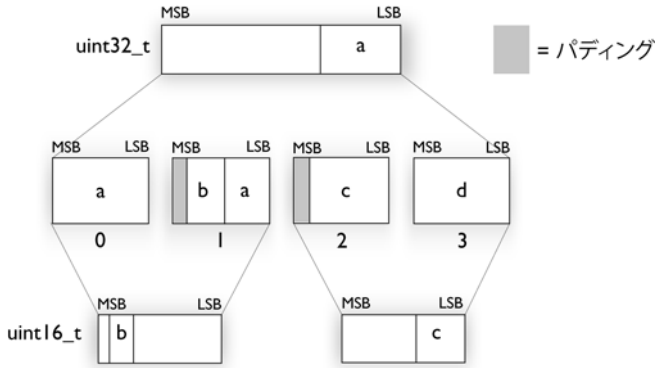
最初のビットフィールド `a` を配置するために、コンパイラはオフセット `0` に 32 ビットを割当て、`a` をコンテナの最初のバイトと 2 番目のバイトに入れます。

2 番目のビットフィールド `b` については、16 ビットのコンテナが必要です。オフセット `0` に 4 つの空いているビットがまだあるため、ビットフィールドはそこに配置されます。

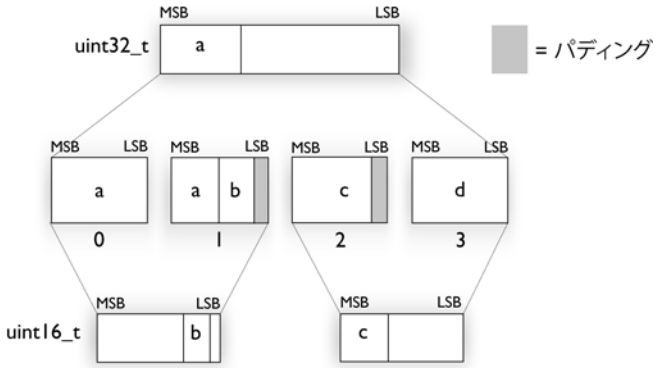
3 番目のビットフィールド `c` は、最初の 16 ビットコンテナに 1 ビットしか残っていないため、オフセット `2` に新しいコンテナが割り当てられ、`c` はこのコンテナの最初のバイトに入れられます。

4 番目のメンバである `d` は、次に使用可能なフルバイト、つまりオフセット `3` のバイトに配置できます。

リトルエンディアンモードでは、各ビットフィールドは左から右の順にバイトに配置されるように、コンテナの最下位の空いているビットから割り当てられます。



ビッグエンディアンモードでは、各ビットフィールドは左から右の順にバイトに配置されるように、コンテナの最上位の空いているビットから割り当てられます。



分割された型のビットフィールド割当て方式の例

最初のビットフィールド a を配置するために、コンパイラはオフセット 0 に 32 ビットを割当て、a を最下位の 12 ビットのコンテナに入れます。

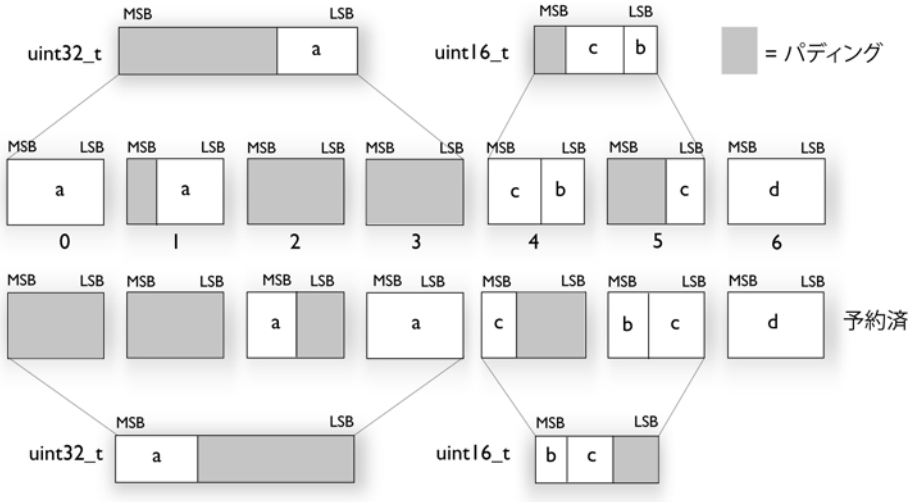
2 番目のビットフィールド b を配置するために、新しいコンテナがオフセット 4 に割当てられます。これは、ビットフィールドの型が前のビットフィールドと同じではないためです。b は、このコンテナの最下位の 3 ビットに配置されます。

3 番目のビットフィールド c は b と同じ型なので、同じコンテナに入ります。

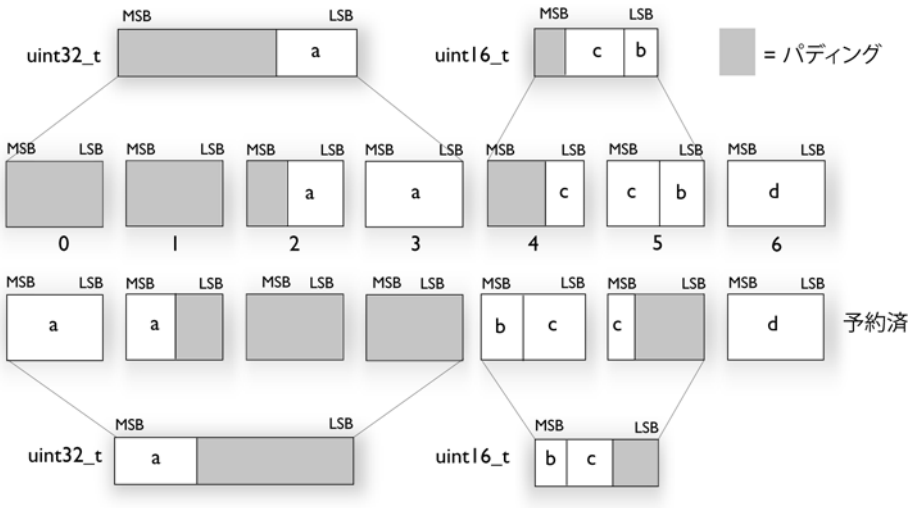
4 番目のメンバ d は、オフセット 6 のバイトに割り当てられます。d は、b や c と同じコンテナには配置できません。その理由は、これがビットフィールドではなく、同じ型でもないために、合わないからです。

逆順（逆順の分割された型）を使用する際は、各ビットフィールドはそのコンテナの最上位ビットから配置されます。

これは、リトルエンディアンモードの `bitfield_example` のレイアウトです。



これは、ビッグエンディアンモードの `bitfield_example` のレイアウトです。



基本データ型浮動小数点数型

ARM 用 IAR C/C++ コンパイラでは、浮動小数点の値を標準 IEEE 754 フォーマットで表現します。各浮動小数点数型のサイズを以下に示します。

タイプ	サイズ	範囲 (+/-)	10 進数	指数部	仮数部	アラインメント
`float`	32 ビット	±1.18E-38 ～ ±3.40E+38	7	8 ビット	23 ビット	4
`double`	64 ビット	±2.23E-308 ～ ±1.79E+308	15	11 ビット	52 ビット	8
`long double`	64 ビット	±2.23E-308 ～ ±1.79E+308	15	11 ビット	52 ビット	8

表 33: 浮動小数点数型

Cortex-M0 および Cortex-M1 の場合、コンパイラは非正規化数をサポートしません。非正規化数を生成する演算では、非正規化数の代わりにすべてゼロが生成されます。その他のコアにおける非正規化数の表現については、333 ページの *特殊な浮動小数点数の表現*を参照してください。

浮動小数点環境

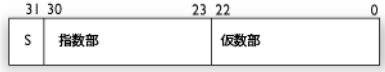
浮動小数点値の例外フラグは、VFP ユニットを持つデバイスでサポートされており、これらは `fenv.h` ファイルで定義されます。VFP ユニットを持たな

いデフォルトバイスの場合、fenv.h ファイルで定義された関数は存在しますが、これらに機能はありません。

feraiseexcept 関数は、FE_OVERFLOW や FE_UNDERFLOW を使用して呼出された場合、inexact 浮動小数点例外を引き起こしません。

32 ビット浮動小数点数フォーマット

32 ビット浮動小数点数を整数として表現すると、以下のようになります。



指数部は 8 ビット、仮数部は 23 ビットです。

値は以下のようになります。

$(-1)^S * 2^{(指数部 - 127)} * 1.$ 仮数部

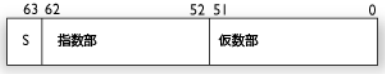
範囲は少なくとも以下のようになります。

±1.18E-38 から 3.39E+38

浮動小数点数の演算子 (+、-、*、/) は、約 7 桁です。

64 ビット浮動小数点数フォーマット

64 ビット浮動小数点数を整数として表現すると、以下のようになります。



指数部は 11 ビット、仮数部は 52 ビットです。

値は以下のようになります。

$(-1)^S * 2^{(指数部 - 1023)} * 1.$ 仮数部

範囲は少なくとも以下のようになります。

±2.23E-308 ～ ±1.79E+308

浮動小数点数の演算子 (+、-、*、/) は、約 15 桁です。

特殊な浮動小数点数の表現

特殊な浮動小数点数の表現を以下に挙げます。

- ゼロは、仮数部や指数部でゼロとして表現されます。符号ビットは、正または負のゼロを示します。

- 無限大は、指数部を最大値に、仮数部をゼロに設定することで表現されます。符号ビットは、正または負の無限大を示します。
- 非数 (NaN) は、指数部を正の最大値に設定し、仮数部の最上位ビットを 1 に設定して表現されます。符号ビットの値は無視されます。
- 非正規化数は、正規化数で表せる数値よりも小さい値を表現するときに使用します。この場合、値が小さくなるほど精度が低下するという欠点があります。指数部は 0 に設定され、数値が非正規化数であることを示します。ただし、数値は指数部が 1 である場合と同様に扱われます。正規化数とは異なり、非正規化数では仮数部の最上位ビット (MSB) に暗黙の 1 が設定されていません。非正規化数の値は次のようになります。

  $(-1)^S * 2^{(1-BIAS)} * 0.$仮数部

  ここで、BIAS は、32 ビットおよび 64 ビット浮動小数点値の場合、それぞれ 127 および 1023 です。

ポインタ型

コンパイラには、関数ポインタとデータポインタという 2 種類の基本ポインタ型があります。

関数ポインタ

関数ポインタのサイズは常に 32 ビットで、範囲は 0x0-0xFFFFFFFF です。

関数ポインタ型が宣言されると、属性が * 記号の前に挿入されます。次に例を示します。

```
typedef void (__thumb __interwork * IntHandler) (void);
```

これは、#pragma ディレクティブを使用して再書き込みすることができます。

```
#pragma type_attribute=__thumb __interwork
typedef void IntHandler_function(void);
typedef IntHandler_function *IntHandler;
```

データポインタ

使用できるデータポインタは 1 つだけです。このサイズは 32 ビットで、範囲は 0x0-0xFFFFFFFF です。

キャスト

ポインタ間のキャストには以下の特徴があります。

- 整数型の*値*からそれよりも小さな型のポインタへのキャストは、切捨てにより実行されます

- ポインタ型からそれよりも小さな整数型へのキャストは、切捨てにより実行されます
- *ポインタ型*からそれよりも大きな整数型へのキャストは、ゼロ拡張により実行されます
- *データポインタ*と関数ポインタ間のキャストは不正です
- *関数ポインタ*を整数型にキャストすると、結果は不定になります
- 符号なし整数型の値からそれよりも大きな型のポインタへのキャストは、ゼロ拡張により実行されます

size_t

`size_t` は `sizeof` 演算子の結果の符号なし整数型です。ARM 用 IAR C/C++ コンパイラでは、`size_t` で使用する型は `unsigned int` です。

ptrdiff_t

`ptrdiff_t` は 2 つのポインタを差し引いた結果の符号付きの整数型です。ARM 用 IAR C/C++ コンパイラでは、`ptrdiff_t` に使用する型は `size_t` 型の符号付きの整数型です。

intptr_t

`intptr_t` は、`void *` を保持するのに十分大きな符号付整数型です。ARM 用 IAR C/C++ コンパイラでは、`intptr_t` で使用する型は `signed long int` です。

uintptr_t

`uintptr_t` は、符号なしであることを除き、`intptr_t` と同じです。

構造体型

`struct` のメンバは、宣言された順に連続して格納されます。最初のメンバが最下位のメモリアドレスを持ちます。

構造体型のアライメント

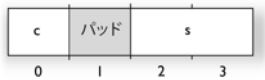
`struct` 型や `union` 型のアラインメントは、最高のアラインメント要件の数と同じになります。このアライメン要件は構造になるメンバにも適用されます。アラインメントされた構造体オブジェクトの配列が可能になるようにするには、`struct` のサイズをアライメントの偶数の倍数に調整します。

一般的なレイアウト

struct のメンバは、常に宣言で指定された順に割り当てられます。各メンバは、指定したアラインメント（オフセット）に従って struct 内に配置されます。

```
struct First
{
  char c;
  short s;
} s;
```

以下の図に、メモリでのレイアウトを示します。



構造体のアラインメントは 2 バイトです。また、short s に正しいアラインメントを与えるためにパッドバイトが挿入されている必要がありあります。

パック構造体型

構造のメンバのアライメント要件を和するために使用されます。これにより、構造体のレイアウトが変更されます。メンバは、宣言時と同じ順序で配置されますが、メンバ間のパッドエリアが少なくなることがあります。

正しくアラインメントされていないオブジェクトにアクセスする場合には、コードのサイズが大きくなり速度が低下します。そのような構造体へのアクセスが多数ある場合、パックされていない struct に正しい値を構成し、この struct にアクセスする方が通常は適しています。

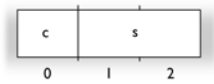
アラインメントが正しく設定されていないメンバへのポインタ作成および使用には、特別な注意も必要です。パックされた struct のアラインメントが正しく設定されていないメンバに直接アクセスする場合、コンパイラは、必要に応じて正しいコード（ただし、サイズが大きく低速）を出力します。しかし、アラインメントが正しく設定されていないメンバへのポインタを使用してそのメンバにアクセスする場合には、通常のコード（サイズが小さく高速）が使用されます。一般的なケースでは、これは機能しません。なぜなら、通常のコードは正しいアラインメントに依存することがあるからです。

以下の例では、パックされた構造体を宣言します。

```
#pragma pack(1)
struct S
{
  char c;
  short s;
};

#pragma pack()
```

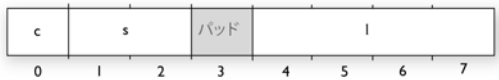
構造 S にはこのメモリレイアウトがあります。



次の例では、パックされていない別の構造体 S2 を宣言します。この構造体には、前の例で宣言した構造体 s が含まれます。

```
struct S2
{
  struct S s;
  long l;
};
```

構造 S2 にはこのメモリレイアウトがあります



構造体 S は、前の例で宣言したメモリレイアウト、サイズ、アラインメントを使用します。メンバ l のアラインメントは 4 です。これは、構造体 S2 のアラインメントが 4 になることを意味します。

詳細については、214 ページの *構造体エレメントのアラインメント*を参照してください。

型修飾子

C 規格では、volatile と const は型修飾子です。

オブジェクトの VOLATILE 宣言

オブジェクト volatile を宣言することによって、オブジェクトの値がコンパイラの制限以上に変化する可能性があることがコンパイラに伝えられます。またコンパイラは、あらゆるアクセスに副作用があると想定する必要があり

ます。よって、volatile オブジェクトへのすべてのアクセスは保持されなければなりません。

オブジェクトを volatile として宣言する主な理由は、以下の 3 つです。

- 共有アクセス : マルチタスク環境で、オブジェクトを複数のタスクで共有する場合
- トリガアクセス : メモリマップされた特殊機能レジスタ (SFR) のように、アクセス発生により影響が生じる場合
- 変更アクセス : コンパイラが認識できない方法で、オブジェクトの内容が変更される可能性がある場合

volatile オブジェクトへのアクセスの定義

C 規格では、抽象マシンが定義されています。これは、volatile 宣言したオブジェクトへのアクセスの動作を制御します。一般的に、抽象マシンに従うコンパイラの動作は、以下のとおりです。

- volatile として宣言されたオブジェクトへの各リード / ライトアクセスをアクセスと見なします
- アクセスは、オブジェクト単位になります。複合オブジェクト（配列、構造体、クラス、共用体など）へのアクセスの場合は、エレメント単位になります。次に例を示します

```
char volatile a;
a = 5;   /* ライトアクセス */
a += 6;  /* 最初はリードアクセスで次にライトアクセス */
```

- ビットフィールドへのアクセスは、その根底型へのアクセスとして処理されます
- const 修飾子を volatile オブジェクトに追加すると、オブジェクトへのライトアクセスが不可能になります。ただし、オブジェクトは C 規格で指定されたとおりに RAM 内に配置されます

ただし、これらの規則は大まかなもので、ハードウェア関連の要件には対応していません。ARM 用 IAR C/C++ コンパイラに固有の規則について、以下に説明します。

アクセス規則

ARM 用 IAR C/C++ コンパイラでは、volatile で宣言したオブジェクトは、以下の規則に従います。

- すべてのアクセスが実行されます
- すべてのアクセスは最後まで実行されます。すなわち、オブジェクト全体がアクセスされます

- すべてのアクセスは、抽象マシンでの場合と同一の順序で実行されます
- すべてのアクセスはアトミックアクセスになります。すなわち、割込みはできません

コンパイラは、8 ビット、16 ビット、32 ビットのすべてのスカラ型へのメモリアクセスに関して、これらの規則に準拠しています。ただし、パック構造体型のアラインメントされていない 16 ビットおよび 32 ビットのフィールドへのアクセスは例外です。

記載されていないすべてのオブジェクト型の組合せについては、すべてのアクセスが維持されるという規則だけが適用されます。

オブジェクト VOLATILE および CONST の宣言

`volatile` オブジェクト `const` を宣言する場合、ライト禁止になりますが、C 規格の仕様に従って RAM メモリに格納されます。

代わりにリードオンリーのメモリにオブジェクトを格納して、`const volatile` オブジェクトとしてアクセス可能にするには、以下のように変数を定義します。

```
const volatile int x @ "FLASH";
```

コンパイラは、リード / ライトセクション `FLASH` を生成します。このセクションは ROM に配置して、アプリケーション起動時に変数を手動で初期化するときに使用します。

これ以降は、イニシャライザは他の値とともにいつでも再度フラッシュすることができます。

オブジェクトの CONST 宣言

`const` 型修飾子は、データオブジェクト（直接またはポインタを使用してアクセス）がリードオンリーであることを示します。`const` として宣言したデータへのポインタは、定数と非定数の両方のオブジェクトを参照できます。可能な限り `const` として宣言したポインタを使用することをお勧めします。これにより、コンパイラによる生成コードの最適化が改善され、誤って修正したデータが原因でアプリケーションに障害が発生する危険性が低下します。

`const` として宣言した静的オブジェクトやグローバルオブジェクトは、ROM に配置されます。

C++ では、ランタイムの初期化が必要なオブジェクトは ROM に配置できません。

C++ のデータ型

C++ では、通常の C データ型はすべて、前述の方法で表現されます。ただし、その型で拡張 C++ 機能を使用している場合は、データ表現に関する想定はできなくなります。たとえば、クラスメンバにアクセスするアセンブラコードを記述することはサポートされません。

拡張キーワード

- 拡張キーワードの一般的な構文規則
- 拡張キーワードの一覧
- 拡張キーワードの詳細

拡張キーワードの一般的な構文規則

コンパイラは、ARM コア固有の機能をサポートする関数やデータオブジェクトで使用可能な一連の属性を提供しています。これらの属性には、*型属性*と*オブジェクト属性*の 2 種類があります。

- 型属性は、データオブジェクトや関数の*外部機能*に影響します
- オブジェクト属性は、データオブジェクトや関数の*内部機能*に影響します

キーワードの構文は、型属性であるかオブジェクト属性であるか、適用対象がデータオブジェクトであるか関数であるかによって異なります。

各属性の詳細については、345 ページの*拡張キーワードの詳細*を参照してください。

注 : 拡張キーワードは、コンパイラで言語拡張が有効化されている場合にのみ使用可能です。

IDE では、デフォルトで言語拡張が有効になっています。

言語拡張を有効にするには、`-e` コンパイラオプションを使用します。263 ページの *-e* を参照してください。

型属性

型属性は、関数の呼出し方法、またはデータオブジェクトのアクセス方法を定義します。すなわち、型属性を使用する場合には、関数またはデータオブジェクトの定義時と宣言時の両方で型属性を指定する必要があります。

型属性を明示的に宣言に配置するか、プラグマディレクティブ `#pragma type_attribute` を使用します。

型属性はさらに*メモリ型属性*と*汎用型属性*に分類できます。メモリ型属性はドキュメントのその他の部分の*メモリ属性*としてだけ参照されます。

汎用型属性

利用可能な*関数型属性*（関数の呼び出し方法に影響）:

```
__arm、__fiq、__interwork、__irq、__swi、__task、__thumb
```

利用可能な*データ型属性*:

```
__big_endian、__little_endian__packed
```

間接参照レベルごとに、必要な数の属性を指定できます。

データオブジェクトで使用される型属性の構文

型属性は構文規則の型修飾子 `const` と `volatile` とほぼ同じように使用します。次に例を示します。

```
__little_endian int i;
int __little_endian j;
```

`i` と `j` は両方ともリトルエンディアンのバイトオーダーでアクセスされます。

`const` や `volatile` とは異なり、構造体メンバの宣言の場合を除いて、型属性は派生した型の型指定子の前に使用されるとき、型属性がオブジェクトまたは typedef 自体に適用されます。

型定義を使用することはコードをより明確にできる場合があります。

```
typedef __packed int packed_int;
packed_int *q1;
```

`packed_int` はパックされた整数のための typedef です。変数 `q1` はそのような整数を指し示すことができます。

`#pragma type_attributes` ディレクティブを使用して宣言の型属性を指定することもできます。プラグマディレクティブで指定した型属性は、データオブジェクトまたは制限される typedef に適用されます。

```
#pragma type_attribute=__packed
int * q2;
```

変数 `q2` はパックされました。

関数で使用される型属性の構文

関数の型属性に使用する構文は、データオブジェクトの型属性の構文とはわずかに異なります。関数の場合、以下のように、属性はリターン型の前に置くか、括弧内に置く必要があります。

```
__irq __arm void my_handler(void);
```

または

```
void (__irq __arm my_handler)(void);
```

以下の my_handler の宣言は、前の例と同一の結果になります。

```
#pragma type_attribute=__irq __arm
void my_handler(void);
```

オブジェクト属性

オブジェクト属性は通常、関数やデータオブジェクトの内部機能に影響しますが、関数の呼出し方法やデータのアクセス方法には直接影響しません。したがって、通常はオブジェクトの宣言でオブジェクト属性を指定する必要はありません。

以下のオブジェクト属性を指定できます。

- 変数に使用可能なオブジェクト属性 :

  __absolute、__no_init

- 関数や変数に使用可能なオブジェクト属性 :

  location、@、__root、__weak

- 関数に使用可能なオブジェクト属性 :

  __intrinsic、__nested、__noreturn、__ramfunc、__stackless

特定の関数やデータオブジェクトに対して、必要な数のオブジェクト属性を指定できます。

location および @ の詳細については、217 ページの*データと関数のメモリ配置制御*を参照してください。

オブジェクト属性の構文

オブジェクト属性は、型の前に記述する必要があります。たとえば、起動時に初期化されないメモリに myarray を配置するには、以下のように記述します。

```
__no_init int myarray[10];
```

#pragma object_attribute ディレクティブも使用できます。以下の宣言は、前の例と同一の結果になります。

```
#pragma object_attribute=__no_init
int myarray[10];
```

注 : オブジェクト属性は、typedef キーワードと併用できません。

拡張キーワードの一覧

以下の表に、拡張キーワードの一覧を示します。

拡張キーワード	説明
__absolute	オブジェクトへの参照が絶対アドレスを使用するようにします
__arm	関数を ARM モードで実行します
__big_endian	ビッグエンディアンバイトオーダを使用する変数を宣言します
__fiq	高速割込み関数を宣言します
__interwork	ARM および Thumb モードの両方から呼出し可能な関数を宣言します
__intrinsic	コンパイラの内部使用専用に予約されています
__irq	割込み関数を宣言します
__little_endian	リトルエンディアンバイトオーダを使用する変数を宣言します
__no_alloc、__no_alloc16	実行ファイルで定数を使用可能にします
__no_alloc_str、__no_alloc_str16	実行ファイルで文字列リテラルを使用可能にします
__nested	__irq で宣言した割込み関数をネストできるようにします。つまり、同じ割込みタイプによる割込みを許可します
__no_init	データオブジェクトを不揮発性メモリに配置します
__noreturn	関数がリターンしないことをコンパイラに通知します
__packed	データ型アラインメントを 1 に減らします
__pcrel	--ropi コンパイラオプションの使用時に、コンパイラで内部的に使用されます
__ramfunc	関数を RAM モードで実行します
__root	関数や変数を、未使用の場合でもオブジェクトに含めます
__sbrel	--rwpi コンパイラオプションの使用時に、コンパイラで内部的に使用されます
__stackless	機能するスタックなしに関数を呼出し可能にします
__swi	ソフトウェア割込み関数を宣言します
__task	レジスタ保護の規則を緩和します

表 34: 拡張キーワードの一覧

拡張キーワード	説明
__thumb	関数を Thumb モードで実行します
__weak	外部的に弱いリンクになるようにシンボルを宣言します

表 34: 拡張キーワードの一覧（続き）

拡張キーワードの詳細

ここでは、それぞれの拡張キーワードについて詳細に説明します。

__absolute

構文

343 ページの *オブジェクト属性の構文*を参照してください。

説明

__absolute キーワードによって、オブジェクトへの参照で絶対アドレスを使用するようにします。

次の制限が適用されます。

- --ropi または --rwpi コンパイラオプションの使用時のみ利用可能
- 外部宣言でのみ使用可能

例

```
extern __absolute char otherBuffer[100];
```

__arm

構文

342 ページの *関数で使用される型属性の構文*を参照してください。

説明

__arm キーワードは、関数を ARM モードで実行します。__arm により宣言される関数は、__interwork が宣言されていない限り、ARM モードでも実行できる関数からのみ呼出しが可能です。

__arm で宣言された関数は __thumb として宣言することはできません。

注 : 非相互作用 ARM 関数は、Thumb モードから呼出すことはできません。

例

```
__arm int func1(void);
```

関連項目

346 ページの *__interwork*。

__big_endian

構文

342 ページの *データオブジェクトで使用される型属性の構文*を参照してください。

説明

__big_endian キーワードは、残りのアプリケーションで使用されるバイトオーダに関係なく、ビッグエンディアンバイトオーダに格納される変数へのアクセスに使用されます。__big_endian キーワードは、ARMv6 以上でコンパイルする場合に使用できます。

このキーワードはポインタ上では使用できません。また、配列上でも使用できません。

例

```
__big_endian long my_variable;
```

関連項目

347 ページの *__little_endian*。

__fiq

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

__fiq キーワードは、高速割込み関数を宣言します。すべての割込み関数は、ARM モードでコンパイルする必要があります。__fiq で宣言された関数には、パラメータを渡すことができないため、値を返しません。このキーワードは、Cortex-M デバイス向けにコンパイルするときは使用できません。

例

```
__fiq __arm void interrupt_function(void);
```

__interwork

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

__interwork により宣言された関数は、ARM または Thumb のいずれかのモードで実行する関数から呼出すことができます。

注 : デフォルトでは、関数は、--interwork コンパイラオプションが使用される場合、および --cpu オプションが使用され、相互作用がデフォルトであるコアを指定する場合に、相互作用になります。

例

```
typedef void (__thumb __interwork *IntHandler)(void);
```

__intrinsic

説明

__intrinsic キーワードは、コンパイラでの内部使用専用に予約されています。

__irq

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

__irq キーワードは、割込み関数を宣言します。すべての割込み関数は、ARM モードでコンパイルする必要があります。__irq により宣言された関数には、パラメータを渡すことができないため、値を返しません。このキーワードは、Cortex-M デバイス向けにコンパイルするときは使用できません。

例

```
__irq __arm void interrupt_function(void);
```

関連項目

253 ページの *--align_sp_on_irq*。

__little_endian

構文

342 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

__little_endian キーワードは、残りのアプリケーションで使用されるバイトオーダに関係なく、リトルエンディアンバイトオーダに格納される変数へのアクセスに使用されます。__little_endian キーワードは、ARMv6 以上でコンパイルする場合に使用できます。

このキーワードはポインタ上では使用できません。また、配列上でも使用できません。

例

```
__little_endian long my_variable;
```

関連項目

346 ページの *__big_endian*。

__nested

構文

343 ページの*オブジェクト属性の構文*を参照してください。

説明

__nested キーワードは、ネストされる割込みを許可する割込み関数の起動および終了コードを修正します。これにより、割込みが有効になります。つま

り、R14 の SPSR およびリターンアドレスを上書きすることなく、新しい割込みを割込み関数に含めることができます。ネストされた割込みは、__irq により宣言された関数のみでサポートされます。

注： __nested キーワードでは、プロセッサモードがユーザモードまたはシステムモードのいずれかであることが必要です。

例

```
__irq __nested __arm void interrupt_handler(void);
```

関連項目 71 ページの*ネスト割込み*、253 ページの *--align_sp_on_irq*。

__no_alloc、__no_alloc16

構文 343 ページの*オブジェクト属性の構文*を参照してください。

説明 定数で __no_alloc または __no_alloc16 オブジェクト属性を使用すると、リンクされたアプリケーション内でスペースをとることなく、実行ファイルでその定数が使用可能になります。

このような定数の内容にはアプリケーションからはアクセスできません。そのアドレス、つまり定数のセクションに対する整数オフセットを取得することはできます。__no_alloc を使用する場合はオフセットの型が unsigned long となり、__no_alloc16 を使用する場合は unsigned short となります。

例

```
__no_alloc const struct MyData my_data @ "XXX" = {...};
```

関連項目 348 ページの *__no_alloc_str*、*__no_alloc_str16*。

__no_alloc_str、__no_alloc_str16

構文

```
__no_alloc_str(string_literal @ section)
```

および

```
__no_alloc_str16(string_literal @ section)
```

説明

string_literal	実行ファイルで使用可能にする文字列リテラル。
section	文字列リテラルを配置するセクション名。

説明

定数で __no_alloc_str または __no_alloc_str16 演算子を使用すると、リンクされたアプリケーション内でスペースをとることなく、実行ファイルで文字列リテラルが使用可能になります。

この式の値は、セクション内の文字列リテラルのオフセットです。
__no_alloc_str の場合、オフセットの型は unsigned long です。
__no_alloc_str16 の場合、オフセットの型は unsigned short です。

例

```
#define MYSEG "YYY"
#define X(str) __no_alloc_str(str @ MYSEG)

extern void dbg_printf(unsigned long fmt, ...)

#define DBGPRINTF(fmt, ...) dbg_printf(X(fmt), __VA_ARGS__)

void
foo(int i, double d)
{
  DBGPRINTF("i の値 : %d、d の値 : %f",i,d);
}
```

使用するデバッガとランタイムサポートによっては、これによってホストコンピュータ上でトレース出力が生成されることがあります。外部のプラグインモジュールを使用しない限り、C-SPY ではこうしたランタイムサポートはない点に注意してください。

関連項目

348 ページの *__no_alloc*、*__no_alloc16*。

__no_init

構文

343 ページの*オブジェクト属性の構文*を参照してください。

説明

__no_init キーワードは、データオブジェクトを不揮発性メモリに配置する場合に使用します。すなわち、変数の初期化（起動時など）が行われなくなります。

例

```
__no_init int myarray[10];
```

関連項目

232 ページの*非初期化変数*、451 ページの *do not initialize ディレクティブ*。

__noreturn

構文

343 ページの*オブジェクト属性の構文*を参照してください。

説明

__noreturn キーワードは、関数がリターンしないことをコンパイラに通知するために使用できます。このような関数でこのキーワードを使用する場合、コンパイラでは、さらに効率的に最適化が可能です。リターンしない関数の例としては、abort や exit などがあります。

注 : 最適化レベル「中」と「高」では、現在の関数がリターン値を返さないと判断された場合は、__noreturn キーワードにより、不正確なコールスタックデバッグ情報が生成されることがあります。

例

```
__noreturn void terminate(void);
```

__packed

構文

342 ページの*データオブジェクトで使用される型属性の構文*を参照してください。例外は struct または union 宣言でキーワードが構造型を変更するために使用されるときです。下記を参照してください。

説明

__packed キーワードを使用して、データ型の 1 のデータアライメントを指定します。__packed を 2 つの方法で使用できます。

- 構造定義で struct または union キーワードの前に使用するとき、構造の各メンバの最大アライメントは 1 に設定され、メンバ間のギャップのために必要になるものを除去します。また、各メンバの型は、__packed 型属性を取り込みます。

  構造宣言で __packed キーワードも使用できますが、__packed キーワードの構造宣言を使用して、__packed キーワードなしで定義した構造型を参照することは不正になります。
- ほかのどの位置を使用しても、型属性の構文規則に従っていて、その全体の型に影響します。__packed 型属性の型は __packed 型属性のない型属性と同じです。違いは 1 のデータアライメントがあることです。1 のデータアライメントがすでにある型は、__packed 型属性によって影響されません。

通常のポインタを __packed へのポインタに明示的に変換することは可能ですが、その逆の変換にはキャストが必要になります。

注 : 通常のアラインメント以外のアラインメントでそのデータ型をアクセスする場合、コードの大幅な増大と速度低下が発生する可能性があります。

ある型のアラインメントおよびその型を使用して定義されるオブジェクトの制限を緩和するには、__packed または #pragma pack を使用します。__packed と #pragma pack を混在させると、予期しない動作が発生することがあります。

例

```
/* X にはパッドバイトなし : */
__packed struct X { char ch; int i; };
/* __packed はここではオプション : */
struct X * xp;

/* 注 : __packed なし : */
struct Y { char ch; int i; };
/* エラー : Y は __packed で定義されていない : */
__packed struct Y * yp ;

/* メンバ 'i' にはアライメント 1 がある : */
struct Z { char ch; __packed int i; };

void Foo(struct X * xp)
{
  /* エラー :"int *" -> "int __packed *" は許可されていない : */
  int * p1 = xp->1;
  /* OK: */
  int __packed * p2 = &xp->i;
  /* OK, char は影響しない */
  char * p3 = &xp->ch;
}
```

関連項目

372 ページの *pack*。

__ramfunc

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

__ramfunc キーワードは、関数を RAM モードで実行します。2 つのコードセクションが作成されます。ひとつは RAM 実行 (.textrw) のためで、もうひとつは ROM の初期化 (.textrw_init) のためです。

__ramfunc により宣言された関数が ROM にアクセスしようとすると、警告が発生します。この動作は、たとえば、フラッシュメモリの一部を再書込みするなど、*アップグレード*ルーチンの作成を簡素化することです。__ramfunc により宣言した関数ではない場合、これらの警告は無視するか、無効にしても問題はありません。

__ramfunc により宣言した関数は、デフォルトでは .textrw という名前のセクションに格納されます。

例

```
__ramfunc int FlashPage(char * data, char * page);
```

関連項目

ブレークポイントに関して __ramfunc 宣言された関数の詳細については、*ARM 用 C-SPY® デバッグガイド*を参照してください。

__root

構文

343 ページの*オブジェクト属性の構文*を参照してください。

説明

__root 属性を持つ関数や変数は、そのモジュールが含まれる場合、アプリケーション内で参照されるかどうかに関わらず保持されます。プログラムモジュールは常に含まれ、ライブラリモジュールは必要に応じて含まれます。

例

```
__root int myarray[10];
```

関連項目

ルートシンボルとその保持方法について詳しくは、99 ページの*シンボルおよびセクションの保持*を参照してください。

__stackless

構文

343 ページの*オブジェクト属性の構文*を参照してください。

説明

__stackless キーワードは、機能するスタックなしに呼出し可能な関数を宣言します。

関数により宣言された __stackless は呼出し規約に違反しているため、そこから戻ることはできません。ただし、コンパイラは関数が戻るかどうかを確実に検出することはできず、検出してもエラーを出力しません。

例

```
__stackless void start_application(void);
```

__swi

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

__swi は、ソフトウェア割込み関数を宣言します。関数呼出しを正しく実行するために必要な SVC（以前の SWI）命令と指定のソフトウェア割込み番号が挿入されます。__swi により宣言された関数は、引数を使用し、値を返すこ

とができます。__swi キーワードを使用することにより、特定のソフトウェア割込み関数に対する正しいリターンシーケンスがコンパイラで生成されます。ソフトウェア割込み関数は、スタックの使用以外、パラメータおよびリターン値に関して通常の関数と同じ呼出し規則に従います。

__swi キーワードには、#pragma swi_number=*number* ディレクティブで指定されるソフトウェア割込み番号が必要です。swi_number は、生成されるアセンブラ SVC 命令への引数として使用されます。また、SVC 割込みハンドラ（たとえば SWI_Handler）が複数のソフトウェア割込み関数を含んだシステムで 1 つのソフトウェア割込み関数を選択するときにも使用できます。ソフトウェア割込み番号は、関数宣言のみに指定（通常、割込み関数を呼出すソースコードにインクルードするヘッダファイルで指定）する必要がある点に注意してくだい。関数定義には指定しないでください。

注 : Cortex-M を除くすべての割込み関数は、ARM モードでコンパイルする必要があります。必要に応じて __arm キーワードまたは #pragma type_attribute=__arm ディレクティブを使用して、デフォルトの動作を変更してください。

例

ソフトウェア割込み関数の宣言は、通常、ヘッダファイルで行い、以下の例のように記述します。

```
#pragma swi_number=0x23
__swi int swi0x23_function(int a, int b);
...
```

関数の呼出し

```
...
int x = swi0x23_function(1, 2);  /* SVC 0x23 によって配置されるため、
                                    リンカは決して swi0x23_ 関数
                                    を配置しようとはしません */
...
```

アプリケーションのソースコード内でソフトウェア割込み関数を定義

```
...
__swi __arm int the_actual_swi0x23_function(int a, int b)
{
  ...
  return 42;
}
```

関連項目

72 ページの *ソフトウェア割込み*、164 ページの *呼出し規約*。

__task

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

このキーワードを使用すると、関数でのレジスタ保護の規則を緩和できます。通常、このキーワードは、RTOS でのタスクの開始関数で使用されます。

`__task` を使用して宣言された関数は、呼出し元関数で必要とされるレジスタを壊す可能性があるため、`__task` を使用するのは、リターンしない関数やアセンブラコードからの呼出す関数のみにする必要があります。

関数 `main` は、アプリケーションから明示的に呼出される場合を除き、`__task` を使用して宣言されるのが普通です。複数のタスクを持つリアルタイムアプリケーションにおいては、通常、それぞれのタスクのルート関数が `__task` を使用して宣言されます。

例

```
__task void my_handler(void);
```

__thumb

構文

342 ページの*関数で使用される型属性の構文*を参照してください。

説明

`__thumb` キーワードは、関数を Thumb モードで実行します。関数が `__interwork` により宣言されていない限り、`__thumb` により宣言された関数の呼出しは、Thumb モードで実行する関数からのみ行うことができます。

関数により制限された `__thumb` は、`__arm` として宣言することはできません。

注 : 非相互作用 Thumb 関数は、ARM モードから呼出すことはできません。

例

```
__thumb int func2(void);
```

関連項目

346 ページの *__interwork*。

__weak

構文

343 ページの*オブジェクト属性の構文*を参照してください。

説明

`__weak` オブジェクト属性をシンボルの外部宣言に使用することにより、モジュール内でのそのシンボルへのすべての参照が弱参照になります。

シンボルの公開されている定義上で `__weak` オブジェクト属性を使用すると、その定義は弱くなります。

リンカは、シンボルへの弱い参照を満たすためだけにライブラリからモジュールをインクルードすることはなく、弱い参照の定義の不足がエラーにつながることもありません。定義がインクルードされない場合、オブジェクトのアドレスはゼロになります。

リンク処理の際、シンボルは弱い定義を必要な数だけと、最大で弱くない定義を 1 つ持つことができます。シンボルが必要な場合は、弱くない定義が 1 つあり、この定義が使用されます。弱くない定義がない場合は、弱い定義のいずれかが使用されます。

例

```
extern __weak int foo; /* 弱い参照 */

__weak void bar(void)  /* 弱い定義 */
{
  /* インクルードされた場合は foo をインクリメント */
  if (&foo != 0)
    ++foo;
}
```

プラグマディレクティブ

- プラグマディレクティブの一覧
- プラグマディレクティブの詳細

プラグマディレクティブの一覧

#pragma ディレクティブは、C 規格によって定義されたものであり、ベンダ固有の拡張の使用方法を規定することにより、ソースコードの移植性を維持するための仕組みです。

プラグマディレクティブは、コンパイラの動作（変数や関数用のメモリの割当て方法、拡張キーワードの許可 / 禁止、ワーニングメッセージの表示 / 非表示など）を制御します。

プラグマディレクティブは、コンパイラでは常に有効になっています。

以下の表には、`#pragma` プリプロセッサディレクティブまたは `_Pragma()` プリプロセッサ演算子で使用可能なコンパイラのプラグマディレクティブの一覧を示します。

プラグマディレクティブ	説明
`bitfields`	ビットフィールドメンバの順序を設定します。
`calls`	間接的なコールに対するコールされうる関数の一覧を表示します。
`call_graph_root`	関数がコールグラフルートであるように指定します。
`data_alignment`	変数のアラインメントを高く（より厳密に）します。
`default_function_attributes`	関数の宣言および定義に対するデフォルトの型とオブジェクトを設定します。
`default_no_bounds`	`#pragma no_bounds` を関数全体のセットに適用します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください。
`default_variable_attributes`	変数の宣言および定義に対するデフォルトの型とオブジェクトを設定します。

表 35: プラグマディレクティブの一覧

プラグマディレクティブ	説明
define_with_bounds	ポインタ変数の境界をトラッキングする関数を組み込みます。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください。
define_without_bounds	C-RUN 用の境界情報を持たない関数のバージョンを定義します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください。
diag_default	診断メッセージの重要度を変更します。
diag_error	診断メッセージの重要度を変更します。
diag_remark	診断メッセージの重要度を変更します。
diag_suppress	診断メッセージを無効にします。
diag_warning	診断メッセージの重要度を変更します。
disable_check	直後の関数が境界へのアクセスをチェックしないように指定します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください。
error	解析の際にエラーについて警告。
generate_entry_without_bounds	直後の関数について、C-RUN 用の境界を持たない追加のエントリの生成を有効化します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください。
include_alias	インクルードファイルのエイリアスを指定します。
inline	関数のインライン化を制御。
language	IAR システムズの言語拡張を設定します。
location	変数の絶対アドレスを指定し、レジスタに変数を配置するか、または指定のセクションに関数のグループを配置します。
message	メッセージを出力します。
no_bounds	直後の関数が C-RUN 用の境界チェックを組み込まれないように指定します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください。
object_attribute	変数または関数の宣言もしくは定義にオブジェクト属性を追加します。
optimize	最適化の種類およびレベルを指定します。

表 35: プラグマディレクティブの一覧 （続き）

プラグマディレクティブ	説明
pack	構造体および共用体メンバのアラインメントを指定します。
__printf_args	printf スタイルフォーマット文字列の関数の呼出しに使用されている引数が正しいかどうかを検証します。
public_equ	パブリックアセンブラのラベルを定義し、それに値を割り当てます。
required	別のシンボルによって必要とされるシンボルが確実にリンク出力に含まれるようにします。
rtmodel	ランタイムモデル属性をモジュールに追加します。
__scanf_args	scanf スタイルフォーマット文字列の関数の呼出しに使用されている引数が正しいかどうかを検証します。
section	組込み関数で使用されるセクション名を宣言します。
segment	このディレクティブは #pragma セクションのエイリアスです。
STDC CX_LIMITED_RANGE	コンパイラで通常の複雑な数式を使用できるかどうかを指定します。
STDC FENV_ACCESS	ソースコードが浮動小数点環境にアクセス可能かどうかを指定します。
STDC FP_CONTRACT	コンパイラが浮動小数点式を縮約できるかどうかを指定します。
swi_number	ソフトウェア割込みの割込み番号を設定します。
type_attribute	宣言または定義に型属性を追加します。
vectorize	ループ用に NEON ベクタ命令の生成を有効または無効にします。
weak	定義を弱くするか、関数または変数に弱いエイリアスを作成します。

表 35: プラグマディレクティブの一覧（続き）

注 : 移植上の理由については、「522 *ページの認識されているプラグマディレクティブ(6.10.6)*」を参照してください。

プラグマディレクティブの詳細

ここでは、各プラグマディレクティブの詳細を説明します。

bitfields

構文

```
#pragma bitfields=disjoint_types|joined_types|
                  reversed_disjoint_types|reversed|default}
```

パラメータ

`disjoint_types`	ビットフィールドメンバは、コンテナ型での最下位ビットから最上位ビットの順に配置されます。基底型が異なるビットフィールドの記憶領域コンテナは重複できません。
`joined_types`	ビットフィールドメンバは、バイトオーダに基づいて配置されます。ビットフィールドの記領域憶コンテナは、他の構造体メンバと重複させることができます。詳細については、328 ページの*ビットフィールド*を参照してください。
`reversed_disjoint_types`	ビットフィールドメンバは、コンテナ型での最上位ビットから最下位ビットの順に配置されます。基底型が異なるビットフィールドの記憶領域コンテナは重複できません。
`reversed`	これは、`reversed_disjoint_types` のエイリアスです。
`default`	ビットフィールドメンバのデフォルトレイアウトを復元します。コンパイラのデフォルトの動作は `joined_types` です。

説明

このプラグマディレクティブは、ビットフィールドメンバのレイアウトを制御する場合に使用します。

例

```
#pragma bitfields=disjoint_types
/* 分割されたビットフィールドの型を使用する構造体 */
struct S
{
  unsigned char  error : 1;
  unsigned char  size  : 4;
  unsigned short code  : 10;
};
#pragma bitfields=default /* デフォルト設定を復元 */
```

関連項目

328 ページの *ビットフィールド*。

calls

構文

```
#pragma calls=function[, function...]
```

パラメータ

function	宣言された任意の関数

説明

このプラグマディレクティブを使用して、次の文で間接的に呼出し可能な関数を一覧表示します。この情報は、リンカでスタック使用量解析に使用できます。

注 : 正確な結果を得るためには、呼出される可能性のある関数をすべて一覧表示する必要があります。

例

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
#pragma calls = Fun1, Fun2
  (*fp)();
}
```

関連項目

87 ページの *スタック使用量解析*。

call_graph_root

構文

```
#pragma call_graph_root[=category]
```

パラメータ

category	オプションのコールグラフルートカテゴリを識別する文字列

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、直後の関数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も指定できます。コンパイラは通常、割込み関数やタスク関数に、コールグラフルートカテゴリを自動的に割り当てます。こうした関数に `#pragma call_graph_root` ディレクティブを使用する場合、デフォルトのカテゴリはオーバライドされます。任意の文字列をカテゴリとして指定できます。

例

```
#pragma call_graph_root="interrupt"
```

関連項目

87 ページの *スタック使用量解析*。

data_alignment

構文

```
#pragma data_alignment=expression
```

パラメータ

expression	定数。2 の累乗（1、2、4 など）を指定する必要があります。

説明

このプラグマディレクティブは、変数に与える開始アドレスのアラインメントを通常よりも高く（より厳密に）する場合に使用します。このディレクティブは、静的 / 自動変数に対して使用できます。

このディレクティブを自動変数に対して使用する場合は、各関数で指定可能なアラインメントに上限が設けられます。この上限は、使用する呼出し規約によって決定されます。

注 : 通常、変数のサイズは、そのアラインメントの倍数です。`data_alignment` ディレクティブは、変数の開始アドレスのみに影響し、サイズには影響しません。そのため、サイズがアラインメントの倍数ではない状況に使用できます。

default_function_attributes

構文

```
#pragma default_function_attributes=[ attribute...]
```

attribute には以下を使用できます。

```
type_attribute
object_attribute
@ section_name
```

パラメータ

type_attribute	341 ページの*型属性*を参照してください。
object_attribute	343 ページの*オブジェクト属性*を参照してください。
@ *section_name*	219 ページの*データと関数のセクションへの配置*を参照してください。

説明

このプラグマディレクティブを使用して、関数の宣言と定義について、デフォルトのセクション配置、型属性、オブジェクト属性を設定します。デフォルト設定は、他の方法で型属性やオブジェクト属性、位置を指定しない宣言および定義に対してのみ使用されます。

属性なしに default_function_attributes プラグマディレクティブを指定すると、デフォルト値が関数の宣言および定義に適用されていない初期の状態が復元されます。

例

```
/* 以下の関数をセクション MYSEC" に配置 */
#pragma default_function_attributes = @ "MYSEC"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1;
/* 関数の MYSEC への配置を停止 */
#pragma default_function_attributes =
```

は以下と同じ効果があります。

```
int fun1(int x) @ "MYSEC" { return x + 1; }
int fun2(int x) @ "MYSEC" { return x - 1; }
```

関連項目

369 ページの *location*

370 ページの *object_attribute*

378 ページの *type_attribute*

default_variable_attributes

構文

```
#pragma default_variable_attributes=[ attribute...]
```

attribute には以下を使用できます。

```
type_attribute
object_attribute
@ section_name
```

パラメータ

type_attribute	341 ページの*型属性*を参照してください。
object_attributes	343 ページの*オブジェクト属性*を参照してください。
@ *section_name*	219 ページの*データと関数のセクションへの配置*を参照してください。

説明

このプラグマディレクティブを使用して、静的記憶寿命変数の宣言と定義について、デフォルトのセクション配置、型属性、オブジェクト属性を設定します。デフォルト設定は、他の方法で型属性やオブジェクト属性、位置を指定しない宣言および定義に対してのみ使用されます。

属性なしに `default_variable_attributes` プラグマディレクティブを指定すると、静的記憶寿命変数にそのようなデフォルト値が適用されていない初期の状態が復元されます。

例

```
/* 以下の変数をセクション MYSEC" に配置 */
#pragma default_variable_attributes = @ "MYSEC"
int var1 = 42;
int var2 = 17;
/* 変数の MYSEC への配置を停止 */
#pragma default_variable_attributes =
```

は以下と同じ効果があります。

```
int var1 @ "MYSEC" = 42;
int var2 @ "MYSEC" = 17;
```

関連項目

369 ページの *location*

370 ページの *object_attribute*

378 ページの *type_attribute*

diag_default

構文

```
#pragma diag_default=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe177` など）

説明

このプラグマディレクティブは、タグで指定される診断メッセージの重要度を変更する場合に使用します。デフォルトの重要度に戻したり、オプション

--diag_error、--diag_remark、--diag_suppress、--diag_warnings を使用してコマンドラインで定義した重要度に変更することができます。

関連項目 241 ページの *診断*。

diag_error

構文

```
#pragma diag_error=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）

説明 このプラグマディレクティブは、指定した診断メッセージの重要度を error に変更する場合に使用します。

関連項目 241 ページの *診断*。

diag_remark

構文

```
#pragma diag_remark=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）

説明 このプラグマディレクティブは、指定した診断メッセージの重要度を remark に変更する場合に使用します。

関連項目 241 ページの *診断*。

diag_suppress

構文

```
#pragma diag_suppress=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明 このプラグマディレクティブは、指定した診断メッセージを無効にする場合に使用します。

関連項目 241 ページの*診断*。

diag_warning

構文
```
#pragma diag_warning=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe826` など）。

説明 このプラグマディレクティブは、指定した診断メッセージの重要度を `warning` に変更する場合に使用します。

関連項目 241 ページの*診断*。

error

構文
```
#pragma error message
```

パラメータ

message	エラーメッセージを表す文字列。

説明 このプラグマディレクティブを使用して、解析時にエラーメッセージを出力します。このメカニズムは、プリプロセッサディレクティブ `#error` とは異なります。`#pragma error` ディレクティブは、`_Pragma` 形式のディレクティブを使用してプリプロセッサマクロにインクルードできるため、マクロが使用されるときにだけエラーとなるためです。

例
```
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error¥"Foo is not available¥"")
#endif
```

`FOO_AVAILABLE` がゼロの場合、`FOO` マクロが実際のソースコードで使用される場合にエラーが警告されます。

include_alias

構文

```
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
```

パラメータ

orig_header	エイリアスを作成するヘッダファイルの名前。
subst_header	元のヘッダファイルのエイリアス。

説明

このプラグマディレクティブは、ヘッダファイルのエイリアスを提供する場合に使用します。これは、あるヘッダファイルを他のヘッダファイルで代用する場合や、相対ファイルへの絶対パスを指定する場合に便利です。

このプラグマディレクティブは、対応する `#include` ディレクティブの前に記述する必要があります。また、`subst_header` は、対応する `#include` ディレクティブに正確に一致する必要があります。

例

```
#pragma include_alias (<stdio.h> , <C:¥MyHeaders¥stdio.h>)
#include <stdio.h>
```

この例では、相対ファイル `stdio.h` を、指定パスにあるファイルで代用します。

関連項目

237 ページの*インクルードファイル検索手順*。

inline

構文

```
#pragma inline[=forced|=never]
```

パラメータ

パラメータはありません	`inline` キーワードと同じ結果になります。
`forced`	コンパイラのヒューリスティックを無効にし、強制的にインライン化します。
`never`	コンパイラのヒューリスティックを無効にして、関数がインライン化されないようにします。

説明

`#pragma inline` を使用して、ディレクティブの直後に定義された関数を C++ のインライン動作に従ってインライン化するようコンパイラに指示します。

`#pragma inline=forced` を指定すると、常に定義された関数がインライン化されます。再帰など何らかの理由でコンパイラが関数をインライン化できない場合、ワーニングメッセージが出力されます。

インライン化は通常、最適化レベル「高」でのみ実行されます。`#pragma inline=forced` を指定すると、最適化レベル「中」でも関数のインライン化が有効になります。

関連項目 74 ページの *インライン関数*。

language

構文

```
#pragma language={extended|default|save|restore}
```

パラメータ

`extended`	プラグマディレクティブを最初に使用してからその後も、IAR システムズの言語拡張を有効にします。
`default`	プラグマディレクティブを最初に使用してからその後も、IAR システムズの言語拡張の設定をコンパイラオプションで指定された状態に復元します。
`save|restore`	ソースコードの一部について、IAR システムズの言語拡張をそれぞれ保存、復元します。 `save` を使用するたびに、`#include` ディレクティブが途中で割込まないように、同じファイル内の一致する `restore` を続いて使用する必要があります。

説明 このプラグマディレクティブを使用して、言語拡張の使用を制御します。

例 IAR システムの拡張を有効にしてコンパイルする必要があるファイルの先頭で：

```
#pragma language=extended
/* ファイルの残りの部分 */
```

IAR システムの拡張を有効にしてコンパイルする必要があるソースコードの特定部分の周辺で、シーケンス前の状態が使用中のコンパイラオプションで指定されたものと同じとは考えられない場合：

```
#pragma language=save
#pragma language=extended
/* ソースコードの一部 */
#pragma language=restore
```

関連項目 263 ページの *-e*、288 ページの *--strict*。

location

構文

```
#pragma location={address|register|NAME}
```

パラメータ

address	絶対位置で指定するグローバル変数や静的変数、または関数の絶対アドレス。
register	ARM コアレジスタ R4-R11 のいずれかに対応する識別子。
NAME	ユーザ定義のセクション名。コンパイラやリンカで使用される定義済のセクション名は指定できません。

説明

このプラグマディレクティブを使用して、以下を指定します。

- グローバル変数または静的変数の場所（絶対アドレス）。プラグマディレクティブの後に宣言が続きます。変数は __no_init として宣言する必要があります。
- レジスタを指定する識別子。プラグマディレクティブの後に定義される変数が、レジスタに配置されます。変数は __no_init として宣言し、ファイルスコープを持つ必要があります。

変数や関数を配置するためのセクションを指定する文字列で、プラグマディレクティブの後に宣言が続きます。通常は異なるセクションにある変数（たとえば、__no_init として宣言される変数と、const として宣言される変数）を、同じ名前のセクションに配置しないでください。

例

```
#pragma location=0xFFFF0400
__no_init volatile char PORT1; /* PORT1 はアドレス
                                  番地に置かれる */

#pragma location=R8
__no_init int TASK; /* TASK が R8 に配置される */

#pragma location="FLASH"
char PORT2; /* PORT2 はセクション FLASH に配置される */

/* 対応メカニズムを利用したより良い方法 */
#define FLASH _Pragma("location=¥"FLASH¥"")
/* ... */
FLASH int i; /* i は FLASH セクションに配置される */
```

関連項目

217 ページの*データと関数のメモリ配置制御*、98 ページの*独自のセクションの宣言および配置*。

message

構文

```
#pragma message(message)
```

パラメータ

message	標準出力ストリームに転送するメッセージ。

説明

このプラグマディレクティブは、コンパイラでファイルのコンパイル時にメッセージを標準出力ストリームに出力する場合に使用します。

例

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

構文

```
#pragma object_attribute=object_attribute[ object_attribute...]
```

パラメータ

このプラグマディレクティブと使用可能なオブジェクト属性の詳細については、343 ページの*オブジェクト属性*を参照してください。

説明

このプラグマディレクティブを使用して、1 つまたは複数の IAR 固有のオブジェクト属性を変数や関数の宣言あるいは定義に追加します。オブジェクト属性は、実際の変数や関数に影響しますが、その型には影響しません。変数や関数を定義する際、定義も含めたあらゆる宣言のオブジェクト属性の共用体が使用されます。

例

```
#pragma object_attribute=__no_init
char bar;
```

は、以下の文と等価です。

```
__no_init char bar;
```

関連項目

341 ページの*拡張キーワードの一般的な構文規則*。

optimize

構文

```
#pragma optimize=[goal][level][no_optimization...]
```

パラメータ

goal	以下から選択します。 size（サイズを重視して最適化） balance（速度とサイズのバランスを最適化） speed（速度を重視して最適化） no_size_constraints: 速度を重視して最適化しますが、コードサイズの拡張のために通常の制限を緩和します。
level	最適化レベルを [none]、[low]、[mid]、[high] から指定します。
no_*optimization*	1 つまたは複数の最適化を無効にします。以下から選択してください。 no_code_motion（コード移動を無効化） no_cse（共通部分式除去を無効化） no_inline（関数のインライン化を無効化） no_tbaa（型ベースエイリアス解析を無効化） no_unroll（ループ展開を無効化） no_vectorize は、NEON ベクタ命令の生成を無効にします no_scheduling（命令スケジューリングを無効化） vectorize は、NEON ベクタ命令の生成を有効にします

説明

このプラグマディレクティブは、最適化レベルを下げる場合や、特定の最適化を無効化する場合に使用します。このプラグマディレクティブは、ディレクティブ直後の関数にのみ影響します。

パラメータ size、balanced、speed、no_size_constraints は、最適化レベル［高］でのみ効果があり、速度とサイズを同時に最適化することはできないため、これらのどれか 1 つだけを使用できます。また、このプラグマディレクティブにプリプロセッサマクロを埋め込むことはできません。埋め込まれたマクロは、プリプロセッサでは展開されません。

注： #pragma optimize ディレクティブを使用して指定した最適化レベルが、コンパイラオプションを使用して指定した最適化レベルよりも高い場合、このプラグマディレクティブは無視されます。

例

```
#pragma optimize=speed
int SmallAndUsedOften()
{
  /* 何らかの処理 */
}

#pragma optimize=size
int BigAndSeldomUsed()
{
  /* 何らかの処理 */
}
```

関連項目

224 ページの *変換の微調整*。

pack

構文

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[,name] [,n])
```

パラメータ

n	次のいずれかの中からオプションの構造体アラインメントを設定します：1、2、4、8、16
空白のリスト	構造体アラインメントをデフォルトに復元します
push	一時的な構造体アラインメントを設定します
pop	構造体アラインメントを一時的にプッシュされたアラインメントから復元します
name	プッシュまたはポップされたオプションのアラインメントラベル

説明

このプラグマディレクティブは、struct および union メンバの最大アラインメントを指定する場合に使用します。

#pragma pack ディレクティブは、プラグマディレクティブに続く構造の宣言を次の #pragma pack またはコンパイルユニットの最後に適用します。

注： この結果、コードが大幅に大きくなり、構造体のメンバにアクセスする際の速度が大幅に低下する可能性があります。

ある型のアラインメントおよびその型を使用して定義されるオブジェクトの制限を緩和するには、__packed または #pragma pack を使用します。__packed と #pragma pack を混在させると、予期しない動作が発生することがあります。

関連項目 335 ページの *構造体型*。

__printf_args

構文

```
#pragma __printf_args
```

説明

このプラグマディレクティブは、printf スタイルフォーマット文字列の関数に使用します。コンパイラは、この関数への任意の呼出しに対して、各変換指定子（たとえば %d）の引数が構文的に正しいかどうかを検証します。

例

```
#pragma __printf_args
int printf(char const *,...);

void PrintNumbers(unsigned short x)
{
  printf("%d", x);  /* コンパイラは x が整数かどうかチェックする */
}
```

public_equ

構文

```
#pragma public_equ="symbol",value
```

パラメータ

symbol	定義するアセンブラシンボルの名前（文字列）。
value	定義済みのアセンブラシンボルの値（整数の定数式）。

説明

このプラグマディレクティブを使用してパブリックのアセンブララベルを定義し、それに値を割り当てます。

例

```
#pragma public_equ="MY_SYMBOL",0x123456
```

関連項目 373 ページの *public_equ*。

required

構文

```
#pragma required=symbol
```

パラメータ

symbol	静的にリンクされた関数または変数。

説明

このプラグマディレクティブは、2 番目のシンボルによって必要とされるシンボルがリンク出力に必ず含まれるようにする場合に使用します。このディレクティブは、2 番目のシンボルの直前に置く必要があります。

このディレクティブは、変数がその格納場所のセクション経由で間接的に参照されるだけの場合など、シンボルが必須かどうかがアプリケーションではわからない場合に使用します。

例

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
  /* 何らかの処理 */
}
```

著作権の文字列がアプリケーションで使用されない場合でも、この文字列がリンカによって含められ、出力に現れます。

rtmodel

構文

```
#pragma rtmodel="key","value"
```

パラメータ

"*key*"	ランタイムモデル属性を指定するテキスト文字列。
"*value*"	ランタイムモデル属性の値を指定するテキスト文字列特殊値 * 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。

説明

このプラグマディレクティブは、ランタイムモデル属性をモジュールに追加する場合に使用します。この属性を使用して、モジュール間の整合性のチェックをリンカで行えます。

このプラグマディレクティブは、モジュール間の整合性を確保するために使用できます。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。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 : 定義済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、ユーザ定義属性ではこのスタイルを使用しないでください。

例

```
#pragma rtmodel="I2C","ENABLED"
```

リンカは、この定義を含むモジュールが、対応するランタイムモデル属性が定義されていないモジュールにリンクされている場合はエラーを生成します。

関連項目 149 ページの *モジュールの整合性チェック*。

__scanf_args

構文

```
#pragma __scanf_args
```

説明

このプラグマディレクティブは、scanf スタイルフォーマット文字列の関数に使用します。コンパイラは、この関数への任意の呼出しに対して、各変換指定子（たとえば %d）の引数が構文的に正しいかどうかを検証します。

例

```
#pragma __scanf_args
int scanf(char const *,...);

int GetNumber()
{
  int nr;
  scanf("%d", &nr);  /* コンパイラが、
                        引数が整数への
                        ポインタであることをチェック */

  return nr;
}
```

section

構文

```
#pragma section="NAME"
```

エイリアス

```
#pragma segment="NAME"
```

パラメータ

NAME	セクションの名前。

説明

このプラグマディレクティブを使用して、セクション演算子の __section_begin、__section_end、__section_size で使用可能なセクション名を定義します。特定のセクションのセクション宣言のメモリ型とアラインメントは、すべて同じでなければなりません。

注 : 変数や関数を特定のセクションに配置するには、#pragma location ディレクティブまたは @ 演算子を使用します。

例

```
#pragma section="MYSECTION"
```

関連項目

179 ページの *専用セクション演算子*セクションおよびセグメントパートについては、「*アプリケーションのリンク*」を参照してください。

STDC CX_LIMITED_RANGE

構文

```
#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}
```

パラメータ

ON	通常の複雑な数式を使用できます。
OFF	通常の複雑な数式は使用できません。
DEFAULT	デフォルトの動作を設定します。つまり OFF です。

説明

このプラグマディレクティブは、コンパイラで *（乗算）、/（除算）、abs に通常の複雑な数式を使用可能に指定するときに使用します。

注 : このディレクティブは、C 規格では必須です。このディレクティブは認識されますが、コンパイラでは何の効果もありません。

STDC FENV_ACCESS

構文

```
#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}
```

パラメータ

ON	ソースコードは浮動小数点環境にアクセスします。この引数はコンパイラではサポートされていない点に注意してください。

OFF	ソースコードは浮動小数点環境にアクセスしません。
DEFAULT	デフォルトの動作を設定します。つまり OFF です。

説明

このプラグマディレクティブを使用して、ソースコードが浮動小数点環境にアクセスするかどうかを指定します。

注 : このディレクティブは、C 規格では必須です。

STDC FP_CONTRACT

構文

```
#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}
```

パラメータ

ON	コンパイラが浮動小数点式を縮約できます。
OFF	コンパイラが浮動小数点式を縮約できません。この引数はコンパイラではサポートされていない点に注意してください。
DEFAULT	デフォルトの動作を設定します。つまり ON です。

説明

このプラグマディレクティブを使用して、コンパイラが浮動小数点式を縮約できるかどうかを指定します。このディレクティブは、C 規格では必須です。

例

```
#pragma STDC FP_CONTRACT=ON
```

swi_number

構文

```
#pragma swi_number=number
```

パラメータ

number	ソフトウェア割込み番号

説明

このプラグマディレクティブは、__swi 拡張キーワードと一緒に使用します。これは、生成されるアセンブラ SVC 命令への引数として使用されます。また、割込み関数などを複数含んだ 1 つのソフトウェア関数をシステムで選択するときにも使用します。

例

```
#pragma swi_number=17
```

関連項目

72 ページの *ソフトウェア割込み*。

type_attribute

構文

```
#pragma type_attribute=type_attr[ type_attr...]
```

パラメータ

このプラグマディレクティブと使用可能な型属性の詳細については、341 ページの*型属性*を参照してください。

説明

このプラグマディレクティブは、C 規格には含まれない IAR 固有の*型属性*を指定する場合に使用します。ただし、指定した型属性がすべてのオブジェクトに適用されるとは限らない点に注意が必要です。

このディレクティブは、プラグマディレクティブ直後の識別子、次の変数、次の関数の宣言に影響します。

例

以下の例では、thumb-mode コードが関数 `foo` に対して生成されます。

```
#pragma type_attribute=__thumb
void foo(void)
{
}
```

以下の宣言は、拡張キーワードを使用して同様の処理を実行します。

```
__thumb void foo(void)
{
}
```

関連項目

*拡張キーワード*の章。

vectorize

構文

```
#pragma vectorize [= never]
```

パラメータ

パラメータはありません	NEON ベクタ命令の生成を有効にします。
`never`	NEON ベクタ命令の生成を無効にします。

説明

このプラグマディレクティブを使用して、このディレクティブの直後に続くループについて、NEON ベクタ命令の生成を有効または無効にします。このプラグマディレクティブは、`for`、`do`、`while` ループの直前にのみ配置できます。最適化レベルが「高」より低い場合、このプラグマディレクティブの効果はありません。

例

```
#pragma vectorize
for (i = 0; i < 1024; ++i)
{
  a[i] = b[i] * c[i];]
}
```

weak

構文

```
#pragma weak symbol1[=symbol2]
```

パラメータ

symbol1	外部リンケージを持つ関数または変数。
symbol2	定義済の関数または変数。

説明

このプラグマディレクティブは次の 2 つのうちどちらかの方法で使用できます。

- 外部リンケージを持つ関数または変数の定義を、弱い定義にする。この目的で、__weak 属性を使用することもできます。
- 別の関数または変数に弱いエイリアスを作成する。同じ関数または変数に、複数のエイリアスを作成できます。

例

foo の定義を弱い定義にするには、次のように記述します。

```
#pragma weak foo
```

NMI_Handler を Default_Handler の弱いエイリアスにするには、次のように記述します。

```
#pragma weak NMI_Handler=Default_Handler
```

NMI_Handler がプログラムの他の場所で定義されていない場合、NMI_Handler へのすべての参照は、Default_Handler も参照します。

関連項目

354 ページの *__weak*。

組込み関数

- 組込み関数の概要
- 組込み関数の詳細

組込み関数の概要

アプリケーションで組込み関数を使用するには、ヘッダファイル `intrinsics.h` をインクルードします。

アプリケーションで Neon 組込み関数を使用するには、ヘッダファイル `arm_neon.h` をインクルードします。詳細については、387 ページの *Neon 命令の組込み関数*を参照してください。

組込み関数名は、次のように最初にダブルアンダースコアが付きます。

```
__disable_interrupt
```

以下の表に、組込み関数の一覧を示します。

組込み関数	説明
`__as_get_base`	パラメータと同じ型のポインタを作成し、そのパラメータによってポイントされるエリアのベースを表します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください
`__as_get_bounds`	パラメータと同じ型のポインタを作成し、そのパラメータによってポイントされるエリアの境界上限を表します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください
`__as_make_bounds`	境界の情報を持つポインタを作成します。*ARM 用 C-SPY® デバッグガイド* で C-RUN のドキュメントを参照してください
`__CLREX`	`CLREX` 命令を挿入します
`__CLZ`	`CLZ` 命令を挿入します
`__disable_fiq`	高速割込み要求 (fiq) を無効にします
`__disable_interrupt`	割込みを禁止します
`__disable_irq`	割込み要求 (irq) を無効にします
`__DMB`	`DMB` 命令を挿入します

表 36: 組込み関数の一覧

組込み関数	説明
__DSB	DSB 命令を挿入します
__enable_fiq	高速割込み要求 (fiq) を有効にします
__enable_interrupt	割込みを有効にします
__enable_irq	割込み要求 (irq) を有効にします
__get_BASEPRI	Cortex-M3/Cortex-M4/Cortex-M7 BASEPRI レジスタの値を返します
__get_CONTROL	Cortex-M CONTROL レジスタの値を返します
__get_CPSR	ARM CPSR（現在のプログラムステータスレジスタ）の値を返します
__get_FAULTMASK	Cortex-M3/Cortex-M4/Cortex-M7 FAULTMASK レジスタの値を返します
__get_FPSCR	FPSCR の値を返します
__get_interrupt_state	割込み状態を返します
__get_IPSR	IPSR レジスタの値を返します
__get_LR	リンクレジスタの値を返します
__get_MSP	MSP レジスタの値を返します
__get_PRIMASK	Cortex-M PRIMASK レジスタの値を返します
__get_PSP	PSP レジスタの値を返します
__get_PSR	PSR レジスタの値を返します
__get_SB	スタティックベースレジスタの値を返します
__get_SP	スタックポインタレジスタの値を返します
__ISB	ISB 命令を挿入します
__LDC	コプロセッサロード命令 LDC を挿入します
__LDCL	コプロセッサロード命令 LDCL を挿入します
__LDC2	コプロセッサロード命令 LDC2 を挿入します
__LDC2L	コプロセッサロード命令 LDC2L を挿入します
__LDC_noidx	コプロセッサロード命令 LDC を挿入します
__LDCL_noidx	コプロセッサロード命令 LDCL を挿入します
__LDC2_noidx	コプロセッサロード命令 LDC2 を挿入します
__LDC2L_noidx	コプロセッサロード命令 LDC2L を挿入します
__LDREX	LDREX 命令を挿入します
__LDREXB	LDREXB 命令を挿入します
__LDREXD	LDREXD 命令を挿入します

表 36: 組込み関数の一覧 （続き）

組込み関数	説明
__LDREXH	LDREXH 命令を挿入します
__MCR	コプロセッサ書込み命令 MCR を挿入します
__MCR2	コプロセッサ書込み命令 MCR2 を挿入します
__MRC	コプロセッサ読取り命令 MRC を挿入します
__MRC2	コプロセッサ読取り命令 MRC2 を挿入します
__no_operation	NOP 命令を挿入します
__PKHBT	PKHBT 命令を挿入します
__PKHTB	PKHTB 命令を挿入します
__PLD	あらかじめロードされたデータ命令 PLD を挿入します
__PLDW	あらかじめロードされたデータ命令 PLDW を挿入します
__PLI	PLI 命令を挿入します
__QADD	QADD 命令を挿入します
__QADD8	QADD8 命令を挿入します
__QADD16	QADD16 命令を挿入します
__QASX	QASX 命令を挿入します
__QCFlag	FPSCR レジスタの累計飽和フラグの値を返します
__QDADD	QDADD 命令を挿入します
__QDOUBLE	QDOUBLE 命令を挿入します
__QDSUB	QDSUB 命令を挿入します
__QFlag	オーバフロー / 飽和が発生しているかどうかを示す Q フラグを返します
__QSAX	QSAX 命令を挿入します
__QSUB	QSUB 命令を挿入します
__QSUB8	QSUB8 命令を挿入します
__QSUB16	QSUB16 命令を挿入します
__RBIT	RBIT 命令を挿入します
__reset_Q_flag	オーバフロー / 飽和が発生しているかどうかを示す Q フラグをクリアします
__reset_QC_flag	FPSCR レジスタの累計飽和フラグ QC の値をクリアします
__REV	REV 命令を挿入します

表 36: 組込み関数の一覧 （続き）

組込み関数	説明
__REV16	REV16 命令を挿入します
__REVSH	REVSH 命令を挿入します
__SADD8	SADD8 命令を挿入します
__SADD16	SADD16 命令を挿入します
__SASX	SASX 命令を挿入します
__SEL	SEL 命令を挿入します
__set_BASEPRI	Cortex-M3/Cortex-M4/Cortex-M7 BASEPRI レジスタの値を設定します
__set_CONTROL	Cortex-M CONTROL レジスタの値を設定します
__set_CPSR	ARM CPSR（現在のプログラムステータスレジスタ）の値を設定します
__set_FAULTMASK	Cortex-M3/Cortex-M4/Cortex-M7 FAULTMASK レジスタの値を設定します
__set_FPSCR	FPSCR レジスタの値を設定します
__set_interrupt_state	割込み状態を復元します
__set_LR	リンクレジスタに新しいアドレスを割り当てます
__set_MSP	MSP レジスタの値を設定します
__set_PRIMASK	Cortex-M PRIMASK レジスタの値を設定します
__set_PSP	PSP レジスタの値を設定します
__set_SB	スタティックベースレジスタに新しいアドレスを割り当てます
__set_SP	スタックポインタレジスタに新しいアドレスを割り当てます
__SEV	SEV 命令を挿入します
__SHADD8	SHADD8 命令を挿入します
__SHADD16	SHADD16 命令を挿入します
__SHASX	SHASX 命令を挿入します
__SHSAX	SHSAX 命令を挿入します
__SHSUB8	SHSUB8 命令を挿入します
__SHSUB16	SHSUB16 命令を挿入します
__SMLABB	SMLABB 命令を挿入します
__SMLABT	SMLABT 命令を挿入します
__SMLAD	SMLAD 命令を挿入します

表 36: 組込み関数の一覧 （続き）

組込み関数	説明
__SMLADX	SMLADX 命令を挿入します
__SMLALBB	SMLALBB 命令を挿入します
__SMLALBT	SMLALBT 命令を挿入します
__SMLALD	SMLALD 命令を挿入します
__SMLALDX	SMLALDX 命令を挿入します
__SMLALTB	SMLALTB 命令を挿入します
__SMLALTT	SMLALTT 命令を挿入します
__SMLATB	SMLATB 命令を挿入します
__SMLATT	SMLATT 命令を挿入します
__SMLAWB	SMLAWB 命令を挿入します
__SMLAWT	SMLAWT 命令を挿入します
__SMLSD	SMLSD 命令を挿入します
__SMLSDX	SMLSDX 命令を挿入します
__SMLSLD	SMLSLD 命令を挿入します
__SMLSLDX	SMLSLDX 命令を挿入します
__SMMLA	SMMLA 命令を挿入します
__SMMLAR	SMMLAR 命令を挿入します
__SMMLS	SMMLS 命令を挿入します
__SMMLSR	SMMLSR 命令を挿入します
__SMMUL	SMMUL 命令を挿入します
__SMMULR	SMMULR 命令を挿入します
__SMUAD	SMUAD 命令を挿入します
__SMUADX	SMUADX 命令を挿入します
__SMUL	符号付き 16 ビット乗算を挿入します
__SMULBB	SMULBB 命令を挿入します
__SMULBT	SMULBT 命令を挿入します
__SMULTB	SMULTB 命令を挿入します
__SMULTT	SMULTT 命令を挿入します
__SMULWB	SMULWB 命令を挿入します
__SMULWT	SMULWT 命令を挿入します
__SMUSD	SMUSD 命令を挿入します
__SMUSDX	SMUSDX 命令を挿入します

表 36: 組込み関数の一覧 （続き）

組込み関数	説明
__SSAT	SSAT 命令を挿入します
__SSAT16	SSAT16 命令を挿入します
__SSAX	SSAX 命令を挿入します
__SSUB8	SSUB8 命令を挿入します
__SSUB16	SSUB16 命令を挿入します
__STC	コプロセッサストア命令 STC を挿入します
__STCL	コプロセッサストア命令 STCL を挿入します
__STC2	コプロセッサストア命令 STC2 を挿入します
__STC2L	コプロセッサストア命令 STC2L を挿入します
__STC_noidx	コプロセッサストア命令 STC を挿入します
__STCL_noidx	コプロセッサストア命令 STCL を挿入します
__STC2_noidx	コプロセッサストア命令 STC2 を挿入します
__STC2L_noidx	コプロセッサストア命令 STC2L を挿入します
__STREX	STREX 命令を挿入します
__STREXB	STREXB 命令を挿入します
__STREXD	STREXD 命令を挿入します
__STREXH	STREXH 命令を挿入します
__SWP	SWP 命令を挿入します
__SWPB	SWPB 命令を挿入します
__SXTAB	SXTAB 命令を挿入します
__SXTAB16	SXTAB16 命令を挿入します
__SXTAH	SXTAH 命令を挿入します
__SXTB16	SXTB16 命令を挿入します
__UADD8	UADD8 命令を挿入します
__UADD16	UADD16 命令を挿入します
__UASX	UASX 命令を挿入します
__UHADD8	UHADD8 命令を挿入します
__UHADD16	UHADD16 命令を挿入します
__UHASX	UHASX 命令を挿入します
__UHSAX	UHSAX 命令を挿入します
__UHSUB8	UHSUB8 命令を挿入します
__UHSUB16	UHSUB16 命令を挿入します

表 36: 組込み関数の一覧 （続き）

組込み関数	説明
__UMAAL	UMAAL 命令を挿入します
__UQADD8	UQADD8 命令を挿入します
__UQADD16	UQADD16 命令を挿入します
__UQASX	UQASX 命令を挿入します
__UQSAX	UQSAX 命令を挿入します
__UQSUB8	UQSUB8 命令を挿入します
__UQSUB16	UQSUB16 命令を挿入します
__USAD8	USAD8 命令を挿入します
__USADA8	USAD8A8 命令を挿入します
__USAT	USAT 命令を挿入します
__USAT16	USAT16 命令を挿入します
__USAX	USAX 命令を挿入します
__USUB8	USUB8 命令を挿入します
__USUB16	USUB16 命令を挿入します
__UXTAB	UXTAB 命令を挿入します
__UXTAB16	UXTAB16 命令を挿入します
__UXTAH	UXTAH 命令を挿入します
__UXTB16	UXTB16 命令を挿入します
__WFE	WFE 命令を挿入します
__WFI	WFI 命令を挿入します
__YIELD	YIELD 命令を挿入します

表 36: 組込み関数の一覧 （続き）

NEON 命令の組込み関数

ARM アーキテクチャで定義された Neon コプロセッサは、Advanced SIMD 命令セット拡張を実装します。アプリケーションで Neon 組込み関数を使用するには、ヘッダファイル arm_neon.h をインクルードします。この関数は、以下のパターンに従って名付けられたベクタ型を使用します。

<*type*><*size*>x<*number_of_lanes*>_t

説明 :

- *type* は、int、unsigned int、float、poly です
- *size* は、8、16、32、64 です
- *number_of_lanes* は、1、2、4、8、16 です

ベクタ型の合計ビット幅は *size* に *number_of_lanes* を掛けた値で、D レジスタ（64 ビット）または Q レジスタ（128 ビット）に収まらなければなりません。

次に例を示します。

```
__intrinsic float32x2_t vsub_f32(float32x2_t, float32x2_t);
```

組込み関数 vsub_f32 は、2 つの 64 ビットベクタ（D レジスタ）上で動作する VSUB.F32 命令を挿入します。それぞれに、32 ビット浮動小数点型の 2 つのエレメント（レーン）があります。

一部の関数はベクタ型の配列を使用します。たとえば、float32x2_t 型の 4 つのエレメントを持つ配列型の定義は以下のようになります。

```
typedef struct
{
  float32x2_t val[4];
}
float32x2x4_t;
```

組込み関数の詳細

ここでは、各組込み関数のリファレンス情報を説明します。

__CLREX

構文

```
void __CLREX(void);
```

説明

CLREX 命令を挿入します。

この組込み関数には、ARM モードの場合はアーキテクチャ ARMv6K または ARMv7、Thumb モードの場合は AVRv7 が必要です。

__CLZ

構文

```
unsigned char __CLZ(unsigned long);
```

説明

CLZ 命令を挿入します。

この組込み関数には ARM モードの場合、ARMv5 アーキテクチャまたはそれ以降、Thumb モードの場合は ARMv6T2 かそれ以降が必要です。

__disable_fiq

構文

```
void __disable_fiq(void);
```

説明

高速割込み要求 (fiq) を無効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__disable_interrupt

構文

```
void __disable_interrupt(void);
```

説明

割込みを禁止します。Cortex-M デバイスに対しては、プライオリティマスクビット PRIMASK を設定することにより、実行優先順位レベルを 0 に上げます。他のデバイスに対しては、割込み要求 (irq) および高速割込み要求 (fiq) を無効にします。

この組込み関数は、特権モードでのみ使用できます。

__disable_irq

構文

```
void __disable_irq(void);
```

説明

割込み要求 (irq) を無効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__DMB

構文

```
void __DMB(void);
```

説明

DMB 命令を挿入します。この組込み関数には、ARMv6M アーキテクチャか ARMv7 アーキテクチャまたはそれ以降が必要です。

__DSB

構文

```
void __DSB(void);
```

説明

DSB 命令を挿入します。この組込み関数には、ARMv6M アーキテクチャか ARMv7 アーキテクチャまたはそれ以降が必要です。

__enable_fiq

構文

```
void __enable_fiq(void);
```

説明

高速割込み要求 (fiq) を有効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__enable_interrupt

構文

```
void __enable_interrupt(void);
```

説明

割込みを有効にします。Cortex-M デバイスに対しては、プライオリティマスクビット PRIMASK をクリアすることにより、実行優先順位レベルをデフォルトに戻し、他のデバイスに対しては、割込み要求 (irq) および高速割込み要求 (fiq) を有効にします。

この組込み関数は、特権モードでのみ使用できます。

__enable_irq

構文

```
void __enable_irq(void);
```

説明

割込み要求 (irq) を有効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__get_BASEPRI

構文

```
unsigned long __get_BASEPRI(void);
```

説明

BASEPRI レジスタの値を返します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3、Cortex-M4、または Cortex-M7 デバイスを使用する必要があります。

__get_CONTROL

構文

```
unsigned long __get_CONTROL(void);
```

説明

CONTROL レジスタの値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__get_CPSR

構文

```
unsigned long __get_CPSR(void);
```

説明

ARM CPSR（現在のプログラムステータスレジスタ）の値を返します。この組込み関数は、特権モードでのみ使用でき、Cortex-M デバイスでは使用できません。また、ARM モードでなければなりません。

__get_FAULTMASK

構文

```
unsigned long __get_FAULTMASK(void);
```

説明

FAULTMASK レジスタの値を返します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3、Cortex-M4、または Cortex-M7 デバイスを使用する必要があります。

__get_FPSCR

構文

```
unsigned long __get_FPSCR(void);
```

説明

FPSCR（浮動小数点ステータスおよび制御レジスタ）の値を返します。

この組込み関数は、VFP コプロセッサを持つデバイスでのみ使用可能です。

__get_interrupt_state

構文

```
__istate_t __get_interrupt_state(void);
```

説明

グローバル割込み状態を返します。リターン値は、__set_interrupt_state 組込み関数の引数として使用して、割込み状態を復元することができます。

この組込み関数は、特権モードでのみ使用でき、--aeabi コンパイラオプションを使用している場合は使用できません。

例

```
#include "intrinsics.h"

void CriticalFn()
{
  __istate_t s = __get_interrupt_state();
  __disable_interrupt();

  /* 何らかの処理 */

  __set_interrupt_state(s);
}
```

__disable_interrupt および __enable_interrupt を使用する場合と比べ、このコードシーケンスを使用する利点は、この例の場合では、__get_interrupt_state の呼出し前に無効化された割込みを有効化することがないことです。

__get_IPSR

構文

```
unsigned long __get_IPSR(void);
```

説明

IPSR レジスタ（割込みプログラムステータスレジスタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_LR

構文

```
unsigned long __get_LR(void);
```

説明

レジスタ (R14) の値を返します。

__get_MSP

構文

```
unsigned long __get_MSP(void);
```

説明

MSP レジスタ（メインスタックポインタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_PRIMASK

構文

```
unsigned long __get_PRIMASK(void);
```

説明

PRIMASK レジスタの値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__get_PSP

構文

```
unsigned long __get_PSP(void);
```

説明

PSP レジスタ（プロセススタックポインタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_PSR

構文

```
unsigned long __get_PSR(void);
```

説明

PSR レジスタ（組み合わせたプログラムステータスレジスタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_SB

構文

```
unsigned long __get_SB(void);
```

説明

スタティックベースレジスタ (R9) の値を返します。

__get_SP

構文

```
unsigned long __get_SP(void);
```

説明

スタックポインタレジスタ (R13) の値を返します。

__ISB

構文

```
void __ISB(void);
```

説明

ISB 命令を挿入します。この組込み関数には、ARMv6M アーキテクチャか ARMv7 アーキテクチャまたはそれ以降が必要です。

__LDC

__LDCL

__LDC2

__LDC2L

構文

```
void __LDCxxx(__ul coproc、__ul CRn、__ul const *src);
```

パラメータ

coproc	コプロセッサ番号 0..15。
CRn	ロードするコプロセッサレジスタです。
src	ロードするデータへのポインタ。

説明

コプロセッサロード命令 LDC（またはその派生形のいずれか）を挿入します。つまり、値がコプロセッサのレジスタにロードされます。パラメータ *coproc* と *CRn* は命令にエンコードされるため、定数でなければなりません。

組込み関数 __LDC と __LDCL には、ARM モードの場合アーキテクチャ ARMv4 またはそれ以降、Thumb モードの場合は ARMv6T2 あるいはそれ以降が必要です。

組込み関数 __LDC2 と __LDC2L には、ARM モードの場合アーキテクチャ ARMv5 またはそれ以降、Thumb モードの場合は ARMv6T2 あるいはそれ以降が必要です。

__LDC_noidx

__LDCL_noidx

__LDC2_noidx

__LDC2L_noidx

構文

```
void __LDCxxx_noidx(__ul coproc, __ul CRn, __ul const *src, __ul option);
```

パラメータ

coproc	コプロセッサ番号 0..15。
CRn	ロードするコプロセッサレジスタです。
src	ロードするデータへのポインタ。
option	追加のコプロセッサオプション 0..255。

説明

コプロセッサロード命令 LDC、またはその派生形のいずれかを挿入します。値は、コプロセッサレジスタにロードされます。パラメータ *coproc* と *CRn*、*option* は命令にエンコードされるため、定数でなければなりません。

組込み関数 __LDC_noidx と __LDCL_noidx には、ARM モードの場合アーキテクチャ ARMv4 またはそれ以降、Thumb モードの場合 ARMv6T2 またはそれ以降が必要です。

組込み関数 __LDC2_noidx と __LDC2L_noidx には、ARM モードの場合アーキテクチャ ARMv5 またはそれ以降、Thumb モードの場合は ARMv6T2 あるいはそれ以降が必要です。

__LDREX

__LDREXB

__LDREXD

__LDREXH

構文

```
unsigned long __LDREX(unsigned long *);
unsigned char __LDREXB(unsigned char *);
unsigned long long __LDREXD(unsigned long long *);
unsigned short __LDREXH(unsigned short *);
```

説明

指定された命令を挿入します。

組込み関数 __LDREX には ARM モードの場合、アーキテクチャ ARMv6 またはそれ以降、Thumb モードの場合は ARMv6T2 かそれ以降が必要です。

組込み関数 __LDREXB と __LDREXH には、ARM モードの場合はアーキテクチャ ARMv6K または ARMv7、Thumb モードの場合は ARMv7 が必要です。

組込み関数 __LDREX には、ARM モードの場合アーキテクチャ ARMv6K または ARMv7、Thumb モードの場合は ARMv7-M は不要ですが ARMv7 が必要です。

__MCR

__MCR2

構文

```
void __MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul
CRm, __ul opcode_2);
void __MCR2(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul
CRm, __ul opcode_2);
```

パラメータ

coproc	コプロセッサ番号 0..15。
opcode_1	コプロセッサ固有の処理コード。
src	コプロセッサに書き込まれる値。
CRn	書込み先のコプロセッサレジスタ。

CRm 追加コプロセッサレジスタ。使用しない場合はゼロに設定します。

opcode_2 追加コプロセッサ固有の処理コード。使用しない場合はゼロに設定します。

説明

コプロセッサ書込み命令（MCR または MCR2）を挿入します。値は、コプロセッサレジスタに書き込まれます。パラメータ *coproc*、*opcode_1*、*CRn*、*CRm*、*opcode_2* は命令内でエンコードされるため、定数でなければなりません。

この組込み関数 __MCR では ARM モードか、Thum モードの場合は ARMv6T2 またはそれ以降が必要です。

この組込み関数 __MCR2 には ARM モードの場合、ARMv5T アーキテクチャまたはそれ以降、Thumb モードの場合は ARMv6T2 かそれ以降が必要です。

__MRC

__MRC2

構文

```
unsigned long __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul
CRm, __ul opcode_2);
unsigned long __MRC2(__ul coproc, __ul opcode_1, __ul CRn, __ul
CRm, __ul opcode_2);
```

パラメータ

coproc コプロセッサ番号 0..15。

opcode_1 コプロセッサ固有の処理コード。

CRn 書込み先のコプロセッサレジスタ。

CRm 追加コプロセッサレジスタ。使用しない場合はゼロに設定します。

opcode_2 追加コプロセッサ固有の処理コード。使用しない場合はゼロに設定します。

説明

コプロセッサ読取り命令（MRC または MRC2）を挿入します。指定されたコプロセッサレジスタの値を返します。パラメータ *coproc*、*opcode_1*、*CRn*、*CRm*、*opcode_2* は命令内でエンコードされるため、定数でなければなりません。

この組込み関数 __MRC では ARM モードか、Thum モードの場合は ARMv6T2 またはそれ以降が必要です。

この組込み関数 __MRC2 には ARM モードの場合、ARMv5T アーキテクチャまたはそれ以降、Thumb モードの場合は ARMv6T2 かそれ以降が必要です。

__no_operation

構文

```
void __no_operation(void);
```

説明

NOP 命令を挿入します。

__PKHBT

構文

```
unsigned long __PKHBT(unsigned long x, unsigned long y, unsigned
long count);
```

パラメータ

x	最初のオペランド。
y	2 番目のオペランド。オプションで左にシフト。
count	シフトカウント 0–31。0 はシフトなし。

説明

カウント用に 1 ～ 31 の範囲でオプションのシフトオペランド (LSL) とともに、PKHBT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R、または ARMv7E-M が必要です。

__PKHTB

構文

```
unsigned long __PKHTB(unsigned long x, unsigned long y, unsigned
long count);
```

パラメータ

x	最初のオペランド。
y	2 番目のオペランド。オプションで右にシフト（算術シフト）。
count	シフトカウント 0–32。0 はシフトなし。

説明

カウント用に 1 ～ 32 の範囲でオプションのシフトオペランド (ASR) とともに、PKHBT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R、または ARM-v7E-M が必要です。

__PLD

__PLDW

構文

```
void __PLD(void const *);
void __PLDW(void const *);
```

説明

あらかじめロードされたデータ命令（PLD または PLDW）を挿入します。

この組込み関数 __PLD では、ARMv7 アーキテクチャを使用する必要があります。__PLDW には MP 拡張（Cortex-A5 など）を持った ARMv7 アーキテクチャが必要です。

__PLI

構文

```
void __PLI(void const *);
```

説明

PLI 命令を挿入します。

この組込み関数では、ARM v7 アーキテクチャを使用する必要があります。

__QADD

__QDADD

__QDSUB

__QSUB

構文

```
signed long __Qxxx(signed long, signed long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv5E かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__QADD8

__QADD16

__QASX

__QSAX

__QSUB8

__QSUB16

構文

```
unsigned long __Qxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__QCFlag

構文

```
unsigned long __QCFlag(void);
```

説明

FPSCR レジスタ（浮動小数点ステータスおよび制御レジスタ）の累計飽和フラグ QC の値を返します。この組込み関数は、Neon (Advanced SIMD) を持つデバイスでのみ使用できます。

__QDOUBLE

構文

```
signed long __QDOUBLE(signed long);
```

説明

ソースレジスタ Rs および宛先レジスタ Rd に対し、命令 QADD Rd,Rs,Rs を挿入します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv5E かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__QFlag

構文

```
int __QFlag(void);
```

説明

オーバフロー / 飽和が発生しているかどうかを示す Q フラグを返します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv5E かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__RBIT

構文

```
unsigned long __RBIT(unsigned long);
```

説明

RBIT 命令を挿入します。これは、32 ビットレジスタのビット順を反転させます。

この組込み関数では、アーキテクチャ ARMv6T2 またはそれ以降を使用する必要があります。

__reset_Q_flag

構文

```
void __reset_Q_flag(void);
```

説明

オーバフロー / 飽和が発生しているかどうかを示す Q フラグをクリアします。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__reset_QC_flag

構文

```
void __reset_QC_flag(void);
```

説明

FPSCR レジスタ（浮動小数点ステータスおよび制御レジスタ）の累計飽和フラグ QC の値をクリアします。この組込み関数は、Neon (Advanced SIMD) を持つデバイスでのみ使用できます。

__REV

__REV16

__REVSH

構文

```
unsigned long __REV(unsigned long);
unsigned long __REV16(unsigned long);
signed long __REVSH(short);
```

説明

指定された命令を挿入します。

これらの組込み関数では、アーキテクチャ ARMv6 またはそれ以降を使用する必要があります。

__SADD8

__SADD16

__SASX

__SSAX

__SSUB8

__SSUB16

構文

```
unsigned long __Sxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SEL

構文

```
unsigned long __SEL(unsigned long, unsigned long);
```

説明

SEL 命令を挿入します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__set_BASEPRI

構文

```
void __set_BASEPRI(unsigned long);
```

説明

BASEPRI レジスタの値を設定します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3、Cortex-M4、または Cortex-M7 デバイスを使用する必要があります。

__set_CONTROL

構文

```
void __set_CONTROL(unsigned long);
```

説明

CONTROL レジスタの値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__set_CPSR

構文

```
void __set_CPSR(unsigned long);
```

説明

ARM CPSR（現在のプログラムステータスレジスタ）の値を設定します。制御フィールドのみが変わります（ビット 0 ～ 7）です。この組込み関数は、特権モードでのみ使用でき、Cortex-M デバイスでは使用できません。また、ARM モードでなければなりません。

__set_FAULTMASK

構文

```
void __set_FAULTMASK(unsigned long);
```

説明

FAULTMASK レジスタの値を設定します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3、Cortex-M4、または Cortex-M7 デバイスを使用する必要があります。

__set_FPSCR

構文

```
void __set_FPSCR(unsigned long);
```

説明

FPSCR（浮動小数点ステータスおよび制御レジスタ）の値を設定します。

この組込み関数は、VFP コプロセッサを持つデバイスでのみ使用可能です。

__set_interrupt_state

構文

```
void __set_interrupt_state(__istate_t);
```

説明

割込み状態を、前回 `__get_interrupt_state` 関数から返された値に復元します。

`__istate_t` 型については、391 ページの *__get_interrupt_state* を参照してください。

__set_LR

構文

```
void __set_LR(unsigned long);
```

説明

リンクレジスタ (R14) に新しいアドレスを割り当てます。

__set_MSP

構文

```
void __set_MSP(unsigned long);
```

説明

MSP レジスタ（メインスタックポインタ）の値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__set_PRIMASK

構文

```
void __set_PRIMASK(unsigned long);
```

説明

PRIMASK レジスタの値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__set_PSP

構文

```
void __set_PSP(unsigned long);
```

説明

PSP レジスタ（プロセススタックポインタ）の値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__set_SB

構文

```
void __set_SB(unsigned long);
```

説明

スタティックベースレジスタ (R9) に新しいアドレスを割り当てます。

__set_SP

構文

```
void __set_SP(unsigned long);
```

説明

スタックポインタレジスタ (R13) に新しいアドレスを割り当てます。

__SEV

構文

```
void __SEV(void);
```

説明

SEV 命令を挿入します。

この組込み関数には、ARM モードの場合はアーキテクチャ ARMv7、Thumb モードの場合 ARMv6-M または ARMv7 が必要です。

__SHADD8

__SHADD16

__SHASX

__SHSAX

__SHSUB8

__SHSUB16

構文

```
unsigned long __SHxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMLABB

__SMLABT

__SMLATB

__SMLATT

__SMLAWB

__SMLAWT

構文

```
unsigned long __SMLAxxx(unsigned long, unsigned long, unsigned
long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMLAD

__SMLADX

__SMLSD

__SMLSDX

構文

```
unsigned long __SMLxxx(unsigned long, unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMLALBB

__SMLALBT

__SMLALTB

__SMLALTT

構文

```
unsigned long long __SMLALxxx(unsigned long, unsigned long, unsigned long long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMLALD

__SMLALDX

__SMLSLD

__SMLSLDX

構文

```
unsigned long long __SMLxxx(unsigned long, unsigned long,
unsigned long long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMMLA

__SMMLAR

__SMMLS

__SMMLSR

構文

```
unsigned long __SMMLxxx(unsigned long, unsigned long, unsigned
long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMMUL

__SMMULR

構文

```
unsigned long __SMMULxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMUAD

__SMUADX

__SMUSD

__SMUSDX

構文

```
unsigned long __SMUxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMUL

構文

```
signed long __SMUL(signed short, signed short);
```

説明

符号付き 16 ビット乗算を挿入します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv5-E かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SMULBB

__SMULBT

__SMULTB

__SMULTT

__SMULWB

__SMULWT

構文

```
unsigned long __SMULxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__SSAT

構文

```
unsigned long __SSAT(unsigned long, unsigned long);
```

説明

SSAT 命令を挿入します。

コンパイラは、可能な場合にはシフト命令をオペランドに組込みます。たとえば、`__SSAT(x << 3,11)` が `SSAT Rd,#11,Rn,LSL #3` にコンパイルされる場合、`x` の値がレジスタ `Rn` に配置されて、`__SSAT` のリターン値はレジスタ `Rd` に配置されます。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R、または ARMv7-M が必要です。

__SSAT16

構文

```
unsigned long __SSAT16(unsigned long, unsigned long);
```

説明

SSAT16 命令を挿入します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R、または ARM v7E-M が必要です。

__STC

__STCL

__STC2

__STC2L

構文

```
void __STCxxx(__ul coproc, __ul CRn, __ul const *dst);
```

パラメータ

coproc	コプロセッサ番号 0..15。
CRn	ロードするコプロセッサレジスタです。
dst	目的の対象へのポインタ。

説明

コプロセッサストア命令 STC（またはその派生形のいずれか）を挿入します。つまり、指定したコプロセッサレジスタの値がメモリの位置に書き込まれます。パラメータ *coproc* と *CRn* は命令にエンコードされるため、定数でなければなりません。

組込み関数 __STC と __STCL には、ARM モードの場合アーキテクチャ ARMv4 またはそれ以降、Thumb モードの場合は ARM v6T2 あるいはそれ以降が必要です。

組込み関数 __STC2 と __STC2L には、ARM モードの場合アーキテクチャ ARMv5 またはそれ以降、Thumb モードの場合は ARMv6-T2 あるいはそれ以降が必要です。

__STC_noidx

__STCL_noidx

__STC2_noidx

__STC2L_noidx

構文

```
void __STCxxx_noidx(__ul coproc, __ul CRn, __ul const *dst, __ul
option);
```

パラメータ

coproc	コプロセッサ番号 0..15。
CRn	ロードするコプロセッサレジスタです。
dst	目的の対象へのポインタ。
option	追加のコプロセッサオプション 0..255。

説明

コプロセッサストア命令 STC（またはその派生形のいずれか）を挿入します。つまり、指定したコプロセッサレジスタの値がメモリの位置に書き込まれます。パラメータ *coproc* と *CRn*、*option* は命令にエンコードされるため、定数でなければなりません。

組込み関数 __STC_noidx と __STCL_noidx には、ARM モードの場合アーキテクチャ ARMv4 またはそれ以降、Thumb モードの場合 ARMv6-T2 またはそれ以降が必要です。

組込み関数 __STC2_noidx と __STC2L_noidx には、ARM モードの場合アーキテクチャ ARMv5 またはそれ以降、Thumb モードの場合 ARMv6-T2 またはそれ以降が必要です。

__STREX

__STREXB

__STREXD

__STREXH

構文

```
unsigned long __STREX(unsigned long, unsigned long *);
unsigned long __STREXB(unsigned char, unsigned char *);
unsigned long __STREXD(unsigned long long, unsigned long long*);
unsigned long __STREXH(unsigned short, unsigned short *);
```

説明

指定された命令を挿入します。

組込み関数 __STREX には ARM モードの場合、アーキテクチャ ARMv6 またはそれ以降、Thumb モードの場合は ARMv6-T2 かそれ以降が必要です。

組込み関数 __STREXB と __STREXH には、ARM モードの場合はアーキテクチャ ARMv6K または ARMv7、Thumb モードの場合は ARMv7 が必要です。

組込み関数 __STREXD には、ARM モードの場合アーキテクチャ ARMv6K または ARMv7、Thumb モードの場合は ARMv7-M は不要ですが ARMv7 が必要です。

__SWP

__SWPB

構文

```
unsigned long __SWP(unsigned long, unsigned long *);
char __SWPB(unsigned char, unsigned char *);
```

説明

指定された命令を挿入します。

これらの組込み関数は、ARM モードでなければなりません。

__SXTAB

__SXTAB16

__SXTAH

__SXTB16

構文

```
unsigned long __SXTxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__UADD8

__UADD16

__UASX

__USAX

__USUB8

__USUB16

構文

```
unsigned long __Uxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__UHADD8

__UHADD16

__UHASX

__UHSAX

__UHSUB8

__UHSUB16

構文

```
unsigned long __UHxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__UMAAL

構文

```
unsigned long long __UMAAL(unsigned long, unsigned long,
unsigned long, unsigned long);
```

説明

UMAAL 命令を挿入します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__UQADD8

__UQADD16

__UQASX

__UQSAX

__UQSUB8

__UQSUB16

構文

```
unsigned long __UQxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__USAD8

__USADA8

構文

```
unsigned long __USADxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__USAT

構文

```
unsigned long __USAT(unsigned long, unsigned long);
```

説明

USAT 命令を挿入します。

コンパイラは、可能な場合にはシフト命令をオペランドに組込みます。たとえば、__USAT(x << 3,11) が USAT Rd,#11,Rn,LSL #3 にコンパイルされる場合、x の値がレジスタ Rn に配置されて、__USAT のリターン値はレジスタ Rd にそれぞれ配置されます。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R、または ARMv7-M が必要です。

__USAT16

構文

```
unsigned long __USAT16(unsigned long, unsigned long);
```

説明

USAT16 命令を挿入します。

この組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__UXTAB

__UXTAB16

__UXTAH

__UXTB16

構文

```
unsigned long __UXTxxx(unsigned long, unsigned long);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合アーキテクチャ ARMv6 かそれ以降、Thumb モードの場合は ARMv7-A、ARMv7-R または ARMv7E-M が必要です。

__WFE

__WFI

__YIELD

構文

```
void long __xxx(void);
```

説明

指定された命令を挿入します。

これらの組込み関数には、ARM モードの場合はアーキテクチャ ARMv7、Thumb モードの場合 ARMv6-M または ARMv7 が必要です。

プリプロセッサ

- プリプロセッサの概要
- 定義済プリプロセッサシンボルの詳細
- その他のプリプロセッサ拡張

プリプロセッサの概要

ARM 用 IAR C/C++ コンパイラのプロプロセッサは、C 規格に準拠しています。また、コンパイラでは、以下のプリプロセッサ関連機能も利用可能です。

- 定義済プリプロセッサシンボル

  これらのシンボルを使用して、コンパイル日時などのコンパイル時の環境を調べることができます。詳細については、420 ページの*定義済プリプロセッサシンボルの詳細*を参照してください。
- コンパイラオプションを使用して定義したユーザ定義プリプロセッサシンボル

  `#define` ディレクティブを使用して独自のプリプロセッサシンボルを定義するほか、`-D` オプションも使用できます（「257 ページの *-D*」を参照）。
- プリプロセッサ拡張

  多数のプラグマディレクティブなど、各種のプリプロセッサ拡張を利用できます。詳細については、「*プラグマディレクティブ*」を参照してください。該当する `_Pragma` 演算子や、その他のプリプロセッサ関連の拡張については、426 ページの*その他のプリプロセッサ拡張*を参照してください。
- プリプロセッサ出力

  プリプロセッサ出力を指定ファイルに出力するには、`--preprocess` オプションを使用します（『283 ページの *--preprocess*』を参照）。

インクルードファイルのパスを指定するには、スラッシュを使用します。

```
#include "mydirectory/myfile"
```

ソースコードでは、スラッシュを使用します。

```
file = fopen("mydirectory/myfile","rt");
```

バックスラッシュも使用可能です。この場合、インクルードファイルパスでは 1 つ、ソースコード文字列では 2 つ使用してください。

定義済プリプロセッサシンボルの詳細

このセクションでは、プログラムプロセッサシンボルの一覧と説明を提供します。

__AAPCS__

説明

--aapcs オプションに基づいて設定される整数。AAPCS の基準の規格が選択された呼出し規約 (--aapcs=std) の場合、シンボルは 1 に設定されます。このシンボルは、他の呼出し規約については未定義です。

関連項目

253 ページの *--aapcs*。

__AAPCS_VFP__

説明

--aapcs オプションに基づいて設定される整数。AAPCS の派生 VFP が選択された呼出し規約 (--aapcs=vfp) の場合、シンボルは 1 に設定されます。このシンボルは、他の呼出し規約については未定義です。

関連項目

253 ページの *--aapcs*。

__ARM_ADVANCED_SIMD__

説明

--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。

関連項目

255 ページの *--cpu*。

__ARM_MEDIA__

説明

--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。

関連項目

255 ページの *--cpu*。

__ARM_PROFILE_M__

説明

--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。

関連項目

255 ページの *--cpu*。

__ARMVFP__

説明

--fpu オプションを反映する整数で、__ARMVFPV2__、__ARMVFPV3__、__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。VFP コード生成が無効な場合（デフォルト）、シンボルの定義は解除されます。

関連項目

267 ページの *--fpu*。

__ARMVFP_D16__

説明

--fpu 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。

関連項目

267 ページの *--fpu*。

__ARMVFP_FP16__

説明

--fpu 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。

関連項目

267 ページの *--fpu*。

__ARMVFP_SP__

説明

--fpu 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。

関連項目

267 ページの *--fpu*。

__BASE_FILE__

説明

コンパイル中の基本ソースファイル（ヘッダファイルでないファイル）の名前を示す文字列です。

関連項目

423 ページの *__FILE__*、276 ページの *--no_path_in_file_macros*。

__BUILD_NUMBER__

説明

使用中のコンパイラのビルド番号を示す固有の整数です。

__CORE__

説明

使用中のチップコアを示す整数です。値は `--cpu` オプションの設定を反映し、`__ARM4TM__`、`__ARM5__`、`__ARM5E__`、`__ARM6__`、`__ARM6M__`、`__ARM6SM__`、`__ARM7M__`、`__ARM7EM__`、`__ARM7A__`、`__ARM7R___` に定義されます。これらのシンボル名は、`__CORE__` シンボルの評価に使用できます。

__COUNTER__

説明

展開されるたびに新しい整数に展開されるマクロ。ゼロ (0) から始まり、増えていきます。

__cplusplus

説明

コンパイラが C++ モードのいずれかで実行する場合に定義される整数です。それ以外の場合には定義されません。定義される場合、その値は `199711L` になります。このシンボルを `#ifdef` で使用し、コンパイラで C++ コードが使用できるかどうかを検出できます。これは、C および C++ のコードで共有するヘッダファイルを作成する場合に特に便利です。

このシンボルは、C 規格で必須です。

__CPU_MODE__

説明

選択した CPU モードを反映する整数です。Thumb の場合 `1`、ARM の場合 `2` と定義されます。

__DATE__

説明

コンパイルした日付を示す文字列です。"Mmm dd yyyy" のフォーマット（"Oct 30 2014" など）で返されます。

このシンボルは、C 規格で必須です。

__embedded_cplusplus

説明

コンパイラが Embedded C++ または拡張 Embedded C++ モードで実行する場合に 1 に定義される整数です。それ以外の場合はシンボルは定義されません。このシンボルを #ifdef で使用し、コンパイラで C++ コードが使用できるかどうかを検出できます。これは、C および C++ のコードで共有するヘッダファイルを作成する場合に特に便利です。

このシンボルは、C 規格で必須です。

__FILE__

説明

コンパイルされるファイルの名前を示す文字列です。基本ソースファイルとインクルードされたヘッダファイルの両方が対象となります。

このシンボルは、C 規格で必須です。

関連項目

422 ページの *__BASE_FILE__*、276 ページの *--no_path_in_file_macros*。

__func__

説明

シンボルが使用される関数名で初期化される定義済の文字列識別子。これは、アサーションやその他のトレースユーティリティで使用します。このシンボルを使用するには、言語拡張を有効にする必要があります。

このシンボルは、C 規格で必須です。

関連項目

263 ページの *-e*、424 ページの *__PRETTY_FUNCTION__*。

__FUNCTION__

説明

シンボルが使用される関数名で初期化される定義済の文字列識別子。これは、アサーションやその他のトレースユーティリティで使用します。このシンボルを使用するには、言語拡張を有効にする必要があります。

関連項目

263 ページの *-e*、424 ページの *__PRETTY_FUNCTION__*。

__IAR_SYSTEMS_ICC__

説明

IAR コンパイラプラットフォームを示す整数です。現行値は 8 です。将来のバージョンでは、番号が大きくなる可能性があります。このシンボルを `#ifdef` で評価し、コードが IAR システムズのコンパイラでコンパイルされたものかどうかを検出できます。

__ICCARM__

説明

コードが ARM 用 IAR C/C++ コンパイラでコンパイルされる場合に、`1` に設定される整数。

__LINE__

説明

コンパイル中のファイルの現在のソースの行番号を示す整数です。基本ソースファイルとインクルードされたヘッダファイルの両方が対象となります。

このシンボルは、C 規格で必須です。

__LITTLE_ENDIAN__

説明

`--endian` オプションの設定を反映する整数で、バイトオーダがリトルエンディアンの場合に `1` に定義されます。バイトオーダーがビッグエンディアンの場合、シンボルは `0` に定義されます。

バイトオーダを反映する整数で、`0`（ビッグエンディアン）に定義されます。

__PRETTY_FUNCTION__

説明

シンボルが使用されている関数の関数名で初期化される定義済みの文字列識別子（パラメータ型、リターン型を含む）。`"void func(char)"` などです。こ

のシンボルは、アサーションやその他のトレースユーティリティで使用します。このシンボルを使用するには、言語拡張を有効にする必要があります。

関連項目 263 ページの *-e*、423 ページの *__func__*。

__ROPI__

説明 `--ropi` コンパイラオプションの使用時に定義される整数。

関連項目 285 ページの *--ropi*。

__RWPI__

説明 `--rwpi` コンパイラオプションの使用時に定義される整数。

関連項目 286 ページの *--rwpi*。

__STDC__

説明 コンパイラが C 規格に準拠する場合に `1` に設定される整数です。このシンボルを `#ifdef` で評価し、使用中のコンパイラが C 規格に準拠しているかどうかを検出できます。

このシンボルは、C 規格で必須です。

__STDC_VERSION__

説明 使用中の C 規格のバージョンを識別する整数。シンボルは `199901L` に拡張します。ただし、`--c89` コンパイラオプションが使用される場合を除きます。この場合は、シンボルは `199409L` に拡張されます。このシンボルは、EC++ モードでは使用できません。

このシンボルは、C 規格で必須です。

__TIME__

説明 コンパイル時刻を `"hh:mm:ss"` というフォーマットで示す文字列です。

このシンボルは、C 規格で必須です。

__TIMESTAMP__

説明

現在のソースファイルの最後の修正日時を識別する文字列定数。文字列のフォーマットは `asctime` 標準関数で使用されるものと同じです（つまり、`"Tue Sep 16 13:03:52 2014"` となります）。

__VER__

説明

使用中の IAR コンパイラのバージョン番号を示す整数です。たとえば、バージョン 5.11.3 の場合、5011003 が返されます。

その他のプリプロセッサ拡張

ここでは、定義済シンボル、プラグマディレクティブ、C 規格ディレクティブ以外に利用可能なプリプロセッサ拡張について説明します。

NDEBUG

説明

このプリプロセッサシンボルは、アプリケーションに記述したアサートマクロをアプリケーションのビルドに含めるかどうかを決定します。

このシンボルを定義していない場合は、すべてのアサートマクロが評価されます。このシンボルを定義している場合は、すべてのアサートマクロがコンパイルから除外されます。すなわち、以下のケースがあります。

- このシンボルを**定義する**場合、アサートコードが含まれ*ない*
- このシンボルを**定義しない**場合、アサートコードが含まれる

したがって、アサートコードを記述し、アプリケーションをビルドする場合、このシンボルを定義することで、アサートコードを最終的なアプリケーションから除外できます。

`assert.h` 標準インクルードファイルでは、アサートマクロは定義されています。

IDE では、リリースビルド構成でアプリケーションをビルドする場合、`NDEBUG` シンボルは自動的に定義されます。

関連項目

142 ページの *Assert*。

#warning message

構文

`#warning` *message*

message には任意の文字列を指定できます。

説明

このプリプロセッサディレクティブは、メッセージを生成する場合に使用します。このディレクティブは、主にアサーションやその他のトレースユーティリティに便利です。C 規格の `#error` ディレクティブの使用方法とよく似ています。このディレクティブは、`--strict` コンパイラオプションの使用時は認識されません。

ライブラリ関数

- ライブラリの概要
- IAR DLIB ライブラリ

ライブラリ関数の詳細については、オンラインヘルプシステムを参照してください。

ライブラリの概要

IAR DLIB ライブラリは、標準の C/C++ に準拠する完全なライブラリです。このライブラリは、IEEE 754 フォーマットの浮動小数点数もサポートしています。また、ロケール、ファイル記述子、マルチバイト文字などのさまざまなレベルのサポートを指定して構成できます。

カスタマイズの詳細は、「*DLIB ランタイム環境*」を参照してください。

ライブラリ関数の詳細については、製品に付属のオンラインドキュメントを参照してください。また、DLIB ライブラリ関数のキーワードのリファレンス情報も提供されています。関数のリファレンス情報を確認するには、エディタウィンドウで関数名を選択し、F1 キーを押します。

ライブラリ関数の詳細については、本ガイドの「*C 規格の処理系定義の動作*」を参照してください。

ヘッダファイル

アプリケーションプログラムは、ヘッダファイルを通じてライブラリ定義にアクセスします。ヘッダファイルは、`#include` ディレクティブを使用して組込みます。ライブラリ定義は、複数の異なるヘッダファイルに分割されています。各ヘッダファイルは、特定の機能領域に対応しており、必要なものだけをインクルードできます。

ライブラリ定義を参照する前に、該当ヘッダファイルをインクルードする必要があります。これを行っていない場合、実行時に呼出しに失敗するか、コンパイルやリンク時にエラーやワーニングメッセージが出力されます。

ライブラリオブジェクトファイル

ほとんどのライブラリ定義は、修正なしで、すなわち製品付属のライブラリオブジェクトから直接使用できます。ランタイムライブラリの設定方法については、「110 ページの *ランタイム環境の設定*」を参照してください。リンカ

は、アプリケーションで直接的または間接的に必要なルーチンのみを含めます。

より高精度な代替ライブラリ関数

cos、sin、tan、pow のデフォルトの実装は、高速かつ小さくなるように設計されています。もうひとつの方法として、より高い精度を提供するように考えられたバージョンがあります。これらの名前は __iar_*xxx*_accuratef（関数の float 派生型）、__iar_*xxx*_accuratel（関数の long double 派生型）で、*xxx* は cos、sin などです。

より正確な以下のバージョンを使用するには、--redirect リンカオプションを使用します。

リエントラント性

関数をメインのアプリケーションや任意の数の割込みで同時に呼出すことが可能な場合、その関数はリエントラントであると言います。したがって、静的に割り当てられたデータを使用するライブラリ関数はリエントラントではありません。

DLIB ライブラリの大部分はリエントラントですが、以下の関数や部分は静的データを必要とするためリエントラントではありません。

- ヒープ関数 — malloc、free、realloc、calloc。C++ 演算子 — new と delete
- ロケール関数 — localeconv、setlocale
- マルチバイト関数 — mbrlen、mbrtowc、mbsrtowc、mbtowc、wcrtomb、wcsrtomb、wctomb
- ランド関数 — rand、srand
- 時間関数 — asctime、localtime、gmtime、mktime
- その他の関数 — atexit、strerror、strtok
- ファイルやヒープを何らかの方法で使用するすべての関数 — これには scanf、sscanf、getchar、putchar があります。また、オプション --enable_multibyte と --dlib_config=Full を使用する場合、printf と sprintf の関数（または任意の派生型）でもヒープを使用できます。

errno を設定できる関数はリエントラントではありません。その理由は、これらの関数のいずれかの結果となる errno の値は、読み込まれる前に後続の関数の使用によって上書きされる可能性があるためです。これは、特に数学関数および文字列変換関数に適用します。

以下の解決方法があります。

- 非リエントラント関数を割込みサービスルーチンで使用しない
- 非リエントラント関数の呼出しをミューテックス、保護エリアなどを利用して保護する

LONGJMP 関数

`longjmp` は、実質的に以前に定義された `setjmp` へのジャンプです。スタックの巻き戻し中にスタック上にある可変長配列や C++ オブジェクトは、どれも破壊されません。これは、リソースのリークや不正なアプリケーション動作の原因となることがあります。

IAR DLIB ライブラリ

IAR DLIB ライブラリは、組込みシステムに利用される最も重要な C/C++ ライブラリ定義を提供します。提供される定義は、以下のとおりです。

- C 規格のフリースタンディング実装への準拠。ライブラリはホストされた機能の大半をサポートしますが、基本機能の一部は実装する必要があります。詳細については、本ガイドの「*C 規格の処理系定義の動作*」を参照してください
- ユーザプログラム用標準 C ライブラリ定義
- C++ ライブラリの定義、ユーザプログラム用
- `CSTARTUP`。起動コードを含むモジュール。本ガイドの *DLIB ランタイム環境*を参照してください
- 低レベルの浮動小数点数ルーチンなどのランタイムサポートライブラリ
- 低レベルの ARM 機能を利用するための組込み関数。詳細については、「*組込み関数*」を参照してください

また、IAR DLIB ライブラリには C の追加機能も含まれています。436 ページの *C の追加機能*を参照してください。

C ヘッダファイル

ここでは、DLIB ライブラリの C 定義専用のヘッダファイルについて説明します。ヘッダファイルには、ターゲット固有の定義が追加されている場合があります。これらについては、「*C の使用*」で説明しています。

次の表は、C ヘッダファイルの一覧を示します。

ヘッダファイル	用途
assert.h	関数実行時のアサーション実行
complex.h	一般的かつ複雑な数学関数の計算
ctype.h	文字の分類
errno.h	ライブラリ関数が出力したエラーコードの評価
fenv.h	浮動小数点例外フラグ
float.h	浮動小数点数型プロパティの評価
inttypes.h	stdint.h で定義されたあらゆるタイプのフォーマットを定義
iso646.h	Amendment I の iso646.h 標準ヘッダ
limits.h	整数型プロパティの評価
locale.h	さまざまな文化圏の慣習への対応
math.h	一般的な数学関数の計算
setjmp.h	非ローカルの goto 文の実行
signal.h	さまざまな例外条件の制御
stdarg.h	可変引数のアクセス
stdbool.h	C の bool 型のサポートを追加
stddef.h	さまざまな有用な型やマクロを定義
stdint.h	整数特性を提供
stdio.h	I/O の実行
stdlib.h	さまざまな処理の実行
string.h	さまざまな種類の文字列の操作
tgmath.h	汎用型の数学関数
time.h	さまざまな時刻 / 日付フォーマットの変換
uchar.h	Unicode 機能（C 規格に対する IAR 拡張）
wchar.h	ワイド文字のサポート
wctype.h	ワイド文字の分類

表 37: 従来の標準 C ヘッダファイル — DLIB

C++ ヘッダファイル

ここでは、C++ ヘッダファイルについて説明します。

- C++ ライブラリヘッダファイル
  標準 C++/Embedded C++ ライブラリを構成するヘッダファイル。

- C++ 標準テンプレートライブラリ (STL) ヘッダファイル
  標準 C++/Extended Embedded C++ ライブラリの STL を構成するヘッダファイル。
- C++ C ヘッダファイル
  C ライブラリからのリソースを提供する C++ ヘッダファイル。

C++ ライブラリヘッダファイル

この表は、C++ および Embedded C++ で使用可能なヘッダファイルの一覧です。

ヘッダファイル	用途
complex	複素数演算をサポートするクラスを定義
exception	例外処理を制御するいくつかの関数を定義（C++ でのみ使用可能）
fstream	外部ファイルを操作する複数の I/O ストリームクラスを定義
iomanip	引数を 1 つ指定する複数の I/O ストリームマニピュレータを宣言
ios	多くの I/O ストリームクラスとして機能するクラスを定義
iosfwd	I/O ストリームクラスの定義が必要となる前に複数の I/O ストリームクラスを宣言
iostream	標準ストリームを操作する I/O ストリームオブジェクトを宣言
istream	抽出を実行するクラスを定義
limits	数値を定義（C++ でのみ使用可能）
locale	さまざまな文化圏の慣習への対応（C++ でのみ使用可能）
new	記憶領域の割当て / 解放を行う複数の関数を宣言
ostream	挿入を実行するクラスを定義
sstream	文字列コンテナを操作する複数の I/O ストリームクラスを定義
stdexcept	例外のレポートに役立ついくつかのクラスを定義（C++ でのみ使用可能）
streambuf	I/O ストリーム処理のバッファ処理を行うクラスを定義
string	文字列コンテナを実装するクラスを定義
strstream	メモリ内の文字列シーケンスを操作する複数の I/O ストリームクラスを定義
typeinfo	型情報のサポートを定義（C++ でのみ使用可能）

表 38: C++ ヘッダファイル

C++ 標準テンプレートライブラリ (STL) ヘッダファイル

次の表は、C++ および Extended Embedded C++ で使用可能な標準テンプレートライブラリ (STL) のヘッダファイルの一覧です。

ヘッダファイル	説明
algorithm	シーケンスに対する一般的な処理を複数定義
bitset	固定サイズのビットシーケンスによりコンテナを定義（C++ でのみ使用可能）
deque	デキューシーケンスコンテナ
functional	複数の関数オブジェクトを定義
hash_map	ハッシュアルゴリズムに基づく map 連想コンテナ
hash_set	ハッシュアルゴリズムに基づく set 連想コンテナ
iterator	共通のイテレータと、イテレータに対する処理を定義
list	双方向リンクリストシーケンスコンテナ
map	map 連想コンテナ
memory	メモリ管理機能定義
numeric	シーケンスに対する一般的な数値操作
queue	キューシーケンスコンテナ
set	set 連想コンテナ
slist	一方向リンクリストシーケンスコンテナ
stack	スタックシーケンスコンテナ
utility	複数のユーティリティコンポーネントを定義
valarray	可変長配列のコンテナを定義（C++ でのみ使用可能）
vector	ベクタシーケンスコンテナ

表 39: 標準テンプレートライブラリヘッダファイル

C++ での標準 C ライブラリの使用

標準 C ライブラリの一部のヘッダファイル（場合によって多少の変更あり）が C++ ライブラリとともに動作します。これらのヘッダファイルは、cassert と assert.h のように、新フォーマットと従来フォーマットの 2 つで提供されます。

次の表は、新しいヘッダファイルの一覧を示します。

ヘッダファイル	用途
cassert	関数実行時のアサーション実行
ccomplex	一般的かつ複雑な数学関数の計算

表 40: 新しい標準 C ヘッダファイル— DLIB

ヘッダファイル	用途
cctype	文字の分類
cerrno	ライブラリ関数が出力したエラーコードの評価
cfenv	浮動小数点例外フラグ
cfloat	浮動小数点数型プロパティの評価
cinttypes	stdint.h で定義されたあらゆるタイプのフォーマットを定義
ciso646	Amendment 1 の iso646.h 標準ヘッダ
climits	整数型プロパティの評価
clocale	さまざまな文化圏の慣習への対応
cmath	一般的な数学関数の計算
csetjmp	非ローカルの goto 文の実行
csignal	さまざまな例外条件の制御
cstdarg	可変引数のアクセス
cstdbool	C の bool データ型のサポートを追加
cstddef	さまざまな有用な型やマクロを定義
cstdint	整数特性を提供
cstdio	I/O の実行
cstdlib	さまざまな処理の実行
cstring	さまざまな種類の文字列の操作
ctgmath	汎用型の数学関数
ctime	さまざまな時刻 / 日付フォーマットの変換
cwchar	ワイド文字のサポート
cwctype	ワイド文字の分類

表 40: 新しい標準 C ヘッダファイル — DLIB （続き）

組込み関数としてのライブラリ関数

特定の C ライブラリ関数は、状況によっては組込み関数として扱われ、memcpy、memset、strcat など、通常の関数呼出しではなくインラインコードを生成します。

C の追加機能

IAR DLIB ライブラリには、追加された C 機能がいくつか含まれています。

これらの機能は、以下のインクルードファイルによって提供されます。

- fenv.h
- iar_dlmalloc.h
- stdio.h
- stdlib.h
- string.h
- time.h

fenv.h

fenv.h では、浮動小数点の数値のトラップ処理のサポートが、関数 fegettrapenable および fegettrapdisable によって定義されています。

iar_dlmalloc.h

iar_dlmalloc.h ヘッダファイルには、アドバンストヒープハンドラ (dlmalloc) のサポートが含まれます。詳細については、199 ページの *ヒープについて*を参照してください。

stdio.h

以下の関数は、追加の I/O 機能を提供します。

fdopen	低レベルのファイル記述子に基づいてファイルを開きます。
fileno	ファイル記述子 (FILE*) から低レベルのファイル記述子を取得します。
__gets	stdin での fgets に相当します。
getw	stdin から wchar_t 文字を取得します。
putw	wchar_t 文字を stdout に配置します。
__ungetchar	stdout での ungetc に相当します。
__write_array	stdout での fwrite に相当します。

string.h

以下は、string.h に定義された追加の関数です。

strdup	ヒープ上の文字列を複製します。
strcasecmp	大文字 / 小文字を区別しない文字列を比較します。
strncasecmp	大文字 / 小文字を区別する境界のある文字列を比較します。
strnlen	境界のある文字列の長さ。

time.h

time_t および関連の関数 time、ctime、difftime、gmtime、localtime、mktime を使用するために、2 つのインタフェースがあります。

- 32 ビットのインタフェースは、1900 年から 2035 年までをサポートし、time_t で 32 ビットの整数を使用します。型と関数は __time32_t、__time32 のような名前を持っています。この派生形は、主に旧バージョンとの互換性のためだけに使用できます。
- 64 ビットのインタフェースは -9999 年から 9999 年をサポートし、time_t で符号付きの long long を使用します。型と関数は __time64_t、__time64 などのような名前を持っています。

どちらのインタフェースでも、time_t は 1970 年から始まります。

インタフェースは、システムヘッダファイル time.h で定義されます。

アプリケーションはどちらのインタフェースも使用でき、32 ビットまたは 64 ビットの派生形を明示的に使用して両方を混在させることも可能です。デフォルトでは、ライブラリとヘッダは time_t や time などを 32 ビットの派生形にリダイレクトします。ただし、これらを明示的に 64 ビットの派生形にリダイレクトするには、time.h または ctime のインクルードの前に _DLIB_TIME_USES_64 を定義します。

139 ページの *時間*を参照してください。

clock_t は、32 ビットの整数型で表します。

ライブラリにより内部的に使用されるシンボル

以下のシンボルはライブラリで使用されます。つまり、ライブラリのソースファイルなどで可視ということです。

__assignment_by_bitwise_copy_allowed

このシンボルは、クラスオブジェクトのプロパティを決定します。

__code, __data

これらのシンボルは、コンパイラでメモリ属性として内部的に使用されます。特定のテンプレートでは引数として使用しなければならないこともあります。

__constrange()

組込み関数へのパラメータの有効範囲と、パラメータの型が const でなければならないことを決定します。

__construction_by_bitwise_copy_allowed

このシンボルは、クラスオブジェクトのプロパティを決定します。

__has_constructor, __has_destructor

これらのシンボルはクラスオブジェクトのプロパティを決定し、sizeof 演算子と同様に機能します。クラス、基底クラス、メンバ（再帰的）にユーザ定義のコンストラクタまたはデストラクタがそれぞれある場合、シンボルは真となります。

__memory_of

クラスメモリを決定します。クラスメモリによって、クラスメモリが存在できるメモリが決まります。このシンボルは、クラスメモリとしてクラス定義の中でのみ発生できます。

注： これらのシンボルは予約済みであり、ライブラリでのみ使用してください。

定義済みのシンボルの値を判断するには、コンパイラオプションの --predef_macros を使用します。

リンカ設定ファイル

- 概要
- メモリおよび領域の定義
- 領域
- セクションの取扱い
- セクションの選択
- シンボル、式、数値の使用
- 構造化設定

この章を読む前に、*セクション*のコンセプトについて知っておく必要があります。78 ページの *モジュールおよびセクション*を参照してください。

概要

要件に合わせてメモリのアプリケーションをリンクおよび検出するため、ILINK には、セクションを扱う方法、および使用できるメモリエリアにセクションを配置する方法に関する情報が必要です。つまり、ILINK には、*リンカ設定ファイル*により渡される*設定*が必要です。

このファイルは、一連のディレクティブで構成され、通常、以下のことを行います。

- 使用できるアドレス可能メモリを定義する
  可能なアドレスの最大サイズに関するリンカ情報を提供し、使用可能な物理メモリを定義します。また、さまざまな方法でのアドレスが可能なメモリを扱います。
- ROM または RAM の使用可能メモリエリアを定義する
  各領域の開始および終了アドレスを提供します。
- セクショングループ
  セクション要件に従ってセクションをブロックまたはオーバレイにグループ化する方法を扱います。

- アプリケーションの初期化を扱う方法を定義する

  初期化されるセクションに関する情報、およびその初期化の方法に関する情報を提供します。
- メモリ割当て

  セクションの各セットが配置されるメモリエリアを定義します。
- シンボル、式、数値の使用

  アドレスやサイズなどを他の設定ディレクティブで表現します。シンボルは、アプリケーション自体でも使用できます。
- 構造化設定

  条件に応じてディレクティブを含める、または除外し、設定ファイルをいくつかの異なるファイルに分割できます。

コメントは、C コメント (`/*...*/`) または C++ コメント (`//...`) のいずれかとして記述できます。

メモリおよび領域の定義

ILINK には、使用可能なメモリ空間に関する情報、具体的には以下の情報が必要です。

- 使用できるアドレス可能メモリの最大サイズ

  `define memory` ディレクティブは、指定したサイズで*メモリ空間*を定義します。これは、アドレス可能メモリの最大サイズで、必ずしも物理的に使用できるサイズではありません。440 ページの *define memory ディレクティブ*を参照してください。
- 使用可能な物理メモリ

  `define region` ディレクティブは、アプリケーションコードの特定のセクションおよびアプリケーションデータのセクションを配置できる使用可能メモリの領域を定義します。441 ページの *define region ディレクティブ*を参照してください。

  領域は、1 つ以上のメモリエリアで構成されます。領域は、メモリ内での連続するバイトで、領域式を使用して複数の領域を表現できます。443 ページの*領域式*を参照してください。

define memory ディレクティブ

構文

```
define memory [ name ] with size = size_expr [ ,unit-size ];
```

ここで、*unit-size* は以下のいずれかです。

```
unitbitsize = bitsize_expr
unitbytesize = bytesize_expr
```

また、*expr* は式です（461 ページの*式*を参照）。

パラメータ

size_expr	メモリ空間に含まれるユニットの量を指定。これは常にアドレスゼロからカウントされます。
bitsize_expr	各ユニットに含まれるビット数を指定します。
bytesize_expr	各ユニットに含まれるバイト数を指定します。各バイトには 8 ビットが含まれます。

説明

define memory ディレクティブは、指定したサイズで*メモリ空間*を定義します。これは、アドレス可能メモリの最大サイズで、必ずしも物理的に使用できるサイズではありません。このディレクティブは、使用可能アドレスの制限をリンカ設定ファイルで設定します。通常のマイクロコントローラでは、1 つのメモリ空間で十分ですが、場合によっては、複数のメモリ空間が必要です。たとえば、ハーバードアーキテクチャでは、通常、コードとデータ用に 1 つずつ、2 つの異なるメモリ空間が必要です。メモリが 1 つだけ定義されている場合、そのメモリ範囲はオプションです。*unit-size* が指定されていない場合、ユニットには 8 ビットが含まれます。

例

```
/* 4 ギガバイトのメモリ空間 Mem を宣言 */
define memory Mem with size = 4G;
```

define region ディレクティブ

構文

```
define region name = region-expr;
```

ここで、*region-expr* は領域式です（442 ページの*領域*を参照）。

パラメータ

name	領域の名前。

説明

define region ディレクティブは、コードの特定のセクションおよびデータのセクションを配置できる領域を定義します。領域は、1 つ以上のメモリエリアで構成されます。各メモリエリアは、特定のメモリ内の連続するバイトで構成されます。領域式を使用して、複数の範囲を組み合わせることができます。これらの範囲では、バイトが連続しなくても、同じメモリになくてもかまいません。

例

```
/* 0x10000 バイトのコード領域の ROM（メモリ Mem のアドレス
   0x10000 に配置）を定義 */
define region ROM = Mem:[from 0x10000 size 0x10000];
```

領域

*領域*は、重複しないメモリ範囲のセットです。*領域式*は、領域の*領域リテラル*およびセット操作（結合、交差、相違）で構成されます。

領域リテラル

構文

```
[ memory-name: ][from expr { to expr | size expr }

  [ repeat expr [ displacement expr ]]]
```

ここで、*expr* は式です（461 ページの*式*を参照）。

パラメータ

memory-name	領域リテラルが配置されるメモリ空間の名前メモリが 1 つだけの場合、名前はオプションです。
`from` *expr*	*expr* は、表示するメモリ範囲の開始アドレス（開始アドレスを含む）。
`to` *expr*	*expr* は、表示するメモリ範囲の終了アドレス（終了アドレスを含む）。
`size` *expr*	*expr* は、メモリ範囲のサイズ。
`repeat` *expr*	*expr* は、領域リテラルに同じメモリの複数の範囲を定義します。
`displacement` *expr*	*expr* は、繰返しシーケンスで前の範囲開始からの移動距離です。デフォルトの移動距離は、範囲サイズと同じ値です。

説明

領域リテラルは、1 つのメモリ範囲で構成されます。範囲を定義する場合、範囲が配置されるメモリ、開始アドレス、サイズを指定する必要があります。範囲サイズは、サイズを指定して明示的に指定するか、範囲の終了アドレスを指定して暗黙的に指定できます。終了アドレスが範囲に含まれ、ゼロサイズ領域にはアドレスのみが含まれます。メモリがどこでラップされるかが認識されているため、範囲がアドレスゼロをスナップしたり、そのような範囲が符号なし値で表現したりできます。

repeat パラメータは、各繰返しに 1 つずつ、複数の範囲を含む領域リテラルを作成します。これは、*バンク*または*ファー*領域で便利です。

例

```
/* 0 番地中心の 5 バイトの領域 */
Mem:[from -2 to 2]

/* 64kB のメモリー中で、0 番地中心の 512 バイトの領域 */
Mem:[from 0xFF00 to 0xFF]

/* 同じメモリ上の、いくつかの繰り返し領域
   リテラル */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* 次の定義と同じ :
   Mem:[from 0 size 0x100]
   Mem:[from 0x1000 size 0x100]
   Mem:[from 0x2000 size 0x100]
*/
```

関連項目

441 ページの *define region ディレクティブ*、443 ページの *領域式*。

領域式

構文

```
region-operand
 | region-expr | region-operand
 | region-expr - region-operand
 | region-expr & region-operand
```

ここで、*region-operand* は以下のいずれかです。

```
( region-expr )
region-name
region-literal
empty-region
```

ここで、*region-name* は領域です（詳細は 441 ページの *define region ディレクティブ*を参照）。

region-literal は領域リテラルです（詳細は 442 ページの *領域リテラル*を参照）。

empty-region は空領域です（詳細は 444 ページの *空の領域*を参照）。

説明

通常、領域は、1 つのメモリ範囲で構成されます。つまり、1 つの *領域リテラル*で領域を表現できます。領域に複数の範囲が（場合によっては異なるメモリに）含まれる場合、*領域式*を使用して領域を表現する必要があります。領域式は、実際、メモリ範囲のセットにおけるセット式です。

領域式を作成するために 3、つの演算子、結合 (|)、交差 (&)、相違 (-) が使用できます。これらの演算子は、*セット理論*に基づいて機能します。たとえば、セット A および B がある場合、演算子の結果は以下のようになります。

- A | B: セット A またはセット B いずれかのすべてのエレメント
- A & B: セット A およびセット B 両方のすべてのエレメント
- A - B: セット A にありセット B にないすべてのエレメント

例

```
/* 結果は Mem 上の 1000 - 2FFF の
   範囲になる */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 1500 - 1FFF の
   範囲になる */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 1000 - 14FF の
   範囲になる */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 2つの範囲 1000 - 1FFF
   と 2501 - 2FFF。
   になる */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

空の領域

構文

```
[ ]
```

説明

空の領域には、メモリ範囲は含まれません。空の領域が、1 つ以上のセクションの配置のために実際に使用される配置ディレクティブで使用される場合、ILINK ではエラーが発生します。

例

```
define region Code = Mem:[from 0 size 0x10000];
if (Banked) {
  define region Bank = Mem:[from 0x8000 size 0x1000];
} else {
  define region Bank = [];
}
define region NonBanked = Code - Bank;

/* シンボル Banked により、NonBanked 領域は
   0x10000 バイトの 1 つの領域か 0x8000 バイトと
   0x7000 バイトの二つの領域になる。*/
```

関連項目　443 ページの *領域式*。

セクションの取扱い

セクションの取扱いは、ILINK で実行イメージのセクションをどう処理するかについて説明します。これは以下のことを意味します。

- セクションを領域に配置する

  `place at` ディレクティブと `place into` ディレクティブは、似ている属性を持つセクションのセットを、以前に定義された領域に配置します。452 ページの *place at ディレクティブ*および 453 ページの *place in ディレクティブ*を参照してください。

- 特殊な要件のセクションのセットを作成する

  `block` ディレクティブを使用すると、特殊なサイズおよびアラインメントを持つ、またはタイプの異なるシーケンシャルにソートされたセクションなど、空のセクションを作成できます。

  `overlay` ディレクティブを使用すると、複数のオーバレイイメージを含むことができるメモリの領域を作成できます。446 ページの *define block ディレクティブ*、447 ページの *define overlay ディレクティブ*を参照。

- アプリケーションの初期化

  ディレクティブ `initialize` と `do not initialize` は、アプリケーションをどのように起動するかを制御します。これらのディレクティブを使用すると、アプリケーションは、起動時にグローバルシンボルを初期化したり、コードの一部をコピーしたりできます。イニシャライザは、たとえば、圧縮するなど、いくつかの方法で格納できます。448 ページの *initialize ディレクティブ*および 451 ページの *do not initialize ディレクティブ*を参照してください。

- 削除したセクションを保持する

  `keep` ディレクティブを使用すると、アプリケーションの残りの部分から参照されない場合でも、セクションが保持されます。つまり、これはアセンブラおよびコンパイラにおける *root* の概念と同じです。452 ページの *keep ディレクティブ*を参照してください。

- `use init table` ディレクティブ

define block ディレクティブ

構文

```
define [movable] block name
  [ with param, param... ]
{
  extended-selectors
}
[except
 {
   section_selectors
  }];
```

ここで、*param* は以下のいずれかです。

```
size = expr
maximum size = expr
alignment = expr
fixed order
static base [basename]
```

また、その他のディレクティブは、ブロックに含めるセクションを選択します（455 ページの *セクションの選択*を参照）。

パラメータ

name	定義するブロックの名前。
size	ブロックのサイズをカスタマイズします。デフォルトでは、ブロックのサイズは、その内容に依存するパーツの合計です。
maximum size	ブロックのサイズの上限を指定します。ブロックのセクションがこのサイズに合わない場合、エラーが発生します。
alignment	ブロックの最小アラインメントを指定します。ブロックの任意のセクションのアラインメントが、最小アラインメントを超える場合、そのアラインメントがブロックのアラインメントになります。
fixed order	セクションを固定順で配置します。指定されない場合、セクションは任意の順序で配置されます。
static base *[basename]*	*basename* という名前のスタティックベースが、特定のスタティックベースに応じて、ブロックの開始部分または中心に配置されるよう指定します。起動コードでは、スタティックベースを保持するレジスタが正しい値に初期化されるようにする必要があります。スタティックベースが 1 つしかない場合、名前は省略できます。

説明

block ディレクティブでは、空のセクションまたはその他のブロックのセットが含まれているメモリの連続エリアを定義します。内容のないブロックはスタックまたはヒープに空間を割り当てるために役立ちます。内容のあるブロックは、通常連続する必要のあるセクションといっしょにグループ化するために使用されます。

__section_begin、__section_end、または __section_size 演算子を使用して、アプリケーションからブロックの開始、終了、およびサイズにアクセスできます。指定の名前のブロックがなくても、その名前のセクションがある場合は、リンカでブロックは作成され、それにはそのセクションがすべて入ります。

movable ブロックは、リードオンリーおよびリード / ライトの位置に依存しない状態で使用します。ブロックを movable（移動可能）にすると、リンカでアプリケーションによるアドレスの使用を検証できるようになります。移動可能なブロックは他のブロックとまったく同じように配置しますが、移動可能ブロック内のシンボルを参照する際に適切な再配置が使用されるかどうかをリンカがチェックします。

例

```
/* ヒープ用に 0x1000 バイトのブロックを作る */
define block HEAP with size = 0x1000, alignment = 8 { };
```

関連項目

200 ページの *ツールとアプリケーション間の相互処理*。アクセスの例については、447 ページの *define overlay ディレクティブ*を参照してください。

define overlay ディレクティブ

構文

```
define overlay name [ with param, param... ]
{
  extended-selectors;
}
[except
 {
   section_selectors
  }];
```

拡張セレクタおよび except 句については、455 ページの *セクションの選択*を参照してください。

パラメータ

name	オーバレイの名前。
size	オーバレイのサイズをカスタマイズします。デフォルトでは、オーバレイのサイズはその内容に依存するパーツの合計です。

maximum size	オーバレイのサイズの上限を指定します。オーバレイのセクションがこのサイズに合わない場合、エラーが発生します。
alignment	オーバレイの最小アラインメントを指定します。オーバレイの任意のセクションのアラインメントが、最小アラインメントを超える場合、そのアラインメントがオーバレイのアラインメントになります。
fixed order	セクションを固定順で配置します。指定されない場合、セクションは任意の順序で配置されます。

説明

overlay ディレクティブは、セクションの指定セットを定義します。block ディレクティブとは対照的に、overlay ディレクティブは、同じ名前を複数回定義できます。各定義は、同じ名前の他のすべての定義と同じメモリ内の場所にまとめることができます。これにより、複数の独立したサブアプリケーションをもつアプリケーションで利用できる、*オーバレイ*メモリエリアが作成されます。

各サブアプリケーションイメージを ROM に配置し、すべてのサブアプリケーションを保持する RAM オーバレイエリアを予約します。サブアプリケーションを実行するには、まず ROM から RAM オーバレイにサブアプリケーションをコピーします。ILINK では、オーバレイ間での参照に関して生成される診断メッセージ以外で、相互依存するオーバレイ定義の管理に関する支援はないので注意してください。

オーバレイのサイズは、そのオーバレイ名で定義されている最大サイズと同じサイズになり、アラインメント要件は、アラインメント要件が最高の定義と同じになります。

注 : オーバレイされたセクションは、RAM および ROM パートに分割する必要があるため、必要なすべてのコピーに注意する必要があります。

関連項目

101 ページの *手動で初期化する*。

initialize ディレクティブ

構文

```
initialize { by copy | manually }
  [ with packing = algorithm ]
{
  section-selectors
}
[except
 {
   section_selectors
  }];
```

その他のディレクティブは、ブロックに含めるセクションを選択します。455 ページの *セクションの選択*を参照してください。

パラメータ

`by copy` セクションをイニシャライザおよび初期化データ用のセクションに分割し、アプリケーション起動時に自動的に初期化を行います。

`manually` セクションをイニシャライザおよび初期化データ用のセクションに分割します。アプリケーション起動時の初期化は、自動的には行われません。

`algorithm` イニシャライザを扱う方法を指定します。以下から選択します。

`none` – 選択したセクションコンテンツの圧縮を無効にします。これは、initialize manually のデフォルト手法です。

`zeros` – 値ゼロの連続するバイトを圧縮します。

`packbits` - PackBits アルゴリズムで圧縮します。この手法は、同じ値のバイトが多数連続するデータにおいて好結果が得られます。

`lz77` – Lempel-Ziv-77 アルゴリズムを使用して圧縮します。この方法ではより広範な入力が処理されますが、デコンプレッサのサイズはわずかに大きくなります。

`bwt` – Burrows-Wheeler アルゴリズムで圧縮します。この手法は、データのブロックを圧縮する前に変換することにより、`packbits` の手法を改善するものです。

`lzw` – Lempel-Ziv-Welch アルゴリズムで圧縮します。この手法では、辞書を使用してバイトパターンをデータに格納します。

`auto` – `smallest` に似ていますが、ILINK は `none`、`packbits`、`lz77` のいずれかを選択します。これは、`initialize by copy` のデフォルト手法です。

`smallest` – ILINK が各パッキング手法（`auto` は除く）を使用して結果のサイズを予測し、推定サイズが最小になるパッキング手法を選択します。ここには、デコンプレッサのサイズも含まれます。

説明

`initialize` ディレクティブは、初期化セクションを、イニシャライザを保持するセクションと初期化データを保持するセクションに分割します。初期化されたデータを保持するセクションには元のセクション名が保持され、イニ

シャライザを保持するセクション名のサフィックスは _init となります。起動時の初期化を自動的に処理するか (initialize by copy)、自分で処理するか (initialize manually) を選択できます。

パッキング手法として auto（initialize by copy のデフォルト）または smallest を使用する場合、ILINK はイニシャライザに適したパッキングアルゴリズムを自動的に選択します。これをオーバライドする場合は、別の packing 手法を選択してください。--log initialization オプションを使用すると、使用するパッケージングアルゴリズムを ILINK がどのように決定するかが示されます。

イニシャライザを圧縮すると、デコンプレッサが自動的にイメージに追加されます。bwt および lzw のデコンプレッサは、他の方式のデコンプレッサよりも実行時間が大幅に長く、RAM の使用量も増加します。bwt では約 9KB のスタックエリアが必要となり、lzw では 3.5KB が必要となります。

イニシャライザを圧縮する場合、圧縮後のイニシャライザの正確なサイズは、未圧縮データの正確な内容がわかるまで確定しません。このデータに他のアドレスが含まれ、これらのアドレスの一部が圧縮後のイニシャライザのサイズに依存する場合には、リンカでエラー Lp017 が発生します。この問題を回避するには、圧縮後のイニシャライザを最後に配置するか、アドレスが既知である必要のないセクションと一緒にメモリ領域に配置します。

initialize manually を使用しない限り、ILINK は初期化テーブルをインクルードすることによってシステム起動時に初期化が行われるようにします。起動コードは、このテーブルを読み込む初期化ルーチンを呼出して、必要なイニシャライザを実行します。

ゼロで初期化するセクションは、initialize ディレクティブの影響を受けません。

通常 initialize ディレクティブは初期化済みの変数に使用されますが、実行可能コードを低速の ROM から高速の RAM にコピーしたり、オーバレイなど任意のセクションをコピーするときにも使用できます。他の例については、447 ページの *define overlay ディレクティブ*を参照してください。

初期化に必要なセクションは、initialize by copy ディレクティブの影響を受けません。このようなセクションとして、__low_level_init 関数およびこの関数が参照する部分のすべてが含まれます。

プログラムエントリラベルから到達可能な部分はすべて*初期化が必要*と考えられます。ただし、__iar_init$$done で始まるラベルを持つセクションフラグメントによって到達される部分は除きます。--log sections オプションは、アプリケーションにインクルードされるセクションフラグメントの作成を記録するほかに、どのセクションが初期化に必要かを決定するプロセスも記録します。

例

```
/* 全ての read-write セクションをプログラムの開始時に自動的に ROM から RAM
   にコピー */
initialize by copy { rw };
place in RAM { rw };
place in ROM { ro };
```

関連項目

84 ページの *システム起動時の初期化*、451 ページの *do not initialize ディレクティブ*。

do not initialize ディレクティブ

構文

```
do not initialize
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、455 ページの *セクションの選択*を参照してください。

説明

do not initialize ディレクティブは、システム起動コードで初期化しないセクションを指定します。このディレクティブを使用できるのは、zeroinit セクションのみです。

コンパイラのキーワード __no_init は、変数を do not initialize ディレクティブによって処理されなければならないセクションに配置します。

例

```
/* プログラムのスタートで、_noinit で終了する read-write
セクションは初期化しない */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

関連項目

84 ページの *システム起動時の初期化*、448 ページの *initialize ディレクティブ*。

keep ディレクティブ

構文

```
keep
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、455 ページの *セクションの選択*を参照してください。

説明

keep ディレクティブは、選択したセクションが参照されない場合であっても、すべての選択したセクションが実行可能イメージを保持するように指定します。

例

```
keep { section .keep* } except {section .keep};
```

place at ディレクティブ

構文

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
           end of region_expr }
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、455 ページの *セクションの選択*を参照してください。

パラメータ

memory: *expr*	特定メモリ内の特定アドレスです。アドレスは、define memory ディレクティブで定義されている供給メモリで使用できなければなりません。メモリが 1 つだけの場合、メモリ指定子はオプションです。
start of *region_expr*	単一の内部領域になる領域式。間隔の開始位置が使用されます。

end of *region_expr*	単一の内部領域になる領域式。間隔の終了位置が使用されます。

説明

place at ディレクティブは、セクションおよびブロックを、特定のアドレス、あるいは領域の開始アドレスまたは終了アドレスのいずれかに配置します。2 つの異なる place at ディレクティブに対して、同一のアドレスを使用することはできません。また、空の領域を place at ディレクティブで使用することもできません。領域に配置される場合、セクションおよびブロックは、place in ディレクティブで同じ領域に配置される他の任意のセクションまたはブロックの前に配置されます。

name が指定されている場合、このディレクティブがマップファイルおよび一部のログメッセージで使用されます。

例

```
/* read-only セクションである .startup を コード領域の最初に配置する */
"START": place at start of ROM { readonly section .startup };
```

関連項目

453 ページの *place in ディレクティブ*。

place in ディレクティブ

構文

```
[ "name": ]
place in region-expr
{
  extended-selectors
}
 [except{
   section-selectors
 }];
```

ここで、*region-expr* は領域式です（442 ページの *領域* を参照）。

また、その他のディレクティブは、ブロックに含めるセクションを選択します。455 ページの *セクションの選択* を参照してください。

説明

place in ディレクティブは、セクションおよびブロックを特定のブロックに配置します。セクションおよびブロックは、任意の順序で領域に配置されます。

特定の順序を指定するには、block ディレクティブを使用します。領域では、複数の範囲を使用できます。

name が指定されている場合、このディレクティブがマップファイルおよび一部のログメッセージで使用されます。

例

```
/* コード領域に read-only セクションを配置する */
"ROM": place in ROM { readonly };
```

関連項目

452 ページの *place at ディレクティブ*。

use init table ディレクティブ

構文

```
use init table name for
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、455 ページの *セクションの選択*を参照してください。

パラメータ

name	init テーブル名。

説明

すべての初期化エントリは通常、1 つの初期化テーブル (Table) にまとめて生成されます。このディレクティブを使用して、一部のエントリが別のテーブルに配置されるようにします。この初期化テーブルは別のときに使用したり、通常の初期化テーブルではない異なる状況で使用することができます。

use init table ディレクティブで言及されていないすべての変数の初期化エントリは、通常の初期化テーブルに入れられます。複数の use init table ディレクティブを使用すると、複数の初期化テーブルを持つことができます。

init テーブルの開始、終了、サイズは "Region$$*name*" の __section_begin、__section_end、__section_size をそれぞれ使用するか、シンボル Region$$*name*$$Base、Region$$*name*$$Limit、Region$$*name*$$Length を使用して、アプリケーションプログラムでアクセスすることができます。

例

```
use init table Core2 for { section *.core2};

/* __section_begin("Region$$Core2") を使用して
   Core2 init テーブルの開始を取得できます。*/
```

セクションの選択

*セクションの選択*の目的は、ILINK ディレクティブが適用されるセクションを指定することです（*section-selector* および *except 句*を使用）。1 つ以上のセクションセレクタに一致するセクションがすべて選択され、except 句が指定されている場合、この句のセレクタのセクションは選択されません。各セクションセレクタは、セクション属性、セクション名、オブジェクト名またはライブラリ名が一致するセクションを選択します。

ディレクティブによっては、セクションおよびブロックの両方に適用できるなど、さらに詳細な選択が必要になることもあります。この場合、*拡張セレクタ*が使用されます。

セクションセレクタ

構文

```
{ [ section-selector ][, section-selector...] }
```

ここで、*section-selector* には以下のように指定します。

```
[ section-attribute ][ section-type ][ section sectionname ]
   [object {module | filename } ]
```

ここで、*section-attribute* には以下のように指定します。

```
[ ro [ code | data ] | rw [ code | data ] | zi ]
```

ここで、ro、rw、zi は、それぞれ readonly、readwrite、zeroinit とすることもできます。

section-type は以下のように指定します。

```
[ preinit_array | init_array ]
```

パラメータ

ro または readonly	リードオンリーセクション。
rw または readwrite	リード / ライトセクション。
zi または zeroinit	ゼロで初期化するセクション。これらのセクションは内容がなく、またシステムのセットアップ中にゼロで初期化される場合があります
code	コードを含むセクション。
data	データを含むセクション。
preinit_array	ELF セクションタイプ SHT_PREINIT_ARRAY のセクション。

init_array	ELF セクションタイプ SHT_INIT_ARRAY のセクション。
sectionname	セクション名。以下の 2 つのワイルドカードを使用できます。 ? は、任意の 1 文字と一致します。 * は、ゼロ以上の文字と一致します。
module	*objectname*(*libraryname*) 形式の名前。オブジェクト名とライブラリ名がそれぞれのパターンに一致するオブジェクトモジュールのセクションが選択されます。空のライブラリ名パターンでは、オブジェクトファイルのセクションのみが選択されます。
filename	オブジェクトファイル、ライブラリ、ライブラリ内のオブジェクトの名前。以下の 2 つのワイルドカードを使用できます。 ? は、任意の 1 文字と一致します。 * は、ゼロ以上の文字と一致します。

説明

セクションセレクタは、*オブジェクト*（オブジェクトファイル、ライブラリ、ライブラリのオブジェクト）のセクション属性、セクションタイプ、セクション名、オブジェクトの名前（オブジェクトファイル名・ライブラリ名・ライブラリに含まれるオブジェクト名）と一致するすべてのセクションを選択します。4 つの条件のうち 3 つまでが省略可能です。セクション属性が省略された場合、セクション属性に関係なく任意のセクションが選択されます。セクションタイプを省略すると、任意のタイプのセクションが選択されます。

セクション名またはオブジェクト名の一部が省略された場合、セクション名やオブジェクト名の制限なしにセクションが選択されます。

セクションセレクタを使用せずに { } のみを使用することもできます。これは、ブロックを定義する場合に便利です。

スコープの狭いセクションセレクタは、より汎用的なセクションセレクタよりも優先順位が高くなります。

複数のセクションセレクタが同じ目的に一致する場合、いずれか 1 つがより具体的でなければなりません。セクションセレクタは以下の場合により具体的となります。

- 他のものと異なり、セクションタイプを指定している
- 他のものと異なり、ワイルドカードを使用せずにセクション名またはオブジェクト名を指定している

- 他のセレクタに一致するセクションがこのセレクタにも一致しているが、その逆が真でない。

セレクタ 1	セレクタ 2	より具体的なセレクタ
section "foo*"	section "f*"	セレクタ 1
section "*x"	section "f*"	どちらも不適格
ro code section "f*"	ro section "f*"	セレクタ 1
init_array	ro section "xx"	セレクタ 1
section ".intvec"	ro section ".int"	セレクタ 1
section ".intvec"	object "foo.o"	どちらも不適格

表 41: セクションセレクタの指定の例

例

```
{ rw }                          /* 全ての read-write セクションを選択する */

{ section .mydata* }            /* .mydata*  セクションのみを選択 */
/* オブジェクトファイル special.o の中の .mydata* セクションを選択する */
{ section .mydata* object special.o }
```

lib.a という名前のライブラリ内に foo.o というオブジェクトがあって、そのオブジェクト内にセクションがある場合、以下のセレクタのいずれかがによってそのセクションが選択されます。

```
object foo.o(lib.a)
object f*{lib*}
object foo.o
object lib.a
```

関連項目

448 ページの *initialize ディレクティブ*、451 ページの *do not initialize ディレクティブ*、452 ページの *keep ディレクティブ*。

拡張セレクタ

構文

{ **[** *extended-selector* **][**, *extended-selector*...**]** }

ここで、*extended-selector* には以下のように指定します。

[first **|** last **|** **midway** **]**
{ *セクションセレクタ* **|**
block *name* **[** *inline-block-def* **]|**
overlay *name* **}**

ここで、*inline-block-def* には以下のように指定します。

[block-params **]** *extended-selectors*

パラメータ

first	選択したセクションやブロック、オーバレイを、それらを含む配置ディレクティブ、ブロック、オーバレイの最初に配置します。
last	選択したセクションやブロック、オーバレイを、それらを含む配置ディレクティブ、ブロック、オーバレイの最後に配置します。
midway	選択したセクション、ブロック、オーバレイを、ブロックの端から、それらを含むブロックの最大サイズの半分よりも遠くならないように配置します。このパラメータは、最大サイズを持つブロック内でしか使用できない点に注意してください。
name	ブロックまたはオーバレイの名前。

説明

配置ディレクティブまたはオーバレイに含める内容を選択するには、extended-selectors を使用します。セクション選択パターンの使用に加えて、含めるブロックやオーバレイを明示的に指定することもできます。

first または last キーワードを使用して、それらを含む配置ディレクティブの最初もしくは最後に配置するパターンやブロック、オーバレイを指定することができます。配置する順序をより正確に制御する必要がある場合は、固定の順序を持つブロックを使用できます。

ブロックは、define block ディレクティブやインラインを extended-selector の一部として使用すれば、個別に定義することができます。

midway パラメータは、主に正と負の両方のオフセットを持つことができるスタティックベースとともに使用すると便利です。

例

```
define block First { ro section .f* }; /* ".f*" に一致するすべての
                                          リードオンリーセクションを */
                                          持つブロックを定義 */
define block Table { first block First, ro section .b };
                                        /* ".b*" に一致する
                                           セクションの前に
                                           くるブロックを
                                           定義 */
```

または、個別の define block ディレクティブ内ではなく、ブロック First をインラインで定義することもできます。

```
define block Table { first block First { ro section .f* },
                     ro section .b* };
```

関連項目　446 ページの *define block ディレクティブ*、447 ページの *define overlay ディレクティブ*、452 ページの *place at ディレクティブ*。

シンボル、式、数値の使用

リンカ設定ファイルでは、以下のことが可能です。

- シンボルを定義およびエクスポートする

  `define symbol` ディレクティブは、設定ファイル内の式に使用可能な指定値を持つシンボルを定義します。また、シンボルは、エクスポートしてアプリケーションやデバッガでも使用できます。460 ページの *define symbol ディレクティブ*、460 ページの *export ディレクティブ*を参照。
- 式および数値を使用する

  リンカ設定ファイルでは、式および数値は、アドレス、サイズなどの指定に使用されます（461 ページの*式*を参照）。

check that ディレクティブ

構文

```
check that expression;
```

パラメータ

expression	ブール値式。

説明

`check that` ディレクティブを使用して、スタック使用量解析の結果をブロックおよび領域のサイズと比較できます。式がゼロに評価される場合、エラーが出力されます。

`check that` 式でのみ、追加の演算子を 3 つ使用できます。

`maxstack`（*カテゴリ*）	カテゴリ内の任意のコールグラフルート関数で最も深いコールチェーンのスタックの深さ。
`totalstack`（*カテゴリ*）	カテゴリ内のそれぞれのコールグラフルート関数で最も深いコールチェーンのスタックの深さ合計。
`size`(*block*)	ブロックのサイズ。

例

```
check that maxstack("Program entry")
             + totalstack("interrupt")
             + 1K
           <= size(block CSTACK);
```

関連項目　87 ページの*スタック使用量解析*。

define symbol ディレクティブ

構文

```
define [ exported ] symbol name = expr;
```

パラメータ

`exported`	実行可能イメージが利用できるシンボルをエクスポートします。
name	シンボルの名前。
expr	シンボルの値。

説明

`define symbol` ディレクティブは、指定値でシンボルを定義します。シンボルは設定ファイル内の式でも使用できます。この方法で定義したシンボルは、設定ファイル外でオプション `--config_def` で定義されたシンボルと同様に機能します。

このディレクティブの派生形 `define exported symbol` は、ディレクティブ `define symbol` を `export symbol` ディレクティブと組み合わせて使用するためのショートカットです。この場合、コマンドラインでは、`--config_def` オプションと `--define_symbol` オプションの両方が同一の効果を達成することが必要とされます。

注：

- シンボルを再定義することはできません
- *_X* プレフィクスの付いたシンボル（*X* は大文字）、または __（下線 2 本）を含むシンボルは、ツールセットベンダの予約語です

例

```
/* シンボル my_symbol を 4 と定義 */
define symbol my_symbol = 4;
```

関連項目

460 ページの *export ディレクティブ*、105 ページの *ILINK とアプリケーション間の相互処理*。

export ディレクティブ

構文

```
export symbol name;
```

パラメータ

name	シンボルの名前。

説明

export ディレクティブは、エクスポートするシンボルを定義し、これを実行可能イメージおよびグローバルラベルから使用できるようにします。アプリケーションまたはデバッガは、これを設定目的などのために参照できます。

例

```
/* シンボル my_symbol をエクスポート */
export symbol my_symbol;
```

式

構文

式は、以下の要素で構成されます。

```
expression binop expression
unop expression
expression ? expression : expression
(expression)
number
symbol
func-operator
```

ここで、*binop* は、以下のいずれかのバイナリ演算子です。

+, -, *, /, %, <<, >>, <, >, ==, !=, &, ^, |, &&, ||

unop は、以下のいずれかの単項演算子です。

+, -, !, ~

number は数値です（詳細は 462 ページの*数値*を参照）。

symbol は、定義済みシンボルです。詳細については、460 ページの *define symbol ディレクティブ*および 299 ページの *--config_def* を参照してください。

func-operator は、以下の関数のような演算子のいずれかです。

minimum(*expr*,*expr*)	2 つのパラメータの小さい方を返します。
maximum(*expr*,*expr*)	2 つのパラメータの大きい方を返します。
isempty(*r*)	領域が空の場合は True、そうでない場合は False を返します。
isdefinedsymbol(*expr-symbol*)	式シンボルが定義されている場合は True、そうでない場合は False を返します。
start(*r*)	領域の最下位アドレスを返します。
end(*r*)	領域の最上位アドレスを返します。
size(*r*)	完全な領域のサイズを返します。

ここで、*expr* は式であり、*r* は領域式です（443 ページの*領域式*を参照）。

説明

リンカ設定ファイルでは、式は、範囲 -2^64 ～ 2^64 の 65 ビット値です。式の構文は、いくつかの例外はありますが、C 構文に従っています。代入やキャスト、事前 / 事後処理、アドレス処理 (*、&、[]、->、.) はありません。領域の式から値を抽出する処理など、いくつかの処理では、関数呼出しに似た構文が使用されます。ブール値式は、0（偽）または 1（真）を返します。

数値

構文

nr [*nr-suffix*]

ここで、nr は、10 進数または 16 進数（0x... または 0X...）のいずれかです。

また、*nr-suffix* は以下のいずれかです。

```
K           /* Kilo = (1 << 10) 1024 */
M           /* Mega = (1 << 20) 1048576 */
G           /* Giga = (1 << 30) 1073741824 */
T           /* Tera = (1 << 40) 1099511627776 */
P           /* Peta = (1 << 50) 1125899906842624 */
```

説明

数値は、通常の C 構文で、または便利なサフィックスのセットを使用して表現できます。

例

1024 は、0x400 と同じです。これは、1K と同じです。

構造化設定

構造化ディレクティブを使用すると、以下のように、リンカ設定ファイル内で構造を作成できます。

- 条件付きインクルード

  if ディレクティブは、条件に基づいて他のディレクティブを含めるまたは除外します。これにより、同じファイルでいくつかの異なるメモリ設定を行うことができます。463 ページの *if ディレクティブ*を参照してください。
- リンカ設定ファイルをいくつかの異なるファイルに分割する

  include ディレクティブを使用すると、設定ファイルをいくつかの論理的に異なるファイルに分割できます。464 ページの *include ディレクティブ*を参照してください。
- サポートされていないケースのエラーについて警告

error ディレクティブ

構文

```
error string
```

パラメータ

string	エラーメッセージ。

説明

error ディレクティブを使用して、条件付きディレクティブのアクティブな部分でディレクティブが発生する場合にエラーを発します。

例

```
エラー " サポートされていない設定 "
```

if ディレクティブ

構文

```
if (expr) {
  directives
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

ここで、*expr* は式です（461 ページの*式*を参照）。

パラメータ

directives	任意の ILINK ディレクティブ。

説明

if ディレクティブは、条件に基づいて他のディレクティブを含めるまたは除外します。これにより、同じファイルで、たとえば、バンクおよび非バンクメモリの両方でいくつかの異なるメモリ設定を行うことができます。

if パート、else if パート、else パート内のディレクティブは、条件式の評価が真か偽かに関係なく、構文がチェックされます。ただし、条件式が真の場合のパートのディレクティブ、またはいずれの条件も真でない場合の else パートのディレクティブは、if ディレクティブ外には影響を与えません。if ディレクティブはネストできます。

例

444 ページの*空の領域*を参照してください。

include ディレクティブ

構文

```
include "filename";
```

パラメータ

filename	/ と ¥ の両方をディレクトリの区切り文字として使用できるパス。

説明

include ディレクティブによって、設定ファイルをそれぞれ個別のファイルに入った論理的に異なるいくつかの部分に分割できるようになります。たとえば、頻繁に変更する必要があるファイルや、編集の必要があまりないファイルなどに分割できます。

通常は、リンカはシステム設定ディレクトリ内の設定インクルードファイルを検索します。--config_search リンカオプションを使用して、検索するディレクトリを追加することができます。

関連項目

299 ページの *--config_search*。

セクションリファレンス

- セクションリファレンス
- セクションおよびブロックの説明

セクションの詳細については、「78 ページの *モジュールおよびセクション*」を参照してください。

セクションの概要

以下の表は、IAR ビルドツールで使用される ELF セクションおよびブロックのリストです。

セクション	説明
.bss	ゼロに初期化される静的 / グローバル変数を保持します。
CSTACK	C/C++ プログラムが使用するスタックを保持します。
.data	初期化される静的 / グローバル変数を保持します。
.data_init	リンカディレクティブ initialize の使用時に、.data セクションの初期値を保持。
__DLIB_PERTHREAD	DLIB モジュールの静的状態を含む変数を保持します。
.exc.text	例外に関連するコードを保持します。
HEAP	動的割当てデータに使用するヒープを保持します。
.iar.dynexit	atexit テーブルを保持します。
.init_array	動的初期化関数のテーブルを保持します。
.intvec	リセットベクタテーブルを保持します。
IRQ_STACK	割込み要求、IRQ、例外のスタックを保持します。
.noinit	静的変数およびグローバル変数 __no_init を保持します。
.preinit_array	動的初期化関数のテーブルを保持します。
.prepreinit_array	動的初期化関数のテーブルを保持します。
.rodata	定数データを保持します。
.text	プログラムコードを保持します。
.textrw	__ramfunc により宣言されたプログラムコードを保持します。

表 42: セクションの概要

セクション	説明
.textrw_init	.textrw で宣言されたセクションのイニシャライザを保持します。

表 42: セクションの概要（続き）

アプリケーションで使用する ELF セクションのほかに、ツールではさまざまな目的で多数の ELF セクションを使用します。

- .debug で始まるセクションは一般的に、DWARF フォーマットのデバッグ情報を含みます
- .iar.debug で始まるセクションには、デバッグの補足情報が IAR フォーマットで含まれます
- セクション .comment は、ファイルのビルドに使用されるツールおよびコマンドラインが含まれます
- .rel または .rela で始まるセクションには、ELF の再配置情報が含まれます
- セクション .symtab には、ファイルのシンボルテーブルが含まれます
- セクション .strtab には、シンボルテーブルのシンボル名が含まれます
- セクション .shstrtab には、セクション名が含まれます

セクションおよびブロックの説明

ここでは、各セグメントのリファレンス情報を説明します。

- *説明*は、セクションが保持する内容のタイプ、また必要に応じて、セクションがリンカによりどのように扱われるかを記述します
- *メモリ配置*は、メモリ配置制限を記述します

リンカ設定ファイルを修正することによるメモリでのセクションの割当て方法については、*「81 ページのコードおよびデータの配置（リンカ設定ファイル）」*を参照してください。

.bss

説明 ゼロに初期化される静的 / グローバル変数を保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

CSTACK

説明 内部データスタックを保持するブロックです。

メモリ配置 このブロックは、メモリ内の任意の場所に配置できます。

関連項目 100 ページの *スタックメモリの設定*。

.data

説明 初期化される静的 / グローバル変数を保持します。オブジェクトファイルでは、これに初期値が含まれます。リンカディレクティブ `initialize` を使用する際、対応する `.data_init` セクションが、それぞれの `.data` セクションに作成され、圧縮された初期値が保持されます。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

.data_init

説明 `.data` セクションの圧縮された初期値を保持します。このセクションは、`initialize` リンカディレクティブが使用された場合に、リンカによって作成されます。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

__DLIB_PERTHREAD

説明 メインのスレッドによって使用される、スレッドのローカルの静的変数およびグローバル初期化済み変数を保持します。

このセクションは自動的に配置されます。配置を変更する場合、初期化を変更しないでください。このセクションの初期化は、`initialize` ディレクティブによって制御されています。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 143 ページの *マルチスレッド環境の管理*。

.exc.text

説明 アプリケーションが例外を処理するときにだけ実行されるコードを保持します。

メモリ配置 .text と同じメモリ内。

関連項目 188 ページの *例外処理*。

HEAP

説明 メモリ内で動的に割り当てられたデータに使用されるヒープ、つまり malloc と free、さらに C++ では new と delete によって割り当てられるデータを保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 100 ページの *ヒープメモリの設定*。

.iar.dynexit

説明 終了時に行われる呼出しのテーブルを保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 100 ページの *atexit 制限の設定*。

.init_array

説明 同じ静的記憶寿命を持つ 1 つ以上の C++ オブジェクトの初期化で呼出すルーチンへのポインタを保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

.intvec

説明 cstartup、割込みサービスルーチンなどへの分岐命令を含む、リセットベクタおよび例外ベクタを保持します。

メモリ配置 アドレス範囲 0x00 〜 0x3F に配置する必要があります。

IRQ_STACK

説明 IRQ 例外の提供時に使用されるスタックを保持します。他の例外タイプに使用するため、FIQ、SVC、ABT、UND など他のスタックを追加できます。cstartup.s ファイルは、使用される例外スタックポインタを初期化するように修正する必要があります。

注 : このセクションは、Cortex-M 用のコンパイルには使用されません。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 198 ページの *例外スタック*。

.noinit

説明 静的 / グローバル変数 __no_init を保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

.preinit_array

説明 .init_array と似ていますが、他より先に C++ 初期化を実行するためにライブラリにより使用されます。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 468 ページの *.init_array*。

.prepreinit_array

説明 .init_array と似ていますが、C 静的初期化が動的初期化として書き直されるときに使用されます。すべての C++ 動的初期化の前に実行されます。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 468 ページの *.init_array*。

.rodata

説明 定数データを保持します。これには、定数変数、文字列、集合リテラルなどをインクルードできます。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

.text

説明 システム初期化用のコードを含むプログラムコードを保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

.textrw

説明 `__ramfunc` により宣言されたプログラムコードを保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 351 ページの *__ramfunc*。

.textrw_init

説明 `.textrw` で宣言されたセクションのイニシャライザを保持します。

メモリ配置 このセクションは、メモリ内の任意の場所に配置できます。

関連項目 351 ページの *__ramfunc*。

スタック使用解析制御ファイル

- 概要
- スタック使用解析制御ディレクティブ
- 構文の構成要素

この章を読む前に、87 ページの*スタック使用量解析*に目を通してください。

概要

スタック使用解析制御ファイルは、スタック使用量解析を制御する一連のディレクティブから構成されます。これらのファイルでは、C ("/*...*/") および C++ ("//...") のコメントを使用できます。

スタック使用解析制御ファイルのデフォルトのファイル名拡張子は、suc です。

C++ 名

スタック使用量制御ファイルで C++ 関数名を指定する際、リンカで使用されるものと同じ名前を使用する必要があります。パラメータの数と名前、型名が一致しなければなりません。ただし、重要でない空白文字の違いは認められます。特に、すべての C++ 関数名には識別子以外の文字が含まれるため、引用符で名前を囲む必要があります。

関数名にはワイルドカードも使用できます。"#*" はあらゆる文字のシーケンスに一致し、"#?" は 1 文字に一致します。これによって、インスタンス化されたあらゆるテンプレート関数の関数名を記述できるようになります。

例 :

```
"operator new(unsigned int)"
"ostream::flush()"                        // EC++
"std::ostream::flush()"                   // C++
"operator <<(ostream &, char const *)"
"void _Sort<#*>(#*, #*, #*)"
```

スタック使用解析制御ディレクティブ

ここでは、スタック使用解析制御ディレクティブの各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。

call graph root ディレクティブ

構文

```
call graph root [ category ] : function-spec [, function-spec... ];
```

パラメータ

category	475 ページの *category* を参照してください
function-spec	475 ページの *function-spec* を参照してください

説明

リストされた関数がコールグラフルートであるように指定します。オプションで、コールグラフルートのカテゴリを指定できます。コールグラフルートは、リンカマップファイルの*スタック使用*の章のカテゴリ下にリストされています。

リンカは通常、アプリケーションで必要な関数で、コールグラフルートでなく、呼出される様子がないものに対してワーニングを発行します。

例

```
call graph root [task]: fun1, fun2;
```

関連項目

361 ページの *call_graph_root*。

exclude ディレクティブ

構文

```
exclude function-spec [, function-spec... ];
```

パラメータ

function-spec	475 ページの *function-spec* を参照してください

説明

指定した関数およびそれから発生する呼出しツリーをスタック使用の計算から除外します。

例

```
exclude fun1, fun2;
```

function ディレクティブ

構文

```
[ override ] function [ category ] function-spec : stack-size
[ , call-info... ];
```

パラメータ

category	475 ページの *category* を参照してください
function-spec	475 ページの *function-spec* を参照してください
call-info	476 ページの *call-info* を参照してください
stack-size	477 ページの *stack-size* を参照してください

説明

関数における最大スタック使用と、その関数から呼出される他の関数を指定します。

関数にスタック使用の情報がすでにあれば通常はエラーは発生しませんが、override で開始するとエラーは非表示になり、ディレクティブに提供された情報が以前の情報の代わりに使用されます。

例

```
function MyFunc1: 32,
  calls MyFunc2,
  calls MyFunc3, MyFunc4: 16;

function [interrupt] nmi: 44
```

max recursion depth ディレクティブ

構文

```
max recursion depth function-spec : size;
```

パラメータ

function-spec	475 ページの *function-spec* を参照してください
size	477 ページの *size* を参照してください

説明

関数がメンバである再帰ネストにおいて、任意のサイクルでの繰り返し回数の最大値を指定します。

再帰ネストはコールグラフのサイクルのセットで、各サイクルは少なくとも 1 つのノードをネスト内の別のサイクルと共有しています。

スタック使用量解析の結果は、最大再帰深度にネスト内で最も深いサイクルのスタック使用量を積算したものに基づきます。最も深いサイクルのいずれ

かのポイントにネストが入力されない場合、その呼出しについてスタック使用量の結果は計算されません。

例

```
max recursion depth fun1: 10;
```

no calls from ディレクティブ

構文

```
no calls from module-spec to function-spec [, function-spec... ];
```

パラメータ

function-spec	475 ページの *function-spec* を参照してください
module-spec	475 ページの *module-spec* を参照してください

説明

スタック使用量情報を持たないモジュール内の関数にスタック使用量情報を提供する場合、そのモジュールから参照されていても、呼出されるものとしてリストされていない関数について警告します。これは主に、ユーザが制御できない、C のランタイムルーチン（コンパイラにより生成されるその呼出し）に関する問題を回避するためです。

これらの関数への呼出しが実際にない場合は、`no calls from` ディレクティブを使用して、指定した関数のワーニングを選択して非表示にします。また、ワーニングを完全に無効化（`--diag_suppress` または **[プロジェクト] > [オプション] > [リンカ] > [診断] > [診断を無効化]**）することができます。

例

```
no calls from [file.o] to fun1, fun2;
```

possible calls ディレクティブ

構文

```
possible calls calling-func : called-func [ , called-func... ];
```

パラメータ

calling-func	475 ページの *function-spec* を参照してください
called-func	475 ページの *function-spec* を参照してください

説明

1 つの関数における間接的なすべての呼出しの宛先の完全なリストを指定します。これは、間接的な呼出しを実行することがわかっている関数で、この特定のアプリケーションでどの関数が呼出されるか正確にわかっているときに使用します。コンパイル時にどの関数が呼出されるか情報が入手可能な場合は、`#pragma calls` ディレクティブの使用を考慮してください。

例 `possible calls afun: bfun, cfun;`

関連項目 361 ページの *calls*。

構文の構成要素

スタック使用制御ディレクティブは、構文の構成要素をいくつか使用します。これについて以下に説明します。

category

構文 `[ name ]`

説明 call graph root のカテゴリ。任意の名前を使用できます。カテゴリでは大文字と小文字は区別されません。

例 カテゴリの例 :

```
[interrupt]
[task]
```

function-spec

構文 **[** *?* **]** *name* **[** *module-spec* **]**

説明 シンボル名、module-local シンボルの場合はそれが定義されているモジュール名を指定します。通常は、関数の仕様がプログラムのシンボルに一致しない場合、ワーニングが出力されます。先頭に ? を付けると、このワーニングは非表示になります。

例 *function-spec* の例 :

```
xFun
MyFun [file.o]
?"fun1(int)"
```

module-spec

構文 `[name [ (name) ]]`

説明

モジュール名、またオプションでモジュールが属するライブラリ名を括弧内で指定します。同じ名前のモジュールを区別するために、以下を指定できます。

- ファイルの完全なパス ("D:¥C1¥test¥file.o")
- パス末尾に必要なだけのパスのエレメント ("test¥file.o")
- パス先頭にパスエレメント、その後に "..."、末尾にパスエレメント ("D:¥...¥file.o")

複数ファイルのコンパイル機能 (--mfc) を使用する際は、複数のファイルが 1 つのモジュールにコンパイルされ、最初のファイルにちなんだ名前が付きます。

例

module-spec の例 :

```
[file.o]
[file.o(lib.a)]
["D:¥C1¥test¥file.o"]
```

name

説明

name は識別子または引用符で囲まれた文字列のどちらかを使用できます。

識別子の最初の文字は、文字または "_"、"$"、または "." のいずれかでなければなりません。残りの文字には数字も使用できます。

引用符で囲まれた文字列の先頭と末尾は " で、すべての文字を含めることができます。2 つの連続した " 文字を引用符で囲まれた文字列内で使用して、1 つの " を表すことができます。

例

name の例 :

```
MyFun
file.o
"file-1.o"
```

call-info

構文

```
calls function-spec [ , function-spec...][ : stack-size ]
```

説明

1 つ以上の呼出された関数のほか、オプションで呼出しのスタックサイズを指定します。

例　*call-info* の例：

```
calls MyFunc1 : stack 16
calls MyFunc2, MyFunc3, MyFunc4
```

stack-size

構文　`[ stack ]` *size*

説明　スタックフレームのサイズを指定します。

例　*stack-size* の例：

```
24
stack 28
```

size

説明　10 進の整数、または 0x とその後に続く 16 進の整数。どちらを使用する場合も、オプションで 2 の累乗 (K=2^{10}、M=2^{20}、G=2^{30}、T=2^{40}、P=2^{50}) を示すサフィックスを付けることができます。

例　*size* の例：

```
24
0x18
2048
2K
```

IAR ユーティリティ

- IARアーカイブツール — iarchive — 複数のELFオブジェクトファイルで構成するライブラリ（アーカイブ）の作成および操作を行います。
- IAR ELF ツール — ielftool — ELF 実行可能イメージ上でさまざまな変換（フィル、チェックサム、フォーマット変換など）を実行します。
- IAR ELF Dumper — ielfdump — ELF 再配置可能イメージまたは実行可能イメージの内容のテキスト表示を作成します。
- IAR ELFオブジェクトツール — iobjmanip — ELFオブジェクトファイルの低レベルの操作に使用します。
- IAR Absolute Symbol Exporter — isymexport — ROM イメージファイルから絶対シンボルをエクスポートします。これらのシンボルは、アドオンアプリケーションのリンク時に使用できます。

IAR アーカイブツール — iarchive

IAR アーカイブツール (`iarchive`) は、複数の ELF オブジェクトファイルから 1 つのライブラリ（アーカイブ）を作成できます。また、`iarchaive` は、ELF ライブラリの操作にも使用できます。

ライブラリファイルには、いくつかの再配置可能 ELF オブジェクトモジュールが含まれており、それぞれ個別にリンカで使用できます。リンカに直接指定されるオブジェクトモジュールとは対照的に、ライブラリの各モジュールは、必要な場合のみ追加されます。

IDE でライブラリをビルドする方法については、*『ARM 用 IDE プロジェクト管理およびビルドガイド』* を参照してください。

呼出し構文

アーカイブビルダの呼出し構文は以下のとおりです。

`iarchive` *パラメータ*

パラメータ

パラメータを以下に示します。

パラメータ	説明
command	実行する操作を定義するコマンドラインオプションです。このようなオプションは、ライブラリファイル名より前に指定する必要があります。
libraryfile	操作対象のライブラリファイルです。
objectfile1 ... objectfileN	指定されたコマンドの操作対象のオブジェクトファイルです。
options	実行する動作を定義するコマンドラインオプションです。これらのオプションは、コマンドラインの任意の場所に配置できます。

表 43: iarchive パラメータ

例

以下の例では、ソースオブジェクトファイル module1.o、module.2.o、module3.o から mylibrary.a という名前のライブラリファイルを作成します。

```
iarchive mylibrary.a module1.o module2.o module3.o.
```

以下の例では、mylibrary.a の内容がリストされます。

```
iarchive --toc mylibrary.a
```

以下の例では、ライブラリ内の module3.o を module3.o ファイルの内容と置き換え、module4.o を mylibrary.a の後に追加します。

```
iarchive --replace mylibrary.a module3.o module4.o
```

IARCHIVE コマンドの概要

以下の表に、iarchive コマンドの概要を示します。

コマンドラインオプション	説明
--create	リストされたオブジェクトファイルを含むライブラリを作成します。
--delete, -d	リストされたオブジェクトファイルをライブラリから削除します。
--extract, -x	リストされたオブジェクトファイルをライブラリから抽出します。

表 44: iarchive コマンドの概要

コマンドラインオプション	説明
--replace, -r	リストされたオブジェクトファイルにより、ライブラリでの置換または追加を行います。
--symbols	ライブラリ内のファイルによって定義されているシンボルをすべてリストします。
--toc, -t	ライブラリ内のファイルすべてリストします。

表 44: iarchive コマンドの概要 （続き）

詳細については、494 ページの*オプションの説明*を参照してください。

IARCHIVE オプションの概要

以下の表に、iarchive オプションの概要を示します。

コマンドラインオプション	説明
-f	コマンドラインを拡張します。
--output, -o	ライブラリファイルを指定。
--silent	出力抑止操作を設定します。
--verbose, -V	実行されたすべての操作を報告します。

表 45: iarchive オプションの概要

詳細については、494 ページの*オプションの説明*を参照してください。

診断メッセージ

ここでは、iarchive で生成されたメッセージについて説明します。

La001: could not open file *filename*

iarchive がオブジェクトファイルを開くことができませんでした。

La002: illegal path *pathname*

パス *pathname* は有効なパスではありません。

La006: too many parameters to *cmd* command

単一のライブラリファイルのみを指定可能なコマンドに、オブジェクトモジュールのリストがパラメータとして指定されました。

La007: too few parameters to *cmd* command

オブジェクトモジュールのリストを指定可能なコマンドが発行されましたが、必要なモジュールが指定されていません。

La008: *lib* is not a library file

ライブラリファイルが基本構文チェックをパスしませんでした。このファイルはライブラリファイルを意図したものではない可能性があります。

La009: *lib* has no symbol table

ライブラリファイルに必要なシンボル情報が含まれていません。ファイルがライブラリファイルを意図したものではないか、ファイルに ELF オブジェクトモジュールが含まれていない可能性があります。

La010: no library parameter given

ツールが操作対象のライブラリを特定できませんでした。ライブラリファイルが指定されていない可能性があります。

La011: file *file* already exists

同じ名前のファイルがすでに存在するため、ファイルを作成できませんでした。

La013: file confusions, *lib* given as both library and object

オブジェクトモジュールのリストにライブラリファイルも記述されています。

La014: module *module* not present in archive *lib*

指定されたオブジェクトモジュールがアーカイブで見つかりませんでした。

La015: internal error

呼出しにより `iarchive` で予期しないエラーが発生しました。

Ms003: could not open file *filename* for writing

`iarchive` が、書込み用のアーカイブファイルを開くことができませんでした。ライト禁止になっていないか確認してください。

Ms004: problem writing to file *filename*

ファイル *`filename`* への書込み中にエラーが発生しました。ボリュームがいっぱいであることが原因と考えられます。

Ms005: problem closing file *filename*

ファイル *`filename`* を閉じているときにエラーが発生しました。

IAR ELF ツール — ielftool

IAR ELF Tool (ichecksum) は、メモリの特定の範囲におけるチェックサムを生成できます。このチェックサムは、使用しているアプリケーションで計算されるチェックサムと比較できます。

ielftool のソースコードおよび Microsoft VisualStudio 2005 のテンプレートプロジェクトは、arm¥src¥elfutils ディレクトリにあります。チェックサムの生成方法に関する特定の要件やフォーマット変換に関する要件がある場合には、それに応じてソースコードを変更できます。

呼出し構文

IAR ELF Tool の呼出し構文は以下のとおりです。

```
ielftool [options] inputfile outputfile [options]
```

ielftool ツールは、最初にすべてのフィルオプションを処理した後、すべてのチェックサムオプションを（左から右に）処理します。

パラメータ

パラメータを以下に示します。

パラメータ	説明
inputfile	ILINK リンカにより生成される絶対 ELF 実行可能イメージ。
options	任意の使用可能なコマンドラインオプション。詳細については、484 ページの *ielftool オプションの概要*を参照してください。
outputfile	絶対 ELF 実行可能イメージ。

表 46: ielftool のパラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

例

以下の例では、メモリ範囲が 0xFF で埋め込まれた後、同じ範囲のチェックサムが計算されます。

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
          --checksum __checksum:4,crc32;0-0xFF
```

IELFTOOL オプションの概要

以下の表に、ielftool のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--bin	出力ファイルのフォーマットをバイナリに設定します。
--checksum	チェックサムを生成します。
--fill	フィルの要件を指定します。
--ihex	出力ファイルのフォーマットをリニアな Intel hex に設定します。
--parity	パリティビットを生成します。
--self_reloc	一般用ではありません。
--silent	出力抑止操作を設定します。
--simple	出力ファイルのフォーマットを簡易コードに設定します。
--simple-ne	--simple と同じですが、エントリレコードを持ちません。
--srec	出力ファイルのフォーマットを Motorola S-records に設定します。
--srec-len	各 S-record 内のデータバイト数を制限します。
--srec-s3only	S-record 出力にレコードのサブセットのみが含まれるように制限します。
--strip	デバッグ情報を削除します。
--titxt	TI-txt 形式で保存します。
--verbose, -V	実行されたすべての操作を出力します。

表 47: ielftool オプションの概要

詳細については、494 ページの*オプションの説明*を参照してください。

IAR ELF Dumper — ielfdump

IAR ELF Dumper for ARM (ielfdumparm) は、再配置可能 ELF ファイルまたは絶対 ELF ファイルの内容のテキスト表示を作成できます。

ielfdumparm は、以下の 3 とおりの方法で使用できます。

- 入力ファイルおよびそのファイルに含まれる ELF セグメント、ELF セクションの一般属性のリストを小さくする。これは、コマンドラインオプションを使用しない場合のデフォルト動作です。
- 入力ファイル内の ELF セクションの内容のテキスト表現も含める。この動作を指定するには、コマンドラインオプション --all を使用します。

- 入力ファイルから選択した ELF セクションのテキスト表現を少なくする。この動作を指定するには、コマンドラインオプション --section を使用します。

呼出し構文

ielfdumparm の呼出し構文は以下のとおりです。

```
ielfdumparm input_file [output_file]
```

注 : ielfdumparm は、本来、IDE での使用を目的としたコマンドラインツールではありません。

パラメータ

パラメータを以下に示します。

パラメータ	説明
input_file	入力として使用する ELF 再配置可能ファイルまたは ELF 実行可能ファイルです。
output_file	出力先のファイルまたはディレクトリ。存在せず --output オプションが指定されない場合、出力先はコンソールになります。

表 48: ielfdumparm parameters

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

IELFDUMP オプションの概要

以下の表に、ielfdumparm のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--all	名前や番号を考慮せず、すべての入力セクションに対して出力を生成します。
--code	実行可能コードを含むすべてのセクションをダンプします。
-f	コマンドラインを拡張します。
--output, -o	出力ファイルを指定します。
--no_strtab	文字列テーブルのセクションのダンプを無効にします。
--raw	すべての選択セクションに対して、そのセクションの専用出力フォーマットではなく、汎用の 16 進数 /ASCII 出力フォーマットを使用します。
--section, -s	選択した入力セクションに対して出力を生成します。

表 49: ielfdumparm オプションの概要

詳細については、494 ページの *オプションの説明*を参照してください。

IAR ELF オブジェクトツール — iobjmanip

IAR ELF オブジェクトツール、iobjmanip を使用して、ELF オブジェクトファイルの低レベルの操作を実行します。

呼出し構文

IAR ELF オブジェクトツール呼出し構文は以下のとおりです。

```
iobjmanip options inputfile outputfile
```

パラメータ

パラメータを以下に示します。

パラメータ	説明
options	実行する動作を定義するコマンドラインオプションです。これらのオプションは、コマンドラインの任意の場所に配置できます。オプションのどれか 1 つを指定する必要があります。
inputfile	再配置可能な ELF オブジェクトファイル。
outputfile	要求されたすべての操作が適用された、再配置可能な ELF オブジェクトファイル。

表 50: iobjmanip パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

例

この例では、input.o 内のセクション .example が .example2 にリネームされ、結果は output.o に保存されます。

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

IOBJMANIP オプションの概要

以下の表に、iobjmanip オプションの概要を示します。

コマンドラインオプション	説明
-f	コマンドラインを拡張します。
--remove_file_path	ファイルシンボルからパス情報を削除します。
--rename_section	セクションをリネームします。

表 51: iobjmanip オプションの概要

コマンドラインオプション	説明
--rename_symbol	シンボルをリネームします。
--strip	デバッグ情報を削除します。

表 51: iobjmanip オプションの概要（続き）

詳細については、494 ページの *オプションの説明*を参照してください。

診断メッセージ

ここでは、iobjmanip で生成されたメッセージについて説明します。

Lm001: No operation given

コマンドラインパラメータで、実行する処理が指定されていません。

Lm002: Expected *nr* parameters but got *nr*

パラメータが少なすぎるか、または多すぎます。iobjmanip および使用されたコマンドラインオプションの呼出し構文をチェックしてください。

Lm003: Invalid section/symbol renaming pattern *pattern*

パターンに有効なリネーム処理が定義されていません。

Lm004: Could not open file *filename*

iobjmanip が入力ファイルを開くことができませんでした。

Lm005: ELF format error *msg*

入力ファイルは有効な ELF オブジェクトファイルではありません。

Lm006: Unsupported section type *nr*

オブジェクトファイルに、iobjmanip で処理できないセクションが含まれています。出力ファイルの生成時に、このセクションは無視されます。

Lm007: Unknown section type *nr*

iobjmanip で、認識されないセクションが検出されました。iobjmanip は、内容をそのままコピーします。

Lm008: Symbol *symbol* has unsupported format

iobjmanip で、処理できないシンボルが検出されました。iobjmanip は、出力ファイルの生成時にこのシンボルを無視します。

Lm009: Group type *nr* not supported

iobjmanip では、グループタイプ GRP_COMDAT のみがサポートされています。他のグループタイプが検出された場合、結果は未定義になります。

Lm010: Unsupported ELF feature in *file*:*msg*

入力ファイルが、iobjmanip でサポートしていない機能を使用しています。

Lm011: Unsupported ELF file type

入力ファイルは再配置可能なオブジェクトファイルではありません。

Lm012: Ambiguous rename for section/symbol *name* (*alt1* and *alt2*)

セクションまたはシンボルのリネーム中に、曖昧な点が見つかりました。代替手段のいずれかが使用されます。

Lm013: Section *name* removed due to transitive dependency on *name*

明示的に削除されたセクションに依存していたため、セクションが削除されました。

Lm014: File has no section with index *nr*

--remove_section または --rename_section へのパラメータとして使用されるセクションインデックスが、入力ファイルのセクションを参照していませんでした。

Ms003: could not open file *filename* for writing

iobjmanip が、書込み用の入力ファイルを開くことができませんでした。ライト禁止になっていないか確認してください。

Ms004: problem writing to file *filename*

ファイル *filename* への書込み中にエラーが発生しました。ボリュームがいっぱいであることが原因と考えられます。

Ms005: problem closing file *filename*

ファイル *filename* を閉じているときにエラーが発生しました。

IAR Absolute Symbol Exporter — isymexport

IAR Absolute Symbol Exporter (isymexport) は、ROM イメージファイルから絶対シンボルをエクスポートします。これらのシンボルは、アドオンアプリケーションのリンク時に使用できます。

最終のアプリケーションでシンボルファイルからのシンボルを保持するには、ソースコードから、あるいはリンカオプションの --keep を使用して、そのシンボルを参照する必要があります。

呼出し構文

IAR Absolute Symbol Exporter の呼出し構文は以下のとおりです。

```
isymexport [options] inputfile outputfile [options]
```

パラメータ

パラメータを以下に示します。

パラメータ	説明
inputfile	実行可能 ELF ファイルの形式 ROM イメージです（リンクからの出力）。
options	任意の使用可能なコマンドラインオプション。詳細については、490 ページの *isymexport のオプションの概要*を参照してください。
outputfile	リンク入力として使用可能な再配置可能 ELF ファイルです。このファイルには、入力ファイル内の絶対シンボルのすべてまたは選択内容が含まれます。出力ファイルには、シンボルのみ含まれ、実際のコードやデータセクションは含まれません。ステアリングファイルを使用して、出力ファイルに含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。

表 52: isymexport のパラメータ

*ファイル名やディレクトリの指定については、247 ページの ファイル名またはディレクトリをパラメータとして指定する場合の規則を参照*してください。

IDE で、ライブラリシンボルのエクスポートを追加するには、**[プロジェクト] > [オプション] > [ビルドアクション]** を選択し、**[ビルド後コマンドライン]** テキストフィールドでコマンドラインをたとえば次のように指定します。

```
$TOOLKIT_DIR$¥bin¥isymexport.exe "$TARGET_PATH$"
"$PROJ_DIR$¥const_lib.symbols"
```

ISYMEXPORT のオプションの概要

以下の表に、isymexport のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--edit	ステアリングファイルを指定します。
-f	コマンドラインを拡張します。
--generate_vfe_header	破棄された可能性のある関数への仮想関数呼出しがイメージに含まれないことを宣言します。
--reserve_ranges	イメージが使用する ROM および RAM 内のエリアを予約するためにシンボルを生成します。

表 53: isymexport オプションの概要

詳細については、494 ページの*オプションの説明*を参照してください。

ステアリングファイル

ステアリングファイルを使用して、出力に含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。ステアリングファイルでは、show ディレクティブおよび hide ディレクティブを使用して、入力ファイルからどのパブリックシンボルを出力ファイルに含めるかの選択ができます。rename ディレクティブを使用すると、入力ファイル内のシンボルの名前を変更できます。

ステアリングファイルを使用する場合は、アクティブにエクスポートされたシンボルのみが出力ファイルで使用可能になります。このため、show ディレクティブを持たないステアリングファイルでは、シンボルのない出力ファイルが生成されます。

構文

以下の構文規則が適用されます。

- それぞれのディレクティブは、別々の行に指定します。
- C のコメント (/*...*/) や C++ のコメント (//...) を使用できます。
- パターンには、シンボル名の複数文字に対応するワイルドカード文字を含めることができます。
- * 文字は、シンボル名の中のゼロ桁または複数桁の文字のシーケンスに一致すると見なされます。
- ? 文字は、シンボル名の中の任意の 1 文字に一致すると見なされます。

例

```
rename xxx_* as YYY_* /* シンボルのプレフィックスを xxx_ から YYY_ に
                         変更 */
show YYY_*            /* すべてのシンボルを YYY パッケージからエクス
                         ポート */
hide *_internal       /* ただし、内部シンボルはエクスポートしない */
show zzz?             /* zzza をエクスポートして、zzzaaa をエクス
                         ポートしない */
hide zzzx             /* zzzx はエクスポートしない */
```

Show ディレクティブ

構文

```
show pattern
```

パラメータ

pattern	シンボル名に一致するパターンです。

説明

パターンに一致する名前のシンボルが出力ファイルに含まれます。ただし、このシンボルが後で hide ディレクティブでオーバライドされる場合は除きます。

例

```
/* _pub で終わるパブリックシンボルをすべてインクルード。*/
show *_pub
```

Hide ディレクティブ

構文

```
hide pattern
```

パラメータ

pattern	シンボル名に一致するパターンです。

説明

パターンに一致する名前のシンボルが出力ファイルから除外されます。ただし、このシンボルが後に show ディレクティブでオーバライドされる場合は除きます。

例

```
/* _sys で終わるパブリックシンボルをインクルードしない */
hide *_sys
```

Rename ディレクティブ

構文

```
rename pattern1 pattern2
```

パラメータ

pattern1	名前を変更するシンボルの検索に使用されるパターンです。パターンには、* または ? のワイルドカード文字を 1 つしか含めることができません。
pattern2	シンボルの新しい名前に使用されるパターンです。パターンにワイルドカード文字が含まれる場合、*pattern1* に含まれるものと同じ種類でなければなりません。

説明

このディレクティブは、出力ファイルから入力ファイルにシンボルをリネーム変更するときに使用します。エクスポートされるシンボルを複数の rename パターンと一致させることはできません。

rename ディレクティブは、ステアリングファイルの任意の場所に配置できますが、show および hide ディレクティブの前に実行されます。したがって、シンボルをリネームする場合、ステアリングファイルにあるすべての show ディレクティブおよび hide ディレクティブは新しい名前を参照します。

シンボルの名前が、ワイルドカードを含まない *pattern1* パターンに一致する場合、出力ファイルで *pattern2* にリネームされます。

シンボルの名前が、ワイルドカードを含む *pattern1* パターンに一致する場合、出力ファイルで *pattern2* にリネームされますが、名前のワイルドカード文字に一致する部分は保持されます。

例

```
/* xxx_start は出力ファイルで Y_start_X にリネームされます。
   xxx_stop は出力ファイルで Y_stop_X にリネームされます。*/
rename xxx_* Y_*_X
```

診断メッセージ

ここでは、isymexport で生成されたメッセージについて説明します。

Es001: could not open file *filename*

isymexport が指定されたファイルを開くことができませんでした。

Es002: illegal path *pathname*

パス *pathname* は有効なパスではありません。

Es003: format error:*message*

入力ファイルの読み取り中に問題が発生しました。

Es004: no input file

入力ファイルが指定されていません。

Es005: no output file

入力ファイルは指定されていますが、出力ファイルが指定されていません。

Es006: too many input files

ファイルが 3 つ以上指定されています。

Es007: input file is not an ELF executable

入力ファイルは ELF 実行可能ファイルではありません。

Es008: unknown directive:*directive*

ステアリングファイルに指定されたディレクティブが認識されません。

Es009: unexpected end of file

必要な入力の途中でステアリングファイルが終了しました。

Es010: unexpected end of line

ディレクティブが終了する前にステアリングファイルの行が終了しました。

Es011: unexpected text after end of directive

ステアリングファイルのディレクティブと同じ行に、ディレクティブの後に別のテキストが存在します。

Es012: expected *text*

指定されたテキストがステアリングファイルに存在しません。このテキストは、ディレクティブを正しく指定するために必要です。

Es013: pattern can contain at most one * or ?

現在のディレクティブの各パターンに含めることができるワイルドカード文字は、* または ? が 1 つだけです。

Es014: rename patterns have different wildcards

現在のディレクティブの両方のパターンに含まれるワイルドカードは、同じ種類でなければなりません。すなわち、両方が以下のいずれかであることが必要です。

- ワイルドカードなし
- * が 1 つ含まれる
- ? が 1 つ含まれる

このエラーは、パターンがワイルドカードに関して両方のパターンが同じでない場合に発生します。

Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

入力ファイル内のシンボルが複数の `rename` パターンに一致しています。

オプションの説明

このセクションでは、さまざまなユーティリティで使用可能な各コマンドラインオプションの詳しいリファレンス情報を提供します。

--all

構文

```
--all
```

ツール

```
ielfdumparm
```

説明

このオプションは、入力ファイルの汎用属性に加え、すべての ELF セクションの内容を出力に含めるときに使用します。セクションは、インデックスの順に出力されます。ただし、再配置可能セクションについては、そのセクションが再配置用に保持しているセクションの直後に各再配置可能セクションが出力されます。

デフォルトでは、セクションの内容は出力に含まれません。

このオプションは、IDE では使用できません。

--bin

構文

```
--bin
```

ツール

ielftool

説明

出力ファイルのフォーマットをバイナリに設定します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [出力コンバータ]

--checksum

構文

```
--checksum
{symbol[+offset]|address}:size,algorithm[:[1|2][m][L|W][r][i|p]]
[,start];range[;range...]
```

パラメータ

symbol	チェックサム値が格納されるシンボルの名前です。これは、入力 ELF ファイルのシンボルテーブルに存在する必要があります。
offset	シンボルへのオフセット。
address	チェックサム値が格納される絶対アドレスです。
size	チェックサムのバイト数です (1、2、4)。これは、チェックサムシンボルのサイズ以下でなければなりません。

algorithm 使用されるチェックサムアルゴリズムです。以下のいずれかです。

• sum: バイト単位で計算される算術合計。結果は 8 ビットに切り詰められます。

• sum8wide: バイト単位で計算される算術合計。結果はシンボルのサイズに切り詰められます。

• sum32: ワード（32 ビット）単位で計算される算術合計。

• crc16: CRC16（生成多項式 0x11021）。デフォルトで使用されます。

• crc32: CRC32（生成多項式 0x104C11DB7）。

• crc64iso: CRC64iso（生成多項式 0x1B）。

• crc64ecma: CRC64ECMA（生成多項式 0x42F0E1EBA9EA3693）。

• crc=*n*: *n* の生成多項式を使用する CRC。

1|2 指定された場合は、以下のどちらかになります :

• 1 - 1 の補数を指定します。

• 2 - 2 の補数を指定します。

m チェックサムを計算するときに各バイト内でビット順を反転させます。

L|W チェックサムを計算するユニットのサイズを指定します。

以下から選択します。

L: 繰返しごとに 32 ビットのチェックサムを計算します。

W: 繰返しごとに 16 ビットのチェックサムを計算します。

ユニットのサイズを指定しなければ、デフォルトで 8 ビットが使用されます。これらのパラメータを使用しても、チェックサムでエラー検出が強化されることはありません。

r サイズ *size* の各ワード内での入力データのバイト順序を逆にします。

i|p	*start* の値が 0 より大きい場合、i または p を使用してください。指定された場合は、以下のどちらかになります： • i – チェックサムの値を開始値で初期化します。 • p – 入力データの先頭に *start* 値を含むサイズ *size* の 1 ワードを付けます。
start	デフォルトでは、チェックサムの初期値は 0 です。異なる初期値を与える必要がある場合には、*start* を使用してください。0 でない場合は、i か p のどちらかを指定する必要があります。
range	チェックサムが計算されるアドレス範囲です。16 進表記および 10 進表記を使用できます（たとえば、0x8002-0x8FFF）。

ツール

ielftool

説明

このオプションは、指定範囲の指定アルゴリズムのチェックサムを計算するときに使用します。チェックサムに外部の定義がある場合（たとえばハードウェアの CRC 実装など）、--checksum オプションに適切なパラメータを使用して、外部の設計に合わせてください。（この場合、ハードウェアのドキュメントでその設計の詳細を参照してください）。チェックサムは、*symbol* の元の値を置き換えます。新しい絶対シンボルが生成されます。計算されたチェックサムを含む *_value* がサフィックスとして symbol 名に付けられます。このシンボルは、デバッグ中など、必要に応じて後でチェックサム値へのアクセスに使用できます。

--checksum オプションがコマンドラインで複数回使用される場合、オプションは、左から右に評価されます。後で評価される --checksum オプションに指定されている *symbol* で、チェックサムが計算される場合、エラーが発生します。

例

この例は、アドレス範囲 0x8000-0x8FFF、開始値 0 の場合の crc16 アルゴリズムの使用方法を示します。

```
ielftool --checksum=__checksum:2,crc16;0x8000-0x8FFF
sourceFile.out destinationFile.out
```

sourceFile.out から読み込まれる入力データ i と、その結果のサイズ 2 バイトのチェックサム値が、シンボル __checksum に格納されます。修正された ELF ファイルは、destinationFile.out として保存されます。sourceFile.out はそのまま変わりません。

関連項目

202 ページの *チェックサムの計算*。

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] > [リンカ] > [チェックサム]

--code

構文

```
--code
```

ツール

```
ielfdump
```

説明

このオプションを使用して、実行可能コード（ELF セクション属性 SHF_EXECINSTR を持つセクション）を含むすべてのセクションをダンプします。

このオプションは、IDE では使用できません。

--create

構文

```
--create libraryfile objectfile1 ... objectfileN
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	ビルドするライブラリを構成するオブジェクトファイルです。

ツール

```
iarchive
```

説明

このコマンドは、一連のオブジェクトファイル（モジュール）から新しいライブラリをビルドするときに使用します。オブジェクトファイルは、コマンドラインで指定した順序どおりにライブラリに追加されます。

コマンドラインでコマンドを指定しない場合、デフォルトで --create が使用されます。

このオプションは、IDE では使用できません。

--delete, -d

構文

```
--delete libraryfile objectfile1 ... objectfileN
-d libraryfile objectfile1 ... objectfileN
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

ツール

iarchive

説明

このコマンドは、オブジェクトファイル（モジュール）を既存のライブラリから削除するときに使用します。コマンドラインで指定したオブジェクトファイルがすべてライブラリから削除されます。

このオプションは、IDE では使用できません。

--edit

構文

```
--edit steering_file
```

ツール

isymexport

説明

このオプションは、ステアリングファイルを指定するときに使用します。ステアリングファイルでは、isymexport の出力ファイルに含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。

関連項目

490 ページの *ステアリングファイル*。

このオプションは、IDE では使用できません。

--extract, -x

構文

```
--extract libraryfile [objectfile1 ... objectfileN]
-x libraryfile [objectfile1 ... objectfileN]
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

ツール iarchive

説明 このコマンドは、オブジェクトファイル（モジュール）を既存のライブラリから抽出するときに使用します。オブジェクトファイルのリストを指定すると、これらのファイルのみ抽出されます。オブジェクトファイルのリストを指定しない場合には、ライブラリ内のすべてのオブジェクトファイルが抽出されます。


このオプションは、IDE では使用できません。

-f

構文 -f *filename*

パラメータ 247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

ツール iarchive、ielfdumparm、iobjmanip、isymexport

説明 このオプションは、ツールで、指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。


このオプションは、IDE では使用できません。

--fill

構文

```
--fill [v;]pattern;range[;range...]
```

パラメータ

v	フィルコマンドについて仮想フィルを生成します。仮想フィルはチェックサムに含まれるフィルバイトですが、出力ファイルには含まれません。これの主な使用目的は、特定の種類のハードウェアで、イメージにより指定されないバイトに既知の値（通常は 0xFF または 0x0）がある場合です。
pattern	プレフィックス 0x を持つ 16 進数文字列（たとえば、0xEF）は、バイトのシーケンスと解釈され、デジットの各ペアが 1 バイトに相当します（たとえば、0x123456 の場合、バイトのシーケンスは 0x12、0x34、0x56 です）。このシーケンスは、フィルエリアで繰り返されます。フィルパターンの長さが、1 バイトより大きい場合、アドレス 0 から開始されるように繰り返されます。
range	フィルのアドレス範囲を指定します。16 進表記および 10 進表記を使用できます（たとえば、0x8002-0x8FFF）。各アドレスは 4 バイトアラインメントでなければならないので注意してください。

ツール

ielftool

説明

このオプションは、1 つ以上の範囲のすべてのギャップにパターンを埋め込むときに使用します。パターンは、式または 16 進数文字列のいずれかです。この内容は、フィルパターンが開始アドレスから終了アドレスまで繰り返し埋め込まれように計算されます。そして、実際の内容でパターンが上書きされます。

--fill オプションがコマンドラインで複数回使用される場合、フィル範囲がそれぞれと重複することはできません。

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] > [リンカ] > [チェックサム]

--generate_vfe_header

構文

```
--generate_vfe_header
```

ツール

isymexport

説明

このオプションを使用して、破棄された可能性のある関数への仮想関数呼出しがイメージに含まれないことを宣言します。

リンカが仮想関数の除去を実行する際、不要と思われる仮想関数は破棄されます。最適化が正しく適用されるためには、破棄された関数に影響する仮想関数呼出しがイメージに必要です。

関連項目

207 ページの *仮想関数の除去*。

このオプションを設定するには、以下を使用します。

[プロジェクト] > [オプション] > [リンカ] > [追加オプション]

--ihex

構文

```
--ihex
```

ツール

ielftool

説明

出力ファイルのフォーマットを線形の Intel hex に設定します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [出力コンバータ]

--no_strtab

構文

```
--no_strtab
```

ツール

ielfdumparm

説明

このオプションを使用して、文字列のテーブルセクション（SHT_STRTAB 型のセクション）のダンプを無効にします。

このオプションは、IDE では使用できません。

--output, -o

構文

```
-o {filename|directory}
--output {filename|directory}
```

パラメータ

247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則* を参照してください。

ツール

iarchive と ielfdumparm。

説明

iarchive

デフォルトでは、iarchive は、iarchive コマンドの後の最初の引数を、作成するライブラリファイルの名前であるとみなします。このオプションは、ライブラリ用に別のファイル名を明示的に指定する場合に使用します。

ielfdumparm

デフォルトでは、ダンプ結果の出力先はコンソールになります。このオプションは、出力先をファイルに変更するときに使用します。出力ファイルのデフォルト名は、入力ファイル名にファイル名拡張子 id を追加したものです。

また、入力ファイル名の後にファイルやディレクトリを指定して、出力ファイルを指定することもできます。

このオプションは、IDE では使用できません。

--parity

構文

```
--parity{symbol[+offset]|address}:size,algo:flashbase[:flags];range[;range...]
```

パラメータ

symbol	パリティバイトが格納されるシンボルの名前です。これは、入力 ELF ファイルのシンボルテーブルに存在する必要があります。
offset	シンボルへのオフセット。デフォルトでは "0 です。
address	パリティバイトが格納される絶対アドレスです。
size	パリティの生成で使用可能な最大バイト数。この値を超えるとエラーが出力されます。このサイズは、ELF ファイルの指定されたシンボルに合っている必要があります。
algo	以下から選択します。 odd：奇数のパリティを使用。 even：偶数のパリティを使用。

flashbase	フラッシュメモリの開始アドレス。*flashbase* から開始アドレスまでについては、パリティビットは生成されません。*flashbase* と開始アドレスの範囲が重なる場合、すべてのアドレスにパリティビットが生成されます。
flags	以下から選択します。 r：各ワード内でバイトオーダを反転させます。 L：一度に 4 バイトを処理します。 W：一度に 2 バイトを処理します。 B：一度に 1 バイトを処理します。
range	パリティバイトを生成するアドレス範囲。16 進表記および 10 進表記を使用できます（たとえば、0x8002-0x8FFF）。

ツール ielftool

説明 指定範囲にパリティバイトを生成するときに使用します。この範囲は左から右の順になり、奇数または偶数のアルゴリズムを使用してパリティビットが生成されます。パリティビットは最終的に指定のシンボル内に格納され、アプリケーションからアクセスできるようになります。

このオプションは、IDE では使用できません。

--ram_reserve_ranges

構文 --ram_reserve_ranges[=*symbol_prefix*]

パラメータ

symbol_prefix	このオプションによって作成されたシンボルのプレフィックス。

ツール isymexport

説明 このオプションを使用して、イメージが使用する RAM のエリアについてシンボルを生成します。各エリアにシンボルが 1 つ生成されます。各シンボルの名前はエリア名に基づき、オプションのパラメータ *symbol_prefix* がプレフィックスとして付きます。

あるエリアをカバーするシンボルをこの方法で生成すると、影響を受けるアドレスにリンカで他の内容を配置しないように防ぐことができます。これは、既存のイメージに対するリンク処理の際に役立ちます。

--ram_reserve_ranges を --reserve_ranges と同時に使用する場合、RAM エリアは --ram_reserve_ranges オプションからプレフィックスを、RAM 以外のエリアは --reserve_ranges オプションからプレフィックスをそれぞれ取得します。

関連項目

508 ページの *--reserve_ranges*。

このオプションは、IDE では使用できません。

--raw

構文

```
--raw
```

ツール

```
ielfdumparm
```

説明

デフォルトでは、特定の種類のセクション専用テキストフォーマットを使用して、多数の ELF セクションがダンプされます。このオプションは、汎用テキストフォーマットを使用して、選択した各 ELF セクションをダンプするときに使用します。

汎用テキストフォーマットを使用する場合、セクション内の各バイトが 16 進数フォーマットまたは、必要に応じて ASCII テキストにダンプされます。

このオプションは、IDE では使用できません。

--remove_file_path

構文

```
--remove_file_path
```

ツール

```
iobjmanip
```

説明

このオプションは、iobjmanip で、生成されたオブジェクトファイルからプロジェクトソースツリーのディレクトリ情報を削除するときに使用します。つまり、ELF オブジェクトファイルのファイルシンボルが変更されることになります。

このオプションは、--remove_section ".comment" と組み合わせて使用する必要があります。

このオプションは、IDE では使用できません。

--remove_section

構文

```
--remove_section {section|number}
```

パラメータ

section	セクションを削除します（複数も可）。
number	削除されるセクションの番号。セクション番号は、ielfdumparm を使用して作成したオブジェクトダンプから取得できます。

ツール

iobjmanip

説明

このオプションでは、出力ファイルを作成するときに iobjmanip で指定のセクションを省略します。

このオプションは、IDE では使用できません。

--rename_section

構文

```
--rename_section {oldname|oldnumber}=newname
```

パラメータ

oldname	セクションをリネームします（複数も可）。
oldnumber	リネームされるセクションの番号。セクション番号は、ielfdumparm を使用して作成したオブジェクトダンプから取得できます。
newname	セクションの新しい名前。

ツール

iobjmanip

説明

このオプションでは、出力ファイルを作成するときに iobjmanip で指定のセクションをリネームします。

このオプションは、IDE では使用できません。

--rename_symbol

構文

```
--rename_symbol oldname =newname
```

パラメータ

oldname	リネームするシンボル。
newname	シンボルの新しい名前。

ツール

iobjmanip

説明

このオプションでは、出力ファイルを作成するときに iobjmanip で指定のシンボルをリネームします。

このオプションは、IDE では使用できません。

--replace, -r

構文

```
--replace libraryfile objectfile1 ... objectfileN
-r libraryfile objectfile1 ... objectfileN
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 247 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

ツール

iarchive

説明

このコマンドは、既存のライブラリでオブジェクトファイル（モジュール）の置換または追加を行うときに使用します。コマンドラインで指定したオブジェクトファイルにより、ライブラリ内の同一名の既存オブジェクトファイルが置換されます。同一名のファイルが存在しない場合には、コマンドラインで指定したファイルがライブラリに追加されます。

このオプションは、IDE では使用できません。

--reserve_ranges

構文

```
--reserve_ranges[=symbol_prefix]
```

パラメータ

symbol_prefix	このオプションによって作成されたシンボルのプレフィックス。

ツール

isymexport

説明

このオプションを使用して、イメージが使用する ROM および RAM のエリアについてシンボルを生成します。各エリアにシンボルが 1 つ生成されます。各シンボルの名前はエリア名に基づき、オプションのパラメータ *symbol_prefix* がプレフィックスとして付きます。

あるエリアをカバーするシンボルをこの方法で生成すると、影響を受けるアドレスにリンカで他の内容を配置しないように防ぐことができます。これは、既存のイメージに対するリンク処理の際に役立ちます。

--reserve_ranges を --ram_reserve_ranges と同時に使用する場合、RAM エリアは --ram_reserve_ranges オプションからプレフィックスを、RAM 以外のエリアは --reserve_ranges オプションからプレフィックスをそれぞれ取得します。

関連項目

504 ページの *--ram_reserve_ranges*。

このオプションは、IDE では使用できません。

--section, -s

構文

```
--section section_number|section_name[,...]
--s section_number|section_name[,...]
```

パラメータ

section_number	ダンプされるセクションの数。
section_name	ダンプされるセクションの名前。

ツール

ielfdumparm

説明

このオプションは、指定した番号のセクションまたは指定した名前のセクションの内容をダンプするときに使用します。選択したセクションに再配置可能セクションが関連付けられている場合には、その内容も出力されます。

このオプションを使用する場合、入力ファイルの一般属性は出力に含まれません。

セクション番号や名前をカンマで区切るか、このオプションを複数回使用することにより、複数のセクション番号や名前を指定できます。

デフォルトでは、セクションの内容は出力に含まれません。

例

```
-s 3,17                    /* セクション No.3 と No.17
-s .debug_frame,42         /* .debug_frame という名の任意のセクションおよびセクション No.42 */
```

このオプションは、IDE では使用できません。

--self_reloc

構文

```
--self_reloc
```

ツール

ielftool

説明

このオプションは一般用ではないため、意図的に文書化されていません。

このオプションは、IDE では使用できません。

--silent

構文

```
--silent
-S（iarchive のみ）
```

ツール

iarchive および ielftool。

説明

ツールが標準出力ストリームにメッセージ送信せずに処理を実行するように設定します。

デフォルトでは、ielftool によりさまざまなメッセージが標準出力ストリームから送信されます。このオプションを使用して、これらのメッセージ送信を抑止できます。エラーおよび警告メッセージは、ielftool から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。

--simple

構文 `--simple`

ツール `ielftool`

説明 出力ファイルのフォーマットを簡易コードに設定します。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［出力コンバータ］

--simple-ne

構文 `--simple-ne`

ツール `ielftool`

説明 出力ファイルのフォーマットを簡易コードに設定しますが、エントリレコードは生成されません。

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］>［出力コンバータ］

--srec

構文 `--srec`

ツール `ielftool`

説明 出力ファイルのフォーマットを Motorola S-records に設定します。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［出力コンバータ］

--srec-len

構文 `--srec-len=`*`length`*

パラメータ

`length`	各 S-record 内のデータバイト数。

ツール　　ielftool

説明　　各 S-record 内のデータバイト数を制限します。このオプションは、--srec オプションと組み合わせて使用できます。

このオプションは、IDE では使用できません。

--srec-s3only

構文　　--srec-s3only

ツール　　ielftool

説明　　S-record 出力にレコードのサブセット（すなわち S0、S3、S7 レコード）のみが含まれるように制限します。このオプションは、--srec オプションと組み合わせて使用できます。

このオプションは、IDE では使用できません。

--strip

構文　　--strip

ツール　　iobjmanip および ielftool。

説明　　このオプションでは、出力ファイル を書き込む前に、デバッグ情報を含むすべてのセクションを削除します。

ielftool では、リンカオプションを使用していない ELF イメージが必要です。リンカで --strip オプションを使用する場合、これを削除し、代わりに ielftool の --strip オプションを使用します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [出力] > [出力ファイルにデバッグ情報を含める]

--symbols

構文

```
--symbols libraryfile
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 247 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

ツール

iarchive

説明

このコマンドは、指定したライブラリ内のオブジェクトファイル（モジュール）によって定義されるすべての外部シンボルを、そのシンボルを定義しているオブジェクトファイル（モジュール）の名前とともにリストするときに使用します。

出力抑止モード (--silent) の場合、このコマンドは、ライブラリファイルのシンボル関連構文チェックを実行し、エラーと警告のみを表示します。

このオプションは、IDE では使用できません。

--titxt

構文

```
--titxt
```

ツール

ielftool

説明

出力ファイルのフォーマットを TI–txt に設定します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [出力コンバータ]

--toc, -t

構文

```
--toc libraryfile
-t libraryfile
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 247 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

ツール

iarchive

説明

このコマンドは、指定したライブラリ内のすべてのオブジェクトファイル（モジュール）をリストするときに使用します。

出力抑止モード (--silent) の場合、このコマンドは、ライブラリファイルの基本的な構文チェックを実行し、エラーと警告のみを表示します。

このオプションは、IDE では使用できません。

--verbose, -V

構文

--verbose
-V（iarchive のみ）

ツール

iarchive および ielftool。

説明

このオプションは、診断メッセージのほかに、実行された操作をツールで報告するときに使用します。

この設定は常に有効化されているため、このオプションは IDE では使用できません。

C 規格の処理系定義の動作

- 処理系定義の動作の詳細

C 規格ではなく C89 を使用する場合、531 ページの *C89 の処理系定義の動作*を参照してください。C 規格と C89 の違いの大まかな概要については、175 ページの *C 言語の概要*を参照してください。

処理系定義の動作の詳細

ここでは、C 規格と同順で各項目を説明します。各項目では、処理系定義の動作を説明する ISO の章 / セクション（括弧で示す）を示しています。

注 : IAR システムズの実装は、標準の C のフリースタンディング実装に準拠しています。すなわち、標準ライブラリの一部を実装で除外できます。

J.3.1 変換

診断 (3.10, 5.1.1.3)

診断は、以下のフォーマットで生成されます。

```
filename,linenumber level[tag]: message
```

ここで、*filename* はエラーが発生したソースファイル名、*linenumber* はコンパイラがエラーを検出した行番号、*level* はメッセージの重要度（リマーク、ワーニング、エラー、致命的なエラー）、*tag* はメッセージを識別する固有のタグ、*message* は数行に及ぶこともある説明のメッセージです。

空白文字 (5.1.1.2)

3 番目の変換フェースでは、空でない空白文字の各シーケンスが保持されます。

J.3.2 環境

文字集合 (5.1.1.2)

ソースの文字集合は、物理的なソースファイルのマルチバイト文字集合と同じです。デフォルトでは、標準の ASCII 文字集合が使用されます。ただし、`--enable_multibytes` コンパイラオプションを使用する場合、ホストの文字集合が代わりに使用されます。

Main (5.1.2.1)

プログラム起動時に呼出される関数は、main 関数です。main にはプロトタイプは宣言されていません。main でサポートされている唯一の定義は以下のとおりです。

```
int main(void)
```

この動作を変更するには、129 ページの *システム初期化のカスタマイズ*を参照してください。

プログラム終了の影響 (5.1.2.1)

アプリケーションを終了すると、（main の呼出し直後に）実行が起動コードに戻ります。

その他の main の定義方法 (5.1.2.2.1)

main 関数を定義する方法は他にありません。

main の argv 引数 (5.1.2.2.1)

argv 引数はサポートされていません。

対話型デバイスとしてのストリーム (5.1.2.3)

ストリーム stdin、stdout、stderr は対話型装置として処理されます。

シグナルとその意味およびデフォルトの処理 (7.14)

DLIB ライブラリでは、サポートされているシグナルのセットは標準の C と同じです。引き起こされたシグナルは、signal 関数がアプリケーションに合うようにカスタマイズされていない限り、何の処理も行いません。

計算の例外のシグナル値 (7.14.1.1)

DLIB ライブラリでは、計算の例外に一致する処理系定義の値はありません。

システム起動時のシグナル (7.14.1.1)

DLIB ライブラリでは、システム起動時に実行される処理系定義のシグナルはありません。

環境名 (7.20.4.5)

DLIB ライブラリでは、getenv 関数に使用される処理系定義の環境名はありません。

システム関数 (7.20.4.6)

`system` 関数はサポートされていません。

J.3.3 識別子

識別子のマルチバイト文字 (6.4.2)

その他のマルチバイト文字は識別子に表示されないことがあります。

識別子における重要な文字 (5.2.4.1, 6.1.2)

外部リンケージの有無に関わらず、識別子の重要な先頭文字の数は 200 文字以上であることが保証されています。

J.3.4 文字

バイト内のビット数 (3.6)

1 バイトには 8 ビットが含まれます。

実行文字集合のメンバ値 (5.2.1)

実行文字集合のメンバ値は、ASCII 文字集合の値です。これらはホストの文字集合の追加文字の値によって追加することが可能です。

英数字のエスケープシーケンス (5.2.2)

標準の英数字のエスケープシーケンスには、`¥a-7`、`¥b-8`、`¥f-12`、`¥n-10`、`¥r-13`、`¥t-9`、`¥v-11` の値があります。

重要な基本文字集合外の文字 (6.2.5)

`char` に保存された重要な基本文字集合外の文字は変換されません。

char 型 (6.2.5, 6.3.1.1)

通常の `char` 型は、`unsigned char` として処理されます。

ソースおよび実行文字集合 (6.4.4.4, 5.1.1.2)

ソース文字集合は、ソースファイルで使用できる正当な文字集合です。デフォルトのソース文字集合は、標準 ASCII 文字集合です。ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、ソース文字集合はホストコンピュータのデフォルトの文字集合になります。

実行文字集合は、実行環境で使用できる正当な文字集合です。デフォルトの実行文字集合は、標準 ASCII 文字集合です。

ただし、コマンドラインオプション --enable_multibytes を使用した場合は、実行文字集合はホストコンピュータのデフォルトの文字集合になります。IAR DLIB ライブラリでは、マルチバイトの実行文字集合をサポートするには、マルチバイト文字スキャナが必要です。136 ページの *ロケール*を参照してください。

複数の文字を含む整数文字定数 (6.4.4.4)

文字数が複数の整数文字定数は、整数定数として処理されます。値は、整数定数で左端の文字を最上位文字、右端の文字を最下位文字として計算されます。値が整数定数で表現できない場合は、診断メッセージが出力されます。

複数の文字を含むワイド文字定数 (6.4.4.4)

複数のマルチバイト文字を含むワイド文字定数を使用すると、診断メッセージが出力されます。

ワイド文字定数に使用されるロケール (6.4.4.4)

デフォルトでは C ロケールが使用されます。--enable_multibytes コンパイラオプションが使用される場合、デフォルトのホストロケールが代わりに使用されます。

ワイド文字列リテラルに使用されるロケール (6.4.5)

デフォルトでは C ロケールが使用されます。--enable_multibytes コンパイラオプションが使用される場合、デフォルトのホストロケールが代わりに使用されます。

重要文字としてのソース文字 (6.4.5)

すべてのソース文字は、重要文字として表すことができます。

J.3.5 整数

拡張整数型 (6.2.5)

拡張整数型はありません。

整数値の範囲 (6.2.6.2)

整数値は、2 の補数で表現されます。最上位ビットは符号を示し、1 の場合は負の値、0 の場合は正の値またはゼロを示します。

各種の整数型の範囲については、327 ページの *基本データ型整数型*を参照してください。

拡張整数型のランク (6.3.1.1)

拡張整数型はありません。

符号付き整数型に変換したときのシグナル (6.3.1.3)

整数が符号付きの整数型に変換された場合、シグナルは引き起こされません。

符号付整数に対するビット単位の演算 (6.5)

符号付整数に対するビット単位の演算は、符号なし整数に対するビット単位演算と同様に行われます。すなわち、符号ビットが他のビットと同様に扱われます。

J.3.6 浮動小数点

浮動小数点処理の精度 (5.2.4.2.2)

浮動小数点処理の精度は不明です。

丸め処理 (5.2.4.2.2)

FLT_ROUNDS の非標準値はありません。

評価方法 (5.2.4.2.2)

FLT_EVAL_METHOD の非標準値はありません。

整数値の浮動小数点値への変換 (6.3.1.4)

整数値がソース値を正確に表現できない浮動小数点値に変換されると、最も近い値に丸めるモードが使用されます（FLT_ROUNDS が 1 に定義されています）。

浮動小数点の浮動小数点への変換 (6.3.1.5)

浮動小数点値がソース値を正確に表現できない浮動小数点値に変換されると、最も近い値に丸めるモードが使用されます（FLT_ROUNDS が 1 に定義されています）。

浮動小数点定数値の記述 (6.4.4.2)

最も近い値への丸めモードが使用されます（FLT_ROUNDS が 1 に定義されています）。

浮動小数点値の縮約 (6.5)

浮動小数点値は縮約されます。ただし、精度のロスはありません。シグナルはサポートされていないため、これは問題にはなりません。

FENV_ACCESS のデフォルトの状態 (7.6.1)

プラグマディレクティブ `FENV_ACCESS` のデフォルトの状態は、`OFF` です。

その他の浮動小数点メカニズム (7.6, 7.12)

浮動小数点例外、丸めモード、環境、分類は他にはありません。

FP_CONTRACT のデフォルトの状態 (7.12.2)

プラグマディレクティブ `FP_CONTRACT` のデフォルトの状態は、`OFF` です。

J.3.7 配列およびポインタ

ポインタの変換 (6.3.2.3)

データポインタおよび関数ポインタのキャストについて詳しくは、334 ページの *キャスト*を参照してください。

ptrdiff_t (6.5.6)

`ptrdiff_t` については、335 ページの *ptrdiff_t* を参照してください。

J.3.8 ヒント

レジスタキーワードの考慮 (6.7.1)

レジスタ変数についてのユーザ要求は考慮されません。

関数のインライン化 (6.7.4)

関数のインライン化へのユーザ要求で確率は高くなりますが、関数が実際に別の関数にインライン化されるか確実ではありません。74 ページの*インライン関数*を参照してください。

J.3.9 構造体、共用体、列挙型、ビットフィールド

プレーンなビットフィールドの符号 (6.7.2, 6.7.2.1)

プレーンな `int` ビットフィールドの処理については、「328 ページの *ビットフィールド*」を参照してください。

ビットフィールドの可能な型 (6.7.2.1)

すべての整数型は、コンパイラの拡張モードでビットフィールドとして使用できます（263 ページの *-e* を参照）。

記憶単位の境界をまたぐビットフィールド (6.7.2.1)

ビットフィールドは常に 1 つの記憶単位にだけ配置されます。つまり、ビットフィールドは記憶単位をまたぐことはできません。

単位内のビットフィールドの割当順序 (6.7.2.1)

記憶単位内のビットフィールドの割当て方法については、328 ページの *ビットフィールド*を参照してください。

ビットフィールド以外の構造体メンバのアラインメント (6.7.2.1)

ビットフィールド以外の構造体メンバのアラインメントは、メンバ型と同じです（325 ページの *アラインメント*を参照）。

列挙型を表すときに使用される整数型 (6.7.2.2)

特定の列挙型用に選択される整数型は、列挙型用に定義された列挙定数によって異なります。最小の整数型が選択されます。

J.3.10 修飾子

volatile オブジェクトへのアクセス (6.7.3)

`volatile` で修飾された型のオブジェクトへの参照は、すべてアクセスされます（337 ページの *オブジェクトの volatile 宣言*を参照）。

J.3.11 プリプロセッサディレクティブ

ヘッダ名のマッピング (6.4.7)

ヘッダ名のシーケンスは、ソースファイル名にそのままマッピングされます。バックスラッシュ '¥' は、エスケープシーケンスとして扱われません。
419 ページの *プリプロセッサの概要*を参照してください。

定数式の文字定数 (6.10.1)

条件付きインクルードを制御する定数式の文字定数は、実行文字集合の同じ文字定数の値と一致します。

単一文字定数の値 (6.10.1)

通常の文字 (char) が符号付き文字として扱われる場合、単一文字定数はマイナスの値しか持つことができません（254 ページの *--char_is_signed* を参照）。

括弧のついたファイル名のインクルード (6.10.2)

角括弧 <> のファイル仕様に使用される検索アルゴリズムについては、237 ページの*インクルードファイル検索手順*を参照してください。

引用符のあるファイル名のインクルード (6.10.2)

引用符で囲まれたファイル仕様に使用される検索アルゴリズムについては、237 ページの*インクルードファイル検索手順*を参照してください。

#include ディレクティブのプリプロセッサトークン (6.10.2)

#include ディレクティブのプリプロセッサトークンは、#include ディレクティブの外側の場合と同じように組み合わされます。

#include ディレクティブのネストの制限 (6.10.2)

#include 処理のネストに明示的な制限はありません。

汎用文字名 (6.10.3.2)

汎用文字名 (UCN) はサポートされていません。

認識されているプラグマディレクティブ (6.10.6)

「*プラグマディレクティブ*」章で説明したプラグマディレクティブ以外にも、以下のディレクティブも認識されます。ただし、これらの影響は不定です。プラグマディレクティブが「*プラグマディレクティブ*」とここの両方に記載されている場合、ここの情報よりも「*プラグマディレクティブ*」の章の情報が優先します。

```
alignment

baseaddr

basic_template_matching

building_runtime

can_instantiate

codeseg

constseg

cspy_support
```

```
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
library_provides
library_requirement_override
memory
module_name
no_pch
once
system_include
vector
warnings
```

Default __DATE__ and __TIME__ (6.10.8)

__TIME__、__DATE__ の定義は常に使用可能です。

J.3.12 ライブラリ関数

その他のライブラリ機能 (5.1.2.1)

標準のライブラリ機能のほとんどがサポートされています。その一部（オペレーティングシステムを必要とするもの）には、アプリケーションに低レベルの実装が必要です。詳細については、109 ページの *DLIB ランタイム環境*を参照してください。

アサート関数によって出力される診断 (7.2.1.1)

assert() 関数で出力される診断は、以下のとおりです。

filename:*linenr expression* -- assertion failed

この診断は、パラメータがゼロに評価される場合に出力されます。

浮動小数点のステータスフラグの表現 (7.6.2.2)

浮動小数点のステータスフラグについては、436 ページの *fenv.h* を参照してください。

浮動小数点の例外を引き起こす feraiseexcept 関数 (7.6.2.3)

浮動小数点の例外を引き起こす feraiseexcept 関数については、332 ページの*浮動小数点環境*を参照してください。

setlocale 関数に渡される文字列 (7.11.1.1)

setlocale 関数に渡される文字列については、136 ページの*ロケール*を参照してください。

float_t および double_t に定義される型 (7.12)

FLT_EVAL_METHOD マクロは、値 0 しか持つことができません。

ドメインエラー (7.12.1)

標準で必要とされるもの以外のドメインエラーを生成する関数はありません。

ドメインエラーのリターン値 (7.12.1)

数学関数は、ドメインエラーに対して浮動小数点 NaN（not a number ＝非数）を返します。

アンダーフローエラー (7.12.1)

数学関数は errno をマクロ ERANGE（errno.h のマクロ）に設定し、アンダーフローエラーにゼロを返します。

fmod 関数のリターン値 (7.12.10.1)

fmod 関数は、2 番目の引数がゼロの場合、浮動小数点 NaN を返します。

remquo の規模 (7.12.10.3)

規模は、合同の剰余 INT_MAX です。

signal 関数 (7.14.1.1)

シグナル関連のライブラリはサポートされていません。

注 : 低レベルインタフェース関数はライブラリには存在しますが、これらの関数は何も実行しません。アプリケーション固有のシグナル処理を実装する場合は、テンプレートソースコードを使用してください。139 ページの *signal と raise* を参照してください。

NULL マクロ (7.17)

`NULL` マクロは、`0` に定義されています。

改行文字による終了 (7.19.2)

`stdout` ストリーム関数は、`newline` または `end of file (EOF)` のどちらかを改行文字として認識します。

改行文字の前の空白文字 (7.19.2)

ストリームで改行文字直前に書き込まれた空白文字は保持されます。

バイナリストリームに書き込まれたデータに追加された Null 文字 (7.19.2)

バイナリストリームにライトされたデータには、NULL 文字は追加されません。

追加モードでのファイル位置 (7.19.3)

ファイル位置は、追加モードで開かれた場合にファイルの先頭に配置されます。

ファイルの切捨て (7.19.3)

テキストストリームへのライトにより、対応するファイルでその書込み位置以降が切り捨てられるかどうかは、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。135 ページの *ファイル I/O* を参照してください。

ファイルのバッファ処理 (7.19.3)

開かれたファイルは、ブロックバッファ、ラインバッファまたはアンバッファのいずれかです。

ゼロ長のファイル (7.19.3)

ゼロ長のファイルが存在するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

有効なファイル名 (7.19.3)

ファイル名が有効かどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

ファイルを開くことができる回数 (7.19.3)

ファイルを何度も開くことができるかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

ファイル内のマルチバイト文字 (7.19.3)

ファイル内のマルチバイト文字のエンコーディングは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

remove 関数 (7.19.4.1)

開いたファイルに対する削除処理の結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。135 ページの *ファイル I/O* を参照してください。

rename 関数 (7.19.4.2)

ファイルの名前を既存のファイル名に変更した結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。135 ページの *ファイル I/O* を参照してください。

開かれた一時ファイルの削除 (7.19.4.3)

開かれた一時ファイルを削除するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

モード変更 (7.19.5.4)

`freopen` は指定のストリームを閉じて、新しいモードで再び開きます。ストリーム `stdin`、`stdout`、`stderr` は、すべての新しいモードで再び開くことができます。

infinity または NaN の出力形式 (7.19.6.1, 7.24.2.1)

浮動小数点定数の infinity または NaN の出力形式は、それぞれ inf、nan（F 変換指定子の場合は INF と NAN）です。n-char-sequence は、nan に対しては使用されません。

printf 関数における %p (7.19.6.1, 7.24.2.1)

printf() の %p 変換指定子（出力ポインタ）の引数は、void * 型として処理されます。値は、%x 変換指定子の場合と同様に、16 進数として出力されます。

scanf 関数の読込み範囲 (7.19.6.2, 7.24.2.1)

-（ダッシュ）文字は、常に範囲記号として処理されます。

scanf 関数における %p (7.19.6.2, 7.24.2.2)

scanf() の %p 変換指定子（スキャンポインタ）は、16 進数を読み取り、void * 型の値に変換します。

ファイル位置のエラー (7.19.9.1, 7.19.9.3, 7.19.9.4)

ファイル位置のエラーでは、関数 fgetpos、ftell、fsetpos は EFPOS を errno に格納します。

nan の後の n-char-sequence (7.20.1.3, 7.24.4.1.1)

NaN の後の n-char-sequence は、読み込まれて無視されます。

アンダーフローの errno 値 (7.20.1.3, 7.24.4.1.1)

errno は、アンダーフローがあった場合には ERANGE に設定されます。

サイズがゼロのヒープオブジェクト (7.20.3)

サイズがゼロのヒープオブジェクトの要求は、NULL ポインタではなく有効なポインタを返します。

abort 関数および exit 関数の動作 (7.20.4.1, 7.20.4.4)

abort() または _Exit() を呼出しても、ストリームバッファはフラッシュされず、オープンのストリームを閉じたり、一時ファイルが削除されることもありません。

終了ステータス (7.20.4.1, 7.20.4.3, 7.20.4.4)

終了ステータスはパラメータとして __exit() に伝播されます。exit() と _Exit() は入力パラメータを使用しますが、abort は EXIT_FAILURE を使用します。

system 関数のリターン値 (7.20.4.6)

system 関数はサポートされていません。

タイムゾーン (7.23.1)

ローカルのタイムゾーンおよび夏時間をアプリケーションで定義する必要があります。詳細については、139 ページの*時間*を参照してください。

時間の範囲および精度 (7.23)

範囲と精度について詳しくは、437 ページの *time.h* を参照してください。アプリケーションは関数 time と clock に実際の実装を提供する必要があります。139 ページの*時間*を参照してください。

clock 関数 (7.23.2.1)

アプリケーションは、clock 関数の実装を提供する必要があります。139 ページの*時間*を参照してください。

%Z 置換文字列 (7.23.3.5, 7.24.5.1)

デフォルトでは ":" が %Z の代わりに使用されます。使用するアプリケーションがタイムゾーンの処理を実装することになります。139 ページの*時間*を参照してください。

数学関数の丸めモード (F.9)

math.h の関数は、FLT-ROUNDS の丸め方向モードに従います。

J.3.13 アーキテクチャ

一部のマクロに割り当てられた値と式 (5.2.4.2, 7.18.2, 7.18.3)

1 バイトは常に 8 ビットです。

MB_LEN_MAX は、使用されるライブラリ設定に応じて最高で 6 バイトになります。

すべての基本型のサイズや範囲などについては、325 ページの*データ表現*を参照してください。

stdint.h で定義される正確な幅、最小幅、最高速かつ最小幅の整数型の制限マクロは、char、short、int、long、long long と同じ範囲になります。

浮動小数点定数 FLT_ROUNDS の値は 1（最も近い値）で、浮動小数点定数 FLT_EVAL_METHOD の値は 0（そのまま処理）です。

バイトの数値、順序、エンコーディング (6.2.6.1)

325 ページの *データ表現* を参照してください。

sizeof 演算子の結果の値 (6.5.3.4)

325 ページの *データ表現* を参照してください。

J.4 ロケール

ソースのメンバおよび実行文字集合 (5.2.1)

デフォルトでは、コンパイラはホストのデフォルト文字集合にあるすべての 1 バイト文字を受け入れます。コンパイラオプション --enable_multibytes を使用する場合、ホストのマルチバイト文字はコメントや文字列リテラルでも受け入れられます。

その他の文字集合の意味 (5.2.1.2)

拡張ソース文字集合のすべてのマルチバイト文字は、拡張実行文字集合にそのまま変換されます。文字が正しく処理されるかどうかは、ライブラリ設定のサポートを持ったアプリケーションに依存します。

マルチバイト文字のエンコードのシフト状態 (5.2.1.2)

コンパイラオプション --enable_multibytes を使用すると、ホストのデフォルトのマルチバイト文字が拡張ソース文字として使用可能になります。

連続する出力文字の方向 (5.2.2)

アプリケーションが表示デバイスの特性を定義します。

小数点の文字 (7.1.1)

デフォルトの小数点の文字は '.' です。ライブラリ設定シンボル _LOCALE_DECIMAL_POINT を定義すれば、設定し直すことができます。

出力文字 (7.4, 7.25.2)

出力文字集合は、選択したロケールによって決まります。

制御文字 (7.4, 7.25.2)

制御文字集合は、選択したロケールによって決まります。

テスト済みの文字 (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

テスト済みの文字集合は、選択したロケールによって決まります。

ネイティブ環境 (7.1.1.1)

ネイティブ環境は、"C" ロケールと同じです。

数値変換関数の対象シーケンス (7.20.1, 7.24.4.1)

数値変換関数で受け入れが可能な他の対象シーケンスはありません。

実行文字集合の照合 (7.21.4.3, 7.24.4.4.2)

実行文字集合の照合は、選択したロケールによって決まります。

strerror 関数によって返されるメッセージ (7.21.6.2)

`strerror` 関数が返すメッセージは、引数に応じて以下のようになります。

引数	メッセージ
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
その他	`error` *nnn*

表 54: strerror() が返すメッセージ— IAR DLIB ライブラリ

C89 の処理系定義の動作

- 処理系定義の動作の詳細

C89 ではなく C 規格を使用する場合、515 ページの *C 規格の処理系定義の動作*を参照してください。C 規格と C89 の違いの大まかな概要については、175 ページの *C 言語の概要*を参照してください。

処理系定義の動作の詳細

ISO の付録と同順で各項目を説明します。各項目では、処理系定義の動作を説明する ISO の章 / セクション（括弧で示す）を示しています。

変換

診断 (5.1.1.3)

診断は、以下のフォーマットで生成されます。

```
filename,linenumber level[tag]: message
```

ここで、*filename* はエラーが発生したソースファイル名、*linenumber* はコンパイラがエラーを検出した行番号、*level* はメッセージの重要度（リマーク、ワーニング、エラー、致命的なエラー）、*tag* はメッセージを識別する固有のタグ、*message* は数行に及ぶこともある説明のメッセージです。

環境

main 関数の引数 (5.1.2.2.2.1)

プログラム起動時に呼出される関数は、`main` 関数です。`main` にはプロトタイプは宣言されていません。`main` でサポートされている唯一の定義は以下のとおりです。

```
int main(void)
```

IAR DLIB ランタイム環境におけるこの動作を変更する場合は、129 ページの *システム初期化のカスタマイズ*を参照してください。

対話型装置 (5.1.2.3)

ストリーム `stdin`、`stdout` は、対話型装置として処理されます。

識別子

外部リンクなしの識別子 (6.1.2)

外部リンクなしの識別子での有効先頭文字数は 200 です。

外部リンクありの識別子 (6.1.2)

外部リンクありの識別子での有効先頭文字数は 200 です。

大文字と小文字の区別 (6.1.2)

外部リンクのある識別子では、大文字と小文字が区別されます。

文字

ソース文字集合・実行文字集合 (5.2.1)

ソース文字集合は、ソースファイルで使用できる正当な文字集合です。デフォルトのソース文字集合は、標準 ASCII 文字集合です。ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、ソース文字集合はホストコンピュータのデフォルトの文字集合になります。

実行文字集合は、実行環境で使用できる正当な文字集合です。デフォルトの実行文字集合は、標準 ASCII 文字集合です。ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、実行文字集合はホストコンピュータのデフォルトの文字集合になります。IAR DLIB ライブラリでは、マルチバイトの実行文字集合をサポートするには、マルチバイト文字スキャナが必要です。

136 ページの*ロケール*を参照してください。

実行文字集合の 1 文字当たりのビット数 (5.2.4.2.1)

1 文字あたりのビット数は、manifest 定数 `CHAR_BIT` で示されます。標準インクルードファイル `limits.h` では、`CHAR_BIT` は 8 と定義されています。

文字のマッピング (6.1.3.4)

ソース文字集合（文字か文字列リテラル）のメンバと実行文字集合のメンバのマッピングは、1 対 1 で設定されています。すなわち、文字集合内の各メンバを表現する値は、ISO 規格で規定されているエスケープシーケンスを除き、両者で同一のものが使用されています。

表現されない文字定数 (6.1.3.4)

基本実行文字集合かワイド文字定数用の拡張文字集合で表現されていない文字やエスケープシーケンス整数文字定数の値を使用すると、診断メッセージが出力され、実行文字集合に合わせて切り詰められます。

1 文字以上の文字定数 (6.1.3.4)

文字数が複数の整数文字定数は、整数定数として処理されます。値は、整数定数で左端の文字を最上位文字、右端の文字を最下位文字として計算されます。値が整数定数で表現できない場合は、診断メッセージが出力されます。

複数のマルチバイト文字を含むワイド文字定数を使用すると、診断メッセージが出力されます。

マルチバイト文字の変換 (6.1.3.4)

サポートされているロケール（IAR C/C++ コンパイラで提供されているロケール）は、C ロケールのみです。コマンドラインオプション `--enable_multibytes` を指定し、マルチバイトをサポートするロケールやマルチバイト文字スキャナをライブラリに追加した場合は、IAR DLIB ライブラリでマルチバイト文字がサポートされます。

136 ページの *ロケール*を参照してください。

プレーンな char の範囲 (6.2.1.1)

プレーンな `char` の範囲は、`unsigned char` と同一です。

整数

整数値の範囲 (6.1.2.5)

整数値は、2 の補数で表現されます。最上位ビットは符号を示し、1 の場合は負の値、0 の場合は正の値またはゼロを示します。

各種の整数型の範囲については、327 ページの *基本データ型整数型*を参照してください。

整数の降格 (6.2.1.2)

整数をより短い符号付整数に変換する場合は、切捨てが行われます。符号なし整数を同一長の符号付整数に変換すると値が表現できなくなる場合も、ビットパターンは変化しません。すなわち、値が十分に大きい場合、負の値に変換されます。

符号付整数に対するビット単位の演算 (6.3)

符号付整数に対するビット単位の演算は、符号なし整数に対するビット単位演算と同様に行われます。すなわち、符号ビットが他のビットと同様に扱われます。

整数除算での余りの符号 (6.3.5)

整数除算での余りの符号は、被除数の符号と同一です。

負の符号付整数型の右シフト (6.3.7)

負の符号付整数型を右シフトした場合、符号ビットが保持されます。たとえば、`0xFF00` を 1 回右シフトすると、結果は `0xFF80` になります。

浮動小数点数

浮動小数点数の表現 (6.1.2.5)

浮動小数点数の表現およびセットは、IEEE 854–1987 に準拠しています。通常の浮動小数点数は、符号ビット (`s`)、バイアス指数 (`e`)、指数部 (`m`) で構成されます。

浮動小数点数型 (`float`、`double`) の範囲およびサイズについては、332 ページの *基本データ型浮動小数点数型*を参照してください。

整数値から浮動小数点値への変換 (6.1.2.3)

整数値を、値を正確に表現できない浮動小数点数値にキャストした場合、値は最近似値に丸められます（切上げまたは切捨て）。

浮動小数点値の降格 (6.2.1.4)

浮動小数点数値を、サイズが小さな型の、値を正確に表現できない浮動小数点数値に変換した場合、値は最近似値に丸められます（切上げまたは切捨て）。

配列、ポインタ

size_t (6.3.3.4, 7.1.1)

`size_t` については、335 ページの *size_t* を参照してください。

ポインタからの変換・ポインタへの変換 (6.3.4)

データポインタと関数ポインタのキャストについては、334 ページのキャストを参照してください。

ptrdiff_t (6.3.6, 7.1.1)

ptrdiff_t については、335 ページの *ptrdiff_t* を参照してください。

レジスタ

レジスタキーワードの扱い (6.5.1)

レジスタ変数についてのユーザ要求は考慮されません。

構造体、共用体、列挙型、ビットフィールド

共用体への不正なアクセス (6.3.2.3)

メンバを使用して共用体に値を格納し、そのメンバとは異なる型のメンバを使用してアクセスすると、結果は最初のメンバの内部記憶エリアに完全に依存します。

構造体メンバのパディングとアラインメント (6.5.2.1)

データオブジェクトの配置要件については、327 ページの*基本データ型整数型*を参照してください。

プレーンなビットフィールドの符号 (6.5.2.1)

プレーンな int ビットフィールドは、unsigned int ビットフィールドとして処理されます。すべての整数型は、ビットフィールドとして使用できます。

ビットフィールドの配置順 (6.5.2.1)

ビットフィールドは、整数内で最下位ビットから最上位ビットの順でアラインメントされます。

記憶単位の境界をまたぐビットフィールド (6.5.2.1)

ビットフィールドは、選択したビットフィールド整数型の記憶単位の境界をまたぐことはできません。

列挙型を表現するために選択される整数型 (6.5.2.2)

特定の列挙型用に選択される整数型は、列挙型用に定義された列挙定数によって異なります。最小の整数型が選択されます。

修飾子

volatile オブジェクトへのアクセス (6.5.3)

volatile で修飾された型のオブジェクトへの参照は、すべてアクセスされます。

宣言子

宣言子数の上限 (6.5.4)

宣言子の個数には上限はありません。個数は、空きメモリ容量にのみ制限されます。

文

case 文の上限数 (6.6.4.2)

switch 文での case 文（case 値）の個数には、上限はありません。個数は、空きメモリ容量にのみ制限されます。

プリプロセッサディレクティブ

文字定数と条件の指定 (6.8.1)

プリプロセッサディレクティブで使用されている文字集合は、実行文字集合と同一です。プリプロセッサは、プレーン char が `signed` char として扱われる場合、負の char を認識します。

角括弧で括ったファイル指定 (6.8.2)

角括弧で括ったファイル指定の場合、プリプロセッサは親ファイルのディレクトリを検索しません。親ファイルは、`#include` ディレクティブを含むファイルになります。その代わりに、コンパイラのコマンドラインで指定したディレクトリで最初にファイル検索を実行します。

引用符で括ったファイル指定 (6.8.2)

引用符で括ったファイル指定の場合、プリプロセッサのディレクトリ検索は、親ファイルのディレクトリでまず実行され、その後でそれより上位の階層のファイルのディレクトリに対して順に実行されます。したがって、処理中のソースファイルを含むディレクトリと相対的に検索が行われます。親よりも上位の階層のファイルがなく、ファイルが見つからなかった場合は、角括弧で括ってファイル名を指定した場合と同様に検索が続行されます。

文字シーケンス (6.8.2)

プリプロセッサディレクティブは、エスケープシーケンスを除き、ソース文字集合を使用します。したがって、インクルードファイルのパスを指定する場合は、次のように円記号を 1 つだけ使用します。

```
#include "mydirectory¥myfile"
```

ソースコード内では、次のように円記号が 2 つ必要です。

```
file = fopen("mydirectory¥¥myfile","rt");
```

認識できるプラグマディレクティブ (6.8.6)

「*プラグマディレクティブ*」で説明したプラグマディレクティブ以外にも、以下のディレクティブも認識されます。ただし、これらの影響は不定です。プラグマディレクティブが「*プラグマディレクティブ*」とここの両方に記載されている場合、ここの情報よりも「*プラグマディレクティブ*」の章の情報が優先します。

```
alignment

baseaddr

basic_template_matching

building_runtime

can_instantiate

codeseg

constseg

cspy_support

dataseg

define_type_info

do_not_instantiate

early_dynamic_initialization

function

function_effects

hdrstop

important_typedef

instantiate

keep_definition
```

```
library_default_requirements

library_provides

library_requirement_override

memory

module_name

no_pch

once

system_include

vector

warnings
```

Default __DATE__ and __TIME__ (6.8.8)

__TIME__、__DATE__ の定義は常に使用可能です。

IAR DLIB ライブラリ関数

以下の内容は、選択したランタイムライブラリ構成がファイル記述子をサポートしている場合にのみ有効です。ランタイムライブラリ構成の詳細については、「*DLIB ランタイム環境*」を参照してください。

NULL マクロ (7.1.6)

NULL マクロは、0 に定義されています。

assert 関数で出力される診断 (7.2)

assert() 関数で出力される診断は、以下のとおりです。

filename:*linenr expression* -- assertion failed

この診断は、パラメータがゼロに評価される場合に出力されます。

定義域エラー (7.5.1)

数学関数で定義域エラーが発生すると、NaN（非数）が返されます。

浮動小数点値のアンダフロー時に ERANGE に設定する errno (7.5.1)

数学関数は、アンダーフロー範囲エラー発生時に、整数式 errno を ERANGE（errno.h で定義されているマクロ）に設定します。

fmod 関数の機能 (7.5.6.4)

`fmod()` の 2 番目の引数がゼロの場合、関数は `NaN` を返します。`errno` は `EDOM` に設定されます。

signal 関数 (7.7.1.1)

シグナル関連のライブラリはサポートされていません。

注 : 低レベルインタフェース関数はライブラリには存在しますが、これらの関数は何も実行しません。アプリケーション固有のシグナル処理を実装する場合は、テンプレートソースコードを使用してください。139 ページの *signal と raise* を参照してください。

行を終了する文字 (7.9.2)

`stdout` ストリーム関数は、`newline` または `end of file (EOF)` のどちらかを改行文字として認識します。

空白行 (7.9.2)

`stdout` ストリームで改行文字直前に書き込まれた空白文字は保持されます。`stdout` ストリームで書き込まれた行を `stdin` ストリームで読み取る方法はありません。

バイナリストリームに書き込まれたデータへの NULL 文字の追加 (7.9.2)

バイナリストリームにライトされたデータには、NULL 文字は追加されません。

ファイル (7.9.3)

追加モードのストリームのファイル位置インジケータが最初にファイルの先頭または末尾に配置されているかどうかは、低レベルファイルルーチンのアプリケーション固有の実装に依存します。

テキストストリームへのライトにより、対応するファイルでその書込み位置以降が切り捨てられるかどうかは、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。135 ページの *ファイル I/O* を参照してください。

ファイルのバッファ化の特徴は、アンバッファされたファイル、ラインバッファされたファイル、完全にバッファされたファイルが実装でサポートされるということです。

ゼロ長のファイルが実際に存在するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

有効なファイル名の作成規則は、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

同じファイルを同時に何度も開くことができるかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

remove 関数 (7.9.4.1)

開いたファイルに対する削除処理の結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。135 ページの *ファイル I/O* を参照してください。

rename 関数 (7.9.4.2)

ファイルの名前を既存のファイル名に変更した結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。135 ページの *ファイル I/O* を参照してください。

printf 関数の %p (7.9.6.1)

`printf()` の `%p` 変換指定子（出力ポインタ）の引数は、`void *` 型として処理されます。値は、`%x` 変換指定子の場合と同様に、16 進数として出力されます。

scanf 関数の %p (7.9.6.2)

`scanf()` の `%p` 変換指定子（スキャンポインタ）は、16 進数を読み取り、`void *` 型の値に変換します。

scanf 関数の読み込み範囲 (7.9.6.2)

`-`（ダッシュ）文字は、常に範囲記号として処理されます。

ファイル位置エラー (7.9.9.1, 7.9.9.4)

ファイル位置エラー発生時に、関数 `fgetpos` および `ftell` store `EFPOS` を `errno` に格納します。

perror 関数で生成されるメッセージ (7.9.10.4)

生成されるメッセージは、次のとおりです。

`usersuppliedprefix``:`*`errormessage`*

ゼロバイトのメモリ割り当て (7.10.3)

calloc()、malloc()、realloc() の各関数では、引数としてゼロを指定できます。メモリが割り当てられ、そのメモリへの有効なポインタが返され、メモリブロックを後で realloc を使用して修正できます。

abort 関数の振る舞い (7.10.4.1)

abort() 関数では、ストリームバッファをフラッシュしません。また、ファイル処理はサポートされていないため、ファイル処理は実行しません。

exit 関数の振る舞い (7.10.4.3)

exit 関数では、main 関数が cstartup に返した値が引数として引き渡されます。

環境 (7.10.4.4)

使用可能な環境名 / 環境リストの変更方法については、138 ページの*環境の操作*を参照してください。

system 関数 (7.10.4.5)

コマンドプロセッサの動作は、system 関数の実装によって異なります。
138 ページの*環境の操作*を参照してください。

strerror 関数が返すメッセージ (7.11.6.2)

strerror() が返すメッセージは、次のとおりです。

引数	メッセージ
EZERO	no error
EDOM	domain error
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 || >99	unknown error
その他	error *nnn*

表 55: strerror() が返すメッセージ— IAR DLIB ライブラリ

タイムゾーン (7.12.1)

ローカルの時間帯と夏時間の実装については、139 ページの*時間*を参照してください。

clock 関数 (7.12.2.1)

システムクロックがカウントを開始する地点は、`clock` 関数の実装によって異なります。139 ページの*時間*を参照してください。

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

あ

い

う

お

か

き

く

こ

さ

し

す

せ

そ

た

ち

つ

て

と

ね

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

ら

り

る

れ

ろ

わ

記号

数字