IAR Embedded Workbench

IAR C/C++ 開発ガイド

コンパイルおよびリンク

Arm Limited

Arm® コア

DARM-28-J

IAR SYSTEMS

改版情報
第 28 版：2022 年 6 月

部品番号：DARM-28-J

本ガイドは、IAR Embedded Workbench® for Arm のバージョン 9.30.x に適用する。

内部参照：BB10、csrct2010.1、V_110411、INIT。

目次（章）

目次

表

はじめに

へようこそ『*Arm 用 IAR C/C++ 開発ガイド*』。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、32 ビットまたは 64 ビット Arm コア用の C/C++ 言語を使用して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。

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。

- 使用する Arm コアの命令セットとアーキテクチャ（チップメーカのドキュメントを参照）
- 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、44 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

IAR C/C++ Compiler および Linker for Arm 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。

この製品を初めてお使いになる場合は、製品の IAR Information Center にあるチュートリアルをまず参照することをお勧めします。IAR Embedded Workbench の使用を開始するにあたって、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、ツールの概要、プログラミング言語、利用可能なデバイスサポート、Arm コアおよびデバイス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 ビルドツールの導入について説明します。
- 「*組み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み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*」は、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.s` 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。呼び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*」C++ サポートの 2 つのレベルの概要を提供します。
- 「*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。
- 「*組み込みアプリケーション用の効率的なコーディング*」では、組み込みアプリケーションのための効率的なコードにコンパイルするコードを書く方法について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、呼び出し構文、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。
- 「*組み込み関数*」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。
- 「*C/C++ 標準ライブラリ関数*」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*」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*」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。
- 「*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

ユーザドキュメントは、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*インストールとライセンス・クイックリファレンス*および*ライセンスガイド*をご覧ください。
- プロジェクト管理および構築のために IDE の使用は、『*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® デバッガの使用および C-RUN ランタイムエラー解析は、『*Arm 用 C-SPY® デバッグガイド*』を参照してください。
- IAR C/C++ Compiler for Arm 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。
- IAR アセンブラ for Arm 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。
- 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。
- I-jet の使用法については、『*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。
- IAR J-Link および IAR J-Trace を使用については、『*J-Link/J-Trace ユーザーガイド*』を参照してください。
- IAR Embedded Workbench for Arm 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行については、『*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 プロジェクト管理およびビルド
- IAR C-SPY® デバッガを使用したデバッグ

- IAR C/C++ Compiler および Linker
- IAR アセンブラ
- C-STAT

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++*. O'Reilly & Associates.
- Furber, Steve. *ARM System-on-Chip Architecture.* Addison-Wesley.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual.* Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.
- Mann, Bernhard. *C für Mikrocontroller.* Franzis-Verlag.［ドイツ語］
- Meyers, Scott. *Effective C++:* 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.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley.

isocpp.org ウェブサイトにも、C++ プログラミングの推奨の本の一覧があります。

Web サイト

推奨 Web サイト :

- チップ製造元のウェブサイト。
- Arm limited 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**) 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**。

- C++ プログラミング言語の Web サイト、**isocpp.org** このウェブサイトには C++ プログラミングの推奨の本の一覧が掲載されています。
- C および C++ 参照のウェブサイト、**en.cppreference.com**。

表記規則

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

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

表記規則

IAR システムズのドキュメントでは、以下の表記規則を使用します。

スタイル	用途
computer	• ソースコードの例、ファイルパス。 • コマンドライン上のテキスト。 • 2 進数、16 進数、8 進数。
parameter	パラメータとして使用される実際の値を表すプレースホルダ。たとえば、*filename*.h の場合、*filename* はファイルの名前を表します。
[option**]**	リンカまたは stack usage control 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ありませんが、[、]、{、または } はいずれもディレクティブ構文の一部です。
{option**}**	リンカまたは stack usage control 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ありませんが、[、]、{、または } はディレクティブ構文の一部です。
[option]	コマンドラインオプション、pragma ディレクティブ、またはライブラリ ファイル名のオプション部分。
[a|b|c]	代替の選択肢を持つコマンドラインオプション、pragma ディレクティブ、またはライブラリ ファイル名のオプション部分。
{a|b|c}	代替の選択肢を持つコマンドラインオプション、pragma ディレクティブ、またはライブラリ ファイル名の必須部分。
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。

表 1: 本ガイドで使用されている表記規則

スタイル	用途
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 インター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
IAR Embedded Workbench® for Arm	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for Arm	IDE
IAR C-SPY® デバッガ for Arm	C-SPY、デバッガ
IAR C-SPY® シミュレータ for Arm	シミュレータ
IAR C/C++ コンパイラ for Arm	コンパイラ
IAR アセンブラ for Arm	アセンブラ
IAR ILINK リンカ ™	ILINK、リンカ
IAR DLIB ランタイム環境 ™	DLIB ランタイム環境

表 2: このガイドで使用されている命名規約

32 ビットモードでは、命令セット T32/T と A32 用に設定された IAR Embedded Workbench for Arm を使用します。

64 ビットモードでは、命令セット A64 用に設定された IAR Embedded Workbench for Arm を使用します。

詳細については、57 ページの「*例外モード*」を参照してください。

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
- 組み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
- 組み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
- 例外モード
- 組み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 ベースの組み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® は強力な統合開発環境 (IDE) で、完全な組み込みアプリケーションプロジェクトを開発および管理できます。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。

構築済みの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

IAR C/C++ コンパイラ for Arm は、C および C++ 言語標準機能に加えて、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

IAR アセンブラ for Arm は、多用途の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。

IAR アセンブラ for Arm では、Arm Limited 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

IAR ILINK リンカ for Arm は、組み込みコントローラー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、大規模、再配置可能な入力、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の混合プログラムの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し生成するので、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。よって 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

IAR C/C++ コンパイラ for Arm は以下をサポートしています。

- C：組み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 － C18 としても知られます。本ガイドでは、この規格を *C 規格*と呼びます。
  - C89 － C94、C90、および ANSI C としても知られます。この規格は MISRA-C が有効なときに必須です。
- C++17 とも呼ばれる標準 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。標準 C++ の IAR 実装は、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。また異なる 2 つの標準ライブラリの選択を提供します。
  - DLIB は C++14 ライブラリで、2 つの設定があります。ノーマルとフルです。ノーマル設定は小さく機能が少なくなります。
  - Libc++ は C++17 ライブラリです。1 つの設定のみで、DLIB ライブラリのフル設定に関連しています。

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。静的およびリラックスした両方のモードには、今後の C/C++ 標準バージョンの機能をサポートすることが含まれる可能性があります。

C の詳細については、「*C の使用*」を参照してください。

C++ の詳細は、「*C++ の使用*」の章を参照してください。

コンパイラが言語の実装定義をどのように処理するかについては、「*C 規格の処理系定義の動作*」の章および「*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。

注 : コンパイラが生成するオブジェクトコードは、コア間でバイナリレベルで互換性があるとは限りません。そのため、プロセッサを指定する必要があります。デフォルトコアは Cortex-M3 です。

32 ビット Arm デバイス

Armv4、Armv5、Armv6、Armv7、および Armv8 世代に属するほとんどのコアおよびデバイスはサポートされています（Armv8.1-M を含む）。

Arm アーキテクチャプロファイル

Armv7 以降、Arm アーキテクチャは 3 つの *architectural profiles* で構成されています。

- A プロファイル、AArch32 と互換性のある Cortex-A シリーズで実装されている *application* プロファイル。
- R プロファイル、Cortex-R シリーズで実装される *real-time* プロファイル。
- M プロファイル、Cortex-M シリーズのほとんどのコアで実装される *microcontroller* プロファイル。

32-bit Arm プロパティ

- 32 ビット Arm デバイス（M プロファイル以外）には、ユーザーモード、割り込み (FIQ, IRQ) モード、スーパーバイザーモードなどの CPU モードがあります。
- 32 ビット Arm デバイスには、3 つの命令セットがあります（すべてのコアには 3 つの命令セットはありません）。
  - Thumb (T)、16 ビットワイド命令。コンパクトなコードに使用。
  - Arm (A32)、32 ビットワイド命令。より速いコードに使用。
  - Thumb-2 (T32)、Thumb 命令セット用に拡張した 32 ビットワイド命令。
- アドレスは常に 32 ビットです。
- レジスタセットは、13 個の汎用 32 ビットレジスタから構成されます。
- 32 ビット Arm デバイスには、VFP（ベクタ浮動小数点）や SIMD（シングルインストラクションの複数のデータ）などのコプロセッサがあります。コプロセッサには、16 個の 64 ビットレジスタと 32 個の 128 ビットレジスタがあります。
- 32 ビット Arm デバイスは、オブジェクトおよびイメージフォーマットとして、32 ビット ELF を使用します。

64 ビット Arm デバイス

- Armv8.4-A 以前のアーキテクチャをベースにした 64 ビット Arm デバイス、および Armv8-R AArch64 をサポートしています。
- Armv8-A/R 世代は 2 つの実行状態を定義します。AArch32 および AArch64 です。（一部のコアは、両方の実行状態をサポートしていません。）

AArch32 実行状態

32 ビット AAarch32 実行状態は Armv7-A アーキテクチャと互換性があり、同じ CPU モード、命令セット、レジスタセットなどがあり、VFP および高度な SIMD があります。この実行状態では、CPU は常に **32 ビットモード**で実行します（57 ページの「*例外モード*」参照）。

AArch64 実行状態

- AArch64 は、4 つのレベルの権限をサポートしています。
  - EL0、実行レベル 0、ユーザーモード。
  - EL1、実行レベル 1、OS モード。
  - EL2、実行レベル 2、ハイパーバイザーモード。オプション。
  - EL3、実行レベル 3、安全なモニターモード。オプション。

  CPU は高い EL から低い EL にトラバースすることができ、その間に AArch64 から AArch32 の実行状態に変更できます。
- AArch64 状態では、CPU は常に **64 ビットモード**で実行します。57 ページの「*例外モード*」を参照してください。
- AArch64 は、32 ビット命令がある 1 つの命令セット、A64 をサポートします。
- アドレスは常に 64 ビットです。
- レジスタセットは、31 個の 64 ビットワイド汎用レジスタがあります。
- A VFP および NEON モジュールは常に表示されます。モジュールには、128 ビットワイドの 32 個のレジスタがあります。
- AArch64 の定義されたデータモデルが 3 つあります。
  - ILP32。32 ビットの `long` およびポインタ型、および 32 ビット `wchar_t` 型があります。オブジェクトおよびイメージフォーマットとして、32 ビット ELF を使用します。
  - LP64。64 ビットの `long` およびポインタ型、および 32 ビット `wchar_t` 型があります。オブジェクトおよびイメージフォーマットとして、64 ビット ELF を使用します。
  - LLP64。32 ビットの `long` 型、64 ビットポインタ型、および 16 ビット `wchar_t` 型があります。IAR Embedded Workbench for Arm ではこのデータモデルをサポートしていません。

注 : ILP32 データモデルを使用している AArch64 の生成したコードは、LP64 データモデルを使用して生成したコードとリンクできません。AArch32 と AArch64 の生成されたコードは、どちらもともにリンクできません。

定義済みサポートファイル

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の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、すべてのサポートされるデバイス用の既製リンカ設定ファイルが含まれます。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、100 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*の章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

アプリケーションのサンプルは IAR Embedded Workbench に同梱されています。これらのサンプルを使用して、IAR システムズの開発ツールを使用する準備を行えます。また、これらのサンプルを基にして、アプリケーションプロジェクトを開始することもできます。

サンプルは現状のまま提供されます。既製のワークスペースファイルが、ソースコードファイルおよび関連する他のすべてのファイルとともに提供されます。サンプルプロジェクトを実行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。

例外モード

IAR Embedded Workbench for Arm は、実行モードによって 32 ビットおよび 64 ビットの Arm アーキテクチャをサポートします。

32 ビットモードは、Armv4/5/6/7 コアまたは Arm v8-A コアでの AArch32 拡張状態のいずれかで、命令セット T32/T および A32 のコードを生成してデバッグするように構成した IAR Embedded Workbench for Arm を使用します。32 ビットモードでは、A32 および T32/T 命令セットの両方を使用し、jump 命令を使用して切り替えることができます。

64 ビットモードは、Arm v8-A コアでの AArch62 実行状態で、命令セット 64 のコードを生成してデバッグするように構成した IAR Embedded Workbench for Arm を使用します。64 ビットモードのコードは、32 ビットモードのコードにトラップし、コードを戻すことができます。ただし、IAR 変換ツールは、単一のリンクしたイメージを使用した、この切り替えをサポートしていません。A32/T32/T コードと A64 コードの切り替えは、複数のイメージを使用して実行する必要があります。例えば、64 ビットモードを使用している OS は 64 ビットまたは 32 ビットモードのいずれかでアプリケーションを開始できます。

AArch32 実行状態は、Arm v7 アーキテクチャと互換性があります。AArch32 実行状態は、AArch64 実行状態内でエミュレーションされます。

組み込みシステム用の特殊サポート

ここでは、コンパイラでさまざま Arm コアおよびデバイスの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、307 ページの「*-e*」を参照してください。

詳細については、「*拡張キーワード*」を参照してください。*データ記憶*および*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保する場合に便利です。

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*」の章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、コンパイル時刻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*」の章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、組み込み関数、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、177 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。

組み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み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み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み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。それらを支援するためのコンパイラオプション、拡張したキーワード、pragma ディレクティブなどが含まれています。

メモリのマッピング

組み込みシステムには、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。

組み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、254 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（100 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ーと接続されている場合、シグナルインターフェイスを初期化とコントロールする必要があり、例えばチップ選択ピンを使用して外部割り込み信号を検出と処理します。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。53 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、268 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*割り込み*を使用します。通常、コード中で割り込みが発生すると、コアはすぐにコードの実行を停止し、その代わりに割り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り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 で割り込みルーチンを記述できるということです（80 ページの「*Cortex-M デバイスの割り込み関数*」と 81 ページの「*Arm7/9/11、Cortex-A、Cortex-R の割り込み関数*」を参照）。87 ページの「*64 ビットモードの例外関数*」を参照してください。

システム起動

すべての組み込みシステムでは、アプリケーションの `main` 関数が呼び出される前に、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。

組み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、システム起動コードとともにリンカに

よって行われます。詳細については、65 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、組み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。

171 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。

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、242 ページの「*AEABI への準拠*」を参照してください。

Arm 用の IAR ビルドツールのバージョン 8.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。詳しくは、『*IAR Embedded Workbench® IDE ユーザガイド*』を参照してください。

詳細については、130 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、個々のビルドツール（コンパイラ、アセンブラ、リンカ）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。

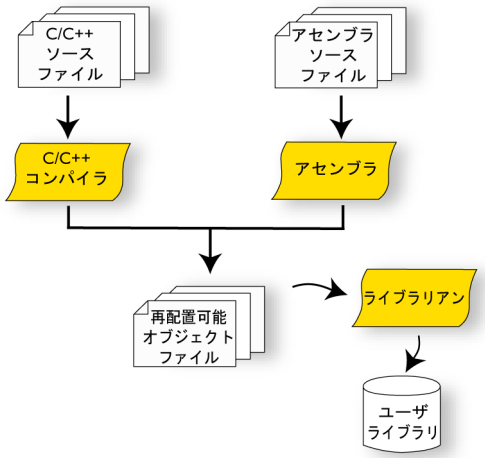
実際のプロセスをより理解できるように、IAR インフォメーションセンタで使用できるチュートリアルをご覧ください。

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。C/C++ コンパイラと 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 : コンパイラは、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。必要に応じて、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圧縮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

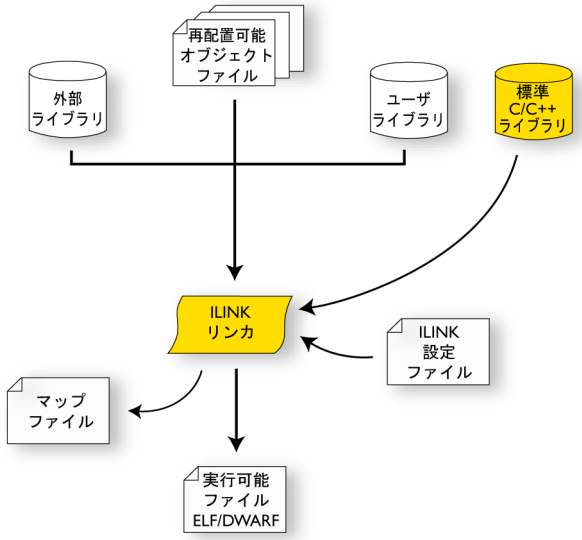
IAR コンパイラおよびアセンブラにより生成されるオブジェクトファイルおよびライブラリの再配置可能モジュールは、そのまま実行することはできません。これらが実行可能なアプリケーションとなるには、*リンク*が必要です。

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点に注意してください。

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。

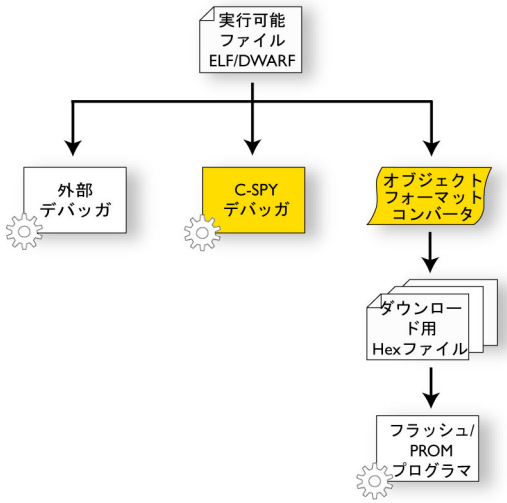
リンカにより実行される手順について詳しくは、97 ページの「*リンクプロセスの詳細*」を参照してください。

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（591 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

このセクションでは、組み込みアプリケーションの実行の概要を以下の 3 つのフェーズに分けて説明します。

- 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。簡潔にすると、初期化フェーズは次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` と、必要に応じて、ユーザが用意する追加の低レベル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 /RESTART 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 関数が呼び出される前に、すべてのグローバル（静的にリンクされた）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。

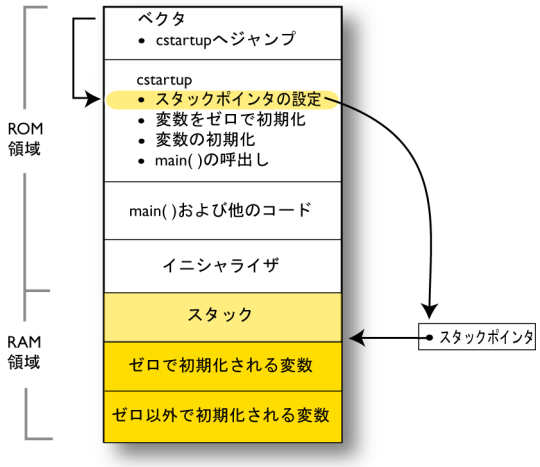
- アプリケーション初期化

  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、さまざまな割り込みの設定、通信の初期化、デバイスの初期化などが含まれます。

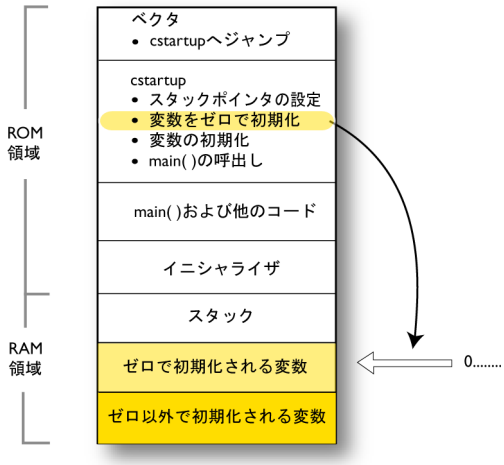
ROM/ フラッシュベースのシステムでは、定数や関数がすでに ROM に配置されています。また、リンカにより、利用可能な RAM は、すでに変数、スタック、ヒープなどの異なるエリアに分割されています。RAM に配置されたすべてのシンボルは、main 関数が呼び出される前に初期化される必要があります。

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。

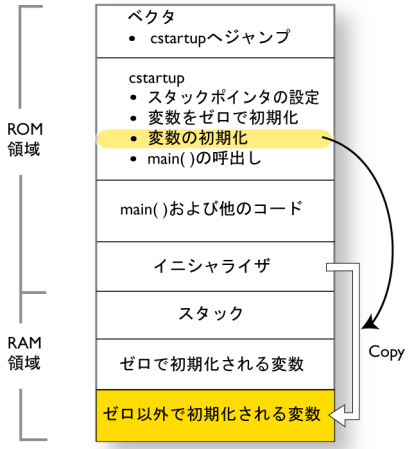


2 次に、ゼロ初期化されるメモリがクリアされます。すなわち、これらのメモリにゼロが埋め込まれます。



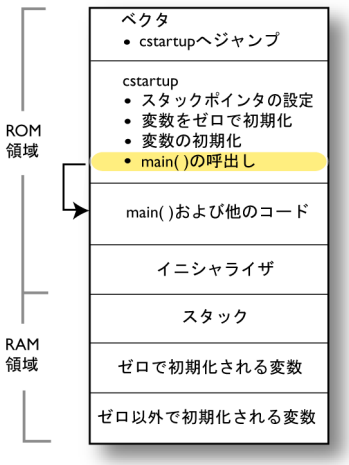
一般的に、これは*ゼロで初期化されるデータ*などのデータで、たとえば `int i = 0;` などで宣言される変数です。

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。



次に、C++ オブジェクトなどの動的に初期化された静的オブジェクトが構築されます。

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、155 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、103 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

組み込みアプリケーションのソフトウェアは、通常、割り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。割り込み駆動型システムの場合、割り込みは、通常、main 関数の開始時に初期化されます。

リアルタイムで動作し、応答性が重要なシステムでは、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 (RTOS) 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、組み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、quick_exit、または abort のいずれかを呼び出すか、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++ デストラクタが呼び出され（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、157 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（70 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。詳細なビルド情報を有効にするには、［**ビルド**］メッセージ画面で右クリックして、コンテキストメニューで［**すべて**］を選択します。

基本的なプロジェクト設定

この章では、使用している Arm デバイスのベストコードを生成するための基本設定の概要について説明します。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。コマンドラインで、それぞれのオプションを別々に指定する必要がありますが、IDE を使用する場合は、ほとんどのオプションが、一部の基本的なオプションの設定をもとに自動的に設定されます。

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（137 ページの「*ランタイム環境の設定*」を参照）。
- 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、その他の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、それぞれの *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。

32 ビットモードプロセッサ構成

コンパイラに最適なコードを生成させるためには、使用している Arm コアに合わせて設定する必要があります。

プロセッサ選択

IAR C/C++ コンパイラ for Arm はほとんどの 32 ビット 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、コア間でバイナリレベルで互換性が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 Cortex-M3 です。

［実行モード］は **32 ビット**です。**Processor variant** オプションの設定の詳細については、『*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。

`--cpu` オプションを使用して、オプションを使用して、Arm コアを指定します。構文情報については、296 ページの「*--arm*」および 339 ページの「*--thumb*」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。

`--Fpu` オプションを使用して、Arm コアを指定します。構文については、310 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。構文については、308 ページの「*--endian*」を参照してください。

64 ビットモードプロセッサ構成

コンパイラに最適なコードを生成させるためには、使用している Arm コアに合わせて設定する必要があります。

プロセッサ選択

IAR C/C++ コンパイラ for Arm をサポートする 64 ビット Armv8-A コアを選択します。コンパイラが生成するオブジェクトコードは、コア間でバイナリレベルで互換性が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。

［**実行モード**］は［**64 ビット**］です。**Processor variant** オプションの設定の詳細については、『*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。

`--Cpu` オプションを使用して、Arm コアを指定します。構文情報については、297 ページの「*--cpu*」および 294 ページの「*--aarch64*」を参照してください。

データモデル

生成したコード ILP32 または LP64 を使用するために、データモードを選択します。

［データモデル］オプションの設定の詳細については、『*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。

`--abi` オプションを使用して、データモデルを指定します。構文については、294 ページの「*--abi*」を参照してください。

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、展開や帰納変数の削除などのループ最適化も実行します。

複数の最適化レベルから選択できます。最高レベルでは、目的の*サイズ*、*速度*、または*バランス*の間で異なる最適条件を選択できます。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、*組み込みアプリケーション用の効率的なコーディング*の章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憶領域
- ヒープ上の動的メモリ

概要

32 ビットの Arm コアは、4GB のシーケンシャルメモリ（範囲は `0x0` ～ `0xFFFF'FFFF`）を持っています。64 ビットの Arm コアは、16EiB のシーケンシャルメモリ（範囲は `0x0~0xFFFF'FFFF'FFFF'FFFF`）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。関数が呼び出し元に戻ると、このメモリ空間は無効になります。詳細については、74 ページの「*自動変数とパラメータの記憶領域*」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 *static* 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。

注 : 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、75 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憶領域

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は解放されます。

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）
- 割り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（呼び出し先保存レジスタ）
- スタック保護される関数が使用するカナリー。93 ページの「*スタック保護*」を参照してください

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、現在の関数を呼び出した関数やその関数を呼び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、割り込みが発生した場合に、呼び出し先の割り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。

228 ページの「*スタックについて*」および 120 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。

関数は自身を呼び出すことができます（再帰関数）。また、呼び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び出し、その関数がさらに別の関数を呼び出す場合など、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。

各ヒープのサイズの設定方法は、120 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、慎重に設計される必要があります。

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
- 32 ビット Arm と Thumb コード
- 64 ビット A64 コード
- RAM での実行
- Cortex-M デバイスの割り込み関数
- Arm7/9/11、Cortex-A、Cortex-R の割り込み関数
- 64 ビットモードの例外関数
- インライン関数
- スタック保護
- TrustZone インターフェース

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。

- 32 ビット CPU モード Arm と Thumb 用にコードの生成
- A64 命令セットのコードの生成
- RAM で関数を実行
- 異なるデバイスに割り込み関数を記述
- 関数インライン化の制御
- 関数の最適化の促進
- ハードウェア機能にアクセス
- TrustZone のインターフェース関数の作成

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、組み込み関数で実現します。

最適化の詳細は、251 ページの「*組み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は、*組み込み関数*の章を参照してください。

32 ビット Arm と Thumb コード

32 ビットの場合、IAR C/C++ コンパイラ for Arm は、32 ビットの Arm、または 16 ビットの Thumb または Thumb2 命令セットのどちらか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。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。

関数呼び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 `0x0-0xFFFF'FFFF` の連続する 4 GB 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 4 MB 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、400 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 関数の呼び出し、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。

64 ビット A64 コード

64 ビットモードの場合、IAR C/C++ コンパイラ for Arm は、A64 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 `--aarch64` や `--abi` 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。

関数呼び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 `0x0-0xFFFF'FFFF'FFFF'FFFF` の連続する 16 Eib 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 64 MB という制限があります。

コードポインタのサイズは 4 または 8 バイトで、データモデルにより異なります。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、400 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 関数の呼び出し、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、155 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。割り込みベクタおよび割り込みサービスルーチンが RAM に格納されていない限り、割り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、124 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 デバイスの割り込み関数

Cortex-M は、以前の Arm アーキテクチャとは割り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 の割り込み

Cortex-M では、割り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` と `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。他の割り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。

組み込み関数 `__get_CPSR` と `__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、270 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。

FPU を備えた Cortex-M の割り込み

FPU を備えた Cortex-M コアの場合、システムレジスタビット FPCCR.ASPEN を 1 に設定して浮動小数点レジスタ（S0–S15 および FPSCR）の自動状態保持を有効にする必要があります。これにより、浮動小数点レジスタを使用している場合でも通常の関数と同じ方法で、割り込みサービスルーチンにエンターし、リターンします。

浮動小数点コンテキスト保存手順 (FPCCR.ASPEN=0) は、次の場合省略することができます。

- 1 つのアプリケーションタスクのみが FPU を使用し、割り込みハンドラは FPU を使用しない。または
- アプリケーションタスクは FPU を使用せず、割り込みハンドラのみで、FPU を使用する。

オペレーティングシステムなしで実行されているアプリケーションは、単一のアプリケーションタスクと見なされます。SVC_Handler を含むすべてのハンドラは影響を受け、ソフトウェア割り込み関数（__svc キーワードで宣言済みの関数）も影響を受けます。

Arm7/9/11、Cortex-A、Cortex-R の割り込み関数

IAR C/C++ コンパイラ for Arm は、Arm7/9/11、Cortex-A、および Cortex-R デバイスの割り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、__nested、
- 組込み関数 :__enable_interrupt、__disable_interrupt、__get_interrupt_state、__set_interrupt_state

注 : Cortex-M は、他の Arm デバイスとは割り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、80 ページの「*Cortex-M デバイスの割り込み関数*」を参照してください。

割り込み関数

組み込みシステムでは、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り込みを使用します。

割り込みサービスルーチン

通常、コード中で割り込みが発生すると、コアはすぐにコードの実行を停止し、その代わりに割り込みルーチンの実行を開始します。割り込み処理の完了後、割り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、割り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、割り込み、ソフトウェア割り込み、高速割り込みをサポートします。割り込みタイプごとに、割り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。

割り込みベクタと割り込みベクタテーブル

各割り込みルーチンは、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。割り込みベクタは、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、ベクタテーブルには無限ループする*デフォルトの割り込みハンドラ*が定義されています。明示的な割り込みサービスルーチンを持たない各割り込みソースに対しては、デフォルトの割り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り込みハンドラを上書きします。

割り込み関数の定義 — 例

割り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。以下に例を示します。

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

割り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 : 割り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。

割り込みと C++ メンバ関数

静的メンバ関数だけが割り込み関数になれます。非静的メンバ関数を呼び出す際には、オブジェクトに割当てる必要があります。割り込みが発生し、割り込み関数が呼び出されるときは、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

すべての割り込み関数およびソフトウェア割り込みハンドラは、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
SVC_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()
{
}
```

割り込み関数は、C リンケージを持つ必要があります。詳細については、192 ページの「*呼び出し規約*」を参照してください。

C++ を使用する場合の割り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。

割り込みおよび高速割り込み

割り込みおよび高速割り込み関数は、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:

- 割り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 415 ページの「*__irq*」および 452 ページの「*type_attribute*」を参照してください。
- 高速割り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 414 ページの「*__fiq*」および 452 ページの「*type_attribute*」を参照してください。

注 : 割り込み関数 (`irq`) および高速割り込み関数 (`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` または `svc`）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、`R0-R3` の 4 つのレジスタのみをパラメータで使用できます。また、`R0-R1` のレジスタのみをリターン値に使用できます。

ネスト割込み

割り込みは、割り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。割り込みハンドラが割り込みを再び有効にし、関数を呼び出して別の割り込みが発生した場合、`LR` に格納されている割り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 番目の割り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。ネストされた割り込みを処理するときに必要な必須ステップを割り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。ネストされた割り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。

ネストされた割り込みを可能にするコンパイラにより生成された割り込みハンドラは、IRQ 割り込みのみでサポートされます。FIQ 割り込みは、迅速な提供を目的としているので、通常、ネストされた割り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 ベクタ割り込みコントローラ (VIC) でネストされた割り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;

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

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

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

  /* この割り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。

ソフトウェア割り込み

ソフトウェア割り込み関数は、ソフトウェア割り込みハンドラ（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（呼び出され）ます。また、引数を使用し、値を返します。このため、他の割り込み関数より少し複雑になります。ここでは、ソフトウェア割り込み関数を呼び出すメカニズムと、ソフトウェア割り込みハンドラが実際のソフトウェア割り込み関数をディスパッチする方法について説明します。

ソフトウェア割り込み関数の呼び出し

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び出すには、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、svc_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 __svc キーワードおよび #pragma svc_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。

__svc 関数は、たとえば、以下のように宣言できます。

```
#pragma svc_number=0x23
__svc int svc_function(int a, int b);
```

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

ソフトウェア割り込み関数は、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び出し規則に従います（192 ページの「*呼び出し規約*」を参照）。

詳細については、422 ページの「*__svc*」と 451 ページの「*svc_number*」を参照してください。

ソフトウェア割り込みハンドラと関数

割込みハンドラ（たとえば SVC_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。割り込みハンドラは、割り込みベクタから呼び出され、ソフトウェア割り込み番号の取得と適切なソフトウェア割り込み関数の呼び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び出す方法はないため、SVC_Handler はアセンブラ内に記述する必要があります。

ソフトウェア割り込み関数

ソフトウェア割り込み関数は、C/C++ で記述できます。__svc 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り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 svc_number ディレクティブは必要ありません。

詳細については、422 ページの「*__svc*」を参照してください。

ソフトウェア割り込みスタックポインタの設定

ソフトウェア割り込みをアプリケーションで使用する場合には、ソフトウェア割り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、割り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（120 ページの「*スタックメモリの設定*」を参照）。

割り込み処理

割り込み関数は、外部イベントが発生するときに呼び出されます。通常、別の関数の実行中に直ちに呼び出されます。割り込み関数の実行が終了すると、元の関数に戻ります。割り込まれた関数の環境の復元が必要で、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。

割り込みが発生すると、以下の処理が実行されます。

- 動作モードが、特定の例外に合わせて変化します。
- 例外の発生した次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 に保存されます。
- 新たな割り込み要求は、CPSR のビット 7 が設定され無効になり、例外が高速割り込みの場合、さらに高速割り込みが CPSR のビット 6 が設定され無効になります。
- PC が、対応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 の割り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。割り込みの開始位置として使用されるメモリエリアは、割り込みベクタテーブルと呼ばれます。割り込みベクタの内容は、通常、割り込みルーチンにジャンプする分岐命令です。

注 : 割り込み関数により割り込みが有効にされると、割り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、これらを割り込みルーチンで格納する必要があります。__nested キーワードが使用されている場合、この処理は自動的に行われます。

64 ビットモードの例外関数

コンパイラは、64 ビットモードの例外関数の書き込みに関連する基本関数を提供します。

- 拡張キーワード __exception、__nested、および __svc
- 組み込み関数 __enable_interrupt および __disable_interrupt
- 特別な関数の名前 Synchronous_Handler_A64、Error_Handler_A64、IRQ_Handler_A64、および FIQ_Handler_A64

例外関数

例外関数は外部の割り込みイベントまたは内部例外を処理するために使用されます。例外が発生すると、コアの実行されたコードが停止され例外のコードが実行し始めます。例外が処理されてた後に予期したコード環境がリストアされることは重要です。これにはプロセッサレジスタ、ステータスレジスタなどの値も含まれます。実行は、例外が発生しなかったように続行できます。

__exception は関数タイプ属性で、例外関数を定義します。リターン値に void が必要で、パラメーターを持つことはできません。使用したレジスタはすべて入口に保存され、終了にリストアされます。ERET 命令で返します。

```
__exception void func(void)
{
    /* 何らかの処理 */
}
```

例外および C++ メンバ関数

静的メンバ関数だけが例外関数になります。非静的メンバ関数を呼び出す際には、オブジェクトに割当てる必要があります。例外が発生し、例外関数が呼び出されるときは、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。

例外ベクタテーブル

IAR C/C++ コンパイラは、3 つの例外レベル EL1、EL2、および EL3 の同じ例外ベクタテーブルを使用します。例外ベクタテーブルはリンカが定義したシンボル `__eevector` から開始します。それには 16 ベクタあり、1 つのベクタは 128 バイトです。

IAR C/C++ コンパイラは、例外レベルを変更せず、または現在の例外レベル（オフセット `0x200`、`0x280`、`0x300`、および `0x380`）に `SP` を使用する、ベクタのみを定義します。それらの 4 つの定義済みベクタの名前は、`Synchronous_Handler_A64`、`Error_Handler_A64`、`IRQ_Handler_A64`、および `FIQ_Handler_A64` です。それらの名前の 1 つと `__exception` 関数で定義することで、上書きされるデフォルトの実装があります。関数がベクタに大きすぎて入らない場合は、コンパイラはエラーを発行します。関数は直接例外関数として使用できません。代わりにできること：

1 `ee` などのグローバルシンボルで始まる、アセンブラモジュールを書き込みます。シンボルは例外関数にジャンプします。

2 リンカ設定ファイルを編集します。`Synchronous_Handler_A64` などの関連する例外関数の `place at` ディレクティブを `place at address synchronous_evector { symbol ee };` で置き換えます。

デフォルトでは、例外ベクタテーブルはアドレス 2048 に配置されます。別のアドレスにそれを配置するには、以下のいずれかの方法を使用します。

- リンカオプション `--config_def` を使用してリンカ設定シンボル `__Exception_table_address` を次のように設定します。
  `--config_def __Exception_table_address=4096`
- プロジェクトが使用する、リンカ設定ファイルを編集します。

例外テーブルは 2KB でアラインメントされています。

ネストされた例外関数

例外関数はネストされます。これは入口で ELR_EL1 システムレジスタも保存します。関数が終了すると、割り込みは無効になり保存したレジスタはすべてリストアされます。

SPSR_EL1 システムレジスタは自動的に保存されません。例外後にステータスフラグを保持するには、割り込みを有効にする前に明示的に保存する必要があります。これを明示的に行うと、SPSR_EL1 の他のビットを操作できるようになります。

例：

```
#include <intrinsics.h>
__exception __nested void func(void)
{
    // All used registers + ELR_EL1 have been saved. SPSR_EL1
    // and ESR_EL1 can be saved/used.

    __enable_interrupt();

    // Do stuff

    __disable_interrupt();

    // The possibly changed SPSR_EL1 and ESR_EL1 can be restored.
    // At exit, interrupts will be disabled and then all used
    // registers are restored. 次に ERET が実行されます。
}
```

Supervisor-defined 関数

関数 type attribute __svc を使用して定義された関数は値を返し、パラメーターを持つことができます。同じレジスタを通常の関数呼び出しとして保存し、ERET 命令で返します。SVC 定義の関数は同期の例外を処理します。

```
__svc void func(void)
{
    /* 何らかの処理 */
}
```

次の例を参照してください。

Supervisor call

SVC は A64 命令で、supervisor を呼び出します。これは例外です。それは同期例外ベクタで処理されます。IAR C/C++ コンパイラは、関数宣言または定義の前にプラグマディレクティブ svc_number を使用することで、SVC 命令で関

数の呼出しに使用する、通常の呼出し命令の交換をサポートしています。提供された番号は ESR_EL1 システムレジスタに格納されます。

```
#pragma svc_number = 23
__svc int Synchronous_Handler_A64(int i)
{
    return i;
}

void f()
{
    int i = Synchronous_Handler_A64(5); // Will use an SVC
}
```

SVC 関数の想定した使用は、より高い例外レベルのより低い例外レベル呼び出しコードでコードを実行することです。

```
// ユーザーコード
#pragma svc_number = 1
int svc1(int);
#pragma svc_number = 2
int svc2(int);

int main(void)
{
    svc1(1);
}

// Supervisor code
__svc int Synchronous_Handler_A64(int a)
{
    // Get syndrome: AARCH64 SVC
    long long nr = 0;
    __asm("MRS %x0, ESR_EL1¥n" : "=r"(nr));
    int ec = (nr >> 26) & 0x3F;
    if (ec != 0x15)
        return -1;

    // Get SVC number.
    nr &= 0xFF'FFFF;

    return nr + a;
}
```

#pragma svc_number で宣言した関数は、同じ関数シグネチャを使用する必要はありません。ことなる署名が使用されている場合、パラメータを通過し正しいリターン値を扱うために、さまざまな呼出しハンドラにトランポリン

として Synchronous_Handler_A64 をアセンブラ言語に書き込む必要があります。

アドレスのリセット

デフォルトではリセットアドレスはアドレス 0 と想定されています。別のアドレスでそれを配置するには、以下のいずれかの方法を使用します。

- リンカオプション --config_def を使用してリンカ設定シンボル __Reset_address を次のように設定します。
  --config_def __Reset_address=4096
- プロジェクトが使用する、リンカ設定ファイルを編集します。

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、呼び出しによるオーバヘッドを解消するために、呼び出し元関数の本体に統合することです。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。サイズを最適化する際は、通常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。

以下に例を示します。

```
// ヘッダファイル内
static int sX;
inline void F(void)
{
  //static int sY; // static を参照できない
  //sX;            // static 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 extern 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、440 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、259 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、262 ページの「*関数インライン化*」を参照してください。

スタック保護

ソフトウェアでは、スタックバッファのオーバーフローは、プログラムが通常固定長バッファである意図したデータ構造外のプログラムのコールスタック上のメモリアドレスに書き込むときに発生します。その結果、ほとんどの場合、近くのデータが破損し、どの関数を返すかまで変更します。それが意図的であれば、それはスタックスマッシングと呼ばれます。スタックバッファオーバーフローから保護する 1 つの方法は、スタックカナリーと呼ばれ、炭鉱にカナリヤを使用することに由来しています。

IAR C/C++ コンパイラのスタック保護

IAR C/C++ コンパイラ for Arm は、スタック保護をサポートします。

スタック保護を必要な関数で有効にするには、コンパイラオプション `--stack_protection` を使用します。詳細については、337 ページの「*--stack_protection*」を参照してください。

スタック保護の IAR Systems の実装には、ヒューリスティックを使用して関数にスタック保護が必要かどうかを決定します。定義したローカル変数に配列型、または配列型として含まれる構造体型がある場合、関数にはスタック保護が必要です。さらに、ローカル変数のアドレスが関数外に伝播される場合、そのような関数もスタック保護が必要です。

関数にスタック保護が必要な場合、ローカル変数は、関数スタックブロックの一番高いところに配置するために、配列型の変数でソートされます。これらの変数の後、カナリー要素が配置されます。カナリーは関数の入口で初期化されます。初期値はグローバル変数 `__stack_chk_guard`から取得されます。関数の終了で、コードはカナリー要素がまだ最初の値を含むかどうかを検証します。含まれない場合は、関数 `__stack_chk_fail` が呼び出されます。

アプリケーションにスタック保護を使用

スタック保護を使用するには、お使いのアプリケーションでこれらのオプションを定義する必要があります。

- `extern uint32_t __stack_chk_guard`

  初めて使用する前に、グローバル変数 `__stack_chk_guard` を初期化しなければなりません。初期値はランダムにすると、より安全です。
- `__interwork __nounwind __noreturn void __stack_chk_fail(void)`

関数 __stack_chk_fail の目的は、問題を通知しアプリケーションを終了することです。

注 : この関数が返すアドレスは、失敗した関数を示すためのものです。

arm¥src¥lib¥runtime ディレクトリの stack_protection.c ファイルは、__stack_chk_guard および __stack_chk_fail の両方のテンプレートとして使用できます。

TrustZone インターフェース

Arm v8-M（**32 ビットモード**）用 TrustZone は、セキュアとセキュアでないコード間のセキュアなインターフェースを作成するには、コンパイラサポートが必要です。この目的のため、コードを生成する方法を制御する関数タイプ属性が 2 つあります :__cmse_nonsecure_entry および __cmse_nonsecure_call。詳細については、246 ページの「*Arm TrustZone*®」を参照してください。

注 : TrustZone サポートは **64 ビットモードでは**自動的に行われます。

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、組み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、C/C++、または混合 C/C++ 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、リンク先が呼び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保証します。ベニアの生成方法の詳細については、126 ページの「ベニア」を参照してください。

リンカは*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を、別々の位置に指定できます。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、52 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

注 : ライブラリには、それぞれのモジュール（ソースファイル）には 1 つの関数を含める必要があります。ライブラリの関数を自分のアプリケーションの関数で上書きする場合は、これは重要です。残りのアプリケーションから参照されている場合は、リンカにはモジュールのみが含まれています。1 つの関数が参照されていて、そのモジュールの別の関数を自分のアプリケーションで定義された関数で上書きする必要があるため、リンカに複数の関数があるライブラリモジュールが含まれている場合、リンカは定義の重複エラーを発行します。

*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。通常の組み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。

セクション属性の主要目的は、ROM に配置されるセクションと RAM に配置されるセクションを識別することです。

`ro|readonly`	ROM セクション
`rw|readwrite`	RAM セクション

各セクションでは、さらにセクションをコード、データで分けることができ、最終的に 4 つの主要カテゴリになります。

`ro code`	通常のコード
`ro data`	定数
`rw code`	RAM にコピーしたコード
`rw data`	変数

`readwrite data` には、アプリケーションの起動時にゼロに初期化されるセクションの `zi|zeroinit` サブカテゴリがあります。

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、119 ページの「*モジュールの保持*」、119 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 section の詳細は、*セクションリファレンス*の章を参照してください。

ブロックを使用して、セクションをまとめて配置することができます。543 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*が必要です。

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点に注意してください。

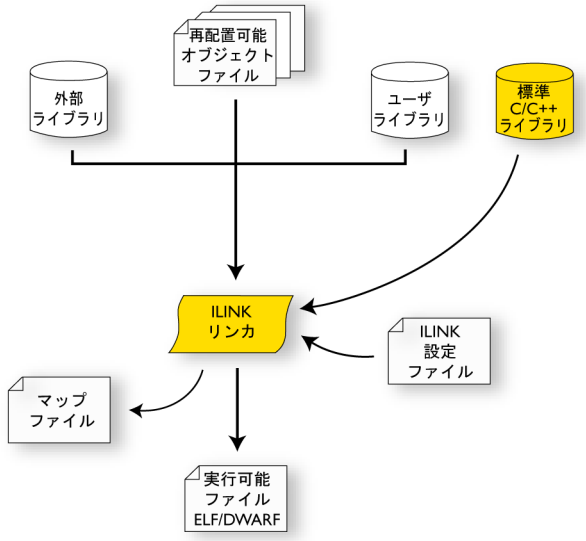
リンカはリンクプロセスに使用されます。通常、以下の手順を実行します (一部の手順は、コマンドラインオプションやリンカ設定ファイルのディレクティブによって無効化できます)。

- アプリケーションに含めるモジュールを判別します。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持つ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します。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します。`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します。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します。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します（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 に生成します。ログメッセージは、アプリケーションがそのように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。594 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 のメモリ空間がある。
- アドレス範囲 `0x0000-0x10000` に ROM メモリがある。
- アドレス範囲 `0x20000-0x30000` に RAM メモリがある。
- スタックのアライメントが 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 */
initialize by copy { readwrite }; /* RW セクションを初期化 */

/* 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
                              ROM: 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 の中に、ROM 領域および RAM 領域がそれぞれ ROM および RAM という名前で定義されています。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。

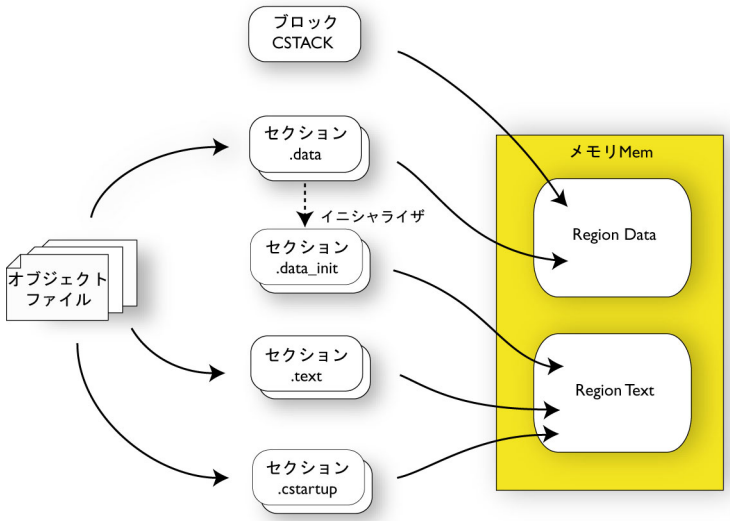
設定ファイルの最後の部分は、使用可能な領域に対してすべてのセクションを実際にどのように配置するかを定義しています。まず、起動コード（リードオンリー (readonly) セクション .cstartup に配置するように定義されている）が、ROM 領域の先頭、つまりアドレス 0x00000 に配置されます。

注 : {} 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。

注：選択セクション { `readonly section .cstartup` } は、より一般的なセクション選択 { `readonly` } よりも優先されます。

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ています。



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。

- ゼロ初期化されたセクション、または初期化されていないセクション (__no_init) は ILINK が自動的に処理します。
- 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りません。

  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ます。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ません。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。

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 */
initialize by copy { readwrite }; /* RW セクションを初期化 */

/* 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
                              ROM: 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（550 ページの「*initialize ディレクティブ*」を参照）、内容のセクション（つまり、_init サフィックス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」の集合に組み込まれます。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、115 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、ルーチンを呼び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。

559 ページの「*section-selectors*」を参照してください。

スタック使用量解析

このセクションでは、リンカを使用したスタック使用量解析の実行方法を説明します。

arm¥src ディレクトリに、スタック使用量解析を実演したサンプルプロジェクトがあります。

スタック使用量解析の概要

適切な状況下では、リンカによってプログラムの開始から割り込み関数、タスクなどの順に、各コールグラフの最大スタック使用量が正確に計算されます（別の関数から呼び出されない各関数、つまりルート）。

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。

この解析は、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、間接的な呼び出し（関数ポインタを使用した呼び出し）がアプリケーション内に

ある場合、各呼び出しコール関数から呼び出しが可能な関数のリストを提供する必要があります。

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。

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、スタック使用量制御ファイルで `check that` ディレクティブを使用できます。

スタック使用量解析の実行

1 スタックの使用量解析の有効化 :

IDE で **［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］** を選択します。

コマンドラインでは、リンカオプション `--enable_stack_usage` を使用します。

360 ページの「*--enable_stack_usage*」を参照してください。

2 リンカマップファイルの有効化 :

IDE で **［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］** を選択します。

コマンドラインでは、リンカオプション `--map` を使用します。

3 プロジェクトをリンクします。

注 : 特定の状況ではリンカはスタック使用量に関するワーニングを発します（111 ページの「*ワーニングが発行される状況*」を参照）。

4 リンカマップファイルを確認します。これには各コールグラフルートのスタック使用量の概要が記載されたスタック使用量の章が含まれています。詳細については、107 ページの「*解析結果 — マップファイルの内容*」を参照してください。

5 詳しくはコールグラフのログを解析してください（111 ページの「*コールグラフログ*」を参照）。

注 : 解析には制限および不正確となる要因があります（110 ページの「*制限*」を参照）。

より正確な結果を得るには、リンカでさらに多くの情報を指定しなければならないことがあります。108 ページの「*追加のスタック使用量情報の指定*」を参照してください。

IDE で［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］>［制御ファイル］を選択します。

コマンドラインでは、リンカオプション `--stack_usage_control` を使用します。

381 ページの「*--stack_usage_control*」を参照してください。

6 スタックに十分なメモリを割り当てたかどうかの自動チェックを追加するには、リンカ設定ファイルの `check that` ディレクティブを使用します。たとえば、`MY_STACK` というスタックブロックがある場合、次のように記述できます。

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

リンクする際、チェックに失敗するとリンカはエラーを発します。この例では、以下の合計が `MY_STACK` block のサイズを超過した場合にエラーが表示されます :

- カテゴリ `Program entry`（メインプログラム）の最大スタック使用量。
- カテゴリ `interrupt` の個々の最大スタック使用量の合計（すべての割り込みルーチンが同時に空間を必要とすると想定した場合）。
- 100 バイトの安全マージン（解析で分からないスタック使用量を想定）。

564 ページの「*check that ディレクティブ*」および 228 ページの「*スタックについて*」を参照してください。

解析結果 — 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。

各コールグラフのルートは、`check that` ディレクティブで便利な計算を有効化するために、コールグラフのルートカテゴリに属します。

追加のスタック使用量情報の指定

追加のスタック使用量情報を指定するには、スタック使用量制御ファイル (`suc`) でスタック使用量制御ディレクティブを指定するか、ソースコードに注釈を付けます。

たとえば、以下のようにします。

- 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

得る呼び出し）を指定します。通常はアセンブラモジュールなどにスタック使用量情報がない場合に、このように指定します。suc ファイルでたとえば以下のように記述することができます。

```
function MyFunc: 32,
  calls MyFunc2,
  calls MyFunc3, MyFunc4: 16;

function [interrupt] MyInterruptHandler: 44;
```

581 ページの「*function ディレクティブ*」を参照してください。

- 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。suc ファイルでたとえば以下のように記述することができます。

  ```
  exclude MyFunc5, MyFunc6;
  ```

  580 ページの「*exclude ディレクティブ*」を参照してください。

- 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び出しの宛先一覧を指定します。これは、間接的な呼び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び出されるか正確にわかっているときに使用します。suc ファイルでたとえば以下のように記述することができます。

  ```
  possible calls MyFunc7: MyFunc8, MyFunc9;
  ```

  どの関数が呼び出されるかという情報がコンパイル時に入手可能な場合、代わりに #pragma calls ディレクティブの使用を検討してください。

  582 ページの「*possible calls ディレクティブ*」および 431 ページの「*calls*」を参照してください。

- スタック使用解析制御ディレクティブ call graph root または #pragma call_graph_root ディレクティブを使用して、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。suc ファイルでたとえば以下のように記述することができます。

  ```
  call graph root [task]: MyFunc10, MyFunc11;
  ```

  コンパイラによって割り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、suc ファイルでたとえば以下のようにディレクティブを指定します。

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

  580 ページの「*call graph root ディレクティブ*」および 432 ページの「*call_graph_root*」を参照してください。

- 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。suc ファイルでたとえば以下のように記述することができます。

  ```
  max recursion depth MyFunc12: 10;
  ```

- 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。たとえば、suc ファイルで no calls from ディレクティブを以下のように使用します。

  ```
  no calls from [file.o] to MyFunc13, MyFunc14;
  ```

- スタック使用量制御ファイルのアセンブラモジュールのスタック使用量情報を指定する代わりに、アセンブラソースのコールフレーム情報に注釈を付けることができます。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。

詳細については、「*スタック使用解析制御ファイル*」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、スタック使用情報を持たないオブジェクトモジュール内のすべての関数を識別できるとは限りません。特に、アセンブリ言語に記述されたオブジェクトモジュールや、IAR 以外のツールにより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。こうしたモジュールにスタック使用量情報を指定するには、スタック使用量制御ファイルを使用したり、アセンブリ言語モジュールの場合は、CFI ディレクティブによりアセンブラソースコードに注釈を付けてスタック使用量情報を付加することも可能です。『*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。
- フレームサイズの変更や関数呼び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、それらは呼出し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。

注 : 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上または偶然に最大コールチェーンに到達しない場合もあります。特に、C++ での仮想関数呼び出しの一連の宛先には、実際にはコード内のそのポイントから呼び出すことができない関数の実装が含まれることがあります。

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。

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。

- スタック使用量情報を持たない関数がある。
- アプリケーション内に間接的な呼び出し元があり、呼び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。
- 既知の間接的な呼び出しはないが、コールグラフルートで認識されていない呼び出されていない関数がある。
- 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び出しがある。
- 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び出されるように記述されていないものがある。

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。

- 関数の呼び出しポイントにおけるスタック使用量。
- 関数名、または単一の '-'。'-' は、関数呼び出しのない関数内（通常はリーフ関数）での使用を示します。
- 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び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
- モジュールの整合性チェック
- リンカの最適化

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

`config` ディレクトリには、サポートされているすべてのコアのリンカ設定ファイル (`*.icf`) の既成のテンプレートが含まれています。

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。すべてのテンプレートは**32 ビットモード**用です。そうでないものには記載があります。

```
/* 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、Region 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。

```
/* region ROM2 が 2 つのエリアを持つように定義する。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。

```
/* セクションに変数を配置する（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
                            ; 定数を
dc16     0xF0F0             ; 置く
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（587 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、96 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。必要ないと思われるシンボル、または実際、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や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。

保持されるシンボルと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、63 ページの「*リンク処理*」を参照してください。

32 ビットモードのアプリケーション起動

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（360 ページの「*--entry*」を参照）。

64 ビットモードのアプリケーション起動

アプリケーション起動を実行する位置は、`__Reset_address` ラベル（`cstartup` モジュールが始まる位置を決定する）で定義されます。`__iar_program_start` ラベルは同じアドレスに配置されます。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。

リセットアドレスの変更方法については、91 ページの「*アドレスのリセット*」を参照してください。

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。**64 ビットモード**では、スタックアライメントは 16 です。

スタックの情報については、228 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。**64 ビットモード**では、ヒープアライメントは 16 です。

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 回呼び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 回の呼び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、最適な圧縮方法を選択します。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。

- 初期化の抑止
- 圧縮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。

初期化の抑止

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。

圧縮アルゴリズムの選択

デフォルトの圧縮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 = lz77 { readwrite };
```

使用可能なその圧縮アルゴリズムの詳細については、550 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

手動で初期化する通常の場合は、initialize by copy ディレクティブは、アプリケーション起動時に内容を含むセクションをコピーする（パッキングの有無にかかわらず）ことによる初期化準備をリンカに行わせるために使用します。リンカは、各セクションに対して、そのセクションの内容を保持する初期化セクションを論理的に作成し、元のセクションは内容を持たないセクションに変換することによりこれを行います。初期化セクションの名前は、サフィックス _init を追加した元のセクションの名前です。例えば、.data セクションの初期化セクションは .data_init と呼ばれます。これはオーバレイのほかに、その他の場合にも便利です。

initialize manually を使用して、テーブルの要素作成を抑止し、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ているセクションを除いて、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。

自動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、リンカが生成する自動ブロックを使用しています。

注 : 自動ブロックは通常のセクションセレクションプロセスを上書きし、セクション名に合致する全ての要素を強制的に 1 つのブロックに配置します。

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 を RAM に、ブロック MYBLOCK_init 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。

注： 初期化を手動で使用しているときは、各 copy init バッチを明示的に扱う必要があります。リンカは、ソースブロックまたは配置ディレクティブと移動先ブロックまたは配置ディレクティブのそれぞれの組み合わせに、異なるバッチを作成します。作成されるバッチを確認知るには、初期化ログ (--log initialization) を使用します。

リンカによって、ブロックが自動的に作成されることがあります。これは複数のバッチに影響します。fixed order のあるブロックを使用しているときや、拡張セクションセレクタで first、last、または midway 修飾子を使用しているときに発生することがあります。

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）

アプリケーションが、コードの一部をフラッシュ ROM から RAM にコピーすることがあります。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ことも、122 ページの「*手動で初期化する*」の説明に従って後でユーザコードで行うこともできます。

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。以下に例を示します。

```
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。122 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。79 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。

関数 __low_level_init が存在する場合、この関数は初期化の前に呼び出されるため、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、割り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,       /* 割り込みテーブルを
                                               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、231 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、231 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、145 ページの「*ビルド済ランタイムライブラリ*」を参照してください。

ELF/DWARF 以外の出力フォーマットを生成

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、591 ページの「*IAR ELF ツール — ielftool*」を参照してください。

ベニア

ベニアは、リンカにより挿入されたコードの小さいシーケンスで、コール関数がその目的先に到達しない、または正しいモードに変更できないときに、そのギャップをつなげます。

ベニアのコードは、任意の呼び出し元および呼び出し先関数間に挿入できます。そのため、一部のレジスタは、アセンブラ言語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。**32 ビットモードの場合**、`R12` はスクラッチレジスタとして扱う必要があります。**64 ビットモードの場合**、`X16` と `X17` は両方ともスクラッチレジスタとして扱う必要があります。

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（367 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（370 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。

```
エラー [Lp002]: relocation failed: out of range or illegal value
    Kind     :   R_XXX_YYY[0x1]
    Location :   0x40000448
                 "myfunc" + 0x2c
                 Module:  somecode.o です
                 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。

表 4: 再配置エラーの説明

メッセージエントリ	説明
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。この例では、0x9000000c および "read" なのでオフセットなし。 • 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（続き）

考えられる解決方法

このケースでは、myfunc から __read にある命令までの長さが、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び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。

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。

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	Color	taste
`file1`	`blue`	**未定義**
`file2`	`red`	**未定義**
`file3`	`red`	*
`file4`	`red`	`spicy`
`file5`	`red`	`lean`

表 5: ランタイムモデル属性の例

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。以下に例を示します。

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

注 : 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、448 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。

リンカの最適化

このセクションの内容は以下のとおりです。

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、どの仮想関数が呼び出されるか、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び出しを実行せず、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。

IDE で、**[プロジェクト] > [オプション] > [リンカ] > [最適化] > [C++ 仮想関数除去]** を選択して、この最適化を有効にします。

注 : 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、386 ページの「*--vfe*」、376 ページの「*--no_vfe*」を参照してください。

小さいルーチンのインライン化

小さいルーチンのインライン化は、リンカの最適化で、小さい関数の呼び出しを、関数の本体と置き換えます。これには、関数を呼び出す命令のスペースに収まる本体が必要です。

IDE で、**[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]** を選択して、この最適化を有効にします。

リンカオプション `--inline` を使用します。

重複セクションのマージ

リンカは、同一の読み取り専用セクションを検出し、そのような各セクションのコピーを 1 つだけ保持し、すべての重複セクションへのリファレンスを保持セクションにリダイレクトします。

IDE で、**[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]** を選択して、この最適化を有効にします。

リンカオプション `--merge_duplicate_sections` を使用します。

注 : これは、異なる関数または定数が同じアドレスをもつ原因になります。よって、テーブルのキーとしてアドレスを使用しているなど、アプリケーションが異なるアドレスに依存している場合は、この最適化を有効にしないでください。

DLIB ランタイム環境

- ランタイム環境の概要
- ランタイム環境の設定
- ランタイム環境についての追加情報
- マルチスレッド環境の管理

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。

このセクションの内容は以下のとおりです。

ランタイム環境の機能

*DLIB ランタイム環境*は標準の C/C++ をサポートし、以下が含まれます。

- *C/C++ 標準ライブラリ*、両方のインタフェース（システムヘッダファイルに同梱）、およびその実装
- 起動 / 終了コード
- 入出力 (I/O) を管理するための低レベル I/O インタフェース
- 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
- ハードウェア機能のサポート
  - *組み込み*関数（割り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り込みの定義
  - ベクタ浮動小数点 (VFP) コプロセッサ

ランタイム環境の関数は*ランタイムライブラリ*で提供されます。

ランタイムライブラリは、ビルド済みと（製品パッケージに応じて）ソースファイルとして提供されます。ビルド済ライブラリは、さまざまなニーズに

対応するよう異なる*構成*のものを用意しています（144 ページの「*ランタイムライブラリ構成*」を参照）。ライブラリは、製品のサブディレクトリ（`arm¥lib` および `arm¥src¥lib`）にあります。

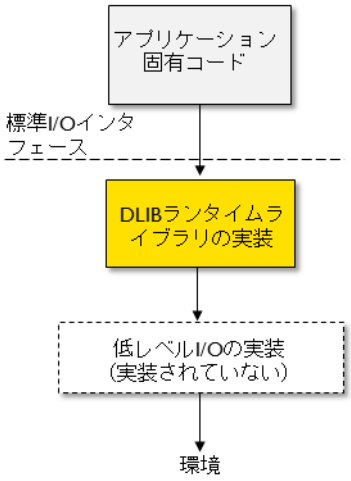
ライブラリの詳細は、「*C/C++ 標準ライブラリ関数*」を参照してください。

入出力 (I/O) の概要

すべてのアプリケーションは、その環境と交信する必要があります。たとえば、アプリケーションが LCD に情報を表示し、センサーから値を読み込んだり、オペレーティングシステムから最新の値を取得するといったことが考えられます。アプリケーションは通常、C/C++ 標準ライブラリまたは何らかのサードパーティ製ライブラリを介して I/O を実行します。

I/O を処理する C/C++ 標準ライブラリには数多くの関数があり、標準文字ストリーム、ファイルシステムアクセス、日時、その他のシステムアクション、終了およびアサートなどがあります。この関数のセットを*標準 I/O インタフェース*といいます。

デスクトップコンピュータやサーバでは、オペレーティングシステムがランタイム環境の標準 I/O インタフェースを通じてアプリケーションに I/O 機能を提供することになっています。ただし、組み込みシステムではランタイムライブラリはそういった機能が存在したり、オペレーティングシステムがあることすら想定でき ません。このため、標準 I/O インタフェースの低レベルの部分はデフォルトでは完全には実装されません。



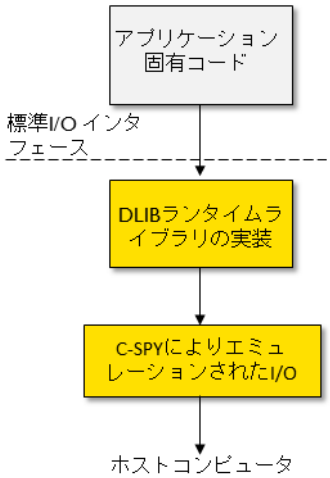
標準 I/O インタフェースを機能させるには、以下のようにします。

- ホストコンピュータ上でC-SPYデバッガによりI/O処理をエミュレーションさせる（135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」を参照）
- 適切なインタフェースの実装を提供することで、標準 I/O インタフェースのターゲットを変更（136 ページの「*再ターゲットの概要*」を参照）

これらの 2 つのアプローチを組み合わせることも可能です。たとえば、デバッグの出力とアサートを C-SPY デバッガによりエミュレーションする一方、自身のファイルシステムを実装するなどです。デバッグの出力とアサートはデバッグ中には便利ですが、アプリケーションをスタンドアロン（C-SPY に接続されていない状態）で実行する際には不要です。

C-SPY によりエミュレーションされた I/O の概要

C-SPY によりエミュレーションされた I/O は、ランタイム環境を C-SPY デバッガと連携させて、ホストコンピュータ上で I/O 処理をエミュレーションするメカニズムです。



たとえば、C-SPY によりエミュレーションされた I/O を有効にした場合、次のようになります。

- 標準文字ストリームが C-SPY［**ターミナル I/O**］ウィンドウに送られる
- ファイルシステムの操作がホストコンピュータ上で実行される
- 時間および日にちの関数がホストコンピュータの時間および日にちを返す

- アプリケーションが終了した、またはアサートが失敗したとき、C-SPY デバッガは通知します。

この動作はアプリケーション開発の初期に非常に役立ちます。たとえば、フラッシュファイルシステムの I/O ドライバを実装する前にファイル I/O を使用するアプリケーションや、I/O を利用可能にする実際のハードウェアデバイスを持たない `stdin` および `stdout` を使用するアプリケーション内の構造体をデバッグしなければならないときなどです。

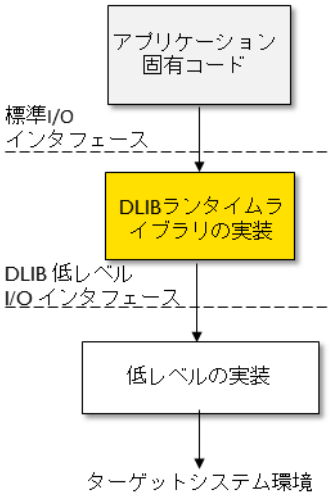
137 ページの「*ランタイム環境の設定*」および 153 ページの「*セミホスティングのメカニズム*」を参照してください。

再ターゲットの概要

*再ターゲット*は、アプリケーションがターゲットシステム上で I/O 処理を実行できるようにランタイム環境を適応するプロセスです。

標準の I/O インタフェースは大きくて複雑です。再ターゲットを簡単にするため、DLIB ランタイム環境は単純な関数の小さいセットを通してすべての I/O 処理を実行するよう設計されています。これらの関数のセットを *DLIB 低レベル I/O インタフェース*といいます。デフォルトでは、低レベルインタフェースの関数には使用可能な実装はありません。実装されていないものもあり、エラーコードを返す以外は何も実行しないスタブ実装のものもあります。

標準の I/O インタフェースを再ターゲットするには、DLIB 低レベル I/O インタフェース内で関数に実装を提供するだけです。



たとえば、アプリケーションが標準 I/O インタフェース内で関数 `printf` と `fputc` を呼び出す場合、これらの関数の実装はどちらも個々の文字を出力するために低レベル関数 `__write` を呼び出します。これらを機能させるには、`__write` 関数の実装を提供するだけでよく、自分で実装するか、サードパーティ製の実装を使用してこれを行います。

自身の実装による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、141 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。インタフェースの一部である関数の詳細は、159 ページの「*DLIB 低レベル I/O インタフェース*」も参照してください。

ランタイム環境の設定

このセクションでは以下のタスクについて説明します。

- 137 ページの「*ランタイム環境の設定*」
  開発初期段階で使用する基本的なプロジェクト設定のランタイム環境。
- 139 ページの「*再ターゲット — ターゲットシステムへの適合*」
- 141 ページの「*ライブラリモジュールのオーバライド*」
- 142 ページの「*独自のランタイムライブラリのカスタマイズおよびビルド*」

関連項目：

- 171 ページの「*マルチスレッド環境の管理*」：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、ランタイム環境を適応させる方法について説明しています。

ランタイム環境の設定

何らかの基本的なプロジェクト設定に基づいてランタイム環境を設定することができます。また、標準ストリームやファイル I/O、その他さまざまなシステムの相互処理を C-SPY デバッガにより管理する方が、たいてい便利です。基本的なランタイム環境は、ターゲットハードウェアを用意する前にシミュレーションに使用できます。

ランタイム環境を設定するには、以下の手順に従います。

1. プロジェクトをビルドする前に、**[プロジェクト]** > **[オプション]** > **[一般オプション]** を選択して **[オプション]** ダイアログボックスを開きます。
2. **[ライブラリ構成]** ページで以下の設定を確認します。
   - **[ライブラリ]**：使用する *ライブラリ構成*を選択します。通常、**[なし]**、**[ノーマル]**、**[フル]**、または **[カスタム]** を選択します。C++17 をサポー

トするライブラリの場合は、フルライブラリ設定を使用する **Libc++** を選択します。

さまざまなライブラリ構成の詳細については、144 ページの「*ランタイムライブラリ構成*」を参照してください。

3 ［**ライブラリオプション**］ページでは、［**Printf フォーマッタ**］および［**Scanf フォーマッタ**］の両方に、［**マルチバイトをサポートする自動**］または［**マルチバイトをサポートしない自動**］を選択します。つまり、リンカはコンパイラからの情報に基づいて適切なフォーマッタを自動的に選択します。使用可能なフォーマッタと手動でそれを選択する方法については、149 ページの「*printf のフォーマッタ*」と 151 ページの「*scanf のフォーマッタ*」をそれぞれ参照してください。

4 C-SPY でエミュレーションされた I/O 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**］>［**ライブラリ構成**］を選択し、［**セミホスティング**］(`--semihosted`) または［**IAR ブレークポイント**］(`--semihosting=iar_breakpoint`) を選びます。

注 : 一部の Cortex-M デバイスでは、SWO 経由で `stdout/stderr` を出力することもできます。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、［**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。

135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」および 153 ページの「*セミホスティングのメカニズム*」を参照してください。

5 一部のシステムでは、ホストコンピュータとターゲットシステムが 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` 関数がランタイム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。

注 : この関数は、約 80 バイトの RAM メモリを使用します。

IDE でこの機能を使用するためには、［**プロジェクト**］>［**オプション**］>［**一般オプション**］>［**ライブラリオプション 1**］から、［**バッファターミナル出力**］を選択します。

コマンドラインでこの関数を有効にするには、以下をリンカコマンドラインに追加します。

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

6 一部の数学関数には異なるバージョンが用意されています。デフォルト、デフォルトより小さいもの、デフォルトのバージョンより大きくてより正確なものがあります。どのバージョンを使用すべきか検討してください。

詳細については、154 ページの「*数学関数*」を参照してください。

7 プロジェクトをビルドする際、選択したプロジェクト設定に基づいて、適切なビルド済ライブラリおよびライブラリ構成ファイルが自動的に使用されます。

どのプロジェクト設定がライブラリファイルの選択に影響するかは、144 ページの「*ランタイムライブラリ構成*」を参照してください。

アプリケーションのソースコードを開発するときに使用できるランタイム環境の設定が終わりました。

再ターゲット — ターゲットシステムへの適合

ターゲットシステムでアプリケーションを実行する前に、ランタイム環境の一部（通常はシステムの初期化と DLIB 低レベル I/O インタフェース関数）を適応させる必要があります。

ターゲットシステムに合わせてランタイム環境を適応させるには、以下の手順に従います。

1 システム初期化を適応させます。

アプリケーションが割り込み処理や I/O 処理、ウォッチドッグタイマなどを初期化しなければならないときなどに、システムの初期化を適応しなければならないことがよくあります。これを行うには、`__low_level_init` ルーチンを実装します。このルーチンはデータセクションの初期化の前に実行されます。155 ページの「*システムの起動と終了*」および 158 ページの「*システム初期化*」を参照してください。

注 : これについてデバイス固有の例が製品のインストールに付属のサンプルプロジェクトにあります。インフォメーションセンタを参照してください。

2 ターゲットシステムに合わせてランタイムライブラリを適応させます。こうした関数を実装するには、DLIB 低レベルインタフェースを十分に理解してください（136 ページの「*再ターゲットの概要*」を参照）。

通常はアプリケーションが以下を使用する場合に、自身の関数を実装する必要があります。

- 標準 I/O ストリーム

  これらのストリームのいずれかが、たとえば `printf` および `scanf` などの関数に使用される場合、独自バージョンの低レベル関数 `__read` と `__write` を実装する必要があります。

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、ファイルハンドル（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 のファイルハンドルを持ちます。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。ストリームは stdio.h で定義されます。

- ファイル I/O

  このライブラリには、fopen、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び出し、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。これらの低レベル関数の独自のバージョンを実装してください。

- signal と raise

  これらの関数のデフォルトの実装から必要な機能が得られなければ、独自のバージョンを実装できます。

- time と date

  time 関数および date 関数を機能させるには、関数 clock、__time32、__time64、__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（530 ページの「*time.h*」を参照）。

- アサート（161 ページの「*__aeabi_assert*」を参照）。

- 環境の操作

  system や getenv のデフォルトの実装から必要な機能が得られなければ、独自のバージョンを実装をできます。

関数の情報については、159 ページの「*DLIB 低レベル I/O インタフェース*」を参照してください。

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。

3 低レベル I/O インタフェースの関数を実装したら、これらの関数の独自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、141 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。

注 : DLIB 低レベル I/O インタフェース関数を実装して、それを C-SPY によりエミュレーションされた I/O のサポートを用いてビルドしたプロジェクトに追加した場合、低レベル関数は使用されますが、C-SPY によりエミュレーションされた I/O に付属の関数は使用されません。たとえば、独自のバージョンの __write を実装すると、C-SPY［**ターミナル I/O**］ウィンドウへの出力はサポートされません。135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」を参照してください。

4 ターゲットシステムでアプリケーションを実行する前に、Release ビルド構成を用いてプロジェクトをリビルドする必要があります。つまり、リンカは C-SPY によりエミュレーションされた I/O メカニズムと、それにより提供される低レベル I/O 関数をインクルードしません。直接または間接的に、標準 I/O インタフェースの低レベル関数のいずれかをアプリケーションが呼び出し、プロジェクトにこれらが含まれない場合、リンカは低レベル関数を持たないものすべてについてエラーを表示します。

注 : デフォルトでは、NDEBUG シンボルはリリースビルド構成で定義される点に注意してください。つまり、アサートは確認されません。詳細については、161 ページの「*__aeabi_assert*」を参照してください。

ライブラリモジュールのオーバライド

ライブラリ関数をオーバライドして独自の実装を置換するには、以下の手順に従います。

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、そのコピーを自分のプロジェクトディレクトリに置きます。

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。

2 ファイルを修正します。

注 : 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、重複定義に関するエラーメッセージが表示されます。

3 他のソースファイルと同じように、修正したファイルを自分のプロジェクトに追加します。

注 : DLIB 低レベル I/O インタフェース関数を実装して、それを C-SPY によりエミュレーションされた I/O のサポートを用いてビルドしたプロジェクトに追加した場合、低レベル関数は使用されますが、C-SPY によりエミュレーションされた I/O に付属の関数は使用されません。たとえば、独自のバージョンの __write を実装すると、C-SPY **［ターミナル I/O］**ウィンドウへの出力はサポートされません。135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」を参照してください。

これで、ライブラリモジュールを独自のバージョンでオーバライドするプロセスが完了しました。

独自のランタイムライブラリのカスタマイズおよびビルド

ビルド済ライブラリ構成が要件を満たさない場合、独自のライブラリ構成をカスタマイズできますが、これにはライブラリの関連した部分の*リビルド*が必要です。

注 : 独自のランタイムライブラリをカスタマイズ、およびビルドするには、ライブラリのソースコードにアクセスする必要があります。これは、IAR Embedded Workbench ライセンスのすべてのタイプでは利用できません。

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ランタイム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。これには、DLIB ランタイム環境の一部の追加 / 削除が含まれます。

このような場合は、以下を行う必要があります。

- ライブラリソースコード (`src¥lib`) をインストールしたことを確認します。まだインストールしていない場合は、IAR License Manager を使用してインストールできます（*ライセンスガイド*を参照）
- ライブラリプロジェクトをセットアップする
- 必要なライブラリのカスタマイズを行う
- カスタマイズしたランタイムライブラリをビルドする
- カスタマイズしたランタイム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。

  カスタマイズしたライブラリは、C および C++ ライブラリ関数のライブラリで実行された DLIB ランタイム環境の一部だけを置き換えることに注意してください。次のライブラリの再構築はサポートされていません。
  - 数学関数
  - ランタイムサポート関数
  - スレッドサポート関数
  - タイムゾーンおよび昼時間関数
  - デバッグサポート関数

ライブラリプロジェクトをセットアップには、以下の手順に従います。

1. IDE で、**[プロジェクト] > [新しいプロジェクトの作成]** を選択し、カスタマイズしたランタイム環境設定に使用するライブラリプロジェクトテンプレートを使用します。フルライブラリ構成のライブラリテンプレートがあります。144 ページの「*ランタイムライブラリ構成*」参照。

注 : テンプレートから新しいライブラリプロジェクトを作成する場合、新しいプロジェクトに含まれるファイルの大半は元のインストールファイルです。これらのファイルを修正する場合、まずコピーを作成してプロジェクトの元のファイルをコピーと置換します。

ライブラリ機能をカスタマイズするには、以下の手順に従います :

1 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おり、これは `arm¥inc¥c` にあります。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。このファイルは修正しないでください。

さらに、`arm¥inc¥c` ディレクトリにある `DLib_Config_configuration.h` ファイルのコピーを作成して、独自の*ライブラリ設定ファイル*を作成できます。また、アプリケーションの要件に応じて設定シンボルの値を設定してカスタマイズできます。

カスタマイズする構成シンボルの詳細については、以下を参照してください。

- 169 ページの「*ファイル I/O の構成シンボル*」
- 169 ページの「*ロケール*」
- 171 ページの「*マルチスレッド環境の管理*」

2 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。

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。

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。

アプリケーションプロジェクト内のカスタマイズされたランタイムライブラリを使用するには、以下の手順に従います。

1 IDE で、**[プロジェクト]** > **[オプション]** > **[一般オプション]** を選択し、**[ライブラリ構成]** タブをクリックします。

2 **[ライブラリ]** ドロップダウンメニューで、**[カスタム]** 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。

ランタイム環境についての追加情報

ここでは、ランタイム環境の追加情報を提供します。

バウンドチェック機能

標準 C の Annex K（*境界チェックインターフェース*）で指定されたバウンドチェック関数を有効にするには、システムヘッダに入れる前に、プリプロセッサシンボル `__STDC_WANT_LIB_EXT1__` を `1` に定義します。529 ページの「*C 境界チェックインターフェース*」を参照してください。

ランタイムライブラリ構成

ランタイムライブラリには異なる *ライブラリ構成*が付属し、それぞれの構成は異なるアプリケーション要件に適合するようになっています。

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。ランタイム環境で必要な機能が少ないほど、環境も小さくなります。

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）	C ロケール、ロケール インターフェースなし、ファイル記述子のサポートなし、`printf` および `scanf` でのマルチバイト文字なし。
フル DLIB	フルの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。

表 6: ライブラリ構成

注：これらの定義済構成に加えて、独自の構成を作成できます（142 ページの「*独自のランタイムライブラリのカスタマイズおよびビルド*」を参照）。

ライブラリ構成を明示的に指定しなければ、デフォルトの構成が使用されます。ビルド済ランタイムライブラリを使用する場合、ランタイムライブラリに合った構成ファイルが自動的に使用されます。137 ページの「*ランタイム環境の設定*」を参照してください。

注： Libc++ ライブラリを使用する場合は、自動的にフル設定になります。これが、Libc++ に存在する唯一の設定です。

デフォルトのライブラリ構成をオーバライドするには、以下のいずれかの方法を使用します。

1 希望するビルド済構成を使用して、ランタイム構成を明示的に指定します。

［**プロジェクト**］>［**オプション**］>［**一般オプション**］>［**ライブラリ構成**］>［**ライブラリ**］を選択して、デフォルト設定を変更します。

`--dlib_config` コンパイラオプションを使用します（305 ページの「*--dlib_config*」を参照）。

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（144 ページの「*ランタイムライブラリ構成*」を参照）。

2 独自のカスタマイズしたライブラリをビルドした場合、独自の構成を使用するには［**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**］を選択して、［**カスタム**］を選びます。詳細については、142 ページの「*独自のランタイムライブラリのカスタマイズおよびビルド*」を参照してください。

ビルド済ランタイムライブラリ

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

- プロセッサ選択
- データモデル

- ライブラリ構成（ノーマルまたはフル）

リンカは、正しいライブラリ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` コンパイラオブジェクトを使用します。

注： Libc++ ライブラリを使用する場合は、自動的にフルライブラリ設定になります。これが、Libc++ に存在する唯一の設定です。`--dlib_config` コンパイラオプションは、別の設定を指定するために使用できません。

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。

arch	CPU アーキテクチャを指定します。 `4t` = Armv4T `5E` = Armv5E `6M` または `6Mx` = Armv6M（6Mx は `--no_literal_pool` で構築） `7M` または `7Mx` = Armv7M（7Mx は `--no_literal_pool` で構築） `7Sx` = Armv7-A および Armv7-R、`--no_literal_pool` で構築 `4as` = 汎用 Armv4、境界チェックで構築 `7as` = 汎用 Armv7、境界チェックで構築 `8A` = Armv8-A
mode	デフォルトのプロセッサ / 実行モードの指定 : `a` = Arm モード `t` = Thumb モード `x` = 6464 ビットモード
endian	バイトオーダーを指定 : `l` = little-endian `b` = big-endian
lib-config	ライブラリ構成を指定 : `n` = ノーマル `f` = フル
rwpi	ライブラリが RWPI をサポートするかどうかを指定 : `s` = RWPI をサポート 存在しない = RWPI をサポートしない

fp	浮動小数点の実行方法を指定 : v = VFP s = 単精度の VFP のみ 存在しない = ソフトウェア実行
abi	**64 ビットモード**のデータモデルの指定 : 44 = ILP32（4 バイト long、4 バイトポインタ） 88 = ILP32（8 バイト long、8 バイトポインタ） 表示なし = ライブラリは **32 ビットモードでの使用**
debug-interface	セミホスティングのメカニズムを指定 : s = SVC b = BKPT i = IAR-breakpoint

ライブラリオブジェクトファイルはディレクトリ arm¥lib¥ にあり、ライブラリ設定ファイルは arm¥inc¥ 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 C により定義された関数で、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{*arch*}_{*mode*}{*endian*}{*lib-config*}[*rwpi*][*abi*].a

具体的には以下のようになります。

dl{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}{n|f}[s][44|88].a

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++ のサポートを使用してコンパイルされます。

Dinkumware C++14 ライブラリ (dlpp) および Libc++ C++17 ライブラリ (dllibcpp) には別のライブラリがあります。

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{*arch*}_{*mode*}{*endian*}{*fp*}_{*lib-config*}c[*rwpi*][*abi*].a
dllibcpp{*arch*}_{*mode*}{*endian*}{*fp*}_fc[*rwpi*][*abi*].a

具体的には以下のようになります。

```
dlpp{4t|5E|6M|6Mx|7M|7Mx|7Sx|4as|7as|8A}_{a|t|x}{l|b}{v|s|}_{n|f}
c[s][44|88].a
dllibcpp{4t|5E|6M|6Mx|7M|7Mx|7Sx|4as|7as|8A}_{a|t|x}{l|b}{v|s|}_f
c[s][44|88].a
```

C++ ランタイムライブラリ関数のライブラリファイル

C++ に必要なランタイム関数があり、Libc++ と DLIB C++ ライブラリの両方で使用されます。

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。

dlpprt{*arch*}_{*mode*}{*endian*}_{*lib-config*}c[*rwpi*][ab*abi*i].a

具体的には以下のようになります。

```
dlpprt{4t|5E|6M|6Mx|7M|7Mx|7Sx|4as|7as|8A}_{a|t|x}{l|b}_{n|f}c[s]
[44|88].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標準の C 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{*arch*}_{*mode*}{*endian*}[*fp*][*abi*].a

具体的には以下のようになります。

```
m{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}[v|s][44|88].a
```

スレッドサポート関数のライブラリファイル

これらはスレッドサポートの関数です。

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。

th{*arch*}_{*mode*}{*endian*}{*lib-config*}[*abi*].a

具体的には以下のようになります。

```
th{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}{n|f}[44|88].a
```

タイムゾーンと夏時間のサポート関数のライブラリファイル

タイムゾーンと夏時間機能をサポートする関数です。

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。

tz{*arch*}_{*mode*}{*endian*}[*rwpi*][*abi*].a

具体的には以下のようになります。

```
tz{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}[s][44|88].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{arch}_{mode}{endian}[abi].a
```

具体的には以下のようになります。

```
rt{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}[44|88].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}_{endian}.a
```

または

```
sh{arch}_{endian}[abi].a
```

具体的には以下のようになります。

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

または

```
sh{6Mx|7Mx|7Sx|8A}_{l|b}[44|88].a
```

printf のフォーマッタ

`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、多くの組み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。メモリの消費を減らすため、3 つのより小さい代替バージョンも用意されています。`wprintf` 派生形は影響を受けない点に注意してください。

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 / 小 NoMb†	大 / 大 NoMb†	フル / フル NoMb†
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
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、スペース	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
wchar_t のサポート	なし	なし	なし	あり

表 7: printf のフォーマッタ

† NoMb は、マルチバイトなしという意味です。

フォーマット文字列が文字列リテラルの場合、コンパイラは printf への直接の呼び出しでどのフォーマット機能が必要かを自動的に検出することができます。この情報はリンカに渡され、すべてのモジュールからの情報と組み合わせてアプリケーションに適切なフォーマッタが選ばれます。ただし、フォーマット文字列が変数であったり、呼び出しが関数ポインタを介した間接的なものの場合、コンパイラは解析を実行できず、リンカはフルのフォーマッタを選択することになります。この場合、自動的に選択された printf フォーマッタをオーバライドすることをお勧めします。

IDE で自動的に選択された printf フォーマッタをオーバライドするには、以下の手順に従います。

1. [プロジェクト] > [オプション] > [一般オプション] を選択して [オプション] ダイアログボックスを開きます。
2. [ライブラリオプション] ページで、適切なフォーマッタを選択します。

コマンドラインから自動的に選択された printf フォーマッタをオーバライドするには、以下の手順に従います。

1. 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
```

コンパイラがマルチバイトサポートを認識しない場合は、それを有効にできます。

［プロジェクト］>［オプション］>［一般オプション］>［ライブラリオプション 1］>［マルチバイトサポートの有効化］を選択します。

リンカオプション `--printf_multibytes` を使用します。

scanf のフォーマッタ

`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、多くの組み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。メモリの消費を減らすため、2 つのより小さい代替バージョンも用意されています。`wscanf` バージョンは影響を受けない点に注意してください。

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 / 小 NoMb†	大 / 大 NoMb†	フル / フル NoMb†
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
スキャン集合 [、]	なし	あり	あり
代入抑止 *	なし	あり	あり
long long のサポート	なし	なし	あり
wchar_t のサポート	なし	なし	あり

表 8: scanf のフォーマッタ

† NoMb は、マルチバイトなしという意味です。

フォーマット文字列が文字列リテラルの場合、コンパイラは scanf への直接の呼び出しでどのフォーマット機能が必要かを自動的に検出することができます。この情報はリンカに渡され、すべてのモジュールからの情報と組み合わせてアプリケーションに適切なフォーマッタが選ばれます。ただし、フォーマット文字列が変数であったり、呼び出しが関数ポインタを介した間接的なものの場合、コンパイラは解析を実行できず、リンカはフルのフォーマッタを選択することになります。この場合、自動的に選択された scanf フォーマッタをオーバライドすることをお勧めします。

IDE で手動により scanf フォーマッタを指定するには、以下の手順に従います。

1. [プロジェクト] > [オプション] > [一般オプション] を選択して [オプション] ダイアログボックスを開きます。
2. [ライブラリオプション] ページで、適切なフォーマッタを選択します。

コマンドラインからの scanf フォーマッタを手動により指定するには、以下の手順に従います。

1. 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
```

コンパイラがマルチバイトサポートを認識しない場合は、それを有効にできます。

［プロジェクト］>［オプション］>［一般オプション］>［ライブラリオプション 1］>［マルチバイトサポートの有効化］を選択します。

リンカオプション `--scanf_multibytes` を使用します。

C-SPY によりエミュレーションされた I/O のメカニズム

1 デバッガは `__DebugBreak` 関数の存在を検出します。この関数は「C-SPY によりエミュレーションされた I/O」のリンカオプションによりアプリケーションとリンクすると、その一部となります。

2 この場合、デバッガは `__DebugBreak` 関数にブレークポイントを自動的に設定します。

3 アプリケーションが `open` などDLIB低レベルI/Oインタフェース内の関数を呼び出す場合、`__DebugBreak` 関数が呼び出され、これによってアプリケーションブレークポイントで停止して必要なサービスを実行します。

4 その後で実行が再開されます。

135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」を参照してください。

セミホスティングのメカニズム

C-SPY によりエミュレーションされた I/O は、Arm Limited が提供するセミホスティングのインタフェースと互換性があります。アプリケーションがセミホスティングを呼び出す際、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、デバッガが呼び出しを処理し、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。

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び出しを実行します。
- 他の Arm コアの場合、32 ビットモードでの**セミホスティングの呼出しには、** `SVC` 命令が使用されます。
- **64 ビットモードの場合**、`HLT` 命令がセミホスティングの呼出しに使用されます。
- *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 の呼び出

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 呼び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、こうした呼び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び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。

注 : 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び出しは機能しません。

数学関数

一部の C/C++ 標準ライブラリ数学関数は、異なるバージョンが用意されています。

- デフォルトのバージョン
- より小さいバージョン（精度は低下します）
- より正確なバージョン（サイズはより大きくなります）

より小さいバージョン

ライブラリには、小さいバージョンの `cos`、`exp`、`log`、`log2`、`log10`、`pow`、`sin`、および `tan` 関数が存在します。これらはデフォルトのバージョンより約 20% 小さく、約 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、ほとんどの場合において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。

より正確なバージョン

関数 `cos`、`pow`、`sin`、および `tan` が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。

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。

開始と終了を処理するコードは、arm¥src¥lib¥arm、arm¥src¥lib¥thumb（Cortex-M の場合）あるいは arm¥src¥lib¥a64 ディレクトリ内のソースファイル cstartup.s、cmain.s、cexit.s、および arm¥src¥lib¥runtime ディレクトリ内の low_level_init.c にあります。

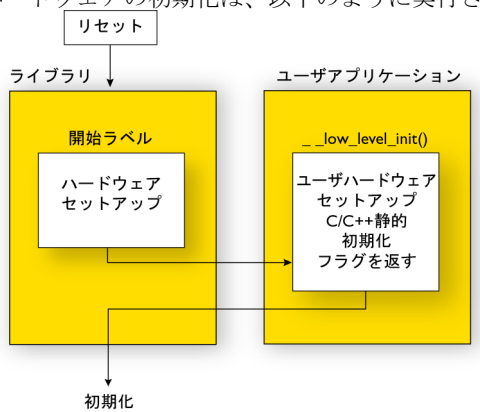
注 : これらのファイルをいくつかインストールするには、IAR Library Source パッケージを展開する必要があります。

システム起動コードのカスタマイズ方法については、158 ページの「*システム初期化*」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。

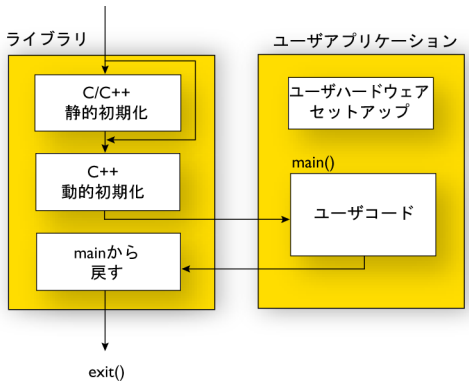
ハードウェアの初期化は、以下のように実行されます。



- 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。**64 ビットモード**の場合、リンカシンボル __Reset_address でセットアップされます。

- **64 ビットモードの場合**、プログラムはハードリセットで入力された例外レベルから例外レベルまでを調べます。
- 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、この関数が呼び出され、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。

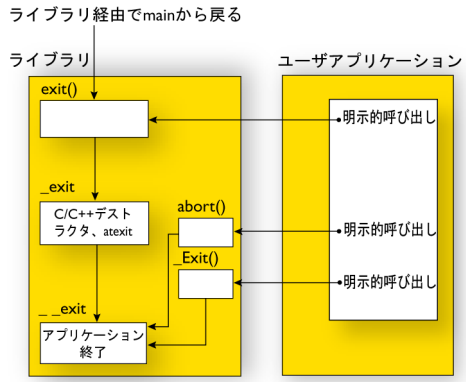


- 静的変数とグローバル変数が初期化されます。つまり、ゼロ初期化変数がクリアされ、他の初期化変数の値が ROM から RAM メモリにコピーされます。このステップは、`__low_level_init` がゼロを返す場合には、省略されます。詳細については、103 ページの「*システム起動時の初期化*」を参照してください。
- 静的 C++ オブジェクトが生成されます。
- `main` 関数が呼び出され、アプリケーションが起動します。

初期化フェーズについて詳しくは、65 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、組み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` 関数を呼び出す

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` 関数を呼び出します。`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、以下を実行する小さなアセンブラ関数 `_exit` を呼び出します。

- 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び出します。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 C 関数 `atexit` で登録された関数が含まれます。121 ページの「*ATEXIT 制限の設定*」を参照してください。
- 開かれているすべてのファイルを閉じます。
- `__exit` を呼び出します。
- `__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`、`_Exit` または `quick_exit` 関数を呼び出すことによっても終了できます。`abort` 関数は、単に `__exit` を呼び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。`quick_exit` 関数は `_Exit` 関数と同等ですが、`__exit` を呼び出す前に、`at_quick_exit` を通過してそれぞれの関数を呼び出します。

終了時にアプリケーションでシステムのリセットなど追加処理を希望する場合（atexit が不十分な場合）、__exit(int) 関数の独自の実装を記述することができます。

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。141 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。

システム終了の C-SPY デバッグサポート

リンクの際に［C-SPY によりエミュレーションされた I/O］を有効にした場合、通常の __exit 関数は特別なものに置換されます。これにより、C-SPY はこの関数が呼び出されたことを認識し、適切な処理を実行して、プログラム終了のエミュレーションを行います。詳細については、135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」を参照してください。

システム初期化

多くの場合、システム初期化の適応が必要になります。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な場合や、システム起動コード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、独自のバージョンの __low_level_init ルーチンを実装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 ファイルから呼び出されます。cmain.s ファイルは直接修正しないでください。

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 に配置されます。cstartup.s は、arm¥src¥lib¥arm、arm¥src¥lib¥thumb (Cortex-M)、または arm¥src¥lib¥a64 ディレクトリに配置され、low_level_init.c は arm¥src¥lib¥runtime ディレクトリに配置されます。

通常は、cmain.s または cexit.s のいずれかのファイルをカスタマイズする必要はありません。

注 : 独自のバージョンの __low_level_init または cstartup.s ファイルを実装する場合も、ライブラリをリビルドする必要はありません。

__low_level_init のカスタマイズ

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。

注 : この時点では変数は初期化されていないため、初期化が必要な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 ファイルの修正

前述のように、独自のバージョンの __low_level_init の実装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（141 ページの「*ライブラリモジュールのオーバライド*」を参照）。

注 : 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、360 ページの「*--entry*」を参照してください。

Cortex-M の場合、割り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。

DLIB 低レベル I/O インタフェース

ランタイムライブラリは、ターゲットシステムとの通信に低レベル関数のセットを使用します。これを *DLIB 低レベル I/O インタフェース*といいます。低レベル関数の多くは実装がありません。

詳細については、134 ページの「*入出力 (I/O) の概要*」を参照してください。

これらは、DLIB 低レベル I/O インタフェースの関数です。

```
abort

__aeabi_assert

clock

__close

__exit

getenv

__getzone

__lseek

__open

raise

__read
```

remove

rename

signal

system

__time32、__time64

__write

注 : 通常、先頭に __ が付く低レベルの関数は、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標準のライブラリ関数を使用してください。たとえば、stdout に書き込むには、printf や puts を使用してください。これらが代わりに低レベル関数 __write を呼び出します。低レベル関数の実装を忘れて、アプリケーションが標準ライブラリ関数を経由してその関数を呼び出す場合、リリースビルド構成でリンクするとリンカはエラーを表示します。

注 : このインタフェースで関数の独自の派生形を実装すると、[C-SPY によりエミュレーションされた I/O] を有効にしている場合でも派生形が使用されます（135 ページの「*C-SPY によりエミュレーションされた I/O の概要*」を参照）。

abort

ソースファイル	arm¥src¥lib¥runtime¥abort.c
宣言	stdlib.h
説明	実行を中止する標準の C ライブラリ。
C-SPY、デバッガ処理	アプリケーションを終了します。
デフォルトの実装	__exit(EXIT_FAILURE) を呼び出します。
関連項目	136 ページの「*再ターゲットの概要*」。 157 ページの「*システム終了*」。

__aeabi_assert

ソースファイル arm¥src¥lib¥runtime¥assert.c

宣言 assert.h

説明 失敗したアサートを処理する低レベル関数。

C-SPY、デバッガ処理 失敗したアサートについて C-SPY デバッガに通知します。

デフォルトの実装 失敗したアサートは、関数 __aeabi_assert よって報告されます。デフォルトでは、エラーメッセージが出力され、abort が呼び出されます。これが必要な動作でなければ、独自のバージョンの関数を実装できます。

アサートマクロは、ヘッダファイル assert.h に定義されます。アサーションを無効にするには、シンボル NDEBUG を定義します。

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。516 ページの「*NDEBUG*」を参照してください。

関連項目 136 ページの「*再ターゲットの概要*」。

clock

ソースファイル arm¥src¥lib¥time¥clock.c

宣言 time.h

説明 プロセッサ時間にアクセスする標準の C ライブラリ。

clock は秒数でカウントすると推定されます。これがその場合ではなく、CLOCKS_PER_SEC を使用している場合、CLOCKS_PER_SEC は、time.h を使用する前の 1 秒あたりの実際のティック数を設定する必要があります。now() 関数を実装するときに、C++ ヘッダ chrono は、CLOCKS_PER_SEC を使用します。

C-SPY、デバッガ処理 ホストコンピュータ上のクロックを返します。

デフォルトの実装 プロセッサ時間が使用できないことを示すために、-1 を返します。

関連項目 136 ページの「*再ターゲットの概要*」。

__close

ソースファイル	arm¥src¥lib¥file¥close.c
宣言	LowLevelIOInterface.h
説明	ファイルを閉じる低レベル関数。
C-SPY、デバッガ処理	ホストコンピュータ上の対応するファイルを閉じます。
デフォルトの実装	なし。
関連項目	136 ページの「*再ターゲットの概要*」。

__exit

ソースファイル	arm¥src¥lib¥runtime¥xxexit.c
宣言	LowLevelIOInterface.h
説明	実行を停止する低レベル関数。
C-SPY、デバッガ処理	アプリケーションの最後に到達したことを通知します。
デフォルトの実装	永久にループします。
関連項目	136 ページの「*再ターゲットの概要*」。 157 ページの「*システム終了*」。

getenv

ソースファイル	arm¥src¥lib¥runtime¥getenv.c arm¥src¥lib¥runtime¥environ.c
宣言	Stdlib.h および LowLevelIOInterface.h
C-SPY、デバッガ処理	ホスト環境にアクセスします。
デフォルトの実装	ライブラリの getenv 関数は、グローバル変数 __environ が指す引数として指定したキー文字列を検索します。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。

以下に例を示します。

```
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。

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は含まれません。

関連項目 136 ページの「*再ターゲットの概要*」。

__getzone

ソースファイル arm¥src¥lib¥time¥getzone.c

宣言 LowLevelIOInterface.h

説明 現在のタイムゾーンを返す低レベル関数。

注： リンカオプション --timezone_lib を使用してライブラリのタイムゾーン機能を有効にする必要があります。

C-SPY、デバッガ処理 適用されません。

デフォルトの実装 ":" を返します。

関連項目 136 ページの「*再ターゲットの概要*」および 383 ページの「*--timezone_lib*」。

詳細については、ソースファイル getzone.c を参照してください。

__lseek

ソースファイル	arm¥src¥lib¥file¥lseek.c
宣言	LowLevelIOInterface.h
説明	開いたファイルで次のアクセスの位置を変更するための低レベル関数。
C-SPY、デバッガ処理	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。
デフォルトの実装	なし。
関連項目	136 ページの「*再ターゲットの概要*」。

__open

ソースファイル	arm¥src¥lib¥file¥open.c
宣言	LowLevelIOInterface.h
説明	ファイルを開く低レベル関数。
C-SPY、デバッガ処理	ホストコンピュータ上のファイルを開きます。
デフォルトの実装	なし。
関連項目	136 ページの「*再ターゲットの概要*」。

raise

ソースファイル	arm¥src¥lib¥runtime¥raise.c
宣言	signal.h
説明	シグナルを引き起こす標準の C ライブラリ関数。
C-SPY、デバッガ処理	適用されません。
デフォルトの実装	引き起こされたシグナルのシグナルハンドラを呼び出すか、__exit(EXIT_FAILURE) への呼び出しをもって終了します。
関連項目	136 ページの「*再ターゲットの概要*」。

__read

ソースファイル　arm¥src¥lib¥file¥read.c

宣言　LowLevelIOInterface.h

説明　stdin およびファイルから文字を読み込む低レベル関数。

C-SPY、デバッガ処理　stdin を［ターミナル I/O］ウィンドウに送ります。他のすべてのファイルは関連のホストファイルを読み込みます。

デフォルトの実装　なし

例　この例のコードは、メモリがマップされた 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>
#include <LowLevelIOInterface.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 記述子が有効な場合にのみ必要) */
  if (handle != 0)
  {
    return -1;
  }

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

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

  return nChars;
}
```

ストリームに関連するハンドルの詳細については、139 ページの「*再ターゲット — ターゲットシステムへの適合*」を参照してください。

@ 演算子の詳細は、254 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。

関連項目 136 ページの「*再ターゲットの概要*」。

remove

ソースファイル `arm¥src¥lib¥file¥remove.c`

宣言 `stdio.h`

説明 ファイルを削除する標準の C ライブラリ関数。

C-SPY、デバッガ処理 ホストコンピュータ上のファイルを削除します。

デフォルトの実装 `0` を返して、ファイルを削除せずに問題なく完了したことを示します。

関連項目 136 ページの「*再ターゲットの概要*」。

rename

ソースファイル `arm¥src¥lib¥file¥rename.c`

宣言 `stdio.h`

説明 ファイル名を変更する標準の C ライブラリ関数。

C-SPY、デバッガ処理 ホストコンピュータ上のファイル名を変更します。

デフォルトの実装 `-1` を返して失敗したことを示します。

関連項目 136 ページの「*再ターゲットの概要*」。

signal

ソースファイル `arm¥src¥lib¥runtime¥signal.c`

宣言 `signal.h`

説明 シグナルハンドラを変更する標準の C ライブラリ関数。

C-SPY、デバッガ処理 適用されません。

デフォルトの実装 標準の C 仕様。非同期のシグナルが環境でサポートされている場合は、この動作を修正することをお勧めします。

関連項目 136 ページの「*再ターゲットの概要*」。

system

ソースファイル
```
arm¥src¥lib¥runtime¥system.c
```

宣言
```
stdlib.h
```

説明 コマンドを実行する標準の C ライブラリ。

C-SPY、デバッガ処理 C-SPY デバッガに、`system` が呼び出されて `-1` を返すよう指示します。

デフォルトの実装 ライブラリで使用可能な `system` 関数は、null ポインタが渡された場合は `0` を返してコマンドプロセッサがないことを示します。それ以外の場合は、失敗を示す `-1` を返します。これが必要な機能でなければ、独自のバージョンを実装できます。この場合、ライブラリのリビルドは必要ありません。

関連項目 136 ページの「*再ターゲットの概要*」。

__time32、__time64

ソースファイル
```
arm¥src¥lib¥time¥time.c
arm¥src¥lib¥time¥time64.c
```

宣言
```
time.h
```

説明 現在のカレンダー時刻を返す低レベル関数。

C-SPY、デバッガ処理 ホストコンピュータ上の時刻を返します。

デフォルトの実装 カレンダーの時刻が使用できないことを示す場合は `-1` を返します。

関連項目 136 ページの「*再ターゲットの概要*」。

__write

ソースファイル　arm¥src¥lib¥file¥write.c

宣言　LowLevelIOInterface.h

説明　stdout、stderr またはファイルに書き込む低レベル関数。

C-SPY、デバッガ処理　stdout および stderr を［**ターミナル I/O**］ウィンドウに送ります。他のすべてのファイルは関連のホストファイルに書き込みます。

デフォルトの実装　なし。

例　この例のコードは、メモリマップド I/O を使用して LCD ディスプレイに書き込み、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>
#include <LowLevelIOInterface.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) */
  if (handle != 1 && handle != 2)
  {
    return -1;
  }
```

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

  return nChars;
}
```

ストリームに関連するハンドルの詳細については、139 ページの「*再ターゲット — ターゲットシステムへの適合*」を参照してください。

関連項目

136 ページの「*再ターゲットの概要*」。

ファイル I/O の構成シンボル

ファイル I/O はフルのライブラリ構成を持つライブラリでのみサポートされています（144 ページの「*ランタイムライブラリ構成*」を参照）。構成シンボル `__DLIB_FILE_DESCRIPTOR` が定義されている場合は、カスタマイズされたライブラリにあります。このシンボルが定義されていない場合は、`FILE *` 引数は使用できません。

ライブラリのカスタマイズおよびリビルドについては、142 ページの「*独自のランタイムライブラリのカスタマイズおよびビルド*」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。

使用しているライブラリ構成に応じて、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。144 ページの「*ランタイムライブラリ構成*」を参照してください。

DLIB ランタイムライブラリは、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があるフルライブラリ設定を使用し、実行中にロケールの切替えを可能にします。

  C ロケールでアプリケーションを開始します。別のロケールを使用するには、`setlocale` または C++ の関連のメカニズムを使用する必要があります。アプリケーションが使用できるロケールはリンク時点で設定されます。

- C ロケールがアプリケーションに組み込まれたロケールインターフェースがない、標準のライブラリ設定を使用します。

注 : マルチバイトがプリントされる場合、DLIB 低レベル I/O インターフェースで `__write` の実行が行えることを確認してください。

アプリケーションで使用できるロケールを指定します

[プロジェクト] > [オプション] > [一般オプション] > [ライブラリオプション 2] > [ロケールサポート] を選択します。

パラメータとしてロケールのタグのあるリンカオプション `--keep` を使用します。例 :

```
--keep _Locale_cs_CZ_iso8859_2
```

利用可能なロケールは、`arm¥config` ディレクトリの `SupportedLocales.json` ファイルに一覧表示されます。例 :

```
['Czech language locale for Czech Republic', 'iso8859-2',
'cs_CZ.iso8859-2', '_Locale_cs_CZ_iso8859_2'],
```

列には、フルロケール名、ロケールのエンコード、省略されたロケール名、およびのリンカオプション `--keep` にパラメータとして使用されるタグが含まれます。

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。使用可能なエンコードは、ISO-8859-1、ISO-8859-2、ISO-8859-4、ISO-8859-5、ISO-8859-7、ISO-8859-8、ISO-8859-9、ISO-8859-15、CP932、および UTF-8 です。

使用可能なロケールとそれらのエンコードの完全な一覧については、`arm¥config` ディレクトリにある `SupportedLocales.json` ファイルを参照してください。

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");
```

マルチスレッド環境の管理

このセクションの内容は以下のとおりです。

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。

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 ランタイム環境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 ランタイム環境でのマルチスレッドのサポート

DLIB ランタイム環境は、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）
- C 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）
- 一時ファイルシステム（つまり tmpnam が使用されるとき）
- C++ 静的記憶寿命を持ち、動的に初期化される関数ローカルオブジェクト
- C++ ロケール ファセットのハンドリング
- C++ 標準式のハンドリング
- C++ 停止および予期しない事象のハンドリング

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
C++ 例外エンジン	適用されません。

表 9: 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、個々のフォーマッタは保護されますが、完全な printf/scanf の呼び出しは保護されません。

マルチスレッドをサポートするランタイム環境で C++ が使用される場合、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します。

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マルチスレッドサポートを構成するには、以下の手順に従います。

1. マルチスレッドサポートを有効にするには、以下の手順に従います。

コマンドラインでは、リンカオプション --threaded_lib を使用します。

C++ を使用する場合、IDE はコンパイラオプション --guard_calls を自動的に使用して、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します。

IDE で、**[プロジェクト] > [オプション] > [一般オプション] > [ライブラリ設定] > [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。これは、リンカオプション --threaded_lib を呼び出し、C++ を使用する場合、IDE はコンパイラオプション --guard_calls を自動的に使用して、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します。

2 ランタイム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。

- ライブラリシステムの実装コードはインターフェースをロックします。
- ファイルストリーミングを使用している場合は、ライブラリファイルストリーミングの実装コードは、インターフェースをロックします。
- 実装コードは、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します。

`DLib_Threads.h` ファイルで必要な宣言の関数を見つけることができます。また詳細もあります。

3 プロジェクトをビルドします。

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。

スレッドの C++ 例外

スレッドの例外の使用は、スレッドの `main` 関数に `noexcept` 例外仕様がある限り、可能です。検出しない例外は、アプリケーションを正しく終了しません。

スレッド ローカル ストレージ (TLS) の設定

スレッド ローカル ストレージ (TLS) は、C（C11 で導入された `_Thread_local` 型指定子を介して）および C++（C++11 で導入された `thread_local` 型指定子を介して）の両方でサポートされます。TLS 変数はスレッド ローカル ストレージに配置され、スレッドが作成されるとメモリ領域がセットアップされます。スレッドが破壊されると、使用しているリソースはすべて、戻ります。C++ 環境では、スレッド ローカル ストレージがセットアップされると TLS オブジェクトが作成され、スレッド ローカル ストレージが破壊される前に TLS オブジェクトが破壊されます。

オペレーションシステムを使用している場合は、関連の TLS ドキュメントを参照してください。追加情報は、IAR ライブラリヘッダファイル `DLib_Threads.h` に記載されています。そのような特定のソースからの情報は、この通常の概要よりも優先されます。

メインスレッド

リンカオプション `--threaded_lib` を指定すると、TLS が有効になります。通常のシステムのスタートアップは、メインスレッドのスレッド ローカル ストレージの初期化を実行します。メインスレッドの初期化した TLS 変数は、リンカセクション `.tdata` に配置され、ゼロ初期化した TLS 変数はセクショ

ン .tbss に配置されます。その他のすべてのスレッドは作成されると、そのスレッド ローカル ストレージをセットアップします。--threaded_lib が指定されていないと、.tdata および .tbss セクションのコンテンツは、.data および .bss のように扱われます。ただし、そのような変数へのアクセスはまだ TLS アクセスです。

TLS のメモリの取得

TLS 変数はメモリに配置する必要があります。スレッドの存続期間中にメモリが利用可能であれば、これがどのように処理されるかは重要ではありません。スレッドのローカル記憶領域のサイズは、関数 __iar_tls_size（DLib_Threads.h で宣言済み）を呼び出すことで、取得できます。

TLS のメモリの取得の一部のオプション :

- OS からのメモリの取得
- ヒープメモリの割り当て
- スレッドが完了するまでリターンしない関数のスタックでのスペースの使用
- 専用のセクションでのスペースの使用

TLS メモリの初期化

TLS メモリを初期化するには、メモリエリアに、ポインタのある関数 __iar_tls_init （DLib_Threads.h で宣言済み）を呼び出します。

初期化関数はリンカセクション __iar_tls$$INIT_DATA の内容をメモリにコピーし、セクション __iar_tls$$DATA のサイズまでの残りのメモリをゼロ初期化します。C++ 環境では、関数 __iar_call_tls_ctors も呼び出されます。これは、セクション __iar_tls$$PREINIT_ARRAY のすべてのコンストラクタを実行します。初期化を実行すると、スレッド ローカル ストレージを使用することができます。すべての TLS 変数 にはその初期値があり、C++ 環境では、すべてのスレッドローカルオブジェクトが構築されています。

TLS メモリの割り当て解除

スレッドを破壊するときは、スレッド ローカル ストレージも破壊されます。C++ 環境では、スレッドローカルオブジェクトは、メモリ自体が処理される前に破壊されます。これは、関数 __call_thread_dtors（DLib_Threads.h で宣言済み）を呼び出すと行われます。メモリがハンドラ（ヒープ、OS など）から取得された場合は、そのメモリを返す必要があります。

例として、このコードの断片情報は、ヒープのスレッド ローカル ストレージを割り当てます。tp は、スレッドコントロールオブジェクトへのポインタです。

```
/* creating a new thread */
  ...
  /* initialize TLS */
  void * tls_mem = malloc(__iar_tls_size()); /* get memory */
  __iar_tls_init(tls_mem);                   /* init TLS in the */
                                             /* new memory */
  tp->tls_area = tls_mem;                    /* set the thread's */
                                  /* TLS area to the new memory */
  ...
  /* destroying a thread */
  ...
  /* destroy the TLS area */
  __call_thread_dtors();                /* only if C++ is used */
  free(tp->tls_area);                   /* return memory */
  ...
```

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び出し
- C++ からのアセンブラルーチンの呼び出し
- 呼び出し規約
- 呼出しフレーム情報

C 言語とアセンブラの結合

IAR C/C++ コンパイラ for Arm 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
- 組み込み関数（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。

組み込み関数

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、組み込み関数と呼びます。組み込み関数は、時間が重要なルーチンなどに便利です。

組み込み関数は、通常の関数呼び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み込み関数です。組み込み関数は、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。

使用可能な組み込み関数の詳細は、*組み込み関数*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。

これはこれは関数呼び出しとリターンの命令シーケンスでオーバヘッドになる原因になります。またコンパイラはいくつかのレジスタをスクラッチレジスタとみなします。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で書いたものの間で明確に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場合、インラインアセンブラの行がコンパイラによって生成されたコードを干渉しないという保証はありません。

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、いくつかの疑問点に遭遇します。

- C から呼び出せるようにアセンブラコードを記述する方法は？
- アセンブラコードが自分のパラメータの場所を探す方法と、呼び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び出す方法は？
- 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、189 ページの「*C からのアセンブラルーチンの呼び出し*」セクションで説明します。次の 2 つの疑問については、192 ページの「*呼び出し規約*」セクションで説明します。

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、202 ページの「*呼出しフレーム情報*」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、189 ページの「*C からのアセンブラルーチンの呼び出し*」と 191 ページの「*C++ からのアセンブラルーチンの呼び出し*」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、適切な組み込み関数がない場合）。
- 速度が重要なコードを C で記述すると遅くなりすぎるため、手動でシーケンス記述する。
- 時間が重要なコードを C で記述するとタイミングが適切でなくなるため、手動でシーケンス記述する。

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メントされることがあります。
- **32 ビットモード**の許可されたレジスタの別名は、`SP` (`R13`)、`LR` (`R14`)、および `PC` (`R15`) のみです。
- **64 ビットモード**の許可されたレジスタの別名は、`IP0` (`X16`)、`IP1` (`X17`)、`FP` (`X29`)、および `LR` (`X30`) 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。
- 実影響のないアセンブラ文の関数のインラインは、実行されません。
- 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*を暗黙的に示しません。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、オペランドおよびクラバーなしで `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、値がレジスタで渡されて戻されると想定しています。これは、関数がインライン化されている場合等は、常にそうはならない可能性がある方法です。
- `Adds` 命令の `s` は、条件フラグが更新されることを示します。これは `cc` クラバー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みなします。

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。コンパイラは、それらにリマークを発行します。

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 には、¥n で区切られた 1 つ以上の処理を含めることができます。各処理は、有効なアセンブラ命令、またはオプションのラベルが前に付いたデータ定義アセンブラディレクティブです。ラベルの前に空白を入れることはできません。また、: を後に付ける必要があります。

以下に例を示します。

```
asm("label:nop¥n"
    "b label");
```

注 : インラインアセンブラ文で定義するラベルは、その文でのみ参照されます。ループまたは状態コードには、これを使用できます。

1 つではなく、2 つのコロンを使用して、インラインアセンブラ文でラベルを定義する場合（例 :"*label*:: nop¥n"）、ラベルはインラインアセンブラ文だけでなく、モジュールでもパブリックになります。この機能はテスト用途だけを目的としています。

宣言した影響のないアセンブラ文は、volatile アセンブラ文とみなされます。これは全く最適化されないことを意味します。コンパイラは、そのようなアセンブラ文にリマークを発行します。

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* "（*式*）

以下に例を示します。

```
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。以下に例を示します。

```
int matrix[M][N];

void MatrixPreloadRow(int row)
{
  asm volatile ("pld [%0]" : : "r" (&matrix[row][0]));
}
```

オペランドの制約

オペランドの制約は、インラインアセンブラコードと前後の C/C++ コード間をオペランドが通過する方法を定義します。

これらは **32 ビットモード**の制約コードです :

制約	説明
r	式に汎用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 のように汎用レジスタペアを使用します。
Te	式に偶数の番号が付けられた汎用目的のレジスタを使用します。
To	式に奇数の番号が付けられた汎用目的のレジスタを使用します。
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定数（Arm および Thumb2）、または 0 ～ 255 の定数 (Thumb1)。

表 10: 32 ビットモードのインラインアセンブラ オペランドの制約

制約	説明
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	否定演算された場合に 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 あるいは 4 つ の連続した S、D、または Q のレジスタのベクタ。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。
digit	入力は、出力オペランド *digit* として同じ場所に配置する必要があります。最初の出力オペランドは 0 で、2 番目は 1 などです（出力としては無効）

表 10: 32 ビットモードのインラインアセンブラ オペランドの制約（続き）

これらは **64 ビットモード**の制約コードです :

制約	説明
r	式に 64 ビットの汎用レジスタを使用します。X0-X30。 代わりに、コンパイラで 32 ビットの汎用レジスタ W0-W31 を使用する場合は、w オペランドの修飾子を使用します。
i	定数値を持つイミディエート整数オペランド。シンボル定数を使用できます。
n	イミディエートオペランドで、i のエイリアスです。
I	オプションの 12 での左シフトがある、範囲 0–4095 の定数。ADD と SUB 命令が許可されている範囲。
J	オプションの 12 での左シフトがある、-4095 ～ 0 の範囲の定数。
K	32 ビット論理命令に有効なイミディエート定数。例、AND、ORR、EOR。

表 11: 64 ビットモードのインラインアセンブラ オペランドの制約

制約	説明
L	64 ビット論理命令に有効なイミディエート定数。例、AND、ORR、EOR。
M	32 ビットレジスタの出力先がある MOV 命令に有効なイミディエート定数。有効な値は、K 定数が許可されているすべての値と、MOVZ、MOVN、MOVK 命令が許可されている値です。
N	64 ビットレジスタの出力先がある MOV 命令に有効なイミディエート定数。有効な値は、L 定数が許可されているすべての値と、MOVZ、MOVN、MOVK 命令が許可されている値です。
w	SIMD または浮動小数点レジスタ V0-V31 を使用します。 b、h、s、d、および q オペランドの修飾子は、動作を上書きできます。
x	オペランドは 128 ビットのベクタ型です。コンパイラは低い SIMD レジスタ V0-V15 を使用します。
digit	入力は、出力オペランド *digit* として同じ場所に配置する必要があります。最初の出力オペランドは 0 で、2 番目は 1 などです（出力としては無効）

表 11: 64 ビットモードのインラインアセンブラ オペランドの制約（続き）

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 12: 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。以下に例を示します。

```
int Add(int term1, int term2)
{
  int sum;

  asm("add %[Rd],%[Rn],%[Rm]"
      : [Rd]"=r"(sum)
      : [Rn]"r" (term1), [Rm]"r" (term2));

  return sum;
}
```

入力オペランド

入力オペランドは制約修飾子を持つことはできませんが、有効な C の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 の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

出力オペランドは制約修飾子として = を持つ必要があり、C の式は l 値で、書き込み可能な場所を指定する必要があります。たとえば、書き込み専用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 の式に（左辺値として）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 の式は左辺値でなければならず、書き込み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） X0-X30、W0-W30（A64 の場合）	汎用レジスタ
Arm のモード S0-S31、D0-D31、Q0-Q15 と Thumb2 A64 の V0-V31、B0-B31、H0-H31、S0-S31、D0-D31、Q0-Q31	浮動小数点レジスタ
cc	条件フラグ (N、Z、V、C)
memory	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 13: 有効な上書きされるリソースの一覧

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、**32 ビットモード**の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	レジスタペアの小さい番号のレジスタ、またはイミディエイト定数の下位の 16 ビット。
H	レジスタペアの大きい番号のレジスタ、またはイミディエイト定数の上位の 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	レジスタペアで最下位のレジスタ。
R	レジスタペアで最上位のレジスタ。
M	1 つのレジスタまたはレジスタペアの場合、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。

表 14: 32 ビットモードのオペランドの修飾子と変換

修飾子	説明
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 14: 32 ビットモードのオペランドの修飾子と変換（続き）

次の表は、**64 ビットモード**の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
c	イミディエー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。
a	オペランドの定数は r です。四角のカッコ内のレジスタ名を出力します。メモリオペランドとしての使用に適しています。
n	イミディエートオペランド用です。# を処理しないマイナス演算子の値を出力します。
w	オペランドの定数は r です。その 32 ビット W 名を使用しているレジスタを出力します。
x	オペランドの定数は r です。その 64 ビット X 名を使用しているレジスタを出力します。
b	オペランドの定数は w または x です。その 8 ビット B 名を使用しているレジスタを出力します。
h	オペランドの定数は w または x です。その 16 ビット H 名を使用しているレジスタを出力します。
s	オペランドの定数は w または x です。その 32 ビット S 名を使用しているレジスタを出力します。
d	オペランドの定数は w または x です。その 64 ビット D 名を使用しているレジスタを出力します。
q	オペランドの定数は w または x です。その 128 ビット Q 名を使用しているレジスタを出力します。

表 15: 64 ビットモードのオペランドの修飾子と変換

上書きされるメモリの使い方の例

```
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び出し

C から呼び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。

- 呼び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び出しの前に 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ことを知る 1 つの方法は、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ことです。

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。

注 : 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、コールフレーム情報 (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。呼び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 では、リストファイルから CFI ディレクティブを除外するには、**[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、**[コールフレーム情報を含める]** サブオプションの選択を解除します。

コマンドラインで、リストファイルから CFI ディレクティブを除外するには、-lA オプションではなくオプション -lB を使用します。

注 : C-SPY の **[コールスタック]** ウィンドウを機能させるには、CFI 情報をソースコードにインクルードする必要があります。

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。

- 呼び出し規約
- 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、202 ページの「*呼出しフレーム情報*」を参照してください。

C++ からのアセンブラルーチンの呼び出し

C の呼び出し規約は、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、呼び出し規約は C の呼び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++ から呼び出されます。

```
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び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、余分な呼び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);
  }
};
```

呼び出し規約

呼び出し規約とは、プログラム内の関数が別の関数を呼び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、呼び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び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、実際の呼び出し規約の例を示します。

コンパイラで使用される呼び出し規約は AEABI の一部である AAPCS (Procedure Call Standard for the Arm Architecture) に準拠します。詳細は 242 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 呼び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び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、この情報から呼び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++ から呼び出すには、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。関数が別の関数を呼び出した後もレジスタの値を必要とする場合は、呼び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。

32 ビットモードの場合、レジスタ `R0` から `R3`、および `R12` は、関数でスクラッチレジスタとして使用されます。**64 ビットモードの場合**、スクラッチレジスタとして使用されるレジスタは `X0` から `X15` です。

注 : ベニアの命令が自動的に挿入されたため、アセンブラ関数を呼び出す場合、**32 ビットモードの場合**は `R12`、**64 ビットモードの場合**は `X16` と `X17` もスクラッチレジスタです。

保護レジスタ

一方、保護レジスタは、他の関数の呼び出し後も保持されます。呼び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。

32 ビットモードの場合、レジスタ `R4` から `R11` は、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。**64 ビットモードの場合**、レジスタ `X18` から `X30` は保護レジスタです。

32 ビットモードの専用レジスタ

32 ビットモードのレジスタの場合、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。割り込みが発生すると、スタックポインタ

が示す位置より下の要素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。

64 ビットモードの専用レジスタ

64 ビットモードのレジスタは、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 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。割り込みが発生すると、スタックポインタが示す位置より下の要素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 16 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。
- リンクレジスタ LR/X30 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、呼び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。パラメータの引渡しに使用できるレジスタ数は非常に少ないため、レジスタが不足する場合は、残りのパラメータはスタックに引き渡されます。これらの規則には次の例外が適用されます。

- パラメータを受け入れて値を返すソフトウェア割り込み関数を除いて、割り込み関数にはどんなパラメータも指定できません。
- ソフトウェア割り込み関数は、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り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、189 ページの「*C からのアセンブラルーチンの呼び出し*」を参照してください。

32 ビットモードのレジスタパラメータ

パラメータの引渡しに利用可能な 32 ビットモードのレジスタ

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` または `R2:R3`

表 16: パラメータの引渡しに使用する 32 ビットモードのレジスタ

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び出される場合、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。

64 ビットモードのレジスタパラメータ

パラメータの引渡しに利用可能な 64 ビットモードのレジスタ：

パラメータ	レジスタでの引渡し
整数、ポインタ、小さい構築（最大 8 バイト）	最初の空きレジスタを使用して渡されます：X0-X7
小さい構造（9–16 バイト）	最初の空きレジスタペアを使用して渡されます：X0-X7

表 17: パラメータの引渡しに使用する 64 ビットモードのレジスタ

浮動小数点数値	最初の空きレジスタを使用して渡されます：V0-V7
均一の構造（浮動小数点またはベクタ型の 1–4 要素）	最初の空きレジスタを使用して渡されます：V0-V7（各レジスタに 1 つの要素）
大きい構造	ポインタは最初の空きレジスタを使用して渡されます：X0-X7

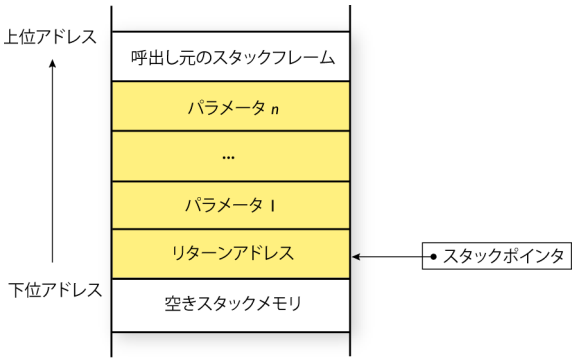
レジスタをパラメータに割り当てるプロセスは単純明快です。パラメータを左から右の順に調べ、最初のパラメータを空きレジスタに代入します。空きレジスタがない場合は、パラメータはスタックを逆方向で使用して引き渡されます。

64 ビットモードの場合、パラメータのサイズと統一されたビットのみが許可されます。よって、呼び出した関数は通常、32 ビットより小さいサイズの符号拡張およびゼロ拡張パラメータです。

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、呼び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。呼び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、呼び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、呼び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。

リターン値の 32 ビットモードに使用するレジスタ

リターン値の 32 ビットモードで使用できるレジスタは、`R0` および `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 18: リターン値の 32 ビットモードに使用するレジスタ

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。

リターン値のために使用した 64 ビットモードのレジスタ

リターン値	レジスタでの引渡し
整数、ポインタ、小さい構築（最大 8 バイト）	X0
小さい構造（9–16 バイト）	X0-X1
浮動小数点数値	V0
均一の構造（浮動小数点またはベクタ型の 1–4 要素）	V0-V3（各レジスタに 1 つの要素）
大きい構造	X8 のて呼出し元に引き渡されるポインタ

表 19: リターン値のために使用した 64 ビットモードのレジスタ

リターン値のサイズと同様のリターン値のビットのみが許可されます。

関数終了時のスタックのレイアウト

呼び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、呼び出し元で行うべきです。

32 ビットモード : 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、終了時に呼び出し元に戻ります。

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}  ; スタックのアライメント 8 を保持
bl      func

; 何らかの処理

pop     {r4-r6,pc}  ; リターン

end
```

64 ビットモード : 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、スタックでプッシュできます。関数の終了では、これらすべてのレジスタはスタックにリストアする必要があります。

以下の例は、この動作を示しています。

```
        name    call
        section .text:CODE
        extern  func

        strp    x9, lr, [sp, #16]!  ; スタックのアライメント 16 を保持
        bl      func

        ; 何らかの処理

        ldrp    x9, x7, [sp, #16]
        ret

        end
```

例

以下では、宣言の例や対応する呼び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);
```

32 ビットモードの場合、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
```

64 ビットモードの場合、この関数は、1 つのパラメータをレジスタ X0 を使用して引き渡し、リターン値をレジスタ X0 を使用して呼出し元に返します。宣言と互換性のある関連のアセンブラルーチンは以下のようになります。

```
name    return
section .text:CODE
add     x0, x0, #1
ret
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);
```

32 ビットモードの場合、構造体メンバ a、b、c、および d の値は、レジスタ R0-R3 に渡されます。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。呼び出し元関数は、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び出し元に返されます。

64 ビットモードの場合、x の値は、X0 と X1 に渡され、y は X2 に渡されます。リターン値は X0 に渡されます。

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、呼び出し元の関数の役割です。**32 ビットモードの場合**、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。**64 ビットモードの場合**、リターン値が格納されるべき位置へのポインタは、X8 内で引き渡されます。パラメータ x は X0 に渡されます。

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。**32 ビットモードの場合**、パラメータ x は R0 に渡され、リターン値は R0 で返されます。**64 ビットモードの場合**、パラメータ x は X0 に渡され、リターン値は X0 で返されます。

呼出しフレーム情報

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び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、呼び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、呼び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。

呼び出し規約に関する詳細記述では、広範なコールフレーム情報を必要とする場合があります。多くの場合は、より限定的なアプローチで十分です。

コールフレーム情報を記述するには、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*
- 呼び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、いつスタックポインタが変更されたか、いつ保護レジスタがスタックに待避、復帰したかについての情報が含まれます。

32 ビットモード呼出しフレーム情報リソース：

リソース	説明
CFA R13	スタックのコールフレーム
R0-R12	プロセッサ汎用 32 ビットレジスタ
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 20: 32 ビットモード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

64 ビットモード呼出しフレーム情報リソース：

リソース	説明
X0-X29	プロセッサ汎用 64 ビットレジスタ
X30	リンクレジスタ、LR
SP	スタックポインタ
CFA SP	スタックのコールフレーム
ELR_mode	例外レベル
V0-V31	ベクタ浮動小数点 (VFP) 64 ビットレジスタ（実際には 128 ビットですが、ABI はこれを扱うことができません）

表 21: 名前ブロックで定義されている 64 ビットモード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

コールフレーム情報を正しく処理するアセンブラ言語ルーチンを作成するには、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。コールフレーム情報（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 **[コール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、**32 ビットモードの**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

IAR C/C++ コンパイラ for Arm は、INCITS/ISO/IEC 9899:2018 標準をサポートし、これは C18 として知られています。C18 は、新しい言語機能を導入しなくても C11 (INCITS/ISO/IEC 9899:2012) の不具合を検出します。これは、C11 標準もサポートしていることを意味します。このガイドでは、C18 標準が *標準 C* を意味し、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。

コンパイラは、C18 標準またはスーパーセットで書かれたソースコードを許可します。

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。

標準 C が有効な状態では、IAR C/C++ コンパイラ for Arm は、スレッドに関連したシステムのヘッダファイルを除いて、すべての C11 ソースコードファイルをコンパイルできます。

ISO/IEC/IEEE 60559 として知られている、標準 C にバインドされた浮動小数点標準は、IEEE 754 フォーマットとほぼ同等です。

Annex K は、標準 C の Annex K（*境界チェックインターフェース*）をサポートします。144 ページの「*バウンドチェック機能*」を参照してください。

C 標準の様々なバージョン間のちがいの概要については、Wikipedia 記事の *C18（C 標準の改訂）*、*C11（C 標準の改訂）*、または *C99* を参照してください。

拡張の概要

コンパイラでは、C 標準の機能のほかに、組み込み業界で使用され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、209 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 つの 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。

- *組み込み関数*

  組み込み関数は、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に便利です。組み込み関数は、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。組み込み関数の使用方法については、177 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、「*組み込み関数*」を参照してください。

- *ライブラリ関数*

  DLIB ランタイム環境は、組み込みシステムに適用される C/C++ 標準ライブラリで C と C++ ライブラリ定義を提供します。詳細については、521 ページの「*DLIB ランタイム環境 — 実装の詳細*」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 規格外の記述は全てエラーになります。
なし	**標準**	すべての *C 規格に対する緩和*が有効ですが、*組み込みシステムのプログラミングの拡張*は有効になりません。拡張については、209 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 22: 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 1］**>**［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。

- *組み込みシステムプログラミングのための拡張* — 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み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、便利ではあるが小規模な構文の拡張。212 ページの「*標準 C に対する緩和*」を参照してください。

組み込みシステムプログラミングのための拡張

以下の言語拡張は、C と C++ の両方のプログラミング言語で使用可能なもので、組み込みシステムのプログラミングに適しています。

- *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。

- *絶対アドレスあるいは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、変数や関数を指定されたセクションに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、254 ページの「*データと関数のメモリ配置制御*」、441 ページの「*location*」を参照してください。

- *アライメントの制御*

  各データタイプには、それ自身のアライメントがあります。詳細については、389 ページの「*アライメント*」を参照してください。アラインメントを変更する場合には、__packed データ型属性、#pragma pack ディレクティブ、および #pragma data_alignment ディレクティブを利用できます。オブジェクトのアライ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。

  標準 C ファイル stdalign.h も参照してください。

- *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、393 ページの「*ビットフィールド*」を参照してください。

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*の最初のアドレスバイトを返します。
__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したりする場合、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することができます。

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。この組み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 23: セクション演算子とそのシンボル

注 : 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りません。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、演算子または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"
...
section_start_address = __section_begin("MYSECTION");
```

449 ページの「*section*」、441 ページの「*location*」を参照してください。

標準 C 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、小規模な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` 型を要素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` や `union` 指定子末尾にセミコロンがなくても受け入れます

  `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、401 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。

- バイナリ整数リテラル (`0b...`) は、サポートされています。

- `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 *`、および `char *`）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。

- - 左辺値でない配列

  左辺値でない配列式は、使用時に配列の最初の要素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、関数の最後でも使用でき、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)。

- 空の翻訳単位

  翻訳単位（入力ファイル）は宣言が空になっていることが可能です。

- ポインタ型の割り当て

  ポインタ型の割り当ては、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 const int ** への代入など)。比較およびポインタが異なるようなペアのポインタ型も可能です。ワーニングが出力されます。

- 異なる関数型のポインタ

  異なる関数型のポインタは明示的な型キャストなしで等式 (==) または不等式 (!=) へ代入されたり、比較されたりすることが可能です。ワーニングが出力されます。この拡張は C++ モードでは使用できません。

- アセンブラ文

  アセンブラ文が許可されます。これは厳密な C モードでは無効です。それは、暗示的に宣言された asm 関数の呼び出すための C 標準と矛盾するためです。

- #include_next

  非標準のプリプロセスする #include_next ディレクティブがサポートされます。これは #include ディレクティブの変数です。現在のソースファイ

ル（#include_next ディレクティブが含まれているもの）があるディレクトリに従う検索パスで、ディレクトリにある名前が付けられたファイルだけを検索します。これは GNU C コンパイラで実装されている拡張です。

- #warning

  非標準のプリプロセスする #warning ディレクティブがサポートされます。それは #error ディレクティブとよく似ていますが、処理するときに、壊滅的なエラーではなく警告を発します。このディレクティブは厳密モードでは認識されません。これは GNU C コンパイラで実装されている拡張です。

- 文字列の連結

  混合した文字連結が許可されています。

  ```
  wchar_t * str="a" L "b";
  ```

- GNU スタイル ステートメント式（中括弧で囲んだステートメントのシーケンス）は許可されます。
- GNU スタイルのケース範囲が許可されます。(case 1..5:).
- GNU スタイルで指定されたイニシャライザの範囲が許可されます。

  例：int widths[] = {[0...9] = 1, [10...99] = 2, [100] = 3};

- typeof

  IAR 拡張機能が有効になっている場合、式タイプへの参照を示すために、非標準の演算子 typeof がサポートされます。構文は sizeof と同じですが、セマンティクスは typedef で定義された型名に似ています。これは GNU C コンパイラで実装されている拡張です。

C++ の使用

- 概要 — 標準 C++
- C++ のサポートの有効化
- C++ の機能の説明
- C++ 言語拡張
- DLIB C++ ライブラリから Libc++ C++ ライブラリへの移行
- EC++ または EEC++ からコードを移植

概要 — 標準 C++

IAR C++ の実装は、スレッド関連のシステムヘッダに依存するソースコードまたは `filesystem` ヘッダを除いて、ISO/IEC 14882:2014 C++ (“C++14”) または 14882:2017 C++ (“C++17”) 標準に完全に準拠しています。このガイドでは、ISO/IEC 14882:2017 C++ 標準は、標準 C を指しています。

アトミック処理は、関数セットがそれらをサポートするコアで使用できます。528 ページの「*アトミック処理*」を参照してください。

IAR C/C++ コンパイラは、C++17 標準またはスーパーセットで書かれたソースコードを許可します。

- DLIB C++14 ライブラリを使用しているときは、ライブラリのサポートで必要な C++17 のそれらの機能は利用できません。
- Libc++ C++17 ライブラリを使用しているときは、特に説明がない限り、C++17 のすべての機能を利用できます。

C++ 標準の様々なバージョン間のちがいの概要については、Wikipedia の記事 *C++17*、*C++14*、*C++11*、または *C++*（C++98 の情報）を参照してください。

注： DLIB ライブラリ (DLIB5) の古いバージョンからの C++ 標準テンプレートライブラリ (STL) ヘッダのセットもあります。機能は少なくなりますが、`map/set` および `vector` の非常に小さいコードになる場合があります。`arm/doc/HelpDLIB5.html` にあるドキュメントを参照してください。

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` を使用すると、以下によってコンパイル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、319 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、322 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 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、スタック上で呼び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び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++ のサポートの有効化

コンパイラのデフォルト言語は 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。299 ページの「*--c++*」を参照してください。

IDE の 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 1］> ［言語］> ［C++］**を選択します。

C++ の機能の説明

IAR C/C++ コンパイラ for Arm 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。

位置演算子 @ と #pragma location ディレクティブは、静的データメンバ上および、すべてのメンバ関数とともに使用することができます。

テンプレート

C++ は、C++ 標準に従ったテンプレートをサポートします。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。

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
}
```

割り込みで静的クラスオブジェクトを使用する

割り込み関数が、（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り込みが発生すると、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、これらの割り込みが有効になっておらず、main から戻ったり、exit を呼び出すときに無効になっていることを確認してください。システムの起動について詳しくは、155 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 ら戻ったり、exit を呼び出す際に破棄されます。

new ハンドラを使用する

メモリの枯渇に対応するには、set_new_handler 関数を使用します。

例外が有効な状態での標準 C++ の NEW ハンドラ

set_new_handler を呼び出さないか、NULL new ハンドラを使用して呼び出す場合、例外が有効になっていて演算子 new が十分なメモリの割当てに失敗すると、演算子 new が std::bad_alloc をスローします。例外が有効でない場合、演算子 new は代わりに abort を呼び出します。

NULL 以外の new ハンドラを用いて set_new_handler を呼び出す場合、演算子 new が十分なメモリの割当てに失敗すると、演算子 new によって、提供された new ハンドラが呼び出されます。new 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、new ハンドラは std::bad_alloc 例外をスローすることもできます。演算子 new の派生型 nothrow は、例外が有効で new ハンドラが std::bad_alloc をスローする場合に、new ハンドラがある状態でのみ NULL を返します。

例外が無効な状態での標準 C++ の NEW ハンドラ

set_new_handler を呼び出さないか、NULL new ハンドラを使用して呼び出す場合、operator new が十分なメモリの割当てに失敗すると、abort が呼び出されます。new 演算子の派生型 nothrow は、代わりに NULL を返します。

NULL 以外の new ハンドラを用いて set_new_handler を呼び出す場合、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 new ハンドラが呼び出されます。new 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、new ハンドラがある状態で NULL を返すことはありません。

これは、new の派生型 nothrow を使用しているのと同じ動作です。

C-SPY でのデバッグサポート

C-SPY には、STL コンテナ用に組み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。

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。

C++ 言語拡張

コンパイラを 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
};
```

- 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、包含するクラスに無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。以下に例を示します。

```
typedef union
{
  int i,j;
} U;  // U は再利用可能な無名共用体を識別

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* および *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。以下に例を示します。

```
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 クラスの構文では、non-class 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。以下に例を示します。

```
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
};
```

- それは、struct の最後のメンバとしてサイズなしまたはサイズ 0 の配列を指定できます。以下に例を示します。

```
typedef struct
{
  int i;
  char ir[0]; // Zero-length array
};

typedef struct
{
  int i;
  char ir[];  // Zero-length array
};
```

- 不完全型の配列

  配列では、不完全な struct、union、enum、または class 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。

- 文字列の連結

  混合した文字列リテラルの連結が許可されています。

```
wchar_t * str = "a" L "b";
```

- 最後のカンマ

  列挙型の定義にある最後のカンマは、表示されないでも許可されます。

ただし、C の拡張が説明されているところは、C++ モードも許可されます。

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。

DLIB C++ ライブラリから Libc++ C++ ライブラリへの移行

Libc++ ライブラリのノーマルの設定はありません。ロケール、ファイル記述子などのサポートは常に含まれています。

Libc++ ライブラリは C++17 ライブラリです。C++17 では、C++14 で廃止された一部の機能は削除されました。例には `std::auto_ptr`、`std::random_shuffle`、および `std::mem_fun` があります。Libc++ ライブラリを使用するときに、プリプロセッサシンボル `_LIBCPP_ENABLE_CXX17_REMOVED_FEATURES` を定義してこれらの機能をサポートできます。

注： DLIB C++14 ライブラリの一部のシステムヘッダは Libc++ ではサポートされていません。その逆もあります。523 ページの「*C++ ヘッダファイル*」の説明を参照してください。

EC++ または EEC++ からコードを移植

標準 C++ は EC++ や EEC++ よりももっと大きな言語という事実から、コンパイルから EC++ と EEC++ コードを回避してしまう 2 つの問題があります。

- ライブラリが `namespace std` に配置される。

  2 つの解決オプション：

  - `std::` のライブラリシンボルを使用したそれぞれの接頭辞。
  - C++ システムヘッダファイルに、最後の include ディレクティブのあとに、`using namespace std;` を挿入します。
- ライブラリシンボルの中には、名前または通過するパラメータが変更されているものがあります。

  これを解決するには、新しい名前と通過するパラメータを検索します。

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
- イメージの整合性を検証するチェックサム計算
- AEABI への準拠
- CMSIS 統合（32 ビットモード）
- Arm TrustZone®
- $Super$$ および $Sub$$ を使用したシンボル定義のパッチ

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

サポートされている出力フォーマットの完全なリストは、オプションなしで `ielftool` を実行します。

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、591 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アプリケーション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は、下の 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する

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、120 ページの「*スタックメモリの設定*」、266 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。

スタックのアライメント

32 ビットモードでは、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 バイトに整列されたアドレスに初期化します。

64 ビットモードでは、デフォルトの cstartup コードは、すべてのスタックを自動的に 16 バイトに整列されたアドレスに初期化します。

スタックのアライメントの詳細については、192 ページの「*呼び出し規約*」、194 ページの「*32 ビットモードの専用レジスタ*」、197 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

64 ビット Arm コアおよび 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
割り込み	IRQ_STACK	汎用 (IRQ) 割り込みハンドラのスタック
高速割り込み	FIQ_STACK	高速 (FIQ) 割り込みハンドラのスタック
未定義	UND_STACK	未定義命令割り込みのスタック。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。
中止	ABT_STACK	命令フェッチおよびデータアクセスメモリ中断割り込みハンドラのスタック

表 24: 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。

- ベーシック、アドバンスト、およびフリーなしヒープのメモリ割当て
- 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、120 ページの「*ヒープメモリの設定*」を参照してください

ヒープメモリハンドラ

システムライブラリには、3 つの異なるヒープメモリハンドラ（*ベーシック*、*アドバンスト*、および*フリーなし*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メモリ割り当てルーチンのコールがあるが、ヒープ割り当て解除ルーチンのコールがない場合は、リンカは、自動的にフリーなしヒープを選択します。
- アプリケーションでヒープメモリ割り当てルーチンのコールがある場合、リンカは自動的にアドバンストヒープを選択します。
- ヒープメモリ割り当てルーチンのコールが、ライブラリなどにある場合、リンカは自動的にベーシックヒープを選択します。

注 : 使用している製品がサイズ制限版の評価ライセンスの場合は、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`) は非常に単純なヒープ割当て機能で、ヒープをあまり使用しないアプリケーションでの使用に適しています。特に、ヒープメモリを割り当てるだけでまったく解放しないアプリケーションに使用できます。ベーシックヒープは特別に速いということはなく、繰り返しメモリを解放するアプリケーションで使用すると、ヒープが不要にフラグメント化する確率が高くなります。ベーシックヒープのコードは、アドバンストヒープの場合に比べて極めて小さいものです。350 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、ヒープを重点的に使用するアプリケーションに対して効率的なメモリ管理を提供します。特に、繰り返しメモリを割り当ててそれを解放するアプリケーションの場合、空間と時間の両方でオーバヘッドが少なくなります。アドバンストヒープのコードは、ベーシックヒープの場合に比べて極めて大きくなります。350 ページの「*--advanced_heap*」を参照してください。この定義については、529 ページの「*iar_dlmalloc.h*」を参照してください。
- フリーなしヒープ (`--no_free_heap`) は、一番小さいヒープの反映です。このヒープは `free` または `realloc` をサポートしていません。373 ページの「*--no_free_heap*」を参照してください。

ヒープサイズと標準 I/O

「ノーマル」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。「フル」の設定のように、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 バイトに設定されていることに注意する必要があります。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。

ヒープアライメント

32 ビットモードの場合、ヒープは 8 バイトのアラインメントされたアドレスにアラインメントされます。

64 ビットモードの場合、ヒープは 16 バイトのアラインメントされたアドレスにアラインメントされます。

ヒープのアライメントについての詳細は、120 ページの「*ヒープメモリの設定*」を参照してください。

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。ILINK 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 `SIZEOF` を使用して名前を付けられたセクションまたはブロック。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 MY_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、指定したサイズの要素の配列を動
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;
}
```

イメージの整合性を検証するチェックサム計算

このページには、チェックサムの計算に関する情報が含まれます。

- 233 ページの「*チェックサム計算の概要*」
- 235 ページの「*チェックサムの計算と検証*」
- 240 ページの「*チェックサム計算のトラブルシューティング*」

詳細については、591 ページの「*IAR ELF ツール — ielftool*」を参照してください。

チェックサム計算の概要

チェックサムを使用して、イメージが実行時に最初にチェックサムが生成されたときと同じかどうかを検証できます。つまり、そのイメージが破損していないかを確認します。

これは次のように機能します。

- 最初のチェックサムが必要です。

  IAR ELF ツール (`ielftool`) を使用して初期のチェックサムを生成するか、サードパーティ製のチェックサムを使用できます。
- 実行時に 2 つ目のチェックサムを生成する必要があります。

  実行時にチェックサムを計算するためのアプリケーションのソースコードに特定のコードを追加するか、または実行時にチェックサムを計算するデバイスとして専用のハードウェアを使用することも可能です。

- 2 つのチェックサムを比較して異なる場合に、適切な処理を行うためのアプリケーションソースコードに、特定のコードを追加する必要があります。

  2 つのチェックサムが同じ方法で計算され、イメージにエラーがない場合、チェックサムは同じになるはずです。同じでなければ、2 つのチェックサムが同じ方法で生成されなかったと考えるべきです。

2 つのチェックサムを生成するときに使用したソリューションが何であれ、両方のチェックサムが*まったく同じ方法*で計算されているか確認する必要があります。初期のチェックサムに `ielftool` を、実行時にはソフトウェアベースの計算にをそれぞれ使用する場合、両方のチェックサムの生成に対して完全なコントロールの権限を持つことになります。ただし、サードパーティ製のチェックサムを初期のチェックサムに使用し、実行時のチェックサム計算に何らかのハードウェアサポートを使用する場合、検討すべき追加の要件が加わることがあります。

2 つのチェックサムについては、常に検討すべき選択肢のほか、追加の要件がある場合にみ選択する項目があります。なお、どちらのチェックサムに対しても、すべての詳細が同じでなければなりません。

常に検討すべき事項 :

- *チェックサムの範囲*

  チェックサムを用いて検証するメモリ範囲（複数可）。一般的には、すべての ROM メモリについてチェックサムを計算することをお勧めします。しかし、特定範囲についてのみ、チェックサムを計算することをお勧めします。以下の点に注意してください。

  - 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。
  - チェックサムは、すべてのメモリ範囲で最下位から最上位アドレスへ計算する必要があります。
  - 各メモリ範囲は定義された順序通りに検証する必要があります（たとえば、`0x100-0x1FF`、`0x400-0x4FF` は `0x400-0x4FF`、`0x100-0x1FF` と同じではありません）。
  - 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。
  - チェックサムは、チェックサムやソフトウェアブレークポイントを含むメモリ範囲については計算しないでください。

- *チェックサムのアルゴリズムおよびサイズ*

  どのアルゴリズムが最適か考える必要があります。Sum（単純な算術合計）または CRC（最もよく使用されるアルゴリズム）という 2 つの基本的な選択肢があります。CRC の場合、チェックサムに使用するサイズは、2 バイ

ト、4 バイト、8 バイトです。定義済の多項式がサイズに応じた十分な幅があると、エラー検出の効果がより高くなります。定義済の多項式はほとんどの場合において機能しますが、データセットによっては機能しない場合もあります。その場合は、独自の多項式を指定することができます。エラー検出メカニズムが必要な場合は、チェックサムのサイズに定義済の CRC アルゴリズムを使用してください。通常は CRC16 または CRC32 です。

注 :*n* ビットの多項式の場合、*n* 番目のビットは常に "1" に設定済みとみなされます。16 ビットの多項式（CRC16 など）の場合、`0x11021` は `0x1021` と同じことになります。

不均一な分布のデータセットの適切な多項式を選択する詳細については、例えば *Tannenbaum, A.S., Computer Networks, Prentice Hall 1981, ISBN: 0131646990* の 3.5.3 章を参照ください。

- *フィル*

  チェックサムの計算の前に、チェックサムの範囲のすべてのバイトについて、十分に定義された値が必要です。不明な値を持つバイトは通常、アライメントのために追加された*パッドバイト*です。つまり、計算時にどのフィルパターンを使用するか指定しなければなりません。一般的には `0xFF` または `0x00` です。

- *初期値*

  チェックサムには常に明示的な初期値が必要です。

これらの必須の詳細に加えて、他の検討事項があることもあります。通常は、サードパーティ製のチェックサムを使用し、それを Rocksoft チェックサムモデルに適合させる場合や、実行時にチェックサムを生成するためのハードウェアサポートを使用するときなどがそうです。`ielftool` にはアライメントや補数、ビット順、ワード内のバイトオーダ、チェックサムのユニットサイズを制御するためのサポートも用意されています。

チェックサムの計算と検証

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。

1. コマンドラインから `ielftool` を使用する場合、まず計算されたチェックサムにメモリの場所を割り当てる必要があります。

   注 : 代わりに IDE（およびコマンドラインではない）`__checksum`、`__checksum_begin` シンボルを使用する場合、および `__checksum_end` チェックサム計算時に `.checksum` セクションが*自動的に*割り当てられる場合、このステップは省略できます。

   2 つの方法でメモリの場所を割り当てできます。

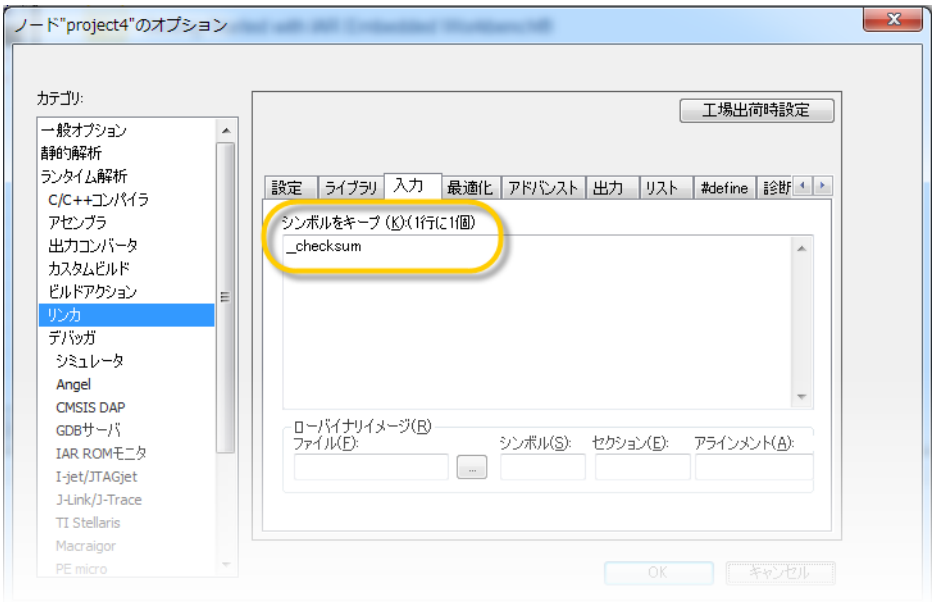
- 適切なサイズのグローバルC/C++またはアセンブラ定数シンボルを作成し、特定のセクション内に存在することで（この例では .checksum）
- リンカオプション --place_holder を使用することで

  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 にアライメント 4 で割り当てるには、以下のように指定します。

  ```
  --place_holder __checksum,2,.checksum,4
  ```

2 .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します。

代わりに、**[プロジェクト] > [オプション] > [リンカ] > [入力]** を選択し __checksum を指定します。



3 .checksum セクションの配置を制御するには、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 block CHECKSUM    { ro section .checksum };
place in ROM_region  { ro, first block CHECKSUM };
```

注 : このステップを省略することは可能ですが、その場合は .checksum セクションは他のリードオンリーのデータとともに自動的に配置されます。

4 チェックサムの計算のために ielftool を設定する場合、いくつか検討すべき基本事項があります。

- チェックサムアルゴリズム

  使用するチェックサムのアルゴリズムを選択します。この例では CRC16 アルゴリズムを使用します。

- メモリ範囲

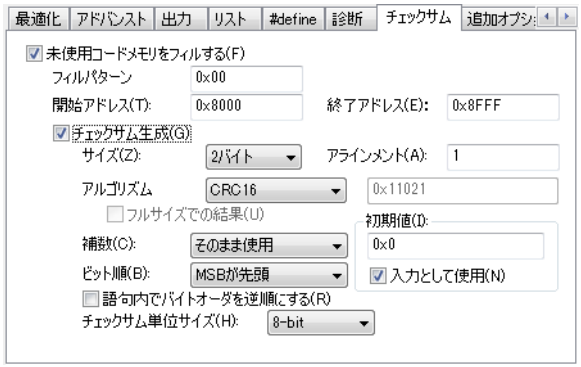
  IDE を使用して、チェックサムを計算する 1 つのメモリ範囲を指定できます。コマンドラインから、範囲を指定できます。

- フィルパターン

  フィルパターンを指定します。不明な値を持つバイトの場合、通常は 0xFF または 0x00 です。フィルパターンはすべてのチェックサムの範囲で使用されます。

詳細については、233 ページの「*チェックサム計算の概要*」を参照してください。

IDE から ielftool を実行するには、**［プロジェクト］>［オプション］>［リンカ］>［チェックサム］**を選択し、たとえば以下のように設定します。



一番簡単な場合は、以下のオプションを無視（またはデフォルト設定のまま）できます。**補数**、**ビット順**、**語句内のバイトオーダを逆順にする**、および**チェックサム単位サイズ**。

コマンドラインから ielftool を実行するには、コマンドをたとえば以下のように指定します。

```
ielftool --fill=0x00;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x8002-0x8FFF sourceFile.out
destinationFile.out
```

注： `ielftool` では、--strip 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。リンカオプションで `--strip` を使用している場合、それを削除し、代わりに `ielftool` の `--strip` オプションを使用します。

チェックサムはプロジェクトをビルドする際に後から作成され、セクション `.checksum` 内の指定されたシンボル `__checksum` 内に自動的に配置されます。

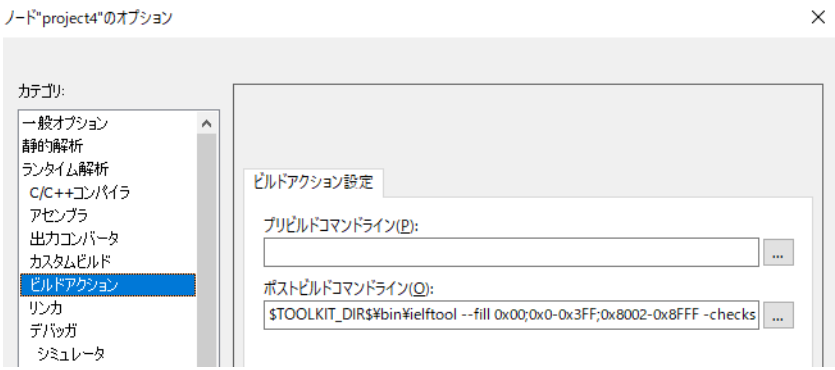
5 1 つではなく、複数範囲を指定できます。

IDE を使用する場合には、次の手順を実行します。

- **［プロジェクト］>［オプション］>［リンカ］>［チェックサム］**を選択し、**未使用コードメモリを埋める**を選択解除します。
- **［プロジェクト］>［オプション］>［ビルドアクション］**を選択し、**ポストビルドコマンドラインの**テキストフィールドで、必要なコマンドの残りと一緒に範囲を指定します。例：

```
$TOOLKIT_DIR$¥bin¥ielftool "$TARGET_PATH$" "$TARGET_PATH$"
--fill 0x00;0x0-0x3FF;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF;0x8002-0x8FFF
```

例えば、ユーザプロジェクトでは、*output.out* をユーザプロジェクトの出ファイル名に書き換えてください。



コマンドラインを使用する場合には、次のように範囲を指定します。

```
ielftool output.out output.out
--fill 0x00;0x0-0x3FF;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF;0x8002-0x8FFF
```

例えば、ユーザプロジェクトでは、*output.out* をユーザプロジェクトの出ファイル名に書き換えてください。

6 チェックサム計算の関数をソースコードに追加します。ielftool で計算されるチェックサムと同じアルゴリズムおよび設定が使用されるように注意してください。たとえば、crc16 アルゴリズムの派生形で、メモリのフットプリントが小さいもの（より多くのメモリを使用する高速の派生形とは対照的に）の場合、以下のように記述します。

```
unsigned short SmallCrc16(uint16_t
 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;
}
```

この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。

7 チェックサムを計算して 2 つのチェックサムを比較し、チェックサムの値が一致しなければ適切に処置する関数への呼び出しがアプリケーションに含まれていることも確認してください。

以下のコードは、アプリケーションでどのようにチェックサムが計算されるか、また ielftool により生成されたチェックサムと比較した場合の例を示します。

```
/* 計算されたチェックサム */

/* リンカにより生成されたシンボル */
extern unsigned short const __checksum;
extern int __checksum_begin;
extern int __checksum_end;


void TestChecksum()
{
  unsigned short calc = 0;
  unsigned char zeros[2] = {0, 0};

  /* チェックサム計算の実行 */
  calc = SmallCrc16(0,
                    (unsigned char *) &__checksum_begin,
                    ((unsigned char *) &__checksum_end -
                    ((unsigned char *) &__checksum_begin)+1));

  /* バイトのシーケンスの末尾をゼロでフィルします。*/
  calc = SmallCrc16(calc, zeros, 2);

  /* チェックサムのテスト */
  if (calc != __checksum)
  {
    printf("Incorrect checksum!¥n");
    abort();   /* 失敗 */
  }

  /* 正しいチェックサムです */
}
```

8 アプリケーションプロジェクトをビルドし、ダウンロードします。

ビルド時、ielftool がチェックサムを作成し、セクション .checksum 内の指定されたシンボル __checksum 内に配置します。

9 [ダウンロードとデバッグ] を選択して、C-SPY デバッガを起動します。

実行時、ielftool のチェックサムと、アプリケーションが計算したチェックサムが同一であることが必要です。

チェックサム計算のトラブルシューティング

2 つのチェックサムが一致しない場合、いくつか原因が考えられます。以下にトラブルシューティングのヒントを示します。

- 可能であれば、チェックサムの突合せの際に小さいサンプルを使用してください。

- 両方のチェックサムの計算で、まったく同じメモリ範囲が使用されているか確認します。

  この確認を簡単にするため、`ielftool` は `stdout` について役立つ情報（使用された正確なアドレス、およびアクセスされた順序）のチェックサムを計算し、範囲をリストします。
- すべてのチェックサムシンボルが、すべてのチェックサムの計算から除外されていることを確認します。

  チェックサムの配置とチェックサムの範囲を比較して、それらが重複していないか確認してください。`ielftool` がチェックサムを生成した後、**［ビルド］**メッセージウィンドウに情報が見つかります。
- チェックサムの計算で同じ多項式が使用されているか確認します。
- バイト内のビットが、最下位ビットから最上位ビット、またはその逆というようにチェックサムの計算と同じ順序で処理されているか確認してください。これは、**［ビット順］**オプション（またはコマンドラインから `--checksum` オプションの `-m` パラメータ）を使用して制御します。
- CRC の小さい派生形を使用している場合は、アルゴリズムにバイト追加する必要があるかどうか確認します。

  バイトのシーケンス末尾に追加するゼロの数は、チェックサムのサイズと一致する必要があります。つまり、1 バイトのチェックサムならゼロが 1 つ、2 バイトのチェックサムであればゼロが 2 つ、4 バイトのチェックサムならゼロが 4 つ、8 バイトのチェックサムではゼロが 8 つです。
- フラッシュメモリ内の任意のブレークポイントは、フラッシュの内容を変更します。つまり、アプリケーションが計算したチェックサムは、`ielftool` が最初に計算したチェックサムともはや一致しません。2 つのチェックサムをもう一度一致させるには、すべてのフラッシュのブレークポイントと C-SPY が内部的にフラッシュに設定するブレークポイントを無効にする必要があります。スタックプラグインとデバッガオプション**［指定位置まで実行］**の両方で、C-SPY のブレークポイント設定を必要とします。可能なブレークポイントの設定元の詳細は、『*Arm 用 C-SPY® デバッグガイド*』を参照してください。
- 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。

AEABI への準拠

Arm 用 IAR ビルドツールは、Arm Limited が提唱する Embedded Application Binary Interface for Arm (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	32 ビットの Arm アーキテクチャのプロシージャ呼び出し標準
CPPABI	32 ビットの Arm アーキテクチャの C++ ABI
AAELF	32 ビットの Arm アーキテクチャの ELF
AADWARF	32 ビットの Arm アーキテクチャの DWARF
RTABI	32 ビットの Arm アーキテクチャのランタイム ABI
CLIBABI	32 ビットの Arm アーキテクチャの C ライブラリ ABI
AAPCS64	64 ビットの Arm アーキテクチャのプロシージャ呼び出し標準
VFABIA64	64 ビットの Arm アーキテクチャのベクタ関数アプリケーションバイナリインターフェース仕様
ELF64	64 ビットの Arm アーキテクチャの ELF
DWARF64	64 ビットの Arm アーキテクチャの DWARF
CPPABI64	64 ビットの Arm アーキテクチャの C++ 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。
- 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。
- 64 ビット Arm にはランタイム ABI または C ABI はありません。よって、**64 ビットモード**のコンパイラオプション `--aeabi` には影響はありません。

AEABI 準拠が有効な場合は、特定のプリプロセッサ定数は実際の定数変数になります。

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。

- 少なくても1つのIAR C/C++コンパイラfor Armでのモジュールビルドを含めます。
- `--cpu` リンカオプションで、使用する CPU。
- 完全な I/O が必要な場合、「フル」のライブラリ設定の標準ライブラリとのリンクの保証。

潜在的な非互換性は以下を含みますが、これらがすべてではありません。

- `enum` のサイズ
- `wchar_t` のサイズ
- 呼び出し規約
- 使用された定数セット

AEABI- 準拠のモジュールをリンクする際には、*ILINK を使用したリンクとアプリケーションのリンク*の章の情報も考慮してください。

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（244 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。

- 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` および `__ramfunc`
- 以下の拡張は使用しても問題ありません。`#pragma location/@`、`__arm`、`__thumb`、`__svc`、`__irq`、`__fiq`、および `__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 統合（32 ビットモード）

このメカニズムは非推奨であり、今後のバージョンでは削除される可能性があります。CMSIS をサポートした新しいプロジェクトをセットアップするには、CMSIS Manager を使用します。

`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。詳しくは、**developer.arm.com/tools-and-software/embedded/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。

注 : CMSIS は **64 ビットモード**ではサポートされていません。

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。

Armv7-M MCU の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|7}{l|b}[s|f]_math.a
```

{0|3|4|7} は Cortex-M 派生、{l|b} はバイトオーダーを選択し、[s|f] は、ライブラリが単精度 / 倍精度 FPU（Cortex-M4 および Cortex-M7 のみ）にビルドされることを示しています。

Armv8-M MCU の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_MCUl[d][fsp|fdp]_math.a
```

MCU は MCU 派生 (ARMv8MBL (M23) または ARMv8MML (M33/M35P) を選択し、l はリトルエンディアンを示し、[d] は DSP 命令のサポートを選択し、[fsp|fdp] は、ライブラリが単精度 / 倍精度 FPU にビルドされることを示しています。

注 : Armv81 (M55) MCU はこのメカニズムではサポートされていません。

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 $EW_DIR$¥arm¥CMSIS¥DSP¥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。

Arm TrustZone®

Arm TrustZone® 技術は、セキュリティのためのシステム・オン・チップ (SOC) および CPU システム全体のアプローチです。

Arm TrustZone は Armv6KZ で紹介され、Armv7-A および Armv8-A でもサポートされています。特定のツールをサポートする必要はありません。同様の性能は Cortex-M as Arm TrustZone for Armv8-M や CMSE（Cortex-M セキュリティエクステンション）で紹介されました。CMSE ではツールのサポートは必要ありません。また CMSE を対象とした開発ツールの標準インターフェースがあります。この拡張には、拡張の 2 つのモード（セキュアおよび非セキュア）が含まれます。またメモリの保護、およびメモリアクセスを有効にしたり、2 つのモデル間の移行を制御するための命令も追加します。

32 ビットモード

Armv8-M に TrustZone を追加するには、セキュアモード用に 1 つ、非セキュアモード用に 1 つの、2 つの別々のイメージをビルドします。セキュアイメージは、非セキュアイメージで使用可能な関数エントリをエクスポートできます。

IAR ビルドツールは、組み込み関数、リンカオプション、コンパイラオプション、定義済プリプロセッサシンボル、拡張キーワード、およびセクション `Veneer$$CMSE` により TrustZone をサポートします。

ヘッダファイル `arm_cmse.h` で TrustZone を使用する際に必要なデータタイプおよびユーティリティ関数を参照することができます。

関数タイプ属性 `__cmse_nonsecure_call` および `__cmse_nonsecure_entry` は、セキュアなコードから非セキュアなコードにコールするとき、使用したレジスタを消去してコードを追加します。

IAR ビルドツールは、Cortex-M セキュリティ拡張 (CMSE) を目的にした開発ツールの標準インターフェースに従っていますが、次の例外があります。

- 可変セキュアエントリ関数は使用できません。

- レジスタに合わないパラメータまたはリターン値があるセキュアエントリ関数は使用できません。
- レジスタに合わないパラメータまたはリターン値での非セキュアなコールは使用できません。
- 浮動ポイントレジスタのパラメータまたはリターン値の非セキュアなコール。
- コンパイラオプション`--cmse`にはセキュリティ拡張のArmv8-Mアーキテクチャが必要です。また ROPI（読み取り専用位置が独立）イメージまたは RWPI（読み取り書き込み位置が独立）イメージを構築しているときは、サポートされません。

ARM TrustZone の詳細については、**www.arm.com** を参照してください。

Armv8-M セキュリティ拡張 (CMSE) を使用した例

`arm¥src¥ARMv8M_Secure` ディレクトリで、Arm TrustZone および CMSE を実際に使用するプロジェクトの例を見つけることができます。

例は、2 つのプロジェクトで構成されます。

- `hello_s`: アプリケーションのセキュアな部分
- `hello_ns`: アプリケーションのセキュアではない部分

注 : セキュアでないプロジェクトをビルドする前に、セキュアなプロジェクトをビルドする必要があります。

2 つのエントリ関数が `hello_s` にあります。セキュアでないコール可能な region のセキュアなゲートウェイを介した `hello_ns` に使用可能です。

- `secure_hello`: クラシック `Hello world` 例のスタイルに挨拶を印刷します。
- `register_secure_goodbye`: コールバックは、既存のセキュア部分で印刷した文字列を返します。

リンカは自動的にセキュアゲートウェイに必要なコードを生成し、それを `Veneers$$CMSE` セクションに配置します。

サンプルプロジェクトをセットアップしビルドするには :

1. `arm¥src¥ARMv8M_Secure¥Hello_Secure.` にあるサンプルプロジェクトのワークスペース `hello_s.eww` を開きます。
2. **［プロジェクト］> ［オプション］> ［一般オプション］> ［32 ビット］**を選択して、**［TrustZone］**および **［モードを選択し］**、`hello_s` プロジェクトをセットアップしセキュアモードで実行します。セキュアモードで実行します。
3. **［プロジェクト > ［オプション］> ［一般オプション］> ［32 ビット］**を選択して、**［TrustZone］**および **［モードを選択し］**、`hello_ns` プロジェクトをセットアップしセキュアモードで実行します。非セキュアモードで実行します。

セキュアでない部分は、0x200000 で初期化ルーチンへのアドレス、セキュアでないスタックのトップ、セキュアでない main といっしょに小さいベクタを読み込む必要があります。このベクタは、セキュアな部分をセットアップし、セキュアでない部分と通信するために使用されます。この例では、nonsecure_hello.c の次のコードで完了します :

```
/* Interface towards the secure part */
#pragma location=NON_SECURE_ENTRY_TABLE
__root const non_secure_init_t init_table =
{
  __iar_data_init3,         /* initialization function */
  __section_end("CSTACK"),  /* non-secure stack */
  main_ns                   /* non-secure main */
};
```

4 セキュアプロジェクトがビルドされると、リンカは、セキュアでない部分からコールされるセキュアな部分の関数のリファレンスだけが含まれている、セキュアでない部分のインポートライブラリファイルを自動的に生成します。**[プロジェクト] > [オプション] > [リンカ] > [出力] > [TrustZone インポートライブラリ]** を使用して、このファイルを指定します。

5 セキュアなプロジェクトをビルドします。

6 追加ライブラリを指定することで、TrustZone インポートライブラリファイルを手動で、hello_ns プロジェクトに含めます : **[プロジェクト] > [オプション] > [リンカ] > [ライブラリ] > [追加ライブラリ]**。

7 セキュアでないプロジェクトをビルドします。

8 セキュアなプロジェクトは、セキュアでないプロジェクト出力ファイルをデバッガによりロードされる追加イメージとして指定する必要があります。これを行うには、**[プロジェクト] > [オプション] > [デバッガ] > [イメージ] > [追加イメージのダウンロード]** を使用します。

デバッグの例 :

1 シミュレータでデバッグするには、プロジェクトを右クリックし **[有効に設定]** を選択して、有効なプロジェクトとして hello_s プロジェクトを設定します。

2 **[プロジェクト] > [オプション] > [デバッガ] > [ドライバ]** を選択し、**[シミュレータ]** を選択します。

3 **[シミュレータ] > [メモリ構成]** を選択します。**使用範囲の設定基準ファイル**オプションが選択されていないことを確認します。

4 ［使用範囲を手動で設定］を選択して、次の新しい範囲を追加します。

アクセスタイプ	開始アドレス	終了アドレス
RAM	0x0000'0000	0x003F'FFFF
RAM	0x2000'0000	0x203F'FFFF
SFR	0x4000'0000	0x5FFF'FFFF
SFR	0xE000'0000	0xE00F'FFFF

表 25: TrustZone 例のメモリ範囲

5 **OK** をクリックして、［メモリ構成］ダイアログボックスを閉じます。

6 ［プロジェクト］>［ダウンロードしてデバッグ］を選択して、C-SPY を開始します。

7 ［表示］>［ターミナル I/O］を選択して［ターミナル I/O］ウィンドウを開きます。

8 ［デバッグ］>［移動］を選択して、実行を開始します。

9 ［ターミナル I/O］ウィンドウは、この文字を印刷します。

```
Hello from secure World!
Hello from non-secure World!
Goodbye, for now.
```

64 ビットモード

TrustZone サポートは **64 ビットモード**では自動的に行われます。

$Super$$ および $Sub$$ を使用したシンボル定義のパッチ

`$Sub$$` および `$Super$$` 特別パターンを使用すると、シンボルが外部ライブラリまたは ROM コードにある場合など、シンボルを変更できないような状況で既存のシンボル定義にパッチを適用できます。

`$Super$$` 特別パターンは、元の関数ディレクトリを呼び出すために使用した元のパッチしてない関数を識別します。

`$Sub$$` 特別パターンは、元の関数の代わりに呼び出される新しい関数を識別します。`$Sub$$` 特別パターンを使用して、元の関数の前後に処理を追加できます。

$Super$$ および $Sub$$ パターンの使用例

次の例では、$Super$$ および $Sub$$ パターンを使用して、元の関数 foo() を呼び出す前に、関数 ExtraFunc() への呼び出しを挿入する方法をお見せします。

```
extern void ExtraFunc(void);
extern void $Super$$foo(void);

/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
                  /* To avoid calling the original foo() function
                   * omit the $Super$$foo(); function call.
                   */
}
```

組み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
- コンパイラの最適化設定
- 良いコードのを生成させる方法

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、コードのサイズや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、ループのインデックスは、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。
- **32 ビットモードでは**、64 ビットデータ型 (`double` や `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び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と、コードサイズと実行速度の両面で効率が低下します。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、3 種類（16、32、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。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。16 ビットフォーマットは一部の特定の状況で役立ちます。

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。以下の例では、`a` が `float` から `double` に変換され、`double` 定数 `1.0` を加えた後、その結果が再度 `float` に変換されます。

```
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、398 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。

構造体要素のアライメント

Arm コアによっては、メモリ内のデータにアクセスする際に、データがアラインメントされている必要があります。構造体の各要素は、指定した型の要件に応じてアライメントされている必要があります。つまり、コンパイラは、

正しいアライメントを保守するため、*パディングバイト*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。

アライメントの要件については、389 ページの「*アライメント*」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メントされていない要素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、445 ページの「*pack*」を参照してください。

無名構造体と無名共用体

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。

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。I/O レジスタには 2 ビットの宣言があり、Way と Out で、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、無名共用体は _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、255 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、256 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、335 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

@ 演算子または #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。（最初のケースでは、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。）両方のオブジェクトは ROM に配置されます。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;  /* エラー、アライメントされてない
*/
```

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++ では良くない */
```

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は注意が必要です。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やアプリケーションの誤動作が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。

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 */
```

リンカは、各変数毎に正しい初期化が行われるようにします。制御または自動初期化を停止したい場合、リンカ設定ファイルの initialize と do not initialize ディレクティブを使用できます。

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);
```

レジスタのデータの配置（32 ビットモード）

32 ビットモードの場合、@ 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 R4-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。
- 高速割り込みハンドラでは、R8-R11 の変数の値は、ハンドラ外部からは保持されません。これらはバンクレジスタだからです。
- 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。

コンパイラの最適化設定

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、高速で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、444 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（1 つ、または複数）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、316 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にも役に立ちます。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、305 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。 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
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。詳細については、261 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、または間違った値を表示することがあります。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。

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。

コマンドラインオプションおよびプラグマディレクティブを使用して、各モジュールや個々の関数に対しても最適化の目標設定を選択することができます（328 ページの「*-O*」および 444 ページの「*optimize*」を参照）。小さい組み込みアプリケーションの場合、これによってコードのサイズを最小限に抑えつつ、適度な速度のパフォーマンスを実現することができます。通常は最も頻繁に実行する内部のループや割り込みハンドラなど、高速でなければならないのはアプリケーションの数箇所のみです。

アプリケーション全体を最適化レベル「高」（バランス）でコンパイルするよりも、「高」（サイズ）を全般に使用しながら、高速のアプリケーションが必要な関数についてのみ、この設定を最適化レベル「高」（速度）でオーバライドすることができます。

注 : 異なる最適化の相互の影響が予測不可能なため、ある最適化によって他の最適化が有効となる場合は、「高」（サイズ）を使用したときよりも「高」（速度）でコンパイルしたときの方が関数が小さくなることがあります。また、複数ファイルのコンパイルを使用すると（316 ページの「*--mfc*」を参照）、多くの最適化を有効にして速度とサイズの両方のパフォーマンスを向上することができます。プロジェクトに最も適したものを選べるように、異なる最適化設定を試してみることをお勧めします。

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
- 命令スケジューリング
- ベクトル化

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、318 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（326 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、その呼び出し元関数の本体に統合し、呼び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、91 ページの「*インライン関数*」を参照してください。

関数のインライン化を無効にするには、コマンドラインオプション `--no_inline` を使用します（320 ページの「*--no_inline*」を参照）。

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、317 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、324 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、317 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、323 ページの「*--no_scheduling*」を参照してください。

ベクトル化

連続するベクトル化変換は、アセンブラコードを書き込んだり、組込み関数を使用したりせずに、NEON ベクタ操作にループします。これは移植を強化します。ターゲットプロセッサに NEON 機能があり、自動ベクトル化が有効な場合にのみ、ループがベクトル化されます。自動ベクトル化は **64 ビットモード**ではサポートされていません。

ベクトル化は、、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。

注 : このオプションは、オプション --do_cross_call を使用する場合を除いて、最適化レベルが［なし］、［低］、［中］、高バランス、または高サイズの場合には動作しません。個々の関数でベクトル化を無効にするには、次のいずれかプラグマディレクティブ `optimize` や `vectorize` を使用します。詳細は 444 ページの「*optimize*」および 453 ページの「*vectorize*」を参照してください。

コマンドラインオプションの情報については、341 ページの「*--vectorize*」を参照してください。

良いコードのを生成させる方法

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。

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、呼び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び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（262 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び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、259 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 や組み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、177 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のテクニックを示します。

- スタックエリアが少ない場合は、長いコール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。

以下に例を示します。

```
int Test();     /* 宣言 */

int Test(ch, i) /* 定義 */
char ch;
int i;
{
  return i + ch;
}
```

整数型とビット反転

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`、`~0x00000080` は `0xFFFFFF7F` になります。左側では、`c1` は範囲 0–255 にある 8 ビットの符号なし文字で、`0xFFFFFF7F` と同等にな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はありません。

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（割り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*読み込み専用*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、404 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、178 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クションに配置します。

__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。

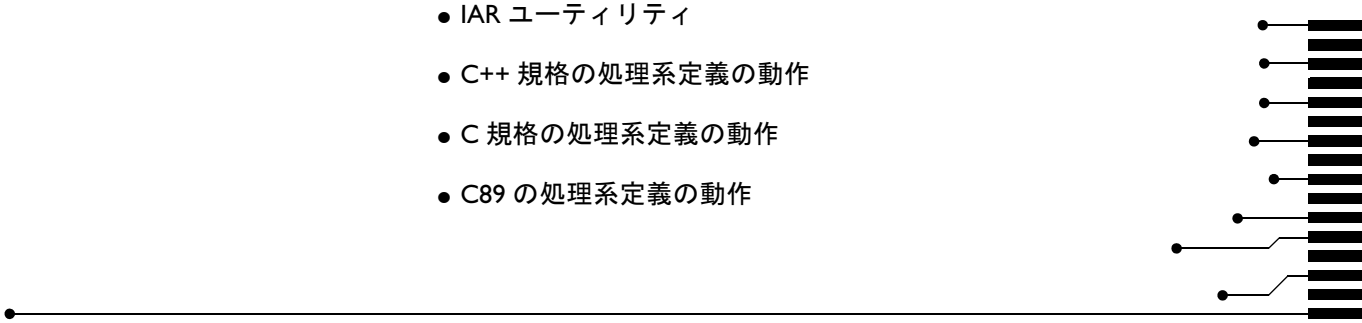
詳細については、418 ページの「*__no_init*」を参照してください。

注 : 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（307 ページの「*-e*」を参照）。詳細については、443 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*』のこの部分は、以下の章で構成されます。

- 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
- 組み込み関数
- プリプロセッサ
- C/C++ 標準ライブラリ関数
- 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
- 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

- 呼び出し構文
- インクルードファイル検索手順
- コンパイラ出力
- リンカオプション
- テキストエンコーディング
- 予約済みの識別子
- 診断

呼び出し構文

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。

コンパイラ呼び出し構文

コンパイラの呼び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` や `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。`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。

リンカ呼出し構文

リンカの呼び出し構文は次のとおりです。

```
ilinkarm [arguments]
```

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。しかし、1 つの例外があります。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リンカに渡す方法は 3 とおりあります。

- コマンドラインから直接渡す方法

  `iccarm` または `ilinkarm` コマンドのあとにコマンドラインのオプションを指定します。273 ページの「*呼び出し構文*」を参照してください。
- 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（275 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（309 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*章の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:¥my_programs¥embedded workbench 8.*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。例： 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 ディレクティブの "" 間および <> 間の文字列は、ソースファイル名としてそのまま使用されます。
- #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（312 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）（275 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。305 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。以下に例を示します。

```
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：¥ と / の両方をディレクトリの区切り文字として使用できます。

詳細については、499 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（312 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。

- プリプロセッサ出力ファイル（オプション）

  `--preprocess` オプションを使用すると、プリプロセッサ出力ファイルが生成されます。デフォルトでファイル拡張子は、`i` となります。

- 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、281 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（277 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	内部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）

リンカオプション

リンカでは、以下の出力を生成できます。

- 絶対実行可能イメージ

  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、リンカは、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、リンカオプション `-map` を使用して生成できます（370 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、281 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

  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提供します（277 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。
- 安全でないイメージ、シンボルとそのアドレスが含まれている再割り当て可能な ELF オブジェクトモジュールをビルドしているときは、重要なライブラリを使用します。リンカオプション 366 ページの「*--import_cmse_lib_out*」も参照してください。

テキストエンコーディング

IAR ツールによるテキストファイルの読み込みまたは書き込みには、さまざまなテキストのエンコードを使用できます。

- Raw

  これは、C/C++ ソースファイルの旧バージョンとの互換性モードです。7 ビット ASCII 文字だけがシンボルの名前に使用できます。その他の文字は、コメントやリテラルなどにだけ使用できます。バイトオーダーマーク (BOM) がない場合、これがデフォルトのソースファイルのエンコードです。
- システムデフォルトロケール

  ご自分の Windows OS で設定されたロケールが使用されます。
- UTF-8

  バイトオーダーマーク (BOM) の有無にかかわらず、8 ビットバイトのシーケンスとしてエンコードされた Unicode。
- UTF-16

  ビッグまたはリトルエンディアン表示を使用した 16 ビットワードのシーケンスとして Unicode がエンコードされます。これらのファイルはいつもバイトオーダーマークで始まります。

Raw 以外のエンコードには、シンボルの名前に適切な種類（英数字など）の Unicode 文字を使用できます。

IAR ツールがバイトオーダーマークのあるテキストファイルを読み込むとき、入力ファイルのエンコードがどのオプションであっても、適切な Unicode エンコードを使用します。

バイトオーダーマークのないソースファイルには、コンパイラオプション `--source_encoding` が指定してない限り、コンパイラは Raw エンコードを使用します。337 ページの「*--source_encoding*」を参照してください。

バイトオーダーマークのないソースファイルには、コンパイラオプション --source_encoding が指定してない限り、コンパイラは Raw エンコードを使用します。

バイトオーダーマークのない拡張コマンドライン（.xcl ファイル）などのその他のテキスト入力ファイルには、コンパイラオプション --utf8_text_in を指定していない限り、IAR ツールは、システムのデフォルトのロケールを使用します。指定している場合は、UTF-8 が使用されます。341 ページの「*--utf8_text_in*」を参照してください。

コンパイラリストファイルおよびプリプロセッサ出力には、メインソースファイルと同じエンコードがデフォルトで使用されます。テキスト出力を生成するその他のツールは、デフォルトでは UTF-8 エンコードを使用します。コンパイラオプション --text_out と --no_bom を使用してこれを変更できます。338 ページの「*--text_out*」および 316 ページの「*--no_bom*」を参照してください。

文字と文字列リテラル

ソースコードをコンパイルするとき、文字 (x) および文字列リテラル (xx) は次のように取り扱われます。

'x'、"xx"	タイプが設定されていない文字と文字列リテラルは、ソースファイルと同じエンコードを使用してそのままコピーされます。
u8"xx"	UTF-8 文字列リテラルの文字は、UTF-8 に変換されます。
u'x'、u"xx"	UTF-16 文字と文字列リテラルの文字は、UTF-16 に変換されます。
U'x'、U"xx"	UTF-32 文字と文字列リテラルの文字は、UTF-32 に変換されます。
L'x'、L"xx"	ワイド文字と文字列リテラルの文字は、UTF-32 に変換されます。

予約済みの識別子

いくつかの識別子は実装に使用するために予約されています。あらゆる用途ために C/C++ 標準に予約される重要な識別子の一部は以下の通りです :

- 二連続の下線 (__) を含む識別子
- アンダースコアで始まりその後に大文字が続く識別子

これに加えて、いろんな用途に IAR ツールが予約する識別子は以下です：

- 2 つのドルマーク ($$) を含む識別子
- 疑問符 (?) を含む識別子

特定の状況ではより特定の予約が影響します。詳細については、C/C++ 標準を参照してください。

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。

```
level[tag]: message filename linenumber
```

各要素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）
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

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
```

各要素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 map 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、332 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によるコードの生成に支障のあるような潜在的な問題はあるが、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（327 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、*コンパイラオプション*の章を参照してください。

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には、コンパイラ用、*プラグマディレクティブ*の章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
- コンパイラまたはリンカのバージョン番号（コンパイラまたはリンカ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
- 内部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、274 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。

`--example_option=`*value*

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。以下に例を示します。

  ```
  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。以下に例を示します。

  ```
  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。以下に例を示します。

  ```
  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	呼び出し規約を指定します
--aarch64	A64 命令セットを使用してコードを生成します
--abi	A64 命令セットを使用してコードを生成するために、データも出デルを指定します
--aeabi	AEABI 準拠のコード生成を有効にします
--align_sp_on_irq	__irq 関数に入るところで SP をアライメントす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
--cmse	CMSE 安全部ジェクトの生成を有効にする
--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
--deprecated_feature_warnings	廃止された機能の警告を有効 / 無効にする
--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

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
--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
--do_explicit_zero_opt_in_named_sections	ユーザー名セクションにおいて、明示的なゼロへの初期化をゼロ初期化とみなします
-e	言語拡張を有効化
--enable_hardware_workaround	個別のハードウェアのワークアラウンドを有効にします
--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
--f	オプションで依存関係を指定して、コマンドラインを拡張します
--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び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
-l	リストファイルを生成
--libc++	コンパイラとリンカが Libc++ ライブラリを使用するようにします。

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
--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
--max_cost_constexpr_call	constexpr 評価コストに制限を指定します
--max_depth_constexpr_call	constexpr 再帰深度に制限を指定します
--mfc	複数ファイルのコンパイルを有効化
--no_alignment_reduction	単純な thumb 関数のアライメント縮小を無効にします
--no_bom	UTF-8 出力ファイルのバイト オーダーマークを省略します
--no_call_frame_info	コールフレーム情報の出力を無効にします
--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_default_fp_contract	STDC FP_CONTRACT のデフォルト値を OFF に設定します
--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_normalize_file_macros	シンボル __FILE__ および __BASE_FILE__ のパスの標準化を無効化
--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

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_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
--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	アライメントされないアクセスを回避します
--no_uniform_attribute_syntax	IAR タイプの属性のデフォルトの構文規則を指定します
--no_unroll	ループ展開を無効化
--no_var_align	そのタイプのアライメントに基づいて、変数オブジェクトを整列します
--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
--nonportable_path_warnings	ソースヘッダファイルを開くために使用するパスが、ファイルシステムのパスと同じケースでないときに、ワーニングを生成します
-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
--pending_instantiations	任意の C++ テンプレートのインスタンス化の最大数を設定します
--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

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
--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データを、PC 相対アドレスでアクセスするコードを生成
--ropi_cb	定数データ、レジスタ R8 への相対をベースにしたアドレスへのすべてのアクセスを作成します
--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
--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
--rwpi_near	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最大 64Kbyte のメモリを指定します
--section	セクション名を変更
--section_prefix	プリフィックスをセクション名前に追加
--silent	サイレント処理を設定
--source_encoding	ソースファイルのエンコードを指定します
--stack_protection	スタック保護を有効にする
--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
--text_out	テキスト出力ファイルのエンコードを指定します
--thumb	デフォルトの関数モードを Thumb に設定します
--uniform_attribute_syntax	const および volatile のように IAR タイプ属性に同じ構文規則を指定します
--use_c++_inline	C で C++ インライン動作を使用
--use_paths_as_written	デバッグ情報の書き込みとしてパスを使用

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
--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
--utf8_text_in	テキスト入力ファイルに UTF-8 エンコードを使用します
--vectorize	NEON ベクタ命令の生成を有効にする
--version	コンパイラの出力をコンソールに送信し、終了します
--vla	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
--warn_about_incomplete_constructors	コンパイラに、すべてのメンバーを初期化しないコンストラクタについて警告させる
--warn_about_missing_field_initializers	コンパイラに、明示的なイニシャライザなしのフィールドについて警告させる
--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、関数呼び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、浮動小数点の呼び出し規約を指定するときに使用します。**64 ビットモードの場合**、このオプションは影響しません。

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。

--aarch64

構文

```
--aarch64
```

説明

このオプションを使用して、アセンブラディレクティブ `CODE` に AArch64 状態の A64 命令セットを使用したコードを生成します。

注 : このオプションの効果は、`--cpu_mode=aarch64` オプションと同じです。

関連項目

294 ページの「*--abi*」および 299 ページの「*--cpu_mode*」。

このオプションを設定するには、**［プロジェクト］> ［オプション］> ［一般オプション］> ［ターゲット］> ［例外モードを使用します］**

--abi

構文

```
--abi={ilp32|lp64}
```

パラメータ

`ilp32`	ILP32 データモデルの A64 コードを生成します。シンボル `__ILP32__` を定義します。
`lp64`	LP62 データモデルの A64 コードを生成します。シンボル `__LP64__` を定義します。

説明

このオプションを使用して、AArch64 ステータスに設定された A64 命令を使用するコードを生成するために、データモデルを指定します。

関連項目

294 ページの「*--aarch64*」および 299 ページの「*--cpu_mode*」。

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] > [ターゲット] > [例外モード]

および

[プロジェクト] > [オプション] > [一般オプション] > [64 ビット] > [データモデル]

--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。**64 ビットモードの場合**、このオプションは影響しません。

注 : 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。

注 : このオプションは、C++ ヘッダファイルと併用できません。

関連項目

242 ページの「*AEABI への準拠*」および 311 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) をアライメントさせます。**64 ビットモードの場合**、このオプションは影響しません。

これは、割り込みコードが割り込みハンドラと同じ SP を使用する、ネストされた割り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 バイトのアライメントではなく、4 バイトのアライメントしか持たない可能性があります。

関連項目

415 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（32 ビットモードの A32）に設定するときに使用します。**64 ビットモードの場合**、このオプションは影響しません。

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。

［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。

関連項目

207 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］

--cmse

構文

```
--cmse
```

説明

このオプションを使用すると、Armv8-M の TrustZone の言語の拡張が可能になります。**64 ビットモード**の場合、このオプションは影響しません。安全なイメージにリックされるオブジェクトファイルに、このオプションを使用します。このオプションは、非セキュアなコードでは使用できない命令、キーワード、タイプを使用できるようにします。

- 関数属性 _cmse_nonsecure_call および _cmse_nonsecure_entry。
- CMSE の関数には、接頭辞 cmse_ の名前があり、ヘッダファイル arm_cmse.h で定義されています。

注 : このオプションを使用するには、最初に［プロジェクト］>［オプション］>［一般オプション］>［32 ビット］>［TrustZone］オプションを選択する必要があります。

関連項目

246 ページの「*Arm TrustZone®*」および *ARMv8-M セキュリティ拡張 : 開発ツールの要件* (ARM-ECM-0359818)。

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。

--cpu

構文

```
--cpu={core|list}
```

パラメータ

core	特定のプロセッサ選択を指定します。
list	オプション --cpu でサポートされているすべての値。

説明

このオプションは、コードの生成対象のアーキテクチャまたはプロセッサ選択をする場合に使用します。

デフォルトコアは Cortex-M3 です。

32 ビットモード: --cpu オプションのサポートされているいくつかの値:

6-M	7-A	7E-M
7-M	7-R	7-S
8-A.AArch32	8-M.baseline	8-M.mainline
8-R.AArch32	Cortex-A5	Cortex-A7
Cortex-A8	Cortex-A9	Cortex-A12
Cortex-A15	Cortex-A17	Cortex-M0
Cortex-M0+	Cortex-M23	Cortex-M23.no_se (TrustZone のサポートなしのコア)
Cortex-M3	Cortex-M33	Cortex-M33.no_dsp (整数 DSP 拡張なしのコア)
Cortex-M33.fp (単精度をサポートした浮動少数単位)	Cortex-M33.no_se (TrustZone のサポートなしのコア)	Cortex-M4
Cortex-M4F	Cortex-M55	Cortex-M85
Cortex-M7	Cortex-M7.fp.dp (倍精度をサポートした浮動少数単位)	Cortex-M7.fp.sp (単精度をサポートした浮動少数単位)
Cortex-R4	Cortex-R5	Cortex-R52
Cortex-R52.no_neon	Cortex-R7	

64 ビットモード: --cpu オプションのサポートされているいくつかの値:

8-a.AArch64	Cortex-A35	Cortex-A53
Cortex-A55	Cortex-A57	Cortex-A72
Cortex-R82		

関連項目

70 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］>［プロセッサ選択］

--cpu_mode

構文

```
--cpu_mode={arm|a|thumb|t|a64}
```

パラメータ

`arm`, `a`（デフォルト）	32 ビットモードで A32 命令セットを選択します。
`thumb`, `t`	32 ビットモードで T32 または T 命令セットを選択します。
`a64`	64 ビットモードで A62 命令セットを選択します。

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。

関連項目

294 ページの「*--aarch64*」。

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

--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

217 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++］

-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][l|w][b]] {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
l	UTF-8 の代わりにロケールエンコードを使用
w	UTF-8 の代わりにリトルエンディアン UTF-16 エンコードを使用
b	UTF-8 出力にバイトオーダーマーク (BOM) を使用
+	-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

287 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。以下に例を示します。

```
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。以下に例を示します。

```
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。

--deprecated_feature_warnings

構文

```
--deprecated_feature_warnings=[+|-]feature[,[+|-]feature,...]
```

パラメータ

feature	feature は attribute_syntax または segment_pragmas です。

説明

このオプションを使用して、廃止予定の機能の使用に関する警告を有効または無効にします。廃止予定の機能 :

- attribute_syntax

  339 ページの「*--uniform_attribute_syntax*」、326 ページの「*--no_uniform_attribute_syntax*」、408 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。

- segment_pragmas

  プラグマディレクティブ dataseg、constseg、および memory を参照します。#pragma location および #pragma default_variable_attributes ディレクティブを代わりに使用します。

廃止予定の機能は、今後の IAR C/C++ コンパイラでは削除されるので、今後のために、ソースコードでそれらの使用を削除することが推奨されます。これを行うには、廃棄予定の機能の警告を有効にします。各警告に対して、コードを修正し、廃棄予定の機能が今後使用されないようにします。

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。

--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回使用できます。

[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回使用できます。

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、診断メッセージの行折り返し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回使用できます。

[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回使用できます。

[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

287 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。

通常このオプションは、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。

--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、割り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

316 ページの「*--mfc*」および 259 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、以下の表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。

287 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*」も参照してください。

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。

注 : このオプションは、コンパイラオプション `--libc++` が指定されている場合のみ、使用できます。

ライブラリオブジェクトファイルはディレクトリ `arm¥lib` にあり、ライブラリ設定ファイルは `arm¥inc¥c` 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、145 ページの「*ビルド済ランタイムライブラリ*」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ことができます。詳細については、142 ページの「*独自のランタイムライブラリのカスタマイズおよびビルド*」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]

--do_explicit_zero_opt_in_named_sections

構文

```
--do_explicit_zero_opt_in_named_sections
```

説明

デフォルトでは、コンパイラは静的な変数の初期化を明示的あるいは暗示的にゼロに初期化します。ただしユーザー名セクションに配置されている変数は例外です。これらの変数に対する明示的ゼロの初期化は、初期化のコピーとしてみなされ、それはゼロ以外に静的に初期化された変数と同じ方法です。

このオプションを使用することで、ユーザー名セクションの変数の例外を無効にし、明示的なゼロへの初期化をゼロの初期化とみなし、初期化のコピーとはみなしません。

例

```
int var1;                // Implicit zero init -> zero inited
int var2 = 0;            // Explicit zero init -> zero inited
int var3 = 7;            // Not zero init      -> copy inited
int var4 @ "MYDATA";     // Implicit zero init -> zero inited
int var5 @ "MYDATA" = 0; // Explicit zero init -> copy inited
                         // If option specified, then zero inited
int var6 @ "MYDATA" = 7; // Not zero init      -> copy inited
```

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。

-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。拡張キーワードや無名構造体 / 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

209 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。

--enable_hardware_workaround

構文

```
--enable_hardware_workaround=waid[,waid...]
```

パラメータ

waid	有効にするワークアラウンドの 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ワークアラウンド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_restrict

構文

```
--enable_restrict
```

説明

C89 と C++ において、標準 C キーワード `restrict` を有効にします。デフォルトでは、`restrict` は標準 C で認識され、`__restrict` は常に認識されます。

このオプションは最適化の際の分析の精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={big|b|little|l}
```

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。このバイトオーダーは **64 ビットモード**では使用できません。
`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。**64 ビットモードの場合**、このバイトオーダーのみが指定できます。

関連項目

390 ページの「*バイトオーダ (32 ビットモードのみ)*」。

[プロジェクト] > [オプション] > [一般オプション] > [32 ビット] > [バイトオーダー]

--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。特定の `enum` 型を使用するソースファイルをコンパイルするときにこのオプションを使用すると、`enum` 型を使用するそれぞれのソースファイルはこのオプションを使用してコンパイルする必要があります。

注 : このオプションでは、整数型よりサイズの大きい enum 型が使用される可能性があるということは考慮されません。

関連項目

392 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

287 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。

コンパイラオプション --dependencies を使用する場合、--f を使用して指定したコマンドライン拡張 (XCL) ファイルは依存関係を生成しますが、-f を使用して指定したものは依存関係を生成しません。

関連項目

301 ページの「*--dependencies*」および 310 ページの「*-f*」。

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。

--f

構文

```
--f filename
```

パラメータ

287 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。

コンパイラオプション `--dependencies` を使用する場合、`--f` を使用して指定したコマンドライン拡張 (XCL) ファイルは依存関係を生成しますが、`-f` を使用して指定したものは依存関係を生成しません。

関連項目

301 ページの「*--dependencies*」および 309 ページの「*-f*」。

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={name|list|none}
```

パラメータ

name	ターゲット FPU アーキテクチャ。
`list`	`--fpu` オプションのサポートされているすべての値の一覧。
`none`（デフォルト）	FPU なし。

説明

このオプションは、浮動小数点単位 (FPU) 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。FPU 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。

ターゲット FPU の名前は、次の方法のいずれかで作成されます。

- なし : FPU なし（デフォルト）
- *fp-architecture*: 指定したアーキテクチャのベースバリアント
- *fp-architecture*`-SP`: 単精度バリアント
- *fp-architecture*`_D16`: 16 `D` レジスタのあるバリアント
- *fp_architecture*`_Fp16`: ハーフ精度拡張の変数

利用可能な組み合わせ :

- `{VFPv2|VFPv3|VFPv4|VFPv5}`
- `{VFPv3|VFPv4|VFPv5}_D16`
- `{VFPv4|VFPv5}-SP`
- `VFPv3_Fp16`
- `VFPv3_D16_Fp16`

注 : 64 ビットモードの場合、このオプションは影響しません。AArch64 アーキテクチャは常に、ハードウェア浮動小数点処理をサポートします。

関連項目

71 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］> ［32 ビット］> ［浮動小数点設定］

--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

171 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

275 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］

-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、コールフレーム情報、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、コールフレーム情報、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。

287 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。

注 : 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］

--libc++

構文

```
--libc++
```

説明

このオプションを使用して、コンパイラが Libc++ システムヘッダを使用するようにして、リンカが、C++17 をサポートする Libc++ ライブラリを使用するようにします。フルライブラリ設定が使用され、ヘッダファイル DLib_Config_Full.h が参照されます。

注 : このオプションは、コンパイラオプション --dlib_config とともに使用することはできません。

関連項目

217 ページの「*概要— 標準 C++*」。

［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］>［Libc++］

--lock_regs

構文

```
--lock_regs=register
```

パラメータ

registers

ロックするレジスタ名とレジスタ区間のコンマ区切りのリスト。

32 ビットモード：範囲 `R4-R11` のレジスタ。

64 ビットモードの場合：範囲 `X9-X15` および `X18-X29` のレジスタ。

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 weak 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

424 ページの「*__weak*」。

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。

--max_cost_constexpr_call

構文

```
--max_cost_constexpr_call=limit
```

パラメータ

limit	コールとループの繰り返し数。デフォルトは、2000000 です。

説明

このオプションを使用して、トップレベルの constexpr コール（関数またはコンストラクタ）を織り込むために、*cost* の上限を指定します。コストは、トップレベルコールの割り込み中、割り込みコール数と実行されたループ割り込み数の組み合わせです。

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。

--max_depth_constexpr_call

構文

```
--max_depth_constexpr_call=limit
```

パラメータ

limit	再帰深度。デフォルトは、1000 です。

説明

このオプションを使用して、トップレベルの constexpr コール（関数またはコンストラクタ）を織り込むために、最大再帰深度を指定します。

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。

--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

305 ページの「*--discard_unused_publics*」、329 ページの「*--output*、*-o*」、259 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_bom

構文

```
--no_bom
```

説明

このオプションを使用して、UTF-8 出力ファイルを生成するときに、バイトオーダーマーク (BOM) を省略します。

関連項目

338 ページの「*--text_out*」、279 ページの「*テキストエンコーディング*」。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［エンコード］>［テキスト出力ファイルのエンコード］

--no_call_frame_info

構文

```
--no_call_frame_info
```

説明

通常、コンパイラは出力にコールフレーム情報を生成し、デバッガがデバッグ情報なしのモジュールからのコードでもコールスタックを表示できるようにします。コールフレーム情報の生成を無効にするには、このオプションを使用します。

関連項目

202 ページの「*呼出しフレーム情報*」。

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、最適化レベルが［中］以上の場合にのみ有効です。

関連項目

264 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

263 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メントに基づいてアライ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メント 1 になります。文字列リテラルは、`const char` 型の要素を持つ配列であり、この型のアライメントが 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

389 ページの「*アライメント*」。

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。

関連項目

262 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_default_fp_contract

構文

```
--no_default_fp_contract
```

説明

このプラグマディレクティブ `STDC FP_CONTRACT` は、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。このプラグマディレクティブのデフォルトは `ON`（縮約を許可）です。このオプションを使用して、デフォルトを `OFF`（縮約を不許可）に変更します。

関連項目

451 ページの「*STDC FP_CONTRACT*」。

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_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

219 ページの「*例外処理*」および 511 ページの「*__EXCEPTIONS__*」。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++]

および選択解除

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++ オプション] > [例外の許可]

--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

119 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

91 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。**64 ビットモードの場合**、このオプションは影響しません。

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。

注 : M アーキテクチャプロファイル (Cortex-M コア) の場合、このオプションはリトルエンディアンのバイトオーダーを使用しているときのみ利用できます。

関連項目

374 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

389 ページの「*アライメント*」。

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び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び出し以外のことが影響してきます。

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_normalize_file_macros

構文

```
--no_normalize_file_macros
```

説明

通常、.. および . コンポーネントの不必要な使用は、定義済のプリプロセッサシンボル `__FILE__` および `__BASE_FILE__` によって返されたパスが破損することになることがあるようです。このオプションを使用して、回避することができます。

例

パス `"D:¥foo¥..¥bar¥baz.c"` は、このオプションが使用されていない限り、シンボル `__FILE__` および `__BASE_FILE__` によって `"D:¥bar¥baz.c"` として返されます。

関連項目

500 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_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

500 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

217 ページの「*C++ の使用*」および 514 ページの「*__RTTI__*」。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++]

および選択解除

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語] 1> [C++ オプション] > [RTTI の許可]

--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

333 ページの「*--ropi*」および 334 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

264 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増加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

261 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

121 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

305 ページの「*--dlib_config*」、338 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

263 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使用可能です。

Armv7-M、Armv8-A、および Armv8-M のアーキテクチャには、ハードウェアは、アライメントしていないアクセスがソフトウェアによって制御されることをサポートします。これらのアーキテクチャには、ライブラリルーチンの変形型があります。アライメントされていないアクセスがハードウェア（接頭辞が `__iar_unaligned_` のシンボル）でサポートされているときは、この変数は速いです。入力モジュールのいずれかが、アライメントされていないアクセスを許可しない場合、IAR リンカは、これらの変形型を使用しません。

関連項目

389 ページの「*アライメント*」。

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_uniform_attribute_syntax

構文

```
--no_uniform_attribute_syntax
```

説明

このオプションを使用して、タイプが指定される前に、デフォルトの構文規則を指定した IAR タイプ属性に適用します。

関連項目

339 ページの「*--uniform_attribute_syntax*」および 408 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*」。

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

262 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_var_align

構文

```
--no_var_align
```

説明

デフォルトでは、サイズが 4 バイト以上の変数オブジェクトに対してアライメント 4 がコンパイラで使用されます。このオプションは、コンパイラで変数オブジェクトをそれらの型のアライメントに基づいてアライメントする場合に使用します。

たとえば、`char` 配列のアライメントは 1 になります。これは型 `char` の要素のアライメントが 1 であるためです。このオプションを使用すると RAM 空間を節約できますが、処理速度が低下することがあります。

関連項目

389 ページの「*アライメント*」および 318 ページの「*--no_const_align*」。

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_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、診断メッセージの行折り返し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。

--nonportable_path_warnings

構文

```
--nonportable_path_warnings
```

説明

このオプションを使用すると、ソースファイルまたはヘッダファイルを開くために使用したパスの文字が、ファイルシステムのパスと比べて、大文字ではなく小文字、またはその逆のときに、コンパイラがワーニングを生成できます。

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）

* レベル「低」で実行されたすべての最適化は、「なし」でも実行されます。ちがいは、レベル「なし」では、すべての非静的変数がそのスコープ全体で有効なことです。

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低が使用されます。-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。

高い最適化レベルで、速度やサイズ（-Ohs や -Ohz）を選ぶとき、コンパイラは、リクエストされた最適化の目標を示した AEABI 属性を出力します。この情報は、DLIB ライブラリ関数のより小さくて速い変数を選択するために、リンカによって使用されます。

- モジュールで参照している関数が、-Ohs でコンパイルされていて、DLIB ライブラリに早い変形型が存在する場合、それが使用されます。
- すべてのモジュールで参照している関数が、-Ohz でコンパイルされていて、DLIB ライブラリは、小さい変形型が存在する場合、それが使用されます。

例えば、Cortex-M0 に -Ohz を使用すると、整数除算にはより小さい AEABI ライブラリルーチンを使用することになります。

関連項目

258 ページの「*コンパイラの最適化設定*」。

［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

コンパイラにおいて、通常はエラー出力ストリーム (stdout) に転送されるメッセージも標準出力ストリーム (stderr) 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

287 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。

--pending_instantiations

構文

```
--pending_instantiations number
```

パラメータ

number	上限を指定する整数で、64 がデフォルト値です。0 を使用すると上限がなくなります。

説明

このオプションを使用して、任意の時点でインスタンス化できる任意の C++ テンプレートのインスタンス化の最大数を指定します。これは、再帰的なインスタンス化を検出するために使用します。

[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]

--predef_macros

構文

```
--predef_macros {filename|directory}
```

パラメータ

287 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、コンパイラまたはコマンドラインで定義したすべてのシンボルをリストします。（ソースコードで定義したシンボルはリストされません。）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。

注 : このオプションは、コマンドラインでソースファイルを指定する必要があります。

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

287 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][s]] {filename|directory}
```

パラメータ

c	コメントを含む
n	プリプロセスのみ
s	#line ディレクティブを無効化

287 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。

注 : 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

282 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び出し
- 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び出し

［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。**64 ビットモードの場合**、このオプションは影響しません。

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

リンカ設定ファイルの `movable` ブロックを使用することを検討します。
543 ページの「*define block ディレクティブ*」を参照してください。

関連項目

322 ページの「*--no_rw_dynamic_init*」、500 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)］

--ropi_cb

構文

```
--ropi_cb
```

説明

このオプションを使用して、定数データ、レジスタ R8 への相対をベースにしたアドレスへのすべてのアクセスを作成します。**64 ビットモードの場合**、このオプションは影響しません。

`--ropi_cb` と `--ropi` をいっしょに使用して、PC を使用する代わりに、読み取り専用データのベースデータとして、Arm コアレジスタ R8 を使用する

ROPI の派生をアクティベートします。たとえば、実行専用メモリから実行されるコードで ROPI を使用する場合に便利です。これはコンパイルして `--no_literal_pool` でリンクすると有効になります。

注 :

- `--ropi_cb` の使用は、AEABI に準拠していません。
- レジスタ `R8` のセットアップは提供されません。これはお使いのアプリケーションで行う必要があります。

このオプションを設定するには、**[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション]** を選択します。

--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。**64 ビットモードの場合**、このオプションは影響しません。

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。

リンカ設定ファイルの `movable` ブロックを使用することを検討します。
543 ページの「*define block ディレクティブ*」を参照してください。

関連項目

322 ページの「*--no_rw_dynamic_init*」、500 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)]

--rwpi_near

構文

```
--rwpi_near
```

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。**64 ビットモードの場合**、このオプションは影響しません。

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。
- 最大 64 Kbytes の範囲のメモリの読み取り / 書き込みが可能です。

関連項目

322 ページの「*--no_rw_dynamic_init*」および 500 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。

--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。

注 : 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ります。

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

254 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]

--section_prefix

構文

```
--section_prefix=prefix
```

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。このオプションを使用して、`@` 表記または

#pragma location ディレクティブを使用して明示的に名前が付けられていないセクション名を変更します。

このオプションは、セクションタイプのデフォルト名の前にプリフィックスを付けてセクション名を作成します。これで、異なる目的に異なるセクションセレクタを使用することができるようになります。下の例ではプリフィックスと一致するために tcm.* を使用できます。例えば、*.bss はゼロ初期化データのセクションと一致します。

注 : 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ります。

例

--section_prefix=tcm の場所を指定 :

- テキストではなく tcm.text のコード
- .rodata でなはく tcm.rodata の読み取り専用データ
- .bss でなはく tcm.bss のゼロ初期化データ
- .data ではなく tcm.data の別のゼロ初期化データ

関連項目

254 ページの「*データと関数のメモリ配置制御*」

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。

--source_encoding

構文

```
--source_encoding {locale|utf8}
```

パラメータ

`locale`	デフォルトのソースエンコードは、システムのロケールのエンコードです。
`utf8`	デフォルトのソースエンコードは、UTF-8 エンコードです。

説明

バイト オーダーマーク (BOM) がないソースファイルを読み込むとき、このオプションを使用してエンコードを指定します。このオプションが指定されていなくて、ソースファイルに BOM がない場合、Raw エンコードが使用されます。

関連項目

279 ページの「*テキストエンコーディング*」。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [エンコード] > [デフォルトソースファイルのエンコード]

--stack_protection

構文

```
--stack_protection
```

説明

このオプションを使用して、スタック保護が必要と考えらる関数のスタック保護を有効にします。

関連項目

93 ページの「*スタック保護*」。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [コード] > [スタック保護]

--strict

構文

```
--strict
```

説明

デフォルトでは、コンパイラで標準の C/C++ に緩和して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

209 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

システムインクルードファイルのパスの指定については、287 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

305 ページの「*--dlib_config*」、324 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。

--text_out

構文

```
--text_out {utf8|utf16le|utf16be|locale}
```

パラメータ

`utf8`	UTF-8 エンコードを使用
`utf16le`	UTF-16 リトルエンディアンエンコードを使用
`utf16be`	UTF-16 ビッグエンディアンエンコードを使用
`locale`	システムのロケールエンコードを使用

説明

このオプションを使用して、テキスト出力ファイルを生成するときに使用するのエンコードを指定します。

コンパイラリストファイルのデフォルトは、メインソースファイルと同じエンコードが使用されます。すべてのその他のテキストファイルのデフォルトは、バイトオーダーマーク (BOM) のある UTF-8 です。

BOM なしの UTF-8 エンコードでテキストを出力する場合、オプション `--no_bom` を使用します。

関連項目

316 ページの「*--no_bom*」および 279 ページの「*テキストエンコーディング*」。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［エンコード］>［テキスト出力ファイルのエンコード］

--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（32 ビットモードの T32 or T) に設定するときに使用します。

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。

［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］

--uniform_attribute_syntax

構文

```
--uniform_attribute_syntax
```

説明

デフォルトでは、タイプ指定の前に指定された IAR タイプ属性がオブジェクトや typedef 自体に適用され、`const` や `volatile` がするようにタイプ指定には適用されません。このオプションを指定すると、IAR タイプ属性が、`const` や `volatile` と同じ構文規則に従います。

デフォルトでは、IAR タイプ属性の均一の属性構文が使用*されません*。

関連項目

326 ページの「*--no_uniform_attribute_syntax*」および 408 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*」。

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。

--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

91 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］>［C++ インライン動作］

--use_paths_as_written

構文

```
--use_paths_as_written
```

説明

デフォルトでは、コンパイラは、最初にそのように指定されてない場合でもデバッグ情報のすべてのパスが絶対パスであることを確実にします。

このオプションを使用すると、最初に相対パスで指定されたパスは、デバッグ情報で相対パスになります。

このオプションで影響のあるパス：

- ソースファイルへのパス
- 相対と指定されたインクルードパスを使用して見つかるヘッダファイルへのパス

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。

--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。

--utf8_text_in

構文

```
--utf8_text_in
```

説明

このオプションを使用して、コンパイラは、バイト オーダーマーク (BOM) のないテキスト入力ファイルを読み込むとき、UTF-8 エンコードを使用できます。

注 : このオプションはソースファイルには適用されません。

関連項目

279 ページの「*テキストエンコーディング*」。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [エンコード] > [デフォルト出力ファイルのエンコード]

--vectorize

構文

```
--vectorize
```

説明

このオプションを使用して、**32 ビットモードで** 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。自動ベクトル化は **64 ビットモード**ではサポートされていません。

[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]

--version

構文

```
--version
```

説明

このオプションを使用して、バージョン情報をコンソールに送信してから終了するようツールに指示します。

このオプションは、IDE では使用できません。

--vla

構文

```
--vla
```

説明

このオプションを使用して、C コード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

207 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。

--warn_about_incomplete_constructors

構文

```
--warn_about_incomplete_constructors
```

説明

このオプションを使用して、イニシャライザが各データメンバーにイニシャライザを提供しない場合にコンパイラにワーニングを作成させます。

このオプションは、IDE では使用できません。

--warn_about_missing_field_initializers

構文

```
--warn_about_missing_field_initializers
```

説明

構造体のイニシャライザが、構造体のすべてのフィールドに明示的なイニシャライザを提供しない場合に、コンパイラの警告を作成するには、このオプションを使用します。

汎用ゼロイニシャライザ { 0 }、または C++ の空のイニシャライザ {} にはワーニングを出力しません。

C では、1 つ以上の専用のイニシャライザを使用するイニシャライザはチェックされません。

標準 C++17 では、専用のイニシャライザは使用できません。言語拡張が有効（-e または #pragma language を使用）な時は、サポートされますが、C++20 のように、専用のイニシャライザがフィールド順オーダーにある場合のみです。この場合、構造体は不明なイニシャライザについてチェックされます。

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

304 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、285 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
`--abi`	リンクするデータモデルを指定します
`--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

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
--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
--default_to_complex_ranges	complex ranges を initialize ディレクティブのデフォルトのデコンプレッサにします
--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
--do_segment_pad	n バイトアライメントになるように、各 ELF セグメントにダミーデータを追加します
--enable_hardware_workaround	特定のハードウェアのワークアラウンドを有効化
--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
--entry_list_in_address_order	アドレス順でソートした追加のエントリリストをマップファイルに追加します
--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び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
--f	オプションで依存関係を指定して、コマンドラインを拡張します
--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

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
--fpu	アプリケーションをリンクする FPU を選択します
--ignore_uninstrumented_pointers	境界が設定されていないメモリへのポインタ経由でのアクセス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
--import_cmse_lib_in	非セキュアイメージをビルドするために、インポートライブラリの以前のバージョンを読み取ります
--import_cmse_lib_out	非セキュアイメージをビルドするために、インポートライブラリを生成します
--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
-L	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。--search のエイリアス
--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
--manual_dynamic_initialization	システム起動時の自動初期化を無効にします
--map	マップファイルを生成します
--merge_duplicate_sections	同等のリードオンリーのセクションをマージ
--no_bom	UTF-8 出力ファイルのバイト オーダーマークを省略します
--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_entry	エントリポイントをゼロに設定します
--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_free_heap	最も小さくなりうるヒープの実装を使用します
--no_inline	小さい関数インラインから関数を除外します

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
--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_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折り返しません
-o	--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（ielftool により計算されるチェックサムなど）
--preconfig	リンカ構成ファイルを読み込む前に指定したファイルを読み込みます
--printf_multibytes	printf フォーマッタがマルチバイトをサポートするようにします
--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
--scanf_multibytes	scanf フォーマッタがマルチバイトをサポートするようにします
--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
--stack_usage_control	スタック使用解析制御ファイルを指定

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
--strip	デバッグ情報を実行可能イメージから削除します
--text_out	テキスト出力ファイルのエンコードを指定します
--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
--timezone_lib	ライブラリでタイムゾーンと夏時間機能を有効にします
--treat_rvct_modules_as_softfp	RVCT が生成したすべてのモジュールを標準の（VFP ではない）呼び出し規約を使用していると扱います
--use_full_std_template_names	標準 C++ テンプレートのフルネームを有効にします
--use_optimized_variants	DLIB ライブラリ関数の最適化された派生型の使用を制御します
--utf8_text_in	テキスト入力ファイルに UTF-8 エンコードを使用します
--version	バージョン情報をコンソールに送信し、終了します
--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。

--abi

構文

```
--abi {ilp32|lp64}
```

パラメータ

`ilp32`	ILP32 データモデルの A64 コードをリンクします
`lp64`	LP64 データモデルの A64 コードをリンクします

説明

A64 命令セットをリンクするとき、リンカはオブジェクトファイルの属性からデータモデルを検出します。このオプションを使用して目的のデータモデルを指定した場合、オブジェクトファイルが想定した属性を持っていないとリンカはエラーを発行します。

関連項目

294 ページの「*--aarch64*」および 299 ページの「*--cpu_mode*」。

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] > [ターゲット] > [例外モード]

および

[プロジェクト] > [オプション] > [一般オプション] > [64 ビット] > [データモデル]

--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

230 ページの「*ヒープメモリハンドラ*」。

[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 2] > [ヒープ選択]

--basic_heap

構文

```
--basic_heap
```

説明

このオプションにより、ベーシックヒープハンドラを使用します。

関連項目

230 ページの「*ヒープメモリハンドラ*」。

［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 2］>［ヒープ選択］

--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。**64 ビットモードの場合**、このオプションは影響しません。

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

71 ページの「*バイトオーダ*」、390 ページの「*バイトオーダ (32 ビットモードのみ)*」、351 ページの「*--BE32*」、308 ページの「*--endian*」。

［プロジェクト］>［オプション］>［一般オプション］>［32 ビット］>［バイトオーダー］

--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。**64 ビットモードの場合**、このオプションは影響しません。

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

71 ページの「*バイトオーダ*」、390 ページの「*バイトオーダ (32 ビットモードのみ)*」、351 ページの「*--BE8*」、308 ページの「*--endian*」。

［プロジェクト］>［オプション］>［一般オプション］>［32 ビット］>［バイトオーダー］

--call_graph

構文

```
--call_graph {filename|directory}
```

パラメータ

287 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

105 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］>［コールグラフ出力 (XML)］

--config

構文

```
--config filename
```

パラメータ

287 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

355 ページの「*--define_symbol*」および 125 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

569 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び出されます。標準ライブラリを使用していない場合など、初期化を行うために別のルーチンが必要な場合にはこのオプションを使用します。

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|list
```

パラメータ

core	特定のプロセッサ選択を指定します。
`list`	オプション `--cpu` でサポートされているすべての値。

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

297 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] > [プロセッサ選択]

--default_to_complex_ranges

構文

```
--default_to_complex_ranges
```

説明

リンカ設定ファイル内の initialize ディレクティブで simple ranges または complex ranges が指定されていない場合、リンカは関連するセクション配置ディレクティブが単一範囲領域を使用する場合、simple ranges を使用します。

リンカがデフォルトで常に complex ranges を使用するようにするには、このオプションを使用します。これは simple ranges と complex ranges が導入される前のリンカの動作でした。

関連項目

550 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] > [リンカ] > [追加オプション]** を使用します。

--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。

注 : 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

353 ページの「*--config_def*」および 125 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|n][s][l|w][b]] {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
l	UTF-8 ではなくロケールエンコードを使用
w	UTF-8 ではなくリトルエンディアン UTF-16 を使用
b	バイトオーダーマーク (BOM) を UTF-8 出力ファイルに使用
+	-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。

287 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。以下に例を示します。

```
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。以下に例を示します。

```
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回使用できます。

[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 `Go109` 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。

注 : すべての診断メッセージが分類される訳ではありません。このオプションは、1 つのコマンドラインで複数回使用できます。

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 `Pa180` など）

説明

このオプションは、診断メッセージの行折り返し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回使用できます。

注 : すべての診断メッセージが分類される訳ではありません。

[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 `Li004` 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、問題のあるエラーや削除を示します。ただし、リンク処理が完了する前にする前にリンカが停止することはありません。このオプションは、1 つのコマンドラインで複数回使用できます。

注 : すべての診断メッセージが分類される訳ではありません。

[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

287 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。

--do_segment_pad

構文

```
--do_segment_pad
```

説明

このオプションを使用して、コンテンツを持つ実行可能ファイル内の各 ELF セグメントを拡張し、（可能であれば）4 バイト長の偶数倍にします。いくつかのランタイムライブラリルーチンは、4 バイト単位でメモリにアクセスでき、ELF セグメントの最後に正しいデータオブジェクトが配置される場合、セグメントの厳格な境界の外側のメモリにアクセスします。これが問題となる環境で実行している場合、これが問題とならないように適切に ELF セグメントを拡張するためにこのオプションを使用できます。

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	有効にするワークアラウンドの ID 番号。使用可能なワークアラウンドのリストは、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ワークアラウンド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際、スタック使用量のセクション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

105 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）。

関連項目

372 ページの「*--no_entry*」。

[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]

--entry_list_in_address_order

構文

```
--entry_list_in_address_order
```

説明

このオプションは、マップファイルに追加のエントリリストを生成する場合に使用します。このエントリリストはアドレス順にソートされます。

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。

--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。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` が呼び出されるようにします。

説明

このオプションを使用して、正しいコール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 関数が正しいコール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コールフレーム情報を生成する必要があります。

関連項目

217 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

287 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び出された後、`main` が呼び出される前に呼び出されます。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。

注 : 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。この理由から、アセンブラ言語で書き込むことが最も安全です。

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] > [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

287 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。

関連項目

363 ページの「*-f*」。

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

287 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。

リンカオプション `--dependencies` を使用する場合、`--f` を使用して指定したコマンドライン拡張 (XCL) ファイルは依存関係を生成しますが、`-f` を使用して指定したものは依存関係を生成しません。

関連項目

301 ページの「*--dependencies*」および 309 ページの「*-f*」。

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することで、リンカがヒューリスティックに例外を使用しないことを認識していても、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。コードのその他の部分にそうした `throw` 式がなければ、リンカは `operator new`、`dynamic_cast`、`typeid` を用意して、失敗したときに例外をスローするのではなく `abort` を呼び出します。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

217 ページの「*C++ の使用*」。

[プロジェクト] > **[オプション]** > **[リンカ]** > **[アドバンスト]** > **[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。

--fpu

構文

```
--fpu={name|none}
```

パラメータ

name	ターゲット FPU アーキテクチャ。
none	FPU なし。

説明

デフォルトでは、リンカは、オブジェクトファイルの属性と互換性のある FPU の自分のアプリケーションをリンクします。このオプションを使用して、自分のアプリケーションをリンクする FPU を明示的に選択します。

注 : 64 ビットモードの場合、このオプションは影響しません。

関連項目

310 ページの「*--fpu*」。

[プロジェクト] > [オプション] > [一般オプション] > [32 ビット] > [FPU]

--image_input

構文

```
--image_input filename [,symbol,[section[,alignment]]]
```

パラメータ

filename	リンクするロウイメージを含むピュアバイナリファイル。287 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。
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
```

ピ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

367 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]

--import_cmse_lib_in

構文

```
--import_cmse_lib_in filename
```

パラメータ

287 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

以前のバージョンのインポートライブラリを読み込みとインポートライブラリで与えられたものと同じアドレスにゲートウェイベニアを作成します。このオプションを使用して、提供されたインポートライブラリに存在するそれぞれのエントリ関数が、出力インポートライブラリの同じアドレスに配置される、セキュアイメージを作成します。

注 : 64 ビットモードの場合、このオプションは影響しません。

関連項目

297 ページの「*--cmse*」および 366 ページの「*--import_cmse_lib_out*」。

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。

--import_cmse_lib_out

構文

```
--import_cmse_lib_out filename|directory
```

パラメータ

287 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

セキュアイメージをビルドする際に、関連の非セキュアイメージで使用するためのインポートライブラリを自動的にを作成するときに、このオプションを使用します。インポートライブラリは、シンボルテーブルだけが含まれている、再割り当て可能な ELF オブジェクトモジュールだけで成り立っています。各シンボルは、セクション `Veneer$$CMSE` のエントリ用のセキュアゲートウェイの絶対アドレスを指定します。

注 : 64 ビットモードの場合、このオプションは影響しません。

関連項目 297 ページの「*--cmse*」および 366 ページの「*--import_cmse_lib_in*」。

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。

--inline

構文 `--inline`

説明 一部のルーチンは小さいため、ルーチンを呼び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び出しをルーチン本体と置き換えるようにします。

関連項目 131 ページの「*小さいルーチンのインライン化*」。

[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[,symbol1,...]`

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。

crt_routine_selection	利用可能な定義、要件、プロセスの結果はどちらになるかなど、ランタイムルーチンの選択プロセスの詳細をリストします。
demangle	ログ出力にある C/C++ シンボルにマングル化した名まえの代わりにデマングル化した名前を使用します。例えば _Z1hic の代わりに void h(int, char) を使用します。
fragment_info	すべてのフラグメントを数字でリストします。情報には、名前、セクション番号、ファイルなどの関連セクションおよびフラグメントのサイズが含まれています。
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。
inlining	インラインされた関数、およびインラインされたセクション（名前、セクション番号、およびファイル）をリストします。リンカのインライン化は --inline リンカオプションで有効にできます。367 ページの「*--inline*」を参照してください。
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。
merging	マージされたセクション（名まえ、セクション番号、ファイル）、およびこの結果になるシンボルリダイレクトをリストします。セクションのマージは --merge_duplicate_sections リンカオプションで有効にする必要があります。371 ページの「*--merge_duplicate_sections*」を参照してください。
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。
veneers	ベニアの作成および使用状況の統計をリストします。

`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。

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　369 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　287 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　367 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。

--manual_dynamic_initialization

構文

```
--manual_dynamic_initialization
```

説明

通常、動的初期化（静的記憶寿命の C++ オブジェクトの通常の初期化）は、アプリケーションのセットアップ中に自動的に実行されます。`--manual_dynamic_initialization` を使用する場合は、初期化を完了するために後で `__iar_dynamic_initialization` を呼び出す必要があります。

関数 `__iar_dynamic_initialization` は、ヘッダファイル `iar_dynamic_init.h` で宣言されます。

スレッドされたアプリケーションでは、`--manual_dynamic_initialization` は、メインスレッドのスレッドローカル変数の自動初期化を無効にします。その場合、スレッドローカル変数を使用する前、また `__iar_dynamic_initialization` を呼び出す前に、`__iar_cstart_tls_init(NULL)` を呼び出す必要があります。

関数 `__iar_dynamic_initialization` は、ヘッダファイル `DLib_Threads.h` で宣言されます。

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。

--map

構文

```
--map {filename|directory}
```

パラメータ

287 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、圧縮手法、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。

注 : 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ります。

関連項目

131 ページの「*重複セクションのマージ*」。

[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_bom

構文

```
--no_bom
```

説明

このオプションを使用して、UTF-8 出力ファイルを生成するときに、バイトオーダーマーク (BOM) を省略します。

関連項目

382 ページの「*--text_out*」および 279 ページの「*テキストエンコーディング*」。

［プロジェクト］>［オプション］>［リンカ］>［エンコード］>［テキスト出力ファイルのエンコード］

--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

217 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_entry

構文

```
--no_entry
```

説明

このオプションを使用して、生成された ELF ファイルのエントリポイント領域をゼロに設定します。

関連項目

360 ページの「*--entry*」。

［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］

--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

217 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] > [アドバンスト] > [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

119 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_free_heap

構文

```
--no_free_heap
```

説明

このオプションを使用して、最も小さいヒープ処理を使用します。このヒープは `free` または `realloc` をサポートしてないので、スタートアップの段階で、さまざまなバッファ、メモリの割り当てを解除することがないアプリケーションなどにヒープメモリを割り当てるアプリケーションだけに適しています。ヒープメモリの割り当てが解放されることはありません。

関連項目

230 ページの「*ヒープメモリハンドラ*」。

［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 2］>［ヒープ選択］

--no_inline

構文

```
--no_inline func[,func...]
```

パラメータ

func	関数シンボルの名前

説明

このオプションを使用しすることで、関数インラインから指定の関数を除外します。

関連項目

367 ページの「*--inline*」。

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。

注 : 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します。標準ライブラリを使用する場合は、一部の手順を再現しなければならないことがあります。どの手順を再現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両方有効にして使用します。

［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。

--no_literal_pool オプションは、Armv6-M および Armv7-M コアのみで使用できます。

関連項目

320 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、すべての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

119 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_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

386 ページの「*--vfe*」および 130 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、診断メッセージの行折り返し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

リンカにおいて、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

287 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、aout.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

126 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` です
alignment	セクションのアライメント。デフォルトは `1` です

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

587 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。

--preconfig

構文

```
--preconfig filename
```

パラメータ

287 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、リンカ構成ファイルを読み込む前に、指定したファイルをリンカに読み込むために使用します。

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。

--printf_multibytes

構文

```
--printf_multibytes
```

説明

このオプションを使用して、リンカが自動的にマルチバイトをサポートする printf フォーマッタを選択するようにします。

[プロジェクト] > [オプション] > [一般オプション] > [ライブラリオプション 1] > [Printf フォーマッタ]

--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

このオプションを使用して、外部シンボルへの参照を別のシンボルを参照するように変更します。

注 : リダイレクトは通常、モジュール内の参照には影響しません。

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] > [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

282 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]

--scanf_multibytes

構文

```
--scanf_multibytes
```

説明

このオプションを使用して、リンカが自動的にマルチバイトをサポートする scanf フォーマッタを選択するようにします。

[プロジェクト] > [オプション] > [一般オプション] > [ライブラリオプション 1] > [Scanf フォーマッタ]

--search、-L

構文

```
--search path

-L 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

97 ページの「*リンクプロセスの詳細*」。

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、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。パラメーターが指定されていない場合は、153 ページの「*セミホスティングのメカニズム*」で説明されている動作になります。

関連項目

153 ページの「*セミホスティングのメカニズム*」。

[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 `stdout`）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。

--stack_usage_control

構文

```
--stack_usage_control=filename
```

パラメータ

287 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

105 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] > [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]

--text_out

構文

```
--text_out {utf8|utf16le|utf16be|locale}
```

パラメータ

`utf8`	UTF-8 エンコードを使用
`utf16le`	UTF-16 リトルエンディアンエンコードを使用
`utf16be`	UTF-16 ビッグエンディアンエンコードを使用
`locale`	システムのロケールエンコードを使用

説明

このオプションを使用して、テキスト出力ファイルを生成するときに使用するのエンコードを指定します。

リンカリストファイルのデフォルトは、メインソースファイルと同じエンコードが使用されます。すべてのその他のテキストファイルのデフォルトは、バイトオーダーマーク (BOM) のある UTF-8 です。

BOM なしの UTF-8 エンコードでテキストを出力する場合、オプション `--no_bom` も使用できます。

関連項目 371 ページの「*--no_bom*」および 279 ページの「*テキストエンコーディング*」。

[プロジェクト] > [オプション] > [リンカ] > [エンコード] > [キスト出力ファイルのエンコード]

--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。

このオプションを使用すると、リンカはセクション `__iar_tls$$DATA` と `__iar_tls$$INIT_DATA` を作成し、セクション `.tdata` と `.tbss` は名前 `.tdata` と `.tbss` を使い続けます。オプション `--threaded_lib` が使用できない場合、セクション `.tdata` のコンテンツは `.data` に配置されているように扱われ、セクション `.tbss` のコンテンツは `.bss.` に配置されているように扱われます。

[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]

--timezone_lib

構文

```
--timezone_lib
```

説明

このオプションを使用して、DLIB ライブラリでタイムゾーンと夏時間機能を有効にします。

注： タイムゾーン機能を自分で実装する必要があります。

関連項目 163 ページの「*__getzone*」。

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。

--treat_rvct_modules_as_softfp

構文

```
--treat_rvct_modules_as_softfp
```

説明

このオプションを使用して、RVCT が生成したすべてのモジュールを標準の（VFP ではない）呼び出し規約を使用していると扱います。

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。

--use_full_std_template_names

構文

```
--use_full_std_template_names
```

説明

C++ エントリのデマングル化された名前で、デフォルトのリンカは一部のクラスで短い名前を使用します。例えば、`"std::basic_string<char、std::char_traits<char>、std::allocator<char>>"` の代わりに `"std::string"`。このオフションを使用して、リンカに完全なマングル化されていない名前を使用するようにさせます。

このオプションは、IDE では使用できません。

--use_optimized_variants

構文

```
--use_optimized_variants={no|auto|small|fast}
```

パラメータ

`no`

標準の最適化でデフォルトの選択を常に使用します。

`auto`

リクエストした最適化のゴールを示す AEABI 属性をもとにした派生型を使用します。

モジュールが `-Ohs` でコンパイルされていて、モジュールで参照している関数の fast 派生型が DLIB ライブラリに存在する場合、それが使用されます。

すべてのモジュールで参照している関数が、`-Ohz` でコンパイルされていて、DLIB ライブラリに small 派生型が存在する場合、それが使用されます。

これはリンカのデフォルトの動作です。

`small`	DLIB ライブラリに small 派生型（コードサイズと実行速度をバランス化し、サイズを選択）がある場合は常にそれを使用します。
`fast`	DLIB ライブラリに fast 派生型（最大実行速度）がある場合は常にそれを使用します。

説明

DLIB ライブラリ関数の最適化された派生型を使用する場合に、このオプションを使用して制御します。（製品に同梱されている一部の DLIB ライブラリには、Cortex-M0 用の小さい整数除算ルーチン、Thumb-2 ISA アーキテクチャをサポートするコア用の高速 `strcpy` 実装など、最適化された派生型が含まれています。）

このオプションが選択した派生型を確認するには、リンカマップファイルにあるリダイレクトのリストを確認します。

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。

--utf8_text_in

構文

```
--utf8_text_in
```

説明

このオプションを使用して、リンカは、バイト オーダーマーク (BOM) のないテキスト入力ファイルを読み込むとき、UTF-8 エンコードを使用できることを指定します。

注 : このオプションはソースファイルには適用されません。

関連項目

279 ページの「*テキストエンコーディング*」。

[プロジェクト] > [オプション] > [リンカ] > [エンコード] > [デフォルト入力ファイルのエンコード]

--version

構文

```
--version
```

説明

このオプションを使用して、リンカがバージョン情報をコンソールに送信してから終了するように指示します。

このオプションは、IDE では使用できません。

--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

デフォルトでは、仮想関数削除は常に実行されますが、すべてのオブジェクトファイルは必要な仮想関数削除情報を含むことが必要です。`--vfe=forced` を使用して、1 つ以上のモジュールで必要な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。

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び出しを実行したり、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

376 ページの「*--no_vfe*」および 130 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

304 ページの「*--diag_warning*」および 358 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

287 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 root 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

119 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

- アライメント
- バイトオーダ（32 ビットモードのみ）
- 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、*組み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` 型整数のアライメントは 2 になります。

構造体のアライメントは、アライ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、アライメントの最初の要素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、403 ページの「*パック構造体型*」を参照してください。

注： #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メント要件を上げることができます。

標準 C ファイル stdalign.h も参照してください。

Arm コア のアライメント

データオブジェクトのアライメントは、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、295 ページの「*--align_sp_on_irq*」、318 ページの「*--no_const_align*」を参照してください。

バイトオーダ（32 ビットモードのみ）

デフォルトのリトルエンディアンバイトオーダでは、*最下位*バイトが、メモリの最下位アドレスに格納されます。*最上位*バイトは、最上位アドレスに格納されます。

ビッグエンディアンバイトオーダは、（**32 ビットモードでのみ選択可能**）では、*最上位*バイトが、メモリの最下位アドレスに格納されます。*最下位*バイトは、最上位アドレスに格納されます。ビッグエンディアンバイトオーダを使用する場合、その他のコンパイラおよび一部のデバイスの I/O レジスタ定義に準拠する #pragma bitfields=reversed ディレクティブを使用しなければならないことがあります。詳細は、393 ページの「*ビットフィールド*」を参照してください。

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。

以下のトピックを解説します :

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

表 32: 整数型

データ型	サイズ	範囲	アライメント
`signed long`			4
32 ビットモードおよび 64 ビットモードの ILP32 64 ビットモードの LP64	32 ビット 64 ビット	-2^{31} ～ $2^{31}-1$ -2^{63} ～ $2^{63}-1$	
`unsigned long`			4
32 ビットモードおよび 64 ビットモードの ILP32 64 ビットモードの LP64	32 ビット 64 ビット	0 ～ $2^{32}-1$ 0 ～ $2^{64}-1$	
`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（続き）

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、論理値の `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` は `long`、または `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。以下に例を示します。

```
/* enum での char 型の使用を無効化 */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

C++ `enum struct` 構文も参照してください。

関連情報については、308 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。

注 : 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 データ型は 4 バイトで UTF-32 のエンコードを使用します。

char16_t 型

char16_t データ型は 2 バイトで UTF-16 のエンコードを使用します。

char32_t 型

char32_t データ型は 4 バイトで UTF-32 のエンコードを使用します。

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。

IAR C/C++ コンパイラ for Arm 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適切に整列された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。必要があればコンテナはタイプで適切に整列される限りは、重ねられることに注意してください。

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 bitfields` 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（430 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

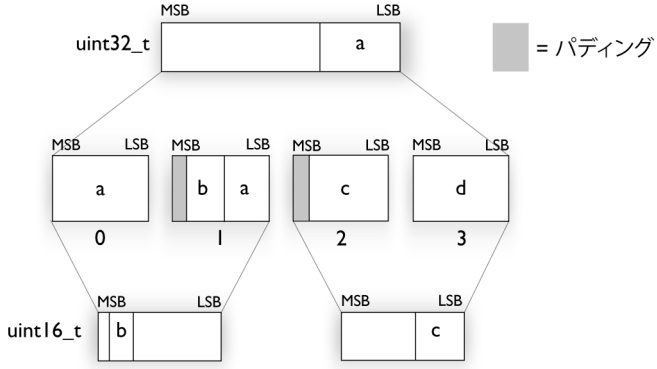
最初のビットフィールド `a` を配置するために、コンパイラは 32 ビットのコンテナをオフセット `0` に割当て、`a` をコンテナの最初のバイトと 2 番目のバイトに入れます。

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。

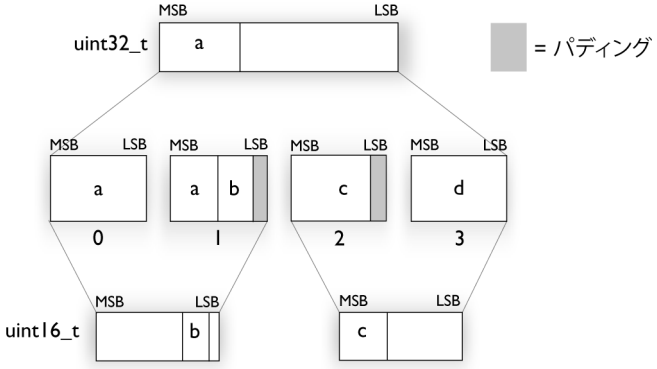
3 番目のビットフィールド `c` は、最初の 16 ビットコンテナに 1 ビットしか残っていないため、オフセット `2` に新しいコンテナが割り当てられ、`c` はこのコンテナの最初のバイトに入れられます。

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 32 ビットのコンテナをオフセット `0` に割当て、`a` を最も重要でない 12 ビットのコンテナに入れます。

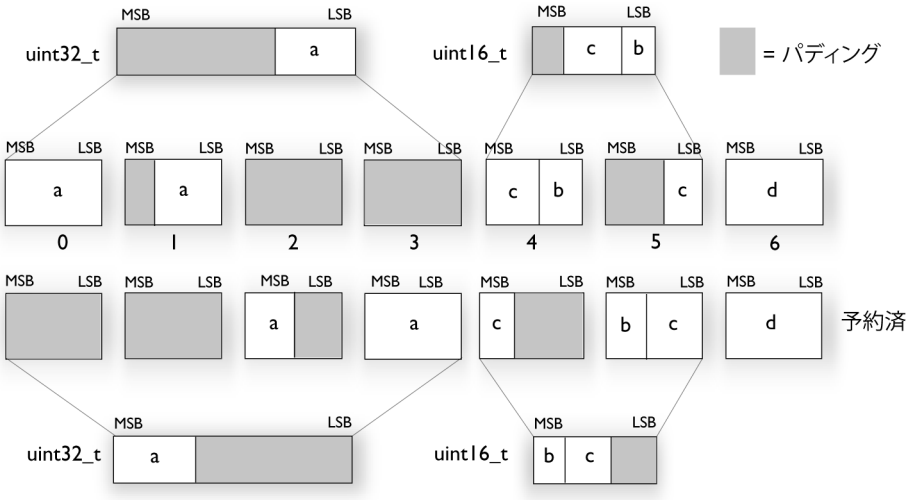
2 番目のビットフィールド `b` を配置するために、新しいコンテナがオフセット `4` に割当てられます。これは、ビットフィールドの型が前のビットフィールドと同じではないためです。`b` は、このコンテナの最下位の 3 ビットに配置されます。

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。

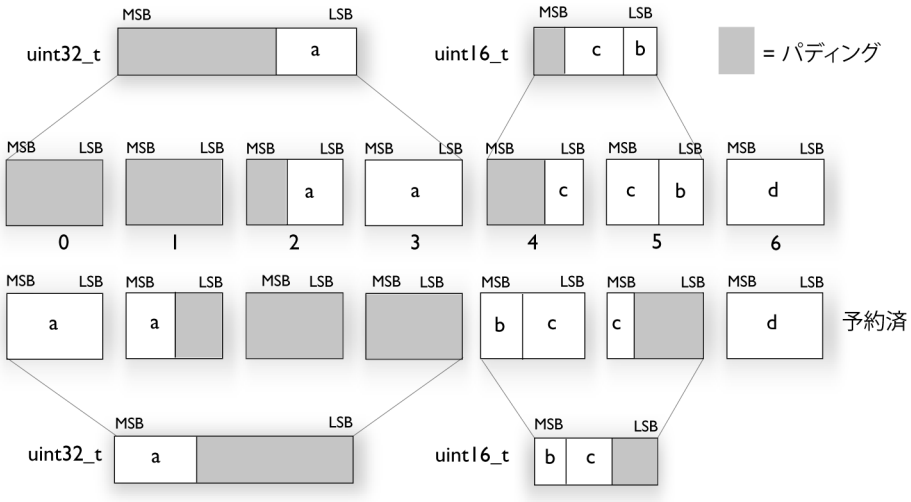
4 番目のメンバ d は、オフセット 6 のバイトに割り当てられます。d は、b や c と同じコンテナには配置できません。その理由は、これがビットフィールドではなく、同じ型でもないために、合わないからです。

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。



パディング

ビットフィールドにアクセスするときに上記のように表示されるように、ビットフィールドコンテナ全体を読み取る / 書き込むために、パディングは通常構造の終わりに追加されます。ただし、ビットが低から高のアドレスに割り当てられているときは、フィールドのアライメントを行う必要がある場合のみパディングは追加されます。

例：

```
struct X { uint32_t x1 : 5; };
```

`uint32_t` ビットフィールドのアライメントが 4 のとき、`struct X` のサイズは 4 で、通常のアライメントでビットフィールドコンテナ (`uint32_t`) 全体を読み取り / 書き込むことができます。ただし、フィールドのアライメントが低く（例えば、`#pragma pack` を使用していることで）、ビットが低アドレスに割り当てられている場合、`struct X` のサイズも小さいです。

基本データ型浮動小数点数型

IAR C/C++ コンパイラ for Arm では、浮動小数点数の値を標準 IEC 60559 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	アライメント
`__fp16`	16 ビット	±2E-14 to 65504	3	5 ビット	11 ビット	2
`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、399 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。

`__fp16` 浮動小数点型は、ストレージ型のみです。すべての数値操作は、`float` に拡張される値で動作します。`__fp16` とレイアウトの互換性がある、標準型 `_Float16` もあります。一部のコアは、`_Float16` 値で直接数値操作をサポートします。その他のコアは、ストレージ専用型です。

注 : C/C++ 標準ライブラリは `_Float16` 型をサポート*しません*。`_Float16` 型で標準ライブラリ関数を使用する場合は、`_Float16` 値を単精度または倍精度にキャストし、適したライブラリ関数を使用する必要があります。`_Float16` 型の文字列フォーマット指定子がないため、明示的なキャストが必要になります。

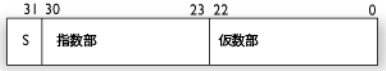
浮動小数点環境

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` を使用して呼び出された場合、`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
```

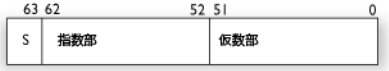
範囲は少なくとも以下のようになります。

```
±1.18E-38 から 3.39E+38
```

浮動小数点数の演算子（+、-、*、および /）の精度は、10 進数で約 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（+、-、*、および /）の精度は、10 進数で約 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

関数ポインタにはこれらのプロパティがあります。

例外モード	データモデル	ポインタサイズ	アドレス範囲
32 ビット	なし	32 ビット	0-0xFFFF'FFFF
64 ビット	ILP32	32 ビット	0-0xFFFF'FFFF
64 ビット	LP64	64 ビット	0-0xFFFF'FFFF'FFFF'FFFF

表 34: 関数ポインタ

注 : ILP32 データモデルでは、レジスタのポインタは常に 64 ビットです。32 ビットポインタがレジスタに読み込まれ、一番低い 32 ビットのみを格納するとき、32 ビットポインタはゼロ拡張です。

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 * IntHandler) (void);
```

これは、#pragma ディレクティブを使用して書き直すことができます。

```
#pragma type_attribute=__thumb
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。これらのプロパティがあります。

例外モード	データモデル	ポインタサイズ	アドレス範囲
32 ビット	なし	32 ビット	0-0xFFFF'FFFF
64 ビット	ILP32	32 ビット	0-0xFFFF'FFFF
64 ビット	LP64	64 ビット	0-0xFFFF'FFFF'FFFF'FFFF

表 35: データポインタ

注 : ILP32 データモデルでは、レジスタのポインタは常に 64 ビットです。32 ビットポインタがレジスタに読み込まれ、一番低い 32 ビットのみを格納するとき、32 ビットポインタはゼロ拡張です。

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。32 ビットモードと、ILP32 データモデルを 64 ビットモードで使用しているとき、`size_t` に使用した型は `unsigned int` です。LP64 データモデルの場合、`size_t` に使用した型は `unsigned long` 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。32 ビットモードと、ILP32 データモデルを 64 ビットモードで使用しているとき、`ptrdiff_t` に使用した型は `size_t` です。LP64 データモデルの場合、`ptrdiff_t` に使用した型は `signed long` 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。IAR C/C++ コンパイラ for Arm では、intptr_t に使用する型は 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、宣言順で連続して格納されます。最初のメンバは、最下位のメモリアドレスに格納されます。

構造体型のアライメント

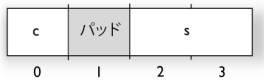
Struct 型および union 型は、一番高いアライメント要求のあるメンバと同じアライメントとなります。このアライメント要求は構造体のメンバにも適用されます。アライメントされた構造体オブジェクトの配列を可能にするに、構造体のサイズはアライメントの偶数倍に調整されます。

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。



構造体のアライメントは 2 バイトです。また、short s に正しいアライメントを保守するため、パディングバイトを挿入しなければならないことがあります。

パック構造体型

__packed データ型属性または #pragma pack ディレクティブを使用して、構造体のメンバのアラインメント要件を緩和します。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ります。

注 : 正しくアライ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構造体で正しい値を構成し、この構造体にアクセスする方が通常は適しています。

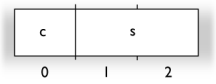
アライメントが正しく設定されていないメンバへのポインタ作成および使用には、特別な注意も必要です。パックされた struct のアライメントが正しく設定されていないメンバに直接アクセスする場合、コンパイラは、必要に応じて正しいコード（ただし、サイズが大きく低速）を出力します。しかし、アライメントが正しく設定されていないメンバへのポインタを使用してそのメンバにアクセスする場合には、通常のコード（サイズが小さく高速）が使用されます。一般的なケースでは、これは機能しません。なぜなら、通常のコードは正しいアライメントに依存することがあるからです。

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()
```

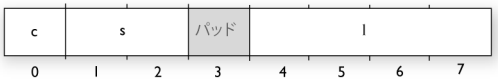
構造 s にはこのメモリレイアウトがあります。



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、アライメントを使用します。メンバ l のアライメントは 4 です。これは構造体 S2 のアライメントが 4 になることを意味します。

詳細については、252 ページの「*構造体要素のアライメント*」を参照してください。

型修飾子

C 規格では、volatile および 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、要素単位になります。以下に例を示します。

```
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。IAR C/C++ コンパイラ for Arm に固有の規則について、以下に説明します。

アクセス規則

IAR C/C++ コンパイラ for Arm 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、アクセス中に他の命令は生成されません。

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 オブジェクトを const 宣言する場合、書き込み禁止になりますが、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、__ro_placement 属性を使用してそれを宣言します。421 ページの「*__ro_placement*」を参照してください。

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 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
- サポートされる GCC 属性

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、412 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。307 ページの「*-e*」を参照してください。

型属性

型属性は、関数の呼び出し方法、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。

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、__cmse_nonsecure_call、__exception、__fiq、
__interwork、__irq、__svc、__swi__no_scratch、__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。

注 : データ型属性 (except `__packed`) は、構造体型フィールドでは使用できません。

データオブジェクトで使用される型属性の構文

*均一の属性構文*を選択すると、データ型属性は型修飾子 `const` と `volatile` と同じ構文規則を使用します。

そうでない場合は、データ型属性は、構文規則の型修飾子 `const` および `volatile` とほぼ同じように使用します。以下に例を示します。

```
__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` はパックされます。

均一の属性構文の詳細については、339 ページの「*--uniform_attribute_syntax*」および 326 ページの「*--no_uniform_attribute_syntax*」を参照してください。

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、属性はリターン型の前に置くか、関数ポインタの括弧内に置く必要があります。例:

```
__irq __arm void my_handler(void);
```

または

```
void (__irq __arm * my_fp)(void);
```

また #pragma type_attribute を使用して、関数型属性を指定できます。

```
#pragma type_attribute=__irq __arm
void my_handler(void);

#pragma type attribute=__irq __arm
typedef void my_fun_t(void);
my_fun_t * my_fp;
```

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、関数の呼び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_alloc、__no_alloc16、__no_alloc_str、__no_alloc_str16、__no_init、__ro_placement
- 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:

  __cmse_nonsecure_entry、__intrinsic、__naked、__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、254 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
__cmse_nonsecure_call	非セキュアのコードを呼び出す関数ポインタを宣言します
__cmse_nonsecure_entry	非セキュアのイメージから呼び出し可能な関数を作成します
__exception	64 ビットモード例外関数を宣言します
__fiq	高速割り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び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	割り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
__naked	関数フレームをセットアップまたはティアダウンするために、コードを生成しないで関数を宣言します

表 36: 拡張キーワードの一覧

拡張キーワード	説明
__nested	__irq で宣言した割り込み関数をネストできるようにします。つまり、同じ割り込みタイプによる割り込みを許可します
__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
__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
__ro_placement	const volatile データをリードオンリーメモリに配置します
__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び出し可能にします
__svc	32 ビットモードのソフトウェア割込み関数、または 64 ビットモードのソフトウェア例外関数を宣言します
__swi	__svc のエイリアス
__task	レジスタ保護の規則を緩和します
__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 36: 拡張キーワードの一覧（続き）

拡張キーワードの詳細

このセクションでは、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

410 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

409 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 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。

64 ビットモードの場合、このキーワードは使用できません。

例

```
__arm int func1(void);
```

__big_endian

構文

408 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、アプリケーションの他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。**64 ビットモード**では使用できません。

注 : このキーワードをポインタで使用している場合、そのポインタは、__big_endian または __little_endian 属性がないポインタと互換性がありません。

例

```
__big_endian long my_variable;
```

関連項目

415 ページの「*__little_endian*」。

__cmse_nonsecure_call

構文

409 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

キーワード `__cmse_nonsecure_call` は関数ポインタに対して使用され、ポインタを介したコールが非セキュア状態に入ることを示します。重要なデータが非セキュア状態にリークすることを回避するために、そのようなコールの前に実行の状態はクリアされます。

`__cmse_nonsecure_call` キーワードは、`--cmse` でコンパイルする際にのみ、関数ポインタに対して使用できます。

キーワード `__cmse_nonsecure_call` は、可変数引数関数、レジスタに合わないパラメータの関数またはリターン値、または浮動ポイントレジスタのパラメータの関数またはリターン値にはサポートされていません。

64 ビットモードの場合、このキーワードは使用できません。

例

```
#include <arm_cmse.h>
typedef __cmse_nonsecure_call void (*fp_ns_t)(void);
static fp_ns_t callback_ns = 0;
__cmse_nonsecure_entry void set_callback_ns(fp_ns_t func_ns) {
  callback_ns = cmse_nsfptr_create(func_ns);
}
```

関連項目

297 ページの「*--cmse*」。

__cmse_nonsecure_entry

構文

410 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

`__cmse_nonsecure_entry` キーワードは非セキュア状態から呼び出されるエントリ関数を宣言します。重要なデータが非セキュア状態にリークすることを回避するために、呼び出し元に戻る前に実行の状態はクリアになります。

キーワード `__cmse_nonsecure_entry` は、レジスタに合わない可変数引数関数、パラメータの関数、またはリターン値をサポートされません。

`__cmse_nonsecure_entry` は、`--cmse` でコンパイルするときだけ使用できます。

64 ビットモードの場合、このキーワードは使用できません。

例

```
#include <arm_cmse.h>
__cmse_nonsecure_entry int secure_add(int a, int b) {
  return cmse_nonsecure_caller() ? a + b : 0;
}
```

関連項目

297 ページの「*--cmse*」。

__exception

構文

409 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

__exception キーワードは、64 ビットモード例外テーブルにある例外ベクタのひとつで関数を使用できるようにします。

例

```
__exception void my_function(void);
```

関連項目

87 ページの「*64 ビットモードの例外関数*」。

__fiq

構文

409 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、高速割り込み関数を宣言します。すべての割り込み関数は、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。

64 ビットモードの場合、このキーワードは使用できません。

例

```
__fiq __arm void interrupt_function(void);
```

__interwork

構文

409 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 のいずれのモードで実行する関数からでも呼び出すことができます。**64 ビットモードの場合**、このキーワードは使用できません。

注 : すべての関数は、interwork です。キーワードは互換性の理由から存在します。

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

409 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、割り込み関数を宣言します。すべての割り込み関数は、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。**64 ビットモードの場合**、このキーワードは使用できません。

例

```
__irq __arm void interrupt_function(void);
```

関連項目

295 ページの「*--align_sp_on_irq*」。

__little_endian

構文

408 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。**64 ビットモード**では使用できません。

注 : このキーワードをポインタで使用している場合、そのポインタは、__big_endian または __little_endian 属性がないポインタと互換性がありません。

例

```
__little_endian long my_variable;
```

関連項目

412 ページの「*__big_endian*」。

__naked

構文

410 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

このキーワードは、コンパイラが、関数フレームをセットアップまた破棄するためのコードを生成しない関数を宣言します。

コンパイラはフレームレイアウトがないと大幅に制限され、宣言した関数の本文は __asm 文から構成する必要があります。拡張したアセンブリ、パラメータリファレンス、または __asm 文と C コードを混合したものを使用することは、正常に動作しない場合があります。

注 : __naked キーワードで宣言した関数を呼び出すことはできません。

例

```
__naked void save_process_state(void);
__naked void restore_process_state(void);
```

__nested

構文

410 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

32 ビットモードの場合、__nested キーワードは、ネストされる割り込みを許可する割り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、割り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、新しい割り込みを割り込み関数に含めることができます。ネストされた割り込みは、__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。

64 ビットモードの場合、__nested キーワードは、ネストされる例外を許可する例外関数の起動および終了を修正します。87 ページの「*64 ビットモードの例外関数*」を参照してください。

例

```
__irq __nested __arm void interrupt_handler(void);
```

関連項目

84 ページの「*ネスト割込み*」および 295 ページの「*--align_sp_on_irq*」。

__no_alloc、__no_alloc16

構文

410 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

417 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することができます。

注 : 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ありません。

関連項目 417 ページの「*__no_alloc*、*__no_alloc16*」。

__no_init

構文 410 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 270 ページの「*非初期化変数*」および 553 ページの「*do not initialize ディレクティブ*」。

__noreturn

構文 410 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。

注 : 拡張キーワード __noreturn は、標準 C キーワード _Noreturn またはマクロ noreturn（stdnoreturn.h が含まれている場合）と同じ意味を持ちます。また標準 C++ 属性 [[noreturn]] とも同様です。

例

```
__noreturn void terminate(void);
```

__packed

構文

408 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、データ型に 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。

  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。
- ほかのどの箇所で使用しても、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 : 通常のアライメント以外のアライ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

```
/* No pad bytes in X: */
__packed struct X { char ch; int i; };
/* __packed is optional here: */
struct X * xp;

/* NOTE: no __packed: */
struct Y { char ch; int i; };
/* ERROR: Yは__packedで定義されていない:*/
__packed struct Y * yp ;

/* Member 'i' has alignment 1: */
struct Z { char ch; __packed int i; };

void Foo(struct X * xp)
{
  /* Error:"int __packed *" -> "int *" not allowed: */
  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

445 ページの「*pack*」。

__ramfunc

構文

410 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

__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 実行用 (.textrw) で、もうひとつは ROM の初期化用 (.textrw_init) 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単にすることです。それが __ramfunc により宣言した目的ではない場合、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

__ramfunc 宣言された関数のブレークポイントの詳細については、『*Arm 用 C-SPY® デバッグガイド*』を参照してください。

__ro_placement

構文

410 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

__ro_placement 属性は、データオブジェクトが読み取り専用メモリに置かれることを指定します。このオブジェクト属性を使用する 2 つの例があります。

- const volatile 宣言されたデータオブジェクトは、デフォルトで読み書き可能メモリとして配置されます。__ro_placement オブジェクト属性は、代わりに読み取り専用メモリのデータオブジェクトに配置されます。
- C++ ではデータオブジェクトは const と宣言され、動的な初期化が必要で、読み書き可能メモリに配置され、システム起動時に初期化されます。__ro_placement オブジェクト属性を使用する場合、データオブジェクトが動的な初期化を必要する場合、コンパイラはエラーメッセージを表示します。

const オブジェクトでは __ro_placement オブジェクト属性のみを使用します。

コンパイラが、データオブジェクトの C++ 動的初期化を静的初期化に最適化できる場合は、C++ オブジェクトの __ro_placement 属性を使用できます。これは、関連のクラス定義のヘッダファイルで、関連の単一のコンストラクタが定義されていて、コンパイラに表示される場合だけ可能です。コンパイラが、コンストラクタを見つけれない場合、またはコンストラクタが複雑すぎる場合は、エラーメッセージ (Error[Go023]) が発行され、完了できません。

例

```
__ro_placement const volatile int x = 10;
```

__root

構文

410 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、119 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

410 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、使用可能なスタックなしに呼び出し可能な関数を宣言します。

関数により宣言された __stackless は呼び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);
```

__svc

構文

409 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

32 ビットモード :

__svc キーワードは、ソフトウェア割込み関数を宣言します。関数呼び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り込み番号が挿入されます。__svc 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。__svc 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り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。ソフトウェア割り込み関数は、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び出し規則に従います。

__svc キーワードには、#pragma svc_number=*number* ディレクティブで指定されるソフトウェア割り込み番号が必要です。svc_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 SVC_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。

注 : ソフトウェア割り込み番号は、関数宣言のみに指定（通常、割り込み関数を呼び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）する必要があります。関数定義には指定しないでください。

注 : Cortex-M を除くすべての割り込み関数は、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。

64 ビットモード :

__svc キーワードは、64 ビットモード例外テーブルにある例外ベクタのひとつで関数を使用できるようにします。87 ページの「*64 ビットモードの例外関数*」を参照してください。

例

ソフトウェア割り込み関数の宣言は、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 svc_number=0x23
__svc int svc0x23_function(int a, int b);
...
```

関数の呼び出し

```
...
int x = svc0x23_function(1, 2);  /* SVC 0x23 によって配置されるため、
                                    リンカは決して
                                    svc0x23_function */
...
```

アプリケーションのソースコード内でソフトウェア割り込み関数を定義

```
...
__svc __arm int the_actual_svc0x23_function(int a, int b)
{
  ...
  return 42;
}
```

関連項目

85 ページの「*ソフトウェア割り込み*」、192 ページの「*呼び出し規約*」、87 ページの「*64 ビットモードの例外関数*」。

__task

構文

409 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。

デフォルトでは、関数は使用された保護レジスタの内容を入口でスタックに保存し、出口で復元します。__task を使用して宣言された関数の場合、レジスタを保存しないため、必要なスタックエリアが小さくなります。

__task を使用して宣言された関数は、呼び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び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び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

409 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。

__thumb として宣言された関数 は、__arm として宣言することはできません。

64 ビットモードの場合、このキーワードは使用できません。

例

```
__thumb int func2(void);
```

__weak

構文

410 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。

public のシンボルの定義上で __weak オブジェクト属性を使用すると、その定義は weak になります。

リンカは、シンボルへの weak 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、weak 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、リンク処理の際、シンボルは weak 定義を必要な数だけと、最大で weak でない定義を 1 つ持つことができます。シンボルが必要で、weak でない定義が 1 つあル場合は、その定義が使用されます。weak でない定義がない場合は、weak 定義のいずれかが使用されます。

例

```
extern __weak int foo; /* A weak reference. */

__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;
}
```

サポートされる GCC 属性

拡張言語モードでは、IAR C/C++ コンパイラは、限られた GCC スタイル属性もサポートします。`__attribute__ ((attribute-list))` 構文をこれらの属性に使用します。

次の属性は、一部またはすべてがサポートされます。詳細については、GCC のドキュメントを参照してください。

- `alias`
- `aligned`
- `always_inline`
- `cmse_nonsecure_call`
- `cmse_nonsecure_entry`
- `const`
- `constructor`
- `deprecated`
- `naked`
- `noinline`
- `noreturn`
- `packed`
- `pcs`（関数に使用される IAR 型属性用）
- `pure`
- `section`
- `target`（関数に使用される IAR 型属性用）
- `unused`
- `used`
- `volatile`
- `weak`

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指定します
`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。
`cstat_disable`	*C-STAT® Static Analysis Guide* を参照してください。
`cstat_enable`	*C-STAT® Static Analysis Guide* を参照してください。
`cstat_restore`	*C-STAT® Static Analysis Guide* を参照してください。
`cstat_suppress`	*C-STAT® Static Analysis Guide* を参照してください。
`data_alignment`	変数のアライメントを高く（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。

表 37: 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
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。
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。
deprecated	非推奨としてエンティティをマークします。
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。
function_category	スタック使用量解析のために関数カテゴリを宣言します。
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。

表 37: 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
no_arith_checks	直後の関数では C-RUN の算術演算チェックを実行しないことを指定します。『*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。
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。
no_stack_protect	直後の関数のスタック保護を無効にします。
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。
once	ヘッダファイルが 1 回以上処理されることを回避します。
optimize	最適化の種類およびレベルを指定します。
pack	構造体および共用体メンバのアライ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び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び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	組み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 section のエイリアスです。
section_prefix	プリフィックスをセクション名に追加します。
stack_protect	直後の関数のスタック保護を強制します。
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。

表 37: 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
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。
svc_number	ソフトウェア割り込み（**32 ビットモード**）あるいは例外（**64 ビットモード**）の割り込み数を設定します。
type_attribute	宣言または定義に型属性を追加します。
unroll	ループを展開します。
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 37: プラグマディレクティブの一覧（続き）

注 : 移植上の理由から、689 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)*」も参照してください。

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、393 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

393 ページの「*ビットフィールド*」。

calls

構文

```
#pragma calls=arg[, arg...]
```

パラメータ

arg はこれらのうちのいずれかです。

function	宣言された関数
category	関数カテゴリの名前を示す文字列

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び出す可能性がある関数を指定します。この情報は、リンカのスタック使用量解析で使用されます。個々の関数と関数カテゴリを指定できます。カテゴリの指定は、そのカテゴリに含まれたすべての関数を指定することと同一です。

例

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
#pragma calls = Fun1, Fun2, "Cat1"
  (*fp)();        // Can call Fun1, Fun2, and all
                  // functions in category "Cat1"
}
```

関連項目

438 ページの「*function_category*」および 105 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、割り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

105 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、直後の変数に与える開始アドレスのアライ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メントに上限が設けられます。この上限は、使用する呼び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、サイズがアライメントの倍数ではない状況に使用できます。

注 : ISO C11 標準およびそれ以降に準拠するために、C コードにアライメント指定子 `_Alignas` を使用することを推奨します。C++11 標準およびそれ以降に準拠するために、C++ コードにアライメント指定子 `alignas` を使用することを推奨します。

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	407 ページの「*型属性*」を参照してください。
object_attribute	409 ページの「*オブジェクト属性*」を参照してください。
@ section_name	256 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

441 ページの「*location*」。

443 ページの「*object_attribute*」。

452 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	407 ページの「*型属性*」を参照してください。
object_attributes	409 ページの「*オブジェクト属性*」を参照してください。
@ section_name	256 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。

注 : 拡張キーワード __packed は、通常の型の属性と構造体型の定義の 2 つの方法で使用できます。プラグマディレクティブ

default_variable_attributes は、型属性として __packed を使用することだけに影響します。構造体定義には、このプラグマディレクティブによる影響はありません。419 ページの「*__packed*」を参照してください。

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

441 ページの「*location*」。

443 ページの「*object_attribute*」。

452 ページの「*type_attribute*」。

deprecated

構文

```
#pragma deprecated=entity
```

説明

タイプの宣言のすぐ前に、このプラグマディレクティブを配置する場合、変数、関数、フィールド、または定数は警告になります。

deprecated プラグマディレクティブは、C++ 属性 [[deprecated]] と同じ効果がありますが、C でも利用できます。

例

```
#pragma deprecated
typedef int * intp_t;  // typedef intp_t is deprecated

#pragma deprecated
extern int fun(void);  // function fun is deprecated

#pragma deprecated
struct xx {            // struct xx is deprecated
  int x;
};
```

```
struct yy {
#pragma deprecated
  int y;                // field y is deprecated
};

intp_t fun(void)        // Warning here
{
  struct xx ax;         // Warning here
  struct yy ay;
  fun();                // Warning here
  return ay.y;          // Warning here
}
```

関連項目　標準 C の Annex K（*境界チェックインターフェース*）。

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。このレベルは、別の診断レベルプラグマディレクティブによって変更されるまで効果を維持します。

関連項目　281 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。このレベルは、別の診断レベルプラグマディレクティブによって変更されるまで効果を維持します。

関連項目 281 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。このレベルは、別の診断レベルプラグマディレクティブによって変更されるまで効果を維持します。

関連項目 281 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`）

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。このレベルは、別の診断レベルプラグマディレクティブによって変更されるまで効果を維持します。

関連項目 281 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。このレベルは、別の診断レベルプラグマディレクティブによって変更されるまで効果を維持します。

関連項目

281 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。

function_category

構文

```
#pragma function_category=category[, category...]
```

パラメータ

category	関数カテゴリの名前を示す文字。

説明

このプラグマディレクティブを使用して、直後の関数に属する関数カテゴリを 1 つ以上指定します。`#pragma calls` と一緒に使用すると、`function_category` ディレクティブは、スタック使用量解析目的の間接的呼び出しの保存先を指定します。

例

```
#pragma function_category="Cat1"
```

関連項目

431 ページの「*calls*」および 105 ページの「*スタック使用量解析*」。

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

275 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|=no_body|=forced_no_body]
```

パラメータ

パラメータなし	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。
no_body	inline キーワードと同じような効果が得られますが、生成した関数には本文がありません。
forced_no_body	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。生成した関数には本文がありません。

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 または #pragma inline=forced_no_body 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 または #pragma inline=forced_no_body を指定すると、すべての最適化レベルの関数をインラインし、再帰などの要因によりエラーになります。

関連項目

91 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り込まないように、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

307 ページの「*-e*」および 337 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。このパラメーターは **64 ビットモード**では使用できません。
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
                                  0xFFFF0400 に配置されます。*/

#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

254 ページの「*データと関数のメモリ配置制御*」および 118 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
```

no_stack_protect

構文

```
#pragma no_stack_protect
```

説明

このプラグマディレクティブを使用して、定義した関数のスタック保護を無効にします。

コンパイラオプション `--stack_protection` が使用されているときだけプラグマディレクティブが影響します。

関連項目

93 ページの「*スタック保護*」。

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、409 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

407 ページの「*拡張キーワードの一般的な構文規則*」。

once

構文

```
#pragma once
```

説明

このプラグマディレクティブをヘッダファイルの最初に配置し、ヘッダファイルがコンパイルで 1 回以上含められることを回避します。1 回以上含められる場合は、1 回目以降は無視されます。

optimize

構文

```
#pragma optimize=[goal][level][vectorize][disable]
```

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`、`medium`、または `high` から指定します。
`vectorize`	NEON ベクタ命令の生成を有効にします。
disable	1 つまたは複数の最適化を無効にします。以下から選択 : `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_relaxed_fp`、浮動小数点の式を極端に最適化する言語の緩和を無効化 `no_tbaa`（型ベースエイリアス解析を無効化） `no_scheduling`、（命令スケジューリングを無効化） `no_vectorize` は、NEON ベクタ命令の生成を無効にします `no_unroll`（ループ展開を無効化）

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

261 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	オプションの構造体アライメントとして、1、2、4、8、16 のいずれか 1 つを設定します。
空白のリスト	構造体アライ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メントを設定します。
pop	構造体アライメントを一時的にプッシュされたアライ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メントラベル。

説明

このプラグマディレクティブは、struct と union メンバの最大アライ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体の宣言から次の #pragma pack まで、またはコンパイル単位の終わりまで影響します。

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

402 ページの「*構造体型*」および 419 ページの「*__packed*」。

__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。

複数のメンバーを持つオーバーロードセットのメンバーである関数には、このプラグマディレクティブを使用できません。

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

331 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 */
}
```

copyright 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。

__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。

複数のメンバーを持つオーバーロードセットのメンバーである関数には、このプラグマディレクティブを使用できません。

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 __section_begin、__section_end、および __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アライメントは、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

210 ページの「*専用セクション演算子*」および*アプリケーションのリンク*の章。

section_prefix

構文

```
#pragma section_prefix="prefix"
```

パラメータ

prefix	すべてのセクション名に追加するプリフィックス。

説明 このプラグマディレクティブには、コンパイラオプション `--section_prefix` と同じ効果があります。`@` 表記や `#pragma location` ディレクティブを使用して明示的に名付けられていない、変換ユニットのすべてのセクション名は変更されます。

関連項目 335 ページの「*--section_prefix*」

stack_protect

構文 `#pragma stack_protect`

説明 このプラグマディレクティブを使用して、定義した関数のスタック保護を強制します。

関連項目 93 ページの「*スタック保護*」。

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。 **注 :** この引数はコンパイラではサポートされていません。
`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。
DEFAULT	デフォルトの動作を設定します。つまり ON です。デフォルトの動作を変更するには、オプション --no_default_fp_contract を使用します。

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
```

関連項目

318 ページの「*--no_default_fp_contract*」。

svc_number

構文

```
#pragma svc_number=number
```

パラメータ

number	ソフトウェア呼び出し数。

説明

このプラグマディレクティブは、__svc 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、割り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 svc_number=17
```

関連項目

85 ページの「*ソフトウェア割り込み*」および 87 ページの「*64 ビットモードの例外関数*」。

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、407 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 `myFunc` に対して生成されます。

```
#pragma type_attribute=__thumb
void myFunc(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 myFunc(void)
{
}
```

関連項目

*拡張キーワード*の章。

unroll

構文

```
#pragma unroll=n
```

パラメータ

n	展開されたループ内にあるループ本体の数で、定数の整数。`#pragma unroll = 1` によって、ループ展開を回避します。

説明

このプラグマディレクティブを使用して、ディレクティブのすぐ後ろにあるループを展開し、展開されたループに *n* のループ本体があるように指示します。このプラグマディレクティブは、`for`、`do`、`while` の各ループの直前にのみ配置でき、繰り返し回数はコンパイル時に決定できます。

展開は通常、比較的小さいループで最も効果的です。ただし、より大きなループを展開すると有効な場合もあります。これはたとえば、共通部分式除去や不要なコードの除去など、展開されていないループの繰り返しの間にさらなる最適化の機会がある場合などです。

`#pragma unroll` ディレクティブを使用して、展開のヒューリスティックがあまり効果的でない場合に、ループを強制的に展開することができます。このプラグマディレクティブはまた、積極的なヒューリスティックの展開を抑える場合にも使用できます。

例

```
#pragma unroll=4
for (i = 0; i < 64; ++i)
{
  foo(i * k, (i + 1) * k);
}
```

関連項目

262 ページの「*ループ展開*」。

vectorize

構文

```
#pragma vectorize [= never]
```

パラメータ

パラメータなし	**32 ビットモード**の 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。

注 : 自動ベクトル化は **64 ビットモード**ではサポートされていません。

例

```
#pragma vectorize
for (i = 0; i < 1024; ++i)
{
  a[i] = b[i] * c[i];]
}
```

関連項目

264 ページの「*ベクトル化*」。

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

424 ページの「*__weak*」。

組み込み関数

- 組み込み関数の概要
- IAR Systems 組み込み関数の説明

組み込み関数の概要

IAR C/C++ コンパイラ for Arm は、組み込み関数のいくつかの異なる設定で使用されます。

アプリケーションで IAR Systems 組み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 *Arm C Language Extensions* (ACLE) 組み込み関数を使用するには、ヘッダファイル `arm_acle.h` をインクルードします。詳細については、455 ページの「*ACLE の組み込み関数*」を参照してください。

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、456 ページの「*Neon 命令の組み込み関数*」を参照してください。

アプリケーションで MVE 組み込み関数を使用するには、ヘッダファイル `arm_mve.h` を含めます。詳細については、456 ページの「*MVE 命令の組み込み関数*」を参照してください。

アプリケーションで CDE 組み込み関数を使用するには、ヘッダファイル `arm_cde.h` を含めます。詳細については、457 ページの「*CDE 命令の組み込み関数*」を参照してください。

アプリケーションで CMSIS 組み込み関数を使用するには、デバイスまたはコアのメインの CMSIS ヘッダファイルをインクルードします。CMSIS ヘッダファイルは、`intrinsics.h` として同じモジュールに含めることはできないので、気をつけてください。CMSIS については、244 ページの「*CMSIS 統合 (32 ビットモード)*」を参照してください。

注 : 組み込み関数名は、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
```

ACLE の組み込み関数

Arm C 言語拡張 (ACLE) は、多くの組み込み関数を指定します。これらはここでは、文書化されていません。代わりに、*Arm C 言語拡張 (IHI 0053D)* を参照してください。

アプリケーションで ACLE の組み込み関数を使用するには、ヘッダファイル arm_acle.h をインクルードします。

Neon 命令の組み込み関数

Neon コプロセッサは、Arm アーキテクチャで定義された Advanced SIMD 命令セット拡張を実装しています。アプリケーションで Neon 組み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、または 64。
- *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。

以下に例を示します。

```
__intrinsic float32x2_t vsub_f32(float32x2_t, float32x2_t);
```

組み込み関数 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 つの要素（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 つの要素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;
```

MVE 命令の組み込み関数

M プロファイルベクタ拡張 (MVE) は Armv8.1-M アーキテクチャで定義されます。アプリケーションで MVE 組み込み関数を使用するには、ヘッダファイル arm_mve.h を含めます。MVE 組み込みに関する詳細は、*Arm C Language Extensions*（ドキュメント番号 :101028）を参照してください。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、または float
- *size* は 8、16、32、または 64
- *number_of_lanes* は、1、2、4、8、または 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、Q レジスタ（128 ビット）に収まらなければなりません。

以下に例を示します。

```
__intrinsic float32x4_t vsubq_f32(float32x4_t, float32x4_t);
```

組み込み関数 vsub_f32 は、2 つの 128 ビットベクタ（Q 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 4 つの要素（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 つの要素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;
```

CDE 命令の組み込み関数

Arm C 言語拡張 (ACLE) は、CDE (Custom Datapath Extension) での使用に多くの組み込み関数を指定します。CDE 組み込みでの情報については、*Arm C Language Extensions* の *Custom Datapath Extension* の章 （ドキュメント番号 : 101028）を参照してください。

IAR Systems 組み込み関数の説明

ここでは、各 IAR Systems 組み込み関数のリファレンス情報を説明します。

__arm_cdp、__arm_cdp2

構文

```
void __arm_cdp(__cpid coproc, __cpopcw opc1, __cpreg CRd,
__cpreg CRn, __cpreg CRm, __cpopc opc2);
void __arm_cdp2(__cpid , __cpopw coprocopc1, __cpreg CRd,
__cpreg CRn, __cpreg CRm, __cpopc opc2);
```

パラメータ

coproc	コプロセッサ番号 0..15。
opc1、*opc2*	コプロセッサ固有の処理コード。
CRd、*CRn*、*CRm*	コプロセッサレジスタ。

説明

コプロセッサ専用データ操作命令 CDP または CDP2 を挿入します。パラメータ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。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

462 ページの「*__CDP*、*__CDP2*」。

__arm_ldc、__arm_ldcl、__arm_ldc2、__arm_ldc12

構文

```
void __arm_ldc(__cpid coproc, __cpreg CRd, const void* p);
void __arm_ldcl(__cpid coproc, __cpreg CRd, const void* p);
void __arm_ldc2(__cpid coproc, __cpreg CRd, const void* p);
void __arm_ldc12(__cpid coproc, __cpreg CRd, const void* p);
```

パラメータ

coproc	コプロセッサ番号 0..15。
CRd	コプロセッサレジスタ。
p	コプロセッサの読み取り先メモリへのポインタ。

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 *CRd* 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。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

472 ページの「*__LDC*、*__LDCL*、*__LDC2*、*__LDC2L*」。

__arm_mcr、__arm_mcr2、__arm_mcrr、__arm_mcrr2

構文

```
void __arm_mcr(__cpid coproc, __cpopc opc1, __ul src, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
void __arm_mcr2(__cpid coproc, __cpopc opc1, __ul src, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
void __arm_mcrr(__cpid coproc, __cpopc opc1, unsigned long long
src, __cpreg CRm);
void __arm_mcrr2(__cpid coproc, __cpopc opc1, unsigned long long
src, __cpreg CRm);
```

パラメータ

coproc	コプロセッサ番号 0..15
opc1、*opc2*	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、*CRm*	読込み元のコプロセッサレジスタ

説明

コプロセッサ書込み命令 MCR、MCR2、MCRR、または MCRR2 を挿入します。パラメータ *coproc*、*opc1*、*opc2*、*CRn*、および *CRm* 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。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

474 ページの「*__MCR*、*__MCR2*」および 475 ページの「*__MCRR*、*__MCRR2*」。

__arm_mrc、__arm_mrc2、__arm_mrrc、__arm_mrrc2

構文

```
unsigned int __arm_mrc(__cpid coproc, __cpopc opc1, __cpreg CRn,
__cpreg CRm, __cpopc opc2);
unsigned int __arm_mrc2(__cpid coproc, __cpopc opc1, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
unsigned long long __arm_mrrc(__cpid coproc, __cpopc opc1,
__cpreg CRm);
unsigned long long __arm_mrrc2(__cpid coproc, __cpopc opc1,
__cpreg CRm);
```

パラメータ

coproc	コプロセッサ番号 0..15
opc1、*opc2*	コプロセッサ固有の処理コード
CRn、*CRm*	読込み元のコプロセッサレジスタ

説明

コプロセッサ読み取り命令 MRC、MRC2、MRRC、または MRRC2 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*、*opc1*、*opc2*、*CRn*、および *CRm* 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。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

474 ページの「*__MCR*、*__MCR2*」、476 ページの「*__MRRC*、*__MRRC2*」。

__arm_rsr、__arm_rsr64、__arm_rsrp

構文

```
unsigned int __arm_rsr(sys_reg special_register);
unsigned long long __arm_rsr64(__sys_reg special_register);
void * __arm_rsrp(sys_reg special_register);
```

パラメータ

special_register	レジスタを指定している文字列リテラル。

説明

システムレジスタを読み取ります。文字列リテラルを使用して、読み取るレジスタを指定します。__arm_rsr および __arm_rsrp は、コンパイラオプション --cpu で指定されたアーキテクチャのために、文字列リテラルは MRS または VMRS 命令で許可されたシステムレジスタの名前を指定できます。

__arm_rsr および __arm_rsrp には、文字列リテラルは、以下のフォーマットを使用して 32 ビットコプロセッサレジスタも指定できます。

coprocessor : *opc1* :c *CRn* :c *CRm* : *opc2*

__arm_rsr64 には、文字列リテラルは、以下のフォーマットを使用して 64 ビットコプロセッサレジスタを指定できます。

coprocessor : *opc1* :c *CRm*

両方のフォーマット用

- *コプロセッサ*は数字で、c0..c15 または cp0..cp15
- *opc1* および *opc2* はコプロセッサ固有の演算コード、0..7

- *CRn* および *CRm* はコプロセッサレジスタ 0..15

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__arm_stc、__arm_stcl、__arm_stc2、__arm_stc2l

構文

```
void __arm_stc(__cpid coproc, __cpreg CRd, const void* p);
void __arm_stcl(__cpid coproc, __cpreg CRd, const void* p);
void __arm_stc2(__cpid coproc, __cpreg CRd, const void* p);
void __arm_stc2l(__cpid coproc, __cpreg CRd, const void* p);
```

パラメータ

coproc	コプロセッサ番号 0..15。
CRd	コプロセッサレジスタ。
p	コプロセッサの書込み先メモリへのポインタ。

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。パラメータ *coproc*、*CRd*、および *p* 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。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

491 ページの「*__STC*、*__STCL*、*__STC2*、*__STC2L*」。

__arm_wsr、__arm_wsr64、__arm_wsrp

構文

```
void __arm_wsr(const char * special_reg, _uint32_t value);
void __arm_wsr64(const char * special_reg, uint64_t value);
void __arm_wsrp(const char * special_reg, const void * value);
```

パラメータ

special_reg	システムレジスタを指定している文字列リテラル。
value	システムレジスタにライトする値。

説明

DP レジスタにライトします。文字列リテラルを使用して、ライトするレジスタを指定します。__arm_wsr および __arm_wsrp は、コンパイラオプション --cpu で指定されたアーキテクチャのために、文字列リテラルは MSR または VMSR 命令で許可されたシステムレジスタの名前を指定できます。

__arm_wsr および __arm_wsrp には、文字列リテラルは、以下のフォーマットを使用して 32 ビットコプロセッサレジスタも指定できます。

coprocessor : *opc1* :c *CRn* :c *CRm* : *opc2*

__arm_wsr64 には、文字列リテラルは、以下のフォーマットを使用して 64 ビットコプロセッサレジスタを指定できます。

coprocessor : *opc1* :c *CRm*

両方のフォーマット用

- *コプロセッサ*はプロセッサ番号で cp0..cp15 または p0..p15
- *opc1* および *opc2* はコプロセッサ固有の演算コード、0..7
- *CRn* および *CRm* はコプロセッサレジスタ 0..15

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__CDP、__CDP2

構文

```
void __CDP(__cpid coproc, __cpopcw opc1, __cpreg CRd, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
void __CDP2(__cpid coproc, __cpopcw opc1, __cpreg CRd, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
```

パラメータ

coproc	コプロセッサ番号 0..15。
opc1、*opc2*	コプロセッサ固有の処理コード。
CRd、*CRn*、*CRm*	コプロセッサレジスタ。

説明

コプロセッサ専用データ操作命令 CDP または CDP2 を挿入します。

パラメータ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。

組み込み関数 __CDP と __CDP2 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 またはそれ以降が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

457 ページの「*__arm_cdp*、*__arm_cdp2*」。

__CLREX

構文

```
void __CLREX(void);
```

説明

CLREX 命令を挿入します。

この組み込み関数には、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__CLZ

構文

```
unsigned int __CLZ(unsigned int);
```

説明

CLZ 命令を挿入します。CLZ 命令が使用できない場合は、同じ結果を得るために、命令の連続シーケンスが挿入されます。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

Arm C 言語拡張 (ACLE) 組み込み関数 __clz、__clzl、および __clzll。

__crc32b、__crc32h、__crc32w、__crc32d

構文

```
unsigned int __crc32b(unsigned int crc, unsigned char data);
unsigned int __CRC32H(unsigned int crc, unsigned short data);
unsigned int __crc32w(unsigned int crc, unsigned int data);
unsigned int __crc32d(unsigned int crc, unsigned long long
data);
```

説明

チェックサム（または初期値）crc とデータの 1 つのアイテムから CRC32 を計算します。

注 : 32 ビット Arm/Thumb 命令には CRC32X を含まないので、__crc32d は __crc32w を 2 回コールするように実装されます。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__crc32cb、__crc32ch、__crc32cw、__crc32cd

構文

```
unsigned int __crc32cb(unsigned int crc, unsigned char data);
unsigned int __CRC32cH(unsigned int crc, unsigned short data);
unsigned int __crc32cw(unsigned int crc, unsigned int data);
unsigned int __crc32cd(unsigned int crc, unsigned long long
data);
```

説明

チェックサム（または初期値）crc とデータの 1 つのアイテムから CRC32C を計算します。

注 : 32 ビット Arm/Thumb 命令には CRC32CX を含まないので、__crc32cd は __crc32cw を 2 回コールするように実装されるので気をつけてください。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__disable_debug

構文

```
void __disable_debug(void);
```

説明

DAIF システムレジスタ（ビット 4）のデバッグリクエストを無効にします。

注 : この組み込み関数は **32 ビットモード**では使用できません。

__disable_fiq

構文

```
void __disable_fiq(void);
```

説明

32 ビットモード : 高速割り込み要求 (fiq) を無効にします。この組み込み関数は、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。

64 ビットモード : DAIF システムレジスタ（ビット 1）の高速割り込み関数を無効にします。

__disable_interrupt

構文

```
void __disable_interrupt(void);
```

説明

割り込みを禁止します。

32 ビットモード : 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 をセットすることにより、実行優先順位レベルを 0 に上げ、他のデバイスに対しては、割り込み要求 (irq) および高速割り込み要求 (fiq) を無効にします。この組み込み関数は、特権モードでのみ使用できます。

64 ビットモード : DAIF システムレジスタ（low 4 ビット以下）にある 4 つの割り込みタイプをすべて無効にします。

__disable_irq

構文

```
void __disable_irq(void);
```

説明

32 ビットモード : 割り込み要求 (irq) を無効にします。この組み込み関数は、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。

64 ビットモード : DAIF システムレジスタ（ビット 2）の割り込み要求を無効にします。

__disable_SError

構文

```
void __disable_SError(void);
```

説明

DAIF システムレジスタ（ビット 3）のデ同期エラー要求を無効にします。

注 : この組み込み関数は **32 ビットモード**では使用できません。

__DMB

構文

```
void __DMB(void);
```

説明

DMB 命令を挿入します。この組み込み関数には、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

Arm C 言語拡張 (ACLE) 組み込み関数 __dmb。

__DSB

構文

```
void __DSB(void);
```

説明

DSB 命令を挿入します。この組み込み関数には、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

Arm C 言語拡張 (ACLE) 組み込み関数 `__dsb`。

__enable_debug

構文

```
void __enable_debug(void);
```

説明

DAIF システムレジスタ（ビット 4）のデバッグ要求を有効にします。

注 : この組み込み関数は **32 ビットモード**では使用できません。

__enable_fiq

構文

```
void __enable_fiq(void);
```

説明

32 ビットモード : 高速割り込み要求 (fiq) を有効にします。この組み込み関数は、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。

64 ビットモード : DAIF システムレジスタ（ビット 1）の高速割り込み要求を有効にします。

__enable_interrupt

構文

```
void __enable_interrupt(void);
```

説明

割り込みを有効にします。

32 ビットモード : 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、割り込み要求 (irq) および高速割り込み要求 (fiq) を有効にします。この組み込み関数は、特権モードでのみ使用できます。

64 ビットモード : DAIF システムレジスタ（low 4 ビット以下）にある 4 つの割り込みタイプをすべて無効にします。

__enable_irq

構文

```
void __enable_irq(void);
```

説明

32 ビットモード : 割り込み要求 (irq) を有効にします。この組み込み関数は、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。

64 ビットモード : DAIF システムレジスタ（ビット 2）の割り込み要求を有効にします。

__enable_SError

構文

```
void __enable_SError(void);
```

説明

DAIF システムレジスタ（ビット 3）の同期エラー要求を有効にします。

注 : この組み込み関数は **32 ビットモード**では使用できません。

__fma、__fmaf

構文

```
double __fma(double x, double y, double z);
float __fmaf(float x, float y, float z);
```

説明

結合浮動小数点積和演算は中間丸めなしで x*y+z を計算し、これは倍精度の場合 __VFMA_F64(z、x、y)、単精度の場合 __VFMA_F32(z, x, y) の組み込みコールに対応します。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__get_BASEPRI

構文

```
unsigned int __get_BASEPRI(void);
```

説明

BASEPRI レジスタの値を返します。この組み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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_CONTROL

構文

```
unsigned int __get_CONTROL(void);
```

説明 CONTROL レジスタの値を返します。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_CPSR

構文

```
unsigned int __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。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_FAULTMASK

構文

```
unsigned int __get_FAULTMASK(void);
```

説明 FAULTMASK レジスタの値を返します。この組み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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_FPSCR

構文

```
unsigned int __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。

この組み込み関数は、VFP コプロセッサを持つデバイスでのみ使用可能です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_interrupt_state

構文

```
__istate_t __get_interrupt_state(void);
```

説明

32 ビットモード :

グローバル割り込み状態を返します。リターン値は、__set_interrupt_state 組み込み関数の引数として使用して、割り込み状態を復元することができます。

この組み込み関数は、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。

64 ビットモード :

DAIF システムレジスタの低位 4 ビット（__istate_t は insigned long long) を返します。

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 と __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 の呼び出し前に無効化された割り込みを有効化することがないことです。

__get_IPSR

構文

```
unsigned int __get_IPSR(void);
```

説明

IPSR レジスタ（割り込みプログラムステータスレジスタ）の値を返します。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_LR

構文

```
unsigned int __get_LR(void);
```

説明

レジスタ (R14) の値を返します。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__get_MSP

構文

```
unsigned int __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。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_PRIMASK

構文

```
unsigned int __get_PRIMASK(void);
```

説明

PRIMASK レジスタの値を返します。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_PSP

構文

```
unsigned int __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。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_PSR

構文

```
unsigned int __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。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 460 ページの「*__arm_rsr*、*__arm_rsr64*、*__arm_rsrp*」。

__get_SB

構文

```
unsigned int __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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__get_SP

構文

```
unsigned int __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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__ISB

構文

```
void __ISB(void);
```

説明

ISB 命令を挿入します。この組み込み関数には、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

Arm C 言語拡張 (ACLE) 組み込み関数 __isb。

__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。

組み込み関数 __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 Armv6T2 またはそれ以降が必要です。

組み込み関数 __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 Armv6T2 またはそれ以降が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目 458 ページの「*__arm_ldc*、*__arm_ldcl*、*__arm_ldc2*、*__arm_ldc12*」。

__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*、*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。

組み込み関数 __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。

組み込み関数 __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 Armv6T2 またはそれ以降が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__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。

__LDREX 組み込み関数には、Arm モードの場合はアーキテクチャ Armv6 またはそれ以降、Thumb モードの場合は Armv6T2 またはベースライン Armv8-M が必要です。

__LDREXB および __LDREXH 組み込み関数には、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 Armv7 またはベースライン Armv8-M が必要です。

__LDREXD 組み込み関数には、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 Armv7-M ではなく Armv7 が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__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。パラメーター *coproc*、*opcode_1*、*CRn*、*CRm*、*opcode_2* は命令でエンコードされ、定数でなければなりません。

この組み込み関数 __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。

組み込み関数 __MCR2 には、Arm モードの場合アーキテクチャ Armv5T 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

459 ページの「*__arm_mcr*、*__arm_mcr2*、*__arm_mcrr*、*__arm_mcrr2*」。

__MCRR、__MCRR2

構文

```
void __MCRR(__cpid coproc, __cpopc opc1, unsigned long long src,
__cpreg CRm);
void __MCRR2(__cpid coproc, __cpopc opc1, unsigned long long
src, __cpreg CRm);
```

パラメータ

coproc	コプロセッサ番号 0..15
opc1	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
CRm	読込み元のコプロセッサレジスタ

説明

コプロセッサ書込み命令 MCRR または MCRR2 を挿入します。パラメータ *coproc*、*opc1*、および *CRm* 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。

組み込み関数 __MCRR と __MCRR2 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 Armv6T2 またはそれ以降が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

459 ページの「*__arm_mcr*、*__arm_mcr2*、*__arm_mcrr*、*__arm_mcrr2*」。

__MRC、__MRC2

構文

```
unsigned int __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul
CRm, __ul opcode_2);
unsigned int __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。パラメーター *coproc*、*opcode_1*、*CRn*、*CRm*、*opcode_2* は命令でエンコードされ、定数でなければなりません。

この組み込み関数 __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。

組み込み関数 __MRC2 には、Arm モードの場合アーキテクチャ Armv5T 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

459 ページの「*__arm_mrc*、*__arm_mrc2*、*__arm_mrrc*、*__arm_mrrc2*」。

__MRRC、__MRRC2

構文

```
unsigned long long __MRRC(__cpid coproc, __cpopc opc1, __cpreg
CRm);
unsigned long long __MRRC2(__cpid coproc, __cpopc opc1, __cpreg
CRm);
```

パラメータ

coproc	コプロセッサ番号 0..15
opc1	コプロセッサ固有の処理コード
CRm	読込み元のコプロセッサレジスタ

説明

コプロセッサ読取り命令（MRRC または MRRC2）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*、*opc1*、および *CRm* 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。

組み込み関数 __MRRC と __MRRC2 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 ArmV6T2 またはそれ以降が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

459 ページの「*__arm_mrc*、*__arm_mrc2*、*__arm_mrrc*、*__arm_mrrc2*」。

__no_operation

構文

```
void __no_operation(void);
```

説明

NOP 命令を挿入します。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ __nop と同等です。

__PKHBT

構文

```
unsigned int __PKHBT(unsigned int x, unsigned int y, unsigned int 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。

この組み込み関数には、Arm の場合は 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__PKHTB

構文

```
unsigned int __PKHTB(unsigned int x, unsigned int y, unsigned int 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、PKHTB 命令を挿入します。

この組み込み関数には、Arm の場合は 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__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。

組み込み関数 __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）を使用している Armv7 アーキテクチャが必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

Arm C 言語拡張 (ACLE) 組み込み関数 __pld。

__PLI

構文

```
void __PLI(void const *);
```

説明

PLI 命令を挿入します。この組み込み関数では、Arm v7 アーキテクチャを使用する必要があります。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

Arm C 言語拡張 (ACLE) 組み込み関数 __pli。

__QADD、__QDADD、__QDSUB、__QSUB

構文

```
signed int __Qxxx(signed int, signed int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm モードの場合はアーキテクチャ Armv5E 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。

この組み込み関数は ACLE（*Arm C 言語拡張*）組み込み関数 `__qadd` および `__qsub` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__QADD8、__QADD16、__QASX、__QSAX、__QSUB8、__QSUB16

構文

```
unsigned int __Qxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

これらの組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__qadd8`、`__qadd16`、`__qasx`、`__qsax`、`__qsub8`、および `__qsub16` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__QCFlag

構文

```
unsigned int __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。この組み込み関数は、Neon (Advanced SIMD) を持つデバイスでのみ使用できます。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__QDOUBLE

構文

```
signed int __QDOUBLE(signed int);
```

説明

ソースレジスタ `Rs` およびデスティネーションレジスタ `Rd` に対し、命令 `QADD Rd,Rs,Rs` を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v5E 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__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。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v5E 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__RBIT

構文

```
unsigned int __RBIT(unsigned int);
```

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。RBIT 命令が使用できない場合は、同じ結果を得るために、命令の連続シーケンスが挿入されます。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組込み関数 `__rbit` と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__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。

この組み込み関数には、Arm の場合は Arm v5E アーキテクチャまたはそれ以降、Thumb モードの場合は Armv7A、Armv7R、または Armv7E-M が必要です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__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。この組み込み関数は、Neon (Advanced SIMD) を持つデバイスでのみ使用できます。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__REV、__REV16、__REVSH

構文

```
unsigned int __REV(unsigned int);
unsigned int __REV16(unsigned int);
signed int __REVSH(short);
```

説明

指定された命令を挿入します。命令が使用できない場合に同じ結果を得るため、命令の個別のシーケンスが挿入されます。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 __rev、__rev16、および __revsh と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__rintn、__rintnf

構文

```
double __rintn(double x);
float __rintnf(float x);
```

説明

数字 x を一番近い整数（偶数になるよう）に丸めます。これにより倍精度の組み込み呼び出し __VRINTN_F64(x)、単精度の __VRINTN_F32(x) のどちらかに一致します。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ROR

構文

```
unsigned int __ROR(unsigned int);
```

説明

ROR 命令を挿入します。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__ror` と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__RRX

構文

```
unsigned int __RRX(unsigned int);
```

説明

`RRX` 命令を挿入します。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__SADD8、__SADD16、__SASX、__SSAX、__SSUB8、__SSUB16

構文

```
unsigned int __Sxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

これらの組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__sadd8`、`__sadd16`、`__sasx`、`__ssax`、`__ssub8`、および `__ssub16` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SEL

構文

```
unsigned int __SEL(unsigned int, unsigned int);
```

説明

`SEL` 命令を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__set_BASEPRI

構文

```
void __set_BASEPRI(unsigned int);
```

説明

BASEPRI レジスタの値を設定します。この組み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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_CONTROL

構文

```
void __set_CONTROL(unsigned int);
```

説明

CONTROL レジスタの値を設定します。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_CPSR

構文

```
void __set_CPSR(unsigned int);
```

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）。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_FAULTMASK

構文

```
void __set_FAULTMASK(unsigned int);
```

説明

FAULTMASK レジスタの値を設定します。この組み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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_FPSCR

構文

```
void __set_FPSCR(unsigned int);
```

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。

この組み込み関数は、VFP コプロセッサを持つデバイスでのみ使用可能です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_interrupt_state

構文

```
void __set_interrupt_state(__istate_t);
```

説明

32 ビットモード : 割り込み状態を、前回 __get_interrupt_state 関数から返された値に復元します。

64 ビットモード : DAIF システムレジスタの 4 低いビットを設定します。

__istate_t 型については、469 ページの「*__get_interrupt_state*」を参照してください。

__set_LR

構文

```
void __set_LR(unsigned int);
```

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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__set_MSP

構文

```
void __set_MSP(unsigned int);
```

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。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_PRIMASK

構文

```
void __set_PRIMASK(unsigned int);
```

説明

PRIMASK レジスタの値を設定します。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_PSP

構文

```
void __set_PSP(unsigned int);
```

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。この組み込み関数は、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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

関連項目 461 ページの「*__arm_wsr*、*__arm_wsr64*、*__arm_wsrp*」。

__set_SB

構文

```
void __set_SB(unsigned int);
```

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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__set_SP

構文

```
void __set_SP(unsigned int);
```

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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__SEV

構文

```
void __SEV(void);
```

説明 SEV 命令を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v7、Thumb モードの場合は Armv6-M または Armv7 が必要です。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 __sev と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__SHADD8、__SHADD16、__SHASX、__SHSAX、__SHSUB8、__SHSUB16

構文

```
unsigned int __SHxxx(unsigned int, unsigned int);
```

説明 指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

これらの組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__shadd8`、`__shadd16`、`__shasx`、`__shsax`、`__shsub8`、および `__shsub16` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMLABB、__SMLABT、__SMLATB、__SMLATT、__SMLAWB、__SMLAWT

構文

```
unsigned int __SMLAxxx(unsigned int, unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMLAD、__SMLADX、__SMLSD、__SMLSDX

構文

```
unsigned int __SMLxxx(unsigned int, unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__smlad`、`__smladx`、`__smlsd`、および `__smlsdx` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMLALBB、__SMLALBT、__SMLALTB、__SMLALTT

構文

```
unsigned long long __SMLALxxx(unsigned int, unsigned int, unsigned long long);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMLALD、__SMLALDX、__SMLSLD、__SMLSLDX

構文

```
unsigned long long __SMLxxx(unsigned int, unsigned int, unsigned
long long);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__smlald`、`__smlaldx`、`__smlsld`、および `__smlsldx` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMMLA、__SMMLAR、__SMMLS、__SMMLSR

構文

```
unsigned int __SMMLxxx(unsigned int, unsigned int, unsigned
int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMMUL、__SMMULR

構文

```
signed int __SMMULxxx(signed int, signed int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMUAD、__SMUADX、__SMUSD、__SMUSDX

構文

```
unsigned int __SMUxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SMUL

構文

```
signed int __SMUL(signed short, signed short);
```

説明

符号付き 16 ビット乗算を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v5-E 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__SMULBB、__SMULBT、__SMULTB、__SMULTT、__SMULWB、__SMULWT

構文

```
unsigned int __SMULxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__sqrt、__sqrtf

構文

```
double __sqrt(double x);
float __sqrtf(float x);
```

説明

オペランド x の平方根を計算します。これにより、倍精度の組み込み呼び出し __VSQRT_F64(x)、単精度の __VSQRT_F32(x) のどちらかに一致します。

これらの組み込み関数は、*Arm C 言語拡張* (ACLE) に従って定義されます。

__SSAT

構文

```
signed int __SSAT(signed int, unsigned int);
```

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み込みます。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。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v6 またはそれ以降、Thumb モードの場合は Armv7-A、Armv7-R、または Armv7-M が必要です。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 __ssat と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__SSAT16

構文

```
unsigned int __SSAT16(unsigned int, unsigned int);
```

説明

SSAT16 命令を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 __ssat16 と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__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	ストア先へのポインタ。

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。

組み込み関数 __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 Armv6T2 またはそれ以降が必要です。

組み込み関数 __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 Armv6-T2 またはそれ以降が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

461 ページの「*__arm_stc*、*__arm_stcl*、*__arm_stc2*、*__arm_stc2l*」。

__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	ストア先へのポインタ。
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*、*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。

組み込み関数 `__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。

組み込み関数 `__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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__STREX、__STREXB、__STREXD、__STREXH

構文

```
unsigned int __STREX(unsigned int, unsigned int *);
unsigned int __STREXB(unsigned char, unsigned char *);
unsigned int __STREXD(unsigned long long, unsigned long long*);
unsigned int __STREXH(unsigned short, unsigned short *);
```

説明

指定された命令を挿入します。

`__STREX` 組み込み関数には、Arm モードの場合はアーキテクチャ Armv6 またはそれ以降、Thumb モードの場合は Armv6-T2 またはベースライン Armv8-M が必要です。

`__STREXB` および `__STREXH` 組み込み関数には、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 Armv7 またはベースライン Armv8-M が必要です。

`__STREXD` 組み込み関数には、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 Armv7-M ではなく Armv7 が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SWP、__SWPB

構文

```
unsigned int __SWP(unsigned int, unsigned int *);
char __SWPB(unsigned char, unsigned char *);
```

説明

指定された命令を挿入します。これらの組み込み関数は、Arm モードでなければなりません。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__SXTAB、__SXTAB16、__SXTAH、__SXTB16

構文

```
unsigned int __SXTxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__TT、__TTT、__TTA、__TTAT

構文

```
unsigned int __TT(unsigned int);
unsigned int __TTT(unsugned int);
unsigned int __TTA(unsigned int);
unsigned int __TTAT(unsigned int);
```

説明

指定された命令を挿入します。これらの組み込み関数を直接使用することは避けてください。代わりに、関数 `cmse_TT`、`cmse_TTT`、`cmse_TT_fptr`、および `cmse_TTT_fptr` を使用します。これらはヘッダファイル `arm_cmse.h` で定義されています。

これらの組み込み関数には、セキュリティ拡張されたアーキテクチャ Armv8-M が必要です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

関連項目

297 ページの「*--cmse*」。

__UADD8、__UADD16、__UASX、__USAX、__USUB8、__USUB16

構文

```
unsigned int __Uxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

これらの組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__uadd8`、`__uadd16`、`__uasx`、`__usax`、`__usub8`、および `__usub16` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__UHADD8、__UHADD16、__UHASX、__UHSAX、__UHSUB8、__UHSUB16

構文

```
unsigned int __UHxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

これらの組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__uhadd8`、`__uhadd16`、`__uhasx`、`__uhsax`、`__uhsub8`、および `__uhsub16` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__UMAAL

構文

```
unsigned long long __UMAAL(unsigned int, unsigned int, unsigned
int, unsigned int);
```

説明

`UMAAL` 命令を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__UQADD8、__UQADD16、__UQASX、__UQSAX、__UQSUB8、__UQSUB16

構文

```
unsigned int __UQxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

これらの組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__uqadd8`、`__uqadd16`、`__uqasx`、`__uqsax`、`__uqsub8`、および `__uqsub16` と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__USAD8、__USADA8

構文

```
unsigned int __USADxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__USAT

構文

```
unsigned int __USAT(signed int, unsigned int);
```

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み込みます。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。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v6 またはそれ以降、Thumb モードの場合は Armv7-A、Armv7-R、または Armv7-M が必要です。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 `__usat` と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__USAT16

構文

```
unsigned int __USAT16(unsigned int, unsigned int);
```

説明

USAT16 命令を挿入します。

この組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 __usat16 と同等です。

注 : この組み込み関数は **64 ビットモード**では使用できません。

__UXTAB、__UXTAB16、__UXTAH、__UXTB16

構文

```
unsigned int __UXTxxx(unsigned int, unsigned int);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v6 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。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__VFMA_F64、__VFMS_F64、__VFNMA_F64、__VFNMS_F64、__VFMA_F32、__VFMS_F32、__VFNMA_F32、__VFNMS_F32

構文

```
double __VFMA_F64(double a, double x, double y);
double __VFMS_F64(double a, double x, double y);
double __VFNMA_F64(double a, double x, double y);
double __VFNMS_F64(double a, double x, double y);
float __VFMA_f32(float a, float x, float y);
float __VFMS_f32(float a, float x, float y);
float __VFNMA_F32(float a, float x, float y);
float __VFNMS_F32(float a, float x, float y);
```

説明

結合浮動小数点積和命令 VFMA、VFMS、VFNMA、または VFNMS を挿入します。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__VMINNM_F64、__VMAXNM_F64、__VMINNM_F32、__VMAXNM_F32

構文

```
double __VMINNM_F64(double x, double y);
double __VMAXNM_F64(double x, double y);
float __VMINNM_F32(float x, float y);
float __VMAXNM_F32(float x, float y);
```

説明

VMINNM または VMAXNM 命令を挿入します。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__VRINTA_F64、__VRINTM_F64、__VRINTN_F64、__VRINTP_F64、__VRINTX_F64、__VRINTR_F64、__VRINTZ_F64、__VRINTA_F32、__VRINTM_F32、__VRINTN_F32、__VRINTP_F32、__VRINTX_F32、__VRINTR_F32、__VRINTZ_F32

構文

```
double __VRINTA_F64(double x);
double __VRINTM_F64(double x);
double __VRINTN_F64(double x);
double __VRINTP_F64(double x);
double __VRINTX_F64(double x);
double __VRINTR_F64(double x);
double __VRINTZ_F64(double x);
float __VRINTA_F32(float x);
float __VRINTM_F32(float x);
float __VRINTN_F32(float x);
float __VRINTP_F32(float x);
float __VRINTX_F32(float x);
float __VRINTR_F32(float x);
float __VRINTZ_F32(float x);
```

説明

指定された丸めを実行し、関連する命令を挿入します :

- VRINTA: 浮動小数点をゼロから遠い方への一番近い整数に丸める
- VRINTM: 浮動小数点を負の無限大方向の整数に丸める
- VRINTN: 浮動小数点を一番近い整数に丸める
- VRINTP: 浮動小数点を正の無限大方向の整数に丸める
- VRINTR: 浮動小数点を整数に丸める（FPSCR の丸めモードを使用）
- VRINTX: 浮動小数点を整数 inexact に丸める（FPSCR の丸めモードを使用）
- VRINTZ: 浮動小数点をゼロ方向の整数に丸める

例えば、__VRINTA_F64 は int に変換される結果になる場合、代わりに命令 VCVTA.S32.F64 が使用されます。unsigned int への変換には、代わりに命令 VCVTA.U32.F64 が使用されます。同様に、VRINTM、VRINTN、VRINTP、および VRINTR は、整数に変換するために VCVTM、VCVTN、VCVTP、および VCVTR が使用されます。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__VSQRT_F64、__VSQRT_F32

構文

```
double __VSQRT_F64(double x);
float __VSQRT_F32(float x);
```

説明

命令 VSQRT の平方根を挿入します。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

__WFE、__WFI、__YIELD

構文

```
void int __xxx(void);
```

説明

指定された命令を挿入します。

これらの組み込み関数には、Arm の場合はアーキテクチャ Arm v7、Thumb モードの場合は Armv6-M または Armv7 が必要です。

この組み込み関数は *Arm C 言語拡張* (ACLE) 組み込み関数 __wfe、__wfi、および __yield と同等です。

注 : これらの組み込み関数は **64 ビットモード**では使用できません。

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

IAR C/C++ コンパイラ for Arm 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、500 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（300 ページの「*-D*」を参照）。
- 定義済プリプロセッサマクロシンボル

  オプション `--predef_macros` を使用して、定義済プリプロセッサマクロシンボル、および特定のコマンドラインのその値を確認します。詳細については、330 ページの「*--predef_macros*」を参照してください。
- 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。プリプロセッサに関する別の拡張子について詳しくは、516 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（331 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");
```

注 : バックスラッシュは、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 つ使用できます。

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。

注 : 定義済みプリプロセッサを一覧にするには、コンパイラオプション `--predef_macros` を使用します。330 ページの「*--predef_macros*」を参照してください。

__AAPCS__

説明

`--aapcs` コンパイラ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び出し規約 (`--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、他の呼び出し規約については未定義です。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_PCS` と同等です。

関連項目

293 ページの「*--aapcs*」。

__AAPCS_VFP__

説明

`--aapcs` コンパイラオプションに基づいて設定される整数。AAPCS の派生 VFP が選択された呼び出し規約 (`--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、他の呼び出し規約については未定義です。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_PCS_VFP` と同等です。

関連項目

293 ページの「*--aapcs*」。

__aarch64__

説明

AArch64 に A64 が選択された命令の場合、シンボルは `1` に設定されます。

関連項目

294 ページの「*--aarch64*」、299 ページの「*--cpu_mode*」、294 ページの「*--abi*」。

__arm__

説明 A32 命令セットのコードを生成するとき、シンボルが定義されます。

関連項目 299 ページの「*--cpu_mode*」。

__ARM_32BIT_STATE

説明 32 ビットモードでコンパイルするとき、シンボルが定義されます。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_64BIT_STATE

説明 64 ビットモードでコンパイルするとき、シンボルが定義されます。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__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。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_NEON` と同等です。

関連項目 297 ページの「*--cpu*」。

__ARM_ALIGN_MAX_PWR

説明 静的オブジェクトの最大アライメントを識別する整数。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_ALIGN_MAX_STACK_PWR

説明 スタックオブジェクトの最大アライメントを識別する整数。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_ARCH

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_ARCH_ISA_A64

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_ARCH_ISA_ARM

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_ARCH_ISA_THUMB

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_ARCH_PROFILE

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_BIG_ENDIAN

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_FEATURE_AES

説明 暗号化 AES 命令がサポートされている場合は、このシンボルは `1` に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_CLZ

説明 `CLZ` 命令がサポートされている場合は、このシンボルは `1` に設定されます。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_FEATURE_CMSE

説明 コンパイラオプション `--cpu` と `--cmse` に基づいて設定される整数。選択したプロセッサの構造に CMSE（Cortex-M セキュリティエクステンション）があり、コンパイラオプション `--cmse` が指定されている場合は、シンボルは `3` に設定されます。

選択したプロセッサの構造に CMSE（Cortex-M セキュリティエクステンション）があり、コンパイラオプション `--cmse` が指定されてない場合は、シンボルは `1` に設定されます。

シンボルは、CMSE なしのコアには未定義です。

関連項目 297 ページの「*--cmse*」および 297 ページの「*--cpu*」。

__ARM_FEATURE_CRC32

説明 CRC32 命令がサポートされる場合、このシンボルは `1` に設定されます（Armv8-A/R ではオプション）。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_CRYPTO

説明 暗号化命令がサポートされる（Neon の Armv8-A/R を指示）場合、このシンボルは `1` に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_DIRECTED_ROUNDING

説明 丸め方向および変換命令がサポートされている場合は、このシンボルは `1` に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_DSP

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目 *Arm C 言語拡張* (IHI 0053D)

__ARM_FEATURE_FMA

説明 FPU が結合浮動小数点積和演算をサポートしている場合は、このシンボルは `1` に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_FP16_FML

説明

Armv8.2-A FP16 乗算命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_IDIV

説明

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__ARM_FEATURE_NUMERIC_MAXMIN

説明

浮動小数点の最大と最小命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_QBIT

説明

Q（飽和）グローバルフラグがサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_QRDMX

説明

SQRDMLAH および SQRDMLSH 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SAT

説明

SSAT および USAT 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SHA2

説明

暗号化 SHA1 および SHA2 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SHA3

説明

暗号化 SHA3 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SHA512

説明

Armv8.2-A 暗号化 SHA2 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SIMD32

説明

32 ビット SIMD 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SM3

説明

暗号化 SM3 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_SM4

説明

暗号化 SM4 命令がサポートされている場合は、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FEATURE_UNALIGNED

説明

このシンボルは、ターゲットがアライメントされていないアクセスをサポートしている、およびアライメントされていないアクセスが許可されている場合のみ設定されます。コンパイラオプション `--no_unaligned_access` はアライメントされていないアクセスを許可しないために使用されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_FP

説明

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__ARM_FP16_ARGS

説明

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__ARM_FP16_FML

説明

FP16 乗算命令がサポートされている場合は、このシンボルは `1` に設定されます。このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__ARM_FP16_FORMAT_IEEE

説明

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__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にマルチメディア用の Armv6 SIMD 拡張がある場合、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。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_FEATURE_SIMD32` と同等です。

関連項目 297 ページの「*--cpu*」。

__ARM_NEON

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_NEON_FP

説明 このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_PCS_AAPCS64

説明 変換ユニットのデフォルトの手順の呼出し標準が、AAPCS64 標準に準拠する場合、このシンボルは `1` に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__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。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_ARCH_PROFILE` と同等です。

関連項目 297 ページの「*--cpu*」。

__ARM_ROPI

説明 変換ユニットが読み込み専用の位置に依存しないモードでコンパイルされる場合、このシンボルは `1` に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_RWPI

説明

変換ユニットが読み込み / 書き込み位置に依存しないモードでコンパイルされる場合、このシンボルは 1 に設定されます。

このシンボルは、*Arm C 言語拡張* (ACLE) に従って定義されます。

__ARM_SIZEOF_MINIMAL_ENUM

説明

列挙型の最小サイズ（1 または 4）を示す整数です。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__ARM_SIZEOF_WCHAR_T

説明

`wchar_t` 型のサイズ（2 または 4）を示す整数です。

このプロプロセッサシンボルは、ACLE (*Arm C Language Extensions*) に従って定義されます。

関連項目

Arm C 言語拡張 (IHI 0053D)

__ARMVFP__

説明

`--fpu` オプションを反映する整数で、`__ARMVFPV2__`、`__ARMVFPV3__`、`__ARMVFPV4__`、または `__ARMVFPV5__` 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

310 ページの「*--fpu*」。

__ARMVFP_D16__

説明

`--fpu` コンパイラオプションに基づいて設定される整数。選択された FPU が 16 D レジスタのみを持つ VFPv3 または VFPv4 ユニットの場合、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

310 ページの「*--fpu*」。

__ARMVFP_SP__

説明

--fpu コンパイラオプションに基づいて設定される整数。選択された FPU が 32 ビットの単精度のみをサポートする場合、このシンボルは 1 に設定されます。それ以外の場合、シンボルは未定義です。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ __ARM_FP と同等です。

関連項目

310 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

511 ページの「*__FILE__*」、321 ページの「*--no_normalize_file_macros*」、322 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__、__ARM8A__、__ARM8M_BASELINE__、__ARM8M_MAINLINE__、__ARM8R__ または __ARM8EM_MAINLINE__ に定義されます。これらの シンボル名は、__CORE__ シンボルをテストする際に使用できます。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ __ARM_ARCH と同等です。

__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 `201703L` 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、Thumb（T または T32）には `1`、Arm (A32) には `2`、または A64 には `3` が定義されます。

__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 2018` など）で返されます。

このシンボルは、C 規格で必須です。

__EXCEPTIONS__

説明

例外が C++ でサポートされるときは、シンボルは定義されます。

関連項目

319 ページの「*--no_exceptions*」。

__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

510 ページの「*__BASE_FILE__*」、321 ページの「*--no_normalize_file_macros*」、322 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。

関連項目 307 ページの「*-e*」および 513 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。`main()` を使用している場合は `char __FUNCTION__[]="main";` と同様です。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 307 ページの「*-e*」および 513 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。現在の値は 9 です。製品の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 IAR C/C++ コンパイラ for Arm でコンパイルされる場合に、`1` に設定される整数。

__ilp32__

説明 AArch64 状態で A64 命令セットにコンパイルするときに ILP32 データモデルが使用される場合、このシンボルが定義されます。

関連項目 294 ページの「*--abi*」。

__LIBCPP

説明 Libc++ ライブラリを使用しているときに定義されたシンボル。

_LIBCPP_ENABLE_CXX17_REMOVED_FEATURES

説明

デフォルトでは、Libc++ ライブラリは廃止された C++17 機能をサポートしていません。これらの機能をサポートできるようにするには、関連のシステムヘッダを含める前に、プリプロセッサシンボル `_LIBCPP_ENABLE_CXX17_REMOVED_FEATURES` を定義します。一部の廃止された機能のリストについては、528 ページの「*サポートされてない C/C++ 関数*」を参照してください。

__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。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_BIG_ENDIAN` と同等です。

__lp64__

説明

AArch64 状態で A64 命令セットにコンパイルするときに LP64 データモデルが使用される場合、このシンボルが定義されます。

関連項目

294 ページの「*--abi*」。

__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

307 ページの「*-e*」および 512 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。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_ROPI` と同等です。

関連項目

333 ページの「*--ropi*」。

__RTTI__

説明

ランタイム情報 (RTTI) が C++ でサポートされるときは、シンボルは定義されます。

関連項目

322 ページの「*--no_rtti*」。

__RWPI__

説明

コンパイラオプション `--rwpi` の使用時に定義される整数。

このプリプロセッサシンボルは ACLE（*Arm C 言語拡張*）マクロ `__ARM_RWPI` と同等です。

関連項目

334 ページの「*--rwpi*」。

__STDC__

説明

整数が `1` に設定されるということは、コンパイラが標準 C に準拠していることを意味します。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_LIB_EXT1__

説明

`201112L` に設定された整数と、C 標準の Annex K シグナル、*境界チェックインターフェース*がサポートされていることを示します。

関連項目

517 ページの「*__STDC_WANT_LIB_EXT1__*」。

__STDC_NO_ATOMICS__

説明 コンパイラがアトミックタイプまたは stdatomic.h をサポートしてない場合は、1 に設定します。

関連項目 528 ページの「*アトミック処理*」。

__STDC_NO_THREADS__

説明 1 に設定すると、反映がスレッドをサポートしないことを示します。

__STDC_NO_VLA__

説明 1 に設定すると、C 変数の長さ配列、VLAs が有効でないことを示します。

関連項目 342 ページの「*--vla*」。

__STDC_UTF16__

説明 1 に設定すると、char16_t タイプの値が UTF-16 エンコードであることを示します。

__STDC_UTF32__

説明 1 に設定すると、char32_t タイプの値が UTF-32 エンコードであることを示します。

__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 201710L 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、C 規格で必須です。

__thumb__

説明

T または T32 命令セットのコードを生成するとき、シンボルが定義されます。

関連項目

299 ページの「*--cpu_mode*」。

__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。

#include_next

説明

これは `#include` ディレクティブの変数です。現在のソースファイル（`#include_next` ディレクティブが含まれているもの）があるディレクトリに従う検索パスで、ディレクトリにある名前が付けられたファイルだけを検索します。

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。

注： `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 161 ページの「*__aeabi_assert*」。

__STDC_WANT_LIB_EXT1__

説明 システムヘッダファイルに含まれる前に、このシンボルを 1 に定義すると、C 標準の Annex K、*境界チェックインターフェース*からの関数を使用できます。

関連項目 144 ページの「*バウンドチェック機能*」および 529 ページの「*C 境界チェックインターフェース*」。

警告

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。

C/C++ 標準ライブラリ関数

- C/C++ 標準ライブラリの概要
- DLIB ランタイム環境 — 実装の詳細

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。

C/C++ 標準ライブラリの概要

コンパイラには、2 つの異なる C/C++ 標準ライブラリの実装があります。

IAR DLIB ランタイム環境は、C および C++ の低レベルなサポートを提供します。このランタイム環境を使用しているときは、2 つのライブラリの選択肢があります。

- IAR DLIB C/C++ ライブラリは、標準の C/C++ に準拠する C/C++ 標準ライブラリの完全な実装で、スレッド関連の機能を除く標準 C および C++14 に準拠しています。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を含めるように設定することもできます。
- IAR Libc++ ライブラリは、標準の C/C++ に準拠する C/C++ 標準ライブラリの完全な実装で、スレッド関連および `filesystem` 関数の機能を除く標準 C および C++14 に準拠しています。Libc++ ライブラリは、オープンソースライセンスで LLVM から取得されます。また、C++17 標準からの拡張と制限でそのまま使用されます。

これらの実装はどちらも IAR DLIB ランタイム環境の上位にビルトインされ、どちらの実装も IEEE 754 フォーマットの浮動小数点の数値をサポートします。この環境についての詳細、および DLIB ライブラリのカスタマイズ方法については、*DLIB ランタイム環境*の章を参照してください。

ライブラリ関数の詳細情報については、製品とともに提供される `arm¥doc` ディレクトリにある `HelpDLIB6.chm` ファイルを参照してください。Libc++ ライブラリ関数についての参照情報はありません。

ライブラリ関数の詳細については、処理系定義の動作の章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

み込みます。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、実行時に呼び出しに失敗するか、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、137 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。

独自バージョンでライブラリモジュールを上書きする方法については、141 ページの「*ライブラリモジュールのオーバライド*」を参照ください。

より高精度な代替ライブラリ関数

cos、sin、tan、および 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。関数の float バリアントは __iar_*xxx*_accuratef、関数の long double バリアントは __iar_*xxx*_accuratel、*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り込みで同時に呼び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 ランタイム環境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。

- ヒープ機能 — malloc、free、realloc、calloc など、C++ 演算子 new および delete
- ロケール関数 — localeconv、setlocale
- マルチバイト関数 — mblen、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、perror、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、getwchar、putchar、および putwchar が含まれますが、これに加えて --printf_multibytes と --dlib_config=Full オプションを使用している場合、printf と sprintf 関数（または任意の派生型）でもヒープを使用する可能性があります。

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破壊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り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び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、どれも破棄されません。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。

DLIB ランタイム環境 — 実装の詳細

以下のトピックを解説します :

- 522 ページの「*DLIB ランタイム環境の概要*」
- 522 ページの「*C ヘッダファイル*」
- 523 ページの「*C++ ヘッダファイル*」
- 527 ページの「*組み込み関数としてのライブラリ関数*」
- 528 ページの「*サポートされてない C/C++ 関数*」
- 528 ページの「*アトミック処理*」
- 529 ページの「*C の追加機能*」
- 532 ページの「*非標準の実装*」
- 532 ページの「*ライブラリにより内部的に使用されるシンボル*」

DLIB ランタイム環境の概要

DLIB ランタイム環境は、組み込みシステムに利用される最も重要な C/C++ 標準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、「*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*の章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み込み関数。詳細については、「*組み込み関数*」を参照してください。

また、DLIB ランタイム環境には C の追加機能も含まれています。529 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 ランタイム環境に固有の C 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*の章で説明しています。

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	代替スペル
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

表 38: 従来の標準 C ヘッダファイル— DLIB

ヘッダファイル	用途
signal.h	さまざまな例外条件の制御
stdalign.h	データオブジェクトでアライメントの取り扱い
stdarg.h	可変引数のアクセス
stdatomic.h	アトミック処理のサポートを追加 アトミック処理は、命令セットがそれらをサポートするコアで使用できます
stdbool.h	C の bool データ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
stdnoreturn.h	リターン値のない関数のサポートを追加
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
threads.h	複数のスレッドの実行のサポートを追加 この機能はサポートされていません。
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
wchar.h	ワイド文字のサポート
wctype.h	ワイド文字の分類

表 38: 従来の標準 C ヘッダファイル — DLIB（続き）

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++ ライブラリ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++ 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
algorithm	コンテイナーおよびその他のシーケンスに対する一般的な処理を複数定義
any	std::any クラスでのサポートを追加。Libc++ が必要
array	配列シーケンスコンテナのサポートを追加
atomic	アトミック処理のサポートを追加 アトミック処理は、命令セットがそれらをサポートするコアで使用できます
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
charconv	std::to_chars および std::from_chars ルーチンのサポートを追加。Libc++ が必要
chrono	時間ユーティリティのサポートを追加
codecvt	エンコード間の変換のサポートを追加
complex	複素数演算をサポートするクラスを定義
condition_variable	スレッド条件変数のサポートを追加します。 この機能はサポートされていません。
deque	デキューシーケンスコンテナ
exception	例外処理を制御する複数の関数を定義
forward_list	転送リストシーケンスコンテナのサポートを追加
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
functional	複数の関数オブジェクトを定義
future	スレッド間の関数情報の受け渡しのサポートを追加 この機能はサポートされていません。
hash_map	ハッシュアルゴリズムに基づく map 連想コンテナ。これは C++14 ヘッダで Libc++ では利用できません。
hash_set	ハッシュアルゴリズムに基づく set 連想コンテナ。これは C++14 ヘッダで Libc++ では利用できません。
initializer_list	initializer_list クラスのサポートを追加
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

表 39: C++ ヘッダファイル

ヘッダファイル	用途
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
iterator	共通のイテレータと、イテレータに対する処理を定義
limits	数値を定義
list	双方向リンクリストシーケンスコンテナ
locale	さまざまな文化圏の慣習への対応
map	map 連想コンテナ
memory	メモリ管理機能定義
mutex	データレース保護オブジェクト mutex のサポートを追加 この機能はサポートされていません。
new	記憶領域の割当て / 解放を行う複数の関数を宣言
numeric	シーケンスに対する一般的な数値操作
optional	std::optional クラステンプレートのサポートを追加。Libc++ が必要
ostream	挿入を実行するクラスを定義
queue	キューシーケンスコンテナ
random	ランダムな数字のサポートを追加
ratio	コンパイル時の有理数演算のサポートの追加
regex	正規表現のサポートを追加
scoped_allocator	メモリリソース scoped_allocator_adaptor のサポートを追加
set	set 連想コンテナ
shared_mutex	データレース保護オブジェクト shared_mutex のサポートを追加 この機能はサポートされていません。
slist	一方向リンクリストシーケンスコンテナ。これは C++14 ヘッダで Libc++ では利用できません。
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
stack	スタックシーケンスコンテナ
stdexcept	例外をレポートする複数のクラスを定義
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

表 39: C++ ヘッダファイル（続き）

ヘッダファイル	用途
string_view	std::basic_string_view クラステンプレートのサポートを追加。Libc++ が必要
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
system_error	グローバル エラーレポートのサポートを追加
thread	複数のスレッドの実行のサポートを追加します。この機能はサポートされていません。
tuple	tuple クラスのサポートを追加
typeinfo	タイプ情報サポートの定義
typeindex	タイプインデックスのサポートを追加
typetraits	タイプでの特性のサポートを追加
unordered_map	命令されてないマップに関連のコンテナのサポートを追加
unordered_set	命令されてないセットに関連のコンテナのサポートを追加
utility	複数のユーティリティコンポーネントを定義
valarray	配列コンテナの長さの変更を定義
variant	std::variant クラステンプレートのサポートを追加。Libc++ が必要
vector	ベクタシーケンスコンテナ

表 39: C++ ヘッダファイル（続き）

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。前者は、すべての宣言されたシンボルを、グローバルと std 名前空間に入れます。後者はグローバル名前空間にのみ入れます。

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
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

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
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	代替スペル
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
cstdalign	データオブジェクトでアライメントの取り扱い
cstdarg	可変引数のアクセス
cstdatomic	アトミック処理のサポートを追加
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
cstdnoreturn	リターン値のない関数のサポートを追加
cstring	さまざまな種類の文字列の操作
ctgmath	汎用型の数学関数
cthreads	複数のスレッドの実行のサポートを追加します。この機能はサポートされていません。
ctime	さまざまな時刻 / 日付フォーマットの変換
cuchar	Unicode 機能
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）

組み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、状況によっては組み込み関数として扱われ、memcpy、memset、strcat など、通常の関数呼び出しではなくインラインコードを生成します。

サポートされてない C/C++ 関数

次のファイルには、IAR C/C++ コンパイラでサポートされてないコンテンツがあります。

- threads.h、condition_variable、future、mutex、shared_mutex、thread、cthreads
- memory_resource
- filesystem

一部のライブラリ関数は同じアドレスを持ちます。これは、cos(double) や cosl(long double) のように、ライブラリ関数のパラメータの型が異なるがサイズが同じ場合に特に発生します。

IAR C/C++ Compiler は、C11 と C++14 標準で説明されているようにスレッドをサポートしていません。ただし、DLib_Threads.h と RTOS を使用して、スレッドをサポートするアプリケーションをビルドできます。詳細については、171 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。

ヘッダ実行のコンテナの C++17 パラレルアルゴニズムは、Libc++ ではサポートされていません。

デフォルトでは、Libc++ ライブラリは、auto_ptr()、auto_ptr_ref()、random_shuffle、set_unexpected()、get_unexpected()、unary_function()、binary_function()、const_mem_fun()、const_mem_fun_ref_t() などの廃止された C++17 関数をサポートしていません。

廃止された C++17 機能のサポートを有効にするには、関連のシステムヘッダを含める前に _LIBCPP_ENABLE_CXX17_REMOVED_FEATURES プリプロセッサシンボルを定義します。

アトミック処理

アトミックアクセスをサポートする命令セットのあるコアをコンパイルするとき、標準 C および C++ アトミック操作は、stdatomic.h および atomic ファイルで利用できます。アトミック操作が利用できない場合、事前定義したプリプロセッサシンボル __STDC_NO_ATOMICS__ が 1 に設定されます。これは、C と C++ の両方で該当します。

ハードウェアによりネイティブに取り扱うことができないアトミック処理は、ライブラリ関数に渡されます。IAR C/C++ コンパイラ for Arm は、これらの関数の実装は含みません。テンプレート実装は src¥lib¥atomic¥libatomic.c ファイルにあります。

C の追加機能

DLIB ランタイム環境には、追加された C 機能がいくつか含まれています：

- C 境界チェックインターフェース
- DLib_Threads.h
- iar_dlmalloc.h
- LowLevelIOInterface.h
- stdio.h
- stdlib.h
- string.h
- time.h (time32.h, time64.h)

C 境界チェックインターフェース

C ライブラリは、標準 C の Annex K（*境界チェックインターフェース*）をサポートします。ヘッダファイル errno.h、stddef.h、stdint.h、stdio.h、stdlib.h、string.h、time.h (time32.h, time64.h)、および wchar.h にシンボル、型、および関数を追加します。

インターフェースを有効にするには、システムヘッダファイルに含まれる前に、プリプロセッサー拡張子 __STDC_WANT_LIB_EXT1__ を 1 に定義します。517 ページの「*__STDC_WANT_LIB_EXT1__*」を参照してください。

追加されることの利点は、コンパイラが、安全でない関数の使用に警告メッセージを発行し、それによりインターフェースはより安全なバージョンを持つことです。例えば、より安全な strcpy_s の代わりに strcpy を使用すると、コンパイラは警告メッセージを発行します。

DLib_Threads.h

DLib_Threads.h ヘッダファイルは、ロックとスレッドローカルストレージ (TLS) 変数のサポートが含まれます。これはスレッドのサポートの実行に役立ちます。詳細については、ヘッダファイルを参照してください。

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、229 ページの「*ヒープについて*」を参照してください。

LowLevelIOInterface.h

ヘッダファイル LowLevelInterface.h には、DLIB によって使用された低レベル I/O 関数の宣言を含みます。159 ページの「*DLIB 低レベル I/O インタフェース*」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。

インタフェースは、3 つのヘッダファイルで定義されます。

- time32.h は __time32_t、time_t、__time32、time、および関連の関数を定義します。
- time64.h は __time64_t、time_t、__time64、time、および関連の関数を定義します。
- time.h には、_DLIB_TIME_USES_64 の定義に従って、time32.h または time64.h が含まれます。

  _DLIB_TIME_USES_64 が次の場合 :
  - 1 に定義されている場合、time64.h を含みます。
  - 0 に定義されている場合、time32.h を含みます。
  - 定義されてない場合、time64.h を含みます。

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。

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。

167 ページの「*__time32*、*__time64*」を参照してください。

long が 8 バイトで 64 ビット time.h が使用される場合、clock_t は 8 バイトで、それ以外の場合は、4 バイトです。

デフォルトでは、時間ライブラリは、タイムゾーンおよび夏時間機能をサポートしません。これらの機能を有効にするには、リンカオプション --timezone_lib を使用します。383 ページの「*--timezone_lib*」を参照してください。

__getzone からタイムゾーンと夏時間情報を読み込みまたは強制読み込みには、2 つの関数が使用できます。

- int _ReloadDstRules (void)
- int _ForceReloadDstRules (void)

これらの両方の関数は、DST ルールが見つかった場合は 0 を返し、見つからなかった場合は -1 を返します。

非標準の実装

これらの関数は、C または C ++ 標準で指定されたようには機能しません。

- `fopen_s` および `freopen`

  これらの C 関数は、排他的属性 `u` を低レベルインターフェースに伝播しません。

- `towupper` および `towlower`

  これらの C 関数は、`A`、...、`Z` および `a`, ...、`z` のみを取り扱います。

- `iswalnum`, ..., `iswxdigit`

  これら C 関数は、0 ～ 127 の範囲の引数だけを取り扱います。

- 比較C関数`strcoll`および`strxfrm`は意図したように動作しません。同様のことが、C++ 同等の関数でも発生します。

- `now`

  C++ ヘッダ `chrono` のこの C++ 関数は、`_Xtime_get_ticks()` 関数と C `time.h` マクロ `CLOCKS_PER_SEC` を使用します。デフォルトでは、`_Xtime_get_ticks()` は `__clock()` を呼び出します。これが適切ではない場合、`chrono` や `clock_t _Xtime_get_ticks()` を使用する前に、マクロ `_XTIME_NSECS_PER_TICK` の設定を上書きする必要があります。

ライブラリにより内部的に使用されるシンボル

システムヘッダファイルは、組み込み関数、シンボル、プラグマディレクティブなどを使用します。ライブラリやコンパイラーで定義されているものもあります。これらの準備されたシンボルは `__` で始まり、またライブラリだけで使用されるべきです。

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。

ライブラリ内で使用されるシンボルは、このガイドではリストされていません。

リンカ設定ファイル

- 概要
- ビルドタイプの宣言
- 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。96 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配置するため、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。

- ビルドタイプの宣言

  ビルドが従来の ROM システム用かは RAM システム用かをリンカに通知し、異なるメモリ領域に適切なセクションのみが配置されているかをリンカがチェックできるようにします。
- 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。
- 名前の特殊文字
  シンボルまたはセクションの名前に識別されない文字を使用するときは、バック クオテーションで名前を囲みます。たとえば、`'My Name'`。

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。

ビルドタイプの宣言

リンカ設定ファイルのビルドタイプを宣言すると、ビルドが従来の ROM システム（変数、プログラム開始時の初期化で使用する）用か、デバッグに使用する（別の初期化スタイルを使用）RAM システム用か、リンカを指定します。

ディレクティブ用のビルド

構文

```
build for { ram | rom };
```

パラメータ

`ram`	ビルドは、デバッグ用または実験的セットアップと想定され一部またはすべての変数初期化がロード時に実行されます。

rom	ビルドは、すべての変数初期化がプログラム開始時に実行される、従来の ROM ビルドになると想定されます。

説明

rom のビルドタイプを宣言し、さらにどのメモリ領域が ROM または RAM であるかを宣言すると、リンカは、異なるメモリ領域に配置された適切なセクションにのみ、より優れたチェックを実行できます。initialize ディレクティブを明示的に指定しない場合（550 ページの「*initialize ディレクティブ*」参照）、リンカは initialize by copy { rw }; を指定したような動作をします。

ram のビルドタイプを宣言すると、リンカはどのセクションタイプがどのメモリ領域に配置されているかを確認しません。

build for ディレクティブをリンカ設定ファイルに含めないと、リンカは制限されたチェックのみを実行します。これは旧バージョンとの互換性の目的には役立ちます。

関連項目

536 ページの「*define region ディレクティブ*」。

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。536 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。このディレクティブを使用して、領域に RAM または ROM メモリが含まれているかどうかを宣言できます。これは従来の ROM システムをビルドするときに役立ちます。536 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、Region 式を使用して複数の領域を表現できます。540 ページの「*Region 式*」を参照してください。

このセクションでは、メモリおよび領域の定義に固有の各リンカディレクティブについて詳しく説明します。

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（566 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数を指定。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 [ ram | rom ] region name = region-expr;
```

ここで、*region-expr* は Region 式です（539 ページの「*領域*」を参照）。

パラメータ

ram	領域に RAM メモリを含めます。

`rom`	領域に ROM メモリを含めます。
`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。いくつかの範囲は、領域の式を使用して結合できます。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。

領域が ROM または RAM になると宣言すると、従来の ROM ベースのシステムをビルドしている場合、リンカはその領域に配置されている適切なセクションのみを確認できます（534 ページの「*ディレクティブ用のビルド*」参照）。

例

```
/* 0x10000 バイトのコード領域の ROM（メモリ Mem のアドレス
   0x10000 */
define rom region ROM = [from 0x10000 size 0x10000];
```

logical ディレクティブ

構文

`logical` *`range-list`* `=` `physical` *`range-list`*

ここで、*`range-list`* は以下のいずれかです。

`[` *`region-expr`*`,...]`*`region-expr`*
`[` *`region-expr`*`,...]from` *`address-expr`*

パラメータ

`region-expr`	Region 式については、539 ページの「*領域*」も参照してください。
`address-expr`	アドレス式。

説明

`logical` ディレクティブは、論理アドレスを物理アドレスにマッピングします。ユーザアプリケーションから見えるのは論理アドレスですが、物理アドレスが通常コンテンツをメモリに読み込んだり、書き込んだりする際に使用されます。使用されている `logical` ディレクティブがない場合や、アドレスが `logical` ディレクティブの指定した範囲内にある場合は、物理アドレスは論理アドレスと同じです。

ELF 出力を生成すると、マッピングはプログラム ヘッダの物理アドレスに影響します。Intel Hex または Motorola S-records フォーマットで出力を生成するときは、物理アドレスが使用されます。

指定された順序の論理範囲リストのそれぞれのアドレスは、指定された順序で物理範囲リストの関連するアドレスにマッピングされます。

1 つまたは両方の範囲リストが from フォームで終わらない限り、論理範囲と物理範囲の合計サイズが同じでなければなりません。片方だけが from フォームで終わる場合は、from フォームで終わる方には、サイズの最後の範囲が含まれ、可能なときは合計サイズを同一にします。両方が from フォームで終わる場合は、範囲は合計サイズが一致するように一番高いアドレスに拡張します。

論理アドレスから物理アドレスにマッピングすることを設定すると、セクションやその他のコンテンツの配置の仕方に影響します。1 つの独立した論理または物理範囲に、コンテンツが重複するように配置されることはありません。また、別の論理範囲から対応する物理範囲へのマッピングがある場合、物理範囲へのマッピングが指定されていない論理範囲（論理ディレクティブで言及されていない）は配置から除外されます。

すべての logical ディレクティブはともに適用されます。同じマッピングを指定するために 1 つまたは複数のディレクティブを使用しても結果は同じです。

例

```
// Logical range 0x8000-0x8FFF maps to physical 0x10000-0x10FFF.
// No content can be placed in the logical range 0x10000-0x10FFF.
logical [from  0x8000 size 4K] = physical [from 0x10000 size 4K];

// Another way to specify the same mapping
logical [from 0x8000 size 4K] = physical from 0x10000;

// Logical range 0x8000-0x8FFF maps to physical 0x10000-0x10FFF.
// Logical range 0x10000-0x10FFF maps to physical 0x8000-0x8FFF.
// No logical range is excluded from placement because of
// this mapping.
logical [from  0x8000 size 4K] = physical [from 0x10000 size 4K];
logical [from 0x10000 size 4K] = physical [from  0x8000 size 4K];

// Logical range 0x1000-0x13FF maps to physical 0x8000-0x83FF.
// Logical range 0x1400-0x17FF maps to physical 0x9000-0x93FF.
// Logical range 0x1800-0x1BFF maps to physical 0xA000-0xA3FF.
// Logical range 0x1C00-0x1FFF maps to physical 0xB000-0xB3FF.
// No content can be placed in the logical ranges 0x8000-0x83FF,
// 0x9000-0x9FFF, 0xA000-0xAFFF, or 0xB000-0xBFFF.
logical [from 0x1000 size 4K] =
   physical [from 0x8000 size 1K repeat 4 displacement 4K];
```

```
// Another way to specify the same mapping.
logical [from 0x1000 to 0x13FF] = physical [from 0x8000 to
0x83FF];
logical [from 0x1400 to 0x17FF] = physical [from 0x9000 to
0x93FF];
logical [from 0x1800 to 0x1BFF] = physical [from 0xA000 to
0xA3FF];
logical [from 0x1C00 to 0x1FFF] = physical [from 0xB000 to
0xB3FF];
```

領域

*領域*は、重複しないメモリ範囲のセットです。*Region 式*は、領域の *Region 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。

Region 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（566 ページの「式」を参照）。

パラメータ

memory-name	Region 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、Region 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

Region 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 Region リテラルを作成します。これは、*バンク*または *far* 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 Region
   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

536 ページの「*define region ディレクティブ*」、540 ページの「*Region 式*」。

Region 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 536 ページの「*define region ディレクティブ*」を参照）。

region-literal は Region リテラルです（詳細は 539 ページの「*Region リテラル*」を参照）。

empty-region は空 Region です（詳細は 541 ページの「*空 Region*」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 *Region 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、*Region 式*を使用して領域を表現する必要があります。Region 式は、実際、メモリ範囲のセットにおけるセット式です。

領域式を作成するには、union (|)、intersection (&)、difference (-) の 3 つの演算子を使用できます。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要素
- `A & B`: セット A およびセット B 両方のすべての要素
- `A - B`: セット A にありセット B にないすべての要素

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 Region

構文

```
[ ]
```

説明

空 Region には、メモリ範囲は含まれません。空 Region 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

540 ページの「*Region 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 in` 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。554 ページの「*place at ディレクティブ*」および 556 ページの「*place in ディレクティブ*」を参照してください。

- メモリのリージョンの予約

  `reserve region` ディレクティブは、特定のメモリリージョンにはコンテンツを配置しないことを指定します。557 ページの「*リージョンの予約*」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。543 ページの「*define block ディレクティブ*」、548 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、アプリケーションがどのように起動されるかを制御します。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。550 ページの

「*initialize ディレクティブ*」および 553 ページの「*do not initialize ディレクティブ*」を参照してください。

- 削除される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。554 ページの「*keep ディレクティブ*」を参照してください。

- リンカが生成されたセクションのコンテンツを指定する

  define section ディレクティブは、リンクタイム だけで使用できるコンテンツと計算がある特定のセクションを作成するために使用します。

- 追加のより専門的なディレクティブ：

  use init table ディレクティブ

このセクションでは、セクションの処理に固有の各リンカディレクティブについて詳しく説明します。

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
minimum size = expr
maximum size = expr
expanding size
alignment = expr
end alignment = expr
fixed order
alphabetical 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（558 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。
minimum size	ブロックのサイズの下限を指定します。コンテンツが必要ない場合でも、ブロックはこれより大きいです。
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。
expanding size	ブロックは、配置されるメモリ範囲のすべての使用可能なスペースに展開します。
alignment	ブロックの最小アライ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メントが、最小アライメントを超える場合、そのアライメントがブロックのアライメントになります。
end alignment	ブロックの終了に最小アライメントを指定します。ブロックの終了アドレスは、通常その開始アドレスとそのサイズによって決まります。これはそのコンテンツによって異なります。ただし、このパラメータが使用されている場合は、必要に応じて終了アドレスは指定したアライメントに従って増やされます。
fixed order	指定されたオーダーにセクションを配置します。それぞれの *extended-selector* は別のネストされたブロックに追加され、これらのブロックは指定されたオーダーに保持されます。
alphabetical order	セクション名のアルファベット順にセクションを配置します。alphabetical order ブロックでは、*section-selector* パターンのみが許可されています。例えば、ネストされたブロックは許可されません。特定の alphabetical order ブロック内のすべてのセクションは、同じ種類の初期化を使用する必要があります（read-only、zero-init、copy-init、no-init、および同等のもの）。alphabetical order ブロックの個々のセクションに __section_begin などを使用することはできません。

`static base [basename]`

32 ビットモードのみ : *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。

`--rwpi` をリンクするとき、書き込み可能なデータの *basename* は、`SB` で、`--ropi_cb` をリンクするときは、読み取り専用データの *basename* は `CB` になります。

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。

expanding size のブロックは、ほとんどヒープやスタックに使用されます。

注 : 別の expanding size があるブロック内、最大サイズのブロック内、またはオーバーレイ内に expanding size のブロックを配置できません。

例

```
/* Create a block with a minimum size for the heap that will use
all remaining space in its memory range */
define block HEAP with minimum size = 4K, expanding size,
alignment = 8 { };
```

関連項目

231 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、548 ページの「*define overlay ディレクティブ*」を参照してください。

define section ディレクティブ

構文

```
define [ root ] section name
  [ with alignment = sec-align ]
{
  section-content-item...
};
```

それぞれの *section-content-item* は以下のいずれかです。

```
udata8 { data | string };
sdata8 data [ ,data ] ...;
udata16 data [ ,data ] ...;
sdata16 data [ ,data ] ...;
udata24 data [ ,data ] ...;
sdata24 data [ ,data ] ...;
udata32 data [ ,data ] ...;
sdata32 data [ ,data ] ...;
udata64 data [ ,data ] ...;
sdata64 data [ ,data ] ...;
pad_to data-align;
[ public ] label:
if-item;
```

if-item は：

```
if ( condition ) {
   section-content-item...
[} else if (condition] {
   section-content-item... ]...
[} else {
   section-content-item...]
}
```

パラメータ

name	セクション名。
sec-align	セクションのアライメント。例外。
root	オプション。root を指定すると、参照しない場合でもセクションが常に含まれます。

udata8 {*data*|*string*};	パラメータが式 (*data*) の場合は、セクションに符号なしの 1 バイトメンバを生成します。*data* 式は再配置中だけ、または値が必要な場合にのみ評価されます。*data* の値が大きすぎてバイトに収まらない場合は再配置エラーになります。可能な値範囲は 0~0xFF です。 パラメータが引用符の場合、文字列のそれぞれの文字に 1 バイトメンバを生成します。
sdata8 *data*;	符号のある 1 バイトメンバを生成する以外は udata8 *data* とします。 可能な値範囲は 0x80~0x7F です。
udata16 *data*;	符号なし 2 バイトメンバを生成する以外は sdata8 とします。可能な値範囲は 0~0xFFFF です。
sdata16 *data*;	符号付き 2 バイトメンバを生成する以外は sdata8 とします。可能な価範囲は 0x8000~0x7FFF です。
udata24 *data*;	符号なし 3 バイトメンバを生成する以外は sdata8 とします。可能な価範囲は 0~0xFFFF'FF です。
sdata24 *data*;	符号付き 3 バイトメンバを生成する以外は sdata8 とします。可能な価範囲は 0x8000'00~0x7FFF'FF です。
udata32 *data*;	符号なし 4 バイトメンバを生成する以外は sdata8 とします。可能な価範囲は 0~0xFFFF'FFFF です。
sdata32 *data*;	符号付き 4 バイトメンバを生成する以外は sdata8 とします。 可能な価範囲は 0x8000'0000~0x7FFF'FFFF です。
udata64 *data*;	符号なし 8 バイトメンバを生成する以外は sdata8 とします。可能な価範囲は 0~0xFFFF'FFFF'FFFF'FFFF です。
sdata64 *data*;	符号付き 8 バイトメンバを生成する以外は sdata8 とします。可能な価範囲は 0x8000'0000'0000'0000~0x7FFF'FFFF'FFFF'FFFF です。
pad_to *data_align*;	パッドバイトを生成して、式 *data-align* にアライメントするためにセクションの開始から現在のオフセットを作成します。

[public] *label*:	セクションの開始から現在のオフセットでのラベルを定義します。public を指定すると、ラベルがほかのプログラム モジュールに表示されます。指定しない場合は、リンカ構成ファイルのその他のデータ式にだけ表示されます。
if-item	アイテムの構成時間の選択。
condition	式。
data	式は再配置中だけ、または値が必要な場合にのみ評価されます。

説明

define section ディレクティブを使用して、アセンブラ言語または C/C++ から使用できないコンテンツのセクションを作成します。これの例は、スタック使用量解析結果、ブロックのサイズ、および再配置として存在しない算術式です。

データ式の不明な識別子は、ラベルと考えます。

注 : データ式だけがラベルスタック使用量解析結果などを使用できます。その他のすべての式は構成ファイルを読み取るとすぐに、評価されます。

例

```
define section data {
  /* 16 ビットワードのアプリケーションエントリは、
     256K 未満で 4 バイトで整列されます。*/
  udata16 __iar_program_start >> 2;
  /* プログラム エントリカテゴリの最大スタック使用量。*/
  udata16 maxstack("Application entry");
  /* DATA ブロックのサイズ */
  udata32 size(block DATA);
};
```

define overlay ディレクティブ

構文

```
define overlay name [ with param, param... ]
{
  extended-selectors;
}
[ except
 {
   section-selectors
  } ];
```

extended selectors および except 句については、558 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メントが、最小アライメントを超える場合、そのアライメントがオーバレイのアライメントになります。
`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、*overlaid* 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。

注 : 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サポートはありません。

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、アライメント要件は、アライ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。

オーバレイされたメモリ領域のコードはデバッグできません。C-SPY Debugger がどのコードをロードしているか決定できないからです。

関連項目

122 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 param, param... ]
{
  section-selectors
}
[ except
 {
   section-selectors
 } ];
```

ここで、*param* は以下のいずれかです。

```
packing = algorithm
simple ranges
complex ranges
no exclusions
```

section-selectors および except 句については、558 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。この方法は、多くの連続した同じバイトを持つデータに対して良い結果となります。

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。

auto – ILINK が各圧縮手法（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圧縮手法を選択します。ここには、デコンプレッサのサイズも含まれます。これは、initialize by copy のデフォルト手法です。

smallest – これは auto と同義です。

説明

initialize ディレクティブは、選択した各セクションをイニシャライザのデータを持つ 1 つのセクションと初期化されたデータ用の空間を持つ別のセクションに分割します。初期化されたデータ用の空間を持つセクションは元のセクション名を保持し、イニシャライザのデータを持つ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。

圧縮手法として auto（initialize by copy のデフォルト）を使用する場合、ILINK はイニシャライザに適した圧縮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。

各デコンプレッサには 2 つの派生形 : 単一のソースのみ処理できるものとより複雑なケースを処理可能なものがあります。デフォルトでは、関連するセクション配置ディレクティブで単一または複数の範囲のメモリ領域のどちらを指定するかによって、リンカはデコンプレッサの派生形を選択します。これは一般的に望ましい動作ですが、initialize ディレクティブで修飾子 with complex ranges または with simple ranges を使用することによって、どのデコンプレッサ派生形を使用するかを指定できます。また、コマンドライン

オプション --default_to_complex_ranges を使用し、initialize ディレクティブで complex ranges をデフォルトで使用するよう指示することも可能です。simple ranges デコンプレッサは通常、complex ranges 派生形よりも数百バイト小さくなります。

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。

内部の依存性のために、初期化された領域のアドレスがイニシャライザのサイズに左右される場合、イニシャライザの圧縮に失敗することがあります（エラー LP021）。これを回避するには、イニシャライザと初期化された領域をメモリの異なる部分に配置します（たとえば、イニシャライザを ROM に、初期化された領域を RAM に配置）。

パラメータ no exclusions を指定すると、セクションが除外された場合にエラーが出力されます（それが初期化に必要だからです）。no exclusions は initialize by copy（自動初期化）とのみ使用でき、initialize manually とは使用できません。

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び出して、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、548 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

103 ページの「*システム起動時の初期化*」、553 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
 } ];
```

section-selectors および except 句については、558 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 ディレクティブを使用して、システム起動コードで自動ゼロ初期化を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。

通常、これはすべてまたは一部の zeroinit セクションでゼロ初期化処理を制御する場合に役立ちます。

変数のゼロ初期化を完全に止める場合にも役立ちます。通常、これはソースで __no_init と指定された変数を自動処理しますが、IAR Systems または他のツールベンダーからの古いツールによって生成されるオブジェクトファイルとリンクした場合、一部のセクションで特別にゼロ初期化を止める必要があるかもしれません。

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

103 ページの「*システム起動時の初期化*」、550 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 | block name }
    [ , {section-selectors | block name }... ] ]
}
[ except
 {
   section-selectors
 } ];
```

selectors および except 句については、558 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、実行可能なイメージのブロック、オーバレイ、またはセクションを含めるために使用できます。そうでない場合はアプリケーションの含められた部分に参照が存在しないために、破棄されます。シンボル名と一致していない場合、このディレクティブは、関連のセクションフラグメントだけでなく、常に入力セクション全体を含めることになります。

さらに、含められたモジュールからのセクションのみが考慮されます。keep ディレクティブは、アプリケーションに追加のモジュールを含めません。

専用シンボルを含めるように定義したモジュール、またはシンボルを定義するセクション フラグメントのために、**Keep symbols** リンカオプション（またはコマンドラインの `--keep` オプション）、または `keep symbol` リンカ ディレクティブを使用します。

例

```
keep { section .keep* } except {section .keep};
```

place at ディレクティブ

構文

```
[ "name": ]
place [ noload ] at { address [ memory: ] address |
      start of region_expr [ with mirroring to mirror_address ] |
      end of region_expr [ with mirroring to mirror_address ] }

{
  extended-selectors
}
[ except
 {
   section-selectors
 } ];
```

extended selectors および except 句については、558 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	オプション。指定されている場合は、マップファイル、ログメッセージに使用され、ディレクティブから得られる ELF 出力セクション名の一部となります。（これはプレゼンテーション用です。アプリケーションで使用される名前には関係ありません。）
noload	オプション。それが指定された場合は、ディレクティブのセクションがターゲットシステムにロードされることを回避します。セクションを使用するには、その他の方法でターゲットシステムにそれらを入れる必要があります。*name* が指定された場合だけ noload を使用できます。
memory: *address*	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 Region 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 Region 式。間隔の終了位置が使用されます。
mirror_address	with mirroring to を指定すると、セクションの内容が このアドレスにミラーリングされているとみなされ、デバッグ情報とシンボルはミラーリングされた範囲に表示されますが、実際のコンテンツバイトは、with mirroring to を指定しない場合と同様に配置されます。 **注 :** この関数は、外部の（ターゲット専用）ミラーリングをサポートするためのものです。

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 Region 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。

注： with mirroring to は、start of と end of といっしょにのみ使用できます。

例

```
/* code_region のはじめで RO セクション .startup に配置 */
"START": place at start of ROM { readonly section .startup };
```

関連項目

556 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 [ noload ] in region-expr
      [ with mirroring to mirror_address ]
{
  extended-selectors
}
 [ except{
   section-selectors
 } ];
```

ここで、*region-expr* は Region 式です（539 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。558 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	オプション。指定されている場合は、マップファイル、ログメッセージに使用され、ディレクティブから得られる ELF 出力セクション名の一部となります。（これはプレゼンテーション用です。アプリケーションで使用される名前には関係ありません。）
noload	オプション。それが指定された場合は、ディレクティブのセクションがターゲットシステムにロードされることを回避します。セクションを使用するには、その他の方法でターゲットシステムにそれらを入れる必要があります。*name* が指定された場合だけ noload を使用できます。

mirror_address

with mirroring to を指定すると、セクションの内容が このアドレスにミラーリングされているとみなされ、デバッグ情報とシンボルはミラーリングされた範囲に表示されますが、実際のコンテンツバイトは、with mirroring to を指定しない場合と同様に配置されます。

注 : この関数は、外部の（ターゲット専用）ミラーリングをサポートするためのものです。

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。

注 : with mirroring to を指定すると、*region-expr* は単一の範囲の結果をださなければなりません。

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

554 ページの「*place at ディレクティブ*」。

リージョンの予約

構文

```
reserve region "name" = region-expr;
```

ここで、*region-expr* は Region 式です（539 ページの「*領域*」を参照）。

パラメータ

name　　予約済みリージョンの名前

説明

reserve region ディレクティブは、place in ディレクティブで使用されるリージョンを除外します。絶対セクション place at ディレクティブは、予約したリージョンを重複する場合は、エラーが出力されます。

予約したリージョンは重複できません。

例

```
reserve region "Stay out" = [ from 0x1000 size 0x100 ];
```

関連項目

556 ページの「*place in ディレクティブ*」。

use init table ディレクティブ

構文

```
use init table name for
{
  section-selectors
}
[ except
 {
   section-selectors
 } ];
```

section-selectors および except 句については、558 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` table 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。

このセクションでは、セクションの選択に固有の各リンカディレクティブについて詳しく説明します。

section-selectors

構文

```
[ section-selector [ , section-selector... ] ]
```

ここで、*section-selector* は以下のように指定します。

```
[ section-attribute ][ section-type ]
[ symbol symbol-name ][ section section-name ]
[ object module-spec ]
```

ここで、*section-attribute* は以下のように指定します。

```
ro [ code | data ] | rw [ code | data ] | zi
```

section-type は以下のように指定します。

```
[ preinit_array | init_array ]
```

パラメータ

section-attribute

指定された属性を持つセクションのみが選択されます。*section-attribute* は以下から構成される。

`ro|readonly`、ROM セクション用。
`rw|readwrite`、RAM セクション用。

各セクションでは、さらにセクションをコード、データで分けることができ、最終的に 4 つの主要カテゴリになります。

`ro code`、通常コード用
`ro data`、定数用
`rw code`、RAM にコピーしたコード用
`rw data`、変数用

`readwrite data` には、アプリケーションの起動時にゼロに初期化されるセクションの `zi|zeroinit` サブカテゴリがあります。

section-type	ELF セクションタイプが選択されたセクションのみ。*section-type* は、以下が可能です。 preinit_array、ELF 選択タイプ SHT_PREINIT_ARRAY のセクション。 init_array、ELF 選択タイプ SHT_INIT_ARRAY のセクション。
symbol *symbol-name*	*シンボル名*のパターンに一致する少なくとも 1 つのパブリックシンボルを定義するセクションのみが選択されます。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。
section *section-name*	*section-name* に一致するセクションのみが選択されます。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。
object *module-spec*	*module-spec* に一致するライブラリモジュールまたはオブジェクトファイルに由来するセクションのみが選択されます。*module-spec* は次の 2 つの形のどちらかです。 *module*、*objectname*(*libraryname*) の形の名前。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。*libraryname* が :sys の場合、パターンはシステムライブラリからのセクションとだけ一致します。 *filename*、オブジェクトファイル名、またはライブラリのオブジェクト。 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、セクション属性、セクションタイプ、シンボル名、セクション名、およびモジュールの名前と一致するすべてのセクションを選択します。5 つの条件のうち 4 つまでが省略可能です。

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。

注 : 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優先順位で以下の場合により具体的となります。

- 他のものと異なり、ワイルドカードを使用せずにシンボル名を指定している
- 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
- 他のものと異なり、セクションタイプを指定している
- 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
ro	ro code	セレクタ 2
symbol mysym	section foo	セレクタ 1
ro code section f*	ro section f*	セレクタ 1
section foo*	section f*	セレクタ 1
section *x	section f*	どちらも不適格
init_array	section f*	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

550 ページの「*initialize ディレクティブ*」、553 ページの「*do not initialize ディレクティブ*」、554 ページの「*keep ディレクティブ*」。

extended-selectors

構文

```
[ extended-selector [ , extended-selector...]]
```

ここで、*extended-selector* は以下のように指定します。

```
[ first | last | midway ]
                  { section-selector |
                    block name [ inline-block-def ]|
                    overlay name }
```

ここで、*inline-block-def* 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
                                           セクションの前にブロック
                                           First を配置するブロックを
                                           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

543 ページの「*define block ディレクティブ*」、548 ページの「*define overlay ディレクティブ*」、554 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。564 ページの「*define symbol ディレクティブ*」、565 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（566 ページの「*式*」を参照）。

このセクションでは、シンボルや式、数値の定義に固有の各リンカディレクティブについて詳しく説明します。

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

105 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 define exported symbol の派生型は、ディレクティブ define symbol を export symbol ディレクティブを組み合わせて使用する場合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。
- シンボル値は、生成したマップファイルにあるエントリリストの **Address** 列にリストされます。

例

```
/* シンボル my_symbol を 4 と定義 */
define symbol my_symbol = 4;
```

関連項目

565 ページの「*export ディレクティブ*」および 125 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。

注 : シンボル値は、生成したマップファイルにあるエントリリストの **Address** 列にリストされます。

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
```

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 567 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、564 ページの「*define symbol ディレクティブ*」および 353 ページの「*--config_def*」を参照してください。

func-operator は、すべての式で使用できる以下の関数のような演算子のいずれかです。

`alignDown(`*expr*`, `*align*`)`	複数の *align* の一番近い値で四捨五入された *expr* 値。*align* は 2 の二乗です。
`alignUp(`*expr*`, `*align*`)`	複数の *align* の一番近い値で繰り上げされた *expr* 値。*align* は 2 の二乗です。
`end(`*region*`)`	領域の最上位アドレス。
`isdefinedsymbol(`*name*`)`	シンボル *name* が設定されると True (`1`)、それ以外は False (`0`)。
`isempty(`*region*`)`	領域が空だと True (`1`)、それ以外は False (`0`)。
`max(`*expr* **[**`, `*expr*`...`**]** `)`	一番大きいパラメータ。
`min(`*expr* **[**`, `*expr*`...`**]** `)`	一番小さいパラメータ。
`size(`*region*`)`	領域のすべての範囲の合計サイズ。
`start(`*region*`)`	領域の最下位アドレス。

align と *expr* が式で、*region* が領域式（540 ページの「*Region 式*」参照）。

func-operator は、これらのオペランドの 1 つで、`check that` 式のブロックやオーバレイのサイズやアライメントの式、および `define section` ディレクティブのデータ式でのみ利用できます。

`imp(`*name*`)`	*name* が定数値を持つシンボルの名前である場合、この演算子はその値です。この演算子は、`#pragma public_equ` と一緒に使用できます（447 ページの「*public_equ*」を参照）アプリケーションのモジュールから値（特定の構造体タイプのサイズなど）をインポートします。
`tlsalignment()`	スレッドのローカル記憶領域のアライメント。
`tlssize()`	スレッドのローカル記憶領域のサイズ。

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、関数呼び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。

keep symbol ディレクティブ

構文

```
keep symbol name;
```

パラメータ

name	シンボルの名前。

説明

一般的にリンカは、アプリケーションで必要なシンボルのみを保存します。このディレクティブを使用して、シンボルを常に最終アプリケーションに含めるようにします。

関連項目

554 ページの「*keep ディレクティブ*」および 367 ページの「*--keep*」。

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。569 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。569 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

このセクションでは、構造化設定に固有の各リンカディレクティブについて詳しく説明します。

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

```
error "Unsupported configuration"
```

if ディレクティブ

構文

```
if (expr) {
  ディレクティブ
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

ここで、*expr* は式です（566 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` ディレクティブの選択していない部分内にあるテキストは構文にチェックされません。そのようなテキストでは、トークン化し、括弧はじめ ({) トークンが、括弧終わり (}) トークンと一致している必要があります。

例

541 ページの「*空 Region*」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 353 ページの「*--config_search*」。

セクションリファレンス

- セクションおよびブロックの概要
- セクションおよびブロックの説明

詳細については、96 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。

セクションおよびブロックの概要

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。
.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_tls$$DATA	プライマリスレッドの TLS 領域を維持します。
__iar_tls$$INITDATA	TLS 領域の初期値を保持します。
.iar.dynexit	atexit テーブルを保持します。
.iar.locale_table	選択したロケールのロケール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	割り込み要求、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。

表 42: セクションの概要

セクション	説明
.tbss	プライマリスレッドのスレッドローカルゼロ初期化された静的およびグローバル変数を維持します。
.tdata	プライマリスレッドのスレッドのスレッドローカル初期化された静的およびグローバル変数を維持します。
.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。
.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。
Veneer$$CMSE	セキュアゲートウェイベニアを保持します。

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

このセクションでは、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、100 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 120 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。

.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 219 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 120 ページの「*ヒープメモリの設定*」。

__iar_tls$$DATA

説明 2 つのセクション .tdata と .tbss を保持します。これらは、一緒になってプライマリスレッドのスレッドローカルストレージ領域を保持します。このセクションの主な用途は、それのサイズ演算子（__section_size、210 ページの「*専用セクション演算子*」参照）を使用して、スレッドのローカル記憶領域のサイズを取得します。演算子 __iar_tls$$DATA$$Align を使用してスレッド ローカル ストレージのアライメントを取得することもできます。

このセクションはリンカオプション --threaded_lib が使用される場合に、リンカによって作成されます。

関連項目 171 ページの「*マルチスレッド環境の管理*」

__iar_tls$$INITDATA

説明 スレッド ローカル ストレージの初期値を保持します。このセクションの主な使用は、スレッドが作成されたときにスレッドのスレッド ローカル ストレージにそれをコピーすることです。このセクションのサイズとスレッド ローカル ストレージ（セクション __iar_tls$$DATA）のサイズの違いは、ゼロに初期化されたスレッド ローカル ストレージのバイト数です。

このセクションはリンカオプション --threaded_lib が使用される場合に、リンカによって作成されます。

関連項目 171 ページの「*マルチスレッド環境の管理*」

.iar.dynexit

説明 終了時に行われる呼び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 121 ページの「*ATEXIT 制限の設定*」。

.iar.locale_table

説明 選択したロケールのロケール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 169 ページの「*ロケール*」。

.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び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、リセットベクタテーブルおよび例外ベクタを保持します。

メモリ配置 このセクションの場所はデバイスによって異なります。ハードウェアの製造元のマニュアルを参照してください。

IRQ_STACK

説明 IRQ 例外を処理する時に使用されるスタックを保持します。他の例外タイプ：`FIQ`、`SVC`、`ABT`、`UND` を処理するために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 228 ページの「*例外スタック*」。

.noinit

説明 静的およびグローバル `__no_init` 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。

.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 575 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 575 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。

.tbss

説明 プライマリスレッドのスレッドローカルゼロ初期化された静的およびグローバル変数を維持します。リンカオプション `--threaded_lib` が使用される場合、これらは `.bss` として扱われます。

関連項目 171 ページの「*マルチスレッド環境の管理*」。

.tdata

説明 プライマリスレッドのスレッドのスレッドローカル初期化された静的およびグローバル変数を維持します。リンカオプション `--threaded_lib` が使用される場合、これらは `.data` として扱われます。

関連項目 171 ページの「*マルチスレッド環境の管理*」。

.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 420 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 420 ページの「*__ramfunc*」。

Veneer$$CMSE

説明 このセクションは、拡張されたキーワード `__cmse_nonsecure_entry` により決定されたように、各エントリ関数用にリンカにより自動的に作成されたセキュアゲートウェイベニアを保持します。

メモリ配置 このセクションは、NSC（非セキュア呼び出し可能）メモリ領域に配置する必要があります。NSC 領域は、SAU（セキュリティ属性ユニット）または IDAU（i 定義された実行属性ユニット）を使用してプログラムできます。

SAU または IDAU のプログラム方法については、Armv8-M コアのドキュメントを参照してください。

関連項目

246 ページの「*Arm TrustZone®*」、297 ページの「*--cmse*」、413 ページの「*__cmse_nonsecure_entry*」、366 ページの「*--import_cmse_lib_out*」。

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、105 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)"
"std::ostream::flush()"
"operator <<(std::ostream &, char const *)"
"void _Sort<#*>(#*, #*, #*)"
```

スタック使用解析制御ディレクティブ

このセクションでは、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 ] : func-spec [, func-spec... ];
```

パラメータ

構文の構成要素の情報を参照 :

583 ページの「*category*」を参照してください。

583 ページの「*func-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。呼び出しグラフルートは、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、呼び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]: MyFunc10, MyFunc11;
```

関連項目

432 ページの「*call_graph_root*」。

exclude ディレクティブ

構文

```
exclude func-spec [, func-spec... ];
```

パラメータ

構文の構成要素の情報を参照 :

583 ページの「*func-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コールツリーをスタック使用の計算から除外します。

例

```
exclude MyFunc5, MyFunc6;
```

function ディレクティブ

構文

```
[ override ] function [ category ] func-spec : stack-size
[ , call-info... ];
```

パラメータ

構文の構成要素の情報を参照 :

583 ページの「*category*」を参照してください。

583 ページの「*func-spec*」を参照してください。

584 ページの「*call-info*」を参照してください。

585 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、その関数から呼び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] MyInterruptHandler: 44;
```

max recursion depth ディレクティブ

構文

```
max recursion depth func-spec : size;
```

パラメータ

構文の構成要素の情報を参照 :

583 ページの「*func-spec*」を参照してください。

585 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、その呼び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 MyFunc12: 10;
```

no calls from ディレクティブ

構文

```
no calls from module-spec to func-spec [, func-spec... ];
```

パラメータ

構文の構成要素の情報を参照 :

583 ページの「*func-spec*」を参照してください。

584 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、呼び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び出し）に関する問題を回避するためです。

これらの関数への呼び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 MyFunc13, MyFun14;
```

possible calls ディレクティブ

構文

```
possible calls calling-func : called-func [ , called-func... ];
```

パラメータ

構文の構成要素の情報を参照 :

583 ページの「*func-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び出しの宛先の完全なリストを指定します。これは、間接的な呼び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び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び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検討してください。

例

```
possible calls MyFunc7: MyFunc8, MyFunc9;
```

関数が呼び出しを一切実行しない場合、このリストは空となります。

```
possible calls MyFunc8: ;
```

関連項目 431 ページの「*calls*」。

構文の構成要素

このセクションでは、スタックの利用制御ディレクティブで使用できる構文構成要素を記載しています。

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]
```

func-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、*func-spec* 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 *func-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要素 ("test¥file.o")
- パス先頭にパス要素、その後に "..."、末尾にパス要素 ("D:¥...¥file.o")

注 : 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 func-spec [ , func-spec... ][ : stack-size ]
```

説明

1 つ以上の呼び出された関数のほか、オプションで呼び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*
（**[** stack **]** サイズ）

説明

スタックフレームのサイズを指定します。スタックは複数回指定することはできません。

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。どちらを使用しても、2 乗を示すサフィックスを任意で使用することができます (K=2^{10}、M=2^{20}、G=2^{30}、T=2^{40}、P=2^{50})。

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 ELF Relocatable Object Creator — iexe2obj — 再配置可能な ELF オブジェクトファイルを実行可能な ELF オブジェクトファイルから作成します。
- オプションの説明では、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。

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、`iarchive` 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。

呼び出し構文

アーカイブビルダの呼び出し構文は以下のとおりです。

```
iarchive [command] [libraryfile] [objectfiles] [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
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。*command* が省略されると、デフォルトでは `--create` が使用されます。コマンドラインのどこででも *command* を指定できます。
libraryfile	操作対象のライブラリファイルです。このように指定した場合、最初のオブジェクトファイルの前に表示されなければなりません。オプション `-o` を使用してライブラリファイルを指定することもできます。
objectfiles	コマンドの引数としての 1 つまたは複数のオブジェクトファイル。コマンドの中にはオブジェクトファイル引数をとらないものもあります。
options	アーカイブツールの動作を変更するオプションのコマンドラインオプション。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。
--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

詳細については、607 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 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
-f	コマンドラインを拡張します。
--f	オプションで依存関係を指定して、コマンドラインを拡張します。
--fake_time	同じタイムスタンプのライブラリファイルを生成します。
--no_bom	UTF-8 出力ファイルのバイト オーダーマークを省略します。
--output、-o	ライブラリファイルを指定。
--text_out	テキスト出力ファイルのエンコードを指定します。
--utf8_text_in	テキスト入力ファイルに UTF-8 エンコードを使用します。
--verbose、-V	実行されたすべての操作を報告します。
--version	ツールの出力をコンソールに送信し、終了します。
--vtoc	ライブラリにファイルの冗長リストを作成します。

表 45: iarchive オプションの概要

詳細については、607 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

呼び出しにより 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 (ielftool) 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 CMake 構成ファイルは、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。CMake ファイルは Microsoft Visual Studio プロジェクトとして使用されるか、Linux で使用するための makefiles の生成などに使用されます。

呼び出し構文

IAR ELF Tool の呼び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完全な 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、592 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、または関連のコマンドラインオプションの 1 つが指定された場合、別のフォーマットのイメージファイル。

表 46: ielftool のパラメータ

287 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ロウバイナリに設定します。
`--bin-multi`	複数のローバイナリファイルに出力を生成します。
`--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。
`--front_headers`	ファイルの最初のヘッダを出力します。
`--ihex`	出力ファイルのフォーマットを 32 ビットリニアな Intel 拡張 hex に設定します。
`--ihex-len`	Intel hex レコードのデータバイト数を設定します。
`--offset`	生成された出力ファイルのすべてのアドレスにオフセットを追加（または差し引き）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。

表 47: ielftool オプションの概要

コマンドラインオプション	説明
--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 のデータバイト数を設定します。
--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	出力ファイルのフォーマットを Texas Instruments TI-TXT に設定します。
--verbose、-V	実行されたすべての操作を出力します。
--version	ツールの出力をコンソールに送信し、終了します。

表 47: ielftool オプションの概要（続き）

詳細については、607 ページの「*オプションの説明*」を参照してください。

ielftool アドレス範囲の指定

最もベーシックレベルで、ielftool のアドレス範囲は、0x8000-0x87FF の 2 つの 16 進数で構成されています。これには 0x8000 と 0x87FF の両方が含まれています。

__checksum_begin-__checksum_end を使用した開始、または終了アドレスとして、処理した ELF ファイルに表示される ELF シンボルを指定できます。この範囲は、__checksum_begin シンボルのアドレス値があるバイトで始まり、__checksum_end シンボルのアドレス値があるバイトで終わります。0x40 と 0x3FD のシンボル値は 0x40-0x3FD で指定したものと同じです。

__start+3-__end+0x10 を使用したシンボル値にオフセットを追加できます。計算は 64 ビットの剰余演算で行われます。よって 0xFFFF'FFFF'FFFF'FFFF は 1 を差し引いたものと同じです。

{BLOCKNAME} を使用して処理した ELF ファイルに表示されまた .icf ファイルからブロックを指定できます。0x400 で始まり、0x535 で終わるブロックは、0x400-0x535 で指定したものと同じになります。

重複していない限り、0x800-1FFF {FARCODE_BLOCK} で分けて、いくつかのアドレス範囲を組み合わせることができます。

正当な範囲として重複がない限り、__FLASH_BASE-__FLASH_END を指定できます。

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。

呼び出し構文

ielfdumparm の呼び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

287 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
-a	文字列テーブルのセクションを除いて、すべてのセクションに対して出力を生成します。
--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。
--disasm_data	コードセクションとしてデータ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。
--f	オプションで依存関係を指定して、コマンドラインを拡張します。
--no_bom	UTF-8 出力ファイルのバイト オーダーマークを省略します。
--no_header	出力のリストヘッダの生成を無効化します。
--no_rel_section	.rel/.rela セクションのダンプを無効化します。
--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。
--no_utf8_in	IAR ELF ファイル以外を UTF-8 と想定しません。
--output、-o	出力ファイルを指定します。
--range	指定した範囲内のアドレスのみを逆アセンブルします。
--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。
--segment、-g	指定した数値のセグメントに対して出力を生成します。
--source	実行可能ファイルを逆アセンブルしたコードにソースをインクルードします。
--text_out	テキスト出力ファイルのエンコードを指定します。
--use_full_std_template_names	一部の標準 C++ テンプレートに完全な省略形のフルネームを使用します。
--utf8_text_in	テキスト入力ファイルに UTF-8 エンコードを使用します。
--version	ツールの出力をコンソールに送信し、終了します。

表 49: ielfdumparm オプションの概要

詳細については、607 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。

呼び出し構文

IAR ELF オブジェクトツール呼び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

287 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。
--f	オプションで依存関係を指定して、コマンドラインを拡張します。
--no_bom	UTF-8 出力ファイルのバイト オーダーマークを省略します。

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
--remove_file_path	ファイルシンボルからパス情報を削除します。
--remove_section	I つまたは複数のセクションを削除します。
--rename_section	セクションをリネームします。
--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。
--text_out	テキスト出力ファイルのエンコードを指定します。
--utf8_text_in	テキスト入力ファイルに UTF-8 エンコードを使用します。
--version	ツールの出力をコンソールに送信し、終了します。

表 51: iobjmanip オプションの概要（続き）

詳細については、607 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

section 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び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。

呼び出し構文

IAR Absolute Symbol Exporter の呼び出し構文は以下のとおりです。

```
isymexport [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
inputfile	実行可能 ELF ファイルの形式 ROM イメージです（リンカからの出力）。
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、600 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は、含めるシンボルを管理するために使用され、またシンボルの名前を変えるためにも使用できます。

表 52: isymexport のパラメータ

287 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。
--export_locals	ローカルのシンボルをエクスポートします。
-f	コマンドラインを拡張します。
--f	オプションで依存関係を指定して、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び出しがイメージに含まれないことを宣言します。
--no_bom	UTF-8 出力ファイルのバイト オーダーマークを省略します。
--ram_reserve_ranges	イメージが使用する RAM 内のエリアのためにシンボルを生成します。
--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。
--show_entry_as	指定された名前を持つアプリケーションのエントリポイントをエクスポートします。
--text_out	テキスト出力ファイルのエンコードを指定します。
--utf8_text_in	テキスト入力ファイルに UTF-8 エンコードを使用します。
--version	ツールの出力をコンソールに送信し、終了します。

表 53: isymexport オプションの概要

詳細については、607 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は、含めるシンボルを管理するために使用され、またシンボルの名前を変えるためにも使用できます。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 */
```

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

pattern1 を *pattern2* に名前を変更

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 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_* as Y_*_X
```

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
```

Show-root ディレクティブ

構文

```
show-root パターン
```

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 root としてマークされ、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。

isymexport で生成されたモジュールでリンクするとき、ビルドにシンボルへのリファレンスがないと表示されても、シンボルは、最終実行可能ファイルに含まれます。

例

```
/* リンクするときに含まれていることを示す myVar をエクスポート */
show-root myVar
```

Show-weak ディレクティブ

構文

```
Show-weak パターン
```

パラメータ

pattern	シンボル名に一致するパターンです。

説明

パターンと一致するシンボル名は、これが hide ディレクティブで上書きされない限り、weak シンボルとして出力ファイルに含まれます。

リンクしているときは、新しいコードに出力したシンボルと同じ名前のシンボルの定義が含まれている場合はエラーは報告されません。

注： isymexport 入力ファイル内の内部参照はすでに解決されているため、新しいコードに定義が存在しても影響を受けることはありません。

例

```
/* weak 定義として myFunc をエクスポート */
show-weak myFunc
```

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。

Es015: 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。

Es016: the entry point symbol is already exported

オプション `--show_entry_as` が、入力ファイルにすでにある名前で使用されました。

IAR ELF Relocatable Object Creator — iexe2obj

IAR ELF Relocatable Object Creator `iexe2obj` は、再配置可能な ELF オブジェクトを実行可能な ELF オブジェクトファイルから作成します。

呼び出し構文

`iexe2obj` の呼び出し構文は次のとおりです。

```
iexe2obj 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。少なくとも 1 つのオプションを指定する必要があります。607 ページの「*iexe2obj オプションの概要*」を参照してください。
inputfile	実行可能な 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結果の再配置可能な ELF オブジェクトファイル名。

表 54: iexe2obj parameters

287 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。

入力ファイルのビルド

入力ファイルは、`--rwpi`、`--ropi`、および `--ropi_cb` でコンパイルされたオブジェクトファイルを使用して、リンカオプション `--no_entry` とリンクされる必要があります。333 ページの「*--ropi_cb*」を参照してください。

ラッパを持つ関数シンボル `FUNC` は、入力ファイルをビルドするときにリンカにより保持される必要があります。これは、`FUNC` の宣言にあるキーワード `__root` を使用するか、リンカコマンドラインオプション `--keep FUNC` を使用して行うことができます。

コードおよび定数のデータ

入力ファイルには最大で出力ファイルに配置される *書き込み不可能*、*実行可能*セクションをひとつ含めることができます。実行可能なセクションを実行専用メモリに配置するには、オプション `--no_literal_pool` をコンパイルするときと、リンクするときの両方で使用する必要があります。

入力ファイルには最大で出力ファイルに配置される *書き込み不可能*、*実行不可能*セクションをひとつ含めることができます。セクションの開始アドレスは、定数ベースのアドレス `CB` として使用されます。

書き込み可能なデータ

入力ファイルには最大で出力ファイルに配置される *書き込み可能*、*実行不可能*セクションをひとつ含めることができます。セクションの開始アドレスは、静的ベースのアドレス `SB` として使用されます。

書き込み可能データのセクションには、動的な初期化が必要です。その場合、iexe2obj が、クライアントアプリケーションの動的初期化中に呼び出される関数 (__sti_routine) を作成します。これが機能するには、ラベル __init（ライブラリ rt7MQx_tl で定義された）が必要です。また入力ファイルを作成するために使用するリンカ設定ファイルには、次を含める必要があります。

```
define block INIT with alignment=4,fixed order {
  section .init_start,
  section .init_a,
  section .init_b,
  section .init_end.
};
```

リンカは、コンテンツなしの定数とセクションが混合したセクションにワーニング (Lp005) を発行します。そのワーニングがセクション .data と .bss を考慮する場合、それは無視されます。

iexe2obj オプションの概要

コマンドラインオプション	説明
--hide_symbols	入力ファイルからすべてのシンボルを非表示にします。
--keep_mode_symbols	入力ファイルから出力ファイルにモードシンボルをコピーします。
--prefix	シンボルとセクション名ののプレフィックスを設定します。
--wrap	*outputfile* のクライアントによって呼び出し可能な *inputfile* の関数シンボルのラッパ関数を生成します。

表 55: iexe2obj オプションの概要

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。

-a

構文 -a

ツール ielfdumparm

説明

このオプションは `--all --no_strtab` のショートカットとして使用します。

このオプションは、IDE では使用できません。

--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[=range]
```

パラメータ

593 ページの「*ielftool アドレス範囲の指定*」を参照してください。

ツール

```
ielftool
```

説明

出力ファイルのフォーマットをロウバイナリに設定します。バイナリフォーマットにはアドレス情報はなく、ロウバイトだけが含まれています。範囲が指定されていない場合は、出力ファイルには、ELF ファイルの内容になる一番小さいアドレスから、内容になる一番大きいアドレスのすべてのバイトが含まれています。範囲が指定されている場合は、範囲のバイトが含まれます。いずれの場合も、コンテンツがないギャップはゼロとして生成されることに注意してください。

注 : コンテンツのない範囲が指定されると。出力ファイルは作成されません。

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]

--bin-multi

構文

```
--bin-multi[=range[;range...]]
```

パラメータ

593 ページの「*ielftool アドレス範囲の指定*」を参照してください。

ツール

ielftool

説明

このオプションを使用して、1 つまたは複数のロウバイナリ出力ファイルを生成します。範囲が指定されていない場合は、ロウバイナリ出力ファイルは、ELF ファイルの内容のそれぞれの範囲に生成されます。範囲が指定されると、ロウバイナリ出力ファイルは、コンテンツのあるそれぞれの指定した範囲に生成されます。それぞれの場合も、各出力ファイルの名前には、その範囲の開始アドレスが含まれます。例えば、出力ファイルが `out.bin` と指定されて範囲 `0x0-0x1F` と `0x8000-0x8147` が出力されるとき、`out-0x0.bin` と `out-0x8000.bin` という名前のファイルが 2 つになります。

このオプションは、IDE では使用できません。

--checksum

構文

```
--checksum {symbol[{+|-}offset]|address}:size,
algorithm[:[1|2][a|m|z][W|L|Q][x][r][R][o][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 オフセットはシンボルに追加されます（またはマイナスのオフセット (-) の場合には差し引き）。+ と - を使用したアドレス式は制限された方法でサポートされます。以下に例を示します。

```
(start+7)-(end-2)
```

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（1、2、または 4）。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。以下から選択します。

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 0x1021）。デフォルトで使用されます。

crc32、CRC32（生成多項式 0x04C11DB7）。

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 指定した場合は、次から選択します。

1、1 の補数を指定します。

2、2 の補数を指定します。

a|m|z チェックサムのビット順序を逆にします。以下から選択します。

a、入力ビットを反転（それ以外作用なし）。

m、入力ビットと最終チェックサムを反転。

z、最終チェックサムを反転（それ以外作用なし）。

a と z の組み合わせは、m と同じ効果を持ちます。

`W|L|Q`　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。

`W`、繰返しごとに 16 ビットのチェックサムを計算します。

`L`、繰返しごとに 32 ビットのチェックサムを計算します。

`Q`、繰返しごとに 64 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。

入力バイトシーケンスは次のように処理されます。

- 8 ビットチェックサム ユニットサイズ byte0、byte1、byte2、byte3 など。
- 16 ビットチェックサム ユニットサイズ byte1、byte0、byte3、byte2 など。
- 32 ビットチェックサム ユニットサイズ byte3、byte2、byte1、byte0、byte7、byte6、byte5、byte4、など。
- 64 ビットチェックサム ユニットサイズ byte7、byte6、byte5、byte4、byte3、byte2、byte1、byte0、byte15、byte14 など。

注 : チェックサム ユニットサイズは、入力バイトシーケンスが実行される順序にのみ影響します。チェックサムシンボルのサイズ、多項式、初期値、プロセッサのアドレスバスの幅などには影響しません。

ほとんどのソフトウェア CRC 実装には、1 バイトのチェックサム ユニットサイズ（8 ビット）が使用されます。ソフトウェア CRC 実装が、ハードウェア CRC 実装で計算されたチェックサムと一致しなければならない場合、`W`、`L`、および `Q` パラメータが独占的に使用されます。ハードウェア CRC 実装とともに動作しない場合は、`W`、`L`、または `Q` パラメータは、異なるオーダーで入力バイトシーケンスを処理するので単に異なるチェックサムを計算します。

`x`　チェックサムのバイトオーダーを逆にします。これは、チェックサムの値にのみ影響します。

`r`　入力データのバイトオーダを逆にします。繰り返しごとのビット数が `L` または `W` パラメータを使用して設定されたのでない限り、これは何の影響ももたらしません。

R	逆順でチェックサム範囲を調べます。 例えば、範囲が 0x100-0xFFF;0x2000-0x2FFF の場合、チェックサム計算は通常は 0x100 から始まり、0xFFF までのすべてのバイトを計算していきます。続いて 0x2000 のバイトから 0x2FFF まで続いて計算します。 R パラメータを使用すると、計算は 0x2FFF から始まり 0x2000 まで戻りながら計算し、次に 0xFFF から 0x100 まで戻ります。
o	チェックサムに Rocksoft モデル仕様を出力します。
i|p	*start* の値が 0 より大きい場合に、i または p を使用します。以下から選択します。 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	*range* は、チェックサムが計算される 1 つまたは複数のメモリ範囲です。 通常、メモリ範囲が変更できる場合は、シンボルまたはブロックを使用することをお勧めします。0x8000-0x8347 などの明示的なアドレスを使用して、コードが変更する場合は、終了アドレスを新しい値に更新する必要があります。{CODE} またはコードの最後にあるシンボルを使用する場合は、--checksum コマンドを更新する必要はありません。 593 ページの「*ielftool アドレス範囲の指定*」を参照してください。

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。

次の例では、範囲の開始を指定するためにシンボルが使用されます。

```
ielftool --checksum=__checksum:2,crc16;__checksum_begin-0x8FFF
sourceFile.out destinationFile.out
```

BLOCK1 が 0x4000-0x4337 を占め、BLOCK2 が 0x8000-0x87FF を占める場合は、この例は 0x4000~0x4337 と 0x8000~0x87FF のバイトのチェックサムを計算します。

```
ielftool --checksum __checksum:2,crc16;{BLOCK1};{BLOCK2}
BlxTest.out BlxTest2.out
```

関連項目

233 ページの「*イメージの整合性を検証するチェックサム計算*」。

593 ページの「*ielftool アドレス範囲の指定*」。

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

ielfdumparm

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。
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。引数はアーカイブファイルとなることもできます。この場合、アーカイブファイルの各メンバーは別々に指定されたように処理されます。

287 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*」も参照してください。

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）および / またはアーカイブツ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。モジュールは、コマンドラインで指定した順序で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。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

287 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*」も参照してください。

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。

--disasm_data

構文

`--disasm_data`

ツール

`ielfdumparm`

説明

このコマンドを使用して、コードセクションであるかのように、ダンパーをダンプデータセクションに指示します。

このオプションは、IDE では使用できません。

--edit

構文

`--edit` *`steering_file`*

パラメータ

287 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

`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

600 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。

--export_locals

構文

```
--export_locals [=symbol_prefix]
```

パラメータ

symbol_prefix	エクスポートしたシンボルの名前のカスタムプレフィックスは、デフォルトのプレフィックス LOCAL を置き換えます。

ツール

isymexport

説明

このオプションを使用して、ROM イメージファイルからローカルシンボルと絶対シンボルをエクスポートします。エクスポートしたシンボルのデフォルト名は LOCAL_*filename_symbolname* です。オプションパラメータ *symbol_prefix* を使用して、LOCAL を自分のカスタムプレフィックスと置き換えます。

例

ROM イメージファイルからエクスポートすると、ソースファイル myFile.c のシンボル symb は、LOCAL_myFile_c_symb になります。

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。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

287 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*」も参照してください。

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

287 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、および 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。

--f

構文

```
--f filename
```

パラメータ

287 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、および 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。

ツールのコマンドラインで --dependencies を指定し、--f を使用して指定したコマンドライン拡張ファイルは依存関係を生成しますが、-f を使用して指定したものは依存関係を生成しません。

関連項目

617 ページの「*-f*」。

このオプションは、IDE では使用できません。

--fake_time

構文

```
--fake_time
```

ツール

```
iarchive
```

説明

このオプションを使用して同じタイムスタンプのライブラリファイルを生成します。使用した値は 0x5CF00000 で、おおよそ 2019 年 5 月 30 日 18:08:32（正確な時間は設定した時間により異なります）です。このオプションでは、同じオブジェクトファイルに同じライブラリを生成できます。このオプションを使用しない場合、タイムスタンプは、同じ入力ファイルから一意のライブラリファイルを生成します。

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	0x のプレフィックスを持つ 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、および 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。各アドレスは 4 バイトアライメントでなければならないので注意してください。

593 ページの「*ielftool アドレス範囲の指定*」を参照してください。

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]

--front_headers

構文 `--front_headers`

ツール `ielftool`

説明 このオプションを使用して、プログラムとセクションヘッダがファイルの最後ではなく最初にある ELF を出力します。

このオプションは、IDE では使用できません。

--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び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び出しがイメージに必要です。

関連項目

130 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]

--hide_symbols

構文

```
--hide_symbols
```

ツール

```
iexe2obj
```

説明

このオプションを使用して、入力ファイルからすべてのシンボルを非表示にします。

このオプションは、IDE では使用できません。

--ihex

構文

```
--ihex
```

ツール

```
ielftool
```

説明

出力ファイルのフォーマットを 32 ビットリニアな Intel 拡張 hex に設定します。hexadecimal text フォーマットは、Intel によって定義されています。

注 : Intel 拡張は 2^{32}-1 より大きいアドレスを示すことができません。アプリケーションにそのようなアドレスが含まれている場合は、別のフォーマットを使用する必要があります。

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] > [リンカ] > [出力コンバータ]

--ihex-len

構文

```
--ihex-len=length
```

パラメータ

length	レコードののデータバイト数。

ツール

```
ielftool
```

説明

Intel Hex レコードのデータバイト数を最大に設定します。このオプションは、`--ihex` オプションと組み合わせてのみ使用できます。デフォルトでは、Intel Hex レコードのデータバイト数は 16 です。

このオプションは、IDE では使用できません。

--keep_mode_symbols

構文

```
--keep_mode_symbols
```

ツール

`iexe2obj`

説明

このオプションを使用して、入力ファイルから出力ファイルにモードシンボルをコピーします。これは、逆アセンブラなどによって使用されます。

このオプションは、IDE では使用できません。

--no_bom

構文

```
--no_bom
```

ツール

`iarchive`、`ielfdumparm`、`iobjmanip`、および `isymexport`

説明

このオプションを使用して、UTF-8 出力ファイルを生成するときに、バイトオーダーマーク (BOM) を省略します。

関連項目

636 ページの「*--text_out*」および 279 ページの「*テキストエンコーディング*」。

このオプションは、IDE では使用できません。

--no_header

構文

```
--no_header
```

ツール

`ielfdumparm`

説明

デフォルトでは、実際のファイル内容の前に標準のリストヘッダが追加されます。このオプションを使用して、リストヘッダの出力を無効化します。

このオプションは、IDE では使用できません。

--no_rel_section

構文

```
--no_rel_section
```

ツール

```
ielfdumparm
```

説明

デフォルトでは、再配置可能なファイルのセクションの内容が出力として生成された場合、関連のセクションがあればそれもまた必ず出力に含まれます。このオプションを使用して、再配置セクションの出力を無効化します。

このオプションは、IDE では使用できません。

--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。

--no_utf8_in

構文

```
--no_utf8_in
```

ツール

```
ielfdumparm
```

説明

通常、ダンパーは IAR ツールによって生成された ELF ファイルが、UTF-8 テキストエンコードを使用するかどうかを決定し、正しい出力を生成できます。IAR ツールでなはいもので生成された ELF ファイルには、ダンパーは、このオプションが使用されてない限り、UTF-8 エンコードを想定します。この場合、エンコードは現在のシステムのデフォルトロケールに従うように想定されます。

注 : これは、文字が 7- ビット以上の ASCII がパスやシンボルなどに使用されている場合にだけ影響します。

関連項目 279 ページの「*テキストエンコーディング*」。

このオプションは、IDE では使用できません。

--offset

構文

```
--offset [-]offset
```

パラメータ

offset	生成された出力ファイルのすべてのアドレスで、オフセットが追加（または - が指定されている場合には差し引き）されます。

ツール ielftool

説明

このオプションを使用して、生成された出力ファイルの各出力レコードのアドレスにオフセットを追加または差し引きします。オプションは Motorola S-records、Intel Hex、TI-Txt、Simple-Code でのみ動作します。オプションは ELF ファイルまたはバイナリファイル（--bin アドレス情報を含まない）を生成するときには影響しません。このオプションでは、エントリーポイントを含むすべてのコンテンツは変更されません。出力フォーマットのアドレスのみです。

例

```
--offset 0x30000
```

これはすべてのアドレスに 0x30000 オフセットを追加します。結果として、アドレス 0x4000 にリンクされたコンテンツは 0x34000 に配置されます。

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 287 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。 593 ページの「*ielftool アドレス範囲の指定*」を参照してください。

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。

--prefix

構文 --prefix *prefix*

パラメータ

prefix	シンボルのプレフィックスとセクション名。

ツール iexe2obj

説明 デフォルトでは、出力ファイルのベース名は、シンボルやラッパで定義されるセクション名にプレフィックスとして使用されます。このオプションを使用して、これらのシンボルとセクション名のカスタムプレフィックスを設定します。

関連項目 639 ページの「*--wrap*」。

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` と `--reserve_ranges` を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

630 ページの「*--reserve_ranges*」。

このオプションは、IDE では使用できません。

--range

構文

```
--range start-end
```

パラメータ

start-end	開始アドレスが *start* 以上、終了アドレスが *end* より小さいコードを逆アセンブルします。

ツール

ielfdumparm

説明

このオプションを使用して、実行可能ファイルのコードをダンプする範囲を指定します。

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。

注 : Raw-binary は、64 ビットアドレスとの問題はありません。

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。
objectfile1 ... objectfileN	コマンドの操作対象のオブジェクトファイルです。引数はアーカイブファイルとなることもできます。この場合、アーカイブファイルの各メンバーは別々に指定されたように処理されます。

287 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*」も参照してください。

ツール

iarchive

説明

このコマンドは、既存のライブラリにオブジェクトファイル（モジュール）の置換または追加、およびまたはファイルのアーカイブを行うときに使用します。コマンドラインで指定したモジュールにより、ライブラリ内の同一名の既存のモジュールが置換されます。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 と --ram_reserve_ranges を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

626 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。

--segment、-g

構文

```
--segment segment_number[,...]
-g segment_number[,...]
```

パラメータ

segment_number	内容が出力に含まれるセグメントの番号。

ツール

ielfdumparm

説明

このオプションを使用して、出力に含める特定のセグメント（プログラムヘッダにより示される実行可能イメージの部分）を選択します。

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。

--show_entry_as

構文

```
--show_entry_as name
```

パラメータ

name	出力ファイルのプログラムエントリーポイントに与えられる名前。

ツール

isymexport

説明

このオプションを使用して、名前 *name* の下に入力として与えられたアプリケーションのエントリーポイント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。

--silent

構文

```
--silent
```

ツール

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、ツール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、この動作を回避することができます。ツールはエラーおよびワーニングメッセージをエラー出力ストリームに送信するため、この設定に関係なくそれらは表示されます。

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。バイナリフォーマットにはアドレス情報が含まれています。

注 : シンプルコードは 2^{32}-1 より大きいアドレスを示すことができます。アプリケーションにそのようなアドレスが含まれている場合は、より高いバージョン番号のシンプルコードファイルが生成されます。そのようなファイルは、この高いバージョンを扱うことができる、シンプルコードリーダーでのみ読み取りできます。

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] > [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]

--source

構文

```
--source
```

ツール

```
ielfdumparm
```

説明

このオプションを使用して、実行可能ファイルからコードをダンプする際、`ielftool` が各文のソースをその文のコードより前にインクルードするようにします。これが機能するためには、実行可能イメージをデバッグ情報とともにビルドし、ソースコードが元の場所でアクセス可能な状態でなければなりません。

このオプションは、IDE では使用できません。

--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。hexadecimal text フォーマットは、Motorola によって定義されています。オプション `ielftool`、`--srec-len`、および `--srec-s3only` を使用して、使用するフォーマットを変更できます。

注 : Motorola S レコードは 2^{32}-1 より大きいアドレスを示すことができません。アプリケーションにそのようなアドレスが含まれている場合は、別のフォーマットを使用する必要があります。

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］>［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

S-record のデータバイト数の最大数を設定します。このオプションは、`--srec` オプションと組み合わせてのみ使用できます。デフォルトでは、S-record のデータバイト数は 16 です。

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。iobjmanip はすべてのモジュールローカル関数、変数、およびセクションシンボルの名前も削除します。

注 : ielftool では、--strip 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、それを削除し、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。

287 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*」も参照してください。

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。

--text_out

構文

```
--text_out {utf8|utf16le|utf16be|locale}
```

パラメータ

`utf8`	UTF-8 エンコードを使用
`utf16le`	UTF-16 リトルエンディアンエンコードを使用
`utf16be`	UTF-16 ビッグエンディアンエンコードを使用
`locale`	システムのロケールエンコードを使用

ツール

`iarchive`、`ielfdumparm`、`iobjmanip`、および `isymexport`

説明

このオプションを使用して、テキスト出力ファイルを生成するときに使用するのエンコードを指定します。

リストファイルのデフォルトは、メインソースファイルと同じエンコードが使用されます。すべてのその他のテキストファイルのデフォルトは、バイトオーダーマーク (BOM) のある UTF-8 です。

BOM なしの UTF-8 エンコードでテキストを出力する場合、オプション `--no_bom` も使用できます。

関連項目

621 ページの「*--no_bom*」および 279 ページの「*テキストエンコーディング*」。

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 Texas Instruments TI–TXT に設定します。hexadecimal text フォーマットは、Texas Instruments によって定義されています。

注 : Texas Instruments TI–TXT は 2^{32}-1 より大きいアドレスを示すことができます。

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] > [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。

287 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*」も参照してください。

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。

--use_full_std_template_names

構文

```
--use_full_std_template_names
```

ツール

ielfdumparm

説明

一部の標準 C++ テンプレート名は通常、マングル化されていないシンボル名で省略されて出力で使用されます（たとえば、"std::basic_string<char、std::char_traits<char>、std_::allocator<char>>" の代わりに "std::string" というように）。このオプションを使用して、ielfdump が省略されていない形式を使用するようにします。

このオプションは、IDE では使用できません。

--utf8_text_in

構文

```
--utf8_text_in
```

ツール

iarchive、ielfdumparm、iobjmanip、および isymexport

説明

このオプションを使用して、ツールは、バイト オーダーマーク (BOM) のないテキスト入力ファイルを読み込むとき、UTF-8 エンコードを使用できることを指定します。

注 : このオプションはソースファイルには適用されません。

関連項目

279 ページの「テキストエンコーディング」。

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。

--version

構文

```
--version
```

ツール

`iarchive`、`ielfdumparm`、`ielftool`、`iobjmanip`、`isymexport`

説明

このオプションを使用して、バージョン情報をコンソールに送信してから終了するようツールに指示します。

このオプションは、IDE では使用できません。

--vtoc

構文

```
--vtoc 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。

287 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*」も参照してください。

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）の名前、サイズ、変更日時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。

--wrap

構文

`--wrap` *symbol*

パラメータ

symbol	`iexe2obj` の出力ファイルのクライアントによって呼び出し可能な関数シンボルです。

ツール

`iexe2obj`

説明

このオプションを使用して、ラッパ関数を関数シンボルに生成します。

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++ の説明
- 実装数

C++ ではなく C を使用している場合は、681 ページの「*C 規格の処理系定義の動作*」または 701 ページの「*C89 の処理系定義の動作*」をそれぞれ参照してください。

処理系定義の動作の詳細 C++ の説明

ここでは、C+17 規格と同順で各項目を説明します。各項目（括弧内）では、コンパイラ（リンカを含むなど）または 1 つまたは両方のライブラリに関連しているかどうかを示しています。各ヘディングは、処理系定義の動作を説明する ISO の章 / セクションのリファレンスで始まります。C++14 標準に関連するリファレンスがある場合は、それも各項目に記載されています。

注： IAR Systems の実装は、標準 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から除外できます。コンパイラは、C++ 標準以降の一部の追加機能がある C++17 標準に準拠します。

トピックの一覧

3.8 診断（コンパイラ）

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。

C++14 リファレンス :1.3.6

4.1 フリースタンディング実装に必要なライブラリ（C++14/C++17 ライブラリ）

IAR C/C++ コンパイラがサポートしない標準 C++ システムヘッダ IAR C/C++ の情報については、523 ページの「*C++ ヘッダファイル*」および 528 ページの「*サポートされてない C/C++ 関数*」を参照してください。

C++14 リファレンス :1.4

4.4 バイトのビット数（コンパイラ）

1 バイトには 8 ビットが含まれます。

C++14 リファレンス :1.7

4.6 対話型装置（C++14/C++17 ライブラリ）

ストリーム `stdin`、`stdout`、および `stderr` は対話型デバイスとして処理されます。

C++14 リファレンス :1.9

4.7 フリースタンディング実装内のプログラムのスレッド数（コンパイラ）

デフォルトでは、IAR システムのランタイム環境は 1 つのスレッドの実行だけをサポートしています。オプションのサードパーティの RTOS を使用すると、いくつかの実行するスレッドをサポートする場合があります。

C++14 リファレンス :1.10

4.7.2 メイン、および std::thread によって作成されたスレッドを実行するスレッドが、同時前進を保証することを提供する要件（コンパイラ）

thread システムヘッダはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

5.2、C.4.1 基本的なソース文字セットに物理ソースファイルの文字をマッピング（コンパイラ）

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、UTF-8、UTF-16、またはシステムのロケールにできます。279 ページの「*テキストエンコーディング*」を参照してください。

C++14 リファレンス :2.2

5.2 物理ソースファイルの文字（コンパイラ）

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、UTF-8、UTF-16、またはシステムのロケールにできます。279 ページの「*テキストエンコーディング*」を参照してください。

C++14 リファレンス :2.2

5.2 ソース文字セットから実行文字セットに文字を変換（コンパイラ）

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。ソースファイルに選択したエンコードによって異なります。279 ページの「*テキストエンコーディング*」を参照してください。デフォルトでは、ソース文字集合は、Raw です。

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。これらは文字定数および文字列リテラルの実行文字セットです。それらのエンコードタイプ :

実行文字集合	エンコードタイプ
L	UTF-32
u	UTF-16
U	UTF-32
u8	UTF-8
none	ソース文字集合

表 56: 実行文字集合およびそのエンコード

IAR DLIB ランタイム環境では、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。169 ページの「*ロケール*」を参照してください。

C++14 リファレンス :2.2

5.2 テンプレートの定義を配置するには、変換ユニットのソースが利用可能あることが必要（コンパイラ）

テンプレートのインスタンス化に関連したテンプレート定義を配置する場合、テンプレートを定義する変換ユニットのソースは、必要ありません。

C++14 リファレンス :2.2

5.3 実行文字セットと実行ワイド文字セット（コンパイラ）

実行文字集合のメンバ値は、ASCII 文字集合の値です。これらはソースファイル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。ソース

ファイルの文字セットは、ソースファイルに選択したエンコードで決定されます。279 ページの「*テキストエンコーディング*」を参照してください。

ワイト文字セットは、ISO/IEC 10646 で定義されたすべてのコードポイントからできています。

C++14 リファレンス :2.3

5.8 ヘッダまたは外部ソースファイルにヘッダ名をマッピング（コンパイラ）

ヘッダ名は、最も直感的な方法で外部ソースファイルに認識されマッピングされます。`#include` プリプロセッサディレクティブの両方のフォームでは、ヘッダ名を指定する文字シーケンスは、他のソース構造と全く同じ方法で認識されます。その後、外部ヘッダソースファイル名にマッピングされます。

C++14 リファレンス :2.9

5.8 q-char-sequence や an h-char-sequence の ', \, /*, または // の意味（コンパイラ）

q-char-sequence および h-char-sequence の文字は文字列リテラルとして変換されます。

C++14 リファレンス :2.9

5.13.3 マルチ文字 LITERAL の値（コンパイラ）

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。

C++14 リファレンス :2.14.3

5.13.3 実行ワイド文字セットにはない、シングル c 文字のあるワイド文字 LITERAL の値（コンパイラ）

すべての可能な c 文字には、実行ワイド文字セットの表現があります。

C++14 リファレンス :2.14.3

5.13.3 複数の文字が含まれているワイド文字 LITERAL の値（コンパイラ）

診断メッセージが発行されますが、最初の c 文字は無視されます。

C++14 リファレンス :2.14.3

5.13.3 非標準エスケープシーケンスの動作（コンパイラ）

非標準エスケープシーケンスはサポートされています。

C++14 リファレンス :2.14.3

5.13.3 関連タイプの範囲外の文字リテラルの値（コンパイラ）

タイプに合わせるために値が切り詰められます。

C++14 リファレンス :2.14.3

5.13.3 汎用文字名のエンコーディングは実行文字セットにはない（コンパイラ）

診断メッセージが発行されます。

C++14 リファレンス :2.14.3

5.13.3 文字リテラルの定義した範囲（コンパイラ）

範囲は `int` と同じです。

C++14 リファレンス :2.14.3

5.13.4 浮動小数点リテラルの大きいまたは小さい値の選択（コンパイラ）

値を測定する浮動小数点リテラルが、浮動小数点として表現できない場合は、一番近い偶数の浮動小数点の値が選択されます。

C++14 リファレンス :2.14.4

5.13.5 文字列リテラルのさまざまなタイプの連結（コンパイラ）

ISO C++ 標準で指定されている場合を除いて、異なるようにプリフィックスされた文字列リテラルのトークンは、連結できません。

C++14 リファレンス :2.14.5

6.6.1 フリースタンディング環境でメインを定義（コンパイラ）

`main` 関数は定義しなければなりません。

C++14 リファレンス :3.6.1

6.6.1 フリースタンディング環境での開始と終了（C++14/C++17 ライブラリ）

アプリケーションの開始と終了の説明については、65 ページの「*アプリケーションの実行— 概要*」および 155 ページの「*システムの起動と終了*」を参照してください。

C++14 リファレンス :3.6.1

6.6.1 Main へのパラメータ（C++14/C++17 ライブラリ）

`main` には 2 つの許可された定義のみがあります。

```
int main()
int main(int, char **)
```

C++14 リファレンス :3.6.1

6.6.1 Main のリンケージ（C++14/C++17 ライブラリ）

`main` 関数には外部リンケージがあります。

C++14 リファレンス :3.6.1

6.6.3 Main の前のスタティック変数の動的初期化（C++14/C++17 ライブラリ）

リンカオプション `--manual_dynamic_initialization` が使用されているとき以外は `main` の最初の文の前に、スタティック変数は初期化されます。

C++14 リファレンス :3.6.2

6.6.3 Entry の前のスレッドされたローカル変数の動的初期化（C++14/C++17 ライブラリ）

デフォルトでは、IAR システムのランタイム環境は 1 つのスレッドの実行だけをサポートしています。オプションのサードパーティの RTOS を使用すると、いくつかの実行するスレッドをサポートする場合があります。

リンカオプション `--threaded_lib` が使用されているとき以外は、スレッドローカル変数は、スタティック変数として扱われます。そして RTOS で初期化されます。

C++14 リファレンス :3.6.2

6.6.3 Main の前のスタティックインライン変数の動的初期化（C++14/C++17 ライブラリ）

リンカオプション `--manual_dynamic_initialization` が使用されているとき以外は、`main` の最初の文の前に、スタティックオブジェクトは初期化されます。

C++14 リファレンス :3.6.2

6.6.3 延期された動的初期化が実行されるスレッドとプログラムポイント（C++14/C++17 ライブラリ）

リンカオプション `--manual_dynamic_initialization` が使用されているとき以外、動的初期化は延期されません。

C++14 リファレンス :3.6.2

6.7 無効なポインタの使用（コンパイラ）

間接的にスローして割り当て解除関数に渡す以外の無効なポインタの他の使用は、有効なポインタのように動作します。

C++14 リファレンス :3.7.4.2

6.7.4.3 実装でのポインタの安全性の緩和と厳密さ（コンパイラ）

標準 C++ の IAR Systems の実装には、緩和されたポインタの安全性があります。

C++14 リファレンス :3.7.4.3

6.9 自明なコピー可能なタイプの値（コンパイラ）

基本的なタイプのすべての ビットは、値表示の一部です。基本的なタイプ間のパディングは、そのままコピーされます。

C++14 リファレンス :3.9

6.9.1 文字の表示と表記（コンパイラ）

通常の `char` 型は、`unsigned char` として処理されます。296 ページの「*--char_is_signed*」および 297 ページの「*--char_is_unsigned*」を参照してください。

C++14 リファレンス :3.9.1

6.9.1 拡張された符号付整数型（コンパイラ）

実装には、拡張された符号付整数型はありません。

C++14 リファレンス :3.9.1

6.9.1 浮動小数点型の値表示（コンパイラ）

398 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。

C++14 リファレンス :3.9.1

6.9.2 ポインタ型の値表示（コンパイラ）

400 ページの「*ポインタ型*」を参照してください。

C++14 リファレンス :3.9.2

6.11 アライメント（コンパイラ）

389 ページの「*アライメント*」を参照してください。

C++14 リファレンス :3.11

6.11 アライメントの追加の値（コンパイラ）

389 ページの「*アライメント*」を参照してください。

C++14 リファレンス :3.11

6.11 expression の追加の値の整列（コンパイラ）

389 ページの「*アライメント*」を参照してください。

C++14 リファレンス :3.11

7.1 オブジェクトの lvalue-to-rvalue 変換には無効なポインタを含む（コンパイラ）

変換が行われますが、ポインタ値が知ようされると何が起こるかは定義されていません。

C++14 リファレンス :4.1

7.8 符号なしから符号付に変換した結果の値（コンパイラ）

整数値が符号付の整数型に変換するときに代入先の型によって表示できない場合、値は、代入先の型のビット数に切り詰められ、代入先の方の値として再度変換されます。

C++14 リファレンス :4.7

7.9 不正確な浮動小数点変換の結果（コンパイラ）

浮動小数点値が異なる浮動小数点型の値に変換され、値が代入先の型の範囲内でも正確に表示されないとき、デフォルトで値は一番近い浮動小数点の値に四捨五入されます。

C++14 リファレンス :4.8

7.10 不正確な整数から浮動小数点への変換の結果値（コンパイラ）

整数値が浮動小数点型の値に変換され、値が代入先の型の範囲内でも正確に表示されないとき、デフォルトで値は一番近い浮動小数点の値に四捨五入されます。

C++14 リファレンス :4.9

7.15 拡張符号付整数型のランク（コンパイラ）

実装には拡張した符号付整数型はありません。

C++14 リファレンス :4.13

8.2.2 点リーダのクラスタイプの引数の通過（コンパイラ）

結果は診断で、その後自明なコピー可能なオブジェクトとして扱われます。

C++14 リファレンス :5.2.2

8.2.2 呼び出し元が返す、または直近の完全な式の終わりではパラメータの有効期限が終了する（コンパイラ）

呼び出し元が返すと、パラメータの有効期限は終了します。

C++14 リファレンス :5.2.2

8.2.6 そのインクリメントされた値を表示できないビットフィールドの値（コンパイラ）

値は、正しいビット数に切り捨てられます。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

8.2.8 Typeid の派生した型（C++14/C++17 ライブラリ）

`typeid` 式のタイプは、動的タイプ `std::type_info` のある式です。

C++14 リファレンス :5.2.8

8.2.10 ポインタから整数への変換（コンパイラ）

401 ページの「キャスト」を参照してください。

C++14 リファレンス :5.2.10

8.2.10 整数からポインタへの変換（コンパイラ）

401 ページの「キャスト」を参照してください。

C++14 リファレンス :5.2.10

8.2.10 関数ポインタからオブジェクトポインタ、またはその逆の変換（コンパイラ）

401 ページの「キャスト」を参照してください。

C++14 リファレンス :5.2.10

8.3.3 char、符号付 char、符号なし char 以外の基本的なタイプに適用した sizeof（コンパイラ）

391 ページの「*基本データ型整数型*」、398 ページの「*基本データ型浮動小数点数型*」、400 ページの「*ポインタ型*」を参照。

C++14 リファレンス :5.3.3

8.3.4, 21.6.3.2 割り当てられたオブジェクトの最大サイズ（C++14/C++17 ライブラリ）

割り当てられたオブジェクトの最大サイズは、理論的に `size_t` の最大値ですが、実際にはヒープに割り当てられたメモリ量によって制限されます。
120 ページの「*ヒープメモリの設定*」を参照してください。

C++14 リファレンス :5.3.4

8.7, 21.2.4 ptrdiff_t のタイプ（コンパイラ）

401 ページの「*ptrdiff_t*」を参照してください。

C++14 リファレンス :5.7、18.2

8.8 負の値の右シフトの結果（コンパイラ）

フォーム `E1 >> E2` のビットごとの右シフト操作には、`E1` が符号付のタイプで負の値がある場合、`E1` が `-1` であるとき以外は、結果の値は商 `E1/(2**E2)` の構成要素です。

C++14 リファレンス :5.8

8.18 ビットフィールドの値に割り当てられた値が表示できない（コンパイラ）

値は、正しいビット数に切り捨てられます。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

10 属性宣言の意味（コンパイラ）

C++ 標準で指定されている属性以外はサポートしていません。サポートしている属性と、オブジェクトとの使用方法については、407 ページの「*拡張キーワード*」を参照してください。

C++14 リファレンス :7

10.1.7.1volatile 修飾型のあるオブジェクトへのアクセス（コンパイラ）

404 ページの「*オブジェクトの volatile 宣言*」を参照してください。

C++14 リファレンス :7.1.6.1

10.2 列挙の根底型（コンパイラ）

392 ページの「*enum 型*」を参照してください。

C++14 リファレンス :7.2

10.4 asm 宣言の意味（コンパイラ）

`asm` 宣言は、アセンブラ命令を直接使用できるようになります。

C++14 リファレンス :7.4

10.5 指定子の動作（コンパイラ）

文字列リテラル `"C"` と `"C++"` のみが、リンケージ指示子で使用できます。

C++14 リファレンス :7.5

10.5 他の言語へのオブジェクトのリンケージ（コンパイラ）

`"C"` リンケージがあるはずです。

C++14 リファレンス :7.5

10.6.1 非標準属性の動作（コンパイラ）

C++ 標準で指定されている属性以外はサポートしていません。サポートしている属性のリストと、オブジェクトとの使用方法については、407 ページの「*拡張キーワード*」を参照してください。

C++14 リファレンス :7.6.1

11.4.1 __func__ からの文字列の結果（コンパイラ）

`__func__` の値は C++ 関数名です。

C++14 リファレンス :8.4.1

11.6 ビットフィールドの値に割り当てられた値が表示できない（コンパイラ）

値は、正しいビット数に切り捨てられます。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

12.2.4 クラスオブジェクト内のビットフィールドの配置（コンパイラ）

393 ページの「*ビットフィールド*」を参照してください。

C++14 リファレンス :9.6

17 テンプレートでのリンケージ指示子の動作（コンパイラ）

文字列リテラル `"C"` と `"C++"` のみが、リンケージ指示子で使用できます。

C++14 リファレンス :14

17.7.1 再帰的なテンプレートのインスタンス化の最大深さ（コンパイラ）

デフォルトの最大深さは 64 です。変更するには、コンパイラオプション329 ページの「*--pending_instantiations*」を使用します。

C++14 リファレンス :14.7.1

18.3, 18.5.1 std::terminate() 呼び出し前のスタックの巻き戻し（C++14/C++17 ライブラリ）

適切なキャッチハンドラが見つからないときは、`std::terminate()` を呼び出す前に、スタックは巻き戻されません。

C++14 リファレンス :15.3、15.5.1

18.5.1 予期しない仕様が攻撃されると、std::terminate() を呼び出す前にスタックが巻き戻されない（C++14/C++17 ライブラリ）

noexcept 仕様が攻撃されると、std::terminate() を呼び出す前にスタックは巻き戻されません。

C++14 リファレンス :15.5.1

19 プリプロセッサディレクティブの追加されたサポートフォーム（コンパイラ）

プリプロセッサディレクティブ #warning および #include_next がサポートされています。517 ページの「*# 警告*」および 516 ページの「*#include_next*」を参照してください。

C++14 リファレンス :16

19.1 #if ディレクティブにある文字リテラルの数字の値（コンパイラ）

#if と #elif プリプロセッサディレクティブの文字リテラルの数字の値は、別の式の値と一致します。

C++14 リファレンス :16.1

19.1 プリプロセッサの文字リテラルの数字の値（コンパイラ）

通常の char 型は、unsigned char として処理されます。296 ページの「*--char_is_signed*」および 297 ページの「*--char_is_unsigned*」を参照してください。char は符号付き文字文字として扱われ、#if と #elif プリプロセッサディレクティブの文字リテラルは、負になります。

C++14 リファレンス :16.1

19.2 < > ヘッダの検索場所（コンパイラ）

275 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。

C++14 リファレンス :16.2

19.2 インクルードソースファイルの検索手順（コンパイラ）

275 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。

C++14 リファレンス :16.2

19.2 "" ヘッダの検索場所（コンパイラ）

275 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。

C++14 リファレンス :16.2

19.2 ヘッダの検索した場所のシーケンス（コンパイラ）

275 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。

C++14 リファレンス :16.2

19.2#include ディレクティブのネスト制限（コンパイラ）

利用可能なメモリ容量に制限を設定します。

C++14 リファレンス :16.2

19.6 #pragma（コンパイラ）

689 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。

C++14 リファレンス :16.6

19.8、C.1.10__STDC__ の定義と意味（コンパイラ）

`__STDC__` は事前に 1 に設定されています。

C++14 リファレンス :16.8

19.8 変換データが利用できない時の __DATE__ のテキスト（コンパイラ）

変換データは常に使用できます。

C++14 リファレンス :16.8

19.8 変換時刻がない時の __TIME__ のテキスト（コンパイラ）

変換時刻は常に使用できます。

C++14 リファレンス :16.8

19.8 __STDC_VERSION__ の定義と意味（コンパイラ）

`__STDC_VERSION__` は事前に `201710L` に事前定義されています。

C++14 リファレンス :16.8

20.5.1.2 C++ ヘッダがインクルードされて時（C++17 ライブラリ）の C 標準ライブラリの Annex K からの関数の宣言

529 ページの「*C 境界チェックインターフェース*」を参照してください。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

20.5.1.3 フリースタンディング実装のヘッダ（C++14/C++17 ライブラリ）

521 ページの「*DLIB ランタイム環境 — 実装の詳細*」を参照してください。

C++14 リファレンス :17.6.1.3

20.5.2.3 標準 C ライブラリからの名前のリンケージ（C++14/C++17 ライブラリ）

C ライブラリからの宣言には `"C"` リンケージがあります。

C++14 リファレンス :17.6.2.3

20.5.5.8 再帰的に再入力できる標準 C++ ライブラリの関数（C++14/C++17 ライブラリ）

関数は、指定した限り ISO C++ 標準により再帰的に再入力できます。

C++14 リファレンス :17.6.5.8

20.5.5.12 例外仕様のない標準ライブラリ関数によりスローされた例外（C++14/C++17 ライブラリ）

これらの関数は、追加例外をスローしません。

C++14 リファレンス :17.6.5.12

20.5.5.14 オペレーティングシステム外からのエラー error_category（C++14/C++17 ライブラリ）

追加エラーカテゴリはありません。

C++14 リファレンス :17.6.5.14

21.2.3, C.5.2.7 NULL の定義（C++14/C++17 ライブラリ）

`NULL` は `0` として事前に定義されています。

C++14 リファレンス :18.2

21.2.4 ptrdiff_t のタイプ（コンパイラ）

401 ページの「*ptrdiff_t*」を参照してください。

C++14 リファレンス :18.2

21.2.4 size_t のタイプ（コンパイラ）

401 ページの「*size_t*」を参照してください。

C++14 リファレンス :18.2

21.2.4 ptrdiff_t のタイプ（コンパイラ）

650 ページの「*8.7, 21.2.4 ptrdiff_t のタイプ（コンパイラ）*」を参照してください。

21.5 Exit ステータス（C++14/C++17 ライブラリ）

制御は `__exit` ライブラリ関数に返されます。162 ページの「*__exit*」を参照してください。

C++14 リファレンス :18.5

21.6.3.1 bad_alloc::what の戻り値（C++14/C++17 ライブラリ）

戻り値は `"bad allocation"` へのポインタです。

C++14 リファレンス :18.6.2.1

21.6.3.2 bad_array_new_length::what の戻り値（C++14/C++17 ライブラリ）

C++17: 戻り値は `"bad array new length"` へのポインタです。C++14: 戻り値は `"bad allocation"` へのポインタです。

C++14 リファレンス :18.6.2.2

21.6.3.2 割り当てられたオブジェクトの最大サイズ（C++14/C++17 ライブラリ）

650 ページの「*8.3.4, 21.6.3.2 割り当てられたオブジェクトの最大サイズ（C++14/C++17 ライブラリ）*」を参照してください。

21.7.2 type_info::name() の戻り値（C++14/C++17 ライブラリ）

戻り値はタイプの名前が含まれている C 文字列へのポインタです。

C++14 リファレンス :18.7.1

21.7.3 Bad_cast::what の戻り値（C++14/C++17 ライブラリ）

戻り値は `"bad cast"` へのポインタです。

C++14 リファレンス :18.7.2

21.7.4 bad_typeid::what の戻り値（C++14/C++17 ライブラリ）

戻り値は `"bad typeid"` へのポインタです。

C++14 リファレンス :18.7.3

21.8.2 exception::what の戻り値（C++14/C++17 ライブラリ）

C++17: 戻り値は `"unknown"` へのポインタです。C++14: 戻り値は `std::exception` へのポインタです。

C++14 リファレンス :18.8.1

21.8.3 Bad_exception::what の戻り値（C++14/C++17 ライブラリ）

戻り値は `"bad exception"` へのポインタです。

C++14 リファレンス :18.8.2

21.10 シングルハンドラとしての非 POF 関数の使用（C++14/C++17 ライブラリ）

Non-Plan Old Function(POF) は、発見されなかった例外がハンドラにスローされる場合や、シングルハンドラの実行が未定義の動作をトリガしない場合、シングルハンドラとして使用できます。

C++14 リファレンス :18.10

23.6.5 bad_optional_access::what の戻り値（C++17 ライブラリ）

戻り値は `bad_optional_access` へのポインタです。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.7.3 オーバーアラインされた型の派生サポート（C++17 ライブラリ）

`variant` はオーバーアラインされた型をサポートします。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.7.11 bad_variant_access::what の戻り値（C++17 ライブラリ）

戻り値は `bad_variant_access` へのポインタです。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.8.2 bad_any_access::what の戻り値（C++17 ライブラリ）

戻り値は `"bad any cast"` へのポインタです。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.10 4 実装に緩和されたポインタの安全性があるときのget_pointer_safety のリターンは pointer_safety::relaxed または pointer_safety::preferred（C++14/C++17 ライブラリ）

関数 `get_pointer_safety` は常に `std::pointer_safety::relaxed` を返します。

C++14 リファレンス :20.7.4

23.11.2.1 bad_weak_ptr::what の戻り値（C++17 ライブラリ）

戻り値は `bad_weak_ptr` へのポインタです。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.11.2.2.1 shared_ptr コンストラクタが失敗する時の例外タイプ（C++14/C++17 ライブラリ）

`std::bad_alloc` のみがスローされます。

C++14 リファレンス :20.8.2.2.1

23.12.5.2 プールで直接満たした最も大きい割り当てを設定する、サポートされている最も大きい値（C++17 ライブラリ）

プールリソースオブジェクトはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.12.5.2 プールを再度満たすブロックの最大数を設定する、サポートされている最も大きい値（C++17 ライブラリ）

プールリソースオブジェクトはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.12.5.4 プールのデフォルト設定（C++17 ライブラリ）

プールリソースオブジェクトはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.12.6.1 monotonic_buffer_resource のデフォルトのnext_buffer_size（C++17 ライブラリ）

`monotonic_buffer_resource` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.12.6.2 monotonic_buffer_resource の成長係数（C++17 ライブラリ）

`monotonic_buffer_resource` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.14.11, 23.14.11.4 バインド式のプレースホルダー数（C++17 ライブラリ）

10 個のプレースホルダーオブジェクトがあります。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.14.11.4 プレースホルダーオブジェクトの割り当ての可能性（C++14/C++17 ライブラリ）

プレースホルダーオブジェクトは `CopyAssignable` です。

C++14 リファレンス :20.9.9.1.4

23.14.13.1.1 bad_function_call::what の戻り値（C++17 ライブラリ）

戻り値は `std::bad_function_call` へのポインタです。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.15.4.3 スカラ型には一意のオブジェクトの表示がある（C++17 ライブラリ）

すべての整数型、boolean、および文字には一意のオブジェクトの表示があります。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

23.15.7.6 拡張したアライメントのサポート（C++14/C++17 ライブラリ）

拡張したアライメントはサポートされています。

C++14 リファレンス :20.10.7.6

23.17.7.1time_t values から time_point objects オブジェクトを変換するときに必要な精度に四捨五入または切り捨てた値（C++14/C++17 ライブラリ）

`time_t` 値から `time_point` オブジェクトに変換するときに必要な精度に値は切り捨てられます。

C++14 リファレンス :20.12.7.1

23.19.3, 28.4.3 パラレルアルゴリズムによってサポートされている追加の実行ポリシー（C++17 ライブラリ）

パラレルアルゴリズムはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

24.2.3.1 streampos のタイプ（C++14/C++17 ライブラリ）

`streampos` のタイプは `std::fpos<mbstate_t>` です。

C++14 リファレンス :21.2.3.1, D.6

24.2.3.1 streamoff のタイプ（C++14/C++17 ライブラリ）

`streamoff` のタイプは `long` です。

C++14 リファレンス :21.2.3.1, D.6

24.2.3.1, 24.2.3.4 サポートされているマルチバイト文字エンコードルール（C++14/C++17 ライブラリ）

169 ページの「*ロケール*」を参照してください。

C++14 リファレンス :21.2.3.1, 21.2.3.4

24.2.3.2 u16streampos のタイプ（C++14/C++17 ライブラリ）

`u16streampos` のタイプは、`streampos` です。

C++14 リファレンス :21.2.3.2

24.2.3.2 char_traits<char16_t>::eof の戻り値（C++14/C++17 ライブラリ）

`char_traits<char16_t>::eof` の戻り値は `EOF` です。

C++14 リファレンス :21.2.3.2

24.2.3.3 u32streampos のタイプ（C++14/C++17 ライブラリ）

`u32streampos` のタイプは、`streampos` です。

C++14 リファレンス :21.2.3.3

24.2.3.3 char_traits<char32_t>::eof の戻り値（C++14/C++17 ライブラリ）

`char_traits<char32_t>::eof` の戻り値は `EOF` です。

C++14 リファレンス :21.2.3.3

24.2.3.4 wstreampos のタイプ（C++14/C++17 ライブラリ）

`wstreampos` のタイプは `streampos` です。

C++14 リファレンス :21.2.3.4

24.2.3.4 char_traits<wchar_t>::eof の戻り値（C++14/C++17 ライブラリ）

`char_traits<wchar_t>::eof` の戻り値は `EOF` です。

C++14 リファレンス :21.2.3.4

24.2.3.4 サポートされているマルチバイト文字エンコードルール（C++14/C++17 ライブラリ）

660 ページの「*24.2.3.1, 24.2.3.4 サポートされているマルチバイト文字エンコードルール（C++14/C++17 ライブラリ）*」を参照してください。

24.3.2 basic_string::const_iterator のタイプ（C++17 ライブラリ）

`basic_string::const_iterator` のタイプは `__wrap_iter<const_pointer>` です。

C++14 リファレンス C++14 の処理系定義の動作の一部ではありません。

24.3.2 basic_string::iterator のタイプ（C++17 ライブラリ）

`basic_string::iterator` のタイプは `__wrap_iter<pointer>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

24.4.2 basic_string_view::const_iterator のタイプ（C++17 ライブラリ）

`basic_string_view::const_iterator` のタイプは `T const *` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

25.3.1 グローバルまたはスレッドごとのロケールオブジェクト（C++14/C++17 ライブラリ）

アプリケーション全体に 1 つのロケールオブジェクトがあります。

C++14 リファレンス :22.3.1

25.3.1.1.1, 30.2.2 iostreams テンプレートがインスタンス化される文字タイプのセット（C++17 ライブラリ）

iostreams テンプレートは char、char16_t、char32_t、および wchar_t にインスタンス化されます。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

25.3.1.2 ロケール名（C++14/C++17 ライブラリ）

169 ページの「*ロケール*」を参照してください。

C++14 リファレンス :22.3.1.2

25.3.1.5C ロケールでの Locale::global 呼び出しの影響（C++14/C++17 ライブラリ）

名前のないロケールがある関数の呼び出しは影響しません。

C++14 リファレンス :22.3.1.5

25.3.1.5 ctype<char>::table_size の値（C++14/C++17 ライブラリ）

ctype<char>::table_size の値は 256 です。

C++14 リファレンス :25.4.1.3

25.4.5.1.2 time_get::do_get_date の追加フォーマット（C++14/C++17 ライブラリ）

time_get::do_get_date には追加フォーマットは許可されていません。

C++14 リファレンス :22.4.5.1.2

25.4.5.1.2 time_get::do_get_year および 2 桁の年数（C++14/C++17 ライブラリ）

time_get::do_get_year により 2 桁の年数が許可されます。0 ～ 68 の年は 2000 ～ 2068、69 ～ 99 は 1969 ～ 1999 の意味として解析されます。

C++14 リファレンス :22.4.5.1.2

25.4.5.3.2 C ロケールの time_put::do_put で生成された文字シーケンスのフォーマット（C++14/C++17 ライブラリ）

動作はライブラリ関数 strftime と同じです。

C++14 リファレンス :22.4.5.3.2

25.4.7.1.2 messages::do_open を呼び出すときの名前からカタログへのマッピング（C++14/C++17 ライブラリ）

この関数はカタログを開かないためマッピングは発生しません。

C++14 リファレンス :22.4.7.1.2

25.4.7.1.2 messages::do_get を呼び出すときにメッセージにマッピング（C++14/C++17 ライブラリ）

この関数はカタログを開かないためマッピングは発生しません。`dflt` が返されます。

C++14 リファレンス :22.4.7.1.2

25.4.7.1.2 messages::do_close を呼び出すときにメッセージにマッピング（C++14/C++17 ライブラリ）

カタログが開けないので関数を呼び出すことはできません。

C++14 リファレンス :22.4.7.1.2

25.4.7.1.2 メッセージカタログでリソースを制限（C++17 ライブラリ）

メッセージカタログはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.7.1 array::const_iterator のタイプ（C++14/C++17 ライブラリ）

`array::const_iterator` のタイプは `T const *` です。

C++14 リファレンス :23.3.2.1

26.3.7.1 array::iterator のタイプ（C++14/C++17 ライブラリ）

`Array::iterator` のタイプは `T *` です。

C++14 リファレンス :23.3.2.1

26.3.8.1 deque::const_iterator のタイプ（C++17 ライブラリ）

`deque::const_iterator` のタイプは `__deque_iterator<T, const_pointer, T const&, __map_const_pointer, difference_type>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.8.1 deque::iterator のタイプ（C++17 ライブラリ）

deque::iterator のタイプは __deque_iterator<T, pointer, T&, __map_pointer, difference_type> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.9.1 forward_list::const_iterator のタイプ（C++17 ライブラリ）

forward_list::const_iterator のタイプは __base::const_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.9.1 forward_list::iterator のタイプ（C++17 ライブラリ）

forward_list::iterator のタイプは __base::iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.10.1 list::const_iterator のタイプ（C++17 ライブラリ）

list::const_iterator のタイプは __list_const_iterator<value_type, __void_pointer> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.10.1 list::iterator のタイプ（C++17 ライブラリ）

list::iterator のタイプは __list_iterator<value_type, __void_pointer> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.11.1 vector::const_iterator のタイプ（C++17 ライブラリ）

vector::const_iterator のタイプは __wrap_iter<const_pointer> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.11.1 vector::iterator のタイプ（C++17 ライブラリ）

vector::iterator のタイプは __wrap_iter<pointer> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.12 vector<bool>::const_iterator のタイプ（C++17 ライブラリ）

vector<bool>::const_iterator のタイプは const_pointer です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.3.12 vector<bool>::iterator のタイプ（C++17 ライブラリ）

`vector<bool>::iterator` のタイプは `pointer` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.4.1 map::const_iterator のタイプ（C++17 ライブラリ）

`map::const_iterator` のタイプは `__map_const_iterator<typename __base::const_iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.4.1 map::iterator のタイプ（C++17 ライブラリ）

`map::iterator` のタイプは `__map_iterator<typename __base::iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.5.1 multimap::const_iterator のタイプ（C++17 ライブラリ）

`multimap::const_iterator` のタイプは `__map_const_iterator<typename __base::const_iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.5.1 multimap::iterator のタイプ（C++17 ライブラリ）

`multimap::iterator` のタイプは `__map_iterator<typename __base::iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.6.1 set::const_iterator のタイプ（C++17 ライブラリ）

`set::const_iterator` のタイプは `__base::const_iterator` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.6.1 set::iterator のタイプ（C++17 ライブラリ）

`set::iterator` のタイプは `__base::const_iterator` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.7.1 multiset::const_iterator のタイプ（C++17 ライブラリ）

`multiset::const_iterator` のタイプは `__base::const_iterator` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.4.7.1 multiset::iterator のタイプ（C++17 ライブラリ）

multiset::iterator のタイプは __base::const_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.4.1 unordered_map::const_iterator のタイプ（C++17 ライブラリ）

unordered_map::const_iterator のタイプは __hash_map_const_iterator<typename __table::const_iterator> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.4.1 unordered_map::const_local_iterator のタイプ（C++17 ライブラリ）

unordered_map::const_local_iterator のタイプは __hash_map_const_iterator<typename __table::const_local_iterator> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.4.1 unordered_map::iterator のタイプ（C++17 ライブラリ）

unordered_map::iterator のタイプは __hash_map_iterator<typename __table::iterator> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.4.1 unordered_map::local_iterator のタイプ（C++17 ライブラリ）

unordered_map::local_iterator のタイプは __hash_map_iterator<typename __table::local_iterator> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.4.2 unordered_map のデフォルトのバケット数（C++14/C++17 ライブラリ）

IAR C/C++ Compiler for Arm は、要素を挿入する前に、unordered_map のデフォルトのコンストラクションを作成します。

C++14 リファレンス :23.5.4.2

26.5.5.2 unordered_multimap のデフォルトのバケット数（C++14/C++17 ライブラリ）

IAR C/C++ Compiler for Arm は、要素を挿入する前に、unordered_multimap のデフォルトのコンストラクションを作成します。

C++14 リファレンス :23.5.5.2

26.5.6.1 unordered_set::const_iterator のタイプ（C++17 ライブラリ）

Unordered_set::const_iterator のタイプは __table::const_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.6.1 unordered_set::const_local_iterator のタイプ（C++17 ライブラリ）

unordered_set::const_local_iterator のタイプは __table::const_local_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.6.1 unordered_set::iterator のタイプ（C++17 ライブラリ）

unordered_set::iterator のタイプは __table::const_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.6.1 unordered_set::local_iterator のタイプ（C++17 ライブラリ）

unordered_set::local_iterator のタイプは __table::const_local_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.6.2 unordered_set のブロックのデフォルト数（C++14/C++17 ライブラリ）

IAR C/C++ Compiler for Arm は、要素を挿入する前に、unordered_set のデフォルトのコンストラクションを作成します。

C++14 リファレンス :23.5.6.2

26.5.7.1 unordered_multiset::const_iterator のタイプ（C++17 ライブラリ）

unordered_multiset::const_iterator のタイプは __table::const_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.7.1 unordered_multiset::const_local_iterator のタイプ（C++17 ライブラリ）

unordered_multiset::const_local_iterator タイプは __table::const_local_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.7.1 unordered_multiset::iterator のタイプ（C++17 ライブラリ）

unordered_multiset::iterator のタイプは __table::const_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.7.1 unordered_multiset::local_iterator のタイプ（C++17 ライブラリ）

unordered_multiset::local_iterator のタイプは __table::const_local_iterator です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.5.7.2 unordered_multiset のバケットのデフォルト数（C++14/C++17 ライブラリ）

IAR C/C++ Compiler for Arm は、要素を挿入する前に、unordered_multiset のデフォルトのコンストラクションを作成します。

C++14 リファレンス :23.5.7.2

26.6.5.1 unordered_multimap::const_iterator のタイプ（C++17 ライブラリ）

unordered_multimap::const_iterator のタイプは __hash_map_const_iterator<typename __table::const_iterator> です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.6.5.1 unordered_multimap::const_local_iterator のタイプ（C++17 ライブラリ）

`unordered_multimap::const_local_iterator` のタイプは `__hash_map_const_iterator<typename __table::const_local_iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.6.5.1 unordered_multimap::iterator のタイプ（C++17 ライブラリ）

`unordered_multimap::iterator` のタイプは `__hash_map_iterator<typename __table::iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

26.6.5.1 unordered_multimap::local_iterator のタイプ（C++17 ライブラリ）

`unordered_multimap::local_iterator` のタイプは `__hash_map_iterator<typename __table::local_iterator>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

28.4.3 前進はパラレルアルゴリズムの暗黙的なスレッドに保証（スレッドに定義されていない場合）（C++17 ライブラリ）

パラレルアルゴリズムはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

28.4.3 実装が定義された実行ポリシーで呼び出されたパラレル アルゴリズムの動作（C++17 ライブラリ）

パラレルアルゴリズムはサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

28.4.3 パラレルアルゴリズムによってサポートされている追加の実行ポリシー（C++17 ライブラリ）

660 ページの「*23.19.3, 28.4.3 パラレルアルゴリズムによってサポートされている追加の実行ポリシー（C++17 ライブラリ）*」を参照してください。

28.613 random_shuffle のランダム数の根底ソース（C++14/C++17 ライブラリ）

根底ソースは `rand()` です。

C++14 リファレンス :25.3.12

29.4.1 浮動小数点数のステータスを管理する <cfenv> 関数の使用（C++17 ライブラリ）

450 ページの「*STDC FENV_ACCESS*」および 398 ページの「*浮動小数点環境*」を参照してください。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.4.1 #pragma FENV_ACCESS のサポート（C++17 ライブラリ）

450 ページの「*STDC FENV_ACCESS*」を参照してください。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.5.8 pow(0,0) の値（C++17 ライブラリ）

`pow(0,0)` は、`ERANGE` を生成し、`NaN` を返します。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.5 default_random_engine のタイプ（C++17 ライブラリ）

`default_random_engine` のタイプは `linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>` です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.6 random_device constructor へのトークンパラメータの動作とデフォルト値（C++17 ライブラリ）

トークンは使用されません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.6 random_device コンストラクタが失敗するときの例外タイプ（C++17 ライブラリ）

コンストラクタは失敗しません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.6 random_device::operator() が失敗するときの例外タイプ（C++17 ライブラリ）

`operator()` は失敗しません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.6 random_device::operator() が値を生成する方法（C++17 ライブラリ）

`random_device::operator()` は、`std::rand()` を使用して値を生成します。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.8.1 標準のランダム数字の配布を生成するために使用されたアルゴリズム（C++17 ライブラリ）

線形合同エンジンは、標準のランダム数字の配布を生成します。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.6.9 rand() およびデータレースの概要（C++17 ライブラリ）

`rand()` はデータレースを引き起こしません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.1 n>=128 or m>=128 での関連の Laguerre 多項式の呼び出しの影響（C++17 ライブラリ）

`cmath assoc_laguerre` 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.2 l>=128 での関連の Laguerre 多項式の呼び出しの影響（C++17 ライブラリ）

`cmath assoc_legendre` 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.7 nu>=128 での標準の変更されたシリンダー状の Bessel 関数の呼び出しの影響（C++17 ライブラリ）

`cyl_bessel_i` 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.8 nu>=128 での、最初の種類シリンダー状の Bessel 関数の呼び出しの影響（C++17 ライブラリ）

`cyl_bessel_j` 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.9 nu>=128 での、非標準の変更されたシリンダー状の Bessel 関数の呼び出しの影響（C++17 ライブラリ）

cyl_bessel_k 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.10 nu>=128 でのシリンダー状の Neumann 関数の呼び出しの影響（C++17 ライブラリ）

cyl_neumann 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.15 n>=128 での Hermite 多項式の呼び出しの影響（C++17 ライブラリ）

hermite 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.16 n>=128 での Laguerre 多項式の呼び出しの影響（C++17 ライブラリ）

laguerre 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.17 l>=128 での Legendre 多項式の呼び出しの影響（C++17 ライブラリ）

legendre 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.19 n>=128 での球状の Bessel 関数の呼び出しの影響（C++17 ライブラリ）

sph_bessel 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.20 l>=128 での球状の関連した Legendre 関数の呼び出しの影響（C++17 ライブラリ）

sph_legendre 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

29.9.5.21 n>=128 での球状の Neumann 関数の呼び出しの影響（C++17 ライブラリ）

sph_neumann 関数はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.2.2 traits::pos_type が streampos でない、または traits::off_type が streamoff でないときの iostream クラスの動作（C++14/C++17 ライブラリ）

この場合特定の動作は実装されません。

C++14 リファレンス :27.2.2

30.2.2 iostreams テンプレートがインスタンス化される文字タイプのセット（C++17 ライブラリ）

662 ページの「*25.3.1.1.1, 30.2.2 iostreams テンプレートがインスタンス化される文字タイプのセット（C++17 ライブラリ）*」を参照してください。

30.5.3.4 標準ストリームでの入出力操作の後の ios_base::sync_with_stdio の呼び出しの影響（C++14/C++17 ライブラリ）

前回の入出力は特別な方法では処理されていません。

C++14 リファレンス :27.5.3.4

30.5.5.4 basic_ios::failure を生成する引数値（C++14/C++17 ライブラリ）

basic_ios::clear が例外をスローすると、badbit/failbit/eofbit セットで構成されたタイプ basic_ios::failure の例外をスローします。

C++14 リファレンス :27.5.5.4

30.7.5.2.3 basic_ostream<charT, traits>& operator<<(nullptr_t) の NTCTS（C++17 ライブラリ）

s は nullptr です。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.8.2.1basic_stringbuf でのコンストラクタの移動およびシーケンスポインタのコピー（C++14/C++17 ライブラリ）

コンストラクタはシーケンスポインタをコピーします。

C++14 リファレンス :27.8.2.1

30.8.2.4 ゼロでない引数のある basic_streambuf::setbuf の呼び出しの影響（C++14/C++17 ライブラリ）

この関数はなにも影響しません。

C++14 リファレンス :27.8.2.4

30.9.2.1Basic_filebuf でのコンストラクタの移動およびシーケンスポインタのコピー（C++14/C++17 ライブラリ）

コンストラクタはシーケンスポインタをコピーします。

C++14 リファレンス :27.9.1.2

30.9.2.4 ゼロでない引数のある basic_filebuf::setbuf の呼び出しの影響（C++14/C++17 ライブラリ）

C++17: 提供されたバッファは `basic_filebuf` で使用されます。C++14: これは、関連のファイルで `setvbuf()` を呼び出すことで、C ストリームにバッファを提供します。問題が発生した場合は、ストリームは再度初期化されます。

C++14 リファレンス :27.9.1.5

30.9.2.4 get 領域があるときの basic_filebuf::sync を呼び出すことの影響（C++14/C++17 ライブラリ）

`get` 領域は存在できません。

C++14 リファレンス :27.9.1.5

30.10.2.2 実装によりオペレーションシステムは異なる（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.6 filesystem trivial clock のタイプ（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.8.1 サポートしているルート名とオペレーティングシステム依存ルート名（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.8.2.1 ルートディレクトリの dot-dot の意味（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.10.1 path::auto_format のパスの文字シーケンスフォーマットの変換（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.10.4 追加のファイルタイプをサポートするファイルシステムの追加の file_type 列挙子（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.13 ディレクトリのようなファイルのタイプ（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.15.3 filesystem::copy の影響（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.15.14 filesystem::file_size の結果（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

30.10.15.35 filesystem::status のファイル引数のファイルタイプ（C++17 ライブラリ）

システムヘッダ `filesystem` はサポートされていません。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

31.5.1 syntax_option_type のタイプ（C++17 ライブラリ）

`syntax_option_type` のタイプは `enum` です。392 ページの「*enum 型*」を参照してください。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

31.5.2 regex_constants::match_flag_type のタイプ（C++17 ライブラリ）

`match_flag_type` のタイプは `enum` です。392 ページの「*enum 型*」を参照してください。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

31.5.3 regex_constants::error_type タイプ（C++14/C++17 ライブラリ）

タイプは `enum` です。392 ページの「*enum 型*」を参照してください。

C++14 リファレンス :28.5.3

32.5 さまざまな ATOMIC_...._LOCK_FREE マクロの値（C++14/C++17 ライブラリ）

アトミック処理がサポートされている場合は、これらのマクロの値は 2 になります。528 ページの「*アトミック処理*」を参照してください。

C++14 リファレンス :29.4

32.6, 32.6.1, 32.6.2, 32.6.3 アトミック型のロックフリー操作（C++17 ライブラリ）

528 ページの「*アトミック処理*」を参照してください。

C++14 リファレンス :C++14 の処理系定義の動作の一部ではありません。

33.2.3 native_handle_type と native_handle の表示と意味（C++14/C++17 ライブラリ）

`thread` システムヘッダはサポートされていません。

C++14 リファレンス :30.2.3

C.1.10__STDC__ の定義と意味（コンパイラ）

654 ページの「*19.8、C.1.10__STDC__ の定義と意味（コンパイラ）*」を参照してください。

C.4.1 基本的なソース文字セットに物理ソースファイルの文字をマッピング（コンパイラ）

642 ページの「*5.2、C.4.1 基本的なソース文字セットに物理ソースファイルの文字をマッピング（コンパイラ）*」を参照してください。

C.5.2.7 NULL の定義（C++14/C++17 ライブラリ）

655 ページの「*21.2.3, C.5.2.7 NULL の定義（C++14/C++17 ライブラリ）*」を参照してください。

D.9 オーバーアラインされた型のサポート（コンパイラ、C++17/C++14 ライブラリ）

オーバーアラインされた型は new 式とデフォルトの割当てでサポートされています。

C++14 リファレンス :5.3.4, 20.7.9.1, 20.7.11

実装数

C++ の IAR Systems 実装は、すべての実装と同じように、正常に実行できるアプリケーションのサイズで制限されます。

これらの制限を適用します。

C++ の機能	制限
複合文、繰返しコントロール構造、選択コントロール構造のレベルのネスト。	メモリごとにのみ制限
条件付きインクルードのレベルのネスト。	メモリごとにのみ制限
クラス、算術式、または宣言の不完全なタイプを変更するポインタ、配列、関数宣言（いろんな組み合わせ）。	メモリごとにのみ制限
完全な式内のカッコで囲まれた式のレベルをネスト。	メモリごとにのみ制限
内部識別子またはマクロ名の文字数。	メモリごとにのみ制限
外部識別子の文字数。	メモリごとにのみ制限
1 つの変換ユニットの外部識別子。	メモリごとにのみ制限

表 57: C++ 実装数

C++ の機能	制限
ブロックの宣言したブロックスコープがある識別子。	メモリごとにのみ制限
I つの変換ユニットで同時に定義されたマクロ識別子。	メモリごとにのみ制限
I つの関数定義のパラメータ。	メモリごとにのみ制限
I つの関数呼び出しの引数。	メモリごとにのみ制限
I つのマクロ定義のパラメータ。	メモリごとにのみ制限
I つのマクロ呼び出しの引数。	メモリごとにのみ制限
I つの論理ソースファイルの文字。	メモリごとにのみ制限
文字列リテラルの文字（連結後）。	メモリごとにのみ制限
オブジェクトのサイズ。	メモリごとにのみ制限
`#include` ファイルのネストレベル。	メモリごとにのみ制限
switch 文のケースラベル（ネストされた switch 文のものは除外）。	メモリごとにのみ制限
シングルクラスのデータメンバ。	メモリごとにのみ制限
シングルクラスの列挙定数。	メモリごとにのみ制限
シングルメンバ仕様のネストしたクラス定義のレベル。	メモリごとにのみ制限
`atexit` で登録された関数。	内蔵のアプリケーションのヒープメモリで制限。
`at_quick_exit` で登録された関数。	内蔵のアプリケーションのヒープメモリで制限。
直接および間接ベースクラス。	メモリごとにのみ制限
シングルクラスの直接ベースクラス。	メモリごとにのみ制限
シングルクラスの宣言されたメンバ。	メモリごとにのみ制限
クラスの仮想関数の最終上書き、アクセス可能かどうか。	メモリごとにのみ制限
クラスの直接および間接仮想ベース	メモリごとにのみ制限
クラスの固定メンバ	メモリごとにのみ制限
クラスの friend 宣言	メモリごとにのみ制限
クラスのアクセスコントロール宣言。	メモリごとにのみ制限
コンストラクタ定義のメンバイニシャライザー。	メモリごとにのみ制限
I つの識別子のスコープ修飾子。	メモリごとにのみ制限

表 57: C++ 実装数（続き）

C++ の機能	制限
ネストした外部仕様。	メモリごとにのみ制限
再帰的な `constexpr` 関数の呼び出し。	1000. この制限は、コンパイラオプション `--max_cost_constexpr_call` を使用して変更できます。
コア定数式の評価済みの完全な式。	メモリごとにのみ制限
テンプレート宣言のテンプレート引数。	メモリごとにのみ制限
テンプレートの引数の差し引き中の、置換を含むテンプレートのインスタンス化を再帰的にネスト (14.8.2)	特定のテンプレートの場合は 64。この制限は、コンパイラオプション `--pending_instantiations` を使用して変更できます。
1 回のブロック試行あたりのハンドラ数。	メモリごとにのみ制限
シングル関数宣言での Throw の仕様	メモリごとにのみ制限
プレースホルダー数 (20.9.9.1.4)	`_1` 〜 `_20` で 20 個のプレースホルダー。

表 57: C++ 実装数（続き）

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、701 ページの「*C89 の処理系定義の動作*」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、UTF-8、UTF-16、またはシステムのロケールにできます。279 ページの「*テキストエンコーディング*」を参照してください。

Main (5.1.2.1)

プログラム起動時に呼び出される関数は、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、158 ページの「*システム初期化*」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 の呼び出し直後に）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、対話型デバイスとして処理されます。

マルチスレッド環境 (5.1.2.4)

デフォルトでは、IAR システムのランタイム環境は 1 つのスレッドの実行だけをサポートしています。オプションのサードパーティの RTOS を使用すると、いくつかの実行するスレッドをサポートする場合があります。

シグナルとその意味およびデフォルトの処理 (7.14)

DLIB ランタイム環境では、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 ランタイム環境では、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 ランタイム環境では、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 (7.22.4.6)

DLIB ラインタイム環境では、`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 (7.22.4.8)

`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)

さらに、マルチバイト文字は、ソースファイルに選択したエンコードによって、識別子に表示することがあります。サポートされたマルチバイト文字は、1 つの汎用文字名 (UCN) に変換可能になります。

識別子における重要な文字 (5.2.4.1, 6.4.2)

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。これらはソースファイル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。ソースファイルの文字セットは、ソースファイルに選択したエンコードで決定されます。279 ページの「*テキストエンコーディング*」を参照してください。

英数字のエスケープシーケンス (5.2.2)

標準の英数字エスケープシーケンスの値は、`¥a-7`、`¥b-8`、`¥f-12`、`¥n-10`、`¥r-13`、`¥t-9`、`¥v-11` です。

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。296 ページの「*--char_is_signed*」および 297 ページの「*--char_is_unsigned*」を参照してください。

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。ソースファイルに選択したエンコードによって異なります。279 ページの「*テキストエンコーディング*」を参照してください。デフォルトでは、ソース文字集合は、Raw です。

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。これらは文字定数および文字列リテラルの実行文字集合です。それらのエンコードタイプ :

実行文字集合	エンコードタイプ
L	UTF-32
u	UTF-16
U	UTF-32
u8	UTF-8
none	ソース文字集合

表 58: 実行文字集合およびそのエンコード

IAR DLIB ランタイム環境では、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。169 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)

684 ページの「*ソースおよび実行文字集合 (6.4.4.4, 5.1.1.2)*」を参照してください。

ワイド文字列リテラルを異なるエンコードタイプと連結 (6.4.5)

異なるエンコードタイプをワイド文字列リテラルと連結できません。

ワイド文字列リテラルに使用されるロケール (6.4.5)

684 ページの「*ソースおよび実行文字集合 (6.4.4.4, 5.1.1.2)*」を参照してください。

実行可能文字としてのソース文字 (6.4.5)

すべてのソース文字は、実行可能文字として表すことができます。

Wchar_t、char16_t、および char32_t のエンコード (6.10.8.2)

`wchar_t` はエンコード UTF-32 があり、`char16_t` は、エンコード UTF-16 があり、`char32_t` は UTF-32 にエンコードされます。

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、391 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。算術右シフトとして動作する演算子 >> を除きます。

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。

浮動小数点変換の精度 (5.2.4.2.2)

浮動小数点変換の精度は不明です。

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 のデフォルトステータスは、コンパイラオプション --no_default_fp_contract を使用していない場合は ON です。

J.3.7 配列およびポインタ

ポインタからの変換・ポインタへの変換 (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、401 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、401 ページの「*ptrdiff_t*」を参照してください。

J.3.8 ヒント

register 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。91 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、393 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（307 ページの「*-e*」を参照）。

ビットフィールドのアトミック型 (6.7.2.1)

アトミック型はビットフィールドとして使用できません。

記憶単位の境界をまたぐビットフィールド (6.7.2.1)

`__attribute__((packed))`（GNU 言語拡張）が使用されない限り、ビットフィールドは常に 1 つの記憶単位にのみ配置されます。よって記憶単位の境界をまたぎません。

ビットフィールドの配置順 (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、393 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メントは、メンバ型と同じです（389 ページの「*アライメント*」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（404 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 #pragma に配置 (6.4、6.4.7)

これらのプラグマディレクティブは、指定した位置でパラメータとしてヘッダ名を取得します。

```
#pragma include_alias ("header", "header")
#pragma include_alias (<header>, <header>)
```

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。
499 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（296 ページの「*--char_is_signed*」を参照）。

角括弧で括ったファイル指定 (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、275 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。

引用符で括ったファイル指定 (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、275 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。

inserts ¥ in front of ¥u (6.10.3.2)

文字定数および文字列リテラル内で # (stringify argument) は、汎用文字名 (UCN) の前に ¥ 文字を挿入します。

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*の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は*プラグマディレクティブ*の章とここの両方でリストされる場合、*プラグマディレクティブ*の章の情報は、ここの情報を上書きします。

```
alias_def

alignment

alternate_target_def
```

```
baseaddr
basic_template_matching
building_runtime
can_instantiate
codeseg
constseg
cplusplus_neutral
cspy_support
cstat_dump
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
exception_neutral
function
function_category
function_effects
hdrstop
important_typedef
ident
implements_aspect
init_routines_only_for_needed_variables
initialization_routine
inline_template
instantiate
keep_definition
library_default_requirements
library_provides
library_requirement_override
```

```
memory
module_name
no_pch
no_vtable_use
once
pop_macro
preferred_typedef
push_macro
separate_init_routine
set_generate_entries_without_bounds
system_include
uses_aspect
vector
warnings
```

Default __DATE__ and __TIME__ (6.10.8)

`__TIME__` と `__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、133 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の表現はありません。

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、398 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、169 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。

定義域エラー (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)

2 番目の引数がゼロの場合、fmod 関数はドメインエラーに errno を設定し、浮動小数点 NaN を返します。

remainder のリターン値 (7.12.10.2)

2 番目の引数がゼロの場合、remainder はドメインエラーに errno を設定し、浮動小数点 NaN を返します。

remquo の規模 (7.12.10.3)

規模は、合同剰余 INT_MAX です。

remquo のリターン値 (7.12.10.3)

2 番目の引数がゼロの場合、remquo はドメインエラーに errno を設定し、浮動小数点 NaN を返します。

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 : `signal` のデフォルトの実装は何の処理も行いません。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。166 ページの「*signal*」および 164 ページの「*raise*」をそれぞれ参照してください。

NULL マクロ (7.19)

`NULL` マクロは、`0` に定義されています。

行を終了する文字 (7.21.2)

stdout ストリーム関数は、`newline` または `end of file (EOF)` を行を終了する文字として認識します。

改行文字の前の空白文字 (7.21.2)

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への NULL 文字の追加 (7.21.2)

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 (7.21.3)

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 (7.21.3)

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。134 ページの「*入出力 (I/O) の概要*」を参照してください。

ファイルのバッファ処理 (7.21.3)

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 (7.21.3)

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 (7.21.3)

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 (7.21.3)

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 (7.21.3)

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 (7.21.4.1)

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。134 ページの「*入出力 (I/O) の概要*」を参照してください。

rename 関数 (7.21.4.2)

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。134 ページの「*入出力 (I/O) の概要*」を参照してください。

開かれた一時ファイルの削除 (7.21.4.3)

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 (7.21.5.4)

`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 (7.21.6.1, 7.29.2.1)

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 (7.21.6.1, 7.29.2.1)

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 (7.21.6.2, 7.29.2.1)

-（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 (7.21.6.2, 7.29.2.2)

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 (7.21.9.1, 7.21.9.3, 7.21.9.4)

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 (7.22.1.3, 7.29.4.1.1)

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 (7.22.1.3, 7.29.4.1.1)

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 (7.22.3)

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 (7.22.4.1, 7.22.4.5)

abort() または _Exit() を呼び出しても、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 (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7)

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()`、`_Exit()` と `quick_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 (7.22.4.8)

その引数が null ポインタでないとき、`system` 関数は 1 を返します。

clock_t と time_t の範囲と精度 (7.27)

`clock_t` の範囲と精度は処理によって異なります。32 ビットの `time_t` が使用される場合は、`time_t` の範囲と精度の 1 秒のティックは、19000101 ～ 20351231 です。64 ビットの `time_t` が使用される場合は、1 秒のティックは -9999 ～ 9999 年間です。530 ページの「*time.h*」を参照してください。

タイムゾーン (7.27.1)

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、530 ページの「*time.h*」を参照してください。

clock() の領域 (7.27.2.1)

`clock` 関数の領域は、処理によって異なります。

TIME_UTC epoch (7.27.2.5)

`TIME_UTC` 関数の領域は、処理によって異なります。

%Z 置換文字列 (7.27.3.5, 7.29.5.1)

デフォルトでは、":" または "" 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。167 ページの「*__time32*、*__time64*」を参照してください。

数学関数の丸めモード (F0.10)

`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, 7.20.2, 7.20.3)

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、389 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。

別のスレッドの autos またはスレッドローカルにアクセス (6.2.4)

IAR システムランタイム環境では、複数のスレッドは許可していません。サードパーティの RTOS を使用して、アクセスしたアイテムが範囲外にならない限り、意図したようにアクセスが確立され作動します。

バイトの数値、順序、エンコーディング (6.2.6.1)

389 ページの「*データ表現*」を参照してください。

拡張したアライメント (6.2.8)

拡張したアライメントの詳細については、432 ページの「*data_alignment*」を参照してください。

有効なアライメント (6.2.8)

基本的なタイプでの有効なアライメントの詳細については、*データ表現*の章を参照してください。

sizeof 演算子の結果の値 (6.5.3.4)

389 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。*エンコード*の章では、ソース文字セットのデフォルトのエンコードセットを変更する方法、および実行文字集合の通常の文字定数および通常の文字列リテラルプレイン文字のエンコードで変更する方法を説明します。

追加文字の意味 (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、次の実行文字集合のエンコードに変換されます。

実行文字集合	エンコーディング
`L` typed	UTF-32
`u` typed	UTF-16
`U` typed	UTF-32
`u8` typed	UTF-8
none typed	ソース文字集合と同じ

表 59: 拡張ソース文字集合のマルチバイト文字の変換

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)

シフト状態はサポートされていません。

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)

設定が Normal または Tiny のライブラリでは、デフォルトの小数点の文字は '.' です。設定が Full のライブラリでは、選択したロケールは小数点に使用する文字を定義します。

出力文字 (7.4, 7.30.2)

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, 7.30.2)

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, 7.30.2.1.2, 7.30.5.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11)

文字ベースの関数用の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。`wchar_t` ベースの関数用のテスト済み文字集合は、UTF-32 コードポイント `0x0` 〜 `0x7F` です。

ネイティブ環境 (7.11.1.1)

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 (7.22.1, 7.29.4.1)

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 (7.24.4.3, 7.29.4.4.2)

照合はサポートされていません。

strerror 関数によって返されるメッセージ (7.24.6.2)

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 60: strerror() が返すメッセージ— DLIB ランタイム環境

時間と日付のフォーマット (7.27.3.5, 7.29.5.1)

タイムゾーン情報は、処理しているように、低レベル関数 __getzone です。

文字マッピング (7.30.1)

文字マッピングは tolower と toupper をサポートします。

文字分類 (7.30.1)

サポートしている文字分類は、alnum、cntrl、digit、graph、lower、print、punct、space、upper、および xdigit です。

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、681 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。

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び出される関数は、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)
```

DLIB ランタイム環境におけるこの動作を変更する場合は、158 ページの「*システム初期化*」を参照してください。

対話型装置 (5.1.2.3)

ストリーム stdin および stdout は、対話型デバイスとして扱われます。

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。ソースファイルに選択したエンコードによって異なります。279 ページの「*テキストエンコーディング*」を参照してください。デフォルトでは、ソース文字集合は、Raw です。

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。これらは文字定数および文字列リテラルの実行文字集合です。それらのエンコードタイプ :

実行文字集合	エンコードタイプ
`L`	UTF-32
`u`	UTF-16
`U`	UTF-32
`u8`	UTF-8
`none`	ソース文字集合

表 61: 実行文字集合およびそのエンコード

IAR DLIB ランタイム環境では、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。169 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)

169 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、391 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。算術右シフトとして動作する演算子 `>>` を除きます。

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、IEC 60559 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、398 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、401 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、401 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、401 ページの「*ptrdiff_t*」を参照してください。

レジスタ

register 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、391 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");
```

認識され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*の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*の章とここでリストされている場合、*プラグマディレクティブ*の章で提供されている情報は、ここの情報で上書きされます。

```
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__ と __DATE__ の定義は常に使用可能です。

IAR DLIBラ ンタイム環境のライブラリ関数

注 : このリストの項目には、ファイル記述子がライブラリ構成でサポートされている場合にのみ当てはまるものもあります。ランタイムライブラリ構成の詳細は、*DLIB ランタイム環境*の章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-errno` 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 : `signal` のデフォ , ルトの実装は何の処理も行いません。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。166 ページの「*signal*」および 164 ページの「*raise*」をそれぞれ参照してください。

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。134 ページの「*入出力 (I/O) の概要*」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。134 ページの「*入出力 (I/O) の概要*」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。134 ページの「*入出力 (I/O) の概要*」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、162 ページの「*getenv*」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。167 ページの「*system*」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 62: strerror() が返すメッセージ— DLIB ランタイム環境

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、167 ページの「*__time32*、*__time64*」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。161 ページの「*clock*」を参照してください。

あ

い

う

お

か

き

く

け

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

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

記号

数字