IAR Embedded Workbench®

IAR C/C++ 開発ガイド

コンパイルおよびリンク

ルネサス エレクトロニクス
RL78 マイクロコントローラファミリ

DRL78_I-2-J

版権事項
© 2011–2015 IAR Systems AB.

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

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

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

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

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

ルネサスはルネサス エレクトロニクス株式会社の登録商標です。RL78 はルネサス エレクトロニクス株式会社の商標です。

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

その他のすべての製品名は、その所有者の商標または登録商標です。

改版情報
第 2 版 : 2015 年 5 月

部品番号 : DRL78_I-2-J

本ガイドは、ルネサス RL78 マイクロコントローラファミリ用 IAR Embedded Workbench® のバージョン 2.x に適用する。

*『RL78 用 IAR C/C++ コンパイラユーザガイド』*は、すべてのバージョンの*『RL78 用 IAR C/C++ コンパイラリファレンスガイド』*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 : M18、csrct2010.1、V_110411、IJOA。

目次（章）

表 29
はじめに 31

パート 1. ビルドツールの使用 39

IAR ビルドツールの概要 41
組込みアプリケーションの開発 47
データ記憶 61
関数 73
ILINK を使用したリンク 83
アプリケーションのリンク 95
DLIB ランタイム環境 109
アセンブラ言語インタフェース 143
C の使用 167
C++ の使用 177
アプリケーションに関する考慮事項 193
組込みアプリケーション用の効率的なコーディング 203

パート 2. リファレンス情報 223

外部インタフェースの詳細 225
コンパイラオプション 235
リンカオプション 273
データ表現 295
拡張キーワード 309

プラグマディレクティブ 325
組込み関数 349
プリプロセッサ 353
ライブラリ関数 361
リンカ設定ファイル 371
セクションリファレンス 397
IAR ユーティリティ 409
C 規格の処理系定義の動作 445
C89 の処理系定義の動作 461
索引 473

目次

表 29
はじめに 31
本ガイドの対象者 31
必要な知識 31
本ガイドの使用方法 31
本ガイドの内容 32
パート 1. ビルドツールの使用 32
パート 2. リファレンス情報 33
その他のドキュメント 33
ユーザガイドおよびリファレンスガイド 34
オンラインヘルプシステムを参照 34
参考資料 35
Web サイト 35
表記規則 36
表記規則 36
命名規約 37
パート 1. ビルドツールの使用 39
IAR ビルドツールの概要 41
IAR ビルドツール — 概要 41
IAR C/C++ コンパイラ 41
IAR アセンブラ 41
IAR ILINK リンカ 42
専用 ELF ツール 42
外部ツール 42
IAR 言語の概要 43
デバイスサポート 43
サポートされている RL78 デバイス 44
事前に定義されているサポートファイル 44
開発を開始するためのサンプルプロジェクト 44

組込みシステム用の特殊サポート ... 44
拡張キーワード ... 45
プラグマディレクティブ ... 45
定義済シンボル ... 45
低レベル機能へのアクセス ... 45
組込みアプリケーションの開発 ... 47
IAR ビルドツールを使用した組込みソフトウェアの開発 ... 47
CPU 機能と制約 ... 47
メモリのマッピング ... 48
周辺ユニットとの通信 ... 48
イベント処理 ... 48
システム起動 ... 49
リアルタイムオペレーティングシステム ... 49
他のビルドツールとの相互運用 ... 49
ビルドプロセス — 概要 ... 50
変換プロセス ... 50
リンク処理 ... 51
リンク後 ... 52
アプリケーションの実行 — 概要 ... 53
初期化フェーズ ... 53
実行フェーズ ... 56
終了フェーズ ... 56
アプリケーションのビルド — 概要 ... 57
基本的なプロジェクト設定 ... 57
プロセッサコア ... 58
データモデル ... 58
コードモデル ... 59
double 浮動小数点型のサイズ ... 59
速度とサイズの最適化 ... 59
ランタイム環境 ... 59
データ記憶 ... 61
概要 ... 61
さまざまなデータ記憶方法 ... 61

メモリタイプ 62
メモリタイプの概要 62
データメモリ属性の使用 63
ポインタとメモリタイプ 65
構造体とメモリタイプ 66
その他の例 66
C++ とメモリタイプ 67
データモデル 68
データモデルの指定 68
自動変数とパラメータの記領域憶 69
スタック 69
ショートアドレス作業エリア 70
ヒープ上の動的メモリ 71
潜在的な問題 71
関数 73
関数関連の拡張 73
関数格納のためのコードモデルとメモリ属性 73
関数メモリ属性の使用 74
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 75
割込み関数 75
モニタ関数 76
インライン関数 79
C と C++ の動作の比較 80
関数のインライン化を制御する機能 80
ILINK を使用したリンク 83
リンクの概要 83
モジュールおよびセクション 84
リンクプロセスの詳細 85
コードおよびデータの配置（リンカ設定ファイル） 87
設定ファイルの簡単な例 88

システム起動時の初期化 90
初期化プロセス 91
C++ 動的初期化 93
コンパイラにより生成された追加のセクション 93

アプリケーションのリンク 95

リンクについて 95
リンカ設定ファイルの選択 95
独自のメモリエリアの定義 96
セクションの配置 97
RAM の空間の予約 98
モジュールの保持 98
シンボルおよびセクションの保持 99
アプリケーションの起動 99
スタックメモリの設定 99
ヒープメモリの設定 100
atexit 制限の設定 100
デフォルト初期化の変更 100
ILINK とアプリケーション間の相互処理 104
標準ライブラリの処理 105
ELF/DWARF 以外の出力フォーマットの生成 105
トラブルシューティングについてのヒント 105
再配置エラー 106

DLIB ランタイム環境 109

ランタイム環境の概要 109
ランタイム環境の機能 109
ランタイム環境の設定 110
ビルド済ライブラリの使用 111
ライブラリファイル名構文 112
ライブラリファイルのグループ 112
ビルド済ライブラリのカスタマイズ（リビルドなし） 113
printf、scanf のフォーマッタの選択 114
printf フォーマッタの選択 114
scanf フォーマッタの選択 115

アプリケーションデバッグサポート 116
C-SPY デバッグサポートを含める 116
ライブラリ機能のデバッグ 117
C-SPY の［ターミナル I/O］ウィンドウ 117
デバッグライブラリの低レベル関数 118
ターゲットハードウェアのライブラリの適合 119
ライブラリの低レベルインタフェース 119
ライブラリモジュールのオーバライド 120
カスタマイズしたライブラリのビルドと使用 120
ライブラリプロジェクトのセットアップ 121
ライブラリ機能の修正 121
カスタマイズしたライブラリの使用 122
システムの起動と終了 122
システム起動 122
システム終了 124
システム初期化のカスタマイズ 125
__low_level_init 125
cstartup.s ファイルの修正 126
ライブラリ構成 126
ランタイム構成の選択 127
標準 I/O ストリーム 127
低レベルキャラクタ I/O の実装 127
printf、scanf の構成シンボル 129
フォーマット機能のカスタマイズ 130
ファイル I/O 131
ロケール 132
ビルド済ライブラリでのロケールサポート 132
ロケールサポートのカスタマイズ 132
実行中のロケール変更 133
環境の操作 134
getenv 関数 134
システム関数 135
signal と raise 135
時間 135

Strtod 136
数学関数 136
より小さいバージョン 136
より正確なバージョン 138
Assert 138
Atexit 139
ヒープ 139
ハードウェアサポート 139
モジュールの整合性チェック 140
ランタイムモデル属性 140
ランタイムモデル属性の使用 141
アセンブラ言語インタフェース 143
C 言語とアセンブラの結合 143
組込み関数 143
C 言語とアセンブラモジュールの結合 144
インラインアセンブラ 145
C からのアセンブラルーチンの呼出し 147
スケルトンコードの作成 147
スケルトンコードのコンパイル 148
C++ からのアセンブラルーチンの呼出し 149
呼出し規約 150
呼出し規約の選択 151
関数の宣言 151
C++ ソースコードでの C リンケージの使用 152
保護レジスタとスクラッチレジスタ 152
関数の入口 153
関数の終了 155
例 156
関数の呼び出しに使用されるアセンブラ命令 158
Near コードモデルの関数の呼び出し 158
Far コードモデルの関数の呼び出し 158
関数表の呼び出しの作成 159

メモリアクセス方法 159
saddr メモリアクセス方法 160
near メモリアクセス方法 160
far メモリアクセス方法 161
huge アクセス方法 161
コールフレーム情報 162
CFI ディレクティブ 162
CFI サポートを持つアセンブラソースの作成 163

C の使用 167

C 言語の概要 167
拡張の概要 168
言語拡張の有効化 169
IAR C 言語拡張 169
組込みシステムプログラミングの拡張 170
C 規格に対する緩和 172

C++ の使用 177

概要 — EC++ および EEC++ 177
Embedded C++ 177
拡張 Embedded C++ 178
C++ のサポートの有効化 179
EEC++ の機能の説明 179
IAR 属性とクラスを使用する 179
関数型 182
演算子 new と delete 183
割込みで静的クラスオブジェクトを使用する 184
新しいハンドラを使用する 185
テンプレート 185
C-SPY でのデバッグサポート 185
EEC++ の機能の説明 186
テンプレート 186
キャスト演算子の派生形 188
Mutable 189
名前空間 189

std 名前空間 ... 189
メンバ関数へのポインタ ... 189
C++ 言語拡張 ... 190
アプリケーションに関する考慮事項 ... 193
出力形式に関する注意事項 ... 193
スタックについて ... 194
スタックサイズについて ... 194
ヒープについて ... 194
DLIB のヒープセクション ... 194
ヒープサイズと標準 I/O ... 195
ツールとアプリケーション間の相互処理 ... 195
チェックサムの計算 ... 197
チェックサムの計算 ... 197
チェックサム関数をソースコードに追加する ... 199
注意事項 ... 200
C-SPY に関する注意事項 ... 201
リンカの最適化 ... 201
仮想関数の除去 ... 201
組込みアプリケーション用の効率的なコーディング ... 203
データ型の選択 ... 203
効率的なデータ型の使用 ... 203
浮動小数点数型 ... 204
構造体エレメントのアラインメント ... 204
匿名構造体と匿名共用体 ... 205
データと関数のメモリ配置制御 ... 206
絶対アドレスへのデータ配置 ... 207
データと関数のセクションへの配置 ... 209
コンパイラ最適化の設定 ... 210
最適化実行のスコープ ... 211
複数ファイルのコンパイルユニット ... 211
最適化レベル ... 212
速度とサイズ ... 213
変換の微調整 ... 213

円滑なコードの生成 216
最適化を容易にするソースコードの記述 217
スタックエリアと RAM メモリの節約 217
関数プロトタイプ 218
整数型とビット否定 219
同時にアクセスされる変数の保護 219
特殊機能レジスタへのアクセス 220
非初期化変数 221

パート 2. リファレンス情報 223

外部インタフェースの詳細 225

呼出し構文 225
コンパイラ呼出し構文 225
ILINK 呼出し構文 226
オプションの受渡し 226
環境変数 227
インクルードファイル検索手順 227
コンパイラ出力 228
エラーリターンコード 229
ILINK 出力 230
診断 231
コンパイラのメッセージフォーマット 231
リンカのメッセージフォーマット 231
重要度 232
重要度の設定 233
インターナルエラー 233

コンパイラオプション 235

オプションの構文 235
オプションのタイプ 235
パラメータの指定に関する規則 235

コンパイラオプションの概要 238
コンパイラオプションの説明 241
--c89 241
--calling_convention 242
--char_is_signed 242
--char_is_unsigned 243
--code_model 243
--code_section 243
--core 244
-D 245
--data_model 245
--debug, -r 246
--dependencies 246
--diag_error 247
--diag_remark 248
--diag_suppress 248
--diag_warning 249
--diagnostics_tables 249
--disable_div_mod_instructions 249
--discard_unused_publics 250
--dlib_config 250
--double 251
-e 251
--ec++ 252
--eec++ 252
--enable_multibytes 253
--enable_restrict 253
--error_limit 253
-f 254
--generate_callt_runtime_library_calls 254
--generate_far_runtime_library_calls 254
--guard_calls 255
--header_context 255
-I 255

-l 256
--macro_positions_in_diagnostics 257
--mfc 257
--near_const_location 258
--no_clustering 258
--no_code_motion 258
--no_cross_call 259
--no_cse 259
--no_dwarf3_cfi 259
--no_fragments 260
--no_inline 260
--no_path_in_file_macros 260
--no_scheduling 261
--no_size_constraints 261
--no_static_destruction 261
--no_system_include 262
--no_tbaa 262
--no_typedefs_in_diagnostics 262
--no_unroll 263
--no_warnings 263
--no_wrap_diagnostics 263
-O 264
--only_stdout 264
--output, -o 265
--pending_instantiations 265
--predef_macros 265
--preinclude 266
--preprocess 266
--public_equ 267
--relaxed_fp 267
--remarks 268
--require_prototypes 268
--silent 268
--strict 269

--system_include_dir 269
--use_c++_inline 269
--use_unix_directory_separators 270
--vla 270
--warn_about_c_style_casts 270
--warnings_affect_exit_code 271
--warnings_are_errors 271
--workseg_area 271

リンカオプション 273

リンカオプションの概要 273
リンカオプションの説明 275
--config 275
--config_def 276
--config_search 276
--cpp_init_routine 277
--debug_lib 277
--define_symbol 278
--dependencies 278
--diag_error 279
--diag_remark 279
--diag_suppress 280
--diag_warning 280
--diagnostics_tables 280
--entry 281
--error_limit 281
--export_builtin_config 282
-f 282
--force_output 282
--image_input 283
--keep 283
--log 284
--log_file 285
--mangled_names_in_messages 285

--map ... 285
--merge_duplicate_sections ... 286
--no_fragments ... 286
--no_library_search ... 287
--no_locals ... 287
--no_range_reservations ... 288
--no_remove ... 288
--no_vfe ... 288
--no_warnings ... 289
--no_wrap_diagnostics ... 289
--only_stdout ... 289
--output, -o ... 289
--place_holder ... 290
--redirect ... 290
--remarks ... 291
--search ... 291
--silent ... 292
--strip ... 292
--vfe ... 292
--warnings_affect_exit_code ... 293
--warnings_are_errors ... 293
--whole_archive ... 293

データ表現 ... 295

アラインメント ... 295
RL78 マイクロコントローラ のアラインメント ... 296

基本データ型整数型 ... 296
整数型概要 ... 296
bool 型 ... 297
enum 型 ... 297
char 型 ... 297
wchar_t 型 ... 297
ビットフィールド ... 298

基本データ型浮動小数点数型 299
浮動小数点環境 300
32 ビット浮動小数点数フォーマット 300
64 ビット浮動小数点数フォーマット 300
特殊な浮動小数点数の表現 300
ポインタ型 301
関数ポインタ 301
データポインタ 301
キャスト 302
構造体型 303
構造体型のアライメント 303
一般的なレイアウト 303
パック構造体型 304
型修飾子 305
オブジェクトの volatile 宣言 305
オブジェクト volatile および const の宣言 307
オブジェクトの const 宣言 307
C++ のデータ型 308
拡張キーワード 309
拡張キーワードの一般的な構文規則 309
型属性 309
オブジェクト属性 312
拡張キーワードの一覧 313
拡張キーワードの詳細 314
__callt 314
__far 314
__far_func 315
__huge 315
__interrupt 316
__intrinsic 316
__monitor 317
__near 317
__near_func 317

__no_alloc、__no_alloc16 ... 318
__no_alloc_str、__no_alloc_str16 ... 318
__no_bit_access ... 319
__no_init ... 319
__noreturn ... 320
__no_save ... 320
__root ... 320
__ro_placement ... 321
__saddr ... 321
__sfr ... 322
__v1_call ... 322
__v2_call ... 322
__weak ... 323

プラグマディレクティブ ... 325

プラグマディレクティブの一覧 ... 325
プラグマディレクティブの詳細 ... 327
bank ... 327
basic_template_matching ... 327
bitfields ... 328
constseg ... 328
data_alignment ... 329
dataseg ... 330
default_function_attributes ... 330
default_variable_attributes ... 331
diag_default ... 332
diag_error ... 332
diag_remark ... 333
diag_suppress ... 333
diag_warning ... 333
error ... 334
include_alias ... 334
inline ... 335
language ... 336

location 336
message 337
no_workseg 338
object_attribute 338
optimize 339
pack 340
__printf_args 341
public_equ 341
required 341
rtmodel 342
__scanf_args 343
section 343
STDC CX_LIMITED_RANGE 344
STDC FENV_ACCESS 344
STDC FP_CONTRACT 345
type_attribute 345
unroll 346
vector 347
weak 347

組込み関数 349

組込み関数の概要 349

組込み関数の詳細 350

__break 350
__disable_interrupt 350
__enable_interrupt 350
__get_interrupt_level 350
__get_interrupt_state 350
__halt 351
__mach 351
__machu 351
__no_operation 351
__set_interrupt_level 351

__set_interrupt_state 352
__stop 352

プリプロセッサ 353

プリプロセッサの概要 353
定義済プリプロセッサシンボルの詳細 354
__BASE_FILE__ 354
__BUILD_NUMBER__ 354
__CALLING_CONVENTION__ 354
__CODE_MODEL__ 354
__CORE__ 354
__COUNTER__ 355
__cplusplus 355
__DATA_MODEL__ 355
__DATE__ 355
__embedded_cplusplus 355
__FAR_RUNTIME_ATTRIBUTE__ 356
__FILE__ 356
__func__ 356
__FUNCTION__ 356
__IAR_SYSTEMS_ICC__ 356
__ICCRL78__ 357
__LINE__ 357
__LITTLE_ENDIAN__ 357
__PRETTY_FUNCTION__ 357
__STDC__ 357
__STDC_VERSION__ 357
__SUBVERSION__ 358
__TIME__ 358
__TIMESTAMP__ 358
__VER__ 358
その他のプリプロセッサ拡張 358
NDEBUG 358
#warning message 359

ライブラリ関数 361
ライブラリの概要 361
ヘッダファイル 361
ライブラリオブジェクトファイル 361
より高精度な代替ライブラリ関数 362
リエントラント性 362
longjmp 関数 363
IAR DLIB ライブラリ 363
C ヘッダファイル 363
C++ ヘッダファイル 364
組込み関数としてのライブラリ関数 367
C の追加機能 367
ライブラリにより内部的に使用されるシンボル 369
リンカ設定ファイル 371
概要 371
メモリおよび領域の定義 372
define memory ディレクティブ 373
define region ディレクティブ 373
領域 374
領域リテラル 374
領域式 375
空の領域 376
セクションの取扱い 377
define block ディレクティブ 378
define overlay ディレクティブ 379
initialize ディレクティブ 380
do not initialize ディレクティブ 383
keep ディレクティブ 384
place at ディレクティブ 384
place in ディレクティブ 385
use init table ディレクティブ 386

セクションの選択 387
section-selectors 387
extended-selectors 389
シンボル、式、数値の使用 391
define symbol ディレクティブ 391
export ディレクティブ 392
式 392
数値 393
構造化設定 394
error ディレクティブ 394
if ディレクティブ 395
include ディレクティブ 395
セクションリファレンス 397
セクションの概要 397
セクションおよびブロックの説明 399
.bss 400
.bss.noinit 400
.bssf 400
.bssf.noinit 400
.callt0 400
.const 401
.constf 401
.consth 401
CSTACK 401
.data 401
.data_init 402
.dataf 402
.dataf_init 402
FAR_HEAP 403
.hbss 403
.hbss.noinit 403
.hdata 403
.hdata_init 404

HUGE_HEAP 404
.iar.dynexit 404
.init_array 404
.intvec 405
NEAR_HEAP 405
.option_byte 405
.preinit_array 405
.sbss 406
.sbss.noinit 406
.sdata 406
.sdata_init 406
.security_id 407
.switch 407
.switchf 407
.text 407
.textf 407
.wrkseg 408

IAR ユーティリティ 409

IAR アーカイブツール — iarchive 409
呼出し構文 409
iarchive コマンドの概要 410
iarchive オプションの概要 411
診断メッセージ 411

IAR ELF ツール — ielftool 413
呼出し構文 413
ielftool オプションの概要 414

IAR ELF Dumper — ielfdump 414
呼出し構文 415
ielfdump オプションの概要 415

IAR ELF オブジェクトツール — iobjmanip 416
呼出し構文 416
iobjmanip オプションの概要 416
診断メッセージ 417

IAR Absolute Symbol Exporter — isymexport 418
呼出し構文 419
isymexport のオプションの概要 420
ステアリングファイル 420
Show ディレクティブ 421
Hide ディレクティブ 421
Rename ディレクティブ 422
診断メッセージ 422
オプションの説明 424
--all 424
--bin 425
--checksum 425
--code 428
--create 428
--delete, -d 429
--edit 429
--extract, -x 430
-f 430
--fill 431
--generate_vfe_header 431
--ihex 432
--no_strtab 432
--output, -o 432
--parity 433
--ram_reserve_ranges 434
--raw 435
--remove_file_path 435
--remove_section 436
--rename_section 436
--rename_symbol 437
--replace, -r 437
--reserve_ranges 438
--section, -s 438
--self_reloc 439

--silent 439
--simple 440
--simple-ne 440
--srec 440
--srec-len 441
--srec-s3only 441
--strip 441
--symbols 442
--titxt 442
--toc, -t 443
--verbose, -V 443

C 規格の処理系定義の動作 445

処理系定義の動作の詳細 445
J.3.1 変換 445
J.3.2 環境 445
J.3.3 識別子 447
J.3.4 文字 447
J.3.5 整数 448
J.3.6 浮動小数点 449
J.3.7 配列およびポインタ 450
J.3.8 ヒント 450
J.3.9 構造体、共用体、列挙型、ビットフィールド 450
J.3.10 修飾子 451
J.3.11 プリプロセッサディレクティブ 451
J.3.12 ライブラリ関数 453
J.3.13 アーキテクチャ 458
J.4 ロケール 459

C89 の処理系定義の動作 461

処理系定義の動作の詳細 461
変換 461
環境 461
識別子 462
文字 462

整数 463
浮動小数点数 464
配列、ポインタ 464
レジスタ 465
構造体、共用体、列挙型、ビットフィールド 465
修飾子 466
宣言子 466
文 466
プリプロセッサディレクティブ 466
IAR DLIB ライブラリ関数 468
索引 473

表

1: 本ガイドで使用されている表記規則 36
2: このガイドで使用されている命名規約 37
3: プロセッサコア 58
4: メモリタイプと対応するキーワード 63
5: データモデルの特徴 68
6: コードモデル 74
7: 関数メモリ属性 74
8: 初期化データを保持するセクション 91
9: 再配置エラーの説明 106
10: カスタマイズ可能な項目 113
11: printf のフォーマッタ 114
12: scanf のフォーマッタ 115
13: 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 ... 118
14: ライブラリ構成 126
15: printf の構成シンボルの詳細 130
16: scanf の構成シンボルの詳細 130
17: 低レベルファイル I/O 131
18: ヒープとメモリタイプ 139
19: ランタイムモデル属性の例 141
20: パラメータの引渡しに使用されるレジスタ 154
21: リターン値に使用されるレジスタ 155
22: 名前ブロックで定義されている呼出しフレーム情報リソース 163
23: 言語拡張 169
24: セクション演算子とそのシンボル 172
25: コンパイラ最適化レベル 212
26: コンパイラの環境変数 227
27: ILINK 環境変数 227
28: エラーリターンコード 229
29: コンパイラオプションの一覧 238
30: リンカオプションの概要 273
31: 整数型 296

32: 浮動小数点数型 299
33: 関数ポインタ 301
34: データポインタ 301
35: 拡張キーワードの一覧 313
36: プラグマディレクティブの一覧 325
37: 組込み関数の一覧 349
38: 従来の標準 C ヘッダファイル — DLIB 364
39: C++ ヘッダファイル 365
40: 標準テンプレートライブラリヘッダファイル 365
41: 新しい標準 C ヘッダファイル — DLIB 366
42: セクションセレクタの指定の例 389
43: セクションの概要 397
44: iarchive パラメータ 410
45: iarchive コマンドの概要 410
46: iarchive オプションの概要 411
47: ielftool のパラメータ 413
48: ielftool オプションの概要 414
49: ielfdumprl78 parameters 415
50: ielfdumprl78 オプションの概要 415
51: iobjmanip パラメータ 416
52: iobjmanip オプションの概要 416
53: isymexport のパラメータ 419
54: isymexport オプションの概要 420
55: strerror() が返すメッセージ — IAR DLIB ライブラリ 460
56: strerror() が返すメッセージ — IAR DLIB ライブラリ 471

はじめに

*RL78 用 IAR C/C++ コンパイラユーザガイド*へようこそ。このガイドは、開発中のアプリケーション要件に対し、最適な方法でビルドツールをご利用いただくのに役立つ、詳細なリファレンス情報を提供します。また、アプリケーションを効率的に開発するための推奨コーディングテクニックも説明しています。

本ガイドの対象者

本ガイドは、C/C++ 言語を使用して RL78 マイクロコントローラ用アプリケーションを開発する予定があり、ビルドツールの使用方法に関する詳細情報を必要とするユーザを対象としています。また、以下について十分な知識があるユーザを対象としています。

必要な知識

IAR Embedded Workbench のツールを使用するには、以下の実践的な知識が必要です。

- RL78 マイクロコントローラ ファミリのアーキテクチャと命令セット（チップメーカーのドキュメントを参照してください）
- C/C++ プログラミング言語
- 組込みシステム用アプリケーションの開発
- ホストコンピュータのオペレーティングシステム

IDE に統合されている他の開発ツールの詳細は、それぞれのドキュメントをご覧ください（33 ページの*その他のドキュメント*を参照）。

本ガイドの使用方法

RL78 用 IAR C/C++ コンパイラコンパイラおよびリンカを使用する際には、本ガイドの「*パート 1. ビルドツールの使用*」を参照してください。

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

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

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

本ガイドの内容

本ガイドの構成および各章の概要を以下に示します。

パート 1. ビルドツールの使用

- 「*IAR ビルドツールの概要*」では、ツール、プログラミング言語、利用可能なデバイスサポート、RL78 マイクロコントローラの特定の機能をサポートするために提供されている拡張機能など、IAR ビルドツールの概要について説明します。
- 「*組込みアプリケーションの開発*」では、IAR ビルドツールを使用する組込みソフトウェアの開発に必要な基礎について説明します。
- 「*データ記憶*」では、メモリへのデータの保存方法について説明します。
- 「*関数*」に関連した拡張（関数を制御するための仕組み）の概要を説明した後、これらの仕組みのいくつかを取り上げて詳しく説明します。
- 「*ILINK を使用したリンク*」では、IAR ILINK リンカを使用するリンクプロセスおよび関連する概念について説明します。
- 「*アプリケーションのリンク*」では、ILINK オプションの使用およびリンカ設定ファイルの調整など、アプリケーションをリンクするときに注意する必要がある多くの事項を示します。
- 「*DLIB ランタイム環境*」では、アプリケーションの実行環境である DLIB ランタイム環境について説明します。オプションの設定、デフォルトライブラリモジュールへのオーバライド、自作ライブラリのビルドにより、ランタイムライブラリを変更する方法を説明します。また、システムの初期化、`cstartup` ファイルの概要、ロケール用モジュールの使用方法、ファイル I/O についても説明します。
- 「*アセンブラ言語インタフェース*」では、アプリケーションの一部をアセンブラ言語で記述する場合に必要な情報を説明します。呼出し規約についても説明しています。
- 「*C の使用*」は、C 言語でサポートされている 2 つの派生型の概要と、C 規格の拡張などコンパイラ拡張の概要を説明します。
- 「*C++ の使用*」C++ サポートの 2 つのレベルの概要を提供します。業界標準の EC++ と IAR 拡張 EC++。
- 「*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 RL78 固有のキーワードのリファレンス情報を提供します。
- 「*プラグマディレクティブ*」では、プラグマディレクティブのリファレンス情報を提供します。
- 「*組込み関数*」では、RL78 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。
- 「*C 規格の処理系定義の動作*」では、コンパイラが C 規格の処理系定義エリアをどのように扱うかについて説明します。
- 「*C89 の処理系定義の動作*」では、コンパイラが C89 言語標準の処理系定義エリアをどのように扱うかについて説明します。

その他のドキュメント

ユーザドキュメンテーションは、ハイパーテキスト PDF 形式、およびコンテキスト依存のオンラインヘルプシステム（HTML フォーマット）があります。

ドキュメンテーションには、インフォメーションセンタあるいは IAR Embedded Workbench IDE の **[ヘルプ]** メニューからアクセスできます。オンラインヘルプシステムは、F1 キーを押しても使用できます。

ユーザガイドおよびリファレンスガイド

IAR システムズの各開発ツールについては、一連のガイドで説明しています。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は、同梱されているクイックレファンレンスのブックレットおよび『*インストールおよびライセンス ガイド*』にあります。
- IAR Embedded Workbench および同梱のツールを使用するにあたっては、『*IAR Embedded Workbench® の使用開始の手順*』を参照してください。
- プロジェクト管理とビルドでの IDE の使用については、『*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、『*RL78 用 C-SPY® デバッガガイド*』を参照してください。
- RL78 用 IAR C/C++ コンパイラのプログラミングおよび IAR ILINK リンカを使用したリンクについては、『*RL78 用 IAR C/C++ コンパイラユーザガイド*』を参照してください。
- RL78 用 IAR アセンブラを使用したプログラミングについては、『*RL78 用 IAR アセンブラリファレンスガイド*』を参照してください。
- IAR DLIB ライブラリの使用については、オンラインヘルプで利用できる *DLIB ライブラリリファレンス情報*を参照してください。
- MISRA-C ガイドラインを使用して、安全性を最重要視したアプリケーションを開発する方法については、『*IAR Embedded Workbench® MISRA-:2004 Reference Guide*』または『*IAR Embedded Workbench® MISRA-C:1998 Reference Guide*』を参照してください。
- 以前の UBROF ベースの製品バージョンから、ELF/DWARF オブジェクトフォーマットを使用する新しいバージョンに移行する際は、『*IAR Embedded Workbench® UBROF から ELF/DWARF の移行*』ガイドを参照してください。

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

オンラインヘルプシステムを参照

コンテキスト依存のオンラインヘルプの内容は以下のとおりです。

- IDE でのプロジェクト管理と編集、ビルドに関する情報
- IAR C-SPY® デバッガを使用したデバッグについての情報

- IDE のメニューやウィンドウ、ダイアログボックスに関するリファレンス情報
- コンパイラのリファレンス情報
- DLIB ライブラリ関数のキーワードリファレンス情報 関数のリファレンス情報を確認するには、エディタウィンドウで関数名を選択し、F1 キーを押します。

参考資料

IAR システムズ開発ツールの使用時は、以下の資料が参考になります。

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

- ルネサスの Web サイト (**www.renesas.com**) には、RL78 マイクロコントローラに関する情報とニュースが記載されています。
- IAR システムズの Web サイト (**www.iar.com/jp**) では、アプリケーションノートおよびその他の製品情報を公開しています。

- C 標準化作業グループの Web サイト、**www.open-std.org/jtc1/sc22/wg14**。
- C++ Standards Committee の Web サイト、**www.open-std.org/jtc1/sc22/wg21**。
- Embedded C++ Technical CommitteeのWebサイト(**www.caravan.net/ec2plus**)には、Embedded C++ 規格についての情報が公開されています。

表記規則

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

製品インストール内のディレクトリについて言及する場合（例 :`rl78¥doc` など)、その場所のフルパスを前提とします。この場合、`c:¥Program Files¥IAR Systems¥Embedded Workbench 7.`*`n`*`¥rl78¥doc` となります。

表記規則

IAR システムズのドキュメント セットでは、次の表記規則を使用します :

スタイル	用途
`computer`	• ソースコードの例、ファイルパス。 • コマンドライン上のテキスト。 • 2 進数、16 進数、8 進数。
`parameter`	パラメータとして使用される実際の値を表すプレースホルダ。たとえば、*`filename`*`.h` の場合、*`filename`* はファイルの名前を表します。
[`option`**]**	ディレクティブのオプション部分、**[** と **]** は実際のディレクティブの一部ではありませんが、`[`、`]`、`{`、`}` はいずれもディレクティブ構文の一部です。
{`option`**}**	ディレクティブの必須部分、**{** と **}** は実際のディレクティブの一部ではありませんが、`[`、`]`、`{`、`}` はいずれもディレクティブ構文の一部です。
`[option]`	コマンドのオプション部分。
`[a|b|c]`	代替の選択肢を持つコマンドのオプション部分。
`{a|b|c}`	コマンドの必須部分に選択肢があることを示します。
太字	画面で表示されるメニュー、メニューコマンド、ボタン、ダイアログボックス の名前を示します。
斜体	• 本ガイドや他のガイドへのクロスリファレンスを示します。 • 強調。

表 1: 本ガイドで使用されている表記規則

スタイル	用途
…	3 点リーダは、その前の項目を任意の回数繰り返せることを示します。
	IAR Embedded Workbench® IDE 固有の内容を示します。
	コマンドライン インタフェース固有の内容を示します。
	開発やプログラミングについてのヒントを示します。
	ワーニングを示します。

表 1: 本ガイドで使用されている表記規則（続き）

命名規約

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

ブランド名	一般名称
RL78 用 IAR Embedded Workbench®	IAR Embedded Workbench®
RL78 用 IAR Embedded Workbench® IDE	IDE
RL78 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
RL78 用 IAR C/C++ コンパイラ	コンパイラ
RL78 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 1. ビルドツールの使用

*『RL78 用 IAR C/C++ コンパイラユーザガイド』*のこのパートは、以下の章で構成されています。

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

IAR ビルドツールの概要

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

IAR ビルドツール — 概要

IAR 製品インストールには、RL78 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、『*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

RL78 用 IAR C/C++ コンパイラは、C/C++ 言語の標準機能に加えて、RL78 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

RL78 用 IAR アセンブラは、柔軟なディレクティブや式演算子セットを備えた強力な再配置マクロアセンブラです。C 言語プリプロセッサを内蔵しており、条件アセンブリをサポートしています。

RL78 用 IAR アセンブラでは、ルネサス RL78 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、*『RL78 用 IAR アセンブラリファレンスガイド』*を参照してください。

IAR ILINK リンカ

RL78 用 IAR ILINK リンカは、組込みコントローラアプリケーションの開発に適した、強力で柔軟性のあるソフトウェアツールです。リンカは、サイズの大きい再配置可能な入力の、マルチモジュールの C/C++ プログラムや C/C++ プログラムとアセンブラプログラムの混合リンクに適していますが、サイズの小さい単一ファイルの絶対アドレスを持つアセンブラプログラムのリンクにも同様に適しています。

専用 ELF ツール

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

- IAR Archive Tool (`iarchive`) は、複数の ELF オブジェクトファイルで構成するライブラリ（アーカイブ）の作成および操作を行います。
- IAR ELF Tool (`ielftool`) は、ELF 実行可能イメージ上でさまざまな変換（フィル、チェックサム、フォーマット変換など）を実行します。
- IAR ELF Dumper for RL78 (`ielfdumprl78`) は、ELF 再配置可能イメージまたは実行可能イメージの内容のテキスト表示を作成します。
- IAR ELF Object Tool (`iobjmanip`) は、ELF オブジェクトファイルの下位レベルの操作を実行する際に使用します。
- IAR Absolute Symbol Exporter (`isymexport`) は、ROM イメージファイルから絶対シンボルをエクスポートします。これらのシンボルは、アドオンアプリケーションのリンク時に使用できます。

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、*『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

RL78 用 IAR C/C++ コンパイラは以下をサポートしています。

- C：組込みシステム業界で最も幅広く使用されている高級プログラミング言語です。以下の標準に準拠したフリースタンディングアプリケーションのビルドが可能です。
  - 標準 C － C99 とも呼ばれる標準の C。本ガイドでは、この規格を *C 規格*と呼びます。
  - C89 は C94、C90、C89、ANSI C とも呼ばれ、この規格は MISRA-C が有効なときに必須です。
- C++：最新のオブジェクト指向プログラミング言語です。モジュール方式のプログラミングに最適なフル機能のライブラリを備えています。以下のすべての標準が使用できます。
  - 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、名前空間、新しいキャスト演算子、標準テンプレートライブラリ (STL) などの追加機能をサポートしています。

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

C の詳細は、「*C の使用*」を参照してください。

Embedded C++ と拡張 Embedded C++ の詳細については、章 *C++ の使用*を参照してください。

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

また、アプリケーションの一部または全部をアセンブラ言語で実装することもできます。*『RL78 用 IAR アセンブラリファレンスガイド』*を参照してください。

デバイスサポート

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

サポートされている RL78 デバイス

RL78 用 IAR C/C++ コンパイラでは、標準のルネサス RL78 マイクロコントローラコアに基づくすべてのデバイスをサポートしています。

注 : バージョン 1.x のコンパイラでは、コア S1、S2、S3 は RL78 0、RL78 1、RL78 2 という名前でした。

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。

リンカ設定ファイル

`rl78¥config` ディレクトリには、サポートされている全デバイスの既成のリンカ設定ファイルが含まれています。これらのファイルのファイル名拡張子は `icf` で、リンカで必要な情報が含まれています。リンカ設定ファイルの詳細は、「87 ページの*コードおよびデータの配置（リンカ設定ファイル）*」を、リファレンス情報については「*リンカ設定ファイル*」を参照してください。

デバイス記述ファイル

デバッガは、使用可能なメモリエリア、周辺レジスタおよびこれらのグループの定義など、いくつかのデバイス固有の要件を、デバイス記述ファイルを使用して処理します。これらのファイルは、`rl78¥config` ディレクトリにあり、そのファイル名拡張子は `ddf` です。周辺レジスタおよびそのグループは、個別のファイル（ファイル名拡張子 `sfr`）で定義できます。これは、`ddf` ファイルに含まれています。これらのファイルの詳細については、*RL78 用 C-SPY® デバッガガイド*を参照してください。

開発を開始するためのサンプルプロジェクト

`rl78¥examples` ディレクトリには、開発を問題なく開始できるように、動作アプリケーションのサンプルが提供されています。

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 RL78 マイクロコントローラ固有の機能をサポートするために提供されている拡張の概要を説明します。

拡張キーワード

コンパイラは、コード生成方法の設定に使用するキーワードセットを提供しています。たとえば、データオブジェクトへのアクセスおよび保存方法を制御するためのキーワードです。また関数が内部でどのように動作するか、およびその呼び出し / 返し方法を制御します。

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、251 ページの *-e* を参照してください。

拡張キーワードの詳細は、「*拡張キーワード*」を参照してください。*61 ページのデータ記憶*、*73 ページの関数*を参照してください。

プラグマディレクティブ

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

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

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

定義済シンボル

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

定義済シンボルの詳細は、「*プリプロセッサ*」を参照してください。

低レベル機能へのアクセス

アプリケーションのハードウェア関連部分では、低レベル機能へのアクセスが必要不可欠です。コンパイラは組込み関数、C モジュールとアセンブラモジュールの混在、インラインアセンブラなどの方法でサポートしています。これらの方法については、143 ページの *C 言語とアセンブラの結合*を参照してください。

組込みアプリケーションの開発

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

IAR ビルドツールを使用した組込みソフトウェアの開発

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

CPU 機能と制約

RL78 マイクロコントローラ 16 ビットの CISC Harvard アーキテクチャを基にしていて、RAM にはフル 16 ビットのデータバス、CPU のレジスタ、周辺ユニット、および 3 段階のパイプラインがあります。低電力アプリケーション用に設計されていて、スタンドバイモードでシリアル通信や ADC 操作ができます。RL78 マイクロコントローラで CISC 命令セットを使用していても、命令の主要な部分は 1–2 のクロックサイクルで実行されます。

RL78 マイクロコントローラにはハードウェアの機能があり、積算、積算 / 累積、および除算などの数学関数で援助します。積算および MACC 関数は 1 および 2 クロックサイクルで実行されます。RL78 バレルシフト関数は、1 クロックサイクルで 16 ビット `SHIFT` または `ROTATE` 命令を実行できます。

コンパイラは、コンパイラオプション拡張キーワード、プラグマディレクティブなどを意味することで、これらのハードウェア機能をサポートしています。

メモリのマッピング

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

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

メモリを効率的に使用するため、コンパイラでは、関数とデータオブジェクトのメモリへの配置を制御するためのさまざまな仕組みを提供しています。詳細については、206 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（87 ページの *コードおよびデータの配置（リンカ設定ファイル）*を参照）。

周辺ユニットとの通信

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

標準の周辺ユニットは、デバイス専用の I/O ヘッダファイル（拡張子は h）で定義されています。43 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、220 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、マイクロコントローラはすぐにコードの実行を停止し、その代わりに割込みルーチンの実行を開始します。

コンパイラには、ハードウェアおよびソフトウェア割込みを管理するためのさまざまな基本関数が用意されています。つまり、C で割込みルーチンを記述できるということです（75 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）。

システム起動

すべての組込みシステムでは、アプリケーションの `main` 関数が呼出される前に、システム起動コードが実行され、ハードウェアとソフトウェアの両方のシステムが初期化されます。

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

C/C++ アプリケーションでは、さらに、すべてのグローバル変数を初期化する必要があります。この初期化は、リンカおよびシステム起動コードで扱われます。詳細については、53 ページの*アプリケーションの実行 — 概要*を参照してください。

リアルタイムオペレーティングシステム

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

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

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

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

他のビルドツールとの相互運用

他のベンダのツールで生成されたオブジェクトファイルを使用できますが、RL78 ABI 仕様に従い、C++ ではなく V2 呼出し規約を使用する必要があります。また、Renesas RL78 コンパイラにより作成された Renesas の標準 C ライブラリを使用するモジュールについては、以下の制限があることが分かっています :

- 標準 C 関数の `printf`/`scanf` ファミリのような可変数引数関数は、サポートされていません
- `strtok` はサポートされていません
- IAR コンパイラの Far コードモデルと Near データモデルを使用する必要があります

RL78 ABI は、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。

ビルドプロセス — 概要

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

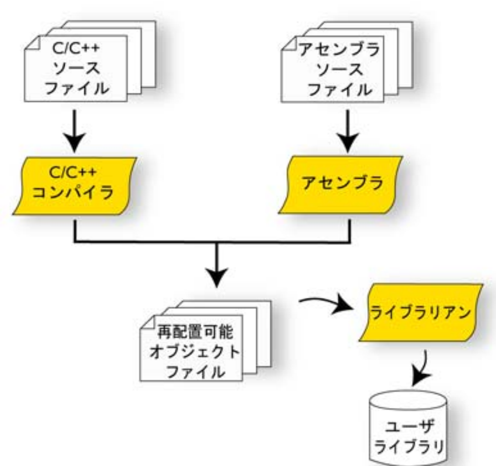
実際のプロセスをより理解できるように、IAR インフォメーションセンタで使用できるチュートリアルを 1 つ以上実行してみてください。

変換プロセス

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

注 : このコンパイラは、C/C++ ソースコードをアセンブラソースコードに変換するときにも使用できます。必要な場合、オブジェクトコードにアセンブルできるアセンブラソースコードを修正できます。IAR アセンブラの詳細については、*『RL78 用 IAR アセンブラリファレンスガイド』*を参照してください。

以下の図は、変換プロセスを示しています。



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

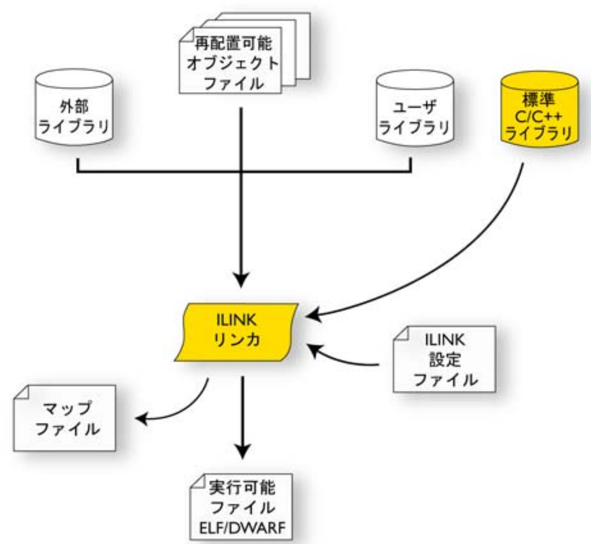
注 : 別のベンダのツールセットにより生成されたモジュールもビルドに含めることができます。ただし、AEABI 準拠でない場合、同じベンダのコンパイラユーティリティライブラリが必要なので注意してください。

最終的なアプリケーションのビルドには、IAR ILINK リンカ (`ilinkrl78.exe`) が使用されます。通常は、リンカでは入力として以下の情報が必要になります。

- いくつかのオブジェクトファイル、場合によっては特定のライブラリ
- プログラムの開始ラベル（デフォルトで設定）

- ターゲットシステムのメモリ内でのコードおよびデータの配置を記述したリンカ設定ファイル

以下の図は、リンク処理を示しています。



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

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

リンカにより実行される手順について詳しくは、85 ページの *リンクプロセスの詳細*を参照してください。

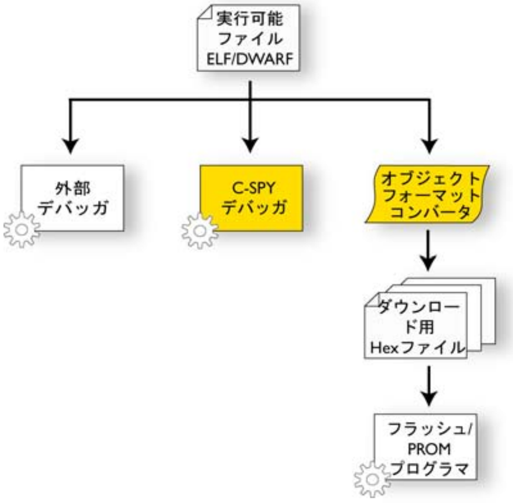
リンク後

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

- IAR C-SPY デバッガ、または ELF や DWARF を読み取るその他の互換性のある外部デバッガへのロード。
- フラッシュ /PROM プログラマを使用したフラッシュ /PROM へのプログラミング。これを実現するには、イメージの実際のバイトを標準の Motorola

32-bit S-record フォーマットまたは Intel Hex-32 フォーマットに変換する必要があります。この変換には、`ielftool` を使用します（413 ページの *IAR ELF ツール — ielftool* 参照）。

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



アプリケーションの実行 — 概要

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

- 初期化フェーズ
- 実行フェーズ
- 終了フェーズ

初期化フェーズ

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

- ハードウェア初期化。通常、少なくともスタックポインタが初期化されます

  ハードウェア初期化は、通常、システム起動コード `cstartup.s` で実行され、必要に応じて、ユーザが提供する最低レベルのルーチンで実行されます。また、ハードウェアの残りの部分のリセット / 起動や、ソフトウェア

C/C++ システム初期化の準備のための CPU などの設定が行われる場合もあります。

- ソフトウェア C/C++ システム初期化

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

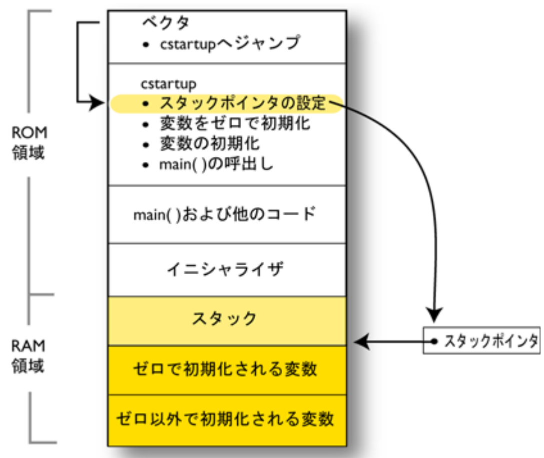
- アプリケーション初期化

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

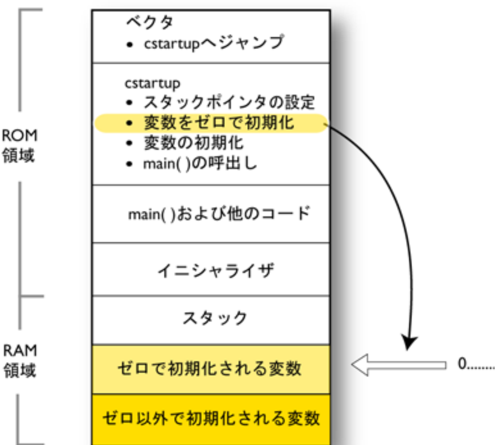
ROM/ フラッシュベースのシステムでは、定数や関数がすでに ROM に配置されています。RAM に配置されたすべてのシンボルは、`main` 関数が呼出される前に初期化される必要があります。また、リンカにより、利用可能な RAM は、すでに変数、スタック、ヒープなどの異なるエリアに分割されています。

以下の一連の図は、初期化の各種段階の概要を簡単に示します。

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、あらかじめ定義されたスタックエリアの最後を指すようにするスタックポインタの初期化など、ハードウェアの初期化を実行します。

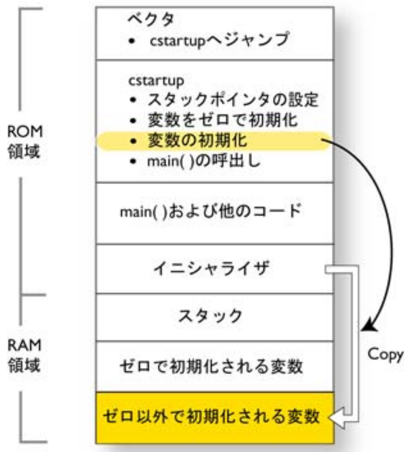


2 次に、ゼロ初期化されるメモリがクリアされます。すなわち、これらのメモリにゼロが埋め込まれます。

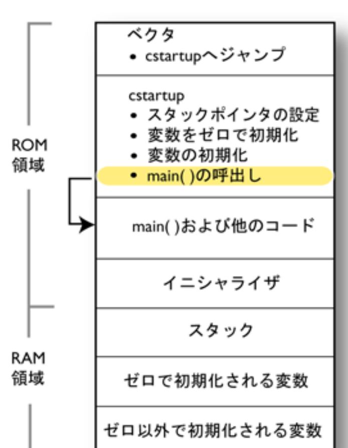


一般的に、これは*ゼロで初期化されるデータ*などのデータで、たとえば `int i = 0;` などで宣言される変数です。

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



4 最後に、main 関数が呼出されます。



各段階の詳細については、122 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、90 ページの *システム起動時の初期化*を参照してください。

実行フェーズ

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

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

終了フェーズ

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

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

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

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

これらの詳細は、124 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。

```
iccrl78 myfile.c
```

また、重要なオプションもいくつか指定する必要があります（57 ページの*基本的なプロジェクト設定*を参照）。

コマンドラインで以下のコマンドを実行すると、リンカが起動します。

```
ilinkrl78 myfile.o myfile2.o -o a.out --config my_configfile.icf
```

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

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

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

基本的なプロジェクト設定

ここでは、使用する RL78 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。

- コア
- データモデル
- コードモデル
- `double` 浮動小数点型のサイズ
- 最適化設定
- ランタイム環境
- ILINK 設定のカスタマイズ（「*アプリケーションのリンク*」を参照）。

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

プロセッサコア

コンパイラは、以下の RL78 ファミリのプロセッサコアをサポートしています。

コア	説明
S1	レジスタバンク 1 つと多重化された 8 ビットバスを 1 つ持つ S2 コア。
S2	ハードウェア乗算 / 除算をサポートする命令なし。ハードウェア乗算 / 除算の使用は、I/O レジスタアクセスによりサポートされています。
S3 （デフォルト）	ハードウェア乗算 / 除算をサポートする命令あり。

表 3: プロセッサコア

プロジェクトにプロセッサコアを指定するには、`--core` オプションを使用します（244 ページの *--core* を参照）。

注 : デフォルトでは、コンパイラは、バンク 0 を使用して、自動的にバンクを切替えることはありません。コンパイラでバンク 1–3 を使用させるには、`#pragma bank` ディレクティブを使用する必要があります。（327 ページの *bank* を参照）

データモデル

コンパイラは、データ要件の異なるアプリケーションのコード作成を容易にするために、2 つのデータモデルを使用します。

- *Near* データモデルはデータメモリ内の最高 64 KB のデータにアクセスできます。

- *Far* データモデルは、データモデル内の 1 MB 全体のデータにアクセスできます。

データモデルの詳細は、章*データ記憶*を参照してください。

コードモデル

コンパイラは、2 つのコードモデルを使用します。これらはファイルまたは関数レベルで設定して、デフォルトでどの関数を呼出すかを制御することができます。

- *Near* コードモデルの上限は 64 KB です。
- *Far* コードモデルは 1 MB メモリ全体にアクセスできます。

データモデルの詳細は、章*関数*を参照してください。

DOUBLE 浮動小数点型のサイズ

浮動小数点の値は、標準の IEEE 754 フォーマットで 32 ビットおよび 64 ビットの数字で表されます。コンパイラオプション `--double={32|64}` を使用する場合、`double` として宣言されたデータを 32 ビットと 64 ビットのどちらで表すかを選択できます。データ型 `float` は、常に 32 ビットを使用して表されます。

速度とサイズの最適化

コンパイラのオプティマイザは、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、精度調整などを実行します。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ライドすることも必要となります。提供されるランタイムライブラリは、IAR DLIB ライブラリです。

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

IDE でのランタイム環境の設定

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

DLIB ライブラリには、異なる設定（ノーマルとフル）があり、これにはロケール、ファイル記述子、マルチバイト文字など異なるレベルのサポートが含まれます（126 ページの *ライブラリ構成*を参照）。

コマンドラインのランタイム環境の設定

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

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

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

```
-I MyApplication¥inc
```

ビルド済のライブラリオブジェクトファイルの詳細については、111 ページの *ビルド済ライブラリの使用*を参照してください。

ライブラリとランタイム環境オプションの設定

オプションの設定により、ライブラリやランタイム環境のサイズを削減できます。

- 関数 `printf`、`scanf`、およびこれらの派生関数で使用されるフォーマッタ。114 ページの *printf*、*scanf のフォーマッタの選択* を参照してください。
- スタックやヒープのサイズ。99 ページの *スタックメモリの設定*、100 ページの *ヒープメモリの設定*を参照してください。

データ記憶

- 概要
- メモリタイプ
- データモデル
- 自動変数とパラメータの記憶領域
- ヒープ上の動的メモリ

概要

コンパイラは、`0x00000` から `0xFFFFF` まで 1MB のシーケンシャルメモリを持つ RL78 デバイスをサポートしています。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。

すべてのコンパイラは、異なる方法でメモリにアクセスできます。アクセス方法は、汎用性がありメモリ空間全体にアクセス可能でコストがかかるものから、ショートアドレスメモリ領域にアクセスできる安価な方法までいろいろあります。これらの詳細は、62 ページの*メモリタイプ*を参照してください。

さまざまなデータ記憶方法

一般的な用途では、データを以下の 3 種類の方法でメモリに格納できます。

- 自動変数

  static として宣言された変数を除き、関数にローカルな変数はすべて、レジスタまたはスタックに格納されます。これらの変数は、関数の実行中にアクセスが可能です。関数が呼出し元に戻ると、このメモリ空間は無効になります。詳細については、69 ページの*自動変数とパラメータの記憶領域*を参照してください。

- グローバル変数、モジュール静的変数、`static` と宣言されたローカル変数

  この場合、メモリは 1 度だけ割り当てられます。ここでの「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。詳細については、68 ページの*データモデル*、62 ページの*メモリタイプ*を参照してください。

- 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、71 ページの*ヒープ上の動的メモリ*を参照してください。

メモリタイプ

ここでは、コンパイラでのデータアクセスに使用される*メモリタイプ*の概念について説明します。複数のメモリタイプがある場合のポインタについても解説します。各メモリタイプについて、機能や制限を説明します。

メモリタイプの概要

コンパイラは、さまざまなメモリタイプを使用して、異なるメモリの領域に配置されたデータにアクセスします。メモリ領域にアクセスする方法はさまざまです。コード空間や実行速度、レジスタの使用を考えると、コストが変わってきます。アクセス方法は、汎用性がありメモリ空間全体にアクセス可能でコストがかかるものから、一部のアドレスメモリ領域にアクセスできる安価な方法までいろいろあります。各メモリタイプは、1 つのメモリアクセス方法に対応しています。異なるメモリ（またはメモリの一部）をメモリタイプに割り当てる場合、コンパイラはデータに効率的にアクセス可能なコードを生成できます。

たとえば、Near アドレスを使用するメモリアクセス方法、Near メモリと呼ばれます。

アプリケーションで使用するデフォルトのメモリタイプを選択するには、*データモデル*を選択します。ただし、（個々の変数やポインタに）異なるメモリタイプを指定することが可能です。このため、大量のデータを保有するアプリケーションを作成して、同時に頻繁に使用される変数を効率的にアクセス可能なメモリ内に配置するのは不可能となります。

次に変数メモリタイプの概要を示します。

saddr

ショートアドレッシングが可能な領域 (saddr) は、アドレス範囲 `0xFFE20–0xFFF1F` の 256 バイトのデータメモリから構成されます。

saddr メモリにだけ saddr オブジェクトを配置できます。このタイプのオブジェクトを使用することで、それらにアクセスするためにコンパイラで生成されるコードが最小化されます。つまり、アプリケーションのフットプリントが小さくなり、ランタイムで実行が速くなります。

70 ページの*ショートアドレス作業エリア*も参照してください。

near

near メモリは、64KB のデータメモリからなります。16 進数表記では、これはアドレス範囲 `0xF0000-0xFFFFF` です。

near メモリにだけ near オブジェクトを配置できます。

far

far メモリは、データメモリ 1MB 全体から構成されます。16 進数表記では、これはアドレス範囲 `0x00000–0xFFFFF` です。

far オブジェクトは far メモリ内にのみ配置でき、これらのオブジェクトのサイズは 64 KB に制限されます。データメモリに 64KB 以上のオブジェクトを配置するには、データメモリ属性 `__huge` をオブジェクトと使用する必要があります。

sfr

特殊機能レジスタ (SFR) 領域は、アドレス範囲 `0xFFF00–0xFFFFF` にあります。

特殊機能レジスタは、デバイス固有のヘッダファイルで定義します。

データメモリ属性の使用

コンパイラには、*データメモリ属性*として使用可能な*拡張キーワード*が用意されています。これらのキーワードを使用すると、個々のデータオブジェクトとポインタについてデフォルトのメモリタイプをオーバライドできます。つまり、デフォルトのメモリ以外の他のメモリ領域にデータオブジェクトを配置することができます。これは、個々のデータオブジェクトへのアクセス方法を微調整でき、コードサイズが小さくなることを意味します。

以下の表は、使用可能なメモリタイプとそれに対応するキーワードを示します。

メモリタイプ	キーワード	アドレス範囲	ポインタサイズ
Saddr	`__saddr`	`0xFFE20-0xFFF1F`	2 バイト

表 4: メモリタイプと対応するキーワード

メモリタイプ	キーワード	アドレス範囲	ポインタサイズ
Near（デフォルト）	`__near`	`0xF0000-0xFFFFF`	2 バイト
Far	`__far` または `__huge`	`0x00000-0xFFFFF`	3 バイト
SFR	`__sfr`	`0xFFF00-0xFFFFF`	なし

表 4: メモリタイプと対応するキーワード（続き）

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。詳細は「251 ページの *-e*」を参照してください。

各キーワードの詳細については、314 ページの*拡張キーワードの詳細*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

型属性は構文規則の型修飾子 `const` と `volatile` とほぼ同じように使用します。次に例を示します。

```
__near int i;
int __near j;
```

`i` と `j` の両方は near メモリに配置されます。

`const` や `volatile` とは異なり、構造体メンバの宣言の場合を除いて、型属性は派生した型の型指定子の前に使用されるとき、型属性がオブジェクトまたは typedef 自体に適用されます。

```
int __near * p;        /* near メモリの整数 */

int * __near p;        /* near メモリのポインタ */

__near int * p;        /* near メモリのポインタ */
```

すべての場合で、メモリ属性が指定されていない場合、適切なデフォルトのメモリタイプが使用されます。それは使用するデータモデルによって異なります。

型定義を使用することはコードをより明確にできる場合があります。

```
typedef __near d16_int;
d16_int *q1;
```

d16_int near メモリの変数のための typedef です。変数 q1 はそのような整数を指し示すことができます。

#pragma type_attributes ディレクティブを使用して宣言の型属性を指定することもできます。プラグマディレクティブで指定した型属性は、データオブジェクトまたは制限される typedef に適用されます。

```
#pragma type_attribute=__near
int * q2;
```

q2 変数は near メモリに配置されます。

メモリ属性の他の使用例については、66 ページのその他の例を参照してください。

タイプの定義

記憶はタイプ定義を使用しても指定することができます。以下の 2 つの宣言は同等です。

```
/* typedef を介して定義 */
typedef char __far Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;

/* 直接的に定義  */
__far char aByte;
char __far *aBytePointer;
```

ポインタとメモリタイプ

ポインタは、データの位置を参照するときに使用されます。一般的にポインタには 1 つのタイプがあります。たとえば、タイプ int * のポインタは整数をポイントします。

コンパイラでは、ポインタは何らかのタイプのメモリもポイントします。アスタリスクの前のキーワードを使用してメモリタイプを指定します。たとえば、near メモリに格納されている整数をポイントするポインタは次のもので宣言されます。

```
int __near * MyPtr;
```

ポインタ変数 MyPtr の場所はキーワードには影響しません。ただし次の例では、ポインタ変数 MyPtr2 は near メモリにあります。MyPtr のように、MyPtr2 は far メモリの文字をポイントします。

```
char __far * __near MyPtr2;
```

たとえば、標準ライブラリの関数は、すべてメモリタイプを明示しないで宣言されます。

ポインタタイプのちがい

ポインタには、あるメモリタイプのメモリの場所を指定するために必要な情報が含まれている必要があります。これはポインタサイズが異なるメモリタイプで異なることを意味します。RL78 用 IAR C/C++ コンパイラには、near と far/huge ポインタのサイズは、それぞれ 16 と 24 ビットです。

ポインタの詳細については、301 ページのポインタ型を参照してください。

構造体とメモリタイプ

構造体の場合、オブジェクト全体が同じメモリタイプ内に配置されます。個々の構造体メンバを異なるメモリタイプに配置することはできません。

次の例では、変数 `gamma` は near メモリ内に配置された構造体です。

```
struct MyStruct
{
  int mAlpha;
  int mBeta;
};

__near struct MyStruct gamma;
```

この宣言は正しくありません：

```
struct MyStruct
{
  int mAlpha;
  __near int mBeta; /* 誤った宣言 */
};
```

その他の例

以下は一連の例と説明です。まず、整数の変数が定義され、ポインタ変数が導入されます。最後に、near メモリ内の整数へのポインタを受け入れる関数

が宣言されます。関数は、far メモリ内の整数へのポインタを返します。メモリ属性がデータタイプの前と後ろのどちらであっても、違いはありません。

`int MyA;`	使用中のデータモデルによって決定される、デフォルトメモリ内に定義された変数。
`int __near MyB;`	near メモリ内の変数。
`__far int MyC;`	far メモリ内の変数。
`int * MyD;`	デフォルトのメモリ内に保存されたポインタ。ポインタはデフォルトのメモリ内の整数をポイントします。
`int __near * MyE;`	デフォルトのメモリ内に保存されたポインタ。ポインタは、near メモリ内の整数をポイントします。
`int __near * __far MyF;`	far メモリに記憶された、near メモリ内に保管された整数をポイントするポインタ。
`int __far * MyFunction(` `   int __near *);`	near メモリ内に記憶された整数へのポインタであるパラメータを受け入れる関数の宣言。関数は、far メモリ内に記憶された整数へのポインタを返します。

C++ とメモリタイプ

C++ クラスのインスタンスは、（他のオブジェクトと同じように）メモリ属性や IAR 言語拡張を使用して、暗黙的または明示的にメモリに配置されます。非静的なメンバ変数は、構造体のフィールドと同じように大きなオブジェクトの一部であり、個別に指定のメモリに配置することはできません。

非静的なメンバ関数では、C++ オブジェクトの非静的メンバ変数は `this` ポインタを介して、暗黙的または明示的に参照が可能です。`this` ポインタは、クラスメモリを使用する場合を除いて、デフォルトのデータポインタ型です（179 ページの *IAR 属性とクラスを使用する*を参照）。

静的メンバ変数は、解放された変数と同じようにデータメモリに個別に配置可能です。

コンストラクタ、デストラクタ以外のすべてのメンバ関数は、解放された関数のように同じ方法でコードメモリに個々に配置されます。

C++ クラスについて詳しくは、179 ページの *IAR 属性とクラスを使用する*を参照してください。

データモデル

*データモデル*を使用して、コンパイラでメモリのどの部分に静的変数およびグローバル変数をデフォルトで配置するかを指定します。つまり、データモデルは以下の内容を制御します。

- デフォルトのメモリタイプ
- 静的変数およびグローバル変数、定数リテラルのデフォルトの配置
- 動的に割り当てられるデータ。たとえば、`malloc` や演算子 `new`（C++ の場合）によって割り当てられたデータです
- デフォルトのポインタタイプ
- 実行時のスタックの配置

データモデルは、デフォルトのメモリタイプのみを指定します。個々の変数やポインタについて、これをオーバライドすることが可能です。個々のオブジェクトについてメモリタイプを指定する方法については、63 ページの*データメモリ属性の使用*を参照してください。

注 : どのデータモデルを選択しても、コードの配置には影響しません。

データモデルの指定

プロジェクトで同時に使用できるデータモデルは 1 つだけです。また、すべてのユーザモジュールとライブラリモジュールで、同一のモデルを使用する必要があります。ただし、明示的にメモリ属性を指定することによってのみ、個々のデータオブジェクトとポインタのデフォルトのメモリタイプをオーバライドできます（63 ページの*データメモリ属性の使用*を参照）。

以下の表は異なるデータモデルの概要です。

データモデル名	デフォルトのメモリおよびポインタ属性	データ配置
Near（デフォルト）	`__near`	最上位 64 KB
Far	`__far`	1MB のメモリ全体

表 5: データモデルの特徴

IDE でのオプション設定については、*『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。

プロジェクトにデータモデルを指定するには、`--data_model` オプションを使用します（245 ページの *--data_model* を参照）。

Near データモデル

Near データモデルは、メモリ内の最上位 64 KB にデータを配置します。このメモリには、2 バイトのポインタを使用してアクセスできます。つまり、ポ

インタの保存に必要なのは 16 ビットのみということです。パラメータとして送信されるデフォルトのポインタタイプでは、16 ビットのレジスタかスタック上の 2 バイトが使用されます。

Far データモデル

Far データモデルは、メモリ内の最初の 1MB にデータを配置します。3 バイトのポインタを使用してアクセス可能なのは、このメモリのみです。パラメータとして送信されるデフォルトのポインタタイプは、スタック上の 4 バイトを使用します。

自動変数とパラメータの記憶領域

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

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

スタック

スタックには、以下を格納できます。

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

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

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

194 ページのスタックについておよび 99 ページのスタックメモリの設定を参照してください。

利点

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

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

潜在的な問題

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

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

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

ショートアドレス作業エリア

`--workseg_area` オプションを使用して、レジスタ変数作業領域のために saddr メモリにスペースを予約できます（271 ページの *--workseg_area* を参照）。最も頻繁に使用されるパラメータおよび自動変数は、指定した最大サイズまでこの領域に格納されます。

`setjmp` 関数の呼出しと、それに対応する `longjmp` 呼出しの間に呼び出された関数の workseg 領域は、使用を避けてください。この workseg 領域は破損する可能性があります。

すべての関数の入り口と終了位置で作業領域を保存して復元すると、オーバヘッドが関係してきます。このオーバヘッドは、`__no_save` 関数属性を使用して関数を宣言すると省略できます。

workseg 領域は `.wrkseg` リンカセクションに配置されます（408 ページの *.wrkseg* を参照）。

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。

コンパイラは、1 つのデータメモリ以上のヒープをサポートします。これらの詳細は、100 ページの*ヒープメモリの設定*を参照してください。

潜在的な問題

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

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

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

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

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

関数

- 関数関連の拡張
- 関数格納のためのコードモデルとメモリ属性
- 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
- インライン関数

関数関連の拡張

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

- メモリ内の関数の記憶領域を制御
- 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そ使用
- 関数インライン化の制御
- 関数の最適化を円滑化
- ハードウェア機能にアクセス

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

最適化の詳細は、203 ページの*組込みアプリケーション用の効率的なコーディング*を参照してください。ハードウェア操作のアクセスに使用可能な組込み関数の詳細については、「*組込み関数*」を参照してください。

関数格納のためのコードモデルとメモリ属性

*コードモデル*を使用して、コンパイラでアプリケーションのコードをどのように生成するかを制御します。

プロジェクトで同時に使用できるコードモデルは 1 つだけです。また、すべてのユーザモジュールとライブラリモジュールで、同一のモデルを使用する必要があります。すべてのコードモデルは、ROM 対応のコードを生成します。また、コードモデルでは関数を格納するデフォルトのメモリ範囲も指定します。

注 : どのコードモデルを選択しても、データの配置には影響しません。

以下のコードモデルが使用可能です :

Near（デフォルト）	関数の呼出しがメモリの最初の 64KB に達します。
Far	関数の呼出しが 1MB のメモリ全体に達します。

表 6: コードモデル

コードモデルを指定しない場合、コンパイラは Near コードモデルをデフォルトとして使用します。

IDE でのコードモデルの設定については、オンラインヘルプシステム を参照してください。

プロジェクトにデータモデルを指定するには、`--code_model` オプションを使用します（243 ページの *--code_model* を参照）。

関数メモリ属性の使用

個々の関数のデフォルトの配置をオーバライドすることが可能です。これを指定するには、適切な*関数メモリ属性*を使用します。以下の属性を選択できます。

関数メモリ属性	アドレス範囲	ポインタサイズ	説明
`__callt`	`0-0x0FFFF`	2 バイト	ベクタ `CALLT` 命令を使用して、メモリ内のどこからでも関数を呼び出せます。
`__near_func`	`0-0x0FFFF`	2 バイト	メモリ内のどこからでも関数を呼び出せます。これは Near コードモデルのデフォルト属性です。
`__far_func`	`0-0xFFFFF`	3 バイト	メモリ内のどこからでも関数を呼び出せます。これは Far コードモデルのデフォルト属性です。

表 7: 関数メモリ属性

関数メモリ属性を持つポインタには、ポインタ間およびポインタと整数型の間の明示的および非明示的なキャスティングについて制限があります。この制限の詳細は、302 ページの *キャスト*を参照してください。

構文および各属性の詳細については、*拡張キーワード*を参照してください。

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

RL78 用 IAR C/C++ コンパイラは、割込み関数、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：`__interrupt`、`__monitor`
- プラグマディレクティブ：`#pragma vector`、`#pragma bank`
- 組込み関数：`__enable_interrupt`、`__disable_interrupt`、`__get_interrupt_state`、`__set_interrupt_state`

割込み関数

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

割込みサービスルーチン

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

RL78 マイクロコントローラは、多くの割込みソースをサポートしています。割込みソースごとに、割込みルーチンを記述できます。各割込みルーチンは、チップメーカの RL78 マイクロコントローラのドキュメントで指定されているベクタ番号に関連付けられます。同じ割込みルーチンを使用していくつかの異なる割込みを処理する場合、割込みベクタを複数指定できます。

割込みベクタと割込みベクタテーブル

RL78 マイクロコントローラには、割込みベクタテーブルは `0x0` アドレスから常に始まり、`.intvec` セクションに配置されます。割込みベクタは、割込みベクタテーブルへのオフセットです。

ヘッダファイル `io`*`device`*`.h`（*`device`* は選択したデバイス）には、事前定義した既存の割込みベクタ名が含まれます。

割込み関数の定義 — 例

割込み関数を定義するには、__interrupt キーワードと #pragma vector ディレクティブを使用できます。次に例を示します。

```
#pragma vector = INTP2_vect /* I/O ヘッダファイルに定義されたシンボル
*/
__interrupt void MyInterruptRoutine(void)
{
  /* 何らかの処理 */
}
```

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

割込みと C++ メンバ関数

static メンバ関数だけが割込み関数になれます。

モニタ関数

モニタ関数は、関数の実行中に割込みを無効にします。関数のエントリで、ステータスレジスタが保存されて割込みが無効になります。関数の終了時には、元のステータスレジスタが復元され、これによって関数呼出し前に存在した割込みステータスも復元されます。

モニタ関数を定義するには、__monitor キーワードを使用できます。詳細については、317 ページの *__monitor* を参照してください。

割込みが長く無効のままになるため、__monitor キーワードを大きな関数に使用することは避けてください。

C でのセマフォの実装の例

次の例では、バイナリセマフォ（ミューテックス）が、静的変数 1 つと 2 つのモニタ関数を使用して実装されます。モニタ関数は重要な領域と同じように機能します。つまり、割込みが発生することはできず、プロセス自体がスワップできません。セマフォは 1 つのプロセスによりロックでき、一度に 1 つのプロセスでしか使用できないリソースをプロセスで同時に使用するのを回避するために使用されます。たとえば、USART などです。__monitor キーワードによって、ロック処理がアトミックであることが確実になります。アトミックを言い換えると、割込みができないということです。

```
/* これはロック変数です。ゼロでない場合、所有者がいます。*/
static volatile unsigned int sTheLock = 0;


/* ロックがオープンかどうかをテストする関数で、オープンの場合はロックを取得
 * します。成功の場合を 1、失敗のときは 0 を返します。
 */

__monitor int TryGetLock(void)
{
  if (sTheLock == 0)
  {
    /* 成功。ロックを誰も所有していません。*/

    sTheLock = 1;
    return 1;
  }
  else
  {
    /* 失敗。誰かがロックを所有しています。*/

    return 0;
  }
}


/* ロックを解放する関数。
 * ロックを持つユーザーからしか呼出せません。
 */

__monitor void ReleaseLock(void)
{
  sTheLock = 0;
}


/* ロックを取得する関数。取得するまで待機します。*/

void GetLock(void)
{
  while (!TryGetLock())
  {
    /* 通常はここでスリープ命令が使用されます。*/
  }
}
```

```
/* セマフォの使用例 */

void MyProgram(void)
{
  GetLock();

  /* 何らかの処理 */

  ReleaseLock();
}
```

C++ でのセマフォの実装の例

C++ では、インライン化する意図で小さいメソッドを実装することは一般的です。ただし、コンパイラは `__monitor` キーワードを使用して宣言された関数およびメソッドのインライン化をサポートしていません。

次の C++ の例では、自動オブジェクトを使用してモニタブロックが制御され、これは `__monitor` キーワードではなく、組込み関数を使用します。

```
#include <intrinsics.h>

// 重要ブロックを制御するためのクラス

class Mutex
{
public:
  Mutex()
  {
    // 現在の割込み状態を取得
    mState = __get_interrupt_state();

    // すべての割込みを無効化
    __disable_interrupt();
  }

  ~Mutex()
  {
    // 割込み状態を復元します。
    __set_interrupt_state(mState);
  }

private:
  __istate_t mState;
};
```

```
class Tick
{
public:
  // チックカウントを安全に読み込むための関数
  static long GetTick()
  {
    long t;

    // 重要ブロックを入力
    {
      Mutex m; // m がスコープ内にある間は割込みは無効

      // テックカウント安全に取得して
      t = smTickCount;
    }
    // それを返す
    return t;
  }

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;

  if (Tick::GetTick() >= nextStop)
  {
    nextStop += 100;
    DoStuff();
  }
}
```

インライン関数

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

に行われるかどうかは、コンパイラのヒューリスティックに基づいて決定されます。

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

C と C++ の動作の比較

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

C では、インライン関数のインライン化されていないバージョンを含むコンパイル単位を手動で 1 つ選択する必要があります。これは、そのコンパイル単位内で関数を extern として明示的に宣言することにより行います。複数のコンパイル単位で関数を extern として宣言すると、リンカは*複数定義*エラーを出力します。また、C ではインライン関数は静的変数や関数を参照できません。

次に例を示します。

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


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

関数のインライン化を制御する機能

関数のインライン化を制御するしくみはいくつかあります。

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

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

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

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

  `#pragma inline` ディレクティブを使用することで、コンパイラのヒューリスティックを無効化して、インライン化を強制するか、完全に無効にすることができます。詳細については、335 ページの *inline* を参照してください。
- `--use_c++_inline` を使用すると、標準の C ソースコードファイルをコンパイルする際に C++ の動作を使用するようコンパイラに強制的に指示します。
- `--no_inline`、`#pragma optimize=no_inline`、`#pragma inline=never` は、いずれも関数のインライン化を無効にします。デフォルトでは、関数のインライン化は最適化レベル［高］で有効になっています。

コンパイラは、定義が分かっている場合にのみ関数をインライン化することができます。これは通常、現在の翻訳単位にのみ制限されています。ただし、複数ファイルのコンパイルの `--mfc` コンパイラオプションが使用される場合、コンパイラは複数ファイルコンパイル単位のすべてのコンパイル単位からの定義をインライン化できます。詳細については、211 ページの*複数ファイルのコンパイルユニット*を参照してください。

関数のインライン化の最適化の詳細については、214 ページの*関数インライン化*を参照してください。

ILINK を使用したリンク

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

リンクの概要

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

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

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

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

ILINK が作成する最終出力は、ELF（デバッグ情報の DWARF を含む）形式の実行可能なイメージを含む、絶対オブジェクトファイルです。このファイルは、C-SPY のほか ELF/DWARF をサポートする互換性のあるデバッガにダウ

ンロードできます。あるいは、EPROM またはフラッシュに格納することができます。

ELF ファイルを使用するために、さまざまなツールが提供されています。付属のユーティリティについては、42 ページの*専用 ELF ツール*を参照してください。

モジュールおよびセクション

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

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

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

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

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

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

セクションは、最小のリンク可能ユニットです。ただし、可能な場合、ILINK は、最終アプリケーションからさらに小さいユニット（セクションフ

ラグメント）を実行できます。詳細については、98 ページの*モジュールの保持*、99 ページの*シンボルおよびセクションの保持*を参照してください。

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

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

ブロックを使用して配置するためにセクションをまとめてグループ化できます。378 ページの *define block ディレクティブ*を参照してください。

リンクプロセスの詳細

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

注 : 別のベンダのツールセットで生成されたモジュールも同様にビルドに含めることができます。ただし、モジュールが RL78 ABI (RL78 Application Binary Interface) 準拠の場合に限ります。RL78 ABI 準拠でない場合、同じベンダのコンパイラユーティリティライブラリが必要なので注意してください。

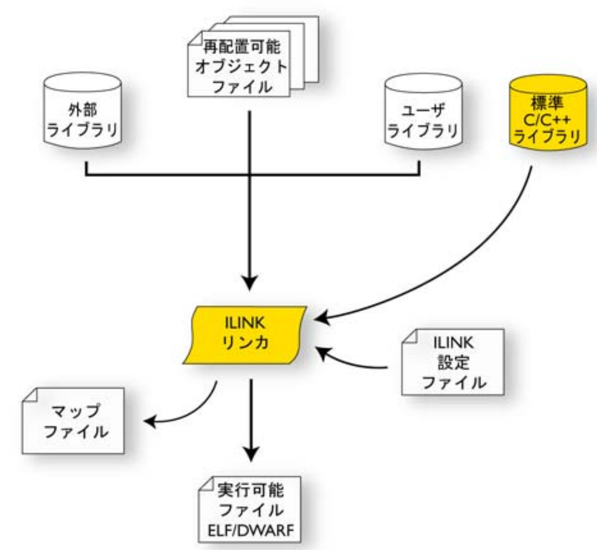
リンカはリンクプロセスに使用されます。通常、以下の手順を実行します（一部の手順は、コマンドラインオプションやリンカ設定ファイルのディレクティブによって無効化できます）。

- アプリケーションに含めるモジュールを判別する。オブジェクトファイルで提供されるモジュールは、常に含まれます。ライブラリファイルのモジュールは、インクルードされるモジュールから参照されるグローバルシンボルの定義を与えるものだけが含まれます。
- 使用する標準ライブラリファイルを選択する。選択は、インクルードするモジュールの属性に基づいて行われます。これらのライブラリは、依然として解決されていないすべての未定義シンボルを充足するために使用されます。
- 複数の定義を持つシンボルを処理します。Weak ではない定義が複数あると、エラーが出力されます。それ以外の場合、いずれかひとつの定義が選択され（weak でない定義がある場合はそれが選択されます）、その他は無効化されます。Weak 定義は通常、インライン関数およびテンプレート関数に使用されます。ライブラリモジュールからの weak でない定義のいくつかをオーバライドする必要がある場合は、ライブラリモジュールがインクルードされていないことを確認してください（通常は、そのライブラリ

モジュールでアプリケーションが使用するすべてのシンボルについて、代替の定義を指定します)。

- 追加したモジュールのうちアプリケーションに含めるセクション/セクションフラグメントを判別する。アプリケーションで実際に必要なセクション/セクションフラグメントのみが含まれます。必要なセクション/セクションフラグメントを判別する方法は、いくつかあります。たとえば、`__root` オブジェクト属性、`#pragma required` ディレクティブ、`keep` リンカディレクティブなどが使用できます。セクションが重複する場合は、1 つのみ含まれます。
- 必要に応じて、RAM 内の初期化変数およびコードの初期化を実行する。`initialize` ディレクティブを使用すると、リンカによって追加のセクションが作成され、ROM から RAM へのコピーが可能になります。コピーによって初期化される各セクションは、2 つのセクションに分割され、1 つが ROM パート、もう 1 つが RAM パートに使用されます。手動の初期化を使用しない場合、初期化を実行するための起動コードもリンカで作成されます。
- *リンカ設定ファイル*のセクション配置ディレクティブに従って、各セクションの配置場所を判別する。コピーによって初期化されるセクションは、配置ディレクティブに照らして ROM パート用と RAM パート用の 2 か所あり、それぞれ異なる属性を持ちます。
- 実行可能イメージおよび提供されたすべてのデバッグ情報を含む絶対ファイルを生成する。再配置可能な入力ファイルの必要な各セクションの内容は、そのファイルおよびセクションの配置時に決定されたアドレスで提供された再配置情報を使用して計算されます。このプロセスによって、特定セクションの要件の一部が満たされないと、1 つまたは複数の再配置エラーとなることがあります。たとえば、配置によって PC 関連のジャンプ命令の目的地のアドレスが、その範囲外となる場合などです。
- セクション配置の結果、各グローバルシンボルのアドレス、各モジュールおよびライブラリ用のメモリ使用量のサマリをリストするマップファイルを生成する（オプション)。

以下の図は、リンク処理を示しています。



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

注：ELF オブジェクトファイルの実際の内容を確認するには、`ielfdumprl78` を使用します。414 ページの *IAR ELF Dumper — ielfdump* を参照してください。

コードおよびデータの配置（リンカ設定ファイル）

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

一般的なリンカ設定ファイルには、以下の定義が含まれます。

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

- 作成されるセクション
- 使用できる領域にセクションを配置する方法

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

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

設定ファイルの簡単な例

以下のメモリ条件を持つ単純な 32 ビットのアーキテクチャを想定します。

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

ここで想定するアーキテクチャの単純な構成ファイルは、以下のようになります。

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

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

/* スタックを定義 */
define block STACK with size = 0x1000, alignment = 2 { };

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

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

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

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

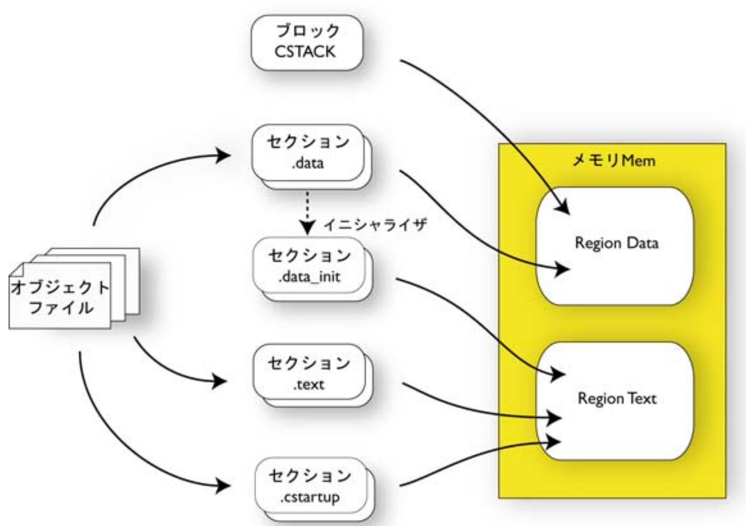
次に、このファイルは、アプリケーションスタックが常駐する STACK という名前のサイズ 4KB の空のブロックを作成します。*ブロック*を作成するのが、配置やサイズなどを詳細に制御する基本的な方法です。この方法を使用してセクションをグループ化したり、この例にあるように、メモリエリアのサイズと配置を指定することもできます。

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

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

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

以下の図は、アプリケーションがメモリにどのように配置されるかを示します。



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

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

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

リンカ設定ファイルの詳細は、「*リンカ設定ファイル*」を参照してください。

システム起動時の初期化

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

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

コンパイラは、変数初期化の各型に対して、特定の型のセクションを生成します。

宣言データのカテゴリ	ソース	セクション型	セクション名のエレメント *	セクションの内容
ゼロで初期化されるデータ	int i;	リード / ライトデータ、ゼロ初期化	.bss	なし
ゼロで初期化されるデータ	int i = 0;	リード / ライトデータ、ゼロ初期化	.bss	なし
初期化されるデータ（ゼロ以外）	int i = 6;	リード / ライトデータ	.data	イニシャライザ
非初期化データ	__no_init int i;	リード / ライトデータ、ゼロ初期化	.bss.noinit	なし
定数	const int i = 6;	リードオンリーデータ	.const	定数

表 8: 初期化データを保持するセクション

* セクション名には、このエレメントのほかにセクションに関連するメモリを説明する文字が含まれます : f(far)、h(huge)、s(saddr) のようになります。1 この文字は、次の表に説明するセクション名のエレメントの前後どちらにつけても問題ありません。near メモリの場合は、このような追加の文字はありません。セクション名の詳細は、397 ページの *セクションの概要*を参照してください。

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

初期化プロセス

データの初期化は、ILINK およびシステム起動コードで扱われます。

変数の初期化を設定するには、以下のことを考慮する必要があります。

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

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

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

リンカ設定ファイルでは、次のように定義されています。

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

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

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

注 : 圧縮されたイニシャライザが使用される場合（380 ページの *initialize ディレクティブ*を参照）、内容（イニシャライザ）のセクション（つまり、サフィックス _init の付いたセクション）はマップファイルで個別のセクションとしてはリストされません。その代わりに、これらは「イニシャライザバイト」の集合に組込まれます。内容のセクションを通常のようにリンカ設定ファイルに配置できますが、こうすることでイニシャライザバイトの集合の配置（およびその番号）に影響が出ます。

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

C++ 動的初期化

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

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

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

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

387 ページの *section-selectors* も参照してください。

コンパイラにより生成された追加のセクション

コンパイラは内部で生成された一連のセクションを使用します。これらは、アプリケーションの動作に重要な情報を格納するために使用されます。

- 切替えライブラリルーチンで使用されるデータ文を格納する .switch セクションと .switchf セクション。これらのテーブルは、なるべく少ない空間を使用するようにエンコードされています。
- --workseg_area オプションによって予約された空間と浮動小数点演算で使用される空間を保持する .wrkseg セクション。

リンカ設定ファイルでは、次のように定義されています。

```
"ROMPAGE":place in ROM_page        { ro section .const,
                                     ro section .switch  };
```

アプリケーションのリンク

- リンクについて
- トラブルシューティングについてのヒント

リンクについて

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

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

リンカ設定ファイルの選択

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

リンカ設定ファイルを編集するには、IDE のエディタ、またはその他の適切なエディタを使用します。

元のテンプレートファイルは変更しないでください。作業ディレクトリにコピーを作成し、そのコピーを修正することをお勧めします。

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

独自のメモリエリアの定義

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

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

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

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

各領域定義は、実際のハードウェアに合わせて調整する必要があります。

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

領域の追加

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

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

異なるエリアを 1 つの領域にマージする

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

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

以下の例も同じです。

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

セクションの配置

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

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

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

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

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

セクションをメモリの特定のアドレスに配置する

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

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

セクションを領域の開始または終了位置に配置する

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

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

独自のセクションの宣言および配置

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

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

以下はアセンブラ言語の場合の例です。

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

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

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

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

RAM の空間の予約

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

リンカ設定ファイルでは、次のように定義されています。

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

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

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

モジュールの保持

モジュールがオブジェクトファイルとしてリンクされている場合、これは常に保持されます。つまり、モジュールは、リンクされたアプリケーションに含まれます。ただし、モジュールがライブラリの一部の場合、モジュールが含まれるのは、アプリケーションの他の部分からシンボルで参照されている場合のみです。これは、ライブラリモジュールにルートシンボルが含まれている場合でも同様です。このようなライブラリモジュールが常に確実に含ま

れるようにするには、iarchive を使用してライブラリからモジュールを抽出します（409 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、84 ページの*モジュールおよびセクション*を参照してください。

シンボルおよびセクションの保持

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

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

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

シンボルとセクションを保持するリンク手順の詳細については、51 ページの*リンク処理*を参照してください。

アプリケーションの起動

デフォルトでは、アプリケーションが実行を開始する位置は __iar_program_start ラベルで定義されています。これは、コードの開始位置をポイントするよう定義されています。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（281 ページの *--entry* を参照）。

スタックメモリの設定

CSTACK ブロックのサイズは、リンカ設定ファイルで定義されています。割り当てられたメモリの量を変更するには、シンボル _STACK_SIZE を必要な値に設定します：

```
define block CSTACK with alignment = 2, size = _STACK_SIZE
                                        { rw section CSTACK };
```

アプリケーションで必要なサイズを指定してください。

スタックの情報については、「194 ページの*スタックについて*」を参照してください。

ヒープメモリの設定

ヒープのサイズは、リンカ設定ファイルでブロックとして定義されます。割り当てられたヒープメモリの量を変更するには、シンボル `_NEAR_HEAP_SIZE`、`_FAR_HEAP_SIZE`、`_HUGE_HEAP_SIZE` を必要な値に設定します：

```
define block NEAR_HEAP with alignment = 2,
                            size = _NEAR_HEAP_SIZE {  };
define block FAR_HEAP with alignment = 2,
                            size = _FAR_HEAP_SIZE {  };
define block HUGE_HEAP with alignment = 2,
                            size = _HUGE_HEAP_SIZE {  };
```

ヒープを使用する場合は、少なくとも 50 バイトを割り当てる必要があります。

ATEXIT 制限の設定

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

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

デフォルト初期化の変更

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

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

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

初期化の無効化

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

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

パッキングアルゴリズムの選択

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

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

使用可能なそのパッキングアルゴリズムの詳細については、380 ページの *initialize ディレクティブ*を参照してください。

手動で初期化する

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

イニシャライザを持たないセクション（ゼロ初期化されたセクション）の場合、状況は逆になります。`do not initialize` ディレクティブで記述されたものを除いて、リンカはアプリケーション起動時にこうしたすべてのセクションのゼロ初期化を行います。通常は `.noinit` セクションのみが `do not initialize` ディレクティブで指定されますが、必要なだけゼロ初期化されたセクションを追加して、いつどのようにセクションが初期化されるかを制御することができます。

暗黙的なブロックを使用した単純なコピーの例

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

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

このソースコードサンプルを使用して、次のセクションを初期化できます。

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

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

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

明示的なブロックの例

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

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

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

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

オーバレイの例

これは、自動ブロック作成を活用した単純なオーバレイの例です。

```
initialize manually { section MYOVERLAY* };

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

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

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

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

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

コードを初期化する（ROM から RAM にコピーする）

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

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

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

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

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

すべてのコードを RAM から実行

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

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

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

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

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

使用可能なその圧縮アルゴリズムの詳細については、380 ページの *initialize ディレクティブ*を参照してください。

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

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

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

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

ILINK とアプリケーション間の相互処理

ILINK は、アプリケーションの制御に使用できるシンボルを定義するコマンドラインオプション --config_def および --define_symbol を提供しています。また、リンカ設定ファイルで定義される連続するメモリエリアの開始および終了位置を表すシンボルを使用することもできます。詳細については、195 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、195 ページの *ツールとアプリケーション間の相互処理*を参照してください。

標準ライブラリの処理

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

ライブラリの自動追加を無効にするには、オプション `--no_library_search` を使用します。この場合、ライブラリに含めるすべてのライブラリファイルを明示的に指定する必要があります。使用可能なライブラリファイルについては、111 ページの *ビルド済ライブラリの使用*を参照してください。

ELF/DWARF 以外の出力フォーマットの生成

ILINK は、ELF/DWARF フォーマットでのみ出力ファイルを生成できます。このフォーマットを PROM/FLASH のプログラムに適したフォーマットに変換するには、413 ページの *IAR ELF ツール— ielftool* 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（284 ページの *--log* を参照）
- ILINK でメモリマップファイルを生成する `--map` オプション。このファイルには、リンカ設定ファイルの結果が含まれます（285 ページの *--map* を参照）

再配置エラー

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

ILINK で発生する再配置エラーの例を以下に示します。

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

このメッセージエントリについて、以下の表で説明します。

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

表 9: 再配置エラーの説明

メッセージエントリ	説明
Destination	命令のターゲット。詳細については、以下を参照してください。 • 16 進数およびオフセットを持つラベルとして表される命令アドレス。この例の場合、0x9000000c および "read" です（オフセットなし）。 • モジュール、およびライブラリ（適切な場合）。この例の場合、モジュールは read.o、ライブラリは iolib.a です。 • セクション番号およびセクション名。この例の場合、セクション番号は 6 で、名前は .text です。 • バイト数で指定されるセクション内のオフセット。この例のオフセットを以下に示します。0x0

表 9: 再配置エラーの説明（続き）

考えられる解決方法

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

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

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

DLIB ランタイム環境

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

ランタイム環境の概要

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

ランタイム環境の機能

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

提供されるランタイムライブラリには、（製品パッケージに応じて）ビルド済ライブラリとソースファイルの両方があり、これらはそれぞれ、製品のサブディレクトリ `r178¥lib` と `r178¥src¥lib` に格納されています。

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

- ハードウェア機能のサポート
  - *組込み*関数（割込みマスク処理用関数など）による低レベルプロセッサ処理へ直接アクセス
  - インクルードファイルでの周辺ユニットレジスタと割込みの定義
- ランタイム環境サポート（起動 / 終了コード、一部のライブラリ関数との低レベルインタフェース）
- 浮動小数点演算のサポートを含む浮動小数点環境 (fenv)（367 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

ライブラリの詳細は、「*ライブラリ関数*」を参照してください。

ランタイム環境の設定

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

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

- 使用するランタイム環境のオブジェクトファイルを選択します

  ILINK により正しいライブラリファイルが自動的に使用されるため、ライブラリファイルを明示的に指定する必要はありません。111 ページの *ビルド済ライブラリの使用*を参照してください。

- どの定義済ランタイムライブラリ設定を使用するかを選択します（ノーマルまたはフル）

  特定のライブラリ機能のサポートレベルを設定できます。たとえば、ロケールやファイル記述子、マルチバイト文字などがあります。何も指定しない場合、ライブラリオブジェクトファイルに一致するデフォルトのライブラリ設定ファイルが自動的に使用されます。ライブラリ設定を明示的に指定するには、`--dlib_config` コンパイラオブジェクトを使用します。126 ページの *ライブラリ構成*を参照してください。

- ランタイムライブラリのサイズの最適化

  関数 `printf`、`scanf`、およびこれらの派生関数で使用されるフォーマッタを指定できます（114 ページの *printf、scanf のフォーマッタの選択*を参照）。

  また、スタックとヒープのサイズおよび配置を指定できます（それぞれ 99 ページの *スタックメモリの設定*、100 ページの *ヒープメモリの設定*を参照）。

- ランタイムのデバッグサポートおよび I/O デバッグのインクルード

  ライブラリは、C-SPY の［**ターミナル I/O**］ウィンドウへの標準入力および出力のリダイレクトや、ホストコンピュータ上のファイルへのアクセスのサポートを提供します（116 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（119 ページの*ターゲットハードウェアのライブラリの適合*を参照)。

- ライブラリモジュールのオーバライド

  ライブラリの機能をカスタマイズしている場合、デフォルトのモジュールではなく、自分のライブラリモジュールのバージョンが使用されるようにしてください。オーバライドは、ライブラリ全体をリビルドせずに行うことができます（120 ページの*ライブラリモジュールのオーバライド*を参照)。

- システム初期化のカスタマイズ

  システム初期化のソースコードをカスタマイズしなければならないことがほとんどです。たとえば、使用するアプリケーションがメモリをマッピングされた特殊な関数レジスタを初期化したり、データセクションのデフォルトの初期化を省略しなければならないことがあります。この場合、ルーチン `__low_level_init` をカスタマイズします。これによって、データ セクションが初期化される前に実行されるようにします。122 ページの*システムの起動と終了*および 125 ページの*システム初期化のカスタマイズ*を参照してください。

- 独自のライブラリ設定ファイルの設定

  ビルド済のライブラリ設定のほかに、独自のライブラリ設定を作成できますが、これにはライブラリの*リビルド*が必要です。この機能により、ランタイム環境を完全に管理できます。120 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。

- 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（140 ページの*モジュールの整合性チェック*を参照)。

ビルド済ライブラリの使用

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

- コードモデル
- データモデル
- プロセッサコア
- `double` 浮動小数点型のサイズ
- 呼出し規約
- ライブラリ構成（ノーマルまたはフル）
- `--generate_far_runtime_library_calls` オプションの使用

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、59 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。

{*library*}	は、IAR DLIB ランタイム環境のための dl です
{*cpu*}	は、RL78 マイクロコントローラ用 rl78 です
{*code_model*}	は Near または Far コードモデルの場合、それぞれ n または f のどちらかです
{*data_model*}	は Near または Far データモデルの場合、それぞれ n または f のどちらかです
{*size_of_double*}	は、f（32 ビットの場合）か d（64 ビットの場合）です
{*calling_c*}	1（V1 呼出し規約）または 2（V2 呼出し規約）
{*core*}	2（S1 または S2）、もしくは 3(S3)
{*lib_config*}	ノーマルとフルの場合は、それぞれ n か f です
{*far_rt_calls*}	は、--generate_far_runtime_library_calls コンパイラオプションが使用されている場合は f、それ以外の場合は空白になります
{*debug_io*}	d（デバッグ I/O サポート）、または n（デバッグ I/O サポートなし）

注 : ライブラリ構成ファイルは、ライブラリと同じベース名を持っています。

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/C++ 標準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/C++ により定義される関数で、たとえば printf や scanf などです。

ライブラリファイルの名前は、以下のように構成されます。

dl{*cpu*}{*code_model*}{*data_model*}{*size_of_double*}{*calling_c*}{*core*}{*lib_config*}{*far_rt_calls*}.a

これは、特に次のことを表します。

```
dlrl78{n|f}{n|f}{f|d}{1|2}{2|3}{n|f}{f| }.a
```

C-SPY デバッグサポート用のライブラリファイル

これらは、C-SPY のデバッグサポートのための関数です。

ライブラリファイルの名前は、以下のように構成されます。

dbg{*cpu*}{*code_model*}{*data_model*}{*size_of_double*}{*calling_c*}{*core*}{*debug_io*}.a

これは、特に次のことを表します。

```
dbgrl78{n|f}{n|f}{f|d}{1|2}{2|3}{d|n}.a
```

ビルド済ライブラリのカスタマイズ（リビルドなし）

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

カスタマイズ可能な項目は、以下のとおりです。

カスタマイズ可能な項目	参照先
printf、scanf のフォーマッタ	114 ページの *printf、scanf のフォーマッタの選択*
起動 / 終了コード	122 ページの *システムの起動と終了*
低レベル I/O	127 ページの *標準 I/O ストリーム*
ファイル I/O	131 ページの *ファイル I/O*
低レベル環境関数	134 ページの *環境の操作*
低レベルシグナル関数	135 ページの *signal と raise*
低レベル時間関数	135 ページの *時間*
一部のライブラリ数学関数	136 ページの *数学関数*
ヒープ、スタック、セクションのサイズ	194 ページの *スタックについて* 194 ページの *ヒープについて* 87 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 10: カスタマイズ可能な項目

ライブラリモジュールのオーバライドの詳細については、120 ページの *ライブラリモジュールのオーバライド*を参照してください。

printf、scanf のフォーマッタの選択

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

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

注 : ライブラリをリビルドする場合は、これらの関数をさらに最適化できます（129 ページの *printf、scanf の構成シンボル*を参照）。

PRINTF フォーマッタの選択

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

以下の表に、各種フォーマッタの機能の概要を示します。

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

表 11: printf 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、129 ページの *printf、scanf の構成シンボル*を参照してください。

IDE での手動による printf のフォーマッタの指定

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

コマンドラインからの手動による printf のフォーマッタの指定

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

```
--redirect __Printf=__PrintfFull
--redirect __Printf=__PrintfFullNoMb
--redirect __Printf=__PrintfLarge
--redirect __Printf=__PrintfLargeNoMb
--redirect __Printf=__PrintfSmall
--redirect __Printf=__PrintfSmallNoMb
--redirect __Printf=__PrintfTiny
--redirect __Printf=__PrintfTinyNoMb
```

SCANF フォーマッタの選択

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

以下の表に、各種フォーマッタの機能の概要を示します。

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

表 12: 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、129 ページの *printf*、*scanf の構成シンボル*を参照してください。

IDE での手動による scanf のフォーマッタの指定

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

コマンドラインからの手動による scanf のフォーマッタの指定

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

```
--redirect __Scanf=__ScanfFull
--redirect __Scanf=__ScanfFullNoMb
--redirect __Scanf=__ScanfLarge
--redirect __Scanf=__ScanfLargeNoMb
--redirect __Scanf=__ScanfSmall
--redirect __Scanf=__ScanfSmallNoMb
```

アプリケーションデバッグサポート

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

C-SPY デバッグサポートを含める

ライブラリで以下についてデバッグサポートを提供できます。

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

IDE で、［**プロジェクト**］>［**オプション**］>［**リンカ**］を選択します。［**ライブラリ**］ページで、［**C-SPY デバッグサポートを含める**］オプションを選択します。

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

注 : コンパイル中にデバッグ情報を有効にすると、リンカオプション `--strip` を使用しない限り、この情報はリンカ出力にも含まれます。

ライブラリ機能のデバッグ

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

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

この機能は、以下のようにして実装されています。

まず、デバッガが関数 __DebugBreak が存在するかどうかを検出します。この関数は、C-SPY デバッグサポート用の ILINK オプションを使用してアプリケーションをリンクしている場合に、アプリケーションに含まれています。検出された場合は、デバッガは __DebugBreak 関数にブレークポイントを自動設定します。アプリケーションが open などを呼び出すと、__DebugBreak 関数が呼び出され、アプリケーションが一時停止して必要なサービスを実行します。その後、実行が再開されます。

C-SPY の［ターミナル I/O］ウィンドウ

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

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

［ターミナル **I/O**］ウィンドウの詳細については、*『RL78 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、IDE で［プロジェクト］>［オプション］>［リンカ］>［ライブラリ］を選択し、［バッファした書込み］を選択するか、以下の行をリンカコマンドラインに追加します。

```
--redirect ___write=___write_buffered
```

デバッグライブラリの低レベル関数

デバッグライブラリには、以下の低レベル関数の実装が含まれています。

DLIB の低レベルインタフェースの関数	C-SPY の対応
abort	アプリケーションが abort を呼び出したことを通知します
clock	ホストコンピュータ上のクロックを返します
__close	ホストコンピュータ上の対応するファイルを閉じます
__exit	アプリケーションの最後に到達したことを通知します
__lseek	ホストコンピュータ上の対応するファイル内を検索します
__open	ホストコンピュータ上のファイルを開きます
__read	stdin、stdout、stderr を［**ターミナル I/O**］ウィンドウにダイレクトします。他のすべてのファイルは、関連のホストファイルを読み取ります
remove	［**デバッグログ**］ウィンドウにメッセージを出力し、-1 を返します
rename	［**デバッグログ**］ウィンドウにメッセージを出力し、-1 を返します
_ReportAssert	アサートの失敗を処理します
system	［**デバッグログ**］ウィンドウにメッセージを出力し、-1 を返します
time	ホストコンピュータ上の時刻を返します
__write	stdin、stdout、stderr を［**ターミナル I/O**］ウィンドウにダイレクトします。他のすべてのファイルは、関連のホストファイルに書き込みます

表 13: 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：先頭に _ または __ が付く低レベルのインタフェースは、直接アプリケーションで使用しないでください。代わりに、これらの関数を使用する高レベルの関数を使用してください。詳細については、119 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、120 ページの *ライブラリモジュールのオーバライド*を参照してください。

ライブラリの低レベルインタフェース

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

ただし、デバッグバージョンのライブラリ低レベルインタフェースがライブラリに含まれており、ここでターゲットハードウェアではなくデバッガを介してホストコンピュータとやりとりするように低レベル関数が実装されます。デバッグライブラリを使用する場合、［**ターミナル I/O**］ウィンドウへの書込みや、ホストコンピュータ上のファイルへのアクセス、ホストコンピュータからの時間の取得といったタスクをアプリケーションが実行できます。詳しくは 117 ページの *ライブラリ機能のデバッグ*を参照してください。

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

自作モジュールでオーバライドできるライブラリファイルは、`rl78¥src¥lib` ディレクトリにあります。

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

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

ライブラリモジュールのオーバライド

実装したライブラリ低レベルのインタフェースを使用するには、それをアプリケーションに追加します。119 ページの*ターゲットハードウェアのライブラリの適合*を参照してください。そうでなければ、カスタマイズしたバージョンでデフォルトのライブラリルーチンをオーバライドすると便利です。どちらの場合も、以下の手順に従ってください。

1 テンプレートソースファイル（ライブラリソースファイルまたは別のテンプレート）を使用して、それを自分のプロジェクトディレクトリにコピーします。

2 ファイルを修正します。

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

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

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

自作モジュールでオーバライドできるライブラリファイルは、`rl78¥src¥lib` ディレクトリにあります。

カスタマイズしたライブラリのビルドと使用

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

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

このような場合は、以下を行う必要があります。

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

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

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

ライブラリプロジェクトのセットアップ

IDE では、ランタイム環境構成のカスタマイズに使用できるライブラリプロジェクトテンプレートが提供されています。このライブラリテンプレートは、ライブラリ構成がフルに設定されています。表 14 *「ライブラリ構成」* を参照してください。

IDE で、作成したライブラリプロジェクトの［一般オプション］をアプリケーションに応じて変更します（57 ページの*基本的なプロジェクト設定*を参照）。

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

ライブラリ機能の修正

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

ライブラリの機能は、*構成シンボル*で決定されます。これらのシンボルのデフォルト値は、`DLib_Defaults.h` ファイルで定義されています。このファイルはリードオンリーで、設定可能な値が記述されています。ライブラリには独自のライブラリ設定ファイル `libraryname.h` もあり、これは特定のライブラリを必要なライブラリ設定でセットアップします。詳細については、113 ページの *ビルド済ライブラリのカスタマイズ（リビルドなし）* を参照してください。

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

ライブラリ設定ファイルの修正

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

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

カスタマイズしたライブラリの使用

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

IDE で以下の手順を実行する必要があります。

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

2 ［**ライブラリ**］ドロップダウンリストから、［**カスタム DLIB**］を選択します。

3 ［**設定ファイル**］テキストボックスで、ライブラリ設定ファイルを指定します。

4 ［**リンカ**］カテゴリで、［**ライブラリ**］タブをクリックします。［**追加ライブラリ**］テキストボックスで、ライブラリファイルを指定します。

システムの起動と終了

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

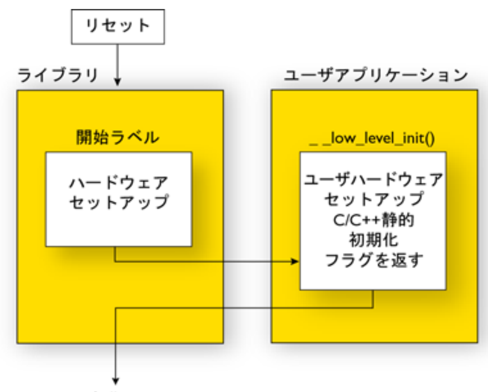
起動と終了を取り扱うコードは、`rl78¥src¥lib` ディレクトリのソースファイル `cstartup.s`、`cexit.s`、`low_level_init.c` にあります。

システム起動コードのカスタマイズ方法については、「125 ページの *システム初期化のカスタマイズ*」を参照してください。

システム起動

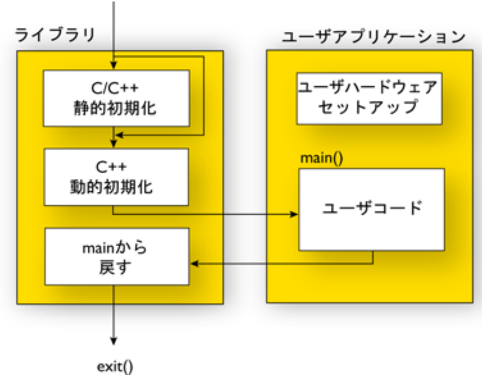
システムの起動時、`main` 関数が入力される前に初期化シーケンスが実行されます。このシーケンスでは、ターゲットハードウェアと C/C++ 環境で必要とされる初期化を実行します。

ハードウェアの初期化は、以下のように実行されます。



- CPU がリセットされると、システム起動コードのプログラムエントリラベル `__iar_program_start` で起動を開始します。
- スタックポインタが初期化されます。
- 関数 `__low_level_init` を定義している場合、この関数が呼出され、アプリケーションで低レベルの初期化を実行できます。

C/C++ の初期化は、以下のように実行されます。



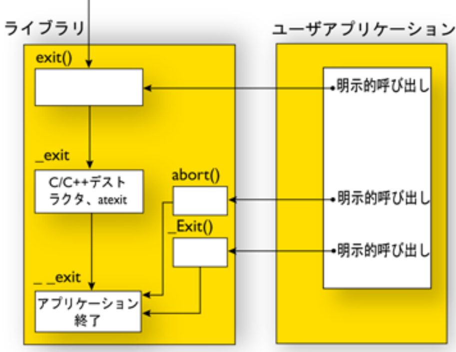
- 静的変数とグローバル変数が初期化されます。つまり、ゼロ初期化変数がクリアされ、他の初期化変数の値が ROM から RAM メモリにコピーされます。このステップは、`__low_level_init` がゼロを返す場合には、省略されます。詳細については、90 ページの *システム起動時の初期化*を参照してください。
- 静的 C++ オブジェクトが生成されます。

- main 関数が呼出され、アプリケーションが起動します。

初期化フェーズについて詳しくは、53 ページの*アプリケーションの実行 — 概要*を参照してください。

システム終了

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



アプリケーションは、以下の 2 つの方法で正常終了できます。

- main 関数から戻る
- exit 関数を呼出す

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

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

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

アプリケーションは、abort または _Exit 関数を呼出すことによっても終了できます。abort 関数は、単に __exit を呼出してシステムの停止を行うだけ

で、終了処理は実行しません。_Exit 関数も abort 関数とほとんど同じですが、_Exit では、終了ステータス情報を渡すための引数をとります。

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

システム終了と C-SPY のインタフェース

プロジェクトが C-SPY デバッグライブラリにリンクされている場合、通常の __exit 関数と abort 関数は特殊なものに置き換えられます。これにより、C-SPY はこれら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、116 ページの *アプリケーションデバッグサポート*を参照してください。

システム初期化のカスタマイズ

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

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

システム起動を処理するコードはソースファイル cstartup.s と low_level_init.c にあります。これらは rl78¥src¥lib ディレクトリに格納されています。

注 : 通常は cexit.s のカスタマイズは不要です。

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（120 ページの *カスタマイズしたライブラリのビルドと使用*を参照)。

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

__LOW_LEVEL_INIT

製品には low_level_init.c という低レベル初期化ファイルのスケルトンが付属しています。これは、ビルド済ランタイムライブラリの一部です。この時点では変数は初期化されていないため、初期化された静的変数はこのファイル内では使用できません。

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

CSTARTUP.S ファイルの修正

前述のように、__low_level_init のカスタマイズによって目的の動作を達成できる場合は、cstartup.s ファイルを修正する必要はありません。ただし、cstartup.s ファイルの修正が必要な場合は、一般的な手順に従いファイルをコピーして修正し、プロジェクトに追加することをお勧めします（120 ページの *ライブラリモジュールのオーバライド*を参照）。

使用している cstartup.s のバージョンで使用される開始ラベルが、確実にリンカで使用されるようにする必要があります。リンカで使用される開始ラベルの変更方法については、281 ページの *--entry* を参照してください。

ライブラリ構成

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

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

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

以下のいずれかの定義済ライブラリ構成を使用できます。

ライブラリ構成	説明
ノーマル DLIB（デフォルト）	ロケールインタフェースなし、C ロケール、ファイル記述子サポートなし、printf と scanf でのマルチバイト文字なし、strtod での 16 進数浮動小数点数なし。
フル DLIB	フルロケールインタフェース、C ロケール、ファイル記述子サポート、printf と scanf でのマルチバイト文字、strtod での 16 進数浮動小数点数。

表 14: 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。250 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、120 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。

標準 I/O ストリーム

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

C/C++ ですべてのキャラクタベース I/O を実行する際に使用される、低レベルの I/O 関数があります。キャラクタベース I/O を使用する場合は、ハードウェア環境で提供される機能を使用して、これらの関数を定義する必要があります。低レベル関数の実装について詳しくは、119 ページの*ターゲットハードウェアのライブラリの適合*を参照してください。

低レベルキャラクタ I/O の実装

ストリーム `stdin` および `stdout` の低レベル関数を実装するには、`__read` および `__write` をそれぞれ記述する必要があります。これらの関数のテンプレートソースコードは、`r178¥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（120 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（116 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 ディスプレイに書込み、そのポートがアドレス `0xFFF70` にあると想定しています。

```
#include <stddef.h>

__no_init volatile __sfr unsigned char lcdIO @ 0xFFF70;

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

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

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

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

  return nChars;
}
```

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

__read の使用例

この例のコードは、メモリがマップされた I/O を使用してキーボードから読込み、そのポートがアドレス 0xFFF72 にあると想定しています。

```
#include <stddef.h>

__no_init volatile __sfr unsigned char kbIO @ 0xFFF72;

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

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

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

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

  return nChars;
}
```

@ 演算子の詳細は、*206 ページのデータと関数のメモリ配置制御*を参照してください。

printf、scanf の構成シンボル

アプリケーションプロジェクトをセットアップする場合は、アプリケーションで必要な printf と scanf のフォーマット機能を考慮する必要があります（114 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 15: 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 16: 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（120 ページの*カスタマイズしたライブラリのビルドと使用*を参照）。
2. アプリケーションの必要に応じて、構成シンボルを定義します。

ファイル I/O

このライブラリには、fopen や fclose、fprintf、fputs など、ファイルの I/O 処理のための数多くの強力な関数が含まれています。これらの関数はすべて、いくつかの低レベル関数を呼出します。これらの関数は、それぞれ特定のタスクを 1 つ実行するように設計されています。たとえば、__open はファイルを開き、__write は文字を出力します。アプリケーションでファイルの I/O 処理用のライブラリ関数を使用する前に、ターゲットハードウェアにあわせて、対応する低レベル関数を実装する必要があります。詳細については、119 ページのターゲットハードウェアのライブラリの適合を参照してください。

ライブラリのファイル I/O 機能は、フルのライブラリ構成を持つライブラリのみによってサポートされます（126 ページのライブラリ構成を参照）。すなわち、ファイル I/O は、構成シンボル __DLIB_FILE_DESCRIPTOR が有効な場合にのみサポートされます。有効でない場合は、関数に FILE * 引数を指定することはできません。

以下のファイル I/O 用テンプレートコードが付属しています。

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

表 17: 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、116 ページのアプリケーションデバッグサポートを参照してください。

ロケール

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

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

DLIB ライブラリは、以下の 2 つのメインモードで使用できます。

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

ビルド済ライブラリでのロケールサポート

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

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

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

ロケールサポートのカスタマイズ

ライブラリをリビルドする場合は、いずれかのロケールを選択できます。

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

ロケールの構成シンボル

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

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

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

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

ロケールサポートをカスタマイズする場合は、アプリケーションで必要なロケール構成シンボルを定義します。詳細については、120 ページのカスタマイズしたライブラリのビルドと使用を参照してください。

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

ロケールインタフェースをサポートしないライブラリのビルド

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

ロケールインタフェースをサポートするライブラリのビルド

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

実行中のロケール変更

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

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

lang_REGION

または

lang_REGION.encoding

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

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

例

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

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

環境の操作

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

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

GETENV 関数

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

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

```
key=value¥0
```

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

次に例を示します。

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

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

ビルドは必要ありません。ソーステンプレートは、getenv.c や environ.c の各ファイルに含まれています。これらのファイルは、r178¥src¥lib ディレクトリにあります。デフォルトのライブラリモジュールのオーバライドについては、120 ページのライブラリモジュールのオーバライドを参照してください。

システム関数

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

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、120 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 getenv および system は C-SPY 版の関数に置き換えられます。詳細については、116 ページのアプリケーションデバッグサポートを参照してください。

signal と raise

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

この場合、ライブラリのリビルドは必要ありません。ソーステンプレートは、signal.c や raise.c の各ファイルに含まれています。これらのファイルは、r178¥src¥lib ディレクトリにあります。デフォルトのライブラリモジュールのオーバライドについては、120 ページのライブラリモジュールのオーバライドを参照してください。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、120 ページのカスタマイズしたライブラリのビルドと使用を参照してください。

時間

関数 __time32、__time64、date が機能するためには、関数 clock、__time32、__time64、および __getzone を実装する必要があります。__time32 と __time64 のどちらを使用するかは、time_t でどのインタフェースを使用するかによって決まります（368 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、r178¥src¥lib ディレクトリのファイル clock.c、

time.c、time64.c、getzone.c にあります。デフォルトのライブラリモジュールのオーバライドについては、120 ページの*ライブラリモジュールのオーバライド*を参照してください。

ライブラリをリビルドする場合は、ライブラリプロジェクトに含まれるソーステンプレートを使用できます。詳細については、120 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、116 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。

Strtod

関数 strtod は、ライブラリ構成が Normal のライブラリの 16 進数浮動小数点文字列を受け付けません。strtod で 16 進数の浮動小数点文字列を受け入れるには、以下を行う必要があります。

1. ライブラリ設定ファイルで、構成シンボル _DLIB_STRTOD_HEX_FLOAT を有効にしてください。
2. ライブラリのリビルド（120 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）。

数学関数

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

より小さいバージョン

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

関数の名前は以下のように構成されます。

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

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

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

```
--redirect _sin=___iar_sin_small
--redirect _cos=___iar_cos_small
--redirect _tan=___iar_tan_small
--redirect _log=___iar_log_small
--redirect _log2=___iar_log2_small
--redirect _log10=___iar_log10_small
--redirect _exp=___iar_exp_small
--redirect _pow=___iar_pow_small
--redirect ___iar_Sin=___iar_Sin_small
--redirect ___iar_Log=___iar_Log_small

--redirect _sinf=___iar_sin_smallf
--redirect _cosf=___iar_cos_smallf
--redirect _tanf=___iar_tan_smallf
--redirect _logf=___iar_log_smallf
--redirect _log2f=___iar_log2_smallf
--redirect _log10f=___iar_log10_smallf
--redirect _expf=___iar_exp_smallf
--redirect _powf=___iar_pow_smallf
--redirect ___iar_FSin=___iar_Sin_smallf
--redirect ___iar_FLog=___iar_Log_smallf

--redirect _sinl=___iar_sin_smalll
--redirect _cosl=___iar_cos_smalll
--redirect _tanl=___iar_tan_smalll
--redirect _logl=___iar_log_smalll
--redirect _log2l=___iar_log2_smalll
--redirect _log10l=___iar_log10_smalll
--redirect _expl=___iar_exp_smalll
--redirect _powl=___iar_pow_smalll
--redirect ___iar_LSin=___iar_Sin_smalll
--redirect ___iar_LLog=___iar_Log_smalll
```

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

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

より正確なバージョン

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

関数の名前は以下のように構成されます。

__iar_*xxx*_accurate<f|l>

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

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

```
--redirect _sin=___iar_sin_accurate
--redirect _cos=___iar_cos_accurate
--redirect _tan=___iar_tan_accurate
--redirect _pow=___iar_pow_accurate
--redirect ___iar_Sin=___iar_Sin_accurate
--redirect ___iar_Pow=___iar_Pow_accurate

--redirect _sinf=___iar_sin_accuratef
--redirect _cosf=___iar_cos_accuratef
--redirect _tanf=___iar_tan_accuratef
--redirect _powf=___iar_pow_accuratef
--redirect ___iar_FSin=___iar_Sin_accuratef
--redirect ___iar_FPow=___iar_Pow_accuratef

--redirect _sinl=___iar_sin_accuratel
--redirect _cosl=___iar_cos_accuratel
--redirect _tanl=___iar_tan_accuratel
--redirect _powl=___iar_pow_accuratel
--redirect ___iar_LSin=___iar_Sin_accuratel
--redirect ___iar_LPow=___iar_Pow_accuratel
```

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

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

Assert

オプション［**C-SPY デバッグサポートを含める**］を使用してアプリケーションをリンクした場合、C-SPY は失敗したアサートについて通知を受け取りま

す。これが必要でない場合は、ソースファイル xreportassert.c をアプリケーションプロジェクトに追加することができます。__ReportAssert 関数は、アサート通知を生成します。r178¥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、120 ページの *ライブラリモジュールのオーバライド*を参照してください。assert を無効にするには、シンボル NDEBUG を定義する必要があります。

IDE では、このシンボル NDEBUG がリリースプロジェクトにデフォルトで定義されており、デバッグプロジェクトには*定義されていません*。コマンドラインでビルドする場合は、必要に応じてこのシンボルを明示的に定義する必要があります。358 ページの *NDEBUG* を参照してください。

Atexit

リンカは、atexit 関数呼出しの静的メモリエリアを割り当てます。デフォルトでは、atexit 関数の呼出し数は 32 バイトに制限されています。この制限を変更するには、100 ページの *atexit 制限の設定*を参照してください。

ヒープ

ランタイム環境は、次のメモリタイプのヒープをサポートします。

メモリタイプ	セクション名	メモリ属性	データモデルでデフォルトで使用
near	NEAR_HEAP	__near	Near
far	FAR_HEAP	__far	Far
	HUGE_HEAP	__huge	—

表 18: ヒープとメモリタイプ

各ヒープのサイズの設定方法は、100 ページの *ヒープメモリの設定*を参照してください。特定のヒープを使用する場合は、表のプレフィックスがメモリ属性で、これを malloc や free、calloc、realloc の前に追加します。たとえば、__near_malloc というようになります。デフォルトの関数は、データモデルなどプロジェクトの設定に応じて、特定のヒープ変数のいずれかを使用します。C++ で特定のヒープを使用する方法については、183 ページの *演算子 new と delete* を参照してください。

ハードウェアサポート

一部の RL78 マイクロコントローラは、ハードウェア積算 / 除算ユニットを持っています。このユニットのランタイムサポートをインクルードするには、

rl78¥src¥lib¥hw_multiply_division_units ディレクトリのサンプルライブラリコードを使用します。

IDE で、［**プロジェクトオプション**］**>**［**一般オプション**］**>**［**ライブラリ設定**］ページの［**ハードウェア積算 / 除算ユニットを使用**］オプションを選択します。現在選択されている RL78 デバイスにハードウェア積算 / 除算ユニットがない場合、このオプションは使用できません。

モジュールの整合性チェック

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

注 : 定義済の属性のほかに、RL78 ABI ランタイム属性に対しても互換性がチェックされます。これらの属性は、主にオブジェクトコードの互換性を対象にします。コンパイル設定を反映して、ユーザ設定はできません。

たとえば、コンパイラでは double の浮動小数点型のサイズを指定できます。64 ビットの double のみで機能するルーチンを記述した場合、そのルーチンが 32 ビット double を使用してビルドされたアプリケーションで使用されていないことをチェックできます。

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、リンカはいくつかの方法でそれらをチェックします。

例

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

オブジェクトファイル	色	taste
file1	blue	未定義
file2	red	未定義
file3	red	*
file4	red	spicy
file5	red	lean

表 19: ランタイムモデル属性の例

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

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

ランタイムモデル属性の使用

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

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

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

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

先頭に 2 つの下線を持つ名前は、コンパイラで予約済です。構文について詳しくは、「342 ページの *rtmodel*」と『*RL78 用 IAR アセンブラリファレンスガイド*』をそれぞれ参照してください。

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

アセンブラ言語インタフェース

- C 言語とアセンブラの結合
- C からのアセンブラルーチンの呼出し
- C++ からのアセンブラルーチンの呼出し
- 呼出し規約
- 関数の呼び出しに使用されるアセンブラ命令
- メモリアクセス方法
- コールフレーム情報

C 言語とアセンブラの結合

RL78 用 IAR C/C++ コンパイラには、低レベルのリソースにアクセスする方法がいくつか用意されています。

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

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

組込み関数

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

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

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

これによって CALL と RET 命令という形である程度のオーバヘッドが生じ、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、コンパイラはまたすべてのレジスタはインラインアセンブラ命令によって破棄されたとみなします。多くの場合、外部命令によるオーバヘッドは、オプティマイザにより削除されます。

重要な利点は、コンパイラが生成したものとアセンブラに自分で書いたものの間でよく定義されたインタフェースを持つことができることです。インラインアセンブラを使用するときは、ご自分のインラインアセンブラがコンパイラが生成したコードに邪魔しないという保証は一切ありません。

アプリケーションで、一部をアセンブラ言語、一部を C/C++ で記述する場合、多くの疑問点に遭遇します。

- C から呼出せるようにアセンブラコードを記述する方法は ?
- アセンブラコードのパラメータの場所は ? また、呼出し元へ値を返す方法は ?
- C で記述した関数をアセンブラコードから呼出す方法は ?
- アセンブラ言語で記述したコードから、C のグローバル変数にアクセスする方法は ?
- アセンブラコードをデバッグする際に、デバッガでコールスタックが表示されない理由は ?

最初の質問については、「147 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、「150 ページの *呼出し*

規約」で説明します。

最後の疑問については、アセンブラコードをデバッガで実行する際、コールスタックを表示できるというのが答えです。ただし、デバッガではコールフレームについての情報が必要になります。この情報は、アセンブラソースファイルでコメントとして記述する必要があります。詳細については、162 ページの *コールフレーム情報*を参照してください。

C/C++ とアセンブラモジュールを混在させる望ましい方法は、147 ページの *C からのアセンブラルーチンの呼出し*と 149 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。

注 : RL78 ABI に準拠するために、コンパイラはシンボル名および関数名の先頭に下線を付けてアセンブララベルを生成します。アセンブラから C のシンボルにアクセスするときは、この下線を 1 つ追加することを忘れないでください。たとえば、`main` は `_main` と記述する必要があります。

同様に、C から外部のアセンブリモジュールを参照する場合、C モジュールで使用するシンボルに下線が付けらるため、アセンブリモジュール名の先頭の文字を下線にする必要があります。

インラインアセンブラ

インラインアセンブラを使用して、アセンブラ命令を C/C++ 関数に直接挿入できます。

`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"
    "jmp 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。

asm キーワードの使用法を次の例で説明します。また、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 __saddr char UART1_SR;
#pragma required=UART1_SR

static __near char sFlag;

void Foo(void)
{
  while( !sFlag )
  {
    asm("MOV A, S:UART1_SR");
    asm("MOV sFlag, A");
  }
}
```

この例では、グローバル変数 sFlag 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、DC16 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
- ラベルを宣言できません

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。

C からのアセンブラルーチンの呼出し

C から呼出すアセンブラルーチンは、以下を満たしている必要があります。

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

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

または

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

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

コンパイラは 2 つの異なる呼出し規約をサポートしています（150 ページの*呼出し規約*を参照）。

スケルトンコードの作成

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

以下の例は、ルーチン本体を簡単に追加できるスケルトンードの作成方法を示します。スケルトンソースコードで必要な処理は、必要な変数を宣言し、

それらにアクセスするだけです。この例では、アセンブラルーチンは int、char を引数に指定し、int を返します。

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
  int locInt = arg1;
  gInt = arg1;
  gChar = arg2;
  return locInt;
}

int main()
{
  int locInt = gInt;
  gInt = Func(locInt, gChar);
  return 0;
}
```

注 : この例では、ローカル変数とグローバル変数のアクセスを示すため、コードのコンパイル時の最適化レベルを低くしています。最適化レベルを高くすると、ローカル変数への必要な参照が最適化で削除される場合があります。最適化レベルによって実際の関数宣言が変更されることはありません。

スケルトンコードのコンパイル

IDE において、リストオプションをファイルレベルで指定します。［ワークスペース］ウィンドウでファイルを選択します。次に、［**プロジェクト**］**>**［**オプション**］を選択します。［**C/C++ コンパイラ**］カテゴリで、［**継承した設定をオーバライド**］を選択します。［**リスト**］ページで［**リストファイルの出力**］の選択を解除し、代わりに［**アセンブラファイルの出力**］オプションおよびそのサブオプション［**ソースのインクルード**］を選択します。また、低い最適化レベルを指定します。

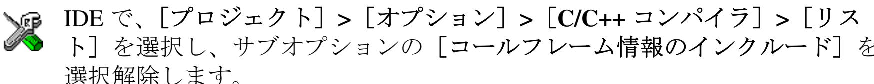
スケルトンコードをコンパイルするには、以下のオプションを使用します。

```
iccrl78 skeleton.c -lA . -On -e
```

-lA オプションは、アセンブラ言語出力ファイルを作成します。このファイルでは、C/C++ ソース行がアセンブラのコメントとして記述されています。.（ピリオド）は、アセンブラファイル名を C/C++ モジュール (skeleton) と同様の方法で設定し、拡張子のみを s に変更するように指定します。-On オプションは、最適化が使用されないことを意味し、-e は言語拡張を有効なことを意味します。さらに、関連のコンパイラオプションを必ず使用してください。通常ご自分のプロジェクトの他の C/C++ ソースファイルで使用するものと同じです。

その結果、アセンブラソース出力ファイル `skeleton.s` が生成されます。

注 : `-lA` オプションは、呼出しフレーム情報 (`CFI`) ディレクティブを含むリストファイルを作成します。これは、これらのディレクティブと使用方法について調べる意図がある場合に便利です。呼出し規約のみを調べるのであれば、`CFI` ディレクティブをリストファイルから除外できます。



IDE で、**［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［リスト］**を選択し、サブオプションの**［コールフレーム情報のインクルード］**を選択解除します。

コマンドラインで、-lA ではなく `-lB` を使用します。C-SPY の**［コールスタック］**ウィンドウを機能させるためには、ソースコードに `CFI` 情報が含まれている必要があります。

出力ファイル

出力ファイルには、以下の重要情報が含まれています。

- 呼出し規約
- リターン値
- グローバル変数
- 関数パラメータ
- スタック（自動変数）空間を作成する方法
- コールフレーム情報 (CFI)

`CFI` ディレクティブは、デバッガの**［コールスタック］**ウィンドウで必要なコールフレーム情報を記述します。詳細については、162 ページの *コールフレーム情報*を参照してください。

C++ からのアセンブラルーチンの呼出し

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

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

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

```
extern "C"
{
  int MyRoutine(int);
}
```

以下の例は、非静的メンバ関数と同等の処理を実現する方法を示します。すなわち、暗黙的な this ポインタを明示する必要があります。アセンブラルーチンの呼出しをメンバ関数内にラップできます。関数インライン化が有効になっていれば、インラインメンバ関数を使用することで、余分な呼出しによるオーバヘッドが解消されます。

```
class MyClass;

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

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

呼出し規約

呼出し規約とは、プログラム内の関数が別の関数を呼出す方法を規定したものです。コンパイラはこれを自動的に処理しますが、関数をアセンブラ言語で記述している場合は、そのパラメータの位置や特定方法、呼出されたプログラム位置に戻る方法、結果を返す方法がわかっている必要があります。

また、アセンブラレベルのルーチンがどのレジスタを保存する必要があるかを知ることも重要です。プログラムが保存するレジスタが多すぎると、効率が低下する場合があります。保存するレジスタが少なすぎると、不正なプログラムになる可能性があります。

コンパイラには 2 つの呼出し規約があります。1 つはバージョン 1.x のコンパイラで使用される旧来のもので、もう一方は新しいデフォルトのものです。

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。

- 呼出し規約の選択
- 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。

注 : V2 呼出し規約は RL78 ABI 標準に準拠しています。

呼出し規約の選択

以下の呼出し規約から選択できます :

- V1 呼出し規約は、RL78 バージョン 1.40.x 用 IAR C/C++ コンパイラと互換性がありますが、スタック引数のクリーンアップの方法は異なります。この呼出し規約は、旧バージョンのコンパイラで記述されたアセンブラコードがある場合に推奨です。個々の関数でこの呼出し規約を使用するには、`__v1_call` キーワードを使用します。以下のようになります。

```
extern
__v1_call void doit(int arg);
```

- V2 呼出し規約は RL78 ABI に適合しており、他のベンダのツールにより生成されたコードをリンクする際に推奨です。個々の関数でこの呼出し規約を使用するには、`__v2_call` キーワードを使用します。以下のようになります。

V2 呼出し規約がデフォルトで使用されます。コンパイラで V1 呼出し規約を使用するには、`--calling_convention` コマンドラインオプションを使用します。

IDE では、［プロジェクト］>［一般オプション］>［ターゲット］ページで呼出し規約を選択します。

関数の宣言

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

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

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

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

C++ ソースコードでの C リンケージの使用

C++ では、関数は C または C++ のいずれかのリンケージを持つことができます。アセンブラルーチンを C++ から呼出すには、C++ 関数に C リンケージを持たせるのが最も簡単です。

C リンケージを持つ関数の宣言例を示します。

```
extern "C"
{
  int F(int);
}
```

多くの場合は、ヘッダファイルを C と C++ で共有するのが実用的です。C と C++ の両方で C リンケージを持つ関数を宣言する例を示します。

```
#ifdef __cplusplus
extern "C"
{
#endif

int F(int);

#ifdef __cplusplus
}
#endif
```

保護レジスタとスクラッチレジスタ

通常の RL78 CPU のレジスタは、以下で説明する 3 種類に分類されます。

スクラッチレジスタ

スクラッチレジスタの内容は、任意の関数により破壊される可能性があります。関数が別の関数の呼出した後もレジスタの値を必要とする場合は、呼出し中はその値をスタックなどで保存する必要があります。

次のレジスタがスクラッチレジスタとして使用されます。

- AX、HL、CS、ES の各レジスタ。V2 呼出し規約では、BC と DE レジスタもスクラッチレジスタです。
- レジスタパラメータとして、また関数で値を返すために使用されるレジスタ。

保護レジスタ

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

- V1 呼出し規約では、BC および DE レジスタは保護レジスタです。
- V2 呼出し規約では、割込み関数でない呼出しに対して保護されているレジスタはありません。割込み全体に対して、レジスタ AX、HL、BC、DE はアクティブなレジスタバンクで保護され、システムレジスタの PC と PSW も保護されています。

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。割込みが発生した場合、スタックポインタが示すポイントより下にあるすべては廃棄されます。

関数の入口

これらの基本的な方法ひとつを使用して、パラメータを関数に渡すことができます。

- レジスタ内
- スタック上

メモリ経由で迂回して引き渡すよりもレジスタを使用する方が大幅に効率的です。そのため、呼出し規約では可能な限りレジスタを使用するように規定されています。パラメータの引渡しに使用できるレジスタ数は非常に少ないため、レジスタが不足する場合は、残りのパラメータはスタックに引き渡されます。V1 呼出し規約では、パラメータも以下の場合にスタックに引き渡されます。

- 構造体型 : `struct`, `union`, and classes, ただしサイズ 1、2、4 の structs を除く（サイズが 2 か 4 の場合の 2 の最小アライメントで）
- 可変長（可変数引数）関数への未指定パラメータ。つまり、`foo(`*`param1`*`, ...)` として宣言された関数。たとえば、`printf` など

つまり、4 バイトより大きいオブジェクトはスタックに引き渡され、V1 呼出し規約ではすべての 3 バイトのオブジェクトが渡されます。詳細については、154 ページの *レジスタパラメータ*を参照してください。

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。

関数が構造体を返す場合、割り当てられた空間へのポインタは隠しパラメータとして引き渡されます。隠しパラメータは、普通のパラメータとして処理されます。

レジスタパラメータ

パラメータを引き渡すために使用可能なレジスタは、AX、BC、および DE です。

パラメータ	レジスタでの引渡し V1 呼出し規約	レジスタでの引渡し V2 呼出し規約
8 ビットの値	A、B、C、X、D、E	A、B、C、X、D、E
16 ビットの値	AX、BC、DE	AX、BC、DE
24 ビットの値	スタックへ引渡し	C:AX、X:BC、E:BC、X:DE、B:DE (struct) A:DE、X:DE、C:DE、B:DE、X:BC（far ポインタ）
32 ビットの値	BC:AX	BC:AX、DE:BC
64 ビットの値	スタックへ引渡し	スタックへ引渡し

表 20: パラメータの引渡しに使用されるレジスタ

レジスタをパラメータに割り当てるプロセスは単純明快です。パラメータを厳密に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。適切なレジスタがない場合は、パラメータはスタックを使用して引き渡されます。

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メインメモリに格納されています。スタックポインタ以下（下位メモリ方向）には、呼出し先関数が使用可能な空きエリアがあります。最初のスタックパラメータは、スタックポインタが指す位置に格納されています。次のパラメータは、2 などで分割可能なスタック上の位置に格納されます。スタック上のオブジェクトは、アラインメント 2 に格納されます。これはつまり、1 バイトのオブジェクト (char) が 2 バイトを占めるということです。

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。

上位アドレス
呼出し元のスタックフレーム
パラメータ *n*
...
パラメータ 1
リターンアドレス
スタックポインタ
下位アドレス
空きスタックメモリ

関数の終了

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

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

リターン値に使用されるレジスタ

リターン値に使用できるレジスタは `A`、`B`、`C`、および `X` です。

リターン値	**レジスタでの引渡し** **V1 呼出し規約**	**レジスタでの引渡し** **V2 呼出し規約**
8 ビットの値	`A`	`A`
16 ビットの値	`AX`	`AX`
24 ビットの値	`A:HL`	`C:AX` (struct) `A:DE`（far ポインタ）
32 ビットの値	`BC:AX`	`BC:AX`
64 ビットの値	ポインタを介しての引渡し	ポインタを介しての引渡し

表 21: リターン値に使用されるレジスタ

関数終了時のスタックのレイアウト

関数終了の前にスタックをクリーンするのは、呼び出された関数の役目です。つまり、プッシュされたパラメータをリターンの前にクリーンにする働きをします。

リターンアドレスの処理

アセンブラ言語で記述された関数は、終了時に呼出し元に戻るのが普通です。関数が呼び出されると、リターンアドレスはスタック上に格納されて、次のようになります。

```
            CALL    func
```

関数は通常、RET 命令を使用することで値を返します。

この規則には 2 つの例外があります。

- 割込み関数の場合、関数のリターンに命令 RETI が使用されます。
- ブレーク割込みの BRK 命令の場合、命令 RETB が関数のリターンに使用されます。

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 AX を使用して引き渡し、リターン値をレジスタ AX を使用して呼出し元に返します。

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

```
            name    return
            section CODE:CODE
            public  _add1
add1:
            incw    ax
            ret
            end
```

例 2

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

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

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

呼出し元関数は、スタックの上位 10 バイトを確保し、`struct` の内容をその位置にコピーします。整数パラメータ `y` は、レジスタ `AX` 内で引き渡されます。リターン値は、`AX` レジスタを使用して呼出し元に返されます。

例 3

次の関数は、`struct MyStruct` 型の構造体を返します。

```
struct MyStruct
{
  int mA[20];
};

struct MyStruct MyFunction(int x);
```

リターン値のメモリ位置を割り当てて、それにポインタを最初の隠しパラメータとして引き渡すのは、呼出し元の関数の役割です。near データモデルを使用している場合、リターン値が格納されるべき場所へのポインタは `AX` 内で引き渡されます。呼出し元は、これらのレジスタがそのままになっていると見なします。パラメータ `x` は `BC` 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、リターン値はポインタであり、隠しパラメータはありません。パラメータ `x` は `AX` で引き渡され、リターン値は `AX` で返されます。

関数の呼び出しに使用されるアセンブラ命令

この章では、RL78 マイクロコントローラで関数から呼び出したり返したりするために使用できるアセンブラ命令について説明します。

関数は関数ポインタは、または関数ポインタ表を介して、直接さまざまな方法で呼び出すことができます。この章では、それぞれのコードモデル用にこれらの種類の呼び出しを実行する方法を説明します。

通常の関数を呼び出す命令は call 命令です。

```
call label
```

次の章では異なるコードも出るが関数呼び出しを実行する方法を示します。

NEAR コードモデルの関数の呼び出し

このコードモデルを使用して直接呼び出すことは簡単です。

```
call n:label16
```

関数ポインタを介して関数呼び出しが行われると、このコードが生成されます。

```
name    callFuncPtr
rseg    CODE:CODE
extern  funcPtr

movw    ax, n:funcPtr ; Load function address
mov     cs, #0        ; clear byte 3
call    ax            ; Make function call
end
```

アドレスはレジスタに格納され、関数の呼び出しに使用されます。関数ポインタを介した呼び出しは 32 ビットアドレス空間全体に到達します。

FAR コードモデルの関数の呼び出し

このコードモデルを使用して直接呼び出すことは簡単です。

```
call f:label24
```

関数ポインタを介して関数呼び出しが行われると、このコードが生成されます。

```
            name    callFuncPtr
            rseg    CODE:CODE
            extern  funcPtr

            mov     a, n:funcPtr+2 ; Load byte3 of function
                                   ; address
            mov     cs, A
            movw    ax, n:funcPtr ; Load function address
            call    ax            ; Make function call
            end
```

アドレスはレジスタに格納され、関数の呼び出しに使用されます。関数ポインタを介した呼び出しは 32 ビットアドレス空間全体に到達します。

関数表の呼び出しの作成

関数表の呼び出しは簡単です。

```
            callt [index]
```

関数表の呼び出しがされると、このコードが生成されます。

```
            name    callTable
            rseg    CLTVEC:CODE:REORDER:NOROOT(1)
__T_func:
            dw      func
            rseg    CODE:CODE
            extern  func

            callt   [__T_func]        ; Make function call
            end
```

メモリアクセス方法

この章では、*データ記憶*章で示された異なるメモリの種類について説明します。さらに、データのアクセスに使用するアセンブラコードについても説明します。またさまざまなメモリの種類の裏にある理由を説明します。

RL78 命令セットを熟知している必要があります。特に、命令で使用されるさまざまなアドレッシングモードがメモリにアクセスできます。

ここでそれぞれのアクセス方法について、次の 3 つの例を説明します。

- グローバル変数のアクセス
- 不明なインデックスを使用してグローバル配列をアクセス

- ポインタを使用して構造体をアクセスします。

これらの 3 つの例はこの C プログラムで示すことができます。

```
char myVar;
char MyArr[10];

struct MyStruct
{
  long mA;
  char mB;
};

char Foo(int i, struct MyStruct *p)
{
  return myVar + MyArr[i] + p->mB;
}
```

SADDR メモリアクセス方法

Saddr（ショートアドレッシングが可能な領域）メモリは、near メモリの 256 バイト領域に配置されます。8 ビットアドレスを使用して、または near ポインタを介してアクセスすることができます。

例

これらの例は saddr メモリにアクセスするさまざまな方法です。

```
        mov     a, s:x          ; グローバル変数 x へのアクセス。

        mov     hl, #y          ; グローバル配列 y の
        addw    ax, hl          ; エントリにアクセス。
        movw    hl, ax
        mov     a, [hl]

        mov     a, [hl+2]       ; ポインタを介したアクセス。
```

NEAR メモリアクセス方法

Near メモリは最後の 64 Kb のメモリに配置されます。

例

これらの例は near メモリにアクセスするさまざまな方法です。

```
mov     a, s:x           ; グローバル変数 x へのアクセス。

mov     a, (y & 0xffff)[bc] ; グローバル配列 y
                            ; へのアクセス。

mov     a, [hl+2]        ; ポインタを介したアクセス。
```

FAR メモリアクセス方法

Far メモリは、制約の 1MB のメモリ空間全体で、各オブジェクトは 64 Kb ページ内にする必要があります。

例

これらの例は near メモリにアクセスするさまざまな方法です。

```
mov     es, byte3(x)     ; グローバル変数 x へのアクセス。
mov     a, es:x

addw    ax, #lwrd(y)     ; グローバル配列のエントリに
movw    hl, ax           ; アクセス。
mov     es, byte3(y)
mov     a, [hl]

mov     a, [hl+2]        ; ポインタを介したアクセス。
```

HUGE アクセス方法

huge 方法は far 方法と同様に同じメモリ範囲にアクセスしますが、64 Kb 内に制限されています。オブジェクトが、とても大きく、読み取り / 書き込み操作が 1 つ以上必要なため、far アクセスよりもかなり遅いです。

例

これらの例はさまざまな方法でメモリに巨大アクセスする方法です。

```
mov     es, byte3(x)    ; グローバル変数 x へのアクセス。
mov     a, es:x

addw    ax, #lwrd(y)    ; グローバル配列のエントリに
movw    hl, ax          ; アクセス。
mov     a, c
addc    a, #byte3(y)
mov     es, a
mov     a, [hl]

mov     a, es:[hl]      ; ポインタを介したアクセス。
```

コールフレーム情報

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

アセンブラ言語で記述したルーチンのデバッグ時にコールスタックを使用できるようにするには、アセンブラディレクティブ CFI を使用して、同等のデバッグ情報をアセンブラソースで提供する必要があります。このディレクティブの詳細は、『*RL78 用 IAR アセンブラリファレンスガイド*』を参照してください。

CFI ディレクティブ

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

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

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

- 追跡可能なリソースを示す*名前ブロック*
- 呼出し規約に対応する*共通ブロック*

- コールフレームで実行された変更を示すデータブロック。通常、これには、スタックポインタが変更された時点、保護レジスタがスタックで待避、復帰した時点についての情報が含まれます。

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	通常のスタックおよび割込みスタックの呼出しフレーム
A、B、C、D、E、H、L、X	通常のレジスタ
CS_REG	コードアドレス空間のビット 16 - 19
ES_REG	データアドレス空間のビット 16 - 19
MACRH	MAC レジスタの上位 16 ビット
MACRL	MAC レジスタの下位 16 ビット
SP	スタックポインタレジスタ
?SPH	?SP20 の上位 4 ビット（常に 0xF）
?RET	リターンアドレス用（20 ビットアドレス）
W0-W127	wrkseg エリア

表 22: 名前ブロックで定義されている呼出しフレーム情報リソース

CFI サポートを持つアセンブラソースの作成

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

1 適当な C ソースコードを使用して開始します。以下に例を示します。

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

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

コマンドラインでは、-lA オプションを使用します。

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

この例のソースコードの場合、リストファイルは次のようになります（スペースの理由でファイルは縮小されます）：

```
NAME Cfi

RTMODEL "__SystemLibrary", "DLib"
RTMODEL "__calling_convention", "iar"
RTMODEL "__code_model", "near"
RTMODEL "__core", "rl78_2"
RTMODEL "__data_model", "near"
RTMODEL "__double_size", "32"
RTMODEL "__far_rt_calls", "false"
RTMODEL "__rt_version", "1"

#define SHT_PROGBITS 0x1

EXTERN F

PUBLIC cfiExample

CFI Names cfiNames0
CFI StackFrame CFA SP NEARDATA
CFI Resource A:8, X:8, B:8, C:8, D:8, E:8, H:8, L:8,
             CS_REG:4, ES_REG:4
CFI VirtualResource ?RET:20
CFI Resource MACRH:16, MACRL:16, W0:8, W1:8, W2:8,
             W3:8, W4:8, W5:8
CFI Resource W6:8, W7:8, W8:8, W9:8, W10:8, W11:8,
             W12:8, W13:8, W14:8
CFI Resource W15:8, W16:8, W17:8, W18:8, W19:8, W20:8,
             W21:8, W22:8
CFI Resource W23:8, W24:8, W25:8, W26:8, W27:8, W28:8,
             W29:8, W30:8
CFI Resource W31:8, W32:8, W33:8, W34:8, W35:8, W36:8,
             W37:8, W38:8
CFI Resource W39:8, W40:8, W41:8, W42:8, W43:8, W44:8,
             W45:8, W46:8
CFI Resource W47:8, W48:8, W49:8, W50:8, W51:8, W52:8,
             W53:8, W54:8
```

```
CFI Resource W55:8, W56:8, W57:8, W58:8, W59:8, W60:8,
             W61:8, W62:8
CFI Resource W63:8, W64:8, W65:8, W66:8, W67:8, W68:8,
             W69:8, W70:8
CFI Resource W71:8, W72:8, W73:8, W74:8, W75:8, W76:8,
             W77:8, W78:8
CFI Resource W79:8, W80:8, W81:8, W82:8, W83:8, W84:8,
             W85:8, W86:8
CFI Resource W87:8, W88:8, W89:8, W90:8, W91:8, W92:8,
             W93:8, W94:8
CFI Resource W95:8, W96:8, W97:8, W98:8, W99:8, W100:8,
             W101:8, W102:8
CFI Resource W103:8, W104:8, W105:8, W106:8, W107:8,
             W108:8, W109:8
CFI Resource W110:8, W111:8, W112:8, W113:8, W114:8,
             W115:8, W116:8
CFI Resource W117:8, W118:8, W119:8, W120:8, W121:8,
             W122:8, W123:8
CFI Resource W124:8, W125:8, W126:8, W127:8, SP:16
             ?SPH:4
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 1
CFI DataAlign 1
CFI ReturnAddress ?RET CODE
CFI CFA SP+4
CFI A Undefined
CFI X Undefined
CFI B SameValue
CFI C SameValue
CFI D SameValue
CFI E SameValue
CFI H Undefined
CFI L Undefined
CFI CS_REG Undefined
CFI ES_REG Undefined
CFI ?RET Frame(CFA, -4)
CFI MACRH Undefined
CFI MACRL Undefined
```

```
        CFI W0 SameValue
        CFI W1 SameValue
             ...
        CFI Wn SameValue...
             ...
        CFI W126 SameValue
        CFI W127 SameValue
        CFI SP Undefined
        CFI EndCommon cfiCommon0

        SECTION LOVE:CODE:NOROOT(0)
cfiExample:
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        PUSH      DE
        CFI E Frame(CFA, -6)
        CFI D Frame(CFA, -5)
        CFI CFA SP+6
        MOVW      DE, AX
        CALL      _F
        MOVW      HL, AX
        MOVW      AX, DE
        ADDW      AX, HL
        POP       DE
        CFI E SameValue
        CFI D SameValue
        CFI CFA SP+4
        RET
        CFI EndBlock cfiBlock0

        END
```

注 : ヘッダファイル `cfi.m` にはマクロ `cfi_Names0` と `cfi_Common0` が含まれており、これらは一般的な名前ブロックおよび一般的な共通ブロックを各 1 つ宣言します。これらの 2 つのマクロは、仮想リソースと具体的なリソースの両方を宣言します。

C の使用

- C 言語の概要
- 拡張の概要
- IAR C 言語拡張

C 言語の概要

RL78 用 IAR C/C++ コンパイラは、ISO/IEC 9899:1999 規格（最新の技術的誤植 No.3 も含む）、通称 C99 をサポートしています。このガイドでは、この規格を*標準の C* と呼び、これがコンパイラで使用されるデフォルト標準です。この標準は C89 よりも厳密です。

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

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

- `inline` キーワードは、キーワードの直後に定義された関数をインライン化するようコンパイラに指示します。
- 宣言と文は、同じスコープ内で混在させることが可能です。
- `for` ループの初期化式における宣言
- `bool` データ型
- `long long` データ型
- 複雑な浮動小数点型
- C++ スタイルのコメント
- 複合リテラル
- 構造体終端の不完全な配列
- 16 進数の浮動小数点定数
- 構造体と配列における指定イニシャライザ
- プリプロセッサ演算子 `_Pragma()`
- 可変数引数マクロは、`printf` スタイルの関数に相当するプリプロセッサマクロです

- VLA（可変長配列）は、コンパイラオプション `--vla` によって明示的に有効化する必要があります
- `asm` キーワードまたは `__asm` キーワードを使用したインラインアセンブラ（145 ページの*インラインアセンブラ*を参照）

注 : たとえ C99 の機能であっても、RL78 用 IAR C/C++ コンパイラは UCN（universal character name ＝汎用文字名）をサポートしていません。

拡張の概要

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

以下は使用可能な拡張の概要です。

- IAR C 言語拡張

  使用可能な言語拡張については、「169 ページの *IAR C 言語拡張*」を参照してください。拡張キーワードの詳細は、「*拡張キーワード*」を参照してください。C++、言語の 2 レベルのサポート、C++ 言語拡張については、「*C++ の使用*」の章を参照してください。
- プラグマディレクティブ

  `#pragma` ディレクティブは、C 規格によって定義されたものであり、ベンダ固有の拡張の使用方法を規定することにより、ソースコードの移植性を維持するための仕組みです。

  コンパイラでは、コンパイラの動作（メモリの割当て方法、拡張キーワードの許可 / 禁止、ワーニングメッセージの表示 / 非表示など）の制御に使用可能な定義済プラグマディレクティブを提供します。ほとんどのプラグマディレクティブは前処理され、マクロに置換されます。プラグマディレクティブは、コンパイラでは常に有効になっています。これらのいくつかに対しては、対応する C/C++ 言語拡張も用意されています。使用可能なプラグマディレクティブについては、「*プラグマディレクティブ*」の章を参照してください。
- プリプロセッサ拡張

  コンパイラのプリプロセッサは、C 規格に準拠しています。また、コンパイラにより、いくつかのプリプロセッサ関連拡張も利用可能になります。詳細については、「*プリプロセッサ*」を参照してください。
- 組込み関数

  組込み関数は、低レベルのプロセッサ処理に直接アクセスするための関数であり、時間が重要なルーチンなどで非常に便利です。組込み関数は、単一の命令か短い命令シーケンスとして、インラインコードにコンパイルさ

れます。組込み関数の使用方法については、「143 ページの *C 言語とアセンブラの結合*」を参照してください。使用可能な関数については、「*組込み関数*」を参照してください。

- ライブラリ関数

  IAR DLIB ライブラリは、組込みシステムに利用される最も重要な C/C++ ライブラリ定義を提供します。詳細については、363 ページの *IAR DLIB ライブラリ*を参照してください。

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

言語拡張の有効化

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

コマンドライン	IDE*	説明
`--strict`	**厳密**	すべての IAR C 言語拡張が無効になり、C 規格外のすべてに対してエラーが発せられます。
なし	**標準**	*C 規格に対するすべての拡張*が有効ですが、組込みシステムのプログラミングの*拡張は一切有効になりません*。拡張については、169 ページの *IAR C 言語拡張*を参照してください。
`-e`	**標準（IAR 拡張あり）**	すべての *IAR C 言語拡張*が有効になります。

表 23: 言語拡張

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

IAR C 言語拡張

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

- *組込みシステムプログラミングの拡張* — 一般的にメモリの制限を満たすために、使用する特定のマイクロコントローラでの効率的な組込みプログラミングに特化した拡張。
- *C 規格に対する緩和* — つまり、C 規格の重要でない問題の緩和や、便利ではあるが重要性の低い構文の拡張。172 ページの *C 規格に対する緩和*を参照してください。

組込みシステムプログラミングの拡張

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

- メモリ属性、型属性およびオブジェクト属性

  関連する概念、一般的な構文規則、リファレンス情報については、「*拡張キーワード*」を参照してください。

- 絶対アドレスへの配置、指定セクションへの配置

  `@` 演算子や `#pragma location` ディレクティブを使用して、グローバル変数や静的変数を絶対アドレスに配置することや、指定されたセクションに変数や関数を配置することができます。これらの機能の使用方法については、206 ページのデータと関数のメモリ配置制御、336 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、295 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は、`#pragma pack` と `#pragma data_alignment` ディレクティブが使用できます。available 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、205 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、298 ページのビットフィールドを参照してください。

- `static_assert()`

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

- 可変数引数マクロのパラメータ

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

専用セクション演算子

コンパイラは、以下のビルトインセクション演算子を使用して、開始アドレスや終了アドレス、セクションのサイズ取得をサポートします。

`__section_begin`	指定のセクションまたはブロックの最初のバイトアドレスを返します。
`__section_end`	指定のセクションまたはブロック*の後にくる*最初のバイトアドレスを返します。
`__section_size`	指定のセクションまたはブロックのサイズ（バイト）を返します。

これらの演算子は、リンカ設定ファイルで定義された指定のセクションまたは指定ブロック上で使用できます。

これらの演算子は構文的に以下のように宣言された場合と同じように動作します。

```
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t __section_size(char const * section)
```

@ 演算子または #pragma location ディレクティブを使用してデータオブジェクトや関数をユーザ定義のセクションに配置したり、指定ブロックをリンカ設定ファイルで使用する場合、セクション演算子を使用して、セクションまたはブロックが配置されたメモリ範囲の開始アドレスと終了アドレスを取得できます。

指定の*セクション*は文字列リテラルである必要があり、#pragma *section* ディレクティブで先に宣言されている必要があります。セクションがメモリ属性 *memattr* により宣言されている場合、__section_begin 演算子の型は、*memattr* void へのポインタとなります。それ以外の場合、型は void へのデフォルトのポインタです。この組込み演算子を使用するには、言語拡張を有効にしておく必要があります。

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

演算子	**シンボル**
`__section_begin(sec)`	`sec$$Base`
`__section_end(sec)`	`sec$$Limit`
`__section_size(sec)`	`sec$$Length`

表 24: セクション演算子とそのシンボル

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

例

以下の例では、`__section_begin` 演算子の型は、`void __near *` です。

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

343 ページの *section*、336 ページの *location* を参照してください。

C 規格に対する緩和

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

- 不完全型の配列

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

- `enum` 型の前方宣言

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

- `struct` または `union` 指定子の最後のセミコロンの欠損を容認する

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

- NULL と void

  ポインタの処理において、void へのポインタは必要に応じて別の型に暗黙的に変換されます。また、NULL ポインタ定数は、必要に応じて適切な型の NULL ポインタに暗黙的に変換されます。C 規格では、一部の演算子でこの動作が認められていますが、そうでないものもあります。

- 静的イニシャライザでのポインタから整数へのキャスト

  イニシャライザでは、ポインタ定数値を整数型にキャストできます（整数型のサイズが十分に大きい場合）。ポインタのキャストに関する詳細については、302 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 (「251 ページの *-e*」を参照)。

- 関数およびブロックスコープ内の静的関数

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

- 数値の構文に従って数値の走査が行われる

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

C++ の使用

- 概要 — EC++ および EEC++
- C++ のサポートの有効化
- EEC++ の機能の説明
- EEC++ の機能の説明
- C++ 言語拡張

概要 — EC++ および EEC++

IAR システムズは C++ 言語をサポートしています。業界標準の Embedded C++ と拡張 Embedded C++ のどちらかを選択できます。ここでは、*C++ 言語を使用する*際の注意事項について説明します。

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

EMBEDDED C++

以下の C++ 機能がサポートされています。

- クラス。データ構造と動作の両方をまとめたユーザ定義の型のことです。本質的な機能である継承により、データ構造と動作を複数のクラスで共有できます
- ポリモフィズム。1 つの処理が異なるクラスで異なる動作を実現できる機能です。仮想関数によって提供されます
- 演算子と関数名のオーバーロード。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ヘッドが、実行時間やコードサイズに生じるような C++ 機能は除外されています。他に除外されているのは、C++ 標準が定義される直前に追加された機能です。したがって、Embedded C++ は、効率性が高く、既存の開発ツールで完全にサポートされている C++ のサブセットを提供します。

Embedded C++ は、以下の C++ の機能が削除されています。

- テンプレート
- 多重継承と仮想継承
- 例外処理
- ランタイムの型情報
- 新しいキャスト構文（演算子 dynamic_cast、static_cast、reinterpret_cast、const_cast）
- 名前空間
- mutable 属性

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

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

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

拡張 EMBEDDED C++

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

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

これらの追加機能は、C++ 規格に準拠しています。

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

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

C++ のサポートの有効化

コンパイラでは、デフォルトの言語は C です。

Embedded C++ で記述されたファイルをコンパイルするには、`--ec++` コンパイラオプションを使用します。252 ページの *--ec++* を参照してください。

拡張 Embedded C++ の機能をソースコードで利用するには、`--eec++` コンパイラオプションを使用します。252 ページの *--eec++* を参照してください。

IDE で EC++ または EEC++ 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］**に続いて、適切な標準を選択します。

EEC++ の機能の説明

RL78 用 IAR C/C++ コンパイラ用の C++ ソースコードを記述する際、C++ の機能（クラス、クラスメンバなど）と IAR 言語拡張（IAR 固有の属性など）を組み合わせる場合の利点や、特異な動作の可能性について認識しておく必要があります。

IAR 属性とクラスを使用する

C++ クラスの静的データメンバは、グローバル変数と同じように処理され、適切なすべての IAR 型、メモリ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、メモリとオブジェクト属性を持つことができます。仮想メンバ関数はデフォルトの関数ポンタと互換性のある属性しか持つことができません。コンストラクタとデストラクタはこうした属性を持つことはできません。

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

属性とクラスの使用例

```
class MyClass
{
public:
  // 静的変数の場所を __saddr メモリ内の
  // アドレス 0xFFE30 に指定
  static __saddr __no_init int mI @ 0xFFE30;

  // 静的関数を __near_func メモリ内に配置
  static __near_func void F();

  // 関数を __far_func メモリ内に配置
  __far_func void G();

  // 仮想関数を __near_func メモリ内に配置
  virtual __near_func void H();

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

this ポインタ

クラスオブジェクトの参照やクラスオブジェクトのメンバ関数の呼出しに使用される `this` ポインタは、デフォルトのデータポインタ型のデータメモリ属性をデフォルトで持ちます。これはつまり、このようなクラスオブジェクトは、ポインタがそこからデフォルトのデータポインタに黙示的に変換できるようなメモリ内に存在するようにしか定義できないことを意味します。この制限は、一時オブジェクトや自動オブジェクトなど、スタック上にあるオブジェクトに適用されることもあります。

class メモリ

この制限を補うために、クラスを *class メモリタイプ*に関連付けることができます。class メモリタイプは次の変更を行います。

- すべてのメンバ関数、コンストラクタ、デストラクタ中の `this` ポインタ型を class メモリへのポインタに変更
- class 型の静的記憶寿命変数のデフォルトメモリ、すなわち自動でない変数を、指定された class メモリに変更
- class 型のオブジェクトをポイントするときに使用されるポインタ型を class メモリに変更

例

```
class __far C
{
public:
  void MyF();        // C __far 型の this ポインタを 1 つ持ちます *
  void MyF() const;  // C __far const 型の
                     // this ポインタを 1 つ持ちます *
  C();               // this ポインタが far メモリを
                     // ポイント
  C(C const &);      // C __far
                     // const & 型のパラメータ 1 つを受け入れ
                        (生成されたコピー
                     // コンストラクタにも適用)
  int mI;
};

C Ca;                // デフォルトメモリではなく far メモリ内に
                     // 存在
C __near Cb;         // near メモリ内に存在、'this'
                     // ポインタはそのまま far メモリをポイント

void MyH()
{
  C cd;              // スタック上に存在
}

C *Cp1;              // far メモリへのポインタを作成
C __near *Cp2;       // near メモリへのポインタを作成
```

class メモリタイプに関連付けられた class 型（c など）を宣言しなければならないときは、class メモリタイプも記述する必要があります。

```
class __far C;
```

また、異なる class メモリに関連付けられた class 型は、互換性のある型ではないことに注意してください。

組込み演算子はクラス、`__memory_of(class)` に関連付けられた class メモリタイプを返します。たとえば、`__memory_of(C)` は `__far` を返します。

継承する際は、サブクラスへのポインタをその基底クラスへのポインタに黙示的に変換できなければなりません。つまり、サブクラスはその基底クラスよりも多くの制限を持つ class メモリを持つことができますが、制限の*より少ない* class メモリを持つことはありえません。

```
class __far D : public C
{ // OK。同じ class メモリです
public:
  void MyG();
  int mJ;
};

class __near E : public C
{ // OK。near メモリが far の内部に
public:
  void MyG() // this ポインタが near メモリをポイント
  {
    MyF();    // far メモリへの this ポインタを取得
  }
  int mJ;
};

class F : public C
{ // OK。C と同じ class メモリに関連付けられます
public:
  void MyG();
  int mJ;
};
```

クラス上の式 new によって、class メモリに関連付けられたヒープ内でメモリが割り当てられます。式 delete は、割当てを同じヒープに自然に解除します。クラスのデフォルトの new と delete 演算子をオーバライドするには、次のように宣言します。

```
void *operator new(size_t);
void operator delete(void *);
```

通常の C++ と同じようにメンバ関数としてです。

メモリタイプについて詳しくは、62 ページの*メモリタイプ*を参照してください。

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
}
```

演算子 NEW と DELETE

各メモリ属性とそれに関連するヒープには、演算子 new と delete があります。これらのヒープは、__near、__far、__huge です。

```
#include <stddef.h>

// __near、__far、__huge のヒープがあることを想定します
void __far   *operator new __far (__far_size_t);
void __near  *operator new __near(__near_size_t);
void __huge   *operator new __huge (__huge_size_t);
void operator delete(void __far  *);
void operator delete(void __near *);
void operator delete(void __huge *);

// new 演算子と delete 演算子の配列に従います
void __far   *operator new[] __far (__far_size_t);
void __near  *operator new[] __near(__near_size_t);
void __huge  *operator new[] __huge(__huge_size_t);
void operator delete[](void __far  *);
void operator delete[](void __near *);
void operator delete[](void __huge *);
```

任意のデータメモリについて、グローバルとクラス固有の演算子 new と演算子 delete をどちらもオーバライドする場合は、この構文を使用します。

各メモリに operator new 関数に名前を指定するための特殊な構文があることに注意してください。ただし、operator delete 関数のネーミングは、通常のオーバロードに依存します。

式 New と delete

式 new は、指定したタイプのメモリについて operator new 関数を呼び出します。class メモリとともに class、struct、union 型を使用する場合、呼び出される operator new 関数はその class メモリによって決まります。以下に例を示します。

```
void MyF()
{
  // Calls operator new __near(__near_size_t)
  int __near *p = new __near int;

  // operator new __near(__near_size_t) の呼出し
  int __near *q = new int __near;

  // operator new[] __near(__near_size_t) の呼出し
  int __near *r = new __near int[10];

  // operator new __far(__far_size_t) の呼出し
  class __far S
  {
  };
  S *s = new S;

  // operator delete(void __near *) の呼出し
  delete p;
  // operator delete(void __far  *) の呼出し
  delete s;

  int __far *t = new __near int;
  delete t; // エラー : ヒープが破損します
}
```

delete 式で使用されるポインタは正しいタイプでなければなりません。つまり、new 式で返されるものと同じタイプということです。間違ったメモリへのポインタを使用すると、ヒープが破損することがあります。

割込みで静的クラスオブジェクトを使用する

割込み関数が、（コンストラクタを使用して）生成または（デストラクタを使用して）破棄しなければならない静的クラスオブジェクトを使用する場合、オブジクトの生成前、または破棄の後や途中に割込みが発生すると、アプリケーションが正しく機能しなくなります。

これを回避するために、静的オブジェクトが構築されるまで、これらの割込みが有効になっておらず、main から戻ったり、exit を呼出すときに無効になっていることを確認します。システムの起動について詳しくは、122 ページの *システムの起動と終了*を参照してください。

関数ローカルの静的クラスオブジェクトは、実行が最初に宣言を通過すると生成され、main から戻ったり、exit を呼出す際に破棄されます。

新しいハンドラを使用する

メモリの消耗に対応するには、set_new_handler 関数を使用します。

Embedded C++ の新しいハンドラ

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

NULL 以外の新規ハンドラを用いて set_new_handler を呼出す場合、operator new がメモリの割当てに失敗すると、operator new によって提供された新規ハンドラが呼出されます。新規ハンドラはより多くのメモリを使用できるようにして、何らかの形で実行を返すか、中止する必要があります。operator new の派生型 nothrow は、新規ハンドラがある状態で NULL を返すことはありません。

テンプレート

拡張 EC++ は、C++ 標準に基づいてテンプレートをサポートしますが、export キーワードはサポートしません。実装では、2 段階のルックアップを使用します。すなわち、必要なときには常に typename キーワードを挿入する必要があります。さらに、テンプレートを使用するたびに、使用可能なすべてのテンプレート定義が可視になる必要があります。すなわち、すべてのテンプレートの定義がインクルードファイルまたは実際のソースファイルに存在する必要があります。

C-SPY でのデバッグサポート

C-SPY には、STL コンテナ用に組込みの表示サポートがあります。コンテナの論理構造は、わかりやすく追跡しやすい方法で包括的に［ウォッチ］ビューに表示されます。

これらの詳細は、『*RL78 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

コンパイラは、標準 C++ で定義された構文および動作を持つテンプレートをサポートしています。ただし、製品に付属の STL（標準テンプレートライブラリ）は、拡張 EC++ 用に調整されています（178 ページの*拡張 Embedded C++* を参照）。

テンプレートおよびデータメモリ属性

データメモリ属性が予期したとおりにテンプレートで機能するように、標準の C++ テンプレート処理の 2 つの要素が変更されました — クラステンプレートの部分的特化照合と関数テンプレートのパラメータ減算です。

拡張 Embedded C++ では、クラステンプレートの部分的特化照合アルゴリズムが次のように機能します。

ポインタや参照タイプがテンプレートパラメータタイプへのポインタや参照に対して照合されると、ポイントされるタイプはテンプレートパラメータタイプになり、結果のポインタや参照タイプが同じ場合には、すべてのデータメモリ属性が削除されます。

例

```
// __far がデフォルトポインタのメモリタイプ
// とします。
template<typename> class Z {};
template<typename T> class Z<T *> {};

Z<int __near *> Zn;    // T = int __near
Z<int __far  *> Zf;    // T = int
Z<int        *> Zd;    // T = int
```

拡張 Embedded C++ では、関数テンプレートのパラメータ減算アルゴリズムが次のように機能します。

関数テンプレート照合が実行され、引数が減算に使用されます。その引数がデフォルトのポインタに黙示的に換算できるメモリへのポインタである場合、デフォルトのポインタであるかのようにパラメータ減算を実行してください。

引数が参照に対して照合されるときは、引数とパラメータがどちらもポインタであるかのように減算を実行します。

例

```
// __far がデフォルトポインタのメモリタイプ
// とします。
template<typename T> void fun(T *);

void MyF()
{
  fun((int __near *) 0); // T = int. クラステンプレートの特化がある
                         // 類似した場合とは
                         // 結果が異なります。
  fun((int        *) 0); // T = int
  fun((int __far  *) 0); // T = int
}
```

この修正済アルゴリズムを使用して照合されるテンプレートの場合は、small メモリタイプへのポインタ用に特殊なコードを自動生成することはできません。large および他のメモリタイプ（デフォルトのポインタではポイントできないメモリ）では、これは可能です。メモリを完全に認識するテンプレートの作成を可能にするには（その方が便利という稀な場合のために）、テンプレート関数の宣言の前に `#pragma basic_template_matching` ディレクティブを使用してください。そのテンプレート関数によって、上記の修正なしに照合が行われます。

例

```
// __far がデフォルトポインタのメモリタイプ
// とします。
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
  fun((int __near *) 0); // T = int __near
```

標準テンプレートライブラリ

`vector` および `map` のような STL のコンテナは、メモリ属性を認識します。つまり、特定のメモリタイプ内に存在するようにコンテナを宣言できます。こうすると次のような結果になります。

- コンテナ自体が選択したメモリ内に存在するようになる
- コンテナ内のエレメントの割当てでは選択したメモリのヒープが使用される
- その中にあるすべての参照では、選択したメモリへのポインタが使用される

例

```
#include <vector>

vector<int> D;                 // デフォルトメモリで D
                               // デフォルトのヒープを使用
                               // デフォルトのポインタを使用
vector<int __near> __near X;   // X を near メモリに配置、
                               // near からヒープを割当て、
                               // near メモリへのポインタを使用
vector<int __far> __near Y;    // Y を near メモリに配置、
                               // far からヒープを割当て、
                               // far メモリへのポインタを使用
```

`map<key、T>`、`multimap<key、T>`、`hash_map<key、T>`、`hash_multimap<key、T>` はすべて `T` のメモリを使用する点に注意してください。これはつまり、これらのコレクションの `value_type` は `pair<key, const T> mem` で、*mem* はメモリタイプが `T` ということです。メモリタイプを持ったキーの指定は実用的ではありません。

例

使用するデータメモリ属性だけが異なる 2 つのコンテナを、相互に割り当てることはできないので注意してください。代わりに、テンプレートを持つメンバ割当て方法を使用する必要があります。

```
#include <vector>

vector<int __near> X;
vector<int __far>  Y;

void MyF()
{
  // テンプレートメンバ割当て方法を使用できます
  X.assign(Y.begin(), Y.end());
  Y.assign(X.begin(), X.end());
}
```

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。

メンバ関数へのポインタ

メンバ関数へのポインタに含めることができるのは、デフォルトの関数ポインタ、またはデフォルトの関数ポインタに暗黙的にキャストできる関数ポインタのみです。メンバ関数へのポインタを使用するには、ポイント先のすべての関数がデフォルトのメモリまたはデフォルトのメモリに含まれるメモリ内に存在することを確認してください。

例

```
class X
{
public:
  __near_func void F();
};

void (__near_func X::*PMF)(void) = &X::F;
```

C++ 言語拡張

コンパイラをいずれかの C++ モードで使用し、IAR 言語拡張を有効にする場合、以下の C++ 言語拡張がコンパイラで使用できます。

- クラスの friend 宣言において、class キーワードを省略できます。以下に例を示します。

```
class B;
class A
{
  friend B;        // IAR 言語拡張を用いると
                   // 可能
  friend class B; // 標準規格に従った書き方
};
```

- スカラ型の定数は、クラス内で定義できます。以下に例を示します。

```
class A
{
  const int mSize = 10; // IAR 言語拡張を用いると
                        // 可能
  int mArr[mSize];
};
```

  規格では、初期化した静的データメンバを代わりに使用することになっています。

- クラスメンバの宣言において、修飾名を使用できます。以下に例を示します。

```
struct A
{
  int A::F(); // IAR 言語拡張を用いると可能
  int G();    // 標準規格に従った書き方
};
```

- C リンケージ (extern "C") を持つ関数のポインタと、C++ リンケージ (extern "C++") を持つ関数のポインタとの間の暗黙的な型変換の使用が許可されています。以下に例を示します。

```
extern "C" void F(); // C リンケージを持つ関数
void (*PF)()         // pf は C++ リンケージを持つ関数を指す
             = &F;   // ポインタの暗黙の変換
```

  規格では、ポインタは明示的に変換する必要があります。

- ? 演算子を含む構造体の 2 番目または 3 番目のオペランドが文字列リテラルまたはワイド文字列リテラル（C++ の場合の定数）の場合、オペランドを暗黙的に char * または wchar_t * に変換できます。以下に例を示します。

```
bool X;

char *P1 = X ?"abc" :"def";          // IAR 言語拡張を用いると
                                     // 可能
char const *P2 = X ? "abc" : "def"; // 標準規格に従った書き方
```

- 関数パラメータに対するデフォルトの引数は、規格に従ったトップレベルの関数宣言の中ではなく、typedef 宣言の中、関数へのポインタの関数宣言の中、メンバへのポインタの関数宣言の中でも指定できます。
- 非静的ローカル変数を含む関数、評価されない式（sizeof 式など）を含むクラスにおいては、式から非静的ローカル変数を参照できます。ただし、ワーニングが出力されます。
- typedef 名によって、含有クラスに匿名共用体を導入できます。最初に共用体を宣言する必要はありません。次に例を示します。

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

class A
{
public:
  U;  // OK -- A::i および A::j への参照が許可されています。
};
```

また、この拡張は *anonymous classes* と *anonymous structs* も許可します。ただし、C++ の機能がなく（たとえば、静的データメンバやメンバ関数を持たず、パブリックでないメンバがないなど）、他の匿名クラスや構造体、共用体以外のネスト型を持たないことが条件です。次に例を示します。

```
struct A
{
  struct
  {
    int i,j;
  }; // OK -- A::i および A::j への参照が許可されています。
};
```

- friend class の構文では、nonclass 型のほか、精密型名なしに `typedef` によって表現されたクラス型も使用可能です。次に例を示します。

```
typedef struct S ST;

class C
{
public:
  friend S;  // OK（S がスコープ内にある必要あり）
  friend ST; // OK（"friend S;" と同じ）
  // friend S const; // エラー、cv-qualifiers は直接
                     // 現れることはできません
};
```

注 : 最初に言語拡張を有効にせずに、これらの構造体のいずれかを使用すると、エラーが出力されます。

アプリケーションに関する考慮事項

- 出力形式に関する注意事項
- スタックについて
- ヒープについて
- ツールとアプリケーション間の相互処理
- チェックサムの計算
- リンカの最適化

出力形式に関する注意事項

リンカは、ELF/DWARF オブジェクトファイル形式で絶対実行可能イメージを生成します。

絶対 ELF イメージは、IAR ELF Tool (`ielftool`) を使用して、メモリへの直接ロードや PROM またはフラッシュメモリなどへの書込みに適したフォーマットに変換できます。

`ielftool` では、以下の出力形式を生成できます。

- バイナリ
- Motorola S-records
- Intel hex

注：`ielftool` は、絶対イメージ内のチェックサムの埋め込みや計算など、別のタイプの変換にも使用できます。

`ielftool` のソースコードは、`rl78¥src` ディレクトリにあります。`ielftool` の詳細は、413 ページの *IAR ELF ツール — ielftool* を参照してください。

スタックについて

お使いのアプリケーションでスタックメモリを効果的に使用するには、考慮することがいくつかあります。

スタックサイズについて

必要なスタックサイズはアプリケーションの動作によって大きく異なります。スタックサイズが大きすぎる場合は、RAM が無駄に消費されます。スタックサイズが小さすぎる場合には、2 つうちのどちらかが発生します。これは、スタックのメモリ上の位置により異なり、

- 変数記憶領域が上書きされ、未定義の動作を引き起こします
- スタックの位置がメモリエリアを超えアプリケーションが異常終了します

いずれの場合もアプリケーション障害が発生します。後者は発見が容易なため、メモリの最後に向かって大きくなるようにスタックを配置するのが得策です。

スタックサイズの詳細については、99 ページの*スタックメモリの設定*、217 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。

ヒープについて

ヒープには、C 関数 `malloc`（あるいは関連関数）か C++ の演算子 `new` を使用して割り当てられた動的データが格納されます。

アプリケーションで動的メモリ割当てを使用する場合には、以下の内容に精通しておく必要があります。

- ヒープに使用されるリンカセクション
- ヒープサイズの割当て。詳細については、100 ページの*ヒープメモリの設定*を参照してください

DLIB のヒープセクション

特定のメモリ内のヒープにアクセスするには、標準関数の `malloc`、`free`、`calloc`、`realloc` の先頭に適切なメモリ属性をプレフィックスとして使用します。たとえば、次のようになります。

```
__near_malloc
```

標準関数のいずれかにプレフィックスを付けて使用すると、その関数はデフォルトのメモリタイプ near にマップされます。

それぞれのヒープは、たとえば NEAR_HEAP など、メモリ属性をプレフィックスに持つ _HEAP という名前のセクションに配置されます。

使用可能なヒープについては、139 ページの *ヒープ*を参照してください。

ヒープサイズと標準 I/O

通常の設定のように FILE 記述子を DLIB ランタイムライブラリから除外すると、I/O バッファが無効になります。詳細設定のように、それ以外の場合は、stdio ライブラリのヘッダファイルで I/O バッファが 512 バイトに設定されます。ヒープが小さすぎる場合は、I/O がバッファされず、I/O がバッファされた場合よりも大幅に低速になります。IAR C-SPY® デバッガのシミュレータドライバを使用してアプリケーションを実行する場合には、速度低下が現れない可能性がありますが、アプリケーションを RL78 マイクロコントローラで実行すると、速度低下を明確に認識できます。標準 I/O ライブラリを使用する場合は、ヒープサイズを標準 I/O バッファの必要に応じたサイズに設定してください。

ツールとアプリケーション間の相互処理

リンクプロセスとアプリケーションでシンボルを相互処理する方法は以下の 4 種類があります。

- リンカコマンドラインオプション --define_symbol を使用してシンボルを作成する。リンカは、アプリケーションがラベル、サイズ、デバッガのセットアップなどとして使用できるパブリック絶対定数シンボルを作成します。
- コマンドラインオプション --config_def または設定ディレクティブ define symbol を使用し、export symbol ディレクティブを使用しシンボルをエクスポートして、エクスポート済み設定シンボルを作成する。LINK は、アプリケーションがラベル、サイズ、デバッガのセットアップなどとして使用できるパブリック絶対定数シンボルを作成します。

  このシンボル定義の利点の 1 つは、このシンボルを設定ファイルで式として使用できる点です。たとえば、メモリ範囲へのセクションの配置を制御するときなどに使用します。
- コンパイラ演算子 __section_begin、__section_end または __section_size、またはアセンブラ演算子 SFB、SFE、または指定の SIZEOF セクションまたは block を使用します。これらの演算子は、開始アドレス、終了アドレス、セクションの連続シーケンスに、同じ名前、またはリンカ設定ファイルで指定されたリンカブロックのシーケンスを提供します。

- コマンドラインオプション --entry は、アプリケーションの開始ラベルをリンカに通知します。これは、リンカより、実行の開始位置をデバッガに通知するときのルートシンボルとして使用されます。

次のラインは -D を使用してシンボルを作成する方法を示します。この方法を使用する必要がある場合は、これらのオプションをこのようなコマンドラインに追加します：

```
--define_symbol NrOfElements=10
--config_def HEAP_SIZE=1024
```

リンカ設定ファイルでは、次のように定義されています。

```
define memory Mem with size = 4G;
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* シンボルをエクスポートする */
export symbol MY_HEAP_SIZE;

/* ILINK のオプションで定義された大きさのヒープエリアを設定 */
define block MyHEAP with size = MY_HEAP_SIZE, alignment = 4 {};

place in RAM { block MyHEAP };
```

以下の行をアプリケーションソースコードに追加します。

```
#include <stdlib.h>

/* ILINK オプションで定義したシンボルを使い、指定したサイズのエレメント配列
を動的に割り当てます。値はラベルの形式をとります。
 */
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
  return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* ILINK オプションで定義したシンボルを使う。
 * リンカ設定ファイル内のシンボルはアプリケーションで使用できるようになってい
る。
 */
extern char MY_HEAP_SIZE;
```

```
/* ヒープを含むセクションを宣言 */
#pragma section = "MYHEAP"

char *MyHeap()
{
  /* 最初に、静的に配置されたセクションの最初アドレスを得る */
  char *p = __section_begin("MYHEAP");

  /* インポートしたヒープサイズを使用し、0 で初期化する */
  for (int i = 0; i < (int) &MY_HEAP_SIZE; ++i)
  {
    p[i] = 0;
  }
  return p;
}
```

チェックサムの計算

IAR ELF Tool (`ielftool`) は、特定の範囲のメモリをパターンで埋め、これらの範囲のチェックサムを計算します。計算されるチェックサムは、入力 ELF イメージの既存シンボルの値を置換します。アプリケーションは、これらの範囲が変更されていないか検証できます。

チェックサムを使用してアプリケーションの整合性を検証する場合、以下のことを行う必要があります。

- `ielftool` により計算されるチェックサムについて、配置を関連する名前とサイズとともに予約する
- チェックサムアルゴリズムを選択し、その `ielftool` を設定して、アルゴリズムのソースコードをアプリケーションに含める
- アプリケーションソースコードで`ielftool`とそのソースコードの両方を検証および設定するメモリ範囲を決定する

IDE に `ielftool` を設定するには、［**プロジェクト**］**>**［**オプション**］**>**［**リンカ**］**>**［**チェックサム**］を選択します。

チェックサムの計算

この例では、`0x8002` ～ `0x8FFF` にある ROM メモリのチェックサムが計算されます。また、計算された 2 バイトチェックサムが `0x8000` に配置されます。

計算されるチェックサムの配置の作成

計算されるチェックサムの配置は、2 とおりの方法で作成できます。特定のセクション（この例では `.checksum`）に常駐する正しいサイズのグローバル

C/C++ またはアセンブラ定数シンボルを作成する方法と、リンカオプション --place_holder を使用する方法です。

たとえば、シンボル _checksum の 2 バイトスペースをセクション .checksum にアラインメント 4 で作成するには、以下のようにします。

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

注：.checksum セクションは、必要と思われる場合に、アプリケーションにのみインクルードされます。アプリケーション自体でチェックサムが必要でない場合、リンカオプション --keep=__checksum か、リンカディレクティブ keep を使用して、セクションを強制的にインクルードすることができます。

.checksum セクションを配置するには、リンカ設定ファイルを修正する必要があります。例えば、これを次に示します（ブロック CHECKSUM の扱いに注意してください）。

```
define memory Mem with size = 4G;

define region ROM_region = Mem:[from 0x8000 to 0x80000000 - 1];
define region RAM_region = Mem:[from 0x80000000 to 0x100000000 -2
];

initialize by copy { rw };
do not initialize  { section .noinit };

define block HEAP      with alignment = 2, size = 16M {};
define block CSTACK    with alignment = 2, 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 };
```

ielftool の実行

チェックサムを計算するには、ielftool を実行します。

```
ielftool --fill=0x00;0x8000-0x8FFF
--checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out
destinationFile.out
```

チェックサムを計算するには、フィル操作を定義する必要があります。この例では、フィルパターン 0x0 が使用されます。使用されるチェックサムアルゴリズムは crc16 です。

ielftool では、リンカオプションを使用していない ELF イメージが必要です。--strip リンカオプションを使用する場合、これを削除し、代わりに --strip ielftool オプションを使用します。

チェックサム関数をソースコードに追加する

ielftool により生成されるチェックサムの値をチェックするには、アプリケーションにより計算されたチェックサムと比較する必要があります。つまり、チェックサム計算用の関数（ielftool と同じアルゴリズムを使用）をアプリケーションソースコードに追加する必要があります。アプリケーションには、この関数へのコールを含める必要があります。

チェックサム計算用の関数

以下の関数（計算時間は遅いがメモリ使用量は少ない版）では、crc16 アルゴリズムが使用されます。

```
unsigned short SlowCrc16(unsigned short sum,
                         unsigned char *p,
                         unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

    for (i = 0; i < 8; ++i)
    {
      unsigned long oSum = sum;
      sum <<= 1;
      if (byte & 0x80)
        sum |= 1;
      if (oSum & 0x8000)
        sum ^= 0x1021;
      byte <<= 1;
    }
  }
  return sum;
}
```

チェックサムアルゴリズムのソースコードは、製品インストールの rl78¥src¥linker ディレクトリにあります。

チェックサムの計算の例

以下のコードは、チェックサムがどのように計算されるかを示した例です。

```
/* The checksum calculated
 * (これはアドレス 0x8000 上にあります)
 */
extern unsigned short const _checksum;


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

  /* チェックサム計算の実行 */
  calc = SlowCrc16(0,
                   (unsigned char *) checksumStart,
                   (checksumEnd - checksumStart+1));

  /* 結果を格納 */
  calc = SlowCrc16(calc, zeros, 2);

  /* チェックサムのテスト */
  if (calc != _checksum)
  {
    abort();   /* 失敗 */
  }
}
```

注意事項

チェックサムを計算する場合、以下のことに注意してください。

- チェックサムは、すべてのメモリ範囲で最下位アドレスから最上位アドレスに計算する必要があります
- 各メモリ範囲は定義されている順序で検証する必要があります（ABC は ACB とは異なります）
- 1 つのチェックサムに対して複数の範囲が存在することは問題ありません
- 複数のチェックサムが使用される場合、セクションごとに一意のシンボル名を使用する必要があります
- 低速の関数バリアントが使用される場合、チェックサム計算の最後の呼出しは、チェックサム内のバイト数と同じバイト数（値 0x00）で行う必要があります

- チェックサムが含まれている場所のチェックサムは計算しません。

詳細については、413 ページの *IAR ELF ツール— ielftool* を参照してください。

C-SPY に関する注意事項

デフォルトでは、リンカオプション `--place_holder` を使用してメモリ内に割り当てたシンボルは、C-SPY によって `int` 型であると見なされます。チェックサムのサイズが `int` のサイズとは異なる場合、そのサイズに合わせてチェックサムシンボルの表示フォーマットを変更できます。

C-SPY の［ウォッチ］ウィンドウでシンボルを選択し、コンテキストメニューから［表示フォーマット］を選択します。チェックサムシンボルのサイズに合った表示フォーマットを選択します。

リンカの最適化

仮想関数の除去

仮想関数除去 (VFE) は、不要な仮想関数と動的ランタイム型情報を除去するリンカの最適化です。

仮想関数除去が機能するためには、仮想関数テーブルについての情報や、どの仮想関数が呼出されるか、どのクラスの動的ランタイム型情報が必要かをすべての適切なモジュールで指定する必要があります。1 つまたは複数のモジュールでこの情報が得られない場合、リンカでワーニングが生成され、仮想関数除去は実行されません。

このような情報を持たないモジュールが仮想関数の呼出しを実行せず、仮想関数テーブルを定義しないことが分かっている場合、`--vfe=forced` リンカオプションを使用して、仮想関数除去を有効にできます。

現在は IAR システムズが、リンカで使用可能な形での仮想関数除去に必要な情報を提供しています。

仮想関数除去は、`--no_vfe` リンカオプションを使用して完全に無効にすることができます。この場合、VFE 情報を持たないモジュールについてワーニングは出力されません。

詳細については、292 ページの *--vfe*、288 ページの *--no_vfe* を参照してください。

組込みアプリケーション用の効率的なコーディング

- データ型の選択
- データと関数のメモリ配置制御
- コンパイラ最適化の設定
- 円滑なコードの生成

データ型の選択

データを効率的に処理するため、使用するデータ型や最も効率的な変数配置を検討する必要があります。

効率的なデータ型の使用

使用するデータ型は、コードのサイズ / 速度に大きく影響することがあるため、慎重に検討する必要があります。

- アプリケーションで signed の値が必要でない限り、small と unsigned のデータタイプ （`unsigned char` と `unsigned short`）を使用してください。
- `double` や `long long` など、64 ビットのデータ型をなるべく使用しないようにしてください。
- サイズが 1 ビット以外のビットフィールドは、ビット処理に比べて不十分なコードの原因となるため、回避してください。
- 配列を使用する場合は、インデックスの式が配列のメモリのインデックスタイプと一致すると効率がさらに良くなります。
- 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` データをポイントするポインタパラメータを宣言すると、呼出し元の関数でより高度な最適化が可能になることがあります。

サポートされているデータ型、ポインタ、構造体の表現の詳細は、「*データ表現*」を参照してください。

浮動小数点数型

数値演算コプロセッサのないマイクロプロセッサで浮動小数点数型を使用するのは、コードサイズと実行速度の両面で非常に非効率です。そのため、浮動小数点数演算を使用するコードを、整数演算を使用するコードに置き換えることを検討してください。これにより効率が向上します。

コンパイラは、2 種類（32 ビットと 64 ビット）の浮動小数点数フォーマットをサポートしています。32 ビット浮動小数点数型の `float` の方は、コードサイズと実行速度の両面において効率が優れます。一方、64 ビットフォーマットの `double` は、より高い精度とより大きな数値に対応します。

コンパイラでは、浮動小数点型の `float` は常に 32 ビットフォーマットを使用します。`double` 浮動小数点型で使用するフォーマットは、コンパイラオプション `--double` の設定によって異なります。

浮動小数点型の詳細については、299 ページの*基本データ型浮動小数点数型*を参照してください。

構造体エレメントのアラインメント

RL78 マイクロコントローラでは、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、2 の位置でアラインメントを保守するため、*パディングバイト*を挿入しなければならないことがあります。

これが問題になりうる状況があります。

- 外部要件。たとえば、ネットワーク通信プロトコルは通常、間にパディングのないデータ型に関して指定されます。
- データメモリを節約する必要がある場合。

アラインメントの要件については、295 ページの*アラインメント*を参照してください。

構造体のレイアウト密度を高くするには、`#pragma pack` ディレクティブを使用してください。欠点は構造体のアラインメントされていないエレメントにアクセスするたびにコードが使用されることです。

または構造体の*パック / アンパック*用のユーザカスタム関数を記述する。こちらの方が移植性が高く、ユーザカスタム関数以外の追加コードは生成されません。欠点として、構造体のデータをパックとアンパックの 2 つの状態で確認する必要があります。

`#pragma pack` ディレクティブの詳細は、340 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。RL78 用 IAR C/C++ コンパイラでは、言語拡張が有効になっていれば、これらを C で使用できます。

IDE では、デフォルトで言語拡張が有効になっています。

言語拡張を有効にするには、`-e` コンパイラオプションを使用します。詳細については、251 ページの *-e* を参照してください。

例

以下の例では、匿名の `union` のメンバに、`union` 名を明示的に指定せずに、関数 `F` でアクセスできます。

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;

void F(void)
{
  St.mL = 5;
}
```

メンバ名は、その前後のスコープ内で固有なものであることが必要です。匿名の `struct/union` を、ファイルスコープレベルで、グローバル変数、外部変

数、静的変数のいずれかとして使用することもできます。これは、以下の例のように、I/O レジスタの宣言などに使用できます。

```
__no_init volatile __sfr
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char way: 1;
    unsigned char out: 1;
  };
} @ 0xFFF80;

/* 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 `0xFFF80` で宣言します。この I/O レジスタでは、`way` と `out` の 2 ビットが宣言されます。内部の構造体と外部の共用体のいずれも匿名になっています。

匿名の構造体や共用体はオブジェクトとして実装されます。このオブジェクトの名前は、最初のフィールドにプレフィックス `_A_` を付けた名前となり、名前空間の実装部で配置されます。この例では、匿名共用体は `_A_IOPORT` というオブジェクトを使用して実装されます。

データと関数のメモリ配置制御

コンパイラでは、関数とデータオブジェクトのメモリへの配置を制御するためのさまざまな仕組みを提供しています。メモリを効率的に使用するためには、これらの仕組みに精通し、さまざまな状況に応じて最適な方法を判別できることが必要です。これらを以下に示します。

- コードモデル

  コードモデルを選択して、関数のデフォルトのメモリ配置を設定できます。詳細については、73 ページの*関数格納のためのコードモデルとメモリ属性*を参照してください。

- データモデル

  データモデルを選択して、変数と定数を配置するためのデフォルトのメモリを設定できます。詳細については、68 ページのデータモデルを参照してください。

- データメモリ属性

  IAR 指定のキーワードまたはプログラムディレクティブを使用して、関数、変数、およびコンスタントのデフォルトの場所を上書きできます。詳細については、74 ページの関数メモリ属性の使用、63 ページのデータメモリ属性の使用を参照してください。

- `@` 演算子および `#pragma location` ディレクティブによる絶対配置

  `@` 演算子または `#pragma location` ディレクティブを使用して、個々のグローバル変数および静的変数を絶対アドレスに配置できます。ただし、この表記を個々の関数の絶対配置に使用することはできません。詳細については、207 ページの絶対アドレスへのデータ配置を参照してください。

- `@` 演算子および `#pragma location` ディレクティブによるセクションの配置

  `@` 演算子または `#pragma location` ディレクティブを使用して、セクションの名前で個々の関数、変数、および定数を配置できます。これらセクションの配置はリンカディレクティブによって制御されます。詳細については、209 ページのデータと関数のセクションへの配置を参照してください。

- `--code_section` オプション

  `--code_section` オプションを使用して、指定の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のメモリに転送する場合に便利です。`--code_section` オプションの詳細については、243 ページの *--code_section* を参照してください。

絶対アドレスへのデータ配置

`@` 演算子か `#pragma location` ディレクティブを使用して、グローバル変数や静的変数を絶対アドレスに配置できます。変数は、次のキーワードの組合せのいずれかを使用して宣言する必要があります。

- `__no_init`
- `__no_init` と `const`（イニシャライザなし）

変数を絶対アドレスに配置するには、`@` 演算子や `#pragma location` ディレクティブの引数に、実際のアドレスを示す定数を指定します。配置する変数のアラインメント条件を満たしている絶対アドレスを指定する必要があります。

注：絶対アドレスに配置される `__no_init` 変数のすべての宣言は、*仮定義*です。仮定義の変数は、コンパイルするモジュールで必要な場合に、コンパイラからの出力にのみ保持されます。こうした変数は、使用されるすべてのモジュールで定義され、同じ方法で定義されている限りは機能します。こうし

た宣言は、変数を使用する全モジュールにインクルードされるすべてのヘッダファイルに配置することをお勧めします。

絶対アドレスに配置される他の変数は、通常の宣言と定義の区別を使用します。こうした変数については、1 つのモジュール（通常はイニシャライザ）でのみ定義を提供する必要があります。他のモジュールは、明示的アドレスの有無に関わらず、extern 宣言を使用すれば変数を参照できます。

例

この例では、__no_init で宣言した変数が絶対アドレスに配置されます。これは、複数のプロセス、アプリケーションなどの間でインタフェースする場合に便利です。

```
__no_init volatile char alpha @ 0xF2000;/* 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。次の例は、初期化されていない const で宣言されたオブジェクトを含みます。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=0x12002
__no_init __far const int beta;          /* OK */

__far const int gamma @ 0x12004 = 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。

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 __far 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 __far const __no_init int x @ 0x100;
```

注 : C++ の静的メンバ変数は、他の静的変数と同様に、絶対アドレスに配置できます。

データと関数のセクションへの配置

データまたは関数をデフォルト以外の指定セクションに配置する場合、以下の方法を使用できます。

- @ 演算子または #pragma location ディレクティブを使用して、個々の変数や関数を指定のセクションに配置できます。指定のセクション は定義済セクションまたはユーザ定義セクションです。
- --code_section オプションは、コンパイルユニット全体の一部である関数を指定 セクションに配置するときに使用できます。

C++ の静的メンバ変数は、他の静的変数と同様に、指定セクションに配置できます。

独自のセクションを使用する場合、定義済セクションに加えて、セクションをリンカ設定ファイルで定義する必要があります。

注 : デフォルトで使用している以外の定義済セクションの変数や関数を、明示的に配置する場合に注意してください。状況によっては有益なオプションですが、配置を間違えると、コンパイル時やリンク時のエラーメッセージからアプリケーションの誤動作までを発生することがあります。状況を慎重に考慮し、宣言および関数や変数の使用に関する要件に、厳密に従ってください。

セクションの位置は、リンカ設定ファイルから制御できます。

セクションの詳細は、「*セクションリファレンス*」を参照してください。

指定セクションへの変数の配置例

以下の例では、データオブジェクトがユーザ定義セクションに配置されます。メモリ属性が指定されている場合、他の変数のように変数は、デフォルトの

メモリに配置されているように扱われます。リンクの際には、セクションが適切なメモリエリアに配置されていることを必ず確認してください。

```
__no_init int alpha @ "MY_NOINIT";  /* OK */

#pragma location="MY_CONSTANTS"
const int beta = 42;                /* OK */

const int gamma @ "MY_CONSTANTS" = 17;/* OK */
int theta @ "MY_ZEROS";             /* OK */
int phi @ "MY_INITED" = 4711;       /* OK */
```

ゼロまたは初期化されたケースのリンカは、変数について正しいタイプの初期化を行います。__no_init 変数をユーザ定義セクションに配置する際、そのセクションに一致するパターンをリンカ設定ファイルの do not initialize ディレクティブに追加する必要があります。初期化された変数の場合は、initialize manually ディレクティブを使用すれば自動初期化を無効にできます。

通常、変数のメモリを選択するために、メモリ属性を使用できます。リンクの際には、セクションが適切なメモリエリアに配置されていることを必ず確認してください。

```
__near __no_init int alpha @ "MY_NEAR_NOINIT";/* near に配置 */
```

指定セクションへの関数の配置例

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_FUNCTIONS"
void h(void);
```

メモリ属性を指定して、関数を特定のメモリに出力し、リンカ設定ファイルのセグメントの配置をそれに合わせて変更します。

```
__near_func void f(void) @ "MY_NEAR_FUNC_FUNCTIONS";
```

コンパイラ最適化の設定

コンパイラは、可能な限り最良のコードを生成するために、アプリケーションで多くの変換を行います。変換の例としては、値をメモリではなくレジスタに格納する、余剰なコードを削除する、計算の順序をより効率的に変更する、数値演算をより安価な処理に置換するなどが挙げられます。

リンカによって実行される最適化もあるため、リンカもコンパイルシステムの構成要素の一部と考える必要があります。たとえば、すべての未使用の関数、変数は削除され、最終的な出力には含まれません。

最適化実行のスコープ

実行される最適化の対象を、アプリケーション全体とするか個々のファイルとするか指定できます。デフォルトでは、プロジェクト全体で同一の最適化タイプが使用されますが、個々のファイルに対して異なる最適化設定を使用することを検討してください。たとえば、非常に高速に実行する必要があるコードは別ファイルに記述して、実行時間が最小になるようにコンパイルし、残りのコードはコードサイズを最小にするようにします。これにより、プログラムが小さくなり、重要部分での十分な高速性も実現できます。

また、個々の関数を最適化の実行から除外することも可能です。`#pragma optimize` ディレクティブでは、最適化レベルを下げることや、別のタイプの最適化の実行を指定できます。プラグマディレクティブについては、339 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コール、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、257 ページの *--mfc* を参照してください。

注 : オブジェクトファイルが 1 つだけ生成されるため、すべてのシンボルはこのオブジェクトファイルの一部となります。

アプリケーション全体を 1 つのコンパイルユニットとしてコンパイルする場合、プロシージャ間の最適化を実行する前に、コンパイラで未使用のパブリック関数と変数を破棄するのも非常に役に立ちます。こうすることで、最適化のスコープが実際に使用される関数と変数に限定されます。詳細については、250 ページの *--discard_unused_publics* を参照してください。

最適化レベル

コンパイラでは、さまざまな最適化のレベルをサポートしています。以下の表は、各レベルで一般的に実行される最適化の一覧です。

最適化レベル	説明
なし（デバッグサポートに最適）	変数は、そのスコープ全体を通して有効です 不要なコードの除去 冗長なラベルの除去 冗長な分岐の除去
低	上記と同じですが、変数は必要時のみ有効となり、スコープ全体を通して有効とは限りません
中	上記のほかに以下があります 生死解析と最適化 コードホイスト ピープホール最適化 レジスタ内容の解析と最適化 共通部分式除去 静的クラスタ
高（バランス）	上記のほかに以下があります クロスジャンプ 命令スケジューリング（速度またはバランスを最適化する際） クロスコール（サイズまたはバランスの最適化時） 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 25: コンパイラ最適化レベル

注 : 一部の最適化は、個別に有効化 / 無効化が可能です。これらの詳細については、213 ページの*変換の微調整*を参照してください。

最適化レベルを高くするとコンパイル時間が長くなることがあり、また、生成されたコードとソースコードの関係がわかりにくくなるため、デバッグも困難になります。たとえば、最適化レベルの低、中、高の場合、変数がスコープ全体を通して有効とは限らないため、変数の格納に使用されたプロセッサレジスタは、最後に使用された状態のままで再使用される可能性があります。このため、C-SPY の［ウォッチ］ウィンドウには、スコープ全体を通した変数の値が表示できないことがあります。コードのデバッグが困難な場合は、最適化レベルを下げてください。

速度とサイズ

最適化レベルが高の場合、コンパイラはサイズの最適化と速度の最適化の間のバランスを取ります。ただし、サイズまたは速度に対して明示的に最適化を微調整することも可能です。それらは、使用するしきい値のみの違いです。速度の場合は、サイズを犠牲にして速度を上げ、サイズの場合は、速度を犠牲にしてサイズを小さくします。ある最適化により別の最適化が実行可能になり、サイズよりも速度を優先して最適化した場合でも、アプリケーションのサイズが小さくなることがあります。

最適化レベル高（速度）を使用する場合、`--no_size_constraints` コンパイラオプションは、コードサイズ拡張に対する通常の制限を緩和して、より積極的な最適化を可能にします。

変換の微調整

最適化レベルごとに、一部の変換を個別に無効にできます。変換を無効にするには、適当なオプション（コマンドラインオプション `--no_inline`、IDE での同等オプションの［**関数インライン化**］など）か、`#pragma optimize` ディレクティブを使用します。以下の変換は、個別に無効化することができます。

- 共通部分式除去
- ループ展開
- 関数インライン化
- コード移動
- 型ベースエイリアス解析
- 静的クラスタ
- クロスコール
- 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、259 ページの *--no_cse* を参照してください。

ループ展開

ループ展開とは、コンパイル時に繰り返し回数を決定できるループのコード本体の複製を意味します。ループ展開によって、複数の繰り返しに分割することにより、ループのオーバヘッドが少なくなります。

この最適化は、ループのオーバヘッドがループ本体合計の大部分を占めるような、小さいループの場合に最も効率的です。

ループ展開は、最適化レベルが［高］の場合に実行可能で、通常は実行時間が短縮されますが、コードサイズは増加します。また、生成されるコードのデバッグも困難になる場合があります。

コンパイラは、ヒューリスティックにより、展開するループを決定します。ループのオーバヘッド減少が明確な、比較的小さいループのみが展開されます。実行する最適化の内容（速度、サイズ、速度とサイズのバランス）に応じて、異なるテクニックが使用されます。

注 : このオプションは、最適化レベルが［なし］、［低］、［中］の場合には動作しません。

プラグマディレクティブについては、346 ページの *unroll* を参照してください。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（263 ページの *--no_unroll* を参照）。

関数インライン化

関数インライン化とは、定義がコンパイル時に判明している関数を、その呼出し元関数の本体に統合し、呼出しによるオーバヘッドを解消することです。通常この最適化では実行時間は短縮されますが、コードサイズが大きくなることがあります。

詳細については、79 ページの*インライン関数*を参照してください。

コード移動

ループ不変式や共通部分式の評価式を移動し、冗長な再評価を回避します。この最適化は、最適化レベルが［中］またはそれ以上の場合に実行可能で、通常はコードサイズと実行時間が短縮されます。ただし、生成されるコードのデバッグは困難になる場合があります。

注 : このオプションは、最適化レベルが中以上の場合にのみ有効です。

コマンドラインオプションの詳細については、258 ページの *--no_code_motion* を参照してください。

型ベースエイリアス解析

複数のポインタが同一メモリ位置を参照する場合、これらのポインタをそれぞれの*エイリアス*といいます。エイリアスが存在すると、特定の値が変更されるかどうかがコンパイル時にわからない場合があるため、最適化が困難になります。

型ベースエイリアス解析による最適化では、同一オブジェクトへのすべてのアクセスは、そのオブジェクトの宣言型または `char` 型の使用を前提としています。これにより、コンパイラはポインタが同一のメモリ位置を参照しているかどうかを検出することができます。

型ベースエイリアス解析は、最適化レベルが［高］の場合のみ実行されます。標準の C/C++ アプリケーションコードに準拠するアプリケーションコードの場合、この最適化によってコードサイズが減少して実行時間が短縮されることがあります。ただし、非標準の C/C++ コードの場合は、予期せぬ動作の原因となるコードをコンパイラが生成することがあります。そのため、この最適化を無効にできるようになっています。

注 : このオプションは、最適化レベルが［なし］、［低］、［中］の場合には動作しません。

コマンドラインオプションの詳細については、262 ページの *--no_tbaa* を参照してください。

例

```
short F(short *p1, long *p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

型ベースエイリアス解析では、`p1` にポイントされる `short` へのライトアクセスは、`p2` がポイントする `long` の値に影響しないと見なされます。そのため、この関数が 0 を返すことをコンパイル時に判定できます。しかし、規格に準拠していない C/C++ コードでは、これらのポインタが同一の共用体に含まれ、相互に重複することがあります。明示的なキャストを使用する場合、異なるポインタ型のポインタが同一メモリ位置を参照するように強制することもできます。

静的クラスタ

静的クラスタが有効にされている場合、同じモジュール内で定義される静的およびグローバル変数は、同じ関数でアクセスされる変数がそれぞれ近くに

格納されるように配置されます。これにより、コンパイラは、いくつかのアクセスに対して同じベースポインタを使用できるようになります。

注 : このオプションは、最適化レベルが［なし]、[低］の場合には動作しません。

コマンドラインオプションの詳細については、258 ページの *--no_clustering* を参照してください。

クロスコール

共通コードシーケンスが、ローカルのサブルーチンに抽出されます。この最適化は、最適化レベル［高］で行われ、コードサイズを小さくします。ときには実行時間とスタックサイズの面で、劇的にコードサイズが小さくなることもあります。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。この最適化は、`#pragma optimize` ディレクティブを使用して無効にすることはできません。

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。

関連のコマンドラインオプションの詳細については、259 ページの *--no_cross_call* 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、261 ページの *--no_scheduling* を参照してください。

円滑なコードの生成

ここでは、コンパイラで良いコードを生成するためのヒントについて説明します。たとえば、次のようなものがあります。

- 効率的なアドレッシングモードの使用

- コンパイラの最適化の促進
- より有意義なエラーメッセージの生成

最適化を容易にするソースコードの記述

以下に、コンパイラでのアプリケーション最適化を改善できるプログラミングテクニックを示します。

- 静的 / グローバル変数よりもローカル変数（自動変数とパラメータ）を使用することをお勧めします。これは、呼出し先関数がローカル以外の変数を変更する可能性などをオプティマイザが想定する必要があるためです。
  ローカル変数の使用期間が終了すると、占有されていたメモリを再利用できます。グローバルに宣言した変数は、プログラムの実行中はデータメモリを占有します。
- & 演算子を使用してローカル変数のアドレスを取ることは避けてください。これは、主に 2 つの理由で非効率です。まず、変数はメモリに配置する必要があり、プロセッサのレジスタに配置できません。そのため、コードのサイズが大きくなり、速度が低下します。次に、ローカル変数が関数呼出しの影響を受けないとオプティマイザが想定できなくなります。
- グローバル変数（非静的）よりも、モジュール内でローカルな変数（static として宣言された変数）を使用することをお勧めします。また、頻繁にアクセスされる静的変数のアドレスを取ることは避けてください。
- コンパイラによる関数のインライン化が可能です（214 ページの*関数インライン化*を参照）。インライン化の効果を最大限にするには、複数のモジュールから呼出される小さい関数の定義を、実装ファイルではなくヘッダファイルに配置することをお薦めします。または、複数ファイルコンパイルを使用します。詳細については、211 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を回避その代わりに、C/C++ にコードを書いて、組込み関数を使用する、またはアセンブラ言語で別のモジュールに書きます。詳細については、143 ページの *C 言語とアセンブラの結合*を参照してください。

スタックエリアと RAM メモリの節約

以下に、メモリやスタックエリアを節約できるプログラミングテクニックを示します。

- スタックエリアが少ない場合は、長い呼出しチェーンや再帰関数の使用は避けてください。
- 大きなサイズの非スカラ型（構造体など）をパラメータやリターン型として使用することは避けてください。スタックエリアを節約するため、代わりにポインタか、C++ の場合は参照として引き渡してください。

関数プロトタイプ

2 つのスタイルのいずれかを使用して、関数の宣言と定義を行うことができます。

- プロトタイプ
- カーニハン & リッチー C (K&R C)

どちらのスタイルも有効な C ですが、できる限りプロトタイプスタイルを使用して、関数を定義するコンパイルユニットおよびそれを使用するすべてのコンパイルユニットの両方に含まれるヘッダの public 関数ごとに、プロトタイプ宣言を指定するようお勧めします。

コンパイラは、K&R スタイルを使用して宣言された関数に引き渡されるパラメータに対してはチェックを実行しません。プロトタイプ宣言を使用すると、コードがより効率的になることがあります。これは、これらの関数に型変換が必要ないためです。

すべての関数定義が適切なプロトタイプスタイルを使用し、すべての public 関数が定義される前に宣言されていることをコンパイラで義務づけるには、［**プロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**言語 1**］**>**［**プロトタイプの強制**］コンパイラオプション (`--require_prototypes`) を使用します。

プロトタイプスタイル

プロトタイプ関数の宣言では、各パラメータの型を指定する必要があります。

```
int Test(char, int); /* 宣言 */

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

カーニハン & リッチースタイル

カーニハン & リッチースタイル（標準 C 以前）では、プロトタイプ化された関数を宣言することはできません。その代わりに、空のパラメータリストを関数宣言で使用します。また、定義の記述が異なります。

次に例を示します。

```
int Test();     /* 宣言 */

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

整数型とビット否定

場合によっては、整数型とその変換の規則が、混乱を招く動作の原因となることがあります。異なるサイズの型や論理演算（特にビット否定）が関係する代入文や条件文（評価式）に注意する必要があります。この場合、*types* 型には定数型も含まれます。

ワーニング（定数の条件文や無意味な比較など）が発生する場合と、予期した結果と異なるだけの場合があります。コンパイラが定数の条件文のインスタンスを特定するために最適化を使用する場合などは、より高水準の最適化でのみ、コンパイラがワーニングを生成することがあります。以下の例では、8 ビット文字、16 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 `0x0080` であり、`~0x0080` は `0xFF7F` 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 8 ビットが設定されることもありません。

同時にアクセスされる変数の保護

非同期でアクセスされる変数（割込みルーチンからアクセスされる変数、独立したスレッドで実行しているコードからアクセスされる変数など）は、適切にマークして保護する必要があります。例外は、常に*リードオンリー*の変数のみです。

変数を適切にマークするには、`volatile` キーワードを使用します。このキーワードは、変数が他のスレッドから変更される可能性があることをコンパイラに示します。すると、コンパイラでは、変数の最適化を回避（レジスタ内の変数を追跡するなど）し、変数へのライトを遅延させず、ソースコードで指定された回数だけ変数にアクセスするように注意します。

割り込まれたくない変数へのアクセスのシーケンスについては、__monitor キーワードを使用してください。これは、ライトとリードシーケンスの両方に対して行わねばなりません。そうしなければ、部分的に更新された変数を読み取ることになりかねません。サイズの小さい volatile 変数へのアクセスはアトミック処理とすることもできますが、継続的にコンパイラの出力を調べるのでない限りは、これに依存すべきではありません。シーケンスが確実にアトミック処理であるようにするには、__monitor キーワードを使用する方が安全です。詳細については、317 ページの *__monitor* を参照してください。

volatile 型修飾子および volatile オプションのアクセス規則について詳しくは、305 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、RL78 デバイス用の専用ヘッダファイルがいくつか付属しています。ヘッダファイルは、io*device*.h という形式で命名され、プロセッサ固有の特殊機能レジスタ (SFR) を定義します。

注 : 各ヘッダファイルには、コンパイラが使用するセクションが 1 つ、アセンブラが使用するセクションが 1 つ含まれています。

ビットフィールド付きの SFR が、ヘッダファイルで定義されています。次の例は、iorl78.h からのものです。

```
typedef struct
{
  unsigned char no0:1;
  unsigned char no1:1;
  unsigned char no2:1;
  unsigned char no3:1;
  unsigned char no4:1;
  unsigned char no5:1;
  unsigned char no6:1;
  unsigned char no7:1;
} __BITS8;

__sfr __no_init volatile union
{
  unsigned char PM0;
  __BITS8 PM0_bit;
} @ 0xFFF20;
```

```
/* 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 */
  PM0 = 0x12;

  /* ビットフィールドアクセス */
  PM0_bit.no1 = 1;
  PM0_bit.no5 = 0;
}
```

他の RL78 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。@ 演算子の詳細については、206 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。

非初期化変数

通常は、アプリケーション起動時に、ランタイム環境がすべてのグローバル変数と静的変数を初期化します。

コンパイラは、`__no_init` 型修飾子により、初期化されない変数の宣言をサポートしています。このような変数は、キーワードとして指定することや、`#pragma object_attribute` ディレクティブを使用して指定することができます。コンパイラは、指定したメモリキーワード基づいて、こうした変数を別のセグメントにに配置します。

`__no_init` を使用した場合、`const` キーワードは、リードオンリーメモリにオブジェクトが格納されるのではなく、オブジェクトがリードオンリーであることを意味します。`__no_init` オブジェクトに初期値を指定することはできません。

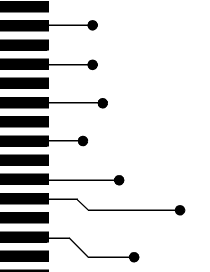
`__no_init` キーワードを使用して宣言した変数は、大きな入力バッファとして使用することや、アプリケーション終了後も内容を保持する特殊な RAM にマッピングすることができます。

詳細については、319 ページの *__no_init* を参照してください。このキーワードを使用するには、言語拡張を有効にする必要があります(251 ページの *-e* を参照)。詳細については、338 ページの *object_attribute* を参照してください。

パート 2. リファレンス情報

*『RL78 用 IAR C/C++ コンパイラユーザガイド』*のこのパートは、以下の章で構成されています。

- 外部インタフェースの詳細
- コンパイラオプション
- リンカオプション
- データ表現
- 拡張キーワード
- プラグマディレクティブ
- 組込み関数
- プリプロセッサ
- ライブラリ関数
- リンカ設定ファイル
- セクションリファレンス
- IAR ユーティリティ
- C 規格の処理系定義の動作
- C89 の処理系定義の動作



外部インタフェースの詳細

- 呼出し構文
- インクルードファイル検索手順
- コンパイラ出力
- ILINK 出力
- 診断

呼出し構文

コンパイラとリンカは、IDE またはコマンドラインインタフェースから使用できます。IDE からのビルドツールの使用については、*『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。

```
iccrl78 [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。

```
iccrl78 prog.c --debug
```

ソースファイルには、C/C++ ファイルを使用でき、それぞれ c または cpp のファイル名拡張子を指定します。ファイル名拡張子を指定しない場合、コンパイルするファイルの拡張子は c でなければばなりません。

通常、コマンドラインでのオプションの順序とソースファイル名の前後のどちらに入力するかは、重要ではありません。ただし、例外が 1 つあります。-I オプションを使用する場合には、ディレクトリの検索はコマンドラインに指定した順序で行われます。

コマンドラインから引数なしでコンパイラを実行する場合、コンパイラのバージョン番号と利用可能なすべてのオプション（簡単な説明を含む）が stdout に転送され、画面に表示されます。

ILINK 呼出し構文

ILINK の呼出し構文は次のとおりです。

```
ilinkrl78 [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。

```
ilinkrl78 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、iccrl78 または ilinkrl78 コマンドの後にオプションを指定します（225 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（227 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（254 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 7.`*n*`¥rl78¥inc;c:¥headers`
`QCCRL78`	コマンドラインのオプションを指定します。`QCCRL78=-lA asm.lst`

表 26: 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
`ILINKRL78_CMD_LINE`	コマンドラインのオプションを指定します。`ILINKRL78_CMD_LINE=--config full.icf --silent`

表 27: 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（255 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）（227 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。250 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。

```
iccrl78 ..¥src¥src.c -I..¥include -I..¥debugconfig
```

すると、以下のディレクトリ（記載順）で config.h ファイルが検索されます。この例では、このファイルは dir¥debugconfig ディレクトリにあります。

dir¥include	現在のファイルは src.h です。
dir¥src	現在のファイルが含まれるファイル (src.c)。
dir¥include	最初の -I オプションで指定した通りになります。
dir¥debugconfig	2 番目の -I オプションで指定した通りになります。

stdio.h などの標準ヘッダファイルは角括弧、アプリケーション用のヘッダファイルは二重引用符で囲んでください。

注 : ¥ および / は両方ともディレクトリの区切り文字として使用できます。

ヘッダファイルをインクルードする構文については、353 ページの*プリプロセッサの概要*を参照してください。

コンパイラ出力

コンパイラでは、以下の出力を生成できます。

- リンク可能オブジェクトファイル

  コンパイラにより生成されるオブジェクトファイルは、業界標準フォーマットの ELF を使用します。デフォルトでは、オブジェクトファイルは o のファイル名拡張子を持ちます。

- リストファイル（オプション）

  コンパイラオプション `-l` を使用して、さまざまな種類のリストファイルを指定できます（256 ページの *-l* を参照）。デフォルトでは、これらのファイルのファイル名拡張子は `lst` です。

- プロセッサ出力ファイル（オプション）

  プロセッサ出力ファイルを作成するには、`--preprocess` オプションを使用します。デフォルトでは、このファイルのファイル名拡張子は `i` です。

- 診断メッセージ

  診断メッセージは、標準エラーストリームに転送されて画面上に表示されるほか、オプションのリストファイルにも出力されます。診断メッセージの情報については、「231 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（229 ページの*エラーリターンコード*を参照）。

- サイズ情報

  関数およびメモリごとのデータに対して生成されたバイト数に関する情報が標準出力ストリームに転送され、画面上に表示されます。それらのバイトの一部が*「共有」*として報告されることもあります。

  共有オブジェクトとは、モジュール間で共有される関数またはデータオブジェクトのことです。このような共有が 2 つ以上のモジュールで発生した場合、1 つの関数 / データオブジェクトのみが保持されます。たとえば、インライン関数がインライン化されない場合があります。これは、これらの関数が共有とマークされていて、各関数の 1 つのインスタンスしか最終的なアプリケーションにインクルードされないためです。この仕組みは、特定の関数や変数に直接的には関連しないコンパイラ生成コードやデータで、最終的なアプリケーションには 1 つのインスタンスしか必要とされない場合にも使用されることがあります。

エラーリターンコード

コンパイラおよびリンカは、バッチファイル内で評価可能なステータス情報をオペレーティングシステムに返します。

以下のコマンドラインエラーコードがサポートされています。

コード	説明
0	コンパイルまたはリンク処理は成功していますが、ワーニングが発生した可能性があります。

表 28: エラーリターンコード

コード	説明
1	オプション --warnings_affect_exit_code の使用中にワーニングが発生しました。
2	エラーが発生しました。
3	致命的なエラーが発生したためツールが停止しました。
4	インターナルなエラーが発生したためツールが停止しました。

表 28: エラーリターンコード（続き）

ILINK 出力

ILINK では、以下の出力を生成できます。

- 絶対実行可能イメージ

  IAR ILINK リンカは、最終出力として、実行可能イメージを含む絶対オブジェクトファイルを生成します。このファイルは、EPROM に格納したり、ハードウェアエミュレータにダウンロードしたり、IAR C-SPY デバッガシミュレータを使用して PC 上で実行できます。デフォルトでは、ファイルは `out` のファイル名拡張子を持ちます。出力フォーマットは、常に ELF です。これは、オプションで DWARF フォーマットのデバッグ情報を含みます。

- オプションのログイン情報

  操作中、ILINK は、その決定を `stdout`、およびオプションでファイルに記録します。たとえば、ライブラリが検索される場合、必要なシンボルがライブラリモジュールで見つかったかどうか、またはモジュールが出力の一部になるかどうかが記録されます。各 ILINK サブシステムのタイミング情報も記録されます。

- オプションのマップファイル

  リンカマップファイル（リンク、ランタイム属性、メモリ、配置のサマリ、エントリリストを含む）は、ILINK オプション `--map` を使用して生成できます（285 ページの *--map* を参照）。デフォルトでは、マップファイルのファイル名拡張子は `map` です。

- 診断メッセージ

  診断メッセージは、`stderr` に転送され、画面上に表示されるほか、オプションのマップファイルにも出力されます。診断メッセージの情報については、「231 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（229 ページの*エラーリターンコード*を参照）。

- 使用メモリのサイズ情報および経過時間

  関数およびメモリごとのデータに対して生成されたバイト数に関する情報が stdout に転送され、画面上に表示されます。

診断

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

コンパイラのメッセージフォーマット

診断メッセージはすべて、説明を要しない完結型のメッセージとして出力されます。コンパイラの診断メッセージは、次のフォーマットで生成されます。

filename,*linenumber level*[*tag*]: *message*

各エレメントの意味は以下のとおりです。

filename	問題が発生したソースファイルの名前
linenumber	コンパイラが問題を検出した行の番号
level	問題の重要度のレベル
tag	診断メッセージを示す固有のタグ
message	説明（場合によっては複数行）

診断メッセージは、オプションのリストファイルに出力されるとともに、画面に表示されます。

オプション --diagnostics_tables を使用すると、すべてのコンパイラ診断メッセージが一覧表示されます。

リンカのメッセージフォーマット

診断メッセージはすべて、説明を要しない完結型のメッセージとして出力されます。ILINK が生成する診断メッセージは、通常次のような形式です。

level[*tag*]: *message*

各エレメントの意味は以下のとおりです。

level	問題の重要度のレベル
tag	診断メッセージを示す固有のタグ
message	説明（場合によっては複数行）

診断メッセージは、オプションのマップファイルに出力されるとともに、画面に表示されます。

オプション `--diagnostics_tables` を使用すると、すべてのリンカ診断メッセージが一覧表示されます。

重要度

診断メッセージは、以下の重要度に分類されます。

リマーク

生成したコードで誤った動作を引き起こす可能性がある構造をコンパイラまたはリンカが検出した場合に生成される診断メッセージ。リマークはデフォルトでは出力されません。有効にする方法については、268 ページの *--remarks* を参照してください。

ワーニング

コンパイラまたはリンカがコードの生成に支障のあるような潜在的な問題プログラミングのエラーや漏れはあるが、コンパイルやリンクの途中終了の原因にはならないものを示します。ワーニングは、`--no_warnings` コマンドラインオプションを使用して無効にできます（263 ページの *--no_warnings* を参照）。

エラー

コンパイラまたはリンカで重大なエラーが見つかったときに生成される診断メッセージ。エラーが発生した場合は、ゼロ以外の終了コードが生成されます。

致命的なエラー

コードが生成できなくなるだけでなく、それ以降の処理が無意味となるような状態をコンパイラが検出した場合に生成される診断メッセージ。このメッセージが出力された後、コンパイルが終了します。致命的なエラーが発生した場合は、ゼロ以外の終了コードが生成されます。

重要度の設定

致命的なエラーや一部の通常エラーを除くすべての診断メッセージに対し、診断メッセージの出力抑制や重要度の変更ができます。

重要度の設定に使用可能なコンパイラオプションについては、*コンパイラオプション*を参照してください。

コンパイラについては、重要度レベルの設定に使用可能なプラグマディレクティブについては、*プラグマディレクティブ*を参照してください。

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。

```
Internal error: 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

- オプションの構文
- コンパイラオプションの概要
- コンパイラオプションの説明

オプションの構文

コンパイラオプションとは、コンパイラのデフォルトの動作を変更するためのパラメータです。オプションの指定は、コマンドラインまたは IDE 内から行えます。ここでは、コマンドラインからの指定について詳細に説明します。

IDE で使用可能なコンパイラオプションとそれらの設定方法については、オンラインヘルプシステムを参照してください。

オプションのタイプ

コマンドラインオプションには、*省略形*の名前と*完全形*の名前の 2 *種類*があります。一部のオプションは両方の名前を持ちます。

- オプションの省略形は 1 文字で構成され、パラメータが付くこともあります。このフォーマットで指定する場合は、`-e` のようにダッシュを付けて入力します。
- オプションの完全形は、複数の語をアンダースコアで連結した形で構成され、パラメータが付くこともあります。このフォーマットで指定する場合は、`--char_is_signed` のようにダッシュを 2 個付けて入力します。

さまざまなオプションの渡し方については、226 ページの*オプションの受渡し*を参照してください。

パラメータの指定に関する規則

オプションパラメータの指定に関する一般的な構文規則があります。最初に、パラメータが*任意指定*か*必須*かどうか、オプション名が省略形か完全形かどうかに分けて規則を説明します。次に、ファイル名およびディレクトリを指定するための規則を一覧で示します。最後に、残りの規則を一覧で示します。

任意指定パラメータの規則

省略形のオプションで任意指定パラメータを伴なう場合は、以下のように、パラメータの前にスペースを空けずに指定します。

`-O` または `-Oh`

完全形のオプションで任意指定パラメータを伴なう場合は、以下のように、パラメータの前に等号 (=) を付けて指定します。

--misrac2004=*n*

必須パラメータの規則

省略形のオプションで必須パラメータを伴なう場合は、以下のように、パラメータの前にスペースを空けても空けなくてもかまいません。

`-I..¥src` または `-I ..¥src¥`

完全形のオプションで必須パラメータを伴なう場合は、以下のように、パラメータの前に等号 (=) を付けるかスペースを空けて指定します。

```
--diagnostics_tables=MyDiagnostics.lst
```

または

```
--diagnostics_tables MyDiagnostics.lst
```

任意指定パラメータと必須パラメータの両方を伴なうオプションの規則

任意指定パラメータと必須パラメータの両方をとるオプションでのパラメータ指定の規則は以下のとおりです。

- 省略形のオプションでは、任意指定パラメータの前にスペースを空けずに指定します。
- 完全形のオプションでは、任意指定パラメータの前に等号 (=) を付けて指定します。
- 省略形および完全形のオプションでは、必須パラメータの前にスペースを空けて指定します。

省略形のオプションで、任意指定パラメータの後に必須パラメータを指定する例を以下に示します。

```
-lA MyList.lst
```

完全形のオプションで、任意指定パラメータの後に必須パラメータを指定する例を以下に示します。

```
--preprocess=n PreprocOutput.lst
```

ファイル名またはディレクトリをパラメータとして指定する場合の規則

ファイル名またはディレクトリをパラメータとして指定するオプションの規則は以下のとおりです。

- ファイル名をパラメータとして指定するオプションは、ファイルパスの指定も可能です。パスは、相対パスと絶対パスのいずれでもかまいません。たとえば、..¥listings¥ ディレクトリのファイル List.lst のリストを生成するには、以下のように入力します。

```
iccrl78 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。

```
iccrl78 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。

```
iccrl78 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。

```
iccrl78 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。

```
iccrl78 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
--c89	C89 の派生言語を指定します
--calling_convention	呼出し規約を指定します
--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
--code_model	コードモデルを指定
--code_section	関数のデフォルトのセクションを変更
--core	CPU コアを指定
-D	プリプロセッサシンボルを定義
--data_model	データモデルを指定
--debug	デバッガ情報を生成
--dependencies	ファイル依存関係をリスト化
--diag_error	エラーとして処理
--diag_remark	リマークとして処理
--diag_suppress	診断を無効化
--diag_warning	ワーニングとして処理
--diagnostics_tables	すべての診断メッセージをリスト化
--disable_div_mod_instructions	RL78 S3 コアで、DIVH/DIVHU 命令の生成を無効にします
--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
--double	コンパイラで 32 ビットまたは 64 ビットの double を使用するよう強制
-e	言語拡張を有効化
--ec++	Embedded C++ を指定

表 29: コンパイラオプションの一覧

コマンドラインオプション	説明
--eec++	Extended Embedded C++ を指定
--enable_multibytes	ソースファイルのマルチバイト文字のサポートを有効化
--enable_restrict	標準の C のキーワード制限を有効にします
--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
--generate_callt_runtime_library_calls	`__callt` ランタイムライブラリ呼出しを生成
--generate_far_runtime_library_calls	`__far` ランタイムライブラリ呼出しを生成
--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
-l	リストファイルを生成
--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
--misrac1998	MISRA-C:1998 固有のエラーメッセージを有効にします。*『IAR Embedded Workbench® MISRA C:1998 リファレンスガイド』*を参照してください
--misrac2004	MISRA-C:2004 固有のエラーメッセージを有効にします。*『IAR Embedded Workbench® MISRA C:2004 リファレンスガイド』*を参照してください
--misrac_verbose	*『IAR Embedded Workbench® MISRA C:1998 リファレンスガイド』*または*『IAR Embedded Workbench® MISRA C:2004 リファレンスガイド』*を参照してください
--near_const_location	ROM から書込み可能にするか、ミラーにする near 定数を指定
--no_clustering	静的クラスタ最適化を無効にします

表 29: コンパイラオプションの一覧（続き）

コマンドラインオプション	説明
--no_code_motion	コード移動最適化を無効化
--no_cross_call	クロスコールの最適化を無効化
--no_cse	共通部分式除去を無効化
--no_dwarf3_cfi	DWARF 3 の呼出しフレーム命令の生成を無効にします
--no_fragments	セクションフラグメント処理を無効化
--no_inline	関数インライン化を無効化
--no_path_in_file_macros	シンボル __FILE__ および __BASE_FILE__ のリターン値からパスを削除
--no_scheduling	命令スケジューラを無効にします。
--no_size_constraints	速度を優先して最適化するときに、コードサイズ拡張の通常の制限を緩和します
--no_static_destruction	プログラム終了時に C++ 静的変数の破壊を無効化します
--no_system_include	システムインクルードファイルの自動検索を無効化します
--no_tbaa	型ベースエイリアス解析を無効化
--no_typedefs_in_diagnostics	診断での typedef 名の使用を無効化
--no_unroll	ループ展開を無効化
--no_warnings	すべての警告を無効化
--no_wrap_diagnostics	診断メッセージのラッピングを無効化
-O	最適化レベルを設定
-o	オブジェクトファイル名を設定。--output のエイリアス
--only_stdout	標準出力のみを使用
--output	オブジェクトファイル名を設定
--pending_instantiations	任意の C++ テンプレートのインスタンス化の最大数を設定します
--predef_macros	定義済シンボルの一覧を表示
--preinclude	ソースファイルを読み込む前にインクルードファイルをインクルード
--preprocess	プリプロセッサ出力を生成
--public_equ	グローバル名のアセンブララベルを定義

表 29: 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
`-r`	デバッガ情報を生成。`--debug` のエイリアス
`--relaxed_fp`	浮動小数点式の最適化規則を緩和します
`--remarks`	リマークを有効化
`--require_prototypes`	関数が定義前に宣言されていることを検証
`--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
`--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
`--warn_about_c_style_casts`	C- スタイルキャストが C++ ソースコードで使用されるときにコンパイラを警告
`--warnings_affect_exit_code`	ワーニングが終了コードに影響
`--warnings_are_errors`	ワーニングをエラーとして処理
`--workseg_area`	ショートアドレス作業セクション領域を任意のサイズで有効化

表 29: コンパイラオプションの一覧（続き）

コンパイラオプションの説明

次のセクションでは、それぞれのコンパイラオプションについて詳細に説明します。

［追加オプション］ページを使用して特定のコマンドラインオプションを指定する場合、IDE では、オプションの競合、オプションの重複、不適切なオプションの使用などの整合性問題のインスタントチェックは実行しません。

--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

167 ページの *C 言語の概要*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［C の派生言語］>［C89］

--calling_convention

構文

`--calling_convention=convention`

パラメータ

convention は以下のいずれかです。

v1	V1 呼出し規約を選択します。
v2（デフォルト）	V2 呼出し規約を選択します。

説明

このオプションを使用して、モジュールのデフォルトの呼出し規約を指定します。アプリケーション内のすべてのランタイムモジュールで、同じ呼出し規約を使用する必要があります。ただし、キーワードを使用すれば、個々の関数についてこの設定をオーバライドすることができます。

注 : バージョン 1.x のコンパイラから移行する場合、スタックは使用する呼出し規約に関わらず、呼び出された関数ではなく、呼び出した関数によりクリーンされる点に注意してください。スタックのパラメータを受け入れるアセンブラ関数は、アップデートしてから使用する必要があります。

関連項目

150 ページの *呼出し規約*。

［プロジェクト］>［オプション］>［一般オプション］>［ターゲット］>［呼出し規約］

--char_is_signed

構文

`--char_is_signed`

説明

デフォルトでは、コンパイラは単純な `char` 型を符号なしとして解釈します。このオプションは、コンパイラで単純な `char` 型を符号付きと解釈する場合に使用します。これは、他のコンパイラとの互換性を確保する場合などに便利です。

注 : ランタイムライブラリは --char_is_signed オプションを使用せずにコンパイルされ、このオプションによってコンパイルされたコードとは一緒に使用できません。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **2**］>［**'CHAR'** の型］

--char_is_unsigned

構文 `--char_is_unsigned`

説明 このオプションは、コンパイラで単純な char 型を符号なしと解釈する場合に使用します。これは、単純な char 型のデフォルトの解釈です。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **2**］>［**'CHAR'** の型］

--code_model

構文 `--code_model={near|n|far|f}`

パラメータ

near（デフォルト）	関数の呼出しがメモリの最初の 64KB に達します。
Far	関数の呼出しが 1MB のメモリ全体に達します。

説明 このオプションでは、コードモデルを選択します。コードモデルを選択しない場合、コンパイラはデフォルトのコードモデルを使用します。アプリケーションの全モジュールで同じコードモデルを使用する必要があることに注意してください。

関連項目 73 ページの*関数格納のためのコードモデルとメモリ属性*。

［プロジェクト］>［オプション］>［一般オプション］>［ターゲット］>［コードモード］

--code_section

構文 `--code_section=section_name`

説明 デフォルトでは、コンパイラはコードモデルによって決められたコードセクションに 関数を配置します。*section_name* という名前のセクションにすべ

ての関数をデフォルトで配置するには、--code_section オプションを使用します。そうすれば、このセクションをリンカ設定ファイルの固定アドレスに割り当てることができます。(セクション名は大文字と小文字を区別します)。

異なるアドレス範囲にコードを配置し、@ 表記または #pragma location ディレクティブでは不十分な場合に、このオプションが有益です。セクション名の変更時には、対応するリンカ設定ファイルも変更する必要があることに、注意してください。

注 : このオプションは、デフォルトのメモリに明示的または暗示的に配置された関数のみに適用されます。関数をデフォルト以外のセクションに明示的に配置する関数メモリ属性を使用する場合、その関数は --code_section オプションの影響を受けません。

関連項目

206 ページの*データと関数のメモリ配置制御*。

[プロジェクト] > [オプション] > [**C/C++** コンパイラ] > [出力] > [コードセクション]

--core

構文

```
--core={s1|s2|s3}
```

パラメータ

s1	レスタバンク 1 つのみと多重化された 8 ビットバスを持つ RL78 1 コアである S1 用にコードを生成します。
s2	ハードウェア積算 / 除算をサポートする命令を持たないコアである S2 用にコードを生成します。
s3(デフォルト)	ハードウェア積算 / 除算をサポートする命令を持つコアである S3 用にコードを生成します。

説明

このオプションを使用して、コードを生成するプロセッサコアを選択します。このオプションを使用してコアを指定しない場合、コンパイラは S3 コアのコードをデフォルトとして生成します。アプリケーションの全モジュールで同じコアを使用する必要がある点に注意してください。

コンパイラは、これらのコアに基づいて、RL78 マイクロコントローラコアと派生品をサポートします。

[プロジェクト] > [オプション] > [一般オプション] > [ターゲット] > [コア]

-D

構文

`-D symbol[=value]`

パラメータ

symbol	プリプロセッサシンボルの名前
value	プリプロセッサシンボルの値

説明

このオプションは、プリプロセッサシンボルの定義に使用します。値を指定しない場合、1 が使用されます。このオプションは、コマンドラインで任意の回数使用できます。

-D オプションは、ソースファイルの最初に #define 文を記述した場合と同様に機能します。

```
-Dsymbol
```

は、以下の文と等価です。

```
#define symbol 1
```

以下の文と等価なコマンドを考えてみます。

```
#define FOO
```

この文と同じ結果を得るには、以下のように、= 記号を付け、その後には何も付けずに指定します。

```
-DFOO=
```

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［プリプロセッサ］>［シンボル定義］

--data_model

構文

`--data_model={near|n|far|f}`

パラメータ

near（デフォルト）	データはデフォルトでメモリの最上位 64KB に配置されます。
Far	データはデフォルトでメモリの 1MB 全体に配置されます。

説明

このオプションを使用してデータモデルを選択します。つまり、データオブジェクトのデフォルトの配置です。データモデルを選択しない場合、コンパ

イラはデフォルトのデータモデルを使用します。アプリケーションの全モジュールで同じデータモデルを使用する必要があることに注意してください。

関連項目

68 ページの*データモデル*。

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

--debug, -r

構文

```
--debug
-r
```

説明

`--debug` や `-r` オプションは、IAR C-SPY® デバッガまたは他のシンボリックデバッガで必要なコンパイラのインクルード情報をオブジェクトモジュールに含める場合に使用します。

注 : デバッグ情報を含めると、オブジェクトファイルのサイズが増加します。

[プロジェクト] > [オプション] > [**C/C++** コンパイラ] > [出力] > [デバッグ情報の生成]

--dependencies

構文

```
--dependencies[=[i|m|n][s]] {filename|directory|+}
```

パラメータ

`i`（デフォルト）	ファイルの名前のみをリスト化
`m`	makefile スタイルでリスト化（複数のルール）
`n`	makefile スタイルでリスト化（1 つのルール）
`s`	システムファイルの無効化
`+`	`-o` と同じ内容を出力しますが、ファイル名拡張子が `d` となります

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

説明

このオプションは、ファイルへの入力のために開かれたすべてのソースファイルおよびヘッダファイルの一覧を、デフォルトのファイル名拡張子 `i` を持つファイルに含める場合に使用します。

例

--dependencies や --dependencies=i を使用すると、開かれている各入力ファイルの名前とフルパス（ある場合）が独立した行に出力されます。次に例を示します。

```
c:¥iar¥product¥include¥stdio.h
d:¥myproject¥include¥foo.h
```

--dependencies=m を使用した場合は、makefile スタイルで出力されます。各入力ファイルについて、makefile の依存関係規則を含む行が生成されます。各行は、オブジェクトファイル名、コロン、空白文字、入力ファイル名で構成されます。次に例を示します。

```
foo.o: c:¥iar¥product¥include¥stdio.h
foo.o: d:¥myproject¥include¥foo.h
```

たとえば、gmake (GNU make) などの一般的な make ユーティリティで --dependencies を使用するには、以下のように操作します。

1 以下のように、ファイルのコンパイル規則を設定します。

```
%.o : %.c
	$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

すなわち、このコマンドを使用すると、オブジェクトファイルの生成に加えて、makefile スタイルで依存関係ファイルが生成されます（この例では、拡張子に .d を使用）。

2 以下のようにして、すべての依存関係ファイルを makefile に含めます。

```
-include $(sources:.c=.d)
```

ダッシュ (-) があるため、.d ファイルがまだ存在しない最初の時点でも機能します。

このオプションは、IDE では使用できません。

--diag_error

構文

--diag_error=*tag*[,*tag*,...]

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージをエラーとして再分類する場合に使用します。エラーは、オブジェクトコードが生成されなくなるような重大

な C/C++ 言語規則の違反を示します。終了コードはゼロ以外になります。このオプションは、1 つのコマンドラインで複数個使用できます。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［診断］>［エラーとして処理］

--diag_remark

構文

`--diag_remark=`*`tag`*`[,`*`tag,...`*`]`

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe177` など）

説明

このオプションは、特定の診断メッセージをリマークとして再分類する場合に使用します。リマークは、最も軽微な診断メッセージです。生成されたコードに異常動作の原因となる可能性があるソースコード構造が存在することを示します。このオプションは、1 つのコマンドラインで複数個使用できます。

注 : デフォルトでは、リマークは表示されません。リマークを表示するには、`--remarks` オプションを使用します。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［診断］>［リマークとして処理］

--diag_suppress

構文

`--diag_suppress=`*`tag`*`[,`*`tag,...`*`]`

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 `Pe117` など）

説明

このオプションは、特定の診断メッセージを無効にする場合に使用します。これらのメッセージは表示されなくなります。このオプションは、1 つのコマンドラインで複数個使用できます。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［診断］>［診断を無効化］

--diag_warning

構文

```
--diag_warning=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe826 など）

説明

このオプションは、特定の診断メッセージをワーニングとして再分類する場合に使用します。ワーニングは、問題はあるが、コンパイルの途中終了の原因にはならないエラーや脱落を示します。このオプションは、1 つのコマンドラインで複数個使用できます。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［ワーニングとして処理］

--diagnostics_tables

構文

```
--diagnostics_tables {filename|directory}
```

パラメータ

237 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。

--disable_div_mod_instructions

構文

```
--disable_div_mod_instructions
```

説明

このオプションを使用して、S3 コアの DIVH/DIVHU 命令の生成を無効化します。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［DIV/MOD 命令を無効化］。

--discard_unused_publics

構文

`--discard_unused_publics`

説明

このオプションは、`--mfc` コンパイラオプションでコンパイルする際に未使用のパブリック関数と変数を破棄するときに使用します。

注 : このオプションをアプリケーションの一部のみで使用しないでください。生成された出力から必要なシンボルが削除されることがあります。オブジェクト属性 `__root` を使用して、割込みハンドラなどコンパイルユニット外から使用されるシンボルを保持します。シンボルが `__root` 属性を持たず、ライブラリで定義される場合、代わりにそのライブラリ定義が使用されます。

関連項目

257 ページの *--mfc*、211 ページの*複数ファイルのコンパイルユニット*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［未使用のパブリックを破棄］

--dlib_config

構文

`--dlib_config` *`filename`*`.h|`*`config`*

パラメータ

filename	DLIB 設定ヘッダファイル、237 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、`rl78¥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、111 ページのビルド済ライブラリの使用を参照してください。

自分でビルドしたランタイムライブラリの場合は、対応するカスタマイズしたライブラリ設定ファイルも作成し、コンパイラで指定する必要があります。詳細については、120 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]

--double

構文　`--double={32|64}`

パラメータ

`32`（デフォルト）	32 ビットの double を使用
`64`	64 ビットの double を使用

説明

このオプションを使用して、浮動小数点型 `double` と `long double` を表すためにコンパイラが使用する精度を選択します。コンパイラでは、32 ビットまたは 64 ビットのどちらかの精度を使用できます。デフォルトではコンパイラは 32 ビットの精度を使用します。

関連項目

299 ページの*基本データ型浮動小数点数型*。

[プロジェクト] > [オプション] > [一般オプション] > [ターゲット] > ['double' 型のサイズ]

-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　*169 ページの言語拡張の有効化。*

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［標準（**IAR** 拡張あり）］

注 : IDE では、このオプションがデフォルトで選択されています。

--ec++

構文

```
--ec++
```

説明

コンパイラでは、デフォルトの言語は C 言語です。Embedded C++ を使用する場合は、このオプションを使用してコンパイラで使用する言語に Embedded C++ を設定する必要があります。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［**C++**］

および

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［**C++** の派生言語］>［**Embedded C++**］

--eec++

構文

```
--eec++
```

説明

名前空間や標準テンプレートライブラリなどの拡張 Embedded C++ の機能をソースコードで利用する場合は、このオプションを使用して、コンパイラが使用する言語を拡張 Embedded C++ に設定する必要があります。

関連項目　178 ページの *拡張 Embedded C++*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［**C++**］

および

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［**C++** の派生言語］>［拡張 **Embedded C++**］

--enable_multibytes

構文 `--enable_multibytes`

説明 デフォルトでは、マルチバイト文字を C/C++ ソースコードで使用することはできません。このオプションを有効にすると、ソースコード内のマルチバイト文字は、ホストコンピュータのデフォルトのマルチバイト文字サポート設定に従って解釈されます。

マルチバイト文字は、C/C++ スタイルのコメント、文字列リテラル、文字定数で使用できます。これらはそのまま生成コードに移動します。

［プロジェクト］>［オプション］>［**C/C++ コンパイラ**］>［**言語 2**］>［**マルチバイト文字サポートを有効にする**］

--enable_restrict

構文 `--enable_restrict`

説明 標準の C のキーワード制限を有効にします。このオプションは最適化中の分析の制度を向上するために役立ちます。

このオプションを設定するには、［**プロジェクト**］>［**オプション**］>［**C/C++ コンパイラ**］>［**追加オプション**］を選択します。

--error_limit

構文 `--error_limit=`*n*

パラメータ

n	コンパイラがコンパイルを中止するエラー発生回数（*n* は正の整数）。`0` は制限なしを示します。

説明 `--error_limit` オプションは、コンパイラがコンパイルを停止するエラー数を指定するために使用します。デフォルトでは、エラー数の上限は 100 です。

このオプションは、IDE では使用できません。

-f

構文 `-f filename`

パラメータ 237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明 このオプションは、コンパイラで、指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。

このオプションを設定するには、［**プロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**追加オプション**］を選択します。

--generate_callt_runtime_library_calls

構文 `--generate_callt_runtime_library_calls`

説明 このオプションは、__callt ランタイムライブラリを生成するために使用します。つまり、最も頻繁に使用されるランタイムライブラリルーチンを呼び出すときに、CALLT 命令が使用されます。これによって、コードのサイズが小さくなります。

［**ぷロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**コード**］**>**［**callt ランタイムライブラリ呼出しの有効化**］。

--generate_far_runtime_library_calls

構文 `--generate_far_runtime_library_calls`

説明 このオプションを使用して、アセンブラのサポートルーチンへの __far ランタイムライブラリ呼出しを生成します。このオプションは、対応するアセンブラオプションと併用して、カスタマイズされたライブラリをビルドする際に、サポートルーチンを near から far メモリに移動することができます。起動コードは near メモリに配置される点に注意してください。

このオプションは、ランタイムモデル属性の __far_rt_calls を True（真）に設定します。

注： このオプションは、オプション `--generate_callt_runtime_library_calls` をオーバライドします。

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。

--guard_calls

構文

`--guard_calls`

説明

このオプションは、関数の静的変数初期化のガードを有効にするときに使用します。このオプションは、スレッド環境で使用してください。

注： このオプションには C++ スレッド環境が必要で、これは C/C++ 用 IAR RL78 コンパイラではサポートされていません。

このオプションは、IDE では使用できません。

--header_context

構文

`--header_context`

説明

問題の原因を特定するために、どのソースファイルからどのヘッダファイルがインクルードされたかの確認が必要となる場合があります。このオプションは、診断メッセージごとに、ソースでの問題の位置に加えて、その時点でのインクルードスタック全体を表示する場合に使用します。

このオプションは、IDE では使用できません。

-I

構文

`-I` *`path`*

パラメータ

path	`#include` ファイルの検索パス

説明

このオプションは、`#include` ファイルの検索パスの指定に使用します。このオプションは、1 つのコマンドラインで複数個使用できます。

関連項目

227 ページの*インクルードファイル検索手順*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［プリプロセッサ］>［追加インクルードディレクトリ］

-l

構文

`-l[a|A|b|B|c|C|D][N][H] {filename|directory}`

パラメータ

`a`（デフォルト）	アセンブラリストファイル
`A`	C/C++ ソースをコメントとして記述したアセンブラリストファイル
`b`	基本アセンブラリストファイル。このファイルは、`-la` を使用して生成したリストファイルと同様の内容になっていますが、コンパイラが生成する追加情報（ランタイムモデル属性、呼出しフレーム情報、フレームサイズ情報）は含まれていません *
`B`	基本アセンブラリストファイル。このファイルは、`-lA` を使用して生成したリストファイルと同様の内容になっていますが、コンパイラが生成する追加情報（ランタイムモデル属性、呼出しフレーム情報、フレームサイズ情報）は含まれていません *
`c`	C/C++ リストファイル
`C`（デフォルト）	アセンブラソースをコメントとして記述した C/C++ リストファイル
`D`	C/C++ コメントとしてアセンブラソースを持つリストファイル。ただし、命令オフセットと 16 進数バイト値はなし
`N`	ファイルに診断を含めません
`H`	ヘッダファイルのソース行を出力に含めます。このオプションを指定しない場合は、1 次ソースファイルのソース行のみ含まれます

* この場合、リストファイルはアセンブラへの入力としては使用しにくくなりますが、人には読みやすくなります。

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

説明

このオプションは、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］

--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

```
iccrl78 myfile1.c myfile2.c myfile3.c --mfc
```

関連項目

250 ページの *--discard_unused_publics*、265 ページの *--output, -o*、211 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］

--near_const_location

構文

```
--near_const_location [RAM|ROM0|ROM1]
```

パラメータ

RAM	定数は、RAM の 0xF0000 〜 0xFFFFF の範囲に配置されます
ROM0	定数は ROM の 0x00000 〜 0x0FFFF の範囲に配置され、ハードウェアによって RAM の 0xF0000 〜 0xFFFFF の範囲にミラーされます
ROM1	定数は ROM の 0x10000 〜 0x1FFFF の範囲に配置され、ハードウェアによって RAM の 0xF0000 〜 0xFFFFF の範囲にミラーされます

説明

このオプションは、__near で宣言された定数および文字列の配置を、RAM または ROM からのミラーで指定するときに使用します。

このしくみと使用可能なメモリ範囲については、チップメーカのマニュアル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］>［Near 定数の配置］

--no_clustering

構文

```
--no_clustering
```

説明

このオプションを使用して静的クラスタ最適化を無効にします。

注 : このオプションは、最適化レベルが［中］以上の場合にのみ有効です。

関連項目

215 ページの*静的クラスタ*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［最適化］>［使用可能な変換］>［静的クラスタ］

--no_code_motion

構文

```
--no_code_motion
```

説明

このオプションは、コード移動最適化を無効にする場合に使用します。

注 : このオプションは、最適化レベルが［中］以上の場合にのみ有効です。

関連項目　　214 ページの *コード移動*。

[プロジェクト] > [オプション] > [**C/C++** コンパイラ] > [最適化] > [使用可能な変換] > [コード移動]

--no_cross_call

構文

```
--no_cross_call
```

説明　　このオプションは、クロスコールの最適化を無効化するときに使用します。

注 : このオプションは、最適化レベルが高より下またはスピードを重視して最適化する場合は、効果がありません。これは、その場合にクロスコールの最適化が有効ではないためです。

関連項目　　216 ページの *クロスコール*。

[プロジェクト] > [オプション] > [**C/C++** コンパイラ] > [最適化] > [使用可能な変換] > [クロスコール]

--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　　213 ページの *共通部分式除去*。

[プロジェクト] > [オプション] > [**C/C++** コンパイラ] > [最適化] > [使用可能な変換] > [共通部分式除去]

--no_dwarf3_cfi

構文

```
--no_dwarf3_cfi
```

説明　　DWARF 3 の呼出しフレーム命令の生成を無効にします。これによって、呼出しフレーム情報の質に影響する点に注意してください。

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。

--no_fragments

構文

```
--no_fragments
```

説明

このオプションはセクションフラグメント処理を無効にします。通常、ツールセットは、セクションフラグメント情報をリンカに転送するために IAR 独自の情報を使用します。リンカは、この情報を使用して、未使用のコードおよびデータを削除し、実行可能イメージのサイズをさらに最小化します。このオプションを使用すると、情報はオブジェクトファイルに出力されません。

関連項目

99 ページの*シンボルおよびセクションの保持*。

このオプションを設定するには、［**プロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**追加オプション**］を選択します。

--no_inline

構文

```
--no_inline
```

説明

このオプションは、関数インライン化を無効にする場合に使用します。

関連項目

79 ページの*インライン関数*。

［**プロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**最適化**］**>**［**使用可能な変換**］**>**［**関数インライン化**］

--no_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

354 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_scheduling

構文 `--no_scheduling`

説明 このオプションは、命令スケジューラを無効にするときに使用します。

注 : このオプションは、最適化レベルが高より下、またはサイズを重視して最適化する場合は、効果がありません。これは、その場合に命令スケジューリングが有効ではないためです。

関連項目 216 ページの*命令スケジューリング*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［最適化］>［使用可能な変換］>［命令スケジューリング］

--no_size_constraints

構文 `--no_size_constraints`

説明 このオプションを使用して、高速度の最適化の時にコードサイズの拡張に対する通常の制限を緩和します。

注 : このオプションは、`-Ohs` とともに使用したときだけ効果があります。

関連項目 213 ページの*速度とサイズ*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［最適化］>［使用可能な変換］>［サイズの制約なし］

--no_static_destruction

構文 `--no_static_destruction`

説明 通常は、コンパイラはコードを出力して、プログラム終了時に破壊が必要な C++ 静的変数を破壊します。こうした破壊が不要なこともあります。

このオプションを使用して、こうしたコードの出力を無効にします。

関連項目 100 ページの *atexit 制限の設定*。

このオプションを設定するには、［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［追加オプション］を選択します。

--no_system_include

構文

```
--no_system_include
```

説明

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

関連項目

250 ページの *--dlib_config*、269 ページの *--system_include_dir*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［プリプロセッサ］>［標準のインクルードディレクトリを無視］

--no_tbaa

構文

```
--no_tbaa
```

説明

このオプションは、型ベースエイリアス解析を無効にする場合に使用します。

注 : このオプションは、最適化レベルが［高］の場合にのみ有効です。

関連項目

215 ページの*型ベースエイリアス解析*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［最適化］>［使用可能な変換］>［型ベースエイリアス解析］

--no_typedefs_in_diagnostics

構文

```
--no_typedefs_in_diagnostics
```

説明

このオプションは、診断で typedef 名の使用を無効化する場合に使用します。通常は、コンパイラからのメッセージ（ほとんどの場合は何らかの診断メッセージ）で型についての記述がある場合、元の宣言で使用されていた typedef 名の方のテキストが短くなるときは typedef 名で記述されます。

例

```
typedef int (*MyPtr)(char const *);
MyPtr p = "My text string";
```

この場合、以下のようなエラーメッセージが表示されます。

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

--no_typedefs_in_diagnostics オプションを使用した場合は、エラーメッセージは以下のようになります。

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

このオプションを設定するには、［**プロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**追加オプション**］を選択します。

--no_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　214 ページの*ループ展開*。

［**プロジェクト**］**>**［**オプション**］**>**［**C/C++ コンパイラ**］**>**［**最適化**］**>**［**使用可能な変換**］**>**［**ループ展開**］

--no_warnings

構文　--no_warnings

説明　デフォルトでは、コンパイラは警告メッセージを出力します。このオプションは、すべてのワーニングを無効にする場合に使用します。

このオプションは、IDE では使用できません。

--no_wrap_diagnostics

構文　--no_wrap_diagnostics

説明　デフォルトでは、診断メッセージ中の長い行は、読みやすくするため複数行に分割されます。このオプションは、診断メッセージのラインラッピングを無効にする場合に使用します。

このオプションは、IDE では使用できません。

-O

構文 `-O[n|l|m|h|hs|hz]`

パラメータ

n	なし *（デバッグサポートに最適）
l（デフォルト）	低 *
m	中
h	高（バランス）
hs	高（速度優先）
hz	高（サイズ優先）

*「なし」と「低」の最も重要な違いは、「なし」ではすべての非静的変数がその変数のスコープ内全体で有効になることです。

説明 このオプションは、コンパイラでコードを最適する際に使用される最適化レベルを設定する場合に使用します。最適化オプションを指定していない場合は、デフォルトで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 210 ページの *コンパイラ最適化の設定*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［最適化］

--only_stdout

構文 `--only_stdout`

説明 コンパイラで、通常はエラー出力ストリーム (`stderr`) に転送されるメッセージにも標準出力ストリーム (`stdout`) を使用する場合に、このオプションを使用します。

このオプションは、IDE では使用できません。

--output, -o

構文

```
--output {filename|directory}
-o {filename|directory}
```

パラメータ

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

デフォルトでは、コンパイラで生成されたオブジェクトコード出力は、ソースファイルと同じ名前で、拡張子が o のファイルに配置されます。このオプションは、オブジェクトコード出力用に別の出力ファイル名を明示的に指定する場合に使用します。

このオプションは、IDE では使用できません。

--pending_instantiations

構文

```
--pending_instantiations number
```

パラメータ

number	上限を指定する整数で、64 がデフォルト値です。0 を使用すると上限がなくなります。

説明

このオプションを使用して、任意の時点でインスタンス化できる任意の C++ テンプレートのインスタンス化の最大数を指定します。これは、再帰的なインスタンス化を検出するために使用します。

［プロジェクト］>［オプション］>［**C/C++ コンパイラ**］>［追加オプション］

--predef_macros

構文

```
--predef_macros {filename|directory}
```

パラメータ

237 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

237 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

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

説明

このオプションは、プリプロセッサ出力を指定ファイルに生成する場合に使用します。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［プリプロセッサ］>［ファイルへのプリプロセッサ出力］

--public_equ

構文

`--public_equ` *`symbol`*`[=`*`value`*`]`

パラメータ

`symbol`	定義するアセンブラシンボルの名前
value	定義したアセンブラシンボルの値（任意指定）

説明

このオプションは、`EQU` ディレクティブを使用してアセンブラ言語でラベルを定義し、`PUBLIC` ディレクティブを使用してエクスポートする操作と等価です。このオプションは、1 つのコマンドラインで複数個使用できます。

このオプションは、IDE では使用できません。

--relaxed_fp

構文

`--relaxed_fp`

説明

このオプションを使用して、コンパイラで言語規則を緩和させ、浮動小数点式の最適化をより積極的に実行します。このオプションは、以下の条件を満たす浮動小数点式のパフォーマンスを向上させます。

- 式に単精度および倍精度の値が両方含まれている。
- 倍精度の値が精度を失わずに単精度に変換できる。
- 式の結果は単精度に変換されます。

倍制度の代わりに単精度で計算を実行すると、精度が失われることがあります。

例

```
float F(float a, float b)
{
  return a + b * 3.0;
}
```

C 標準では、この例における `3.0` の型が `double` であるため、式全体が倍精度で評価される必要があります。ただし、`--relaxed_fp` オプションを使用する場合、3.0 は `float` に変換され、式全体が `float` 精度で評価できるようになります。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **2**］>［浮動小数点数動作］

--remarks

構文
```
--remarks
```

説明

最も軽度の診断メッセージを、リマークと呼びます。リマークは、ソースコード中で、生成したコードで異常な動作の原因となる可能性がある部分を示します。デフォルトでは、コンパイラはリマークを生成しません。このオプションは、コンパイラでリマークを生成する場合に使用します。

関連項目

232 ページの*重要度*。

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［診断］>［リマークの有効化］

--require_prototypes

構文
```
--require_prototypes
```

説明

このオプションは、すべての関数が正しいプロトタイプを持つかどうかをコンパイラで強制的に検証する場合に使用します。このオプションを使用すると、以下のいずれかが含まれるコードではエラーが発生します。

- 宣言のない関数、またはカーニハン & リッチー C 形式で宣言された関数の呼出し
- 先にプロトタイプが宣言されていない public 関数の関数定義
- プロトタイプを含まない型の関数ポインタによる間接的な関数呼出し

［プロジェクト］>［オプション］>［C/C++ コンパイラ］>［言語 1］>［プロトタイプの強制］

--silent

構文
```
--silent
```

説明

デフォルトでは、コンパイラは開始メッセージや最終的な統計レポートを出力します。このオプションは、コンパイラがこれらのメッセージを標準出力ストリーム（通常は画面）に送信しないようにする場合に使用します。

このオプションは、エラー / ワーニングメッセージの表示には影響しません。

このオプションは、IDE では使用できません。

--strict

構文

```
--strict
```

説明

デフォルトでは、コンパイラで標準の C/C++ に緩く対応したスーパセットを使用できます。このオプションを使用して、アプリケーションのソースコードが厳密な標準の C/C++ に準拠するよう徹底します。

注 : `-e` オプションと `--strict` オプションは、同時に使用できません。

関連項目

169 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、237 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

250 ページの *--dlib_config*、262 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。

--use_c++_inline

構文

```
--use_c++_inline
```

説明

標準の C では、`inline` キーワードに対して C++ の場合とはわずかに異なる動作が使用されます。C を使用する際に C++ の動作が必要な場合は、このオプションを使用してください。

関連項目

79 ページの*インライン関数*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［**C** の派生言語］>［**C99**］>［**C++** インライン動作］

--use_unix_directory_separators

構文

```
--use_unix_directory_separators
```

説明

このオプションを使用して、DWARF デバッグ情報で / を（¥ の代わりに）ファイルパスのディレクトリ区切り文字として使用します。

このオプションは、UNIX 形式のディレクトリ区切り文字を必要とするデバッガを使用する場合に便利です。

このオプションを設定するには、［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［追加オプション］を選択します。

--vla

構文

```
--vla
```

説明

このオプションを使用して、C99 の可変長配列のサポートを有効にします。こうした配列は、ヒープに配置されます。このオプションには標準の C が必要であり、`--c89` コンパイラオプションとは併用できません。

注 : `--vla` と `longjmp` ライブラリ関数は併用できません。併用するとメモリリークとなることがあります。

関連項目

167 ページの *C 言語の概要*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［言語 **1**］>［**C** の派生言語］>［**VLA** の許可］

--warn_about_c_style_casts

構文

```
--warn_about_c_style_casts
```

説明

このオプションを使用して、C- スタイルキャストが C++ ソースコードで使用されるときにコンパイラを警告します。

このオプションは、IDE では使用できません。

--warnings_affect_exit_code

構文 `--warnings_affect_exit_code`

説明 デフォルトでは、ゼロ以外の終了コードが生成されるのはエラーが発生した場合のみであるため、ワーニングは終了コードには影響しません。このオプションを使用すると、ワーニングが発生した場合もゼロ以外の終了コードが生成されます。

このオプションは、IDE では使用できません。

--warnings_are_errors

構文 `--warnings_are_errors`

説明 このオプションは、コンパイラでワーニングをエラーとして処理する場合に使用します。コンパイラがエラーを検出した場合、オブジェクトコードは生成されません。リマークに変更されたワーニングは、エラーとして処理されません。

注：オプション `--diag_warning` または `#pragma diag_warning` ディレクティブにより警告として再分類された診断メッセージも、`--warnings_are_errors` 使用時はエラーとして処理されます。

関連項目 249 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］

--workseg_area

構文 `--workseg_area[=`*size*`]`

パラメータ

size	workseg 領域のサイズ。パラメータを省略した場合、デフォルトのサイズは 20 となり、最大値は 128 です。

説明 `.wrkseg` セクションの saddr メモリに予約する空間を指定します。これは、*レジスタ*変数作業領域として使用されます。

注：関数で修飾子 `__no_save` が使用されていない限り、各関数の入口と出口で作業エリアを保存して復元すると、オーバヘッドがあります。

関連項目

70 ページの*ショートアドレス作業エリア*、408 ページの *.wrkseg*。

［プロジェクト］>［オプション］>［**C/C++** コンパイラ］>［コード］>［ショートアドレス作業領域］

リンカオプション

- リンカオプションの概要
- リンカオプションの説明

一般的な構文規則については、235 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
`--config`	リンカによって使用されるリンカ設定ファイルを指定します
`--config_def`	設定ファイルのシンボルを定義します
`--config_search`	リンカ設定ファイルの検索に使用する追加のディレクトリを指定します
`--cpp_init_routine`	ユーザ定義 C++ 動的初期化ルーチンを指定します
`--debug_lib`	C-SPY デバッグライブラリを使用
`--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
`--error_limit`	リンクを停止するエラー数の上限を指定
`--export_builtin_config`	デフォルト設定の `icf` ファイルを生成します
`-f`	コマンドラインを拡張

表 30: 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
--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
--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
--merge_duplicate_sections	同等のリードオンリーのセクションをマージ
--misrac1998	MISRA-C:1998 固有のエラーメッセージを有効にします。『*IAR Embedded Workbench® MISRA C:1998 リファレンスガイド*』を参照してください
--misrac2004	MISRA-C:2004 固有のエラーメッセージを有効にします。『*IAR Embedded Workbench® MISRA C:2004 リファレンスガイド*』を参照してください
--misrac_verbose	MISRA-C チェックの冗長なロギングを有効にします。『*IAR Embedded Workbench® MISRA-C:1998 リファレンスガイド*』および『*IAR Embedded Workbench® MISRA-C:2004 リファレンスガイド*』を参照してください
--no_fragments	セクションフラグメント処理を無効化
--no_library_search	自動ランタイムライブラリ検索を無効にします
--no_locals	ローカルシンボルを ELF 実行可能イメージから削除します
--no_range_reservations	絶対シンボルの範囲予約を無効化
--no_remove	未使用のセクションの削除を無効にします
--no_vfe	仮想関数の除去を無効化
--no_warnings	ワーニングの生成を無効にします
--no_wrap_diagnostics	診断メッセージの長い行をラップしません
-o	オブジェクトファイル名を設定。--output のエイリアス

表 30: 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
--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
--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（ichecksum 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
--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
--whole_archive	アーカイブにあるすべてのオブジェクトファイルを、コマンドライン上で指定したときと同じように扱います

表 30: リンカオプションの概要（続き）

リンカオプションの説明

次のセクションでは、それぞれのリンカオプションについて詳細に説明します。

［追加オプション］ページを使用して特定のコマンドラインオプションを指定する場合、IDE では、オプションの競合、オプションの重複、不適切なオプションの使用などの整合性問題のインスタントチェックは実行しません。

--config

構文

`--config filename`

パラメータ

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションは、リンカにより使用される設定ファイルを指定するときに使用します（デフォルトのファイル名拡張子は `icf` です)。設定ファイルが指定されていない場合、デフォルトの設定が使用されます。このオプションは、1 つのコマンドラインで 1 回だけ使用できます。

関連項目

*リンカ設定ファイル*の章。

[プロジェクト] > [オプション] > [リンカ] > [設定] > [リンカ設定ファイル]

--config_def

構文

`--config_def` *`symbol=constant_value`*

パラメータ

`symbol`	設定ファイルで使用されるシンボルの名前。
`constant_value`	設定シンボルの定数値。

説明

このオプションは、設定ファイルで使用される定数設定シンボルを定義するときに使用します。このオプションの効果は、リンカ設定ファイルの `define symbol` ディレクティブと同じです。このオプションは、1 つのコマンドラインで複数個使用できます。

関連項目

278 ページの *--define_symbol*、104 ページの *ILINK とアプリケーション間の相互処理*。

[プロジェクト] > [オプション] > [リンカ] > [設定] > [設定ファイルに定義されたシンボル]

--config_search

構文

`--config_search` *`path`*

パラメータ

`path`	リンカがリンカ設定インクルードファイルを検索するディレクトリのパス。

説明

このオプションを使用して、リンカ設定ファイルで `include` ディレクティブを処理する際に、ファイル検索で使用する追加のディレクトリを指定します。

デフォルトでは、リンカはシステム設定ディレクトリ内の設定インクルードファイルのみを検索します。複数の検索ディレクトリを指定するには、各パスについてこのオプションを使用します。

関連項目

395 ページの *include ディレクティブ*。

このオプションを設定するには、[**プロジェクト**] > [**オプション**] > [**リンカ**] > [**追加オプション**] を使用します。

--cpp_init_routine

構文

`--cpp_init_routine routine`

パラメータ

routine	ユーザ定義 C++ 動的初期化ルーチン。

説明

IAR C/C++ コンパイラおよび標準ライブラリを使用する場合、C++ 動的初期化は自動的に処理されます。これ以外の場合、このオプションの使用が必要になることがあります。

セクション型が `INIT_ARRAY` または `PREINIT_ARRAY` のセクションがアプリケーションに含まれている場合、C++ 動的初期化ルーチンは必須です。デフォルトでは、このルーチンの名前は `__iar_cstart_call_ctors` で、標準ライブラリの起動コードで呼出されます。このオプションは、標準ライブラリを使用していないときに、別ルーチンでこれらのセクション型を処理する必要がある場合に使用します。

このオプションを設定するには、[**プロジェクト**] > [**オプション**] > [**リンカ**] > [**追加オプション**] を使用します。

--debug_lib

構文

`--debug_lib`

説明

このオプションを使用して C-SPY デバッグライブラリをインクルードします。

関連項目

116 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**] > [**リンカ**] > [**ライブラリ**] > [**C-SPY デバッグサポートを含める**]

--define_symbol

構文

```
--define_symbol symbol=constant_value
```

パラメータ

symbol	アプリケーションで使用できる定数シンボルの名前。
constant_value	シンボルの定数値。

説明

このオプションは、アプリケーションで使用できる定数シンボル、すなわちラベルを定義するときに使用します。このオプションは、1 つのコマンドラインで複数個使用できます。このオプションは、define symbol ディレクティブと異なる点に注意してください。

関連項目

276 ページの *--config_def*、104 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。

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

説明

このオプションは、ファイル入力のために開かれたリンカ設定ファイル、オブジェクトファイル、ライブラリファイルのファイル名を、デフォルトのファイル名拡張子 i を持つファイルに含める場合に使用します。

例

--dependencies や --dependencies=i を使用すると、開かれている各入力ファイルの名前とフルパス（ある場合）が独立した行に出力されます。次に例を示します。

```
c:¥myproject¥foo.o
d:¥myproject¥bar.o
```

--dependencies=m を使用した場合は、makefile スタイルで出力されます。各入力ファイルについて、makefile の依存関係規則を含む行が生成されます。

各行は出力ファイル名、コロン、空白文字、入力ファイル名で構成されます。次に例を示します。

```
a.out: c:¥myproject¥foo.o
a.out: d:¥myproject¥bar.o
```

このオプションは、IDE では使用できません。

--diag_error

構文

`--diag_error=tag[,tag,...]`

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージをエラーとして再分類する場合に使用します。エラーは、実行可能イメージが生成されなくなるような重要度の問題を示します。終了コードはゼロ以外になります。このオプションは、1H つのコマンドラインで複数個使用できます。

［プロジェクト］>［オプション］>［リンカ］>［診断］>［エラーとして処理］

--diag_remark

構文

`--diag_remark=tag[,tag,...]`

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）

説明

このオプションは、特定の診断メッセージをリマークとして再分類する場合に使用します。リマークは、最も軽微な診断メッセージです。実行可能イメージでの異常な動作の原因となる可能性がある構造が存在することを示します。このオプションは、1 つのコマンドラインで複数個使用できます。

注：デフォルトでは、リマークは表示されません。リマークを表示するには、`--remarks` オプションを使用します。

［プロジェクト］>［オプション］>［リンカ］>［診断］>［リマークとして処理］

--diag_suppress

構文
```
--diag_suppress=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe117 など）

説明

このオプションは、特定の診断メッセージを無効にする場合に使用します。これらのメッセージは表示されなくなります。このオプションは、1 つのコマンドラインで複数個使用できます。

［プロジェクト］>［オプション］>［リンカ］>［診断］>［診断を無効化］

--diag_warning

構文
```
--diag_warning=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe826 など）

説明

このオプションは、特定の診断メッセージをワーニングとして再分類する場合に使用します。ワーニングは、問題はあるが、リンク処理の終了前にリンカが終了する原因にはならないエラーや脱落を示します。このオプションは、1 つのコマンドラインで複数個使用できます。

［プロジェクト］>［オプション］>［リンカ］>［診断］>［ワーニングとして処理］

--diagnostics_tables

構文
```
--diagnostics_tables {filename|directory}
```

パラメータ

237 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。

--entry

構文

`--entry symbol`

パラメータ

symbol	ルートシンボルおよび開始ラベルとして処理されるシンボルの名前

説明

このオプションは、ルートシンボルおよびアプリケーションの開始ラベルとして処理されるシンボルを作成するときに使用します。これは、ローダを使用する場合に便利です。このオプションを使用しない場合、デフォルトの開始シンボルは `__iar_program_start` です。ルートシンボルは、そのモジュールが含まれる場合、アプリケーション内で参照されるかどうかに関わらず保持されます。オブジェクトファイルのモジュールは常に含まれますが、ライブラリのモジュールパートは必要な場合にのみ含まれます。

注 : 参照先のラベルは、アプリケーション内で使用可能でなければなりません。また、リセットベクタが必ず新しい開始ラベルを参照するようにしてください（たとえば、`--redirect __iar_program_start=_myStartLabel` など）。

［プロジェクト］>［オプション］>［リンカ］>［ライブラリ］>［デフォルトプログラムエントリのオーバライド］

--error_limit

構文

`--error_limit=n`

パラメータ

n	リンカがリンクを中止するエラー発生回数（*n* は正の整数）。`0` は制限なしを示します。

説明

`--error_limit` オプションは、リンカがリンクを停止する前のエラー数を指定するために使用します。デフォルトでは、エラー数の上限は 100 です。

このオプションは、IDE では使用できません。

--export_builtin_config

構文 `--export_builtin_config filename`

パラメータ
237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明
デフォルトで使用される設定をファイルにエクスポートします。

このオプションは、IDE では使用できません。

-f

構文 `-f filename`

パラメータ
237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明
このオプションは、リンカで、指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。

このオプションを設定するには、［プロジェクト］>［オプション］>［リンカ］>［追加オプション］を使用します。

--force_output

構文 `--force_output`

説明
このオプションは、リンクエラーに関係なく出力実行可能イメージを生成するときに使用します。

このオプションを設定するには、［プロジェクト］>［オプション］>［リンカ］>［追加オプション］を使用します。

--image_input

構文

`--image_input filename [,symbol,[section[,alignment]]]`

パラメータ

filename	リンクする raw イメージを含むピュアバイナリファイル。
symbol	バイナリデータを参照できるシンボル。
section	バイナリデータを配置するセクション。デフォルトは .text です。
alignment	セクションのアラインメント。デフォルトは 1 です。

説明

このオプションは、通常の入力ファイルの他に、ピュアバイナリファイルをリンクします。ファイルの全体の内容がセクションに配置されます。つまり、ピュアバイナリデータのみを含むことができます。

注：オブジェクトファイルからのセクションの場合のみ、`--image_input` オプションを使用して作成されたセクションは実際に必要でない限り含まれません。オプションでシンボルを指定してアプリケーションでこのシンボルを参照（または `--keep` オプションを使用）するか、あるいはセクション名を指定してリンカ設定ファイルで `keep` ディレクティブを使用し、セクションが含まれていることを確認します。

例

`--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4`

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

283 ページの *--keep*。

［プロジェクト］>［オプション］>［リンカ］>［入力］>［ロウバイナリイメージ］

--keep

構文

`--keep symbol`

パラメータ

symbol	ルートシンボルとして処理されるシンボルの名前

説明

一般的にリンカでは、アプリケーションで必要なシンボルのみを保存します。このオプションは、シンボルを常に最終アプリケーションに含めるときに使用します。

[プロジェクト] > [オプション] > [リンカ] > [入力] > [シンボルをキープ]

--log

構文

`--log topic[,topic,...]`

パラメータ

ここで、*topic* は以下のいずれかです。

`initialization`	各バッチに選択されたコピーバッチおよび圧縮をリストします。
`libraries`	自動ライブラリセレクタによって行われるすべての決定をリストします。これには、必要なその他のシンボル (`--keep`)、リダイレクト (`--redirect`)、どのランタイムライブラリが選択されたかが含まれることがあります。
`modules`	アプリケーションにインクルードするよう選択された各モジュールと、インクルードの要因となったシンボルをリストします。
`redirects`	リダイレクトされたシンボルをリストします。
`sections`	アプリケーションにインクルードするよう選択された各モジュールとセクションのフラグメント、インクルードの要因となった依存関係をリストします。
`unused_fragments`	アプリケーションにインクルードされなかったセクションフラグメントをリストします。

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

285 ページの *--log_file*。

[プロジェクト] > [オプション] > [リンカ] > [リンカ] > [ログの生成]

--log_file

構文
`--log_file filename`

パラメータ
237 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
284 ページの *--log*。

［プロジェクト］>［オプション］>［リンカ］>［リンカ］>［ログの生成］

--mangled_names_in_messages

構文
`--mangled_names_in_messages`

説明
このオプションは、メッセージの C/C++ シンボルにマングル化した名前とデマングル化した名前の両方を生成するときに使用します。マングル化とは、複雑な C 名や C++ 名を簡単な名前にマッピングするときに使用するテクニックです（オーバロードなどに利用）。たとえば、`void h(int, char)` は、`_Z1hic` になります。

このオプションは、IDE では使用できません。

--map

構文
`--map {filename|directory}`

説明
このオプションは、リンカメモリマップファイルを生成するときに使用します。マップファイルのデフォルトのファイル名拡張子は `map` です。このマップファイルの内容は、以下のとおりです。

- リンカのバージョン、現在の日時、使用されたコマンドラインをリストするマップファイルヘッダのリンクの概要。
- ランタイム属性をリストするランタイム属性の概要。
- 配置ディレクティブでソートしアドレス順序で各セクション/ブロックをリストした配置の概要。
- データ範囲、パッキング方手法、圧縮率をリストする初期化テーブルのレイアウト。

- 各モジュールからイメージへのメモリ使用率を、ディレクトリおよびライブラリでソートしてリストするモジュールの概要。
- すべてのパブリックシンボルおよび一部のローカルシンボルをアルファベット順に表示し、そのシンボルを含むモジュールを一覧表示したエントリリスト。
- それらのバイトの一部が*「共有」*として報告されることもあります。

  共有オブジェクトとは、モジュール間で共有される関数またはデータオブジェクトのことです。このような共有が 2 つ以上のモジュールで発生した場合、1 つの関数 / データオブジェクトのみが保持されます。たとえば、インライン関数がインライン化されない場合があります。これは、これらの関数が共有とマークされていて、各関数の 1 つのインスタンスしか最終的なアプリケーションにインクルードされないためです。この仕組みは、特定の関数や変数に直接的には関連しないコンパイラ生成コードやデータで、最終的なアプリケーションには 1 つのインスタンスしか必要とされない場合にも使用されることがあります。

このオプションは、1 つのコマンドラインで 1 回だけ使用できます。

［プロジェクト］>［オプション］>［リンカ］>［リスト］>［リンカマップファイルの表示］

--merge_duplicate_sections

構文

```
--merge_duplicate_sections
```

説明

このオプションを使用して、リードオンリーのセクションのコピーを 1 つだけ保持します。これによって異なる関数や定数が同じアドレスを持つことがあるため、このオプションを有効にすると、異なるアドレスに依存するアプリケーションが正しく機能しなくなるため注意してください。

［プロジェクト］>［オプション］>［リンカ］>［最適化］>［重複セクションのマージ］

--no_fragments

構文

```
--no_fragments
```

説明

このオプションはセクションフラグメント処理を無効にします。通常、ツールセットは、セクションフラグメント情報をリンカに転送するために IAR 独自の情報を使用します。リンカは、この情報を使用して、未使用のコードおよびデータを削除し、実行可能イメージのサイズをさらに最小化します。このオプションでは、セクションのフラグメントの除去を無効にして、各セク

ション全体をインクルードまたは除外します。通常、アプリケーションのサイズが大きくなります。

関連項目

99 ページの *シンボルおよびセクションの保持*。

このオプションを設定するには、[プロジェクト] > [オプション] > [リンカ] > [追加オプション] を使用します。

--no_library_search

構文

`--no_library_search`

説明

このオプションは、自動ランタイムライブラリ検索を無効にするときに使用します。このオプションは、正しい標準ライブラリの自動追加を無効にします。これは、たとえば、ユーザが構築した標準ライブラリをアプリケーションで必要な場合などに役に立ちます。

このオプションは、自動ライブラリ選択のすべての手順を無効にする点に注意してください。標準ライブラリを使用する場合は、一部の手順を複製しなければならないことがあります。 どの手順を複製するか判断するには、`--log libraries` リンカオプションと自動ライブラリ選択を有効にしてともに使用します。

[プロジェクト] > [オプション] > [リンカ] > [ライブラリ] > [自動ランタイムライブラリ選択]

--no_locals

構文

`--no_locals`

説明

このオプションは、ローカルシンボルを ELF 実行可能イメージから削除するときに使用します。

注 : このオプションは、実行可能イメージの DWARF 情報からはローカルシンボルを削除しません。

[プロジェクト] > [オプション] > [リンカ] > [出力]

--no_range_reservations

構文　`--no_range_reservations`

説明　通常は、リンカは絶対シンボルが使用するすべての範囲をサイズゼロとして予約し、`place in` コマンドの対象から除外します。

このオプションを使用する場合、これらの予約は無効になり、リンカは絶対シンボルの範囲と重複する形でセクションを自由に配置することができます。

このオプションを設定するには、［**プロジェクト**］>［**オプション**］>［**リンカ**］>［**追加オプション**］を使用します。

--no_remove

構文　`--no_remove`

説明　このオプションが使用されている場合、未使用のセクションは削除されません。つまり、実行可能イメージに含まれる各モジュールには、その元のセクションがすべて含まれます。

関連項目　99 ページの*シンボルおよびセクションの保持*。

このオプションを設定するには、［**プロジェクト**］>［**オプション**］>［**リンカ**］>［**追加オプション**］を使用します。

--no_vfe

構文　`--no_vfe`

説明　このオプションは、仮想関数除去の最適化を無効化するときに使用します。少なくとも 1 つのインスタンスを持つ全クラスのすべての仮想関数が保持され、ランタイム型情報データはすべてのポリモーフィッククラスで保持されます。また、VFE 情報を持たないモジュールについてワーニングメッセージは出力されません。

関連項目　292 ページの *--vfe*、201 ページの*仮想関数の除去*。

オプションを設定するには、以下のように選択します。

［**プロジェクト**］>［**オプション**］>［**リンカ**］>［**最適化**］>［**C++ 仮想関数除去を実行**］

--no_warnings

構文 `--no_warnings`

説明 デフォルトでは、リンカは警告メッセージを出力します。このオプションは、すべてのワーニングを無効にする場合に使用します。

このオプションは、IDE では使用できません。

--no_wrap_diagnostics

構文 `--no_wrap_diagnostics`

説明 デフォルトでは、診断メッセージ中の長い行は、読みやすくするため複数行に分割されます。このオプションは、診断メッセージのラインラッピングを無効にする場合に使用します。

このオプションは、IDE では使用できません。

--only_stdout

構文 `--only_stdout`

説明 リンカで、通常はエラー出力ストリーム (`stderr`) に転送されるメッセージにも標準出力ストリーム (`stdout`) を使用する場合に、このオプションを使用します。

このオプションは、IDE では使用できません。

--output, -o

構文

```
--output {filename|directory}
-o {filename|directory}
```

パラメータ 237 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］

--place_holder

構文

`--place_holder symbol[,size[,section[,alignment]]]`

パラメータ

symbol	作成するシンボルの名前
size	ROM のサイズ、デフォルトは 4 バイト
section	使用するセクション名。デフォルトは .text
alignment	セクションのアラインメント。デフォルトは 1

説明

このオプションは、たとえば、ielftool により計算されるチェックサムなど、他のツールによってフィルされる ROM の部分を予約するときに使用します。このリンカオプションを使用するたびに、指定した名前、サイズ、アラインメントのセクションが生成されます。シンボルは、そのセクションを参照するためにアプリケーションで使用できます。

注：他のセクションと同様、--place_holder オプションにより作成されるセクションは、そのセクションが必要だとみなされた場合のみアプリケーションに含まれます。--keep リンカオプション、または keep リンカディレクティブを使用すると、このようなセクションを強制的に含めることができます。

関連項目

409 ページの *IAR ユーティリティ*。

このオプションを設定するには、**［プロジェクト］>［オプション］>［リンカ］>［追加オプション］**を使用します。

--redirect

構文

`--redirect from_symbol=to_symbol`

パラメータ

from_symbol	変更前のシンボルの名前
to_symbol	変更後のシンボルの名前

説明

このオプションは、シンボルの参照を変更するときに使用します。

このオプションを設定するには、[プロジェクト] > [オプション] > [リンカ] > [追加オプション] を使用します。

--remarks

構文

```
--remarks
```

説明

最も軽度の診断メッセージを、リマークと呼びます。リマークは、ソースコード中で、生成したコードで異常な動作の原因となる可能性がある部分を示します。デフォルトでは、リンカはリマークを生成しません。このオプションは、リンカでリマークを生成する場合に使用します。

関連項目

232 ページの*重要度*。

[プロジェクト] > [オプション] > [リンカ] > [診断] > [リマークの有効化]

--search

構文

`--search` *path*

パラメータ

path	リンカがオブジェクトやライブラリファイルを検索するディレクトリのパス。

説明

このオプションを使用して、リンカがオブジェクトやライブラリファイルを検索するディレクトリを追加して指定します。

デフォルトでは、リンカは作業ディレクトリにあるオブジェクトおよびライブラリファイルのみを検索します。コマンドライン上でこのオプションを使用するたびに、検索ディレクトリが 1 つ追加されます。

関連項目

51 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。

--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で必要な仮想関数除去の情報が不足している場合でも、仮想関数除去を実行します。

説明

このオプションを使用して、1 つまたは複数のモジュールで必要な仮想関数除去の情報が不足している場合でも、仮想関数除去を実行します。パラメータ `forced` がないと、このオプションは効果がありません。

仮想関数除去を強制的に使用すると、必要な情報を持たないモジュールが仮想関数の呼出しを実行したり、動的ランタイム型情報を使用する場合に、安全でなくなる可能性があります。

関連項目

288 ページの *--no_vfe*、201 ページの *仮想関数の除去*。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［リンカ］>［最適化］>［**C++** 仮想関数除去を実行］

--warnings_affect_exit_code

構文

`--warnings_affect_exit_code`

説明

デフォルトでは、ゼロ以外の終了コードが生成されるのはエラーが発生した場合のみであるため、ワーニングは終了コードには影響しません。このオプションを使用すると、ワーニングが発生した場合もゼロ以外の終了コードが生成されます。

このオプションは、IDE では使用できません。

--warnings_are_errors

構文

`--warnings_are_errors`

説明

このオプションは、リンカでワーニングをエラーとして処理する場合に使用します。リンカがエラーを検出した場合は、実行可能イメージは生成されません。リマークに変更されたワーニングは、エラーとして処理されません。

注：オプション `--diag_warning` オプションによりワーニングとして再分類された診断メッセージも、`--warnings_are_errors` 使用時はエラーとして処理されます。

関連項目

249 ページの *--diag_warning*、280 ページの *--diag_warning*。

［プロジェクト］>［オプション］>［リンカ］>［診断］>［すべてのワーニングをエラーとして処理］

--whole_archive

構文

`--whole_archive` *`filename`*

パラメータ

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

説明

このオプションを使用して、アーカイブにあるすべてのオブジェクトファイルを、コマンドライン上で指定したときと同じように扱います。これは、常

にオブジェクトファイル（ファイル名拡張子 o）から含まれるルートコンテンツがアーカイブに含まれているものの、モジュールから何らかのエントリが参照されている場合にのみアーカイブからインクルードされるときに役立ちます。

例

`archive.a` にオブジェクトファイル `file1.o`、`file2.o`、`file3.o` が含まれる場合、`--whole_archive archive.a` を使用すると、`file1.o file2.o file3.o` と指定したときと同じになります。

関連項目

98 ページの *モジュールの保持*。

このオプションを設定するには、［**プロジェクト**］**>**［**オプション**］**>**［**リンカ**］**>**［**追加オプション**］を使用します。

データ表現

- アラインメント
- 基本データ型整数型
- 基本データ型浮動小数点数型
- ポインタ型
- 構造体型
- 型修飾子
- C++ のデータ型

特定のアプリケーションで最も効率的なコードを作成するためのデータ型やポインタについては、章*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るとします。この場合、`long` 型の整数など、4 バイトオブジェクトのアラインメントは 4 になります。

また、一度に 2 バイトしか読み取ることができないプロセッサの場合には、4 バイトの `long` 型整数のアラインメントは 2 になります。

構造体のアラインメントは、アラインメントが最も厳密な構造体メンバと同じです。構造体とそのメンバへのアラインメント要件を軽減するには、`#pragma pack` 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ディングバイトを追加しなければならないことがあります。パディングバイトについては、304 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。

RL78 マイクロコントローラ のアラインメント

RL78 マイクロコントローラは、8 ビットまたは 16 ビットアクセスを使用してメモリにアクセスできます。ただし、16 ビットのアクセスを実行する場合、データは偶数アドレスの位置になければなりません。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、RL78 マイクロコントローラ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。この場合、ワードアセンブラ命令 `MOVW` が使用されます。

注 : すべての配列と文字列リテラルがアラインメントとして 2 を受け取るため、バイトのサイズが奇数の場合はコンパイラがパディングバイトを追加することがあります。

基本データ型整数型

コンパイラは、標準の C の基本データ型のすべてと追加のデータ型の両方をサポートします。

整数型概要

以下の表に、各整数型のサイズと範囲の一覧を示します。

データ型	サイズ	範囲	アラインメント
`bool`	8 ビット	0 ～ 1	1
`char`	8 ビット	0 ～ 255	1
`signed char`	8 ビット	-128 ～ 127	1
`unsigned char`	8 ビット	0 ～ 255	1
`signed short`	16 ビット	-32768 ～ 32767	2
`unsigned short`	16 ビット	0 ～ 65535	2
`signed int`	16 ビット	-32768 ～ 32767	2

表 31: 整数型

データ型	サイズ	範囲	アラインメント
unsigned int	16 ビット	0 ～ 65535	2
signed long	32 ビット	-2^{31} ～ $2^{31}-1$	2
unsigned long	32 ビット	0 ～ $2^{32}-1$	2
signed long long	64 ビット	-2^{63} ～ $2^{63}-1$	2
unsigned long long	64 ビット	0 ～ $2^{64}-1$	2

表 31: 整数型（続き）

符号付変数は、2 の補数フォーマットで表現されます。

BOOL 型

bool データ型は、C++ 言語のデフォルトでサポートされています。言語拡張を有効化し、stdbool.h ファイルをインクルードした場合は、bool 型を C ソースコードでも使用できます。この場合は、ブール値の false および true も使用可能になります。

ENUM 型

コンパイラは必要な最小の型を使用して、enum 定数を保持します。unsigned よりも signed が優先的に使用されます。

IAR システムズの言語拡張が有効化されている場合や、C++ においては、enum 手数および型を long、unsigned long、long long、unsigned long long 型にすることも可能です。

コンパイラで自動的に使用される型より大きな型を使用するには、enum 定数を十分に大きな値で定義します。次に例を示します。

```
/* enum での char 型の使用を無効化 */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

CHAR 型

char 型は、コンパイラのデフォルトでは符号なしですが、--char_is_signed コンパイラオプションを使用して符号付にすることが可能です。ただし、ライブラリは char 型は符号なしでコンパイルされています。

WCHAR_T 型

wchar_t 型は、サポートされるロケール間で設定された最大の拡張文字集合のすべてのメンバに対して個別のコードを表現できる値の範囲を持つ整数型です。

wchar_t 型は、C++ 言語のデフォルトでサポートされています。wchar_t 型を C ソースコードでも使用するには、ランタイムライブラリから stddef.h ファイルをインクルードする必要があります。

ビットフィールド

標準の C では、int、signed int と unsigned int を整数ビットフィールドの基本型として使用できます。標準の C++ および C では、コンパイラで言語拡張が有効になっている場合、任意の整数または列挙型を基本型として使用できます。単純な整数型（char、short、int など）が符号つきまたは符号なしのビットフィールドになるかどうかは、実装によって定義されます。

RL78 用 IAR C/C++ コンパイラでは、単純な整数型は符号なしとして処理されます。

式のビットフィールドは、int がビットフィールドのすべての値を表せる場合、int として扱われます。それ以外の場合は、ビットフィールドの基本型として処理されます。

各ビットフィールドは、最下位ビットから最上位ビットの順に、その基本型のコンテナに配置されます。最後のコンテナが同じ型で十分なビットがある場合、ビットフィールドはこのコンテナに配置されます。それ以外の場合は、新しいコンテナが割り当てられます。この割り当てスキームは分割された型のビットフィールド割当て方式として参照されます。

ディレクティブ #pragma bitfield=reversed を使用する場合、ビットフィールドは最上位ビットから最下位ビットの順に各コンテナに配置されます。328 ページの *bitfields* を参照してください。

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;
};
```

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。

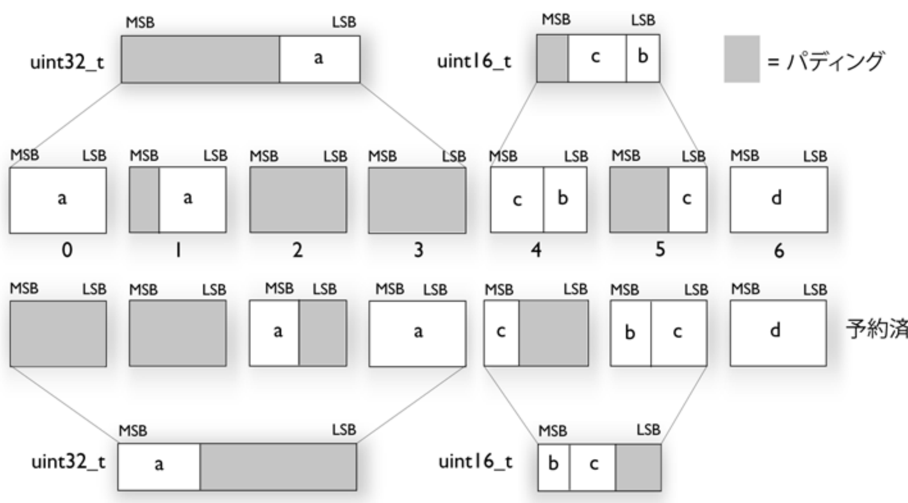
2 番目のビットフィールド b を配置するために、新しいコンテナがオフセット 4 に割当てられます。これは、ビットフィールドの型が前のビットフィールドと同じではないためです。b は、このコンテナの最下位の 3 ビットに配置されます。

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 `bitfield_example`：のレイアウトです。



基本データ型浮動小数点数型

RL78 用 IAR C/C++ コンパイラでは、浮動小数点の値を標準 IEEE 754 フォーマットで表現します。各浮動小数点数型のサイズを以下に示します。

タイプ	double=32 の場合のサイズ	double=64 の場合のサイズ
`float`	32 ビット	32 ビット
`double`	32 ビット（デフォルト）	64 ビット
`long double`	32 ビット	64 ビット

表 32: 浮動小数点数型

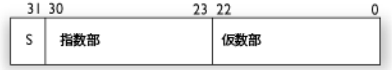
注：`double` と `long double` のサイズは `--double={32|64}` オプションに依存します（251 ページの *--double* を参照）。`long double` 型は、`double` と同じ精度を使用します。

浮動小数点環境

例外フラグはサポートしていません。feraiseexcept 関数では例外は引き起こされません。

32 ビット浮動小数点数フォーマット

32 ビット浮動小数点数を整数として表現すると、以下のようになります。



指数部は 8 ビット、仮数部は 23 ビットです。

値は以下のようになります。

(-1)^{S} * 2 $^{(指数部 -127)}$ * 1. 仮数部

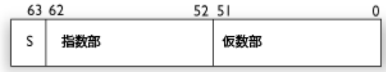
範囲は少なくとも以下のようになります。

±1.18E-38 から 3.39E+38

浮動小数点数の演算子 (+、-、*、/) は、約 7 桁です。

64 ビット浮動小数点数フォーマット

64 ビット浮動小数点数を整数として表現すると、以下のようになります。



指数部は 11 ビット、仮数部は 52 ビットです。

値は以下のようになります。

(-1)^{S} * 2 $^{(指数部 -1023)}$ * 1. 仮数部

範囲は少なくとも以下のようになります。

±2.23E-308 ～ ±1.79E+308

浮動小数点数の演算子 (+、-、*、/) は、約 15 桁です。

特殊な浮動小数点数の表現

特殊な浮動小数点数の表現を以下に挙げます。

- ゼロは、仮数部や指数部でゼロとして表現されます。符号ビットは、正または負のゼロを示します。
- 無限大は、指数部を最大値に、仮数部をゼロに設定することで表現されます。符号ビットは、正または負の無限大を示します。

- (NaN) が標準実装の場合、この項目を含める非数は指数部を最大値に設定し、仮数部をゼロに設定することで表現されます。符号ビットの値は無視されます。符号ビットの値は無視されます。最も高いビットを設定する必要があります。
- 非正規化数は、正規化数で表せる数値よりも小さい値を表現するときに使用します。この場合、値が小さくなるほど精度が低下するという欠点があります。指数部は 0 に設定され、数値が非正規化数であることを示します。ただし、数値は指数部が 1 である場合と同様に扱われます。正規化数とは異なり、非正規化数では仮数部の最上位ビット (MSB) に暗黙の 1 が設定されていません。非正規化数の値は次のようになります。

  $(-1)^{S} * 2^{(1-BIAS)} * 0.$仮数部

  BIAS は 127 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 16 ビットか 24 ビットです。以下の関数ポインタを使用できます。

キーワード	ポインタサイズ	コードモデルのデフォルト	説明
__near_func	2 バイト	Near	メモリの最初の 64 KB にアドレッシングできます。
__far_func	3 バイト *	Far	1MB のメモリ空間全体にアドレッシングできます。

表 33: 関数ポインタ

* アラインメントの制限のため、3 バイトのポインタはメモリで 4 バイトを使用します。

データポインタ

データポインタのサイズは常に 16 ビットか 24 ビットです。以下のデータポインタを使用できます。

キーワード	ポインタサイズ	アドレス範囲	説明
__near	2 バイト	0xF0000–0xFFFFF	最上位 64KB。
__far	3 バイト *	0x00000–0xFFFFF	1MB アドレス領域全体。

表 34: データポインタ

キーワード	ポインタサイズ	アドレス範囲	説明
`__huge`	3 バイト *	`0x00000–0xFFFFF`	1MB アドレス領域全体。64 Kb 以上のオブジェクトに使用します。

表 34: データポインタ（続き）

* アラインメントの制限のため、3 バイトのポインタはメモリで 4 バイトを使用します。

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
- 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
- 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
- `__near` ポインタから `__far` または `__huge` ポインタへのキャスティングは、`0xF0000` 拡張により実行されます
- `__far` または `__huge` ポインタから `__near` ポインタへのキャスティングは無効な処理です
- `__near_func` ポインタから `__far_func` ポインタへのキャスティングは、ゼロ拡張により実行されます
- `__far_func` ポインタから `__near_func` ポインタへのキャスティングは、無効な処理です
- `__far` ポインタから `__huge` ポインタへのキャスティングは同じビットパターンの結果になります
- `__huge` ポインタから `__far` ポインタへのキャスティングは、無効な処理です

size_t

`size_t` は `sizeof` 演算子の結果の符号なし整数型です。RL78 用 IAR C/C++ コンパイラでは、`size_t` で使用する型は `unsigned int` です。

ptrdiff_t

ptrdiff_t は 2 つのポインタを差し引いた結果の符号付きの整数型です。RL78 用 IAR C/C++ コンパイラでは、ptrdiff_t に使用する型は size_t 型の符号付きの整数型です。

デフォルトポインタ以外からポインタを差し引くことは、より小さいまた大きい整数型の結果になります。それぞれの場合で、この整数型は size_t 型の符号付きの整数型です。

注 : 2 つのポインタを差し引いた結果が負で、かなり大きいオブジェクトを作成する場合もあります。たとえば次のようになります。

```
char buff[60000];           /* ptrdiff_t が 16 ビットとします */
char *p1 = buff;            /* 符号付整数型 */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;   /* 結果 -5536 */
```

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。RL78 用 IAR C/C++ コンパイラでは、intptr_t で使用する型は signed long です。

uintptr_t

uintptr_t は、符号なしであることを除き、intptr_t と同じです。

構造体型

struct のメンバは、宣言された順に連続して格納されます。最初のメンバが最下位のメモリアドレスを持ちます。

構造体型のアライメント

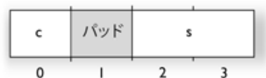
struct 型や union 型のアラインメントは、最高のアラインメント要件の数と同じになります。このアライメン要件は構造になるメンバにも適用されます。アラインメントされた構造体オブジェクトの配列が可能になるようにするには、struct のサイズをアライメントの偶数の倍数に調整します。

一般的なレイアウト

struct のメンバは、常に宣言で指定された順に割り当てられます。各メンバは、指定したアラインメント（オフセット）に従って struct 内に配置されます。

```
struct First
{
  char c;
  short s;
} s;
```

以下の図に、メモリでのレイアウトを示します。



構造体のアラインメントは 2 バイトです。また、short s に正しいアラインメントを与えるためにパディングバイトが挿入されている必要がありあります。

パック構造体型

#pragma pack ディレクティブは、構造体メンバのアラインメント要件を緩和するときに使用されます。これにより、構造体のレイアウトが変更されます。メンバは、宣言時と同じ順序で配置されますが、メンバ間のパッドエリアが少なくなることがあります。

正しくアラインメントされていないオブジェクトにアクセスする場合には、コードのサイズが大きくなり速度が低下します。そのような構造体へのアクセスが多数ある場合、パックされていない struct に正しい値を構成し、この struct にアクセスする方が通常は適しています。

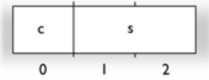
アラインメントが正しく設定されていないメンバへのポインタ作成および使用には、特別な注意も必要です。パックされた struct のアラインメントが正しく設定されていないメンバに直接アクセスする場合、コンパイラは、必要に応じて正しいコード（ただし、サイズが大きく低速）を出力します。しかし、アラインメントが正しく設定されていないメンバへのポインタを使用してそのメンバにアクセスする場合には、通常のコード（サイズが小さく高速）が使用されます。一般的なケースでは、これは機能しません。なぜなら、通常のコードは正しいアラインメントに依存することがあるからです。

以下の例では、パックされた構造体を宣言します。

```
#pragma pack(1)
struct S
{
  char c;
  short s;
};

#pragma pack()
```

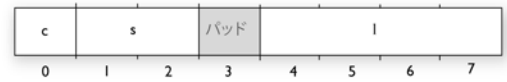
構造 s にはこのメモリレイアウトがあります。



次の例では、パックされていない別の構造体 S2 を宣言します。この構造体には、前の例で宣言した構造体 s が含まれます。

```
struct S2
{
  struct S s;
  long l;
};
```

構造 S2 にはこのメモリレイアウトがあります



構造体 s は、前の例で宣言したメモリレイアウト、サイズ、アラインメントを使用します。メンバ l のアラインメントは 2 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、204 ページの*構造体エレメントのアラインメント*を参照してください。

型修飾子

C 規格では、volatile と const は型修飾子です。

オブジェクトの VOLATILE 宣言

オブジェクト volatile を宣言することによって、オブジェクトの値がコンパイラの制限以上に変化する可能性があることがコンパイラに伝えられます。またコンパイラは、あらゆるアクセスに副作用があると想定する必要があります。よって、volatile オブジェクトへのすべてのアクセスは保持されなければなりません。

オブジェクトを volatile として宣言する主な理由は、以下の 3 つです。

- 共有アクセス : マルチタスク環境で、オブジェクトを複数のタスクで共有する場合
- トリガアクセス : メモリマップされた特殊機能レジスタ (SFR) のように、アクセス発生により影響が生じる場合

- 変更アクセス : コンパイラが認識できない方法で、オブジェクトの内容が変更される可能性がある場合

volatile オブジェクトへのアクセスの定義

C 規格では、抽象マシンが定義されています。これは、volatile 宣言したオブジェクトへのアクセスの動作を制御します。一般的に、抽象マシンに従うコンパイラの動作は、以下のとおりです。

- volatile として宣言されたオブジェクトへの各リード / ライトアクセスをアクセスと見なします
- アクセスは、オブジェクト単位になります。複合オブジェクト（配列、構造体、クラス、共用体など）へのアクセスの場合は、エレメント単位になります。次に例を示します

```
char volatile a;
a = 5;    /* ライトアクセス */
a += 6;  /* 最初はリードアクセスで次にライトアクセス */
```

- ビットフィールドへのアクセスは、その根底型へのアクセスとして処理されます
- const 修飾子を volatile オブジェクトに追加すると、オブジェクトへのライトアクセスが不可能になります。ただし、オブジェクトは C 規格で指定されたとおりに RAM 内に配置されます

ただし、これらの規則は大まかなもので、ハードウェア関連の要件には対応していません。RL78 用 IAR C/C++ コンパイラに固有の規則について、以下に説明します。

アクセス規則

RL78 用 IAR C/C++ コンパイラでは、volatile で宣言したオブジェクトは、以下の規則に従います。

- すべてのアクセスが実行されます
- すべてのアクセスは最後まで実行されます。すなわち、オブジェクト全体がアクセスされます
- すべてのアクセスは、抽象マシンでの場合と同一の順序で実行されます
- すべてのアクセスはアトミックアクセスになります。すなわち、割込みはできません

コンパイラは、すべての 8 ビットデータ型についてこれらの規則に従います。volatile と __sfr によって宣言された変数は、それらも __nobitaccess として宣言されている場合を除いて、ビットアクセスを使用してアクセスできます。

その他すべてのオブジェクト型については、すべてのアクセスが維持されるという規則だけが適用されます。

オブジェクト VOLATILE および CONST の宣言

volatile オブジェクトを const 宣言する場合、書き込み禁止になりますが、C 規格の仕様に従って RAM メモリに格納されます。

代わりにリードオンリーのメモリにオブジェクトを格納して、const volatile オブジェクトとしてアクセス可能にするには、__ro_placement 属性を使用してそれを宣言します。321 ページの *__ro_placement* を参照してください。

代わりにリードオンリーのメモリにオブジェクトを格納して、const volatile オブジェクトとしてアクセス可能にするには、以下のように変数を定義します。

```
const volatile int x @ "FLASH";
```

コンパイラは、リード / ライトセクション FLASH を生成します。このセクションは ROM に配置して、アプリケーション起動時に変数を手動で初期化するときに使用します。

これ以降は、イニシャライザは他の値とともにいつでも再度フラッシュすることができます。

オブジェクトの CONST 宣言

const 型修飾子は、データオブジェクト（直接またはポインタを使用してアクセス）がリードオンリーであることを示します。const として宣言したデータへのポインタは、定数と非定数の両方のオブジェクトを参照できます。可能な限り const として宣言したポインタを使用することをお勧めします。これにより、コンパイラによる生成コードの最適化が改善され、誤って修正したデータが原因でアプリケーションに障害が発生する危険性が低下します。

const で宣言され、far メモリ内に配置されている静的オブジェクトおよびグローバルオブジェクトは、ROM 内で割り当てられます。

const により宣言された saddr オブジェクトは、RAM 内で割り当てられ起動時にランタイムシステムによって初期化されます。const により宣言された near オブジェクトは、オプション --near_const_location に従って割り当てられます。

C++ では、ランタイムの初期化が必要なオブジェクトは ROM に配置できません。

C++ のデータ型

C++ では、通常の C データ型はすべて、前述の方法で表現されます。ただし、その型で拡張 C++ 機能を使用している場合は、データ表現に関する想定はできなくなります。たとえば、クラスメンバにアクセスするアセンブラコードを記述することはサポートされません。

拡張キーワード

- 拡張キーワードの一般的な構文規則
- 拡張キーワードの一覧
- 拡張キーワードの詳細

異なるメモリ領域のアドレス範囲について詳しくは、*セクションリファレンス*を参照してください。

拡張キーワードの一般的な構文規則

コンパイラは、RL78 マイクロコントローラ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、314 ページの*拡張キーワードの詳細*を参照してください。属性を使用してデータを修正する方法については、章*データ記憶*を参照してください。属性を使用してデータを修正する方法については、章*関数*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。
251 ページの *-e* を参照してください。

型属性

型属性は、関数の呼出し方法、またはデータオブジェクトのアクセス方法を定義します。すなわち、型属性を使用する場合には、関数またはデータオブジェクトの定義時と宣言時の両方で型属性を指定する必要があります。

型属性を明示的に宣言に配置するか、プラグマディレクティブ #pragma type_attribute を使用します。

型属性はさらに*メモリ型属性*と*汎用型属性*に分類できます。メモリ型属性はドキュメントのその他の部分の*メモリ属性*としてだけ参照されます。

メモリ属性

メモリ属性は、マイクロコントローラ内の特定の論理メモリまたは物理メモリに対応しています。

使用可能な*関数メモリ属性* :

__near_func, far_func, __callt

使用可能な*データ関数メモリ属性* :

__saddr__near, __far, __huge, __sfr.

データオブジェクト、関数、ポインタまたは C++ 参照の指す場所には、常にメモリ属性が 1 つあります。宣言中またはプラグマディレクティブ #pragma type_attribute によって明示的に属性が指定されていなければ、適切なデフォルトの属性が使用されます。間接参照レベルごとにメモリ属性を 1 つ指定できます。

汎用型属性

利用可能な*関数型属性*（関数の呼び出し方法に影響）:

__interrupt, __no_save, __v1_call, __v2_call

利用可能な*データ型属性* :

__no_bit_access

間接参照レベルごとに、必要な数の属性を指定できます。

データオブジェクトで使用される型属性の構文

型属性は構文規則の型修飾子 const と volatile とほぼ同じように使用します。次に例を示します。

```
__near int i;
int __near j;
```

i と j の両方は near メモリに配置されます。

const や volatile とは異なり、構造体メンバの宣言の場合を除いて、型属性は派生した型の型指定子の前に使用されるとき、型属性がオブジェクトまたは typedef 自体に適用されます。

```
int __near * p;        /* near メモリの整数 */
int * __near p;        /* near メモリのポインタ */
__near int * p;        /* near メモリのポインタ */
```

すべての場合で、メモリ属性が指定されていない場合、適切なデフォルトのメモリタイプが使用されます。それは使用するデータモデルによって異なります。

型定義を使用することはコードをより明確にできる場合があります。

```
typedef __near d16_int;
d16_int *q1;
```

d16_int near メモリの変数のための typedef です。変数 q1 はそのような整数を指し示すことができます。

#pragma type_attributes ディレクティブを使用して宣言の型属性を指定することもできます。プラグマディレクティブで指定した型属性は、データオブジェクトまたは制限される typedef に適用されます。

```
#pragma type_attribute=__near
int * q2;
```

q2 変数は near メモリに配置されます。

メモリ属性の他の使用例については、66 ページの*その他の例*を参照してください。

関数で使用される型属性の構文

関数の型属性に使用する構文は、データオブジェクトの型属性の構文とはわずかに異なります。関数の場合、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。

```
__interrupt void my_handler(void);
```

または

```
void (__interrupt 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=__interrupt
void my_handler(void);
```

関数ポインタを宣言するには、次の構文を使用します。

```
int (__near_func * fp) (double);
```

この宣言の後、関数ポインタ fp はは near メモリをポイントします。

より簡単な記憶領域の指定方法は、タイプ定義を使用することです。

```
typedef __near_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

#pragma type_attribute は typedef 宣伝とともに使用できます。

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。この規則に対する例外は、属性の記述に示されます。

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 :

  __no_alloc、__no_alloc16、__no_alloc_str、__no_alloc_str16、__no_init、__ro_placement
- 関数や変数に使用可能なオブジェクト属性 :

  location、@、__root、__weak
- 関数に使用可能なオブジェクト属性 :

  __intrinsic、__monitor、__noreturn、vector

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、206 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。vector の詳細については、347 ページの vector 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
__callt	関数の記憶領域の制御
__far	データオブジェクトの記憶領域の制御
__far_func	関数の記憶領域の制御
__huge	データオブジェクトの記憶領域の制御
__interrupt	割込み関数を指定
__intrinsic	コンパイラの内部使用専用に予約されています
__monitor	関数のアトミック実行を指定
__near	データオブジェクトの記憶領域の制御
__near_func	関数の記憶領域の制御
__no_alloc、__no_alloc16	実行ファイルで定数を使用可能にします
__no_alloc_str、__no_alloc_str16	実行ファイルで文字列リテラルを使用可能にします
__no_bit_access	オブジェクトの個々のビットにアクセスする際にビット命令の使用を抑制
__no_init	データオブジェクトを不揮発性メモリに配置します
__noreturn	関数がリターンしないことをコンパイラに通知します
__no_save	使用した workseg 領域を保存および復元しないよう関数を指定
__root	関数や変数を、未使用の場合でもオブジェクトに含めます

表 35: 拡張キーワードの一覧

拡張キーワード	説明
`__ro_placement`	`const volatile` データをリードオンリーメモリに配置します
`__saddr`	データオブジェクトの記憶領域の制御
`__sfr`	データオブジェクトの記憶領域の制御
`__v1_call`	V1 呼出し規約を指定します
`__v2_call`	V2 呼出し規約を指定します
`__weak`	外部的に弱いリンクになるようにシンボルを宣言します

表 35: 拡張キーワードの一覧（続き）

拡張キーワードの詳細

ここでは、それぞれの拡張キーワードについて詳細に説明します。

__callt

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

`__callt` メモリ属性は、選択されたコードモデルによって指定されたデフォルトの関数の記憶領域をオーバライドし、ここの関数が `CALLT` 命令を使用して呼び出されるように指定します。この関数は、どの XLINK セグメントにも配置できます。

記憶領域情報

- アドレス範囲：`0-0xFFFF` (64KB)
- 最大サイズ：65535 バイト
- ポインタサイズ：2 バイト

例

```
__callt void myfunction(void);
```

関連項目

254 ページの *--generate_callt_runtime_library_calls*。

__far

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

`__far` メモリ属性は、選択したデータモデルで与えられたデフォルトの変数の記憶領域をオーバライドし、個々の変数や定数を far メモリに配置します。

__far 属性を使用して、far メモリ内にあるオブジェクトをポイントするポインタを明示的に作成することもできます。

記憶領域情報

- アドレス範囲 : 0-0xFFFFF (1MB)
- 最大オブジェクトサイズ : 65535 バイト。オブジェクトは 64Kb をまたぐことはできません
- ポインタサイズ : 3 バイト。演算は、2 つの下位バイトでのみ実行されます。ただし、等式比較は 24 ビットのアドレス全体で実行されます

例

```
__far int x;
```

関連項目

62 ページの*メモリタイプ*。

__far_func

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

__far_func メモリ属性は、選択されたコードモデルによって指定されたデフォルトの関数の記憶領域をオーバライドし、個々の関数を far_func メモリに配置します。__far_func 属性を使用して、far_func メモリ内にあるオブジェクトをポイントするポインタを明示的に作成することもできます。

記憶領域情報

- アドレス範囲 : 0-0xFFFFF (1MB)
- 最大サイズ : 65535 バイト。オブジェクトは 64KB の境界を越えることはできません
- ポインタサイズ : 3 バイト

例

```
__far_func void myfunction(void);
```

関連項目

73 ページの*関数格納のためのコードモデルとメモリ属性*。

__huge

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

__huge メモリ属性は、選択したデータモデルによって指定された変数のデフォルトの記憶領域をオーバライドし、そのサイズにかかわらずメモリに個々の変数と定数を配置します。

また __huge 属性を使用して、サイズにかかわらずメモリ内にあるオブジェクトをポイントするポインタを明示的に作成することもできます。

記憶領域情報

- アドレス範囲：0-0xFFFFF (1MB)
- 最大オブジェクトサイズ：1 Mb
- ポインタサイズ：3 バイト

例

```
__huge int x;
```

関連項目

62 ページの*メモリタイプ*。

__interrupt

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

__interrupt キーワードは、割込み関数を指定します。1 つ以上の割込みベクタを指定するには、#pragma vector ディレクティブを使用します。割込みベクタの範囲は、使用するデバイスによって異なります。ベクタなしの割込み関数を定義することは可能ですが、その場合はコンパイラは割込みベクタテーブルにエントリを一切生成しません。

割込み関数はリターン型に void を持つことができ、パラメータは何も設定できません。

ヘッダファイル io*device*.h（*device* は選択したデバイス）には、事前定義した既存の割込みベクタ名が含まれます。

例

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

関連項目

75 ページの*割込み関数*、および 405 ページの *.intvec*。

__intrinsic

説明

__intrinsic キーワードは、コンパイラでの内部使用専用に予約されています。

__monitor

構文

313 ページの*オブジェクト属性の構文*を参照してください。

説明

`__monitor` キーワードは、関数の実行中に割込みを無効にします。これによって、複数のプロセスによってリソースへのアクセスを制御するセマフォでの処理など、アトミック処理が可能になります。`__monitor` キーワードで宣言された関数は、他のあらゆる関数と他のすべての面で同等です。

例

```
__monitor int get_lock(void);
```

関連項目

76 ページの*モニタ関数*。関連の組込み関数の詳細については、350 ページの *__disable_interrupt*、350 ページの *__enable_interrupt*、350 ページの *__get_interrupt_state*、352 ページの *__set_interrupt_state* をそれぞれ参照してください。

__near

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

`__near` メモリ属性は、選択したデータモデルで与えられたデフォルトの変数の記憶領域をオーバライドし、個々の変数や定数を near メモリに配置します。

`__near` 属性を使用して、near メモリ内にあるオブジェクトをポイントするポインタを明示的に作成することもできます。

記憶領域情報

- アドレス範囲：`0xF0000-0xFFFFF`
- 最大オブジェクトサイズ：65535 バイト
- ポインタサイズ：2 バイト

例

```
__near int x;
```

関連項目

62 ページの*メモリタイプ*。

__near_func

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

`__near_func` メモリ属性は、選択されたコードモデルによって指定されたデフォルトの関数の記憶領域をオーバライドし、個々の関数を near_func メモリ

に配置します。__near_func 属性を使用して、near_func メモリ内にあるオブジェクトをポイントするポインタを明示的に作成することもできます。

記憶領域情報

- アドレス範囲：0-0xFFFF (64KB)
- 最大サイズ：65535 バイト
- ポインタサイズ：2 バイト

例

```
__near_func void myfunction(void);
```

関連項目

73 ページの*関数格納のためのコードモデルとメモリ属性*。

__no_alloc、__no_alloc16

構文

313 ページの*オブジェクト属性の構文*を参照してください。

説明

定数で __no_alloc または __no_alloc16 オブジェクト属性を使用すると、リンクされたアプリケーション内でスペースをとることなく、実行ファイルでその定数が使用可能になります。

このような定数の内容にはアプリケーションからはアクセスできません。そのアドレス、つまり定数のセクションに対する整数オフセットを取得することはできます。__no_alloc を使用する場合はオフセットの型が unsigned long となり、__no_alloc16 を使用する場合は unsigned short となります。

例

```
__no_alloc const struct MyData my_data @ "XXX" = {...};
```

関連項目

318 ページの *__no_alloc_str、__no_alloc_str16*。

__no_alloc_str、__no_alloc_str16

構文

`__no_alloc_str(`*`string_literal`*` @ `*`section`*`)`

および

`__no_alloc_str16(`*`string_literal`*` @ `*`section`*`)`

説明

string_literal	実行ファイルで使用可能にする文字列リテラル。
section	文字列リテラルを配置するセクション名。

説明

定数で __no_alloc_str または __no_alloc_str16 演算子を使用すると、リンクされたアプリケーション内でスペースをとることなく、実行ファイルで文字列リテラルが使用可能になります。

この式の値は、セクション内の文字列リテラルのオフセットです。__no_alloc_str の場合、オフセットの型は unsigned long です。__no_alloc_str16 の場合、オフセットの型は unsigned short です。

例

```
#define MYSEG "YYY"
#define X(str) __no_alloc_str(str @ MYSEG)

extern void dbg_printf(unsigned long fmt, ...)

#define DBGPRINTF(fmt, ...) dbg_printf(X(fmt), __VA_ARGS__)

void
foo(int i, double d)
{
  DBGPRINTF("i の値 : %d、d の値 : %f",i,d);
}
```

使用するデバッガとランタイムサポートによっては、これによってホストコンピュータ上でトレース出力が生成されることがあります。外部のプラグインモジュールを使用しない限り、C-SPY ではこうしたランタイムサポートはない点に注意してください。

関連項目

318 ページの *__no_alloc*、*__no_alloc16*。

__no_bit_access

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

__no_bit_access として宣言されたデータオブジェクトの場合、ビット命令の使用は抑制され、オブジェクトの個々のビットがアクセスされます。

__no_init

構文

313 ページの*オブジェクト属性の構文*を参照してください。

説明

__no_init キーワードは、データオブジェクトを不揮発性メモリに配置する場合に使用します。すなわち、変数の初期化（起動時など）が行われなくなります。

例

```
__no_init int myarray[10];
```

関連項目

221 ページの*非初期化変数*、383 ページの *do not initialize ディレクティブ*。

__noreturn

構文

313 ページの*オブジェクト属性の構文*を参照してください。

説明

`__noreturn` キーワードは、関数がリターンしないことをコンパイラに通知するために使用できます。このような関数でこのキーワードを使用する場合、コンパイラでは、さらに効率的に最適化が可能です。リターンしない関数の例としては、`abort` や `exit` などがあります。

注 : 最適化レベル「中」と「高」では、現在の関数がリターン値を返さないと判断された場合は、`__noreturn` キーワードにより、不正確なコールスタックデバッグ情報が生成されることがあります。

例

```
__noreturn void terminate(void);
```

__no_save

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

`__no_save` として宣言された関数は、使用した workseg 領域の保存や復元を行いません。

関数は通常、使用された `WRKSEG` セグメントの領域をスタック上に保存します。`__no_save` キーワードは、`--workseg_area` オブジェクトを使用するときに、この情報を保存しないよう関数に指示します。

`--workseg_area` オブジェクトを使用しない限り、このキーワードは効果がありません。

例

```
__no_save void myfunction(void);
```

__root

構文

313 ページの*オブジェクト属性の構文*を参照してください。

説明

`__root` 属性を持つ関数や変数は、そのモジュールが含まれる場合、アプリケーション内で参照されるかどうかに関わらず保持されます。プログラムモジュールは常に含まれ、ライブラリモジュールは必要に応じて含まれます。

例

```
__root int myarray[10];
```

関連項目

ルートシンボルとその保持方法について詳しくは、99 ページの*シンボルおよびセクションの保持*を参照してください。

__ro_placement

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

これはオブジェクト属性ですが、データオブジェクトで使用可能な型属性の一般的な構文規則に従います。型属性と同じように、データオブジェクトが定義されているときと宣言されるときの両方で指定する必要があります。

説明

`__ro_placement` 属性は、型修飾子 `const` および `volatile` と組み合わせて使用し、変数をリードオンリー（コード）メモリに配置するようコンパイラに指示します。これはデフォルトのメモリタイプの属性を変更します。

例

```
__ro_placement const volatile int x = 10;
```

__saddr

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

`__saddr` メモリ属性は、選択したデータモデルで与えられたデフォルトの変数の記憶領域をオーバライドし、個々の変数や定数を saddr メモリに配置します。

`__saddr` 属性を使用して、saddr メモリ内にあるオブジェクトをポイントするポインタを明示的に作成することもできます。

記憶領域情報

- アドレス範囲：`0xFFE20–0xFFF1F`
- 最大オブジェクトサイズ：256 バイト
- ポインタサイズ：2 バイト

例

```
__saddr int x;
```

関連項目

62 ページの*メモリタイプ*。

__sfr

構文

310 ページの*データオブジェクトで使用される型属性の構文*を参照してください。

説明

`__sfr` メモリ属性は、選択したデータモデルで与えられたデフォルトの変数の記憶領域をオーバライドし、個々の変数や定数をメモリに配置します。

記憶領域情報

- アドレス範囲：`0xFFF00`–`0xFFFFF`
- 最大オブジェクトサイズ：通常、256 バイト（デバイスによって異なる）
- ポインタサイズ：なし

例

```
__sfr int x;
```

関連項目

62 ページの*メモリタイプ*。

__v1_call

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

`__v1_call` キーワードにより、関数についてコンパイラでデフォルトの呼出し規約ではなく、RL78 バージョン 1.x 用 IAR C/C++ コンパイラの呼出し規約を使用するよう指示します。RL78 バージョン 1.x 用 IAR C/C++ コンパイラのために記述されたコードには、下位互換性のために V1 呼び出し規約が必要です。

注 : バージョン 1.x のコンパイラから移行する場合、スタックは使用する呼出し規約に関わらず、呼び出された関数ではなく、呼び出した関数によりクリーンされる点に注意してください。スタックのパラメータを受け入れるアセンブラ関数は、アップデートしてから使用する必要があります。

例

```
__v1_call int func(int arg1, double arg2);
```

関連項目

150 ページの*呼出し規約*。

__v2_call

構文

311 ページの*関数で使用される型属性の構文*を参照してください。

説明

`__v2_call` キーワードは、コンパイラで関数に対して V2 呼出し規約を使用するよう指定します。これはデフォルトの呼出し規約です。V2 呼出し規約は

RL78 ABI に適合しており、他のベンダのツールにより生成されたコードをリンクする際に必要です。

例

```
__v2_call int func(int arg1, double arg2);
```

関連項目

150 ページの*呼出し規約*。

__weak

構文

313 ページの*オブジェクト属性の構文*を参照してください。

説明

__weak オブジェクト属性をシンボルの外部宣言に使用することにより、モジュール内でのそのシンボルへのすべての参照が弱参照になります。

シンボルの公開されている定義上で __weak オブジェクト属性を使用すると、その定義は弱くなります。

リンカは、シンボルへの弱い参照を満たすためだけにライブラリからモジュールをインクルードすることはなく、弱い参照の定義の不足がエラーにつながることもありません。定義がインクルードされない場合、オブジェクトのアドレスはゼロになります。

リンク処理の際、シンボルは弱い定義を必要な数だけと、最大で弱くない定義を 1 つ持つことができます。シンボルが必要な場合は、弱くない定義が 1 つあり、この定義が使用されます。弱くない定義がない場合は、弱い定義のいずれかが使用されます。

例

```
extern __weak int foo; /* 弱い参照 */

__weak void bar(void)  /* 弱い定義 */
{
  /* インクルードされた場合は foo をインクリメント */
  if (&foo != 0)
    ++foo;
}
```

プラグマディレクティブ

- プラグマディレクティブの一覧
- プラグマディレクティブの詳細

プラグマディレクティブの一覧

#pragma ディレクティブは、C 規格によって定義されたものであり、ベンダ固有の拡張の使用方法を規定することにより、ソースコードの移植性を維持するための仕組みです。

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

プラグマディレクティブは、コンパイラでは常に有効になっています。

以下の表には、`#pragma` プリプロセッサディレクティブまたは `_Pragma()` プリプロセッサ演算子で使用可能なコンパイラのプラグマディレクティブの一覧を示します。

プラグマディレクティブ	説明
`bank`	直後に続く割込み関数のレジスタバンク番号を指定します。
`basic_template_matching`	テンプレート関数がメモリ属性を完全に認識するようにします。
`bitfields`	ビットフィールドメンバの順序を設定します。
`constseg`	定数変数を指定のセクションに配置します。
`data_alignment`	変数のアラインメントを高く（より厳密に）します。
`dataseg`	変数を指定のセクションに配置します。
`default_function_attributes`	関数の宣言および定義に対するデフォルトの型とオブジェクトを設定します。
`default_variable_attributes`	変数の宣言および定義に対するデフォルトの型とオブジェクトを設定します。
`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。

表 36: 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
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。
location	変数の絶対アドレスを指定します。または、指定したセクションに関数や変数のグループを配置します。
message	メッセージを出力します。
no_workseg	直後の関数が workseg 領域を使用しないように指定します。
object_attribute	変数または関数の宣言もしくは定義にオブジェクト属性を追加します。
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。
public_equ	パブリックアセンブラのラベルを定義し、それに値を割り当てます。
required	別のシンボルによって必要とされるシンボルが確実にリンク出力に含まれるようにします。
rtmodel	ランタイムモデル属性をモジュールに追加します。
__scanf_args	scanf スタイルフォーマット文字列の関数の呼出しに使用されている引数が正しいかどうかを検証します。
section	組込み関数で使用されるセクション名を宣言します。
segment	このディレクティブは #pragma セクションのエイリアスです。

表 36: プラグマディレクティブの一覧（続き）

プラグマディレクティブ	説明
STDC 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。
unroll	ループを展開。
vector	割込みベクタまたはトラップ関数を指定します。
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。
type_attribute	宣言または定義に型属性を追加します。

表 36: プラグマディレクティブの一覧（続き）

注 : 移植上の理由については、「452 ページの*認識されているプラグマディレクティブ (6.10.6)*」を参照してください。

プラグマディレクティブの詳細

ここでは、各プラグマディレクティブの詳細を説明します。

bank

構文

`#pragma bank=`*`bank_number`*

パラメータ

bank_number	レジスタバンク 0 から 3 を記述する定数の整数式。

説明

このプログラムディレクティブは、関数の実行を開始する前に別のレジスタバンクに切り替える割込み関数を宣言する場合に使用します。`#pragma bank` ディレクティブは、直後に続く割込み関数のレジスタバンク番号（0 から 3）を制御します。

basic_template_matching

構文

`#pragma basic_template_matching`

説明

このプラグマディレクティブはテンプレート関数の宣言の前で使用して、関数がメモリ属性を完全に認識するようにます。その方が都合が良い場合のみ

使用します。そのテンプレート関数によって、修正なしに照合が行われます（186 ページの*テンプレートおよびデータメモリ属性*を参照）。

例

```
#pragma basic_template_matching
template<typename T> void fun(T *);

fun((int __near *) 0); /* テンプレートパラメータ T が
                          int __near になります */
```

bitfields

構文

```
#pragma bitfields={reversed|default}
```

パラメータ

reversed	ビットフィールドメンバは、最上位ビットから最下位ビットの順に配置されます。
default	ビットフィールドメンバは、最下位ビットから最上位ビットの順に配置されます。

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順序を制御する場合に使用します。

例

```
#pragma bitfields=reversed
/* 逆順のビットフィールドを使用する構造体 */
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

298 ページの *ビットフィールド*。

constseg

構文

```
#pragma constseg=[__memoryattribute ]{SECTION_NAME|default}
```

パラメータ

__memoryattribute	セクションを配置するメモリを記述するオプションのメモリ属性。指定しない場合、デフォルトのメモリが使用されます。

SECTION_NAME	ユーザ定義のセクション名。コンパイラやリンカで使用される定義済のセクション名は指定できません。
`default`	定数のデフォルトのセクションを使用します。

説明

このプラグマディレクティブは、指定したセクションに定数を配置するときに使用します。コンパイラやリンカで使用される定義済のセクション名をセクション名として指定することはできません。この設定は、`#pragma constseg=default` ディレクティブを使用して再び無効にするまでそのまま有効になります。

例

```
#pragma constseg=__far MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

data_alignment

構文

`#pragma data_alignment=`*`expression`*

パラメータ

expression	定数。2 の累乗（1、2、4 など）を指定する必要があります。

説明

このプラグマディレクティブは、変数に与える開始アドレスのアラインメントを通常よりも高く（より厳密に）する場合に使用します。このディレクティブは、静的 / 自動変数に対して使用できます。

このディレクティブを自動変数に対して使用する場合は、各関数で指定可能なアラインメントに上限が設けられます。この上限は、使用する呼出し規約によって決定されます。

注 : 通常、変数のサイズは、そのアラインメントの倍数です。`data_alignment` ディレクティブは、変数の開始アドレスのみに影響し、サイズには影響しません。そのため、サイズがアラインメントの倍数ではない状況に使用できます。

dataseg

構文

```
#pragma dataseg=[__memoryattribute]{SECTION_NAME|default}
```

パラメータ

__memoryattribute	セクションを配置するメモリを記述するオプションのメモリ属性。指定しない場合、デフォルトのメモリが使用されます。
SECTION_NAME	ユーザ定義のセクション名。コンパイラやリンカで使用される定義済のセクション名は指定できません。
default	デフォルトのセクションを使用します。

説明

このプラグマディレクティブは、指定したセクションに変数を配置するときに使用します。コンパイラやリンカで使用される定義済のセクション名をセクション名として指定することはできません。変数は起動時に初期化されないため、イニシャライザを持つことはできません。つまり、`__no_init` として宣言される必要があります。この設定は、`#pragma dataseg=default` ディレクティブを使用して再び無効にするまで、そのまま有効になります。

例

```
#pragma dataseg=__far MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

default_function_attributes

構文

```
#pragma default_function_attributes=[ attribute...]
```

attribute には以下を使用できます。

type_attribute
object_attribute
@ *section_name*

パラメータ

type_attribute	309 ページの*型属性*を参照してください。
object_attribute	312 ページの*オブジェクト属性*を参照してください。
@ *section_name*	209 ページの*データと関数のセクションへの配置*を参照してください。

説明

このプラグマディレクティブを使用して、関数の宣言と定義について、デフォルトのセクション配置、型属性、オブジェクト属性を設定します。デ

フォルト設定は、他の方法で型属性やオブジェクト属性、位置を指定しない宣言および定義に対してのみ使用されます。

属性なしに default_function_attributes プラグマディレクティブを指定すると、デフォルト値が関数の宣言および定義に適用されていない初期の状態が復元されます。

例

```
/* 以下の関数をセクション MYSEC" に配置 */
#pragma default_function_attributes = @ "MYSEC"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1;
/* 関数の MYSEC への配置を停止 */
#pragma default_function_attributes =
```

は以下と同じ効果があります。

```
int fun1(int x) @ "MYSEC" { return x + 1; }
int fun2(int x) @ "MYSEC" { return x - 1; }
```

関連項目

336 ページの *location*

338 ページの *object_attribute*

345 ページの *type_attribute*

default_variable_attributes

構文

```
#pragma default_variable_attributes=[ attribute...]
```

attribute には以下を使用できます。

type_attribute
object_attribute
@ *section_name*

パラメータ

type_attribute	309 ページの*型属性*を参照してください。
object_attributes	312 ページの*オブジェクト属性*を参照してください。
@ *section_name*	209 ページの*データと関数のセクションへの配置*を参照してください。

説明

このプラグマディレクティブを使用して、静的記憶寿命変数の宣言と定義について、デフォルトのセクション配置、型属性、オブジェクト属性を設定します。デフォルト設定は、他の方法で型属性やオブジェクト属性、位置を指定しない宣言および定義に対してのみ使用されます。

属性なしに default_variable_attributes プラグマディレクティブを指定すると、静的記憶寿命変数にそのようなデフォルト値が適用されていない初期の状態が復元されます。

例

```
/* 以下の変数をセクション MYSEC" に配置 */
#pragma default_variable_attributes = @ "MYSEC"
int var1 = 42;
int var2 = 17;
/* 変数の MYSEC への配置を停止 */
#pragma default_variable_attributes =
```

は以下と同じ効果があります。

```
int var1 @ "MYSEC" = 42;
int var2 @ "MYSEC" = 17;
```

関連項目

336 ページの *location*

338 ページの *object_attribute*

345 ページの *type_attribute*

diag_default

構文

```
#pragma diag_default=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）。

説明

このプラグマディレクティブは、タグで指定される診断メッセージの重要度を変更する場合に使用します。デフォルトの重要度に戻したり、オプション --diag_error、--diag_remark、--diag_suppress、--diag_warnings を使用してコマンドラインで定義した重要度に変更することができます。

関連項目

231 ページの*診断*。

diag_error

構文

```
#pragma diag_error=tag[,tag,...]
```

パラメータ

tag	診断メッセージの番号（たとえば、メッセージ番号 Pe177 など）。

説明

このプラグマディレクティブは、指定した診断メッセージの重要度を error に変更する場合に使用します。

関連項目

231 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

231 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

231 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

231 ページの*診断*。

error

構文

```
#pragma error message
```

パラメータ

message	エラーメッセージを表す文字列。

説明

このプラグマディレクティブを使用して、解析時にエラーメッセージを出力します。このメカニズムは、プリプロセッサディレクティブ `#error` とは異なります。`#pragma error` ディレクティブは、`_Pragma` 形式のディレクティブを使用してプリプロセッサマクロにインクルードできるため、マクロが使用されるときにだけエラーとなるためです。

例

```
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error¥"Foo is not available¥"")
#endif
```

`FOO_AVAILABLE` がゼロの場合、`FOO` マクロが実際のソースコードで使用される場合にエラーが警告されます。

include_alias

構文

```
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
```

パラメータ

orig_header	エイリアスを作成するヘッダファイルの名前。
subst_header	元のヘッダファイルのエイリアス。

説明

このプラグマディレクティブは、ヘッダファイルのエイリアスを提供する場合に使用します。これは、あるヘッダファイルを他のヘッダファイルで代用する場合や、相対ファイルへの絶対パスを指定する場合に便利です。

このプラグマディレクティブは、対応する `#include` ディレクティブの前に記述する必要があります。また、`subst_header` は、対応する `#include` ディレクティブに正確に一致する必要があります。

例

```
#pragma include_alias (<stdio.h> , <C:¥MyHeaders¥stdio.h>)
#include <stdio.h>
```

この例では、相対ファイル `stdio.h` を、指定パスにあるファイルで代用します。

関連項目

227 ページの*インクルードファイル検索手順*。

inline

構文

```
#pragma inline[=forced|=never]
```

パラメータ

パラメータはありません	`inline` キーワードと同じ結果になります。
`forced`	コンパイラのヒューリスティックを無効にし、強制的にインライン化します。
`never`	コンパイラのヒューリスティックを無効にして、関数がインライン化されないようにします。

説明

`#pragma inline` を使用して、ディレクティブの直後に定義された関数を C++ のインライン動作に従ってインライン化するようコンパイラに指示します。

`#pragma inline=forced` を指定すると、常に定義された関数がインライン化されます。再帰など何らかの理由でコンパイラが関数をインライン化できない場合、ワーニングメッセージが出力されます。

インライン化は通常、最適化レベル「高」でのみ実行されます。`#pragma inline=forced` を指定すると、最適化レベル「中」でも関数のインライン化が有効になります。

関連項目

79 ページの*インライン関数*。

language

構文

```
#pragma language={extended|default|save|restore}
```

パラメータ

`extended`	プラグマディレクティブを最初に使用してからその後も、IAR システムズの言語拡張を有効にします。
`default`	プラグマディレクティブを最初に使用してからその後も、IAR システムズの言語拡張の設定をコンパイラオプションで指定された状態に復元します。
`save|restore`	ソースコードの一部について、IAR システムズの言語拡張をそれぞれ保存、復元します。 `save` を使用するたびに、`#include` ディレクティブが途中で割込まないように、同じファイル内の一致する `restore` を続いて使用する必要があります。

説明

このプラグマディレクティブを使用して、言語拡張の使用を制御します。

例

IAR システムの拡張を有効にしてコンパイルする必要があるファイルの先頭で：

```
#pragma language=extended
/* ファイルの残りの部分 */
```

IAR システムの拡張を有効にしてコンパイルする必要があるソースコードの特定部分の周辺で、シーケンス前の状態が使用中のコンパイラオプションで指定されたものと同じとは考えられない場合：

```
#pragma language=save
#pragma language=extended
/* ソースコードの一部 */
#pragma language=restore
```

関連項目

251 ページの *-e*、269 ページの *--strict*。

location

構文

```
#pragma location={address|NAME}
```

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。

NAME	ユーザ定義のセクション名。コンパイラやリンカで使用される定義済のセクション名は指定できません。

説明

このプラグマディレクティブは、このプラグマディレクティブの直後に宣言されたグローバル変数または静的変数の位置（絶対アドレス）を指定する場合に使用します。セクション別の方法として、このプラグマディレクティブの後に宣言された変数または関数を配置するためのを文字列で指定することもできます。通常は異なるセクションにある変数（たとえば、__no_init として宣言される変数と、const として宣言される変数）を、同じ名前のセクションに配置しないでください。

例

```
#pragma location=0xFFF20
__no_init volatile char __sfr PORT1;
/* PORT1 をアドレス 0xFFF20 に配置 */

#pragma segment="FLASH"
#pragma location="FLASH"
__no_init char PORT2; /* PORT2 はセグメント FLASH に配置 */

/* 対応メカニズムを利用したより良い方法 */
#define FLASH _Pragma("location=¥"FLASH¥"")

FLASH __no_init int i; /* i は FLASH セグメントに配置 */
```

関連項目

206 ページの*データと関数のメモリ配置制御*、97 ページの*独自のセクションの宣言および配置*。

message

構文

```
#pragma message(message)
```

パラメータ

message	標準出力ストリームに転送するメッセージ。

説明

このプラグマディレクティブは、コンパイラでファイルのコンパイル時にメッセージを標準出力ストリームに出力する場合に使用します。

例

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

no_workseg

構文

```
#pragma no_workseg
```

説明

このプラグマディレクティブを関数の宣言の前に使用して、その関数が --workseg_area コンパイラオプションで作成されたレジスタ変数作業領域を使用しないようにします。

関連項目

70 ページの*ショートアドレス作業エリア*。

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、312 ページの*オブジェクト属性*を参照してください。

説明

このプラグマディレクティブを使用して、1 つまたは複数の IAR 固有のオブジェクト属性を変数や関数の宣言あるいは定義に追加します。オブジェクト属性は、実際の変数や関数に影響しますが、その型には影響しません。変数や関数を定義する際、定義も含めたあらゆる宣言のオブジェクト属性の共用体が使用されます。

例

```
#pragma object_attribute=__no_init
char bar;
```

は、以下の文と等価です。

```
__no_init char bar;
```

関連項目

309 ページの*拡張キーワードの一般的な構文規則*。

optimize

構文

```
#pragma optimize=[goal][level][no_optimization...]
```

パラメータ

goal	以下から選択します。 size（サイズを重視して最適化） balance（速度とサイズのバランスを最適化） speed（速度を重視して最適化） no_size_constraints: 速度を重視して最適化しますが、コードサイズの拡張のために通常の制限を緩和します。
level	最適化レベルを [none]、[low]、[mid]、[high] から指定します。
no_*optimization*	1 つまたは複数の最適化を無効にします。以下から選択してください。 no_code_motion（コード移動を無効化） no_cse（共通部分式除去を無効化） no_inline（関数のインライン化を無効化） no_tbaa（型ベースエイリアス解析を無効化） no_unroll（ループ展開を無効化）

説明

このプラグマディレクティブは、最適化レベルを下げる場合や、特定の最適化を無効化する場合に使用します。このプラグマディレクティブは、ディレクティブ直後の関数にのみ影響します。

パラメータ size、balanced、speed、no_size_constraints は、最適化レベル［高］でのみ効果があり、速度とサイズを同時に最適化することはできないため、これらのどれか 1 つだけを使用できます。また、このプラグマディレクティブにプリプロセッサマクロを埋め込むことはできません。埋め込まれたマクロは、プリプロセッサでは展開されません。

注：#pragma optimize ディレクティブを使用して指定した最適化レベルが、コンパイラオプションを使用して指定した最適化レベルよりも高い場合、このプラグマディレクティブは無視されます。

例

```
#pragma optimize=speed
int SmallAndUsedOften()
{
  /* 何らかの処理 */
}

#pragma optimize=size
int BigAndSeldomUsed()
{
  /* 何らかの処理 */
}
```

関連項目

213 ページの*変換の微調整*。

pack

構文

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[,name] [,n])
```

パラメータ

n	次のいずれかの中からオプションの構造体アラインメントを設定します：1、2、4、8、16
空白のリスト	構造体アラインメントをデフォルトに復元します
push	一時的な構造体アラインメントを設定します
pop	構造体アラインメントを一時的にプッシュされたアラインメントから復元します
name	プッシュまたはポップされたオプションのアラインメントラベル

説明

このプラグマディレクティブは、struct および union メンバの最大アラインメントを指定する場合に使用します。

#pragma pack ディレクティブは、プラグマディレクティブに続く構造の宣言を次の #pragma pack またはコンパイルユニットの最後に適用します。

注：この結果、コードが大幅に大きくなり、構造体のメンバにアクセスする際の速度が大幅に低下する可能性があります。

関連項目

303 ページの*構造体型*。

__printf_args

構文

```
#pragma __printf_args
```

説明

このプラグマディレクティブは、printf スタイルフォーマット文字列の関数に使用します。コンパイラは、この関数への任意の呼出しに対して、各変換指定子（たとえば %d）の引数が構文的に正しいかどうかを検証します。

例

```
#pragma __printf_args
int printf(char const *,...);

void PrintNumbers(unsigned short x)
{
  printf("%d", x);  /* コンパイラは x が整数かどうかチェックする */
}
```

public_equ

構文

```
#pragma public_equ="symbol",value
```

パラメータ

symbol	定義するアセンブラシンボルの名前（文字列）。
value	定義済みのアセンブラシンボルの値（整数の定数式）。

説明

このプラグマディレクティブを使用してパブリックのアセンブララベルを定義し、それに値を割り当てます。

例

```
#pragma public_equ="MY_SYMBOL",0x123456
```

関連項目

267 ページの *--public_equ*。

required

構文

```
#pragma required=symbol
```

パラメータ

symbol	静的にリンクされた関数または変数。

説明

このプラグマディレクティブは、2 番目のシンボルによって必要とされるシンボルがリンク出力に必ず含まれるようにする場合に使用します。このディレクティブは、2 番目のシンボルの直前に置く必要があります。

このディレクティブは、変数がその格納場所のセクション経由で間接的に参照されるだけの場合など、シンボルが必須かどうかがアプリケーションではわからない場合に使用します。

例

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
  /* 何らかの処理 */
}
```

著作権の文字列がアプリケーションで使用されない場合でも、この文字列がリンカによって含められ、出力に現れます。

rtmodel

構文

```
#pragma rtmodel="key","value"
```

パラメータ

"*key*" ランタイムモデル属性を指定するテキスト文字列。

"*value*" ランタイムモデル属性の値を指定するテキスト文字列特殊値 * を使用すると、属性が未定義である場合と等価になります。

説明

このプラグマディレクティブは、ランタイムモデル属性をモジュールに追加する場合に使用します。この属性を使用して、モジュール間の整合性のチェックをリンカで行えます。

このプラグマディレクティブは、モジュール間の整合性を確保するために使用できます。一緒にリンクされ、同一のランタイムモジュール属性のキーを定義するすべてのモジュールは、そのキーに対応する値が同一であるか、特殊な * という値を持つ必要があります。ただし、この値を使用することで、モジュールがランタイムモデルに対応していることを明示できます。

1 つのモジュールで複数のランタイムモデルを定義できます。

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

例

```
#pragma rtmodel="I2C","ENABLED"
```

リンカは、この定義を含むモジュールが、対応するランタイムモデル属性が定義されていないモジュールにリンクされている場合はエラーを生成します。

関連項目

140 ページの*モジュールの整合性チェック*。

__scanf_args

構文

```
#pragma __scanf_args
```

説明

このプラグマディレクティブは、scanf スタイルフォーマット文字列の関数に使用します。コンパイラは、この関数への任意の呼出しに対して、各変換指定子（たとえば `%d`）の引数が構文的に正しいかどうかを検証します。

例

```
#pragma __scanf_args
int scanf(char const *,...);

int GetNumber()
{
  int nr;
  scanf("%d", &nr);  /* コンパイラが、
                        引数が整数への
                        ポインタであることをチェック */

  return nr;
}
```

section

構文

`#pragma section="`*`NAME`*`" [`*`__memoryattribute`*`]`

エイリアス

`#pragma segment="`*`NAME`*`" [`*`__memoryattribute`*`]`

パラメータ

NAME	セクションの名前。
__memoryattribute	セクションを配置するメモリを記述するオプションのメモリ属性。指定しない場合、デフォルトのメモリが使用されます。

説明

このプラグマディレクティブを使用して、セクション演算子の `__section_begin`、`__section_end`、`__section_size` で使用可能なセク

ション名を定義します。特定のセクションのセクション宣言のメモリ型とアラインメントは、すべて同じでなければなりません。

`__memoryattribute` パラメータは、セクション演算子の `__section_begin`、`__section_end`、`__section_size` とともに使用した場合のみ、適用可能です。

オプションのメモリ属性を使用する場合、セクション operators `__section_begin` と `__section_end` のリターン型は次のとおりです。

`void __memoryattribute *.`

注 : 変数や関数を特定のセクションに配置するには、`#pragma location` ディレクティブまたは `@` 演算子を使用します。

例

```
#pragma section="MYNEAR" __near
```

関連項目

171 ページの*専用セクション演算子*セクションおよびセグメントパートについては、「*アプリケーションのリンク*」を参照してください。

STDC CX_LIMITED_RANGE

構文

```
#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}
```

パラメータ

ON	通常の複雑な数式を使用できます。
OFF	通常の複雑な数式は使用できません。
DEFAULT	デフォルトの動作を設定します。つまり `OFF` です。

説明

このプラグマディレクティブは、コンパイラで `*`（乗算）、`/`（除算）、`abs` に通常の複雑な数式を使用可能に指定するときに使用します。

注 : このディレクティブは、C 規格では必須です。このディレクティブは認識されますが、コンパイラでは何の効果もありません。

STDC FENV_ACCESS

構文

```
#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}
```

パラメータ

ON	ソースコードは浮動小数点環境にアクセスします。この引数はコンパイラではサポートされていない点に注意してください。

OFF	ソースコードは浮動小数点環境にアクセスしません。
DEFAULT	デフォルトの動作を設定します。つまり OFF です。

説明

このプラグマディレクティブを使用して、ソースコードが浮動小数点環境にアクセスするかどうかを指定します。

注 : このディレクティブは、C 規格では必須です。

STDC FP_CONTRACT

構文

```
#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}
```

パラメータ

ON	コンパイラが浮動小数点式を縮約できます。
OFF	コンパイラが浮動小数点式を縮約できません。この引数はコンパイラではサポートされていない点に注意してください。
DEFAULT	デフォルトの動作を設定します。つまり ON です。

説明

このプラグマディレクティブを使用して、コンパイラが浮動小数点式を縮約できるかどうかを指定します。このディレクティブは、C 規格では必須です。

例

```
#pragma STDC FP_CONTRACT=ON
```

type_attribute

構文

```
#pragma type_attribute=type_attr[ type_attr...]
```

パラメータ

このプラグマディレクティブと使用可能な型属性の詳細については、309 ページの*型属性*を参照してください。

説明

このプラグマディレクティブは、C 規格には含まれない IAR 固有の*型属性*を指定する場合に使用します。ただし、指定した型属性がすべてのオブジェクトに適用されるとは限らない点に注意が必要です。

このディレクティブは、プラグマディレクティブ直後の識別子、次の変数、次の関数の宣言に影響します。

例

この例では、int オブジェクトメモリ属性 __near を持つ型属性が定義されます。

```
#pragma type_attribute=__near
int x;
```

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。

```
__near int x;
```

関連項目

*拡張キーワード*の章。

unroll

構文

`#pragma unroll=n`

パラメータ

n	展開されたループ内にあるループ本体の数で、定数の整数です。

説明

このプラグマディレクティブを使用して、ディレクティブのすぐ後ろにあるループを展開し、展開されたループに *n* のループ本体があるように指示します。このプラグマディレクティブは、for、do、while の各ループの直前にのみ配置でき、繰り返し回数はコンパイル時に決定できます。

展開は通常、比較的小さいループの場合に最も効率的です。ただし、大きいループを展開することが有益な場合もあります。これは、展開することで、たとえば共通部分式除去や不要なコードの除去など、展開されたループの繰り返し間にさらなる最適化の機会が生まれる場合などです。

#pragma unroll ディレクティブを使用して、展開のヒューリスティックがあまり効果的でない場合に、ループを強制的に展開することができます。また、展開のヒューリスティックの効果を低減するために、このプラグマディレクティブを使用することもできます。#pragma unroll = 1 とすると、ループが展開されなくなります。

例

```
#pragma unroll=4
for (i = 0; i < 64; ++i)
{
  foo(i * k; (i + 1) * k);
}
```

関連項目

214 ページの*ループ展開*。

vector

構文

```
#pragma vector=vector1[, vector2, vector3, ...]
```

パラメータ

vectorN	割込み関数のベクタ番号。

説明

このプラグマディレクティブは、プラグマディレクティブの直後に宣言されている割込みまたはトラップ関数のベクタを指定する場合に使用します。関数ごとに複数のベクタを定義できます。

例

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

weak

構文

```
#pragma weak symbol1[=symbol2]
```

パラメータ

symbol1	外部リンケージを持つ関数または変数。
symbol2	定義済の関数または変数。

説明

このプラグマディレクティブは次の 2 つのうちどちらかの方法で使用できます。

- 外部リンケージを持つ関数または変数の定義を、弱い定義にする。この目的で、`__weak` 属性を使用することもできます。
- 別の関数または変数に弱いエイリアスを作成する。同じ関数または変数に、複数のエイリアスを作成できます。

例

`foo` の定義を弱い定義にするには、次のように記述します。

```
#pragma weak foo
```

`NMI_Handler` を `Default_Handler` の弱いエイリアスにするには、次のように記述します。

```
#pragma weak NMI_Handler=Default_Handler
```

`NMI_Handler` がプログラムの他の場所で定義されていない場合、`NMI_Handler` へのすべての参照は、`Default_Handler` も参照します。

関連項目

323 ページの *__weak*。

組込み関数

- 組込み関数の概要
- 組込み関数の詳細

組込み関数の概要

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。

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。

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
__break	BRK 命令を挿入
__disable_interrupt	割込みを禁止します
__enable_interrupt	割込みを有効にします
__get_interrupt_level	割込みレベルを返します
__get_interrupt_state	割込み状態を返します
__halt	停止 / 無動作の命令ペアを挿入
__mach	MACH 命令を挿入 (S3 コア)
__machu	MACHU 命令を挿入 (S3 コア)
__no_operation	NOP 命令を挿入します
__set_interrupt_level	割込みレベルを設定します
__set_interrupt_state	割込み状態を復元します
__stop	STOP 命令を挿入

表 37: 組込み関数の一覧

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。

__break

構文

```
void __break(void);
```

説明

BRK 命令を挿入します。

__disable_interrupt

構文

```
void __disable_interrupt(void);
```

説明

DI 命令を挿入して割込みを無効にします。

__enable_interrupt

構文

```
void __enable_interrupt(void);
```

説明

EI 命令を挿入して割込みを有効にします。

__get_interrupt_level

構文

```
__ilevel_t __get_interrupt_level(void);
```

説明

現在の割込みレベルを返します。リターン型 __ilevel_t は次の定義があります。

```
typedef unsigned char __ilevel_t;
```

__get_interrupt_level リターン値は、__set_interrupt_level 命令関数の引数として使用できます。

__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。

例

`__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。

__halt

構文

```
void __halt(void);
```

説明

停止 / 無動作の命令ペアを挿入します。

__mach

構文

```
void __mach(signed short, signed short);
```

説明

`MACH` 命令を挿入します。この組込み関数は、S3 コアでコンパイルされたコードでしか使用できません。

__machu

構文

```
void __machu(unsigned short, unsigned short);
```

説明

`MACHU` 命令を挿入します。この組込み関数は、S3 コアでコンパイルされたコードでしか使用できません。

__no_operation

構文

```
void __no_operation(void);
```

説明

命令を挿入します。

__set_interrupt_level

構文

```
void __set_interrupt_level(__ilevel_t);
```

説明

割込みレベルを設定します。`__ilevel_t` 型については、350 ページの *__get_interrupt_level* を参照してください。

__set_interrupt_state

構文

```
void __set_interrupt_state(__istate_t);
```

説明

割込み状態を、前回 `__get_interrupt_state` 関数から返された値に復元します。

`__istate_t` 型については、350 ページの *__get_interrupt_state* を参照してください。

__stop

構文

```
void __stop(void);
```

説明

`STOP` 命令を挿入します。

プリプロセッサ

- プリプロセッサの概要
- 定義済プリプロセッサシンボルの詳細
- その他のプリプロセッサ拡張

プリプロセッサの概要

RL78 用 IAR C/C++ コンパイラのプロプロセッサは、C 規格に準拠しています。また、コンパイラでは、以下のプリプロセッサ関連機能も利用可能です。

- 定義済プリプロセッサシンボル

  これらのシンボルを使用して、コンパイル日時などのコンパイル時の環境を調べることができます。詳細については、354 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（「245 ページの *-D*」を参照）。

- プリプロセッサ拡張

  多数のプラグマディレクティブなど、各種のプリプロセッサ拡張を利用できます。詳細については、「*プラグマディレクティブ*」を参照してください。該当する `_Pragma` 演算子や、その他のプリプロセッサ関連の拡張については、358 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（『266 ページの *--preprocess*』を参照）。

インクルードファイルのパスを指定するには、スラッシュを使用します。

```
#include "mydirectory/myfile"
```

ソースコードでは、スラッシュを使用します。

```
file = fopen("mydirectory/myfile","rt");
```

バックスラッシュも使用可能です。この場合、インクルードファイルパスでは 1 つ、ソースコード文字列では 2 つ使用してください。

定義済プリプロセッサシンボルの詳細

このセクションでは、プログラムプロセッサシンボルの一覧と説明を提供します。

__BASE_FILE__

説明

コンパイル中の基本ソースファイル（ヘッダファイルでないファイル）の名前を示す文字列です。

関連項目

356 ページの __FILE__、260 ページの *--no_path_in_file_macros*。

__BUILD_NUMBER__

説明

使用中のコンパイラのビルド番号を示す固有の整数です。ビルド番号は、必ずしも後でリリースされたコンパイラの方が遅い番号になるとは限りません。

__CALLING_CONVENTION__

説明

使用中のデフォルトの呼出し規約を示します。値は `--calling_convention` オプションの設定を反映し、`__CC_V1__` または `__CC_V2__` に定義されます。これらのシンボル名は、`__CALLING_CONVENTION__` シンボルの評価に使用できます。

関連項目

150 ページの*呼出し規約*、242 ページの *--calling_convention*。

__CODE_MODEL__

説明

使用中のコードモデルを示す整数です。この値は `--code_model` オプションの設定を反映し、`__CODE_MODEL_NEAR__` または `__CODE_MODEL_FAR__` に定義されます。これらのシンボル名は、`__CODE_MODEL__` シンボルの評価に使用できます。

__CORE__

説明

使用中のチップコアを示す整数です。この値は `--core` オプションの設定を反映し、`__S1__`、`__S2__`、`__S3__` に定義されます。これらのシンボル名は、`__CORE__` シンボルの評価に使用できます。

注：下位互換性のために、このシンボルは値 __RL78_0__、__RL78_1__、__RL78_2__ に対してもテストすることができます。

__COUNTER__

説明

展開されるたびに新しい整数に展開されるマクロ。ゼロ (0) から始まり、増えていきます。

__cplusplus

説明

コンパイラが C++ モードのいずれかで実行する場合に定義される整数です。それ以外の場合には定義されません。定義される場合、その値は 199711L になります。このシンボルを #ifdef で使用し、コンパイラで C++ コードが使用できるかどうかを検出できます。これは、C および C++ のコードで共有するヘッダファイルを作成する場合に特に便利です。

このシンボルは、C 規格で必須です。

__DATA_MODEL__

説明

使用中のデータモデルを示す整数です。この値は --data_model オプションの設定を反映し、__DATA_MODEL_NEAR__ または __DATA_MODEL_FAR__ に定義されます。これらのシンボル名は、__DATA_MODEL__ シンボルの評価に使用できます。

__DATE__

説明

コンパイルした日付を示す文字列です。"Mmm dd yyyy" のフォーマット（"Oct 30 2014" など）で返されます。

このシンボルは、C 規格で必須です。

__embedded_cplusplus

説明

コンパイラが C++ モードのいずれかで実行する場合に 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。

このシンボルは、C 規格で必須です。

__FAR_RUNTIME_ATTRIBUTE__

説明

--generate_far_runtime_library_calls オプションの使用時に __far_func に定義されるシンボル。それ以外のときは __near_func に定義されます。

関連項目

254 ページの *--generate_far_runtime_library_calls*。

__FILE__

説明

コンパイルされるファイルの名前を示す文字列です。基本ソースファイルとインクルードされたヘッダファイルの両方が対象となります。

このシンボルは、C 規格で必須です。

関連項目

354 ページの __BASE_FILE__、260 ページの *--no_path_in_file_macros*。

__func__

説明

シンボルが使用される関数名で初期化される定義済の文字列識別子。これは、アサーションやその他のトレースユーティリティで使用します。このシンボルを使用するには、言語拡張を有効にする必要があります。

このシンボルは、C 規格で必須です。

関連項目

251 ページの *-e*、357 ページの *__PRETTY_FUNCTION__*。

__FUNCTION__

説明

シンボルが使用される関数名で初期化される定義済の文字列識別子。これは、アサーションやその他のトレースユーティリティで使用します。このシンボルを使用するには、言語拡張を有効にする必要があります。

関連項目

251 ページの *-e*、357 ページの *__PRETTY_FUNCTION__*。

__IAR_SYSTEMS_ICC__

説明

IAR コンパイラプラットフォームを示す整数です。現行値は 8 です。将来のバージョンでは、番号が大きくなる可能性があります。このシンボルを #ifdef で評価し、コードが IAR システムズのコンパイラでコンパイルされたものかどうかを検出できます。

__ICCRL78__

説明

コードが RL78 用 IAR C/C++ コンパイラでコンパイルされる場合に、1 に設定される整数。

__LINE__

説明

コンパイル中のファイルの現在のソースの行番号を示す整数です。基本ソースファイルとインクルードされたヘッダファイルの両方が対象となります。

このシンボルは、C 規格で必須です。

__LITTLE_ENDIAN__

説明

バイトオーダを反映する整数で、1（リトルエンディアン）に定義されます。

__PRETTY_FUNCTION__

説明

シンボルが使用されている関数の関数名で初期化される定義済みの文字列識別子（パラメータ型、リターン型を含む)。"void func(char)" などです。このシンボルは、アサーションやその他のトレースユーティリティで使用します。このシンボルを使用するには、言語拡張を有効にする必要があります。

関連項目

251 ページの *-e*、356 ページの *__func__*。

__STDC__

説明

コンパイラが C 規格に準拠する場合に 1 に設定される整数です。このシンボルを #ifdef で評価し、使用中のコンパイラが C 規格に準拠しているかどうかを検出できます。*

このシンボルは、C 規格で必須です。

__STDC_VERSION__

説明

使用中の C 規格のバージョンを識別する整数。シンボルは 199901L に拡張します。ただし、--c89 コンパイラオプションが使用される場合を除きます。この場合は、シンボルは 199409L に拡張されます。このシンボルは、EC++ モードでは使用できません。

このシンボルは、C 規格で必須です。

__SUBVERSION__

説明

新しいスタイル （R8 から）コンパイラのサブバージョン番号を識別する整数。たとえば 1.2.3.4. の 3 など。

__TIME__

説明

コンパイル時刻を "hh:mm:ss" というフォーマットで示す文字列です。

このシンボルは、C 規格で必須です。

__TIMESTAMP__

説明

現在のソースファイルの最後の修正日時を識別する文字列定数。文字列のフォーマットは asctime 標準関数で使用されるものと同じです（つまり、"Tue Sep 16 13:03:52 2014" となります)。

__VER__

説明

使用中の IAR コンパイラのバージョン番号を示す整数です。番号の値は次のようにして計算されます。(100 * メジャーバージョン番号 + マイナーバージョン番号)。たとえば、コンパイラのバージョンが 3.34 の場合、3 はメジャーバージョン番号で、34 がマイナーバージョン番号です。これにより、__VER__ は 334 となります。

その他のプリプロセッサ拡張

ここでは、定義済シンボル、プラグマディレクティブ、C 規格ディレクティブ以外に利用可能なプリプロセッサ拡張について説明します。

NDEBUG

説明

このプリプロセッサシンボルは、アプリケーションに記述したアサートマクロをアプリケーションのビルドに含めるかどうかを決定します。

このシンボルを定義していない場合は、すべてのアサートマクロが評価されます。このシンボルを定義している場合は、すべてのアサートマクロがコンパイルから除外されます。すなわち、以下のケースがあります。

- このシンボルを**定義する**場合、アサートコードが含まれない
- このシンボルを**定義しない**場合、アサートコードが含まれる

したがって、アサートコードを記述し、アプリケーションをビルドする場合、このシンボルを定義することで、アサートコードを最終的なアプリケーションから除外できます。

`assert.h` 標準インクルードファイルでは、アサートマクロは定義されています。

IDE では、リリースビルド構成でアプリケーションをビルドする場合、`NDEBUG` シンボルは自動的に定義されます。

関連項目

138 ページの *Assert*。

#warning message

構文

`#warning` *`message`*

`message` には任意の文字列を指定できます。

説明

このプリプロセッサディレクティブは、メッセージを生成する場合に使用します。このディレクティブは、主にアサーションやその他のトレースユーティリティに便利です。C 規格の `#error` ディレクティブの使用方法とよく似ています。このディレクティブは、`--strict` コンパイラオプションの使用時は認識されません。

ライブラリ関数

- ライブラリの概要
- IAR DLIB ライブラリ

ライブラリ関数の詳細については、オンラインヘルプシステムを参照してください。

ライブラリの概要

IAR DLIB ライブラリは、標準の C/C++ に準拠する完全なライブラリです。このライブラリは、IEEE 754 フォーマットの浮動小数点数もサポートしています。また、ロケール、ファイル記述子、マルチバイト文字などのさまざまなレベルのサポートを指定して構成できます。

カスタマイズの詳細は、「*DLIB ランタイム環境*」を参照してください。

ライブラリ関数の詳細については、製品に付属のオンラインドキュメントを参照してください。また、DLIB ライブラリ関数のキーワードのリファレンス情報も提供されています。関数のリファレンス情報を確認するには、エディタウィンドウで関数名を選択し、F1 キーを押します。

ライブラリ関数の詳細については、本ガイドの「*C 規格の処理系定義の動作*」を参照してください。

ヘッダファイル

アプリケーションプログラムは、ヘッダファイルを通じてライブラリ定義にアクセスします。ヘッダファイルは、`#include` ディレクティブを使用して組込みます。ライブラリ定義は、複数の異なるヘッダファイルに分割されています。各ヘッダファイルは、特定の機能領域に対応しており、必要なものだけをインクルードできます。

ライブラリ定義を参照する前に、該当ヘッダファイルをインクルードする必要があります。これを行っていない場合、実行時に呼出しに失敗するか、コンパイルやリンク時にエラーやワーニングメッセージが出力されます。

ライブラリオブジェクトファイル

ほとんどのライブラリ定義は、修正なしで、すなわち製品付属のライブラリオブジェクトから直接使用できます。ランタイムライブラリの設定方法については、「110 ページの *ランタイム環境の設定*」を参照してください。リンカ

は、アプリケーションで直接的または間接的に必要なルーチンのみを含めます。

より高精度な代替ライブラリ関数

cos、sin、tan、pow のデフォルトの実装は、高速かつ小さくなるように設計されています。もうひとつの方法として、より高い精度を提供するように考えられたバージョンがあります。これらの名前は __iar_*xxx*_accuratef（関数の float 派生型）、__iar_*xxx*_accuratel（関数の long double 派生型）で、*xxx* は cos、sin などです。

より正確な以下のバージョンを使用するには、--redirect リンカオプションを使用します。

リエントラント性

関数をメインのアプリケーションや任意の数の割込みで同時に呼出すことが可能な場合、その関数はリエントラントであると言います。したがって、静的に割り当てられたデータを使用するライブラリ関数はリエントラントではありません。

DLIB ライブラリの大部分はリエントラントですが、以下の関数や部分は静的データを必要とするためリエントラントではありません。

- ヒープ関数 — malloc、free、realloc、calloc。C++ 演算子 — new と delete
- ロケール関数 — localeconv、setlocale
- マルチバイト関数 — mbrlen、mbrtowc、mbsrtowc、mbtowc、wcrtomb、wcsrtomb、wctomb
- ランド関数 — rand、srand
- 時間関数 — asctime、localtime、gmtime、mktime
- その他の関数 — atexit、strerror、strtok
- ファイルやヒープを何らかの方法で使用するすべての関数 — これには scanf、sscanf、getchar、putchar があります。また、オプション --enable_multibyte と --dlib_config=Full を使用する場合、printf と sprintf の関数（または任意の派生型）でもヒープを使用できます。

errno を設定できる関数はリエントラントではありません。その理由は、これらの関数のいずれかの結果となる errno の値は、読み込まれる前に後続の関数の使用によって上書きされる可能性があるためです。これは、特に数学関数および文字列変換関数に適用します。

以下の解決方法があります。

- 非リエントラント関数を割込みサービスルーチンで使用しない
- 非リエントラント関数の呼出しをミューテックス、保護エリアなどを利用して保護する

LONGJMP 関数

`longjmp` は、実質的に以前に定義された `setjmp` へのジャンプです。スタックの巻き戻し中にスタック上にある可変長配列や C++ オブジェクトは、どれも破壊されません。これは、リソースのリークや不正なアプリケーション動作の原因となることがあります。

IAR DLIB ライブラリ

IAR DLIB ライブラリは、組込みシステムに利用される最も重要な C/C++ ライブラリ定義を提供します。提供される定義は、以下のとおりです。

- C 規格のフリースタンディング実装への準拠。ライブラリはホストされた機能の大半をサポートしますが、基本機能の一部は実装する必要があります。詳細については、本ガイドの「*C 規格の処理系定義の動作*」を参照してください。
- ユーザプログラム用標準 C ライブラリ定義
- C++ ライブラリの定義、ユーザプログラム用
- `CSTARTUP`。起動コードを含むモジュール。本ガイドの *DLIB ランタイム環境*を参照してください。
- 低レベルの浮動小数点数ルーチンなどのランタイムサポートライブラリ
- 低レベルの RL78 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。367 ページの *C の追加機能*を参照してください。

C ヘッダファイル

ここでは、DLIB ライブラリの C 定義専用のヘッダファイルについて説明します。ヘッダファイルには、ターゲット固有の定義が追加されている場合があります。これらについては、「*C の使用*」で説明しています。

次の表は、C ヘッダファイルの一覧を示します。

ヘッダファイル	用途
assert.h	関数実行時のアサーション実行
complex.h	一般的かつ複雑な数学関数の計算
ctype.h	文字の分類
errno.h	ライブラリ関数が出力したエラーコードの評価
fenv.h	浮動小数点例外フラグ
float.h	浮動小数点数型プロパティの評価
inttypes.h	stdint.h で定義されたあらゆるタイプのフォーマットを定義
iso646.h	Amendment 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

表 38: 従来の標準 C ヘッダファイル — DLIB

C++ ヘッダファイル

ここでは、C++ ヘッダファイルについて説明します。

- C++ ライブラリヘッダファイル

  Embedded C++ ライブラリを構成するヘッダファイル。

- C++ 標準テンプレートライブラリ (STL) ヘッダファイル
  Extended Embedded C++ ライブラリの STL を構成するヘッダファイル。
- C++ C ヘッダファイル
  C ライブラリからのリソースを提供する C++ ヘッダファイル。

C++ ライブラリヘッダファイル

この表は Embedded C++ で使用可能なヘッダファイルの一覧です

ヘッダファイル	用途
complex	複素数演算をサポートするクラスを定義
fstream	外部ファイルを操作する複数の I/O ストリームクラスを定義
iomanip	引数を 1 つ指定する複数の I/O ストリームマニピュレータを宣言
ios	多くの I/O ストリームクラスとして機能するクラスを定義
iosfwd	I/O ストリームクラスの定義が必要となる前に複数の I/O ストリームクラスを宣言
iostream	標準ストリームを操作する I/O ストリームオブジェクトを宣言
istream	抽出を実行するクラスを定義
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
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

表 39: C++ ヘッダファイル

C++ 標準テンプレートライブラリ (STL) ヘッダファイル

次の表は Extended Embedded C++ で使用可能な標準テンプレートライブラリ (STL) のヘッダファイルの一覧です。

ヘッダファイル	説明
algorithm	シーケンスに対する一般的な処理を複数定義
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

表 40: 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
iterator	共通のイテレータと、イテレータに対する処理を定義
list	双方向リンクリストシーケンスコンテナ
map	map 連想コンテナ
memory	メモリ管理機能定義
numeric	シーケンスに対する一般的な数値操作
queue	キューシーケンスコンテナ
set	set 連想コンテナ
slist	一方向リンクリストシーケンスコンテナ
stack	スタックシーケンスコンテナ
utility	複数のユーティリティコンポーネントを定義
vector	ベクタシーケンスコンテナ

表 40: 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。

次の表は、新しいヘッダファイルの一覧を示します。

ヘッダファイル	用途
cassert	関数実行時のアサーション実行
cctype	文字の分類
cerrno	ライブラリ関数が出力したエラーコードの評価
cfloat	浮動小数点数型プロパティの評価
cinttypes	stdint.h で定義されたあらゆるタイプのフォーマットを定義
climits	整数型プロパティの評価
clocale	さまざまな文化圏の慣習への対応
cmath	一般的な数学関数の計算
csetjmp	非ローカルの goto 文の実行
csignal	さまざまな例外条件の制御
cstdarg	可変引数のアクセス
cstdbool	C の bool データ型のサポートを追加
cstddef	さまざまな有用な型やマクロを定義

表 41: 新しい標準 C ヘッダファイル — DLIB

ヘッダファイル	用途
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
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 41: 新しい標準 C ヘッダファイル — DLIB（続き）

組込み関数としてのライブラリ関数

特定の C ライブラリ関数は、状況によっては組込み関数として扱われ、memcpy、memset、strcat など、通常の関数呼出しではなくインラインコードを生成します。

C の追加機能

IAR DLIB ライブラリには、追加された C 機能がいくつか含まれています。

これらの機能は、以下のインクルードファイルによって提供されます。

- fenv.h
- stdio.h
- stdlib.h
- string.h
- time.h

fenv.h

fenv.h では、浮動小数点の数値のトラップ処理のサポートが、関数 fegettrapenable および fegettrapdisable によって定義されています。

stdio.h

以下の関数は、追加の I/O 機能を提供します。

fdopen	低レベルのファイル記述子に基づいてファイルを開きます。
fileno	ファイル記述子 (FILE*) から低レベルのファイル記述子を取得します。
__gets	stdin での fgets に相当します。

getw	stdin から wchar_t 文字を取得します。
putw	wchar_t 文字を stdout に配置します。
__ungetchar	stdout での ungetc に相当します。
__write_array	stdout での fwrite に相当します。

string.h

以下は、string.h に定義された追加の関数です。

strdup	ヒープ上の文字列を複製します。
strcasecmp	大文字 / 小文字を区別しない文字列を比較します。
strncasecmp	大文字 / 小文字を区別する境界のある文字列を比較します。
strnlen	境界のある文字列の長さ。

time.h

time_t および関連の関数 time、ctime、difftime、gmtime、localtime、mktime を使用するために、2 つのインタフェースがあります。

- 32 ビットのインタフェースは、1900 年から 2035 年までをサポートし、time_t で 32 ビットの整数を使用します。型と関数は __time32_t、__time32 のような名前を持っています。この派生形は、主に旧バージョンとの互換性のためだけに使用できます。
- 64 ビットのインタフェースは -9999 年から 9999 年をサポートし、time_t で符号付きの long long を使用します。型と関数は __time64_t、__time64 などのような名前を持っています。

どちらのインタフェースでも、time_t は 1970 年から始まります。

インタフェースは、システムヘッダファイル time.h で定義されます。

アプリケーションはどちらのインタフェースも使用でき、32 ビットまたは 64 ビットの派生形を明示的に使用して両方を混在させることも可能です。デフォルトでは、ライブラリとヘッダは time_t や time などを 32 ビットの派生形にリダイレクトします。ただし、これらを明示的に 64 ビットの派生形にリダイレクトするには、time.h または ctime のインクルードの前に _DLIB_TIME_USES_64 を定義します。

135 ページの*時間*を参照してください。

clock_t は、32 ビットの整数型で表します。

ライブラリにより内部的に使用されるシンボル

以下のシンボルはライブラリで使用されます。つまり、ライブラリのソースファイルなどで可視ということです。

`__assignment_by_bitwise_copy_allowed`

このシンボルは、クラスオブジェクトのプロパティを決定します。

`__constrange()`

組込み関数へのパラメータの有効範囲と、パラメータの型が `const` でなければならないことを決定します。

`__construction_by_bitwise_copy_allowed`

このシンボルは、クラスオブジェクトのプロパティを決定します。

`__has_constructor`, `__has_destructor`

これらのシンボルはクラスオブジェクトのプロパティを決定し、`sizeof` 演算子と同様に機能します。クラス、基底クラス、メンバ（再帰的）にユーザ定義のコンストラクタまたはデストラクタがそれぞれある場合、シンボルは真となります。

`__memory_of`

クラスメモリを決定します。クラスメモリによって、クラスメモリが存在できるメモリが決まります。このシンボルは、クラスメモリとしてクラス定義の中でのみ発生できます。

注：これらのシンボルは予約済みであり、ライブラリでのみ使用してください。

定義済みのシンボルの値を判断するには、コンパイラオプションの `--predef_macros` を使用します。

リンカ設定ファイル

- 概要
- メモリおよび領域の定義
- 領域
- セクションの取扱い
- セクションの選択
- シンボル、式、数値の使用
- 構造化設定

この章を読む前に、*セクション*のコンセプトについて知っておく必要があります。84 ページの *モジュールおよびセクション*を参照してください。

概要

要件に合わせてメモリのアプリケーションをリンクおよび検出するため、ILINK には、セクションを扱う方法、および使用できるメモリエリアにセクションを配置する方法に関する情報が必要です。つまり、ILINK には、*リンカ設定ファイル*により渡される*設定*が必要です。

このファイルは、一連のディレクティブで構成され、通常、以下のことを行います。

- 使用できるアドレス可能メモリを定義する

  可能なアドレスの最大サイズに関するリンカ情報を提供し、使用可能な物理メモリを定義します。また、さまざまな方法でのアドレスが可能なメモリを扱います。
- ROM または RAM の使用可能メモリエリアを定義する

  各領域の開始および終了アドレスを提供します。
- セクショングループ

  セクション要件に従ってセクションをブロックまたはオーバレイにグループ化する方法を扱います。

- アプリケーションの初期化を扱う方法を定義する

  初期化されるセクションに関する情報、およびその初期化の方法に関する情報を提供します。
- メモリ割当て

  セクションの各セットが配置されるメモリエリアを定義します。
- シンボル、式、数値の使用

  アドレスやサイズなどを他の設定ディレクティブで表現します。シンボルは、アプリケーション自体でも使用できます。
- 構造化設定

  条件に応じてディレクティブを含める、または除外し、設定ファイルをいくつかの異なるファイルに分割できます。

コメントは、C コメント (`/*...*/`) または C++ コメント (`//...`) のいずれかとして記述できます。

メモリおよび領域の定義

ILINK には、使用可能なメモリ空間に関する情報、具体的には以下の情報が必要です。

- 使用できるアドレス可能メモリの最大サイズ

  `define memory` ディレクティブは、指定したサイズで*メモリ空間*を定義します。これは、アドレス可能メモリの最大サイズで、必ずしも物理的に使用できるサイズではありません。373 ページの *define memory ディレクティブ*を参照してください。
- 使用可能な物理メモリ

  `define region` ディレクティブは、アプリケーションコードの特定のセクションおよびアプリケーションデータのセクションを配置できる使用可能メモリの領域を定義します。373 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。375 ページの*領域式*を参照してください。

このセクションではメモリと領域の定義に固有の各リンカディレクティブについての詳しい情報を提供します。

define memory ディレクティブ

構文

```
define memory [ name ] with size = size_expr [ ,unit-size ];
```

ここで、*unit-size* は以下のいずれかです。

```
unitbitsize = bitsize_expr
unitbytesize = bytesize_expr
```

また、*expr* は式です（392 ページの*式*を参照）。

パラメータ

size_expr	メモリ空間に含まれるユニットの量を指定。これは常にアドレスゼロからカウントされます。
bitsize_expr	各ユニットに含まれるビット数を指定します。
bytesize_expr	各ユニットに含まれるバイト数を指定します。各バイトには 8 ビットが含まれます。

説明

define memory ディレクティブは、指定したサイズで*メモリ空間*を定義します。これは、アドレス可能メモリの最大サイズで、必ずしも物理的に使用できるサイズではありません。このディレクティブは、使用可能アドレスの制限をリンカ設定ファイルで設定します。通常のマイクロコントローラでは、1 つのメモリ空間で十分ですが、場合によっては、複数のメモリ空間が必要です。たとえば、ハーバードアーキテクチャでは、通常、コードとデータ用に 1 つずつ、2 つの異なるメモリ空間が必要です。メモリが 1 つだけ定義されている場合、そのメモリ範囲はオプションです。*unit-size* が指定されていない場合、ユニットには 8 ビットが含まれます。

例

```
/* 4 ギガバイトのメモリ空間 Mem を宣言 */
define memory Mem with size = 4G;
```

define region ディレクティブ

構文

```
define region name = region-expr;
```

ここで、*region-expr* は領域式です（374 ページの*領域*を参照）。

パラメータ

name	領域の名前。

説明

define region ディレクティブは、コードの特定のセクションおよびデータのセクションを配置できる領域を定義します。領域は、1 つ以上のメモリエリアで構成されます。各メモリエリアは、特定のメモリ内の連続するバイト

で構成されます。領域式を使用して、複数の範囲を組み合わせることができます。これらの範囲では、バイトが連続しなくても、同じメモリになくてもかまいません。

例

```
/* 0x10000 バイトのコード領域の ROM（メモリ Mem のアドレス
   0x10000 に配置）を定義 */
define region ROM = Mem:[from 0x10000 size 0x10000];
```

領域

*領域*は、重複しないメモリ範囲のセットです。*領域式*は、領域の*領域リテラル*およびセット操作（結合、交差、相違）で構成されます。

領域リテラル

構文

[*memory-name*: **]**[from *expr* **{** to *expr* **|** size *expr* **}**

  [repeat *expr* **[** displacement *expr* **]]**]

ここで、*expr* は式です（392 ページの*式*を参照）。

パラメータ

memory-name	領域リテラルが配置されるメモリ空間の名前メモリが 1 つだけの場合、名前はオプションです。
from *expr*	*expr* は、表示するメモリ範囲の開始アドレス（開始アドレスを含む）。
to *expr*	*expr* は、表示するメモリ範囲の終了アドレス（終了アドレスを含む）。
size *expr*	*expr* は、メモリ範囲のサイズ。
repeat *expr*	*expr* は、領域リテラルに同じメモリの複数の範囲を定義します。
displacement *expr*	*expr* は、繰返しシーケンスで前の範囲開始からの移動距離です。デフォルトの移動距離は、範囲サイズと同じ値です。

説明

領域リテラルは、1 つのメモリ範囲で構成されます。範囲を定義する場合、範囲が配置されるメモリ、開始アドレス、サイズを指定する必要があります。範囲サイズは、サイズを指定して明示的に指定するか、範囲の終了アドレスを指定して暗黙的に指定できます。終了アドレスが範囲に含まれ、ゼロサイズ領域にはアドレスのみが含まれます。メモリがどこでラップされるかが認

識されているため、範囲がアドレスゼロをスナップしたり、そのような範囲が符号なし値で表現したりできます。

`repeat` パラメータは、各繰返しに 1 つずつ、複数の範囲を含む領域リテラルを作成します。これは、バンクまたはファー領域で便利です。

例

```
/* 0 番地中心の 5 バイトの領域 */
Mem:[from -2 to 2]

/* 64kB のメモリー中で、0 番地中心の 512 バイトの領域 */
Mem:[from 0xFF00 to 0xFF]

/* 同じメモリ上の、いくつかの繰り返し領域
   リテラル */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* 次の定義と同じ :
   Mem:[from 0 size 0x100]
   Mem:[from 0x1000 size 0x100]
   Mem:[from 0x2000 size 0x100]
*/
```

関連項目

373 ページの *define region ディレクティブ*、375 ページの*領域式*。

領域式

構文

```
region-operand
 | region-expr | region-operand
 | region-expr - region-operand
 | region-expr & region-operand
```

ここで、*region-operand* は以下のいずれかです。

```
( region-expr )
region-name
region-literal
empty-region
```

ここで、*region-name* は領域です（詳細は 373 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 374 ページの*領域リテラル*を参照）。

empty-region は空領域です（詳細は 376 ページの*空の領域*を参照）。

説明

通常、領域は、1 つのメモリ範囲で構成されます。つまり、1 つの*領域リテラル*で領域を表現できます。領域に複数の範囲が（場合によっては異なるメモリに）含まれる場合、*領域式*を使用して領域を表現する必要があります。領域式は、実際、メモリ範囲のセットにおけるセット式です。

領域式を作成するために 3、つの演算子、結合 (|)、交差 (&)、相違 (-) が使用できます。これらの演算子は、*セット理論*に基づいて機能します。たとえば、セット A および B がある場合、演算子の結果は以下のようになります。

- `A | B`: セット A またはセット B いずれかのすべてのエレメント
- `A & B`: セット A およびセット B 両方のすべてのエレメント
- `A - B`: セット A にありセット B にないすべてのエレメント

例

```
/* 結果は Mem 上の 1000 - 2FFF の
   範囲になる */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 1500 - 1FFF の
   範囲になる */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の 1000 - 14FF の
   範囲になる */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]

/* 結果は Mem 上の ２つの範囲 1000 - 1FFF
   と 2501 - 2FFF。
   になる */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

空の領域

構文

```
[ ]
```

説明

空の領域には、メモリ範囲は含まれません。空の領域が、1 つ以上のセクションの配置のために実際に使用される配置ディレクティブで使用される場合、ILINK ではエラーが発生します。

例

```
define region Code = Mem:[from 0 size 0x10000];
if (Banked) {
  define region Bank = Mem:[from 0x8000 size 0x1000];
} else {
  define region Bank = [];
}
define region NonBanked = Code - Bank;

/* シンボル Banked により、NonBanked 領域は
   0x10000 バイトの 1 つの領域か 0x8000 バイトと
   0x7000 バイトの 2 つの領域になる。*/
```

関連項目

375 ページの *領域式*。

セクションの取扱い

セクションの取扱いは、ILINK で実行イメージのセクションをどう処理するかについて説明します。これは以下のことを意味します。

- セクションを領域に配置する

  `place at` ディレクティブと `place into` ディレクティブは、似ている属性を持つセクションのセットを、以前に定義された領域に配置します。384 ページの *place at ディレクティブ*および 385 ページの *place in ディレクティブ*を参照してください。

- 特殊な要件のセクションのセットを作成する

  `block` ディレクティブを使用すると、特殊なサイズおよびアラインメントを持つ、またはタイプの異なるシーケンシャルにソートされたセクションなど、空のセクションを作成できます。

  `overlay` ディレクティブを使用すると、複数のオーバレイイメージを含むことができるメモリの領域を作成できます。378 ページの *define block ディレクティブ*、379 ページの *define overlay ディレクティブ*を参照。

- アプリケーションの初期化

  ディレクティブ `initialize` と `do not initialize` は、アプリケーションをどのように起動するかを制御します。これらのディレクティブを使用すると、アプリケーションは、起動時にグローバルシンボルを初期化したり、コードの一部をコピーしたりできます。イニシャライザは、たとえば、圧縮するなど、いくつかの方法で格納できます。380 ページの *initialize ディレクティブ*および 383 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。384 ページの *keep ディレクティブ*を参照してください。

- `use init table` ディレクティブ

このセクションではセクションの取扱いに固有の各リンカディレクティブについての詳しい情報を提供します。

define block ディレクティブ

構文

```
define  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
```

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（387 ページの*セクションの選択*を参照）。

パラメータ

name	定義するブロックの名前。
size	ブロックのサイズをカスタマイズします。デフォルトでは、ブロックのサイズは、その内容に依存するパーツの合計です。
maximum size	ブロックのサイズの上限を指定します。ブロックのセクションがこのサイズに合わない場合、エラーが発生します。
alignment	ブロックの最小アラインメントを指定します。ブロックの任意のセクションのアラインメントが、最小アラインメントを超える場合、そのアラインメントがブロックのアラインメントになります。
fixed order	セクションを固定順で配置します。指定されない場合、セクションは任意の順序で配置されます。

説明

block ディレクティブでは、空のセクションまたはその他のブロックのセットが含まれているメモリの連続エリアを定義します。内容のないブロックはスタックまたはヒープに空間を割り当てるために役立ちます。内容のあるブロックは、通常連続する必要のあるセクションといっしょにグループ化するために使用されます。

__section_begin、__section_end、または __section_size 演算子を使用して、アプリケーションからブロックの開始、終了、およびサイズにアクセスできます。指定の名前のブロックがなくても、その名前のセクションがある場合は、リンカでブロックは作成され、それにはそのセクションがすべて入ります。

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 =  { };
```

関連項目

195 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、379 ページの *define overlay ディレクティブ*を参照してください。

define overlay ディレクティブ

構文

```
define overlay name [ with param, param... ]
{
  extended-selectors;
}
[except
 {
   section_selectors
 }];
```

拡張セレクタおよび except 句については、387 ページの*セクションの選択*を参照してください。

パラメータ

name	オーバレイの名前。
size	オーバレイのサイズをカスタマイズします。デフォルトでは、オーバレイのサイズはその内容に依存するパーツの合計です。
maximum size	オーバレイのサイズの上限を指定します。オーバレイのセクションがこのサイズに合わない場合、エラーが発生します。

`alignment` オーバレイの最小アラインメントを指定します。オーバレイの任意のセクションのアラインメントが、最小アラインメントを超える場合、そのアラインメントがオーバレイのアラインメントになります。

`fixed order` セクションを固定順で配置します。指定されない場合、セクションは任意の順序で配置されます。

説明

`overlay` ディレクティブは、セクションの指定セットを定義します。`block` ディレクティブとは対照的に、overlay ディレクティブは、同じ名前を複数回定義できます。各定義は、同じ名前の他のすべての定義と同じメモリ内の場所にまとめることができます。これにより、複数の独立したサブアプリケーションをもつアプリケーションで利用できる、*オーバレイ*メモリエリアが作成されます。

各サブアプリケーションイメージを ROM に配置し、すべてのサブアプリケーションを保持する RAM オーバレイエリアを予約します。サブアプリケーションを実行するには、まず ROM から RAM オーバレイにサブアプリケーションをコピーします。ILINK では、オーバレイ間での参照に関して生成される診断メッセージ以外で、相互依存するオーバレイ定義の管理に関する支援はないので注意してください。

オーバレイのサイズは、そのオーバレイ名で定義されている最大サイズと同じサイズになり、アラインメント要件は、アラインメント要件が最高の定義と同じになります。

注 : オーバレイされたセクションは、RAM および ROM パートに分割する必要があるため、必要なすべてのコピーに注意する必要があります。

関連項目

101 ページの*手動で初期化する*。

initialize ディレクティブ

構文

```
initialize { by copy | manually }
  [ with packing = algorithm ]
{
  section-selectors
}
[except
 {
   section_selectors
  }];
```

その他のディレクティブは、ブロックに含めるセクションを選択します。387 ページの*セクションの選択*を参照してください。

パラメータ

`by copy` セクションをイニシャライザおよび初期化データ用のセクションに分割し、アプリケーション起動時に自動的に初期化を行います。

`manually` セクションをイニシャライザおよび初期化データ用のセクションに分割します。アプリケーション起動時の初期化は、自動的には行われません。

`algorithm` イニシャライザを扱う方法を指定します。以下から選択します。

`none` – 選択したセクションコンテンツの圧縮を無効にします。これは、initialize manually のデフォルト手法です。

`zeros` – 値ゼロの連続するバイトを圧縮します。

`packbits` – PackBits アルゴリズムで圧縮します。この手法は、同じ値のバイトが多数連続するデータにおいて好結果が得られます。

`lz77` – Lempel-Ziv-77 アルゴリズムを使用して圧縮します。この方法ではより広範な入力が処理されますが、デコンプレッサのサイズはわずかに大きくなります。

`bwt` – Burrows-Wheeler アルゴリズムで圧縮します。この手法は、データのブロックを圧縮する前に変換することにより、`packbits` の手法を改善するものです。

`lzw` – Lempel-Ziv-Welch アルゴリズムで圧縮します。この手法では、辞書を使用してバイトパターンをデータに格納します。

`auto` – `smallest` に似ていますが、ILINK は `none`、`packbits`、`lz77` のいずれかを選択します。これは、`initialize by copy` のデフォルト手法です。

`smallest` – ILINK が各パッキング手法（`auto` は除く）を使用して結果のサイズを予測し、推定サイズが最小になるパッキング手法を選択します。ここには、デコンプレッサのサイズも含まれます。

説明

`initialize` ディレクティブは、初期化セクションを、イニシャライザを保持するセクションと初期化データを保持するセクションに分割します。初期化されたデータを保持するセクションには元のセクション名が保持され、イニシャライザを保持するセクション名のサフィックスは `_init` となります。起動時の初期化を自動的に処理するか (`initialize by copy`)、自分で処理するか (`initialize manually`) を選択できます。

パッキング手法として auto（initialize by copy のデフォルト）または smallest を使用する場合、ILINK はイニシャライザに適したパッキングアルゴリズムを自動的に選択します。これをオーバライドする場合は、別の packing 手法を選択してください。--log initialization オプションを使用すると、使用するパッケージングアルゴリズムを ILINK がどのように決定するかが示されます。

イニシャライザを圧縮すると、デコンプレッサが自動的にイメージに追加されます。bwt および lzw のデコンプレッサは、他の方式のデコンプレッサよりも実行時間が大幅に長く、RAM の使用量も増加します。bwt では約 9KB のスタックエリアが必要となり、lzw では 3.5KB が必要となります。

イニシャライザを圧縮する場合、圧縮後のイニシャライザの正確なサイズは、未圧縮データの正確な内容がわかるまで確定しません。このデータに他のアドレスが含まれ、これらのアドレスの一部が圧縮後のイニシャライザのサイズに依存する場合には、リンカでエラー Lp017 が発生します。この問題を回避するには、圧縮後のイニシャライザを最後に配置するか、アドレスが既知である必要のないセクションと一緒にメモリ領域に配置します。

内部の依存性のために、初期化された領域のアドレスがイニシャライザのサイズに依存する場合、圧縮されたイニシャライザの生成が失敗（エラー LP021）することがあります。これを防ぐには、イニシャライザと初期化された領域をメモリの異なる部分に配置します（たとえばイニシャライザを ROM に、初期化された領域を RAM に）。

initialize manually を使用しない限り、ILINK は初期化テーブルをインクルードすることによってシステム起動時に初期化が行われるようにします。起動コードは、このテーブルを読み込む初期化ルーチンを呼出して、必要なイニシャライザを実行します。

ゼロで初期化するセクションは、initialize ディレクティブの影響を受けません。

通常 initialize ディレクティブは初期化済みの変数に使用されますが、実行可能コードを低速の ROM から高速の RAM にコピーしたり、オーバレイなど任意のセクションをコピーするときにも使用できます。他の例については、379 ページの *define overlay ディレクティブ*を参照してください。

初期化に必要なセクションは、initialize by copy ディレクティブの影響を受けません。このようなセクションとして、__low_level_init 関数およびこの関数が参照する部分のすべてが含まれます。

プログラムエントリラベルから到達可能な部分はすべて*初期化が必要*と考えられます。ただし、__iar_init$$done で始まるラベルを持つセクションフラグメントによって到達される部分は除きます。--log sections オプションは、アプリケーションにインクルードされるセクションフラグメントの作成

を記録するほかに、どのセクションが初期化に必要かを決定するプロセスも記録します。

例

```
/* 全ての read-write セクションをプログラムの開始時に自動的に ROM から RAM
にコピー */
initialize by copy { rw };
place in RAM { rw };
place in ROM { ro };
```

関連項目

90 ページの *システム起動時の初期化*、383 ページの *do not initialize ディレクティブ*。

do not initialize ディレクティブ

構文

```
do not initialize
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、387 ページの *セクションの選択*を参照してください。

説明

`do not initialize` ディレクティブは、システム起動コードで初期化しないセクションを指定します。このディレクティブを使用できるのは、`zeroinit` セクションのみです。

コンパイラのキーワード `__no_init` は、変数を `do not initialize` ディレクティブによって処理されなければならないセクションに配置します。

例

```
/* プログラムのスタートで、_noinit で終了する read-write セクションは
   初期化しない */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

関連項目

90 ページの *システム起動時の初期化*、380 ページの *initialize ディレクティブ*。

keep ディレクティブ

構文

```
keep
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、387 ページのセクションの選択を参照してください。

説明

keep ディレクティブは、選択したセクションが参照されない場合であっても、すべての選択したセクションが実行可能イメージを保持するように指定します。

例

```
keep { section .keep* } except {section .keep};
```

place at ディレクティブ

構文

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
           end of region_expr }
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、387 ページのセクションの選択を参照してください。

パラメータ

memory: *expr*	特定メモリ内の特定アドレスです。アドレスは、define memory ディレクティブで定義されている供給メモリで使用できなければなりません。メモリが 1 つだけの場合、メモリ指定子はオプションです。
start of *region_expr*	単一の内部領域になる領域式。間隔の開始位置が使用されます。

`end of region_expr`	単一の内部領域になる領域式。間隔の終了位置が使用されます。

説明

`place at` ディレクティブは、セクションおよびブロックを、特定のアドレス、あるいは領域の開始アドレスまたは終了アドレスのいずれかに配置します。2 つの異なる `place at` ディレクティブに対して、同一のアドレスを使用することはできません。また、空の領域を `place at` ディレクティブで使用することもできません。領域に配置される場合、セクションおよびブロックは、`place in` ディレクティブで同じ領域に配置される他の任意のセクションまたはブロックの前に配置されます。

name が指定されている場合、このディレクティブがマップファイルおよび一部のログメッセージで使用されます。

例

```
/* read-only セクションである .startup を コード領域の最初に配置する */
"START": place at start of ROM { readonly section .startup };
```

関連項目

385 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（374 ページの*領域*を参照）。

また、その他のディレクティブは、ブロックに含めるセクションを選択します。387 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

384 ページの *place at ディレクティブ*。

use init table ディレクティブ

構文

```
use init table name for
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

拡張セレクタおよび except 句については、387 ページの *セクションの選択*を参照してください。

パラメータ

name	init テーブル名。

説明

すべての初期化エントリは通常、1 つの初期化テーブル (Table) にまとめて生成されます。このディレクティブを使用して、一部のエントリが別のテーブルに配置されるようにします。この初期化テーブルは別のときに使用したり、通常の初期化テーブルではない異なる状況で使用することができます。

use init table ディレクティブで言及されていないすべての変数の初期化エントリは、通常の初期化テーブルに入れられます。複数の use init table ディレクティブを使用すると、複数の初期化テーブルを持つことができます。

init テーブルの開始、終了、サイズは "Region$$*name*" の __section_begin、__section_end、__section_size をそれぞれ使用するか、シンボル Region$$*name*$$Base、Region$$*name*$$Limit、Region$$*name*$$Length を使用して、アプリケーションプログラムでアクセスすることができます。

例

```
use init table Core2 for { section *.core2};

/* __section_begin("Region$$Core2") を使用して
   Core2 init テーブルの開始を取得できます。*/
```

セクションの選択

*セクションの選択*の目的は、ILINK ディレクティブが適用されるセクションを指定することです（*section-selector* および *except 句*を使用）。1 つ以上のセクションセレクタに一致するセクションがすべて選択され、except 句が指定されている場合、この句のセレクタのセクションは選択されません。各セクションセレクタは、セクション属性、セクション名、オブジェクト名またはライブラリ名が一致するセクションを選択します。

ディレクティブによっては、セクションおよびブロックの両方に適用できるなど、さらに詳細な選択が必要になることもあります。この場合、*拡張セレクタ*が使用されます。

このセクションではセクションの選択に固有の各リンカディレクティブについての詳しい情報を提供します。

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 ]
```

ここで、ro、rw、zi は、それぞれ readonly、readwrite、zeroinit とすることもできます。

section-type は以下のように指定します。

```
[ preinit_array | init_array ]
```

パラメータ

ro または readonly	リードオンリーセクション。
rw または readwrite	リード / ライトセクション。
zi または zeroinit	ゼロで初期化するセクション。これらのセクションは内容がなく、またシステムのセットアップ中にゼロで初期化される場合があります。
code	コードを含むセクション。
data	データを含むセクション。

`preinit_array`	ELF セクションタイプ `SHT_PREINIT_ARRAY` のセクション。
`init_array`	ELF セクションタイプ `SHT_INIT_ARRAY` のセクション。
sectionname	セクション名。以下の 2 つのワイルドカードを使用できます。 `?` は、任意の 1 文字と一致します。 `*` は、ゼロ以上の文字と一致します。
module	*objectname*(*libraryname*) 形式の名前。オブジェクト名とライブラリ名がそれぞれのパターンに一致するオブジェクトモジュールのセクションが選択されます。空のライブラリ名パターンでは、オブジェクトファイルのセクションのみが選択されます。
filename	オブジェクトファイル、ライブラリ、ライブラリ内のオブジェクトの名前。以下の 2 つのワイルドカードを使用できます。 `?` は、任意の 1 文字と一致します。 `*` は、ゼロ以上の文字と一致します。

説明

セクションセレクタは、*オブジェクト*（オブジェクトファイル、ライブラリ、ライブラリのオブジェクト）のセクション属性、セクションタイプ、セクション名、オブジェクトの名前（オブジェクトファイル名・ライブラリ名・ライブラリに含まれるオブジェクト名）と一致するすべてのセクションを選択します。4 つの条件のうち 3 つまでが省略可能です。セクション属性が省略された場合、セクション属性に関係なく任意のセクションが選択されます。セクションタイプを省略すると、任意のタイプのセクションが選択されます。

セクション名またはオブジェクト名の一部が省略された場合、セクション名やオブジェクト名の制限なしにセクションが選択されます。

セクションセレクタを使用せずに `{ }` のみを使用することもできます。これは、ブロックを定義する場合に便利です。

スコープの狭いセクションセレクタは、より汎用的なセクションセレクタよりも優先順位が高くなります。

複数のセクションセレクタが同じ目的に一致する場合、いずれか 1 つがより具体的でなければなりません。セクションセレクタは以下の場合により具体的となります。

- 他のものと異なり、セクションタイプを指定している
- 他のものと異なり、ワイルドカードを使用せずにセクション名またはオブジェクト名を指定している
- 他のセレクタに一致するセクションがこのセレクタにも一致しているが、その逆が真でない。

セレクタ 1	セレクタ 2	より具体的なセレクタ
section "foo*"	section "f*"	セレクタ 1
section "*x"	section "f*"	どちらも不適格
ro code section "f*"	ro section "f*"	セレクタ 1
init_array	ro section "xx"	セレクタ 1
section ".intvec"	ro section ".int"	セレクタ 1
section ".intvec"	object "foo.o"	どちらも不適格

表 42: セクションセレクタの指定の例

例

```
{ rw }                    /* 全ての read-write セクションを選択する */

{ section .mydata* }      /* .mydata*  セクションのみを選択 */
/* オブジェクトファイル special.o の中の .mydata* セクションを選択する */
{ section .mydata* object special.o }
```

`lib.a` という名前のライブラリ内に `foo.o` というオブジェクトがあって、そのオブジェクト内にセクションがある場合、以下のセレクタのいずれかがによってそのセクションが選択されます。

```
object foo.o(lib.a)
object f*{lib*}
object foo.o
object lib.a
```

関連項目

380 ページの *initialize ディレクティブ*、383 ページの *do not initialize ディレクティブ*、384 ページの *keep ディレクティブ*。

extended-selectors

構文

```
{ [ extended-selector ][, extended-selector...] }
```

ここで、*extended-selector* には以下のように指定します。

```
 [ first | last | midway ]
                    { セクションセレクタ |
                    block name [ inline-block-def ] |
                    overlay name }
```

ここで、*inline-block-def* には以下のように指定します。

[block-params] *extended-selectors*

パラメータ

first	選択したセクションやブロック、オーバレイを、それらを含む配置ディレクティブ、ブロック、オーバレイの最初に配置します。
last	選択したセクションやブロック、オーバレイを、それらを含む配置ディレクティブ、ブロック、オーバレイの最後に配置します。
midway	選択したセクション、ブロック、オーバレイを、ブロックの端から、それらを含むブロックの最大サイズの半分よりも遠くならないように配置します。このパラメータは、最大サイズを持つブロック内でしか使用できない点に注意してください。
name	ブロックまたはオーバレイの名前。

説明

配置ディレクティブまたはオーバレイに含める内容を選択するには、extended-selectors を使用します。セクション選択パターンの使用に加えて、含めるブロックやオーバレイを明示的に指定することもできます。

first または last キーワードを使用して、それらを含む配置ディレクティブの最初もしくは最後に配置するパターンやブロック、オーバレイを指定することができます。配置する順序をより正確に制御する必要がある場合は、固定の順序を持つブロックを使用できます。

ブロックは、define block ディレクティブやインラインを extended-selector の一部として使用すれば、個別に定義することができます。

midway パラメータは、主に正と負の両方のオフセットを持つことができるスタティックベースとともに使用すると便利です。

例

```
define block First { ro section .f* }; /* ".f*" に一致するすべての
                                          リードオンリーセクションを */
                                           持つブロックを定義 */
define block Table { first block First, ro section .b };
                                       /* ".b*" に一致する
                                          セクションの前に
                                          くるブロックを
                                          定義 */
```

または、個別の define block ディレクティブ内ではなく、ブロック First をインラインで定義することもできます。

```
define block Table { first block First { ro section .f* },
                     ro section .b* };
```

関連項目

378 ページの *define block ディレクティブ*、379 ページの *define overlay ディレクティブ*、384 ページの *place at ディレクティブ*。

シンボル、式、数値の使用

リンカ設定ファイルでは、以下のことが可能です。

- シンボルを定義およびエクスポートする

  define symbol ディレクティブは、設定ファイル内の式に使用可能な指定値を持つシンボルを定義します。また、シンボルは、エクスポートしてアプリケーションやデバッガでも使用できます。391 ページの *define symbol ディレクティブ*、392 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（392 ページの*式*を参照）。

このセクションではシンボルと式、数値の定義に固有の各リンカディレクティブについて詳しい情報を提供します。

define symbol ディレクティブ

構文

```
define [ exported ] symbol name = expr;
```

パラメータ

exported	実行可能イメージが利用できるシンボルをエクスポートします。
name	シンボルの名前。
expr	シンボルの値。

説明

define symbol ディレクティブは、指定値でシンボルを定義します。シンボルは設定ファイル内の式でも使用できます。この方法で定義したシンボルは、設定ファイル外でオプション --config_def で定義されたシンボルと同様に機能します。

このディレクティブの派生形 define exported symbol は、ディレクティブ define symbol を export symbol ディレクティブと組み合わせて使用するためのショートカットです。この場合、コマンドラインでは、--config_def オプションと --define_symbol オプションの両方が同一の効果を達成することが必要とされます。

注：

- シンボルを再定義することはできません。
- _x プレフィクスの付いたシンボル（x は大文字)、または __（下線 2 本）を含むシンボルは、ツールセットベンダの予約語です。

例

```
/* シンボル my_symbol を 4 と定義 */
define symbol my_symbol = 4;
```

関連項目

392 ページの *export ディレクティブ*、104 ページの *ILINK とアプリケーション間の相互処理*。

export ディレクティブ

構文

```
export symbol name;
```

パラメータ

name	シンボルの名前。

説明

export ディレクティブは、エクスポートするシンボルを定義し、これを実行可能イメージおよびグローバルラベルから使用できるようにします。アプリケーションまたはデバッガは、これを設定目的などのために参照できます。

例

```
/* シンボル my_symbol をエクスポート */
export symbol my_symbol;
```

式

構文

式は、以下の要素で構成されます。

```
expression binop expression
unop expression
expression ? expression : expression
(expression)
number
symbol
func-operator
```

ここで、*binop* は、以下のいずれかのバイナリ演算子です。

`+, -, *, /, %, <<, >>, <, >, ==, !=, &, ^, |, &&, ||`

unop は、以下のいずれかの単項演算子です。

`+, -, !, ~`

number は数値です（詳細は 393 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、391 ページの *define symbol ディレクティブ*および 276 ページの *--config_def* を参照してください。

func-operator は、以下の関数のような演算子のいずれかです。

minimum(*expr*,*expr*)	2 つのパラメータの小さい方を返します。
maximum(*expr*,*expr*)	2 つのパラメータの大きい方を返します。
isempty(*r*)	領域が空の場合は True、そうでない場合は False を返します。
isdefinedsymbol(*expr-symbol*)	式シンボルが定義されている場合は True、そうでない場合は False を返します。
start(*r*)	領域の最下位アドレスを返します。
end(*r*)	領域の最上位アドレスを返します。
size(*r*)	完全な領域のサイズを返します。

ここで、*expr* は式であり、*r* は領域式です（375 ページの*領域式*を参照）。

説明

リンカ設定ファイルでは、式は、範囲 -2^64 ～ 2^64 の 65 ビット値です。式の構文は、いくつかの例外はありますが、C 構文に従っています。代入やキャスト、事前 / 事後処理、アドレス処理 (*、&、[]、->、.) はありません。領域の式から値を抽出する処理など、いくつかの処理では、関数呼出しに似た構文が使用されます。ブール値式は、0（偽）または 1（真）を返します。

数値

構文

nr [*nr-suffix*]

ここで、nr は、10 進数または 16 進数（0x... または 0X...）のいずれかです。

また、*nr-suffix* は以下のいずれかです。

```
K          /* Kilo = (1 << 10) 1024 */
M          /* Mega = (1 << 20) 1048576 */
G          /* Giga = (1 << 30) 1073741824 */
T          /* Tera = (1 << 40) 1099511627776 */
P          /* Peta = (1 << 50) 1125899906842624 */
```

説明

数値は、通常の C 構文で、または便利なサフィックスのセットを使用して表現できます。

例

`1024` は、`0x400` と同じです。これは、`1K` と同じです。

構造化設定

構造化ディレクティブを使用すると、以下のように、リンカ設定ファイル内で構造を作成できます。

- 条件付きインクルード

  `if` ディレクティブは、条件に基づいて他のディレクティブを含めるまたは除外します。これにより、同じファイルでいくつかの異なるメモリ設定を行うことができます。395 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。395 ページの *include ディレクティブ*を参照してください。

- サポートされていないケースのエラーについて警告

このセクションでは構造化設定に固有の各リンカディレクティブについて詳しい情報を提供します。

error ディレクティブ

構文

`error` *`string`*

パラメータ

string	エラーメッセージ。

説明

`error` ディレクティブを使用して、条件付きディレクティブのアクティブな部分でディレクティブが発生する場合にエラーを発します。

例

```
エラー " サポートされていない設定 "
```

if ディレクティブ

構文

```
if (expr) {
  ディレクティブ
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

ここで、*expr* は式です（392 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

376 ページの*空の領域*を参照してください。

include ディレクティブ

構文

```
include "filename";
```

パラメータ

filename	/ と ¥ の両方をディレクトリの区切り文字として使用できるパス。

説明

include ディレクティブによって、設定ファイルをそれぞれ個別のファイルに入った論理的に異なるいくつかの部分に分割できるようになります。たとえば、頻繁に変更する必要があるファイルや、編集の必要があまりないファイルなどに分割できます。

通常は、リンカはシステム設定ディレクトリ内の設定インクルードファイルを検索します。--config_search リンカオプションを使用して、検索するディレクトリを追加することができます。

関連項目

276 ページの *--config_search*。

セクションリファレンス

- セクションの概要
- セクションおよびブロックの説明

セクションの詳細については、「84 ページの *モジュールおよびセクション*」を参照してください。

セクションの概要

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。

以下の表は、IAR ビルドツールで使用される ELF セクションおよびブロックのリストです。

セクション	説明
.bss	ゼロに初期化される __near 静的 / グローバル変数を保持します。
.bss.noinit	静的変数およびグローバル変数 __no_init __near を保持します。
.bssf	ゼロに初期化される __far 静的 / グローバル変数を保持します。
.bssf.noinit	静的変数およびグローバル変数 __no_init __far を保持します。
.bss_unit64kp	このセクションは、リンクにより内部的に使用されます。
.callt0	__callt 拡張キーワードの使用によって生成される呼出しテーブルベクタを保持します。
.const	__near の定数データを保持します。
.constf	__far の定数データを保持します。
.consth	__huge の定数データを保持します。
CSTACK	C/C++ プログラムが使用するスタックを保持します。
.data	初期化される __near の静的 / グローバル変数を保持します。

表 43: セクションの概要

セクション	説明
.data_init	リンカディレクティブ initialize の使用時に、.data セクションの初期値を保持。
.dataf	初期化される __far の静的 / グローバル変数を保持します。
.dataf_init	リンカディレクティブ initialize の使用時に、.dataf セクションの初期値を保持。
.data_unit64kp	このセクションは、リンクにより内部的に使用されます。
FAR_HEAP	動的に割り当てられた __far データに使用するヒープを保持します。
.hbss	ゼロに初期化される __huge 静的 / グローバル変数を保持します。
.hbss.noinit	静的変数およびグローバル変数 __no_init __huge を保持します。
.hdata	初期化される __huge の静的 / グローバル変数を保持します。
.hdata_init	リンカディレクティブ initialize の使用時に、.hdata セクションの初期値を保持。
HUGE_HEAP	動的に割り当てられた __huge データに使用するヒープを保持します。
.iar.dynexit	atexit テーブルを保持します。
.init_array	動的初期化関数のテーブルを保持します。
.intvec	割込みベクタテーブルを保持します。
NEAR_HEAP	動的に割り当てられた __near データに使用するヒープを保持します。
.option_byte	オンチップのデバッグインタフェースを設定するための OCD オプションバイトを保持します。
.preinit_array	動的初期化関数のテーブルを保持します。
.sbss	ゼロに初期化される静的 / グローバル変数を保持します。
.sbss.noinit	静的変数およびグローバル変数 __no_init __saddr を保持します。
.sdata	初期化される __saddr の静的 / グローバル変数を保持します。
.sdata_init	リンカディレクティブ initialize の使用時に、.sdata セクションの初期値を保持。

表 43: セクションの概要 （続き）

セクション	説明
.security_id	デバッグセッションを開始する前の認証チェックを可能にするセキュリティ ID を保持します。
.switch	__near_func 関数の切替えテーブルを保持します。
.switchf	__far_func 関数の切替えテーブルを保持します。
.text	__near_func 関数、割込み、__callt 関数を保持します。
.textf	__far_func 関数を保持します。
.text_unit64kp	このセクションは、リンクにより内部的に使用されます。
.wrkseg	追加のレジスタ変数領域であるショートアドレスの作業領域を保持します。
.vector	このセクションは、リンクにより内部的に使用されます。

表 43: 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、*87 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 `__near` 静的 / グローバル変数を保持します。
メモリ配置	RAM メモリの `0xF0000–0xFFE1F`。
関連項目	62 ページの*メモリタイプ*。

.bss.noinit

説明	静的変数およびグローバル変数 `__no_init __near` を保持します。
メモリ配置	RAM メモリの `0xF0000–0xFFE1F`。
関連項目	62 ページの*メモリタイプ*。

.bssf

説明	`__far` 静的 / グローバルのゼロ初期化済変数を保持します。
メモリ配置	RAM メモリの `0x10000–0xFFE1F`。
関連項目	62 ページの*メモリタイプ*。

.bssf.noinit

説明	`__no_init __far` 静的 / グローバル変数を保持します。
メモリ配置	RAM メモリの `0x10000–0xFFE1F`。
関連項目	62 ページの*メモリタイプ*。

.callt0

説明	`__callt` キーワードの使用によって生成される呼出しテーブルベクタを保持します。
メモリ配置	ROM メモリの `0x00080–0x000BF`。
関連項目	314 ページの *__callt*。

.const

説明
__near の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
ハードウェアにより RAM にミラーされる ROM 領域。この領域の場所はチップにより異なります。

関連項目
62 ページの*メモリタイプ*。

.constf

説明
__far の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
ROM メモリの 0x000D0–0xEFFFF。

関連項目
62 ページの*メモリタイプ*。

.consth

説明
__huge の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
ROM メモリの 0x000D0–0xEFFFF。

関連項目
62 ページの*メモリタイプ*。

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
RAM メモリの 0xF0000–0xFFE1F。

関連項目
99 ページの*スタックメモリの設定*。

.data

説明
初期化される __near 静的 / グローバル変数を保持します。オブジェクトファイルでは、これに初期値が含まれます。リンカディレクティブ initialize

を使用する際、対応する .near.data_init セクションが、それぞれの .near.data セクションに作成され、圧縮された初期値が保持されます。

メモリ配置 RAM メモリの 0xF0000–0xFFE1F。

関連項目 62 ページの*メモリタイプ*。

.data_init

説明 .data セクションの圧縮された初期値を保持します。このセクションは、initialize リンカディレクティブが使用された場合に、リンカによって作成されます。

メモリ配置 RAM メモリの 0xF0000–0xFFE1F。

関連項目 62 ページの*メモリタイプ*。

.dataf

説明 __far 静的およびグローバルの初期化済変数を保持します。オブジェクトファイルでは、これに初期値が含まれます。リンカディレクティブ initialize を使用する際、対応する .dataf_init セクションが、それぞれの .dataf 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 RAM メモリの 0x10000–0xFFE1F。

関連項目 62 ページの*メモリタイプ*。

.dataf_init

説明 .dataf セクションの圧縮された初期値を保持します。このセクションは、initialize リンカディレクティブが使用された場合に、リンカによって作成されます。

メモリ配置 RAM メモリの 0x10000–0xFFE1F。

関連項目 62 ページの*メモリタイプ*。

FAR_HEAP

説明	far メモリ内で動的に割り当てられたデータに使用されるヒープを保持します。つまり、far_malloc と far_free によって割り当てられたデータ、C++ では new と delete によって割り当てられたデータのことです。
メモリ配置	RAM メモリの 0x10000–0xFFE1F。
関連項目	100 ページの*ヒープメモリの設定*。

.hbss

説明	__huge 静的 / グローバルのゼロ初期化済変数を保持します。
メモリ配置	RAM メモリの 0x10000–0xFFE1F。
関連項目	62 ページの*メモリタイプ*。

.hbss.noinit

説明	__no_init __huge の静的 / グローバル変数を保持します。
メモリ配置	RAM メモリの 0x10000–0xFFE1F。
関連項目	62 ページの*メモリタイプ*。

.hdata

説明	初期化される __huge 静的 / グローバル変数を保持します。オブジェクトファイルでは、これに初期値が含まれます。リンカディレクティブ initialize を使用する際、対応する .hdata_init セクションが、それぞれの .hdata 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	RAM メモリの 0x10000–0xFFE1F。
関連項目	62 ページの*メモリタイプ*。

.hdata_init

説明

.hdata セクションの圧縮された初期値を保持します。このセクションは、initialize リンカディレクティブが使用された場合に、リンカによって作成されます。

メモリ配置

RAM メモリの 0x10000–0xFFE1F。

関連項目

62 ページの*メモリタイプ*。

HUGE_HEAP

説明

huge メモリ内で動的に割り当てられたデータに使用されるヒープを保持します。つまり、huge_malloc と huge_free によって割り当てられたデータ、C++ では new と delete によって割り当てられたデータのことです。

メモリ配置

RAM メモリの 0x10000–0xFFE1F。

関連項目

100 ページの*ヒープメモリの設定*。

.iar.dynexit

説明

終了時に行われる呼出しのテーブルを保持します。

メモリ配置

このセクションは、ROM メモリ内の任意の場所に配置できます。

関連項目

100 ページの *atexit 制限の設定*。

.init_array

説明

同じ静的記憶寿命を持つ 1 つ以上の C++ オブジェクトの初期化で呼出すルーチンへのポインタを保持します。

メモリ配置

ROM メモリの 0x000D0–0xEFFFF。

.intvec

説明

`__interrupt` 拡張キーワードを使用して生成される割込みベクタテーブルを保持します。

メモリ配置

ROM メモリの `0x00000–0x0007F`。

NEAR_HEAP

説明

near メモリ内で動的に割り当てられたデータに使用されるヒープを保持します。つまり、`near_malloc` と `near_free` によって割り当てられたデータ、C++ では `new` と `delete` によって割り当てられたデータのことです。

メモリ配置

RAM メモリの `0xF0000–0xFFE1F`。

関連項目

100 ページの *ヒープメモリの設定*。

.option_byte

説明

オンチップのデバッグインタフェースの設定に使用する OCD オプションバイトを保持します。

メモリ配置

ROM メモリの `0x000C0–0x000C3`。

関連項目

RL78 用 C-SPY® デバッガガイド

.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

ROM メモリの `0x000D0–0xEFFFF`。

関連項目

404 ページの *.init_array*。

.sbss

説明 ゼロに初期化された __saddr 静的 / グローバル変数を保持します。

メモリ配置 RAM メモリの 0xFFE20–0xFFEDF。

関連項目 62 ページの*メモリタイプ*。

.sbss.noinit

説明 __no_init __saddr の静的 / グローバル変数を保持します。

メモリ配置 RAM メモリの 0xFFE20–0xFFEDF。

関連項目 62 ページの*メモリタイプ*。

.sdata

説明 __saddr 静的 / グローバル初期化済変数を保持します。オブジェクトファイルでは、これに初期値が含まれます。リンカディレクティブ initialize を使用する際、対応する .sdata_init セクションが、それぞれの .sdata 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 RAM メモリの 0xFFE20–0xFFEDF。

関連項目 62 ページの*メモリタイプ*。

.sdata_init

説明 .sdata セクションの圧縮された初期値を保持します。このセクションは、initialize リンカディレクティブが使用された場合に、リンカによって作成されます。

メモリ配置 RAM メモリの 0xFFE20–0xFFEDF。

関連項目 62 ページの*メモリタイプ*。

.security_id

説明

デバッグセッションを開始する前の認証チェックを可能にするセキュリティID を保持します。

メモリ配置

ROM メモリの `0x000C4–0x000CD`。

関連項目

RL78 用 C-SPY® デバッガガイド

.switch

説明

Near コードモデルの切替えテーブルを保持します。

メモリ配置

ROM メモリの `0x000D0–0x0FFFF`。

.switchf

説明

Far コードモデルの切替えテーブルを保持します。

メモリ配置

ROM メモリの `0x000D0–0xEFFFF`。

.text

説明

プログラムコードのデフォルトのセクション。このセクションは、起動コードおよびランタイムライブラリコード、`__near_func`、`__interrupt`、`__callt` 属性で宣言されたコードを保持します。

メモリ配置

ROM メモリの `0x000D0–0x0FFFF`。

関連項目

62 ページの*メモリタイプ*。

.textf

説明

このセクションは、`__far_func` 属性で宣言されたプログラムコードを保持します。

メモリ配置

ROM メモリの `0x000D0–0xEFFFF`。

関連項目

62 ページの*メモリタイプ*。

.wrkseg

説明

`--workseg_area` オプションの使用時に、ショートアドレス作業領域を保持します。これは追加のレジスタ変数領域です。

メモリ配置

RAM メモリの `0xFFE20-0xFFEDF`。

IAR ユーティリティ

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

IAR アーカイブツール — iarchive

IAR アーカイブツール (`iarchive`) は、複数の ELF オブジェクトファイルから 1 つのライブラリ（アーカイブ）を作成できます。また、`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、『*IDE プロジェクト管理およびビルドガイド*』を参照してください。

呼出し構文

アーカイブビルダの呼出し構文は以下のとおりです。

`iarchive` *パラメータ*

パラメータ

パラメータを以下に示します。

パラメータ	説明
command	実行する操作を定義するコマンドラインオプションです。このようなオプションは、ライブラリファイル名より前に指定する必要があります。
libraryfile	操作対象のライブラリファイルです。
objectfile1 ... objectfileN	指定されたコマンドの操作対象のオブジェクトファイルです。
options	実行する動作を定義するコマンドラインオプションです。これらのオプションは、コマンドラインの任意の場所に配置できます。

表 44: iarchive パラメータ

例

以下の例では、ソースオブジェクトファイル `module1.o`、`module.2.o`、`module3.o` から `mylibrary.a` という名前のライブラリファイルを作成します。

```
iarchive mylibrary.a module1.o module2.o module3.o.
```

以下の例では、`mylibrary.a` の内容がリストされます。

```
iarchive --toc mylibrary.a
```

以下の例では、ライブラリ内の `module3.o` を `module3.o` ファイルの内容と置き換え、`module4.o` を `mylibrary.a` の後に追加します。

```
iarchive --replace mylibrary.a module3.o module4.o
```

IARCHIVE コマンドの概要

以下の表に、`iarchive` コマンドの概要を示します。

コマンドラインオプション	説明
`--create`	リストされたオブジェクトファイルを含むライブラリを作成します。
`--delete, -d`	リストされたオブジェクトファイルをライブラリから削除します。
`--extract, -x`	リストされたオブジェクトファイルをライブラリから抽出します。

表 45: iarchive コマンドの概要

コマンドラインオプション	説明
--replace, -r	リストされたオブジェクトファイルにより、ライブラリでの置換または追加を行います。
--symbols	ライブラリ内のファイルによって定義されているシンボルをすべてリストします。
--toc, -t	ライブラリ内のファイルすべてリストします。

表 45: iarchive コマンドの概要（続き）

詳細については、424 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 46: iarchive オプションの概要

詳細については、424 ページの*オプションの説明*を参照してください。

診断メッセージ

ここでは、iarchive で生成されたメッセージについて説明します。

La001: could not open file *filename*

iarchive がオブジェクトファイルを開くことができませんでした。

La002: illegal path *pathname*

パス *pathname* は有効なパスではありません。

La006: too many parameters to *cmd* command

単一のライブラリファイルのみを指定可能なコマンドに、オブジェクトモジュールのリストがパラメータとして指定されました。

La007: too few parameters to *cmd* command

オブジェクトモジュールのリストを指定可能なコマンドが発行されましたが、必要なモジュールが指定されていません。

La008: *lib* is not a library file

ライブラリファイルが基本構文チェックをパスしませんでした。このファイルはライブラリファイルを意図したものではない可能性があります。

La009: *lib* has no symbol table

ライブラリファイルに必要なシンボル情報が含まれていません。ファイルがライブラリファイルを意図したものではないか、ファイルに ELF オブジェクトモジュールが含まれていない可能性があります。

La010: no library parameter given

ツールが操作対象のライブラリを特定できませんでした。ライブラリファイルが指定されていない可能性があります。

La011: file *file* already exists

同じ名前のファイルがすでに存在するため、ファイルを作成できませんでした。

La013: file confusions, *lib* given as both library and object

オブジェクトモジュールのリストにライブラリファイルも記述されています。

La014: module *module* not present in archive *lib*

指定されたオブジェクトモジュールがアーカイブで見つかりませんでした。

La015: internal error

呼出しにより `iarchive` で予期しないエラーが発生しました。

Ms003: could not open file *filename* for writing

`iarchive` が、書込み用のアーカイブファイルを開くことができませんでした。ライト禁止になっていないか確認してください。

Ms004: problem writing to file *filename*

ファイル *`filename`* への書込み中にエラーが発生しました。ボリュームがいっぱいであることが原因と考えられます。

Ms005: problem closing file *filename*

ファイル *`filename`* を閉じているときにエラーが発生しました。

IAR ELF ツール — ielftool

IAR ELF Tool (ichecksum) は、メモリの特定の範囲におけるチェックサムを生成できます。このチェックサムは、使用しているアプリケーションで計算されるチェックサムと比較できます。

ielftool のソースコードおよび Microsoft VisualStudio 2005 のテンプレートプロジェクトは、rl78¥src¥elfutils ディレクトリにあります。チェックサムの生成方法に関する特定の要件やフォーマット変換に関する要件がある場合には、それに応じてソースコードを変更できます。

呼出し構文

IAR ELF Tool の呼出し構文は以下のとおりです。

```
ielftool [options] inputfile outputfile [options]
```

ielftool ツールは、最初にすべてのフィルオプションを処理した後、すべてのチェックサムオプションを（左から右に）処理します。

パラメータ

パラメータを以下に示します。

パラメータ	説明
inputfile	ILINK リンカにより生成される絶対 ELF 実行可能イメージ。
options	任意の使用可能なコマンドラインオプション。詳細については、414 ページの *ielftool オプションの概要*を参照してください。
outputfile	絶対 ELF 実行可能イメージ。

表 47: ielftool のパラメータ

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

例

以下の例では、メモリ範囲が 0xFF で埋め込まれた後、同じ範囲のチェックサムが計算されます。

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
           --checksum __checksum:4,crc32;0-0xFF
```

IELFTOOL オプションの概要

以下の表に、ielftool のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--bin	出力ファイルのフォーマットをバイナリに設定します。
--checksum	チェックサムを生成します。
--fill	フィルの要件を指定します。
--ihex	出力ファイルのフォーマットをリニアな Intel hex に設定します。
--parity	パリティビットを生成します。
--self_reloc	一般用ではありません。
--silent	出力抑止操作を設定します。
--simple	出力ファイルのフォーマットを簡易コードに設定します。
--simple-ne	--simple と同じですが、エントリレコードを持ちません。
--srec	出力ファイルのフォーマットを Motorola S-records に設定します。
--srec-len	各 S-record 内のデータバイト数を制限します。
--srec-s3only	S-record 出力にレコードのサブセットのみが含まれるように制限します。
--strip	デバッグ情報を削除します。
--titxt	TI-txt 形式で保存します。
--verbose, -V	実行されたすべての操作を出力します。

表 48: ielftool オプションの概要

詳細については、424 ページの*オプションの説明*を参照してください。

IAR ELF Dumper — ielfdump

IAR ELF Dumper for RL78 (ielfdumprl78) 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。

ielfdumprl78 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

ielfdumprl78 の呼出し構文は以下のとおりです。

```
ielfdumprl78 input_file [output_file]
```

注 : ielfdumprl78 は、本来、IDE での使用を目的としたコマンドラインツールではありません。

パラメータ

パラメータを以下に示します。

パラメータ	説明
input_file	入力として使用する ELF 再配置可能ファイルまたは ELF 実行可能ファイルです。
output_file	出力先のファイルまたはディレクトリ。存在せず --output オプションが指定されない場合、出力先はコンソールになります。

表 49: ielfdumprl78 parameters

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

IELFDUMP オプションの概要

以下の表に、ielfdumprl78 のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--all	名前や番号を考慮せず、すべての入力セクションに対して出力を生成します。
--code	実行可能コードを含むすべてのセクションをダンプします。
-f	コマンドラインを拡張します。
--output, -o	出力ファイルを指定します。
--no_strtab	文字列テーブルのセクションのダンプを無効にします。
--raw	すべての選択セクションに対して、そのセクションの専用出力フォーマットではなく、汎用の 16 進数 /ASCII 出力フォーマットを使用します。
--section, -s	選択した入力セクションに対して出力を生成します。

表 50: ielfdumprl78 オプションの概要

詳細については、424 ページの*オプションの説明*を参照してください。

IAR ELF オブジェクトツール — iobjmanip

IAR ELF オブジェクトツール、iobjmanip を使用して、ELF オブジェクトファイルの低レベルの操作を実行します。

呼出し構文

IAR ELF オブジェクトツール呼出し構文は以下のとおりです。

```
iobjmanip options inputfile outputfile
```

パラメータ

パラメータを以下に示します。

パラメータ	説明
options	実行する動作を定義するコマンドラインオプションです。これらのオプションは、コマンドラインの任意の場所に配置できます。オプションのどれか 1 つを指定する必要があります。
inputfile	再配置可能な ELF オブジェクトファイル。
outputfile	要求されたすべての操作が適用された、再配置可能な ELF オブジェクトファイル。

表 51: iobjmanip パラメータ

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

例

この例では、input.o 内のセクション .example が .example2 にリネームされ、結果は output.o に保存されます。

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

IOBJMANIP オプションの概要

以下の表に、iobjmanip オプションの概要を示します。

コマンドラインオプション	説明
-f	コマンドラインを拡張します。
--remove_file_path	ファイルシンボルからパス情報を削除します。
--rename_section	セクションをリネームします。
--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 52: iobjmanip オプションの概要

詳細については、424 ページの*オプションの説明*を参照してください。

診断メッセージ

ここでは、iobjmanip で生成されたメッセージについて説明します。

Lm001: No operation given

コマンドラインパラメータで、実行する処理が指定されていません。

Lm002: Expected *nr* parameters but got *nr*

パラメータが少なすぎるか、または多すぎます。iobjmanip および使用されたコマンドラインオプションの呼出し構文をチェックしてください。

Lm003: Invalid section/symbol renaming pattern *pattern*

パターンに有効なリネーム処理が定義されていません。

Lm004: Could not open file *filename*

iobjmanip が入力ファイルを開くことができませんでした。

Lm005: ELF format error *msg*

入力ファイルは有効な ELF オブジェクトファイルではありません。

Lm006: Unsupported section type *nr*

オブジェクトファイルに、iobjmanip で処理できないセクションが含まれています。出力ファイルの生成時に、このセクションは無視されます。

Lm007: Unknown section type *nr*

iobjmanip で、認識されないセクションが検出されました。iobjmanip は、内容をそのままコピーします。

Lm008: Symbol *symbol* has unsupported format

iobjmanip で、処理できないシンボルが検出されました。iobjmanip は、出力ファイルの生成時にこのシンボルを無視します。

Lm009: Group type *nr* not supported

iobjmanip では、グループタイプ GRP_COMDAT のみがサポートされています。他のグループタイプが検出された場合、結果は未定義になります。

Lm010: Unsupported ELF feature in *file*: *msg*

入力ファイルが、iobjmanip でサポートしていない機能を使用しています。

Lm011: Unsupported ELF file type

入力ファイルは再配置可能なオブジェクトファイルではありません。

Lm012: Ambiguous rename for section/symbol *name* (*alt1* and *alt2*)

セクションまたはシンボルのリネーム中に、曖昧な点が見つかりました。代替手段のいずれかが使用されます。

Lm013: Section *name* removed due to transitive dependency on *name*

明示的に削除されたセクションに依存していたため、セクションが削除されました。

Lm014: File has no section with index *nr*

--remove_section または --rename_section へのパラメータとして使用されるセクションインデックスが、入力ファイルのセクションを参照していませんでした。

Ms003: could not open file *filename* for writing

iobjmanip が、書込み用の入力ファイルを開くことができませんでした。ライト禁止になっていないか確認してください。

Ms004: problem writing to file *filename*

ファイル *filename* への書込み中にエラーが発生しました。ボリュームがいっぱいであることが原因と考えられます。

Ms005: problem closing file *filename*

ファイル *filename* を閉じているときにエラーが発生しました。

IAR Absolute Symbol Exporter — isymexport

IAR Absolute Symbol Exporter (isymexport) は、ROM イメージファイルから絶対シンボルをエクスポートします。これらのシンボルは、アドオンアプリケーションのリンク時に使用できます。

最終のアプリケーションでシンボルファイルからのシンボルを保持するには、ソースコードから、あるいはリンカオプションの --keep を使用して、そのシンボルを参照する必要があります。

呼出し構文

IAR Absolute Symbol Exporter の呼出し構文は以下のとおりです。

```
isymexport [options] inputfile outputfile [options]
```

パラメータ

パラメータを以下に示します。

パラメータ	説明
inputfile	実行可能 ELF ファイルの形式 ROM イメージです（リンクからの出力）。
options	任意の使用可能なコマンドラインオプション。詳細については、420 ページの *isymexport のオプションの概要*を参照してください。
outputfile	リンク入力として使用可能な再配置可能 ELF ファイルです。このファイルには、入力ファイル内の絶対シンボルのすべてまたは選択内容が含まれます。出力ファイルには、シンボルのみ含まれ、実際のコードやデータセクションは含まれません。ステアリングファイルを使用して、出力ファイルに含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。

表 53: isymexport のパラメータ

237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*も参照してください。

IDE で、ライブラリシンボルのエクスポートを追加するには、［プロジェクト］>［オプション］>［ビルドアクション］を選択し、［ビルド後コマンドライン］テキストフィールドでコマンドラインをたとえば次のように指定します。

```
$TOOLKIT_DIR$¥bin¥isymexport.exe "$TARGET_PATH$"
"$PROJ_DIR$¥const_lib.symbols"
```

ISYMEXPORT のオプションの概要

以下の表に、isymexport のコマンドラインオプションの一覧を示します。

コマンドラインオプション	説明
--edit	ステアリングファイルを指定します。
-f	コマンドラインを拡張します。
-f	コマンドラインを拡張します。
--generate_vfe_header	破棄された可能性のある関数への仮想関数呼出しがイメージに含まれないことを宣言します。
--reserve_ranges	イメージが使用する ROM および RAM 内のエリアを予約するためにシンボルを生成します。

表 54: isymexport オプションの概要

詳細については、424 ページの*オプションの説明*を参照してください。

ステアリングファイル

ステアリングファイルを使用して、出力に含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。ステアリングファイルでは、show ディレクティブおよび hide ディレクティブを使用して、入力ファイルからどのパブリックシンボルを出力ファイルに含めるかの選択ができます。rename ディレクティブを使用すると、入力ファイル内のシンボルの名前を変更できます。

ステアリングファイルを使用する場合は、アクティブにエクスポートされたシンボルのみが出力ファイルで使用可能になります。このため、show ディレクティブを持たないステアリングファイルでは、シンボルのない出力ファイルが生成されます。

構文

以下の構文規則が適用されます。

- それぞれのディレクティブは、別々の行に指定します。
- C のコメント (/*...*/) や C++ のコメント (//...) を使用できます。
- パターンには、シンボル名の複数文字に対応するワイルドカード文字を含めることができます。
- * 文字は、シンボル名の中のゼロ桁または複数桁の文字のシーケンスに一致すると見なされます。
- ? 文字は、シンボル名の中の任意の 1 文字に一致すると見なされます。

例

```
rename xxx_* as YYY_* /* シンボルのプレフィックスを xxx_ から YYY_ に
                         変更 */
show YYY_*            /* すべてのシンボルを YYY パッケージからエクス
                         ポート */
hide *_internal       /* ただし、内部シンボルはエクスポートしない */
show zzz?             /* zzza をエクスポートして、zzzaaa をエクス
                         ポートしない */
hide zzzx             /* zzzx はエクスポートしない */
```

Show ディレクティブ

構文

`show` *`pattern`*

パラメータ

pattern	シンボル名に一致するパターンです。

説明

パターンに一致する名前のシンボルが出力ファイルに含まれます。ただし、このシンボルが後で `hide` ディレクティブでオーバライドされる場合は除きます。

例

```
/* _pub で終わるパブリックシンボルをすべてインクルード。*/
show *_pub
```

Hide ディレクティブ

構文

`hide` *`pattern`*

パラメータ

pattern	シンボル名に一致するパターンです。

説明

パターンに一致する名前のシンボルが出力ファイルから除外されます。ただし、このシンボルが後に `show` ディレクティブでオーバライドされる場合は除きます。

例

```
/* _sys で終わるパブリックシンボルをインクルードしない */
hide *_sys
```

Rename ディレクティブ

構文

```
rename pattern1 as pattern2
```

パラメータ

pattern1 名前を変更するシンボルの検索に使用されるパターンです。パターンには、* または ? のワイルドカード文字を 1 つしか含めることができません。

pattern2 シンボルの新しい名前に使用されるパターンです。パターンにワイルドカード文字が含まれる場合、*pattern1* に含まれるものと同じ種類でなければなりません。

説明

このディレクティブは、出力ファイルから入力ファイルにシンボルをリネーム変更するときに使用します。エクスポートされるシンボルを複数の rename パターンと一致させることはできません。

rename ディレクティブは、ステアリングファイルの任意の場所に配置できますが、show および hide ディレクティブの前に実行されます。したがって、シンボルをリネームする場合、ステアリングファイルにあるすべての show ディレクティブおよび hide ディレクティブは新しい名前を参照します。

シンボルの名前が、ワイルドカードを含まない *pattern1* パターンに一致する場合、出力ファイルで *pattern2* にリネームされます。

シンボルの名前が、ワイルドカードを含む *pattern1* パターンに一致する場合、出力ファイルで *pattern2* にリネームされますが、名前のワイルドカード文字に一致する部分は保持されます。

例

```
/* xxx_start は出力ファイルで Y_start_X にリネームされます。
   xxx_stop  は出力ファイルで Y_stop_X  にリネームされます。*/
rename xxx_* Y_*_X
```

診断メッセージ

ここでは、isymexport で生成されたメッセージについて説明します。

Es001: could not open file *filename*

isymexport が指定されたファイルを開くことができませんでした。

Es002: illegal path *pathname*

パス *pathname* は有効なパスではありません。

Es003: format error: *message*

入力ファイルの読み取り中に問題が発生しました。

Es004: no input file

入力ファイルが指定されていません。

Es005: no output file

入力ファイルは指定されていますが、出力ファイルが指定されていません。

Es006: too many input files

ファイルが 3 つ以上指定されています。

Es007: input file is not an ELF executable

入力ファイルは ELF 実行可能ファイルではありません。

Es008: unknown directive: *directive*

ステアリングファイルに指定されたディレクティブが認識されません。

Es009: unexpected end of file

必要な入力の途中でステアリングファイルが終了しました。

Es010: unexpected end of line

ディレクティブが終了する前にステアリングファイルの行が終了しました。

Es011: unexpected text after end of directive

ステアリングファイルのディレクティブと同じ行に、ディレクティブの後に別のテキストが存在します。

Es012: expected *text*

指定されたテキストがステアリングファイルに存在しません。このテキストは、ディレクティブを正しく指定するために必要です。

Es013: pattern can contain at most one * or ?

現在のディレクティブの各パターンに含めることができるワイルドカード文字は、* または ? が 1 つだけです。

Es014: rename patterns have different wildcards

現在のディレクティブの両方のパターンに含まれるワイルドカードは、同じ種類でなければなりません。すなわち、両方が以下のいずれかであることが必要です。

- ワイルドカードなし
- `*` が 1 つ含まれる
- `?` が 1 つ含まれる

このエラーは、パターンがワイルドカードに関して両方のパターンが同じでない場合に発生します。

Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

入力ファイル内のシンボルが複数の `rename` パターンに一致しています。

オプションの説明

このセクションでは、さまざまなユーティリティで使用可能な各コマンドラインオプションの詳しいリファレンス情報を提供します。

--all

構文 `--all`

ツール `ielfdumprl78`

説明 このオプションは、入力ファイルの汎用属性に加え、すべての ELF セクションの内容を出力に含めるときに使用します。セクションは、インデックスの順に出力されます。ただし、再配置可能セクションについては、そのセクションが再配置用に保持しているセクションの直後に各再配置可能セクションが出力されます。

デフォルトでは、セクションの内容は出力に含まれません。

このオプションは、IDE では使用できません。

--bin

構文 `--bin`

ツール `ielftool`

説明 出力ファイルのフォーマットをバイナリに設定します。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［出力コンバータ］

--checksum

構文

```
--checksum
{symbol[+offset]|address}:size,algorithm[:[1|2][m][L|W][r][i|p]]
[,start];range[;range...]
```

パラメータ

symbol	チェックサム値が格納されるシンボルの名前です。これは、入力 ELF ファイルのシンボルテーブルに存在する必要があります。
offset	シンボルへのオフセット。
address	チェックサム値が格納される絶対アドレスです。
size	チェックサムのバイト数です (1、2、4)。これは、チェックサムシンボルのサイズ以下でなければなりません。

algorithm	使用されるチェックサムアルゴリズムです。以下のいずれかです。 • sum: バイト単位で計算される算術合計。結果は 8 ビットに切り詰められます。 • sum8wide: バイト単位で計算される算術合計。結果はシンボルのサイズに切り詰められます。 • sum32: ワード（32 ビット）単位で計算される算術合計。 • crc16: CRC16（生成多項式 0x11021）。デフォルトで使用されます。 • crc32: CRC32（生成多項式 0x104C11DB7）。 • crc64iso：CRC64iso（生成多項式 0x1B）。 • crc64ecma：CRC64ECMA（生成多項式 0x42F0E1EBA9EA3693）。 • crc=*n*: *n* の生成多項式を使用する CRC。
1|2	指定された場合は、以下のどちらかになります： • 1 －1 の補数を指定します。 • 2 －2 の補数を指定します。
m	チェックサムを計算するときに各バイト内でビット順を反転させます。
L|W	チェックサムを計算するユニットのサイズを指定します。 以下から選択します。 L: 繰返しごとに 32 ビットのチェックサムを計算します。 W: 繰返しごとに 16 ビットのチェックサムを計算します。 ユニットのサイズを指定しなければ、デフォルトで 8 ビットが使用されます。これらのパラメータを使用しても、チェックサムでエラー検出が強化されることはありません。
r	サイズ *size* の各ワード内での入力データのバイト順序を逆にします。

i|p	*start* の値が 0 より大きい場合、i または p を使用してください。指定された場合は、以下のどちらかになります： • i – チェックサムの値を開始値で初期化します。 • p – 入力データの先頭に *start* 値を含むサイズ *size* の 1 ワードを付けます。
start	デフォルトでは、チェックサムの初期値は 0 です。異なる初期値を与える必要がある場合には、*start* を使用してください。0 でない場合は、i か p のどちらかを指定する必要があります。
range	チェックサムが計算されるアドレス範囲です。16 進表記および 10 進表記を使用できます（たとえば、0x8002-0x8FFF）。

ツール

ielftool

説明

このオプションは、指定範囲の指定アルゴリズムのチェックサムを計算するときに使用します。チェックサムに外部の定義がある場合（たとえばハードウェアの CRC 実装など）、--checksum オプションに適切なパラメータを使用して、外部の設計に合わせてください。（この場合、ハードウェアのドキュメントでその設計の詳細を参照してください）。チェックサムは、*symbol* の元の値を置き換えます。新しい絶対シンボルが生成されます。計算されたチェックサムを含む *_value* がサフィックスとして symbol 名に付けられます。このシンボルは、デバッグ中など、必要に応じて後でチェックサム値へのアクセスに使用できます。

--checksum オプションがコマンドラインで複数回使用される場合、オプションは、左から右に評価されます。後で評価される --checksum オプションに指定されている *symbol* で、チェックサムが計算される場合、エラーが発生します。

例

この例は、アドレス範囲 0x8000-0x8FFF、開始値 0 の場合の crc16 アルゴリズムの使用方法を示します。

```
ielftool --checksum=__checksum:2,crc16;0x8000-0x8FFF
sourceFile.out destinationFile.out
```

sourceFile.out から読み込まれる入力データ i と、その結果のサイズ 2 バイトのチェックサム値が、シンボル __checksum に格納されます。修正された ELF ファイルは、destinationFile.out として保存されます。sourceFile.out はそのまま変わりません。

関連項目

197 ページの*チェックサムの計算*。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [チェックサム]

--code

構文

`--code`

ツール

`ielfdump`

説明

このオプションを使用して、実行可能コード（ELF セクション属性 `SHF_EXECINSTR` を持つセクション）を含むすべてのセクションをダンプします。

このオプションは、IDE では使用できません。

--create

構文

`--create libraryfile objectfile1 ... objectfileN`

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	ビルドするライブラリを構成するオブジェクトファイルです。

ツール

`iarchive`

説明

このコマンドは、一連のオブジェクトファイル（モジュール）から新しいライブラリをビルドするときに使用します。オブジェクトファイルは、コマンドラインで指定した順序どおりにライブラリに追加されます。

コマンドラインでコマンドを指定しない場合、デフォルトで `--create` が使用されます。

このオプションは、IDE では使用できません。

--delete, -d

構文

```
--delete libraryfile objectfile1 ... objectfileN
-d libraryfile objectfile1 ... objectfileN
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

ツール

iarchive

説明

このコマンドは、オブジェクトファイル（モジュール）を既存のライブラリから削除するときに使用します。コマンドラインで指定したオブジェクトファイルがすべてライブラリから削除されます。

このオプションは、IDE では使用できません。

--edit

構文

`--edit steering_file`

ツール

isymexport

説明

このオプションは、ステアリングファイルを指定するときに使用します。ステアリングファイルでは、isymexport の出力ファイルに含めるシンボルの選択を制御できるほか、必要に応じて、シンボルの名前を変更することもできます。

関連項目

420 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。237 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
237 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、ielfdumprl78、iobjmanip、isymexport

説明
このオプションは、ツールで、指定ファイル（デフォルトのファイル名拡張子は xcl）からコマンドラインオプションを読み取る場合に使用します。

コマンドファイルでは、項目はコマンドライン上でのフォーマットと同様に記述します。ただし、改行復帰文字は空白文字やタブと同様に処理されるため、複数行にわたって記述できます。

ファイルでは、C と C++ の両スタイルのコメントを使用できます。二重引用符は、Microsoft Windows のコマンドライン環境の場合と同様に機能します。

このオプションは、IDE では使用できません。

--fill

構文

`--fill [v;]`*pattern*`;`*range*`[;`*range*`...]`

パラメータ

`v`	フィルコマンドについて仮想フィルを生成します。仮想フィルはチェックサムに含まれるフィルバイトですが、出力ファイルには含まれません。これの主な使用目的は、特定の種類のハードウェアで、イメージにより指定されないバイトに既知の値（通常は `0xFF` または `0x0`）がある場合です。
pattern	プレフィックス `0x` を持つ 16 進数文字列（たとえば、`0xEF`）は、バイトのシーケンスと解釈され、デジットの各ペアが 1 バイトに相当します（たとえば、`0x123456` の場合、バイトのシーケンスは `0x12`、`0x34`、`0x56` です）。このシーケンスは、フィルエリアで繰り返されます。フィルパターンの長さが、`1` バイトより大きい場合、アドレス 0 から開始されるように繰り返されます。
range	フィルのアドレス範囲を指定します。16 進表記および 10 進表記を使用できます（たとえば、`0x8002-0x8FFF`）。各アドレスは 4 バイトアラインメントでなければならないので注意してください。

ツール

`ielftool`

説明

このオプションは、1 つ以上の範囲のすべてのギャップにパターンを埋め込むときに使用します。パターンは、式または 16 進数文字列のいずれかです。この内容は、フィルパターンが開始アドレスから終了アドレスまで繰り返し埋め込まれように計算されます。そして、実際の内容でパターンが上書きされます。

`--fill` オプションがコマンドラインで複数回使用される場合、フィル範囲がそれぞれと重複することはできません。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［リンカ］>［チェックサム］

--generate_vfe_header

構文

`--generate_vfe_header`

ツール

`isymexport`

説明 このオプションを使用して、破棄された可能性のある関数への仮想関数呼出しがイメージに含まれないことを宣言します。

リンカが仮想関数の除去を実行する際、不要と思われる仮想関数は破棄されます。最適化が正しく適用されるためには、破棄された関数に影響する仮想関数呼出しがイメージに必要です。

関連項目 201 ページの*仮想関数の除去*。

このオプションを設定するには、以下を使用します。

[プロジェクト] > [オプション] > [リンカ] > [追加オプション]

--ihex

構文 `--ihex`

ツール `ielftool`

説明 出力ファイルのフォーマットを線形の Intel hex に設定します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [リンカ] > [出力コンバータ]

--no_strtab

構文 `--no_strtab`

ツール `ielfdumprl78`

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 237 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 ielfdumprl78。

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。

ielfdumprl78

デフォルトでは、ダンプ結果の出力先はコンソールになります。このオプションは、出力先をファイルに変更するときに使用します。出力ファイルのデフォルト名は、入力ファイル名にファイル名拡張子 id を追加したものです。

また、入力ファイル名の後にファイルやディレクトリを指定して、出力ファイルを指定することもできます。

このオプションは、IDE では使用できません。

--parity

構文

--parity{*symbol*[+*offset*]|*address*}:*size*,*algo*:*flashbase*[:*flags*];*range*[;*range*...]

パラメータ

symbol	パリティバイトが格納されるシンボルの名前です。これは、入力 ELF ファイルのシンボルテーブルに存在する必要があります。
offset	シンボルへのオフセット。デフォルトでは 0 です。
address	パリティバイトが格納される絶対アドレスです。
size	パリティの生成で使用可能な最大バイト数。この値を超えるとエラーが出力されます。このサイズは、ELF ファイルの指定されたシンボルに合っている必要があります。
algo	以下から選択します。 odd：奇数のパリティを使用。 even：偶数のパリティを使用。

flashbase	フラッシュメモリの開始アドレス。*flashbase* から開始アドレスまでについては、パリティビットは生成されません。*flashbase* と開始アドレスの範囲が重なる場合、すべてのアドレスにパリティビットが生成されます。
flags	以下から選択します。 r：各ワード内でバイトオーダを反転させます。 L：一度に 4 バイトを処理します。 W：一度に 2 バイトを処理します。 B：一度に 1 バイトを処理します。
range	パリティバイトを生成するアドレス範囲。16 進表記および 10 進表記を使用できます（たとえば、0x8002-0x8FFF）。

ツール

ielftool

説明

指定範囲にパリティバイトを生成するときに使用します。この範囲は左から右の順になり、奇数または偶数のアルゴリズムを使用してパリティビットが生成されます。パリティビットは最終的に指定のシンボル内に格納され、アプリケーションからアクセスできるようになります。

このオプションは、IDE では使用できません。

--ram_reserve_ranges

構文

--ram_reserve_ranges[=*symbol_prefix*]

パラメータ

symbol_prefix	このオプションによって作成されたシンボルのプレフィックス。

ツール

isymexport

説明

このオプションを使用して、イメージが使用する RAM のエリアについてシンボルを生成します。各エリアにシンボルが 1 つ生成されます。各シンボルの名前はエリア名に基づき、オプションのパラメータ *symbol_prefix* がプレフィックスとして付きます。

あるエリアをカバーするシンボルをこの方法で生成すると、影響を受けるアドレスにリンカで他の内容を配置しないように防ぐことができます。これは、既存のイメージに対するリンク処理の際に役立ちます。

`--ram_reserve_ranges` を `--reserve_ranges` と同時に使用する場合、RAM エリアは `--ram_reserve_ranges` オプションからプレフィックスを、RAM 以外のエリアは `--reserve_ranges` オプションからプレフィックスをそれぞれ取得します。

関連項目

438 ページの *--reserve_ranges*。

このオプションは、IDE では使用できません。

--raw

構文

`--raw`

ツール

`ielfdumprl78`

説明

デフォルトでは、特定の種類のセクション専用テキストフォーマットを使用して、多数の ELF セクションがダンプされます。このオプションは、汎用テキストフォーマットを使用して、選択した各 ELF セクションをダンプするときに使用します。

汎用テキストフォーマットを使用する場合、セクション内の各バイトが 16 進数フォーマットまたは、必要に応じて ASCII テキストにダンプされます。

このオプションは、IDE では使用できません。

--remove_file_path

構文

`--remove_file_path`

ツール

`iobjmanip`

説明

このオプションは、`iobjmanip` で、生成されたオブジェクトファイルからプロジェクトソースツリーのディレクトリ情報を削除するときに使用します。つまり、ELF オブジェクトファイルのファイルシンボルが変更されることになります。

このオプションは、--remove_section ".comment" と組み合わせて使用する必要があります。

このオプションは、IDE では使用できません。

--remove_section

構文

--remove_section {*section*|*number*}

パラメータ

section	セクションを削除します（複数も可)。
number	削除されるセクションの番号。セクション番号は、ielfdumprl78 を使用して作成したオブジェクトダンプから取得できます。

ツール

iobjmanip

説明

このオプションでは、出力ファイルを作成するときに iobjmanip で指定のセクションを省略します。

このオプションは、IDE では使用できません。

--rename_section

構文

--rename_section {*oldname*|*oldnumber*}=*newname*

パラメータ

oldname	セクションをリネームします（複数も可)。
oldnumber	リネームされるセクションの番号。セクション番号は、ielfdumprl78 を使用して作成したオブジェクトダンプから取得できます。
newname	セクションの新しい名前。

ツール

iobjmanip

説明

このオプションでは、出力ファイルを作成するときに `iobjmanip` で指定のセクションをリネームします。

このオプションは、IDE では使用できません。

--rename_symbol

構文

```
--rename_symbol oldname =newname
```

パラメータ

oldname	リネームするシンボル。
newname	シンボルの新しい名前。

ツール

`iobjmanip`

説明

このオプションでは、出力ファイルを作成するときに `iobjmanip` で指定のシンボルをリネームします。

このオプションは、IDE では使用できません。

--replace, -r

構文

```
--replace libraryfile objectfile1 ... objectfileN
-r libraryfile objectfile1 ... objectfileN
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。
objectfile1 ...objectfileN	コマンドの操作対象のオブジェクトファイルです。

ツール

`iarchive`

説明

このコマンドは、既存のライブラリでオブジェクトファイル（モジュール）の置換または追加を行うときに使用します。コマンドラインで指定したオブジェクトファイルにより、ライブラリ内の同一名の既存オブジェクトファイ

ルが置換されます。同一名のファイルが存在しない場合には、コマンドラインで指定したファイルがライブラリに追加されます。

このオプションは、IDE では使用できません。

--reserve_ranges

構文

`--reserve_ranges[=symbol_prefix]`

パラメータ

symbol_prefix	このオプションによって作成されたシンボルのプレフィックス。

ツール

`isymexport`

説明

このオプションを使用して、イメージが使用する ROM および RAM のエリアについてシンボルを生成します。各エリアにシンボルが 1 つ生成されます。各シンボルの名前はエリア名に基づき、オプションのパラメータ *symbol_prefix* がプレフィックスとして付きます。

あるエリアをカバーするシンボルをこの方法で生成すると、影響を受けるアドレスにリンカで他の内容を配置しないように防ぐことができます。これは、既存のイメージに対するリンク処理の際に役立ちます。

`--reserve_ranges` を `--ram_reserve_ranges` と同時に使用する場合、RAM エリアは `--ram_reserve_ranges` オプションからプレフィックスを、RAM 以外のエリアは `--reserve_ranges` オプションからプレフィックスをそれぞれ取得します。

関連項目

434 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

ielfdumprl78

説明

このオプションは、指定した番号のセクションまたは指定した名前のセクションの内容をダンプするときに使用します。選択したセクションに再配置可能セクションが関連付けられている場合には、その内容も出力されます。

このオプションを使用する場合、入力ファイルの一般属性は出力に含まれません。

セクション番号や名前をカンマで区切るか、このオプションを複数回使用することにより、複数のセクション番号や名前を指定できます。

デフォルトでは、セクションの内容は出力に含まれません。

例

```
-s 3,17                /* セクション No.3 と No.17
-s .debug_frame,42     /* .debug_frame という名の任意のセクション
                          およびセクション No.42 */
```

このオプションは、IDE では使用できません。

--self_reloc

構文

--self_reloc

ツール

ielftool

説明

このオプションは一般用ではないため、意図的に文書化されていません。

このオプションは、IDE では使用できません。

--silent

構文

--silent
-S（iarchive のみ）

ツール

iarchive および ielftool。

説明

ツールが標準出力ストリームにメッセージ送信せずに処理を実行するように設定します。

デフォルトでは、ielftoolによりさまざまなメッセージが標準出力ストリームから送信されます。このオプションを使用して、これらのメッセージ送信を抑止できます。エラーおよび警告メッセージは、ielftoolからエラー出力ストリームに送信されるため、この設定にかかわらず表示されます。

このオプションは、IDE では使用できません。

--simple

構文　--simple

ツール　ielftool

説明　出力ファイルのフォーマットを簡易コードに設定します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [出力コンバータ]

--simple-ne

構文　--simple-ne

ツール　ielftool

説明　出力ファイルのフォーマットを簡易コードに設定しますが、エントリレコードは生成されません。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [出力コンバータ]

--srec

構文　--srec

ツール　ielftool

説明　出力ファイルのフォーマットを Motorola S-records に設定します。

オプションを設定するには、以下のように選択します。

[プロジェクト] > [オプション] > [出力コンバータ]

--srec-len

構文 `--srec-len=length`

パラメータ

length 各 S-record 内のデータバイト数。

ツール `ielftool`

説明

各 S-record 内のデータバイト数を制限します。このオプションは、`--srec` オプションと組み合わせて使用できます。

このオプションは、IDE では使用できません。

--srec-s3only

構文 `--srec-s3only`

ツール `ielftool`

説明

S-record 出力にレコードのサブセット（すなわち S0、S3、S7 レコード）のみが含まれるように制限します。このオプションは、`--srec` オプションと組み合わせて使用できます。

このオプションは、IDE では使用できません。

--strip

構文 `--strip`

ツール `iobjmanip` and `ielftool`。

説明

このオプションでは、出力ファイル を書き込む前に、デバッグ情報を含むすべてのセクションを削除します。

`ielftool` では、リンカオプションを使用していない ELF イメージが必要です。リンカで `--strip` オプションを使用する場合、これを削除し、代わりに `ielftool` の `--strip` オプションを使用します。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［リンカ］>［出力］>［出力ファイルにデバッグ情報を含める］

--symbols

構文

`--symbols libraryfile`

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

ツール

`iarchive`

説明

このコマンドは、指定したライブラリ内のオブジェクトファイル（モジュール）によって定義されるすべての外部シンボルを、そのシンボルを定義しているオブジェクトファイル（モジュール）の名前とともにリストするときに使用します。

出力抑止モード (`--silent`) の場合、このコマンドは、ライブラリファイルのシンボル関連構文チェックを実行し、エラーと警告のみを表示します。

このオプションは、IDE では使用できません。

--titxt

構文

`--titxt`

ツール

`ielftool`

説明

出力ファイルのフォーマットを TI-txt に設定します。

オプションを設定するには、以下のように選択します。

［プロジェクト］>［オプション］>［出力コンバータ］

--toc, -t

構文

```
--toc libraryfile
-t libraryfile
```

パラメータ

libraryfile	コマンドの操作対象のライブラリファイルです。 237 ページの*ファイル名またはディレクトリをパラメータとして指定する場合の規則*を参照してください。

ツール

iarchive

説明

このコマンドは、指定したライブラリ内のすべてのオブジェクトファイル（モジュール）をリストするときに使用します。

出力抑止モード (--silent) の場合、このコマンドは、ライブラリファイルの基本的な構文チェックを実行し、エラーと警告のみを表示します。

このオプションは、IDE では使用できません。

--verbose, -V

構文

```
--verbose
-V（iarchive のみ）
```

ツール

iarchive および ielftool。

説明

このオプションは、診断メッセージのほかに、実行された操作をツールで報告するときに使用します。

この設定は常に有効化されているため、このオプションは IDE では使用できません。

C 規格の処理系定義の動作

- 処理系定義の動作の詳細

C 規格ではなく C89 を使用する場合、461 ページの *C89 の処理系定義の動作*を参照してください。C 規格と C89 の違いの大まかな概要については、167 ページの *C 言語の概要*を参照してください。

処理系定義の動作の詳細

ここでは、C 規格と同順で各項目を説明します。各項目では、処理系定義の動作を説明する ISO の章 / セクション（括弧で示す）を示しています。

注 : IAR システムズの実装は、標準の C のフリースタンディング実装に準拠しています。すなわち、標準ライブラリの一部を実装で除外できます。

J.3.1 変換

診断 (3.10, 5.1.1.3)

診断は、以下のフォーマットで生成されます。

`filename,linenumber level[tag]: message`

ここで、*filename* はエラーが発生したソースファイル名、*linenumber* はコンパイラがエラーを検出した行番号、*level* はメッセージの重要度（リマーク、ワーニング、エラー、致命的なエラー）、*tag* はメッセージを識別する固有のタグ、*message* は数行に及ぶこともある説明のメッセージです。

空白文字 (5.1.1.2)

3 番目の変換フェースでは、空でない空白文字の各シーケンスが保持されます。

J.3.2 環境

文字集合 (5.1.1.2)

ソースの文字集合は、物理的なソースファイルのマルチバイト文字集合と同じです。デフォルトでは、標準の ASCII 文字集合が使用されます。ただし、`--enable_multibytes` コンパイラオプションを使用する場合、ホストの文字集合が代わりに使用されます。

Main (5.1.2.1)

プログラム起動時に呼出される関数は、main 関数です。main にはプロトタイプは宣言されていません。main でサポートされている唯一の定義は以下のとおりです。

```
int main(void)
```

この動作を変更するには、125 ページの*システム初期化のカスタマイズ*を参照してください。

プログラム終了の影響 (5.1.2.1)

アプリケーションを終了すると、(main の呼出し直後に)実行が起動コードに戻ります。

その他の main の定義方法 (5.1.2.2.1)

main 関数を定義する方法は他にありません。

main の argv 引数 (5.1.2.2.1)

argv 引数はサポートされていません。

対話型デバイスとしてのストリーム (5.1.2.3)

ストリーム stdin、stdout、stderr は対話型装置として処理されます。

シグナルとその意味およびデフォルトの処理 (7.14)

DLIB ライブラリでは、サポートされているシグナルのセットは標準の C と同じです。引き起こされたシグナルは、signal 関数がアプリケーションに合うようにカスタマイズされていない限り、何の処理も行いません。

計算の例外のシグナル値 (7.14.1.1)

DLIB ライブラリでは、計算の例外に一致する処理系定義の値はありません。

システム起動時のシグナル (7.14.1.1)

DLIB ライブラリでは、システム起動時に実行される処理系定義のシグナルはありません。

環境名 (7.20.4.5)

DLIB ライブラリでは、getenv 関数に使用される処理系定義の環境名はありません。

システム関数 (7.20.4.6)

system 関数はサポートされていません。

J.3.3 識別子

識別子のマルチバイト文字 (6.4.2)

その他のマルチバイト文字は識別子に表示されないことがあります。

識別子における重要な文字 (5.2.4.1, 6.1.2)

外部リンケージの有無に関わらず、識別子の重要な先頭文字の数は 200 文字以上であることが保証されています。

J.3.4 文字

バイト内のビット数 (3.6)

1 バイトには 8 ビットが含まれます。

実行文字集合のメンバ値 (5.2.1)

実行文字集合のメンバ値は、ASCII 文字集合の値です。これらはホストの文字集合の追加文字の値によって追加することが可能です。

英数字のエスケープシーケンス (5.2.2)

標準の英数字のエスケープシーケンスには、¥a-7、¥b-8、¥f-12、¥n-10、¥r-13、¥t-9、¥v-11 の値があります。

重要な基本文字集合外の文字 (6.2.5)

char に保存された重要な基本文字集合外の文字は変換されません。

char 型 (6.2.5, 6.3.1.1)

通常の char 型は、unsigned char として処理されます。

ソースおよび実行文字集合 (6.4.4.4, 5.1.1.2)

ソース文字集合は、ソースファイルで使用できる正当な文字集合です。デフォルトのソース文字集合は、標準 ASCII 文字集合です。ただし、コマンドラインオプション --enable_multibytes を使用した場合は、ソース文字集合はホストコンピュータのデフォルトの文字集合になります。

実行文字集合は、実行環境で使用できる正当な文字集合です。デフォルトの実行文字集合は、標準 ASCII 文字集合です。

ただし、コマンドラインオプション --enable_multibytes を使用した場合は、実行文字集合はホストコンピュータのデフォルトの文字集合になります。IAR DLIB ライブラリでは、マルチバイトの実行文字集合をサポートするには、マルチバイト文字スキャナが必要です。132 ページの*ロケール*を参照してください。

複数の文字を含む整数文字定数 (6.4.4.4)

文字数が複数の整数文字定数は、整数定数として処理されます。値は、整数定数で左端の文字を最上位文字、右端の文字を最下位文字として計算されます。値が整数定数で表現できない場合は、診断メッセージが出力されます。

複数の文字を含むワイド文字定数 (6.4.4.4)

複数のマルチバイト文字を含むワイド文字定数を使用すると、診断メッセージが出力されます。

ワイド文字定数に使用されるロケール (6.4.4.4)

デフォルトでは C ロケールが使用されます。--enable_multibytes コンパイラオプションが使用される場合、デフォルトのホストロケールが代わりに使用されます。

ワイド文字列リテラルに使用されるロケール (6.4.5)

デフォルトでは C ロケールが使用されます。--enable_multibytes コンパイラオプションが使用される場合、デフォルトのホストロケールが代わりに使用されます。

重要文字としてのソース文字 (6.4.5)

すべてのソース文字は、重要文字として表すことができます。

J.3.5 整数

拡張整数型 (6.2.5)

拡張整数型はありません。

整数値の範囲 (6.2.6.2)

整数値は、2 の補数で表現されます。最上位ビットは符号を示し、1 の場合は負の値、0 の場合は正の値またはゼロを示します。

各種の整数型の範囲については、296 ページの*基本データ型整数型*を参照してください。

拡張整数型のランク (6.3.1.1)

拡張整数型はありません。

符号付き整数型に変換したときのシグナル (6.3.1.3)

整数が符号付きの整数型に変換された場合、シグナルは引き起こされません。

符号付整数に対するビット単位の演算 (6.5)

符号付整数に対するビット単位の演算は、符号なし整数に対するビット単位演算と同様に行われます。すなわち、符号ビットが他のビットと同様に扱われます。

J.3.6 浮動小数点

浮動小数点処理の精度 (5.2.4.2.2)

浮動小数点処理の精度は不明です。

丸め処理 (5.2.4.2.2)

`FLT_ROUNDS` の非標準値はありません。

評価方法 (5.2.4.2.2)

`FLT_EVAL_METHOD` の非標準値はありません。

整数値の浮動小数点値への変換 (6.3.1.4)

整数値がソース値を正確に表現できない浮動小数点値に変換されると、最も近い値に丸めるモードが使用されます（`FLT_ROUNDS` が 1 に定義されています）。

浮動小数点の浮動小数点への変換 (6.3.1.5)

浮動小数点値がソース値を正確に表現できない浮動小数点値に変換されると、最も近い値に丸めるモードが使用されます（`FLT_ROUNDS` が 1 に定義されています）。

浮動小数点定数値の記述 (6.4.4.2)

最も近い値への丸めモードが使用されます（`FLT_ROUNDS` が 1 に定義されています）。

浮動小数点値の縮約 (6.5)

浮動小数点値は縮約されます。ただし、精度のロスはありません。シグナルはサポートされていないため、これは問題にはなりません。

FENV_ACCESS のデフォルトの状態 (7.6.1)

プラグマディレクティブ `FENV_ACCESS` のデフォルトの状態は、`OFF` です。

その他の浮動小数点メカニズム (7.6, 7.12)

浮動小数点例外、丸めモード、環境、分類は他にはありません。

FP_CONTRACT のデフォルトの状態 (7.12.2)

プラグマディレクティブ `FP_CONTRACT` のデフォルトの状態は、`OFF` です。

J.3.7 配列およびポインタ

ポインタの変換 (6.3.2.3)

データポインタおよび関数ポインタのキャストについて詳しくは、302 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、303 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。79 ページの*インライン関数*を参照してください。

J.3.9 構造体、共用体、列挙型、ビットフィールド

プレーンなビットフィールドの符号 (6.7.2, 6.7.2.1)

プレーンな `int` ビットフィールドの処理については、「298 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（251 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、298 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（295 ページの *アラインメント*を参照）。

列挙型を表すときに使用される整数型 (6.7.2.2)

特定の列挙型用に選択される整数型は、列挙型用に定義された列挙定数によって異なります。最小の整数型が選択されます。

J.3.10 修飾子

volatile オブジェクトへのアクセス (6.7.3)

`volatile` で修飾された型のオブジェクトへの参照は、すべてアクセスされます（305 ページの *オブジェクトの volatile 宣言*を参照）。

J.3.11 プリプロセッサディレクティブ

ヘッダ名のマッピング (6.4.7)

ヘッダ名のシーケンスは、ソースファイル名にそのままマッピングされます。バックスラッシュ '¥' は、エスケープシーケンスとして扱われません。353 ページの *プリプロセッサの概要*を参照してください。

定数式の文字定数 (6.10.1)

条件付きインクルードを制御する定数式の文字定数は、実行文字集合の同じ文字定数の値と一致します。

単一文字定数の値 (6.10.1)

通常の文字 (`char`) が符号付き文字として扱われる場合、単一文字定数はマイナスの値しか持つことができません（242 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、227 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、227 ページの*インクルードファイル検索手順*を参照してください。

#include ディレクティブのプリプロセッサトークン (6.10.2)

`#include` ディレクティブのプリプロセッサトークンは、`#include` ディレクティブの外側の場合と同じように組み合わされます。

#include ディレクティブのネストの制限 (6.10.2)

`#include` 処理のネストに明示的な制限はありません。

汎用文字名 (6.10.3.2)

汎用文字名 (UCN) はサポートされていません。

認識されているプラグマディレクティブ (6.10.6)

*プラグマディレクティブ*章で説明したプラグマディレクティブ以外にも、以下のディレクティブも認識されます。ただし、これらの影響は不定です。プラグマディレクティブが*プラグマディレクティブ*とここの両方に記載されている場合、ここの情報よりも*プラグマディレクティブ*の章の情報が優先します。

`alignment`

`baseaddr`

`building_runtime`

`can_instantiate`

`codeseg`

`context_handler`

`cspy_support`

`define_type_info`

do_not_instantiate

early_dynamic_initialization

function

function_effects

hdrstop

important_typedef

instantiate

keep_definition

library_default_requirements

library_provides

library_requirement_override

memory

module_name

no_pch

once

system_include

warnings

Default __DATE__ and __TIME__ (6.10.8)

__TIME__、__DATE__ の定義は常に使用可能です。

J.3.12 ライブラリ関数

その他のライブラリ機能 (5.1.2.1)

標準のライブラリ機能のほとんどがサポートされています。その一部（オペレーティングシステムを必要とするもの）には、アプリケーションに低レベルの実装が必要です。詳細については、109 ページの *DLIB ランタイム環境*を参照してください。

アサート関数によって出力される診断 (7.2.1.1)

assert() 関数で出力される診断は、以下のとおりです。

```
filename: linenr expression -- assertion failed
```

この診断は、パラメータがゼロに評価される場合に出力されます。

浮動小数点のステータスフラグの表現 (7.6.2.2)

浮動小数点のステータスフラグについては、367 ページの *fenv.h* を参照してください。

浮動小数点の例外を引き起こす feraiseexcept 関数 (7.6.2.3)

浮動小数点の例外を引き起こす `feraiseexcept` 関数については、300 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、132 ページの*ロケール*を参照してください。

float_t および double_t に定義される型 (7.12)

`FLT_EVAL_METHOD` マクロは、値 `0` しか持つことができません。

ドメインエラー (7.12.1)

標準で必要とされるもの以外のドメインエラーを生成する関数はありません。

ドメインエラーのリターン値 (7.12.1)

数学関数は、ドメインエラーに対して浮動小数点 `NaN`（not a number ＝非数）を返します。

アンダーフローエラー (7.12.1)

数学関数は `errno` をマクロ `ERANGE`（`errno.h` のマクロ）に設定し、アンダーフローエラーにゼロを返します。

fmod 関数のリターン値 (7.12.10.1)

`fmod` 関数は、2 番目の引数がゼロの場合、浮動小数点 `NaN` を返します。

remquo の規模 (7.12.10.3)

規模は、合同の剰余 `INT_MAX` です。

signal 関数 (7.14.1.1)

シグナル関連のライブラリはサポートされていません。

注 : 低レベルインタフェース関数はライブラリには存在しますが、これらの関数は何も実行しません。アプリケーション固有のシグナル処理を実装する場合は、テンプレートソースコードを使用してください。135 ページの *signal*

と *raise* を参照してください。

NULL マクロ (7.17)

`NULL` マクロは、`0` に定義されています。

改行文字による終了 (7.19.2)

`stdout` ストリーム関数は、`newline` または `end of file (EOF)` のどちらかを改行文字として認識します。

改行文字の前の空白文字 (7.19.2)

ストリームで改行文字直前に書き込まれた空白文字は保持されます。

バイナリストリームに書き込まれたデータに追加された Null 文字 (7.19.2)

バイナリストリームにライトされたデータには、NULL 文字は追加されません。

追加モードでのファイル位置 (7.19.3)

ファイル位置は、追加モードで開かれた場合にファイルの先頭に配置されます。

ファイルの切捨て (7.19.3)

テキストストリームへのライトにより、対応するファイルでその書込み位置以降が切り捨てられるかどうかは、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。131 ページの *ファイル I/O* を参照してください。

ファイルのバッファ処理 (7.19.3)

開かれたファイルは、ブロックバッファ、ラインバッファまたはアンバッファのいずれかです。

ゼロ長のファイル (7.19.3)

ゼロ長のファイルが存在するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

有効なファイル名 (7.19.3)

ファイル名が有効かどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

ファイルを開くことができる回数 (7.19.3)

ファイルを何度も開くことができるかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

ファイル内のマルチバイト文字 (7.19.3)

ファイル内のマルチバイト文字のエンコーディングは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

remove 関数 (7.19.4.1)

開いたファイルに対する削除処理の結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。131 ページの *ファイル I/O* を参照してください。

rename 関数 (7.19.4.2)

ファイルの名前を既存のファイル名に変更した結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。131 ページの *ファイル I/O* を参照してください。

開かれた一時ファイルの削除 (7.19.4.3)

開かれた一時ファイルを削除するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

モード変更 (7.19.5.4)

`freopen` は指定のストリームを閉じて、新しいモードで再び開きます。ストリーム `stdin`、`stdout`、`stderr` は、すべての新しいモードで再び開くことができます。

infinity または NaN の出力形式 (7.19.6.1, 7.24.2.1)

浮動小数点定数の infinity または `NaN` の出力形式は、それぞれ `inf`、`nan`（`F` 変換指定子の場合は `INF` と `NAN`）です。n-char-sequence は、`nan` に対しては使用されません。

printf 関数における %p (7.19.6.1, 7.24.2.1)

`printf()` の `%p` 変換指定子（出力ポインタ）の引数は、`void *` 型として処理されます。値は、`%x` 変換指定子の場合と同様に、16 進数として出力されます。

scanf 関数の読込み範囲 (7.19.6.2, 7.24.2.1)

-（ダッシュ）文字は、常に範囲記号として処理されます。

scanf 関数における %p (7.19.6.2, 7.24.2.2)

`scanf()` の `%p` 変換指定子（スキャンポインタ）は、16 進数を読み取り、`void *` 型の値に変換します。

ファイル位置のエラー (7.19.9.1, 7.19.9.3, 7.19.9.4)

ファイル位置のエラーでは、関数 `fgetpos`、`ftell`、`fsetpos` は `EFPOS` を `errno` に格納します。

nan の後の n-char-sequence (7.20.1.3, 7.24.4.1.1)

NaN の後の n-char-sequence は、読み込まれて無視されます。

アンダーフローの errno 値 (7.20.1.3, 7.24.4.1.1)

`errno` は、アンダーフローがあった場合には `ERANGE` に設定されます。

サイズがゼロのヒープオブジェクト (7.20.3)

サイズがゼロのヒープオブジェクトの要求は、NULL ポインタではなく有効なポインタを返します。

abort 関数および exit 関数の動作 (7.20.4.1, 7.20.4.4)

`abort()` または `_Exit()` を呼出しても、ストリームバッファはフラッシュされず、オープンのストリームを閉じたり、一時ファイルが削除されることもありません。

終了ステータス (7.20.4.1, 7.20.4.3, 7.20.4.4)

終了ステータスはパラメータとして `__exit()` に伝播されます。`exit()` と `_Exit()` は入力パラメータを使用しますが、abort は `EXIT_FAILURE` を使用します。

system 関数のリターン値 (7.20.4.6)

`system` 関数はサポートされていません。

タイムゾーン (7.23.1)

ローカルのタイムゾーンおよび夏時間をアプリケーションで定義する必要があります。詳細については、135 ページの*時間*を参照してください。

時間の範囲および精度 (7.23)

範囲と精度について詳しくは、368 ページの *time.h* を参照してください。アプリケーションは関数 `time` と `clock` に実際の実装を提供する必要があります。135 ページの*時間*を参照してください。

clock 関数 (7.23.2.1)

アプリケーションは、`clock` 関数の実装を提供する必要があります。135 ページの*時間*を参照してください。

%Z 置換文字列 (7.23.3.5, 7.24.5.1)

デフォルトでは `":"` が `%Z` の代わりに使用されます。使用するアプリケーションがタイムゾーンの処理を実装することになります。135 ページの*時間*を参照してください。

数学関数の丸めモード (F.9)

`math.h` の関数は、`FLT-ROUNDS` の丸め方向モードに従います。

J.3.13 アーキテクチャ

一部のマクロに割り当てられた値と式 (5.2.4.2, 7.18.2, 7.18.3)

1 バイトは常に 8 ビットです。

`MB_LEN_MAX` は、使用されるライブラリ設定に応じて最高で 6 バイトになります。

すべての基本型のサイズや範囲などについては、295 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)

295 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)

295 ページの*データ表現*を参照してください。

J.4 ロケール

ソースのメンバおよび実行文字集合 (5.2.1)

デフォルトでは、コンパイラはホストのデフォルト文字集合にあるすべての 1 バイト文字を受け入れます。コンパイラオプション `--enable_multibytes` を使用する場合、ホストのマルチバイト文字はコメントや文字列リテラルでも受け入れられます。

その他の文字集合の意味 (5.2.1.2)

拡張ソース文字集合のすべてのマルチバイト文字は、拡張実行文字集合にそのまま変換されます。文字が正しく処理されるかどうかは、ライブラリ設定のサポートを持ったアプリケーションに依存します。

マルチバイト文字のエンコードのシフト状態 (5.2.1.2)

コンパイラオプション `--enable_multibytes` を使用すると、ホストのデフォルトのマルチバイト文字が拡張ソース文字として使用可能になります。

連続する出力文字の方向 (5.2.2)

アプリケーションが表示デバイスの特性を定義します。

小数点の文字 (7.1.1)

デフォルトの小数点の文字は '.' です。ライブラリ設定シンボル `_LOCALE_DECIMAL_POINT` を定義すれば、設定し直すことができます。

出力文字 (7.4, 7.25.2)

出力文字集合は、選択したロケールによって決まります。

制御文字 (7.4, 7.25.2)

制御文字集合は、選択したロケールによって決まります。

テスト済みの文字 (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

テスト済みの文字集合は、選択したロケールによって決まります。

ネイティブ環境 (7.1.1.1)

ネイティブ環境は、"C" ロケールと同じです。

数値変換関数の対象シーケンス (7.20.1, 7.24.4.1)

数値変換関数で受け入れが可能な他の対象シーケンスはありません。

実行文字集合の照合 (7.21.4.3, 7.24.4.4.2)

実行文字集合の照合は、選択したロケールによって決まります。

strerror 関数によって返されるメッセージ (7.21.6.2)

`strerror` 関数が返すメッセージは、引数に応じて以下のようになります。

引数	メッセージ
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
その他	`error` *nnn*

表 55: strerror() が返すメッセージ— IAR DLIB ライブラリ

C89 の処理系定義の動作

- 処理系定義の動作の詳細

C89 ではなく C 規格を使用する場合、445 ページの *C 規格の処理系定義の動作*を参照してください。C 規格と C89 の違いの大まかな概要については、167 ページの *C 言語の概要*を参照してください。

処理系定義の動作の詳細

ISO の付録と同順で各項目を説明します。各項目では、処理系定義の動作を説明する ISO の章 / セクション（括弧で示す）を示しています。

変換

診断 (5.1.1.3)

診断は、以下のフォーマットで生成されます。

filename,*linenumber* *level*[*tag*]: *message*

ここで、*filename* はエラーが発生したソースファイル名、*linenumber* はコンパイラがエラーを検出した行番号、*level* はメッセージの重要度（リマーク、ワーニング、エラー、致命的なエラー）、*tag* はメッセージを識別する固有のタグ、*message* は数行に及ぶこともある説明のメッセージです。

環境

main 関数の引数 (5.1.2.2.2.1)

プログラム起動時に呼出される関数は、main 関数です。main にはプロトタイプは宣言されていません。main でサポートされている唯一の定義は以下のとおりです。

```
int main(void)
```

IAR DLIB ランタイム環境におけるこの動作を変更する場合は、125 ページの*システム初期化のカスタマイズ*を参照してください。

対話型装置 (5.1.2.3)

ストリーム stdin、stdout は、対話型装置として処理されます。

識別子

外部リンクなしの識別子 (6.1.2)

外部リンクなしの識別子での有効先頭文字数は 200 です。

外部リンクありの識別子 (6.1.2)

外部リンクありの識別子での有効先頭文字数は 200 です。

大文字と小文字の区別 (6.1.2)

外部リンクのある識別子では、大文字と小文字が区別されます。

文字

ソース文字集合・実行文字集合 (5.2.1)

ソース文字集合は、ソースファイルで使用できる正当な文字集合です。デフォルトのソース文字集合は、標準 ASCII 文字集合です。ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、ソース文字集合はホストコンピュータのデフォルトの文字集合になります。

実行文字集合は、実行環境で使用できる正当な文字集合です。デフォルトの実行文字集合は、標準 ASCII 文字集合です。ただし、コマンドラインオプション `--enable_multibytes` を使用した場合は、実行文字集合はホストコンピュータのデフォルトの文字集合になります。IAR DLIB ライブラリでは、マルチバイトの実行文字集合をサポートするには、マルチバイト文字スキャナが必要です。

132 ページの*ロケール*を参照してください。

実行文字集合の 1 文字当たりのビット数 (5.2.4.2.1)

1 文字あたりのビット数は、manifest 定数 `CHAR_BIT` で示されます。標準インクルードファイル `limits.h` では、`CHAR_BIT` は 8 と定義されています。

文字のマッピング (6.1.3.4)

ソース文字集合（文字か文字列リテラル）のメンバと実行文字集合のメンバのマッピングは、1 対 1 で設定されています。すなわち、文字集合内の各メンバを表現する値は、ISO 規格で規定されているエスケープシーケンスを除き、両者で同一のものが使用されています。

表現されない文字定数 (6.1.3.4)

基本実行文字集合かワイド文字定数用の拡張文字集合で表現されていない文字やエスケープシーケンス整数文字定数の値を使用すると、診断メッセージが出力され、実行文字集合に合わせて切り詰められます。

1 文字以上の文字定数 (6.1.3.4)

文字数が複数の整数文字定数は、整数定数として処理されます。値は、整数定数で左端の文字を最上位文字、右端の文字を最下位文字として計算されます。値が整数定数で表現できない場合は、診断メッセージが出力されます。

複数のマルチバイト文字を含むワイド文字定数を使用すると、診断メッセージが出力されます。

マルチバイト文字の変換 (6.1.3.4)

サポートされているロケール（IAR C/C++ コンパイラで提供されているロケール）は、C ロケールのみです。コマンドラインオプション `--enable_multibytes` を指定し、マルチバイトをサポートするロケールやマルチバイト文字スキャナをライブラリに追加した場合は、IAR DLIB ライブラリでマルチバイト文字がサポートされます。

132 ページの*ロケール*を参照してください。

プレーンな char の範囲 (6.2.1.1)

プレーンな `char` の範囲は、`unsigned char` と同一です。

整数

整数値の範囲 (6.1.2.5)

整数値は、2 の補数で表現されます。最上位ビットは符号を示し、1 の場合は負の値、0 の場合は正の値またはゼロを示します。

各種の整数型の範囲については、296 ページの*基本データ型整数型*を参照してください。

整数の降格 (6.2.1.2)

整数をより短い符号付整数に変換する場合は、切捨てが行われます。符号なし整数を同一長の符号付整数に変換すると値が表現できなくなる場合も、ビットパターンは変化しません。すなわち、値が十分に大きい場合、負の値に変換されます。

符号付整数に対するビット単位の演算 (6.3)

符号付整数に対するビット単位の演算は、符号なし整数に対するビット単位演算と同様に行われます。すなわち、符号ビットが他のビットと同様に扱われます。

整数除算での余りの符号 (6.3.5)

整数除算での余りの符号は、被除数の符号と同一です。

負の符号付整数型の右シフト (6.3.7)

負の符号付整数型を右シフトした場合、符号ビットが保持されます。たとえば、`0xFF00` を 1 回右シフトすると、結果は `0xFF80` になります。

浮動小数点数

浮動小数点数の表現 (6.1.2.5)

浮動小数点数の表現およびセットは、IEEE 854–1987 に準拠しています。通常の浮動小数点数は、符号ビット (s)、バイアス指数 (e)、指数部 (m) で構成されます。

浮動小数点数型 (`float`、`double`) の範囲およびサイズについては、299 ページの*基本データ型浮動小数点数型*を参照してください。

整数値から浮動小数点値への変換 (6.1.2.3)

整数値を、値を正確に表現できない浮動小数点数値にキャストした場合、値は最近似値に丸められます（切上げまたは切捨て）。

浮動小数点値の降格 (6.2.1.4)

浮動小数点数値を、サイズが小さな型の、値を正確に表現できない浮動小数点数値に変換した場合、値は最近似値に丸められます（切上げまたは切捨て）。

配列、ポインタ

size_t (6.3.3.4, 7.1.1)

`size_t` については、302 ページの *size_t* を参照してください。

ポインタからの変換・ポインタへの変換 (6.3.4)

データポインタと関数ポインタのキャストについては、302 ページの *キャスト*を参照してください。

ptrdiff_t (6.3.6, 7.1.1)

`ptrdiff_t` については、303 ページの *ptrdiff_t* を参照してください。

レジスタ

レジスタキーワードの扱い (6.5.1)

レジスタ変数についてのユーザ要求は考慮されません。

構造体、共用体、列挙型、ビットフィールド

共用体への不正なアクセス (6.3.2.3)

メンバを使用して共用体に値を格納し、そのメンバとは異なる型のメンバを使用してアクセスすると、結果は最初のメンバの内部記憶エリアに完全に依存します。

構造体メンバのパディングとアラインメント (6.5.2.1)

データオブジェクトの配置要件については、296 ページの*基本データ型整数型*を参照してください。

プレーンなビットフィールドの符号 (6.5.2.1)

プレーンな `int` ビットフィールドは、`unsigned int` ビットフィールドとして処理されます。すべての整数型は、ビットフィールドとして使用できます。

ビットフィールドの配置順 (6.5.2.1)

ビットフィールドは、整数内で最下位ビットから最上位ビットの順でアラインメントされます。

記憶単位の境界をまたぐビットフィールド (6.5.2.1)

ビットフィールドは、選択したビットフィールド整数型の記憶単位の境界をまたぐことはできません。

列挙型を表現するために選択される整数型 (6.5.2.2)

特定の列挙型用に選択される整数型は、列挙型用に定義された列挙定数によって異なります。最小の整数型が選択されます。

修飾子

volatile オブジェクトへのアクセス (6.5.3)

volatile で修飾された型のオブジェクトへの参照は、すべてアクセスされます。

宣言子

宣言子数の上限 (6.5.4)

宣言子の個数には上限はありません。個数は、空きメモリ容量にのみ制限されます。

文

case 文の上限数 (6.6.4.2)

switch 文での case 文（case 値）の個数には、上限はありません。個数は、空きメモリ容量にのみ制限されます。

プリプロセッサディレクティブ

文字定数と条件の指定 (6.8.1)

プリプロセッサディレクティブで使用されている文字集合は、実行文字集合と同一です。プリプロセッサは、プレーン char が `signed` char として扱われる場合、負の char を認識します。

角括弧で括ったファイル指定 (6.8.2)

角括弧で括ったファイル指定の場合、プリプロセッサは親ファイルのディレクトリを検索しません。親ファイルは、`#include` ディレクティブを含むファイルになります。その代わりに、コンパイラのコマンドラインで指定したディレクトリで最初にファイル検索を実行します。

引用符で括ったファイル指定 (6.8.2)

引用符で括ったファイル指定の場合、プリプロセッサのディレクトリ検索は、親ファイルのディレクトリでまず実行され、その後でそれより上位の階層のファイルのディレクトリに対して順に実行されます。したがって、処理中のソースファイルを含むディレクトリと相対的に検索が行われます。親よりも上位の階層のファイルがなく、ファイルが見つからなかった場合は、角括弧で括ってファイル名を指定した場合と同様に検索が続行されます。

文字シーケンス (6.8.2)

プリプロセッサディレクティブは、エスケープシーケンスを除き、ソース文字集合を使用します。したがって、インクルードファイルのパスを指定する場合は、次のように円記号を 1 つだけ使用します。

```
#include "mydirectory¥myfile"
```

ソースコード内では、次のように円記号が 2 つ必要です。

```
file = fopen("mydirectory¥¥myfile","rt");
```

認識できるプラグマディレクティブ (6.8.6)

*プラグマディレクティブ*で説明したプラグマディレクティブ以外にも、以下のディレクティブも認識されます。ただし、これらの影響は不定です。プラグマディレクティブが*プラグマディレクティブ*とここの両方に記載されている場合、ここの情報よりも*プラグマディレクティブ*の章の情報が優先します。

```
alignment

baseaddr

building_runtime

can_instantiate

codeseg

context_handler

cspy_support

define_type_info

do_not_instantiate

early_dynamic_initialization

function

function_effects

hdrstop

important_typedef

instantiate

keep_definition

library_default_requirements

library_provides

library_requirement_override
```

memory

module_name

no_pch

once

system_include

warnings

Default __DATE__ and __TIME__ (6.8.8)

`__TIME__`、`__DATE__` の定義は常に使用可能です。

IAR DLIB ライブラリ関数

以下の内容は、選択したランタイムライブラリ構成がファイル記述子をサポートしている場合にのみ有効です。ランタイムライブラリ構成の詳細については、*DLIB ランタイム環境*を参照してください。

NULL マクロ (7.1.6)

`NULL` マクロは、`0` に定義されています。

assert 関数で出力される診断 (7.2)

`assert()` 関数で出力される診断は、以下のとおりです。

filename: *linenr expression* -- assertion failed

この診断は、パラメータがゼロに評価される場合に出力されます。

定義域エラー (7.5.1)

数学関数で定義域エラーが発生すると、`NaN`（非数）が返されます。

浮動小数点値のアンダフロー時に ERANGE に設定する errno (7.5.1)

数学関数は、アンダーフロー範囲エラー発生時に、整数式 `errno` を `ERANGE`（`errno.h` で定義されているマクロ）に設定します。

fmod 関数の機能 (7.5.6.4)

`fmod()` の 2 番目の引数がゼロの場合、関数は `NaN` を返します。`errno` は `EDOM` に設定されます。

signal 関数 (7.7.1.1)

シグナル関連のライブラリはサポートされていません。

注 : 低レベルインタフェース関数はライブラリには存在しますが、これらの関数は何も実行しません。アプリケーション固有のシグナル処理を実装する場合は、テンプレートソースコードを使用してください。135 ページの *signal と raise* を参照してください。

行を終了する文字 (7.9.2)

`stdout` ストリーム関数は、`newline` または `end of file (EOF)` のどちらかを改行文字として認識します。

空白行 (7.9.2)

`stdout` ストリームで改行文字直前に書き込まれた空白文字は保持されます。`stdout` ストリームで書き込まれた行を `stdin` ストリームで読み取る方法はありません。

バイナリストリームに書き込まれたデータへの NULL 文字の追加 (7.9.2)

バイナリストリームにライトされたデータには、NULL 文字は追加されません。

ファイル (7.9.3)

追加モードのストリームのファイル位置インジケータが最初にファイルの先頭または末尾に配置されているかどうかは、低レベルファイルルーチンのアプリケーション固有の実装に依存します。

テキストストリームへのライトにより、対応するファイルでその書込み位置以降が切り捨てられるかどうかは、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。131 ページの *ファイル I/O* を参照してください。

ファイルのバッファ化の特徴は、アンバッファされたファイル、ラインバッファされたファイル、完全にバッファされたファイルが実装でサポートされるということです。

ゼロ長のファイルが実際に存在するかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

有効なファイル名の作成規則は、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

同じファイルを同時に何度も開くことができるかどうかは、下位レベルのファイルルーチンのアプリケーション固有の実装によって異なります。

remove 関数 (7.9.4.1)

開いたファイルに対する削除処理の結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。131 ページの *ファイル I/O* を参照してください。

rename 関数 (7.9.4.2)

ファイルの名前を既存のファイル名に変更した結果は、アプリケーションごとの低レベルファイルルーチンの実装によって異なります。131 ページの *ファイル I/O* を参照してください。

printf 関数の %p (7.9.6.1)

`printf()` の `%p` 変換指定子（出力ポインタ）の引数は、`void *` 型として処理されます。値は、`%x` 変換指定子の場合と同様に、16 進数として出力されます。

scanf 関数の %p (7.9.6.2)

`scanf()` の `%p` 変換指定子（スキャンポインタ）は、16 進数を読み取り、`void *` 型の値に変換します。

scanf 関数の読み込み範囲 (7.9.6.2)

-（ダッシュ）文字は、常に範囲記号として処理されます。

ファイル位置エラー (7.9.9.1, 7.9.9.4)

ファイル位置エラー発生時に、関数 `fgetpos` および `ftell` store `EFPOS` を `errno` に格納します。

perror 関数で生成されるメッセージ (7.9.10.4)

生成されるメッセージは、次のとおりです。

`usersuppliedprefix`:*`errormessage`*

ゼロバイトのメモリ割り当て (7.10.3)

`calloc()`、`malloc()`、`realloc()` の各関数では、引数としてゼロを指定できます。メモリが割り当てられ、そのメモリへの有効なポインタが返され、メモリブロックを後で `realloc` を使用して修正できます。

abort 関数の動作 (7.10.4.1)

abort() 関数では、ストリームバッファをフラッシュしません。また、ファイル処理はサポートされていないため、ファイル処理は実行しません。

exit 関数の動作 (7.10.4.3)

exit 関数では、main 関数が cstartup に返した値が引数として引き渡されます。

環境 (7.10.4.4)

使用可能な環境名 / 環境リストの変更方法については、134 ページの*環境の操作*を参照してください。

system 関数 (7.10.4.5)

コマンドプロセッサの動作は、system 関数の実装によって異なります。134 ページの*環境の操作*を参照してください。

strerror 関数が返すメッセージ (7.11.6.2)

strerror() が返すメッセージは、次のとおりです。

引数	メッセージ
EZERO	no error
EDOM	domain error
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 || >99	unknown error
その他	error *nnn*

表 56: strerror() が返すメッセージ— IAR DLIB ライブラリ

タイムゾーン (7.12.1)

ローカルの時間帯と夏時間の実装については、135 ページの*時間*を参照してください。

clock 関数 (7.12.2.1)

システムクロックがカウントを開始する地点は、clock 関数の実装によって異なります。135 ページの*時間*を参照してください。

A

abort
  C89 における処理系定義の動作（DLIB）. 471
  システム終了（DLIB）. 124
  処理系定義の動作. 457
algorithm（STL ヘッダファイル）. 365
alignment（プラグマディレクティブ）. 452, 467
__ALIGNOF__（演算子）. 170
--all（ielfdump オプション）. 424
ANSI C. C89 を*参照*
argv（引数）、処理系定義の動作 446
asm、__asm（言語拡張）. 145
assert.h（DLIB ヘッダファイル）. 364
__assignment_by_bitwise_copy_allowed、
ライブラリで使用されるシンボル. 369
atexit. 139
  呼出し用空間の予約 100
atexit 制限、設定 100
@（演算子）
  セクション内への配置 209
  絶対アドレスに配置 207
auto、イニシャライザのパッキングアルゴリズム. . . 381

B

bank（プラグマディレクティブ）. 327
Barr, Michael 35
baseaddr（プラグマディレクティブ）. 452, 467
__BASE_FILE__（定義済シンボル）. 354
basic_template_matching（プラグマディレ
クティブ）. 327
  使用. 187
--bin（ielftool オプション）. 425
bitfields（プラグマディレクティブ）. 328
bool（データ型）. 296
  サポートを追加、DLIB 364, 366
__break（組込み関数）. 350
BRK（アセンブラ命令）. 350
.bss（ELF セクション）. 400
.bssf（ELF セクション）. 400
.bssf_unit64kp（ELF セクション）. 397
.bssf.noinit（ELF セクション）. 400
.bss.noinit（ELF セクション）. 400
building_runtime（プラグマディレクティブ）. . . 452, 467
__BUILD_NUMBER__（定義済シンボル）. 354
Burrows-Wheeler アルゴリズム、
パッキングイニシャライザ. 381
bwt、イニシャライザのパッキングアルゴリズム . . . 381

C

C/C++ 呼出し規約。呼出し規約
__CALLING_CONVENTION__（定義済シンボル）. . 354
--calling_convention（コンパイラオプション）. 242
calloc（ライブラリ関数）. 71
  ヒープも*参照*
  C89 における処理系定義の動作（DLIB）. 470
__callt（拡張キーワード）. 314
CALLT（アセンブラ命令）. 314
.callt0（ELF セクション）. 400
can_instantiate（プラグマディレクティブ）. 452, 467
cassert（ライブラリヘッダファイル）. 366
cctype（DLIB ヘッダファイル）. 366
cerrno（DLIB ヘッダファイル）. 366
cexit（システム終了コード）
  システム終了のカスタマイズ 125
CFI（アセンブラディレクティブ）. 162
cfloat（DLIB ヘッダファイル）. 366
char 型、処理系定義の動作. 447
--char_is_signed（コンパイラオプション）. 242
--char_is_unsigned（コンパイラオプション）. 243
char（データ型）. 296
  signed と unsigned. 297
  デフォルト表現の変更（--char_is_signed）. 242
  処理系定義の動作. 447
  表現の変更（--char_is_unsigned）. 243
cinttypes（DLIB ヘッダファイル）. 366

class メモリ（拡張 EC++） 180
climits（DLIB ヘッダファイル） 366
clocale（DLIB ヘッダファイル） 366
clock.c 135
clock（DLIB ライブラリ関数）、
C89 における処理系定義の動作 471
clock（ライブラリ関数）
処理系定義の動作 458
__close（DLIB ライブラリ関数） 131
clustering（コンパイラ変換） 215
disabling（--no_clustering） 258
cmath（DLIB ヘッダファイル） 366
codeseg（プラグマディレクティブ） 452, 467
__CODE_MODEL__（定義済シンボル） 354
--code_section（コンパイラオプション） 243
--code_section（コンパイラオプション 243
.comment（ELF セクション） 399
complex.h（ライブラリヘッダファイル） 364
complex（ライブラリヘッダファイル） 365
--config（リンカオプション） 275
configuration
基本的なプロジェクト設定 57
__low_level_init 125
リンカの設定ファイル。リンカ設定ファイルを*参照*
--config_def（リンカオプション） 276
--config_search（リンカオプション） 276
const
オブジェクトの宣言 307
非トップレベル 173
.const（ELF セクション） 401
.constf（ELF セクション） 401
.consth（ELF セクション） 401
__constrange（）、ライブラリで
使用されるシンボル 369
__construction_by_bitwise_copy_allowed、
ライブラリで使用されるシンボル. 369
constseg（プラグマディレクティブ） 328
const_cast（キャスト演算子） 178
context_handler（プラグマディレクティブ） 452, 467
__CORE__（定義済シンボル） 354
core
RL78 0（旧名） 44
RL78 1（旧名） 44
RL78 2（旧名） 44
selecting（--core） 244
コマンドラインで指定 244
特定. 354
--core（コンパイラオプション） 244
cosf（ライブラリルーチン） 137–138
cosl（ライブラリルーチン） 137–138
cos（ライブラリルーチン） 137–138
cos（ライブラリ関数） 362
__COUNTER__（定義済シンボル） 355
__cplusplus（定義済シンボル） 355
--cpp_init_routine（リンカオプション） 277
--create（iarchive オプション） 428
csetjmp（DLIB ヘッダファイル） 366
csignal（DLIB ヘッダファイル） 366
cspy_support（プラグマディレクティブ） 452, 467
CSTACK（ELF ブロック） 401
サイズの設定 99
スタックも参照
cstartup（システム起動コード）
システム初期化のカスタマイズ 125
ソースファイル（DLIB） 122
cstdarg（DLIB ヘッダファイル） 366
cstdbool（DLIB ヘッダファイル） 366
cstddef（DLIB ヘッダファイル） 366
cstdio（DLIB ヘッダファイル） 367
cstdlib（DLIB ヘッダファイル） 367
cstring（DLIB ヘッダファイル） 367
ctime（DLIB ヘッダファイル） 367
ctype.h（ライブラリヘッダファイル） 364
cwctype.h（ライブラリヘッダファイル） 367
C ヘッダファイル. 363
C 言語、概要 167
C_INCLUDE（環境変数） 227
C-SPY
C++ のデバッグサポート 185

システム終了用のインタフェース 125
デバッグサポートを含める 116
C/C++ のリンケージ 152
C++
Embedded C++、拡張 Embedded C++ *も参照*
サポート 43
ヘッダファイル 364
言語拡張 190
呼出し規約 149
静的メンバ変数 209
絶対アドレス 209
標準テンプレートライブラリ（STL）. 365
C++ オブジェクト、メモリタイプへの配置 67
C++ スタイルのコメント 167
C++ ヘッダファイル 365
C++ 用語 36
C89
サポート 167
処理系定義の動作 461
--c89（コンパイラオプション） 241
C89 におけるバイナリストリーム（DLIB） 469
C90.C89 *を参照*
C94.C89 *を参照*
C99. *標準 C を参照*

D

-D（コンパイラオプション） 245
-d（iarchive オプション）. 429
data
さまざまな記憶方法 61
記憶 61
配置 206, 295, 330, 397
絶対アドレス 207
配置、extern として宣言 208
表現 295
.data（ELF セクション）. 401
.dataf（ELF セクション） 402
.dataf_init（ELF セクション） 402
dataseg（プラグマディレクティブ）. 330
data_alignment（プラグマディレクティブ）. 329
.data_init（ELF セクション） 402
__DATA_MODEL__（定義済シンボル）. 355
.data_unit64kp（ELF セクション）. 398
__DATE__（定義済シンボル）. 355
date（ライブラリ関数）、サポートの設定 135
DC32（アセンブラディレクティブ） 146
--debug_lib（リンカオプション） 277
.debug（ELF セクション） 399
define block（リンカディレクティブ）. 378
define memory（リンカディレクティブ）. 373
define overlay（リンカディレクティブ） 379
define region（リンカディレクティブ） 373
define symbol（リンカディレクティブ） 391
--define_symbol（リンカオプション）. 278
define_type_info（プラグマディレクティブ） . . . 452, 467
--delete（iarchive オプション）. 429
delete 演算子（拡張 EC++）. 183
delete（キーワード）. 71
--dependencies（コンパイラオプション）. 246
--dependencies（リンカオプション）. 278
deque（STL ヘッダファイル）. 365
DI（アセンブラ命令） 350
--diagnostics_tables（コンパイラオプション）. 249
--diagnostics_tables（リンカオプション）. 280
diag_default（プラグマディレクティブ）. 332
--diag_error（コンパイラオプション） 247
--diag_error（リンカオプション） 279
--no_fragments（コンパイラオプション） 260
--no_fragments（リンカオプション） 286
diag_error（プラグマディレクティブ） 332
--diag_remark（コンパイラオプション） 248
--diag_remark（リンカオプション） 279
diag_remark（プラグマディレクティブ）. 333
--diag_suppress（コンパイラオプション） 248
--diag_suppress（リンカオプション） 280
diag_suppress（プラグマディレクティブ）. 333
--diag_warning（コンパイラオプション）. 249

--diag_warning（リンカオプション）. 280
diag_warning（プラグマディレクティブ）. 333
--disable_div_mod_instructions（コンパイラオプション）. 249
__disable_interrupt（組込み関数）. 350
--discard_unused_publics（コンパイラオプション）. 250
DIVHU 命令、無効化. 249
DIVH 命令、無効化. 249
DLIB. 363
デバッグサポートを含める 116
ドキュメント. 34
ランタイム環境. 109
リファレンス情報。オンラインヘルプシステムを参照. 361
構成. 126
設定. 110, 250
命名規約. 37
--dlib_config（コンパイラオプション）. 250
DLib_Defaults.h（ライブラリ設定ファイル）. . . 121, 126
__DLIB_FILE_DESCRIPTOR（構成シンボル）. 131
do not initialize（リンカディレクティブ）. 383
--double（コンパイルオプション）. 251
double（データ型）. 299
回避. 203
浮動小数点型のサイズ設定 59
do_not_instantiate（プラグマディレクティブ）. . 453, 467

E

-e（コンパイラオプション）. 251
early_initialization（プラグマディレクティブ）. . 453, 467
--ec++（コンパイラオプション）. 252
--edit（isymexport オプション）. 429
--eec++（コンパイラオプション）. 252
EI（アセンブラ命令）. 350
ELF ユーティリティ 409
Embedded C++. 177
C++ との違い. 178
概要. 177
関数リンケージ. 152
言語拡張. 177
有効. 252
Embedded C++ Technical Committee 36
__embedded_cplusplus（定義済シンボル）. 355
__enable_interrupt（組込み関数）. 350
--enable_multibytes（コンパイラオプション）. 253
--enable_restrict（コンパイラオプション）. 253
--entry（リンカオプション）. 281
enums
データ表現. 297
前方宣言. 172
EQU（アセンブラディレクティブ）. 267
ERANGE 454
ERANGE（C89）. 468
error（リンカディレクティブ）. 394
--error_limit（コンパイラオプション）. 253
--error_limit（リンカオプション）. 281
error（プラグマディレクティブ）. 334
_Exit（ライブラリ関数）. 125
exit（ライブラリ関数）. 124
C89 における処理系定義の動作 471
処理系定義の動作. 457
_exit（ライブラリ関数）. 124
__exit（ライブラリ関数）. 124
expf（ライブラリルーチン）. 137
expl（ライブラリルーチン）. 137
export キーワード、拡張 EC++ から除外 185
--export_builtin_config（リンカオプション）. 282
export（リンカディレクティブ）. 392
exp（ライブラリルーチン）. 137
extended-selectors（リンカ設定ファイル）. 389
extern "C" リンケージ. 182
--extract（iarchive オプション）. 430

F

-f（IAR ユーティリティオプション）. 430
-f（コンパイラオプション）. 254

-f（リンカオプション）. 282
__far（拡張キーワード）. 314
Far（コードモデル）. 74
Far（データモデル）. 69
far（メモリタイプ）. 63
__far（拡張キーワード）. 301
__far_func（拡張キーワード）. 315
__far_func（関数ポインタ）. 301
FAR_HEAP（セクション）. 403
__FAR_RUNTIME_ATTRIBUTE__
（定義済シンボル）. 356
fdopen、stdio.h 367
fegettrapdisable. 367
fegettrapenable 367
FENV_ACCESS、処理系定義の動作 450
fenv.h（ライブラリヘッダファイル）. 364
　C の追加機能 367
fgetpos（ライブラリ関数）、
C89 における処理系定義の動作. 470
fgetpos（ライブラリ関数）、処理系定義の動作 457
__FILE__（定義済シンボル）. 356
filename
　オブジェクトファイル 265, 289
　オブジェクト実行可能イメージ 289
　デバイス記述ファイルの拡張子 44
　パラメータとして指定 237
　ヘッダファイルの拡張子 44
　検索手順. 227
fileno、stdio.h 367
--fill（ielftool オプション）. 431
float.h（ライブラリヘッダファイル）. 364
float（データ型）. 299
FLT_EVAL_METHOD、処理系定義の
動作 449, 454, 458
FLT_ROUNDS、処理系定義の動作 449, 458
fmod（ライブラリ関数）、
C89 における処理系定義の動作. 468
--force_output（リンカオプション）. 282
for ループ、宣言. 167
FP_CONTRACT、処理系定義の動作. 450
free（ライブラリ関数）*ヒープも 参照* 71
fsetpos（ライブラリ関数）、処理系定義の動作 457
fstream（ライブラリヘッダファイル）. 365
ftell（ライブラリ関数）、処理系定義の動作. 457
　C89 470
Full DLIB（ライブラリ構成）. 126
__func__（定義済シンボル）. 174, 356
__FUNCTION__（定義済シンボル）. 174, 356
functional（STL ヘッダファイル）. 365
function_effects（プラグマディレクティブ）. . . . 453, 467
function（プラグマディレクティブ）. 453, 467

G

--generate_callt_runtime_library_calls
（コンパイラオプション）. 254
--generate_far_runtime_library_calls
（コンパイラオプション）. 254
--generate_vfe_header（isymexport オプション）. 431
getenv（ライブラリ関数）、サポートの設定. 134
getw、stdio.h 368
getzone.c. 135
getzone（ライブラリ関数）、サポートの設定. 135
__get_interrupt_level（組込み関数）. 350
__get_interrupt_state（組込み関数）. 350
GRP_COMDAT、グループタイプ 417
--guard_calls（コンパイラオプション）. 255

H

__halt（組込み関数）. 351
Harbison, Samuel P. 35
hash_map（STL ヘッダファイル）. 365
hash_set（STL ヘッダファイル）. 365
__has_constructor、ライブラリで
使用されるシンボル 369
__has_destructor、ライブラリで
使用されるシンボル 369
.hbss（ELF セクション）. 403

.hbss.noinit（ELF セクション） 403
.hdata（ELF セクション）. 403
.hdata_init（ELF セクション） 404
hdrstop（プラグマディレクティブ）. 453, 467
--header_context（コンパイラオプション） 255
hide（isymexport ディレクティブ）. 421
__huge（拡張キーワード）. 315
HUGE_HEAP（セクション）. 404

I

-I（コンパイラオプション） 255
iarbuild.exe（ユーティリティ） 121
iarchive 409
    コマンドの概要. 410
    オプションの概要. 411
IAR コマンドラインビルドユーティリティ 121
IAR システムズの技術サポート. 233
IAR 言語の概要 43
__iar_cos_accurate（ライブラリルーチン） 138
__iar_cos_accuratef（ライブラリルーチン）. 138
__iar_cos_accuratef（ライブラリ関数）. 362
__iar_cos_accuratel（ライブラリルーチン）. 138
__iar_cos_accuratel（ライブラリ関数）. 362
__iar_cos_small（ライブラリルーチン）. 137
__iar_cos_smallf（ライブラリルーチン） 137
__iar_cos_smalll（ライブラリルーチン） 137
__iar_exp_small（ライブラリルーチン). 137
__iar_exp_smallf（ライブラリルーチン） 137
__iar_exp_smalll（ライブラリルーチン） 137
__iar_FPow（ライブラリルーチン） 138
__iar_FSin（ライブラリルーチン）. 137
__iar_log_small（ライブラリルーチン）. 137
__iar_log_smallf（ライブラリルーチン） 137
__iar_log_smalll（ライブラリルーチン） 137
__iar_log10_small（ライブラリルーチン）. 137
__iar_log10_smallf（ライブラリルーチン） 137
__iar_log10_smalll（ライブラリルーチン） 137
__iar_LPow（ライブラリルーチン）. 138
__iar_LSin（ライブラリルーチン）. 137–138
__iar_maximum_atexit_calls 100
__iar_Pow（ライブラリルーチン） 138
__iar_Pow_accurate（ライブラリルーチン）. 138
__iar_pow_accurate（ライブラリルーチン）. 138
__iar_Pow_accuratef（ライブラリルーチン） 138
__iar_pow_accuratef（ライブラリルーチン） 138
__iar_pow_accuratef（ライブラリ関数）. 362
__iar_Pow_accuratel（ライブラリルーチン） 138
__iar_pow_accuratel（ライブラリルーチン） 138
__iar_pow_accuratel（ライブラリ関数） 362
__iar_pow_small（ライブラリルーチン） 137
__iar_pow_smallf（ライブラリルーチン） 137
__iar_pow_smalll（ライブラリルーチン） 137
__iar_program_start（ラベル） 123
__iar_Sin（ライブラリルーチン）. 137–138
__iar_Sin_accurate（ライブラリルーチン） 138
__iar_sin_accurate（ライブラリルーチン）. 138
__iar_Sin_accuratef（ライブラリルーチン）. 138
__iar_sin_accuratef（ライブラリルーチン） 138
__iar_sin_accuratef（ライブラリ関数） 362
__iar_Sin_accuratel（ライブラリルーチン）. 138
__iar_sin_accuratel（ライブラリルーチン） 138
__iar_sin_accuratel（ライブラリ関数） 362
__iar_Sin_small（ライブラリルーチン）. 137
__iar_sin_small（ライブラリルーチン） 137
__iar_Sin_smallf（ライブラリルーチン）. 137
__iar_sin_smallf（ライブラリルーチン） 137
__iar_Sin_smalll（ライブラリルーチン） 137
__iar_sin_smalll（ライブラリルーチン） 137
__IAR_SYSTEMS_ICC__（定義済シンボル）. 356
__iar_tan_accurate（ライブラリルーチン）. 138
__iar_tan_accuratef（ライブラリルーチン）. 138
__iar_tan_accuratef（ライブラリ関数）. 362
__iar_tan_accuratel（ライブラリルーチン） 138
__iar_tan_accuratel（ライブラリ関数） 362
__iar_tan_small（ライブラリルーチン）. 137
__iar_tan_smallf（ライブラリルーチン） 137
__iar_tan_smalll（ライブラリルーチン） 137

.iar.debug（ELF セクション）. 399
.iar.dynexit（ELF セクション）. 404
__ICCRL78__（定義済シンボル）. 357
IDE
　ビルドツールの概要 41
　ライブラリのビルド 121
IEEE フォーマット、浮動小数点数値 299
ielfdump 414
　オプションの概要 415
ielftool 413
　オプションの概要 414
if（リンカディレクティブ）. 395
--ihex（ielftool オプション）. 432
ILINK オプション . リンカオプションを参照
ILINK「リンカ」*を参照してください。*
--image_input（リンカオプション）. 283
important_typedef（プラグマディレクティブ）. . 453, 467
include_alias（プラグマディレクティブ）. 334
include（リンカディレクティブ）. 395
infinity（出力形式）、処理系定義の動作 456
initialization
　C++ 動的. 93
　デフォルトの変更 100
　パッキングアルゴリズム 101
　手動. 101
　単一の値. 174
　動的. 122
　無効化. 101
initialize（リンカディレクティブ）. 380
.init_array（セクション）. 404
inline（プラグマディレクティブ）. 335
instantiate（プラグマディレクティブ）. 453, 467
Intel hex 193
__interrupt（拡張キーワード）. 76, 316
　プラグマディレクティブで使用 347
intptr_t（整数型）. 303
intrinsics.h（ヘッダファイル）. 349
inttypes.h（ライブラリヘッダファイル）. 364
.intvec（セクション）. 405
int（データ型）signed および unsigned 296
iobjmanip 416
　オプションの概要 416
iomanip（ライブラリヘッダファイル）. 365
iosfwd（ライブラリヘッダファイル）. 365
iostream（ライブラリヘッダファイル）. 365
ios（ライブラリヘッダファイル）. 365
iso646.h（ライブラリヘッダファイル）. 364
istream（ライブラリヘッダファイル）. 365
isymexport 418
　オプションの概要 420
iterator（STL ヘッダファイル）. 366
I/O レジスタ . SFR を参照

J

Josuttis, Nicolai M.. 35

K

--keep（リンカオプション）. 283
keep_definition（プラグマディレクティブ）. . . . 453, 467
keep（リンカディレクティブ）. 384
Kernighan, Brian W.. 35

L

-l（コンパイラオプション）. 256
　スケルトンコードの作成 148
Labrosse, Jean J.. 35
Lajoie, Jos 仔 35
language（プラグマディレクティブ）. 336
Lempel-Ziv-Welch アルゴリズム、
パッキングイニシャライザ. 381
library_default_requirements（プラグマディレク
ティブ）. 453, 467
library_provides（プラグマディレクティブ）. . . . 453, 467
library_requirement_override（プラグマディレク
ティブ）. 453, 467

lightbulb アイコン、本ガイドの 37
limits.h（ライブラリヘッダファイル）. 364
__LINE__（定義済シンボル） 357
Lippman, Stanley B. 35
list（STL ヘッダファイル）. 366
locale.h（ライブラリヘッダファイル） 364
location（プラグマディレクティブ） 207, 336
logf（ライブラリルーチン） 137
logl（ライブラリルーチン） 137
--log_file（リンカオプション） 285
log（ライブラリルーチン） 137
log10f（ライブラリルーチン） 137
log10l（ライブラリルーチン）. 137
log10（ライブラリルーチン） 137
long double（データ型） 299
long float（データ型）、double の同義語 173
long long（データ型）
　回避. 203
long long（データ型）signed および unsigned 297
longjmp、使用の制限. 363
long（データ型）signed および unsigned 297
__low_level_init 123
　カスタマイズ. 125
　初期化フェーズ. 53
low_level_init.c. 122
__lseek（ライブラリ関数） 131
lzw、イニシャライザのパッキングアルゴリズム . . . 381
lz77、イニシャライザのパッキングアルゴリズム. . . 381

M

__mach（組込み関数） 351
MACH（アセンブラ命令)、挿入. 351
__machu（組込み関数）. 351
MACHU（アセンブラ命令)、挿入 351
--macro_positions_in_diagnostics（コンパイラオプション） 257
main（関数）
　処理系定義の動作. 446
　定義（C89）. 461
malloc（ライブラリ関数）
　C89 における処理系定義の動作 470
　ヒープも*参照*. 71
--mangled_names_in_messages（リンカオプション). . 285
Mann, Bernhard 35
--map（リンカオプション）. 285
map（STL ヘッダファイル）. 366
math.h（ライブラリヘッダファイル） 364
MB_LEN_MAX、処理系定義の動作 458
__memory_of
　ライブラリで使用されるシンボル 369
　演算子. 181
memory（STL ヘッダファイル）. 366
memory（プラグマディレクティブ） 453, 468
-merge_duplicate_sections（リンカオプション). 286
message（プラグマディレクティブ） 337
Meyers, Scott 35
--mfc（コンパイラオプション） 257
--misrac_verbose（コンパイラオプション）. 239
--misrac_verbose（リンカオプション）. 274
--misrac1998（コンパイラオプション） 239
--misrac1998（リンカオプション） 274
--misrac2004（コンパイラオプション） 239
--misrac2004（リンカオプション） 274
MISRA-C、ドキュメント 34
module_name（プラグマディレクティブ). 453, 468
__monitor（拡張キーワード） 317
Motorola S-records. 193
mutable 属性、拡張 EC++ 178, 189

N

NaN
　実装. 301
　処理系定義の動作. 456
NDEBUG（プリプロセッサシンボル） 358

__near（拡張キーワード）. 317
Near（コードモデル）. 74
Near（データモデル）. 68
near（メモリタイプ）. 63
__near（拡張キーワード）. 301
--near_const_location（コンパイラオプション） 258
__near_func（拡張キーワード）. 317
__near_func（関数ポインタ）. 301
NEAR_HEAP（セクション）. 405
new 演算子（拡張 EC++）. 183
new（キーワード）. 71
new（ライブラリヘッダファイル） 365
__noreturn（拡張キーワード）. 320
Normal DLIB（ライブラリ構成） 126
__no_alloc（拡張キーワード）. 318
__no_alloc_str（演算子）. 319
__no_alloc_str16（演算子）. 319
__no_alloc16（拡張キーワード）. 318
__no_bit_access（拡張キーワード）. 319
--no_clustering（コンパイラオプション）. 258
--no_code_motion（コンパイラオプション） 258
--no_cross_call（コンパイラオプション） 259
--no_cse（コンパイラオプション）. 259
--no_dwarf3_cfi（コンパイラオプション）. 259
__no_init（拡張キーワード）. 221, 319
--no_inline（コンパイラオプション）. 260
-- no_library_search（リンカオプション）. 287
--no_locals（リンカオプション）. 287
__no_operation（組込み関数）. 351
--no_path_in_file_macros（コンパイラオプション） . . 260
no_pch（プラグマディレクティブ）. 453, 468
--no_range_reservations（リンカオプション). 288
--no_remove（リンカオプション） 288
__no_save（拡張キーワード） 320
--no_scheduling（コンパイラオプション）. 261
--no_size_constraints（コンパイラオプション）. 261
--no_static_destruction（コンパイラオプション）. 261
--no_strtab（ielfdump オプション） 432
--no_system_include（コンパイラオプション） 262
--no_tbaa（コンパイラオプション）. 262
--no_typedefs_in_diagnostics（コンパイラオプション） 262
--no_unroll（コンパイラオプション）. 263
--no_vfe（リンカオプション）. 288
--no_warnings（コンパイラオプション） 263
--no_warnings（リンカオプション） 289
no_workseg（プラグマディレクティブ） 338
--no_wrap_diagnostics（コンパイラオプション）. 263
--no_wrap_diagnostics（リンカオプション）. 289
NULL
C89 における処理系定義の動作（DLIB）. 468
ポインタ定数、C 規格の緩和 173
処理系定義の動作. 455
numeric（STL ヘッダファイル） 366

O

-o（コンパイラオプション）. 264
-o（iarchive オプション） 432
-o（ielfdump オプション） 432
-o（コンパイラオプション）. 265
-o（リンカオプション）. 289
object_attribute（プラグマディレクティブ） 221, 338
once（プラグマディレクティブ）. 453, 468
--only_stdout（コンパイラオプション） 264
--only_stdout（リンカオプション） 289
__open（ライブラリ関数）. 131
optimize（プラグマディレクティブ）. 339
.option_byte（セクション）. 405
Oram, Andy 35
ostream（ライブラリヘッダファイル） 365

P

packbits、イニシャライザのパッキングアルゴリズム 381
packing、イニシャライザのアルゴリズム. 381
pack（プラグマディレクティブ）. 304, 340

--parity（ielftool オプション）. 433
--pending_instantiations（コンパイラオプション）. . . . 265
perror（ライブラリ関数）、
C89 における処理系定義の動作. 470
place at（リンカディレクティブ）. 384
place in（リンカディレクティブ） 385
--place_holder（リンカオプション） 290
powf（ライブラリルーチン）. 137–138
powl（ライブラリルーチン）. 137–138
pow（ライブラリルーチン） 137–138
  代替の実装. 362
_Pragma（プリプロセッサ演算子）. 167
--predef_macro（コンパイラオプション） 265
--preinclude（コンパイラオプション）. 266
.preinit_array（セクション）. 405
--preprocess（コンパイラオプション）. 266
__PRETTY_FUNCTION__（定義済シンボル） 357
__printf_args（プラグマディレクティブ） 341
printf（ライブラリ関数）. 114
  C89 における処理系定義の動作 470
  フォーマッタの選択 114
  構成シンボル. 129
  処理系定義の動作. 456
__program_start（ラベル） 123
ptrdiff_t（整数型）. 303
--public_equ（コンパイラオプション）. 267
public_equ（プラグマディレクティブ）. 341
PUBLIC（アセンブラディレクティブ）. 267
putenv（ライブラリ関数）、DLIB には存在しない . . 134
putw、stdio.h 368

Q

QCCRL78（環境変数）. 227
queue（STL ヘッダファイル）. 366

R

-r（iarchive オプション） 437
-r（コンパイラオプション） 246
raise.c 135
raise（ライブラリ関数）、サポートの設定 135
RAM
  イニシャライザを ROM からコピー. 55
  コードの実行. 103
  メモリの節約. 217
  領域の宣言の例. 89
--ram_reserve_ranges（isymexport オプション） 434
--raw（ielfdump オプション）. 435
__read（ライブラリ関数） 131
  カスタマイズ. 127
realloc（ライブラリ関数） 71
  C89 における処理系定義の動作 470
  ヒープも*参照*
--redirect（リンカオプション） 290
reinterpret_cast（キャスト演算子） 178
--relaxed_fp（コンパイラオプション）. 267
.rela（ELF セクション） 399
.rel（ELF セクション）. 399
--remove_file_path（iobjmanip オプション）. 435
--remove_section（iobjmanip オプション） 436
remove（ライブラリ関数）. 131
  C89 における処理系定義の動作（DLIB）. 470
  処理系定義の動作. 456
remquo、規模 454
--rename_section（iobjmanip オプション） 436
--rename_symbol（iobjmanip オプション） 437
rename（isymexport ディレクティブ） 422
rename（ライブラリ関数）. 131
  C89 における処理系定義の動作（DLIB）. 470
  処理系定義の動作. 456
--replace（iarchive オプション）. 437
__ReportAssert（ライブラリ関数）. 139
required（プラグマディレクティブ）. 341
--require_prototypes（コンパイラオプション） 268

--reserve_ranges（isymexport オプション）438
Ritchie, Dennis M. 35
RL78
  サポートされているデバイス 44
  機能、サポートするもの 47
RL78 0 コア（旧名）44
RL78 1 コア（旧名）44
RL78 2 コア（旧名）44
ROM から RAM、コピー 103
__ro_placement（拡張キーワード）321
rrno.h（ライブラリヘッダファイル）364
rtmodel（アセンブラディレクティブ）141
rtmodel（プラグマディレクティブ）342
rtti のサポート、STL から除外 179

S

-S（iarchive オプション）439
-s（ielfdump オプション）438
__saddr（拡張キーワード）321
saddr（メモリタイプ）62
.sbss（ELF セクション）406
.sbss.noinit（ELF セクション）406
__scanf_args（プラグマディレクティブ）343
scanf（ライブラリ関数）
  C89 における処理系定義の動作（DLIB）470
  フォーマッタの選択（DLIB）115
  構成シンボル 129
  処理系定義の動作 457
scheduling（コンパイラ変換）216
  無効 261
.sdata（ELF セクション）406
.sdata_init（ELF セクション）406
--search（リンカオプション）291
sections 397
  概要 397
  指定（--code_section）243
  宣言（#pragma section）343
  定義 87
__section_begin（拡張演算子）171
__section_end（拡張演算子）171
__section_size（拡張演算子）171
section-selectors（リンカ設定ファイル）387
.security_id（セクション）407
segment（プラグマディレクティブ）343
--self_reloc（ielftool オプション）439
setjmp.h（ライブラリヘッダファイル）364
setlocale（ライブラリ関数）133
__set_interrupt_level（組込み関数）351
__set_interrupt_state（組込み関数）352
set（STL ヘッダファイル）366
SFR
  特殊機能レジスタへのアクセス 220
  特殊機能レジスタを extern として宣言 208
__sfr（拡張キーワード）322
SFR（メモリタイプ）63
short（データ型）296
Show（isymexport ディレクティブ）421
.shstrtab（ELF セクション）399
signal.c 135
signal（ライブラリ関数）
  C89 における処理系定義の動作 469
  サポートの設定 135
  処理系定義の動作 454
signed char（データ型）296–297
  指定 242
signed int（データ型）296
signed long long（データ型）297
signed long（データ型）297
signed short（データ型）296
signed の値、回避 203
--silent（iarchive オプション）439
--silent（ielftool オプション）439
--silent（コンパイラオプション）268
--silent（リンカオプション）292
--simple（ielftool オプション）440
--simple-ne（ielftool オプション）440
sinf（ライブラリルーチン）137–138

sinl（ライブラリルーチン）. 137–138
sin（ライブラリルーチン）. 137–138
sin（ライブラリ関数）. 362
size_t（整数型）. 302
slist（STL ヘッダファイル）. 366
smallest、イニシャライザのパッキングアル
ゴリズム 381
sprintf（ライブラリ関数）. 114
フォーマッタの選択 114
--srec（ielftool オプション）. 440
--srec-len（ielftool オプション）. 441
--srec-s3only（ielftool オプション）. 441
sscanf（ライブラリ関数）
フォーマッタの選択（DLIB）. 115
sstream（ライブラリヘッダファイル）. 365
stack（STL ヘッダファイル）. 366
static clustering（コンパイラ変換）. 215
static_assert（）. 170
static_cast（キャスト演算子）. 178
stdbool.h（ライブラリヘッダファイル）. 297, 364
__STDC__（定義済シンボル）. 357
STDC CX_LIMITED_RANGE
（プラグマディレクティブ）. 344
STDC FENV_ACCESS
（プラグマディレクティブ）. 344
STDC FP_CONTRACT
（プラグマディレクティブ）. 345
__STDC_VERSION__（定義済シンボル）. 357
stddef.h（ライブラリヘッダファイル）. 298, 364
stderr. 131, 264, 289
stdin 131
C89 における処理系定義の動作（DLIB）. 469
stdint.h（ライブラリヘッダファイル）. 364, 367
stdio.h、C の追加機能 367
stdio.h（ライブラリヘッダファイル）. 364
stdout 131, 264, 289
C89 における処理系定義の動作（DLIB）. 469
処理系定義の動作 455
std 名前空間、EC++、拡張 EC++ から除外 189
Steele, Guy L. 35
STL. 186
__stop（組込み関数）. 352
strcasecmp、string.h. 368
strdup、string.h 368
streambuf（ライブラリヘッダファイル）. 365
strerror（ライブラリ関数）、C89 における
処理系定義の動作（DLIB）. 471
strerror（ライブラリ関数）、処理系定義の動作 460
--strict（コンパイラオプション）. 269
string.h、C の追加機能. 368
string.h（ライブラリヘッダファイル）. 364
string（ライブラリヘッダファイル）. 365
--strip（ielftool オプション）. 441
--strip（iobjmanip オプション）. 441
--strip（リンカオプション）. 292
strncasecmp、string.h. 368
strnlen、string.h 368
Stroustrup, Bjarne 35
strstream（ライブラリヘッダファイル）. 365
.strtab（ELF セクション）. 399
strtod（ライブラリ関数）、サポートの設定 136
Sutter, Herb. 35
.switch（セクション）. 407
.switchf（セクション）. 407
.symtab（ELF セクション）. 399
system 関数、処理系定義の動作. 447, 457
--system_include_dir（コンパイラオプション）. 269
system_include（プラグマディレクティブ）. . . . 453, 468
system（ライブラリ関数）
C89 における処理系定義の動作（DLIB）. 471
サポートの設定 134

T

-t（iarchive オプション）. 443
tanf（ライブラリルーチン）. 137–138
tanl（ライブラリルーチン）. 137–138
tan（ライブラリルーチン）. 137–138
tan（ライブラリ関数）. 362

.text（ELF セクション）........................407
.textf（ELF セクション）........................407
.text_unit64kp（ELF セクション）..................399
tgmath.h（ライブラリヘッダファイル）............364
this（ポインタ）.................................149
class メモリ..................................180
クラスオブジェクトへの参照....................180
__TIME__（定義済シンボル）.....................358
time zone
（ライブラリ関数）、C89 における処理系定義の
動作...471
time zone（ライブラリ関数）、処理系定義の動作...457
__TIMESTAMP__（定義済シンボル）..............358
time.c..135
time.h（ライブラリヘッダファイル）..............364
C の追加機能..................................368
time32（ライブラリ関数）、サポートの設定.......135
time64（ライブラリ関数）、サポートの設定.......135
--titxt（ielftool オプション）...................442
--toc（iarchive オプション）....................443
typedefs
繰返し..173
診断から除外..................................262
type_attribute（プラグマディレクティブ）..........345

U

uchar.h（ライブラリヘッダファイル）.............364
uintptr_t（整数型）..............................303
__ungetchar, stdio.h.............................368
unroll（プラグマディレクティブ）.................346
unsigned char（データ型）....................296–297
signed char に変更............................242
unsigned int（データ型）.........................296
unsigned long long（データ型）...................297
unsigned long（データ型）........................297
unsigned short（データ型）.......................296
use init table（リンカディレクティブ）.............386
--use_c++_inline（コンパイラオプション）.........269
--use_unix_directory_separators
（コンパイラオブジェクト）.........................270
utility（STL ヘッダファイル）.....................366

V

-V（iarchive オプション）........................443
.vector（ELF セクション）........................399
vector（STL ヘッダファイル）.....................366
vector（プラグマディレクティブ）.............76, 347
--verbose（iarchive オプション）.................443
--verbose（ielftool オプション）..................443
--vfe（リンカオプション）........................292
--vla（コンパイラオプション）.....................270
void、ポインタ...................................173
volatile
const、オブジェクトの宣言.....................307
アクセス規則..................................306
オブジェクトの宣言............................305
同時にアクセスされる変数の保護...............219
V1 呼出し規約....................................151
__v1_call（拡張キーワード）.....................322
V2 呼出し規約....................................151
__v2_call（拡張キーワード）.....................322

W

#warning message（プリプロセッサ拡張）..........359
--warnings_affect_exit_code
（コンパイラオプション）....................230, 271
--warnings_affect_exit_code（リンカオプション）....293
--warnings_are_errors（コンパイラオプション）.....271
--warnings_are_errors（リンカオプション）.........293
warnings（プラグマディレクティブ）..........453, 468
--warn_about_c_style_casts
（コンパイラオプション）.........................270
wchar_t（データ型）、AC でのサポートの追加.....297
wchar.h（ライブラリヘッダファイル）.........364, 367
wctype.h（ライブラリヘッダファイル）............364

__weak（拡張キーワード）. 323
weak（プラグマディレクティブ）. 347
Web サイト、推奨 35
--whole_archive（リンカオプション）. 293
workseg 領域 70
--workseg_area（コンパイラオプション）. 271
__write（ライブラリ関数）. 131
　カスタマイズ. 127
__write_array, in stdio.h 368
__write_buffered（DLIB ライブラリ関数）. 117
.wrkseg（ELF セクション）. 408

X

-x（iarchive オプション）. 430
xreportassert.c. 139

Z

zeros、イニシャライザのパッキングアルゴリズム . . 381

あ

アサート関数 138
　C89 における処理系定義の動作、(DLIB) 468
　出力に含める 358
　処理系定義の動作 453
アセンブラコード
　C からの呼出し 147
　C++ から呼び出す. 149
　インライン挿入. 145
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 146
　呼出しフレーム情報 162
アセンブララベル
　public 化（--public_equ）. 267
　アプリケーション起動時のデフォルト設定 . . . 57, 99
　先頭に追加の下線 145
アセンブラリストファイル、生成. 256
アセンブラ言語インタフェース. 143
　呼び出し規約。アセンブラコードを参照
アセンブラ出力ファイル. 149
アセンブラ命令
　CALLT 314
　インライン挿入. 145
アトミック処理 76
　__monitor 317
アドレスメモリタイプ、データモデル、
およびコードモデルを参照
アプリケーション
　ビルド、概要. 57
　起動と終了（DLIB）. 122
　実行、概要. 53
アラインメント 295
　インラインアセンブラの制約 146
　オブジェクト（__ALIGNOF__）. 170
　データ型. 296
　厳密に設定（#pragma data_alignment）. 329
　構造体（#pragma pack）. 340
　構造体、問題の原因 204
アンダーフローエラー、処理系定義の動作. 454
アンダーフローの errno 値、処理系定義の動作. 457
アンダーフロー範囲エラー、
C89 における処理系定義の動作. 468

い

イニシャライザ、静的 173
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 266
　指定. 227
インストール先ディレクトリ. 36
インターナルエラー 233
インラインアセンブラ 145
　アセンブラ言語インタフェース*も参照*
　回避. 217
インライン関数 167
　コンパイラ 214

う

ウィンドウ
　サポートされない場合 120
　使用可能にする（DLIB） 117
ウィンドウ、デバッグサポートに含める 117
エスケープシーケンス、処理系定義の動作 447
エラーメッセージ 232
　range 105
　コンパイラ用の分類 247
　リンカ用の分類 279
　分類 260, 286
エラーリターンコード 229
エリアエラー、処理系定義の動作 454
エントリラベル、プログラム 123

お

オブジェクトファイルの検索パス（--search） 291
オブジェクトファイル名、指定（-O） 265, 289
オブジェクトファイル、リンカの検索パス
（--search） 291
オブジェクト属性 312
オプションパラメータ 235
オプション、iarchiveiarchive オプションを参照
オプション、ielfdump.ielfdump のオプションを参照
オプション、ielftool.ielftool のオプションを参照
オプション、iobjmanipiobjmanip オプションを*参照*
オプション、isymexportisymexport 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*
オーバヘッド、削減 214

か

ガイドラインの確認 31
ガイドライン、確認 31
カーニハン & リッチー関数宣言 218
　不許可 268

き

キャスト
　ポインタと整数 302
　整数へのポインタ、言語拡張 173
キャスト演算子
　Embedded C++ から削除 178
　拡張 EC++. 178, 188
キャラクタベース I/O 127
キーワード 309
　拡張、概要 45
キーワードの制限の有効化 253
キーワードの制限、有効 253

く

クラステンプレートの部分的特化照合
（拡張 EC++） 186
クロスコール（コンパイラ変換） 216
グローバル配列、アクセス 159
グローバル変数
　アクセス 159
　グローバル変数 215
　システム起動中の初期化 123
　システム終了時に処理 124
　使用しないためのヒント 217

こ

このガイドで使用されている規則 36
コマンドプロンプト アイコン、本ガイド 37
コマンドラインオプション
　また リンカオプションを参照
　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 225
　リンカ呼出し構文のパート 226
　受渡し 226
　表記規則 36

コメント
C++ スタイル、C コードで使用 167
プリプロセッサディレクティブ後 173
コンパイラ
環境変数 227
呼出し構文 225
出力元 228
コンパイラオブジェクトファイル 50
コンパイラからの出力 228
(--debug, -r) にデバッグ情報を含める 246
コンパイラオプション 235
コンパイラへの受渡し 226
ファイル (-f) からの読取り 254
パラメータの指定 237
概要 238
構文 235
スケルトンコードの作成 148
命令スケジューリング 216
--generate_far_runtime_library_calls 254
--warnings_affect_exit_code 230
コンパイラでのハードウェアサポート 109
コンパイラのサブバージョン番号 358
コンパイラのバージョン番号 358
コンパイラプラットフォーム、特定 356
コンパイラリスト、生成 (-l) 256
コンパイラ最適化レベル 212
コンパイラ変換 210
コンパイル
コマンドラインから 57
構文 225
コンパイル日
正確な時刻 (__TIME__) 358
(__DATE__) の特定 355
コンピュータスタイル、表記規則 36
コード
コードの生成を円滑化 216
実行 59
実行の割込み 75
--code (ielfdump オプション) 428
コードメモリ、データ配置 321
コードモデル 73
configuration 59
Far 74
Near 74
関数の呼び出し 158
特定 (__CODE_MODEL__) 354
コード移動 (コンパイラ変換) 214
無効化 (--no_code_motion) 258
コールスタック 162

さ

サポート、技術 233

し

シグナル、処理系定義の動作 446
システム起動時 446
システムの終了「システムの終了」を*参照*
システム起動
DLIB 122
カスタマイズ 125
初期化フェーズ 53
システム終了
C-SPY のインタフェース 125
DLIB 124
ショートアドレス作業エリア。workseg 領域を*参照*
シンボル
プリプロセッサ、定義 245, 278
ローカル、ELF イメージから削除 287
参照の変更 290
出力に含める 341
定義済シンボルの概要 45
匿名、作成 167
--symbols (iarchive オプション) 442
シンボル名、先頭に追加の下線 145

す

スカラ以外のパラメータ、回避 217
スクラッチレジスタ 152
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 147
スタック 69
  size 194
  エリアの節約 217
  クリーン 155, 242, 322
  サイズの設定 99
  レイアウト 154
  使用の利点、問題点 70
  内容 69
  保持するブロック 401
スタックパラメータ 153–154
スタックポインタ 69
スタックポインタレジスタ、注意事項 153
ステアリングファイル、isymexport への入力 420
ストリーム
  Embedded C++ でサポート 178
  処理系定義の動作 446

せ

--section（ielfdump オプション） 438
セマフォ
  C の例 76
  C++ の例 78
  処理 317

そ

ソースファイル、すべての参照先のリスト 255

た

タイプ定義、メモリ記憶の指定に使用 65
［ターミナル I/O］ウィンド 117, 120
C-SPY
  ［ターミナル I/O］ウィンドウ、
  デバッグサポートに含める 117
ターミナル I/O、デバッガのランタイムインタ
フェース 117
ターミナル出力、高速化 117

ち

チェックサム
  C-SPY でのシンボルの表示フォーマット 201
  計算 197
--checksum（ielftool オプション） 425

つ

ツールアイコン、本ガイド 37

て

ディレクティブ
  プラグマ 45, 325
  リンカ 371
ディレクトリ、パラメータとして指定 237
デストラクタおよび割込み、使用 184
デバイス記述ファイル、C-SPY 用に事前定義 44
--debug（コンパイラオプション） 246
デバッグ情報、オブジェクトファイルに含める 246
テンプレートのサポート
  Embedded C++ から削除 178
  拡張 EC++ 178, 185
データブロック（呼出しフレーム情報） 163
データポインタ 301
データメモリ属性、使用 63
データモデル 68
  configuration 58
  Far 69
  Near 68
  （__DATA_MODEL__）の特定 355

データ型 296
C++ 308
signed の回避 203
整数型. 296
浮動小数点数 299

と

ドキュメント
ガイドの概要 33
内容. 32
本ガイドの使用方法 31
本ガイドの対象者 31
トラップベクタ、プラグマディレクティブでの
指定 347

ね

ネイティブ環境、処理系定義の動作. 459

は

バイト内のビット、処理系定義の動作. 447
バイナリストリーム 455
バックトレース情報、呼出しフレーム情報 *も参照*
パック構造体型 304
バッチファイル
エラーリターンコード 229
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） 121
パラメータ
function 153
register 153–154
スカラ以外、回避 217
スタック 153–154
ファイルまたはディレクトリを
指定する場合の規則 237
隠し 154
指定. 237
表記規則 36
バンク、切替え 327
バージョン
コンパイラ（__VER__） 358
コンパイラのサブバージョン番号 358
使用中の C 規格の識別（__STDC_VERSION__）. 357
本ガイド 2
ハードウェア積算 / 除算ユニット 139

ひ

ビットフィールド
C89 における処理系定義の動作 465
データ表現. 298
ヒント 203
処理系定義の動作 450
非標準型. 170
ビット否定 219
ヒント
円滑なコードの生成 216
効率的なデータ型の使用 203
処理系定義の動作 450
ヒント、プログラミング. 216
ヒープ
DLIB サポート. 139
VLA の割当て 270
データ記憶. 62
動的メモリ 71
ヒープサイズ
デフォルトの変更 100
標準 I/O. 195
ヒープセクション
DLIB 194
配置. 100
ヒープ（サイズがゼロ）、処理系定義の動作 457

ふ

ファイルのバッファ処理、処理系定義の動作......455
ファイルパス、#include ファイル用パスの指定.....255
ファイル位置、処理系定義の動作................455
ファイル依存関係、追跡.......................246
ファイル名（有効）、処理系定義の動作...........455
ファイル、処理系定義の動作
　マルチバイト文字..........................456
　一時ファイルの処理........................456
　開く....................................456
ファイル（ゼロ長）、処理系定義の動作...........455
フォーマット
　標準 IEEE（浮動小数点数）...................299
　浮動小数点数値............................299
プラグマディレクティブ.......................45
　概要....................................325
　basic_template_matching、使用.................187
　pack..................................304, 340
　絶対配置データ............................207
　認識されたすべての一覧.....................452
　認識されたすべての一覧（C89）...............467
プリプロセッサ
　演算子（_Pragma）.........................167
　出力....................................266
プリプロセッサシンボル.......................354
　定義..................................245, 278
プリプロセッサディレクティブ
　C89 における処理系定義の動作...............466
　終了後のコメント..........................173
　処理系定義の動作..........................451
　#pragma.................................325
プリプロセッサ拡張
　__VA_ARGS__.............................167
　#warning message..........................359
プログラミングのヒント.......................216
プログラムエントリラベル.....................123
プログラム終了、処理系定義の動作..............446
プロジェクト
　ライブラリのためのセットアップ..............121
　基本設定..................................57
プロセッサコア、サポート.....................58
プロセッサ処理
　アクセス................................143
　低レベル..............................168, 349
プロトタイプ、強制..........................268

へ

ヘッダファイル
　C.......................................363
　C++..................................364–365
　ライブラリ..............................361
　STL.....................................365
　bool での stdbool.h のインクルード.............297
　DLib_Defaults.h........................121, 126
　wchar_t での stddef.h のインクルード...........298
　特殊機能レジスタ..........................220
ヘッダ名、処理系定義の動作...................451

ほ

ポインタ
　C89 における処理系定義の動作...............464
　data.....................................301
　function.................................301
　キャスト................................302
　処理系定義の動作..........................450
ポインタ型.................................301
　ちがい...................................66
　混在....................................173
ポリモフィズム、Embedded C++.................177

ま

マクロ
ERANGE（in errno.h）454, 468
NULL、処理系定義の動作 455
C89 における DLIB 468
アサートのインクルード 358
可変数引数 167
#pragma optimize への埋め込み 339
#pragma ディレクティブで代用 168
マップファイル、生成 285
マルチバイト文字のサポート 253
マルチバイト文字、処理系定義の動作 447, 459

め

メッセージ
強制 337
無効 268, 292
メモリ
C++ での解放 71
C++ での割当て 71
RAM、節約 217
アクセス 62, 159
saddr 方法を使用 160
データ 16 方法の使用 160–161
グローバル / 静的変数で使用 61
スタック 69
節約 217
ヒープ 71
動的 71
非初期化 221
メモリタイプ 62
C++ 67
SFR 63
ポインタ 65
概要 63
構造体 66
指定 63
変数の配置 67
メモリマップ
SFR の初期化 125
リンカ設定 95
生成（--map）285
隣家からの出力 230
メモリ管理、型安全 177
メモリ配置
タイプ定義の使用 65
メンバ関数、ポインタ 189

も

モジュールの互換性 140
rtmodel 342
モジュール、概要 84
モニタ関数 76, 317
モード変更、処理系定義の動作 456

ゆ

ユーティリティ（ELF）409

ら

ライブラリ
ビルド済 111
使用の理由 51
標準テンプレートライブラリ 365
ライブラリオブジェクトファイル 361
ライブラリオプション、設定 60
ライブラリドキュメント 361
ライブラリファイルの検索パス（--search）291
ライブラリファイル、検索パス（--search）291
ライブラリプロジェクト、テンプレートを使用したビルド 121
ライブラリヘッダファイル 361
ライブラリモジュール
オーバライド 120

概要 84
ライブラリ関数
一覧、DLIB 363
オンラインヘルプ 35
ライブラリ機能、Embedded C++ から削除 178
ライブラリ設定ファイル
DLIB 126
DLib_Defaults.h 121, 126
指定 250
修正 121
ラベル 174
アセンブラ、public 化 267
アセンブラ、先頭に追加の下線 145
__iar_program_start 123
__program_start 123
ランタイムの型情報、Embedded C++ から削除 178
ランタイムモデル属性 140
ランタイムモデル定義 342
ランタイムライブラリ
IDE から設定 60
コマンドラインからの設定 60
ランタイムライブラリ（DLIB）
概要 361
システム起動コードのカスタマイズ 125
ビルド済 111
ファイル名の構文 112
モジュールのオーバライド 120
リビルドなしでのカスタマイズ 113
ランタイム環境
DLIB 109
オプションの設定 60
設定（DLIB） 110

り

リエントラント性（DLIB） 362
リスト、生成 256
リターンアドレス 156
リターン値、関数 155
リテラル、複合 167
--remarks（コンパイラオプション） 268
--remarks（リンカオプション） 291
リマーク（診断メッセージ） 232
コンパイラで有効化 268
コンパイラ用の分類 248
リンカで有効化 291
リンカ用の分類 279
リンカ 83
出力元 230
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 289
リンカオプション 273
ファイル（-f）からの読取り 282
概要 273
表記規則 36
リンカ設定ファイル
コードおよびデータの配置 87
概要 371
詳細 371
選択 95
リンク
コマンドラインから 57
ビルド処理 51
プロセス 85
概要 83
リンケージ、C/C++ 152
リードオンリーメモリ、データ配置 321

る

ルーチン、時間が重要 143, 168, 349
__root（拡張キーワード） 320
ループ展開（コンパイラ変換） 214
無効 263
#pragma unroll 346
ループ不変式 214

れ

レジスタ
- C89 における処理系定義の動作 465
- アセンブラレベルルーチン 150
- スクラッチ 152
- パラメータへの割当て 154
- 関数のリターン 155
- 呼出し先保存、スタックに格納 69
- 保護 153

レジスタキーワード、処理系定義の動作 450
レジスタパラメータ 153–154
レジスタバンク、切替え 327
レジスタ変数作業領域。workseg 領域*を参照*

ろ

--log（リンカオプション）. 284
ロケール
- サポート 132
- サポートの削除 133
- ライブラリでのサポートを追加 133
- 実行中のロケール変更 133
- 処理系定義の動作 448, 459

ローカルシンボル、ELF イメージから削除 287
ローカル変数、自動変数*を参照*

記号

_Exit（ライブラリ関数）. 125
_exit（ライブラリ関数）. 124
_low_level_init、カスタマイズ 125
__ALIGNOF__（演算子）. 170
__asm（言語拡張）. 145
__assignment_by_bitwise_copy_allowed、ライブラリで使用されるシンボル 369
__BASE_FILE__（定義済シンボル）. 354
__break（組込み関数）. 350
__BUILD_NUMBER__（定義済シンボル）. 354
__CALLING_CONVENTION__（定義済シンボル）. . 354
__callt runtime library calls、生成. 254
__callt（拡張キーワード）. 314
__close（ライブラリ関数）. 131
__CODE_MODEL__（定義済シンボル）. 354
__constrange（）、ライブラリで使用されるシンボル 369
__construction_by_bitwise_copy_allowed、ライブラリで使用されるシンボル. 369
__CORE__（定義済シンボル）. 354
__COUNTER__（定義済シンボル）. 355
__cplusplus（定義済シンボル）. 355
__DATA_MODEL__（定義済シンボル）. 355
__DATE__（定義済シンボル）. 355
__disable_interrupt（組込み関数）. 350
__DLIB_FILE_DESCRIPTOR（構成シンボル）. 131
__embedded_cplusplus（定義済シンボル）. 355
__enable_interrupt（組込み関数）. 350
__exit（ライブラリ関数）. 124
__far（拡張キーワード）. 301, 314
__far_func（拡張キーワード）. 315
__far_func（関数ポインタ）. 301
__FAR_RUNTIME_ATTRIBUTE__（定義済シンボル）. 356
__FILE__（定義済シンボル）. 356
__FUNCTION__（定義済シンボル）. 174, 356
__func__（定義済シンボル）. 174, 356
__gets、stdio.h 367
__get_interrupt_level（組込み関数）. 350
__get_interrupt_state（組込み関数）. 350
__halt（組込み関数）. 351
__has_constructor、ライブラリで使用されるシンボル 369
__has_destructor、ライブラリで使用されるシンボル 369
__huge（拡張キーワード）. 302, 315
__iar_cos_accuratef（ライブラリルーチン）. 138
__iar_cos_accuratel（ライブラリルーチン）. 138
__iar_cos_accurate（ライブラリルーチン）. 138
__iar_cos_smallf（ライブラリルーチン）. 137

__iar_cos_smalll（ライブラリルーチン）. 137
__iar_cos_small（ライブラリルーチン）. 137
__iar_exp_smallf（ライブラリルーチン）. 137
__iar_exp_smalll（ライブラリルーチン）. 137
__iar_exp_small（ライブラリルーチン）. 137
__iar_FPow（ライブラリルーチン）. 138
__iar_FSin（ライブラリルーチン）. 137–138
__iar_log_smallf（ライブラリルーチン）. 137
__iar_log_smalll（ライブラリルーチン）. 137
__iar_log_small（ライブラリルーチン）. 137
__iar_log10_smallf（ライブラリルーチン）. 137
__iar_log10_smalll（ライブラリルーチン）. 137
__iar_log10_small（ライブラリルーチン）. 137
__iar_LPow（ライブラリルーチン）. 138
__iar_LSin（ライブラリルーチン）. 137–138
__iar_maximum_atexit_calls 100
__iar_Pow_accuratef（ライブラリルーチン）. 138
__iar_pow_accuratef（ライブラリルーチン）. 138
__iar_pow_accuratel（ライブラリルーチン）. 138
__iar_Pow_accurate（ライブラリルーチン）. 138
__iar_pow_accurate（ライブラリルーチン）. 138
__iar_pow_smallf（ライブラリルーチン）. 137
__iar_pow_smalll（ライブラリルーチン）. 137
__iar_pow_small（ライブラリルーチン）. 137
__iar_Pow（ライブラリルーチン）. 138
__iar_program_start（ラベル）. 123
__iar_Sin_accuratef（ライブラリルーチン）. 138
__iar_sin_accuratef（ライブラリルーチン）. 138
__iar_Sin_accuratel（ライブラリルーチン）. 138
__iar_sin_accuratel（ライブラリルーチン）. 138
__iar_Sin_accurate（ライブラリルーチン）. 138
__iar_sin_accurate（ライブラリルーチン）. 138
__iar_Sin_smallf（ライブラリルーチン）. 137
__iar_sin_smallf（ライブラリルーチン）. 137
__iar_Sin_smalll（ライブラリルーチン）. 137
__iar_sin_smalll（ライブラリルーチン）. 137
__iar_Sin_small（ライブラリルーチン）. 137
__iar_sin_small（ライブラリルーチン）. 137
__iar_Sin（ライブラリルーチン）. 137–138
__IAR_SYSTEMS_ICC__（定義済シンボル）. 356
__iar_tan_accuratef（ライブラリルーチン）. 138
__iar_tan_accuratel（ライブラリルーチン）. 138
__iar_tan_accurate（ライブラリルーチン）. 138
__iar_tan_smallf（ライブラリルーチン）. 137
__iar_tan_smalll（ライブラリルーチン）. 137
__iar_tan_small（ライブラリルーチン）. 137
__ICCRL78__（定義済シンボル）. 357
__interrupt（拡張キーワード）. 76, 316
　プラグマディレクティブで使用 347
__intrinsic（拡張キーワード）. 316
__LINE__（定義済シンボル）. 357
__low_level_init 123
　初期化フェーズ 53
__lseek（ライブラリ関数）. 131
__mach（組込み関数）. 351
__machu（組込み関数）. 351
__memory_of
　ライブラリで使用されるシンボル 369
　演算子 181
__monitor（拡張キーワード）. 317
__near（拡張キーワード）. 301, 317
__near_func（拡張キーワード）. 317
__near_func（関数ポインタ）. 301
__near_size_t 184
__noreturn（拡張キーワード）. 320
__no_alloc（拡張キーワード）. 318
__no_alloc_str（演算子）. 318–319
__no_alloc_str16（演算子）. 318–319
__no_alloc16（拡張キーワード）. 318
__no_bit_access（拡張キーワード）. 319
__no_init（拡張キーワード）. 221, 319
__no_operation（組込み関数）. 351
__no_save（拡張キーワード）. 320
__open（ライブラリ関数）. 131
__PRETTY_FUNCTION__（定義済シンボル）. 357
__printf_args（プラグマディレクティブ）. 341
__program_start（ラベル）. 123
__read（ライブラリ関数）. 131

カスタマイズ 127
__ReportAssert（ライブラリ関数） 139
__root（拡張キーワード） 320
__ro_placement（拡張キーワード） 321
__saddr（拡張キーワード） 321
__scanf_args（プラグマディレクティブ） 343
__section_begin（拡張演算子） 171
__section_end（拡張演算子） 171
__section_size（拡張演算子） 171
__set_interrupt_level（組込み関数） 351
__set_interrupt_state（組込み関数） 352
__sfr（拡張キーワード） 322
__STDC_VERSION__（定義済シンボル） 357
__STDC__（定義済シンボル） 357
__stop（組込み関数） 352
__TIMESTAMP__（定義済シンボル） 358
__TIME__（定義済シンボル） 358
__ungetchar, stdio.h 368
__VA_ARGS__（プリプロセッサ拡張） 167
__v1_call（拡張キーワード） 322
__v2_call（拡張キーワード） 322
__weak（拡張キーワード） 323
__write_array, in stdio.h 368
__write_buffered（DLIB ライブラリ関数） 117
__write（ライブラリ関数） 131
カスタマイズ 127
-d（iarchive オプション） 429
-D（コンパイラオプション） 245
-e（コンパイラオプション） 251
-f（IAR ユーティリティオプション） 430
-f（コンパイラオプション） 254
-f（リンカオプション） 282
-I（コンパイラオプション） 255
-l（コンパイラオプション） 256
スケルトンコードの作成 148
-o（iarchive オプション） 432
-o（ielfdump オプション） 432
-o（コンパイラオプション） 264–265
-o（リンカオプション） 289
-r（iarchive オプション） 437
-r（コンパイラオプション） 246
-S（iarchive オプション） 439
-s（ielfdump オプション） 438
-t（iarchive オプション） 443
-V（iarchive オプション） 443
-x（iarchive オプション） 430
-- no_library_search（リンカオプション） 287
--all（ielfdump オプション） 424
--bin（ielftool オプション） 425
--calling_convention（コンパイラオプション） 242
--char_is_signed（コンパイラオプション） 242
--char_is_unsigned（コンパイラオプション） 243
--checksum（ielftool オプション） 425
--code_model（コンパイラオプション） 243, 245
--code_section（コンパイラオプション） 243
--code（ielfdump オプション） 428
--config_def（リンカオプション） 276
--config_search（リンカオプション） 276
--config（リンカオプション） 275
--core（コンパイラオプション） 244
--cpp_init_routine（リンカオプション） 277
--create（iarchive オプション） 428
--c89（コンパイラオプション） 241
--debug_lib（リンカオプション） 277
--debug（コンパイラオプション） 246
--define_symbol（リンカオプション） 278
--delete（iarchive オプション） 429
--dependencies（コンパイラオプション） 246
--dependencies（リンカオプション） 278
--diagnostics_tables（コンパイラオプション） 249
--diagnostics_tables（リンカオプション） 280
--diag_error（コンパイラオプション） 247
--diag_error（リンカオプション） 279
--diag_remark（コンパイラオプション） 248
--diag_remark（リンカオプション） 279
--diag_suppress（コンパイラオプション） 248
--diag_suppress（リンカオプション） 280
--diag_warning（コンパイラオプション） 249

--diag_warning（リンカオプション）. 280
--disable_div_mod_instructions
（コンパイラオプション）. 249
--discard_unused_publics（コンパイラオプション）. . 250
--dlib_config（コンパイラオプション）. 250
--double（コンパイルオプション）. 251
--ec++（コンパイラオプション）. 252
--edit（isymexport オプション）. 429
--eec++（コンパイラオプション）. 252
--enable_multibytes（コンパイラオプション）. 253
--enable_restrict（コンパイラオプション）. 253
--entry（リンカオプション）. 281
--error_limit（コンパイラオプション）. 253
--error_limit（リンカオプション）. 281
--export_builtin_config（リンカオプション）. 282
--extract（iarchive オプション）. 430
--fill（ielftool オプション）. 431
--force_output（リンカオプション）. 282
--generate_callt_runtime_library_calls
（コンパイラオプション）. 254
--generate_far_runtime_library_calls
（コンパイラオプション）. 254
--generate_vfe_header（isymexport オプション）. 431
--guard_calls（コンパイラオプション）. 255
--header_context（コンパイラオプション）. 255
--ihex（ielftool オプション）. 432
--image_input（リンカオプション）. 283
--keep（リンカオプション）. 283
--log_file（リンカオプション）. 285
--log（リンカオプション）. 284
--macro_positions_in_diagnostics
（コンパイラオプション）. 257
--mangled_names_in_messages
（リンカオプション）. 285
--map（リンカオプション）. 285
--merge_duplicate_sections（リンカオプション）. 286
--mfc（コンパイラオプション）. 257
--misrac_verbose（コンパイラオプション）. 239
--misrac_verbose（リンカオプション）. 274
--misrac1998（コンパイラオプション）. 239
--misrac1998（リンカオプション）. 274
--misrac2004（コンパイラオプション）. 239
--misrac2004（リンカオプション）. 274
--near_const_location（コンパイラオプション）. 258
--no_clustering（コンパイラオプション）. 258
--no_code_motion（コンパイラオプション）. 258
--no_cross_call（コンパイラオプション）. 259
--no_cse（コンパイラオプション）. 259
--no_dwarf3_cfi（コンパイラオプション）. 259
--no_fragments（コンパイラオプション）. 260
--no_fragments（リンカオプション）. 286
--no_inline（コンパイラオプション）. 260
--no_locals（リンカオプション）. 287
--no_path_in_file_macros（コンパイラオプション）. . 260
--no_range_reservations（リンカオプション）. 288
--no_remove（リンカオプション）. 288
--no_scheduling（コンパイラオプション）. 261
--no_size_constraints（コンパイラオプション）. 261
--no_static_destruction（コンパイラオプション）. 261
--no_strtab（ielfdump オプション）. 432
--no_system_include（コンパイラオプション）. 262
--no_typedefs_in_diagnostics
（コンパイラオプション）. 262
--no_unroll（コンパイラオプション）. 263
--no_vfe（リンカオプション）. 288
--no_warnings（コンパイラオプション）. 263
--no_warnings（リンカオプション）. 289
--no_wrap_diagnostics（コンパイラオプション）. 263
--no_wrap_diagnostics（リンカオプション）. 289
--only_stdout（コンパイラオプション）. 264
--only_stdout（リンカオプション）. 289
--output（iarchive オプション）. 432
--output（ielfdump オプション）. 432
--output（コンパイラオプション）. 265
--output（リンカオプション）. 289
--parity（ielftool オプション）. 433
--pending_instantiations（コンパイラオプション）. . . . 265
--place_holder（リンカオプション）. 290
--predef_macro（コンパイラオプション）. 265
--preinclude（コンパイラオプション）. 266

--preprocess（コンパイラオプション）. 266
--ram_reserve_ranges（isymexport オプション）. 434
--raw（[ielfdump] オプション）. 435
--redirect（リンカオプション）. 290
--relaxed_fp（コンパイラオプション）. 267
--remarks（コンパイラオプション）. 268
--remarks（リンカオプション）. 291
--remove_file_path（iobjmanip オプション）. 435
--remove_section（iobjmanip オプション）. 436
--rename_section（iobjmanip オプション）. 436
--rename_symbol（iobjmanip オプション）. 437
--replace（iarchive オプション）. 437
--require_prototypes（コンパイラオプション）. 268
--reserve_ranges（isymexport オプション）. 438
--search（リンカオプション）. 291
--section（ielfdump オプション）. 438
--self_reloc（ielftool オプション）. 439
--silent（iarchive オプション）. 439
--silent（ielftool オプション）. 439
--silent（コンパイラオプション）. 268
--silent（リンカオプション）. 292
--simple-ne（ielftool オプション）. 440
--simple（ielftool オプション）. 440
--srec-len（ielftool オプション）. 441
--srec-s3only（ielftool オプション）. 441
--srec（ielftool オプション）. 440
--strict（コンパイラオプション）. 269
--strip（ielftool オプション）. 441
--strip（iobjmanip オプション）. 441
--strip（リンカオプション）. 292
--symbols（iarchive オプション）. 442
--system_include_dir（コンパイラオプション）. 269
--titxt（ielftool オプション）. 442
--toc（iarchive オプション）. 443
--use_c++_inline（コンパイラオプション）. 269
--use_unix_directory_separators
（コンパイラオブジェクト）. 270
--verbose（iarchive オプション）. 443
--verbose（ielftool オプション）. 443
--vfe（リンカオプション）. 292
--vla（コンパイラオプション）. 270
--warnings_affect_exit_code
（コンパイラオプション）. 230, 271
--warnings_affect_exit_code（リンカオプション）. . . . 293
--warnings_are_errors（コンパイラオプション）. 271
--warnings_are_errors（リンカオプション）. 293
--warn_about_c_style_casts
（コンパイラオプション）. 270
--whole_archive（リンカオプション）. 293
--workseg_area（コンパイラオプション）. 271
.bssf（ELF セクション）. 400
.bssf_unit64kp（ELF セクション）. 397
.bssf.noinit（ELF セクション）. 400
.bss.noinit（ELF セクション）. 400
.bss（ELF セクション）. 400
.callt0（ELF セクション）. 400
.comment（ELF セクション）. 399
.const（ELF セクション）. 401
.constf（ELF セクション）. 401
.consth（ELF セクション）. 401
.dataf（ELF セクション）. 402
.dataf_init（ELF セクション）. 402
.data_init（ELF セクション）. 402
.data_unit64kp（ELF セクション）. 398
.data（ELF セクション）. 401
.debug（ELF セクション）. 399
.hbss（ELF セクション）. 403
.hbss.noinit（ELF セクション）. 403
.hdata（ELF セクション）. 403
.hdata_init（ELF セクション）. 404
.iar.debug（ELF セクション）. 399
.iar.dynexit（ELF セクション）. 404
.init_array（セクション）. 404
.intvec（セクション）. 405
.option_byte（セクション）. 405
.preinit_array（セクション）. 405
.rela（ELF セクション）. 399
.rel（ELF セクション）. 399
.sbss（ELF セクション）. 406
.sbss.noinit（ELF セクション）. 406

.sdata（ELF セクション）........................ 406
.sdata_init（ELF セクション）.................... 406
.security_id（セクション）...................... 407
.shstrtab（ELF セクション）...................... 399
.strtab（ELF セクション）........................ 399
.switch（セクション）.......................... 407
.switchf（セクション）.......................... 407
.symtab（ELF セクション）........................ 399
.textf（ELF セクション）........................ 407
.text_unit64kp（ELF セクション）.................. 399
.text（ELF セクション）......................... 407
.vector（ELF セクション）........................ 399
.wrkseg（ELF セクション）........................ 408
@（演算子）
　セクション内への配置 209
　絶対アドレスに配置 207
#include ファイル、指定 227, 255
#warning message（プリプロセッサ拡張）.......... 359
%Z 置換文字列、処理系定義の動作 458

数字

16 ビットポインタ、メモリへのアクセス 69
24 ビットポインタ、メモリへのアクセス 69
32 ビット（浮動小数点数フォーマット）.......... 300
64 ビットデータタイプ、回避 203
64 ビット（浮動小数点数フォーマット）.......... 300