

IAR C/C++ 開発ガイド

コンパイルおよびリンク

Advanced RISC Machines Ltds
ARM[®] コア



著作権事項

Copyright ©1999–2011 IAR Systems AB.

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

免責事項

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

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

商標

IAR Systems、IAR Embedded Workbench、C-SPY、visualSTATE、From Idea to Target、IAR KickStart Kit、IAR PowerPac、IAR YellowSuite、IAR Advanced Development Kit、IAR、および IAR Systems のロゴタイプは、IAR Systems AB が所有権を有する商標または登録商標です。J-Link は IAR Systems AB がライセンスを受けた商標です。

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

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

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

その他のすべての製品名は、その所有者の商標または登録商標です。

改版情報

第 7 版 : 2011 年 4 月

部品番号 : DARM-7-J

本ガイドは、ARM® 用 IAR Embedded Workbench® のバージョン 6.2x に適用する。

『ARM 用 IAR C/C++ 開発ガイド』は、以前のすべてのバージョンの『ARM IAR C/C++ Compiler Reference Guide』および『IAR リンカおよびライブラリ ツールリファレンスガイド』の内容に代わるものです。

内部参照 : M10、Too6.3、csrct2010.1、V_110411、ISUD。

目次（章）

表	19
はじめに	21
パート 1. ビルドツール	29
IAR ビルドツールの概要	31
組込みアプリケーションの開発	37
データ記憶	53
関数	57
ILINK を使用したリンク	67
アプリケーションのリンク	77
DLIB ランタイムライブラリ	89
アセンブラ言語インタフェース	129
C の使用	147
C++ の使用	157
アプリケーションに関する考慮事項	169
組込みアプリケーション用の効率的なコーディング	183
パート 2. リファレンス情報	203
外部インタフェースの詳細	205
コンパイラオプション	215
リンカオプション	259
データ表現	285
拡張キーワード	301

プラグマディレクティブ	315
組込み関数	333
プリプロセッサ	377
ライブラリ関数	383
リンカ設定ファイル	393
セクションリファレンス	417
IAR ユーティリティ	423
処理系定義の動作	453
索引	471

目次

表	19
はじめに	21
本ガイドの対象者	21
このガイドの使用方法	21
本ガイドの内容	22
その他のドキュメント	23
ユーザガイドおよびリファレンスガイド	24
オンラインヘルプシステムを参照	24
参考資料	25
Web サイト	26
表記規則	26
表記規則	26
命名規約	27
パート I. ビルドツール	29
IAR ビルドツールの概要	31
IAR ビルドツール — 概要	31
IAR C/C++ コンパイラ	31
IAR アセンブラ	32
IAR ILINK リンカ	32
専用 ELF ツール	32
外部ツール	32
IAR 言語の概要	33
デバイスサポート	34
サポートされている ARM デバイス	34
事前に定義されているサポートファイル	34
開発を開始するための例	34
組込みシステム用の特殊サポート	35
拡張キーワード	35
プラグマディレクティブ	35

定義済シンボル	35
特殊な関数型	35
低レベル機能へのアクセス	36
組込みアプリケーションの開発	37
IAR ビルドツールを使用した組込みソフトウェアの開発	37
内部および外部メモリのマッピング	37
周辺ユニットとの通信	38
イベント処理	38
システム起動	38
リアルタイムオペレーティングシステム	39
他のビルドツールとの相互運用	39
ビルドプロセス — 概要	40
変換プロセス	40
リンク処理	41
リンク後	42
アプリケーションの実行 — 概要	43
初期化フェーズ	44
実行フェーズ	47
終了フェーズ	47
アプリケーションのビルド概要	47
基本的なプロジェクト設定	48
プロセッサ構成	48
速度とサイズの最適化	50
ランタイム環境	50
データ記憶	53
はじめに	53
さまざまなデータ記憶方法	53
自動変数 — スタック	54
スタック	54
ヒープ上の動的メモリ	55

関数	57
関数関連の拡張	57
ARM および Thumb コード	57
RAM での実行	58
割込み、並列処理、OS 関連のプログラミング用の	
基本コマンド	59
割込み関数	59
例外関数のインストール	60
割込みおよび高速割込み	61
ネスト割込み	62
ソフトウェア割込み	63
割込み処理	64
ARM Cortex-M の割込み	65
C++ と特殊な関数型	66
ILINK を使用したリンク	67
リンクの概要	67
モジュールおよびセクション	68
リンク処理	69
コードおよびデータの配置（リンカ設定ファイル）	71
設定ファイルの簡単な例	71
システム起動時の初期化	74
初期化プロセス	75
C++ 動的初期化	76
アプリケーションのリンク	77
リンクについて	77
リンカ設定ファイルの選択	77
独自のメモリエリアの定義	78
セクションの配置	79
RAM の空間の予約	80
モジュールの保持	81
シンボルおよびセクションの保持	81
アプリケーション起動	81

スタックの設定	82
ヒープの設定	82
atexit 制限の設定	82
デフォルト初期化の変更	82
ILINK とアプリケーション間の相互処理	85
標準ライブラリの処理	86
ELF/DWARF 以外の出力フォーマットの生成	86
ベニア	86
トラブルシューティングについてのヒント	87
再配置エラー	87
DLIB ランタイムライブラリ	89
ランタイムライブラリの概要	89
ランタイムライブラリの機能	89
ランタイムライブラリの設定	90
ビルド済ライブラリ	91
ライブラリファイル名構文	92
ライブラリファイルのグループ	93
ビルド済ライブラリのカスタマイズ（リビルドなし）	94
printf、scanf のフォーマッタの選択	95
printf フォーマッタの選択	95
scanf フォーマッタの選択	96
アプリケーションデバッグサポート	97
C-SPY デバッグサポートを含める	98
ライブラリ機能のデバッグ	98
C-SPY の [ターミナル I/O] ウィンドウ	99
デバッグライブラリの低レベル関数	100
ターゲットハードウェアのライブラリの適合	101
ライブラリの低レベルインタフェース	101
ライブラリモジュールのオーバーライド	102
カスタマイズしたライブラリのビルドと使用	102
ライブラリプロジェクトのセットアップ	103
ライブラリ機能の修正	103
カスタマイズしたライブラリの使用	104

システムの起動と終了	104
システム起動	104
システム終了	107
システム初期化のカスタマイズ	108
__low_level_init	108
cstartup.s ファイルの修正	109
ライブラリ構成	109
ランタイム構成の選択	110
標準 I/O ストリーム	110
低レベルキャラクタ I/O の実装	110
printf、scanf の構成シンボル	112
フォーマット機能のカスタマイズ	113
ファイル I/O	114
ロケール	115
ビルド済ライブラリでのロケールサポート	115
ロケールサポートのカスタマイズ	115
実行中のロケール変更	116
環境の操作	117
getenv 関数	117
システム関数	118
シグナル関数	118
時間関数	118
Pow	119
assert 関数	119
Atexit	119
マルチスレッド環境の管理	120
DLIB ライブラリでのマルチスレッドのサポート	120
マルチスレッドのサポートの有効化	121
リンカ設定ファイルにおける変更	125
モジュールの整合性チェック	125
ランタイムモデル属性	126
ランタイムモデル属性の使用	126

アセンブラ言語インタフェース	129
C 言語とアセンブラの結合	129
組込み関数	129
C 言語とアセンブラモジュールの結合	130
インラインアセンブラ	131
C からのアセンブラルーチンの呼出し	133
スケルトンコードの作成	133
コードのコンパイル	134
C++ からのアセンブラルーチンの呼出し	135
呼出し規約	136
関数の宣言	137
C++ ソースコードでの C リンケージの使用	137
保護レジスタとスクラッチレジスタ	137
関数の入口	138
関数の終了	140
例	141
呼出しフレーム情報	143
CFI ディレクティブ	143
CFI サポートを持つアセンブラソースの作成	144
C の使用	147
C 言語の概要	147
拡張の概要	148
言語拡張の有効化	149
IAR C 言語拡張	150
組込みシステムプログラミングの拡張	150
C 規格に対する緩和	153
C++ の使用	157
概要 — EC++ および EEC++	157
Embedded C++	157
拡張 Embedded C++	158

概要 — 標準 C++	159
例外および RTTI サポートのモード	159
例外処理	160
C++ および派生言語のサポートを有効にする	161
C++ および EC++ の機能の説明	162
IAR 属性とクラスを使用する	162
関数型	163
割込みで静的クラスオブジェクトを使用する	163
新しいハンドラを使用する	163
テンプレート	164
C-SPY でのデバッグサポート	164
EEC++ の機能の説明	164
テンプレート	164
キャスト演算子の派生形	165
Mutable	165
名前空間	165
std 名前空間	165
EC++ および C++ の言語拡張	165
アプリケーションに関する考慮事項	169
出力形式に関する注意事項	169
スタックについて	169
スタックサイズについて	170
スタックのアラインメント	170
例外スタック	170
ヒープについて	171
ツールとアプリケーション間の相互処理	172
チェックサムの計算	173
チェックサムの計算	174
チェックサム関数をソースコードに追加する	175
注意事項	177
C-SPY に関する注意事項	177
リンカの最適化	178
仮想関数の除去	178

AEABI への準拠	178
IAR ILINK リンカを使用して AEABI 準拠モジュール をリンクする	179
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	180
AEABI 準拠をコンパイラで有効にする	180
CMSIS の統合	181
CMSIS DSP ライブラリ	181
CMSIS DSP ライブラリのカスタマイズ	181
コマンドラインでの CMSIS を使用したビルド	181
IAR Embedded Workbench での CMSIS を使用したビルド	182
組込みアプリケーション用の効率的なコーディング	183
データ型の選択	183
効率的なデータ型の使用	183
浮動小数点数型	184
構造体エレメントのアラインメント	185
匿名構造体と匿名共用体	185
データと関数のメモリ配置制御	187
絶対アドレスへのデータ配置	188
データと関数のセクションへの配置	189
レジスタへのデータ配置	190
コンパイラ最適化の設定	191
最適化実行のスコープ	191
複数ファイルのコンパイルユニット	192
最適化レベル	192
速度とサイズ	193
変換の微調整	194
円滑なコードの生成	197
最適化を容易にするソースコードの記述	197
スタックエリアと RAM メモリの節約	198
関数プロトタイプ	198
整数型とビット否定	199
同時にアクセスされる変数の保護	200

特殊機能レジスタへのアクセス	200
C およびアセンブラオブジェクト間での値の受渡し	201
非初期化変数	202
パート 2. リファレンス情報	203
外部インタフェースの詳細	205
呼出し構文	205
コンパイラ呼出し構文	205
ILINK 呼出し構文	205
オプションの受渡し	206
環境変数	207
インクルードファイル検索手順	207
コンパイラ出力	208
ILINK 出力	210
診断	211
コンパイラのメッセージフォーマット	211
リンカのメッセージフォーマット	211
重要度	212
重要度の設定	212
インターナルエラー	213
コンパイラオプション	215
オプションの構文	215
オプションのタイプ	215
パラメータの指定に関する規則	215
コンパイラオプションの概要	219
コンパイラオプションの説明	222
リンカオプション	259
リンカオプションの概要	259
リンカオプションの説明	262

データ表現	285
アラインメント	285
ARM コアのアラインメント	286
バイトオーダー	286
基本データ型	287
整数型	287
浮動小数点数型	292
ポインタ型	294
関数ポインタ	294
データポインタ	295
キャスト	295
構造体型	296
アラインメント	296
一般的なレイアウト	296
パック構造体型	296
型修飾子	298
オブジェクトの volatile 宣言	298
オブジェクト volatile および const の宣言	299
オブジェクトの const 宣言	300
C++ のデータ型	300
拡張キーワード	301
拡張キーワードの一般的な構文規則	301
型属性	301
オブジェクト属性	303
拡張キーワードの一覧	304
拡張キーワードの詳細	305
プラグマディレクティブ	315
プラグマディレクティブの一覧	315
プラグマディレクティブの詳細	317

組込み関数	333
組込み関数の概要	333
Neon 命令の組込み関数	339
組込み関数の詳細	339
プリプロセッサ	377
プリプロセッサの概要	377
定義済プリプロセッサシンボルの詳細	378
その他のプリプロセッサ拡張	381
NDEBUG	381
#warning message	382
ライブラリ関数	383
ライブラリの概要	383
ヘッダファイル	383
ライブラリオブジェクトファイル	384
高精度な代替ライブラリ関数	384
リエントラント性	384
longjmp 関数	385
IAR DLIB ライブラリ	385
C ヘッダファイル	385
C++ ヘッダファイル	386
組込み関数としてのライブラリ関数	389
C の追加機能	389
ライブラリにより内部的に使用されるシンボル	391
リンカ設定ファイル	393
概要	393
メモリおよび領域の定義	394
define memory ディレクティブ	394
define region ディレクティブ	395
領域	395
領域リテラル	396
領域式	397
空の領域	398

セクションの取扱い	398
define block ディレクティブ	399
define overlay ディレクティブ	401
initialize ディレクティブ	402
do not initialize ディレクティブ	404
keep ディレクティブ	405
place at ディレクティブ	406
place in ディレクティブ	407
セクションの選択	407
Section-selectors	408
拡張セクタ	410
シンボル、式、数値の使用	411
define symbol ディレクティブ	411
export ディレクティブ	412
式	413
数値	414
構造化構成	414
if ディレクティブ	414
include ディレクティブ	415
セクションリファレンス	417
セクションの概要	417
セクションおよびブロックの説明	418
IAR ユーティリティ	423
IAR アーカイブツール — iarchive	423
呼出し構文	424
iarchive コマンドの概要	424
iarchive オプションの概要	425
診断メッセージ	425
IAR ELF ツール — ielftool	427
呼出し構文	427
ielftool オプションの概要	428

IAR ELF Dumper for ARM — ielfdumparm	428
呼出し構文	429
ielfdumparm オプションの概要	429
IAR ELF オブジェクトツール — iobjmanip	430
呼出し構文	430
iobjmanip オプションの概要	431
診断メッセージ	431
IAR Absolute Symbol Exporter — isymexport	433
呼出し構文	433
isymexport のオプションの概要	434
ステアリングファイル	434
Show ディレクティブ	435
Hide ディレクティブ	435
Rename ディレクティブ	435
診断メッセージ	436
オプションの説明	438
処理系定義の動作	453
処理系定義の動作の詳細	453
J.3.1 変換	453
J.3.2 環境	453
J.3.3 識別子	455
J.3.4 文字	455
J.3.5 整数	456
J.3.6 浮動小数点	457
J.3.7 配列およびポインタ	458
J.3.8 ヒント	458
J.3.9 構造体、共用体、列挙型、ビットフィールド	459
J.3.10 修飾子	459
J.3.11 プリプロセッサディレクティブ	460
J.3.12 ライブラリ関数	462
J.3.13 アーキテクチャ	467
J.4 ロケール	467
索引	471

表

1: このガイドの表記規則	26
2: このガイドで使用されている命名規約	27
3: 初期化データを保持するセクション	74
4: 再配置エラーの説明	88
5: カスタマイズ可能な項目	94
6: printf のフォーマッタ	95
7: scanf のフォーマッタ	97
8: デバッグライブラリ付きでリンクした場合に特殊な意味を持つ関数	100
9: ライブラリ構成	109
10: printf の構成シンボルの詳細	113
11: scanf の構成シンボルの詳細	113
12: 低レベルファイル I/O	114
13: TLS を使用するライブラリオブジェクト	121
14: TLS 割当てを実装するためのマクロ	124
15: ランタイムモデル属性の例	126
16: パラメータの引渡しに使用されるレジスタ	139
17: リターン値に使用されるレジスタ	140
18: 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	143
19: 言語拡張	149
20: セクション演算子とそのシンボル	152
21: 例外スタック	170
22: コンパイラ最適化レベル	192
23: コンパイラの環境変数	207
24: ILINK 環境変数	207
25: エラーリターンコード	209
26: コンパイラオプションの一覧	219
27: リンカオプションの概要	259
28: 整数型	287
29: 浮動小数点数型	292
30: 拡張キーワードの一覧	304
31: プラグマディレクティブの一覧	315

32: 組み込み関数の一覧	333
33: 定義済シンボル	378
34: 従来の標準 C ヘッダファイル — DLIB	386
35: Embedded C++ ヘッダファイル	387
36: 標準テンプレートライブラリヘッダファイル	387
37: 新しい標準 C ヘッダファイル — DLIB	388
38: セクションセクタの指定の例	409
39: セクションの概要	417
40: iarchive パラメータ	424
41: iarchive コマンドの概要	424
42: iarchive オプションの概要	425
43: ielftool のパラメータ	427
44: ielftool オプションの概要	428
45: ielfdumparm のパラメータ	429
46: ielfdumparm オプションの概要	429
47: iobjmanip パラメータ	430
48: iobjmanip オプションの概要	431
49: ielftool のパラメータ	433
50: isymexport オプションの概要	434
51: strerror() が返すメッセージ — IAR DLIB ライブラリ	469

はじめに

ARM 用 IAR C/C++ 開発ガイドへようこそ。このガイドは、ご使用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頂くのに役立つ、詳細なリファレンス情報を提供します。また、アプリケーションを効率的に開発するための推奨コーディングテクニックも説明しています。

本ガイドの対象者

本ガイドは、C/C++ 言語を使用して ARM コア用アプリケーションを開発する予定があり、ビルドツールの使用方法に関する詳細情報を必要とするユーザを対象としています。また、以下について十分な知識があるユーザを対象としています。

- ARM コアのアーキテクチャおよび命令セット。(Advanced RISC Machines Ltd については、ARM コアの提供する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

このガイドの使用方法

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

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

IAR システムズのビルドツールを初めて使用する場合は、まず『ARM® 用 IDE プロジェクト管理およびビルドガイド』の内容を確認してください。本ガイドには、IDE および IAR C-SPY® デバッガの製品の概要、コンセプト、ユーザ情報、リファレンス情報が含まれています。

本ガイドの内容

本ガイドの構成および各章の概要を以下に示します。

パート I. ビルドツール

- 「*IAR ビルドツールの概要*」では、ツール、プログラミング言語、利用可能なデバイスサポート、ARM コアの特定の機能をサポートするために提供されている拡張機能など、IAR ビルドツールの概要について説明します。
- 「*組込みアプリケーションの開発*」では、IAR ビルドツールを使用する組込みソフトウェアの開発に必要な基礎について説明します。
- 「*データ記憶*」では、メモリへのデータの保存方法について説明します。
- 「*関数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 ランタイムライブラリ*」では、アプリケーションの実行環境である DLIB ランタイムライブラリ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ーライド、自作ライブラリのビルドにより、ランタイムライブラリを変更する方法を説明します。また、システムの初期化、cstartup ファイルの概要、ロケール用モジュールの使用方法、ファイル I/O についても説明します。
- 「*アセンブラ言語インタフェース*」では、アプリケーションの一部をアセンブラ言語で記述する場合に必要な情報を説明します。呼出し規約についても説明しています。
- 「*C の使用*」は、C 言語でサポートされている 2 つの派生型の概要と、C 規格の拡張などコンパイラ拡張の概要を説明します。
- 「*C++ の使用*」では、業界標準の EC++ と IAR 拡張 EC++ という 2 種類の C++ サポートの概要を説明します。
- 「*アプリケーションに関する考慮事項*」では、コンパイラおよびリンカの使用に関連する一部の範囲のアプリケーション問題について説明します。
- 「*組込みアプリケーション用の効率的なコーディング*」では、組込みアプリケーションに適した効率的なコーディング方法のヒントを提供します。

パート2. リファレンス情報

- 「外部インタフェースの詳細」では、コンパイラおよびリンカがそれらの環境を操作する方法として、呼出し構文、コンパイラおよびリンカにオプションを渡すための手法、環境変数、インクルードファイル検索手順、さまざまな種類のコンパイラおよびリンカ出力について説明します。また、診断システムの機能についても説明します。
- 「コンパイラオプション」では、オプションの設定方法、オプションの要約、各コンパイラオプションの詳細なリファレンス情報について説明します。
- 「リンカオプション」では、オプションの要約について説明し、各リンカオプションの詳細なリファレンス情報について説明します。
- 「データ表現」では、使用可能なデータ型、ポインタ、構造体について説明します。また、型やオブジェクト属性についても説明します。
- 「拡張キーワード」では、標準 C/C++ 言語を拡張した ARM 固有のキーワードのリファレンス情報を提供します。
- 「プラグマディレクティブ」では、プラグマディレクティブのリファレンス情報を提供します。
- 「組み込み関数」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。
- 「ライブラリ関数」では、C/C++ ライブラリ関数の概要と、ヘッダファイルの要約を説明します。
- 「リンカ設定ファイル」では、リンカ設定ファイルの目的およびその内容について説明します。
- 「セクションリファレンス」では、セクション使用に関するリファレンス情報を収録しています。
- 「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。
- 「処理系定義の動作」では、コンパイラが C 言語標準の処理系定義エリアをどのように扱うかについて説明します。

その他のドキュメント

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

ユーザガイドおよびリファレンスガイド

IAR システムズの各開発ツールについては、一連のガイドで説明しています。知りたい情報に対応するドキュメントを以下に示します。

- IAR システムズの製品のインストールおよび登録の要件と詳細については、同梱されているクイックレファレンスのブックレットおよび『インストールとライセンス登録ガイド』をご覧ください。
- IAR Embedded Workbench および提供されるツールの利用にあたっては、『IAR Embedded Workbench® の使用開始の手順』を参照してください。
- プロジェクト管理とビルドでの IDE の使用については、『ARM® 用 IDE プロジェクト管理およびビルドガイド』を参照してください。
- IAR C-SPY® デバッガの使用については、『ARM® 用 C-SPY® デバッガガイド』を参照してください。
- IAR C/C++ コンパイラのプログラミングについては、製品パッケージに IAR XLINK リンカが含まれていれば、『IAR C/C++ コンパイラリファレンスガイド』、IAR ILINK リンカが含まれていれば『IAR C/C++ 開発ガイド、コンパイルおよびリンク』をそれぞれ参照してください。
- ARM 用 IAR アセンブラを使用したプログラミングについては、『ARM® IAR アセンブラリファレンスガイド』を参照してください。
- IAR DLIB ライブラリの使用については、オンラインヘルプで利用できる DLIB ライブラリリファレンス情報を参照してください。
- ARM 用 IAR Embedded Workbench の旧バージョンで開発したアプリケーションコードやプロジェクトの移植については、『ARM® 用 IAR Embedded Workbench® 移行ガイド』を参照してください。
- MISRA-C ガイドラインを使用して、安全性を最重要視したアプリケーションを開発する方法については、『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』または『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』を参照してください。

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

オンラインヘルプシステムを参照

コンテキスト依存のオンラインヘルプの内容は以下のとおりです。

- IAR C-SPY® デバッガを使用したデバッグについての包括的な情報
- IDE のメニューやウィンドウ、ダイアログボックスに関するリファレンス情報
- コンパイラのリファレンス情報

- DLIB ライブラリ関数のキーワードリファレンス情報 関数のリファレンス情報を確認するには、エディタウィンドウで関数名を選択し、F1 キーを押します

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.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Jose Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [ドイツ語]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Sloss, Andrew N. et al, *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

WEB サイト

推奨 Web サイト：

- Advanced RISC Machines Ltd のWebサイト (www.arm.com) には、ARM コアに関する情報やニュースのほか、ARM アーキテクチャの ARM Embedded Application Binary Interface (AEABI) の情報が掲載されており、ELF/DWARF 規格についての文書へのリンクもあります。
- IAR システムズの Web サイト (www.iar.com/jp) では、アプリケーションノートおよびその他の製品情報を公開しています。
- C 標準化作業グループの Web サイト、www.open-std.org/jtc1/sc22/wg14。
- C++ Standards Committee の Web サイト、www.open-std.org/jtc1/sc22/wg21。
- Embedded C++ Technical Committee の Web サイト (www.caravan.net/ec2plus) には、Embedded C++ 規格についての情報が公開されています。

表記規則

本ガイド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。

製品インストール先のディレクトリ (arm¥doc) の記述がある場合、その場所までのフルパス（例：c:¥Program Files¥IAR Systems¥Embedded Workbench 6.n¥arm¥doc）を意味します。

表記規則

このガイドでは、次の表記規則を使用します。

スタイル	用途
コンピュータ	• ソースコードの例、ファイルパス。 • コマンドライン上のテキスト。 • 2 進数、16 進数、8 進数。
パラメータ	パラメータとして使用される実際の値を表すプレースホルダ。 たとえば、 <i>filename.h</i> の場合、 <i>filename</i> はファイルの名前を表します。
[option]	リンカディレクティブのオプション部分。ここで、[と] は実際のディレクティブの一部ではありませんが、[、]、{、} はいずれもディレクティブ構文の一部です。
{option}	リンカ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。

表 1: このガイドの表記規則





スタイル	用途
[オプション]	コマンドのオプション部分。
[a b c]	代替の選択肢を持つコマンドのオプション部分。
{a b c}	コマンドの必須部分に選択肢があることを示します。
太字	画面に表示されるメニュー名、メニューコマンド、ボタン、およびダイアログボックス。
斜体	<ul style="list-style-type: none">• 本ガイドや他のガイドへのクロスリファレンスを示します。• 強調。
...	3 点リーダーは、その前の項目を任意の回数繰り返せることを示します。
	IAR Embedded Workbench IDE 固有の内容を示します。
	コマンドラインインタフェース固有の内容を示します。
	開発やプログラミングについてのヒントを示します。
	ワーニングを示します。

表 1: このガイドの表記規則 (続き)

命名規約

以下の命名規約は、このガイドに記述されている IAR システムズの製品およびツールで使用されています。

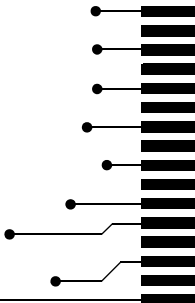
ブランド名	一般名称
ARM 用 IAR Embedded Workbench®	IAR Embedded Workbench®
ARM 用 IAR Embedded Workbench® IDE	IDE
ARM 用 IAR C-SPY® デバッガ	C-SPY、デバッガ
IAR C-SPY® シミュレータ	シミュレータ
ARM 用 IAR C/C++ コンパイラ	コンパイラ
ARM 用 IAR アセンブラ	アセンブラ
IAR ILINK リンカ	ILINK、リンカ
IAR DLIB ライブラリ	DLIB ライブラリ

表 2: このガイドで使用されている命名規約

パート I. ビルドツール

『ARM 用 IAR C/C++ 開発ガイド』のパート I 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 ランタイムライブラリ
- 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

この章では、以下に関する概要など、ARM コアについて紹介します。
すなわち、の概要を理解することになります。

- IAR ビルドツール（ビルドインタフェース、コンパイラ、アセンブラ、リンカ）
- プログラミング言語
- 利用可能なデバイスサポート
- ARM コアの特定の機能をサポートするために IAR C/C++ Compiler for ARM により提供される拡張機能

IAR ビルドツール — 概要

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



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

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



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

IAR C/C++ コンパイラ

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

IAR アセンブラ

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

IAR Assembler for ARM では、Advanced RISC Machines Ltd ARM アセンブラと同じニーモニックとオペランド構文を使用するため、既存のコードを容易に移行できます。詳細については『*ARM® 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 Linker for ARM は、組込みコントローラアプリケーションの開発に適した、強力な柔軟性のあるソフトウェアツールです。リンカは、サイズの大きい再配置可能な入力マルチモジュールの C/C++ プログラムや C/C++ プログラムとアセンブラプログラムの混合リンクに適していますが、サイズの小さい単一ファイルの絶対アドレスを持つアセンブラプログラムのリンクにも同様に適しています。

専用 ELF ツール

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

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

注：これらの ELF ユーティリティは、IAR システムズのツールにより生成されるオブジェクトファイルに非常に適しています。このため、GNU バイナリユーティリティではなく、これ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++ Compiler for ARM では、以下の 2 種類の高度なプログラミング言語を使用できます。

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

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

C の詳細は、「C の使用」を参照してください。

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

コンパイラでの言語の処理系定義の処理方法については、「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® IAR* アセンブラリファレンスガイドを参照してください。

デバイスサポート

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

サポートされている ARM デバイス

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

事前に定義されているサポートファイル

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

I/O ヘッダファイル

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

デバイス記述ファイル

デバッグは、周辺機器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\inc` ディレクトリにあり、そのファイル名拡張子は `ddf` です。これらのファイルの詳細については、『*ARM® 用 C-SPY® デバッグガイド*』および `EWARM_DDFFormat.pdf`（`arm\doc\` ディレクトリ内）を参照してください。

開発を開始するための例

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

組込みシステム用の特殊サポート

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

拡張キーワード

コンパイラは、コード生成方法の設定に使用するキーワードセットを提供しています。たとえば、特殊な関数型を宣言するためのキーワードが提供されています。



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、232 ページの `-e` を参照してください。

拡張キーワードの詳細は、「[拡張キーワード](#)」を参照してください。

プラグマディレクティブ

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

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

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

定義済シンボル

定義済プリプロセッサシンボルを使用して、コンパイル時の環境（CPU モデル、コンパイル時刻など）を調べることができます。

定義済シンボルの詳細は、「[プリプロセッサ](#)」を参照してください。

特殊な関数型

ARM コアの特殊なハードウェア機能は、`software interrupts`、`interrupts`、`fast interrupts` という特殊関数型によりサポートされています。これらの関数をアセンブラ言語で記述することなく、完全なアプリケーションを記述できます。

詳細については、59 ページの [割込み](#)、[並列処理](#)、[OS 関連のプログラミング用の基本コマンド](#) 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、129 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

この章では、IAR ビルドツールを使用する ARM コアの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。

最初に、組み込みソフトウェア開発に関するタスクの概要について説明してから、アプリケーションのコンパイルとリンク処理に関するステップなど、ビルドプロセスの概要について説明します。

次に、実行アプリケーションのプログラムフローについて説明します。

最後に、プロジェクトに必要な基本設定の概要について説明します。

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。

内部および外部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、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。詳しくは、187 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（71 ページの *コードおよびデータの配置*（リンク設定ファイル）を参照）。

周辺ユニットとの通信

外部デバイスがマイクロコントローラに接続される場合、シグナル伝達用インタフェースを初期化および制御し、たとえば、チップを使用して、ピンを選択し、外部割込みシグナルを検出および処理して行います。通常、初期化および制御はランタイムに実行する必要があります。通常、これは関数レジスタまたは 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 ヘッドファイル（拡張子は h）で定義されています。34 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、200 ページの *特殊機能レジスタへのアクセス* を参照してください。

イベント処理

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

コンパイラは、プロセッサ例外型割込み、ソフトウェア割込み、高速割込みをサポートします。すなわち、割込みルーチンを C で記述できます。詳細については、59 ページの *割込み関数* を参照してください。

システム起動

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

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

C/C++ アプリケーションでは、さらに、すべてのグローバル変数を初期化する必要があります。この初期化は、リンクおよびシステム起動コードで扱われます。詳細については、43 ページの *アプリケーションの実行— 概要* を参照してください。

リアルタイムオペレーティングシステム

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

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

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

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

他のビルドツールとの相互運用

IAR コンパイラおよびリンカは、AEABI (ARM Embedded Application Binary Interface) 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、178 ページの *AEABI への準拠* を参照してください。

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

ビルドプロセス — 概要

このセクションでは、ビルドプロセスの概要について説明します。つまり、コンパイラ、アセンブラ、リンカなどさまざまなビルドツールがどのように組み合わせられ、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利用できるチュートリアルを 1 つ以上実演しておいてください。

変換プロセス

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

注： このコンパイラは、C/C++ ソースコードをアセンブラソースコードに変換するときにも使用できます。必要な場合、オブジェクトコードにアセンブルできるアセンブラソースコードを修正できます。IAR アセンブラの詳細については、『*ARM® 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。

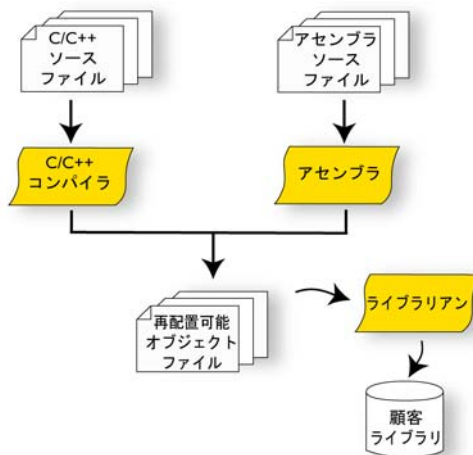


図 1: リンク前のビルドプロセス

変換後は、任意の数のモジュールを1つのアーカイブ、つまりライブラリにパッキングすることができます。ライブラリを使用する重要な理由は、ライブラリの各モジュールが条件付きでアプリケーションにリンクされるということです。すなわち、オブジェクトファイルとして提供されたモジュールによって直接的または間接的に使用されるモジュールのみアプリケーションに含まれることになります。また、ライブラリを作成してから、IAR ユーティリティ `iarchive` を使用することもできます。

リンク処理

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

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

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

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

以下の図は、リンク処理を示しています。

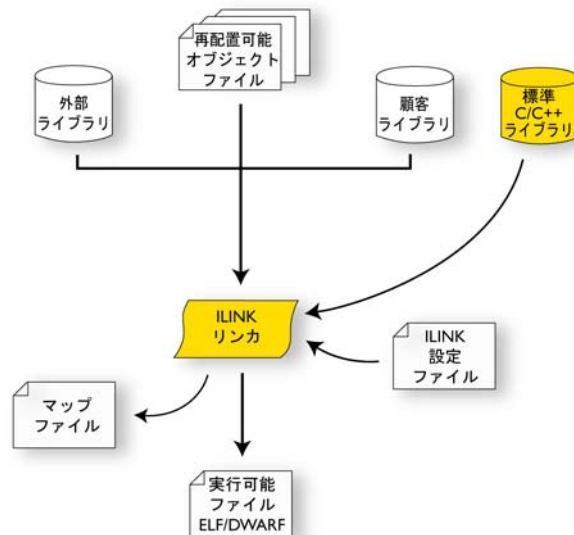


図2: リンク処理

注： 標準の C/C++ ライブラリには、コンパイラ用および C/C++ 標準ライブラリ関数の実装用のサポートルーチンが含まれています。

リンク中、ILINK は、エラーメッセージおよびログメッセージを stdout および stderr に生成することがあります。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。

ILINK により実行される手順について詳しくは、69 ページの *リンク処理* 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（427 ページの *LAR 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ています。

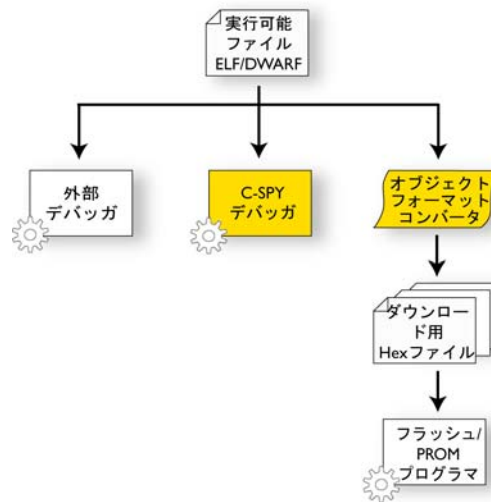


図 3: 絶対出力 ELF/DWARF で可能な使用方法

アプリケーションの実行 — 概要

このセクションでは、組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。初期化フェーズは、簡単に以下のように分割できます。

- ハードウェア初期化。通常、少なくともスタックポインタが初期化されます。

ハードウェア初期化は、通常、システム起動コード `cstartup.s` で実行され、必要に応じて、ユーザが提供する超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 / 起動や、ソフトウェア C/C++ システム初期化の準備のための CPU などの設定が行われる場合もあります。

- ソフトウェア C/C++ システム初期化

一般的に、この初期化フェーズでは、main 関数が呼び出される前に、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、さまざまな割込みの設定、通信の初期化、デバイスの初期化などが含まれます。

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

以下の一連の図は、初期化の各種段階の概要を簡単に示しています。

- 1 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、事前定義されたスタックエリアの最後にあ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。

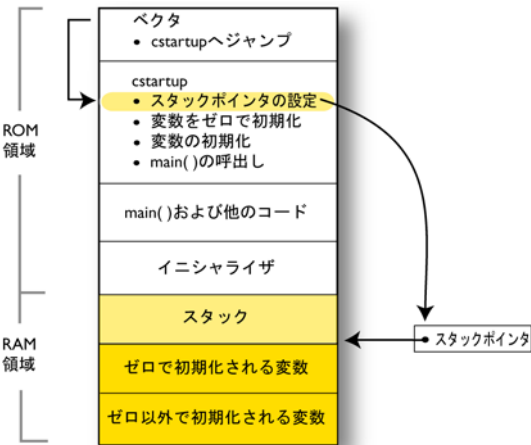


図4: ハードウェアの初期化

- 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。

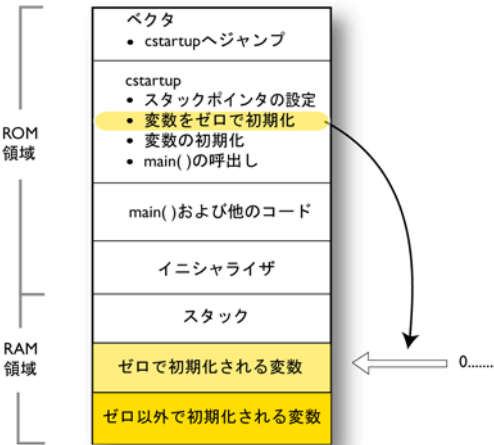


図5: 変数のゼロ初期化

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。

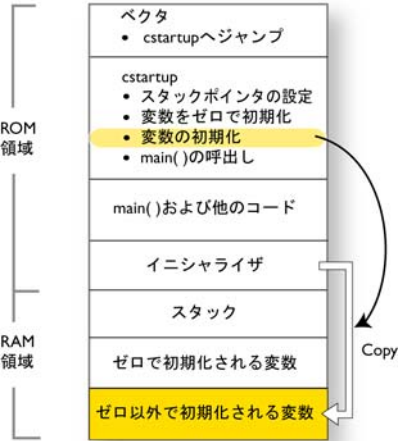


図 6: 変数の初期化

- 4 最後に、main 関数が呼び出されます。

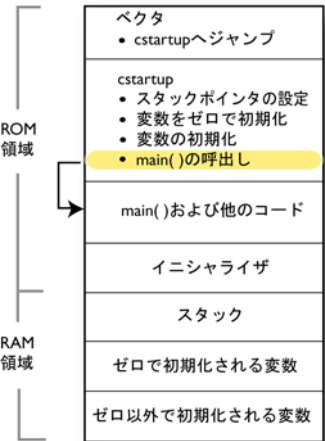


図 7: main 関数の呼出し

各段階について詳しくは、104 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、74 ページのシステム起動時の初期化を参照してください。

実行フェーズ

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

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

終了フェーズ

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

アプリケーションを制御したまま終了するには、標準 C ライブラリ関数の `exit`、`_Exit`、`abort` のいずれかを呼び出すか、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。

この詳細については、107 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。

```
icccarm myfile.c
```

コマンドラインで以下のコマンドを実行すると、ILINK 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。

基本的なプロジェクト設定

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

以下の設定が必要です。

- プロセッサ設定。すなわち、派生プロセッサ、CPU モード、相互作用、VFP および浮動小数点演算、バイトオーダーの設定です
- 最適化設定
- ランタイムライブラリ
- ILINK 設定のカスタマイズ（「アプリケーションのリンク」を参照）

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

プロセッサ構成

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

派生プロセッサ

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



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



ARM コアを指定するには、`--cpu` オプションを使用します。構文については、225 ページの `--cpu` を参照してください。

CPU モード

IAR C/C++ Compiler for ARM は、ARM および Thumb という 2 つの CPU モードをサポートします。

`__arm` または `__thumb` により明示的に宣言されている関数以外、すべての関数および関数ポインタは、指定したモードでコンパイルされます。



IDE での **〔派生プロセッサ〕** オプションや **〔チップ〕** 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。



プロジェクトの CPU モードを指定するには、`--arm` または `--thumb` オプションを使用します。構文については、224 ページの `--arm` および 256 ページの `--thumb` を参照してください。

相互作用

`--interwork` オプションを使用してコードをコンパイルすると、ARM および Thumb コードを自由に混在させることができ、相互作用関数は、ARM と Thumb の両方のコードから呼び出すことができます。相互作用は、命令セットバージョン 5、6、7 に基づいたデバイス、または `--aeabi` コンパイルオプションを使用した場合のデフォルトです。



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



プロジェクトで相互作用機能を指定するには、`--interwork` オプションを使用します。構文については、238 ページの `--interwork` を参照してください。

VFP および浮動小数点演算

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



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



浮動小数点演算でコプロセッサを使用するには、`--fpu` オプションを使用します。構文については、236 ページの `--fpu` を参照してください。

バイトオーダー

IAR C/EC++ Compiler for ARM は、ビッグエンディアンおよびリトルエンディアンバイトオーダーをサポートします。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、同じバイトオーダー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ーを指定するには、`--endian` オプションを使用します。234 ページの `--endian` 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ションがあります。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。

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

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

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

ランタイム環境

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

提供されているランタイムライブラリは、IAR DLIB ライブラリです。このライブラリは、標準の C/C++ をサポートします。このライブラリは、IEEE 754 フォーマットの浮動小数点数もサポートしています。また、ロケール、ファイル記述子、マルチバイト文字などのさまざまなレベルのサポートを指定して構成することができます。

ランタイムライブラリには、C/C++ 標準で定義された関数と、ライブラリインタフェースを定義するインクルードファイル（システムヘッダファイル）が含まれています。

ランタイムライブラリとしては、ビルド済ライブラリのいずれか1つ、またはユーザ自身でカスタマイズおよびビルドしたライブラリを選択できます。IDE は、ライブラリプロジェクトテンプレートを提供しています。これを使用して自作ライブラリのビルドが可能です。この機能により、ランタイムライブラリを完全管理できます。アセンブラソースコードだけで構成されたプロジェクトの場合は、ランタイムライブラリを選択する必要はありません。

ランタイム環境の詳細は、「*DLIB ランタイムライブラリ*」を参照してください。



IDE でのランタイム環境の設定

ライブラリは、[プロジェクト] > [オプション] > [一般オプション] から [ライブラリ構成]、[ライブラリオプション]、[ライブラリ用途] の各ページで設定した内容に従って、リンカにより自動的に選択されます。

DLIB ライブラリには、異なる設定 (Normal と Full — ロケール、ファイル記述子、マルチバイト文字などについてさまざまなレベルのサポートが含まれます。詳細は 109 ページの *ライブラリ構成* を参照してください。

選択したライブラリ構成および他のプロジェクト設定に基づいて、適切なライブラリファイルが自動的に使用されます。デバイス固有のインクルードファイルについては、正しいインクルードパスがセットアップされます。



コマンドラインのランタイム環境の設定

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

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

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

```
-I armvinc
```

IAR DLIB ライブラリのビルド済オブジェクトファイルについては、91 ページの *ビルド済ライブラリ* を参照してください。

ライブラリとランタイム環境オプションの設定

オプションの設定により、ライブラリやランタイムライブラリのサイズを削減できます。

- 関数 `printf` と `scanf`、およびこれらの派生関数で使用するフォーマット。
95 ページの *printf*、*scanf* のフォーマットの選択を参照してください。
- スタックやヒープのサイズ。82 ページの *スタックの設定*、82 ページの *ヒープの設定* をそれぞれ参照してください。

データ記憶

この章では、ARM コアのメモリレイアウトの概要と、メモリでの基本的なデータ記憶方法（スタック、静的（グローバル）メモリ、ヒープメモリ）について説明します。最後に、データをスタックやヒープに記憶する場合の詳細を説明します。

はじめに

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

さまざまなデータ記憶方法

一般的な用途では、データを以下の 3 種類の方法でメモリに格納できます。

- 自動変数
static として宣言された変数を除き、関数にローカルな変数はすべて、スタック上に格納されます。これらの変数は、関数の実行中にアクセスが可能です。関数が呼出し元に戻ると、このメモリ空間は無効になります。
- グローバル変数、モジュール静的変数、静的と宣言されたローカル変数
この場合、メモリは 1 度だけ割り当てられます。ここでの「静的」とは、このタイプの変数に割り当てられたメモリ容量がアプリケーション実行中に変化しないことを意味します。ARM コアには、単一アドレス空間があり、コンパイラはフルメモリアドレッシングをサポートします。
- 動的に割り当てられたデータ
アプリケーションは、データをヒープ上に割り当てることができます。この場合、アプリケーションが明示的にヒープをシステムに解放するまでデータは有効な状態で保持されます。このタイプのメモリは、アプリケーションを実行するまで必要なオブジェクト量がわからない場合に便利です。動的に割り当てられたデータを、メモリ容量が限られているシステムや、長期間実行するシステムで使用すると、問題が生じる危険性があります。詳細については、55 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 — スタック

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。自動変数の一部はプロセッサのレジスタに、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。

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

スタック

スタックには、以下を格納できます。

- レジスタに格納されていないローカル変数、パラメータ
- 式の間結果
- 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があるプロセッサレジスタ（呼出し先保存レジスタ）

スタックは、2つのパートで構成される固定メモリブロックです。最初のパートは、現在の関数を呼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。

利点

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

関数が自身を直接的または間接的に呼び出すことができ（*再帰関数*）、呼出しごとに自身のデータをスタックに格納できます。

潜在的な問題

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

```
int * MyFunction()
{
    int x;
    /* 何かの処理 */
    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非常に便利です。

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

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

関連情報については、262 ページの `--basic_heap` を参照してください。

潜在的な問題

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

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

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

この章では、関数について説明します。関数に関連した拡張（関数を制御するための仕組み）の概要を説明した後、これらの仕組みのいくつかを取り上げて詳しく説明します。

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、C で関数を記述するための拡張が利用できます。これらの拡張により、以下の操作が可能になります。

- 異なる CPU モードである ARM と Thumb 用にコードを生成
- RAM モードでの関数実行
- 基本コマンドによる割込み、並列処理、OS 関連のプログラミング
- 関数の最適化
- ハードウェア機能へのアクセス

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

最適化の詳細については、183 ページの *組込みアプリケーション用の効率的なコーディング* を参照してください。ハードウェア操作のアクセスに使用可能な組込み関数の詳細は、「*組込み関数*」を参照してください。

ARM および Thumb コード

IAR C/C++ Compiler 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ーライドできます。コードが相互に作用する限り、ARM および Thumb コードが同じアプリケーションに混在しても問題はありません。

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

すべてのコードポインタのサイズは4バイトです。コードポインタからデータポインタや整数型、およびその逆の場合での暗黙的および明示的なキャストには制限があります。制限の詳細については、294 ページの *ポインタ型* を参照してください。

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

RAM での実行

`__ramfunc` キーワードは、関数を RAM モードで実行します。つまり、リード/ライト属性を持つセクションに関数が配置されます。関数は、初期化された変数のように、システム起動時に ROM から RAM にコピーされます。詳細については、104 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、警告が発生します。

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

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

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

    /* ROM 内の文字列リテラル */
    msg("Hello");
}
```

これを次のように記述し直すことができます。

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

```

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

msg(hello);
}

```

詳細については、84 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。

割り込み、並列処理、OS 関連のプログラミング用の基本コマンド

IAR C/C++ Compiler for ARM では、割り込み関数、並列処理関数、OS 関連関数に関連する以下の基本関数を提供します。

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

注：ARM 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、65 ページの *ARM Cortex-M の割り込み* 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。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ーライドします。

割込みベクタと割込みベクタテーブル

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

割込み関数の定義 — 例

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

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

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

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

例外関数のインストール

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

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

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

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

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

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

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

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

割込み関数は、C リンケージを持つ必要があります。詳細については、*136 ページの 呼出し規約*を参照してください。

C++ を使用する場合は割込み関数の例を以下に示します。

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

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

その他の変更は必要ありません。

割込みおよび高速割込み

割込みおよび高速割込み関数は、パラメータを受け取らず、値を返さないため、簡単に扱うことができます。

- 割込み関数を宣言するには、__irq 拡張キーワードまたは #pragma type_attribute=__irq ディレクティブを使用します。構文については、それぞれ *307 ページの __irq* および *331 ページの type_attribute* を参照してください。
- 高速割込み関数を宣言するには、__fiq 拡張キーワードまたは #pragma type_attribute=__fiq ディレクティブを使用します。構文については、それぞれ *306 ページの __fiq* および *331 ページの type_attribute* を参照してください。

注：割込み関数 (irq) および高速割込み関数 (fiq) には、リターン型 void を指定する必要があります。また、パラメータを指定することはできません。ソフトウェア割込み関数 (swi) にはパラメータを指定できます。指定した場合、値を返すことができます。デフォルトでは、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

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

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

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

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

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

    /* VIC で割込みを承諾 */
    VICVectAddr = 0;

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

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

    /* この割込みに関連するタスクを実行 */
    (*interrupt_task)();
}
```

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

ソフトウェア割込み

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

ソフトウェア割込み関数の呼出し

ソフトウェア割込み関数をアプリケーションソースコードから呼び出すには、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、`swi_number` と表記）。コンパイラでは、この命令を C/C++ ソースコードから暗黙的に生成するための簡単な方法を提供しています。関数を宣言する際に `__swi` キーワードおよび `#pragma swi_number` ディレクティブを使用することによって生成できます。

`__swi` 関数は、たとえば、以下のように宣言できます。

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

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

ソフトウェア割込み関数は、スタックの使用以外、パラメータおよびリターン値に関して通常の関数と同じ呼出し規則に従います（*136 ページの呼出し規約*を参照）。

詳細については、*310 ページの __swi*、*330 ページの swi_number* を参照してください。

ソフトウェア割込みハンドラと関数

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

ソフトウェア割込み関数

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

詳細については、310 ページの __swi を参照してください。

ソフトウェア割込みスタックポインタの設定

ソフトウェア割込みをアプリケーションで使用する場合には、ソフトウェア割込みスタックポインタ (SVC_STACK) を設定し、スタックにエリアを割り当てる必要があります。SVC_STACK ポインタは、他のスタックと一緒に cstartup.s ファイルで設定できます。例としては、割込みスタックポインタの設定を参照してください。SVC_STACK ポインタの関連エリアは、リンカ設定ファイルで設定します (82 ページの スタックの設定を参照)。

割込み処理

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

割込みが発生すると、以下の処理が実行されます。

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

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

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

ARM CORTEX-M の割込み

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

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

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

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

ベクタテーブルは配列として実装されます。また、ベクタテーブルには常に `__vector_table` という名前が必要です。これは、`cmain` が、ベクタテーブルの位置を判別するときにそのシンボルと **C-SPY** を参照するためです。

事前定義の例外関数は、**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、リンカのみに含まれます。別のシンボルが同じ名前で作成されている場合、これが優先されます。そのためアプリケーションは、上記の一覧から正しい名前を使用するだけで、独自の例外関数を定義できます。他の割込みまたは他の例外ハンドラが必要な場合には、`cstartup_M.c` または `cstartup_M.s` ファイルのコピーを作成し、ベクタテーブルに正しく追加する必要があります。

組込み関数 `__get_CPSR` および `__set_CPSR` は、ARM Cortex-M のこれら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、201 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。

C++ と特殊な関数型

C++ のメンバ関数は、特殊な関数型を使用して宣言できます。ただし、割込みメンバ関数は静的関数でなければならないという例外があります。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する必要があります。割込みが発生し、割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。

静的メンバ関数には、特殊な関数型を使用できます。たとえば、以下の例では、関数 `handler` は、割込み関数として宣言されます。

```
class Device
{
    static __arm __irq void handler();
};
```

ILINK を使用したリンク

この章では、IAR ILINK リンカを使用したリンクプロセスについて説明します。また、関連する概念について、基本的な情報から詳細まで説明します。

リンクの概要

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

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

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

ILINK では、ARM と Thumb の両方のコード、およびこれらの組み合わせのリンクが可能です。自動的に追加命令（ベニア）を挿入することで、ILINK では、リンク先が呼出しや分岐に到達し、プロセッサの状態が必要に応じて切り変わることを確認します。ベニアの生成方法の詳細については、86 ページの *ベニア*を参照してください。

ILINK では設定ファイルを使用します。このファイルでは、ターゲットシステムのメモリマップのコードやデータ領域に対して、別々の位置を指定できます。このファイルではアプリケーションの初期化フェーズの自動処理もサポートしています。すなわち、イニシャライザのコピーや、場合によっては解凍も行って、グローバル変数領域とコード領域のイニシャライズを行います。

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

ELF ファイルを使用するために、さまざまなツールが提供されています。付属のユーティリティについては、32 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
- 使用されているデバイスなど、さまざまな情報を指定するランタイム属性
- DWRAFF フォーマットのデバッグ情報（オプション）
- 使用されているすべてのグローバルシンボルおよびすべての外部シンボルのシンボルテーブル

セクションとは、メモリ内の物理位置に配置されるデータやコードを含む論理エンティティです。セクションは、いくつかのセクションフラグメントで構成できます。セクションフラグメントは、通常、各変数または関数（シンボル）に対して 1 つです。セクションは、RAM または ROM のいずれかに配置できます。通常の組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。一般的に使用される属性を以下に示します。

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

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

セクションは、最小のリンク可能ユニットです。ただし、可能な場合、ILINK は、最終アプリケーションからさらに小さいユニット（セクションフラグメント）を実行できます。詳細については、*81 ページのモジュールの保持*、*81 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、リンクの最も重要な機能の 1 つは、アプリケーションで使用するさまざまなセクションにアドレスを割り当てることです。

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

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

IAR ILINK 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ク設定ファイルのディレクティブによって無効化できます）。

- アプリケーションに含めるモジュールを判別する。オブジェクトファイルで提供されるモジュールは、常に含まれます。ライブラリファイルのモジュールは、インクルードされるモジュールから参照されるグローバルシンボルの定義を与えるものだけが含まれます。
- 使用する標準ライブラリファイルを選択する。選択は、インクルードするモジュールの属性に基づいて行われます。これらのライブラリは、依然として解決されていないすべての未定義シンボルを充足するために使用されます。
- 追加したモジュールのうちアプリケーションに含めるセクション/セクションフラグメントを判別する。アプリケーションで実際に必要なセクション/セクションフラグメントのみが含まれます。必要なセクション/セクションフラグメントを判別する方法は、いくつかあります。たとえば、__root オブジェクト属性、#pragma required ディレクティブ、keep リンカディレクティブなどが使用できます。セクションが重複する場合は、1 つのみ含まれます。
- 必要に応じて、RAM 内の初期化変数およびコードの初期化を実行する。initialize ディレクティブを使用すると、リンク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クで作成されます。
- リンカ設定ファイルのセクション配置ディレクティブに従って、各セクションの配置場所を判別する。コピーによって初期化されるセクションは、配置ディレクティブに照らして ROM パート用と RAM パート用の 2 か所あり、それぞれ異なる属性を持ちます。配置の過程において、リンクは、コード参照をその宛先まで進めるためや CPU モードを切り替えるために必要なベニアの追加も行います。

- 実行可能イメージおよび提供されたすべてのデバッグ情報を含む絶対ファイルを生成する。再配置可能な入力ファイルの必要な各セクションの内容は、そのファイルおよびセクションの配置時に決定されたアドレスで提供された再配置情報を使用して計算されます。このプロセスによって、特定セクションの要件の一部が満たされないと、1つまたは複数の再配置エラーとなることがあります。たとえば、配置によって PC 関連のジャンプ命令の目的地のアドレスが、その範囲外となる場合などです。
- セクション配置の結果、各グローバルシンボルのアドレス、各モジュールおよびライブラリ用のメモリ使用量のサマリをリストするマップファイルを生成する（オプション）。

以下の図は、リンク処理を示しています。

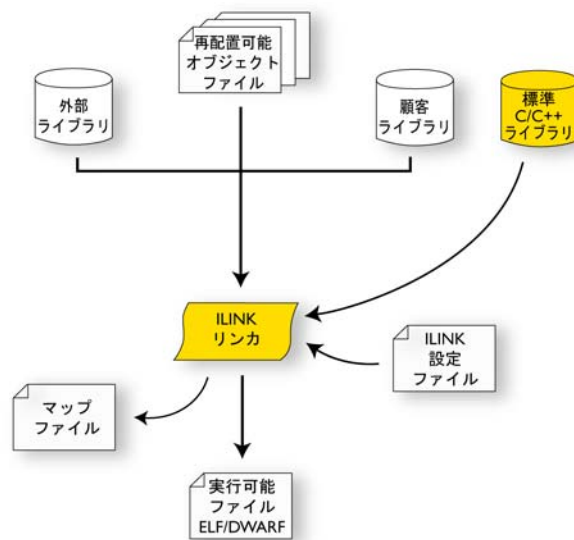


図 8: リンク処理

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

注： ELF オブジェクトファイルの実際の内容を確認するには、`ielfdumparm` を使用します。428 ページの *IAR ELF Dumper for ARM — ielfdumparm* を参照してください。

コードおよびデータの配置（リンク設定ファイル）

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、*リンク設定ファイル*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。

一般的なリンク設定ファイルには、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

簡単な設定ファイルの例を以下に示します。

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

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

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

/* 初期化操作 */
do not initialize { section .noinit };
initialize by copy { readwrite }; /*RW セクションを初期化、
                                   ゼロに初期化される
                                   セクションを除く */

/* 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 }; /* ROM: .romdata と .data_init
                             にある定数と初期化 データを配置する */
place in RAM { readwrite, /* .data、.bss、.noinit、STACK を */
                block STACK }; /* を配置する */

```

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

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

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

設定ファイルの最後のパートは、使用可能な領域に対してすべてのセクションを実際にどのように配置するかを定義しています。まず、リードオンリー (readonly) セクション .cstartup に常駐するように定義されている起動コードが、ROM 領域の開始位置、アドレス 0x00000 に配置されます。{} 内は、セクション選択と呼ばれ、ディレクティブが適用されるセクションを選択します。次に、リードオンリーセクションの残りの部分が、ROM 領域に配置されます。選択セクション { readonly section .cstartup } は、さらに一般的なセクション選択 { readonly } より優先されます。

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

以下の図は、アプリケーションがメモリにどのように配置されるかを示しています。

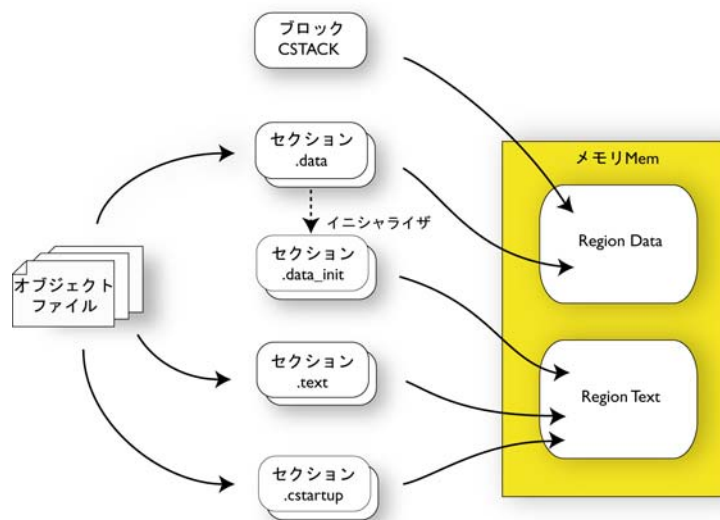


図9: メモリ内のアプリケーション

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えるシンボルの作成
- 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。コンパイラでは、この規則に 1 つだけ例外があります。それは、初期化されない `__no_init` により宣言される変数です。

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	<code>int i;</code>	リード/ライトデータ、ゼロ初期化	<code>.bss</code>	なし
ゼロで初期化されるデータ	<code>int i = 0;</code>	リード/ライトデータ、ゼロ初期化	<code>.bss</code>	なし
初期化されるデータ (ゼロ以外)	<code>int i = 6;</code>	リード/ライトデータ	<code>.data</code>	イニシャライザ
非初期化データ	<code>__no_init int i;</code>	リード/ライトデータ、ゼロ初期化	<code>.noinit</code>	なし
定数	<code>const int i = 6;</code>	リードオンリーデータ	<code>.rodata</code>	定数
コード	<code>__ramfunc void myfunc() {}</code>	リード/ライトコード	<code>.text</code>	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と一緒に `.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ないように、コンパイラは定数を `.text` セクションに配置するよう決定できます。

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

初期化プロセス

データの初期化は、ILINK およびシステム起動コードで扱われます。

変数の初期化を設定するには、以下のことを考慮する必要があります。

- ゼロ初期化されるセクションは、ILINK により自動的に扱われる。これらは RAM にのみ配置されます
- 初期化されるセクションは、ゼロ初期化されるセクションを除き、initialize ディレクティブにリストされていなければならない
通常、リンク時に、初期化されるセクションは、2つのセクションに分割される。ここで、元の初期化されるセクションはその名前を保持します。その内容は、新しいイニシャライザセクションに配置され、元の名前に付いたサフィックス _init が保持されます。配置ディレクティブにより、イニシャライザは ROM に、初期化されるセクションは RAM に配置されます。この最も一般的な例は、.data セクションです。このセクションは、リンカにより .data および .data_init に分割されます。
- 定数を含むセクションは初期化されない。このようなセクションは、フラッシュ /ROM にのみ配置されます
- __no_init により宣言される変数を保持するセクションは、初期化されず、do not initialize ディレクティブにリストされる。このようなセクションは RAM に配置されます

リンカ設定ファイルでは、次のように定義されています。

```
/* 初期化操作 */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* ゼロで初期化ゼロに初期化される
                                   セクションを除外する */

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

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

初期化の設定の詳細例については、77 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クタを使用できます。

関連項目 408 ページの *Section-selectors*。

アプリケーションのリンク

この章では、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事項を示します。また、ILINK オプションの使用およびリンカ設定ファイルの調整についても説明します。

さらに、この章では、トラブルシューティングに関するヒントについてもいくつか紹介します。

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、以下の項目について考慮する必要があります。

- 独自のメモリエリアの定義
- セクションの配置
- アプリケーションでのモジュールの保持
- シンボルおよびシンボルおよびセクションの保持
- アプリケーションの起動
- スタックおよびヒープの設定
- atexit 制限の設定
- デフォルト初期化の変更
- アプリケーションを制御するシンボル
- 標準ライブラリの処理
- ELF/DWARF 以外の出力フォーマット
- ベニア

リンカ設定ファイルの選択

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

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

これらのファイルには、ILINK で必要な情報が含まれています。この付属の設定ファイルは、ターゲットシステムメモリマップに合わせて各領域の開始および終了アドレスをカスタマイズするだけで簡単に使用できます。たとえ

ば、アプリケーションが追加外部 RAM を使用する場合は、外部 RAM のメモリエリアについての情報を追加する必要があります。

リンカ構成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。

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

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

独自のメモリエリアの定義

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

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

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

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

各領域定義は、実際のハードウェアに合わせて調整する必要があります。

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

領域の追加

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

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

異なるエリアを1つの領域にマージする

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

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

以下の例も同じです。

```
define region ROM2 = Mem:[from 0x80000 to 0xc7FFF]
-Mem:[from 0xa0000 to 0xbFFFF];
```

セクションの配置

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

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

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

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

セクションをメモリの特定のアドレスに配置する

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

```
/* .vectors セクションを 0 番地に配置する */
place at address Mem:[0] {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 사용되는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。

```
/* 変数セクションを作成 */
#pragma section = "MYOWNSECTION"

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

以下はアセンブラ言語の場合の例です。

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

新しいセクションを配置するには、元の place in ROM {readonly}; ディレクティブで十分です。

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宣言 */
#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（423 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、68 ページの *モジュールおよびセクション* を参照してください。

シンボルおよびセクションの保持

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

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

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

シンボルとセクションを保持するリンク手順の詳細については、69 ページの *リンク処理* を参照してください。

アプリケーション起動

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` 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 (268 ページの `--entry` を参照)。

スタックの設定

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{ };
```

アプリケーションに必要なサイズを指定してください。

スタックの情報については、「169 ページの *スタックについて*」を参照してください。

ヒープの設定

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};
```

アプリケーションに必要なサイズを指定してください。

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 回の呼出しを保持する空間を予約するには、以下のように記述します。

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

デフォルト初期化の変更

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

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

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

パッキングアルゴリズムの選択

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

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

使用可能なそのパッキングアルゴリズムの詳細については、402 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

initialize manually ディレクティブを使用すると、初期化を完全に制御できます。ILINK は、関連する各セクションに対して初期化データを含む追加セクションを作成しますが、実際のコピーについて処理を行いません。このディレクティブは、たとえば、オーバーレイに便利です。

```
/* MYOVERLAY1 セクションと MYOVERLAY2 セクションは MyOverlay にオーバーレイする */
define overlay MyOverlay { section MYOVERLAY1 };
define overlay MyOverlay { section MYOVERLAY2 };
```

```
/* オーバレイセクションを分割するが、システムスタートアップ時に初期化しない */
initialize manually { section MYOVERLAY* };
```

```
/* それぞれのブロックに初期化セクションを配置する */
define block MyOverlay1InRom { section MYOVERLAY1_init };
define block MyOverlay2InRom { section MYOVERLAY2_init };
```

```
/* オーバレイと初期化ブロックを配置する */
place in RAM { overlay MyOverlay };
place in ROM { block MyOverlay1InRom, block MyOverlay2InRom };
```

アプリケーションは、特定のオーバーレイを起動できます。この場合、ROM から RAM へのコピーが行われます。

```
#include <string.h>
```

```
/* オーバレイのセクションを宣言 */
```

```
#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1INROM"
```

```
/* イメージ 1 でオーバーレイに切り替わる関数 */
```

```
void SwitchToOverlay1()
```

```

{
    char *targetAddr      = __section_begin("MYOVERLAY");
    char *sourceAddr      = __section_begin("MYOVERLAY1INROM");
    char *sourceAddrEnd   = __section_end("MYOVERLAY1INROM");
    int size = sourceAddrEnd - sourceAddr;

    memcpy(targetAddr, sourceAddr, size);
}

```

コードを初期化する（ROM から RAM にコピーする）

場合によっては、アプリケーションは、コードの一部をフラッシュ ROM から RAM にコピーします。これは、コードの全体の領域に対して ILINK により簡単に実行されます。ただし、個々の関数に対して、__ramfunc キーワードを使用できます（32 ページの *RAM* での実行を参照）。

initialize ディレクティブで初期化されるコードセクションをリストし、イニシャライザと初期化されたセクションをそれぞれ ROM および RAM に配置します。

リンク設定ファイルでは、次のように定義されています。

```

/* .textrw セクションを readonly と readwrite セクションに分割 */
initialize by copy { section .textrw };

/* ブロックに配置する */
define block RamCode { section .textrw };
define block RamCodeInit { section .textrw_init };

/* これらを ROM と RAM に配置する */
place in ROM { block RamCodeInit };
place in RAM { block RamCode };

```

このブロック定義では、アプリケーションからのブロックの開始および終了位置を参照できます。

例については、172 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。

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、initialize by copy ディレクティブを使用して、以下のようを実現できます。

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

readwrite パターンは、静的に初期化されるすべての変数に一致し、これらが起動時に初期化されるように準備します。readonly パターンは、初期化に

必要なコードおよびデータを除くすべてのリードオンリーコードおよびデータに対して同様に機能します。

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

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

利用可能な圧縮アルゴリズムの詳細は、402 ページの *initialize* ディレクティブを参照してください。

関数 `__low_level_init` が存在する場合、この関数は初期化の前に呼び出されるため、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、割込みベクタテーブルなどに適用できます。

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

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

ILINK とアプリケーション間の相互処理

ILINK は、アプリケーションの制御に使用できるシンボルを定義するコマンドラインオプション `--config_def` および `--define_symbol` を提供しています。また、リンカ設定ファイルで定義される連続するメモリエリアの開始および終了位置を表すシンボルを使用することもできます。詳細については、172 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、172 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エーション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利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、91 ページの *ビルド済ライブラリ* を参照してください。

ELF/DWARF 以外の出力フォーマットの生成

ILINK は、ELF/DWARF フォーマットでのみ出力ファイルを生成できます。このフォーマットを PROM/フラッシュのプログラムに適したフォーマットに変換するには、427 ページの *IAR ELF ツール—elftool* を参照してください。

ベニア

ARM コアは、以下の 2 つの状況でベニアを使用する必要があります。

- ARM 関数を Thumb モードから呼び出す場合、または Thumb 関数を ARM モードから呼び出す場合、ベニアにより、マイクロプロセッサの状態が変更されます。コアが BLX 命令をサポートしている場合、モード変更のためのベニアは必要ありません。
- 通常は到達できない関数を呼び出す場合、ベニアは、呼出しを確実に宛先に到達させるコードを導入します。

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

詳細については、278 ページの `--no_veneers` を参照してください。

トラブルシューティングについてのヒント

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

- リンク時のメッセージ（たとえば、再配置エラーが発生した場合など）。
- ILINK で情報を stdout に記録させる `--log` オプション。この情報は、実行可能イメージが現在の状態になった理由を理解するときに役に立ちます。詳細については、273 ページの `--log` を参照してください。
- 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。詳細については、274 ページの `--map` を参照してください。

再配置エラー

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

ILINK で発生する再配置エラーの例を以下に示します。

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

このメッセージエントリについて、以下の表で説明します。

メッセージエントリ	説明
Kind	失敗した再配置ディレクティブ。ディレクティブは、使用される命令により異なります。
Location	問題が発生した場所。詳細については、以下を参照してください。 <ul style="list-style-type: none">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。 <ul style="list-style-type: none">16 進数およびオフセットを持つラベルとして表される命令アドレス。この例の場合、0x9000000c および "read" です（オフセットなし）。モジュール、およびライブラリ（適切な場合）。この例の場合、モジュールは read.o、ライブラリは iolib.a です。セクション番号およびセクション名。この例の場合、セクション番号は 6 で、名前は .text です。バイト数で指定されるセクション内のオフセット。この例のオフセットを以下に示します。0x0

表 4: 再配置エラーの説明

考えられる解決方法

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

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

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

DLIB ランタイムライブラリ

この章では、アプリケーションを実行するランタイムライブラリについて説明します。特に、DLIB ランタイムライブラリと、使用するアプリケーション向けにそれを最適化する方法について解説します。

ランタイムライブラリの概要

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

ランタイムライブラリの機能

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

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

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

- ハードウェア機能のサポート
 - 組込み関数（割込みマスク処理用関数など）による低レベルプロセッサ処理へ直接アクセス
 - インクルードファイルでの周辺ユニットレジスタと割込みの定義
 - ベクタ浮動小数点 (VFP) コプロセッサ
- ランタイムライブラリサポート（起動 / 終了コード、一部のライブラリ関数との低レベルインタフェース）
- 浮動小数点演算のサポートを含む浮動小数点環境 (`fenv`)（390 ページの `fenv.h` を参照）
- 特殊なコンパイラのサポート。たとえば、切替えの処理や整数演算のための関数など

詳細については、178 ページの *AEABI* への準拠を参照してください。

ライブラリの詳細は、「*ライブラリ関数*」を参照してください。

ランタイムライブラリの設定

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

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

- 使用するランタイムライブラリのオブジェクトファイルを選択します。
ILINK が正しいライブラリファイルを自動的に使用するため、ライブラリファイルを明示的に指定する必要はありません。91 ページの *ビルド済ライブラリ* を参照してください。
- どの定義済ランタイムライブラリ設定を使用するかを選択します (Normal/Full)
特定のライブラリ機能のサポートレベルを設定できます。たとえば、ロケールやファイル記述子、マルチバイト文字などがあります。何も指定しない場合、ライブラリオブジェクトファイルに一致するデフォルトのライブラリ設定ファイルが自動的に使用されます。ライブラリ設定を明示的に指定するには、`--dlib_config` コンパイラオブジェクトを使用します。109 ページの *ライブラリ構成* を参照してください。
- ランタイムライブラリのサイズの最適化
関数 `printf`、`scanf`、およびこれらの派生関数で使用するフォーマットを指定できます (95 ページの *printf*、*scanf* のフォーマッタの選択を参照)。スタックとヒープのサイズと配置も指定できます (82 ページの *スタックの設定* と 82 ページの *ヒープの設定* をそれぞれ参照)。
- ランタイムのデバッグサポートおよび I/O デバッグのインクルード
ライブラリは、C-SPY の [ターミナル I/O] ウィンドウへの標準入力および出力のリダイレクトや、ホストコンピュータ上のファイルへのアクセスのサポートを提供します (97 ページの *アプリケーションデバッグサポート* を参照)。
- ターゲットハードウェアのライブラリの適合
ライブラリは、ターゲットシステムへのアクセス処理に低レベルの関数のセットを使用します。これらのアクセスを機能させるには、これらの関数の自分のバージョンを実装する必要があります。たとえば、`printf` でボード上の LCD ディスプレイに書き込むようにするには、ターゲットに適合したバージョンの低レベルの関数 `__write` を実装して、文字をディスプレイに書き込めるようにする必要があります。こうした関数をカスタマイズするには、ライブラリ低レベルを十分に理解してください (101 ページの *ターゲットハードウェアのライブラリの適合* を参照)。

- ライブラリモジュールのオーバーライド
ライブラリの機能をカスタマイズしている場合、デフォルトのモジュールではなく、自分のライブラリモジュールのバージョンが使用されるようにしてください。オーバーライドは、ライブラリ全体をリビルドせずに行うことができます（102 ページの *ライブラリモジュールのオーバーライド* を参照）。
- システム初期化のカスタマイズ
システム初期化のソースコードをカスタマイズしなければならないことがほとんどです。たとえば、使用するアプリケーションがメモリをマッピングされた特殊な関数レジスタを初期化したり、データセグメントのデフォルトの初期化を省略しなければならないことがあります。この場合、ルーチン `__low_level_init` をカスタマイズします。これによって、データセグメントが初期化される前に実行されるようにします。104 ページの *システムの起動と終了* および 108 ページの *システム初期化のカスタマイズ* を参照してください。
- 独自のライブラリ設定ファイルの設定
ビルド済のライブラリ設定のほかに、独自のライブラリ設定を作成できますが、これにはライブラリの *リビルド* が必要です。この機能により、ランタイムライブラリを完全管理できます。102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。
- マルチスレッド環境の管理
マルチスレッド環境では、すべてのライブラリオブジェクトがスレッドにとってグローバルかローカルに応じて処理されるように、ランタイムライブラリを適応させる必要があります。120 ページの *マルチスレッド環境の管理* を参照してください。
- モジュールの整合性チェック
ランタイムモデル属性を使用して、モジュールが互換性のある設定を使用してビルドされるようにします（125 ページの *モジュールの整合性チェック* を参照）。

ビルド済ライブラリ

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

- アーキテクチャ
- CPU モード
- バイトオーダー
- ライブラリ構成 — Normal/Full
- 浮動小数点数実装

リンカは、正しいライブラリオブジェクトファイルとライブラリ設定ファイルを自動的にインクルードします。ライブラリ設定を明示的に指定するには、`--dlib_config` オブジェクトを使用します。詳細については、50 ページの *ランタイム環境* を参照してください。



ライブラリファイル名構文

ライブラリの名前は以下のように構成されます。

<code>{architecture}</code>	アーキテクチャの名前です。ARM アーキテクチャ v4T、v5TE、v6M、v7M に対して、4t、5E、6M、7M をそれぞれ使用できます。v5TE アーキテクチャ用にビルドされたライブラリは、v6 アーキテクチャとそれ以降でも使用されます (v6M 用と v7M 用を除く)。
<code>{cpu_mode}</code>	Thumb と ARM の場合、それぞれ t または a のどちらかです。
<code>{byte_order}</code>	リトルエンディアンとビッグエンディアンの場合、それぞれ l または b です。
<code>{fp_implementation}</code>	<p>– VFP のサポートなしにライブラリをコンパイルする場合。つまり、ソフトウェア実装の浮動小数点関数を使用する場合です。</p> <p>v アーキテクチャ VFPv2 またはそれ以降について、VFP のサポートがある状態でライブラリをコンパイルする場合。VFP のサポートがある状態でコンパイルされたライブラリには、それぞれの関数に浮動小数点の署名を持つ 2 つのエントリがあります。1 つのエントリは AAPCS の派生 VFP に、もう一方は AAPCS ベースの標準に準拠します。VFP 呼出し規約を用いてコンパイルされたモジュールの場合、リンカは派生 VFP エントリを使用します。他のモジュールについては、基本の標準エントリが使用されます。</p>
<code>{language}</code>	標準の C++ サポートに対してライブラリがコンパイルされる場合は c、Embedded C++ サポートでコンパイルされる場合は e です。
<code>{lib_config}</code>	ノーマルとフルの場合は、それぞれ n か f です。
<code>{debug_interface}</code>	SWI/SVC メカニズム、BKPT メカニズム、IAR 固有のブレークポイントメカニズムについては、それぞれ s、b、i のいずれかです。詳細については、282 ページの <i>--semihosting</i> を参照してください。

{*rwpi*} ライブラリにリード/ライトかつ位置に依存しないコードが含まれる場合、s になります (254 ページの --*rwpi* を参照)。

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

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

ライブラリファイルのグループ

ライブラリは、以下のグループのライブラリ関数で提供されます。

C ライブラリ関数のライブラリファイル

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

ライブラリファイルの名前は、以下のように構成されています。

```
dl<architecture>_<cpu_mode><byte_order><lib_config><rwpi>.a
```

これは、特に次のことを表します。

```
dl<4t|5E|6M|7M>_<a|t><l|b><n|f><s>.a
```

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

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

ライブラリファイルの名前は、以下のように構成されています。

```
dlpp<architecture>_<cpu_mode><byte_order><fp_implementation>  
<lib_config><language>.a
```

これは、特に次のことを表します。

```
dlpp<4t|5E|6M|7M>_<a|t><l|b><_|v><n|f><c|e>.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ています。

```
m<architecture>_<cpu_mode><byte_order><fp_implementation>.a
```

これは、特に次のことを表します。
m<4t|5E|6M|7M>_<a|t><1|b><|v>.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++ 標準の一部の関数部分用です。

ライブラリファイルの名前は、以下のように構成されています。

```
rt<architecture>_<cpu_mode><byte_order>.a
```

これは、特に次のことを表します。
rt<4t|5E|6M|7M>_<a|t><1|b>.a

デバッグサポート関数のライブラリファイル

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

```
sh<debug_interface>_<byte_order>.a
```

これは、特に次のことを表します。
sh<s|b|i>_<1|b>.a

ビルド済ライブラリのカスタマイズ（リビルドなし）

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

カスタマイズ可能な項目は、以下のとおりです。

カスタマイズ可能な項目	参照先
printf、scanf のフォーマット	95 ページの printf、scanf のフォーマットの選択
起動 / 終了コード	104 ページの システムの起動と終了
低レベル I/O	110 ページの 標準 I/O ストリーム
ファイル I/O	114 ページの ファイル I/O
低レベル環境関数	117 ページの 環境の操作
低レベルシグナル関数	118 ページの シグナル関数
低レベル時間関数	118 ページの 時間関数

表 5: カスタマイズ可能な項目

カスタマイズ可能な項目	参照先
ヒープ、スタック、セクションのサイズ	169 ページの スタックについて
	171 ページの ヒープについて
	71 ページの コードおよびデータの配置 (リンカ設定ファイル)

表 5: カスタマイズ可能な項目 (続き)

デフォルトのライブラリモジュールのオーバーライドについては、102 ページの [ライブラリモジュールのオーバーライド](#)を参照してください。

printf、scanf のフォーマットの選択

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

すべての printf- および scanf 関連の関数 (wprintf および wscanf バリエーションを除く) のデフォルトフォーマットをオーバーライドするために必要な作業は、該当するライブラリオプションを設定することだけです。ここでは、使用可能なオプションについて説明します。

注: ライブラリをリビルドする場合は、これらの関数をさらに最適化できます (112 ページの [printf、scanf の構成シンボル](#)を参照)。

PRINTF フォーマットの選択

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

以下の表に、各種フォーマットの機能の概要を示します。

フォーマット機能	極小	小 / SmallNoMb	大 / LargeNoMb	フル / FullNoMb
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

表 6: printf のフォーマット

フォーマット機能	極小	小 / SmallNoMb	大 / LargeNoMb	フル / FullNoMb
変換指定子 n	なし	なし	あり	あり
フォーマットフラグ +、-、#、0、空白	なし	あり	あり	あり
サイズ修飾子 h、l、L、s、t、Z	なし	あり	あり	あり
フィールド幅、精度（*を含む）	なし	あり	あり	あり
long long のサポート	なし	なし	あり	あり

表 6: printf のフォーマッタ (続き)

フォーマット機能をさらに調整する方法については、112 ページの *printf*、*scanf* の構成シンボルを参照してください。



IDE での printf のフォーマッタの指定

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



コマンドラインからの printf のフォーマッタの指定

フォーマッタを明示的に指定するには、使用するリンカファイルの以下の行のいずれかを **ILINK** コマンドラインオプション：

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

SCANF フォーマッタの選択

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

以下の表に、各種フォーマッタの機能の概要を示します。

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

表 7: scanf のフォーマッタ

フォーマット機能をさらに調整する方法については、112 ページの printf、scanf の構成シンボルを参照してください。



IDE での scanf のフォーマッタの指定

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



コマンドラインからの scanf のフォーマッタの指定

フォーマッタを明示的に指定するには、使用するリンカファイルの以下の行のいずれかを ILINK コマンドラインオプション：

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

アプリケーションデバッグサポート

デバッグ情報を生成するツールのほかに、ライブラリ低レベルインタフェース（通常は I/O 処理と基本ランタイムサポート）のデバッグバージョンがあります。デバッグライブラリを使用すると、ホストコンピュータ上でファイルを開いて stdout をデバッグの [ターミナル I/O] ウィンドウにリダイレクトするなどの処理を速く実行できます。

C-SPY デバッグサポートを含める

ライブラリで以下についてデバッグサポートを提供できます。

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

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



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

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



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



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

ライブラリ機能のデバッグ

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

これらの機能は、アプリケーション開発の初期段階、たとえばファイル I/O を使用するアプリケーションで、フラッシュファイルシステム I/O ドライバを実装する前などに非常に便利な場合があります。また、stdin や stdout を使用するアプリケーションで、実際の I/O 用ハードウェアデバイスがない状態でデバッグする必要がある場合にも使用します。もう 1 つの使用法は、デバッグ出力の生成です。

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

セミホスティングのメカニズム

使用可能なセミホスティングのメカニズムには、以下の 3 つがあります。

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

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

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

C-SPY の [ターミナル I/O] ウィンドウ

[ターミナル I/O] ウィンドウを使用可能にするには、I/O デバッグのサポートを使用してアプリケーションをリンクする必要があります。これは、関数 `__read` や `__write` を呼び出して、ストリーム `stdin`、`stdout`、`stderr` 上で I/O 操作を実行する場合、C-SPY の [ターミナル I/O] ウィンドウを介してデータの送信や読み取りが行われることを意味します。

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

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

ターミナル出力の高速化

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

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

この機能を使用するには、[プロジェクト] > [オプション] > [一般オプション] > [ライブラリオプション] を選択して、IDE でオプション [バッファしたターミナル出力] を選ぶか、以下をリンカコマンドファイルに追加します。

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

デバッグライブラリの低レベル関数

デバッグライブラリには、以下の低レベル関数の実装が含まれています。

DLIB の低レベルインタフェースの関数	アクション
<code>abort</code>	アプリケーションを終了します
<code>clock</code>	ホストコンピュータ上のクロックを返します
<code>__close</code>	ホストコンピュータ上の対応するファイルを閉じます
<code>__exit</code>	アプリケーションの最後に到達したことを通知します
<code>__open</code>	ホストコンピュータ上のファイルを開きます
<code>__read</code>	<code>stdin</code> が [ターミナル I/O] ウィンドウにダイレクトされ、それ以外のすべてのファイルはホスト上の対応するファイルを読み取ります
<code>remove</code>	ホストコンピュータ上のファイルを削除します
<code>rename</code>	ホストコンピュータ上のファイル名を変更します
<code>__iar_ReportAssert</code>	ターミナル I/O にアサートメッセージを出力します
<code>__lseek</code>	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
<code>__write</code>	<code>stdout</code> と <code>stderr</code> が [ターミナル I/O] ウィンドウにダイレクトされ、それ以外のすべてのファイルはホスト上の対応するファイルに書き込まれます

表 8: デバッグライブラリ付きでリンクした場合に特殊な意味を持つ関数

注：これらの低レベルの関数を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してアクションを実行する高レベルの関数を使用してください。詳細については、101 ページの *ライブラリの低レベルインタフェース* を参照してください。

ターゲットハードウェアのライブラリの適合

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

低レベルインタフェースを実装したら、これらの関数のバージョンを自分のプロジェクトに追加する必要があります。これについては、102 ページの **ライブラリモジュールのオーバーライド**を参照してください。

ライブラリの低レベルインタフェース

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

ただし、デバッグバージョンのライブラリ低レベルインタフェースがライブラリに含まれており、ここでターゲットハードウェアではなくデバッグを介してホストコンピュータとやりとりするように低レベル関数が実装されます。デバッグライブラリを使用する場合、[ターミナル I/O] ウィンドウへの書込みや、ホストコンピュータ上のファイルへのアクセス、ホストコンピュータからの時間の取得といったタスクをアプリケーションが実行できます。詳しくは 98 ページの **ライブラリ機能のデバッグ**を参照してください。

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

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

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

- 110 ページの **標準 I/O ストリーム**
- 114 ページの **ファイル I/O**
- 118 ページの **シグナル関数**
- 118 ページの **時間関数**
- 119 ページの **assert 関数**

ライブラリモジュールのオーバーライド

実装したライブラリ低レベルのインタフェースを使用するには、それをアプリケーションに追加します。*101* ページの *ターゲットハードウェアのライブラリの適合* を参照してください。そうでなければ、カスタマイズしたバージョンでデフォルトのライブラリルーチンをオーバーライドすると便利です。どちらの場合も、以下の手順に従ってください。

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

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

カスタマイズしたライブラリのビルドと使用

カスタマイズされたライブラリのビルドは、複雑なプロセスです。そのため、本当に必要かどうかを慎重に検討する必要があります。ローケルのサポート、ファイル記述子、でマルチバイト文字などのサポートを備えた独自のライブラリ設定を定義する場合、自分の C/C++ 標準ライブラリをビルドする必要があります。

このような場合は、以下を行う必要があります。

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

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

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

ライブラリプロジェクトのセットアップ

IDE では、ランタイムライブラリ構成のカスタマイズに使用できるライブラリプロジェクトテンプレートが提供されています。このライブラリテンプレートは、ライブラリ構成がフルに設定されています。表 9 「ライブラリ構成」、109 ページを参照してください。



IDE で、作成したライブラリプロジェクトの [一般オプション] をアプリケーションに応じて変更します (48 ページの *基本的なプロジェクト設定* を参照)。

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

ライブラリ機能の修正

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

ライブラリの機能は、*構成シンボル* で決定されます。これらのシンボルのデフォルト値は、DLib_Defaults.h ファイルで定義されています。このファイルはリードオンリーで、設定可能な値が記述されています。さらに、ユーザのライブラリには独自のライブラリ設定ファイルがあり、これによって必要なライブラリ設定を使用して特定のライブラリを設定します。詳細については、ページ 94 ページの表 5 「カスタマイズ可能な項目」を参照してください。

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

ライブラリ設定ファイルの修正

ライブラリプロジェクトで、ライブラリ設定ファイルを開き、アプリケーション要件に従って設定シンボルの値を変更することによりカスタマイズします。

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

カスタマイズしたライブラリの使用

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



IDE で以下の手順を実行する必要があります。

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

システムの起動と終了

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

起動と終了を処理するコードは、`¥src¥libs` ディレクトリの `cstartup.s`、`cmain.s`、`cexit.`、および `low_level_init.c` または `.slow_level_initarm` にあります。

Cortex-M の場合、`cstartup.s` の代わりに以下のファイルのいずれかが使用されます。

`thumb¥cstartup_M.s` または `thumb¥cstartup_M.c`

システム起動コードのカスタマイズ方法については、108 ページの *システム初期化のカスタマイズ* を参照してください。

システム起動

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

ハードウェアの初期化は、以下のように実行されます。

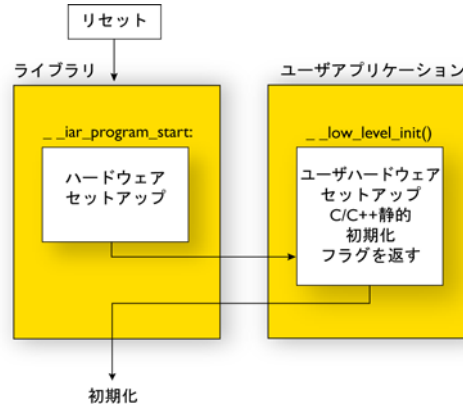


図10: ターゲットハードウェアの初期化フェーズ

- CPU は、リセットされると、システム起動コード内のプログラムエントリラベル `__iar_program_start` にジャンプします
- 例外スタックポインタは、それぞれ対応するセクションの最後の位置に初期化されます
- 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
- 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

注：Cortex-M デバイスでは、上記のリスト 2 番目の項目は有効ではありません。最初の項目と 3 番目の項目は、処理方法がわずかに異なります。リセット時、Cortex-M CPU により、PC および SP がベクタテーブル (`__vector_table`) から初期化されます。これは、`cstartup_M.c` ファイルで定義されます。

C/C++ の初期化は、以下のように実行されます。

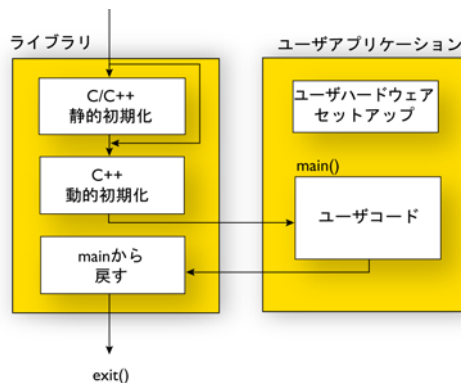


図 11: C/C++ 初期化フェーズ

- 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、74 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、43 ページの *アプリケーションの実行概要* を参照してください。

システム終了

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

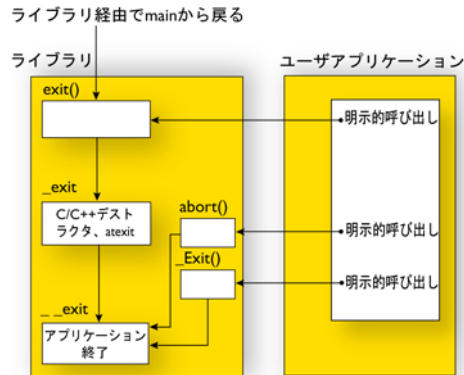


図12: システム終了フェーズ

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、これらの動作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
- 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または_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。

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

システム終了と C-SPY のインタフェース

プロジェクトがにリンクされている場合、通常の `__exit` 関数が特殊なものに置き換わります。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、プログラム終了のシミュレーションを行います。詳細については、97 ページの *アプリケーションデバッグサポート* を参照してください。

システム初期化のカスタマイズ

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

これを行うには、カスタマイズされたバージョンのルーチン `__low_level_init` を提供します。このルーチンはデータ `.s` 初期化の前に、`cmain` セクションから呼び出されます。`cstartups` 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` および `low_level_init.c` に記述されています。これらのファイルは、`arm¥src¥lib` ディレクトリにあります。

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



ライブラリをリビルドする場合は、テンプレートライブラリプロジェクトに含まれるソースファイルを使用できます。102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。

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

`__LOW_LEVEL_INIT`

製品には、C ソースファイルの `low_level_init.c` と代替アセンブラソースファイルの `low_level_init.s` という 2 つの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。後者は、ビルド済ランタイムライブラリの一部です。C ソース版を使用する場合は、変数初期化がこの時点では実行されていないため、静的初期化変数をファイル内で使用できないという制限があります。

`__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まります。関数が 0 を返す場合、データセクションは初期化されません。

CSTARTUP.S ファイルの修正

前述のように、`__low_level_init` のカスタマイズによって目的の動作を達成できる場合は、`cstartup.s` ファイルを修正する必要はありません。ただし、`cstartup.s` ファイルの修正が必要な場合は、一般的な手順に従いファイルをコピーして修正し、プロジェクトに追加することをお勧めします（102 ページの *ライブラリモジュールのオーバーライド* を参照）。

使用している `cstartup.s` のバージョンで使用される開始ラベルが、確実にリンカで使用されるようにする必要があります。リンカで使用される開始ラベルの変更方法については、268 ページの *--entry* を参照してください。

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

ライブラリ構成

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

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

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

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
Normal DLIB (デフォルト)	ロケールインタフェース、C ロケールなし、ファイル記述子のサポートなし、また <code>printf</code> および <code>scanf</code> でのマルチバイト文字もなし。
Full 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 <code>printf</code> および <code>scanf</code> でのマルチバイト文字。

表 9: ライブラリ構成

ランタイム構成の選択

ランタイム構成を選択するには、以下のいずれかの方法を使用します。

- デフォルトのビルド済構成。ライブラリ構成を明示的に指定しない場合、デフォルト構成が使用されます。ランタイムライブラリのオブジェクトファイルに一致する構成ファイルが、自動的に使用されます。
- 選択したビルド済構成。ランタイム構成を明示的に指定するには、`--dlib_config` コンパイラオプションを使用します。231 ページの `--dlib_config` を参照してください。
- 独自の構成。自分用に構成を定義できます。つまり、構成ファイルを修正する必要があります。ライブラリ設定ファイルには、ライブラリがどのようにビルドされたかが記述されています。そのため、変更を有効にするには、ライブラリをリビルドする必要があります。詳細については、102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。

ビルド済ライブラリは、デフォルト構成を使用してビルドされています (表 9 「ライブラリ構成」を参照)。

標準 I/O ストリーム

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

C/C++ ですべてのキャラクターベース I/O を実行する際に使用される、低レベルの I/O 関数があります。キャラクターベース I/O を使用する場合は、ハードウェア環境で提供される機能を使用して、これらの関数を定義する必要があります。低レベル関数の実装について詳しくは、101 ページの *ターゲットハードウェアのライブラリの適合* を参照してください。

低レベルキャラクタ I/O の実装

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

ライブラリをリビルドする場合は、テンプレートライブラリプロジェクトに含まれるソースファイルを使用できます (102 ページの *カスタマイズしたライブラリのビルドと使用* を参照)。I/O 用の低レベルルーチンをカスタマイズする場合は、ライブラリのリビルドは必要ありません。

注：__read や __write を自分で記述する場合は、C-SPY ランタイムインタフェースを考慮する必要があります（97 ページの *アプリケーションデバッグサポート* を参照）。

__write の使用例

この例のコードは、メモリがマップされた 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>

__no_init volatile unsigned char lcdIO @ 0x1000;

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

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

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

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

    return nChars;
}
```

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

__read の使用例

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

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0x1000;

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

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

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

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

    return nChars;
}
```

@ 演算子の詳細は、187 ページの *データと関数のメモリ配置制御* を参照してください。

printf、scanf の構成シンボル

アプリケーションプロジェクトをセットアップする場合は、アプリケーションに必要な printf と scanf のフォーマット機能を考慮する必要があります (95 ページの *printf、scanf のフォーマッタの選択* を参照)。

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

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

以下の構成シンボルで、printf 関数の機能を決定します。

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

表 10: printf の構成シンボルの詳細

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

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

表 11: scanf の構成シンボルの詳細

フォーマット機能のカスタマイズ

フォーマット機能をカスタマイズするには、以下を行う必要があります。

- 1 ライブラリプロジェクトをセットアップする (102 ページの *カスタマイズしたライブラリのビルドと使用* を参照)。
- 2 アプリケーションの必要に応じて、構成シンボルを定義します。

ファイル I/O

このライブラリには、`fopen` や `fclose`、`fprintf`、`fputs` など、ファイルの I/O 処理のための数多くの強力な関数が含まれています。これらの関数はすべて、いくつかの低レベル関数を呼び出します。これらの関数は、それぞれ特定のタスクを 1 つ実行するように設計されています。たとえば、`__open` はファイルを開き、`__write` は文字を出力します。アプリケーションでファイルの I/O 処理用のライブラリ関数を使用する前に、ターゲットハードウェアにあわせて、対応する低レベル関数を実装する必要があります。詳細については、101 ページの *ターゲットハードウェアのライブラリの適合* を参照してください。

ライブラリのファイル I/O 機能は、Full ライブラリ構成を持つライブラリのみによってサポートされます (109 ページの *ライブラリ構成* を参照)。すなわち、ファイル I/O は、構成シンボル `__DLIB_FILE_DESCRIPTOR` が有効な場合にのみサポートされます。有効でない場合は、関数に `FILE *` 引数を指定することはできません。

以下のファイル I/O 用テンプレートコードが付属しています。

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

表 12: 低レベルファイル I/O

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

注：I/O デバッグサポートを使用してアプリケーションをリンクする場合は、C-SPY との通信用に C-SPY 版の低レベル I/O 関数がリンクされます。詳細については、97 ページの *アプリケーションデバッグサポート* を参照してください。

ロケール

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

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

DLIB ライブラリは、以下の 2 つのメインモードで使用できます。

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

ビルド済ライブラリでのロケールサポート

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

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

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

ロケールサポートのカスタマイズ

ライブラリをリビルドする場合は、いずれかのロケールを選択できます。

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

ロケールの構成シンボル

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

`_ENCODING_USE_ENCODING` は、サポートされているすべてのロケールおよびエンコーディングを定義します。

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

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

ロケールサポートをカスタマイズする場合は、アプリケーションで必要なロケール構成シンボルを定義します。詳細については、102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。

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

ロケールインタフェースをサポートしないライブラリのビルド

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

ロケールインタフェースをサポートするライブラリのビルド

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

実行中のロケール変更

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

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

`lang_REGION`

または

```
lang_REGION.encoding
```

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

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

例

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

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

環境の操作

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

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

GETENV 関数

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

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

```
key=value¥0
```

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

次に例を示します。

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

より高度な環境変数の操作が必要な場合は、独自の `getenv` 関数や、場合によっては `putenv` 関数を実装する必要があります。この場合、ライブラリのビルドは必要ありません。ソーステンプレートは、`getenv.c` や `environ.c` の

各ファイルに含まれています。これらのファイルは、`arm¥src¥lib` ディレクトリにあります。デフォルトのライブラリモジュールのオーバーライドについては、102 ページの *ライブラリモジュールのオーバーライド* を参照してください。

システム関数

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

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。

注：I/O デバッグのサポートを使用してアプリケーションをリンクする場合、関数 `system` は C-SPY 版の関数に置き換えられます。詳細については、97 ページの *アプリケーションデバッグサポート* を参照してください。

シグナル関数

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

この場合、ライブラリのリビルドは必要ありません。ソーステンプレートは、`signal.c` や `raise.c` の各ファイルに含まれています。これらのファイルは、`arm¥src¥lib` ディレクトリにあります。デフォルトのライブラリモジュールのオーバーライドについては、102 ページの *ライブラリモジュールのオーバーライド* を参照してください。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。

時間関数

`__time32__time64`、`date` 関数が機能するためには、関数 `clock`、`__time32`、`__time64` および `__getzone` を実装する必要があります。`__time32` と `__time64` のどちらを使用するかは、`time_t` でどのインタフェースを使用するかによって決まります (390 ページの *time.h* を参照)。

これらの関数を実装する場合、ライブラリのリビルドは必要ありません。ソーステンプレートは、`¥src¥lib` ディレクトリ内のファイル `clock.c`、`time.c`、`time64.c` および `getzone.carm` にあります。デフォルトのライブラリモジュールのオーバーライドについては、102 ページの *ライブラリモジュールのオーバーライド* を参照してください。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。

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

注: I/O デバッグのサポートを使用してアプリケーションをリンクする場合、関数 `clock` および `time` は、C-SPY 版の関数に置き換えられます。これらの代替関数は、ホストのクロックと時刻をそれぞれ返します。詳細については、97 ページの *アプリケーションデバッグサポート* を参照してください。

Pow

DLIB ランタイムライブラリには、拡張精度を備えた代替の電源関数 `powXp` が含まれています。精度の低い `pow` 関数がデフォルトで使用されます。代わりに `powXp` 関数を使用するには、リンクオプション `--redirect pow=powXp` でアプリケーションをリンクしてください。

assert 関数

ランタイムデバッグのサポートを使用してアプリケーションをリンクしている場合、アサートにより `stdout` にメッセージが出力されます。これが必要でない場合は、ソースファイル `xreportassert.c` をアプリケーションプロジェクトに追加する必要があります。 `__iar_ReportAssert` 関数は、アサート通知を生成します。 `arm¥src¥lib` ディレクトリにあるテンプレートコードを使用できます。詳細については、102 ページの *カスタマイズしたライブラリのビルドと使用* を参照してください。アサーションを無効にするには、シンボル `NDEBUG` を定義する必要があります。



IDE では、このシンボル `NDEBUG` がリリースプロジェクトにデフォルトで定義されており、デバッグプロジェクトには定義されていません。コマンドラインでビルドする場合は、必要に応じてこのシンボルを明示的に定義する必要があります。381 ページの *NDEBUG* を参照してください。

Atexit

リンクは、`atexit` 関数呼出しの静的メモリエリアを割り当てます。デフォルトでは、`atexit` 関数の呼出し数は 32 バイトに制限されています (82 ページの *atexit 制限の設定* を参照)。

マルチスレッド環境の管理

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

考えられるシナリオは 3 つあり、そのうちどれが当てはまるか考慮する必要があります。

- DLIB ライブラリから提供されるマルチスレッドをサポートする RTOS を使用している場合、RTOS と DLIB ライブラリがマルチスレッドを処理します。つまり、DLIB ライブラリを適合させる必要はありません。
- DLIB ライブラリによって提供されるマルチスレッドをサポートしない、または部分的にしかサポートしない RTOS を使用する場合、おそらく RTOS と DLIB ライブラリを両方とも適合させる必要があります。
- RTOS を使用しない場合は、マルチスレッドのサポートを得るために DLIB ライブラリを適合させる必要があります。

DLIB ライブラリでのマルチスレッドのサポート

DLIB ライブラリは、2 種類のロック（システムロックとファイルストリームロック）を使用します。ファイルストリームロックは、ファイルストリームの状態が更新されたときにガードとして使用され、フルライブラリ設定でのみ必要になります。以下のライブラリオブジェクトは、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、または calloc が使用されるとき。
- 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、または freopen が使用されるときです。
- シグナルシステム。つまり signal が使用されるときです。
- 一時ファイルシステム。つまり tmpnam が使用されるときです。
- 静的関数オブジェクトの初期化。

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	以下の関数を使用される場合
エラー関数	errno, strerror
ロケール関数	localeconv, setlocale
時間関数	asctime, localtime, gmtime, mktime
マルチバイト関数	mbrlen, mbrtowc, mbsrtowc, mbtowc, wctomb, wcsrtomb, wctomb
ランド関数	rand, srand
その他の関数	atexit, strtok
C++ 例外エンジン	なし

表 13: TLS を使用するライブラリオブジェクト

マルチスレッドのサポートの有効化

ライブラリでマルチスレッドのサポートを有効にするには、以下を行う必要があります。

- #define _DLIB_THREAD_SUPPORT を DLib_product.h に追加して、ライブラリをリビルド
- ライブラリのシステムロックインタフェースのコードを実装
- ファイルストリームを使用する場合、ライブラリのファイルストリームロックインタフェースのコードを実装するか、インタフェースをシステムロックインタフェースにリダイレクト（リンカオプション --redirect を使用）
- スレッドの作成と破棄、およびライブラリの TLS アクセス方法を処理するソースコードを実装
- リンカ設定ファイルを適切に修正
- C++ 派生言語のいずれかを使用する場合は、コンパイラオプション --guard_calls を使用します。それ以外の場合は、動的にイニシャライザを持つ関数静的変数が、いくつかのスレッドによって同時に初期化されることがあります

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

システムロックインタフェース

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

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

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

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

ファイルストリームロックインタフェース

このインタフェースは、フルライブラリ設定でのみ必要です。ファイルストリームが使用される場合、完全に実装するか、またはシステムロックインタフェースにリダイレクトすることもできます。ファイルストリームロックが機能するためには、このインタフェースが実装されている必要があります。

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

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

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

DLIB ロックの使用

DLIB ライブラリによって以下の数のロックが存在すると想定されます。

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

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

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

TLS 処理

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

メインスレッドの TLS メモリエリアには以下の特徴があります。

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

各セカンダリスレッドの TLS メモリエリアには以下の特徴があります

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

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

```
void *__iar_dlib_perthread_access(void *symp);
```

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

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

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

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

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

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

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

表 14: TLS 割当てを実装するためのマクロ

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

以下はスレッド処理の一例です。

```
#include <yvals.h>

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

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

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

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

/* スレッドに対してローカルの
   TLS メモリエリアにあるオブジェクトにアクセス */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
    void _DLIB_TLS_MEMORY *sympb)
```

```

{
    char _DLIB_TLS_MEMORY *p = 0;
    if (InSecondaryThread)
        p = (char _DLIB_TLS_MEMORY *) TLSp;
    else
        p = (char _DLIB_TLS_MEMORY *)
            __segment_begin("__DLIB_PERTHREAD");

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

```

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

リンカ設定ファイルにおける変更

通常は、リンカは静的データの初期化方法を自動的に選択します。スレッドが使用される場合、メインスレッドの TLS メモリエリアはナイーブコピーによって初期化される必要があります。これは、各セカンダリスレッドの TLS メモリエリアについてもイニシャライザが使用されるためです。リンカ設定ファイルに次の文を挿入してください。

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

モジュールの整合性チェック

ここでは、ランタイムモデル属性の概念と、互換性のある設定を使用してモジュールがビルドされるように徹底するために使用できるしくみについて説明します。

アプリケーションの開発では、互換性のないモジュールの同時使用の防止が重要です。たとえば、2つのモードで実行可能な UART がある場合、uart というように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。

IAR システムズが提供するツールは、定義済みのランタイムモデル属性を使用します。自動的にモジュールの整合性を確認できます。

ランタイムモデル属性

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

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

注：IAR の定義済ランタイムモデル属性の場合、リンク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 15: 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 は相互にリンクできます。また、file4 と file5 のいずれかにリンクできますが、両方に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。次に例を示します。

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

327 ページの rtmodel 構文の詳細については、を参照してください。

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

```
rtmodel "color", "red"
```

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

アセンブラ言語インタフェース

組込みシステム用アプリケーションの開発では、正確なタイミングや特殊な命令シーケンスを要求する ARM コアのメカニズムを使用する場合など、コードの一部をアセンブラで記述する必要があることがあります。

この章では、アセンブラでの記述方法、C 言語で代替する方法、それぞれの利点と欠点を説明します。また、C/C++ で記述されたアプリケーションで使用できる関数をアセンブラ言語で記述する方法についても説明します。

最後に、関数の呼出し方法、および C-SPY® の [呼出しスタック] ウィンドウで使用する呼出しフレーム情報のサポートをアセンブラルーチンで実装する方法について説明します。

C 言語とアセンブラの結合

IAR C/C++ Compiler 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
- 組込み関数 (C の代替関数)
- インラインアセンブラ

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

組込み関数

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

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

インラインアセンブラを使用する場合と比較した場合の組込み関数の利点は、レジスタの割当てや変数とシーケンスとのインタフェースに必要な情報のすべてをコンパイラが把握できることです。また、そのようなシーケンスのある関数を最適化する方法もコンパイラで特定できます。これは、インラインアセンブラシーケンスでは不可能です。その結果、目的のシーケンスをコードに適切に統合し、その結果をコンパイラで最適化できます。

使用可能な組込み関数の詳細は、「*組込み関数*」を参照してください。

C 言語とアセンブラモジュールの結合

アプリケーションの一部をアセンブラで記述し、C/C++ モジュールと混在させることができます。インラインアセンブラを使用する場合と比較して、これにはいくつか利点があります。

- 関数呼出しの仕組みが明確に定義されている
- コードが読みやすくなる
- オプティマイザで C/C++ 関数を処理できる

関数呼出しとリターンの命令シーケンスにおいて、ある程度のオーバーヘッドが生じます。また、コンパイラは一部のレジスタをスクラッチレジスタと見なします。ただし、コンパイラは同時に、すべてのスクラッチレジスタの内容がインラインアセンブラ命令により破壊されるものと見なします。多くの場合、外部命令によるオーバーヘッドは、オプティマイザにより削除されます。

重要な利点は、コンパイラの生成内容とアセンブラでの記述内容との間のインタフェースが明確に定義されることです。インラインアセンブラを使用する場合には、インラインアセンブラの行がコンパイラ生成コードと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、多く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？
- アセンブラコードのパラメータの場所は？また、呼出し元へ値を返す方法は？
- C で記述した関数をアセンブラコードから呼び出す方法は？
- 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、デバッガで呼出しスタックが表示されない理由は？

最初の質問については、133 ページの *C* からのアセンブラルーチンの呼出しで説明します。2 番目と 3 番目の疑問については、136 ページの *呼出し規約* で説明します。

131 ページの *インラインアセンブラ* のセクションでは、インラインアセンブラの使用方法について説明していますが、メモリのデータへのアクセス方法についても説明しています。

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、呼出し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 *呼出しフレーム* 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、143 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、133 ページの *C* からのアセンブラルーチンの呼出し、135 ページの C++ からのアセンブラルーチンの呼出し、でそれぞれ説明しています。

インラインアセンブラ

アセンブラコードを C/C++ の関数に直接挿入できます。asm と __asm の各キーワードは、どちらも指定のアセンブラ文をインラインで挿入します。詳細については 148 ページの *インラインアセンブラ* を参照してください。asm キーワードの使用法を次の例で説明します。この例（ARM モードの場合）は、インラインアセンブラを使用する場合のリスクも示します。

```
extern volatile char UART1_SR;
#pragma required=UART1_SR

char sFlag;

void Foo(void)
{
    while(!sFlag)
    {
        asm(" ldr r2,[pc,#0] %n" /* r2 = sFlag のアドレス */
            " b .+8 %n" /* 定数をジャンプ */
            " DCD sFlag %n" /* sFlag のアドレス */
            " ldr r3,[pc,#0] %n" /* r3 = UART1_SR のアドレス */
            " b .+8 %n" /* 定数をジャンプ */
            " DCD UART1_SR %n" /* UART1_SR のアドレス */
            " ldr r0,[r3] %n" /* r0 = UART1_SR */
            " str r0,[r2]"); /* sFlag = r0 */
    }
}
```

この例では、グローバル変数 `Flag` の割当てをコンパイラが認識できません。すなわち、前後のコードがインラインアセンブラ文に依存することはできません。

インラインアセンブラ命令は、単純にプログラムフローの与えられた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。

インラインアセンブラシーケンスでは、C/C++ で記述された前後のコードとのインタフェースが明確に定義されていません。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、将来コンパイラをアップグレードした場合に保守面で問題が生じる可能性もあります。また、インラインアセンブラの使用にはいくつか制限があります。

- コンパイラによる最適化では、インラインシーケンス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。
- 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。
- アラインメントは制御できません。つまり、DC32 などのディレクティブの位置が誤ってアラインメントされることがあります。
- 自動変数にアクセスできません。
- その他のレジスタ名、ニーモニック、演算子はサポートされていません。
-j アセンブラオプションの詳細については、『ARM® IAR アセンブラリファレンスガイド』を参照してください。

これらの理由から、通常はインラインアセンブラの使用を避けてください。適当な組込み関数がない場合は、インラインアセンブラを使用する代わりに、アセンブラ言語で記述したモジュールを使用することをお勧めします。アセンブラルーチンへの関数呼出しの方が通常は性能低下が小さいためです。

対応する Thumb モードの例では、Thumb2 命令セットが必要です。

```
extern volatile char UART1_SR;
#pragma required=UART1_SR

char sFlag;

void Foo(void)
{
    while(!sFlag)
    {
        /* Thumb2 が必要です : Thumb1 LDR 命令は
         * LDR のソースラベルに 4 バイトのアラインメントが必要です
         * これは保証できません
         *
         * LDR に使用される基準アドレスは 4 バイトでアラインメントされていますが
         * コードのアラインメントは不明なため、ラベルを使用して
```

```

* アセンブラで正しい PC に相対するオフセットを計算する必要が
* あります。インラインアセンブラはパスが 1 つのみなので、ラベルは
* 使用される前に導入する必要があります
*/
asm(" b.n    .+11        %n" /* 定数をジャンプ          */
    "const:    %n" /* ラベルが奇数 (Thumb)          */
    " DCD sFlag    %n" /* sFlag のアドレス          */
    " DCD UART1_SR %n" /* UART1_SR のアドレス       */
    " ldr.w r3,const+3 %n" /* r3 = UART1_SR のアドレス */
    " ldr.w r2,const-1 %n" /* r2 = sFlag のアドレス    */
    " ldr r0,[r3]    %n" /* r0 = UART1_SR            */
    " str r0,[r2]"); /* sFlag = r0                */
}
}

```

C からのアセンブラルーチンの呼出し

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。

- 呼出し規約に準拠していること
- PUBLIC エントリポイントラベルがあること
- 型チェックやパラメータの型変換（オプション）を可能にするため、以下の例のようにすべての呼出しの前に **external** として宣言されていること

```
extern int foo(void);
```

または

```
extern int foo(int i, int j);
```

これらの条件を満たすには、C でスケルトンコードを作成してコンパイルし、アセンブラリストファイルを調べるという方法があります。

スケルトンコードの作成

正しいインタフェースを持つアセンブラ言語ルーチンを作成するには、C コンパイラで作成されたアセンブラ言語ソースファイルから開始することをお勧めします。関数プロトタイプごとにスケルトンコードを作成する必要があります。

以下の例は、ルーチン本体を簡単に追加できるスケルトンコ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;
    int locInt = gInt;
    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。

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ーライド] 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、[アセンブラ出力ファイル] オプションを選択し、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 .
```

-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。また、言語拡張を有効にするために低い最適化レベルおよび `-e` を指定することに注意してください。

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： `-1A` オプションは、呼出しフレーム情報 (CFI) ディレクティブを含むリストファイルを作成します。これは、これらのディレクティブと使用方法について調べる意図がある場合に便利です。呼出し規約のみを調べるのであれば、CFI ディレクティブをリストファイルから除外できます。IDE で **【プロジェクト】 > 【オプション】 > 【C/C++ コンパイラ】 > 【リスト】** を選択し、サブオプションの **【呼出しフレーム情報のインクルード】** の選択を解除します。コマンドラインでは、`-1A` ではなく `-1B` オプションを使用します。C-SPY の **【呼出しスタック】** 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
- 呼出しフレーム情報 (CFI)

CFI ディレクティブは、デバッガの **【呼出しスタック】** ウィンドウで必要な呼出しフレーム情報を記述します。詳細については、143 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出し

C の呼出し規約は、C++ 関数には適用されません。最も重要なことは、関数名だけでは C++ 関数を特定できない点です。型安全なリンケージを保証し、オーバーロードを解決するために、関数のスコープと型も必要になります。

もう 1 つの違いとして、非静的メンバ関数が、別の隠し引数である `this` ポインタを取る点があります。

ただし、C リンケージを使用する場合は、呼出し規約は C の呼出し規約に準拠します。したがって、アセンブラルーチンは以下の方法で宣言した場合に C++ から呼び出されます。

```
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出しをメンバ関数内にラップできます。関数インライン化が有効になっていれば、インラインメンバ関数を使用することで、余分な呼出しによるオーバーヘッドが解消されます。

```
class MyClass;

extern "C"
{
    void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
    inline void DoIt(int arg)
    {
        ::DoIt(this, arg);
    }
};
```

呼出し規約

呼出し規約とは、プログラム内の関数が別の関数を呼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出し規約について説明します。内容は以下のとおりです。

- 関数の宣言
- C/C++ のリンケージ
- 保護レジスタとスクラッチレジスタ
- 関数の入口
- 関数の終了
- リターンアドレスの処理

最後に、実際の呼出し規約の例を示します。

特に説明がない限り、コンパイラで使用される呼出し規約は AEABI の一部である AAPCS に準拠します。詳細は、178 ページの AEABI への準拠を参照してください。

関数の宣言

C では、コンパイラで関数の呼出し方法を特定できるように、関数は規則に沿って宣言する必要があります。宣言の例を次に示します。

```
int MyFunction(int first, char * second);
```

この宣言は、整数と文字へのポインタの 2 つのパラメータを関数で指定することを示します。この関数は、整数を返します。

通常は、コンパイラが関数について特定できるのはこれだけです。したがって、この情報から呼出し規約を推定できる必要があります。

C++ ソースコードでの C リンケージの使用

C++ では、関数は C または C++ のいずれかのリンケージを持つことができます。アセンブラルーチンを C++ から呼び出すには、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。関数が別の関数の呼出し後もレジスタの値を必要とする場合は、呼出し中はその値をスタックなどで保存する必要があります。

レジスタ R0 ～ R3 のすべてと R12 は、スクラッチレジスタとして関数で使えます。**Veneer** のために自動的に挿入される命令のため、R12 は、アセンブラ関数間の呼出し時にもスクラッチレジスタとみなされるので注意してください。

保護レジスタ

一方、保護レジスタは、他の関数の呼出し後も保持されます。呼び出された関数は保護レジスタを他の用途で使えますが、使用前に値を保存し、関数終了時に値を復元する必要があります。

レジスタ R4 ～ R11 が保護レジスタです。これらは、呼出し先関数で保持されます。

専用レジスタ

レジスタによっては、考慮すべき特別な要件があります。

- スタックポインタレジスタ R13/SP は、常にスタック上の最後のエレメントかその下の位置を示します。割込みが発生すると、スタックポインタが示す位置より下のエレメントはすべて破壊されます。関数の入り口および終了位置では、スタックポインタは 8 バイトに整列されている必要があります。関数内では、スタックポインタは常にワード整列されていなければなりません。終了位置では、SP の値はエントリ位置の値と同じである必要があります。
- レジスタ R15/PC は、プログラムカウンタ専用です。
- リンクレジスタ R14/LR は、関数の入口にリターンアドレスを保持します。

関数の入口

パラメータは、2 つの基本的な方法（レジスタ、スタック）のいずれかを使用して引き渡すことができます。メモリ経由で迂回して引き渡すよりもレジスタを使用する方が大幅に効率的です。そのため、呼出し規約では可能な限りレジスタを使用するように規定されています。パラメータの引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を使用して引き渡されます。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込み関数を除いて、割込み関数にはどんなパラメータも指定できません。

- ソフトウェア割込み関数は、通常の関数と同じ様にはスタックを使用できません。svc 命令が実行されると、プロセッサはスーパーバイザモードに切り替わり、スーパーバイザスタックが使用されるので、引数は、アプリケーションが割込み発生前にスーパーバイザモードで実行していない場合、スタックに渡すことはできません。

隠しパラメータ

関数の宣言や定義で明示されるパラメータに加えて、隠しパラメータが存在する場合があります。

- 関数より返された構造体が 32 ビットを超えている場合、その構造体が格納されるメモリ位置は、追加パラメータとして渡されます。これは、常に最初のパラメータとして扱われることに注意してください。
- 関数が非静的 C++ メンバ関数の場合、this ポインタが最初のパラメータとして渡されます（ただし、1 つのみの場合、配置されるのは構造体ポインタのリターン後）。詳細については、135 ページの C++ からのアセンブラルーチンの呼出しを参照してください。

レジスタパラメータ

パラメータの引渡しに使用できるレジスタは R0-R3 です。

パラメータ	レジスタでの引渡し
32 ビット以下のスカラ値と浮動小数点値、および単精度（32 ビット）浮動小数点値	最初の空きレジスタを使用して渡されます：R0 ~ R3
long long および倍精度（64 ビット）値	最初に使用できるレジスタペアで渡されます：R0:R1、R2:R3

表 16: パラメータの引渡しに使用されるレジスタ

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

32 ビット未満のパラメータを持つ関数が呼び出される場合、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。

スタックパラメータとレイアウト

スタックパラメータは、メモリのスタックポインタが指す位置を開始位置として格納されています。スタックポインタ以下（下位メモリ方向）には、呼出し先関数で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。

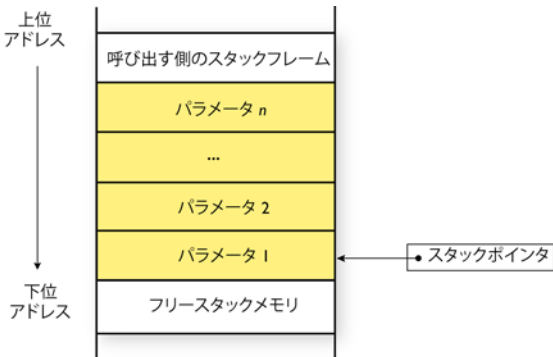


図 13: 関数呼出し後のスタックのイメージ

スタックは関数の入り口で 8 に整列されます。

関数の終了

関数は、呼出し元の関数やプログラムに値を返すことができます。または、関数のリターン型が void の場合もあります。

関数のリターン値がある場合は、スカラ（整数、ポインタなど）、浮動小数点数、構造体のいずれかになります。

リターン値に使用されるレジスタ

リターン値に使用可能なレジスタは R0 および R0:R1 です。

リターン値	レジスタ / レジスタペアで渡されます
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 17: リターン値に使用されるレジスタ

リターン値が 32 ビット未満の場合、値は 32 ビットに符号拡張またはゼロ拡張されます。

関数終了時のスタックのレイアウト

呼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、呼出し元で行うべきです。

リターンアドレスの処理

アセンブラ言語で記述した関数は、レジスタ `LR` によりポイントされるアドレスまでジャンプすることで、終了時に呼出し元に戻ります。

関数の入口で、非スカラレジスタおよび `LR` レジスタは、1つの命令でプッシュできます。関数の出口では、これらすべてのレジスタは1つの命令でポップできます。リターンアドレスは、`PC` に直接ポップできます。

以下の例は、この動作を示しています。

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

push      {r4-r6,lr}    ; スタックのアラインメント 8 を保持
bl        func

; 何らかの処理

pop       {r4-r6,pc}    ; リターン
end
```

例

以下では、宣言の例や対応する呼出し規約を紹介します。後の例ほど複雑になっています。

例 1

以下の関数が宣言されているとします。

```
int add1(int);
```

この関数は、1つのパラメータをレジスタ `R0` を使用して引き渡し、リターン値をレジスタ `R0` を使用して呼出し元に戻します。

以下のアセンブラルーチンは、この宣言に適合します。このルーチンは、パラメータの値よりも1つ大きな値を返します。

```
name      return
section   .text:CODE
adds      r0, r0, #1
bx        lr
end
```

例2

この例は、構造体がスタックを使用して引き渡される方法を説明しています。以下が宣言されているとします。

```
struct MyStruct
{
    short a;
    short b;
    short c;
    short d;
    short e;
};

int MyFunction(struct MyStruct x, int y);
```

構造値メンバの値 a、b、c、d は、レジスタ R0-R3 で渡されます。最後の構造体メンバ e および整数パラメータ y はスタックで渡されます。呼出し元関数は、スタックの上位 8 バイトを確保し、2 つのスタックパラメータの内容をその位置にコピーします。リターン値は、R0 レジスタを使用して呼出し元に返されます。

例3

次の関数は、struct MyStruct 型の構造体を返します。

```
struct MyStruct
{
    int mA;
};

struct MyStruct MyFunction(int x);
```

リターン値のメモリ位置を割り当てて、それにポインタを最初の隠しパラメータとして引き渡すのは、呼出し元の関数の役割です。リターン値が格納されるべき位置へのポインタは、R0 内で引き渡されます。パラメータ x は R1 で引き渡されます。

関数が構造体へのポインタを返すよう宣言されているとします。

```
struct MyStruct *MyFunction(int x);
```

この場合、リターン値はスカラであり、隠しパラメータはありません。パラメータ x は R2 で引き渡され、リターン値はデータモデルに応じて R0 または (R1、R0) 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、呼出しスタック、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、呼出しフレームのレイアウトを説明するデバッグ情報、特にリターンアドレスの格納されている場所を提供することで、これを可能にします。

アセンブラ言語で記述したルーチンのデバッグ時に呼出しスタックを使用できるようにするには、アセンブラディレクティブ CFI を使用して、同等のデバッグ情報をアセンブラソース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® IAR アセンブラリファレンスガイド』を参照してください。

CFI ディレクティブ

CFI ディレクティブは、呼出し元関数のステータス情報を C-SPY に提供します。この中で最も重要な情報は、リターンアドレスと、関数やアセンブラルーチンのエントリ時点でのスタックポインタの値です。C-SPY は、この情報を使用して、呼出し元関数の状態を復元し、スタックを巻き戻すことができます。

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

呼出しフレーム情報を記述するには、以下の 3 つのコンポーネントが必要です。

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

以下の表に、コンパイラが使用する名前ブロックで定義されているすべてのリソースを示します。

リソース	説明
CFA R13	スタックの呼出しフレーム
R0-R12	プロセッサ汎用 32 ビットレジスタ
R13	スタックポインタ、SP
R14	リンクレジスタ、LR
S0-S31	ベクトル浮動小数点 (VFP) 32 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 18: 名前ブロックで定義されている呼出しフレーム情報リソース

CFI サポートを持つアセンブラソースの作成

呼出しフレーム情報を正しく処理するアセンブラ言語ルーチンを作成するには、コンパイラで作成されたアセンブラ言語ソースファイルから開始することをお勧めします。

- 1 適当な C ソースコードを使用して開始します。以下に例を示します。

```
int F(int);
int cfiExample(int i)
{
    return i + F(i);
}
```

- 2 C ソースコードをコンパイルします。呼出しフレーム情報（CFI ディレクティブ）を含むリストファイルを必ず作成してください。



コマンドラインでは、-lA オプションを使用します。



IDE で [プロジェクト] > [オプション] > [C/C++ コンパイラ] > [リスト] を選択し、サブオプションの [呼出しフレーム情報のインクルード] が選択されていることを確認します。

この例のソースコードの場合、リストファイルは以下のようになります。

```
#if Compile
int F(int);
int cfiExample(int i)
{
    return i + F(i);
}
#endif

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 `cfiCommon.i` には、`CFI_NAME_BLOCK`、`CFI_COMMON_ARM`、`CFI_COMMON_Thumb` の各マクロが含まれており、これらは一般的な名前ブロックと2つの共通ブロックを宣言します。これらのマクロは、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

この章では、コンパイラの C 言語のサポートについて概要を説明します。また、IAR の C 言語拡張の概要についても簡単に解説します。

C 言語の概要

IAR C/C++ Compiler for ARM は、ISO/IEC 9899:1999 規格（最新の技術的誤植 No.3 も含む）、通称 C99 をサポートしています。このガイドでは、この規格を *標準の C* と呼び、これがコンパイラで使用されるデフォルト標準です。この標準は C89 よりも厳密です。

また、コンパイラは ISO 9899:1990 規格（すべての技術的誤植と追加事項を含む）、通称 C94、C90、C89、ANSI C もサポートしています。本ガイドでは、この規格を *C89* といいます。この規格を有効にするには、`--c89` コンパイラオプションを使用します。

C99 規格は C89 から派生したものですが、以下のような特長が追加されています。

- `inline` キーワードは、ディレクティブの直後に宣言された関数をインライン化するようにコンパイラに指示します
- 宣言と文は、同じスコープ内で混在させることが可能です
- `for` ループの初期化式における宣言
- `bool` データ型
- `long long` データ型
- 複雑な浮動小数点型
- C++ スタイルのコメント
- 複合リテラル
- 構造体終端の不完全な配列
- 16 進数の浮動小数点定数
- 構造体と配列における指定イニシャライザ
- プリプロセッサ演算子 `_Pragma()`
- 可変引数マクロは、`printf` スタイルの関数に相当するプリプロセッサマクロです
- VLA（可変長配列）は、コンパイラオプション `--vla` によって明示的に有効化する必要があります
- `asm` キーワードまたは `__asm` キーワードを使用したインラインアセンブラ

注：たとえ C99 の機能であっても、IAR C/C++Compiler for ARM は、UCN (universal character name = 汎用文字名) をサポートしていません。

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、生成される関数にアセンブラ命令を挿入することができます。

asm 拡張キーワードおよびそのエイリアス __asm は、どちらも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。

注：これらのキーワードでは、一部のアセンブラディレクティブや演算子は挿入できません。

構文は以下のとおりです。

```
asm ("string");
```

string には、有効なアセンブラ命令またはデータ定義用アセンブラディレクティブを指定できます。ただし、コメントは指定できません。以下のように、複数の連続したインラインアセンブラ命令を記述できます。

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

ここで、¥n (改行) は各アセンブラ命令の区切り文字として使用しています。インラインアセンブラ命令では、ローカルラベルを定義して使用できます。

インラインアセンブラの詳細については、129 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。

拡張の概要

コンパイラでは、C 標準の機能のほかに、組込み業界における効率的なプログラミング専用の機能から、規格上の軽微な問題の緩和にいたるまで、幅広い拡張を提供します。

以下は使用可能な拡張の概要です。

- IAR C 言語拡張

使用可能な言語拡張については、「150 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については、「*拡張キーワード*」を参照してください。C++、言語の 2 レベルのサポート、C++ 言語拡張については、「*C++ の使用*」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」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非常に便利です。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については、「129 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
IAR DLIB ライブラリは、組み込みシステムに利用される最も重要な C/C++ ライブラリ定義を提供します。詳細については、385 ページの IAR DLIB ライブラリを参照してください。

注：プラグマディレクティブ以外の拡張を使用する場合、アプリケーションは C 規格との整合性がなくなります。

言語拡張の有効化

プロジェクトオプションを使用して、さまざまな言語の適合レベルを選択できます。

コマンドライン	IDE*	説明
--strict	厳密	すべての IAR C 言語拡張が無効になり、C 規格外のすべてに対してエラーが发せられます。

表 19: 言語拡張

コマンドライン	IDE*	説明
なし	標準	C 規格に対するすべての拡張が有効ですが、組込みシステムのプログラミングの拡張は一切有効になりません。拡張については、150 ページの IAR C 言語拡張を参照してください。
-e	IAR 拡張ありの標準	すべての IAR C 言語拡張が有効になります。

表 19: 言語拡張 (続き)

* IDE では、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語] > [言語の適合] を選択して、適切なオプションを選びます。言語拡張はデフォルトで有効になっています。

IAR C 言語拡張

コンパイラには、幅広い C 言語拡張セットが用意されています。アプリケーションに必要な拡張が簡単に見つかるように、このセクションでは拡張は以下のようにグループ化されています。

- 組込みシステムプログラミングの拡張 — 一般的にメモリの制限を満たすために、使用する特定のコアでの効率的な組込みプログラミングに特化した拡張。
- C 規格に対する緩和 — つまり、C 規格の重要でない問題やの緩和や、便利ではあるが重要性の低い構文の拡張。153 ページの C 規格に対する緩和を参照してください。

組込みシステムプログラミングの拡張

以下の言語拡張は、C/C++ の両方のプログラミング言語で使用可能なもので、組込みシステムのプログラミングに最適です。

- 型属性およびオブジェクト属性
関連する概念、一般的な構文規則、リファレンス情報については、「拡張キーワード」を参照してください。
- 絶対アドレスへの配置、指定 section への配置
@ 演算子や #pragma location ディレクティブを使用して、グローバル変数や静的変数を絶対アドレスに配置することや、指定された section に変数や関数を配置することができます。これらの機能の使用方法については、187 ページの データと関数のメモリ配置制御、322 ページの location を参照してください。
- アラインメントの制御
それぞれのデータ型には独自のアラインメントがあります。詳細については、285 ページの アラインメントを参照してください。アラインメントを変更する場合には、__packed データ型属性、#pragma pack ディレクティブ

ブ、`#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ンメントをチェックする場合は、`__ALIGNOF__()` 演算子を使用します。

`__ALIGNOF__` 演算子は、オブジェクトのアラインメントの取得に使用できます。以下の 2 つのフォーマットのいずれかで指定します。

- `__ALIGNOF__ (type)`
- `__ALIGNOF__ (expression)`

2 番目のフォーマットの `expression` は評価されません。

- 匿名構造体と匿名共用体

C++ には、匿名共用体という機能があります。コンパイラでは、C プログラミング言語において、構造体と共用体の両方に対する同様の機能を使用できます。詳細については、185 ページの *匿名構造体と匿名共用体* を参照してください。

- ビットフィールドと非標準型

標準の C では、ビットフィールドの型は `int` か `unsigned int` でなければなりません。IAR C の言語拡張を使用することで、任意の整数型や列挙型を使用できます。これには、場合によって構造体のサイズが小さくなるという利点があります。詳細については、288 ページの *ビットフィールド* を参照してください。

- `static_assert()`

構造 `static_assert(const-expression, "message");` は、C/C++ で使用できます。この構造はコンパイル時に評価され、`const-expression` が偽であれば、`message` 文字列を含むメッセージが出力されます。

- 可変数引数マクロのパラメータ

可変数引数マクロは、`printf` スタイルの関数に相当するプリプロセッサマクロです。プリプロセッサは、引数なしで可変数引数マクロを受け入れます。つまり、... パラメータに一致するパラメータがない場合、`"`, `##_VA_ARGS__` マクロ定義でコンマが削除されます。標準の C では、... パラメータは少なくとも 1 つの引数と一致する必要があります。

専用 section 演算子

コンパイラは、以下の section 内蔵演算子を使用して、開始アドレスや終了アドレス、section の取得をサポートします。

<code>__section_begin</code>	指定の <code>section</code> またはブロックの最初のバイトアドレスを返します。
<code>__section_end</code>	指定の <code>section</code> またはブロックの後にある最初のバイトアドレスを返します。

`__section_size` 指定の `section` 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、リンカ構成ファイルで定義された指定のセクション `section` 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 `section` に配置したり、指定ブロックをリンカ構成ファイルで使用する場合、`section` 演算子を使用して、`section` またはブロックが配置されたメモリ範囲の開始アドレスと終了アドレスを取得できます。

指定の `section` は文字列リテラルである必要があり、`#pragma section` ディレクティブで先に宣言されている必要があります。`__section_begin` 演算子のタイプは、`void` へのポインタです。この組込み演算子を使用するには、言語拡張を有効にしておく必要があります。

これらの演算子は、専用の名前を持つシンボルとして実装され、以下の名前でリンカマップファイルに表示されます。

演算子	シンボル
<code>__section_begin(sec)</code>	<code>sec\$\$Base</code>
<code>__section_end(sec)</code>	<code>sec\$\$Limit</code>
<code>__section_size(sec)</code>	<code>sec\$\$Length</code>

表 20: セクション演算子とそのシンボル

これらの演算子を使用しない場合、リンカは同じ名前を持つセクションを連続して配置するとは限らない点に注意してください。これらの演算子（または同等のシンボル）を使用すると、`section` が指定のブロック内にあるかのようにリンカが動作します。これは、演算子に意味のある値が割り当てられるように、セクションが連続して配置されるためです。このことで、リンカ構成ファイルで指定した `section` 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 __huge *` です。

```
#pragma section="MYSECTION"
...
section_start_address = __section_begin("MYSECTION");
```

328 ページの *section*、322 ページの *location* も参照してください。

C 規格に対する緩和

このセクションでは、一部の C 規格の問題の一覧と、緩和について説明するとともに、重要度の低い構文の拡張についても解説します。

● 不完全型の配列

配列では、不完全な `struct` 型、`union` 型、`enum` 型をエレメントの型として使用できます。型は、配列が使用される場合はその前に、使用されない場合はコンパイル単位の終了までに完全にする必要があります。

● `enum` 型の前方宣言

拡張を使用して、`enum` の名前を先に宣言しておき、後で中括弧で囲んだリストを指定することでその名前を解決できます。

● `struct` や `union` 指定子末尾にセミコロンがなくても受け入れる。

`struct` や `union` 指定子の末尾にセミコロンがないと、ワーニング（エラーの代わりとして）が出力されます。

● `NULL` と `void`

ポインタの処理において、`void` へのポインタは必要に応じて別の型に暗黙的に変換されます。また、`NULL` ポインタ定数は、必要に応じて適切な型の `NULL` ポインタに暗黙的に変換されます。C 規格では、一部の演算子でこのような動作が可能ですが、他の演算子ではこれはできません。

● 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、295 ページの *キャスト* を参照してください。

● レジスタ変数のアドレスの取得

C 規格では、レジスタ変数として指定した変数のアドレスを取得することは不正です。コンパイラではこれは可能ですが、ワーニングが出力されます。

● `long float` は `double` を意味します。

`long float` 型は、`double` 型の同義語として扱われます。

● `typedef` 宣言の繰返し

同一スコープ内で `typedef` を繰り返し宣言することは可能ですが、ワーニングが出力されます。

- ポインタ型の混在

交換可能だが同一ではない型へのポインタ間 (`unsigned char *` と `char *` など) で代入、差分計算を行うことが可能です。これには、同一サイズの整数型へのポインタが含まれます。ワーニングが出力されます。

文字列リテラルを任意の種類の文字へのポインタに代入することは可能であり、ワーニングは出力されません。

- トップレベルでない `const`

ポインタの代入は、代入先の型に、トップレベルでない型修飾子が追加されている場合には可能です (`int **` から `int const **` への代入など)。また、このようなポインタの差分の比較および取得も可能です。

- `-lvalue` 以外の配列

`lvalue` 以外の配列式は、使用時に配列の最初のエレメントへのポインタに変換されます。

- プリプロセッサディレクティブ終了後のコメント

この拡張は、プリプロセッサディレクティブの後にテキストを配置できるようにするもので、厳密な C 規格モードを使用していない場合に有効になります。この言語拡張の目的は、レガシーコードのコンパイルをサポートすることであり、このフォーマットで新しいコードを記述することは推奨しません。

- `enum` リスト最後の余分なカンマ

`enum` リストの最後に、余分なカンマを付けてもかまいません。厳密な C 規格モードでは、ワーニングが出力されます。

- `}` の前のラベル

C 規格では、ラベルに続けて少なくとも 1 つの文を記述する必要があります。したがって、ラベルをブロックの最後に配置するのは不正になります。コンパイラはこれを許可しますが、ワーニングが出力されます。

これは、`switch` 文のラベルについても同様です。

- 空白の宣言

空白の宣言 (セミコロンのみ) は可能ですが、リマークが出力されます (リマークが有効な場合)。

- 単一の値の初期化

C 規格では、静的な配列、`struct`、`union` のイニシャライザ式は、すべて中括弧で囲む必要があります。

単一の値のイニシャライザは、中括弧なしで記述できますが、ワーニングが出力されます。コンパイラは次の式を受け入れます。

```
struct str
{
    int a;
} x = 10;
```

- 他のスコープでの宣言

他のスコープでの外部 / 静的宣言は可視になります。以下の例では、変数 *y* は *if* 文の本体でのみ可視になるべきですが、関数の最後で使用できます。ワーニングが出力されます。

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- 関数をコンテキストとして持つ文字列への関数名の拡張

シンボル `__func__` または `__FUNCTION__` を関数本体の中で使用すると、そのシンボルが現在の関数名を持つ文字列に拡張されます。

また、シンボル `__PRETTY_FUNCTION__` を使用すると、パラメータ型とリターン型も含まれます。シンボル `__PRETTY_FUNCTION__` を使用した場合の結果は、以下の例のようになります。

```
"void func(char)"
```

これらのシンボルは、アサーションやその他のトレースユーティリティに便利です。これらは、言語拡張が有効化されていることが必要です (232 ページの *-e* を参照)。

- 関数およびブロックスコープ内の静的関数

静的関数を関数およびブロックスコープ内で宣言可能。宣言はファイルスコープに移動します。

- 数値の構文に従って数値の走査が行われる

数値は、`pp-number` 構文ではなく、数値の構文に従って走査されます。このため、`0x123e+1` は 1 つの有効なトークンではなく、3 つのトークンとして走査されます。(`--strict` オプションを使用する場合、代わりに `pp-number` 構文が使用されます)。

C++ の使用

IAR システムズは C++ 言語をサポートしています。標準の C++、業界標準の Embedded C++、拡張 Embedded C++ の規格から選択できます。ここでは、C++ 言語を使用する際の注意事項について説明します。

概要 — EC++ および EEC++

Embedded C++ は、ISO/IEC C++ 標準の正当なサブセットで、組み込みシステムのプログラミング用に設計されています。業界団体である Embedded C++ Technical Committee により定義されています。組み込みシステム開発においてはパフォーマンスと移植性が特に重要であり、言語の定義時には、このことが考慮されています。EC++ には C++ と同じオブジェクト指向の利点がありますが、予測が付きにくいコードサイズや実行時間の増加につながる一部の機能はありません。

EMBEDDED C++

以下の C++ 機能がサポートされています。

- クラス。データ構造と動作の両方をまとめたユーザ定義の型のことです。本質的な機能である継承により、データ構造と動作を複数のクラスで共有できます
- ポリモフィズム。1 つの処理が異なるクラスで異なる動作を実現できる機能です。仮想関数によって提供されます
- 演算子と関数名のオーバーロード。引数リストが明確に異なる場合に、名前が同一の演算子や関数を複数使用できます
- 型安全なメモリ管理。演算子 new、delete を使用します
- インライン関数。特にインライン展開に適しています

プログラマによる制御が不可能なオーバーヘッドが、実行時間やコードサイズに生じるような C++ 機能は除外されています。他に除外されているのは、C++ 標準が定義される直前に追加された機能です。したがって、Embedded C++ は、効率性が高く、既存の開発ツールで完全にサポートされている C++ のサブセットを提供します。

Embedded C++ は、以下の C++ の機能が削除されています。

- テンプレート
- 多重継承と仮想継承
- 例外処理

- ランタイムの型情報
- 新しいキャスト構文（演算子 `dynamic_cast`、`static_cast`、`reinterpret_cast`、`const_cast`）
- 名前空間
- `mutable` 属性

これらの言語機能の除外により、ランタイムライブラリの効率性が大幅に向上しています。他にも、Embedded C++ ライブラリはフル C++ ライブラリと以下の点で異なります。

- 標準テンプレートライブラリ (STL) が除外されています
- テンプレートを使用せずにストリーム、文字列、複素数をサポートしています
- 例外処理、ランタイムの型情報 `except`、`stdexcept`、`typeid` の各ヘッダに関連するライブラリ機能が除外されています

注： Embedded C++ は名前空間をサポートしていないため、ライブラリは `std` 名前空間中には存在しません。

拡張 EMBEDDED C++

IAR システムズの拡張 EC++ は、標準の EC++ に以下の機能を追加した C++ のサブセットです。

- フルテンプレートサポート
- 多重継承と仮想継承
- 名前空間のサポート
- `mutable` 属性
- キャスト演算子 `static_cast`、`const_cast`、`reinterpret_cast`

これらの追加機能は、C++ 規格に準拠しています。

拡張 EC++ をサポートするため、本製品には、標準テンプレートライブラリ (STL) が付属しています。STL には C++ 標準チャプタユーティリティ、コンテナ、イテレータ、アルゴリズム、数値が含まれています。この STL は、拡張 EC++ 言語の使用に合わせて変更されており、例外処理、多重継承、ランタイムの型情報 (`rtti`) をサポートしていません。また、ライブラリは `std` 名前空間には存在しません。

注： 拡張 EC++ を有効化してコンパイルされたモジュールは、拡張 EC++ を有効化せずにコンパイルされたモジュールと完全なリンク互換性を持ちます。

概要 — 標準 C++

IAR C++ 実装は、ISO/IEC 1488:2003 C++ 標準に完全に準拠しています。本ガイドでは、この規格を C++ といいます。

EC++ や EEC++ ではなく標準の C++ を使用する主な理由は、以下のいずれかの必要性がある場合です。

- 例外のサポート
- ランタイム型情報 (RTTI) のサポート
- 標準の C++ ライブラリ (EC++ ライブラリは C++ ライブラリの簡易版で、ストリームと文字列はテンプレートではありません)

コードサイズが重要で、アプリケーションにこれらの機能が必要なければ、EC++ (または EEC++) の方が適しています。

例外および RTTI サポートのモード

例外とランタイム型の情報がアプリケーションにインクルードされることによって、コードサイズは増加します。サイズの増加を避けるために、以下のどちらか一方または両方を無効にした方がいい場合もあります。

- ランタイム型情報のコンストラクトのサポートは、コンパイラオプション `--no_rtti` を使用すれば無効にできます
- 例外のサポートは、コンパイラオプション `--no_exceptions` を使用して無効にできます

コンパイル中にサポートが有効な場合でも、リンカは余分なコードやテーブルの最終アプリケーションへのインクルードを避けることができます。アプリケーションで例外が発生しない場合、例外の使用をサポートするコードやテーブルは、アプリケーションイメージにインクルードされません。また、動的ランタイム型情報コンストラクト (`dynamic_cast/typeid`) がポリモフィズム型と併用されない場合、それらのサポートに必要なオブジェクトは、アプリケーションのコードイメージにインクルードされません。この動作を制御するには、リンカオプション `--no_exceptions`、`--force_exceptions`、`--no_dynamic_rtti_elimination` を使用します。

例外サポートを無効にする

コンパイラオプション `--no_exceptions` を使用する場合、以下によってコンパイ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、243 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、245 ページの `--no_rtti` を参照してください。

例外処理

例外処理は以下の 3 つの部分に分けることができます。

- 例外の発生メカニズム — C++ では throw 式および rethrow 式です
- 例外のキャッチメカニズム — C++ では try-catch 文や関数の例外仕様、main から例外がリークするのを防ぐための暗黙的キャッチです
- 現在アクティブな関数についての情報 — try-catch 文と自動オブジェクトのセットがあって、例外が関数を介して伝播されるときにそれらのデストラクタを実行する必要がある場合

例外が引き起こされると、関数の呼出しスタックが関数およびブロックごとに巻き戻されます。それぞれの関数やブロックについて、破棄が必要な自動オブジェクトのデストラクタが実行され、例外のキャッチハンドラがあるかどうかチェックが行われます。ある場合は、そのキャッチハンドラから実行が続けられます。

C++ コードをアセンブラおよび C コードと併用するアプリケーション、およびアセンブラルーチンと C 関数を介して、ある C++ 関数から別の関数へ例外をスローするアプリケーションは、リンカオプション `--exception_tables` と引数 `unwind` を併用する必要があります。

例外の実装

例外はテーブル方式を使用して実装されます。それぞれの関数について、テーブルで以下を記述します。

- 関数の巻き戻し方法。つまり、スタック上で呼出し元を探して復元が必要なレジスタを復元する方法です
- 関数にどのキャッチハンドラがあるのか
- 関数に例外仕様があるかどうか、どの例外の伝播が許可されているか
- デストラクタを実行する必要がある自動オブジェクトのセット

例外が引き起こされると、ランタイムは2つのフェーズで進行します。最初のフェーズは例外テーブルを使用して、スタックの巻き戻しをその時点で停止させるキャッチハンドラまたは例外仕様を含む関数呼出しのスタックを検索します。このポイントが見つければ、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使用量でのオーバー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成本のかかる処理になります。

例外の結果によるスタックの巻き戻し中の自動オブジェクトの破棄は、通常の関数の処理を扱うコードとは別にコードに実装されます。このコードはキャッチハンドラのコードとともに、通常のコード（本来は .text に配置）とは別のセクション(.exc.text)に配置されます。場合によっては、たとえば高速と低速のROMメモリがある場合、リンカ設定ファイルにセクションを配置する際に、この違いに基づいて選択すると有益なことがあります。

C++ および派生言語のサポートを有効にする



コンパイラでは、デフォルトの言語はCです。

標準のC++で記述されたファイルをコンパイルするには、`--c++` コンパイラオプションを使用する必要があります。227 ページの `--c++` を参照してください。

Embedded C++で記述されたファイルをコンパイルするには、`--ec++` コンパイラオプションを使用する必要があります。233 ページの `--ec++` を参照してください。

拡張 Embedded C++ の機能をソースコードで利用するには、`--eec++` コンパイラオプションを使用する必要があります。233 ページの `--eec++` を参照してください。



IDEでEC++EEC++、またはC++を有効にするには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語] を選択して、適切な標準を選びます。

C++ および EC++ の機能の説明

IAR C/C++ Compiler 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インタと互換性のある属性しか持つことができません。コンストラクタとデストラクタはこうした属性を持つことはできません。

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

例

```
class MyClass
{
public:
    // 静的変数の場所を __memattr メモリ内のアドレス 60 に指定します
    static __no_init int mI @ 60;

    // 静的 Thumb 関数
    static __thumb void F();

    // A Thumb 関数
    __thumb void G();

    // インタワーク
    virtual __arm void ArmH();

    // インタワーク
    virtual __thumb void ThumbH();

    // 仮想関数の場所を SPECIAL に指定します
    virtual void M() const volatile @ "SPECIAL";
};
```

関数型

extern "C" リンケージを持つ関数型は、C++ リンケージを持つ関数と互換性があります。

例

```
extern "C"
{
    typedef void (*FpC)(void);    // C 関数 typedef
}

typedef void (*FpCpp)(void);    // C++ 関数 typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
    MyF(F1);                      // 常に機能する
    MyF(F2);                      // fpCpp は fpC と互換
}
```

割込みで静的クラスオブジェクトを使用する

割込み関数が、(コンストラクタを使用して) 生成または (デストラクタを使用して) 破棄しなければならない静的クラスオブジェクトを使用する場合、オブジェクトの生成前または破棄の後に割込みが発生すると、アプリケーションが正しく機能しなくなります。

これを回避するために、静的オブジェクトが構築されるまで、これらの割込みが有効になっておらず、main から戻ったり、exit を呼び出すときに無効になっていることを確認します。システムの起動について詳しくは、104 ページのシステムの起動と終了を参照してください。

関数ローカルな静的クラスオブジェクトは、実行が最初に宣言を通過すると生成され、main から戻ったり、exit を呼び出す際に破棄されます。

新しいハンドラを使用する

メモリの消耗に対応するには、set_new_handler 関数を使用します。

set_new_handler を呼び出さないか、NULL 新規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 operator new が十分なメモリの割当てに失敗すると、operator new が std::bad_alloc をスローします。例外が有効でない場合、operator new は代わりに abort を呼び出します。

NULL でない新規ハンドラを用いて `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新規ハンドラが呼び出されます。新規ハンドラはより多くのメモリを使用できるようにして、何らかの形で実行を返すか、中止する必要があります。例外が有効な場合、新規ハンドラは `std::bad_alloc` 例外をスローすることもできます。`operator new` の派生型である `nothrow` は、例外が有効で新規ハンドラが `std::bad_alloc` をスローする場合に、新規ハンドラがある状態でのみ NULL を返します。

テンプレート

C++ および拡張 EC++ は、C++ 標準に基づいてテンプレートをサポートしますが、`export` キーワードはサポートしません。実装では、2 段階のルックアップを使用します。すなわち、必要なときには常に `typename` キーワードを挿入する必要があります。さらに、テンプレートを使用するたびに、使用可能なすべてのテンプレート定義が可視になる必要があります。すなわち、すべてのテンプレートの定義がインクルードファイルまたは実際のソースファイルに存在する必要があります。

C-SPY でのデバッグサポート

C-SPY® は、C++ の一部の機能についてデバッグサポートを提供しています。

- C-SPY には、STL コンテナ用の内蔵ディスプレイ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。
- `throw` 文の位置や、引き起こされた例外に対応する `catch` 文がない場合に、C-SPY を停止させることができます。

これらの詳細は、『ARM® 用 C-SPY® デバッガガイド』を参照してください。

EEC++ の機能の説明

ここでは、拡張 EC++ と EC++ で大きく異なる機能について説明します。

テンプレート

標準テンプレートライブラリ

製品に付属の STL（標準テンプレートライブラリ）は、拡張 EC++ 用に調整されています（158 ページの *拡張 Embedded C++* を参照）。

キャスト演算子の派生形

拡張 EC++ では、以下の C++ キャスト演算子の派生形を使用できます。

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

MUTABLE

拡張 EC++ では mutable 属性がサポートされています。クラスオブジェクト全体が const である場合でも、mutable シンボルを変更できます。

名前空間

名前空間機能は、拡張 EC++ でのみサポートされています。すなわち、名前空間を使用してコードを分割できます。ただし、ライブラリそのものは std 名前空間には配置されません。

STD 名前空間

std 名前空間は、標準 EC++ と拡張 EC++ のいずれでも使用されません。std 名前空間内のシンボルを参照するコードがある場合は、以下の例のように std を無定義とします。

```
#define std
```

使用するアプリケーション内の識別子が、ランタイムライブラリの識別子を妨害しないように注意する必要があります。

EC++ および C++ の言語拡張

コンパイラをいずれかの C++ モードで使用し、IAR 言語拡張を有効にする場合、以下の C++ 言語拡張がコンパイラで使用できます。

- クラスの friend 宣言において、class キーワードを省略できます。以下に例を示します。

```
class B;
class A
{
    friend B;           // IAR 言語拡張を用いると
                        // 可能
    friend class B;    // 標準規格に従った書き方
};
```

- スカラ型の定数は、クラス内で定義できます。以下に例を示します。

```
class A
{
    const int mSize = 10; //IAR 言語拡張を用いると
                          // 可能

    int mArr[mSize];
};
```

規格では、初期化した静的データメンバを代わりに使用することになっています。

- クラスメンバの宣言において、修飾名を使用できます。以下に例を示します。

```
struct A
{
    int A::F(); //IAR 言語拡張を用いると可能
    int G();    // 標準規格に従った書き方
};
```

- C リンケージ (extern "C") を持つ関数のポインタと、C++ リンケージ (extern "C++") を持つ関数のポインタとの間の暗黙的な型変換の使用が許可されています。以下に例を示します。

```
extern "C" void F(); //C リンケージを持つ関数
void (*PF)()         //pf は C++ リンケージを持つ関数を指す
                    = &F;    // ポインタの暗黙の変換
```

規格では、ポインタは明示的に変換する必要があります。

- ? 演算子を含む構造体の 2 番目または 3 番目のオペランドが文字列リテラルまたはワイド文字列リテラル (C++ の場合の定数) の場合、オペランドを暗黙的に char * または wchar_t * に変換できます。以下に例を示します。

```
bool X;

char *P1 = X ? "abc" : "def";          //IAR 言語拡張を用いると
                                        // 可能
char const *P2 = X ? "abc" : "def"; // 標準規格に従った書き方
```

- 関数パラメータに対するデフォルトの引数は、規格に従ったトップレベルの関数宣言ではなく、typedef 宣言の中、関数へのポインタの関数宣言の中、メンバへのポインタの関数宣言の中でも指定できます。
- 非静的ローカル変数を含む関数、評価されない式 (sizeof 式など) を含むクラスにおいては、式から非静的ローカル変数を参照できます。ただし、ワーニングが出力されます。

- `typedef` 名によって、含有クラスに匿名共用体を導入できます。最初に共用体を宣言する必要はありません。以下に例を示します。

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

class A
{
public:
    U; // OK -- A::i および A::j への参照が許可されています
};
```

また、この拡張は *anonymous classes* と *anonymous structs* も許可します。ただし、C++ の機能がなく（たとえば、静的データメンバやメンバ関数を持たず、パブリックでないメンバがないなど）、他の匿名クラスや構造体、共用体以外のネスト型を持たないことが条件です。次に例を示します。

```
struct A
{
    struct
    {
        int i,j;
    }; // OK -- A::i および A::j への参照が許可されています
};
```

- `friend class` の構文では、`nonclass` 型のほか、精密型名なしに `typedef` によって表現されたクラス型も使用可能です。次に例を示します。

```
typedef struct S ST;

class C
{
public:
    friend S; // OK (S がスコープ内にある必要あり)
    friend ST; // OK ("friend S;" と同じ)
    // friend S const; // エラー、cv-qualifiers は直接
                        // 現れることはできません
};
```

注：最初に言語拡張を有効にせずに、これらの構造体のいずれかを使用すると、エラーが出力されます。

アプリケーションに関する考慮事項

この章では、組込みアプリケーションの開発に関連する特定の範囲のアプリケーション問題について説明します。

通常、この章では、コンパイラまたはリンクのみに特別には関連しない問題は強調表示されています。

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き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。

- 通常のバイナリ
- Motorola S-records
- Intel hex

注：`ielftool` は、絶対イメージ内のチェックサムの埋め込みや計算など、別のタイプの変換にも使用できます。

`ielftool` のソースコードは、`arm/src` ディレクトリにあります。`ielftool` の詳細は、427 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

スタックは、関数がローカルで使用する変数やその他の情報を保存するために使用されます（「データ記憶」を参照）。スタックは、メモリの連続したブロックであり、プロセッサのスタックポインタレジスタ `SP` によって位置が示されます。

スタックを保持するために使用するデータセクションを、`CSTACK` といいます。システム起動コードは、スタックポインタをスタックセグメントの最後の位置に初期化します。

スタックサイズについて

コンパイラは、内部データスタックをさまざまなユーザアプリケーション処理に使用します。必要なスタックサイズは、これらの処理の内容によって大きく異なります。スタックサイズが大きすぎる場合は、RAMが無駄に消費されます。スタックサイズが小さすぎる場合には、2つのことが発生します。これは、スタックのメモリ上の位置により異なり、いずれの場合もアプリケーション障害が発生します。これらの障害は、変数記憶領域が上書きされ動作が不安定になるか、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かのいずれかです。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、82 ページの *スタックの設定*、198 ページの *スタックエリアと RAM メモリの節約*を参照してください。

スタックのアラインメント

デフォルトの `cstartup` コードは、すべてのスタックを自動的に 8 バイトに整列されたアドレスに初期化します。

スタックの整列の詳細については、136 ページの *呼出し規約*、138 ページの *専用レジスタ*、139 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

ARM アーキテクチャでは、さまざまな例外が発生したときに切り替わる 5 種類の例外モードがサポートされます。各例外モードには、システム / ユーザモードスタックの破損を回避するための独自のスタックがあります。

以下の表に、さまざまな例外スタックの推奨スタック名を示します。ただし、スタックには任意の名前を付けることができます。

プロセッサのモード	推奨スタックセクション名	説明
Supervisor	SVC_STACK	オペレーティングシステムスタック。
IRQ	IRQ_STACK	汎用 (IRQ) 割込みハンドラのスタック。
FIQ	FIQ_STACK	高速 (FIQ) 割込みハンドラのスタック。
未定義	UND_STACK	未定義命令割込みのスタック。ハードウェアコプロセッサおよび命令セット拡張のソフトウェアエミュレーションをサポートします。
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込みハンドラのスタック。

表 21: 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。

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。



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、その関連関数の 1 つ）か C++ の演算子 `new` を使用して割り当てられた動的データが格納されます。

アプリケーションで動的メモリ割当てを使用する場合には、以下の内容に精通しておく必要があります。

- ヒープに使用されるリンカセクション
- ヒープサイズの割当て。詳細については、82 ページの *ヒープの設定* を参照してください

ヒープに割り当てられるメモリは、セクション `HEAP` にあります。これは、動的メモリ割当てが実際に行われた場合のみアプリケーションに含まれます。



ヒープサイズと標準 I/O

通常の設定のように FILE 記述子を DLIB ランタイムライブラリから除外すると、I/O バッファが無効になります。詳細設定のように、それ以外の場合は、`stdio` ライブラリのヘッダファイルで I/O バッファが 512 バイトに設定されます。ヒープが小さすぎる場合は、I/O がバッファされず、I/O がバッファされた場合よりも大幅に低速になります。IAR C-SPY® デバッガのシミュレータドライバを使用してアプリケーションを実行する場合には、速度低下が現れない可能性があります。アプリケーションを ARM コアで実行すると、速度低下を明確に認識できます。標準 I/O ライブラリを使用する場合は、ヒープサイズを標準 I/O バッファの必要に応じたサイズに設定してください。

ツールとアプリケーション間の相互処理

リンクプロセスとアプリケーションでシンボルを相互処理する方法は以下の4種類があります。

- **ILINK** 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。**ILINK** は、アプリケーションがラベル、サイズ、デバッグのセットアップなどとして使用できるパブリック絶対定数シンボルを作成します。
- コマンドラインオプション `--config_def` または設定ディレクティブ `define symbol` を使用し、`export symbol` ディレクティブを使用しシンボルをエクスポートして、エクスポート済み設定シンボルを作成する。**LINK** は、アプリケーションがラベル、サイズ、デバッグ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`、`__section_size`、またはアセンブラ演算子 `SFB`、`SFE`、`SIZEOF` を指定のセクションまたはブロックで使用する。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、またはリンカ構成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 **ILINK** に通知します。これは、**ILINK** により、実行の開始位置をデバッグに通知するときのルートシンボルとして使用されます。

以下の行は、上記のメカニズムを使用する方法を示します。以下のオプションをコマンドラインに追加します。

```
--define_symbol NrOfElements=10
--config_def HeapSize=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 HeapSize;

/* 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 = HeapSize, alignment = 8 {};

place in RAM { block MyHEAP };
```

以下の行をアプリケーションソースコードに追加します。

```
#include <stdlib.h>

/* ILINK オプションで定義したシンボルを使い、指定したサイズのエレメント配列
を動的に割り当てます。値はラベルの形式をとります。
*/
extern int NrOfElements;

typedef char Elements;
Elements * GetElementArray()
{
    return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* ILINK オプションで定義したシンボルを使う。
 * リンカ設定ファイル内のシンボルはアプリケーションで使用できるようになって
いる。
*/
extern char HeapSize;

/* 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) &HeapSize; ++i)
    {
        p[i] = 0;
    }
    return p;
}
```

チェックサムの計算

IAR ELF Tool (ielftool) は、特定の範囲のメモリをパターンで埋め、これらの範囲のチェックサムを計算します。計算されるチェックサムは、入力 ELF イメージの既存シンボルの値を置換します。アプリケーションは、これらの範囲が変更されていないか検証できます。

チェックサムを使用してアプリケーションの整合性を検証する場合、以下のことを行う必要があります。

- ielftool により計算されるチェックサムについて、配置に関連する名前とサイズとともに予約する
- チェックサムアルゴリズムを選択し、その ielftool を設定して、アルゴリズムのソースコードをアプリケーションに含める
- アプリケーションソースコードで ielftool とそのソースコードの両方を検証および設定するメモリ範囲を決定する



注：IDE で ielftool を設定するには、[プロジェクト] > [オプション] > [リンカ] > [チェックサム] を選択します。

チェックサムの計算

この例では、0x8002 ～ 0x8FFF にある ROM メモリのチェックサムが計算されます。また、計算された 2 バイトチェックサムが 0x8000 に配置されます。

計算されるチェックサムの配置の作成

計算されるチェックサムの配置は、2 とおりの方法で作成できます。特定のセクション（この例では .checksum）に常駐する正しいサイズのグローバル C/C++ またはアセンブラ定数シンボルを作成する方法と、リンカオプション --place_holder を使用する方法です。

たとえば、シンボル __checksum の 2 バイトスペースをセクション .checksum にアラインメント 4 で作成するには、以下のようになります。

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

注：.checksum セクションは、必要と思われる場合に、アプリケーションにのみインクルードされます。アプリケーション自体でチェックサムが必要でない場合、リンカオプション --keep=__checksum か、リンカディレクティブ keep を使用して、セクションを強制的にインクルードすることができます。

.checksum セクションを配置するには、リンカ設定ファイルを修正する必要があります。これを次に示します（ブロック CHECKSUM の扱いに注意してください）。

```
define memory Mem with size = 4G;

define region ROM_region = Mem:[from 0x8000 to 0x80000000 - 1];
define region RAM_region = Mem:[from 0x80000000 to 0x100000000 - 2];

initialize by copy { rw };
do not initialize { section .noinit };
```

```

define block HEAP          with alignment = 8, size = 16M {};
define block CSTACK       with alignment = 8, size = 16K {};
define block IRQ_STACK    with alignment = 8, size = 16K {};
define block FIQ_STACK     with alignment = 8, size = 16K {};

define block CHECKSUM      { ro section .checksum };
place at address Mem:0x0 { ro section .intvec};
place in ROM_region        { ro, first block CHECKSUM };
place in RAM_region { rw, block HEAP, block CSTACK, block
                        IRQ_STACK, block FIQ_STACK };

```

ielftool の実行

チェックサムを計算するには、ielftool を実行します。

```

ielftool --fill=0x00;0x8000-0x8FFF
--checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out
destinationFile.out

```

チェックサムを計算するには、フィル操作を定義する必要があります。この例では、フィルパターン 0x0 が使用されます。使用されるチェックサムアルゴリズムは **crc16** です。

ielftool では、**--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。**--strip** リンカオプションを使用する場合、これを削除し、代わりに **--strip ielftool** オプションを使用します。

チェックサム関数をソースコードに追加する

ielftool により生成されるチェックサムの値をチェックするには、アプリケーションにより計算されたチェックサムと比較する必要があります。つまり、チェックサム計算用の関数（ielftool と同じアルゴリズムを使用）をアプリケーションソースコードに追加する必要があります。アプリケーションには、この関数への呼出しを含める必要があります。

チェックサム計算用の関数

以下の関数（計算時間は遅いがメモリ使用量は少ない版）では、`crc16` アルゴリズムが使用されます。

```
unsigned short SlowCrc16(unsigned short sum,
                        unsigned char *p,
                        unsigned int len)
{
    while (len--)
    {
        int i;
        unsigned char byte = *(p++);

        for (i = 0; i < 8; ++i)
        {
            unsigned long oSum = sum;
            sum <<= 1;
            if (byte & 0x80)
                sum |= 1;
            if (oSum & 0x8000)
                sum ^= 0x1021;
            byte <<= 1;
        }
    }
    return sum;
}
```

チェックサムアルゴリズムのソースコードは、製品インストールの `arm\src\linker` ディレクトリにあります。

チェックサムの計算

以下のコードは、チェックサムがどのように計算されるかを示した例です。

```
/* チェックサム範囲の最初と最後のアドレス */
unsigned long ChecksumStart = 0x8000+2;
unsigned long ChecksumEnd = 0x8FFF;

/* ielftool によって計算されるチェックサム
 * （これはアドレス 0x8000 上にあります）
 */
extern unsigned short const __checksum;

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



```

/* チェックサム計算の実行 */
calc = slow_crc16(0,
                  (unsigned char *) ChecksumStart,
                  (ChecksumEnd - ChecksumStart+1));

/* 結果を格納 */
calc = slow_crc16(calc, zeros, 2);

/* チェックサムのテスト */
if (calc != __checksum)
{
    abort();    /* 失敗 */
}
}

```

注意事項

チェックサムを計算する場合、以下のことに注意してください。

- チェックサムは、すべてのメモリ範囲で最下位アドレスから最上位アドレスに計算する必要があります
- 各メモリ範囲は、定義されている順序で検証する必要があります
- 1つのチェックサムに対して複数の範囲が存在することは問題ありません
- 複数のチェックサムが使用される場合、セクションごとに一意のシンボル名を使用する必要があります
- 低速の関数派生型が使用される場合、チェックサム計算の最後の呼出しは、チェックサム内のバイト数と同じバイト数（値 0x00）で行う必要があります

詳細については、427 ページの *IAR ELF ツール* — *ieelftool* を参照してください。

C-SPY に関する注意事項

デフォルトでは、リンカオプション `--place_holder` を使用してメモリ内に割り当てたシンボルは、**C-SPY** によって `int` 型であると見なされます。チェックサムのサイズが `int` のサイズとは異なる場合、そのサイズに合わせてチェックサムシンボルの表示フォーマットを変更できます。



[**C-SPY** ウォッチ] ウィンドウでシンボルを選択し、コンテキストメニューから [**表示フォーマット**] を選択します。チェックサムシンボルのサイズに合った表示フォーマットを選択します。

リンカの最適化

仮想関数の除去

仮想関数除去 (VFE) は、不要な仮想関数と動的ランタイム型情報を除去するリンカの最適化です。

仮想関数除去が機能するためには、仮想関数テーブルについての情報や、どの仮想関数が呼び出されるか、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出しを実行せず、仮想関数テーブルを定義しないことが分かっている場合、`--vfe=forced` リンカオプションを使用して、仮想関数除去を有効にできます。

現在は IAR システムズおよび RealView のツールが、リンカで使用可能な形で仮想関数除去に必要な情報を提供しています。

仮想関数除去は、`--no_vfe` リンカオプションを使用して完全に無効にすることができます。この場合、VFE 情報を持たないモジュールについてワーニングは出力されません。

詳細については、283 ページの `--vfe`、278 ページの `--no_vfe` を参照してください。

AEABI への準拠

ARM 用 IAR ビルドツールは、ARM Limited が提唱する ARM Embedded Application Binary Interface (AEABI) をサポートしています。このインタフェースは、Intel IA64 ABI インタフェースに基づいています。AEABI に準拠すると、他のベンダにより提供されるツールで生成されていても、モジュールを他の任意の AEABI 準拠モジュールとリンクできるというメリットがあります。

ARM 用 IAR ビルドツールは、AEABI の以下のパートをサポートしています。

AAPCS	ARM アーキテクチャのプロシージャ呼出し標準
CPPABI	ARM アーキテクチャの C++ ABI (EC++ パーツのみ)
AAELF	ARM アーキテクチャの ELF
AADWARF	ARM アーキテクチャの DWARF
RTABI	ARM アーキテクチャのランタイム ABI
CLIBABI	ARM アーキテクチャの C ライブラリ ABI

IAR ビルドツールは、明示的なオペレーティングシステムがない ROM ベースシステムのみをサポートします。

注：

- AEABI は C89 専用です
- IAR ビルドツールは、デフォルトおよび C ロケールの使用のみをサポートします
- AEABI は、C++ ライブラリとの互換性を指定しません
- enum および wchar_t のいずれのサイズも、AEABI では一定ではありません

AEABI 準拠が有効な場合、システムヘッダファイルで実行されるほとんどすべての最適化が無効になり、特定のプロセッサ定数が、実定数の変数になります。

IAR ILINK リンカを使用して AEABI 準拠モジュールをリンクする

IAR ILINK リンカを使用してアプリケーションを構築する場合、以下のタイプのモジュールを組み合わせることができます。

- IAR ビルドツールを使用して生成されたモジュール（AEABI 準拠モジュールと AEABI 準拠でないモジュールの両方）
- 別のベンダのビルドツールを使用して生成された AEABI 準拠モジュール

注：別のベンダのコンパイラで生成されたモジュールをリンクするには、そのベンダの追加サポートライブラリが必要になることがあります。

IAR ILINK リンカでは、オブジェクトファイルに含まれる属性に基づき、使用する適切な標準 C/C++ ライブラリが自動的に選択されます。インポートされるオブジェクトファイルには、これらのすべての属性が含まれないことがあります。そのため、このような場合、以下の 1 つ以上の項目を検証して、ILINK による標準ライブラリの選択をサポートする必要があります。

- 使用する CPU (--cpu リンカオプションを指定)
- フル I/O が必要な場合、フルライブラリ設定の標準ライブラリとリンクする必要がある
- --no_library_search リンカオプションとの組み合わせが可能な、ランタイムライブラリファイルを明示的に指定する

リンクする際は、仮想関数除去についても考慮してください（178 ページの仮想関数の除去を参照）。

サードパーティ製リンカを使用して AEABI 準拠のモジュールをリンクする

IAR C/C++ コンパイラを使用してモジュールを生成し、このモジュールを別のベンダのリンカを使用してリンクする場合、このモジュールは AEABI 準拠モジュールでなければなりません（180 ページの *AEABI 準拠をコンパイラで有効にする* を参照）。

また、このモジュールが任意の IAR 固有コンパイラ拡張を使用する場合、これらの機能が他のベンダのツールによりサポートされている必要があります。特に以下のことに注意してください。

- 以下の拡張のサポートを検証する必要があります。#pragma pack、__no_init、__root、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、__swi、__irq、__fig、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 と --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>
#ifdef _AEABI_PORTABLE
    #error "header.h not AEABI compatible"
#endif
```

CMSIS の統合

arm¥CMSIS サブディレクトリには、CMSIS (ARM Cortex Microcontroller Software Interface Standard) および CMSIS DSP ヘッダとライブラリファイル、ドキュメントが含まれます。CMSIS について詳しくは、<http://www.arm.com/cmsis> をご覧ください。

特殊なヘッダファイル inc¥c¥cmsis_iar.h が、現行バージョンの IAR C/C++ コンパイラの CMSIS 版として用意されています。

CMSIS DSP ライブラリ

IAR Embedded Workbench には、arm¥CMSIS¥Lib¥IAR ディレクトリにビルド済の CMSIS DSP ライブラリが用意されています。ライブラリファイルの名前は次のように作成されます：

```
iar_cortexM<0|3|4><1|b>[v]_math.a
```

<0|3|4> は Cortex-M 派生品、<1|b> はバイトオーダーをそれぞれ選択し、[v] はライブラリが FPU (Cortex-M4 のみ) 用にビルドされていることを示します。

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ビルドする際のコマンドラインの例を示します。

CMSIS のみ (DSP ライブラリなし)

```
iccarm -I $EW_DIR¥arm¥CMSIS¥Include
```

DSP ライブラリあり、Cortex-M4、リトルエンディアン、FPU あり

```
iccarm --endian=little --cpu=Cortex-M4 --fpu=VFPv4_sp -I  
$EW_DIR¥arm¥CMSIS¥Include -D ARM_MATH_CM4
```

```
ilinkarm $EW_DIR¥arm¥CMSIS¥Lib¥IAR¥iar_cortexM4lv_math.a
```

DSP ライブラリあり、Cortex-M3、リトルエンディアン

```
iccam --endian=little --cpu=Cortex-M3 -I  
$EW_DIR$Yarm¥CMSIS¥Include -D ARM_MATH_CM3  
  
ilinkarm $EW_DIR$Yarm¥CMSIS¥Lib¥IAR¥iar_cortexM3l_math.a
```



IAR EMBEDDED WORKBENCH での CMSIS を使用したビルド

[プロジェクト] > [オプション] > [一般オプション] > [ライブラリ構成]
を選択して、CMSIS のサポートを有効にします。

有効にすると、CMSIS のインクルードパスと DSP ライブラリが自動的に使用されます。詳細については、『*ARM® 用 IDE プロジェクト管理およびビルドガイド*』を参照してください。

組込みアプリケーション用の効率的なコーディング

組込みシステムでは、外部メモリやオンチップメモリの容量を削減することで、システムの費用、消費電力を大幅に削減できるため、生成されたコードとデータのサイズが非常に重要です。

以下の項目について説明します。

- データ型の選択
- データと関数のメモリ配置制御
- コンパイラ最適化の設定
- 円滑なコードの生成

また、この中で、一般的な誤りと回避方法、優れたコーディングテクニックについても説明します。

データ型の選択

データを効率的に処理するため、使用するデータ型や最も効率的な変数配置を検討する必要があります。

効率的なデータ型の使用

使用するデータ型は、コードのサイズ / 速度に大きく影響することがあるため、慎重に検討する必要があります。

- 可能な場合、char または short ではなく、int または long を使用して、符号拡張またはゼロ拡張を回避します。特に、ループインデックスは、コード生成を最小に抑えるため、必ず int または long にしてください。また、Thumb モードでは、スタックポインタ (SP) を介したアクセスは、32 ビットデータ型に制限されます。これにより、このようなデータ型を使用するメリットを利用できます。
- アプリケーションで符号付き値が必要でない限り、符号なしデータ型を使用してください。

- 64 ビットデータ型（double、long long など）を使用する場合は、短所を理解しておいてください。
- ビットフィールドとパック構造体が大きくて低速のコードを生成します。
- 数値演算コプロセッサのないマイクロプロセッサで浮動小数点数型を使用すると、コードサイズと実行速度の両面で非常に効率が低下します。
- const 型データへのポインタを宣言すると、参照先のデータが変化しないことが呼出し元関数に通知され、最適化の向上につながります。

サポートされているデータ型、ポインタ、構造体の表現については、「データ表現」を参照してください。

浮動小数点数型

数値演算コプロセッサのないマイクロプロセッサで浮動小数点数型を使用するのは、コードサイズと実行速度の両面で非常に非効率です。そのため、浮動小数点数演算を使用するコードを、整数演算を使用するコードに置き換えることを検討してください。これにより効率が向上します。

コンパイラは、2 種類（32 ビットと 64 ビット）の浮動小数点数フォーマットをサポートしています。32 ビット浮動小数点数型の float の方は、コードサイズと実行速度の両面において効率が優れます。一方、64 ビットフォーマットの double は、より高い精度とより大きな数値に対応します。

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、292 ページの浮動小数点数型を参照してください。

構造体エレメントのアラインメント

ARM コアでは、メモリ内のデータがアラインメントされている必要があります。構造体の各エレメントは、指定した型の要件に応じてアラインメントされている必要があります。つまり、コンパイラは、正しいアラインメントを保守するため、パッドバイトを挿入しなければならないことがあります。

これが問題になりうる状況があります。

- 外部要件。たとえば、ネットワーク通信プロトコルは通常、間にパディングのないデータ型に関して指定されます。
- データメモリを節約する必要がある場合。

アラインメントの要件については、285 ページの [アラインメント](#) を参照してください。

これを解決する方法は2つあります。

- `#pragma pack` ディレクティブまたは `__packed` データ型属性を使用して、構造体のレイアウトをより密にする。この場合、構造体のアラインメントされていないエレメントにアクセスするたびにコードが使用されるという欠点があります。
- 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、325 ページの [pack](#) を参照してください。

匿名構造体と匿名共用体

構造体や共用体を名前なしで宣言すると、それらは匿名になります。その結果、それらのメンバは前後のスコープでのみ認識されます。

匿名構造体は C++ 言語の機能の一部ですが、C にはこの機能はありません。ARM 用 IAR C/C++ C/C++ コンパイラでは ARM では、言語拡張が有効な場合にこれらを C で使用できます。



IDE では、デフォルトで言語拡張が有効になっています。



言語拡張を有効にするには、`-e` コンパイラオプションを使用します。詳細については、232 ページの [-e](#) を参照してください。

例

以下の例では、匿名の union のメンバに、union 名を明示的に指定せずに、関数 F でアクセスできます。

```
struct S
{
    char mTag;
    union
    {
        long mL;
        float mF;
    };
} St;
```

```
void F(void)
{
    St.mL = 5;
}
```

メンバ名は、その前後のスコープ内で固有なものである必要があります。匿名の struct/union を、ファイルスコープレベルで、グローバル変数、外部変数、静的変数のいずれかとして使用することもできます。これは、以下の例のように、I/O レジスタの宣言などに使用できます。

```
__no_init volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 0x1000;

/* ここで変数を使用 */
void Test(void)
{
    IOPORT = 0;
    way = 1;
    out = 1;
}
```

この例は、I/O レジスタバイト IOPORT をアドレス 0x1000 で宣言します。この I/O レジスタでは、way と out の 2 ビットが宣言されます。内部の構造体と外部の共用体のいずれも匿名になっています。

匿名の構造体や共用体はオブジェクトとして実装されます。このオブジェクトの名前は、最初のフィールドにプレフィックス `_A_` を付けた名前となり、名前空間の実装部で配置されます。この例では、匿名共用体は `_A_IOPORT` というオブジェクトを使用して実装されます。

データと関数のメモリ配置制御

コンパイラでは、関数とデータオブジェクトのメモリへの配置を制御するためのさまざまな仕組みを提供しています。メモリを効率的に使用するためには、これらの仕組みに精通し、さまざまな状況に応じて最適な方法を判別できる必要があります。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。変数は、`__no_init` で宣言する必要があります。
 これは、外部要件を確認するために固定アドレス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便利です。たとえば、割込みベクタや他のハードウェアテーブルを読み込む場合などです。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。
- **@ 演算子および #pragma location ディレクティブによるセクション配置**
 @ 演算子や #pragma location ディレクティブを使用して、関数またはグローバル / 静的変数のグループを指定セクションに配置できます。各オブジェクトの明示的な制御は不要です。セクションは、たとえば、メモリの特定エリアに配置することや、セクションの開始 / 終了演算子を使用して制御された方法で初期化やコピーを行うことができます。これは、アプリケーションプロジェクトやブートローダプロジェクトのように、別々にリンクされたユニット間でインタフェースする必要がある場合にも便利です。指定セクションは、変数の個別配置の絶対制御が不要な場合や有効ではない場合に使用します。
- **@ 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。
- **-- セクションオプション**
 -- セクションオプションを使用して、指定セクションに関数やデータオブジェクトを配置します。これは、たとえば、異なる高速または低速メモリに転送する場合に便利です。-- セクションオプションについて詳しくは、254 ページの `--section` を参照してください。

コンパイル時、データおよび関数は異なるセクションに配置されます。
 68 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*。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ロードアドレスを割り当てることです。すべてのセクション（絶対配置データを保持するセクションは除く）は、リンカ設定ファイル（71 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）* 参照）の仕様に従って自動的にメモリに割り当てられます。

絶対アドレスへのデータ配置

@ 演算子、または #pragma location ディレクティブは、グローバル変数および静的変数を絶対アドレスに配置するときに使用できます。

変数を絶対アドレスに配置するには、@ 演算子や #pragma location ディレクティブの引数に、実際のアドレスを示す定数を指定します。配置する変数のアラインメント条件を満たしている絶対アドレスを指定する必要があります。

注：絶対アドレスに配置される __no_init 変数のすべての宣言は、*仮定義*です。仮定義の変数は、コンパイルするモジュールが必要な場合に、コンパイラからの出力にのみ保持されます。こうした変数は、使用されるすべてのモジュールで定義され、同じ方法で定義されている限りは機能します。こうした宣言は、変数を使用する全モジュールにインクルードされるすべてのヘッダファイルに配置することをお勧めします。

絶対アドレスに配置される他の変数は、通常の宣言と定義の区別を使用します。こうした変数については、1つのモジュール（通常はイニシャライザ）でのみ定義を提供する必要があります。他のモジュールは、明示的アドレスの有無に関わらず、extern 宣言を使用すれば変数を参照できます。

例

この例では、__no_init で宣言した変数が絶対アドレスに配置されます。これは、複数のプロセス、アプリケーションなどの間でインタフェースする場合に便利です。

```
__no_init volatile char alpha @ 0xFF2000; /* OK */
```

次の例では、2つの const 宣言オブジェクトが含まれます。1つは初期化されず、もう1つは特定の値に初期化されます。両方のオブジェクトはROMに配置されます。これは、外部インタフェースからアクセス可能な構成パラメータに便利です。2番目においては、値が既知であるため、必ずコンパイラが変数から実際に読み出すとは限りません。

```
#pragma location=0xFF2002
__no_init const int beta; /* OK */

const int gamma @ 0xFF2004 = 3; /* OK */
```

1 番目においては、値はコンパイラで初期化されません。別の方法で値を設定する必要があります。代表的な用途は、値が別々に ROM にロードされる構成や、リードオンリーの特殊機能レジスタです。

以下の例は、間違った用法を示します。

```
int delta @ 0xFF2006;          /* エラー、__no_init でない */
エラー /*                          /* "__no_init" と "const" のど
                                     ちらでもなし */

__no_init int epsilon @ 0xFF2008 /* エラー、不正なアラインメント */
```

C++ についての注意

C++ では、モジュールスコープの `const` 変数は静的（モジュールローカル）ですが、C ではこれらはグローバルです。つまり、特定の `const` 変数を宣言する各モジュールには、この名前で別の変数が含まれるということです。このようなモジュールの複数とアプリケーションをリンクする場合において、これらのモジュールがすべて、たとえば以下の宣言を（ヘッダファイル経由で）含む場合、

```
volatile const __no_init int x @ 0x100;          /* C++ では無効 */
```

リンカは、複数の変数がアドレス 0x100 に配置されていることを報告します。

この問題を回避し、プロセスを C と C++ で同じにするには、以下の例のように、これらの変数を `extern` として宣言します。

```
/* extern キーワードによって x がパブリックに */
extern volatile const __no_init int x @ 0x100;
```

注：C++ の静的メンバ変数は、他の静的変数と同様に、絶対アドレスに配置できます。

データと関数のセクションへの配置

データまたは関数をデフォルト以外の指定セクションに配置する場合、以下の方法を使用できます。

- @ 演算子か、または `#pragma location` ディレクティブを使用して、個々の変数または関数を指定したセクションに配置することができます。指定セクションとして、定義済セクションまたはユーザ定義セクションを使用できます。
- --セクションオプションは、コンパイルユニット全体の一部である変数および関数を指定セクションに配置するときに使用できます。

C++ の静的メンバ変数は、他の静的変数と同様に、指定セクションに配置できます。

定義済セクションに加え、独自のセクションを使用する場合、セクションは、リンカ構成ファイルに定義する必要があります。

注：デフォルトで使用している以外の定義済セクションの変数や関数を、明示的に配置する場合に注意してください。状況によっては有益なオプションですが、配置を間違えると、コンパイル時やリンク時のエラーメッセージからアプリケーションの誤動作までを発生することがあります。状況を慎重に考慮し、宣言および関数や変数の使用に関する要件に、厳密に従ってください。

セクションの位置は、リンカ構成ファイルから制御できます。

セクションの詳細については、「[セクションリファレンス](#)」を参照してください。

指定セクションへの変数の配置例

以下の例では、データオブジェクトがユーザ定義セクションに配置されます。

```
__no_init int alpha @ "MY_NOINIT"; /* OK */

#pragma location="MY_CONSTANTS"
const int beta; /* OK */
```

指定セクションへ関数の配置例

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_FUNCTIONS"
void h(void);
```

レジスタへのデータ配置

@ 演算子か #pragma location ディレクティブを使用して、グローバル変数や静的変数をレジスタに配置できます。

レジスタに変数を配置するには、@ 演算子の引数と #pragma location ディレクティブが R4-R11 の範囲の ARM コアレジスタに対応する識別子である必要があります (R9 は --rwp1 コマンドラインオプションとともに指定できません)。

変数をレジスタに配置できるのは、変数が __no_init として宣言され、ファイルスコープがあって、サイズが 4 バイトの場合のみです。レジスタに配置される変数は、メモリアドレスを持たないため。アドレス演算子 & は使用できません。

変数がレジスタに配置されているモジュール内では、指定されたレジスタはその変数へのアクセスのみに使用されます。変数の値は他のモジュールへの関数呼出しで保持されます。その理由は、レジスタ R14-R11 が呼出し先で保存され、実行が返されるときに復元されるためです。ただし、レジスタに配置される変数の値は、常に予想通り保持されるとは限りません。

- 例外ハンドラやライブラリコールバックルーチン（qsort に引き渡されたコンパレータ関数など）では、値が保持されないことがあります。コマンドラインオプション `--lock_regs` が、ライブラリモジュールも含むアプリケーションの全モジュール内のレジスタのロックに使用される場合、値は保持されます。
- 高速割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。これらのレジスタがバンクされているためです。
- `longjmp` 関数および C++ の例外によって、保持されない他の静的記憶寿命変数とは異なり、レジスタに配置された変数は古い値に復元されることがあります。

リンカは、モジュールが同じレジスタに異なる変数を配置するのを防止することはありません。異なるモジュールにある変数は同じレジスタ内に配置でき、別のモジュールはそのレジスタを他の目的で使用できます。

注： レジスタに配置された変数は、インクルードファイルで定義され、その変数を使用するすべてのモジュールにインクルードされる必要があります。モジュール内の未使用の定義によって、そのモジュールでレジスタが使用されなくなります。

コンパイラ最適化の設定

コンパイラは、可能な限り最良のコードを生成するために、アプリケーションで多くの変換を行います。変換の例としては、値をメモリではなくレジスタに格納する、余剰なコードを削除する、計算の順序をより効率的に変更する、数値演算をより安価な処理に置換するなどが挙げられます。

リンカによって実行される最適化もあるため、リンカもコンパイルシステムの構成要素の一部と考える必要があります。たとえば、すべての未使用の関数、変数は削除され、最終的な出力には含まれません。

最適化実行のスコープ

実行される最適化の対象を、アプリケーション全体とするか個々のファイルとするか指定できます。デフォルトでは、プロジェクト全体で同一の最適化タイプが使用されますが、個々のファイルに対して異なる最適化設定を使用することを検討してください。たとえば、非常に高速に実行する必要がある

コードは別ファイルに記述して、実行時間が最小になるようにコンパイルし、残りのコードはコードサイズを最小にします。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、個々のファイル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することができます。プラグマディレクティブについては、324 ページの `optimize` を参照してください。

複数ファイルのコンパイルユニット

さまざまな最適化を異なるソースファイルや関数に適用するだけでなく、コンパイルユニットに含まれるソースコードのファイル数（なし、または複数など）を指定することもできます。

デフォルトでは、コンパイルユニットは1つのソースファイルから構成されますが、複数ファイルのコンパイルを使用して、いくつかのソースファイルを1つのコンパイルユニットに作成することも可能です。この利点は、インライン化やクロスジャンプなど、プロシージャ間の最適化で対象のソースコードが多くなることです。アプリケーション全体を1つのコンパイルユニットとしてコンパイルするのが理想的です。ただし、大きなアプリケーションの場合はホストコンピュータにリソースの制限があるため、この方法は実用的ではありません。詳細については、240 ページの `--mfc` を参照してください。

アプリケーション全体を1つのコンパイルユニットとしてコンパイルする場合、プロシージャ間の最適化を実行する前に、コンパイラで未使用のパブリック関数と変数を破棄するのも非常に役に立ちます。こうすることで、最適化の範囲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、231 ページの `--discard_unused_publics` を参照してください。

最適化レベル

コンパイラでは、さまざまな最適化のレベルをサポートしています。以下の表は、各レベルで一般的に実行される最適化の一覧です。

最適化レベル	説明
なし（デバッグサポートに最適）	変数は、そのスコープ全体を通して有効です
低	上記と同じですが、変数は必要時のみ有効となり、スコープ全体を通して有効とは限りません

表 22: コンパイラ最適化レベル

最適化レベル	説明
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 ピーブホール最適化 一部のレジスタ内容の解析と最適化 静的クラスタ 共通部分式除去
High（最大の最適化）	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 コード移動 型ベースエイリアス解析

表 22: コンパイラ最適化レベル（続き）

注：一部の最適化は、個別に有効化 / 無効化が可能です。これらの詳細については、194 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もあり、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、スコープ全体を通じた変数の値が表示できないことがあります。コードのデバッグが困難な場合は、最適化レベルを下げてください。

速度とサイズ

最適化レベルが高の場合、コンパイラはサイズの最適化と速度の最適化の間のバランスを取ります。ただし、サイズまたは速度に対して明示的に最適化を微調整することも可能です。それらは、使用するしきい値のみの違いです。速度の場合は、サイズを犠牲にして速度を上げ、サイズの場合は、速度を犠牲にしてサイズを小さくします。ある最適化により別の最適化が実行可能になり、サイズよりも速度を優先して最適化した場合でも、アプリケーションのサイズが小さくなる場合があります。

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せん。

- 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

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、242 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、ループのオーバーヘッドが少なくなります。

この最適化は、ループのオーバー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。ループのオーバー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せん。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 (248 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単純な関数を、その呼出し元関数の本体に統合し、呼出しによるオーバーヘッドを解消することです。この最適化は、最適化レベルが [高] の場合に実行可能で、通常は実行時間が短縮されますが、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。

インライン化する関数は、コンパイラがヒューリスティックにより決定します。実行する最適化の内容 (速度、サイズ、速度とサイズのバランス) に応じて、異なるテクニックが使用されます。通常はサイズを最適化する際はコードサイズは増加しません。個々の関数についてヒューリスティックを制御するには、`#pragma inline` ディレクティブまたは標準の C の `inline` キーワードを使用します。

モジュール全体のインライン化を無効にしない場合、個々の関数に `#pragma inline=never` を使用します。

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、243 ページの `--no_inline` を参照してください。プラグマディレクティブについては、321 ページの `inline` を参照してください。

コード移動

ループ不変式や共通部分式の評価式を移動し、冗長な再評価を回避します。この最適化は、最適化レベルが [高] 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 [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、241 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、247 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` でポイントされる `short` 型へのライトにより、`p2` がポイントする `long` 値は変更されないことを前提としています。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、241 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。すべてのコアがスケジューリングの恩恵を受けるわけではありません。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 [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、246 ページの `--no_scheduling` を参照してください。

円滑なコードの生成

ここでは、コンパイラで良いコードを生成するためのヒントについて説明します。たとえば、次のようなものがあります。

- 効率的なアドレッシングモードの使用
- コンパイラの最適化の促進
- より有意義なエラーメッセージの生成

最適化を容易にするソースコードの記述

以下に、コンパイラでのアプリケーション最適化を改善できるプログラミングテクニックを示します。

- 静的 / グローバル変数よりもローカル変数（自動変数とパラメータ）を使用することをお勧めします。これは、呼出し先関数がローカル以外の変数を変更する可能性などをオブ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ます、プロセッサのレジスタに配置できません。そのため、コードの

サイズが大きくなり、速度が低下します。次に、ローカル変数が関数呼出しの影響を受けないとオブ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（195 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勧めします。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、192 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。
- インラインアセンブラの使用は避けてください。その代わりに、コードを C/C++ で記述する、組み込み関数を使用する、アセンブラ言語で別モジュールとして記述するなどを検討してください。詳細については、129 ページの *C 言語とアセンブラの結合* を参照してください。

スタックエリアと RAM メモリの節約

以下に、メモリやスタックエリアを節約できるプログラミングテクニックを示します。

- スタックエリアが少ない場合は、長い呼出しチェーンや再帰関数の使用は避けてください。
- 大きなサイズの非スカラ型（構造体など）をパラメータやリターン型。として使用することは避けてください。スタックエリアを節約するため、代わりにポインタか、C++ の場合は参照として引き渡してください。

関数プロトタイプ

2 つのスタイルのいずれかを使用して、関数の宣言と定義を行うことができます。

- プロトタイプ
- カーニハン & リッチー C (K&R C)

いずれのスタイルも C 規格に含まれていますが、プロトタイプスタイルの方がコンパイラでコードの問題が検出されやすいため、こちらの使用をお勧めします。また、プロトタイプスタイルを使用することで、型変換（暗黙のキャスト）が不要になるため、より効率的なコードを生成できます。K&R スタイルは、互換性の理由からのみサポートされています。

すべての関数が適切なプロトタイプを持つかどうかをコンパイラで検証するには、コンパイラオプション **[プロトタイプの強制]** (`--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します。また、定義の記述が異なります。

例

```
int Test();      /* 宣言 */

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

整数型とビット否定

場合によっては、整数型とその変換の規則が、混乱を招く動作の原因となることがあります。異なるサイズの型や論理演算（特にビット否定）が関係する代入文や条件文（評価式）に注意する必要があります。この場合、*types* 型には定数型も含まれます。

ワーニング（定数の条件文や無意味な比較など）が発生する場合と、期待した結果と異なるだけの場合があります。コンパイラが定数の条件文のインスタンスを特定するために最適化を使用する場合などは、より高水準の最適化でのみ、コンパイラがワーニングを生成することがあります。以下の例では、8 ビット文字、32 ビット整数、2 の補数を想定しています。

```
void F1(unsigned char c1)
{
    if (c1 == ~0x80)
        ;
}
```

この場合、評価結果は常に `false` になります。右辺の `0x80` は `0x00000080` であ、`~0x00000080` は `0xFFFFF7F` になります。左辺の `c1` は 8 ビットの符号なし文字であるため、255 を超えることはありません。また、負の値にもならないため、汎整数拡張された値の上位 24 ビットが設定されることはありません。

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（割込みルーチンからアクセスされる変数、独立したスレッドで実行しているコードからアクセスされる変数など）は、適切にマークして保護する必要があります。例外は、常にリードオンリーの変数のみです。

変数を適切にマークするには、`volatile` キーワードを使用します。このキーワードは、変数が他のスレッドから変更される可能性があることをコンパイラに示します。すると、コンパイラでは、変数の最適化を回避（レジスタ内の変数を追跡するなど）し、変数へのライトを遅延させず、ソースコードで指定された回数だけ変数にアクセスするように注意します。

`volatile` 型修飾子および `volatile` オブジェクトへのアクセスの規則については、298 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、`iodevice.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 `ioxs32c5000a.h` の例を示します。

```
__no_init volatile union
{
    unsigned short mwctl12;
    struct
    {
        unsigned short edr   : 1;
        unsigned short edw   : 1;
        unsigned short lee   : 2;
        unsigned short lemd  : 2;
        unsigned short lepl  : 2;
    } mwctl12bit;
} @ 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 */
    mwctl12 = 0x1234;

    /* ビットフィールドアクセス */
    mwctl12bit.edw = 1;
    mwctl12bit.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。

```

#pragma diag_suppress=Pe940
#pragma optimize=no_inline
static unsigned long get_APSR( void )
{
    /* 関数の出口では、
     * 戻り値は R0 に置かれる */
    asm( "MRS R0, APSR" );
}

#pragma diag_default=Pe940

#pragma optimize=no_inline
static void set_APSR( unsigned long value)
{
    /* 関数の入り口では、最初のパラメータは R0 に置かれる */
    asm( "MSR APSR, R0" );
}

```

汎用レジスタ R0 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。関数にはインラインアセンブラのみが含まれるので、コンパイラは、レジスタの使用には干渉しません。レジスタ R0 は、常に、リターン値用に使用されます。最初のパラメータは、その型が 32 ビット以下の場合、常に、R0 に渡されます。

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を使用する場合のリスクについては、131 ページの *インラインアセンブラ* を参照してください。インラインアセンブラの詳細については、148 ページの *インラインアセンブラ* を参照してください。

注： インラインアセンブラを使用する前に、代わりに組み込み関数を使用できるかどうか検討してください。333 ページの *組み込み関数の概要* 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、ランタイムライブラリ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。

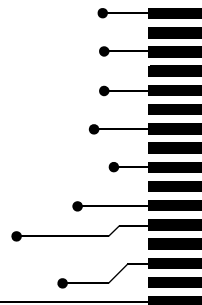
`__no_init` キーワードを使用して宣言した変数は、大きな入力バッファとして使用することや、アプリケーション終了後も内容を保持する特殊な RAM にマッピングすることができます。

`__no_init` キーワードについては、308 ページのを参照してください。このキーワードを使用するには、言語拡張を有効にする必要があります。232 ページの *-e* を参照してください。`#pragma object_attribute` については、324 ページの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』のパート 2 は、以下の章で構成されています。

- 外部インタフェースの詳細
- コンパイラオプション
- リンカオプション
- データ表現
- 拡張キーワード
- プラグマディレクティブ
- 組込み関数
- プリプロセッサ
- ライブラリ関数
- リンカ設定ファイル
- セクションリファレンス
- IAR ユーティリティ
- 処理系定義の動作





外部インタフェースの詳細

この章では、コンパイラおよびリンカがそれらの環境とやりとりする方法について説明します。ここでは、呼出し構文、ツール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とリンカ出力の一覧と簡単な説明を提供します。

呼出し構文

コンパイラとリンカは、IDE またはコマンドラインインタフェースから使用できます。IDE からのビルドツールの使用については、『*ARM® 用 IDE プロジェクト管理およびビルドガイド*』を参照してください。

コンパイラ呼出し構文

コンパイラの呼出し構文は次のとおりです。

```
icarm [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。

```
icarm prog.c --debug
```

ソースファイルには、C/C++ ファイルを使用でき、通常、それぞれ c/cpp のファイル名拡張子を指定します。ファイル名の拡張子を指定しない場合、コンパイルするファイルの拡張子は c でなければなりません。

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

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

ILINK 呼出し構文

ILINK の呼出し構文は次のとおりです。

```
ilinkarm [引数]
```

各引数は、コマンドラインオプション、オブジェクトファイル、ライブラリのいずれかです。

たとえば、オブジェクトファイル `prog.o` をリンクする場合、以下のコマンドを使用します。

```
ilinkarm prog.o --config configfile
```

リンカ設定ファイルの拡張子を指定しない場合、設定ファイルの拡張子は `icf` でなければなりません。

通常、コマンドラインの引数の順序は重要ではありません。ただし、例外が1つあります。複数のライブラリを適用する場合、ライブラリの検索はコマンドラインに指定した順序で行われます。デフォルトライブラリは常に最後に検索されます。

出力実行可能イメージは、`-o` オプションが使用されない限り、`a.out` という名前のファイルに置かれます。

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

オプションの受渡し

オプションをコンパイラおよび **ILINK** に渡す方法は3とおりあります。

- コマンドラインから直接渡す方法
コマンドラインで、`iccarm` または `ilinkarm` コマンドの後にオプションを指定します（205 ページの *呼出し構文* を参照）。
- 環境変数経由で渡す方法
コンパイラおよびリンカは、自動的に環境変数の値を各コマンドラインの後に付加します（207 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（235 ページの *-f* を参照）。

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

環境変数

以下の環境変数をコンパイラで使用できます。

環境変数	説明
C_INCLUDE	インクルードファイルを検索するディレクトリを指定します。例： C_INCLUDE=c:\program files\iar systems\embedded workbench 6.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 23: コンパイラの環境変数

以下の環境変数が ILINK で使用できます。

環境変数	説明
ILINKARM_CMD_LINE	コマンドラインのオプションを指定します。 ILINKARM_CMD_LINE=---config full.icf --silent

表 24: ILINK 環境変数

インクルードファイル検索手順

コンパイラの #include ファイル検索手順の詳細を以下に示します。

- #include ファイルの名前が角括弧または二重引用符で指定された絶対パスの場合は、そのファイルが開きます
- 以下のように #include ファイルの名前が角括弧で囲まれている場合
#include <stdio.h>
以下のディレクトリでインクルード対象ファイルが検索されます。
 - 1 -I オプションで指定されるディレクトリ。指定順に検索されます (237 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) (207 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。231 ページの --dlib_config を参照してください。
- 次のように #include ファイルの名前が二重引用符で囲まれている場合
#include "vars.h"
#include 文が記述されているソースファイルのあるディレクトリが検索され、その後、角括弧で囲まれたファイル名の場合と同じ手順が実行されます。

`#include` ファイルが入れ子になっている場合は、最後にインクルードされたファイルのあるディレクトリから検索が開始され、上位方向に各インクルードファイルの検索が繰り返され、最後にソースファイルのディレクトリが検索されます。次に例を示します。

```
src.c in directory dir¥src
#include "src.h"
...
src.h in directory dir¥include
#include "config.h"
...
```

`dir¥exe` がカレントディレクトリの場合は、以下のコマンドを使用してコンパイルします。

```
icccarm ..¥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。

<code>dir¥include</code>	現在のファイルは <code>src.h</code> です。
<code>dir¥src</code>	現在のファイルが含まれるファイル (<code>src.c</code>)。
<code>dir¥include</code>	最初の <code>-I</code> オプションで指定した通りになります。
<code>dir¥debugconfig</code>	2 番目の <code>-I</code> 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。

ヘッダファイルをインクルードする構文については、377 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
コンパイラで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 FLE を使用しています。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（238 ページの `-l` を参照）。デフォルトでは、これらのファイルのファイル名拡張子は `lst` です。

- プロセッサ出力ファイル（オプション）
プロセッサ出力ファイルを作成するには、`--preprocess` オプションを使用します。デフォルトでは、このファイルのファイル名拡張子は `i` です。
- 診断メッセージ
診断メッセージは、標準エラーストリームに転送されて画面上に表示されるほか、オプションのリストファイルにも出力されます。診断メッセージの詳細については、『211 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（209 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。
1	オプション <code>--warnings_affect_exit_code</code> の使用中にワーニングが発生しました。
2	エラーが発生しました。
3	致命的なエラーが発生したためツールが停止しました。
4	インターナルエラーが発生したためツールが停止しました。

表 25: エラーリターンコード

ILINK 出力

ILINK では、以下の出力を生成できます。

- 絶対実行可能イメージ

IAR ILINK リンカは、最終出力として、実行可能イメージを含む絶対オブジェクトファイルを生成します。このファイルは、EPROM に格納したり、ハードウェアエミュレータにダウンロードしたり、IAR C-SPY デバッガシミュレータを使用して PC 上で実行できます。デフォルトでは、ファイルは out のファイル名拡張子を持ちます。出力フォーマットは、常に ELF です。これは、オプションで DWARF フォーマットのデバッグ情報を含みます。

- オプションのログイン情報

操作中、ILINK は、その決定を stdout、およびオプションでファイルに記録します。たとえば、ライブラリが検索される場合、必要なシンボルがライブラリモジュールで見つかったかどうか、またはモジュールが出力の一部になるかどうか記録されます。各 ILINK サブシステムのタイミング情報も記録されます。

- オプションのマッピングファイル

リンカマッピングファイル（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む）は、ILINK オプション --map を使用して生成できます（274 ページの --map を参照）。デフォルトでは、マッピング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、オプションのマッピングファイルにも出力されます。診断メッセージの詳細については、『211 ページの 診断』を参照してください。

- エラーリターンコード

ILINK は、バッチファイル内で評価可能なステータス情報をオペレーティングシステムに提供します（209 ページの エラーリターンコードを参照）。

- 使用メモリのサイズ情報および経過時間

関数およびメモリごとのデータに対して生成されたバイト数に関する情報が stdout に転送され、画面上に表示されます。

診断

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

コンパイラのメッセージフォーマット

診断メッセージはすべて、説明を要しない完結型のメッセージとして出力されます。コンパイラの診断メッセージは、次のフォーマットで生成されます。

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

各エLEMENTの意味は以下のとおりです。

<i>filename</i>	問題が発生したソースファイルの名前
<i>linenumber</i>	コンパイラが問題を検出した行の番号
<i>level</i>	問題の重要度のレベル
<i>tag</i>	診断メッセージを示す固有のタグ
<i>message</i>	説明（場合によっては複数行）

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

オプション `--diagnostics_tables` を使用すると、すべてのコンパイラ診断メッセージが一覧表示されます。

リンカのメッセージフォーマット

診断メッセージはすべて、説明を要しない完結型のメッセージとして出力されます。ILINK が生成する診断メッセージは、通常次のような形式です。

```
level[tag]: message
```

各エLEMENTの意味は以下のとおりです。

<i>level</i>	問題の重要度のレベル
<i>tag</i>	診断メッセージを示す固有のタグ
<i>message</i>	説明（場合によっては複数行）

診断メッセージは、オプションのマッピング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、253 ページの `--remarks` を参照してください。

ワーニング

コンパイラまたはリンカがコードの生成に支障のあるような潜在的な問題はあるが、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 (248 ページのを参照)。

エラー

コンパイラまたはリンカがコードの生成に支障のあるような重大なエラーを検出した場合に生成される診断メッセージ。エラーが発生した場合は、ゼロ以外の終了コードが生成されます。

致命的なエラー

コードが生成できなくなるだけでなく、それ以降の処理が無意味となるような状態をコンパイラが検出した場合に生成される診断メッセージ。このメッセージが出力された後、コンパイルが終了します。致命的なエラーが発生した場合は、ゼロ以外の終了コードが生成されます。

重要度の設定

致命的なエラーや一部の通常エラーを除くすべての診断メッセージに対し、診断メッセージの出力抑制や重要度の変更ができます。

重要度の設定に使用可能なコンパイラオプションについては、219 ページの *コンパイラオプションの概要* を参照してください。

コンパイラについては、「プラグマディレクティブ」を参照してください。

インターナルエラー

インターナルエラーは、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。

インターナルエラー : *message*

ここで、*message* はエラーの説明を示します。インターナルエラーが発生した場合は、ソフトウェアの配布元か IAR システムズの技術サポートまでご報告ください。その際、問題を再現できるように、以下の情報をお知らせください。

- 製品名
- コンパイラまたは ILINK のバージョン番号 (コンパイラまたは ILINK が生成するリストまたはマップファイルのヘッダ部分にあります)
- ライセンス番号
- インターナルエラーメッセージ本文
- インターナルエラーの原因となったアプリケーションの関連ファイル
- インターナルエラー発生時に指定していたオプションの一覧

コンパイラオプション

この章では、コンパイラオプションの構文とオプションパラメータを指定するための一般的な構文規則について説明するとともに、各オプションの詳細なリファレンス情報を提供します。

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、『*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。

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、206 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う場合は、以下のように、パラメータの前に等号 (=) を付けて指定します。

```
--misrac2004=n
```

必須パラメータの規則

省略形のオプションで必須パラメータを伴う場合は、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ともかまいません。

```
-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。

```
icccarm prog.c -l ..¥listings¥List.lst
```

- ファイル名を出力先として指定するオプションの場合、ファイル名のないパスとしてパラメータを指定できます。コンパイラは、このディレクトリ内のオプションに基づいた拡張子を持つファイルに出力を保存します。ファイル名は、コンパイルしたソースファイルの名前と同じになります。ただし、`-o` オプションで別の名前を指定した場合は、その名前が使用されます。次に例を示します。

```
icccarm prog.c -l ..¥listings¥
```

生成されるリストファイルには、デフォルト名の `..¥listings¥prog.lst` が付けられます。

- *現在のディレクトリ*は、以下のように、ピリオド (`.`) で指定します。次に例を示します。

```
icccarm prog.c -l .
```

- `/` をディレクトリの区切り文字として `¥` の代わりに使用できます。
- `-` を指定することにより、入力ファイルおよび出力ファイルがそれぞれ、標準の入力および出力ストリームにリダイレクトされます。次に例を示します。

```
icccarm 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。

```
icccarm prog.c -l ---r
```

- 同じ型の複数の引数を指定可能なオプションの場合、引数は、以下の例のようにカンマ区切り（スペースなし）のリストとして指定できます。

```
--diag_warning=Be0001,Be0002
```

また、以下のように、引数ごとにオプションを繰り返して指定することもできます。

```
--diag_warning=Be0001
```

```
--diag_warning=Be0002
```

コンパイラオプションの概要

以下の表に、コンパイラのコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--aapcs	呼出し規約を指定します
--aeabi	AEABI 準拠のコード生成を有効にします
--align_sp_on_irq	__irq 関数への入り口で SP を整列されるコードを生成します
--arm	デフォルトの関数モードを ARM に設定します
--c89	C89 の派生言語を指定します
--char_is_signed	char を符号付として処理
--char_is_unsigned	char を符号なしとして処理
--cpu	派生プロセッサを指定します
--cpu_mode	関数のデフォルトモードを選択します
--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	デバッグ情報を生成
--dependencies	ファイル依存関係をリスト化
--diag_error	エラーとして処理
--diag_remark	リマークとして処理
--diag_suppress	診断を無効化
--diag_warning	ワーニングとして処理
--diagnostics_tables	すべての診断メッセージをリスト化
--discard_unused_publics	未使用のパブリックシンボルを破棄
--dlib_config	DLIB ライブラリのシステムインクルードファイルを使用して、どのライブラリの設定を使用するかを決定
-e	言語拡張を有効化
--ec++	Embedded C++ を指定
--eec++	Extended Embedded C++ を指定
--enable_hardware_workaround	個別のハードウェア対策を有効化します
--enable_multibytes	ソースファイルのマルチバイト文字のサポートを有効化

表 26: 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
--endian	生成されるコードおよびデータのバイトオーダーを指定します
--enum_is_int	列挙型の最小サイズを設定します
--error_limit	コンパイルを停止するエラー数の上限を指定
-f	コマンドラインを拡張
--fpu	浮動小数点ユニット型を選択します
--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
--interwork	相互作用コードを生成します
-l	リストファイルを生成
--legacy	古いツールチェーンとリンク可能なオブジェクトコードを生成します
--lock_regs	コンパイラが指定のレジスタを使用しないようにします
--macro_positions_in_diagnostics	診断メッセージでマクロ内の位置を取得
--mfc	複数ファイルのコンパイルを有効化
--migration_preprocessor_extensions	プリプロセッサを拡張
--misrac	MISRA-C:1998 固有のエラーメッセージを有効化。このオプションは --misrac1998 と同義で、旧バージョンとの互換性のためだけに使用できます
--misrac1998	MISRA-C:1998 固有のエラーメッセージを有効化 (『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』を参照)
--misrac2004	MISRA-C:2004 固有のエラーメッセージを有効化 (『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』を参照)
--misrac_verbose	MISRA-C チェックの冗長なログギングを有効化。『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』または『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』を参照してください

表 26: 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
<code>--no_clustering</code>	静的クラスタ最適化を無効にします
<code>--no_code_motion</code>	コード移動最適化を無効化
<code>--no_const_align</code>	定数のアラインメント最適化を無効にします
<code>--no_cse</code>	共通部分式除去を無効化
<code>--no_exceptions</code>	C++ 例外のサポートを無効化
<code>--no_fragments</code>	セクションフラグメント処理を無効化
<code>--no_inline</code>	関数インライン化を無効化
<code>--no_loop_align</code>	ループ内のラベルのアラインメントを無効化 (Thumb2)
<code>--no_mem_idioms</code>	<code>memcpy/memset/memclr</code> でイディオムの認識を無効化
<code>--no_path_in_file_macros</code>	シンボル <code>__FILE__</code> および <code>__BASE_FILE__</code> のリターン値からパスを削除
<code>--no_rtti</code>	C++ RTTI のサポートを無効化
<code>--no_rw_dynamic_init</code>	静的 C 変数のランタイムの初期化を無効化
<code>--no_scheduling</code>	命令スケジューラを無効にします
<code>--no_static_destruction</code>	プログラム終了時に C++ 静的変数の破壊を無効化します
<code>--no_system_include</code>	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
<code>--no_tbaa</code>	型ベースエイリアス解析を無効化
<code>--no_typedefs_in_diagnostics</code>	診断での <code>typedef</code> 名の使用を無効化
<code>--no_unaligned_access</code>	アラインメントされないアクセスを回避します
<code>--no_unroll</code>	ループ展開を無効化
<code>--no_warnings</code>	すべての警告を無効化
<code>--no_wrap_diagnostics</code>	診断メッセージのラッピングを無効化
<code>-O</code>	最適化レベルを設定
<code>-o</code>	オブジェクトファイル名を設定。--output のエイリアス
<code>--only_stdout</code>	標準出力のみを使用
<code>--output</code>	オブジェクトファイル名を設定
<code>--predef_macros</code>	定義済シンボルの一覧を表示

表 26: 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
--preinclude	ソースファイルを読み込む前にインクルードファイルをインクルード
--preprocess	プリプロセッサ出力を生成
--public_equ	グローバル名のアセンブララベルを定義
-r	デバッグ情報を生成。--debug のエイリアス
--relaxed_fp	浮動小数点式の最適化規則を緩和します
--remarks	リマークを有効化
--require_prototypes	関数が定義前に宣言されていることを検証
--ropi	アドレスコードおよびリードオンリーのデータへの PC 関連の参照を使用するコードを生成
--rwp	静的ベースレジスタからアドレス書き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
--セクション	セクション名を変更
--separate_cluster_for_initialized_variables	初期化変数と非初期化変数を分離
--silent	サイレント処理を設定
--strict	標準 C/C++ への厳密な準拠を確認
--system_include_dir	システムインクルードファイルのパスを指定
--thumb	デフォルトの関数モードを Thumb に設定します
--use_c++_inline	C99 で C++ インライン動作を使用
--use_unix_directory_separators	パス内で / をディレクトリの区切り文字として使用
--vla	C99 VLA のサポートを有効化
--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 26: 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

ここでは、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。



[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	<code>--aapcs={std vfp}</code>	
パラメータ	std	AAPCS 規格に従って、関数呼出しでの浮動小数点パラメータおよびリターン値にプロセッサレジスタが使用されます。ソフトウェア FPU が選択されている場合は、std がデフォルトです。
	vfp	浮動小数点パラメータおよびリターン値に VFP レジスタが使用されます。生成されたコードは AEAB コードに準拠していません。VFP ユニットが使用されている場合、vfp がデフォルトです。
説明	このオプションは、浮動小数点の呼出し規約を指定するときに使用します。	



[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]

--aeabi

構文	<code>--aeabi</code>	
説明	このオプションは、AEABI 準拠のオブジェクトコードを生成するときに使用します。このオプションは、 <code>--guard_calls</code> オプションとともに使用する必要があります。	
関連項目	178 ページの <i>AEABI</i> への準拠、237 ページの <code>--guard_calls</code> 。	



[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]

--align_sp_on_irq

構文	<code>--align_sp_on_irq</code>	
説明	このオプションを使用して、 <code>__irq</code> により宣言された関数への入り口でスタックポインタ (SP) を整列させます。 これは、割込みコードが割込みハンドラと同じ SP を使用する、ネストされた割込みに特に便利です。つまり、スタックは AEABI (および一部のコアでコンパイラにより生成される特定の命令) で必要とされる 8 バイトのアラインメントではなく、4 バイトのアラインメントしか持たない可能性があります。	

関連項目

307 ページの `__irq`。

[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]

--arm

構文

`--arm`

説明

このオプションは、デフォルトの関数モードを ARM に設定するときに使用します。この設定は、相互作用していない限り、プログラムに含まれるすべてのファイルで同じでなければなりません。

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

関連項目

238 ページの `--interwork`、306 ページの `__interwork`。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [コード] > [プロセッサのモード] > [Arm]

--c89

構文

`--c89`

説明

このオプションを使用して、標準の C ではなく C89 C の派生言語を有効にします。

注： このオプションは、MISRA-C のチェックが有効な場合に必須です。

関連項目

147 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 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	<code>--char_is_unsigned</code>
説明	このオプションは、コンパイラで単純な <code>char</code> 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 <code>char</code> 型のデフォルトの解釈です。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 2] > ['char' の型]

--cpu

構文	<code>--cpu=core</code>
パラメータ	<code>core</code> 特定の派生プロセッサを指定します
説明	このオプションは、コードの生成対象の派生プロセッサを選択する場合に使用します。デフォルトは、ARM7TDMI です。以下のコアおよびプロセッサマクロセルが認識されます。

ARM7TDMI	ARM946E-S	ARM1176JF (ARM1176JZF のエイリアス)
ARM7TDMI-S	ARM966E-S	ARM1176JF-S (ARM1176JZF-S のエイリアス)
ARM710T	ARM968E-S	Cortex-A5
ARM720T	ARM10E	Cortex-M0
ARM740T	ARM1020E	Cortex-M1
ARM7EJ-S	ARM1022E	Cortex-Ms1*
ARM9TDMI	ARM1026EJ-S	Cortex-M3

ARM920T	ARM1136J	Cortex-M4
ARM922T	ARM1136J-S	Cortex-M4F
ARM940T	ARM1136JF	Cortex-R4
ARM9E	ARM1136JF-S	Cortex-R4F
ARM9E-S	ARM1176J (ARM1176JZ のエイリアス)	XScale
ARM926EJ-S	ARM1176J-S (ARM1176JZ-S のエイリアス)	XScale-IR7

*オペレーティングシステム拡張を持つ Cortex-M1

関連項目 48 ページの 派生プロセッサ。



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

--cpu_mode

構文 --cpu_mode={arm|a|thumb|t}

パラメータ

arm, a (デフォルト) ARM モードを関数のデフォルトモードとして選択します
thumb, t Thumb モードを関数のデフォルトモードとして選択します


説明 このオプションは、関数のデフォルトモードを選択するときに使用します。
この設定は、相互作用していない限り、プログラムに含まれるすべてのファイルで同じでなければなりません。

関連項目 238 ページの --interwork、306 ページの __interwork。




[プロジェクト] > [オプション] > [一般オプション] > [ターゲット] > [プロセッサのモード]

--c++

構文	--c++
説明	デフォルトでは、コンパイラでサポートされている言語はC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、コンパイラで使用する言語をC++に設定する必要があります。
関連項目	233 ページの --ec++、233 ページの --eec++、157 ページの C++ の使用。
	 [プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++] および [プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++ 派生言語] > [C++]

-D

構文	-D <i>symbol</i> [= <i>value</i>]				
パラメータ	<table> <tr> <td><i>symbol</i></td><td>プリプロセッサシンボルの名前</td></tr> <tr> <td><i>value</i></td><td>プリプロセッサシンボルの値</td></tr> </table>	<i>symbol</i>	プリプロセッサシンボルの名前	<i>value</i>	プリプロセッサシンボルの値
<i>symbol</i>	プリプロセッサシンボルの名前				
<i>value</i>	プリプロセッサシンボルの値				
説明	<p>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。</p> <p>-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。</p> <p>-D<i>symbol</i></p> <p>は、以下の文と等価です。</p> <pre>#define <i>symbol</i> 1</pre> <p>以下の文と等価なコマンドを考えてみます。</p> <pre>#define FOO</pre> <p>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。</p> <pre>-DFOO=</pre> <p>  [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] </p>				

--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]] {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

ファイル名やディレクトリの指定については、217 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、入力用に開かれたすべてのソースおよびヘッダファイルの名前を、デフォルトのファイル名拡張子 i を持つ 1 つのファイルにリスト化するときに使用します。

例

--dependencies や --dependencies=i を使用すると、開かれている各入力ファイルの名前とフルパス（ある場合）が独立した行に出力されます。次に例を示します。

```
c:¥iar¥product¥include¥stdio.h
d:¥myproject¥include¥foo.h
```

--dependencies=m を使用した場合は、makefile スタイルで出力されます。各入力ファイルについて、makefile の依存関係規則を含む行が生成されます。各行は、オブジェクトファイル名、コロン、空白文字、入力ファイル名で構成されます。次に例を示します。

```
foo.o: c:¥iar¥product¥include¥stdio.h
foo.o: d:¥myproject¥include¥foo.h
```

たとえば、**gmake** (GNU make) などの一般的な **make** ユーティリティで **--dependencies** を使用するには、以下のように操作します。

- 1 以下のように、ファイルのコンパイル規則を設定します。

```
%o : %.c
$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

すなわち、このコマンドを使用すると、オブジェクトファイルの生成に加えて、**makefile** スタイルで依存関係ファイルが生成されます（この例では、拡張子に **.d** を使用）。

- 2 以下のようにして、すべての依存関係ファイルを **makefile** に含めます。

```
-include $(sources:.c=.d)
```

ダッシュ (-) があるため、**.d** ファイルがまだ存在しない最初の時点でも機能します。



このオプションは、IDE では使用できません。

--diag_error

構文

```
--diag_error=tag[, tag, ...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージをエラーとして再分類する場合に使用します。エラーは、C/C++ 言語の規則違反のうち、オブジェクトコードが生成されないようなものを示します。終了コードはゼロ以外になります。このオプションは、1つのコマンドラインで複数個使用できます。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [診断] > [エラーとして処理]

--diag_remark

構文

```
--diag_remark=tag[, tag, ...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）

説明

このオプションは、特定の診断メッセージをリマークとして再分類する場合に使用します。リマークは、最も軽微な診断メッセージです。生成したコードに異常動作の原因となる可能性があるソースコード構造が存在することを示します。このオプションは、1つのコマンドラインで複数個使用できます。

注：デフォルトでは、リマークは表示されません。リマークを表示するには、`--remarks` オプションを使用します。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [診断] > [リマークとして処理]

--diag_suppress

構文

```
--diag_suppress=tag[, tag, ...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージを無効にする場合に使用します。これらのメッセージは表示されなくなります。このオプションは、1つのコマンドラインで複数個使用できます。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [診断] > [診断を無効化]

--diag_warning

構文

```
--diag_warning=tag[, tag, ...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe826 など）

説明

このオプションは、特定の診断メッセージをワーニングとして再分類する場合に使用します。ワーニングは、問題はあるが、コンパイラでのコンパイルの途中終了の原因にはならないエラーや漏れを示します。このオプションは、1つのコマンドラインで複数個使用できます。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [診断] > [ワーニングとして処理]

--diagnostics_tables

構文	<code>--diagnostics_tables {filename directory}</code>
パラメータ	ファイル名やディレクトリの指定については、217 ページの ファイル名またはディレクトリをパラメータとして指定する場合の規則 を参照してください。
説明	このオプションは、すべての診断メッセージを指定ファイルに保存する場合に使用します。これは、プラグマディレクティブを使用して診断メッセージの重要度を無効化または変更したが、その理由を記述し忘れた場合などに便利です。 このオプションは、他のオプションと併用できません。



このオプションは、IDE では使用できません。

--discard_unused_publics

構文	<code>--discard_unused_publics</code>
説明	このオプションは、 <code>--mfc</code> 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。
関連項目	240 ページの <code>--mfc</code> 、192 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	<code>--dlib_config filename.h config</code>
パラメータ	<i>filename</i> DLIB 設定ヘッダファイル。ファイル名の指定については、217 ページの ファイル名またはディレクトリをパラメータとして指定する場合の規則 を参照してください。

`config`

指定された設定のデフォルト設定ファイルが使用されます。
 以下から選択します。
 none。設定は使用されません。
 normal。通常のライブラリ設定が使用されます（デフォルト）。
 full。フルライブラリ設定が使用されます。

説明

このオプションを使用して、明示的なファイルを指定するか、ライブラリ設定を指定することによって、どのライブラリ設定を使用するかを指定します。ライブラリ設定を指定する場合は、ライブラリ設定のデフォルトファイルが使用されます。使用するライブラリに対応する設定を指定してください。このオプションを指定しない場合、デフォルトのライブラリ設定ファイルが使用されます。

すべてのビルド済ランタイムライブラリについて、対応する設定ファイルが提供されています。ライブラリオブジェクトファイルやライブラリ設定ファイルは、`arm¥lib` ディレクトリにあります。ビルド済ランタイムライブラリの例と詳細については、*91 ページの ビルド済ライブラリ*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必要があります。詳細については、*102 ページのカスタマイズしたライブラリのビルドと使用*を参照してください。



オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [一般オプション] > [ライブラリ構成]

-e**構文**`-e`**説明**

コマンドラインバージョンのコンパイラのデフォルトでは、言語拡張が無効になっています。拡張キーワードや匿名構造体 / 共用体などの言語拡張をソースコードで使用する場合には、このオプションを使用して有効化する必要があります。

注：-e オプションと --strict オプションは、同時に使用できません。

関連項目

149 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] > [IAR 拡張ありの標準]

注：IDE では、このオプションがデフォルトで選択されています。

--ec++

構文

`--ec++`

説明

コンパイラでは、デフォルトの言語は C です。Embedded C++ を使用する場合は、このオプションを使用して、コンパイラで使用する言語を Embedded C++ に設定する必要があります。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++]

および

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++ 派生言語] > [Embedded C++]

--eec++

構文

`--eec++`

説明

名前空間や標準テンプレートライブラリなどの拡張 Embedded C++ の機能をソースコードで利用する場合は、このオプションを使用して、コンパイラが使用する言語を拡張 Embedded C++ に設定する必要があります。

関連項目

158 ページの *拡張 Embedded C++*。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++]

および

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++ 派生言語] > [Extended Embedded C++]

--enable_hardware_workaround

構文

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

パラメータ

`wait`

有効化対策の ID 番号です。使用可能な有効化対策の一覧については、リリースノートを参照してください

説明

このオプションは、特定のハードウェア問題の対策をコンパイラで生成する場合に使用します。

関連項目

使用可能なパラメータの一覧については、リリースノートを参照してください。



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--enable_multibytes

構文

```
--enable_multibytes
```

説明

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

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



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 2] > [マルチバイト文字サポートを有効にする]

--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ーとして指定します
little, l (デフォルト)	リトルエンディアンをデフォルトのバイトオーダーとして指定します

説明

このオプションは、生成されるコードおよびデータのバイトオーダー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ーでコードを生成します。

関連項目

286 ページの *バイトオーダー*、



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


--enum_is_int

構文	--enum_is_int
説明	このオプションは、列挙型のサイズを少なくとも 4 バイトに強制的に指定するときに使用します。 注： このオプションでは、整数型よりサイズの大きい enum 型を使用できるということは考慮されません。
関連項目	287 ページの <i>enum</i> 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= <i>n</i>
パラメータ	<i>n</i> コンパイラがコンパイルを中止するエラー発生回数 (<i>n</i> 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、エラー数の上限を指定します。この値を超えると、コンパイラがコンパイルを停止します。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 <i>filename</i>
パラメータ	ファイル名の指定については、217 ページの <i>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</i> 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 `xc1`）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。

 このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--fpu

構文

`--fpu={VFPv2|VFPv3|VFPv3_d16|VFPv4|VFPv4_sp|VFP9-S|none}`

パラメータ


VFPv2	VFPv2 アーキテクチャに準拠した VFP ユニットを実装するシステム
VFPv3	VFPv3 アーキテクチャに準拠した VFP ユニットを実装するシステム
VFPv3_d16	VFPv3 アーキテクチャの D16 派生品に準拠した VFP ユニットを実装するシステム
VFPv4	VFPv4 アーキテクチャに準拠した VFP ユニットを実装するシステム
VFPv4_sp	VFPv4 アーキテクチャの単精度派生品に準拠した VFP ユニットを実装するシステム
VFP9-S	CPU コアの ARM9E ファミリと使用できる VFPv2 アーキテクチャの実装である VFP9-S。そのため、VFP9-S コプロセッサを選択することは、VFPv2 アーキテクチャを選択することと同じです
none（デフォルト）	ソフトウェア浮動小数点ライブラリが使用されます

説明

このオプションは、ベクタ浮動小数点 (VFP) コプロセッサを使用して浮動小数点演算を実行するコードを生成するときに使用します。VFP コプロセッサを選択することで、ソフトウェア浮動小数点ライブラリの使用を、サポートされたすべての浮動小数点演算にオーバーライドします。

関連項目

49 ページの *VFP および浮動小数点演算*。

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

--guard_calls

構文	<code>--guard_calls</code>
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	120 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	<code>--header_context</code>
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	<code>-I path</code>
パラメータ	<code>path</code> <code>#include</code> ファイルの検索パス
説明	このオプションは、 <code>#include</code> 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	207 ページの インクルードファイル検索手順。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [プリプロセッサ] > [追加インクルードディレクトリ]

--interwork

構文

--interwork

説明

このオプションは、相互作用コードを生成するときに使用します。

このオプションでコンパイルされたコードでは、関数は、デフォルトで相互作用型になります。すべてが --interwork オプションでコンパイルされている限り、arm および thumb (--cpu_mode オプションを使用) としてコンパイルされたファイルを混合できます。

注：ARM アーキテクチャ v5 以降でコンパイルされたソースコード、または AEABI 準拠のコードは、デフォルトで相互作用します。

関連項目

306 ページの __interwork。



[プロジェクト] > [オプション] > [一般オプション] > [ターゲット] > [相互作用コードを生成します]

-l

構文

-l[a|A|b|B|c|C|D][N][H] {filename|directory}

パラメータ

a (デフォルト) アセンブラリストファイル。

A C/C++ ソースをコメントとして記述したアセンブラリストファイル。

b 基本アセンブラリストファイル。このファイルは、-la を使用して生成したリストファイルと同様の内容になっていますが、コンパイラが生成する追加情報（ランタイムモデル属性、呼出しフレーム情報、フレームサイズ情報）は含まれていません。*

B 基本アセンブラリストファイル。このファイルは、-lA を使用して生成したリストファイルと同様の内容になっていますが、コンパイラが生成する追加情報（ランタイムモデル属性、呼出しフレーム情報、フレームサイズ情報）は含まれていません。*

c C/C++ リストファイル。

C (デフォルト) アセンブラソースをコメントとして記述した C/C++ リストファイル。

D C/C++ アセンブラソース（命令オフセット、16 進数バイト値を除く）をコメントとして記述したリストファイル。

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。**

ファイル名やディレクトリの指定については、217 ページの **ファイル名またはディレクトリをパラメータとして指定する場合の規則**を参照してください。

説明

このオプションは、アセンブラまたは C/C++ リストをファイルに出力する場合に使用します。このオプションは、コマンドラインで任意の回数使用できます。



オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [リスト]

--legacy

構文

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

パラメータ

RVCT3.0 RVCT3.0 でリンクとリンク可能なオブジェクトコードを生成します。RVCT3.0 でリンクとリンクされる必要のあるコードをエクスポートする場合には、このモードとともに --aeabi オプションを使用してください

説明

このオプションは、古いツールチェーンと互換性のあるコードを生成するときに使用します。



[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]

--lock_regs

構文

```
--lock_regs=registers
```

パラメータ

レジスタ コンマ区切りレジスタ名リスト、およびロックするレジスタ間隔 (R4 ~ R11 の範囲)

説明

このオプションは、指定されたレジスタを使用するコードをコンパイラで生成しないようにするときに使用します。

例

```
--lock_regs=R4
--lock_regs=R8-R11
--lock_regs=R4,R8-R11
```



[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]

--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。

--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 `iccarms myfile1.c myfile2.c myfile3.c --mfc`

関連項目 231 ページの `--discard_unused_publics`、250 ページの `--output, -o`、192 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]

--migration_preprocessor_extensions

構文	<code>--migration_preprocessor_extensions</code>
説明	<p>以前の IAR Systems C または C/C++ コンパイラからコードを移行する必要がある場合に、このオプションを使用します。プリプロセッサ式で以下を使用する場合に、このオプションを使用します。</p> <ul style="list-style-type: none"> ● 浮動小数点数式 ● 基本型名および <code>sizeof</code> ● すべてのシンボル名 (<code>typedef</code> および変数を含む) <p>注： このオプションを使用すると、標準の C に準拠していないコードがコンパイラで許可されるだけでなく、同規格に準拠しているコードが拒否されることもあります。</p> <p>重要： 新しく記述するコードでは、これらの拡張を使用しないでください。これらの拡張は、将来のコンパイラではサポートされなくなる可能性があります。</p>



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [IAR 移行プリプロセッサ拡張を有効にする]

--no_clustering

構文	<code>--no_clustering</code>
説明	<p>このオプションは、静的クラスタ最適化を無効にする場合に使用します。</p> <p>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。</p>
関連項目	196 ページの <i>静的クラスタ</i> を参照してください。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [静的クラスタ]

--no_code_motion

構文	<code>--no_code_motion</code>
説明	<p>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。</p> <p>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。</p>

関連項目

195 ページの *コード移動* を参照してください。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [コード移動]

--no_const_align

構文

--no_const_align

説明

デフォルトでは、サイズが 4 バイト以上のオブジェクトに対してアラインメント 4 がコンパイラで使用されます。このオプションは、コンパイラで const オブジェクトをそれらの型のアラインメントに基づいてアラインメントする場合に使用します。

たとえば、文字列リテラルはアラインメント 1 になります。文字列リテラルは、const char 型のエレメントを持つ配列であり、この型のアラインメントが 1 になるためです。このオプションを使用すると、ROM 空間が節約される可能性があります、場合によっては処理速度が犠牲になります。

関連項目

285 ページの *アラインメント*。



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--no_cse

構文

--no_cse

説明

このオプションは、共通部分式除去を無効にする場合に使用します。

注：このオプションは、最適化レベルが [中] 以上の場合にのみ有効です。

関連項目

194 ページの *共通部分式除去* を参照してください。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [共通部分式除去]

--no_exceptions

構文

--no_exceptions

説明

このオプションを使用して、C++ 言語での例外のサポートを無効にします。throw や try-catch のような例外文、および関数定義の例外仕様によって、エラーメッセージが生成されます。関数宣言の例外仕様は無視されます。このオプションは、--c++ コンパイラオプションと併用した場合にのみ有効です。

アプリケーションで例外が使用されない場合、このオプションを使用して例外のサポートを無効化するよう推奨します。この理由は、例外によってコードサイズが大幅に増加するためです。

関連項目

160 ページの *例外処理*。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++]
および

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++
派生言語] > [C++] > [例外あり]

--no_fragments

構文

--no_fragments

説明

このオプションはセクションフラグメント処理を無効にします。通常、ツールセットは、セクションフラグメント情報をリンカに転送するために IAR 独自の情報を使用します。リンカは、この情報を使用して、未使用のコードおよびデータを削除し、実行可能イメージのサイズをさらに最小化します。コンパイラでこのオプションを使用すると、オブジェクトサイズが小さくなります。

関連項目

81 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。

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

195 ページの *関数インライン化* を参照してください。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [関数インライン化]

--no_loop_align

構文

--no_loop_align

説明

このオプションを使用して、ループ内にあるラベルの 4 バイトのアラインメントを無効にします。このオプションは、Thumb2 モードでのみ役に立ちます。

ARM/Thumb1 モードでは、このオプションは有効ですが何の処理も実行しません。

関連項目

285 ページの *アラインメント*。



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--no_mem_idioms

構文

--no_mem_idioms

説明

このオプションを使用して、メモリ領域を消去、設定またはコピーするコードシーケンスをコンパイラが最適化しないようにします。これを使用しなければ、これらのメモリアクセスのパターン（イディオム）は極端に最適化され、場合によってはランタイムライブラリの呼出しが使用されることもあります。原則的に、変換にはライブラリの呼出し以外のことが関係してきます。



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--no_path_in_file_macros

構文

--no_path_in_file_macros

説明

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

関連項目 378 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 157 ページの *C++ の使用*。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++]
および

[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C++
派生言語] > [C++] > [RTTI あり]

--no_rw_dynamic_init

構文 --no_rw_dynamic_init

説明 このオプションを使用して、静的 C 変数のラ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 253 ページの *--ropi*、254 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	<code>--no_scheduling</code>
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	「197 ページの 命令スケジューリング」。を参照してください。
	 [プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [命令スケジューリング]

--no_static_destruction

構文	<code>--no_static_destruction</code>
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	82 ページの <i>atexit</i> 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	<code>--no_system_include</code>
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、 <code>-I</code> 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	231 ページの <code>--dlib_config</code> 、256 ページの <code>--system_include_dir</code> 。
	 [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 196 ページの *型ベースエイリアス解析* を参照してください。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [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 = "foo";
```

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ようになります。

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--no_unaligned_access

構文

--no_unaligned_access

説明

このオプションは、コンパイラでアラインメントされないアクセスを回避するときに使用します。データアクセスは、通常、パフォーマンス上の理由でアラインメントされて実行されます。ただし、アクセスによっては、特にパックデータ構造体に対する読み込みまたは書き込みの場合、アラインメントされない場合があります。このオプションを使用すると、このようなすべてのアクセスは、アラインメントされないアクセスを拒否するため、小さいデータサイズを使用して実行されます。このオプションは、ARMv6 アーキテクチャ以降で役に立ちます。

関連項目

285 ページの *アラインメント*。



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--no_unroll

構文

--no_unroll

説明

このオプションは、ループ展開を無効にする場合に使用します。

注：このオプションは、最適化レベルが [高] の場合にのみ有効です。

関連項目

194 ページの *ループ展開* を参照してください。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化] > [変換の有効化] > [ループ展開]

--no_warnings

構文

--no_warnings

説明

デフォルトでは、コンパイラはワーニングメッセージを出力します。このオプションは、すべてのワーニングを無効にする場合に使用します。



このオプションは、IDE では使用できません。

--no_wrap_diagnostics

構文 `--no_wrap_diagnostics`

説明 デフォルトでは、診断メッセージ中の長い行は、読みやすくするため複数行に分割されます。このオプションは、診断メッセージのラインラッピングを無効にする場合に使用します。



このオプションは、IDE では使用できません。

-O

構文 `-O[n|l|m|h|hs|hz]`

パラメータ

n	なし * (デバッグサポートに最適)
l (デフォルト)	低 *
m	中
h	高 (バランス)
hs	高 (速度優先)
hz	高 (サイズ優先)

* 「なし」と「低」の最も重要な違いは、「なし」ではすべての非静的変数がその変数のスコープ内全体で有効になることです。

説明 このオプションは、コンパイラでコードを最適する際に使用される最適化レベルを設定する場合に使用します。最適化オプションを指定していない場合は、デフォルトで低の最適化レベルが使用されます。`-o`のみを使用し、パラメータを指定しない場合は、高 (バランス) の最適化レベルが使用されます。

最適化レベルを低くすると、デバッガでプログラムのフローを追跡するのが比較的容易になります。逆に、最適化レベルを高くすると、追跡が比較的難しくなります。

関連項目 191 ページの *コンパイラ最適化の設定*。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [最適化]

--only_stdout

構文

```
--only_stdout
```

説明

コンパイラにおいて、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にも標準出力ストリーム (stdout) を使用する場合に、このオプションを使用します。



このオプションは、IDE では使用できません。

--output, -o

構文

```
--output {filename|directory}  
-o {filename|directory}
```

パラメータ

ファイル名やディレクトリの指定については、217 ページの **ファイル名またはディレクトリをパラメータとして指定する場合の規則**を参照してください。

説明

デフォルトでは、コンパイラで生成されたオブジェクトコード出力は、ソースファイルと同じ名前で、拡張子が `.o` のファイルに配置されます。このオプションは、オブジェクトコード出力用に別の出力ファイル名を明示的に指定する場合に使用します。



このオプションは、IDE では使用できません。

--predef_macros

構文

```
--predef_macros {filename|directory}
```

パラメータ

ファイル名の指定については、217 ページの **ファイル名またはディレクトリをパラメータとして指定する場合の規則**を参照してください。

説明

このオプションは、定義済シンボルを一覧表示するときに使用します。このオプションを使用する場合には、プロジェクトの他のファイルと同一のオプションを使用する必要もあります。

`filename` を指定した場合は、コンパイラは出力をそのファイルに保存します。ディレクトリを指定した場合、コンパイラは出力をそのディレクトリ内のファイル（拡張子 `predef`）に保存します。

このオブジェクトでは、コマンドラインでソースファイルを指定する必要がある点に注意してください。



このオプションは、IDE では使用できません。

--preinclude

構文

```
--preinclude includefile
```

パラメータ

ファイル名の指定については、217 ページの [ファイル名またはディレクトリをパラメータとして指定する場合の規則](#)を参照してください。

説明

このオプションは、コンパイラでソースファイルのリードを開始する前に、指定のインクルードファイルをリードする場合に使用します。これは、アプリケーション全体のソースコードで変更を行う場合（新しいシンボルを定義する場合など）に便利です。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [プリプロセッサ] > [プリインクルードファイル]

--preprocess

構文

```
--preprocess [= [c] [n] [l]] {filename|directory}
```

パラメータ

c	コメントの保持
n	プリプロセスのみ
l	#line ディレクティブを生成

ファイル名やディレクトリの指定については、217 ページの [ファイル名またはディレクトリをパラメータとして指定する場合の規則](#)を参照してください。

説明

このオプションは、プリプロセッサ出力を指定ファイルに生成する場合に使用します。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [プリプロセッサ] > [ファイルへのプリプロセッサ出力]

--public_equ

構文

`--public_equ symbol[=value]`

パラメータ

symbol

定義するアセンブラシンボルの名前

value

定義したアセンブラシンボルの値（任意指定）

説明

このオプションは、EQU ディレクティブを使用してアセンブラ言語でラベルを定義し、PUBLIC ディレクティブを使用してエクスポートする操作と等価です。このオプションは、1つのコマンドラインで複数個使用できます。



このオプションは、IDE では使用できません。

--relaxed_fp

構文

`--relaxed_fp`

説明

このオプションを使用して、コンパイラで言語規則を緩和させ、浮動小数点式の最適化をより積極的に実行します。このオプションは、以下の条件を満たす浮動小数点式のパフォーマンスを向上させます。

- 式に単精度および倍精度の値が両方含まれている
- 倍精度の値が精度を失わずに単精度に変換できる
- 式の結果は単精度に変換されます

倍精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

212 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 Kernighan & Ritchie C 形式で宣言された関数の呼出し
- 先にプロトタイプが宣言されていない `public` 関数の関数定義
- プロトタイプを含まない型の関数ポインタによる間接的な関数呼出し



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [プロトタイプの強制]

--ropi

構文

`--ropi`

説明

このオプションを使用して、アドレスコードおよびリードオンリーのデータへの `pc` 関連の参照を使用するコードをコンパイラで生成します。

このオプションを使用する場合、以下の制限が適用されます。

- C++ の構文は使用できません
- オブジェクト属性 `__ramfunc` は使用できません

- 別の定数、文字列リテラル、関数のアドレスを使用してポインタ定数は初期化できません。ただし、書込み可能な変数は実行時に定数アドレスに初期化することができます

関連項目

プリプロセッサシンボル `__ROPI__` については 245 ページの `--no_rw_dynamic_init` および 378 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ーデータ (ropi)]

--rwp

構文

```
--rwp
```

説明

このオプションを使用して、静的ベースレジスタ (R9) からアドレス書込みが可能なデータへのオフセットを使用するコードをコンパイラで生成します。

このオプションを使用する場合、以下の制限が適用されます。

- オブジェクト属性 `__ramfunc` は使用できません
- ポインタ定数は、書込み可能な変数のアドレスでは初期化できません。
ただし、書込み可能な静的変数は実行時に書込み可能な変数アドレスに初期化することができます

関連項目

プリプロセッサシンボル `__RWPI__` については、245 ページの `--no_rw_dynamic_init` および 378 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 (rwp)]

--section

構文

```
--section OldName=NewName
```

説明

コンパイラは、関数およびデータオブジェクトを、IAR ILINK リンカが参照する指定セクションに配置します。このオプションは、セクションの名前を `OldName` から `NewName` に変更するときに使用します。

異なるアドレス範囲にコードやデータを配置し、@ 表記または `#pragma location` ディレクティブでは不十分な場合に、このオプションが有益です。セクション名の変更時には、対応するリンカ設定ファイルも変更する必要がありますことに、注意してください。

--strict

構文	--strict
説明	デフォルトでは、コンパイラで標準の C/C++ に緩く対応したスーパー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	149 ページの 言語拡張の有効化。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [言語の適合] > [厳密]

--system_include_dir

構文	--system_include_dir path
パラメータ	path システムインクルードファイルのパス。パスの指定については、217 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	231 ページの --dlib_config、246 ページの --no_system_include。



このオプションは、IDE では使用できません。

--thumb

構文	--thumb
説明	このオプションは、デフォルトの関数モードを Thumb に設定するときに使用します。この設定は、相互作用していない限り、プログラムに含まれるすべてのファイルで同じでなければなりません。 注： このオプションの効果は、--cpu_mode=thumb オプションと同じです。

関連項目

238 ページの `--interwork`、306 ページの `__interwork`。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [コード] > [プロセッサのモード] > [Thumb]

--use_c++_inline

構文

```
--use_c++_inline
```

説明

標準の C では、`inline` キーワードに対して C++ の場合とはわずかに異なる動作が使用されます。このオプションを使用すると、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++ の動作となります。

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。コンパイルユニットのひとつでインライン定義を外部と定義することにより、外部の定義を提供する必要があります。



[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [言語 1] > [C 派生言語] > [C99] > [C++ インライン動作]

--use_unix_directory_separators

構文

```
--use_unix_directory_separators
```

説明

このオプションを使用して、DWARF デバッグ情報で `/` を (`¥` の代わりに) ファイルパスのディレクトリ区切り文字として使用します。

このオプションは、UNIX 形式のディレクトリ区切り文字を必要とするデバッガを使用する場合に便利です。



このオプションを設定するには、[プロジェクト] > [オプション] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--vla

構文

```
--vla
```

説明

このオプションを使用して、C99 の可変長配列のサポートを有効にします。こ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 147 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 派生言語] > [VLA の許可]

--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。コンパイラでエラーが検出された場合、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` の使用時はエラーとして処理されます。

関連項目 230 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

この章では、各リンカオプションの詳細なリファレンス情報を説明します。

一般的な構文規則については、215 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
--basic_heap	高度なヒープではなく、基本のヒープを使用します
--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
--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
--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	派生プロセッサを指定します
--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
--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

表 27: 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
--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_builtint_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
--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
--image_input	イメージファイルをセクションに配置します
--inline	小さいルーチンをインライン化します
--keep	シンボルをアプリケーションに強制的に追加します
--log	選択したトピックのログ出力を有効にします
--log_file	ログをファイルに記録します
--mangled_names_in_messages	マングル化名をメッセージに追加します
--map	マップファイルを生成します
--misrac1998	MISRA-C:1998 固有のエラーメッセージを有効化 (『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』を参照)
--misrac2004	MISRA-C:2004 固有のエラーメッセージを有効化 (『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』を参照)
--misrac_verbose	MISRA-C チェックの冗長なロギングを有効化。『IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド』および『IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド』を参照してください
--no_dynamic_rtti_elimination	不要な場合でも動的ランタイム型の情報をインクルードします
--no_exceptions	例外が使用された場合にエラーを出力します
--no_fragments	セクションフラグメント処理を無効化
--no_library_search	自動ランタイムライブラリ検索を無効にします

表 27: 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_locals	ローカルシンボルを ELF 実行可能イメージから削除します
--no_range_reservations	絶対シンボルの範囲予約を無効化
--no_remove	未使用のセクションの削除を無効にします
--no_veneers	ベニアの生成を無効にします
--no_vfe	仮想関数の除去を無効化
--no_warnings	ワーニングの生成を無効にします
--no_wrap_diagnostics	診断メッセージの長い行をラップしません
-o	オブジェクトファイル名を設定。--output のエイリアス
--only_stdout	標準出力のみを使用
--output	オブジェクトファイル名を設定
--pi_veneers	位置独立コードのベニアを生成します
--place_holder	他のツールによってフィルされる ROM の部分を予約するときに使用します (ichcksum により計算されるチェックサムなど)
--redirect	シンボルへの参照を別のシンボルにリダイレクトします
--remarks	リマークを有効化
--search	オブジェクトとライブラリファイルを検索するディレクトリを追加して指定します
--semihosting	デバッグインタフェースでリンクします
--silent	サイレント処理を設定
--strip	デバッグ情報を実行可能イメージから削除します
--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

表 27: リンカオプションの概要 (続き)

リンカオプションの説明

以下では、各コンパイラおよびリンカオプションの詳細なリファレンス情報を説明します。



[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。

--basic_heap

構文

--basic_heap

説明

このオプションを使用して、高度なヒープではなく基本のヒープを使用します。基本ヒープはオーバーヘッドが少なく、ヒープメモリのみを割り当てるアプリケーションの例に適していますが、free は呼び出されません。ヒープの他の使用については、一般的に高度なヒープの方がより効率的です。



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]** を選択します。

--BE8

構文

--BE8

説明

このオプションは、Byte Invariant Addressing モードを指定するときに使用します。

つまり、リンカは命令のバイトオーダーを反転させ、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

50 ページの *バイトオーダー*、286 ページの *バイトオーダー*、263 ページの *--BE32*、234 ページの *--endian*。




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

--BE32

構文	--BE32
説明	<p>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。</p> <p>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。</p>
関連項目	50 ページの <i>バイトオーダー</i> 、286 ページの <i>バイトオーダー</i> 、262 ページの <i>--BE8</i> 、234 ページの <i>--endian</i> 。
	 [プロジェクト] > [オプション] > [一般オプション] > [ターゲット] > [エンディアンモード]

--config

構文	--config <i>filename</i>
パラメータ	ファイル名の指定については、217 ページの <i>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</i> を参照してください。
説明	<p>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 <i>icf</i> 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。</p>
関連項目	<i>リンカ設定ファイル</i> 。
	 [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 <i>symbol</i> [= <i>constant_value</i>]				
パラメータ	<table> <tr> <td><i>symbol</i></td><td>設定ファイルで使用するシンボルの名前。デフォルトでは、値 0（ゼロ）が使用されます。</td></tr> <tr> <td><i>constant_value</i></td><td>設定シンボルの定数値。</td></tr> </table>	<i>symbol</i>	設定ファイルで使用するシンボルの名前。デフォルトでは、値 0（ゼロ）が使用されます。	<i>constant_value</i>	設定シンボルの定数値。
<i>symbol</i>	設定ファイルで使用するシンボルの名前。デフォルトでは、値 0（ゼロ）が使用されます。				
<i>constant_value</i>	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す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 <code>define symbol</code> 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回使用できます。
関連項目	265 ページの <code>--define_symbol</code> 、85 ページの <i>ILINK</i> 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] > [設定ファイルの定義済シンボル]

--cpp_init_routine

構文	<code>--cpp_init_routine routine</code>
パラメータ	<code>routine</code> ユーザ定義 C++ 動的初期化ルーチン。
説明	<p>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。</p> <p>セクション型が <code>INIT_ARRAY</code> または <code>PREINIT_ARRAY</code> 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 <code>__iar_cstart_call_ctors</code> で、標準ライブラリの起動コードで呼び出されます。このオプションは、標準ライブラリを使用していないときに、別ルーチンでこれらのセクション型を処理する必要がある場合に使用します。</p> <p> このオプションを設定するには、[プロジェクト] > [オプション] > [リンカ] > [追加オプション] を選択します。</p>

--cpu

構文	<code>--cpu=core</code>
パラメータ	<code>コア</code> 特定の派生プロセッサを指定します。
説明	このオプションは、コードの生成対象の派生プロセッサを選択する場合に使用します。デフォルトは、ARM7TDMI です。

関連項目

認識されているコアおよびプロセッサマクロセルのリストについては、225 ページの `--cpu` を参照してください。



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

--define_symbol

構文

```
--define_symbol symbol=constant_value
```

パラメータ

<i>symbol</i>	アプリケーションで使用する定数シンボルの名前。
<i>constant_value</i>	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する定数シンボル、すなわちラベルを定義するときに使用します。このオプションは、1つのコマンドラインで複数個使用できます。このオプションは、`define symbol` ディレクティブと異なる点に注意してください。

関連項目

263 ページの `--config_def`、85 ページの `ILINK` とアプリケーション間の相互処理。



[プロジェクト] > [オプション] > [リンカ] > [#define] > [シンボル定義]

--dependencies

構文

```
--dependencies=[i|m] {filename|directory}
```

パラメータ

i (デフォルト)	ファイルの名前のみをリスト化。
m	makefile スタイルでリスト化。

ファイル名やディレクトリの指定については、217 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、入力用に開かれたリンカ設定、オブジェクト、ライブラリファイルの名前を、デフォルトのファイル名拡張子 `i` を持つ1つのファイルにリスト化するときに使用します。

例

--dependencies や --dependencies=i を使用すると、開かれている各入力ファイルの名前とフルパス（ある場合）が独立した行に出力されます。次に例を示します。

```
c:¥myproject¥foo.o
d:¥myproject¥bar.o
```

--dependencies=m を使用した場合は、**makefile** スタイルで出力されます。各入力ファイルについて、**makefile** の依存関係規則を含む行が生成されます。各行は、出力ファイル名、コロン、空白文字、入力ファイル名で構成されます。次に例を示します。

```
a.out: c:¥myproject¥foo.o
a.out: d:¥myproject¥bar.o
```



このオプションは、IDE では使用できません。

--diag_error

構文

```
--diag_error=tag[, tag,...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）。

説明

このオプションは、特定の診断メッセージをエラーとして再分類する場合に使用します。エラーは、実行可能イメージが生成されないような重要度の問題を示します。終了コードはゼロ以外になります。このオプションは、1 つのコマンドラインで複数個使用できます。



[プロジェクト] > [オプション] > [リンカ] > [診断] > [エラーとして処理]

--diag_remark

構文

```
--diag_remark=tag[, tag,...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）。

説明

このオプションは、特定の診断メッセージをリマークとして再分類する場合に使用します。リマークは、最も軽微な診断メッセージです。実行可能イメージに異常動作の原因となる可能性がある構造が存在することを示します。このオプションは、1 つのコマンドラインで複数個使用できます。

注：デフォルトでは、リマークは表示されません。リマークを表示するには、`--remarks` オプションを使用します。



[プロジェクト] > [オプション] > [リンカ] > [診断] > [リマークとして処理]

--diag_suppress

構文

```
--diag_suppress=tag[, tag,...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）。

説明

このオプションは、特定の診断メッセージを無効にする場合に使用します。これらのメッセージは表示されなくなります。このオプションは、1つのコマンドラインで複数個使用できます。



[プロジェクト] > [オプション] > [リンカ] > [診断] > [診断を無効化]

--diag_warning

構文

```
--diag_warning=tag[, tag,...]
```

パラメータ

tag 診断メッセージの番号（たとえば、メッセージ番号 Pe826 など）。

説明

このオプションは、特定の診断メッセージをワーニングとして再分類する場合に使用します。ワーニングは、問題はあるが、リンカでのリンクの途中終了の原因にはならないエラーや漏れを示します。このオプションは、1つのコマンドラインで複数個使用できます。



[プロジェクト] > [オプション] > [リンカ] > [診断] > [ワーニングとして処理]

--diagnostics_tables

構文

```
--diagnostics_tables {filename|directory}
```

パラメータ

ファイル名やディレクトリの指定については、217 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。

--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。



[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ーライド]

--error_limit

構文 `--error_limit=n`

パラメータ

n リンカがリンクを中止するエラー発生回数 (*n* は正の整数)。0 は制限なしを示します。

説明 [`--error_limit`] オプションは、エラー数の上限を指定します。この値を超えると、リンカがリンクを停止します。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	<code>--exception_tables={ncreate unwind cantunwind}</code>						
パラメータ	<table><tr><td><code>ncreate</code> (デフォルト)</td><td>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。</td></tr><tr><td><code>unwind</code></td><td>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。</td></tr><tr><td><code>cantunwind</code></td><td>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 <code>terminate</code> が呼び出されるようにします。</td></tr></table>	<code>ncreate</code> (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。	<code>unwind</code>	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。	<code>cantunwind</code>	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 <code>terminate</code> が呼び出されるようにします。
<code>ncreate</code> (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。						
<code>unwind</code>	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。						
<code>cantunwind</code>	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 <code>terminate</code> が呼び出されるようにします。						
説明	<p>このオプションを使用して、正しい呼出しフレーム情報を持っているが例外情報を持たない関数をリンカでどう処理するかを決定します。</p> <p>コンパイラは、C 関数が正しい呼出しフレーム情報を取得するよう徹底します。アセンブラ言語で記述された関数の場合、アセンブラディレクティブを使用して呼出しフレーム情報を生成する必要があります。</p>						
関連項目	157 ページの <i>C++ の使用</i> 。						



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] を選択します。

--export_builtin_config

構文	<code>--export_builtin_config filename</code>	
パラメータ	ファイル名の指定については、217 ページの <i>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</i> を参照してください。	
説明	デフォルトで使用する設定をファイルにエクスポートします。	
	<div> <div></div> <div>このオプションは、IDE では使用できません。</div> </div>	

--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。このルーチンは、レジスタ R0 で引き渡された値を保持する必要があります。ルーチンが定義されていないければ、エントリは追加されません。



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

-f

構文

-f filename

パラメータ

ファイル名の指定については、217 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、リンカで指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。



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

--force_exceptions

構文	<code>--force_exceptions</code>
説明	<p>このオプションを使用して、リンカのヒューリスティックで例外を使用しないことが指示されていても、リンカが例外テーブルと例外コードをインクルードするようにします。</p> <p>インクルードされるコードに <code>rethrow</code> ではない <code>throw</code> 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 <code>throw</code> 式がなければ、リンカは <code>operator new</code>、<code>dynamic_cast</code>、<code>typeid</code> 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 <code>abort</code> 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ぬ場合、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。</p>
関連項目	157 ページの <i>C++ の使用</i> 。



[プロジェクト] > [オプション] > リンカ > [最適化] > [C++ 例外] > [許可] > [常にインクルード]

--force_output

構文	<code>--force_output</code>
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] を選択します。

--image_input

構文	<code>--image_input filename [,symbol,[section[,alignment]]]</code>	
パラメータ	<i>filename</i>	リンクする raw イメージを含むピュアバイナリファイル。
	<i>symbol</i>	バイナリデータを参照できるシンボル。
	<i>section</i>	バイナリデータを配置するセクション。デフォルトは <code>.text</code> です。
	アラインメント	セクションのアラインメント。デフォルトは 1 です。

説明

このオプションは、通常の入力ファイルの他に、ピュアバイナリファイルをリンクします。ファイルの全体の内容がセクションに配置されます。つまり、ピュアバイナリデータのみを含むことができます。

filename で指定したファイルの内容が配置されるセクションは、*symbol* で指定したシンボルをアプリケーションが必要とする場合にのみ含まれます。セクションを強制的に参照するには、`--keep` オプションを使用します。

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ンメントされ、アプリケーションがシンボル `Bootstrap` を参照している場合（またはコマンドラインオプション `--keep`）にのみ含まれます。

関連項目

272 ページの `--keep`。



[プロジェクト] > [オプション] > [リンカ] > [入力] > [未処理バイナリイメージ]

--inline

構文

```
--inline
```

説明

一部のルーチンは小さいため、ルーチンを呼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出しをルーチン本体と置き換えるようにします。



[プロジェクト] > [オプション] > [リンカ] > [最適化] > [小さいルーチンのインライン化]

--keep

構文

```
--keep symbol
```

パラメータ

symbol

ルートシンボルとして処理されるシンボルの名前。

説明

一般的にリンカでは、アプリケーションに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。

initialization	各バッチに選択されたコピーバッチおよび圧縮をリストします。
libraries	自動ライブラリセクタによって行われるすべての決定をリストします。これには、必要なその他のシンボル (--keep)、リダイレクト (--redirect)、どのランタイムライブラリが選択されたかが含まれることがあります。
modules	アプリケーションにインクルードするよう選択された各モジュールと、インクルードの要因となったシンボルをリストします。
redirects	リダイレクトされたシンボルをリストします。
sections	アプリケーションにインクルードするよう選択された各モジュールとセクションのフラグメント、インクルードの要因となった依存関係をリストします。
unused_fragments	アプリケーションにインクルードされなかったセクションフラグメントをリストします。
veneers	一部のベニア作成および使用統計をリストします。

説明

このオプションは、リンカログ情報を `stdout` に出力するときに使用します。ログ情報は、実行可能なイメージが現在の状態になった原因を把握するために利用できる場合があります。

関連項目

274 ページの `--log_file`。



[プロジェクト] > [オプション] > [リンカ] > [リスト] > [ログの生成]

--log_file

構文	<code>--log_file filename</code>
パラメータ	ファイル名の指定については、217 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	273 ページの <code>--log</code> 。



[プロジェクト] > [オプション] > [リンカ] > [リスト] > [ログの生成]

--mangled_names_in_messages

構文	<code>--mangled_names_in_messages</code>
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（オーバーロードなどに利用）。たとえば、 <code>void h(int, char)</code> は、 <code>_Z1hic</code> になります。



このオプションは、IDE では使用できません。

--map

構文	<code>--map {filename directory}</code>
説明	<p>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 <code>map</code> です。このマップファイルの内容は、以下のとおりです。</p> <ul style="list-style-type: none"> ● リンカのバージョン、現在の日時、使用されたコマンドラインをリストするマップファイルヘッダのリンクの概要 ● ランタイム属性をリストするランタイム属性の概要 ● 配置ディレクティブでソートしアドレス順序で各セクション/ブロックをリストした配置の概要 ● データ範囲、パッキング手法、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]

--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：ポリモーフィックでない型の 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

157 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、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

157 ページの *C++ の使用*。



オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [最適化] > [C++ 例外] > [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。

関連項目

81 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] を選択します。

--no_library_search

構文

--no_library_search

説明

このオプションは、自動ランタイムライブラリ検索を無効にするときに使用します。このオプションは、正しい標準ライブラリの自動追加を無効にします。これは、たとえば、ユーザが構築した標準ライブラリをアプリケーションで必要な場合などに役に立ちます。

このオプションは、自動ライブラリ選択のすべての手順を無効にする点に注意してください。標準ライブラリを使用する場合は、一部の手順を複製しなければならぬことがあります。どの手順を複製するか判断するには、

--log libraries リンカオプションと自動ライブラリ選択を有効にしてともに使用します。



[プロジェクト] > [オプション] > [リンカ] > [ライブラリ] > [自動ランタイムライブラリ選択]

--no_locals

構文

--no_locals

説明

このオプションは、ローカルシンボルを ELF 実行可能イメージから削除するときに使用します。

注：このオプションは、実行可能イメージの DWARF 情報からはローカルシンボルを削除しません。



[プロジェクト] > [オプション] > [リンカ] > [出力]

--no_range_reservations

構文

--no_range_reservations

説明

通常は、リンカは絶対シンボルが使用するすべての範囲をサイズゼロとして予約し、place in コマンドの対象から除外します。

このオプションを使用する場合、これらの予約は無効になり、リンカは絶対シンボルの範囲と重複する形でセクションを自由に配置することができます。



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

--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

81 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] を選択します。

--no_veneers

構文

--no_veneers

説明

このオプションは、実行可能イメージで必要とする場合でもベニアの挿入を無効にするときに使用します。ベニアの挿入を無効にすると、ベニアを必要とする各参照に対して再配置エラーが発生します。

関連項目

86 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] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--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多形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

283 ページの *--vfe* および 178 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++ 仮想関数除去の実行]

--no_warnings

構文

--no_warnings

説明

デフォルトでは、リンカはワーニングメッセージを出力します。このオプションは、すべてのワーニングを無効にする場合に使用します。



このオプションは、IDE では使用できません。

--no_wrap_diagnostics

構文

`--no_wrap_diagnostics`

説明

デフォルトでは、診断メッセージ中の長い行は、読みやすくするため複数行に分割されます。このオプションは、診断メッセージのラインラッピングを無効にする場合に使用します。



このオプションは、IDE では使用できません。

--only_stdout

構文

`--only_stdout`

説明

リンカにおいて、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にも標準出力ストリーム (stdout) を使用する場合に、このオプションを使用します。



このオプションは、IDE では使用できません。

--output, -o

構文

`--output {filename|directory}`
`-o {filename|directory}`

パラメータ

ファイル名やディレクトリの指定については、217 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

デフォルトでは、リンカで生成されたオブジェクト実行可能イメージは、`a.out` という名前のファイルに配置されます。このオプションは、別の出力ファイル名（デフォルトのファイル名拡張子は `out`）を明示的に指定する場合に使用します。



[プロジェクト] > [オプション] > [リンカ] > [出力] > [出力ファイル]

--pi_veneers

構文	<code>--pi_veneers</code>
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	86 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] > [C/C++ コンパイラ] > [追加オプション] を選択します。

--place_holder

構文	<code>--place_holder symbol[,size[,section[,alignment]]]</code>		
パラメータ	<i>symbol</i>	作成するシンボルの名前。	
	<i>size</i>	ROM のサイズ、デフォルトは 4 バイト。	
	<i>section</i>	使用するセクション名。デフォルトは .text。	
	<i>alignment</i>	セクションのアラインメント。デフォルトは 1。	

説明 このオプションは、たとえば、ielftool により計算されるチェックサムなど、他のツールによってフィルされる ROM の部分を予約するときに使用します。このリンカオプションを使用するたびに、指定した名前、サイズ、アラインメントのセクションが生成されます。シンボルは、そのセクションを参照するためにアプリケーションで使用できます。


注：他のセクションと同様、--place_holder オプションにより作成されるセクションは、そのセクションが必要だとみなされた場合のみアプリケーションに含まれます。--keep リンカオプション、または keep リンカディレクティブを使用すると、このようなセクションを強制的に含めることができます。

関連項目 423 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] を選択します。

--redirect

構文	<code>--redirect from_symbol=to_symbol</code>	
パラメータ	<code>from_symbol</code>	変更前のシンボルの名前。
	<code>to_symbol</code>	変更後のシンボルの名前。
説明	このオプションは、シンボルの参照を変更するときに使用します。	
	 このオプションを設定するには、[プロジェクト] > [オプション] > [リンカ] > [追加オプション] を選択します。	

--remarks

構文	<code>--remarks</code>	
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	212 ページの <i>重要度</i> 。	
	 [プロジェクト] > [オプション] > [リンカ] > [診断] > [リマークの有効化]	

--search

構文	<code>--search path</code>	
パラメータ	<code>path</code>	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	<p>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。</p> <p>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。</p>	

関連項目 69 ページの *リンク処理*。



このオプションは、IDE では使用できません。

--semihosting

構文 --semihosting[=iar_breakpoint]

パラメータ

iar_breakpoint IAR 固有のメカニズムは、SWI/SVC を広範囲に使用するアプリケーションのデバッグ時に使用できます。

説明 このオプションは、デバッグインタフェース（ブレイクポイントメカニズム）を出力イメージに含めるときに使用します。パラメータを指定しない場合、SWI/SVC メカニズムが ARM7/9/11 用に含まれ、BKPT メカニズムが Cortex-M 用に含まれます。

関連項目 97 ページの *アプリケーションデバッグサポート*。



[プロジェクト] > [オプション] > [一般オプション] > [ライブラリ構成] > [セミホスティング]

--silent

構文 --silent

説明 デフォルトでは、リンカは開始メッセージや最終的な統計レポートを出力します。このオプションは、リンカがこれらのメッセージを標準出力ストリーム（通常は画面）に送信しないようにする場合に使用します。

このオプションは、エラー / ワーニングメッセージの表示には影響しません。



このオプションは、IDE では使用できません。

--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]

--vfe

構文

`--vfe[=forced]`

パラメータ

`forced`

1 つまたは複数のモジュールに必要な仮想関数除去の情報が不足している場合でも、仮想関数除去を実行します。

説明

このオプションは、仮想関数除去の最適化を有効にするときに使用します。デフォルトでは、この最適化は有効になっています。

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

278 ページの `--no_vfe` および 178 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++ 仮想関数除去の実行]

--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。リンカでエラーが検出された場合、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` 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 267 ページの `--diag_warning`、230 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]

データ表現

この章では、コンパイラでサポートされているデータ型、ポインタ、構造体について説明します。

特定のアプリケーションで最も効率的なコードを作成するためのデータ型やポインタについては、「[組込みアプリケーション用の効率的なコーディング](#)」を参照してください。

アラインメント

すべての C データオブジェクトには、オブジェクトをメモリ内に記憶する方法を指定するためのアラインメントが設定されています。たとえば、オブジェクトのアラインメントが 4 の場合は、このオブジェクトは 4 で割り切れるアドレスに格納する必要があります。

一部のプロセッサではメモリのアクセス方法に制限があるため、アラインメントのコンセプトが採用されています。

4 で割り切れるアドレスにメモリ読取りが設定された場合にのみ、プロセッサが 1 回の命令で 4 バイトのメモリを読み取ることができます。この場合、long 型の整数など、4 バイトオブジェクトのアラインメントは 4 になります。

また、一度に 2 バイトしか読み取ることができないプロセッサの場合には、4 バイトの long 型整数のアラインメントは 2 になります。

構造体のアラインメントは、アラインメントが最も厳密な構造体メンバと同じです。構造体およびそのメンバのアラインメント要件を下げるには、`#pragma pack` または `__packed` データ型属性を使用します。

すべてのデータ型は、それらのアラインメントの倍数のサイズにする必要があります。これ以外については、アラインメントの最初のエレメントのみ配列の要件に従って配置されることが保証されます。つまり、コンパイラが構造体の最後にパッドバイトを追加しなければならないことがあります。パッドバイトについては、[296 ページのパック構造体型](#)を参照してください。

`#pragma data_alignment` ディレクティブを使用すると、特定の変数のアラインメント要件を上げることができます。

ARM コアのアラインメント

データオブジェクトのアラインメントは、データオブジェクトがメモリでどのように格納されるかを制御します。アラインメントを使用する理由は、4 バイトオブジェクトが 4 で割り切れるアドレスに格納されている場合、ARM コアがより効率的に 4 バイトオブジェクトにアクセスできるためです。

アラインメント 4 のオブジェクトは、4 で割り切れるアドレスに格納する必要があります、アラインメント 2 のオブジェクトは、2 で割り切れるアドレスに格納する必要があります。

コンパイラでは、すべてのデータ型のアラインメントを割り当てることで、このようなデータの配置を保証し、ARM コアが確実にデータを読み取ることができるようにしています。

関連情報については、223 ページの `--align_sp_on_irq`、242 ページの `--no_const_align` を参照してください。

バイトオーダー

ARM コアは、リトルエンディアンまたはビッグエンディアンバイトオーダーのいずれかでデータを格納します。バイトオーダーを指定するには、`--endian` コンパイラオプションを使用します (234 ページの `--endian` を参照)。

デフォルトのリトルエンディアンバイトオーダーでは、**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ーでは、**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ーを使用する場合、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、288 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

コンパイラは、標準の C の基本データ型のすべてと追加のデータ型の両方をサポートします。

整数型

以下の表に、各整数型のサイズと範囲の一覧を示します。

データ型	サイズ	範囲	アラインメント
bool	8 ビット	0 ~ 1	1
char	8 ビット	0 ~ 255	1
signed char	8 ビット	-128 ~ 127	1
unsigned char	8 ビット	0 ~ 255	1
signed short	16 ビット	-32768 ~ 32767	2
unsigned short	16 ビット	0 ~ 65535	2
signed int	32 ビット	$-2^{31} \sim 2^{31}-1$	4
unsigned int	32 ビット	0 ~ $2^{32}-1$	4
signed long	32 ビット	$-2^{31} \sim 2^{31}-1$	4
unsigned long	32 ビット	0 ~ $2^{32}-1$	4
signed long long	64 ビット	$-2^{63} \sim 2^{63}-1$	8
unsigned long long	64 ビット	0 ~ $2^{64}-1$	8

表 28: 整数型

符号付変数は、2 の補数フォーマットで表現されます。

bool 型

bool 型は、C++ 言語のデフォルトでサポートされています。言語拡張を有効化し、stdbool.h ファイルをインクルードした場合は、bool 型を C ソースコードでも使用できます。この場合は、ブール値の false および true も使用可能になります。

enum 型

コンパイラでは、enum 定数の保持に必要な最小の型を使用し、unsigned よりも signed を優先します。

IAR システムズの言語拡張が有効化されている場合や、C++ においては、enum 定数および型を long、unsigned long、long long、unsigned long long 型にすることも可能です。

コンパイラで自動的に使用される型より大きな型を使用するには、enum 定数を十分に大きな値で定義します。次に例を示します。

```
/* enum での char 型の使用を無効化 */  
enum Cards{Spade1, Spade2,  
           DontUseChar=257};
```

関連情報については、235 ページの `--enum_is_int` を参照してください。

char 型

char 型は、コンパイラのデフォルトでは符号なしですが、`--char_is_signed` コンパイラオプションを使用して符号付にすることが可能です。ただし、ライブラリは char 型は符号なしでコンパイルされています。

wchar_t 型

wchar_t 型は、サポートされるロケール間で設定された最大の拡張文字セットのすべてのメンバに対して個別のコードを表現できる値の範囲を持つ整数型です。

wchar_t 型は、C++ 言語のデフォルトでサポートされています。wchar_t 型を C ソースコードでも使用するには、ランタイムライブラリから `stddef.h` ファイルをインクルードする必要があります。

ビットフィールド

標準の C では、`int`、`signed int` と `unsigned int` を整数ビットフィールドの基本型として使用できます。標準の C++ および C では、コンパイラで言語拡張が有効になっている場合、任意の整数または列挙型を基本型として使用できます。単純な整数型（`char`、`short`、`int` など）が符号つきまたは符号なしのビットフィールドになるかどうかは、実装によって定義されます。

IAR C/C++ Compiler 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コンテナに配置されます。それぞれのコンテナの中では、バイトオーダを考慮して、最初に使用可能なバイト内にビットフィールドが配置されます。

また、異なる型のビットフィールドコンテナが重複できない場合、コンパイラは代替のビットフィールド割当て方式（分割された型）に対応します。この割当て方式を使用すると、各ビットフィールドは型が前のビットフィールド

のものと異なったり、前のビットフィールドと同じコンテナにビットフィールドが合わない場合、新しいコンテナに配置されます。各コンテナ内では、ビットフィールドは最下位のビットから最上位ビット（分割された型）へ、あるいは最上位ビットから最下位ビット（逆順の分割された型）へと配置されます。この割当て方式では、デフォルトの割当て方式（連結された型）より使用スペースが少なくなることは決してなく、ビットフィールドの型が混在する場合は格段に多くのスペースを使用することがあります。

#pragma bitfield ディレクティブを使用して、ビットフィールドの割当て方式を選択してください（317 ページの *bitfields* を参照）。

例

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。

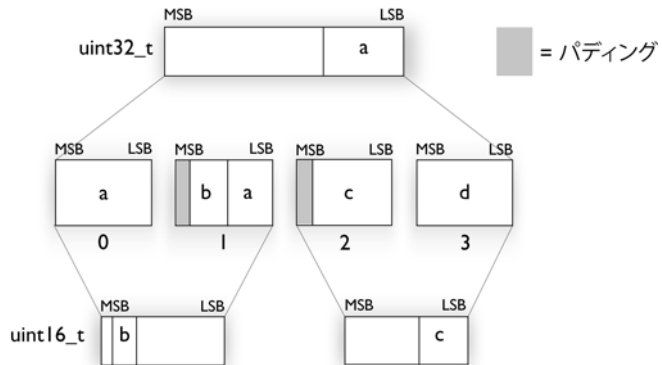


図 14: リトルエンディアンモードにおける結合された型のビットフィールドメンバのレイアウト

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。

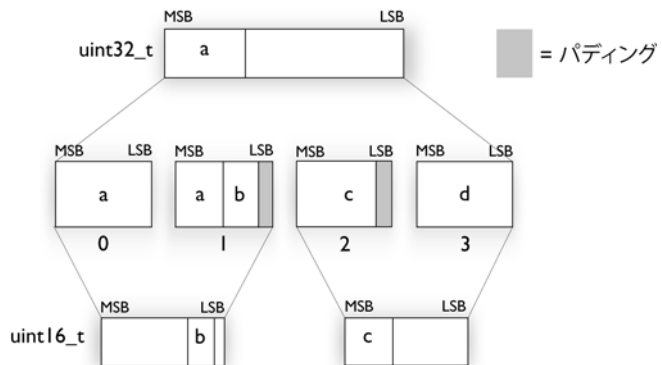


図 15: ビッグエンディアンモードにおける結合された型のビットフィールドメンバのレイアウト

分割された型のビットフィールド割当て方式の例

最初のビットフィールド `a` を配置するために、コンパイラはオフセット 0 に 32 ビットを割当て、`a` を最下位の 12 ビットのコンテナに入れます。

2 番目のビットフィールド `b` を配置するために、新しいコンテナがオフセット 4 に割当てられます。これは、ビットフィールドの型が前のビットフィールドと同じではないためです。`b` は、このコンテナの最下位の 3 ビットに配置されます。

3 番目のビットフィールド `c` は `b` と同じ型なので、同じコンテナに入ります。

4 番目のメンバ `d` は、オフセット 6 のバイトに割り当てられます。`d` は、`b` や `c` と同じコンテナには配置できません。その理由は、これがビットフィールドではなく、同じ型でもないために、合わないからです。

逆順（逆順の分割された型）を使用する際は、各ビットフィールドはそのコンテナの最上位ビットから配置されます。

これは、リトルエンディアンモードの `bitfield_example` のレイアウトです。

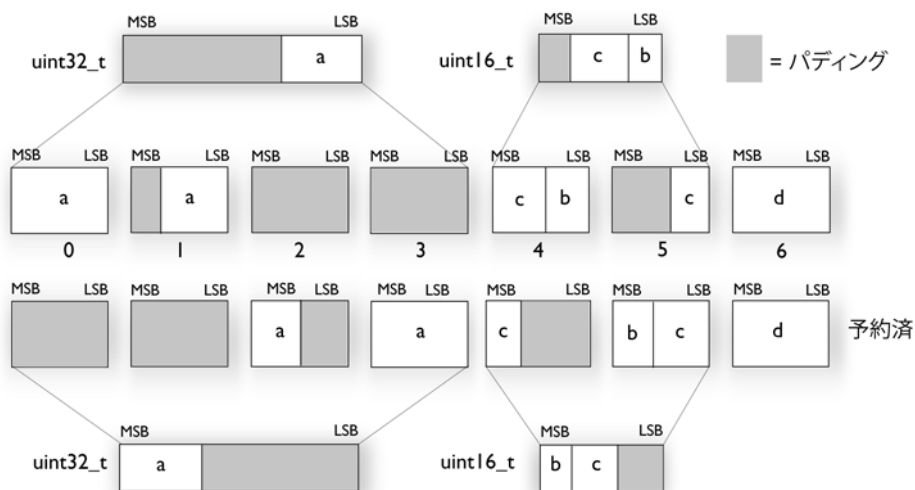
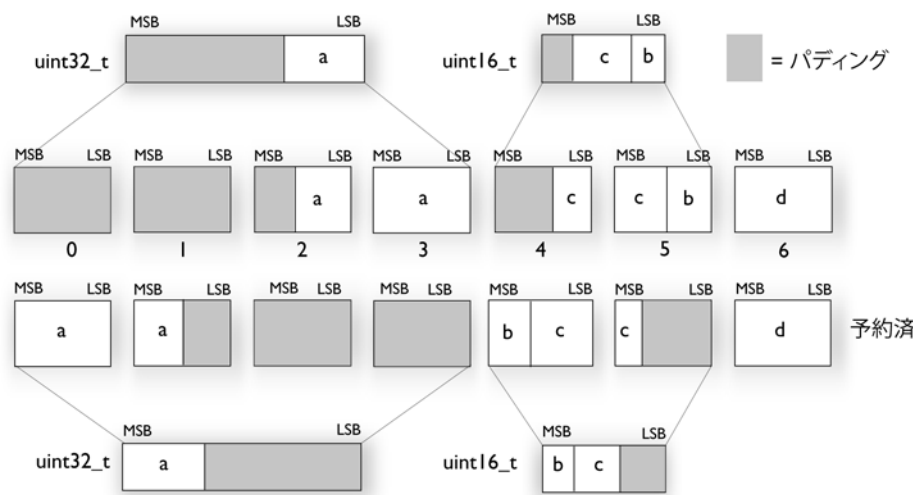


図 16: リトルエンディアンモードでの分割された型の `bitfield_example` のレイアウト

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。



浮動小数点数型

IAR C/C++ Compiler for ARM では、浮動小数点数の値を標準 IEEE 754 フォーマットで表現します。各浮動小数点数型のサイズを以下に示します。

型	サイズ	範囲 (+/-)	10 進数	指数部	仮数部	アラインメント
float	32 ビット	$\pm 1.18\text{E}-38 \sim \pm 3.40\text{E}+38$	7	8 ビット	23 ビット	4
double	64 ビット	$\pm 2.23\text{E}-308 \sim \pm 1.79\text{E}+308$	15	11 ビット	52 ビット	8
long double	64 ビット	$\pm 2.23\text{E}-308 \sim \pm 1.79\text{E}+308$	15	11 ビット	52 ビット	8

表 29: 浮動小数点数型

Cortex-M0 と 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、294 ページの特殊な浮動小数点数の表現。を参照してください。

浮動小数点環境

浮動小数点値の例外フラグは、VFP ユニットを持つデバイスでサポートされており、これらは `fenv.h` ファイルで定義されます。VFP ユニットを持たないデフォルトバイスの場合、`fenv.h` ファイルで定義された関数は存在しますが、これらに機能はありません。

`feraiseexcept` 関数は、`FE_OVERFLOW` や `FE_UNDERFLOW` を使用して呼び出された場合、`inexact` 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。

31	30	23	22	0
S	指数部			仮数部

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。

$$(-1)^S * 2^{(\text{指数部} - 127)} * 1.\text{仮数部}$$

範囲は少なくとも以下のようになります。

$$\pm 1.18\text{E-}38 \text{ から } 3.39\text{E+}38$$

浮動小数点数の演算子 (+、-、*、/) 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。

63	62	52	51	0
S	指数部			仮数部

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。

$$(-1)^S * 2^{(\text{指数部} - 1023)} * 1.\text{仮数部}$$

範囲は少なくとも以下のようになります。

$$\pm 2.23\text{E-}308 \sim \pm 1.79\text{E+}308$$

浮動小数点数の演算子 (+、-、*、/) は、約 15 桁です。

特殊な浮動小数点数の表現

特殊な浮動小数点数の表現を以下に挙げます。

- ゼロは、仮数部や指数部でゼロとして表現されます。符号ビットは、正または負のゼロを示します
- 無限大は、指数部を最大値に、仮数部をゼロに設定することで表現されます。符号ビットは、正または負の無限大を示します
- 非数 (NaN) は、指数部を最大値に設定し、少なくとも 1 ビットを仮数部の上位 20 ビットに設定して、表現されます。残りのビットはゼロです
- 非数 (NaN) は、指数部を最大値に設定し、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、正規化数で表現可能な値よりも小さな値を表現する場合に使用されます。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

$$(-1)^S * 2^{(1-BIAS)} * 0.\text{仮数部}$$

ここで、BIAS は 32 ビットおよび 64 ビット浮動小数点値の場合、それぞれ 127 および 1023 です。

ポインタ型

コンパイラには、関数ポインタとデータポインタという 2 種類の基本ポインタ型があります。

関数ポインタ

関数ポインタのサイズは常に 32 ビットで、範囲は 0x0-0xFFFFFFFF です。

関数ポインタ型が宣言されると、属性が * 記号の前に挿入されます。次に例を示します。

```
typedef void (__thumb __interwork *IntHandler)(void);
```

これは、#pragma ディレクティブを使用して再書き込みすることができます。

```
#pragma type_attribute=__thumb __interwork
typedef void IntHandler_function(void);
typedef IntHandler_function *IntHandler;
```

データポインタ

使用できるデータポインタは 1 つだけです。このサイズは 32 ビットで、範囲は 0x0-0xFFFFFFFF です。

キャスト

ポインタ間のキャストには以下の特徴があります。

- *整数型の値*からそれよりも小さな型のポインタへのキャストは、切捨てにより実行されます
- 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
- 符号付き整数型の *値*からそれよりも大きな型のポインタへのキャストは、符号拡張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

size_t

size_t は、オブジェクトの最大サイズの格納に必要な符号なし整数型です。IAR C/C++ Compiler for ARM では、size_t のサイズは 32 ビットです。

ptrdiff_t

ptrdiff_t は、同一配列のエレメントと 2 つのポインタ間の差を格納するために必要な符号付整数型です。IAR C/C++ Compiler for ARM では、ptrdiff_t のサイズは 32 ビットです。

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++ Compiler for ARM では、intptr_t のサイズは 32 ビットです。

uintptr_t

uintptr_t は、符号なしであることを除き、intptr_t と同じです。

構造体型

struct のメンバは、宣言された順に連続して格納されます。最初のメンバが最下位のメモリアドレスを持ちます。

アラインメント

struct 型や union 型のアラインメントは、最高のアラインメント要件のメンバと同じになります。また、struct のサイズは、アラインメントされた構造体オブジェクトの配列が可能になるように調整されます。

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、指定したアラインメント（オフセット）に従って struct 内に配置されます。

例

```
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。

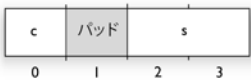


図 18: 構造体のレイアウト

構造体のアラインメントは 2 バイトです。また、short s に正しいアラインメントを与えるためにパッドバイトが挿入されている必要があります。

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ることがあります。

正しくアラインメントされていないオブジェクトにアクセスする場合には、コードのサイズが大きくなり速度が低下します。そのような構造体へのアクセスが多数ある場合、パックされていない struct に正しい値を構成し、この struct にアクセスする方が通常は適しています。

アラインメントが正しく設定されていないメンバへのポインタ作成および使用には、特別な注意も必要です。パックされた struct のアラインメントが正しく設定されていないメンバに直接アクセスする場合、コンパイラは、必要に応じて正しいコード（ただし、サイズが大きく低速）を出力します。しかし、アラインメントが正しく設定されていないメンバへのポインタを使用してそのメンバにアクセスする場合には、通常のコード（サイズが小さく高速）が使用されます。通常、このコードは正しく動作しません。

例

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()
```

この例の場合、構造体 s のメモリレイアウトは以下のようになります。



図19: パックされた構造体のレイアウト

以下の例では、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;
};
```

s2 のメモリレイアウトは以下のようになります。

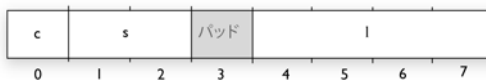


図20: パックされた構造体のレイアウト

構造体 `s` は、前の例で宣言したメモリレイアウト、サイズ、アラインメントを使用します。メンバ `1` のアラインメントは 4 です。これは、構造体 `s2` のアラインメントが 2 になることを意味します。

詳細については、185 ページの [構造体エレメントのアラインメント](#) を参照してください。

型修飾子

C 規格では、`volatile` と `const` は型修飾子です。

オブジェクトの VOLATILE 宣言

オブジェクト `volatile` を宣言することによって、オブジェクトの値がコンパイラの制限以上に変化する可能性があることがコンパイラに伝えられます。またコンパイラは、あらゆるアクセスに副作用があると想定する必要があります。よって、`volatile` オブジェクトへのすべてのアクセスは保持されなければなりません。

オブジェクトを `volatile` として宣言する主な理由は、以下の 3 つです。

- 共有アクセス: マルチタスク環境で、オブジェクトを複数のタスクで共有する場合
- トリガアクセス: メモリマップド SFR のように、アクセス発生により影響が生じる場合
- 変更アクセス: コンパイラが認識できない方法で、オブジェクトの内容が変更される可能性がある場合

volatile オブジェクトへのアクセスの定義

C 規格では、抽象マシンが定義されています。これは、`volatile` 宣言したオブジェクトへのアクセスの動作を制御します。一般的に、抽象マシンに従うコンパイラの動作は、以下のとおりです。

- `volatile` として宣言されたオブジェクトへの各リード/ライトアクセスをアクセスと見なします
- アクセスは、オブジェクト単位になります。複合オブジェクト（配列、構造体、クラス、共用体など）へのアクセスの場合は、エレメント単位になります。次に例を示します

```
char volatile a;
a = 5; /* ライトアクセス */
a += 6; /* 最初はリードアクセスで次にライトアクセス */
```

- ビットフィールドへのアクセスは、その根底型へのアクセスとして処理されます
- `const` 修飾子を `volatile` オブジェクトに追加すると、オブジェクトへのライトアクセスが不可能になります。ただし、オブジェクトは C 規格で指定されたとおりに RAM 内に配置されます

ただし、これらの規則は大まかなもので、ハードウェア関連の要件には対応していません。IAR C/C++ Compiler for ARM に固有の規則について、以下で説明します。

アクセス規則

IAR C/C++ Compiler for ARM では、`volatile` 宣言したオブジェクトは、以下の規則に従います。

- すべてのアクセスが実行されます
- すべてのアクセスは最後まで実行されます。すなわち、オブジェクト全体がアクセスされます
- すべてのアクセスは、抽象マシンでの場合と同一の順序で実行されます
- すべてのアクセスはアトミックアクセスになります。すなわち、割込みはできません

コンパイラは、8 ビット、16 ビット、32 ビットのすべてのスカラ型へのメモリアクセスに関して、これらの規則に準拠しています。ただし、バック構造体型のアラインメントされていない 16 ビットおよび 32 ビットのフィールドへのアクセスは例外です。

記載されていないすべてのオブジェクト型の組合せについては、すべてのアクセスが維持されるという規則だけが適用されます。

オブジェクト VOLATILE および CONST の宣言

`volatile` オブジェクト `const` を宣言する場合、ライト禁止になりますが、C 規格の仕様に従って RAM メモリに格納されます。

代わりにリードオンリーのメモリにオブジェクトを格納して、`const volatile` オブジェクトとしてアクセス可能にするには、以下のように変数を定義します。

```
const volatile int x @ "FLASH";
```

コンパイラは、リード/ライトセクション `FLASH` を生成します。このセクションは ROM に配置して、アプリケーション起動時に変数を手動で初期化するとき 사용합니다。

これ以降は、イニシャライザは他の値とともにいつでも再度フラッシュすることができます。

オブジェクトの CONST 宣言

`const` 型修飾子は、データオブジェクト（直接またはポインタを使用してアクセス）がリードオンリーであることを示します。`const` として宣言したデータへのポインタは、定数と非定数の両方のオブジェクトを参照できます。可能な限り `const` として宣言したポインタを使用することをお勧めします。これにより、コンパイラによる生成コードの最適化が改善され、誤って修正したデータが原因でアプリケーションに障害が発生する危険性が低下します。

`const` として宣言した静的オブジェクトやグローバルオブジェクトは、ROM に配置されます。

C++ では、ランタイムの初期化が必要なオブジェクトは ROM に配置できません。

C++ のデータ型

C++ では、通常の C データ型はすべて、前述の方法で表現されます。ただし、その型で拡張 C++ 機能を使用している場合は、データ表現に関する想定はできなくなります。たとえば、クラスメンバにアクセスするアセンブラコードを記述することはサポートされません。

拡張キーワード

この章では、ARM コア固有の機能をサポートする拡張キーワード、およびそれらのキーワードの一般的な構文規則について説明します。また、この章では、各キーワードの詳細についても説明します。

拡張キーワードの一般的な構文規則

拡張キーワードの構文規則を理解するには、関連するコンセプトに精通することが重要です。

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、305 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。詳しくは、232 ページの *-e* を参照してください。

型属性

型属性は、関数の呼出し方法、またはデータオブジェクトのアクセス方法を定義します。すなわち、型属性を使用する場合には、関数またはデータオブジェクトの定義時と宣言時の両方で型属性を指定する必要があります。

型属性を明示的に宣言に配置するか、プラグマディレクティブ `#pragma type_attribute` を使用します。

汎用型属性

以下の汎用型属性を指定できます。

- **関数型属性** (関数の呼出し方法に影響) : `__arm`、`__fiq`、`__interwork`、`__irq`、`__swi`、`__task`、`__thumb`
- **データ型属性**: `__big_endian`、`const`、`__little_endian`、`__packed`、`volatile`

間接参照レベルごとに、必要な数の属性を指定できます。

型修飾子 `const` と `volatile` の詳細については、298 ページの **型修飾子** を参照してください。

データオブジェクトで使用される型属性の構文

概して、データオブジェクトの型属性の構文は、型修飾子 `const` および `volatile` と同じです。

以下の宣言では、`__little_endian` 型属性が変数 `i` と `j` に割り当てられます。つまり、変数 `i` と `j` は、リトルエンディアンのバイトオーダを使用してアクセスされます。ただし、`struct` や `union` の個々のメンバは型属性を持つことはできません。変数 `k` と `l` の挙動は同一になります。

```
__little_endian int i, j;
int __little_endian k, l;
```

属性は、両方の識別子に影響します。

以下の `i` および `j` の宣言は、前の例と同一の結果になります。

```
#pragma type_attribute=__little_endian
int i, j;
```

キーワードを指定するプラグマディレクティブを使用した場合は、ソースコードの移植性を向上できるという利点があります。

データポインタの型属性の構文

型属性を使用するポインタを宣言するための構文は、型修飾子 `const` および `volatile` と同じです。

<code>int __packed * p;</code>	パックされた整数へのポインタ
<code>int * __packed p;</code>	整数へのパックされたポインタ
<code>__packed int * p;</code>	整数へのパックされたポインタ

関数の型属性の構文

関数の型属性に使用する構文は、データオブジェクトの型属性の構文とはわずかに異なります。関数の場合、以下のように、属性はリターン型の前に置くか、括弧内に置く必要があります。

```
__irq __arm void my_handler(void);
```

または

```
void (__irq __arm my_handler)(void);
```

以下の `my_handler` の宣言は、前の例と同一の結果になります。

```
#pragma type_attribute=__irq __arm
void my_handler(void);
```

オブジェクト属性

オブジェクト属性は通常、関数やデータオブジェクトの内部機能に影響しますが、関数の呼出し方法やデータのアクセス方法には直接影響しません。したがって、通常はオブジェクトの宣言でオブジェクト属性を指定する必要はありません。

以下のオブジェクト属性を指定できます。

- 変数に使用可能なオブジェクト属性: `__absolute` と `__no_init`
- 関数や変数に使用可能なオブジェクト属性: `location`、`@`、`__root`、`__weak`
- 関数に使用可能なオブジェクト属性: `__intrinsic`、`__nested`、`__noreturn`、`__ramfunc`、`__stackless`。

特定の関数やデータオブジェクトに対して、必要な数のオブジェクト属性を指定できます。

`location` および `@` の詳細については、187 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
<code>__absolute</code>	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
<code>__arm</code>	関数を ARM モードで実行します
<code>__big_endian</code>	ビッグエンディアンバイトオーダーを使用する変数を宣言します
<code>__fiq</code>	高速割込み関数を宣言します
<code>__interwork</code>	ARM および Thumb モードの両方から呼出し可能な関数を宣言します
<code>__intrinsic</code>	コンパイラの内部使用専用予約されています
<code>__irq</code>	割込み関数を宣言します
<code>__little_endian</code>	リトルエンディアンバイトオーダーを使用する変数を宣言します
<code>__nested</code>	<code>__irq</code> で宣言した割込み関数をネストできるようにします。 つまり、同じ割込みタイプによる割込みを許可します
<code>__no_init</code>	不揮発性メモリをサポートします
<code>__noreturn</code>	関数がリターンしないことをコンパイラに通知します
<code>__packed</code>	データ型アラインメントを 1 に減らします
<code>__pcrel</code>	<code>--ropi</code> 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
<code>__ramfunc</code>	関数を RAM モードで実行します
<code>__root</code>	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
<code>__sbrel</code>	<code>--rwp</code> 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
<code>__stackless</code>	機能するスタックなしに関数を呼出し可能にします
<code>__swi</code>	ソフトウェア割込み関数を宣言します
<code>__task</code>	レジスタ保護の規則を緩和します
<code>__thumb</code>	関数を Thumb モードで実行します
<code>__weak</code>	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 30: 拡張キーワードの一覧

拡張キーワードの詳細

ここでは、それぞれの拡張キーワードについて詳細に説明します。

__absolute

構文	データに使用可能なオブジェクト属性の汎用構文規則に従います (303 ページの <i>オブジェクト属性</i> を参照)。
説明	<p>__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。</p> <p>次の制限が適用されます。</p> <ul style="list-style-type: none"> ● --ropi または --rwpі 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	<pre>extern __absolute char otherBuffer[100];</pre>

__arm

構文	関数に使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
説明	<p>__arm キーワードは、関数を ARM モードで実行します。__arm により宣言される関数は、__interwork が宣言されていない限り、ARM モードでも実行できる関数からのみ呼出しが可能です。</p> <p>__arm により宣言された関数、__thumb として宣言することはできません。</p> <p>注： 非相互作用 ARM 関数は、Thumb モードから呼び出すことはできません。</p>
例	<pre>__arm int func1(void);</pre>
関連項目	306 ページの <i>__interwork</i> 。

__big_endian

構文	データオブジェクトに使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
----	---

説明	<p><code>__big_endian</code> 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ーに関係なく、ビッグエンディアンバイトオーダーに格納される変数へのアクセスに使用されます。<code>__big_endian</code> 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。</p> <p>このキーワードはポインタ上では使用できません。また、配列上でも使用できません。</p>
例	<code>__big_endian long my_variable;</code>
関連項目	307 ページの <code>__little_endian</code> 。

`__fiq`

構文	関数に使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
説明	<p><code>__fiq</code> キーワードは、高速割込み関数を宣言します。すべての割込み関数は、ARM モードでコンパイルする必要があります。<code>__fiq</code> 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。</p>
例	<code>__fiq __arm void interrupt_function(void);</code>

`__interwork`

構文	関数に使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
説明	<p><code>__interwork</code> により宣言された関数は、ARM または Thumb のいずれかのモードで実行する関数から呼び出すことができます。</p> <p>注： デフォルトでは、関数は、<code>--interwork</code> コンパイラオプションが使用される場合、および <code>--cpu</code> オプションが使用され、相互作用がデフォルトであるコアを指定する場合に、相互作用になります。</p>
例	<code>typedef void (__thumb __interwork *IntHandler)(void);</code>

`__intrinsic`

説明	<code>__intrinsic</code> 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	関数に使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
説明	__irq キーワードは、割込み関数を宣言します。すべての割込み関数は、ARM モードでコンパイルする必要があります。__irq により宣言された関数には、パラメータを渡すことができないため、値を返しません。
例	<pre>__irq __arm void interrupt_function(void);</pre>

__little_endian

構文	データオブジェクトに使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
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ーに関係なく、リトルエンディアンバイトオーダーに格納される変数へのアクセスに使用されます。__little_endian キーワードは、ARMv6 以上でコンパイルする場合に使用できます。 このキーワードはポインタ上では使用できません。また、配列上でも使用できません。
例	<pre>__little_endian long my_variable;</pre>
関連項目	305 ページの <i>__big_endian</i> 。

__nested

構文	関数に使用可能なオブジェクト属性の汎用構文規則に従います (303 ページの <i>オブジェクト属性</i> を参照)。
説明	__nested キーワードは、ネストされる割込みを許可する割込み関数の起動および終了コードを修正します。これにより、割込みが有効になります。つまり、R14 の SPSR およびリターンアドレスを上書きすることなく、新しい割込みを割込み関数に含めることができます。ネストされた割込みは、__irq により宣言された関数のみでサポートされます。 注： __nested キーワードでは、プロセッサモードがユーザモードまたはシステムモードのいずれかであることが必要です。

例 `__irq __nested __arm void interrupt_handler(void);`

関連項目 62 ページの *ネスト割込み* を参照してください。

__no_init

構文 オブジェクト属性の汎用構文規則に従います (303 ページの *オブジェクト属性* を参照)。

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 404 ページの *do not initialize* ディレクティブ。

__noreturn

構文 オブジェクト属性の汎用構文規則に従います (303 ページの *オブジェクト属性* を参照)。

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。

例 `__noreturn void terminate(void);`

__packed

構文 データに使用可能な型属性の汎用構文規則に従います (301 ページの *型属性* を参照)。

説明 `__packed` キーワードは、データ型アライメントを 1 に減らすときに使用します。`__packed` は、以下の 2 つの目的に使用できます。

- `struct` または `union` 型属性とともに使用すると、その `struct` または `union` のメンバの最大アラインメントが 1 に設定され、メンバ間のすべてのギャップが解消されます。また、各メンバの型は、`__packed` 型属性を取り込みます

- それ以外の型に使用する場合、その結果は、`__packed` 型属性を持たない型と同一の型になりますが、アラインメントは 1 に設定されます。すでにアラインメントが 1 に設定されている型は、`__packed` 型属性の影響を受けません

通常のポインタを `__packed` へのポインタに明示的に変換することは可能ですが、その逆の変換にはキャストが必要になります。

注： 通常のアラインメント以外のアラインメントでそのデータ型にアクセスする場合、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。

例

```
__packed struct X {char ch; int i;};           /* パッドバイトなし */
void Foo (struct X * xp)                       /* ここでは __packed 不要 */
{
    int * p1      = &xp->i; /* エラー："int __packed *" */
    int __packed * p2 = &xp->i;           /* OK */
    char * p2      = &xp->ch;           /* OK, char に影響なし */
}
```

関連項目 325 ページの *pack*。

`__ramfunc`

構文 オブジェクト属性の汎用構文規則に従います (303 ページの *オブジェクト属性* を参照)。

説明 `__ramfunc` キーワードは、関数を RAM モードで実行します。RAM 実行用 (`.textwr`) と ROM 初期化用 (`.textwr_init`) の 2 つのコードセクションが作成されます。

`__ramfunc` により宣言された関数が ROM にアクセスしようすると、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き込みするなど、アップグレードルーチンの作成を簡素化することです。

`__ramfunc` により宣言した関数ではない場合、これらの警告は無視するか、無効にしても問題はありません。

`__ramfunc` により宣言した関数は、デフォルトでは `.textwr` という名前のセクションに格納されます。

例


```
__ramfunc int FlashPage(char * data, char * page);
```

関連項目 ブレークポイントに関して `__ramfunc` 宣言された関数の詳細については、『*ARM® 用 C-SPY® デバッガガイド*』を参照してください。

__root

構文	オブジェクト属性の汎用構文規則に従います (303 ページの <i>オブジェクト属性</i> を参照)。
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	<pre>__root int myarray[10];</pre>
関連項目	ルートシンボルおよびその保持方法に関する詳細については、81 ページの <i>シンボルおよびセクションの保持</i> を参照してください。

__stackless

構文	関数に使用可能なオブジェクト属性の汎用構文規則に従います (303 ページの <i>オブジェクト属性</i> を参照)。
説明	__stackless キーワードは、機能するスタックなしに呼出し可能な関数を宣言します。
	 ワーニング: 関数により宣言された __stackless は呼出し規約に違反しているため、そこから戻ることはできません。ただし、コンパイラは関数が戻るかどうかを確実に検出することはできず、検出してもエラーを出力しません。
例	<pre>__stackless void start_application(void);</pre>

__swi

構文	関数に使用可能な型属性の汎用構文規則に従います (301 ページの <i>型属性</i> を参照)。
説明	__swi キーワードは、ソフトウェア割込み関数を宣言します。関数呼出しを正しく実行するために必要な SVC (以前の SWI) 命令と指定のソフトウェア割込み番号が挿入されます。__swi により宣言された関数は、引数を使用し、値を返すことができます。__swi キーワードを使用することにより、特定のソフトウェア割込み関数に対する正しいリターンシーケンスがコンパイラで生成されます。ソフトウェア割込み関数は、スタックの使用以外、パラメータおよびリターン値に関して通常の関数と同じ呼出し規則に従います。

`__swi` キーワードには、`#pragma swi_number=number` ディレクティブで指定されるソフトウェア割込み番号が必要です。`swi_number` は、生成されるアセンブラ `SWI` 命令への引数として使用されます。また、**SVC** 割込みハンドラ（たとえば `SWI_Handler`）が複数のソフトウェア割込み関数を含んだシステムで 1 つのソフトウェア割込み関数を選択するときにも使用できます。ソフトウェア割込み番号は、関数宣言のみに指定（通常、割込み関数を呼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）する必要がある点に注意してください。関数定義には指定しないでください。

注： Cortex-M を除くすべての割込み関数は、ARM モードでコンパイルする必要があります。必要に応じて `__arm` キーワードまたは `#pragma type_attribute=__arm` ディレクティブを使用して、デフォルトの動作を変更してください。

例

ソフトウェア割込み関数の宣言は、通常、ヘッダファイルで行い、以下の例のように記述します。

```
#pragma swi_number=0x23
__swi int swi0x23_function(int a, int b);
...
```

関数の呼出し

```
...
int x = swi0x23_function(1, 2); /* SVC 0x23 によって配置されるため、
                                リンカは決して swi0x23_ 関数を配置
                                しようとはしません */
...
```

アプリケーションのソースコード内でソフトウェア割込み関数を定義

```
...
__swi __arm int the_actual_swi0x23_function(int a, int b)
{
    ...
    return 42;
}
```

関連項目

63 ページの *ソフトウェア割込み*、136 ページの *呼出し規約*。

__task

構文

関数に使用可能な型属性の汎用構文規則に従います（301 ページの *型属性* を参照）。

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 を使用するのには、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

関数に使用可能な型属性の汎用構文規則に従います (301 ページの *型属性* を参照)。

説明

__thumb キーワードは、関数を Thumb モードで実行します。関数が __interwork により宣言されていない限り、__thumb により宣言された関数の呼出しは、Thumb モードで実行する関数からのみ行うことができます。

関数により宣言される __thumb は、__arm として宣言することはできません。

注：非相互作用 Thumb 関数は、ARM モードから呼び出すことはできません。

例

```
__thumb int func2(void);
```

関連項目

306 ページの *__interwork*。

__weak

構文

オブジェクト属性の汎用構文規則に従います (303 ページの *オブジェクト属性* を参照)。

説明

__weak オブジェクト属性をシンボルの外部宣言に使用することにより、モジュール内でのそのシンボルへのすべての参照が弱参照になります。

シンボルの公開されている定義上で `__weak` オブジェクト属性を使用すると、その定義は弱くなります。

リンカは、シンボルへの弱い参照を満たすためにライブラリからモジュールをインクルードすることはなく、弱い参照の定義の不足がエラーにつながることもありません。定義がインクルードされない場合、オブジェクトのアドレスはゼロになります。

リンク処理の際、シンボルは弱い定義を必要な数だけと、最大で弱くない定義を1つ持つことができます。シンボルが必要な場合は、弱くない定義が1つあり、この定義が使用されます。弱くない定義がない場合は、弱い定義のいずれかが使用されます。

例

```
extern __weak int foo; /* 弱い参照 */

__weak void bar(void) /* 弱い定義 */
{
    /* インクルードされた場合はfooをインクリメント */
    if (&foo != 0)
        ++foo;
}
```


プラグマディレクティブ

この章では、コンパイラのプラグマディレクティブについて説明します。

#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。

プラグマディレクティブの一覧

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
data_alignment	変数のアラインメントを高く（より厳密に）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
error	解析の際にエラーについて警告
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

表 31: 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
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 section 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
object_attribute	変数または関数の定義を変更します
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ンメントを指定します
__printf_args	printf スタイルフォーマット文字列の関数の呼出しに使用されている引数が正しいかどうかを検証します
required	別のシンボルによって必要とされるシンボルが確実にリンク出力に含まれるようにします
rtmodel	ランタイムモデル属性をモジュールに追加します
__scanf_args	scanf スタイルフォーマット文字列の関数の呼出しに使用されている引数が正しいかどうかを検証します
section	組み関数で使用するセクション名を宣言
segment	このディレクティブは #pragma section のエイリアスです
STDC CX_LIMITED_RANGE	コンパイラで通常の複雑な数式を使用できるかどうかを指定します
STDC FENV_ACCESS	ソースコードが浮動小数点環境にアクセス可能かどうかを指定します
STDC FP_CONTRACT	コンパイラが浮動小数点式を縮約できるかどうかを指定します
swi_number	ソフトウェア割込み関数の割込み番号を設定します
type_attribute	変数または関数の宣言および定義を変更します
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 31: プラグマディレクティブの一覧 (続き)

注：移植性のため、プラグマディレクティブ alignment、baseaddr、codeseg、constseg、dataseg、function、memory、warnings も認識されますが、使用すると診断メッセージが出力されます。こうしたプラグマディレクティブを含む既存のコードを移植する必要がある場合は、この点に注意してください。[460 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憶領域コンテナは、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、288 ページの <i>ビットフィールド</i> 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	288 ページの <i>ビットフィールド</i> 。	

data_alignment

構文	<code>#pragma data_alignment=expression</code>
パラメータ	<code>expression</code> 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	<p>このプラグマディレクティブは、変数に与える開始アドレスのアラインメントを通常よりも高く（より厳密に）する場合に使用します。このディレクティブは、静的 / 自動変数に対して使用できます。</p> <p>このディレクティブを自動変数に対して使用する場合は、各関数で指定可能なアラインメントに上限が設けられます。この上限は、使用する呼出し規約によって決定されます。</p> <p>注：通常、変数のサイズは、そのアラインメントの倍数です。data_alignment ディレクティブは、変数の開始アドレスのみに影響し、サイズには影響しません。そのため、サイズがアラインメントの倍数ではない状況に使用できます。</p>

diag_default

構文	<code>#pragma diag_default=tag[, tag, ...]</code>
パラメータ	<code>tag</code> 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	<p>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すことや、オプション <code>--diag_error</code>、<code>--diag_remark</code>、<code>--diag_suppress</code>、<code>--diag_warnings</code> のいずれか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せん。</p>
関連項目	211 ページの <i>診断</i> 。

diag_error

構文	<code>#pragma diag_error=tag[, tag, ...]</code>
パラメータ	<code>tag</code> 診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）。

説明 このプラグマディレクティブは、指定した診断メッセージの重要度を `error` に変更する場合に使用します。

関連項目 211 ページの *診断*。

diag_remark

構文 `#pragma diag_remark=tag[, tag, ...]`

パラメータ `tag` 診断メッセージの番号（たとえば、メッセージ番号 `Pe177` など）。

説明 このプラグマディレクティブは、指定した診断メッセージの重要度を `remark` に変更する場合に使用します。

関連項目 211 ページの *診断*。

diag_suppress

構文 `#pragma diag_suppress=tag[, tag, ...]`

パラメータ `tag` 診断メッセージの番号（たとえば、メッセージ番号 `Pe117` など）。

説明 このプラグマディレクティブは、指定した診断メッセージを無効にする場合に使用します。

関連項目 211 ページの *診断*。

diag_warning

構文 `#pragma diag_warning=tag[, tag, ...]`

パラメータ `tag` 診断メッセージの番号（たとえば、メッセージ番号 `Pe826` など）。

説明 このプラグマディレクティブは、指定した診断メッセージの重要度を `warning` に変更する場合に使用します。

関連項目 211 ページの *診断*。

error

構文	<code>#pragma error message</code>	
パラメータ	<code>message</code>	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 <code>#error</code> とは異なります。 <code>#pragma error</code> ディレクティブは、 <code>_Pragma</code> 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	<pre>#if FOO_AVAILABLE #define FOO ... #else #define FOO _Pragma("error¥\"Foo is not available¥\"") #endif</pre> <p><code>FOO_AVAILABLE</code> がゼロの場合、<code>FOO</code> 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。</p>	

include_alias

構文	<code>#pragma include_alias ("orig_header" , "subst_header")</code> <code>#pragma include_alias (<orig_header> , <subst_header>)</code>	
パラメータ	<code>orig_header</code>	エイリアスを作成するヘッダファイルの名前。
	<code>subst_header</code>	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 <code>#include</code> ディレクティブの前に記述する必要があります。また、 <code>subst_header</code> は、対応する <code>#include</code> 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 [207 ページの インクルードファイル検索手順。](#)

inline

構文

```
#pragma inline[=forced|=never]
```

パラメータ

パラメータなし	<code>inline</code> キーワードと同じ結果になります。
<code>forced</code>	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。
<code>never</code>	コンパイラのヒューリスティックを無効にして、関数がインライン化されないようにします。

説明

`#pragma inline` を使用して、このディレクティブ直後に宣言された関数を、呼出し元関数本体へインライン展開可能であることをコンパイラに通知します。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。

`#pragma inline` は、C++ の `inline` キーワードと同等です。違うのは、`#pragma inline` ディレクティブの場合はコンパイラは C++ のインライン動作を使用しますが、`inline` キーワードには標準の C 動作を使用する点です。

`#pragma inline=never` を指定すると、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。

`#pragma inline=forced` を指定することにより、コンパイラのヒューリスティックに関係なく、ディレクティブ直後に宣言された関数を強制的にインライン化の候補として扱うようにコンパイラに指示します。再帰など何らかの理由でコンパイラが候補の関数をインライン化できない場合、ワーニングメッセージが出力されます。

インライン化は通常、最適化レベル「高」でのみ実行されます。`#pragma inline=forced` を指定すると、最適化レベル「中」でも関数のインライン化が有効になります。

関連項目 [195 ページの 関数インライン化。](#)

language

構文	<code>#pragma language={extended default save restore}</code>	
パラメータ	extended	pragma 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	pragma 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 1	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： <code>#pragma language=extended</code> <code>/* ファイルの残りの部分 */</code>	
例 2	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： <code>#pragma language=save</code> <code>#pragma language=extended</code> <code>/* ソースコードの一部 */</code> <code>#pragma language=restore</code>	
関連項目	232 ページの <code>-e</code> 、256 ページの <code>--strict</code> 。	

location

構文	<code>#pragma location={address register NAME}</code>	
パラメータ	address	絶対位置で指定するグローバル変数または静的変数の絶対アドレス。

<code>register</code>	ARM コアレジスタ R4-R11 のいずれかに対応する識別子。
<code>NAME</code>	ユーザ定義の section 名。コンパイラやリンカで使用される定義済の section 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 section 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 section に配置しないでください

例

```
#pragma location=0xFFFF0400
__no_init volatile char PORT1; /* PORT1 はアドレス
                                番地に置かれる */

#pragma location=R8
__no_init int TASK; /* TASK が R8 に配置される */

#pragma section="FLASH"
#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

187 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。

message

構文

```
#pragma message (message)
```

パラメータ

`message` 標準出力ストリームに転送するメッセージ。

説明 このプラグマディレクティブは、コンパイラでファイルのコンパイル時にメッセージを標準出力ストリームに出力する場合に使用します。

例 :

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

構文 `#pragma object_attribute=object_attribute[,object_attribute,...]`

パラメータ このプラグマディレクティブと使用可能なオブジェクト属性の詳細については、303 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は、オブジェクト属性を持つ変数または関数を宣言する場合に使用します。このディレクティブは、ディレクティブ直後の識別子の定義に影響します。オブジェクトが修正され、その型は修正されません。変数または関数の保存およびアクセスを指定する `#pragma type_attribute` ディレクティブとは異なり、宣言にオブジェクト属性を指定する必要はありません。

例

```
#pragma object_attribute=__no_init
char bar;
```

関連項目 301 ページの *拡張キーワードの一般的な構文規則*。

optimize

構文 `#pragma optimize=[goal][level][no_optimization...]`

パラメータ

<i>goal</i>	以下から選択します。 バランス（速度とサイズのバランスを最適化） size（サイズを重視して最適化） speed（速度を重視して最適化）
<i>level</i>	最適化レベルを指定します。[なし]、[低]、[中]、[高] から選択します。

`no_optimization` 1 つまたは複数の最適化を無効にします。以下から選択してください。

- `no_code_motion` (コード移動を無効化)
- `no_cse` (共通部分式除去を無効化)
- `no_inline` (関数のインライン化を無効化)
- `no_tbaa` (型ベースエイリアス解析を無効化)
- `no_unroll` (ループ展開を無効化)
- `no_scheduling` (命令スケジューリングを無効化)

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 `speed`、`size`、`balanced` は、最適化レベルが高の場合にのみ有効です。ただし、速度優先の最適化とサイズ優先の最適化を同時に実行できないため、指定できるパラメータは、これらのいずれか 1 つです。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 */
}
```

pack

構文

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[,name] [,n])
```

パラメータ

`n` 次のいずれかの中からオプションの構造体アラインメントを設定します：1、2、4、8、16。

空白のリスト	構造体アラインメントをデフォルトに復元します。
<code>push</code>	一時的な構造体アラインメントを設定します。
<code>pop</code>	構造体アラインメントを一時的にプッシュされたアラインメントから復元します。
<code>name</code>	プッシュまたはポップされたオプションのアラインメントラベル。
説明	<p>このプラグマディレクティブは、<code>struct</code> および <code>union</code> メンバの最大アラインメントを指定する場合に使用します。</p> <p><code>#pragma pack</code> ディレクティブは、このプラグマディレクティブの後から次の <code>#pragma pack</code> まで、またはコンパイルユニットの最後まで構造体の宣言に影響します。</p> <p>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。</p>
関連項目	296 ページの <i>構造体型</i> 、308 ページの <i>__packed</i> 。

__printf_args

構文	<code>#pragma __printf_args</code>
説明	このプラグマディレクティブは、 <code>printf</code> 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 <code>%d</code> ）の引数が構文的に正しいかどうかを検証します。
例	<pre>#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 */ }</pre>

required

構文	<code>#pragma required=symbol</code>
パラメータ	<p><i>symbol</i></p> <p>静的にリンクされた関数または変数。</p>

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、変数とその格納場所の **section** 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アプリケーションではわからない場合に使用します。

例

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
    /* 何らかの処理 */
}
```

著作権の文字列がアプリケーションで使用されない場合でも、この文字列がリンカによって含められ、出力に現れます。

rtmodel

構文

```
#pragma rtmodel="key", "value"
```

パラメータ

"key"	ランタイムモデル属性を指定するテキスト文字列
"value"	ランタイムモデル属性の値を指定するテキスト文字列特殊値 * を使用すると、属性が未定義である場合と等価になります。

説明

このプラグマディレクティブは、ランタイムモデル属性をモジュールに追加する場合に使用します。この属性を使用して、モジュール間の整合性のチェックをリンカで行えます。

このプラグマディレクティブは、モジュール間の整合性を確保するために使用できます。一緒にリンクされ、同一のランタイムモジュール属性のキーを定義するすべてのモジュールは、そのキーに対応する値が同一であるか、特殊な * という値を持つ必要があります。ただし、この値を使用することで、モジュールがランタイムモデルに対応していることを明示できます。

1 つのモジュールで複数のランタイムモデルを定義できます。

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

例	<pre>#pragma rtmodel="I2C","ENABLED"</pre> <p>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。</p>
関連項目	125 ページの <i>モジュールの整合性チェック</i> を参照してください。

__scanf_args

構文	<pre>#pragma __scanf_args</pre>
説明	このプラグマディレクティブは、scanf スタイルフォーマット文字列の関数に使用します。コンパイラは、この関数への任意の呼出しに対して、各変換指定子（たとえば %d）の引数が構文的に正しいかどうかを検証します。
例	<pre>#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; }</pre>

section

構文	<pre>#pragma section="NAME" [align] alias #pragma segment="NAME" [align]</pre>				
パラメータ	<table> <tr> <td><i>NAME</i></td><td>section の名前。</td></tr> <tr> <td><i>align</i></td><td>section 値には、定数（2 の累乗の整数）を指定する必要があります。</td></tr> </table>	<i>NAME</i>	section の名前。	<i>align</i>	section 値には、定数（2 の累乗の整数）を指定する必要があります。
<i>NAME</i>	section の名前。				
<i>align</i>	section 値には、定数（2 の累乗の整数）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を使用して、section 演算子 __section_begin、__section_end、__section_size で使用可能な section 名を定義します。特定 section の section 定義はすべて、メモリタイプ属性とアラインメントが同じである必要があります。				

例	<code>#pragma section="MYHUGE" __huge 4</code>
関連項目	151 ページの 専用 <i>section</i> 演算子。 <i>section</i> については、「アプリケーションのリンク」を参照してください。

STDC CX_LIMITED_RANGE

構文	<code>#pragma STDC CX_LIMITED_RANGE {ON OFF DEFAULT}</code>	
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 <code>x</code> (乗算)、 <code>/</code> (除算)、 <code>abs</code> に通常の複雑な数式を使用可能なように指定するときに使用します。 注： このディレクティブは、標準の C 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	<code>#pragma STDC FENV_ACCESS {ON OFF DEFAULT}</code>	
パラメータ	ON	ソースコードは浮動小数点環境にアクセスします。この引数はコンパイラではサポートされていない点に注意してください。
	OFF	ソースコードは浮動小数点環境にアクセスしません。
	DEFAULT	デフォルトの動作を設定します。つまり OFF です。
説明	このプラグマディレクティブを使用して、ソースコードが浮動小数点環境にアクセスするかどうかを指定します。 注： このディレクティブは、標準の C では必須です。	

STDC FP_CONTRACT

構文	#pragma STDC FP_CONTRACT {ON OFF DEFAULT}	
パラメータ	ON	コンパイラが浮動小数点式を縮約できます。
	OFF	コンパイラが浮動小数点式を縮約できません。この引数はコンパイラではサポートされていない点に注意してください。
	DEFAULT	デフォルトの動作を設定します。つまり ON です。
説明	このプラグマディレクティブを使用して、コンパイラが浮動小数点式を縮約できるかどうかを指定します。このディレクティブは、標準の C では必須です。	
例	#pragma STDC_FP_CONTRACT=ON	

swi_number

構文	#pragma swi_number= <i>number</i>	
パラメータ	<i>number</i>	ソフトウェア割込み番号。
説明	このプラグマディレクティブは、__swi 拡張キーワードと一緒に使用します。これは、生成されるアセンブラ SWC 命令への引数として使用されます。また、割込み関数などを複数含んだ 1 つのソフトウェア関数をシステムで選択するときにも使用します。	
例	#pragma swi_number=17	
関連項目	63 ページの <i>ソフトウェア割込み</i> を参照してください。	

type_attribute

構文	<code>#pragma type_attribute=type_attribute[, type_attribute, ...]</code>
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、301 ページの <i>型属性</i> を参照してください。
説明	<p>このプラグマディレクティブは、標準の C には含まれない IAR 固有の <i>型属性</i> 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。</p> <p>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。</p>
例	<p>以下の例では、thumb-mode コードが関数 <code>foo</code> に対して生成されます。</p> <pre>#pragma type_attribute=__thumb void foo(void) { }</pre> <p>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。</p> <pre>__thumb void foo(void) { }</pre>
関連項目	詳細については、 <i>拡張キーワード</i> を参照してください。

weak

構文	<code>#pragma weak symbol1={symbol2}</code>				
パラメータ	<table> <tr> <td><code>symbol1</code></td><td>外部リンケージを持つ関数または変数。</td></tr> <tr> <td><code>symbol2</code></td><td>定義済の関数または変数。</td></tr> </table>	<code>symbol1</code>	外部リンケージを持つ関数または変数。	<code>symbol2</code>	定義済の関数または変数。
<code>symbol1</code>	外部リンケージを持つ関数または変数。				
<code>symbol2</code>	定義済の関数または変数。				
説明	<p>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。</p> <ul style="list-style-type: none"> ● 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、<code>__weak</code> 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

312 ページの `__weak`。

組込み関数

この章では、コンパイラで使用可能な定義済関数である組込み関数について説明します。

組込み関数は、低レベルのプロセッサ処理に直接アクセスするための関数であり、時間が重要なルーチンなどで非常に便利です。組込み関数は、単一の命令か短い命令シーケンスとして、インラインコードにコンパイルされます。

組込み関数の概要

アプリケーションで組込み関数を使用するには、ヘッダファイル `intrinsics.h` をインクルードします。

アプリケーションで Neon 組込み関数を使用するには、ヘッダファイル `arm_neon.h` をインクルードします。詳細については、NEON 命令の組込み関数を参照してください。

組込み関数名は、次のように最初にダブルアンダースコアが付きます。

`__disable_interrupt`

以下の表に、組込み関数の一覧を示します。

組込み関数	説明
<code>__CLZ</code>	CLZ 命令を挿入します
<code>__disable_fiq</code>	高速割り込み要求 (fiq) を無効にします
<code>__disable_interrupt</code>	割り込みを禁止します
<code>__disable_irq</code>	割り込み要求 (irq) を無効にします
<code>__DMB</code>	DMB 命令を挿入します
<code>__DSB</code>	DSB 命令を挿入します
<code>__enable_fiq</code>	高速割り込み要求 (fiq) を有効にします
<code>__enable_interrupt</code>	割り込みを有効にします
<code>__enable_irq</code>	割り込み要求 (irq) を有効にします
<code>__get_BASEPRI</code>	Cortex-M3/Cortex-M4 BASEPRI レジスタの値を返します
<code>__get_CONTROL</code>	Cortex-M CONTROL レジスタの値を返します

表 32: 組込み関数の一覧

組み込み関数	説明
__get_CPSR	ARM CPSR（現在のプログラムステータスレジスタ）の値を返します
__get_FAULTMASK	Cortex-M3/Cortex-M4 FAULTMASK レジスタの値を返します
__get_interrupt_state	割り込み状態を返します
__get_IPSR	IPSR レジスタの値を返します
__get_LR	リンクレジスタの値を返します
__get_MSP	MSP レジスタの値を返します
__get_PRIMASK	Cortex-M PRIMASK レジスタの値を返します
__get_PSP	PSP レジスタの値を返します
__get_PSR	PSR レジスタの値を返します
__get_SB	静的ベースレジスタの値を返します
__get_SP	スタックポインタレジスタの値を返します
__ISB	ISB 命令を挿入します
__LDC	コプロセッサロード命令 LDC を挿入します
__LDCL	コプロセッサロード命令 LDCL を挿入します
__LDC2	コプロセッサロード命令 LDC2 を挿入します
__LDC2L	コプロセッサロード命令 LDC2L を挿入します
__LDC_noidx	コプロセッサロード命令 LDC を挿入します
__LDCL_noidx	コプロセッサロード命令 LDCL を挿入します
__LDC2_noidx	コプロセッサロード命令 LDC2 を挿入します
__LDC2L_noidx	コプロセッサロード命令 LDC2L を挿入します
__LDREX	LDREX 命令を挿入します
__MCR	コプロセッサ書き込み命令 (MCR) を挿入します
__MRC	コプロセッサ読取り命令 (MRC) を挿入します
__no_operation	NOP 命令を挿入します
__PKHBT	PKHBT 命令を挿入します
__PKHTB	PKHTB 命令を挿入します
__QADD	QADD 命令を挿入します
__QADD8	QADD8 命令を挿入します
__QADD16	QADD16 命令を挿入します
__QASX	QASX 命令を挿入します

表 32: 組み込み関数の一覧（続き）

組込み関数	説明
__QCFIag	FPSCR レジスタの累計飽和フラグの値を返します
__QDADD	QDADD 命令を挿入します
__QDOUBLE	QADD 命令を挿入します
__QDSUB	QDSUB 命令を挿入します
__QFlag	オーバフロー / 飽和が発生しているかどうかを示す Q フラグを返します
__QSAX	QSAX 命令を挿入します
__QSUB	QSUB 命令を挿入します
__QSUB8	QSUB8 命令を挿入します
__QSUB16	QSUB16 命令を挿入します
__reset_Q_flag	オーバフロー / 飽和が発生しているかどうかを示す Q フラグをクリアします
__reset_QC_flag	FPSCR レジスタの累計飽和フラグ QC の値をクリアします
__REV	REV 命令を挿入します
__REVSH	REVSH 命令を挿入します
__SADD8	SADD8 命令を挿入します
__SADD16	SADD16 命令を挿入します
__SASX	SASX 命令を挿入します
__SEL	SEL 命令を挿入します
__set_BASEPRI	Cortex-M3/Cortex-M4 BASEPRI レジスタの値を設定します
__set_CONTROL	Cortex-M CONTROL レジスタの値を設定します
__set_CPSR	ARM CPSR（現在のプログラムステータスレジスタ）の値を設定します
__set_FAULTMASK	Cortex-M3/Cortex-M4 FAULTMASK レジスタの値を設定します
__set_interrupt_state	割り込み状態を復元します
__set_LR	リンクレジスタに新しいアドレスを割り当てます
__set_MSP	MSP レジスタの値を設定します
__set_PRIMASK	Cortex-M PRIMASK レジスタの値を設定します
__set_PSP	PSP レジスタの値を設定します

表 32: 組込み関数の一覧 (続き)

組み込み関数	説明
__set_SB	スタティックベースレジスタに新しいアドレスを割り当てます
__set_SP	スタックポインタレジスタに新しいアドレスを割り当てます
__SHADD8	SHADD8 命令を挿入します
__SHADD16	SHADD16 命令を挿入します
__SHASX	SHASX 命令を挿入します
__SHSAX	SHSAX 命令を挿入します
__SHSUB8	SHSUB8 命令を挿入します
__SHSUB16	SHSUB16 命令を挿入します
__SMLABB	SMLABB 命令を挿入します
__SMLABT	SMLABT 命令を挿入します
__SMLAD	SMLAD 命令を挿入します
__SMLADX	SMLADX 命令を挿入します
__SMLALBB	SMLALBB 命令を挿入します
__SMLALBT	SMLALBT 命令を挿入します
__SMLALD	SMLALD 命令を挿入します
__SMLALDX	SMLALDX 命令を挿入します
__SMLALTB	SMLALTB 命令を挿入します
__SMLALTT	SMLALTT 命令を挿入します
__SMLATB	SMLATB 命令を挿入します
__SMLATT	SMLATT 命令を挿入します
__SMLAWB	SMLAWB 命令を挿入します
__SMLAWT	SMLAWT 命令を挿入します
__SMLSD	SMLSD 命令を挿入します
__SMLSDX	SMLSDX 命令を挿入します
__SMLSLD	SMLSLD 命令を挿入します
__SMLSLDX	SMLSLDX 命令を挿入します
__SMMLA	SMMLA 命令を挿入します
__SMMLAR	SMMLAR 命令を挿入します
__SMMLS	SMMLS 命令を挿入します
__SMMLSR	SMMLSR 命令を挿入します

表 32: 組み込み関数の一覧 (続き)

組込み関数	説明
__SMMUL	SMMUL 命令を挿入します
__SMMULR	SMMULR 命令を挿入します
__SMUAD	SMUAD 命令を挿入します
__SMUADX	SMUADX 命令を挿入します
__SMUL	符号付き 16 ビット乗算を挿入します
__SMULBB	SMULBB 命令を挿入します
__SMULBT	SMULBT 命令を挿入します
__SMULTB	SMULTB 命令を挿入します
__SMULTT	SMULTT 命令を挿入します
__SMULWB	SMULWB 命令を挿入します
__SMULWT	SMULWT 命令を挿入します
__SMUSD	SMUSD 命令を挿入します
__SMUSDX	SMUSDX 命令を挿入します
__SSAT	SSAT 命令を挿入します
__SSAT16	SSAT16 命令を挿入します
__SSAX	SSAX 命令を挿入します
__SSUB8	SSUB8 命令を挿入します
__SSUB16	SSUB16 命令を挿入します
__STC	コプロセッサストア命令 STC を挿入します
__STCL	コプロセッサストア命令 STCL を挿入します
__STC2	コプロセッサストア命令 STC2 を挿入します
__STC2L	コプロセッサストア命令 STC2L を挿入します
__STC_noidx	コプロセッサストア命令 STC を挿入します
__STCL_noidx	コプロセッサストア命令 STCL を挿入します
__STC2_noidx	コプロセッサストア命令 STC2 を挿入します
__STC2L_noidx	コプロセッサストア命令 STC2L を挿入します
__STREX	STREX 命令を挿入します
__SWP	SWP 命令を挿入します
__SWPB	SWPB 命令を挿入します
__SXTAB	SXTAB 命令を挿入します
__SXTAB16	SXTAB16 命令を挿入します
__SXTAH	SXTAH 命令を挿入します

表 32: 組込み関数の一覧 (続き)

組込み関数	説明
__SXTB16	SXTB16 命令を挿入します
__UADD8	UADD8 命令を挿入します
__UADD16	UADD16 命令を挿入します
__UASX	UASX 命令を挿入します
__UHADD8	UHADD8 命令を挿入します
__UHADD16	UHADD16 命令を挿入します
__UHASX	UHASX 命令を挿入します
__UHSAX	UHSAX 命令を挿入します
__UHSUB8	UHSUB8 命令を挿入します
__UHSUB16	UHSUB16 命令を挿入します
__UMAAL	UMAAL 命令を挿入します
__UQADD8	UQADD8 命令を挿入します
__UQADD16	UQADD16 命令を挿入します
__UQASX	UQASX 命令を挿入します
__UQSAX	UQSAX 命令を挿入します
__UQSUB8	UQSUB8 命令を挿入します
__UQSUB16	UQSUB16 命令を挿入します
__USAD8	USAD8 命令を挿入します
__USADA8	USAD8A8 命令を挿入します
__USAT	USAT 命令を挿入します
__USAT16	USAT16 命令を挿入します
__USAX	USAX 命令を挿入します
__USUB8	USUB8 命令を挿入します
__USUB16	USUB16 命令を挿入します
__UXTAB	UXTAB 命令を挿入します
__UXTAB16	UXTAB16 命令を挿入します
__UXTAH	UXTAH 命令を挿入します
__UXTB16	UXTB16 命令を挿入します

表 32: 組込み関数の一覧 (続き)

NEON 命令の組込み関数

ARM アーキテクチャで定義された Neon コプロセッサは、Advanced SIMD 命令セット拡張を実装します。アプリケーションで Neon 組込み関数を使用するには、ヘッダファイル `arm_neon.h` をインクルードします。この関数は、以下のパターンに従って名付けられたベクタ型を使用します。

```
<type><size>x<number_of_lanes>_t
```

説明：

- *type* は int、unsigned int、float、poly
- *size* は 8、16、32、64
- *number_of_lanes* は 1、2、4、8、16

ベクタ型の合計ビット幅は *size* に *number_of_lanes* を掛けた値で、D レジスタ（64 ビット）または Q レジスタ（128 ビット）に収まらなければなりません。

次に例を示します。

```
__intrinsic float32x2_t vsub_f32(float32x2_t, float32x2_t);
```

組込み関数 `vsub_f32` は、2 つの 64 ビットベクタ（D レジスタ）上で動作する `VSUB.F32` 命令を挿入します。それぞれに、32 ビット浮動小数点型の 2 つのエレメント（レーン）があります。

一部の関数はベクタ型の配列を使用します。たとえば、`float32x2_t` 型の 4 つのエレメントを持つ配列型の定義は以下のようになります。

```
typedef struct
{
    float32x2_t val[4];
}
float32x2x4_t;
```

組込み関数の詳細

ここでは、各組込み関数のリファレンス情報を説明します。

__CLZ

構文

```
unsigned char __CLZ(unsigned long);
```

説明

CLZ 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v5 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v6T2 またはそれ以降が必要です。

__disable_fiq

構文

```
void __disable_fiq(void);
```

説明

高速割込み要求 (fiq) を無効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__disable_interrupt

構文

```
void __disable_interrupt(void);
```

説明

割込みを禁止します。Cortex-M デバイスに対しては、プライオリティマスクビット PRIMASK を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。

__disable_irq

構文

```
void __disable_irq(void);
```

説明

割込み要求 (irq) を無効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__DMB

構文

```
void __DMB(void);
```

説明

DMB 命令を挿入します。この組込み関数には、ARM v6M アーキテクチャか ARM v7 アーキテクチャまたはそれ以上が必要です。

__DSB

構文

```
void __DSB(void);
```

説明

DSB 命令を挿入します。この組込み関数には、ARM v6M アーキテクチャか ARM v7 アーキテクチャまたはそれ以上が必要です。

__enable_fiq

構文

```
void __enable_fiq(void);
```

説明

高速割り込み要求 (fiq) を有効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__enable_interrupt

構文

```
void __enable_interrupt(void);
```

説明

割り込みを有効にします。Cortex-M デバイスに対しては、プライオリティマスクビット PRIMASK をクリアすることにより、実行優先順位レベルをデフォルトに戻し、他のデバイスに対しては、割り込み要求 (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。

__enable_irq

構文

```
void __enable_irq(void);
```

説明

割り込み要求 (irq) を有効にします。

この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスには使用できません。

__get_BASEPRI

構文

```
unsigned long __get_BASEPRI(void);
```

説明

BASEPRI レジスタの値を返します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3 か Cortex-M4 デバイスを使用する必要があります。

__get_CONTROL

構文

```
unsigned long __get_CONTROL(void);
```

説明

CONTROL レジスタの値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__get_CPSR

構文

```
unsigned long __get_CPSR(void);
```

説明

ARM CPSR（現在のプログラムステータスレジスタ）の値を返します。この組込み関数は、特権モードでのみ使用でき、Cortex-M デバイスでは使用できません。また、ARM モードでなければなりません。

__get_FAULTMASK

構文

```
unsigned long __get_FAULTMASK(void);
```

説明

FAULTMASK レジスタの値を返します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3 か Cortex-M4 デバイスを使用する必要があります。

__get_interrupt_state

構文

```
__istate_t __get_interrupt_state(void);
```

説明

グローバル割込み状態を返します。リターン値は、__set_interrupt_state 組込み関数の引数として使用して、割込み状態を復元することができます。

この組込み関数は、特権モードでのみ使用でき、--aeabi コンパイラオプションを使用している場合は使用できません。

例

```
#include "intrinsics.h"
```

```
void CriticalFn()
{
    __istate_t s = __get_interrupt_state();
    __disable_interrupt();

    /* 何らかの処理 */

    __set_interrupt_state(s);
}
```

__disable_interrupt および __enable_interrupt を使用する場合と比べ、このコードシーケンスを使用する利点は、この例の場合では、__get_interrupt_state の呼出し前に無効化された割込みを有効化することがないことです。

__get_IPSR

構文

```
unsigned long __get_IPSR(void);
```

説明

IPSR レジスタ（割り込みプログラムステータスレジスタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_LR

構文

```
unsigned long __get_LR(void);
```

説明

レジスタ（R14）の値を返します。

__get_MSP

構文

```
unsigned long __get_MSP(void);
```

説明

MSP レジスタ（メインスタックポインタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_PRIMASK

構文

```
unsigned long __get_PRIMASK(void);
```

説明

PRIMASK レジスタの値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__get_PSP

構文

```
unsigned long __get_PSP(void);
```

説明

PSP レジスタ（プロセススタックポインタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_PSR

構文	<code>unsigned long __get_PSR(void);</code>
説明	PSR レジスタ（組み合わせたプログラムステータスレジスタ）の値を返します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__get_SB

構文	<code>unsigned long __get_SB(void);</code>
説明	静的ベースレジスタ (R9) の値を返します。

__get_SP

構文	<code>unsigned long __get_SP(void);</code>
説明	スタックポインタレジスタ (R13) の値を返します。

__ISB

構文	<code>void __ISB(void);</code>
説明	ISB 命令を挿入します。この組込み関数には、ARM v6M アーキテクチャか ARM v7 アーキテクチャまたはそれ以上が必要です。

**__LDC
__LDCL
__LDC2
__LDC2L**

構文	<code>void __nnn(__ul coproc, __ul CRn, __ul const *src);</code> <i>nnn</i> は LDC、LDCL、LDC2、LDC2L のいずれかです。
----	---

パラメータ	
<i>coproc</i>	コプロセッサ番号 0..15。
<i>CRn</i>	ロードするコプロセッサレジスタです。
<i>src</i>	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。

__LDC_noidx
__LDCL_noidx
__LDC2_noidx
__LDC2L_noidx

構文

```
void __nnn_noidx(__ul coproc, __ul CRn, __ul const *src, __ul option);
```

nnn は LDC、LDCL、LDC2、LDC2L のいずれかです。

パラメータ

<i>coproc</i>	コプロセッサ番号 0..15。
<i>CRn</i>	ロードするコプロセッサレジスタです。
<i>src</i>	ロードするデータへのポインタ。
<i>option</i>	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。

__LDREX

構文

```
unsigned long __LDREX(unsigned long *);
```

説明

LDREX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v6T2 またはそれ以降が必要です。

__MCR

構文

```
void __MCR(__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) を挿入します。値は、コプロセッサレジスタに書き込まれます。パラメータ coproc、opcode_1、CRn、CRm、opcode_2 は、MCR 命令処理コードにエンコードされるので、定数でなければなりません。

この組込み関数では ARM モードか、Thum モードの場合は ARM v6T2 またはそれ以降が必要です。

__MRC

構文

```
unsigned long __MRC(__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) を挿入します。指定されたコプロセッサレジスタの値を返します。パラメータ coproc、opcode_1、CRn、CRm、opcode_2 は、MRC 命令処理コードにエンコードされるので、定数でなければなりません。

この組込み関数では ARM モードか、Thum モードの場合は ARM v6T2 またはそれ以降が必要です。

__no_operation

構文 `void __no_operation(void);`

説明 NOP 命令を挿入します。

__PKHBT

構文 `unsigned long __PKHBT(unsigned long x, unsigned long y, unsigned long count);`

パラメータ

<i>x</i>	最初のオペランド。
<i>y</i>	2 番目のオペランド。オプションで左にシフト
<i>count</i>	シフトカウント 0-31。0 はシフトなし。

説明 カウント用に 1 ～ 31 の範囲でオプションのシフトオペランド (LSL) とともに、PKHBT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__PKHTB

構文 `unsigned long __PKHTB(unsigned long x, unsigned long y, unsigned long count);`

パラメータ

<i>x</i>	最初のオペランド。
<i>y</i>	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)。
<i>count</i>	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。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QADD

構文

```
signed long __QADD(signed long, signed long);
```

説明

QADD 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QADD8

構文

```
unsigned long __QADD8(unsigned long, unsigned long);
```

説明

QADD8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QADD16

構文

```
unsigned long __QADD16(unsigned long, unsigned long);
```

説明

QADD16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QASX

構文

```
unsigned long __QASX(unsigned long, unsigned long);
```

説明

QASX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QCFlag

構文

```
unsigned long __QCFlag(void);
```

説明

FPSCR レジスタ（浮動小数点ステータスおよび制御レジスタ）の累計飽和フラグ `QC` の値を返します。この組込み関数は、Neon (Advanced SIMD) を持つデバイスでのみ使用できます。

__QDADD

構文

```
signed long __QDADD(signed long, signed long);
```

説明

`QDADD` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QDOUBLE

構文

```
signed long __QDOUBLE(signed long);
```

説明

ソースレジスタ `Rs` および宛先レジスタ `Rd` に対し、命令 `QDADD Rd, Rs, Rs` を挿入します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QDSUB

構文

```
signed long __QDSUB(signed long, signed long);
```

説明

`QDSUB` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QFlag

構文

```
int __QFlag(void);
```

説明

オーバフロー / 飽和が発生しているかどうかを示す Q フラグを返します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QSAX

構文

```
unsigned long __QSAX(unsigned long, unsigned long);
```

説明

QSAX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QSUB

構文

```
signed long __QSUB(signed long, signed long);
```

説明

QSUB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QSUB8

構文

```
unsigned long __QSUB8(unsigned long, unsigned long);
```

説明

QSUB8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__QSUB16

構文 `unsigned long __QSUB16(unsigned long, unsigned long);`

説明 QSUB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__reset_Q_flag

構文 `void __reset_Q_flag(void);`

説明 オーバフロー / 飽和が発生しているかどうかを示す Q フラグをクリアします。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__reset_QC_flag

構文 `void __reset_QC_flag(void);`

説明 FPSCR レジスタ（浮動小数点ステータスおよび制御レジスタ）の累計飽和フラグ QC の値をクリアします。この組込み関数は、Neon (Advanced SIMD) を持つデバイスでのみ使用できます。

__REV

構文 `unsigned long __REV(unsigned long);`

説明 REV 命令を挿入します。この組込み関数では、ARM v6 以降のアーキテクチャを使用する必要があります。

__REVSH

構文 `signed long __REVSH(short);`

説明 REVSH 命令を挿入します。この組込み関数では、ARM v6 以降のアーキテクチャを使用する必要があります。

__SADD8

構文 `unsigned long __SADD8(unsigned long, unsigned long);`

説明 SADD8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SADD16

構文 `unsigned long __SADD16(unsigned long, unsigned long);`

説明 SADD16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SASX

構文 `unsigned long __SASX(unsigned long, unsigned long);`

説明 SASX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SEL

構文 `unsigned long __SEL(unsigned long, unsigned long);`

説明 SEL 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__set_BASEPRI

構文

```
void __set_BASEPRI(unsigned long);
```

説明

BASEPRI レジスタの値を設定します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3 か Cortex-M4 デバイスを使用する必要があります。

__set_CONTROL

構文

```
void __set_CONTROL(unsigned long);
```

説明

CONTROL レジスタの値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__set_CPSR

構文

```
void __set_CPSR(unsigned long);
```

説明

ARM CPSR（現在のプログラムステータスレジスタ）の値を設定します。制御フィールドのみが変わります（ビット 0 ～ 7）です。この組込み関数は、特権モードでのみ使用でき、Cortex-M デバイスでは使用できません。また、ARM モードでなければなりません。

__set_FAULTMASK

構文

```
void __set_FAULTMASK(unsigned long);
```

説明

FAULTMASK レジスタの値を設定します。この組込み関数は特権モードでのみ使用できます。また、Cortex-M3 か Cortex-M4 デバイスを使用する必要があります。

__set_interrupt_state

構文

```
void __set_interrupt_state(__istate_t);
```

説明

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。

__istate_t 型については、342 ページの __get_interrupt_state を参照してください。

__set_LR

構文

```
void __set_LR(unsigned long);
```

説明

リンクレジスタ (R14) に新しいアドレスを割り当てます。

__set_MSP

構文

```
void __set_MSP(unsigned long);
```

説明

MSP レジスタ（メインスタックポインタ）の値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__set_PRIMASK

構文

```
void __set_PRIMASK(unsigned long);
```

説明

PRIMASK レジスタの値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスを使用する必要があります。

__set_PSP

構文

```
void __set_PSP(unsigned long);
```

説明

PSP レジスタ（プロセススタックポインタ）の値を設定します。この組込み関数は、特権モードでのみ使用できます。また、Cortex-M デバイスでのみ利用できます。

__set_SB

構文

```
void __set_SB(unsigned long);
```

説明

スタティックベースレジスタ (R9) に新しいアドレスを割り当てます。

__set_SP

構文

```
void __set_SP(unsigned long);
```

説明

スタックポインタレジスタ (R13) に新しいアドレスを割り当てます。

__SHADD8

構文 `unsigned long __SHADD8(unsigned long, unsigned long);`

説明 SHADD8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SHADD16

構文 `unsigned long __SHADD16(unsigned long, unsigned long);`

説明 SHADD16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SHASX

構文 `unsigned long __SHASX(unsigned long, unsigned long);`

説明 SHASX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SHSAX

構文 `unsigned long __SHSAX(unsigned long, unsigned long);`

説明 SHSAX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SHSUB8

構文 `unsigned long __SHSUB8(unsigned long, unsigned long);`

説明
SHSUB8 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SHSUB16

構文 `unsigned long __SHSUB16(unsigned long, unsigned long);`

説明
SHSUB16 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLABB

構文 `unsigned long __SMLABB(unsigned long, unsigned long, unsigned long);`

説明
SMLABB 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLABT

構文 `unsigned long __SMLABT(unsigned long, unsigned long, unsigned long);`

説明
SMLABT 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLAD

構文 `unsigned long __SMLAD(unsigned long, unsigned long, unsigned long);`

説明 SMLAD 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLADX

構文 `unsigned long __SMLADX(unsigned long, unsigned long, unsigned long);`

説明 SMLADX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLALBB

構文 `unsigned long long __SMLALBB(unsigned long, unsigned long, unsigned long);`

説明 SMLALBB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLALBT

構文 `unsigned long long __SMLALBT(unsigned long, unsigned long, unsigned long);`

説明 SMLALBT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLALD

構文

```
unsigned long long __SMLALD(unsigned long, unsigned long,  
unsigned long);
```

説明

SMLALD 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLALDX

構文

```
unsigned long long __SMLALDX(unsigned long, unsigned long,  
unsigned long);
```

説明

SMLALDX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLALTB

構文

```
unsigned long long __SMLALTB(unsigned long, unsigned long,  
unsigned long);
```

説明

SMLALTB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLALTT

構文

```
unsigned long long __SMLALTT(unsigned long, unsigned long,  
unsigned long);
```

説明

SMLALTT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLATB

構文 `unsigned long __SMLATB(unsigned long, unsigned long, unsigned long);`

説明
SMLATB 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLATT

構文 `unsigned long __SMLATT(unsigned long, unsigned long, unsigned long);`

説明
SMLATT 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLAWB

構文 `unsigned long __SMLAWB(unsigned long, unsigned long, unsigned long);`

説明
SMLAWB 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLAWT

構文 `unsigned long __SMLAWT(unsigned long, unsigned long, unsigned long);`

説明
SMLAWT 命令を挿入します。
この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLSD

構文	<code>unsigned long __SMLSD(unsigned long, unsigned long, unsigned long);</code>
説明	SMLSD 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLSDX

構文	<code>unsigned long __SMLSDX(unsigned long, unsigned long, unsigned long);</code>
説明	SMLSDX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLS�D

構文	<code>unsigned long long __SMLS�D(unsigned long, unsigned long, unsigned long);</code>
説明	SMLS�D 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMLS�DX

構文	<code>unsigned long long __SMLS�DX(unsigned long, unsigned long, unsigned long);</code>
説明	SMLS�DX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMMLA

構文 `unsigned long __SMMLA(unsigned long, unsigned long, unsigned long);`

説明 `SMMLA` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMMLAR

構文 `unsigned long __SMMLAR(unsigned long, unsigned long, unsigned long);`

説明 `SMMLAR` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMMLS

構文 `unsigned long __SMMLS(unsigned long, unsigned long, unsigned long);`

説明 `SMMLS` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMMLSR

構文 `unsigned long __SMMLSR(unsigned long, unsigned long, unsigned long);`

説明 `SMMLSR` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMMUL

構文 `unsigned long __SMMUL(unsigned long, unsigned long);`

説明 `SMMUL` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMMULR

構文 `unsigned long __SMMULR(unsigned long, unsigned long);`

説明 `SMMULR` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMUAD

構文 `unsigned long __SMUAD(unsigned long, unsigned long);`

説明 `SMUAD` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMUADX

構文 `unsigned long __SMUADX(unsigned long, unsigned long);`

説明 `SMUADX` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMUL

構文 `signed long __SMUL(signed short, signed short);`

説明 符号付き 16 ビット乗算を挿入します。

この組込み関数には ARM モードの場合、ARM v5E アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMULBB

構文 `unsigned long __SMULBB(unsigned long, unsigned long);`

説明 SMULBB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMULBT

構文 `unsigned long __SMULBT(unsigned long, unsigned long);`

説明 SMULBT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMULTB

構文 `unsigned long __SMULTB(unsigned long, unsigned long);`

説明 SMULTB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMULTT

構文

```
unsigned long __SMULTT(unsigned long, unsigned long);
```

説明

SMULTT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMULWB

構文

```
unsigned long __SMULWB(unsigned long, unsigned long);
```

説明

SMULWB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMULWT

構文

```
unsigned long __SMULWT(unsigned long, unsigned long);
```

説明

SMULWT 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMUSD

構文

```
unsigned long __SMUSD(unsigned long, unsigned long);
```

説明

SMUSD 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SMUSDX

構文 `unsigned long __SMUSDX(unsigned long, unsigned long);`

説明 `SMUSDX` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SSAT

構文 `unsigned long __SSAT(unsigned long, unsigned long);`

説明 `SSAT` 命令を挿入します。

コンパイラは、可能な場合にはシフト命令をオペランドに組み込みます。たとえば、`__SSAT(x << 3, 11)` は `SSAT Rd, #11, Rn, LSL #3` にコンパイルされ、`x` の値がレジスタ `Rn` に配置されて、`__SSAT` のリターン値はレジスタ `Rd` に配置されます。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R、または ARM v7-M が必要です。

__SSAT16

構文 `unsigned long __SSAT16(unsigned long, unsigned long);`

説明 `SSAT16` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SSAX

構文 `unsigned long __SSAX(unsigned long, unsigned long);`

説明 `SSAX` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SSUB8

構文	<code>unsigned long __SSUB8(unsigned long, unsigned long);</code>
説明	SSUB8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SSUB16

構文	<code>unsigned long __SSUB16(unsigned long, unsigned long);</code>
説明	SSUB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__STC
__STCL
__STC2
__STC2L

構文	<code>void __nnn(__ul coproc, __ul CRn, __ul const *dst);</code>
----	--

nnn は、STC、STCL、STC2、STC2L のいずれかです。

パラメータ	
<i>coproc</i>	コプロセッサ番号 0..15。
<i>CRn</i>	ロードするコプロセッサレジスタです。
<i>dst</i>	目的の対象へのポインタ。

説明	コプロセッサのストア命令 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 <i>coproc</i> と <i>CRn</i> 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。
----	---

__STC_noidx __STCL_noidx __STC2_noidx __STC2L_noidx

構文

```
void __nnn_noidx(__ul coproc, __ul CRn, __ul const *dst, __ul
option);
```

nnn は、STC、STCL、STC2、STC2L のいずれかです。

パラメータ

<i>coproc</i>	コプロセッサ番号 0..15。
<i>CRn</i>	ロードするコプロセッサレジスタです。
<i>dst</i>	目的の対象へのポインタ。
<i>option</i>	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を挿入します。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。

__STREX

構文

```
unsigned long __STREX(unsigned long, unsigned long *);
```

説明

STREX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v6T2 またはそれ以降が必要です。

__SWP

構文

```
unsigned long __SWP(unsigned long, unsigned long *);
```

説明

SWP 命令を挿入します。この組込み関数では、ARM モードでなければなりません。

__SWPB

構文

```
char __SWPB(unsigned char, unsigned char *);
```

説明

SWPB 命令を挿入します。この組込み関数では、ARM モードでなければなりません。

__SXTAB

構文

```
unsigned long __SXTAB(unsigned long, unsigned long);
```

説明

SXTAB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SXTAB16

構文

```
unsigned long __SXTAB16(unsigned long, unsigned long);
```

説明

SXTAB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SXTAH

構文

```
unsigned long __SXTAH(unsigned long, unsigned long);
```

説明

SXTAH 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__SXTB16

構文 `unsigned long __SXTB16(unsigned long);`

説明 `SXTB16` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UADD8

構文 `unsigned long __UADD8(unsigned long, unsigned long);`

説明 `UADD8` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UADD16

構文 `unsigned long __UADD16(unsigned long, unsigned long);`

説明 `UADD16` 命令を挿入します。

__UASX

構文 `unsigned long __UASX(unsigned long, unsigned long);`

説明 `UASX` 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UHADD8

構文

```
unsigned long __UHADD8(unsigned long, unsigned long);
```

説明

UHADD8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UHADD16

構文

```
unsigned long __UHADD16(unsigned long, unsigned long);
```

説明

UHADD16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UHASX

構文

```
unsigned long __UHASX(unsigned long, unsigned long);
```

説明

UHASX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UHSAX

構文

```
unsigned long __UHSAX(unsigned long, unsigned long);
```

説明

UHSAX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UHSUB8

構文 `unsigned long __UHSUB8(unsigned long, unsigned long);`

説明 UHSUB8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UHSUB16

構文 `unsigned long __UHSUB16(unsigned long, unsigned long);`

説明 UHSUB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UMAAL

構文 `unsigned long long __UMAAL(unsigned long, unsigned long, unsigned long, unsigned long);`

説明 UMAAL 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UQADD8

構文 `unsigned long __UQADD8(unsigned long, unsigned long);`

説明 UQADD8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UQADD16

構文

```
unsigned long __UQADD16(unsigned long, unsigned long);
```

説明

UQADD16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UQASX

構文

```
unsigned long __UQASX(unsigned long, unsigned long);
```

説明

UQASX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UQSAX

構文

```
unsigned long __UQSAX(unsigned long, unsigned long);
```

説明

UQSAX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UQSUB8

構文

```
unsigned long __UQSUB8(unsigned long, unsigned long);
```

説明

UQSUB8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UQSUB16

構文 `unsigned long __UQSUB16(unsigned long, unsigned long);`

説明 UQSUB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__USAD8

構文 `unsigned long __USAD8(unsigned long, unsigned long);`

説明 USAD8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__USADA8

構文 `unsigned long __USADAS8(unsigned long, unsigned long);`

説明 USADA8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__USAT

構文 `unsigned long __USAT(unsigned long, unsigned long);`

説明 USAT 命令を挿入します。

コンパイラは、可能な場合にはシフト命令をオペランドに組み込みます。たとえば、`__USAT(x << 3, 11)` は `USAT Rd, #11, Rn, LSL #3` にコンパイルされ、`x` の値がレジスタ `Rn` に配置され、`__USAT` のリターン値はレジスタ `Rd` に配置されます。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R、または ARM v7M が必要です。

__USAT16

構文 `unsigned long __USAT16(unsigned long, unsigned long);`

説明 USAT16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__USAX

構文 `unsigned long __USAX(unsigned long, unsigned long);`

説明 USAX 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__USUB8

構文 `unsigned long __USUB8(unsigned long, unsigned long);`

説明 USUB8 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__USUB16

構文 `unsigned long __USUB16(unsigned long, unsigned long);`

説明 USUB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UXTAB

構文 `unsigned long __UXTAB(unsigned long, unsigned long);`

説明 UXTAB 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UXTAB16

構文 `unsigned long __UXTAB16(unsigned long, unsigned long);`

説明 UXTAB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UXTAH

構文 `unsigned long __UXTAH(unsigned long, unsigned long);`

説明 UXTAH 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

__UXTB16

構文 `unsigned long __UXTB16(unsigned long);`

説明 UXTB16 命令を挿入します。

この組込み関数には ARM モードの場合、ARM v6 アーキテクチャまたはそれ以降、Thumb モードの場合は ARM v7A、ARM v7R または ARM v7E-M が必要です。

プリプロセッサ

この章では、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。

プリプロセッサの概要

IAR C/C++ Compiler 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。詳しくは、378 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 (227 ページの *-D* を参照)。
- プリプロセッサ拡張
多数のプラグマディレクティブなど、各種のプリプロセッサ拡張を利用できます。詳細については、このガイドの「プラグマディレクティブ」の章を参照してください。該当する _Pragma 演算子や、その他のプリプロセッサ関連の拡張については、381 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 (251 ページの *--preprocess* を参照)。

インクルードファイルのパスを指定するには、スラッシュを使用します。

```
#include "mydirectory/myfile"
```

ソースコードでは、スラッシュを使用します。

```
file = fopen("mydirectory/myfile", "rt");
```

バックスラッシュも使用可能です。この場合、インクルードファイルパスでは 1 つ、ソースコード文字列では 2 つ使用してください。

定義済プリプロセッサシンボルの詳細

以下の表に、定義済プリプロセッサシンボルの一覧を示します。

定義済シンボル	説明
__AAPCS__	--aapcs オプションに基づいて設定される整数。AAPCS の基準の規格が選択された呼出し規約 (--aapcs=std) の場合、シンボルは 1 に設定されます。このシンボルは、他の呼出し規約については未定義です。
__AAPCS_VFP__	--aapcs オプションに基づいて設定される整数。AAPCS の派生 VFP が選択された呼出し規約 (--aapcs=vfp) の場合、シンボルは 1 に設定されます。このシンボルは、他の呼出し規約については未定義です。
__ARM_ADVANCED_SIMD__	--cpu オプションに基づいて設定される整数。選択されたプロセッサのアーキテクチャが Advanced SIMD アーキテクチャの拡張の場合、このシンボルは 1 に設定されます。このシンボルは、他のコアについては未定義です。
__ARM_MEDIA__	--cpu オプションに基づいて設定される整数。選択されたプロセッサのアーキテクチャがマルチメディア用の ARMv6 SIMD 拡張である場合、このシンボルは 1 に設定されます。このシンボルは、他のコアについては未定義です。
__ARM_PROFILE_M__	--cpu オプションに基づいて設定される整数。選択されたプロセッサ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。
__ARMVFP__	--fpu オプションを反映する整数で、__ARMVFPV2__、__ARMVFPV3__、または __ARMVFPV4__ に定義されています。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。
__ARMVFP_D16__	--fpu オプションに基づいて設定される整数。選択された FPU が 16 D レジスタのみを持つ VFPv3 または VFPv4 ユニットの 경우、このシンボルは 1 に設定されます。それ以外の場合、シンボルは未定義です。
__ARMVFP_FP16__	--fpu オプションに基づいて設定される整数。選択された FPU が 16 ビットの浮動小数点数のみをサポートする場合、このシンボルは 1 に設定されます。それ以外の場合、シンボルは未定義です。

表 33: 定義済シンボル

定義済シンボル	説明
__ARMVFP_SP__	--fpu オプションに基づいて設定される整数。選択された FPU が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。
__BASE_FILE__	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。380 ページの __FILE__、244 ページの --no_path_in_file_macros も参照してください。
__BUILD_NUMBER__	現在使用中のコンパイラのビルド番号を示す一意の整数です。
__CORE__	使用中のプロセッサアーキテクチャを示す整数です。 --cpu オプションを反映するシンボルで、__ARM4M__、__ARM4TM__、__ARM5__、__ARM5E__、__ARM6__、__ARM6M__、__ARM6SM__、__ARM7M__、__ARM7EM__、__ARM7A__、または __ARM7R__ に対して定義されます。これらのシンボル名は、__CORE__ シンボルの評価に使用できます。
__cplusplus	コンパイラが C++ モードのいずれかで実行する場合に定義される整数です。それ以外の場合には定義されません。定義される場合、その値は 199711L になります。このシンボルを #ifdef で使用し、コンパイラで C++ コードが使用できるかどうかを検出できます。これは、C および C++ のコードで共有するヘッダファイルを作成する場合に特に便利です。*
__CPU_MODE__	選択した CPU モードを反映する整数です。Thumb の場合 1、ARM の場合 2 と定義されます。
__DATE__	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 2010" など) で返されます。*
__DOUBLE__	--double データ型のサイズを示す整数です。 シンボルは 64 に定義されます。
__embedded_cplusplus	コンパイラが Embedded C++ または拡張 Embedded C++ で実行する場合に 1 と定義される整数です。それ以外の場合にはシンボルは定義されません。このシンボルを #ifdef で使用し、コンパイラで C++ コードが使用できるかどうかを検出できます。これは、C および C++ のコードで共有するヘッダファイルを作成する場合に特に便利です。*

表 33: 定義済シンボル (続き)

定義済シンボル	説明
__FILE__	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。379 ページの <code>__BASE_FILE__</code> 、244 ページの <code>-no_path_in_file_macros</code> も参照してください。*
__func__	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 (232 ページの <code>-e</code> を参照)。380 ページの <code>__PRETTY_FUNCTION__</code> も参照してください。*
__FUNCTION__	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 (232 ページの <code>-e</code> を参照)。380 ページの <code>__PRETTY_FUNCTION__</code> も参照してください。
__IAR_SYSTEMS_ICC__	IAR コンパイラプラットフォームを示す整数です。現行値は 8 です。将来のバージョンでは、番号が大きくなる可能性があります。このシンボルを <code>#ifdef</code> 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__	コードが IAR C/C++ Compiler for ARM でコンパイルされる場合に 1 に設定される整数。
__LINE__	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。*
__LITTLE_ENDIAN__	<code>--endian</code> オプションを反映する整数で、バイトオーダー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。
__PRETTY_FUNCTION__	シンボルが使用されている関数の関数名 (パラメータ型、リターン型を含む) を示す文字列です (" <code>void func(char)</code> " など)。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 (232 ページの <code>-e</code> を参照)。380 ページの <code>__func__</code> も参照してください。

表 33: 定義済シンボル (続き)

定義済シンボル	説明
__ROPI__	--ropi コンパイラオプションの使用時に定義される整数 (253 ページの <code>-ropi</code> を参照)。
__RWPI__	--rwpj コンパイラオプションの使用時に定義される整数 (254 ページの <code>-rwpj</code> を参照)。
__STDC__	コンパイラが C 規格に準拠する場合に 1 に設定される整数です。このシンボルを <code>#ifdef</code> 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。*
__STDC_VERSION__	使用中の C 規格のバージョンを示す整数です。シンボルは 199901L に拡張します。ただし、--c89 コンパイラオプションが使用される場合を除きます。この場合は、シンボルは 199409L に拡張されます。このシンボルは、EC++ モードでは使用できません。*
__TIME__	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。*
__VER__	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。

表 33: 定義済シンボル (続き)

* このシンボルは、C 規格で必須です。

その他のプリプロセッサ拡張

ここでは、定義済シンボル、プリAGMAディレクティブ、C 規格ディレクティブ以外に利用可能なプリプロセッサ拡張について説明します。

NDEBUG

説明	<p>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。</p> <p>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評価されます。このシンボルが定義していない場合は、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。</p> <ul style="list-style-type: none">● このシンボルを定義する場合、アサートコードが含まれない● このシンボルを定義しない場合、アサートコードが含まれる
----	---

したがって、アサートコードを記述し、アプリケーションをビルドする場合、このシンボルを定義することで、アサートコードを最終的なアプリケーションから除外できます。

`assert.h` 標準インクルードファイルでは、アサートマクロは定義されていません。

関連項目

119 ページの *assert* 関数。



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。

#warning message

構文

```
#warning message
```

`message` には任意の文字列を指定できます。

説明

このプリプロセッサディレクティブは、メッセージを生成する場合に使用します。このディレクティブは、主にアサーションやその他のトレースユーティリティに便利です。C 規格の `#error` ディレクティブの使用方法とよく似ています。このディレクティブは、`--strict` コンパイラオプションの使用時は認識されません。

ライブラリ関数

この章では、C/C++ のライブラリ関数の概要を説明します。また、ライブラリ定義にアクセスするためのヘッダファイルも説明しています。

ライブラリ関数の詳細については、オンラインヘルプシステムを参照してください。

ライブラリの概要

コンパイラには、IAR DLIB ライブラリが用意されています。これは C/C++ 規格に準拠した包括的ライブラリです。このライブラリは、IEEE 754 フォーマットの浮動小数点数もサポートしています。また、ロケール、ファイル記述子、マルチバイト文字などのさまざまなレベルのサポートを指定して構成することができます。

カスタマイズの詳細は、「*DLIB ランタイムライブラリ*」を参照してください。

ライブラリ関数の詳細については、製品に付属のオンラインドキュメントを参照してください。また、DLIB ライブラリ関数のキーワードのリファレンス情報も提供されています。関数のリファレンス情報を確認するには、エディタウィンドウで関数名を選択し、F1 キーを押します。

ライブラリ関数の詳細については、本ガイドの*処理系定義の動作*を参照してください。

ヘッダファイル

アプリケーションプログラムは、ヘッダファイルを通じてライブラリ定義にアクセスします。ヘッダファイルは、`#include` ディレクティブを使用して組み込みます。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、90 ページの *ランタイムライブラリの設定* 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。

高精度な代替ライブラリ関数

`cos`、`sin`、`tan`、`pow` のデフォルトの実装は、高速かつ小さくなるように設計されています。もうひとつの方法として、より高い精度を提供するように考えられたバージョンがあります。これらは `__iar_xxx_accuratef`（関数の `float` 派生形）と `__iar_xxx_accuratel`（関数 `long double` の派生形）という名称で、`xxx` は `is cos` や `sin` などです。

より正確な以下のバージョンを使用するには、`--redirect` リンカオプションを使用します。

リエントラント性

関数をメインのアプリケーションや任意の数の割込みで同時に呼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 ライブラリの大部分はリエントラントですが、以下の関数や部分は静的データを必要とするためリエントラントではありません。

- ヒープ関数：`malloc`、`free`、`realloc`、`calloc`、および C++ 演算子 `new`、`delete`
- ロケール関数：`localeconv`、`setlocale`
- マルチバイト関数：`mbrlen`、`mbrtowc`、`mbsrtowc`、`mbtowc`、`wcrtomb`、`wcsrtomb`、`wctomb`
- ランド関数：`rand`、`srand`
- 時間関数：`asctime`、`localtime`、`gmtime`、`mktime`
- その他の関数 `atexit`、`strerror`、`strtok`
- ファイルやヒープを何らかの方法で使用するすべての関数：これには、`printf`、`sprintf`、`scanf`、`sscanf`、`getchar`、`putchar` などが含まれます

`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込みサービスルーチンで使用しない
- 非リエントラント関数の呼出しをミューテックス、保護エリアなどを利用して保護する

LONGJMP 関数



`longjmp` は、実質的に以前に定義された `setjmp` へのジャンプです。スタックの巻き戻し中にスタック上にある可変長配列や C++ オブジェクトは、どれも破壊されません。これは、リソースのリークや不正なアプリケーション動作の原因となることがあります。

IAR DLIB ライブラリ

IAR DLIB ライブラリは、組込みシステムに利用される最も重要な C/C++ ライブラリ定義を提供します。提供される定義は、以下のとおりです。

- C 規格のフリースタANDING実装への準拠。ライブラリはホストされた機能の大半をサポートしますが、基本機能の一部は実装する必要があります。詳細については、本ガイドの「[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* ランタイムライブラリを参照してください
- 低レベルの浮動小数点数ルーチンなどのランタイムサポートライブラリ
- 低レベルの ARM 機能を利用するための組込み関数。詳細については、「[組込み関数](#)」を参照してください

また、IAR DLIB ライブラリには C の追加機能も含まれています。389 ページの *C の追加機能* を参照してください。

C ヘッダファイル

ここでは、DLIB ライブラリの C 定義専用のヘッダファイルについて説明します。ヘッダファイルには、ターゲット固有の定義が追加されている場合があります。これらについては、「[C の使用](#)」で説明しています。

次の表は、C ヘッダファイルの一覧を示します。

ヘッダファイル	用途
assert.h	関数実行時のアサーション実行
表 34: 従来の標準 C ヘッダファイル — DLIB	
complex.h	一般的かつ複雑な数学関数の計算
ctype.h	文字の分類
errno.h	ライブラリ関数が出力したエラーコードの評価
fenv.h	浮動小数点例外フラグ
float.h	浮動小数点数型プロパティの評価
inttypes.h	stdint.h で定義されたあらゆるタイプのフォーマットを定義
iso646.h	Amendment 1 の iso646.h 標準ヘッダ
limits.h	整数型プロパティの評価
locale.h	さまざまな文化圏の慣習への対応
math.h	一般的な数学関数の計算
setjmp.h	非ローカルの goto 文の実行
signal.h	さまざまな例外条件の制御
stdarg.h	可変引数のアクセス
stdbool.h	C の bool 型のサポートを追加
stddef.h	さまざまな有用な型やマクロを定義
stdint.h	整数特性を提供
stdio.h	I/O の実行
stdlib.h	さまざまな処理の実行
string.h	さまざまな種類の文字列の操作
tgmath.h	汎用型の数学関数
time.h	さまざまな時刻 / 日付フォーマットの変換
uchar.h	Unicode 機能 (C 規格に対する IAR 拡張)
wchar.h	ワイド文字のサポート
wctype.h	ワイド文字の分類

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++ および Embedded C++

次の表は、C++ および Embedded C++ で使用可能なヘッダファイルの一覧を示します。

ヘッダファイル	用途
complex	複素数演算をサポートするクラスを定義
exception	例外処理を制御するいくつかの関数を定義（C++ でのみ使用可能）
fstream	外部ファイルを操作する複数の I/O ストリームクラスを定義
omanip	引数を 1 つ指定する複数の I/O ストリームマニピュレータを宣言
ios	多くの I/O ストリームクラスとして機能するクラスを定義
iosfwd	I/O ストリームクラスの定義が必要となる前に複数の I/O ストリームクラスを宣言
iostream	標準ストリームを操作する I/O ストリームオブジェクトを宣言
istream	抽出を実行するクラスを定義
limits	数値を定義（C++ でのみ使用可能）
locale	さまざまな文化圏の慣習への対応（C++ でのみ使用可能）
new	記憶領域の割当て / 解放を行う複数の関数を宣言
ostream	挿入を実行するクラスを定義
sstream	文字列コンテナを操作する複数の I/O ストリームクラスを定義
stdexcept	例外のレポートに役立ついくつかのクラスを定義（C++ でのみ使用可能）
streambuf	I/O ストリーム処理のバッファ処理を行うクラスを定義
string	文字列コンテナを実装するクラスを定義
strstream	メモリ内の文字列シーケンスを操作する複数の I/O ストリームクラスを定義
typeinfo	型情報のサポートを定義（C++ でのみ使用可能）

表 35: Embedded C++ ヘッダファイル

C++ 標準テンプレートライブラリ

次の表は、C++ および拡張 Embedded C++ で使用可能な標準テンプレートライブラリ (STL) のヘッダファイルの一覧を示します。

ヘッダファイル	説明
algorithm	シーケンスに対する一般的な処理を複数定義
bitset	固定サイズのビットシーケンスによりコンテナを定義（C++ でのみ使用可能）

表 36: 標準テンプレートライブラリヘッダファイル

ヘッダファイル	説明
deque	デキューシーケンスコンテナ
functional	複数の関数オブジェクトを定義
hash_map	ハッシュアルゴリズムに基づく map 連想コンテナ
hash_set	ハッシュアルゴリズムに基づく set 連想コンテナ
iterator	共通のイテレータと、イテレータに対する処理を定義
list	双方向リンクリストシーケンスコンテナ
map	map 連想コンテナ
memory	メモリ管理機能定義
numeric	シーケンスに対する一般的な数値操作
queue	キューシーケンスコンテナ
set	set 連想コンテナ
slist	一方向リンクリストシーケンスコンテナ
スタック	スタックシーケンスコンテナ
utility	複数のユーティリティコンポーネントを定義
valarray	可変長配列のコンテナを定義 (C++ でのみ使用可能)
vector	ベクタシーケンスコンテナ

表 36: 標準テンプレートライブラリヘッダファイル (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。

以下の表に、新フォーマットのヘッダファイルを示します。

ヘッダファイル	用途
<code>cassert</code>	関数実行時のアサーション実行
<code>ccomplex</code>	一般的かつ複雑な数学関数の計算
<code>cctype</code>	文字の分類
<code>cerrno</code>	ライブラリ関数が出力したエラーコードの評価
<code>cfenv</code>	浮動小数点例外フラグ
<code>cfloat</code>	浮動小数点数型プロパティの評価
<code>cinttypes</code>	<code>stdint.h</code> で定義されたあらゆるタイプのフォーマットを定義
<code>ciso646</code>	Amendment 1 の <code>iso646.h</code> 標準ヘッダ

表 37: 新しい標準 C ヘッダファイル — DLIB

ヘッダファイル	用途
climits	整数型プロパティの評価
locale	さまざまな文化圏の慣習への対応
cmath	一般的な数学関数の計算
csetjmp	非ローカルの goto 文の実行
csignal	さまざまな例外条件の制御
cstdarg	可変引数のアクセス
cstdbool	C の bool データ型のサポートを追加
cstddef	さまざまな有用な型やマクロを定義
cstdint	整数特性を提供
cstdio	I/O の実行
cstdlib	さまざまな処理の実行
cstring	さまざまな種類の文字列の操作
ctgmath	汎用型の数学関数
ctime	さまざまな時刻 / 日付フォーマットの変換
cwchar	ワイド文字のサポート
cwctype	ワイド文字の分類

表 37: 新しい標準 C ヘッダファイル — DLIB (続き)

組込み関数としてのライブラリ関数

ある状況下において、特定の C ライブラリ関数は組込み関数として扱われ、通常の間数呼出しの代わりに memcpy、memset、strcat などのインラインコードを生成します。

C の追加機能

IAR DLIB ライブラリには、追加された C 機能がいくつか含まれています。これらの機能は、以下のインクルードファイルによって提供されます。

- fenv.h
- stdio.h
- stdlib.h
- string.h
- time.h

fcntl.h

fcntl.h では、浮動小数点の数値のトラップ処理のサポートが、関数 fegettrapenable および fegettrapdisable によって定義されています。

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 など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、システムヘッダファイル `time.h` で定義されます。

アプリケーションはどちらのインタフェースも使用でき、32 ビットまたは 64 ビットの派生形を明示的に使用して両方を混在させることも可能です。デフォルトでは、ライブラリとヘッダは `time_t` や `time` などを 32 ビットの派生形にリダイレクトします。ただし、これらを明示的に 64 ビットの派生形にリダイレクトするには、`time.h` または `ctime` のインクルードの前に `_DLIB_TIME_USES_64` を定義します。

118 ページの *時間関数* も参照してください。

ライブラリにより内部的に使用されるシンボル

以下のシンボルはライブラリで使用されます。つまり、ライブラリのソースファイルなどで可視ということです。

`__assignment_by_bitwise_copy_allowed`

このシンボルは、クラスオブジェクトのプロパティを決定します。

`__code, __data`

これらのシンボルは、コンパイラでメモリ属性として内部的に使用されます。特定のテンプレートでは引数として使用しなければならないこともあります。

`__constrange()`

組込み関数へのパラメータの有効範囲と、パラメータの型が `const` でなければならないことを決定します。

`__construction_by_bitwise_copy_allowed`

このシンボルは、クラスオブジェクトのプロパティを決定します。

`__has_constructor, __has_destructor`

これらのシンボルはクラスオブジェクトのプロパティを決定し、`sizeof` 演算子と同様に機能します。クラス、基底クラス、メンバ（再帰的）にユーザ定義のコンストラクタまたはデストラクタがそれぞれある場合、シンボルは真となります。

`__memory_of`

クラスメモリを決定します。クラスメモリによって、クラスメモリが存在できるメモリが決まります。このシンボルは、クラスメモリとしてクラス定義の中でのみ発生できます。

注：これらのシンボルは予約済みであり、ライブラリでのみ使用してください。

定義済みのシンボルの値を判断するには、コンパイラオプションの `--predef_macros` を使用します。

リンカ設定ファイル

この章では、リンカ設定ファイルの目的およびその内容について説明します。

この章を読む前に、**セクション**の概念について理解しておいてください。詳細については、68 ページの **モジュールおよびセクション**を参照してください。

概要

要件に合わせてメモリのアプリケーションをリンクおよび検出するため、ILINK には、セクションを扱う方法、および使用できるメモリエリアにセクションを配置する方法に関する情報が必要です。つまり、ILINK には、**リンカ設定ファイル**により渡される**設定**が必要です。

このファイルは、一連のディレクティブで構成され、通常、以下のことを行います。

- 使用できるアドレス可能メモリを定義する
可能なアドレスの最大サイズに関するリンカ情報を提供し、使用可能な物理メモリを定義します。また、さまざまな方法でのアドレスが可能なメモリを扱います。
- ROM または RAM の使用可能メモリエリアを定義する
各領域の開始および終了アドレスを提供します。
- セクショングループ
セクション要件に従ってセクションをブロックまたはオーバーレイにグループ化する方法を扱います。
- アプリケーションの初期化を扱う方法を定義する
初期化されるセクションに関する情報、およびその初期化の方法に関する情報を提供します。
- メモリ割当て
セクションの各セットが配置されるメモリエリアを定義します。
- シンボル、式、数値の使用
アドレスやサイズなどを他の設定ディレクティブで表現します。シンボルは、アプリケーション自体でも使用できます。

- 構造化設定

条件に応じてディレクティブを含める、または除外し、設定ファイルをいくつかの異なるファイルに分割できます。

コメントは、C コメント (`/*...*/`) または C++ コメント (`//...`) のいずれかとして記述できます。

メモリおよび領域の定義

ILINK には、使用可能なメモリ空間に関する情報、具体的には以下の情報が必要です。

- 使用できるアドレス可能メモリの最大サイズ

`define memory` ディレクティブは、指定したサイズでメモリ空間を定義します。これは、アドレス可能メモリの最大サイズで、必ずしも物理的に使用できるサイズではありません。394 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なメモリの領域を定義します。395 ページの *define region* ディレクティブを参照してください。

領域は、1 つ以上のメモリエリアで構成されます。領域は、メモリ内での連続するバイトで、領域式を使用して複数の領域を表現できます。

397 ページの *領域式* を参照してください。

define memory ディレクティブ

構文

```
define memory [ name ] with size = size_expr [ , unit-size ] ;
```

ここで、*unit-size* は以下のいずれかです。

```
unitbitsize = bitsize_expr
```

```
unitbytesize = bytesize_expr
```

また、*expr* は式です（詳細は 413 ページの *式* を参照）。

パラメータ

size_expr

メモリ空間に含まれるユニットの量を指定。これは常にアドレスゼロからカウントされます。

bitsize_expr

各ユニットに含まれるビット数を指定します。

bytesize_expr

各ユニットに含まれるバイト数を指定します。各バイトには 8 ビットが含まれます。

領域リテラル

構文

```
[ memory-name: ][from expr { to expr | size expr }  
  
[ repeat expr [ displacement expr ] ]]
```

ここで、*expr* は式です（詳細は 413 ページの 式を参照）。

パラメータ

<i>memory-name</i>	領域リテラルが配置されるメモリ空間の名前。メモリが 1 つだけの場合、名前はオプションです。
<i>from expr</i>	<i>expr</i> は、表示するメモリ範囲の開始アドレス（普 gl）。
<i>to expr</i>	<i>expr</i> は、表示するメモリ範囲の終了アドレス（普 gl）。
<i>size expr</i>	<i>expr</i> は、メモリ範囲のサイズ。
<i>repeat expr</i>	<i>expr</i> は、領域リテラルに同じメモリの複数の範囲を定義します。
<i>displacement expr</i>	<i>expr</i> は、繰り返しシーケンスで前の範囲開始からの移動距離です。デフォルトの移動距離は、範囲サイズと同じ値です。

説明

領域リテラルは、1 つのメモリ範囲で構成されます。範囲を定義する場合、範囲が配置されるメモリ、開始アドレス、サイズを指定する必要があります。範囲サイズは、サイズを指定して明示的に指定するか、範囲の終了アドレスを指定して暗黙的に指定できます。終了アドレスが範囲に含まれ、ゼロサイズ領域にはアドレスのみが含まれます。メモリがどこでラップされるかが認識されているため、範囲がアドレスゼロをスナップしたり、そのような範囲が符号なし値で表現したりできます。

repeat パラメータは、各繰り返しに 1 つずつ、複数の範囲を含む領域リテラルを作成します。これは、バンクまたはフー領域で便利です。

例

```
/* 0 番地中心の 5 バイトの領域 */  
Mem:[from -2 to 2]  
  
/* 64kB のメモリー中で、0 番地中心の 512 バイトの領域 */  
Mem:[from 0xFF00 to 0xFF]  
  
/* 同じメモリ上の、いくつかの繰り返し領域  
   リテラル */  
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]
```

```
/* 次の定義と同じ :
   Mem:[from 0 size 0x100]
   Mem:[from 0x1000 size 0x100]
   Mem:[from 0x2000 size 0x100]
*/
```

関連項目

395 ページの *define region* ディレクティブ、397 ページの *領域式*。

領域式

構文

```
region-operand
| region-expr | region-operand
| region-expr - region-operand
| region-expr & region-operand
```

ここで、*region-operand* は以下のいずれかです。

```
( region-expr )
region-name
region-literal
empty-region
```

region-name は領域です（詳細は 395 ページの *define region* ディレクティブを参照）。

region-literal は領域リテラルです（詳細は 396 ページの *領域リテラル* を参照）。

empty-region は空領域です（詳細は 398 ページの *空の領域* を参照）。

説明

通常、領域は、1 つのメモリ範囲で構成されます。つまり、1 つの *領域リテラル* で領域を表現できます。領域に複数の範囲が（場合によっては異なるメモリに）含まれる場合、*領域式* を使用して領域を表現する必要があります。領域式は、実際、メモリ範囲のセットにおけるセット式です。

領域式を作成するために、3 つの演算子、union (*|*)、intersection (*&*)、difference (*-*) が使用できます。これらの演算子は、*セット理論* に基づいて機能します。たとえば、セット A および B がある場合、演算子の結果は以下のようになります。

- A | B: セット A またはセット B いずれかのすべてのエレメント
- A & B: セット A およびセット B 両方のすべてのエレメント
- A - B: セット A にありセット B にはないすべてのエレメント

例

```
/* 結果は Mem 上の 1000 - 2FFF の範囲になる */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 1500 - 1FFF の範囲になる */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 1000 - 14FF の範囲になる */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 2つの範囲 . 1000 - 1FFF と 2501 - 2FFF.
   になる Mem */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

空の領域

構文

[]

説明

空の領域には、メモリ範囲は含まれません。空の領域が、1つ以上のセクションの配置のために実際に使用される配置ディレクティブで使用される場合、ILINK ではエラーが発生します。

例

```
define region Code = Mem:[from 0 size 0x10000];
if (Banked) {
    define region Bank = Mem:[from 0x8000 size 0x1000];
} else {
    define region Bank = [];
}
define region NonBanked = Code - Bank;

/* シンボル Banked により、NonBanked 領域は 0x10000 バイトの1つの領域
   か 0x8000 バイトと 0x7000 バイトの二つの領域になる。*/
```

関連項目

397 ページの *領域式*。

セクションの取扱い

セクションの取扱いは、ILINK で実行イメージのセクションをどう処理するかについて説明します。これは以下のことを意味します。

- セクションを領域に配置する
place at および place into ディレクティブは、同様の属性をもつセクションのセットを、以前定義された領域に配置します。*406 ページの place at* ディレクティブおよび *407 ページの place in* ディレクティブを参照してください。

- 特殊な要件のセクションのセットを作成する

`block` ディレクティブを使用すると、特殊なサイズおよびアラインメントを持つ、またはタイプの異なるシーケンシャルにソートされたセクションなど、空のセクションを作成できます。

`overlay` ディレクティブを使用すると、複数のオーバー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。399 ページの *define block* ディレクティブ、401 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` および `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。402 ページの *initialize* ディレクティブおよび 404 ページの *do not initialize* ディレクティブを参照してください。

- 削除したセクションを保持する

`keep` ディレクティブを使用すると、アプリケーションの残りの部分から参照されない場合でも、セクションが保持されます。つまり、これは、アセンブラおよびコンパイラにおける *root* の概念と同じです。405 ページの *keep* ディレクティブを参照してください。

define block ディレクティブ

構文

```
define [movable] block name
    [ with param, param... ]
{
    extended-selectors
}
[except
{
    section_selectors
}];
```

ここで、*param* は以下のいずれかです。

```
size = expr
maximum size = expr
alignment = expr
fixed order
static base [name]
static base
```

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 (407 ページの *セクションの選択* を参照)。

パラメータ

<code>name</code>	定義するブロックの名前。
<code>size</code>	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。
<code>maximum size</code>	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。
<code>alignment</code>	ブロックの最小アラインメントを指定します。ブロックの任意のセクションのアラインメントが、最小アラインメントを超える場合、そのアラインメントがブロックのアラインメントになります。
<code>fixed order</code>	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。
<code>static base name</code>	アプリケーション実行時に、指定した名前の静的ベースをブロック開始位置にするよう指定します。
<code>static base</code>	アプリケーション実行時に静的ベースをブロック開始位置にするよう指定します。

説明

`block` 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。ブロックを定義することで、たとえば、スタックまたはヒープとして使用できるバイトの空のブロックを作成できます。また、このディレクティブを使用して、連続する、または連続しない特定のタイプのセクションをグループ化することもできます。さらに、セクションを1つのメモリエリアにまとめて、アプリケーションからそのエリアの開始および終了位置にアクセスすることもできます。

`movable` ブロックは、リードオンリーおよびリード/ライトの位置に依存しない状態で使用します。ブロックを `movable` (移動可能) にすると、リンカでアプリケーションによるアドレスの使用を検証できるようになります。移動可能なブロックは他のブロックとまったく同じように配置しますが、移動可能ブロック内のシンボルを参照する際に適切な再配置が使用されるかどうかをリンカがチェックします。

例

```
/* ヒープ用に 0x1000 バイトのブロックを作る */
define block HEAP with size = 0x1000, alignment = 8 { };
```

関連項目

172 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、401 ページの `define overlay` ディレクティブを参照してください。

define overlay ディレクティブ

```

構文      define overlay name [ with param, param... ]
          {
            extended-selectors;
          }
          [except
            {
              section_selectors
            }
          ];

```

拡張セクタおよび **except** 句については、407 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

<code>name</code>	オーバーレイの名前。
<code>size</code>	オーバーレイのサイズをカスタマイズします。デフォルトでは、オーバーレイのサイズは、そのコンテンツに依存するパーツの合計です。
<code>maximum size</code>	オーバーレイのサイズの上限を指定します。オーバー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。
<code>alignment</code>	オーバーレイの最小アラインメントを指定します。オーバーレイの任意のセクションのアラインメントが、最小アラインメントを超える場合、そのアラインメントがオーバーレイのアラインメントになります。
<code>fixed order</code>	セクションを固定順で配置します。指定されない場合、セクションは任意の順序で配置されます。

説明

`overlay` ディレクティブは、セクションの指定セットを定義します。`block` ディレクティブとは対照的に、`overlay` ディレクティブは、同じ名前を複数回定義できます。各定義は、同じ名前の他のすべての定義と同じメモリ内の場所にまとめることができます。これにより、複数の独立したサブアプリケーションをもつアプリケーションで利用できる、オーバーレイメモリエリアが作成されます。

各サブアプリケーションイメージを ROM に配置し、すべてのサブアプリケーションを保持する RAM オーバー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 オーバーレイにサブアプリケーションをコピーします。ILINK では、オーバー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ーレイ定義の管理に関する支援はないので注意してください。

オーバーレイのサイズは、そのオーバーレイ名で定義されている最大サイズと同じサイズになり、アラインメント要件は、アラインメント要件が最高の定義と同じになります。

注： オーバー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ので、必要なすべてのコピーに注意する必要があります。

関連項目

83 ページの *手動で初期化する* を参照してください。

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 = { none | zeros | packbits | bwt | lzw | auto |  
           smallest }
```

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。407 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ズ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ズ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。

packing イニシャライザを扱う方法を指定します。以下から選択します。

none - 選択したセクションコンテンツの圧縮を無効にします。これは、initialize manually のデフォルト手法です。

zeros - 値ゼロの連続するバイトを圧縮します。

packbits - PackBits アルゴリズムで圧縮します。この手法は、同じ値のバイトが多数連続するデータにおいて好結果が得られます。

bwt - Burrows-Wheeler アルゴリズムで圧縮します。この手法は、データのブロックを圧縮する前に変換することにより、packbits の手法を改善するものです。

lzw - Lempel-Ziv-Welch アルゴリズムで圧縮します。この手法では、辞書を使用してバイトパターンをデータに格納します。

auto - smallest とよく似ていますが、ILINK は none または packbits を選択します。これは、initialize by copy のデフォルト手法です。

smallest - ILINK が各パッキング手法（auto は除く）を使用して結果のサイズを予測し、推定サイズが最小になるパッキング手法を選択します。ここには、デコンプレッサのサイズも含まれます。

説明

initialize ディレクティブは、初期化セクションを、イニシャライザを保持するセクションと初期化データを保持するセクションに分割します。起動時の初期化が自動的に行われるか (initialize by copy)、手動で行うか (initialize manually) を選択できます。

パッキング手法として auto (initialize by copy のデフォルト) または smallest を使用する場合、ILINK はイニシャライザに適したパッキングアルゴリズムを自動的に選択します。これをオーバーライドする場合は、別の packing 手法を選択してください。--log initialization オプションを使用すると、使用するパッケージングアルゴリズムを ILINK がどのように決定するかが示されます。

イニシャライザを圧縮すると、デコンプレッサが自動的にイメージに追加されます。bwt および lzw 用のデコンプレッサは、zeros および packbits と比べ、大幅に実行時間が長く、大規模な RAM も必要とします。bwt では約 9KB のスタックエリアが必要となり、lzw では 3.5KB が必要となります。

イニシャライザを圧縮する場合、圧縮後のイニシャライザの正確なサイズは、未圧縮データの正確な内容がわかるまで確定しません。このデータに他のアドレスが含まれ、これらのアドレスの一部が圧縮後のイニシャライザのサイズに依存する場合には、リンカでエラー Lp017 が発生します。この問題を回避するには、圧縮後のイニシャライザを最後に配置するか、アドレスが既知である必要のないセクションと一緒にメモリ領域に配置します。

initialize manually を使用しない限り、ILINK は初期化テーブルをインクルードすることによってシステム起動時に初期化が行われるようにします。

起動コードは、このテーブルを読み込む初期化ルーチンを呼び出して、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、オーバー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、[401 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な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

[74 ページの システム起動時の初期化](#)、[404 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
}】;
```

`extended selectors` および `except` 句については、[407 ページの セクションの選択](#)を参照してください。

説明	<p>do not initialize ディレクティブは、システム起動コードで初期化しないセクションを指定します。このディレクティブを使用できるのは、zeroinit セクションのみです。</p> <p>コンパイラキーワード <code>__no_init</code> は、do not initialize ディレクティブで扱う必要があるセクションに変数を配置します。</p>
例	<pre>/* 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 };</pre>
関連項目	74 ページの <i>システム起動時の初期化</i> 、402 ページの <i>initialize ディレクティブ</i> 。

keep ディレクティブ

構文	<pre>keep { section-selectors } 【except { section-selectors }】;</pre> <p>extended selectors および except 句については、407 ページの <i>セクションの選択</i> を参照してください。</p>
説明	keep ディレクティブは、選択したセクションが参照されない場合あっても、すべての選択したセクションが実行可能イメージを保持するように指定します。
例	<pre>keep { section .keep* } except {section .keep};</pre>

place at ディレクティブ

構文

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
          end of region_expr }

{
    extended-selectors
}
【except
{
    section-selectors
}】;
```

extended selectors および except 句については、407 ページの *セクションの選択* を参照してください。

パラメータ

memory: expr 特定メモリ内の特定アドレスです。アドレスは、define memory ディレクティブで定義されている供給メモリで使用できなければなりません。メモリが1つだけの場合、メモリ指定子はオプションです。

start of region_expr 領域式。使用する領域の開始位置です。

end of region_expr 領域式。使用する領域の終了位置です。

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、空の領域を place at ディレクティブで使用することもできません。領域に配置される場合、セクションおよびブロックは、place in ディレクティブで同じ領域に配置される他の任意のセクションまたはブロックの前に配置されます。

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。

name が指定されている場合、このディレクティブがマップファイルおよび一部のログメッセージで使用されます。

例

```
/* read-only セクションである .startup を コード領域の最初に配置する */
"START": place at start of ROM { readonly section .startup };
```

関連項目

407 ページの *place in* ディレクティブ。

place in ディレクティブ

構文

```
[ "name": ]
place in region-expr
{
    extended-selectors
}
[except{
    section-selectors
}];
```

ここで、*region-expr* は領域式です。詳細については、395 ページの *領域* を参照してください。

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。407 ページの *セクションの選択* を参照してください。

説明

place in ディレクティブは、セクションおよびブロックを特定のブロックに配置します。セクションおよびブロックは、任意の順序で領域に配置されます。

特定の順序を指定するには、*block* ディレクティブを使用します。領域では、複数の範囲を使用できます。

name が指定されている場合、このディレクティブがマップファイルおよび一部のログメッセージで使用されます。

例

```
/* コード領域に read-only セクションを配置する */
"ROM": place in ROM { readonly };
```

関連項目

406 ページの *place at* ディレクティブ。

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（*セクションセクタ*および *except* 句を使用）。1 つ以上のセクションセ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、この句のセクタのセクションは選択されません。各セクションセクタは、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、*拡張セクタ*が使用されます。

Section-selectors

構文

```
{ [ section-selector ] [, section-selector... ] }
```

ここで、*section-selector* には以下のように指定します。

```
[ section-attribute ] [ section-type ] [ section sectionname ]
    [ object { module | filename } ]
```

ここで、*section-attribute* には以下のように指定します。

```
[ ro [ code | data ] | rw [ code | data ] | zi ]
```

また、ro、rw、zi は、それぞれ readonly、readwrite、zeroinit です。

section-type は以下のように指定します。

```
[ preinit_array | init_array ]
```

パラメータ

ro or readonly	リードオンリーセクション。
rw または readwrite	リード/ライトセクション。
zi または zeroinit	ゼロで初期化されるセクション。これらのセクションには内容はなため、システム起動時にゼロで初期化するのが望ましいと言えます。
コード	コードを含むセクション。
データ	データを含むセクション。
preinit_array	ELF セクションタイプ SHT_PREINIT_ARRAY のセクション。
init_array	ELF セクションタイプ SHT_INIT_ARRAY のセクション。
sectionname	セクション名。以下の 2 つのワイルドカードを使用できます。 ? は、任意の 1 文字と一致します。 * は、ゼロ以上の文字と一致します。
module	objectname(libraryname) 形式の名前。オブジェクト名とライブラリ名がそれぞれのパターンに一致するオブジェクトモジュールのセクションが選択されます。空のライブラリ名パターンでは、オブジェクトファイルのセクションのみが選択されます。
filename	オブジェクトファイル、ライブラリ、ライブラリ内のオブジェクトの名前。以下の 2 つのワイルドカードを使用できます。 ? は、任意の 1 文字と一致します。 * は、ゼロ以上の文字と一致します。

説明

セクションセクタは、*object*（オブジェクトファイル、ライブラリ、ライブラリのオブジェクト）のセクション属性、セクションタイプ、セクション名、名前と一致するすべてのセクションを選択します。4つの条件のうち3つまでを省略できます。セクション属性が省略された場合、セクション属性に関係なく任意のセクションが選択されます。セクションタイプを省略すると、任意のタイプのセクションが選択されます。

セクション名またはオブジェクト名の一部が省略された場合、セクション名やオブジェクト名の制限なしにセクションが選択されます。

セクションセクタを使用せずに { } のみを使用することもできます。これは、ブロックを定義する場合に便利です。

スコープの狭いセクションセクタは、より汎用的なセクションセ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。セクションセクタは以下の場合により具体的となります。

- 他のものと異なり、セクションタイプを指定している
- 他のものと異なり、ワイルドカードを使用せずにセクション名またはオブジェクト名を指定している
- 他のセクタに一致するセクションがこのセクタにも一致しているが、その逆が真でない

セクタ 1	セクタ 2	より具体的なセクタ
セクション "foo*"	セクション "f*"	セクタ 1
セクション "*x"	セクション "f*"	どちらも不適格
ro コードセクション "f*""	ro セクション "f*"	セクタ 1
init_array	ro セクション "xx"	セクタ 1
セクション ".intvec"	ro セクション ".int*"	セクタ 1
セクション ".intvec"	オブジェクト "foo.o"	どちらも不適格

表 38: セクションセクタの指定の例

例

```
{ rw } /* 全ての read-write セクションを選択する */

{ section .mydata* } /* .mydata* セクションを * 選択する */
/* 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

402 ページの *initialize* ディレクティブ、404 ページの *do not initialize* ディレクティブ、405 ページの *keep* ディレクティブ。

拡張セクタ

構文

```
{ [ extended-selector ] [ , extended-selector... ] }

ここで、extended-selector には以下のように指定します。

[ first | last ] { section-selector |
                  block name [ inline-block-def ] |
                  overlay name }
```

ここで、inline-block-def には以下のように指定します。

```
[ block-params ] extended-selectors
```

パラメータ

first	選択した名前を最初に領域、ブロック、オーバーレイに配置します。
last	選択した名前を最後に領域、ブロック、オーバーレイに配置します。
block	ブロックの名前。
overlay	オーバーレイの名前。

説明

section-selector の機能のほかに、extended-selector は、ブロックまたはオーバーレイを最初または最後にセクションのセット、ブロック、オーバーレイに配置する機能を提供します。また、ブロックのインライン定義を作成することもできます。つまり、セクション配置をさらに正確に制御できます。

例	<pre>define block First { section .first }; /* セクション .first を含 むブロックを定義する */ define block Table { first block First }; /* ブロック First が最初に 配置されたブロックを定義 する */</pre> <p>または、ブロック First のインライン定義を使用した場合は以下ようになります。</p> <pre>define block Table { first block First { section .first }};</pre>
関連項目	399 ページの <i>define block</i> ディレクティブ、401 ページの <i>define overlay</i> ディレクティブ、406 ページの <i>place at</i> 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。411 ページの *define symbol* ディレクティブ、412 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。413 ページの *式* を参照してください。

define symbol ディレクティブ

構文	<pre>define [exported] symbol name = expr;</pre>	
パラメータ	<pre>exported</pre>	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。
	<pre>name</pre>	シンボルの名前。
	<pre>expr</pre>	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` プレフィックスの付いたシンボル (x は大文字)、または `__` (下線 2 本) を含むシンボルは、ツールセットベンダの予約語です

例

```
/* シンボル my_symbol を 4 と定義 */  
define symbol my_symbol = 4;
```

関連項目

412 ページの *export* ディレクティブ、85 ページの *ILINK* とアプリケーション間の相互処理。

export ディレクティブ

構文

```
export symbol name;
```

パラメータ

name

シンボルの名前。

説明

`export` ディレクティブは、エクスポートするシンボルを定義し、これを実行可能イメージおよびグローバルラベルから使用できるようにします。アプリケーションまたはデバッガは、これを設定目的などのために参照できます。

例

```
/* シンボル my_symbol をエクスポート */  
export symbol my_symbol;
```

式

構文

式は、以下の要素で構成されます。

```
expression binop expression
unop expression
expression ? expression : expression
(expression)
number
symbol
func-operator
```

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 414 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、411 ページの *define symbol* ディレクティブおよび 263 ページの *--config_def* を参照してください。

func-operator は、以下の関数のような演算子のいずれかです。

<code>minimum(expr, expr)</code>	2つのパラメータの小さい方を返します。
<code>maximum(expr, expr)</code>	2つのパラメータの大きい方を返します。
<code>isempty(r)</code>	領域が空の場合は True、そうでない場合は False を返します。
<code>isdefinedsymbol(expr-symbol)</code>	式シンボルが定義されている場合は True、そうでない場合は False を返します。
<code>start(r)</code>	領域の最下位アドレスを返します。
<code>end(r)</code>	領域の最上位アドレスを返します。
<code>size(r)</code>	完全な領域のサイズを返します。

ここで、*expr* は式で、*r* は領域式です（詳細は 397 ページの *領域式* を参照）。

説明

リンカ設定ファイルでは、式は、範囲 $-2^{64} \sim 2^{64}$ の 65 ビット値です。式の構文は、いくつかの例外がありますが、C 構文に従っています。割当て、キャスト、事前/事後処理、アドレス処理（`*`、`&`、`[]`、`->`、`.`）はありません。領域の式から値を抽出する処理など、いくつかの処理では、関数呼出しに似た構文が使用されます。ブール値式は、0（偽）または 1（真）を返します。

数値

構文

```
nr [nr-suffix]
```

ここで、`nr` は、10 進数または 16 進数（`0x...` または `0X...`）のいずれかです。
また、`nr-suffix` は以下のいずれかです。

```
K      /* Kilo = (1 << 10) 1024 */  
M      /* Mega = (1 << 20) 1048576 */  
G      /* Giga = (1 << 30) 1073741824 */  
T      /* Tera = (1 << 40) 1099511627776 */  
P      /* Peta = (1 << 50) 1125899906842624 */
```

説明

数値は、通常の C 構文で、または便利なサフィックスのセットを使用して表現できます。

例

1024 は、`0x400` と同じです。これは、`1K` と同じです。

構造化構成

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することができます。

- 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。[414 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。[415 ページの `include` ディレクティブ](#)を参照してください。

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（詳細は [413 ページの 式](#)を参照）。

パラメータ

ディレクティブ

任意の ILINK ディレクティブ。

説明

if ディレクティブは、条件に基づいて他のディレクティブを含めるまたは除外します。これにより、同じファイルで、たとえば、バンクおよび非バンクメモリの両方でいくつかの異なるメモリ設定を行うことができます。

if パート、else if パート、else パート内のディレクティブは、条件式の評価が真か偽かに関係なく、構文がチェックされ、処理されます。ただし、条件式が真の場合のパートのディレクティブ、またはいずれの条件も真でない場合の else パートのディレクティブは、if ディレクティブ外には影響を与えません。if ディレクティブはネストできます。

例

398 ページの *空の領域*を参照してください。

include ディレクティブ

構文

```
include filename;
```

パラメータ

filename

/ と ¥ の両方をディレクトリの区切り文字として使用できる文字列リテラル。

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。

セクションリファレンス

コンパイラは、コードおよびデータをセクションに配置します。ILINK は、リンカ設定ファイルで指定された設定に基づき、セクションをメモリに配置します。

この章では、ARM 用 IAR ビルドツールで使用可能な、事前定義されたすべての ELF セクションおよびブロックをリストして、各セクションの詳細なリファレンス情報について説明します。

セクションの詳細については、「68 ページの *モジュールおよびセクション*」の章を参照してください。

セクションの概要

以下の表は、IAR ビルドツールで使用する 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。
.cstart	起動コードを保持します。
.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 <code>initialize by copy</code> の使用時に、 <code>.data</code> セクションの初期値を保持。
__DLIB_PERTHREAD	DLIB モジュールの静的状態を含む変数を保持します。
.exc.text	例外に関連するコードを保持します。
HEAP	動的割当てデータに使用するヒープを保持します。
.iar.dynexit	<code>atexit</code> 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	<code>__no_init</code> 静的 / グローバル変数を保持します。
.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。

表 39: セクションの概要

セクション	説明
.rodata	定数データを保持します。
.text	プログラムコードを保持します。
.texttrw	__ramfunc により宣言されたプログラムコードを保持します。
.texttrw_init	.texttrw で宣言されたセクションのイニシャライザを保持します。

表 39: セクションの概要 (続き)

アプリケーションで使用する ELF セクションのほかに、ツールではさまざまな目的で多数の ELF セクションを使用します。

- .debug で始まるセクションは一般的に、DWARF フォーマットのデバッグ情報を含みます。
- .iar.debug で始まるセクションには、デバッグの補足情報が IAR フォーマットで含まれます。
- セクション .comment は、ファイルのビルドに使用されるツールおよびコマンドラインが含まれます。
- .rel または .rela で始まるセクションには、ELF の再配置情報が含まれます。
- セクション .symtab には、ファイルのシンボルテーブルが含まれます。
- セクション .strtab には、シンボルテーブルのシンボル名が含まれます。
- セクション .shstrtab には、セクション名が含まれます。

セクションおよびブロックの説明

ここでは、各セグメントのリファレンス情報を説明します。

- **説明**は、セクションが保持する内容のタイプ、また必要に応じて、セクションがリンカによりどのように扱われるかを記述します。
- **メモリ配置**は、メモリ配置制限を記述します。

リンカ設定ファイルを修正することによるメモリでのセクションの割当て方法については、71 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	82 ページの <i>スタックの設定</i> を参照してください。

.cstart

説明	起動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。

.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 <code>initialize by copy</code> を使用する際、対応する <code>.data_init</code> セクションが、それぞれの <code>.data</code> 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	<code>.data</code> 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、 <code>initialize by copy</code> リンカディレクティブが使用された場合に、リンカによって作成されます。
メモリ配置	このセクションは、メモリ内の任意の場所に配置できます。

__DLIB_PERTHREAD

説明	DLIB モジュールの静的状態を含む変数を保持します。
メモリ配置	このセクションは自動的に正しく配置されます。配置を変更しないでください。
関連項目	120 ページの <i>DLIB</i> ライブラリでのマルチスレッドのサポートを参照してください。

.exc.text

説明	アプリケーションが例外を処理するときだけに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	160 ページの <i>例外処理</i> を参照してください。

HEAP

説明	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	「82 ページの <i>ヒープの設定</i> 」。

.iar.dynexit

説明	終了時に行われる呼出しのテーブルを保持します。
メモリ配置	このセクションは、メモリ内の任意の場所に配置できます。
関連項目	82 ページの <i>atexit 制限の設定</i> 。

.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、リセットベクタおよび例外ベクタを保持します。
メモリ配置	アドレス範囲 0x00 ～ 0x3F に配置する必要があります。

IRQ_STACK

説明	IRQ 例外の提供時に使用されるスタックを保持します。その他の例外タイプの提供に必要な場合、他のスタック FIQ、SVC、ABT、UND が追加されることもあります。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	170 ページの <i>例外スタックを参照してください。</i>

.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	421 ページの .init_array。

.prepreinit_array

説明	<code>.init_array</code> 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	421 ページの <code>.init_array</code> 。

.rodata

説明	定数データを保持します。これには、定数変数、文字列、集合リテラルなどをインクルードできます。
メモリ配置	このセクションは、メモリ内の任意の場所に配置できます。

.text

説明	システム初期化用のコードを除く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	<code>__ramfunc</code> 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	309 ページの <code>__ramfunc</code> 。

.textrw_init

説明	<code>.textrw</code> 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	309 ページの <code>__ramfunc</code> 。

IAR ユーティリティ

この章では、利用可能な IAR コマンドラインユーティリティについて説明します。

- 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 for ARM — `ielfdumparm` — 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。

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。

呼出し構文

アーカイブビルダの呼出し構文は以下のとおりです。

iarchive パラメータ

パラメータ

パラメータを以下に示します。

パラメータ	説明
コマンド	実行する操作を定義するコマンドラインオプションです。このようなオプションは、ライブラリファイル名より前に指定する必要があります。
libraryfile	操作対象のライブラリファイルです。
objectfile1 ... objectfileN	指定されたコマンドの操作対象のオブジェクトファイルです。
オプション	実行する動作を定義するコマンドラインオプションです。これらのオプションは、コマンドラインの任意の場所に配置できます。

表 40: 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。

表 41: iarchive コマンドの概要

コマンドラインオプション	説明
--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をリストします。

表 41: iarchive コマンドの概要 (続き)

詳しくは、438 ページの *オプションの説明* を参照してください。

IARCHIVE オプションの概要

以下の表に、iarchive オプションの概要を示します。

コマンドラインオプション	説明
-f	コマンドラインを拡張します。
--output, -o	ライブラリファイルを指定。
--silent	出力抑止操作を設定します。
--verbose, -V	実行されたすべての操作を報告します。

表 42: iarchive オプションの概要

詳しくは、438 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 (ichecksum) は、メモリの特定の範囲におけるチェックサムを生成できます。このチェックサムは、使用しているアプリケーションで計算されるチェックサムと比較できます。

ielftool のソースコードおよび Microsoft VisualStudio 2005 のテンプレートプロジェクトは、arm\src\elfutils ディレクトリにあります。チェックサムの生成方法に関する特定の要件やフォーマット変換に関する要件がある場合には、それに応じてソースコードを変更できます。

呼出し構文

IAR ELF Tool の呼出し構文は以下のとおりです。

```
ielftool [オプション] inputfile outputfile [オプション]
```

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
<i>inputfile</i>	ILINK リンカにより生成される絶対 ELF 実行可能イメージ。
オプション	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、 428 ページの <i>ielftool</i> オプションの概要を参照してください。
<i>outputfile</i>	絶対 ELF 実行可能イメージ。

表 43: *ielftool* のパラメータ

ファイル名やディレクトリの指定については、217 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

例

以下の例では、メモリ範囲が 0xFF で埋め込まれた後、同じ範囲のチェックサムが計算されます。

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
--checksum __checksum:4,crc32;0-0xFF
```

IELFTOOL オプションの概要

以下の表に、ielftool のコマンドラインオプションの一覧を示します。

コマンドラインオプション 説明	
--bin	出力ファイルのフォーマットをバイナリに設定します。
--checksum	チェックサムを生成します。
--fill	ファイルの要件を指定します。
--ihex	出力ファイルのフォーマットを線形の Intel hex に設定します。
--self_reloc	一般用ではありません。
--silent	サイレント処理を設定します。
--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。
--srec	出力ファイルのフォーマットを Motorola S-records に設定します。
--srec-len	各 S-record 内のデータバイト数を制限します。
--srec-s3only	S-record 出力にレコードのサブセットのみが含まれるように制限します。
--strip	デバッグ情報を削除します。
--verbose, -V	実行されたすべての操作を出力します。

表 44: ielftool オプションの概要

詳しくは、438 ページの オプションの説明を参照してください。

IAR ELF Dumper for ARM — ielfdumparm

IAR ELF Dumper for ARM (ielfdumparm) は、再配置可能 ELF ファイルまたは絶対 ELF ファイルの内容のテキスト表示を作成できます。

ielfdumparm は、以下の 3 とおりの方法で使用できます。

- 入力ファイルおよびそのファイルに含まれる ELF セグメント、ELF セクションの一般属性のリストを小さくする。これは、コマンドラインオプションを使用しない場合のデフォルト動作です。

- 入力ファイル内の ELF セクションの内容のテキスト表現も含める。この動作を指定するには、コマンドラインオプション `--all` を使用します。
- 入力ファイルから選択した ELF セクションのテキスト表現を少なくする。この動作を指定するには、コマンドラインオプション `--section` を使用します。

呼出し構文

`ielfdumparm` の呼出し構文は以下のとおりです。

```
ielfdumparm input_file [output_file]
```

注：`ielfdumparm` は、本来、IDE での使用を目的としたコマンドラインツールではありません。

パラメータ

パラメータを以下に示します。

パラメータ	説明
<code>input_file</code>	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。
<code>output_file</code>	出力先のファイルまたはディレクトリ。存在せず <code>--output</code> 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 45: `ielfdumparm` のパラメータ

ファイル名やディレクトリの指定については、217 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。

IELFDUMPARM 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
<code>--all</code>	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>--code</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。
<code>-f</code>	コマンドラインを拡張します。
<code>--output, -o</code>	出力ファイルを指定します。
<code>--no_strtab</code>	文字列テーブルのセクションのダンプを無効にします。

表 46: `ielfdumparm` オプションの概要

コマンドラインオプション 説明	
--raw	すべての選択セクションに対して、そのセクションの専用出力フォーマットではなく、汎用の 16 進数 /ASCII 出力フォーマットを使用します。
--section, -s	選択した入力セクションに対して出力を生成します。

表 46: ielfdumparm オプションの概要 (続き)

詳しくは、438 ページの オプションの説明を参照してください。

IAR ELF オブジェクトツール — iobjmanip

IAR ELF オブジェクトツール、iobjmanip を使用して、ELF オブジェクトファイルの低レベルの操作を実行します。

呼出し構文

IAR ELF オブジェクトツール呼出し構文は以下のとおりです。

```
iobjmanip オプション 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
オプション	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 47: iobjmanip パラメータ

ファイル名やディレクトリの指定については、217 ページの ファイル名またはディレクトリをパラメータとして指定する場合の規則を参照してください。

例

この例では、input.o 内のセクション .example が .example2 にリネームされ、結果は output.o に保存されます。

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

IOBJMANIP オプションの概要

以下の表に、iobjmanip オプションの概要を示します。

コマンドラインオプション	説明
-f	コマンドラインを拡張します。
--remove_section	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。

表 48: iobjmanip オプションの概要

詳しくは、438 ページの *オプションの説明* を参照してください。

診断メッセージ

ここでは、iobjmanip で生成されたメッセージについて説明します。

Lm001: No operation given

コマンドラインパラメータで、実行する処理が指定されていません。

Lm002: *nr* パラメータではなく、*nr* が使用されています。

パラメータが少なすぎるか、または多すぎます。iobjmanip および使用されたコマンドラインオプションの呼出し構文をチェックしてください。

Lm003: Invalid section/symbol renaming pattern *pattern*

パターンに有効なリネーム処理が定義されていません。

Lm004: Could not open file *filename*

iobjmanip が入力ファイルを開くことができませんでした。

Lm005: ELF format error *msg*

入力ファイルは有効な ELF オブジェクトファイルではありません。

Lm006: Unsupported section type *nr*

オブジェクトファイルに、iobjmanip で処理できないセクションが含まれています。出力ファイルの生成時に、このセクションは無視されます。

Lm007: Unknown section type *nr*

iobjmanip で、認識されないセクションが検出されました。iobjmanip は、内容をそのままコピーします。

Lm008: Symbol *symbol* has unsupported format

iobjmanip で、処理できないシンボルが検出されました。iobjmanip は、出力ファイルの生成時にこのシンボルを無視します。

Lm009: Group type *nr* not supported

iobjmanip では、グループタイプ GRP_COMDAT のみがサポートされています。他のグループタイプが検出された場合、結果は未定義になります。

Lm010: Unsupported ELF feature in file: *msg*

入力ファイルが、iobjmanip でサポートしていない機能を使用しています。

Lm011: Unsupported ELF file type

入力ファイルは再配置可能なオブジェクトファイルではありません。

Lm012: Ambiguous rename for section/symbol name (*alt1* and *alt2*)

セクションまたはシンボルのリネーム中に、曖昧な点が見つかりました。代替手段のいずれかが使用されます。

Lm013: Section name removed due to transitive dependency on name

明示的に削除されたセクションに依存していたため、セクションが削除されました。

Lm014: File has no section with index *nr*

--remove_section または --rename_section へのパラメータとして使用されるセクションインデックスが、入力ファイルのセクションを参照していませんでした。

Ms003: could not open file *filename* for writing

iobjmanip が、書込み用の入力ファイルを開くことができませんでした。ライト禁止になっていないか確認してください。

Ms004: problem writing to file *filename*

ファイル *filename* への書込み中にエラーが発生しました。ボリュームが
いっぱいであることが原因と考えられます。

Ms005: problem closing file *filename*

ファイル *filename* を閉じているときにエラーが発生しました。

IAR Absolute Symbol Exporter — isymexport

IAR Absolute Symbol Exporter (isymexport) は、ROM イメージファイルから絶対シンボルをエクスポートします。これらのシンボルは、アドオンアプリケーションのリンク時に使用できます。

呼出し構文

IAR Absolute Symbol Exporter の呼出し構文は以下のとおりです。

isymexport [*オプション*] *inputfile* *outputfile* [*オプション*]

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
<i>inputfile</i>	実行可能 ELF ファイルの形式の ROM イメージです（リンクからの出力）。
<i>オプション</i>	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、434 ページの <i>isymexport</i> の <i>オプションの概要</i> を参照してください。
<i>outputfile</i>	リンク入力として使用可能な再配置可能 ELF ファイルです。このファイルには、入力ファイル内の絶対シンボルのすべてまたは選択内容が含まれます。出力ファイルには、シンボルのみ含まれ、実際のコードやデータセクションは含まれません。ステアリングファイルを使用して、出力ファイルに含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。

表 49: ielftool のパラメータ

ファイル名やディレクトリの指定については、217 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。

ISYMEXPORT のオプションの概要

以下の表に、isymexport のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--edit	ステアリングファイルを指定します。
-f	コマンドラインを拡張します。
--ram_reserve_ranges	イメージが使用する RAM 内のエリアを予約する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。

表 50: isymexport オプションの概要

詳細については、438 ページの オプションの説明を参照してください。

ステアリングファイル

ステアリングファイルを使用して、出力に含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。ステアリングファイルでは、show ディレクティブおよびhide ディレクティブを使用して、入力ファイルからどのパブリックシンボルを出力ファイルに含めるかの選択ができます。rename ディレクティブを使用すると、入力ファイル内のシンボルの名前を変更できます。

構文

以下の構文規則が適用されます。

- それぞれのディレクティブは、別々の行に指定します。
- C のコメント (/*...*/) や C++ のコメント (//...) を使用できます。
- パターンには、シンボル名の複数文字に対応するワイルドカード文字を含めることができます。
- * 文字は、シンボル名の中のゼロ桁または複数桁の文字のシーケンスに一致すると見なされます。
- ? 文字は、シンボル名の中の任意の 1 文字に一致すると見なされます。

例

```
rename xxx_* as YYY_* /* シンボルのプレフィックスを xxx_ から YYY_ に変更 */
show YYY_*           /* すべてのシンボルを YYY パッケージからエクスポート */
hide *_internal      /* ただし、内部シンボルはエクスポートしない */
show zzz?            /* zzza をエクスポートして、zzzaaa をエクスポートしない */
hide zzzx            /* zzzx はエクスポートしない */
```

Show ディレクティブ

構文	<code>show pattern</code>	
パラメータ	<code>pattern</code>	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 <code>hide</code> ディレクティブでオーバーライドされる場合は除きます。	
例	<pre>/* _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</pre>	

Hide ディレクティブ

構文	<code>hide pattern</code>	
パラメータ	<code>pattern</code>	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 <code>show</code> ディレクティブでオーバーライドされる場合は除きます。	
例	<pre>/* _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</pre>	

Rename ディレクティブ

構文	<code>rename pattern1 pattern2</code>	
パラメータ	<code>pattern1</code>	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。
	<code>pattern2</code>	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、 <code>pattern1</code> 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が複数の `rename` パターンに一致す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せんが、`show` および `hide` ディレクティブの前に実行されます。したがって、シンボルをリネームする場合、ステアリングファイルにあるすべての `show` ディレクティブおよび `hide` ディレクティブは新しい名前を参照します。

シンボルの名前が、ワイルドカードを含まない `pattern1` パターンに一致する場合、出力ファイルで `pattern2` にリネームされます。

シンボルの名前が、ワイルドカードを含む `pattern1` パターンに一致する場合、出力ファイルで `pattern2` にリネームされますが、名前のワイルドカード文字に一致する部分は保持されます。

例

```
/* xxx_start は出力ファイルで Y_start_X にリネームされます。
   xxx_stop は出力ファイルで Y_stop_X にリネームされます。*/
rename xxx_ * Y_*_X
```

診断メッセージ

ここでは、`isymexport` で生成されたメッセージについて説明します。

Es001: could not open file *filename*

`isymexport` が指定されたファイルを開くことができませんでした。

Es002: illegal path *pathname*

パス `pathname` は有効なパスではありません。

Es003: format error: message

入力ファイルの読み取り中に問題が発生しました。

Es004: no input file

入力ファイルが指定されていません。

Es005: no output file

入力ファイルは指定されていますが、出力ファイルが指定されていません。

Es006: too many input files

ファイルが 3 つ以上指定されています。

Es007: input file is not an ELF executable

入力ファイルは ELF 実行可能ファイルではありません。

Es008: unknown directive: *directive*

ステアリングファイルに指定されたディレクティブが認識されません。

Es009: unexpected end of file

必要な入力の途中でステアリングファイルが終了しました。

Es010: unexpected end of line

ディレクティブが終了する前にステアリングファイルの行が終了しました。

Es011: unexpected text after end of directive

ステアリングファイルのディレクティブと同じ行に、ディレクティブの後に別のテキストが存在します。

Es012: expected text

指定されたテキストがステアリングファイルに存在しません。このテキストは、ディレクティブを正しく指定するために必要です。

Es013: pattern can contain at most one * or ?

現在のディレクティブの各パターンに含めることができるワイルドカード文字は、* または ? が 1 つだけです。

Es014: rename patterns have different wildcards

現在のディレクティブの両方のパターンに含まれるワイルドカードは、同じ種類でなければなりません。すなわち、両方が以下のいずれかであることが必要です。

- ワイルドカードなし
- * が 1 つ含まれる
- ? が 1 つ含まれる

このエラーは、パターンがワイルドカードに関して両方のパターンが同じでない場合に発生します。


Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

入力ファイル内のシンボルが複数の `rename` パターンに一致しています。


オプションの説明

このセクションでは、さまざまなユーティリティで使用可能な各コマンドラインオプションの詳しいリファレンス情報を提供します。

--all

構文	--all
ツール	ielfdumparm
説明	<p>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。</p> <p>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。</p> <div> このオプションは、IDE では使用できません。</div>

--bin

構文	--bin
ツール	ielftool
説明	<p>出力ファイルのフォーマットをバイナリに設定します。</p> <div> 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]</div>

--checksum

構文	<pre>--checksum {symbol[+offset] address}:size,algorithm[:[1 2][m][r][i p]] [,start];range[:range...]</pre>	
パラメータ	<i>symbol</i>	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。

<i>offset</i>	シンボルへのオフセット。
<i>address</i>	チェックサム値が格納される絶対アドレスです。
<i>size</i>	チェックサムのバイト数です (1、2、4)。これは、チェックサムシンボルのサイズ以下でなければなりません。
<i>algorithm</i>	<p>使用されるチェックサムアルゴリズムです。以下のいずれかです。</p> <ul style="list-style-type: none"> • <i>sum</i>: 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。 • <i>sum8wide</i>: 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。 • <i>sum32</i>: ワード (32 ビット) 単位で計算される算術合計。 • <i>crc16</i>: CRC16 (生成多項式 0x11021)。デフォルトで使用されます。 • <i>crc32</i>: CRC32 (生成多項式 0x104C11DB7) • <i>crc=n</i>: <i>n</i> の生成多項式を使用する CRC
<i>1 2</i>	<p>指定された場合は、以下のどちらかになります：</p> <ul style="list-style-type: none"> • 1 - 1 の補数を指定します。 • 2 - 2 の補数を指定します。
<i>m</i>	チェックサムの計算時に各バイト内でビットの順序を逆にします。
<i>r</i>	サイズ <i>size</i> の各ワード内での入力データのバイト順序を逆にします。
<i>i p</i>	<p><i>start</i> の値が 0 より大きい場合に、<i>i</i> または <i>p</i> を使用します。指定された場合は、以下のどちらかになります：</p> <ul style="list-style-type: none"> • <i>i</i> - チェックサムの値を開始値で初期化します。 • <i>p</i> - 入力データの先頭に <i>start</i> 値を含むサイズ <i>size</i> の 1 ワードを付けます。
<i>start</i>	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、 <i>start</i> を使用してください。0 でない場合は、 <i>i</i> か <i>p</i> のどちらかを指定する必要があります。
<i>range</i>	チェックサムが計算されるアドレス範囲です。16 進表記および 10 進表記を使用できます (たとえば、0x8002-0x8FFF)。

ツール

ielftool

説明

このオプションは、指定範囲の指定アルゴリズムのチェックサムを計算するときに使用します。チェックサムは、*symbol* の元の値を置き換えます。新しい絶対シンボルが生成されます。計算されたチェックサムを含む *_value* がサフィックスとして *symbol* 名に付けられます。このシンボルは、デバッグ中など、必要に応じて後でチェックサム値へのアクセスに使用できます。

--checksum オプションがコマンドラインで複数回使用される場合、オプションは、左から右に評価されます。後で評価される --checksum オプションに指定されている *symbol* で、チェックサムが計算される場合、エラーが発生します。

例


この例は、アドレス範囲 0x8000-0x8FFF、開始値 0 の場合の **crc16** アルゴリズムの使用方法を示します。

```
ielftool --checksum=__checksum:2,crc16;0x8000-0x8FFF
sourceFile.out destinationFile.out
```

sourceFile.out から読み込まれる入力データ *i* と、その結果のサイズ 2 バイトのチェックサム値が、シンボル __checksum に格納されます。修正された ELF ファイルは、destinationFile.out として保存されます。sourceFile.out はそのまま変わりません。

関連項目


173 ページの *チェックサムの計算*。

 オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [チェックサム]

--code

構文	--code
ツール	ielfdump
説明	このオプションを使用して、実行可能コード (ELF セクション属性 SHF_EXECINSTR を持つセクション) を含むすべてのセクションをダンプします。

 このオプションは、IDE では使用できません。

--create

構文	--create libraryfile objectfile1 ... objectfileN				
パラメータ	<table><tr><td>libraryfile</td><td>コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 ページの <i>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</i>を参照してください。</td></tr><tr><td>objectfile1 ...objectfileN</td><td>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。</td></tr></table>	libraryfile	コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 ページの <i>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</i> を参照してください。	objectfile1 ...objectfileN	ビルドするライブラリを構成するオブジェクトファイルです。
libraryfile	コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 ページの <i>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</i> を参照してください。				
objectfile1 ...objectfileN	ビルドするライブラリを構成するオブジェクトファイルです。				

ツール	iarchive
説明	<p>このコマンドは、一連のオブジェクトファイル（モジュール）から新しいライブラリをビルドするときに使用します。オブジェクトファイルは、コマンドラインで指定した順序どおりにライブラリに追加されます。</p> <p>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 <code>--create</code> が使用されます。</p>



このオプションは、IDE では使用できません。

--delete, -d

構文	<pre>--delete libraryfile objectfile1 ... objectfileN -d libraryfile objectfile1 ... objectfileN</pre>	
パラメータ	<p><i>libraryfile</i> コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 ページの <i>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。</i></p> <p><i>objectfile1</i> ...<i>objectfileN</i> コマンドの操作対象のオブジェクトファイルです。</p>	
ツール	iarchive	
説明	<p>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。</p>	
		このオプションは、IDE では使用できません。

--edit

構文	<code>--edit steering_file</code>	
ツール	isymexport	
説明	<p>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。</p>	

関連項目

434 ページの ステアリングファイル。



このオプションは、IDE では使用できません。

--extract, -x

構文

```
--extract libraryfile [objectfile1 ... objectfileN]
-x libraryfile [objectfile1 ... objectfileN]
```

パラメータ

libraryfile コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

objectfile1
...*objectfileN* コマンドの操作対象のオブジェクトファイルです。

ツール

iarchive

説明

このコマンドは、オブジェクトファイル（モジュール）を既存のライブラリから抽出するときに使用します。オブジェクトファイルのリストを指定すると、これらのファイルのみ抽出されます。オブジェクトファイルのリストを指定しない場合には、ライブラリ内のすべてのオブジェクトファイルが抽出されます。



このオプションは、IDE では使用できません。

-f

構文

```
-f filename
```

パラメータ

ファイル名の指定については、217 ページの *ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

ツール

iarchive、ielfdumparm、iobjmanip、isymexport.

説明

このオプションは、ツールで、指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。



このオプションは、IDE では使用できません。

--fill

構文

```
--fill pattern;range[;range...]
```

パラメータ

<i>range</i>	フィルのアドレス範囲を指定します。16 進表記および 10 進表記を使用できます (たとえば、0x8002-0x8FFF)。各アドレスは 4 バイトアラインメントでなければならないので注意してください。
<i>pattern</i>	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、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。

適用範囲

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]

--ihex

構文

```
--ihex
```

ツール

ielftool

説明

出力ファイルのフォーマットを線形の Intel hex に設定します。



オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > リンカ > [出力コンバータ]

--no_strtab

構文

--no_strtab

ツール

ielfdumparm

説明

このオプションを使用して、文字列のテーブルセクション（SHT_STRTAB 型のセクション）のダンプを無効にします。



このオプションは、IDE では使用できません。

--output, -o

構文

-o {filename|directory}
--output {filename|directory}

パラメータ

ファイル名やディレクトリの指定については、217 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、IAR ELF Dumper から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。

--ram_reserve_ranges

構文	<code>--ram_reserve_ranges[=symbol_prefix]</code>	
パラメータ	<i>symbol_prefix</i>	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	<p>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 <i>symbol_prefix</i> がプレフィックスとして付きます。</p> <p>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ク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。</p> <p><code>--ram_reserve_ranges</code> と <code>--reserve_ranges</code> を同時に使用する場合、RAM エリアは <code>--ram_reserve_ranges</code> オプションからプレフィックスを、RAM 以外のエリアは <code>--reserve_ranges</code> オプションからプレフィックスをそれぞれ取得します。</p>	
関連項目	447 ページの <code>--reserve_ranges</code> 。	



このオプションは、IDE では使用できません。


--raw

構文	<code>--raw</code>	
ツール	ielfdumparm	
説明	<p>デフォルトでは、特定種類のセクション専用の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。</p> <p>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。</p>	




このオプションは、IDE では使用できません。

--remove_section

構文	<code>--remove_section {section number}</code>	
パラメータ	<i>section</i>	セクションを削除します（複数も可）。
	<i>number</i>	削除されるセクションの番号。セクション番号は、 <code>ielfdumparm</code> 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	<code>iobjmanip</code>	
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 <code>iobjmanip</code> 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	<code>--rename_section {oldname oldnumber}=newname</code>	
パラメータ	<i>oldname</i>	セクションをリネームします（複数も可）。
	<i>oldnumber</i>	リネームされるセクションの番号。セクション番号は、 <code>ielfdumparm</code> 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。
	<i>newname</i>	セクションの新しい名前。
ツール	<code>iobjmanip</code>	
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 <code>iobjmanip</code> 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	<code>--rename_symbol oldname =newname</code>	
パラメータ	<i>oldname</i>	リネームするシンボル。
	<i>newname</i>	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	<pre>--replace libraryfile objectfile1 ... objectfileN -r libraryfile objectfile1 ... objectfileN</pre>	
パラメータ	<p><i>libraryfile</i> コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 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。</p> <p><i>objectfile1</i> ...<i>objectfileN</i> コマンドの操作対象のオブジェクトファイルです。</p>	

ツール	iarchive
説明	このコマンドは、既存のライブラリでオブジェクトファイル（モジュール）の置換または追加を行うときに使用します。コマンドラインで指定したオブジェクトファイルにより、ライブラリ内の同一名の既存オブジェクトファイルが置換されます。同一名のファイルが存在しない場合には、コマンドラインで指定したファイルがライブラリに追加されます。



このオプションは、IDE では使用できません。

--reserve_ranges

構文	--reserve_ranges [=symbol_prefix]
パラメータ	symbol_prefix このオプションによって作成されたシンボルのプレフィックス。
ツール	isymexport
説明	このオプションを使用して、イメージが使用する ROM および RAM のエリアについてシンボルを生成します。各エリアにシンボルが 1 つ生成されます。各シンボルの名前はエリア名に基づき、オプションのパラメータ symbol_prefix がプレフィックスとして付きます。

あるエリアをカバーするシンボルをこの方法で生成すると、影響を受けるアドレスにリンカで他の内容を配置しないように防ぐことができます。これは、既存のイメージに対するリンク処理の際に役立ちます。

--reserve_ranges 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 445 ページの --ram_reserve_ranges。



このオプションは、IDE では使用できません。

--section, -s

構文 --section section_number|section_name[,...]
--s section_number|section_name[,...]

パラメータ section_number ダンプされるセクションの数。
section_name ダンプされるセクションの名前。

ツール ielfdumparm

説明 このオプションは、指定した番号のセクションまたは指定した名前のセクションの内容をダンプするときに使用します。選択したセクションに再配置可能セクションが関連付けられている場合には、その内容も出力されます。
このオプションを使用する場合、入力ファイルの一般属性は出力に含まれません。
セクション番号や名前をカンマで区切るか、このオプションを複数回使用することにより、複数のセクション番号や名前を指定できます。
デフォルトでは、セクションの内容は出力に含まれません。

例 -s 3,17 /* セクション No.3 と No.17
-s .debug_frame,42 /* .debug_frame という名の任意のセクションおよびセクション No.42 */



このオプションは、IDE では使用できません。

--self_reloc

構文	<code>--self_reloc</code>
ツール	<code>ielftool</code>
説明	このオプションは一般用ではないため、意図的に文書化されていません。



このオプションは、IDE では使用できません。

--silent

構文	<code>--silent</code> <code>-S</code> (<code>iarchive</code> のみ)
ツール	<code>iarchive</code> および <code>ielftool</code> .
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、 <code>ielftool</code> によりさまざまなメッセージが標準出力ストリームから送信されます。このオプションを使用して、これらのメッセージ送信を抑止できます。エラーおよび警告メッセージは、 <code>ielftool</code> からエラー出力ストリームに送信されるため、この設定にかかわらず表示されます。



このオプションは、IDE では使用できません。

--simple

構文	<code>--simple</code>
ツール	<code>ielftool</code>
説明	出力ファイルのフォーマットを簡易コードに設定します。



オプションを設定するには、以下のように選択します。
[プロジェクト] > [オプション] > [出力コンバータ]

--srec

構文	<code>--srec</code>
ツール	<code>ielftool</code>
説明	出力ファイルのフォーマットを Motorola S-records に設定します。



オプションを設定するには、以下のように選択します。
[プロジェクト] > [オプション] > [出力コンバータ]

--srec-len

構文	<code>--srec-len=length</code>
パラメータ	<i>length</i> 各 S-record 内のデータバイト数。
ツール	<code>ielftool</code>
説明	各 S-record 内のデータバイト数を制限します。このオプションは、 <code>--srec</code> オプションと組み合わせて使用できます。



このオプションは、IDE では使用できません。


--srec-s3only

構文	<code>--srec-s3only</code>
ツール	<code>ielftool</code>
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、 <code>--srec</code> 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	<code>--strip</code>
ツール	iobjmanip および ielftool。
説明	<p>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。</p> <p>ielftool では、<code>--strip</code> 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。<code>--strip</code> オプションをリンカで使用する場合、これを削除して、代わりに ielftool の <code>--strip</code> オプションを使用します。</p> <p> オプションを設定するには、以下のように選択します。</p> <p>[プロジェクト] > [オプション] > [リンカ] > [出力] > [出力ファイルにデバッグ情報を含める]</p>


--symbols

構文	<code>--symbols libraryfile</code>
パラメータ	<p><i>libraryfile</i> コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 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。</p>
ツール	iarchive
説明	<p>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。</p> <p>出力抑止モード (<code>--silent</code>) 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。</p> <p> このオプションは、IDE では使用できません。</p>

--toc, -t

構文	<pre>--toc libraryfile -t libraryfile</pre>	
パラメータ	<i>libraryfile</i>	コマンドの操作対象のライブラリファイルです。ファイル名の指定については、217 ページの ファイル名またはディレクトリをパラメータとして指定する場合の規則 を参照してください。
ツール	iarchive	
説明	<p>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。</p> <p>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。</p> <p> このオプションは、IDE では使用できません。</p>	

--verbose, -V

構文	<pre>--verbose -V (iarchive のみ)</pre>	
ツール	iarchive および ielftool.	
説明	<p>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。</p> <p> 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。</p>	

処理系定義の動作

この章では、IAR システムの C 言語の処理系定義の処理方法について説明します。

注: IAR システムズの実装は、標準の C のフリースタANDING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。

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。

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
```

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

文字セット (5.1.1.2)

ソースの文字セットは、物理的なソースファイルのマルチバイト文字セットと同じです。デフォルトでは、標準の ASCII 文字セットが使用されます。ただし、`--enable_multibytes` コンパイラオプションを使用する場合、ホストの文字セットが代わりに使用されます。

Main (5.1.2.1)

プログラム起動時に呼び出される関数は、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 )
```

108 ページの *システム初期化のカスタマイズ* を参照してください。

プログラム終了の影響 (5.1.2.1)

アプリケーションを終了すると、(main の呼出し直後に) 実行が起動コードに戻ります。

その他の main の定義方法 (5.1.2.2.1)

main 関数を定義する方法は他にありません。

main の argv 引数 (5.1.2.2.1)

argv 引数はサポートされていません。

対話型デバイスとしてのストリーム (5.1.2.3)

ストリーム stdin、stdout、stderr は対話型デバイスとして処理されます。

シグナルとその意味およびデフォルトの処理 (7.14)

DLIB ライブラリでは、サポートされているシグナルのセットは標準の C と同じです。引き起こされたシグナルは、signal 関数がアプリケーションに合うようにカスタマイズされていない限り、何の処理も行いません。

計算の例外のシグナル値 (7.14.1.1)

DLIB ライブラリでは、計算の例外に一致する処理系定義の値はありません。

システム起動時のシグナル (7.14.1.1)

DLIB ライブラリでは、システム起動時に実行される処理系定義のシグナルはありません。

環境名 (7.20.4.5)

DLIB ライブラリでは、getenv 関数に使用される処理系定義の環境名はありません。

システム関数 (7.20.4.6)

system 関数はサポートされていません。

J.3.3 識別子

識別子のマルチバイト文字 (6.4.2)

その他のマルチバイト文字は識別子に表示されないことがあります。

識別子における重要な文字 (5.2.4.1, 6.1.2)

外部リンケージの有無に関わらず、識別子の重要な先頭文字の数は 200 文字以上であることが保証されています。

J.3.4 文字

バイト内のビット数 (3.6)

1 バイトには 8 ビットが含まれます。

実行文字セットのメンバ値 (5.2.1)

実行文字セットのメンバ値は、ASCII 文字セットの値です。これらはホストの文字セットの追加文字の値によって追加することが可能です。

英数字のエスケープシーケンス (5.2.2)

標準の英数字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。

重要な基本文字セット外の文字 (6.2.5)

`char` に保存された重要な基本文字セット外の文字は変換されません。

`char` 型 (6.2.5, 6.3.1.1)

通常の `char` 型は、`unsigned char` として処理されます。

ソースおよび実行文字セット (6.4.4.4, 5.1.1.2)

ソース文字セットは、ソースファイルで使える正当な文字セットです。デフォルトのソース文字セットは、標準 ASCII 文字セットです。ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、ソース文字セットはホストコンピュータのデフォルトの文字セットになります。

実行文字セットは、実行環境で使える正当な文字セットです。デフォルトの実行文字セットは、標準 ASCII 文字セットです。

ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、実行文字セットはホストコンピュータのデフォルトの文字セットになります。IAR DLIB ライブラリでは、マルチバイトの実行文字セット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。*115 ページの ロケールを参照してください。*

複数の文字を含む整数文字定数 (6.4.4.4)

文字数が複数の整数文字定数は、整数定数として処理されます。値は、整数定数で左端の文字を最上位文字、右端の文字を最下位文字として計算されます。値が整数定数で表現できない場合は、診断メッセージが出力されます。

複数の文字を含むワイド文字定数 (6.4.4.4)

複数のマルチバイト文字を含むワイド文字定数を使用すると、診断メッセージが出力されます。

ワイド文字定数に使用されるロケール (6.4.4.4)

デフォルトでは C ロケールが使用されます。`--enable_multibytes` コンパイラオプションが使用される場合、デフォルトのホストロケールが代わりに使用されます。

ワイド文字列リテラルに使用されるロケール (6.4.5)

デフォルトでは C ロケールが使用されます。`--enable_multibytes` コンパイラオプションが使用される場合、デフォルトのホストロケールが代わりに使用されます。

重要文字としてのソース文字 (6.4.5)

すべてのソース文字は、重要文字として表すことができます。

J.3.5 整数

拡張整数型 (6.2.5)

拡張整数型はありません。

整数値の範囲 (6.2.6.2)

整数値は、2 の補数で表現されます。最上位ビットは符号を示し、1 の場合は負の値、0 の場合は正の値またはゼロを示します。

異なる整数型の範囲について詳しくは、287 ページの *基本データ型* 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。

Signed bitwise operations (6.5)

符号付整数に対するビット単位演算は、符号なし整数に対するビット単位演算と同様に行われます。すなわち、符号ビットが他のビットと同様に扱われます。

J.3.6 浮動小数点**浮動小数点処理の精度 (5.2.4.2.2)**

浮動小数点処理の精度は不明です。

丸め処理 (5.2.4.2.2)

FLT_ROUNDS の非標準値はありません。

評価方法 (5.2.4.2.2)

FLT_EVAL_METHOD の非標準値はありません。

整数値の浮動小数点値への変換 (6.3.1.4)

整数値がソース値を正確に表現できない浮動小数点値に変換されると、最も近い値に丸めるモードが使用されます (FLT_ROUNDS が 1 に定義されています)。

浮動小数点の浮動小数点への変換 (6.3.1.5)

浮動小数点値がソース値を正確に表現できない浮動小数点値に変換されると、最も近い値に丸めるモードが使用されます (FLT_ROUNDS が 1 に定義されています)。

浮動小数点定数値の記述 (6.4.4.2)

最も近い値への丸めモードが使用されます (FLT_ROUNDS が 1 に定義されています)。

浮動小数点値の縮約 (6.5)

浮動小数点値は縮約されます。ただし、精度のロスはありません。シグナルはサポートされていないため、これは問題にはなりません。

FENV_ACCESS のデフォルトの状態 (7.6.1)

プラグマディレクティブ FENV_ACCESS のデフォルトの状態は、OFF です。

その他の浮動小数点メカニズム (7.6, 7.12)

浮動小数点例外、丸めモード、環境、分類は他にはありません。

FP_CONTRACT のデフォルトの状態 (7.12.2)

プラグマディレクティブ FP_CONTRACT のデフォルトの状態は、OFF です。

J.3.7 配列およびポインタ

ポインタの変換 (6.3.2.3)

データポインタおよび関数ポインタのキャストについては、295 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、295 ページの *ptrdiff_t* を参照してください。

J.3.8 ヒント

レジスタキーワードの考慮 (6.7.1)

レジスタ変数についてのユーザ要求は考慮されません。

関数のインライン化 (6.7.4)

関数のインライン化へのユーザ要求で確率は高くなりますが、関数が実際に別の関数にインライン化されるか確実ではありません。プラグマディレクティブ「321 ページの *inline*」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

'plain' ビットフィールドの符号 (6.7.2, 6.7.2.1)

'plain' int ビットフィールドの処理については、「288 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 (232 ページの *-e* を参照)。

記憶単位の境界をまたぐビットフィールド (6.7.2.1)

ビットフィールドは常に 1 つの記憶単位にだけ配置されます。つまり、ビットフィールドは記憶単位をまたぐことはできません。

単位内のビットフィールドの割当順序 (6.7.2.1)

記憶単位内のビットフィールドの割当て方法については、288 ページの ビットフィールドを参照してください。

ビットフィールド以外の構造体メンバのアラインメント (6.7.2.1)

ビットフィールド以外の構造体メンバのアラインメントは、メンバ型と同じです (285 ページの アラインメントを参照)。

列挙型を表すときに使用される整数型 (6.7.2.2)

特定の列挙型用を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 (298 ページの オブジェクトの volatile 宣言を参照)。

J.3.11 プリプロセッサディレクティブ

ヘッダ名のマッピング (6.4.7)

ヘッダ名のシーケンスは、ソースファイル名にそのままマップされます。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。377 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、実行文字セットの同じ文字定数の値を突き合わせます。

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 (224 ページの *--char_is_signed* を参照)。

括弧のついたファイル名のインクルード (6.10.2)

角括弧 `<>` のファイル仕様に使用される検索アルゴリズムについては、207 ページの *インクルードファイル検索手順* を参照してください。

引用符のあるファイル名のインクルード (6.10.2)

引用符で囲まれたファイル仕様に使用される検索アルゴリズムについては、207 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せられます。

`#include` ディレクティブのネストの制限 (6.10.2)

`#include` 処理のネストに明示的な制限はありません。

汎用文字名 (6.10.3.2)

汎用文字名 (UCN) はサポートされていません。

認識されているプリプロセッサディレクティブ (6.10.6)

「プリプロセッサディレクティブ」で説明したプリプロセッサ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。プリプロセッサディレクティブが「プリプロセッサディレクティブ」の章とここの両方に

記載されている場合、この情報よりも「プラグマディレクティブ」の章の情報が優先します。

```
alignment
baseaddr
basic_template_matching
building_runtime
can_instantiate
codeseg
constseg
cspy_support
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
library_requirement_override
library_provides
library_requirement_override
memory
module_name
no_pch
once
public_equ
system_include
vector
warnings
```

Default `__DATE__` and `__TIME__` (6.10.8)

`__TIME__`、`__DATE__` の定義は常に使用可能です。

J.3.12 ライブラリ関数

その他のライブラリ機能 (5.1.2.1)

標準のライブラリ機能のほとんどがサポートされています。その一部（オペレーティングシステムを必要とするもの）には、アプリケーションに低レベルの実装が必要です。詳細については、89 ページの *DLIB* ランタイムライブラリを参照してください。

アサート関数によって出力される診断 (7.2.1.1)

`assert()` 関数で出力される診断は、以下のとおりです。

```
filename:linenr expression -- assertion failed
```

この診断は、パラメータがゼロに評価される場合に出力されます。

浮動小数点のステータスフラグの表現 (7.6.2.2)

浮動小数点のステータスフラグについては、390 ページの *fernv.h* を参照してください。

浮動小数点の例外を引き起こす `Feraiseexcept` (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、293 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、115 ページの *ロケール* を参照してください。

`float_t` and `double_t` に定義される型 (7.12)

`FLT_EVAL_METHOD` マクロは、値 0 しか持つことができません。

ドメインエラー (7.12.1)

標準で必要とされるもの以外のドメインエラーを生成する関数はありません。

ドメインエラーのリターン値 (7.12.1)

数学関数は、ドメインエラーに対して浮動小数点 NaN（not a number = 非数）を返します。

アンダーフローエラー (7.12.1)

数学関数は `errno` をマクロ `ERANGE` (`errno.h` のマクロ) に設定し、アンダーフローエラーにゼロを返します。

fmod のリターン値 (7.12.10.1)

`fmod` 関数は、2 番目の引数がゼロの場合、浮動小数点 NaN を返します。

remquo の規模 (7.12.10.3)

規模は、合同剰余 `INT_MAX` です。

signal() (7.14.1.1)

シグナル関連のライブラリはサポートされていません。

注：低レベルインタフェース関数はライブラリには存在しますが、これらの関数は何も実行しません。アプリケーション固有のシグナル処理を実装する場合は、テンプレートソースコードを使用してください。118 ページの *シグナル関数* を参照してください。

NULL マクロ (7.17)

`NULL` マクロは、0 に定義されています。

改行文字による終了 (7.19.2)

`stdout` ストリーム関数は、`newline` または `end of file` (EOF) 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 (7.19.2)

ストリームで改行文字直前に書き込まれた空白文字は保持されます。

バイナリストリームに書き込まれたデータに追加された Null 文字 (7.19.2)

バイナリストリームにライトされたデータには、`NULL` 文字は追加されません。

追加モードでのファイル位置 (7.19.3)

ファイル位置は、追加モードで開かれた場合にファイルの先頭に配置されます。

ファイルの切捨て (7.19.3)

テキストストリームへのライトにより、対応するファイルでその書込み位置以降が切り捨てられるかどうかは、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。114 ページの *ファイル I/O* を参照してください。

ファイルのバッファ処理 (7.19.3)

開かれたファイルは、ブロックバッファ、ラインバッファまたはアンバッファのいずれかです。

ゼロ長のファイル (7.19.3)

ゼロ長のファイルが存在するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

有効なファイル名 (7.19.3)

ファイル名が有効かどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

ファイルを開くことができる回数 (7.19.3)

ファイルを何度も開くことができるかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

ファイル内のマルチバイト文字 (7.19.3)

ファイル内のマルチバイト文字のエンコーディングは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

remove() (7.19.4.1)

開いたファイルに対する削除処理の結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。114 ページの *ファイル I/O* を参照してください。

rename() (7.19.4.2)

ファイルの名前を既存のファイル名に変更した結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。114 ページの *ファイル I/O* を参照してください。

開かれた一時ファイルの削除 (7.19.4.3)

開かれた一時ファイルを削除するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

モード変更 (7.19.5.4)

`freopen` は指定のストリームを閉じて、新しいモードで再び開きます。ストリーム `stdin`、`stdout`、`stderr` は、すべての新しいモードで再び開くことができます。

infinity または NaN の出力形式 (7.19.6.1, 7.24.2.1)

浮動小数点定数の `infinity` または `NaN` の出力形式は、それぞれ `inf` および `nan` (`F` 変換指定子の場合 `INF` と `NAN`) です。`n-char-sequence` は、`nan` に対しては使用されません。

`printf()` における `%p` (7.19.6.1, 7.24.2.1)

`printf()` の `%p` 変換指定子（出力ポインタ）の引数は、`void *` 型として処理されます。値は、`%x` 変換指定子の場合と同様に、16 進数として出力されます。

`scanf` の読み込み範囲 (7.19.6.2, 7.24.2.1)

- (ダッシュ) 文字は、常に範囲記号として処理されます。

`scanf` における `%p` (7.19.6.2, 7.24.2.2)

`scanf()` の `%p` 変換指定子（スキャンポインタ）は、16 進数を読み取り、`void *` 型の値に変換します。

ファイル位置のエラー (7.19.9.1, 7.19.9.3, 7.19.9.4)

ファイル位置のエラーでは、関数 `fgetpos`、`ftell`、`fsetpos` は `EFPOS` を `errno` に格納します。

NaN の後の `n-char-sequence` (7.20.1.3, 7.24.4.1.1)

NaN の後の `n-char-sequence` は、読み込まれて無視されます。

アンダーフローの `errno` 値 (7.20.1.3, 7.24.4.1.1)

`errno` は、アンダーフローがあった場合には `ERANGE` に設定されます。

サイズがゼロのヒープオブジェクト (7.20.3)

サイズがゼロのヒープオブジェクトの要求は、NULL ポインタではなく有効なポインタを返します。

abort および exit の動作 (7.20.4.1, 7.20.4.4)

`abort()` または `_Exit()` を呼出しても、ストリームバッファはフラッシュされず、オープンストリームを閉じたり、一時ファイルが削除されることはありません。

終了ステータス (7.20.4.1, 7.20.4.3, 7.20.4.4)

終了ステータスはパラメータとして `__exit()` に伝播されます。`exit()` と `_Exit()` は入力パラメータを使用しますが、`abort` は `EXIT_FAILURE` を使用します。

システム関数のリターン値 (7.20.4.6)

`system` 関数はサポートされていません。

タイムゾーン (7.23.1)

ローカルのタイムゾーンおよび夏時間をアプリケーションで定義する必要があります。詳細については、118 ページの *時間関数* を参照してください。

時間の範囲および精度 (7.23)

実装では `clock_t` および `time_t` を表現するときに、1970 年の始めに基づいて `signed long` を使用します。これによって、範囲は秒単位で前後に約 69 年となります。ただし、アプリケーションは関数 `time` と `clock` に実際の実装を提供する必要があります。118 ページの *時間関数* を参照してください。

clock() (7.23.2.1)

アプリケーションは、`clock` 関数の実装を提供する必要があります。118 ページの *時間関数* を参照してください。

%Z 置換文字列 (7.23.3.5, 7.24.5.1)

デフォルトでは、`": "` が `%Z` の代わりに使用されます。使用するアプリケーションがタイムゾーンの処理を実装することになります。118 ページの *時間関数* を参照してください。

数学関数の丸めモード (F.9)

`math.h` の関数は、`FLT-ROUNDS` の丸め方向モードに従います。

J.3.13 アーキテクチャ

一部のマクロに割り当てられた値と式 (5.2.4.2, 7.18.2, 7.18.3)

1 バイトは常に 8 ビットです。

MB_LEN_MAX は、使用されるライブラリ設定に応じて最高で 6 バイトになります。

すべての基本型のサイズや範囲などについては、285 ページの *データ表現* を参照してください。

stdint.h で定義される正確な幅、最小幅、最高速かつ最小幅の整数型の制限マクロは、char、short、int、long、long long と同じ範囲になります。

浮動小数点定数 FLT_ROUNDS の値は 1（最も近い値）で、浮動小数点定数 FLT_EVAL_METHOD の値は 0（そのまま処理）です。

バイトの数値、順序、エンコーディング (6.2.6.1)

285 ページの *データ表現* を参照してください。

sizeof 演算子の結果の値 (6.5.3.4)

285 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セット (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セットにあるすべての 1 バイト文字を受け入れます。コンパイラオプション `--enable_multibytes` を使用する場合、ホストのマルチバイト文字はコメントや文字列リテラルでも受け入れられます。

その他の文字セットの意味 (5.2.1.2)

拡張ソース文字セットのすべてのマルチバイト文字は、拡張実行文字セットにそのまま変換されます。文字が正しく処理されるかどうかは、ライブラリ設定のサポートを持ったアプリケーションに依存します。

マルチバイト文字のエンコードのシフト状態 (5.2.1.2)

コンパイラオプション `--enable_multibytes` を使用すると、ホストのデフォルトのマルチバイト文字が拡張ソース文字として使用可能になります。

連続する出力文字の方向 (5.2.2)

アプリケーションが表示デバイスの特性を定義します。

小数点の文字 (7.1.1)

デフォルトの小数点の文字は'.'です。ライブラリ設定シンボル `_LOCALE_DECIMAL_POINT` を定義すれば、設定し直すことができます。

出力文字 (7.4, 7.25.2)

出力文字セットは、選択したロケールによって決まります。

制御文字 (7.4, 7.25.2)

制御文字セットは、選択したロケールによって決まります。

テスト済みの文字 (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

テスト済みの文字セットは、選択したロケールによって決まります。

ネイティブ環境 (7.1.1.1)

ネイティブ環境は、"C" ロケールと同じです。

数値変換関数の対象シーケンス (7.20.1, 7.24.4.1)

数値変換関数で受け入れが可能な他の対象シーケンスはありません。

実行文字セットの照合 (7.21.4.3, 7.24.4.4.2)

実行文字セットの照合は、選択したロケールによって決まります。

strerror によって返されるメッセージ (7.21.6.2)

strerror 関数が返すメッセージは、引数に応じて以下ようになります。

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 <i>nnn</i>

表 51: strerror() が返すメッセージ—IAR DLIB ライブラリ

A

<code>__AAPCS__</code> (定義済シンボル)	378
<code>--aapcs</code> (コンパイラオプション)	223
<code>__AAPCS_VFP__</code> (定義済シンボル)	378
ABI、AEABI、IA64	178
abort	
システム終了 (DLIB)	107
処理系定義の動作	466
Abort_Handler (例外関数)	60
<code>__absolute</code> (拡張キーワード)	305
<code>--aeabi</code> (コンパイラオプション)	223
<code>__AEABI_PORTABILITY_LEVEL</code> (プロセッサ シンボル)	180
<code>__AEABI_PORTABLE</code> (プロセッサシンボル)	180
algorithm (STL ヘッドファイル)	387
alignment (プラグマディレクティブ)	461
<code>__ALIGNOF__</code> (演算子)	151
<code>--align_sp_on_irq</code> (コンパイラオプション)	223
<code>--all</code> (ielfdump オプション)	438
argv (引数)、処理系定義の動作	454
ARM	
CPU モード	49
Thumb コード、概要	57
サポートされているデバイス	34
メモリレイアウト	53
<code>__arm</code> (拡張キーワード)	305
<code>--arm</code> (コンパイラオプション)	224
<code>__ARMVFP__</code> (定義済シンボル)	378
<code>__ARMVFPV2__</code> (定義済シンボル)	378
<code>__ARMVFPV3__</code> (定義済シンボル)	378
<code>__ARMVFPV4__</code> (定義済シンボル)	378
<code>__ARMVFP_D6__</code> (定義済シンボル)	378
<code>__ARMVFP_FP16__</code> (定義済シンボル)	378
<code>__ARMVFP_SP__</code> (定義済シンボル)	379
<code>__ARM_ADVANCED_SIMD__</code> (定義済シンボル) ..	378
<code>__ARM_MEDIA__</code> (定義済シンボル)	378
<code>__ARM_PROFILE_M__</code> (定義済シンボル)	378
<code>__ARM4M__</code> (定義済シンボル)	379

<code>__ARM4TM__</code> (定義済シンボル)	379
<code>__ARM5__</code> (定義済シンボル)	379
<code>__ARM5E__</code> (定義済シンボル)	379
<code>__ARM6__</code> (定義済シンボル)	379
<code>__ARM6M__</code> (定義済シンボル)	379
<code>__ARM6SM__</code> (定義済シンボル)	379
<code>__ARM7A__</code> (定義済シンボル)	379
<code>__ARM7EM__</code> (定義済シンボル)	379
<code>__ARM7M__</code> (定義済シンボル)	379
<code>__ARM7R__</code> (定義済シンボル)	379
asm、 <code>__asm</code> (言語拡張)	148
assert.h (DLIB ヘッドファイル)	386
<code>__assignment_by_bitwise_copy_allowed、</code> ライブラリで使用するシンボル	391
atexit	119
呼出し用空間の予約	82
atexit 制限、設定	82
auto、イニシャライザのパッキングアルゴリズム ..	403

B

Barr, Michael	25
baseaddr (プラグマディレクティブ)	461
<code>__BASE_FILE__</code> (定義済シンボル)	379
<code>--basic_heap</code> (リンカオプション)	262
basic_template_matching (プラグマディレクティブ)	461
<code>--BE8</code> (リンカオプション)	262
<code>--BE32</code> (リンカオプション)	263
<code>__big_endian</code> (拡張キーワード)	306
<code>--bin</code> (ielftool オプション)	438
bitfields (プラグマディレクティブ)	317
bitset (ライブラリヘッドファイル)	387
bool (データ型)	287
サポートを追加、DLIB	386, 389
.bss (ELF セクション)	419
building_runtime (プラグマディレクティブ)	461
<code>__BUILD_NUMBER__</code> (定義済シンボル)	379
Burrows-Wheeler アルゴリズム、 パッキングイニシャライザ	403

BusFault_Handler (例外関数)	65
bwt、イニシャライザのパッキングアルゴリズム	403

C

calloc (ライブラリ関数)	55
ヒープも参照	
can_instantiate (プラグマディレクティブ)	461
cassert (ライブラリヘッダファイル)	388
ccomplex (ライブラリヘッダファイル)	388
cctype (DLIB ヘッダファイル)	388
cerrno (DLIB ヘッダファイル)	388
cexit (システム終了コード)	104
cfenv (ライブラリヘッダファイル)	388
CFI (アセンブラディレクティブ)	143
cfiCommon.i (CFI ヘッダファイル例)	145
CFI_COMMON_ARM (CFI マクロ)	145
CFI_COMMON_Thumb (CFI マクロ)	145
CFI_NAME_BLOCK (CFI マクロ)	145
cfloat (DLIB ヘッダファイル)	388
char (データ型)	287
signed と unsigned	288
デフォルト表現の変更 (--char_is_signed)	224
処理系定義の動作	455
表現の変更 (--char_is_unsigned)	225
char 型、処理系定義の動作	455
--char_is_signed (コンパイラオプション)	224
--char_is_unsigned (コンパイラオプション)	225
cinttypes (DLIB ヘッダファイル)	388
ciso646 (ライブラリヘッダファイル)	388
class メモリ (拡張 EC++)	163
climits (DLIB ヘッダファイル)	389
locale (DLIB ヘッダファイル)	389
clock (ライブラリ関数) 処理系定義の動作	466
clock.c	118
__close (DLIB ライブラリ関数)	114
clustering (コンパイラ変換)	196
disabling (--no_clustering)	241
__CLZ (組込み関数)	339

cmain (システム起動コード)	104
cmath (DLIB ヘッダファイル)	389
CMSIS の統合	181
codeseg (プラグマディレクティブ)	461
__code、ライブラリで使われるシンボル	391
.comment (ELF セクション)	418
complex (ライブラリヘッダファイル)	387
complex.h (ライブラリヘッダファイル)	386
--config (リンカオプション)	263
--config_def (リンカオプション)	263
configuration	
__low_level_init	108
リンカの設定ファイル。リンカ設定ファイルを参照	
const	
オブジェクトの宣言	300
非トップレベル	154
__constrange (), ライブラリで使われるシンボル	391
__construction_by_bitwise_copy_allowed、ライブラリで使われるシンボル	391
constseg (プラグマディレクティブ)	461
const_cast (キャスト演算子)	158
__CORE__ (定義済シンボル)	379
Cortex	
サポート	225
割込み関数に関する特別な考慮事項	65
cos (ライブラリ関数)	384
__cplusplus (定義済シンボル)	379
--cpp_init_routine (リンカオプション)	264
CPU	
コマンドラインでコンパイラを指定	225
コマンドラインでリンカを指定	264
--cpu (コンパイラオプション)	225
--cpu (リンカオプション)	264
CPU モード	49
__CPU_MODE__ (定義済シンボル)	379
--cpu_mode (コンパイラオプション)	226
--create (iarchive オプション)	440
csetjmp (DLIB ヘッダファイル)	389
csignal (DLIB ヘッダファイル)	389

cspy_support (プラグマディレクティブ)	461
CSTACK (ELF ブロック)	419
スタックも参照	
CSTACK (セクション), 例	169
.cstart (ELF セクション)	419
cstartup (システム起動コード)	
システム初期化のカスタマイズ	108
ソースファイル (DLIB)	104
csdarg (DLIB ヘッダファイル)	389
csdbool (DLIB ヘッダファイル)	389
csddef (DLIB ヘッダファイル)	389
csdio (DLIB ヘッダファイル)	389
csdlib (DLIB ヘッダファイル)	389
cstring (DLIB ヘッダファイル)	389
ctgmth (ライブラリヘッダファイル)	389
ctime (DLIB ヘッダファイル)	389
ctype.h (ライブラリヘッダファイル)	386
cwctype.h (ライブラリヘッダファイル)	389
C ヘッダファイル	385
C 言語、概要	147
C/C++ の呼出し規約。呼出し規約を参照	
C_INCLUDE (環境変数)	207
C-SPY	
C++ のデバッグサポート	164
システム終了用のインタフェース	108
デバッグサポートを含める	98
[ターミナル I/O] ウィンドウ、 デバッグサポートを含める	99
C/C++ のリンケージ	137
C++	
Embedded C++、拡張 Embedded C++ も参照	
サポート	33
ヘッダファイル	386
言語拡張	165
呼出し規約	135
静的メンバ変数	189
絶対アドレス	189
特殊な関数型	66
--c++ (コンパイラオプション)	227
C++ スタイルのコメント	147

C++ ヘッダファイル	387
C++ 用語	26
--c89 (コンパイラオプション)	224

D

-D (コンパイラオプション)	227
-d (iarchive オプション)	441
dataseg (プラグマディレクティブ)	461
data_alignment (プラグマディレクティブ)	318
.data_init (ELF セクション)	419
__data、ライブラリで使用するシンボル	391
__DATE__ (定義済シンボル)	379
date (ライブラリ関数)、サポートの設定	118
DC32 (アセンブラディレクティブ)	132
.debug (ELF セクション)	418
DebugMon_Handler (例外関数)	65
define block (リンカディレクティブ)	399
define overlay (リンカディレクティブ)	401
define symbol (リンカディレクティブ)	411
--define_symbol (リンカオプション)	265
define_type_info (プラグマディレクティブ)	461
--delete (iarchive オプション)	441
delete (キーワード)	55
--dependencies (コンパイラオプション)	228
--dependencies (リンカオプション)	265
deque (STL ヘッダファイル)	388
--diagnostics_tables (コンパイラオプション)	231
--diagnostics_tables (リンカオプション)	267
diag_default (プラグマディレクティブ)	318
--diag_error (コンパイラオプション)	229
--diag_error (リンカオプション)	266
--no_fragments (コンパイラオプション)	243
--no_fragments (リンカオプション)	276
diag_error (プラグマディレクティブ)	318
--diag_remark (コンパイラオプション)	229
--diag_remark (リンカオプション)	266
diag_remark (プラグマディレクティブ)	319
--diag_suppress (コンパイラオプション)	230

--diag_suppress (リンカオプション).....	267
diag_suppress (プラグマディレクティブ).....	319
--diag_warning (コンパイラオプション).....	230
--diag_warning (リンカオプション).....	267
diag_warning (プラグマディレクティブ).....	319
__disable_fiq (組込み関数).....	340
__disable_interrupt (組込み関数).....	340
__disable_irq (組込み関数).....	340
--discard_unused_publics (コンパイラオプション) ..	231
DLIB.....	50, 385
デバッグサポートを含める.....	97
ドキュメント.....	24
ランタイムライブラリ.....	89
リファレンス情報 オンラインヘルプシステムを	
参照.....	383
構成.....	109
設定.....	90, 231
--dlib_config (コンパイラオプション).....	231
DLib_Defaults.h (ライブラリ設定ファイル) ...	103, 109
__DLIB_FILE_DESCRIPTOR (構成シンボル)	114
__DMB (組込み関数).....	340
do not initialize (リンカディレクティブ).....	404
__DOUBLE__ (定義済シンボル).....	379
double (データ型).....	292
identifying size of (__DOUBLE__).....	379
do_not_instantiate (プラグマディレクティブ).....	461
__DSB (組込み関数).....	340

E

-e (コンパイラオプション).....	232
early_initialization (プラグマディレクティブ)	461
--ec++ (コンパイラオプション).....	233
--edit (isymexport オプション).....	441
--eec++ (コンパイラオプション).....	233
ELF ユーティリティ.....	423
Embedded C++.....	157
C++ との違い.....	157
概要.....	157

関数リンケージ.....	137
言語拡張.....	157
有効.....	233
Embedded C++ Technical Committee	26
__embedded_cplusplus (定義済シンボル).....	379
__enable_fiq (組込み関数).....	341
--enable_hardware_workaround	
(コンパイラオプション).....	233
__enable_interrupt (組込み関数).....	341
__enable_irq (組込み関数).....	341
--enable_multibytes (コンパイラオプション)	234
--endian (コンパイラオプション).....	234
--entry (リンカオプション).....	268
enums	
データ表現.....	287
前方宣言.....	153
--enum_is_int (コンパイラオプション)	235
EQU (アセンブラディレクティブ).....	252
ERANGE.....	463
error (プラグマディレクティブ).....	320
--error_limit (コンパイラオプション).....	235
--error_limit (リンカオプション).....	268
exception (ライブラリヘッダファイル).....	387
--exception_tables (リンカオプション).....	269
.exc.text (ELF セクション).....	420
_Exit (ライブラリ関数).....	108
exit (ライブラリ関数).....	107
処理系定義の動作.....	466
_exit (ライブラリ関数).....	107
_exit (ライブラリ関数).....	107
export キーワード、拡張 EC++ から除外	164
--export_builtin_config (リンカオプション).....	269
export (リンカディレクティブ).....	412
extended-selectors (リンカ設定ファイル).....	410
extern "C" リンケージ.....	163
--extract (iarchive オプション).....	442
--extra_init (リンカオプション).....	270

F

-f (iobjmanip オプション) 442
 -f (コンパイラオプション) 235
 -f (リンカオプション) 270
 fdopen、stdio.h 390
 fegettrapdisable 390
 fegettrapenable 390
 FENV_ACCESS、処理系定義の動作 458
 fenv.h (ライブラリヘッダファイル) 386, 388
 C の追加機能 390
 fgetpos (ライブラリ関数)、処理系定義の
 動作 465
 __FILE__ (定義済シンボル) 380
 filename
 オブジェクト実行可能イメージ 279
 パラメータとして指定 217
 検索手順 207
 fileno、stdio.h 390
 --fill (ielftool オプション) 443
 __fiq (拡張キーワード) 306
 FIQ_Handler (例外関数) 60
 float (データ型) 292
 float.h (ライブラリヘッダファイル) 386
 FLT_EVAL_METHOD、処理系定義の
 動作 457, 462, 467
 FLT_ROUNDS、処理系定義の
 動作 457, 466–467
 --force_exceptions (リンカオプション) 271
 --force_output (リンカオプション) 271
 for ループ、宣言 147
 --fpu (コンパイラオプション) 236
 FP_CONTRACT、処理系定義の動作 458
 free (ライブラリ関数) ヒープも参照 55
 fsetpos (ライブラリ関数)、処理系定義の
 動作 465
 fstream (ライブラリヘッダファイル) 387
 ftell (ライブラリ関数)、処理系定義の動作 465
 Full DLIB (ライブラリ構成) 109
 __func__ (定義済シンボル) 155, 380

__FUNCTION__ (定義済シンボル) 155, 380
 function (プラグマディレクティブ) 461
 functional (STL ヘッダファイル) 388

G

getenv (ライブラリ関数)、サポートの設定 117
 getw、stdio.h 390
 getzone (ライブラリ関数)、サポートの設定 118
 getzone.c 118
 __get_BASEPRI (組込み関数) 341
 __get_CONTROL (組込み関数) 341
 __get_CPSR (組込み関数) 342
 __get_FAULTMASK (組込み関数) 342
 __get_interrupt_state (組込み関数) 342
 __get_IPSR (組込み関数) 343
 __get_LR (組込み関数) 343
 __get_MSP (組込み関数) 343
 __get_PRIMASK (組込み関数) 343
 __get_PSP (組込み関数) 343
 __get_PSR (組込み関数) 344
 __get_SB (組込み関数) 344
 __get_SP (組込み関数) 344
 GRP_COMDAT、グループタイプ 432
 --guard_calls (コンパイラオプション) 237

H

Harbison, Samuel P. 25
 HardFault_Handler (例外関数) 65
 hash_map (STL ヘッダファイル) 388
 hash_set (STL ヘッダファイル) 388
 __has_constructor、ライブラリで
 使用されるシンボル 391
 __has_destructor、ライブラリで
 使用されるシンボル 391
 hdrstop (プラグマディレクティブ) 461
 --header_context (コンパイラオプション) 237
 HEAP (セクション) 171

hide (isymexport ディレクティブ)435

I

-I (コンパイラオプション) 237
iarbuild.exe (ユーティリティ) 102
iarchive 423
 コマンドの概要 424
 オプションの概要 425
IAR コマンドラインビルドユーティリティ 102
IAR システムズの技術サポート 213
IAR 言語の概要 33
__iar_program_start (ラベル) 105
__iar_ReportAssert (ライブラリ関数) 119
__IAR_SYSTEMS_ICC__ (定義済シンボル) 380
.iar.debug (ELF セクション) 418
.iar.dynexit (ELF セクション) 420
IA64 ABI 178
__ICCARM__ (定義済シンボル) 380
IDE
 ビルドツールの概要 31
 ライブラリのビルド 103
IEEE フォーマット、浮動小数点数値 292
ielfdump 428
 オプションの概要 429
ielftool 427
 オプションの概要 428
if (リンカディレクティブ) 414
--ihex (ielftool オプション) 443
ILINK 「リンカ」を参照してください。
ILINKARM_CMD_LINE (環境変数) 207
ILINK オプション リンカオプションを参照
--image_input (リンカオプション) 271
important_typedef (プラグマディレクティブ) 461
include_alias (プラグマディレクティブ) 320
include (リンカディレクティブ) 415
infinity (出力形式)、処理系定義の動作 465
initialize (リンカディレクティブ) 402
.init_array (セクション) 421

--inline (リンカオプション) 272
inline (プラグマディレクティブ) 321
instantiate (プラグマディレクティブ) 461
int (データ型) signed および unsigned 287
Intel hex 169
Intel IA64 ABI 178
__interwork (拡張キーワード) 306
--interwork (コンパイラオプション) 238
intptr_t (整数型) 295
intrinsics.h (ヘッダファイル) 333
inttypes.h (ライブラリヘッダファイル) 386
.intvec (セクション) 421
iobjmanip 430
 オプションの概要 431
iomanip (ライブラリヘッダファイル) 387
ios (ライブラリヘッダファイル) 387
iosfwd (ライブラリヘッダファイル) 387
iostream (ライブラリヘッダファイル) 387
__irq (拡張キーワード) 307
IRQ_Handler (例外関数) 60
IRQ_STACK (セクション) 421
__ISB (組込み関数) 344
iso646.h (ライブラリヘッダファイル) 386
istream (ライブラリヘッダファイル) 387
isymexport 433
 オプションの概要 434
iterator (STL ヘッダファイル) 388

J

Josuttis, Nicolai M. 25

K

--keep (リンカオプション) 272
keep_definition (プラグマディレクティブ) 461
keep (リンカディレクティブ) 405
Kernighan, Brian W. 25

L	
-l (コンパイラオプション)	238
スケルトンコードの作成	134
Labrosse, Jean J.	25
Lajoie, Jose.	25
language (プラグマディレクティブ)	322
__LDC (組込み関数)	344
__LDCL (組込み関数)	344
__LDCL_nidx (組込み関数)	345
__LDC_nidx (組込み関数)	345
__LDC2 (組込み関数)	344
__LDC2L (組込み関数)	344
__LDC2L_nidx (組込み関数)	345
__LDC2_nidx (組込み関数)	345
__LDREX (組込み関数)	345
--legacy (コンパイラオプション)	239
Lempel-Ziv-Welch アルゴリズム、 パッキングイニシャライザ	403
library_default_requirements (プラグマディレクティブ)	461
library_provides (プラグマディレクティブ)	461
library_requirement_override (プラグマディレクティブ)	461
lightbulb アイコン、本ガイドの	27
limits (ライブラリヘッダファイル)	387
limits.h (ライブラリヘッダファイル)	386
__LINE__ (定義済シンボル)	380
Lippman, Stanley B.	25
list (STL ヘッダファイル)	388
__LITTLE_ENDIAN__ (定義済シンボル)	380
__little_endian (拡張キーワード)	307
locale.h (ライブラリヘッダファイル)	386
location (プラグマディレクティブ)	188, 322
--lock_regs (コンパイラオプション)	239
--log_file (リンクオプション)	274
long double (データ型)	292
long float (データ型)、double の同義語	153
longjmp、使用の制限	385
__low_level_init	105
カスタマイズ	108
low_level_init.c	104
low_level_init.s	104
__lseek (ライブラリ関数)	114
lzw、イニシャライザのパッキングアルゴリズム	403
M	
--map (リンクオプション)	274
--macro_positions_in_diagnostics (コンパイラオプション)	240
main (関数)、処理系定義の動作	454
malloc (ライブラリ関数) ヒープも参照	55
--mangled_names_in_messages (リンクオプション)	274
Mann, Bernhard	25
map (STL ヘッダファイル)	388
math.h (ライブラリヘッダファイル)	386
MB_LEN_MAX、処理系定義の動作	467
__MCR (組込み関数)	346
memattr_HEAP (ELF セクション)	420
MemManage_Handler (例外関数)	65
memory C++ での解放	55
C++ での割当て	55
RAM、節約	198
グローバル / 静的変数で使用	53
スタック	54
節約	198
ヒープ	55
動的	55
非初期化	202
memory (STL ヘッダファイル)	388
memory (プラグマディレクティブ)	461
__memory_of、ライブラリで使用するシンボル	391
message (プラグマディレクティブ)	323
Meyers, Scott	25
--mfc (コンパイラオプション)	240

--migration_preprocessor_extensions (コンパイラオプション).....	241
--misrac (コンパイラオプション).....	220
--misrac_verbose (コンパイラオプション).....	220
--misrac_verbose (リンカオプション).....	260
--misrac1998 (コンパイラオプション).....	220
--misrac1998 (リンカオプション).....	260
--misrac2004 (コンパイラオプション).....	220
--misrac2004 (リンカオプション).....	260
MISRA-C、ドキュメント.....	24
module_name (プラグマディレクティブ).....	461
Motorola S-records.....	169
__MRC (組込み関数).....	346
mutable 属性、拡張 EC++.....	158, 165

N

NaN	
実装.....	294
処理系定義の動作.....	465
NDEBUG (プリプロセッサシンボル).....	381
Neon 組込み関数.....	339
__nested (拡張キーワード).....	307
new (キーワード).....	55
new (ライブラリヘッダファイル).....	387
NMI_Handler (例外関数).....	65
.noinit (ELF セクション).....	421
NOP (アセンブラ命令).....	347
__noreturn (拡張キーワード).....	308
Normal DLIB (ライブラリ構成).....	109
--no_clustering (コンパイラオプション).....	241
--no_code_motion (コンパイラオプション).....	241
--no_const_align (コンパイラオプション).....	242
--no_cse (コンパイラオプション).....	242
--no_dynamic_rtti_elimination (リンカオプション).....	275
--no_exceptions (コンパイラオプション).....	243
--no_exceptions (リンカオプション).....	276
__no_init (拡張キーワード).....	202, 308
--no_inline (コンパイラオプション).....	243

--no_library_search (リンカオプション).....	276
--no_locals (リンカオプション).....	277
--no_loop_align (コンパイラオプション).....	244
--no_mem_idioms (コンパイラオプション).....	244
__no_operation (組込み関数).....	347
--no_path_in_file_macros (コンパイラオプション).....	244
no_pch (プラグマディレクティブ).....	461
--no_range_reservations (リンカオプション).....	277
--no_remove (リンカオプション).....	277
--no_rtti (コンパイラオプション).....	245
--no_rw_dynamic_init (コンパイラオプション).....	245
--no_scheduling (コンパイラオプション).....	246
--no_static_destruction (コンパイラオプション).....	246
--no_strtab (ielfdump オプション).....	444
--no_system_include (コンパイラオプション).....	246
--no_tbaa (コンパイラオプション).....	247
--no_typedefs_in_diagnostics (コンパイラオプション).....	247
--no_unaligned_access (コンパイラオプション).....	248
--no_unroll (コンパイラオプション).....	248
--no_veneer (リンカオプション).....	278
--no_vfe (コンパイラオプション).....	278
--no_warnings (コンパイラオプション).....	248
--no_warnings (リンカオプション).....	278
--no_wrap_diagnostics (コンパイラオプション).....	249
--no_wrap_diagnostics (リンカオプション).....	279
NULL	
ポインタ定数、C 規格の緩和.....	153
処理系定義の動作.....	463
numeric (STL ヘッダファイル).....	388

O

-o (コンパイラオプション).....	249
-o (iarchive オプション).....	444
-o (ielfdump オプション).....	444
-o (コンパイラオプション).....	250
-o (リンカオプション).....	279
object_attribute (プラグマディレクティブ).....	202, 324

once (プラグマディレクティブ)	461
--only_stdout (コンパイラオプション)	250
--only_stdout (リンカオプション)	279
__open (ライブラリ関数)	114
optimize (プラグマディレクティブ)	324
Oram, Andy	25
ostream (ライブラリヘッダファイル)	387
output	
プリプロセッサ	251
リンカ用の指定	48
--output (iarchive オプション)	444
--output (ielfdump オプション)	444
--output (コンパイラオプション)	250
--output (リンカオプション)	279

P

pack (プラグマディレクティブ)	296, 325
packbits、イニシャライザのパッキングアルゴリズム	403
__packed (拡張キーワード)	308
packing、イニシャライザのアルゴリズム	403
__prel (拡張キーワード)	304
PendSV_Handler (例外関数)	65
--pi_veneer (リンカオプション)	280
__PKHBT (組込み関数)	347
__PKHTB (組込み関数)	347
place at (リンカディレクティブ)	406
place in (リンカディレクティブ)	407
--place_holder (リンカオプション)	280
pow (ライブラリ関数)	119
代替の実装	384
powXp (ライブラリ関数)	119
__Pragma (プリプロセッサ演算子)	147
--predef_macro (コンパイラオプション)	250
Prefetch_Handler (例外関数)	60
--preinclude (コンパイラオプション)	251
.preinit_array (セクション)	421
.prepreinit_array (セクション)	422

--preprocess (コンパイラオプション)	251
__PRETTY_FUNCTION__ (定義済シンボル)	380
printf (ライブラリ関数)	95
フォーマッタの選択	95
構成シンボル	112
処理系定義の動作	465
__printf_args (プラグマディレクティブ)	326
ptrdiff_t (整数型)	295
PUBLIC (アセンブラディレクティブ)	252
--public_equ (コンパイラオプション)	252
public_equ (プラグマディレクティブ)	461
putenv (ライブラリ関数)、DLIB には存在しない	117
putw, stdio.h	390

Q

__QADD (組込み関数)	348
__QADD8 (組込み関数)	348
__QADD16 (組込み関数)	348
__QASX (組込み関数)	348
QCCARM (環境変数)	207
__QCFlag (組込み関数)	349
__QDADD (組込み関数)	349
__QDOUBLE (組込み関数)	349
__QDSUB (組込み関数)	349
__QFlag (組込み関数)	350
__QSAX (組込み関数)	350
__QSUB (組込み関数)	350
__QSUB8 (組込み関数)	350
__QSUB16 (組込み関数)	351
queue (STL ヘッダファイル)	388

R

-r (iarchive オプション)	447
-r (コンパイラオプション)	228
raise (ライブラリ関数)、サポートの設定	118
raise.c	118

RAM

イニシャライザを ROM からコピー	46
コードの実行	84
メモリの節約	198
実行	58
領域の宣言の例	72
__ramfunc (拡張キーワード)	58, 309
--ram_reserve_ranges (isymexport オプション)	445
--raw (ielfdump オプション)	445
__read (ライブラリ関数)	114
カスタマイズ	110
realloc (ライブラリ関数)	55
ヒープも参照	
--redirect (リンカオプション)	281
reinterpret_cast (キャスト演算子)	158
.rel (ELF セクション)	418
.rela (ELF セクション)	418
--relaxed_fp (コンパイラオプション)	252
remove (ライブラリ関数)	114
処理系定義の動作	464
--remove_section (iobjmanip オプション)	446
remquo、規模	463
rename (isymexport ディレクティブ)	435
rename (ライブラリ関数)	114
処理系定義の動作	464
--rename_section (iobjmanip オプション)	446
--rename_symbol (iobjmanip オプション)	446
--replace (iarchive オプション)	447
required (プラグマディレクティブ)	326
--require_prototypes (コンパイラオプション)	253
--reserve_ranges (isymexport オプション)	447
__reset_QC_flag (組込み関数)	351
__reset_Q_flag (組込み関数)	351
__REV (組込み関数)	351
__REVSH (組込み関数)	351
Ritchie, Dennis M.	25
.rodata (ELF セクション)	422
ROM から RAM、コピー	84
__ROPI__ (定義済シンボル)	381

--ropi (コンパイラオプション)	253
rmno.h (ライブラリヘッダファイル)	386
rtmodel (プラグマディレクティブ)	327
rtmodel (アセンブラディレクティブ)	127
RTTI データ (動的)、イメージにインクルードする	275
rtti のサポート、STL から除外	158
--rwpi (コンパイラオプション)	254

S

-S (iarchive オプション)	449
-s (ielfdump オプション)	448
__SADD8 (組込み関数)	352
__SADD16 (組込み関数)	352
__SASX (組込み関数)	352
__sbrel (拡張キーワード)	304
scanf (ライブラリ関数)	
フォーマッタの選択 (DLI)	96
構成シンボル	112
処理系定義の動作	465
__scanf_args (プラグマディレクティブ)	328
scheduling (コンパイラ変換)	197
無効	246
--search (リンカオプション)	281
--section (ielfdump オプション)	448
__section_begin (拡張演算子)	152
__section_end (拡張演算子)	152
__section_size (拡張演算子)	152
section-selectors (リンカ設定ファイル)	408
segment (プラグマディレクティブ)	328
__SEL (組込み関数)	352
--semihosting (リンカオプション)	282
--separate_cluster_for_initialized_variables (コンパイラオプション)	255
set (STL ヘッダファイル)	388
setjmp.h (ライブラリヘッダファイル)	386
setlocale (ライブラリ関数)	116
__set_BASEPRI (組込み関数)	353

<code>_set_CONTROL</code> (組込み関数).....	353	<code>size_t</code> (整数型).....	295
<code>_set_CPSR</code> (組込み関数).....	353	<code>skeleton.s</code> (アセンブラソース出力).....	135
<code>_set_FAULTMASK</code> (組込み関数).....	353	<code>slist</code> (STL ヘッドファイル).....	388
<code>_set_interrupt_state</code> (組込み関数).....	353	<code>smallest</code> 、イニシャライザのパッキングアルゴリズム.....	403
<code>_set_LR</code> (組込み関数).....	354	<code>_SMLABB</code> (組込み関数).....	356
<code>_set_MSP</code> (組込み関数).....	354	<code>_SMLABT</code> (組込み関数).....	356
<code>_set_PRIMASK</code> (組込み関数).....	354	<code>_SMLAD</code> (組込み関数).....	357
<code>_set_PSP</code> (組込み関数).....	354	<code>_SMLADX</code> (組込み関数).....	357
<code>_set_SB</code> (組込み関数).....	354	<code>_SMLALBB</code> (組込み関数).....	357
<code>_set_SP</code> (組込み関数).....	354	<code>_SMLALBT</code> (組込み関数).....	357
SFR.....		<code>_SMLALD</code> (組込み関数).....	358
特殊機能レジスタへのアクセス.....	200	<code>_SMLALDX</code> (組込み関数).....	358
特殊機能レジスタを <code>extern</code> として宣言.....	189	<code>_SMLALTB</code> (組込み関数).....	358
<code>_SHADD8</code> (組込み関数).....	355	<code>_SMLALTT</code> (組込み関数).....	358
<code>_SHADD16</code> (組込み関数).....	355	<code>_SMLATB</code> (組込み関数).....	359
<code>_SHASX</code> (組込み関数).....	355	<code>_SMLATT</code> (組込み関数).....	359
<code>short</code> (データ型).....	287	<code>_SMLAWB</code> (組込み関数).....	359
<code>Show</code> (<code>isymexport</code> ディレクティブ).....	435	<code>_SMLAWT</code> (組込み関数).....	359
<code>_SHSAX</code> (組込み関数).....	355	<code>_SMLSD</code> (組込み関数).....	360
<code>.shstrtab</code> (ELF セクション).....	418	<code>_SMLSDX</code> (組込み関数).....	360
<code>_SHSUB8</code> (組込み関数).....	356	<code>_SMLSLD</code> (組込み関数).....	360
<code>_SHSUB16</code> (組込み関数).....	356	<code>_SMLSLDX</code> (組込み関数).....	360
<code>signal</code> (ライブラリ関数).....		<code>_SMMLA</code> (組込み関数).....	361
サポートの設定.....	118	<code>_SMMLAR</code> (組込み関数).....	361
処理系定義の動作.....	463	<code>_SMMLS</code> (組込み関数).....	361
<code>signal.c</code>	118	<code>_SMMLSR</code> (組込み関数).....	361
<code>signed char</code> (データ型).....	287-288	<code>_SMMUL</code> (組込み関数).....	362
指定.....	224	<code>_SMMULR</code> (組込み関数).....	362
<code>signed int</code> (データ型).....	287	<code>_SMUAD</code> (組込み関数).....	362
<code>signed long long</code> (データ型).....	287	<code>_SMUADX</code> (組込み関数).....	362
<code>signed long</code> (データ型).....	287	<code>_SMUL</code> (組込み関数).....	363
<code>signed short</code> (データ型).....	287	<code>_SMULBB</code> (組込み関数).....	363
<code>--silent</code> (<code>iarchive</code> オプション).....	449	<code>_SMULBT</code> (組込み関数).....	363
<code>--silent</code> (<code>ielftool</code> オプション).....	449	<code>_SMULTB</code> (組込み関数).....	363
<code>--silent</code> (コンパイラオプション).....	255	<code>_SMULTT</code> (組込み関数).....	364
<code>--silent</code> (リンカオプション).....	282	<code>_SMULWB</code> (組込み関数).....	364
<code>--simple</code> (<code>ielftool</code> オプション).....	449	<code>_SMULWT</code> (組込み関数).....	364
<code>sin</code> (ライブラリ関数).....	384	<code>_SMUSD</code> (組込み関数).....	364
<code>sizeof</code> 、プリプロセッサ式での使用.....	241	<code>_SMUSDX</code> (組込み関数).....	365

sprintf (ライブラリ関数).....	95	stdio.h (ライブラリヘッダファイル).....	386
フォーマッタの選択.....	95	stdio.h、C の追加機能.....	390
--srec (ielftool オプション).....	450	stdout.....	114, 250, 279
--srec-len (ielftool オプション).....	450	処理系定義の動作.....	463
--srec-s3only (ielftool オプション).....	450	std 名前空間、EC++、拡張 EC++ から除外.....	165
__SSAT (組込み関数).....	365	Steele, Guy L.....	25
__SSAT16 (組込み関数).....	365	STL.....	164
__SSAX (組込み関数).....	365	strcasecmp、string.h.....	390
sscanf (ライブラリ関数)、フォーマッタの 選択 (DLIB).....	96	strdup、string.h.....	390
sstream (ライブラリヘッダファイル).....	387	streambuf (ライブラリヘッダファイル).....	387
__SSUB8 (組込み関数).....	366	strerror (ライブラリ関数)、処理系定義の動作.....	469
__SSUB16 (組込み関数).....	366	__STREX (組込み関数).....	367
stack (STL ヘッダファイル).....	388	--strict (コンパイラオプション).....	256
__stackless (拡張キーワード).....	310	string (ライブラリヘッダファイル).....	387
static clustering (コンパイラ変換).....	196	string.h (ライブラリヘッダファイル).....	386
static_assert ().....	151	string.h、C の追加機能.....	390
static_cast (キャスト演算子).....	158	--strip (ielftool オプション).....	451
__STC (組込み関数).....	366	--strip (iobjmanip オプション).....	451
__STCL (組込み関数).....	366	--strip (リンカオプション).....	283
__STCL_nidx (組込み関数).....	367	strncasecmp、string.h.....	390
__STC_nidx (組込み関数).....	367	strnlen、string.h.....	390
__STC2 (組込み関数).....	366	Stroustrup, Bjarne.....	25
__STC2L (組込み関数).....	366	strstream (ライブラリヘッダファイル).....	387
__STC2L_nidx (組込み関数).....	367	--strtab (ielfdump オプション).....	449
__STC2_nidx (組込み関数).....	367	.strtab (ELF セクション).....	418
stdbool.h (ライブラリヘッダファイル).....	287, 386	Sutter, Herb.....	25
__STDC__ (定義済シンボル).....	381	SVC #immed、ソフトウェア割込み.....	63
STDC CX_LIMITED_RANGE (プラグマディレクティブ).....	329	SVC_Handler (例外関数).....	65
STDC FENV_ACCESS (プラグマディレクティブ).....	329	__swi (拡張キーワード).....	310
STDC FP_CONTRACT (プラグマディレクティブ).....	330	SWI_Handler (例外関数).....	60
__STDC_VERSION__ (定義済シンボル).....	381	swi_number (プラグマディレクティブ).....	330
stddef.h (ライブラリヘッダファイル).....	288, 386	SWO、stdout/stderr の出力.....	98
stderr.....	114, 250, 279	__SWP (組込み関数).....	367
stdexcept (ライブラリヘッダファイル).....	387	__SWPB (組込み関数).....	368
stdin.....	114	__SXTAB (組込み関数).....	368
stdint.h (ライブラリヘッダファイル).....	386, 389	__SXTAB16 (組込み関数).....	368
		__SXTAH (組込み関数).....	368
		__SXTB16 (組込み関数).....	369

symbols

ブリプロセッサ、定義	227, 265
参照の変更	281
出力に含める	327
定義済シンボルの概要	35
匿名、作成	147
--symbols (iarchive オプション)	451
.symtab (ELF セクション)	418
system (ライブラリ関数)、サポートの設定	117
system_include (プラグマディレクティブ)	461
--system_include_dir (コンパイラオプション)	256
SysTick_Handler (例外関数)	65

T

-t (iarchive オプション)	452
tan (ライブラリ関数)	384
__task (拡張キーワード)	312
.text (ELF セクション)	422
tgmath.h (ライブラリヘッダファイル)	386
this (ポインタ)	135
__thumb (拡張キーワード)	312
--thumb (コンパイラオプション)	256
Thumb、CPU モード	49
__TIME__ (定義済シンボル)	381
time zone (ライブラリ関数)、処理系定義の動作	466
time.c	118
time.h (ライブラリヘッダファイル)	386
time.h, C の追加機能	390
time32 (ライブラリ関数)、サポートの設定	118
time64 (ライブラリ関数)、サポートの設定	118
--toc (iarchive オプション)	452
typedefs	
ブリプロセッサ式での使用	241
繰返し	153
診断から除外	247
typeinfo (ライブラリヘッダファイル)	387
type_attribute (プラグマディレクティブ)	331

U

__UADD8 (組込み関数)	369
__UADD16 (組込み関数)	369
__UASX (組込み関数)	369
uchar.h (ライブラリヘッダファイル)	386
__UHADD8 (組込み関数)	370
__UHADD16 (組込み関数)	370
__UHASX (組込み関数)	370
__UHSAX (組込み関数)	370
__UHSUB8 (組込み関数)	371
__UHSUB16 (組込み関数)	371
uintptr_t (整数型)	295
__UMAAL (組込み関数)	371
Undefined_Handler (例外関数)	60
__ungetchar, stdio.h	390
unsigned char (データ型)	287-288
signed char に変更	224
unsigned int (データ型)	287
unsigned long long (データ型)	287
unsigned long (データ型)	287
unsigned short (データ型)	287
__UQADD8 (組込み関数)	371
__UQADD16 (組込み関数)	372
__UQASX (組込み関数)	372
__UQSAX (組込み関数)	372
__UQSUB8 (組込み関数)	372
__UQSUB16 (組込み関数)	373
__USADA8 (組込み関数)	373
__USAD8 (UMAAL)	373
UsageFault_Handler (例外関数)	65
__USAT (組込み関数)	373
__USAT16 (組込み関数)	374
__USAX (組込み関数)	374
--use_c++_inline (コンパイラオプション)	257
--use_unix_directory_separators (コンパイラオブジェクト)	257
__USUB8 (組込み関数)	374
__USUB16 (組込み関数)	374

utility (STL ヘッダファイル)	388
__UXTAB (組込み関数)	375
__UXTAB16 (組込み関数)	375
__UXTAH (組込み関数)	375
__UXTB16 (組込み関数)	375

V

-V (iarchive オプション)	452
valaway (ライブラリヘッダファイル)	388
vector (STL ヘッダファイル)	388
vector (プラグマディレクティブ)	461
__vector_table、ベクタテーブルを保持する配列	65
__VER__ (定義済シンボル)	381
--verbose (iarchive オプション)	452
--verbose (ielftool オプション)	452
--vfe (コンパイラオプション)	283
VFP	236
--vla (コンパイラオプション)	257
void、ポインタ	153
volatile	
const、オブジェクトの宣言	299
アクセス規則	299
オブジェクトの宣言	298
同時にアクセスされる変数の保護	200

W

#warning message (プリプロセッサ拡張)	382
warnings (プラグマディレクティブ)	461
--warnings_affect_exit_code (コンパイラオプション)	209, 258
--warnings_affect_exit_code (リンカオプション)	284
--warnings_are_errors (コンパイラオプション)	258
--warnings_are_errors (リンカオプション)	284
wchar_t (データ型)、AC でのサポートの追加	288
wchar.h (ライブラリヘッダファイル)	386, 389
wctype.h (ライブラリヘッダファイル)	386
__weak (拡張キーワード)	312

weak (プラグマディレクティブ)	331
Web サイト、推奨	26
__write (ライブラリ関数)	114
カスタマイズ	110
__write_array, in stdio.h	390
__write_buffered (DLIB ライブラリ関数)	100

X

-x (iarchive オプション)	442
xreportassert.c	119

Z

zeros、イニシャライザのパッキングアルゴリズム	403
---------------------------------	-----

あ

アサート関数	119
出力に含める	381
処理系定義の動作	462
アセンブラコード	
C からの呼出し	133
C++ から呼び出す	135
インライン挿入	131
アセンブラディレクティブ	
インラインアセンブラコードでの使用	132
呼出しフレーム情報	143
アセンブララベル、public 化 (--public_equ)	252
アセンブラリストファイル、生成	238
アセンブラ言語インタフェース	129
呼出し規約。アセンブラコードを参照	
アセンブラ出力ファイル	135
アセンブラ命令	
インライン挿入	131
ソフトウェア割込み	63
アセンブラ、インライン	148

アプリケーション	
ビルド、概要	47
起動と終了 (DLIB)	104
実行、概要	43
アラインメント	285
インラインアセンブラの制限	132
オブジェクト (<code>__ALIGNOF</code>)	151
データ型	286
厳密に設定 (<code>#pragma data_alignment</code>)	318
構造体 (<code>#pragma pack</code>)	326
構造体、問題の原因	185
アンダーフローエラー、処理系定義の動作	463
アンダーフローの <code>errno</code> 値、処理系定義の動作	465
アーキテクチャ	
ARM	53
詳細情報	21

い

イニシャライザ、静的	153
インクルードファイル	
ソースファイルより前にインクルード	251
指定	207
インストール先ディレクトリ	26
インターナルエラー	213
インラインアセンブラ	131, 148
アセンブラ言語インタフェースも参照	
回避	198
インライン関数	147
コンパイラ	195

う

エスケープシーケンス、処理系定義の動作	455
エラーメッセージ	212
range	87
コンパイラ用の分類	229
リンカ用の分類	266
分類	243, 276

エラーリターンコード	209
エリアエラー、処理系定義の動作	462
エントリラベル、プログラム	105

お

オブジェクトファイルの検索パス (<code>--search</code>)	281
オブジェクトファイル、リンカの検索パス (<code>--search</code>)	281
オブジェクト属性	303
オプションパラメータ	215
オプション、 <code>iarchive iarchive</code> オプションを参照	
オプション、 <code>ielddump. ielddump</code> のオプションを参照	
オプション、 <code>ieftool. ieftool</code> のオプションを参照	
オプション、 <code>iobjmanip iobjmanip</code> オプションを参照	
オプション、 <code>isymexport isymexport</code> オプションを参照	
オプション、コンパイラ コンパイラオプションを参照	
オプション、リンカ。リンカオプションを参照	
オーバーヘッド、削減	194–195

か

ガイドラインの確認	21
ガイドライン、確認	21
カーニハン & リッチー関数宣言	199
不許可	253

き

キャスト	
ポインタと整数	295
整数へのポインタ、言語拡張	153
キャスト演算子	
Embedded C++ から削除	158
拡張 EC++	158, 165
キャラクタベース I/O	
DLIB	110
キーワード	301
拡張、概要	35

目次

コア

選択	49
特定	379
このガイドで使用されている規則	26
コマンドプロンプトアイコン、本ガイド	27
コマンドラインオプション	
コンパイラオプションも参照	
コンパイラ呼出し構文のパート	205
リンカオプションも参照	
リンカ呼出し構文のパート	205
受渡し	206
表記規則	27
コマンド、iarchive	
コメント	
C++ スタイル、C コードで使用	147
プリプロセッサディレクティブ後	154
コンパイラ	
環境変数	207
呼出し構文	205
出力元	208
コンパイラオブジェクトファイル	40
コンパイラからの出力	208
(--debug, -r) にデバッグ情報を含める	228
コンパイラオプション	215
コンパイラへの受渡し	206
ファイル (-f) からの読取り	235
パラメータの指定	217
概要	219
構文	215
スケルトンコードの作成	134
命令スケジューリング	197
--warnings_affect_exit_code	209
コンパイラでのハードウェアサポート	89
コンパイラのバージョン番号	381
コンパイラプラットフォーム、特定	380
コンパイラリスト、生成 (-l)	238
コンパイラ最適化レベル	192

コンパイラ変換	191
コンパイル	
コマンドラインから	47
構文	205
コンパイル日	
正確な時刻 (__TIME__)	381
(__DATE__) の特定	379
コンピュータスタイル、表記規則	26
コード	
ARM および Thumb、概要	57
実行の割込み	59
--code (ielfdump オプション)	440
コードの相互作用	49
コード移動 (コンパイラ変換)	195
無効化 (--no_code_motion)	241

さ

サポート、技術	213
---------	-----

し

シグナル、処理系定義の動作	454
システム起動時	454
システムの終了「システムの終了」を参照	
システム関数、処理系定義の動作	454, 466
システム起動	
DLIB	104
カスタマイズ	108
初期化フェーズ	44
システム終了	
C-SPY のインタフェース	108
DLIB	107
シンボル名、プリプロセッサ式での使用	241

す

スカラ以外のパラメータ、回避	198
スクラッチレジスタ	138

スケルトンコード、アセンブラ	
言語インタフェース用に作成.....	133
スタック	54
exception	170
エリアの節約	198
サイズ	169
レイアウト	139
関数リターン後のクリーン	141
使用の利点、問題点	54
内部データ	419
内容	54
スタックパラメータ	138-139
スタックポインタ	54
ステアリングファイル、isymexport への入力	434
ストリーム	
Embedded C++ でサポート	158
処理系定義の動作	454

せ

セクション	417
概要	417
指定 (--section)	254
宣言 (#pragma section)	328
定義	71
セミホスティング、概要	99

そ

ソフトウェア割込み	63
ソースファイル、すべての参照先のリスト	237

た

ターミナル I/O、デバッグのランタイムイン	
タフェース	98
ターミナル出力、高速化	100

ち

チェックサム	
C-SPY でのシンボルの表示フォーマット	177
計算	173
--checksum (ielftool オプション)	438

つ

ツールアイコン、本ガイド	27
--------------------	----

て

ディレクティブ	
プラグマ	35, 315
リンカ	393
ディレクトリ、パラメータとして指定	217
デストラクタおよび割込み、使用	163
デバイス記述ファイル、C-SPY 用に事前定義	34
--debug (コンパイラオプション)	228
デバッグ情報、オブジェクトファイルに含める	228
テンプレートのサポート	
Embedded C++ から削除	157
拡張 EC++	158, 164
データ	
さまざまな記憶方法	53
レジスタへの配置	190
記憶	53
配置	187, 254, 285, 417
絶対アドレス	188
配置、extern として宣言	189
表現	285
.data (ELF セクション)	419
データブロック (呼出しフレーム情報)	143
データポインタ	295
データ型	287
C++	300
整数型	287
浮動小数点数	292

と

ドキュメント、ガイドの概要.....23

ね

ネイティブ環境
処理系定義の動作468
ネスト割込み62

は

バイトオーダー50
 指定 (--endian)234
 特定380
バイト内のビット、処理系定義の動作.....455
バイナリストリーム463
バックトレース情報、呼出しフレーム情報も参照
バック構造体型296
バッチファイル
 エラーリターンコード209
 コマンドファイルからライブラリをビルド
 (ファイル提供なし)102
バラメータ
 スカラ以外、回避198
 スタック138-139
 ファイルまたはディレクトリを指定する
 場合の規則217
 レジスタ138-139
 隠し139
 関数138
 指定217
 表記規則26
バージョン番号
 コンパイラ381
 本ガイド2

ひ

ビッグエンディアン (バイトオーダー).....50
ビットフィールド
 データ表現288
 ヒント184
 処理系定義の動作459
 非標準型151
ビット否定199
ヒント
 円滑なコードの生成197
 効率的なデータ型の使用183
 処理系定義の動作458
ヒント、プログラミング197
ヒープ
 データ記憶53
 動的メモリ55
ヒープサイズ
 デフォルトの変更82
 標準 I/O171
ヒープ (サイズがゼロ)、処理系定義の動作.....465

ふ

ファイルのバッファ処理、処理系定義の動作.....464
ファイルパス、#include ファイル用パスの指定.....237
ファイル位置、処理系定義の動作.....463
ファイル依存関係、追跡.....228
ファイル名
 デバイス記述ファイルの拡張子34
 ヘッダファイルの拡張子34
ファイル名 (有効)、処理系定義の動作.....464
ファイル、処理系定義の動作
 マルチバイト文字464
 一時ファイルの処理465
 開く464
ファイル (ゼロ長)、処理系定義の動作.....464
フォーマット
 標準 IEEE (浮動小数点数)292

浮動小数点数値	292
プラグマディレクティブ	35
概要	315
pack	296, 325
絶対配置データ	188
認識されたすべての一覧	460
プリプロセッサ	
output	251
演算子 (<code>_Pragma</code>)	147
概要	377
プリプロセッサシンボル	378
定義	227, 265
プリプロセッサディレクティブ	
終了後のコメント	154
処理系定義の動作	460
#pragma	315
プリプロセッサ拡張	
互換性	241
__VA_ARGS__	147
#warning message	382
プログラミングのヒント	197
プログラムエントリラベル	105
プログラム終了、処理系定義の動作	454
プロジェクト	
ライブラリのためのセットアップ	103
基本設定	48
プロセッサ構成	48
プロセッサ処理	
アクセス	129
低レベル	149, 333
プロトタイプ、強制	253

へ

ベクタ浮動小数点ユニット	236
ヘッダファイル	
C	385
C++	386–387
ライブラリ	383

特殊機能レジスタ	200
bool での <code>stdbool.h</code> のインクルード	287
DLib_Defaults.h	103, 109
wchar_t での <code>stddef.h</code> のインクルード	288
ヘッダ名、処理系定義の動作	460
ベニア	86

ほ

ポインタ	
キャスト	295
データ	295
関数	294
処理系定義の動作	458
ポインタ型	294
混在	154
ポリモフィズム、Embedded C++	157

ま

マクロ	
ERANGE (in <code>errno.h</code>)	463
NULL、処理系定義の動作	463
アサートのインクルード	381
可変数引数	147
#pragma optimize への埋め込み	325
#pragma ディレクティブで代用	149
マップファイル、生成	274
マルチバイト文字のサポート	234
マルチバイト文字、処理系定義の動作	455, 467

め

メッセージ	
強制	323
無効	255, 282
メモリマップ、リンカからの出力	210
メモリレイアウト、ARM	53
メモリ管理、型安全	157

メンバ（リンカ設定ファイル）.....	414
---------------------	-----

も

モジュールの互換性	125
rtmodel.....	327
モジュール、概要	68
モード変更、処理系定義の動作.....	465

ゆ

ユーティリティ（ELF）	423
--------------------	-----

ら

ライブラリオブジェクトファイル.....	384
ライブラリオプション、設定.....	52
ライブラリドキュメント.....	383
ライブラリファイルの検索パス（--search）.....	281
ライブラリファイル、検索パス（--search）.....	281
ライブラリプロジェクトテンプレート.....	51
使用.....	103
ライブラリヘッダファイル.....	383
ライブラリモジュール	
オーバーライド.....	102
概要.....	68
ライブラリ関数	383
一覧、DLIB	385
ライブラリ機能、Embedded C++ から削除.....	158
ライブラリ設定ファイル	
DLIB	109
DLib_Defaults.h	103, 109
指定.....	231
修正.....	103
ライブラリ、ビルド済	91
ラベル	154
アセンブラ、public 化.....	252
__iar_program_start.....	105
ランタイムの型情報、Embedded C++ から削除.....	158

ランタイムモデル属性	125
ランタイムモデル定義	327
ランタイムライブラリ	
DLIB	89
オプション設定	52
設定（DLIB）.....	90
ランタイムライブラリ（DLIB）	
概要.....	383
IDE から設定	51
コマンドラインからの設定	51
システム起動コードのカスタマイズ	108
ビルド済.....	91
ファイル名の構文.....	92
モジュールのオーバーライド	102
リビルドなしでのカスタマイズ	94

り

リエントラント性（DLIB）.....	384
リスト、生成.....	238
リターン値、関数	140
リテラル、複合	147
リトルエンディアン（バイトオーダー）.....	50
--remarks（コンパイラオプション）	253
--remarks（リンカオプション）	281
リマーク（診断メッセージ）.....	212
コンパイラで有効化	253
コンパイラ用の分類	229
リンカで有効化	281
リンカ用の分類	266
リレー、ベニアを参照	86
リンカ	67
出力元.....	210
リンカオブジェクト実行可能イメージ	
(-o) のファイル名の指定	279
リンカオプション	259
ファイル（-f）からの読取り	270
概要.....	259
表記規則.....	26

リンカ設定ファイル	
コードおよびデータの配置	71
概要	393
詳細	393
選択	77
リンク	
コマンドラインから	47
プロセス	41, 69
概要	67
リンケージ、C/C++	137

る

ルーチン、時間が重要	129, 149, 333
__root (拡張キーワード)	310
ループ展開 (コンパイラ変換)	194
無効	248
ループ不変式	195

れ

レジスタ	
アセンブラレベルルーチン	136
スラッチ	138
パラメータへの割当て	139
呼出し先保存、スタックに格納	54
保護	138
レジスタキーワード、処理系定義の動作	458
レジスタパラメータ	138-139

ろ

--log (リンカオプション)	273
ロケール	
サポート	115
サポートの削除	116
ライブラリでのサポートを追加	116
ライブラリヘッダファイル	387
実行中のロケール変更	116

処理系定義の動作	456, 467
ローカル変数、自動変数を参照	

わ

ワーニング	212
コンパイラでの終了コード	258
コンパイラで分類	230
コンパイラで無効化	248
リンカでの終了コード	284
リンカで分類	267
リンカで無効化	278

記号

__AEABI_PORTABILITY_LEVEL	
(プロセッサシンボル)	180
__AEABI_PORTABLE (プロセッサシンボル)	180
__Exit (ライブラリ関数)	108
__exit (ライブラリ関数)	107
__low_level_init、カスタマイズ	108
__AAPCS_VFP__ (定義済シンボル)	378
__AAPCS__ (定義済シンボル)	378
__absolute (拡張キーワード)	305
__ALIGNOF__ (演算子)	151
__ARMVFPV2__ (定義済シンボル)	378
__ARMVFPV3__ (定義済シンボル)	378
__ARMVFPV4__ (定義済シンボル)	378
__ARMVFP_D6__ (定義済シンボル)	378
__ARMVFP_FP16__ (定義済シンボル)	378
__ARMVFP_SP__ (定義済シンボル)	379
__ARMVFP__ (定義済シンボル)	378
__ARM_ADVANCED_SIMD__ (定義済シンボル)	378
__ARM_MEDIA__ (定義済シンボル)	378
__ARM_PROFILE_M__ (定義済シンボル)	378
__arm (拡張キーワード)	305
__ARM4M__ (定義済シンボル)	379
__ARM4TM__ (定義済シンボル)	379
__ARM5E__ (定義済シンボル)	379

<u>ARM5</u> (定義済シンボル).....	379	<u>FILE</u> (定義済シンボル).....	380
<u>ARM6M</u> (定義済シンボル).....	379	<u>fiq</u> (拡張キーワード).....	306
<u>ARM6SM</u> (定義済シンボル).....	379	<u>FUNCTION</u> (定義済シンボル).....	155, 380
<u>ARM6</u> (定義済シンボル).....	379	<u>func</u> (定義済シンボル).....	155, 380
<u>ARM7A</u> (定義済シンボル).....	379	<u>gets、stdio.h</u>	390
<u>ARM7EM</u> (定義済シンボル).....	379	<u>get_BASEPRI</u> (組込み関数).....	341
<u>ARM7M</u> (定義済シンボル).....	379	<u>get_CONTROL</u> (組込み関数).....	341
<u>ARM7R</u> (定義済シンボル).....	379	<u>get_CPSR</u> (組込み関数).....	342
<u>asm</u> (言語拡張).....	148	<u>get_FAULTMASK</u> (組込み関数).....	342
<u>assignment_by_bitwise_copy_allowed</u> 、 ライブラリで使用されるシンボル.....	391	<u>get_interrupt_state</u> (組込み関数).....	342
<u>BASE_FILE</u> (定義済シンボル).....	379	<u>get_IPSR</u> (組込み関数).....	343
<u>big_endian</u> (拡張キーワード).....	306	<u>get_LR</u> (組込み関数).....	343
<u>BUILD_NUMBER</u> (定義済シンボル).....	379	<u>get_MSP</u> (組込み関数).....	343
<u>close</u> (ライブラリ関数).....	114	<u>get_PRIMASK</u> (組込み関数).....	343
<u>CLZ</u> (組込み関数).....	339	<u>get_PSP</u> (組込み関数).....	343
<u>code</u> 、ライブラリで使用されるシンボル.....	391	<u>get_PSR</u> (組込み関数).....	344
<u>constrange ()</u> 、ライブラリで使用され るシンボル.....	391	<u>get_SB</u> (組込み関数).....	344
<u>construction_by_bitwise_copy_allowed</u> 、 ライブラリで使用される シンボル.....	391	<u>get_SP</u> (組込み関数).....	344
<u>CORE</u> (定義済シンボル).....	379	<u>has_constructor</u> 、ライブラリで使用さ れるシンボル.....	391
<u>cplusplus</u> (定義済シンボル).....	379	<u>has_destructor</u> 、ライブラリで使用さ れるシンボル.....	391
<u>CPU_MODE</u> (定義済シンボル).....	379	<u>iar_maximum_atexit_calls</u>	82
<u>data</u> 、ライブラリで使用されるシンボル.....	391	<u>iar_program_start</u> (ラベル).....	105
<u>DATE</u> (定義済シンボル).....	379	<u>iar_ReportAssert</u> (ライブラリ関数).....	119
<u>disable_fiq</u> (組込み関数).....	340	<u>IAR_SYSTEMS_ICC</u> (定義済シンボル).....	380
<u>disable_interrupt</u> (組込み関数).....	340	<u>ICCARM</u> (定義済シンボル).....	380
<u>disable_irq</u> (組込み関数).....	340	<u>interwork</u> (拡張キーワード).....	306
<u>DLIB_FILE_DESCRIPTOR</u> (構成シンボル).....	114	<u>intrinsic</u> (拡張キーワード).....	304, 306
<u>DLIB_PERTHREAD</u> (ELF セクション).....	420	<u>irq</u> (拡張キーワード).....	307
<u>DMB</u> (組込み関数).....	340	<u>ISB</u> (組込み関数).....	344
<u>DOUBLE</u> (定義済シンボル).....	379	<u>LDC</u> (組込み関数).....	344
<u>DSB</u> (組込み関数).....	340	<u>LDCL</u> (組込み関数).....	344
<u>embedded_cplusplus</u> (定義済シンボル).....	379	<u>LDCL_noidx</u> (組込み関数).....	345
<u>enable_fiq</u> (組込み関数).....	341	<u>LDC_noidx</u> (組込み関数).....	345
<u>enable_interrupt</u> (組込み関数).....	341	<u>LDC2</u> (組込み関数).....	344
<u>enable_irq</u> (組込み関数).....	341	<u>LDC2L</u> (組込み関数).....	344
<u>exit</u> (ライブラリ関数).....	107	<u>LDC2L_noidx</u> (組込み関数).....	345
		<u>LDC2_noidx</u> (組込み関数).....	345
		<u>LDREX</u> (組込み関数).....	345

<code>__LINE__</code> (定義済シンボル)	380	<code>__REV</code> (組込み関数)	351
<code>__little_endian</code> (拡張キーワード)	307	<code>__REVSH</code> (組込み関数)	351
<code>__LITTLE_ENDIAN__</code> (定義済シンボル)	380	<code>__root</code> (拡張キーワード)	310
<code>__low_level_init</code>	105	<code>__ROPI</code> (定義済シンボル)	381
初期化フェーズ	44	<code>__SADD8</code> (組込み関数)	352
<code>__lseek</code> (ライブラリ関数)	114	<code>__SADD16</code> (組込み関数)	352
<code>__MCR</code> (組込み関数)	346	<code>__SASX</code> (組込み関数)	352
<code>__memory_of</code> 、ライブラリで使用され るシンボル	391	<code>__sbrel</code> (拡張キーワード)	304
<code>__MRC</code> (組込み関数)	346	<code>__scanf_args</code> (プラグマディレクティブ)	328
<code>__nested</code> (拡張キーワード)	307	<code>__section_begin</code> (拡張演算子)	152
<code>__noreturn</code> (拡張キーワード)	308	<code>__section_end</code> (拡張演算子)	152
<code>__no_init</code> (拡張キーワード)	202, 308	<code>__section_size</code> (拡張演算子)	152
<code>__no_operation</code> (組込み関数)	347	<code>__SEL</code> (組込み関数)	352
<code>__open</code> (ライブラリ関数)	114	<code>__set_BASEPRI</code> (組込み関数)	353
<code>__packed</code> (拡張キーワード)	308	<code>__set_CONTROL</code> (組込み関数)	353
<code>__pcrel</code> (拡張キーワード)	304	<code>__set_CPSR</code> (組込み関数)	353
<code>__PKHBT</code> (組込み関数)	347	<code>__set_FAULTMASK</code> (組込み関数)	353
<code>__PKHTB</code> (組込み関数)	347	<code>__set_interrupt_state</code> (組込み関数)	353
<code>__PRETTY_FUNCTION__</code> (定義済シンボル)	380	<code>__set_LR</code> (組込み関数)	354
<code>__printf_args</code> (プラグマディレクティブ)	326	<code>__set_MSP</code> (組込み関数)	354
<code>__QADD</code> (組込み関数)	348	<code>__set_PRIMASK</code> (組込み関数)	354
<code>__QADD8</code> (組込み関数)	348	<code>__set_PSP</code> (組込み関数)	354
<code>__QADD16</code> (組込み関数)	348	<code>__set_SB</code> (組込み関数)	354
<code>__QASX</code> (組込み関数)	348	<code>__set_SP</code> (組込み関数)	354
<code>__QCFlag</code> (組込み関数)	349	<code>__SHADD8</code> (組込み関数)	355
<code>__QDADD</code> (組込み関数)	349	<code>__SHADD16</code> (組込み関数)	355
<code>__QDOUBLE</code> (組込み関数)	349	<code>__SHASX</code> (組込み関数)	355
<code>__QDSUB</code> (組込み関数)	349	<code>__SHSAX</code> (組込み関数)	355
<code>__QFlag</code> (組込み関数)	350	<code>__SHSUB8</code> (組込み関数)	356
<code>__QSAX</code> (組込み関数)	350	<code>__SHSUB16</code> (組込み関数)	356
<code>__QSUB</code> (組込み関数)	350	<code>__SMLABB</code> (組込み関数)	356
<code>__QSUB8</code> (組込み関数)	350	<code>__SMLABT</code> (組込み関数)	356
<code>__QSUB16</code> (組込み関数)	351	<code>__SMLAD</code> (組込み関数)	357
<code>__ramfunc</code> (拡張キーワード)	309	<code>__SMLADX</code> (組込み関数)	357
RAM での実行	58	<code>__SMLALBB</code> (組込み関数)	357
<code>__read</code> (ライブラリ関数)	114	<code>__SMLALBT</code> (組込み関数)	357
カスタマイズ	110	<code>__SMLALD</code> (組込み関数)	358
<code>__reset_QC_flag</code> (組込み関数)	351	<code>__SMLALDX</code> (組込み関数)	358
<code>__reset_Q_flag</code> (組込み関数)	351	<code>__SMLALTB</code> (組込み関数)	358

__SMLALTT (組込み関数)	358	__STC2_noidx (組込み関数)	367
__SMLATB (組込み関数)	359	__STDC_VERSION__ (定義済シンボル)	381
__SMLATT (組込み関数)	359	__STDC__ (定義済シンボル)	381
__SMLAWB (組込み関数)	359	__STREX (組込み関数)	367
__SMLAWT (組込み関数)	359	__swi (拡張キーワード)	310
__SMLSDB (組込み関数)	360	__SWP (組込み関数)	367
__SMLSDBX (組込み関数)	360	__SWPB (組込み関数)	368
__SMLSDB (組込み関数)	360	__SXTAB (組込み関数)	368
__SMLSDBX (組込み関数)	360	__SXTAB16 (組込み関数)	368
__SMMLA (組込み関数)	361	__SXTAH (組込み関数)	368
__SMMLAR (組込み関数)	361	__SXTB16 (組込み関数)	369
__SMMLS (組込み関数)	361	__task (拡張キーワード)	312
__SMMLSR (組込み関数)	361	__thumb (拡張キーワード)	312
__SMMUL (組込み関数)	362	__TIME__ (定義済シンボル)	381
__SMMULR (組込み関数)	362	__UADD8 (組込み関数)	369
__SMUAD (組込み関数)	362	__UADD16 (組込み関数)	369
__SMUADX (組込み関数)	362	__UASX (組込み関数)	369
__SMUL (組込み関数)	363	__UHADD8 (組込み関数)	370
__SMULBB (組込み関数)	363	__UHADD16 (組込み関数)	370
__SMULBT (組込み関数)	363	__UHASX (組込み関数)	370
__SMULTB (組込み関数)	363	__UHSAX (組込み関数)	370
__SMULTT (組込み関数)	364	__UHSUB16 (組込み関数)	371
__SMULWB (組込み関数)	364	__UMAAL (組込み関数)	371
__SMULWT (組込み関数)	364	__ungetchar, stdio.h	390
__SMUSD (組込み関数)	364	__UQADD8 (組込み関数)	371
__SMUSD (組込み関数)	365	__UQADD16 (組込み関数)	372
__SSAT (組込み関数)	365	__UQASX (組込み関数)	372
__SSAT16 (組込み関数)	365	__UQSAX (組込み関数)	372
__SSAX (組込み関数)	365	__UQSUB8 (組込み関数)	372
__SSUB8 (組込み関数)	366	__UQSUB16 (組込み関数)	373
__SSUB16 (組込み関数)	366	__USADA8 (組込み関数)	373
__stackless (拡張キーワード)	310	__USAD8 (UMAAL)	373
__STC (組込み関数)	366	__USAT (組込み関数)	373
__STCL (組込み関数)	366	__USAT16 (組込み関数)	374
__STCL_noidx (組込み関数)	367	__USAX (組込み関数)	374
__STC_noidx (組込み関数)	367	__USUB8 (組込み関数)	374
__STC2 (組込み関数)	366	__USUB16 (組込み関数)	374
__STC2L (組込み関数)	366	__UXTAB (組込み関数)	375
__STC2L_noidx (組込み関数)	367	__UXTAB16 (組込み関数)	375

__UXTAH (組込み関数)	375	--char_is_signed (コンパイラオプション)	224
__UXTB16 (組込み関数)	375	--char_is_unsigned (コンパイラオプション)	225
__VA_ARGS__ (プリプロセッサ拡張)	147	--checksum (ielftool オプション)	438
__VER__ (定義済シンボル)	381	--code (ielfdump オプション)	440
__weak (拡張キーワード)	312	--config (リンカオプション)	263
__write (ライブラリ関数)	114	--config_def (リンカオプション)	263
カスタマイズ	110	--cpp_init_routine (リンカオプション)	264
__write_array, in stdio.h	390	--cpu (コンパイラオプション)	225
__write_buffered (DLIB ライブラリ関数)	100	--cpu (リンカオプション)	264
-d (iarchive オプション)	441	--cpu_mode (コンパイラオプション)	226
-D (コンパイラオプション)	227	--create (iarchive オプション)	440
-e (コンパイラオプション)	232	--c++ (コンパイラオプション)	227
-f (iobjmanip オプション)	442	--c89 (コンパイラオプション)	224
-f (コンパイラオプション)	235	--debug (コンパイラオプション)	228
-f (リンカオプション)	270	--define_symbol (リンカオプション)	265
-I (コンパイラオプション)	237	--delete (iarchive オプション)	441
-l (コンパイラオプション)	238	--dependencies (コンパイラオプション)	228
スケルトンコードの作成	134	--dependencies (リンカオプション)	265
-o (iarchive オプション)	444	--diagnostics_tables (コンパイラオプション)	231
-o (ielfdump オプション)	444	--diagnostics_tables (リンカオプション)	267
-o (コンパイラオプション)	249-250	--diag_error (コンパイラオプション)	229
-o (リンカオプション)	279	--diag_error (リンカオプション)	266
-r (iarchive オプション)	447	--diag_remark (コンパイラオプション)	229
-r (コンパイラオプション)	228	--diag_remark (リンカオプション)	266
-S (iarchive オプション)	449	--diag_suppress (コンパイラオプション)	230
-s (ielfdump オプション)	448	--diag_suppress (リンカオプション)	267
-t (iarchive オプション)	452	--diag_warning (コンパイラオプション)	230
-V (iarchive オプション)	452	--diag_warning (リンカオプション)	267
-x (iarchive オプション)	442	--discard_unused_publics (コンパイラオ プション)	231
--no_library_search (リンカオプション)	276	--dlib_config (コンパイラオプション)	231
--aapcs (コンパイラオプション)	223	--ec++ (コンパイラオプション)	233
--aeabi (コンパイラオプション)	223	--edit (isymexport オプション)	441
--align_sp_on_irq (コンパイラオプション)	223	--eec++ (コンパイラオプション)	233
--all (ielfdump オプション)	438	--enable_hardware_workaround (コンパイラオ プション)	233
--arm (コンパイラオプション)	224	--enable_multibytes (コンパイラオプション)	234
--basic_heap (リンカオプション)	262	--endian (コンパイラオプション)	234
--BE32 (リンカオプション)	263	--entry (リンカオプション)	268
--BE8 (リンカオプション)	262	--enum_is_int (コンパイラオプション)	235
--bin (ielftool オプション)	438		

--error_limit (コンパイラオプション)	235	--no_dynamic_rtti_elimination (リンカオプション) ..	275
--error_limit (リンカオプション)	268	--no_exceptions (コンパイラオプション)	243
--exception_tables (リンカオプション)	269	--no_exceptions (リンカオプション)	276
--export_builtin_config (リンカオプション)	269	--no_fragments (コンパイラオプション)	243
--extract (iarchive オプション)	442	--no_fragments (リンカオプション)	276
--extra_init (リンカオプション)	270	--no_inline (コンパイラオプション)	243
--fill (ielftool オプション)	443	--no_locals (リンカオプション)	277
--force_exceptions (リンカオプション)	271	--no_loop_align (コンパイラオプション)	244
--force_output (リンカオプション)	271	--no_mem_idioms (コンパイラオプション)	244
--fpu (コンパイラオプション)	236	--no_path_in_file_macros (コンパイラオプション) ..	244
--guard_calls (コンパイラオプション)	237	--no_range_reservations (リンカオプション)	277
--header_context (コンパイラオプション)	237	--no_remove (リンカオプション)	277
--ihex (ielftool オプション)	443	--no_rtti (コンパイラオプション)	245
--image_input (リンカオプション)	271	--no_rw_dynamic_init (コンパイラオプション)	245
--inline (リンカオプション)	272	--no_scheduling (コンパイラオプション)	246
--interwork (コンパイラオプション)	238	--no_static_destruction (コンパイラオプション)	246
--keep (リンカオプション)	272	--no_strtab (ielfdump オプション)	444
--legacy (コンパイラオプション)	239	--no_system_include (コンパイラオプション)	246
--lock_regs (コンパイラオプション)	239	--no_typedefs_in_diagnostics	
--log (リンカオプション)	273	(コンパイラオプション)	247
--log_file (リンカオプション)	274	--no_unaligned_access (コンパイラオプション)	248
--macro_positions_in_diagnostics		--no_unroll (コンパイラオプション)	248
(コンパイラオプション)	240	--no_veneer (リンカオプション)	278
--mangled_names_in_messages		--no_vfe (コンパイラオプション)	278
(リンカオプション)	274	--no_warnings (コンパイラオプション)	248
--map (リンカオプション)	274	--no_warnings (リンカオプション)	278
--mfc (コンパイラオプション)	240	--no_wrap_diagnostics (コンパイラオプション)	249
--migration_preprocessor_extensions		--no_wrap_diagnostics (リンカオプション)	279
(コンパイラオプション)	241	--only_stdout (コンパイラオプション)	250
--misrac (コンパイラオプション)	220	--only_stdout (リンカオプション)	279
--misrac_verbose (コンパイラオプション)	220	--output (iarchive オプション)	444
--misrac_verbose (リンカオプション)	260	--output (ielfdump オプション)	444
--misrac1998 (コンパイラオプション)	220	--output (コンパイラオプション)	250
--misrac1998 (リンカオプション)	260	--output (リンカオプション)	279
--misrac2004 (コンパイラオプション)	220	--pi_veneer (リンカオプション)	280
--misrac2004 (リンカオプション)	260	--place_holder (リンカオプション)	280
--no_clustering (コンパイラオプション)	241	--predef_macro (コンパイラオプション)	250
--no_code_motion (コンパイラオプション)	241	--preinclude (コンパイラオプション)	251
--no_const_align (コンパイラオプション)	242	--preprocess (コンパイラオプション)	251
--no_cse (コンパイラオプション)	242	--ram_reserve_ranges (isymexport オプション)	445

--raw (ielfdump) オプション)	445	--verbose (ielftool オプション)	452
--redirect (リンカオプション)	281	--vfe (コンパイラオプション)	283
--relaxed_fp (コンパイラオプション)	252	--vla (コンパイラオプション)	257
--remarks (コンパイラオプション)	253	--warnings_affect_exit_code (コンパイラオプション)	209, 258
--remarks (リンカオプション)	281	--warnings_affect_exit_code (リンカオプション)	284
--remove_section (iobjmanip オプション)	446	--warnings_are_errors (コンパイラオプション)	258
--rename_section (iobjmanip オプション)	446	--warnings_are_errors (リンカオプション)	284
--rename_symbol (iobjmanip オプション)	446	.bss (ELF セクション)	419
--replace (iarchive オプション)	447	.comment (ELF セクション)	418
--require_prototypes (コンパイラオプション)	253	.cstart (ELF セクション)	419
--reserve_ranges (isymexport オプション)	447	.data (ELF セクション)	419
--ropi (コンパイラオプション)	253	.data_init (ELF セクション)	419
--rwp (コンパイラオプション)	254	.debug (ELF セクション)	418
--search (リンカオプション)	281	.exc.text (ELF セクション)	420
--section (ielfdump オプション)	448	.iar.debug (ELF セクション)	418
--section (コンパイラオプション)	254	.iar.dynexit (ELF セクション)	420
--semihosting (リンカオプション)	282	.init_array (セクション)	421
--separate_cluster_for_initialized_variables (コンパイラオプション)	255	.intvec (セクション)	421
--silent (iarchive オプション)	449	.noinit (ELF セクション)	421
--silent (ielftool オプション)	449	.preinit_array (セクション)	421
--silent (コンパイラオプション)	255	.prepreinit_array (セクション)	422
--silent (リンカオプション)	282	.rel (ELF セクション)	418
--simple (ielftool オプション)	449	.rela (ELF セクション)	418
--srec (ielftool オプション)	450	.rodata (ELF セクション)	422
--srec-len (ielftool オプション)	450	.shstrtab (ELF セクション)	418
--srec-s3only (ielftool オプション)	450	.strtab (ELF セクション)	418
--strict (コンパイラオプション)	256	.symtab (ELF セクション)	418
--strip (ielftool オプション)	451	.text (ELF セクション)	422
--strip (iobjmanip オプション)	451	.textrow (ELF セクション)	422
--strip (リンカオプション)	283	.textrow_init (ELF セクション)	422
--strtab (ielfdump オプション)	449	[ターミナル I/O] ウィンドウ	
--symbols (iarchive オプション)	451	サポートされない場合	102
--system_include_dir (コンパイラオプション)	256	使用可能にする (DLIB)	99
--thumb (コンパイラオプション)	256	@ (演算子)	
--toc (iarchive オプション)	452	セクション内への配置	189
--use_c++_inline (コンパイラオプション)	257	絶対アドレスに配置	188
--use_unix_directory_separators (コンパイラオブジェ クト)	257	#include ファイル、指定	207, 237
--verbose (iarchive オプション)	452	#warning message (プリプロセッサ拡張)	382
		%Z 置換文字列、処理系定義の動作	466

数字

32 ビット（浮動小数点数フォーマット）293

64 ビット（浮動小数点数フォーマット）293