



Linux* 版インテル® Fortran コンパイラ
ユーザーズ・ガイド
Vol I: アプリケーションのビルド

[免責条項](#)

資料番号: 253259-002J

目次

免責条項	1
インテル® Fortran コンパイラについて	3
本リリースの新機能	3
バージョン 8.1 の新機能と変更点	3
バージョン 8.0 の新機能と変更点	4
本書の使い方	4
その他のマニュアル	5
表記規則	5
コンパイラの起動	6
コンパイラの起動の概要	6
インテル® Fortran コンパイラの使い方	7
コンパイル・フェーズ	8
前処理フェーズ	9
アセンブラとリンカ	10
アセンブラ	10
リンカ	11
インテル® Fortran コンパイラのデフォルト動作	11
入力ファイルとファイル名拡張子	11
ファイル指定	12
出力ファイル	13
コンパイラまたはリンカにより作成される一時ファイル	14

アプリケーションのビルド	15
アプリケーションのビルドの概要	15
コンパイル処理の制御	16
環境変数の設定と表示	16
設定ファイル環境変数	16
シェル・スクリプトの実行による環境変数の設定	17
インテル® Fortran コンパイラの起動	17
ifort コマンドを使用する	18
make コマンドを使用する	18
ifort コマンドの例	19
複数のファイルのコンパイルおよびリンク	19
リンクの抑止	19
出力ファイル名の変更	19
リンカ・ライブラリの追加	20
モジュール (.mod) ファイルの使用	20
モジュールを持つプログラムのコンパイル	20
マルチディレクトリ・モジュール・ファイルの処理	21
makefile を使用した並列呼び出し	22
インクルード・ファイルと .mod ファイルの検索	23
インクルード・ファイル・パスの指定と除外	24
設定ファイルと応答ファイル	24
設定ファイル	24

設定ファイルのサンプル	25
応答ファイル	25
代替ツールの場所およびオプションの指定	26
-Qlocation を使ってツールの代替場所を指定する	26
-Qoption を使用してオプションをツールに渡す	27
定義済みプリプロセッサ・シンボル	27
プリプロセッサ・シンボルの定義	29
プリプロセッサ・シンボルの抑止	29
コマンドライン出力からファイルへの	30
リダイレクト	30
実行プログラムの作成、実行、デバッグ	30
サンプル・プログラムを作成するためのコマンド	31
サンプル・プログラムの実行	32
サンプル・プログラムのデバッグ	32
共有ライブラリの作成	33
ifort コマンドのみで共有ライブラリを作成する	33
ifort コマンドおよび ld コマンドで共有ライブラリを作成する	34
共有ライブラリの制約	34
共有ライブラリのインストール	35
共通ブロックの割り当て	36
-dyncom オプションの使用ガイドライン	36
ダイナミック共通ブロックを使用する理由	36

ダイナミック共通ブロックへのメモリの割り当て	37
コンパイラ・オプション	37
コンパイラ・オプションの概要.....	37
コンパイラ・オプションの詳細	38
コンパイラ・オプションの形式.....	38
複数の ifort コマンドを使用する	39
OPTIONS 文を使用してオプションを上書きする	40
オプションに関するヘルプを表示する.....	40
オプションに関連するコンパイラ・	40
ディレクティブ	40
コードの生成オプション.....	41
コードの生成オプションの説明	41
-[no]recursive.....	41
-[no]reentrancy [keyword]	41
-sox[-] (IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ).....	42
互換性オプション	42
互換性オプションの説明	42
-1	42
-assume [no]bscc	42
-convert	43
-[no]f77rtl.....	43
-fpscomp all および -fpscomp none.....	44

-fpscomp [no]filesfromcmd	44
-fpscomp [no]general	44
-fpscomp [no]ioformat	44
-fpscomp [no]ldio_spacing	45
-fpscomp [no]libs	45
-fpscomp [no]logicals	45
-prof_format_32 (IA-32 および Itanium® ベース・システムのみ).....	45
-vms.....	46
コンパイル診断オプション	47
コンパイル診断オプションの説明	47
-e90 または -e95	47
-[no]error_limit n.....	48
-openmp_report{0 1 2}	48
-par_report{0 1 2 3}	48
-std, -std90, -std95	49
-vec_report{0 1 2 3 4 5} (IA-32 およびインテル® EM64T システムのみ)	49
-warn all、-warn none、-nowarn	49
-warn [no]alignments	50
-warn [no]declarations	50
-warn [no]errors	50
-warn [no]general	50
-warn [no]ignore_loc	51

-warn [no]stderrs.....	51
-warn [no]truncated_source.....	51
-warn [no]uncalled.....	51
-warn [no]unused.....	51
-warn [no]usage.....	52
データ・オプション	52
データ・オプションの説明	52
-align none	52
-align [no]commons または -align [no]dcommons	52
-align recnbyte	53
-align [no]records.....	53
-align [no]sequence.....	54
-assume [no]byterecl	54
-assume [no]dummy_aliases	54
-assume [no]protect_constants	55
-auto_scalar、-auto、および -save.....	55
-double_size {64 128}.....	56
-dyncom "blk1,blk2,..."	56
-integer_size {16 32 64}.....	57
-pg.....	57
-real_size {32 64 128}.....	57
-safe_cray_ptr.....	58

-zero[-]	59
外部プロシージャ・オプション	59
外部プロシージャ・オプションの説明	59
-assume [no]underscore	59
-[no]mixed_str_len_arg	60
-names keyword	60
浮動小数点オプション	60
浮動小数点オプションの説明	61
-assume [no]minus0	61
-[no]fltconsistency	61
-fp_port (IA-32 システムのみ)	62
-[no]fpconstant	62
-fpen	63
-fpstkchk (IA-32 システムのみ)	63
-fr32 (Itanium ベース・システムのみ)	63
-ftz[-]	64
-IPF_flt_eval_method0 (Itanium ベース・システムのみ)	64
-IPF_ftacc[-] (Itanium ベース・システムのみ)	64
-IPF_fma[-] (Itanium ベース・システムのみ)	64
-IPF_fp_relaxed[-] (Itanium ベース・システムのみ)	65
-IPF_fp_speculationmode (Itanium ベース・システムのみ)	65
-mp1	65

-pc{32 64 80} (IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ).....	65
言語オプション	66
言語オプションの説明	66
-[no]altparam	66
-[no]d_lines	66
-[no]extend_source [size].....	66
-[no]f66	67
-[no]free または -[no]fixed.....	67
-openmp または -openmp_stubs	67
-[no]pad_source	68
ライブラリ・オプション	68
ライブラリ・オプションの説明.....	68
-no_cpprt.....	68
-nodefaultlibs	69
-i_dynamic.....	69
-Ldir.....	69
-[no]threads	69
-nostdlib	69
-shared	70
-shared-libcxa	70
-static-libcxa	70
-static	70

その他のオプション	71
-ansi_alias[-].....	71
-assume cc_omp	71
-assume none	71
-nobss_init	72
-ccdefault keyword.....	72
-debug keyword.....	72
-dryrun.....	73
-dynamic-linkerfile	73
-fpic または -fPIC	73
-fminshared.....	73
-fvisibility=keyword と -fvisibility-keyword=file	74
-g	74
-help	74
-inline_debug_info	75
-[no]logo	75
-nofor_main	75
-noinclude.....	75
-openmp_profile.....	76
-[no]pad	76
-prec_div (IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ).....	76
-rcd (IA-32 システムのみ).....	76

-size_lp64 (Itanium ベース・システムのみ).....	76
-nostartfiles.....	77
-syntax_only.....	77
-T file.....	77
-Tf file.....	77
-tcheck.....	77
-u.....	78
-v.....	78
-V.....	78
-what.....	78
-Wl,option1[,option2,...].....	78
-X.....	78
-Xlinker value.....	79
最適化オプション.....	79
最適化オプションの説明.....	79
-arch keyword (IA-32 システムのみ).....	79
-assume [no]buffered_io.....	80
-auto_ilp32 (Itanium® ベース・システムおよびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ).....	80
-ax{K W N B P} (IA-32 およびインテル EM64T システムのみ).....	81
-complex_limited_range[-].....	82
-f[no-]alias.....	82
-f[no-]fnalias.....	82

-fast.....	82
-fnsplit[-] (Itanium ベース・システムのみ).....	83
-fp (IA-32 およびインテル EM64T システムのみ).....	83
-gp.....	84
-ip.....	84
-ip_no_inlining.....	84
-ip_no_pinning.....	84
-ipo[n].....	84
-ipo_c.....	85
-ipo_obj.....	85
-ipo_S.....	85
-ipo_separate.....	86
-ivdep_parallel (Itanium ベース・システムのみ).....	86
-nolib_inline.....	86
-On.....	86
-opt_report.....	87
-opt_report_file file.....	88
-opt_report_help.....	88
-opt_report_level {min med max}.....	88
-opt_report_phase phase.....	88
-opt_report_routine [routine].....	88
-par_threshold[n].....	89

-parallel	89
-prefetch[-] (IA-32 システムのみ).....	89
-prof_dir dir	90
-prof_file file.....	90
-prof_gen	90
-prof_use	90
-scalar_rep[-] (IA-32 システムのみ).....	91
-tpn.....	91
-unroll[n]	91
-x[K W N B P] (IA-32 およびインテル EM64T システムのみ).....	92
出力ファイル・オプション	93
出力ファイル・オプションの説明	93
-c	93
-fcode-asm.....	93
-fsource-asm.....	93
-f[no]verbose-asm	93
-module path.....	94
-ofilename	94
-Qinstall dir.....	94
-Qlocation,tool,path.....	94
-Qoption,tool,options	94
-S.....	95

-use_asm	95
プリプロセッサ・オプション	95
プリプロセッサ・オプションの説明	95
-assume [no]source_include	95
-Dname[=value]	96
-[no]fpp	96
-Idir	96
-preprocess_only	97
-U name	97
-Wp,option1[,option2,...]	97
ランタイム・オプション	98
ランタイム・オプションの説明	98
-[no]check [all] または -[no]check [none]	98
-check [no]arg_temp_created	98
-check [no]args	98
-check [no]bounds	98
-check [no]format	99
-check [no]output_conversion	99
-check [no]pointer	99
-check [no]shape	99
-[no]traceback	100
idb を使用したデバッグ	100

idb を使用したデバッグの概要	100
デバッグの概要	101
デバッグ・オプション	102
デバッグのためのプログラムの準備	102
デバッグコマンドの使用とブレークポイントの設定	103
その他のデバッグコマンド	104
デバッグコマンドの概要	105
SQUARES サンプル・プログラムのデバッグ	107
デバッグにおける変数の表示	112
モジュール変数	112
共通ブロック変数	112
派生型変数	113
レコード変数	114
ポインタ変数	114
Fortran 95/90 ポインタ	114
整数ポインタ	115
配列変数	116
配列セクション	116
配列への代入	117
複素変数	117
データ型	118
デバッグコマンドにおける表現	119

Fortran 演算子	119
プロシージャ	119
言語が混在したプログラムのデバッグ	119
信号を生成するプログラムのデバッグ	120
アライメントされていないデータの検索	121
データと I/O.....	122
データ表現の概要	122
組込みデータ型	122
整数データの表現.....	125
整数データの表現の概要.....	125
INTEGER(KIND=1) の表現	125
INTEGER(KIND=2) の表現	125
INTEGER(KIND=4) の表現	126
INTEGER(KIND=8) の表現	126
論理データの表現	126
ネイティブ IEEE* 浮動小数点の表現.....	127
ネイティブ IEEE* 浮動小数点の表現の概要	127
REAL(KIND=4) (REAL) の表現.....	129
REAL(KIND=8) (DOUBLE PRECISION) の表現.....	129
REAL(KIND=16) (EXTENDED PRECISION) の表現.....	130
COMPLEX(KIND=4) (COMPLEX) の表現.....	130
COMPLEX(KIND=8) (DOUBLE COMPLEX) の表現.....	131

COMPLEX(KIND=16) の表現	131
fordef.for ファイルとその使用	132
文字表現	134
Hollerith 表現	134
書式なしデータの変換	135
書式なしデータの変換の概要	135
サポートされるネイティブ数値形式と非	136
ネイティブ数値形式	136
数値変換の制限	139
データ書式の指定方法	139
データ書式の指定方法: 概要	139
環境変数 FORT_CONVERTn を使用する方法	141
環境変数 FORT_CONVERT.ext または FORT_CONVERT_ext を使用する方法	142
環境変数 F_UFMTENDIAN を使用する方法	143
リトル・エンディアン – ビッグ・エンディアンの変換環境変数	144
使用例:	145
OPEN 文 CONVERT を使用する方法	147
OPTIONS 文を使用する方法	147
-convert コンパイラ・オプションを使用する方法	148
非ネイティブ・データのポータビリティ	149
Fortran I/O	150
Fortran I/O の概要	150

論理 I/O ユニット	150
I/O 文の種類	151
I/O 文の形式	153
ファイルとファイルの特性	155
ファイルとファイルの特性の概要	155
ファイルの編成	155
シーケンシャル編成	156
相対編成	156
内部ファイルとスクラッチ・ファイル	156
内部ファイル	156
スクラッチ・ファイル	157
レコード型	158
固定長レコード型	158
可変長レコード型	158
セグメント・レコード型	158
ストリーム・レコード型	159
Stream_LF レコード型および Stream_CR レコード型	159
レコード型の選択	159
レコード・オーバーヘッド	160
レコード長	160
ファイルのアクセスと割り当て	161
論理ユニットへのファイルの割り当て	161

デフォルト値の使用	162
OPEN 文でファイル名の指定	162
環境変数の使用	162
暗黙のインテル Fortran 論理ユニット番号	163
デフォルトのパス名とファイル名	163
デフォルトのパス名とファイル名の適用例	164
デフォルトのパス名とファイル名の適用規則	164
事前結合された標準 I/O ファイルの使用	165
ファイルを開く: OPEN 文	166
OPEN 文指定子	166
ファイルとユニット情報の指定子	166
ファイルとレコード特性の指定子	167
特殊なファイルを開くルーチンの指定子	167
ファイルアクセス、処理、および位置の指定子	167
レコード転送特性の指定子	167
エラー処理機能の指定子	168
ファイルを閉じる処理の指定子	168
OPEN 文でファイルの場所をコード化する	168
ファイル情報の取得: INQUIRE 文	169
ユニットによる照会	169
ファイル名による照会	170
出力項目リストによる照会	171

ファイルを閉じる: CLOSE 文	171
レコード操作	172
レコード操作の概要	172
レコード I/O 文指定子	172
レコードアクセス	173
シーケンシャル・アクセス	173
直接アクセス	174
ファイル編成とレコード型によるレコードアクセスの制限	174
ファイル共有	175
開始レコード位置の指定	175
アドバンシング・レコード I/O とノン・アドバンシングレコード I/O	176
レコード転送	177
レコード転送: 入力	178
レコード転送: 出力	178
ユーザが提供する OPEN プロシージャ: USEROPEN 指定子	179
USEROPEN 指定子の構文および動作	179
呼び出された USEROPEN 関数の制限事項	181
USEROPEN プログラムおよび関数の例	181
C およびインテル Fortran プログラムのコンパイルとリンク	182
C 関数およびヘッダファイル用のソースコード	182
インテル Fortran プログラムを呼び出すソースコード	184
レコード型の形式	185

固定長レコード	185
可変長レコード	185
2 GB 未満の可変長レコード	186
2 GB を超える可変長レコード	186
セグメント・レコード	187
ストリーム・ファイル	188
Stream_CR レコードと Stream_LF レコード	188
Microsoft* Fortran PowerStation* 互換ファイル	189
書式付きシーケンシャル・ファイル	189
書式付き直接ファイル	190
書式なしシーケンシャル・ファイル	191
書式なし直接ファイル	193
言語が混在したプログラミング	194
言語が混在したプログラミングの概要	194
メイン・プログラムからのサブプログラムの呼び出し	195
メイン・プログラムからの呼び出し	195
サブプログラムの呼び出し	196
言語が混在したプログラミングのまとめ	196
言語が混在したプログラミングにおける呼び出し規則の調整	198
言語が混在したプログラミングにおける呼び出し規則の調整の概要	198
ATTRIBUTES プロパティと呼び出し規則	199
言語が混在したプログラミングにおける命名規則の調整	202

言語が混在したプログラミングにおける命名規則の調整の概要	202
C/C++ 命名規則	203
Fortran、C、C++ のプロシージャ名	204
名前の大文字・小文字の調整	205
Fortran モジュール名と ATTRIBUTES	205
Fortran でのプロシージャのプロトタイピング	207
言語が混在したプログラミングにおけるデータ交換と参照	208
言語が混在したプログラミングにおけるデータ交換と参照の概要	208
言語が混在したプログラミングにおける引数の渡し方	208
言語が混在したプログラミングにおける共通外部データの使用	211
言語が混在したプログラミングにおけるグローバル変数の使用	211
Fortran の共通ブロックおよび C の構造体の使用	212
共通ブロックおよび C の構造体への直接アクセス	213
共通ブロックのアドレスの受け渡し	214
言語が混在したプログラミングにおけるデータ型の処理	215
言語が混在したプログラミングにおけるデータ型の処理の概要	215
数値、複素数、および論理データ型の処理	217
複素数データ型の戻り	218
配列ポインタと割付け配列の処理	219
整数ポインタの処理	220
整数ポインタの渡し方	220
ポインタの受け取り方	221

配列と Fortran 配列記述子の処理	222
インテル® Fortran の配列記述子の書式	223
文字列の処理	226
文字データ型の戻り	228
ユーザ定義型の処理	229
インテル® Fortran/C が混在したプログラム	230
インテル® Fortran/C が混在したプログラムの概要	230
インテル® Fortran/C プログラムのコンパイルとリンク	230
Fortran と C が混在したプログラミングにおけるモジュールの使用	231
インテル® Fortran プログラムからの C プロシージャの呼び出し	233
命名規則	233
Fortran と C プロシージャ間での引数の受け渡し	234
エラー処理	234
エラー処理の概要	234
ランタイム・ライブラリのデフォルトのエラー処理	234
ランタイム・メッセージの形式	236
プログラム終了時にシェルに返される値	237
重要なエラーのための強制コアダンプ	238
ランタイム・エラーの処理	238
END、EOR、および ERR 分岐指定子の使用	239
IOSTAT 指定子の使用	240
信号処理	241

デフォルトのランタイム・ライブラリ例外ハンドラの上書き	243
TRACEBACKQQ によるトレースバック情報の取得	243
ライブラリの作成と使用	244
ライブラリの使用と作成の概要	244
ライブラリの作成	245
スタティック・ライブラリ	245
共有ライブラリ	245
インテル® Fortran が提供するライブラリ	246
移植ライブラリ	247
移植ライブラリの概要	247
libifport.a 移植ライブラリの使用	248
移植ルーチン	248
情報取得ルーチン	248
プロセス制御ルーチン	249
数値および変換ルーチン	249
入出力ルーチン	250
日付および時刻ルーチン	251
エラー処理ルーチン	251
システム、ドライブ、またはディレクトリ制御および照会ルーチン	251
その他のルーチン	252
数値演算ライブラリ	252
参照情報	253

コンパイル時の環境変数.....	253
ランタイム時の環境変数.....	253
主な IA-32 コンパイラ・ファイルの概要.....	256
主な Itanium® コンパイラ・ファイルの概要.....	257
コンパイラの制限	258
16 進、2 進、8 進、および 10 進の間での変換	259
以前のバージョンのインテル® Fortran との互換性	260
インテル Fortran バージョン 7.1 とバージョン 8.x の相違点	261
マニュアル情報	261
インテル Fortran バージョン 8.x で提供されていないバージョン 7.1 の機能	262
ランタイム時のエラー・メッセージ	262
キーワード.....	280

免責条項

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

本ドキュメントの内容を予告なしに変更することがあります。

インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

本書に含まれている内容は、出荷時の内容を正確に著すよう記述されていますが、製品の不具合の発見とその改良に伴い、製品および本書の内容は予告なく変更される場合があります。現在報告されているソフトウェアの不具合につきましては、お問い合わせください。

いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

著作権法で許可されている場合を除き、文書による事前の許可なく、複製、改変、または翻訳することを禁じます。無断転載を禁じます。

版權制限：米国政府による使用、複製、または開示は、DFARS 252-227-7013 の条項「Rights in Technical Data and Computer Software」の副項 (c)(I)(ii) に規定されている制限を受けます。

非 DOD U.S. Government Departments and Agencies の権利は、FAR 52.227-19(c0(1,2)) に規定されています。

インテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) を利用するには、インテル® EM64T に対応したプロセッサ、チップセット、BIOS、OS、デバイスドライバ、アプリケーションを搭載するコンピュータ・システムが必要です。インテル® EM64T に対応した BIOS がない場合、32 ビットでの動作も含め、プロセッサは動作しません。性能は、ご利用のハードウェアやソフトウェアによって異なります。インテル® EM64T に対応したプロセッサの情報等、詳細については <http://www.intel.co.jp/jp/info/em64t/> を参照、もしくは各システムベンダにお問い合わせください。

Intel、インテル、Celeron、Itanium、MMX、Pentium、i386、Xeon、VTune は、アメリカ合衆国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です。

* その他の名称およびブランド名は、各社の商標および登録商標です。

© 2004, Intel Corporation.

Portions © 2001 Hewlett-Packard Development Company, L.P.

インテル® Fortran コンパイラについて

インテル® Fortran コンパイラは、IA-32 アーキテクチャ・システム向け、およびインテル® Itanium® アーキテクチャ・システム向けにコードをコンパイルします。

インテル Fortran コンパイラ製品には、開発環境用に次のコンポーネントが含まれています。

- 32 ビット・アプリケーション用のインテル Fortran コンパイラ
- Itanium ベース・アプリケーション用のインテル Fortran コンパイラ
- インテル® デバッガ (IDB)

Itanium ベース・アプリケーション用インテル Fortran コンパイラには、インテル Itanium アセンブラとインテル Itanium プロセッサ・リンクが含まれています。

インテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) 対応 IA-32 システムについての詳細は、「Linux* 版インテル® Fortran コンパイラ 8.1 エクステンデッド・メモリ 64 テクノロジ・リリースノート」を参照してください。

「本書の使い方」も参照してください。

本リリースの新機能

バージョン 8.1 の新機能と変更点

本リリースには、新しい定義済みプリプロセッサ・シンボル `_INTEL_COMPILER_BUILD_DATE` や新しいコンパイル・オプションを含む、いくつかの新機能と変更点が含まれています。本リリースの新規オプションをすべて記載したリストは、『インテル® Fortran コンパイラ・オプション・クイック・リファレンス・ガイド』の「新規のコンパイラ・オプション」を参照してください。

本リリースでは、インテル Fortran コンパイラ 8.0 でコンパイルされたオブジェクトおよび `.mod` ファイルを再コンパイルする必要はありません。



注

本リリースに実装された機能の最新情報は、リリースノートを参照してください。

バージョン 8.0 の新機能と変更点

バージョン 8.0 には、以下の新機能と変更点が含まれています。

- Compaq Visual Fortran と互換性のある、より多くのコマンドライン・コンパイラ・オプション
- 派生型拡張子の Fortran 2000 割り付けコンポーネントのサポート
- ランタイム・エラーとソースを関連付けるトレースバック・オプション
- 以前のアプリケーション実行プロファイルを基に、最も効果的なアプリケーション・テストを選択し、その優先度を示すテスト・プライオリタイゼーション・ツール
- 特定のワークロードにおけるアプリケーションのコード適用範囲情報をビジュアルに表示するコード・カバレッジ・ツール
- Itanium® ベース・システム用インテル® デバッガのサポート



注

バージョン 8.0 で実装された機能に関する最新情報は、リリースノートを参照してください。

本書の使い方

『Linux* 版インテル® Fortran コンパイラ・ユーザーズ・ガイド』には、Vol I と Vol II があり、本書は Vol I にあたります。Vol I では、アプリケーションをビルドするための [インテル Fortran コンパイラ](#) の使い方を説明します。Vol II では、アプリケーションの最適化について説明します。

本ユーザーズ・ガイド には、インテル Fortran の使用を開始するにあたっての必要な情報、コンパイルの動作、およびアプリケーションの開発方法が記載されています。

本書は、Fortran 標準プログラミング言語とインテル® プロセッサ・アーキテクチャについてよく理解している読者を対象としています。また、ホスト・コンピュータのオペレーティング・システムにも精通している必要があります。



注

本書では、対象となる各アーキテクチャごとに情報や命令がどのように適用されるかを説明しています。特定のアーキテクチャが指定されていない場合、その説明はすべてのアーキテクチャに適用されます。

その他のマニュアル

本ユーザズ・ガイドの他には、以下のマニュアルがあります。

- *Linux* 版インテル® Fortran コンパイラ・オプション・クイック・リファレンス・ガイド*
- *Intel® Fortran Language Reference (英語)*
- *Intel® Fortran Libraries Reference (英語)*
- *インテル® Fortran リリースノート*

インテル・コンパイラに関するホワイトペーパーを含む、その他のテクニカル製品情報は、次の Web サイトを参照してください。

<http://www.intel.co.jp/jp/developer/software/products/>

表記規則

本書では、次の表記規則を使用しています。

インテル® Fortran	Windows* 版インテル® Visual Fortran コンパイラ製品および Linux* 版インテル Fortran コンパイラ製品でサポートされている共通コンパイラ言語の名前を示します。
インテル® EM64T	インテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) 対応の IA-32 システムを示します。
This type style	構文要素、予約語、キーワード、ファイル名、またはコード例を、モノスペース・フォントで表記します。特に大文字が必要でない限り、小文字で表記されます。
THIS TYPE STYLE	文、キーワード、およびディレクティブを、標準フォントを使用してすべて大文字で表記します。例:「USE 文を追加します」
This type style	メニュー名、メニュー項目、ボタン名、ダイアログ・ウィンドウ名、およびその他のユーザ・インターフェイス項目を、太字の標準テキストで表記します。
[ファイル] > [開く]	(>) 記号で結合したメニュー名およびメニュー項目は、一連の動作を示します。例えば、「[ファイル] > [開く] をクリックします」の場合、[ファイル] メニュー内の [開く] をクリックし、この動作を実行します。
This type style	太字のモノスペース・フォントで表記されているテキ

	ストは、ユーザ入力を示します。例: ユーザが入力するコマンドや値
<i>This type style</i>	斜体のモノスペース・フォントで表記されているテキストは、ユーザの入力個所を示します。また、新しい用語が使われた場合にも、その用語を斜体で表記します。
[options]	大括弧で囲まれている項目は、省略可能です。一部の例では、大括弧は配列を示すためにも使われています。
{value value}	中括弧と縦線は、複数の項目の選択を示します。すべての項目が大括弧で囲まれていない限りは、項目から 1 つを選択しなければなりません。
...	項目の次にくる横方向の反復記号 (3 個のドット) は、反復記号の前の項目が繰り返されていることを示します。コード例では、横方向の反復記号は、一部の文が省略されていることを意味します。
Linux* システム	用語や名前の終わりにあるアスタリスク (*) は、他社の製品登録商標を示します。

コンパイラの起動

コンパイラの起動の概要

次のトピックを参照してください。

[インテル® Fortran コンパイラの使い方](#)

[コンパイル・フェーズ](#)

[前処理フェーズ](#)

[アセンブラとリンカ](#)

[インテル® Fortran コンパイラのデフォルト動作](#)

[入力ファイルとファイル名拡張子](#)

[ファイル指定](#)

[出力ファイル](#)

コンパイラまたはリンカにより作成される一時ファイル

インテル® Fortran コンパイラの使い方

インテル® Fortran コンパイラには、次のような種類があります。

- 32 ビット・アプリケーション用インテル Fortran コンパイラは IA-32 システム用です。IA-32 のコンパイルは、任意の IA-32 インテル® プロセッサ上で実行でき、IA-32 システム上で動作するアプリケーションを生成します。このコンパイラは、インテル® Pentium® M プロセッサ、Pentium 4 プロセッサ、インテル® Xeon™ プロセッサなど、1 種類以上の IA-32 プロセッサ向けに最適です。
- Itanium® アーキテクチャ用インテル Fortran コンパイラ (またはネイティブ・コンパイラ) は、Itanium アーキテクチャ・システム用です。このコンパイラは Itanium ベースのシステムで動作し、Itanium ベース・アプリケーションが生成されます。Itanium アーキテクチャのコンパイラは Itanium ベースのシステムでしか実行できません。

これらのコンパイラを起動させるコマンドは `ifort` です。



注

インテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) 対応 IA-32 システムについての詳細は、「Linux* 版インテル® Fortran コンパイラ 8.1 エクステンデッド・メモリ 64 テクノロジ・リリースノート」を参照してください。

インテル Fortran コンパイラにはさまざまなオプションがあり、それによって、アプリケーションのパフォーマンスを向上させるコンパイラ機能を使用できます。

インテル Fortran コンパイラを使用すると、インテル® アーキテクチャ・ベースのコンピュータ上でソフトウェアの最高のパフォーマンスを引き出せます。このコンパイラにより、さまざまなハイ・パフォーマンスの最適化を行うことができます。以下の表は、それらの特徴と利点の一覧です。

特徴	説明
ストリーミング SIMD 拡張命令 (SSE)、ストリーミング SIMD 拡張命令 2 (SSE2)、およびストリーミング SIMD 拡張命令 3 (SSE3) のサポート	インテル® マイクロアーキテクチャの利点
自動ベクトライザ	コード内の自動並列処理
並列化	ループのマルチスレッド・コードの自動生成。OpenMP*での共用メモリ並列プログ

	ラミング
浮動小数点の最適化	浮動小数点のパフォーマンスが向上
データ・プリフェッチ機能	データ送信の高速化によりパフォーマンスが向上
プロシージャ間の最適化	大きなアプリケーションに対するパフォーマンスが向上
プログラム全体の最適化	大きなアプリケーション内のモジュール間のパフォーマンスが向上
プロファイルに基づく最適化	頻繁に使用されるプロシージャのプロファイリングに基づくパフォーマンスの向上
プロセッサ・ディスパッチ	最新のインテル・アーキテクチャの機能を利用すると同時に、前世代の インテル Pentium プロセッサとのオブジェクト・コードの互換性を確保

コンパイル・フェーズ

コンパイラは、Fortran 言語のソースを処理し、オブジェクト・ファイルを生成します。コンパイラの実行時にオプションを設定することによって、入力と出力を決定します。

コンパイラは、起動時に、ソースファイル名の拡張子、およびコマンドラインに指定されたコンパイル・オプションに基づいて、どのコンパイル・フェーズを実行するかを決定します。

次の表は、コンパイル・フェーズ、および各フェーズを制御するソフトウェアを示しています。

コンパイル・フェーズ	制御するソフトウェア	IA-32 または Itanium® ベース・アプリケーション
前処理 (オプション)	fpp	両方
コンパイル	fortcom	両方
アセンブル (オプション)	as または ias	as は IA-32 アプリケーション用で、ias は Itanium ベース・アプリケーション用です。
リンク	ld (1)	両方

デフォルトでは、コンパイラは、アセンブラを呼び出さずに直接オブジェクト・ファイルを生成します。しかし、特定のアセンブリ入力ファイルを使用して、プロジェクトのその他

の部分とリンクする必要がある場合は、それらのファイルに対してアセンブラを使用できます。

コンパイラは、オブジェクト・ファイルと、認識できないファイル名とをリンクに渡します。次に、リンクは、そのファイルがオブジェクト・ファイル（.o）なのか、ライブラリ（.a）なのか、または 共有ライブラリ（.so）なのかを判断します。コンパイラは、すべてのタイプの入力ファイルを正しく処理するので、どのコンパイル・フェーズの実行にも使用できます。

前処理フェーズ

前処理は、プリプロセッサ・シンボル（マクロ）置換、条件付きコンパイル、ファイルのインクルードといった処理を行います。コンパイルの最初のオプション・フェーズのときに、ファイルが前処理されます。.fpp、.F、.F90、.FOR、.FTN、または.FPP の **ファイル名の拡張子**を持つソースファイルは、コンパイラにより自動的に前処理されます。例えば、次のコマンドは、標準的な Fortran プリプロセッサ・ディレクティブが格納されたソースファイルを前処理した後、このファイルをコンパイラとリンクに渡します。

```
ifort source.fpp
```

その他の拡張子を持つファイルを前処理する場合は、プリプロセッサを明示的に指定する必要があります。

通常は、Fortran ソース・プログラムのための前処理を指定する必要はありません。プログラムが #if、#define などの C 言語形式のプリプロセッシング・コマンドを使用している場合のみ前処理が必要になります。

ソース・プログラムを前処理したい場合は、プリプロセッサ fpp（インテル® Fortran コンパイラとともに提供されているプリプロセッサ）を使用するか、または C コンパイラの前処理機能を使用しなければなりません。fpp を使用することをお勧めします。

別のプリプロセッサを使用する場合は、コンパイラを起動する前にそのプリプロセッサを起動する必要があります。

fpp は、cpp に準拠しており、cpp 方式のディレクティブを受け付けます。cpp（および fpp）では、#if 式で文字列定数値を使用できません。

前処理オプションは、コマンドラインからプリプロセッサの操作を指示するのに使用できます。



注

Fortran をサポートしていないプリプロセッサを使用すると、特に FORMAT 文で、Fortran コードが壊れる可能性があります。例えば、FORMAT (\\I4) は、バックスラッシュ「¥」がレコードの終わりを示すため、プログラムの意味を変更します。

アセンブラとリンカ

次の表は、使用できるアセンブラとリンカをまとめたものです。

ツール	デフォルト	インテル® Fortran コンパイラに 付属
IA-32 アセンブラ	Linux* アセンブラ、as	なし
Itanium® アセンブラ	インテル Itanium アセン ブラ、ias	あり
リンカ	システムリンカ、ld(1)	なし

前処理、コンパイル、アセンブリ、およびリンクについては、[代替ツールの場所およびオプションの指定](#)ができます。

「[インテル® Fortran が提供するライブラリ](#)」も参照してください。

アセンブラ

32 ビット・アプリケーションについては、Linux が独自のアセンブラ as を用意しています。

Itanium ベースのアプリケーションについては、Itanium アセンブラ ias を使用してください。例えば、ある特定の入力ファイルを Fortran プロジェクトのオブジェクト・ファイルにリンクする場合は、次の手順を実行します。

1. -S オプションを指定してコマンドを発行し、アセンブリ・コード・ファイル `file.s: ifort -S -c file.f` を生成します。
2. `file.s` ファイルをアセンブルするには、次のコマンドで Itanium アセンブラを呼び出します: `ias -Nso -p32 -o file.o file.s`

次のアセンブラ・オプションが使用されます。

-Nso は、サインオン・メッセージを抑止します。

-p32 は、32 ビット要素を再配置が可能なデータ要素として定義できるようにします。
(このオプションには、下位互換があります。)

-o *file.o* は、出力オブジェクト・ファイル名を示します。

リンカ

コンパイラは、システムリンカ `ld(1)` を呼び出して、オブジェクト・ファイルから実行ファイルを作成します。

インテル® Fortran コンパイラのデフォルト動作

コンパイラは 1 つまたは複数の入力ファイルに対し、1 つまたは複数の出力ファイルを生成します。デフォルトでは、次の動作を実行します。

- 現在のディレクトリで、ライブラリ・ファイルを含むすべてのファイルを検索します。
- リンク用に指定されたオプションをリンカに渡します。
- ユーザ定義のライブラリをリンカに渡します。
- エラー・メッセージと警告メッセージを表示します。
- デフォルトのオプションが変更されない限り、デフォルトの設定と最適化を実行します。
- IA-32 アプリケーションの場合、[-tpp7 オプション](#)を使用して、コードをインテル® Pentium® 4 プロセッサ用とインテル® Xeon™ プロセッサ用に最適化します。
- Itanium® ベース・アプリケーションの場合、[-tpp 2 オプション](#)を使用して、コードをインテル® Itanium® 2 プロセッサ用に最適化します。



注

Unicode* (マルチバイト) 形式の文字をサポートするオペレーティング・システムでは、コンパイラは Unicode* 文字を含むファイル名を処理します。

入力ファイルとファイル名拡張子

インテル® Fortran コンパイラでは、ファイル名拡張子から各入力ファイルの種類を解釈します。ファイル名拡張子には、`.a`、`.f`、`.for`、`.o` などがあります。

ファイル名	ファイルの種類	処理
<i>filename.a</i>	オブジェクト・ライブラリ	ld に渡されます。
<i>filename.f</i> <i>filename.ftn</i> <i>filename.for</i> <i>filename.i</i>	Fortran の固定形式ソース	インテル Fortran コンパイラによってコンパイルされます。
<i>filename.fpp</i> <i>filename.F</i> <i>filename.FOR</i> <i>filename.FTN</i> <i>filename.FPP</i>	Fortran の固定形式ソース	インテル Fortran プリプロセッサ fpp によって前処理されてから、インテル Fortran コンパイラによってコンパイルされなければなりません。
<i>filename.F90</i>	Fortran の自由形式ソース	インテル Fortran プリプロセッサ fpp によって前処理されてから、インテル Fortran コンパイラによってコンパイルされなければなりません。
<i>filename.f90</i> <i>filename.i90</i>	Fortran の自由形式ソース	インテル Fortran コンパイラによってコンパイルされます。
<i>filename.s</i>	アセンブリ・ファイル	アセンブラ (IA-32 コンパイラ) またはインテル Itanium® アセンブラ (Itanium コンパイラ) に渡されます。
<i>filename.o</i>	コンパイル済みのオブジェクト・ファイル	ld に渡されます。

コンパイラの[設定ファイル](#)を使用して、入カライブラリのデフォルトのディレクトリを指定できます。入力ファイル、一時ファイル、ライブラリ、アセンブラ、およびリンク用の追加ディレクトリを指定するには、[出力ファイル名とディレクトリ名を指定するコンパイラ・オプション](#)を使用します。

ファイル指定

ファイル指定では、任意で、ディレクトリを指定するパス名とそれに続くファイル名が使用されます。パス名は、次の 2 つの形式のいずれかで示します:

- ディレクトリがルート・ディレクトリを基準として決められた絶対パス名。最初の文字はスラッシュ (/) になります。例えば、次に示すディレクトリとファイル名は、
/usr/users/gdata ディレクトリにある testdata ファイルを参照します:
/usr/users/gdata/testdata
- ディレクトリが現在のディレクトリを基準として決められた相対パス名。相対パス名では、スラッシュ (/) を最初の文字として使用しません。次の例では、現

在のディレクトリ `/usr/users` を基準とした相対パス名を使用して、`gdata/` サブディレクトリ内の同一ファイル `testdata` を参照します：
`gdata/testdata`

ディレクトリ名およびファイル名には、オペレーティング・システムのワイルドカード文字（*、?、および [] 構文など）を含めることができません。C シェルで見られるように、パス名の最初の文字にチルド文字（~）を使用することで、最上位ディレクトリを参照できます。

先頭の空白および末尾の空白は、文字式からは削除されますが、Hollerith（数値配列）名からは削除されないため、ファイル指定を行う際は注意してください。

ファイル名では、大文字・小文字が区別されるので、両方を含めることができます。例えば、次の 3 つのファイルはすべて異なるファイルとして扱われます：

```
myfile.for
MYfile.for
MYFILE.for
```

出力ファイル

`ifort` コマンドによって生成される出力は以下のとおりです：

- コマンドラインで `-c` オプションを指定した場合、オブジェクト・ファイル（`test.o` など）が生成されます。オブジェクト・ファイルは、各ソースファイルごとに作成されます。
- `-c` オプションを省略した場合、実行ファイル（`a.out` など）が生成されます。
- ソースファイルに `MODULE` 文が 1 つまたはそれ以上ある場合、1 つまたはそれ以上のモジュール・ファイル（`datadef.mod` など）が生成されます。
- `-shared` オプションを使用した場合、共有ライブラリ（`mylib.so` など）が生成されます。

これらのファイルを生成するかどうかは、コマンドラインで適切なオプションを指定することで制御できます。

`-c` オプションが指定されていない場合、コンパイラによって各ソースファイルごとに一時オブジェクト・ファイルが生成されます。次に、リンカが起動され、オブジェクト・ファイルを 1 つの実行プログラム・ファイルにリンクし、最後に一時オブジェクト・ファイルを削除します。

`-c` オプションを指定した場合は、オブジェクト・ファイルが作成され、作業ディレクトリに保持されます。この場合、後でオブジェクト・ファイルを別の `ifort` コマンドからリ

リンクする必要があります。この方法は、make コマンドで makefile を処理してコンパイルする方法のように、大きなアプリケーションを増分コンパイルするときに役立ちます。

コンパイルの途中で致命的なエラーが発生する場合、または `-c` などの特定のオプションを指定した場合、リンクは行われません。



注

プログラム全体のすべてのオブジェクトをコンパイルするには、`-ipo` オプションを使用します。

実行プログラム・ファイル名を指定するには (`a.out` 以外の名前を指定する場合)、`-o output` オプションを使用します (`output` にはファイル名を指定します)。次のコマンドは、ソースファイル名 `test1.f` に対してファイル名 `prog1.out` を指定します:

```
ifort -o prog1.out test1.f
```

`-o output` オプションとともに `-c` オプションを指定する場合、オブジェクト名を変更できます (実行プログラム名は変更できません)。また、`-c` を指定し、`-o output` オプションを省略する場合、サフィックスが `.o` に変更されたソースファイル名がオブジェクト・ファイル名に使用されます。



注

複数のソースファイルに対しては、`-c` および `-o` を一緒に使用できません。

デフォルトの最適化レベルは `-O2` です (`-g` を指定しない場合のみ)。

コンパイラまたはリンカにより作成される一時ファイル

コンパイラまたはリンカによって作成される一時ファイルは、オペレーティング・システムが一時ファイルを格納するために使用するディレクトリに置かれます。

一時ファイルを格納する際に、ドライバはまず `TMP` 環境変数を確認します。定義されていれば、`TMP` が示すディレクトリが一時ファイルの格納に使用されます。

`TMP` 環境変数が定義されていなければ、ドライバは `TMPDIR` 環境変数を確認します。定義されていれば、`TMPDIR` が示すディレクトリが一時ファイルの格納に使用されます。

TMPDIR 環境変数が定義されていなければ、ドライバは TEMP 環境変数を確認します。定義されていれば、TEMP が示すディレクトリが一時ファイルの格納に使用されます。

TEMP 環境変数が定義されていなければ、/tmp ディレクトリが一時ファイルの格納に使用されます。

アプリケーションのビルド

アプリケーションのビルドの概要

インテル® Fortran については次のトピックを参照してください。

[コンパイル処理の制御](#)

[環境変数の設定と表示](#)

[シェル・スクリプトの実行による環境変数の設定](#)

[インテル® Fortran コンパイラの起動](#)

[ifort コマンドの例](#)

[モジュール \(.mod\) ファイルの使用](#)

[インクルード・ファイルと .mod ファイルの検索](#)

[設定ファイルと応答ファイル](#)

[代替ツールの場所およびオプションの指定](#)

[定義済みプリプロセッサ・シンボル](#)

[コマンドライン出力からファイルへのリダイレクト](#)

[実行プログラムの作成、実行、デバッグ](#)

[共有ライブラリの作成](#)

[共通ブロックの割り当て](#)

コンパイル処理の制御

コンパイル時に使用される環境をカスタマイズするには、以下の変数、オプション、およびファイルを指定します。

- **環境変数**はコンパイラがライブラリや「インクルード」ファイルなどの特殊なファイルを検索するパスを指定します。
- **設定ファイル**は各コンパイルに使用するオプションを指定します。また、**応答ファイル**は個々のプロジェクトに使用するオプションとファイルを指定します。

環境変数の設定と表示

SET コマンドを使用することにより、環境変数を 1 つずつ表示するかまたは設定できます。また、`ifortvars.csh` および `ifortvars.sh` ファイルを使用することにより、一度に複数の環境変数を設定できます。これらのファイルは、`/opt/intel_fc_80/bin` ディレクトリにあります。詳細は、「[シェル・スクリプトの実行による環境変数の設定](#)」を参照してください。

C シェルで環境変数を設定するには、`setenv` コマンドを使用します:

```
setenv FORT8 /usr/users/smith/test.dat
```

C シェルで環境変数を無効にするには、`unsetenv` コマンドを使用します。

```
unsetenv FORT8
```

Bourne シェル (`sh`)、Korn シェル (`ksh`)、および `bash` シェルで環境変数を設定するには、`export` コマンドと割り当てコマンドを使用します:

```
export FORT8
FORT8=/usr/users/smith/test.dat
```

Bourne シェル、Korn シェル、または `bash` シェルで環境変数を無効にするには、`unset` コマンドを使用します:

```
unset FORT8
```

設定ファイル環境変数

デフォルトでは、コンパイラ実行ファイルと同じディレクトリにあるデフォルトの設定ファイル (`ifort.cfg`) が使用されます。ただし、異なるディレクトリにある他の設定ファ

イルを使用する場合は、`IFORTCFG` 環境変数を使って、その設定ファイルのディレクトリとファイル名を指定します。

次のトピックも参照してください。

コンパイル時の環境変数

ランタイム時の環境変数

シェル・スクリプトの実行による環境変数の設定

コンパイラを起動する前に、[環境変数](#)を設定して、各種コンポーネントの場所を設定する必要があります。

インテル® Fortran コンパイラのインストールには、環境変数の設定に使用できるシェル・スクリプトが含まれています。

ソースコマンドを使用して、コマンドラインからシェル・スクリプトを実行します。例えば、`bash` シェルのスクリプト・ファイルを実行するには次のとおりです：

```
source /opt/intel_fc_80/bin/ifortvars.sh
```

C シェルを使用するには、`.csh` バージョンのスクリプト・ファイルを使用します：

```
source /opt/intel_fc_80/bin/ifortvars.csh
```

Linux* の起動時に `ifortvars.sh` を自動的に実行するには、`.bash_profile` ファイルを編集し、ファイルの最後に上記の行を追加します。次に例を示します。

```
# set up environment for Intel compiler
source /opt/intel_fc_80/bin/ifortvars.sh
```

インテル® Fortran コンパイラの起動

次の 2 つの方法のいずれかで、インテル® Fortran コンパイラを起動できます。

- `ifort` コマンドを使用する
- `make` コマンドを使用して `makefile` を指定する

ifort コマンドを使用する

構文は、次のとおりです:

```
ifort [options] input_file(s)
```

option はハイフンが先頭にある 1 文字以上の文字で指定されます。

オプションは、引数をファイル名、文字列、文字、または数字の形をとることができます。特に指定がない限り、オプションとその引数の間にスペースを挿入することができます。「[コンパイラ・オプションの概要](#)」を参照してください。

スペースを区切り文字として使って、複数の *input_file* を指定できます。「[入力ファイルとファイル名拡張子](#)」を参照してください。



注

コマンドラインで指定されたオプションは、すべてのファイルに適用されます。例えば、次の `-c` および `-nowarn` コマンドライン・オプションは `x.f` および `y.f` ファイルに適用されます:

```
ifort -c x.f -nowarn y.f
```

make コマンドを使用する

さまざまなパスを持つ複数のファイルをコンパイルし、この情報を繰返しコンパイルできるようにするには、`makefile` を使用して インテル Fortran コンパイラを起動します。

`makefile` を使用して入力ファイルをコンパイルするには、`/usr/bin` と `/usr/local/bin` がパスになければなりません。

C シェルを使う場合は、`.cshrc` ファイルを編集して、次の行を追加します。

```
setenv PATH /usr/bin:/usr/local/bin:yourpath
```

その後、次のようにコンパイルができます。

```
make -f yourmakefile
```

`-f` は特定の `makefile` を指定する `make` コマンドのオプションです。

ifort コマンドの例

複数のファイルのコンパイルおよびリンク

次の ifort コマンドは、Fortran の自由形式ソースファイル `aaa.f90`、`bbb.f90`、および `ccc.f90` をコンパイルします。このコマンドは、`ld` リンカを起動して、一時オブジェクト・ファイルをリンクに渡し、実行ファイル `a.out` を生成します:

```
ifort aaa.f90 bbb.f90 ccc.f90
```

次の ifort コマンドは、ファイル名が `.f` で終わるすべてのファイルを Fortran 固定形式ソースとしてコンパイルします。`a.out` ファイルがリンクによって生成されます:

```
ifort *.f
```

リンクの抑止

次の ifort コマンドは、MODULE TYPEDEFS_1 を含む自由形式ソースファイル `typedefs_1.f90` をコンパイルしますが、リンクは行いません。このコマンドによって、`typedefs_1.mod` と `typedefs_1.o` が作成されます。オブジェクト・ファイルは自動的に削除されずに残されます。`-c` オプションを指定することでリンクを抑止できます:

```
ifort -c typedefs_1.f90
```

出力ファイル名の変更

次の ifort コマンドは、自由形式の Fortran ソースファイル `circle-calc.f90` および `sub.f90` を続けてコンパイルします:

```
ifort -c circle-calc.f90 sub.f90
```

コンパイル時に、デフォルトの最適化レベルである `-O2` が両方のソースファイルに適用されます。`-c` オプションが指定されているので、オブジェクト・ファイルはリンクに渡されません。この場合、名前付けされた出力ファイルがオブジェクト・ファイルになります。

上記のコマンドと同じように、次の ifort コマンドは複数のソースファイルをコンパイルします:

```
ifort -o circle.out circle-calc.f90 sub.f90
```

この場合、`-c` オプションが指定されていないので、`circle.out` という名前を持つ実行プログラムが生成されます。

リンカ・ライブラリの追加

次の `ifort` コマンドは、デフォルトの最適化設定を使用して、自由形式ソースファイル `myprog.f90` をコンパイルし、検索に追加するライブラリをリンカに渡します：

```
ifort myprog.f90 typedefs_1.o -lmylib
```

ファイルは最適化レベル `-O2` で処理され、オブジェクト・ファイル `typedefs_1.o` とリンクされます。`-lmylib` オプションは、(`ifort` コマンドがリンカに渡す標準ライブラリ・リストの他に) `libmylib` ライブラリで未解決の参照を検索するように指定します。

モジュール (.mod) ファイルの使用

モジュール (.mod ファイル) とは、データ・オブジェクト、パラメータ、構造体、プロシージャ、および演算子のようなエンティティの仕様を持つプログラム・ユニットの種類です。これらのプリコンパイル済みの仕様と定義は、1 つまたはそれ以上のプログラム・ユニットで使用できます。モジュール・エンティティへの部分的あるいは完全なアクセスは、`USE` 文で提供されます。モジュールの一般的な用途は、グローバル・データの仕様、または派生型および関連した演算子の仕様を記述することです。

一部のプログラムでは、複数のディレクトリに格納されたモジュールが必要になります。プログラムのコンパイル時に `-I \textit{dir}` オプションを使用することにより、プログラムに含める必要がある .mod ファイルの位置を指定できます。

`-module \textit{path}` オプションを使用することにより、モジュール・ファイルを作成するディレクトリを指定できます。また、この指定したパスは、モジュール・ファイルを検索するパスとしても使用されます。このオプションを使用しない場合は、モジュール・ファイルが現在のディレクトリに作成されます。

他のプログラムやサブプログラムから参照される前に、モジュール・ファイルが作成されていることを確認する必要があります。

モジュールを持つプログラムのコンパイル

コンパイルするファイルに 1 つ以上のモジュールが定義されている場合、コンパイラは 1 つ以上の .mod ファイルを生成します。例えば、`a.f90` ファイルには、以下のよう

```

module test
integer:: a
contains
  subroutine f()
  end subroutine
end module test

module payroll
.
.
.
end module payroll

```

コンパイラ・コマンド:

```
ifort -c a.f90
```

このコマンドにより、次のファイルが生成されます:

- test.mod
- test.o
- payroll.mod
- payroll.o

.mod ファイルには、プログラム a.f90 で定義されたモジュールに関する必要な情報が含まれています。

プログラムにモジュールが含まれていない場合、.mod ファイルは生成されません。モジュールが含まれていないサンプル・プログラム test2.f90 の例を以下に示します。コンパイラ・コマンド:

```
ifort -c test2.f90
```

上記のコマンドにより、オブジェクト・ファイル test2.o のみが生成されます。

他の例として、file1.f90 には 1 つ以上のモジュールが含まれており、file2.f90 には、USE 文を使ってそのモジュールを参照するプログラム・ユニットが 1 つ以上含まれていると仮定します。この場合、ソースを次のコマンドでコンパイルし、リンクを行います:

```
ifort file1.f90 file2.f90
```

マルチディレクトリ・モジュール・ファイルの処理

.mod ファイルを複数の異なるディレクトリに生成するときのモジュール管理の例を以下に示します。プログラム mod_def.f90 がディレクトリ /usr/yourdir/test/t

にあり、このプログラムには、以下のように定義されたモジュールが含まれているとします:

```
file: mod_def.f90
module definedmod
.
.
.
end module
```

コンパイラ・コマンド:

```
ifort -c mod_def.f90
```

上記のコマンドによって、ディレクトリ /usr/yourdir/test/t 内に mod_def.o と definedmod.mod の 2 つのファイルが生成されます。

上記の .mod ファイルを別のディレクトリで使用する必要がある場合、例えば、プログラム usemod が /usr/yourdir/test/t2 ディレクトリにある definemod.mod を使用する場合、以下を行います:

```
file: use_mod_def.f90
program usemod
use definedmod
.
.
.
end program
```

上記のサンプル・プログラムをコンパイルするには、以下のコマンドを使用します:

```
ifort -c use_mod_def.f90 -I/usr/yourdir/test/t
```

-Idir オプションには、definedmod.mod ファイルが格納されているパスを指定します。

makefile を使用した並列呼び出し

モジュールが定義されているプログラムでは、makefile を使用した並列呼び出しがサポートされています。この makefile は、プロシージャ間の最適化およびプログラム全体の最適化に対応します。以下にサンプル・コードを示します:

```
test1.f90
module m1
.
.
.
```



```

end module
test2.f90
subroutine s2()
use m1
.
.
.
end subroutine
test3.f90
subroutine s3()
use m1
.
.
.
end subroutine

```

上記のコードをコンパイルするための makefile は、以下のようになります:

```

m1.mod: test1.o
test1.o: test1.f90
ifort -c test1.f90
test2.o: m1.mod test2.f90
ifort -c test2.f90
test3.o: m1.mod test3.f90
ifort -c test3.f90

```

インクルード・ファイルと .mod ファイルの検索

インクルード・ファイルは、`#include` プリプロセッサ・ディレクティブまたは Fortran の `INCLUDE` 文によってプログラムに取り込まれます。

ディレクトリからインクルード・ファイルが検索される順序は、次のとおりです。

1. `include` が含まれるソースファイルのディレクトリ
2. `-I` オプションで指定されたディレクトリ
3. 現在の作業ディレクトリ
4. `FPATH` 環境変数で指定されたディレクトリ

検索されるディレクトリの場所は、インクルード・ファイル・パスと呼ばれます。複数のディレクトリをインクルード・ファイル・パスに指定できます。

モジュール (.mod) ファイルは、`USE` 文を使用してプログラムで指定されます。モジュールファイルは、複数のディレクトリにわたって配置できます。

ディレクトリから .mod ファイルが検索される順序は、次のとおりです。

1. USE 文が含まれるソースファイルのディレクトリ
2. `-module path` オプションで指定されたディレクトリ
3. `-I dir` オプションで指定されたディレクトリ
4. 現在の作業ディレクトリ
5. `FPATH` 環境変数で指定されたディレクトリ

インクルード・ファイル・パスの指定と除外

インクルード・ファイルとモジュールファイルの場所を示すには、`-I dir` オプションを使用できます。

`FPATH` 環境変数で指定されたデフォルトのパスをコンパイラが検索しないようにするには、`-x` オプションを使用します。

これらのオプションは、`ifort.cfg` の設定ファイルまたはコマンドラインで指定できます。

例えば、デフォルトのパスの代わりに `/alt/include` のパスを検索するようにコンパイラに命令するときは次のコマンドラインを使用します。

```
ifort -X -I /alt/include newmain.f
```

設定ファイルと応答ファイル

設定ファイルと応答ファイルは、同じコマンドを何度も入力する手間を省く点では同じです。(応答ファイルは、間接コマンドファイルとしても知られています。) 次に、各ファイルについて説明します。

設定ファイル

設定ファイル (`.cfg`) を使用することにより、次のことが可能になります:

- コマンドライン・オプションを入力する時間を短縮できます。
- よく使用されるコマンドの整合性を保証します。

設定ファイルには、あらゆるコマンドライン・オプションを挿入できます。コンパイラは、設定ファイルに記述されたオプションを上から順番に処理し、次にコンパイラの起動時に指定されたコマンドライン・オプションを処理します。

**注**

設定ファイルに記述されたオプションは、コンパイラを実行するたびに使用されます。他のプロジェクトで、異なるオプションを使用する必要がある場合は、応答ファイルを使用してください。

デフォルトでは、`ifort.cfg` という名前の設定ファイルが使用されます。

このファイルは、コンパイラの実行ファイルが格納されているディレクトリにあります。

別の場所にある他の設定ファイルを使用する場合は、`IFORTCFG` 環境変数を使用し、その設定ファイルのディレクトリとファイル名を指定します。

設定ファイルのサンプル

以下に設定ファイルのサンプルを示します。ポンド記号 (#) は、その行がコメント行であることを示します。

```
## Example ifort.cfg file
##
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
##
## Set extended-length source lines.
-extend_source
##
## Set maximum floating-point significand precision.
-pc80
##
```

応答ファイル

応答ファイル（間接コマンドファイルとしても知られています）を使用することにより、次のことが可能です：

- プロジェクトごとに、特定のコンパイルで使用するオプションを指定できます。
- この情報を個々のファイルに保存できます。

応答ファイルは、コマンドラインでオプションとして呼び出します。応答ファイルで指定したオプションは、コマンドライン上の応答ファイルが呼び出された場所に挿入されます。

応答ファイルは、設定ファイルと同じように、次のことが可能です：

- コマンドライン・オプションを入力する時間を短縮できます。

- よく使用されるコマンドの整合性を保証します。

設定ファイルに記述されたオプションは、コンパイラを実行するたびに使用されます。これに対して、応答ファイルは、個々のプロジェクトごとにオプションを維持できます。

応答ファイル（または間接コマンドファイル）では、1 行にオプションまたはファイル名をいくつも記述できます。同じコマンドラインで、複数のファイルを参照することもできます。

応答ファイルを使用するには、次の構文を使用します:

```
ifort @responsefile [@responsefile2...]
```



コマンドラインでは、応答ファイル名の前にアットマーク (@) を入力する必要があります。

代替ツールの場所およびオプションの指定

インテル® Fortran コンパイラでは、前処理、コンパイル、アセンブリ、およびリンクのデフォルト・ツールに代わる代替ツールの場所およびオプションを指定できます。これを行うには、コマンドライン・オプションを使用します。

-Qlocation を使ってツールの代替場所を指定する

-Qlocation を使用すると、ツールのパス名を指定できます。このオプションの構文は次のとおりです:

```
-Qlocation, tool, path
```

tool には次のいずれかを入力します:

- fpp インテル Fortran プリプロセッサ (fpp)
- f インテル Fortran コンパイラ (fortcom)
- as アセンブラ
- link または ld リンカ
- crt スタートアップ・ルーチン (crt%.o)。実行ファイルにリンクされる。

path はツールの場所です。

例:

```
ifort -Qlocation,fpp,/usr/preproc myprog.f
```

-Qoption を使用してオプションをツールに渡す

-Qoption を使用して、オプションをプリプロセッサ、コンパイラ、アセンブラ、またはリンカに渡します。このオプションの構文は次のとおりです:

```
-Qoption, tool, options
```

tool には次のいずれかを入力します:

- fpp インテル Fortran プリプロセッサ (fpp)
- f インテル Fortran コンパイラ (fortcom)
- as アセンブラ
- link リンカ

options は、指定したツールに有効な引数文字列です。1 つでも複数個でも指定できます。

スペースかタブ文字を含む引数を指定するときは、その引数全体を二重引用符 (" ") で囲ってください。引数を複数指定するときは、それぞれをカンマで区切ってください。

次の例では、代替ライブラリとリンクします:

```
ifort -Qoption,link,-lmylib prog1.f
```

定義済みプリプロセッサ・シンボル

プリプロセッサ・シンボル (マクロ) を使用すると、コンパイルする前にプログラムの値を置換することができます。この処理は、[前処理フェーズ](#)で行います。

一部のプリプロセッサ・シンボルはコンパイラ・システムによって事前定義されており、コンパイラ・ディレクティブおよび fpp で使用できます。他のシンボルを使用する場合は、コマンド行で指定します。

「[プリプロセッサ・オプション](#)」も参照してください。

次の表は、インテル® Fortran コンパイラで利用できる定義済みプリプロセッサ・シンボルです。「デフォルト」欄では、デフォルトでそのプリプロセッサ・シンボルが有効 (オン) になるのか、あるいは無効 (オフ) になるのかを示します。

シンボル名	デフォルト	アーキテクチャ (IA-32 または Itanium® ベース)	説明
<code>_INTEL_COMPILER=n</code>	On, <i>n</i> =810	両方	インテル Fortran コンパイラを識別します。
<code>_INTEL_COMPILER_BUILD_DATE=YYYYMMDD</code>		両方	インテル Fortran コンパイラのビルドの日付を示します。
<code>__linux__</code> <code>__linux</code> <code>__gnu_linux__</code> <code>linux</code> <code>__unix__</code> <code>__unix</code> <code>unix</code> <code>__ELF__</code>		両方	コンパイル開始時に定義されます。
<code>__i386__</code> <code>__i386</code> <code>i386</code>		IA-32 アーキテクチャ	
<code>__ia64__</code> <code>__ia64</code> <code>ia64</code>		Itanium アーキテクチャ	
<code>_OPENMP=n</code>	<i>n</i> =200011	両方	このプリプロセッサ・シンボルは、YYYYMM 形式で、YYYY と MM はそれぞれ OpenMP Fortran 使用がサポートされた年と月を表します。fpp と Fortran コンパイラの条件付きコンパイルの両方で、このプリプロセッサ・シンボルを使用できます。また、 <code>-openmp</code> が指定された場合のみ使用できます。
<code>_PRO_INSTRUMENT</code>	オフ	両方	<code>-prof_gen</code> が指定されたときに、定義されます。

プリプロセッサ・シンボルの定義

前処理中に使用されるシンボル名を定義するには、`-D` オプションを使用します。このオプションの機能は、`#define` というプリプロセッサ・ディレクティブと同じです。次の形式でこのオプションを指定します。

```
-Dname [=value]
```

各アイテムの意味は次のとおりです。

- *name* は定義するシンボルの名前です。
- *value* は *name* を置き換える任意の *value* です。

value を入力しなかった場合、*name* は 1 に設定されます。スペースまたは特殊文字がある場合は、*value* が引用符で囲まれていなければなりません。

前処理を行うと、*name* が検出されるたびに、指定されている *value* に置換されます。例えば、*SIZE* という名前のシンボルを定義し、値を 100 にするには、次のコマンドを実行します。

```
ifort -fpp -DSIZE =100 prog1.f
```

前処理を行うと、前処理されたソースコードをコンパイラに渡す前に、*SIZE* が指定されている値の 100 に置換されます。プログラムに次の宣言が含まれていることを仮定します。

```
REAL VECTOR (SIZE)
```

コンパイラに送信されるコードでは、この宣言およびその他 *SIZE* という名前が検出されるすべての箇所において、値 100 が *SIZE* に取って代わります。

プリプロセッサ・シンボルの抑止

プリプロセッサ・シンボルの自動定義を抑止するには、`-U` オプションを使用します。このオプションは、指定された名前について現在有効になっているシンボル定義を抑止するために使用します。`-U` オプションは、`#undef` プリプロセッサ・ディレクティブと同じ機能を果たします。

コマンドライン出力からファイルへのリダイレクト

多くのテキストを表示するプログラムの場合、`stdout` に表示されているテキストをファイルにリダイレクトすることを考慮してください。テキストを多く表示すると、プログラムの実行が遅くなります。ワークステーションのウィンドウでテキストをスクロールすることで、I/O でボトルネック（経過時間の増加）が生じ、CPU 時間がより長くなる可能性があります。

次のコマンドでは、出力をリダイレクトしてから表示することによって、プログラムをより効率的に動作する方法を示します。

```
myprog > results.lis
more results.lis
```

プログラムの出力をリダイレクトすると、画面 I/O が減少するため、レポートされる時間が変わります。

実行プログラムの作成、実行、デバッグ

自由形式を使用し、モジュールおよび外部サブプログラムを使用するメイン・プログラムの例を以下に示します。

`CALC_AVERAGE` 関数は、別個に作成されたファイルに収められており、インターフェイス・ブロックの `ARRAY_CALCULATOR` モジュールに依存します。

`USE` 文が `ARRAY_CALCULATOR` モジュールにアクセスします。このモジュールには、`CALC_AVERAGE` の関数定義が含まれています。

5 要素の配列が `CALC_AVERAGE` 関数に渡され、`PRINT` される値が関数から `AVERAGE` 変数へ返されます。

サンプル・プログラム:

```
!File: main.f90
!This program calculates the average of five numbers
PROGRAM MAIN
  USE ARRAY_CALCULATOR
  REAL, DIMENSION(5) :: A = 0
  REAL :: AVERAGE
  PRINT *, 'Type five numbers: '
  READ  (*, '(F10.3)') A
```



```

    AVERAGE = CALC_AVERAGE(A)
    PRINT *, 'Average of the five numbers is: ', AVERAGE
END PROGRAM MAIN

```

メイン・プログラムが参照するモジュールを以下に示します。この例では、インターフェイス・ブロックや形状引継ぎ配列など、さらに多くの Fortran 95/90 機能が示されています:

```

!File: array_calc.f90.
!Module containing various calculations on arrays.
MODULE ARRAY_CALCULATOR
  INTERFACE
    FUNCTION CALC_AVERAGE(D)
      REAL :: CALC_AVERAGE
      REAL, INTENT(IN) :: D(:)
    END FUNCTION CALC_AVERAGE
  END INTERFACE
  !Other subprogram interfaces...
END MODULE ARRAY_CALCULATOR

```

メイン・プログラムが参照する CALC_AVERAGE の関数宣言を以下に示します:

```

!File: calc_aver.f90.
!External function returning average of array.
FUNCTION CALC_AVERAGE(D)
  REAL :: CALC_AVERAGE
  REAL, INTENT(IN) :: D(:)
  CALC_AVERAGE = SUM(D) / UBOUND(D, DIM = 1)
END FUNCTION CALC_AVERAGE

```

サンプル・プログラムを作成するためのコマンド

プログラム開発の初期段階では、上記の 3 つのサンプル・プログラムのようなファイルは、次のコマンドを使用して個別にコンパイルしリンクできます:

```

ifort -c array_calc.f90
ifort -c calc_aver.f90
ifort -c main.f90
ifort -o calc main.o array_calc.o calc_aver.o

```

コマンドの詳細:

- -c オプションはリンクを抑制し、.o ファイルを保持します。
- 最初のコマンドでは、ファイル array_calculator.mod および array_calc.o (MODULE 文にある名前によって、array_calculator.mod モジュール・ファイルの名前が決定します) が作成されます。モジュール・ファイルは作業ディレクトリに保存されます。

- 2 番目のコマンドでは、ファイル `calc_aver.o` が作成されます。
- 3 番目のコマンドでは、ファイル `main.o` が作成され、モジュール・ファイル `array_calculator.mod` が使用されます。
- 最後のコマンドでは、すべてのオブジェクト・ファイルが `calc` という名前の実行プログラムにリンクされます。ファイルをリンクするには、`ld` コマンドの代わりに `ifort` コマンドを使用します。

ファイル名を指定する順番は重要です。この `ifort` コマンドでは次の順でファイルを指定し、コンパイル、リンクが行われます:

- モジュールを定義するファイル `array_calc.f90` がコンパイルされ、オブジェクト・ファイルおよび `array_calculator.mod` ファイルが作成されます。
- 外部関数 `CALC_AVERAGE` を含むファイル `calc_aver.f90` をコンパイルします。
- ファイル `main.f90` (メイン・プログラム) をコンパイルします。USE 文は、モジュール・ファイルの `array_calculator.mod` を参照します。
- `ld` を使用して、メイン・プログラムとすべてのオブジェクト・ファイルを `calc` という名前の実行プログラム・ファイルにリンクします。

サンプル・プログラムの実行

パス定義に `calc` が格納されたディレクトリが含まれている場合、プログラムの名前を入力するだけで実行できます。

`calc`

サンプル・プログラムを実行すると、メイン・プログラムの `PRINT` 文と `READ` 文によって、ユーザとプログラムの間で次のようなやり取りが行われます:

```
Type five numbers:
55.5
4.5
3.9
9.0
5.6
Average of the five numbers is:    15.70000
```

サンプル・プログラムのデバッグ

デバッガを使用してプログラムをデバッグするには、`-g` オプションを付けてソースファイルをコンパイルし、オブジェクト・ファイルおよび実行プログラム・ファイルにソース行単位のデバッグを行うためのシンボルテーブル情報を追加します。また、`ifort` コマ

ンドで `-o` オプションを使用することにより、実行プログラム・ファイルに `calc_debug` という名前を付けられます:

```
ifort -g -o calc_debug array calc.f90 calc aver.f90
main.f90
```

「[ld を使用したデバッグの概要](#)」および関連するセクションを参照してください。

共有ライブラリの作成

Fortran のソースファイルから共有ライブラリを作成するには、`ifort` コマンドを使用してファイルを処理します。

- `.so` ファイルを作成するには、`-shared` オプションを指定しなければなりません。
- `-o output` オプションを指定して、出力ファイルに名前を付けることができます。
- `-c` オプションを省略すると、コマンド行から直接、共有ライブラリ (`.so` ファイル) を単一ステップで作成できます。
`-o output` オプションを省略すると、コマンド行の最初の Fortran ファイルの名前が、`.so` ファイルのファイル名を作成するのに使用されます。また、共有ライブラリ作成に関連付けられている追加オプションを指定できます。
- `-c` オプションを指定すると、オブジェクト・ファイル (`.o` ファイル) を作成できます。これは、`-o` オプションで名前を付けることができます。共有ライブラリを作成するには、`.o` ファイルを `ld` と一緒に処理します。この際、共有ライブラリ作成に関連付けられているオプションを指定します。
- Itanium® ベース・システムで共有ライブラリをビルドする場合、共有ライブラリに含まれる各オブジェクト・ファイルのコンパイルを行う `-fpic` オプションを指定します。このオプションが使用されていない場合、リンカにより、「@gprel relocation against dynamic symbol.」というようなエラー・メッセージが発行されます。

ifort コマンドのみで共有ライブラリを作成する

`ifort` コマンドのみで共有ライブラリ (`.so`) ファイルを作成できます。

```
ifort -shared octagon.f90
```

`-shared` オプションは共有ライブラリを作成するのに必要です。`octagon.f90` はソースファイルの名前です。複数のソースファイルとオブジェクト・ファイルを指定できます。

-o オプションが省略されているため、共有ライブラリ・ファイルの名前は `octagon.so` になります。

-c オプションが省略されているため、Fortran ライブラリの標準リストを指定する必要はありません。

ifort コマンドおよび ld コマンドで共有ライブラリを作成する

最初に `.o` ファイルを作成する必要があります。次の例では、`octagon.o` ファイルを作成します。

```
ifort -c octagon.f90
```

`octagon.o` ファイルを `ld` コマンドの入力として使用して、`octagon.so` という共有ライブラリを作成します。

```
ld -shared -no archive octagon.o \  
    -lifport -lifcoremt -limf -lm -lirc -lcxa \  
    -lpthread -lirc -lunwind -lc -lirc_s
```

次の点に注意してください。

- `-shared` オプションは共有ライブラリを作成するのに必要です。
- オブジェクト・ファイルの名前は `octagon.o` です。複数のオブジェクト (`.o`) ファイルを指定できます。
- `-lifport` とそれに続くオプションは、ライブラリの標準リストです。これは、`ifort` コマンドで、`ld` に渡すことになるリストです。共有ライブラリを作成する場合、すべてのシンボルが解決される必要があります。

正しくライブラリを指定するために、`-dryrun` コマンドの出力結果から、使用されるすべてのライブラリを確認するとよいでしょう。

`-Qoption` コマンドを使用してオプションを `ld` に渡します。

共有ライブラリの詳細は、「[ライブラリの作成](#)」を参照してください。

「`ld(1)`」リファレンス・ページも参照してください。

共有ライブラリの制約

`ld` で共有ライブラリを作成する場合、次の制約に注意してください。

- 共有ライブラリをアーカイブ・ライブラリにリンクすることはできません。
共有ライブラリを作成する場合、外部参照を解決するために依存できるのは、他の共有ライブラリだけです。アーカイブ・ライブラリに存在するルーチンを参照する必要がある場合は、そのルーチンを個別の共有ライブラリに入れるか、または作成する共有ライブラリに含めます。共有ライブラリの作成時には、複数のオブジェクト(.o) ファイルを指定できます。
ルーチンを個別の共有ライブラリに入れるには、そのルーチンのソースファイルかオブジェクト・ファイル入手して、必要に応じて再コンパイルし、個別の共有ライブラリを作成します。ifort コマンドで再コンパイルする際、または ld コマンドで共有ライブラリを作成する際に、オブジェクト・ファイルを指定することができます。
- 作成する共有ライブラリにルーチンを含めるには、そのルーチン（ソースファイルまたはオブジェクト・ファイル）を共有ライブラリを構成する他のソースファイルと一緒に配置し、必要に応じて再コンパイルします。
次に、共有ライブラリを作成しますが、再コンパイル時または共有ライブラリ作成時に、そのルーチンが含まれるファイルを指定します。ifort コマンドで再コンパイルする際、または ld コマンドで共有ライブラリを作成する際に、オブジェクト・ファイルを指定することができます。
- 共有ライブラリを作成する場合、すべてのシンボルが定義（解決）されている必要があります。
共有ライブラリの作成時には、すべてのシンボルが ld に対して定義されていなければならないため（-Qoption コマンドを使用しない限り）、ld コマンド行にすべてのインテル® Fortran 標準ライブラリを含む共有ライブラリを指定する必要があります。インテル Fortran の標準ライブラリのリストは、-lstring オプションを使って指定することができます。

共有ライブラリのインストール

共有ライブラリを作成したら、これを参照するプログラムを実行する前に、プライベートまたはシステム全体で使用するためにインストールする必要があります。

- プライベート用の共有ライブラリをインストールするには（テストを行う場合など）、「ld(1)」で説明されているように環境変数を LD_LIBRARY_PATH に設定します。
- システム共通の共有ライブラリをインストールするには、ld が使用する標準ディレクトリ・パスの 1 つに共有ライブラリ・ファイルを配置します。「ld(1)」を参照してください。

共通ブロックの割り当て

`-dyncom` (ダイナミック共用) オプションは、実行時に共通ブロックの割り当てを制御するために使用します。

このオプションは、共通ブロックを動的にするために指定します。そのデータの領域がコンパイル時ではなく実行時に割り当てられます。ダイナミック共通ブロックの宣言を含む各ルーチンへの入口で、共通ブロックの領域が割り当てられているかどうかをチェックされます。ダイナミック共通ブロックがまだ割り当てられていない場合は、そのチェック時に領域が割り当てられます。

次の例のコマンドラインでは、実行時に動的に割り当てられる共通ブロックの名前とともに、ダイナミック共用オプションを指定しています。

```
ifort -dyncom "blk1,blk2,blk3" test.f
```

`blk1`、`blk2`、および `blk3` は、動的にされる共通ブロックの名前です。

`-dyncom` オプションの使用ガイドライン

次に、`-dyncom` (ダイナミック共用) オプションを使用する場合に注意する必要があるいくつかの制限を示します。

- ダイナミック COMMON 内のエンティティを、DATA 文で初期化してはいけません。
- 指定された COMMON ブロックだけがダイナミック COMMON として指定されます。
- ダイナミック COMMON 内のエンティティを、スタティック COMMON 内のエンティティまたは DATA で初期化された変数とともに EQUIVALENCE 式で使用してはいけません。

ダイナミック共通ブロックを使用する理由

ダイナミック共通ブロックを使用する主な理由は、独自の割り当てルーチンの提供によって共通ブロックの割り当てが制御できるからです。独自の割り当てルーチンを使用するには、そのルーチンを Fortran ランタイム・ライブラリの前にリンクする必要があります。このルーチンは、正しいルーチン名を生成するために、C 言語で記述されなければなりません。

このルーチンのプロトタイプは、次のとおりです。

```
void _FTN_ALLOC(void **mem, int *size, char *name);
```

ここで

- *mem* は、共通ブロックの基底ポインタの場所です。このポインタは、このルーチンによって、割り当てられるブロックメモリを指すように設定しなければなりません。
- *size* は、プログラム内で宣言され、コンパイラが共通ブロックに割り当てる必要があると判断したメモリのバイト数（整数）。この値を無視して、目的に合った任意の値を使用できます。



注

割り当てる領域のサイズ（バイト数）を返す必要があります。`_FTN_ALLOC()` を呼び出すライブラリ・ルーチンにより、この共通ブロックの他のすべてのオカレンスが、割り当てた領域に適合することが保証されます。`size` パラメータの変更によって、割り当てる領域のサイズ（バイト数）を返します。

- *name* は、動的に割り当てられる共通ブロックの名前です。

ダイナミック共通ブロックへのメモリの割り当て

ランタイム・ライブラリ・ルーチン `f90_dyncom` は、メモリ割り当てを実行します。コンパイラは、ダイナミック共通ブロックを含むプログラム内の各ルーチンの開始時に、このルーチンを呼び出します。次に、このライブラリ・ルーチンが `_FTN_ALLOC()` を呼び出して、メモリを割り当てます。デフォルトでは、コンパイラが、各ルーチンで宣言されたとおりの共通ブロックのサイズ（バイト数）を `f90_dyncom` に渡し、次にそのサイズが `_FTN_ALLOC()` に渡ります。別のルーチンにおいて異なるサイズで宣言された同じ名前の共通ブロックを持つ非標準の拡張を使用する場合、その共通ブロックの宣言を含むルーチンが呼び出される順序によっては、ランタイム・エラーが発生する可能性があります。

Fortran ランタイム・ライブラリには、単に要求されたバイト数を割り当てて返す `_FTN_ALLOC()` のデフォルトのバージョンが含まれます。

コンパイラ・オプション

コンパイラ・オプションの概要

次のトピックを参照してください。

コンパイラ・オプションの詳細

オプションに関連したコンパイラ・ディレクティブ

コードの生成オプション

互換性オプション

コンパイル診断オプション

データ・オプション

外部プロシージャ・オプション

浮動小数点オプション

言語オプション

ライブラリ・オプション

その他のオプション

最適化オプション

出力ファイル・オプション

プリプロセッサ・オプション

ランタイム・オプション

コンパイラ・オプションの詳細

`ifort` コマンドに渡すオプションは、コンパイラがファイル名のサフィックスに応じて、どのようにファイルを処理するかに影響を与えます。多くの場合、`ifort` コマンドは最も単純な形式で使用できます。

コンパイラ・オプションの形式

オプションには、スペースによって分かれた 2 つの単語から構成されているものや、下線 (`_`) によってつながった複数の単語から構成されているものがあります。多くのオプションは短縮可能で、たいていの場合、4 文字 (またはそれ以上の文字数) まで

短縮できます。例えば、オプション `-check output_conversion` を `-check out` に短縮できます。

インテル® Fortran のコンパイラ・オプションには、次の 4 つの形式があります:

- `no` が先頭にあるオプションは、そのオプションを無効にします。この形式は、Compaq* Fortran で使用されていました。例: `logo` および `nologo`
- 最後にハイフンがあるオプションは、そのオプションを無効にします。この形式は、インテル Fortran コンパイラの旧バージョンで使用されていました。例: `-prefetch` および `-prefetch-`
- `no` または `no-` がオプションの途中に挿入されている場合、そのオプションを無効にします。例: `-falias` および `-fno-alias`
- `n` パラメータ (番号を指定します) を持つオプション。`n` をゼロに設定することでオプションを無効にします。



注

コマンドライン上に、オプションを有効にするバージョンと無効にするバージョンの両方 (または同じオプションの 2 つの異なるバージョン) がある場合、最後に指定したバージョンが優先されます。

複数の ifort コマンドを使用する

複数の `ifort` コマンドを使用してプログラムを部分的にコンパイルする場合、プログラムの実行に影響を与えるオプションは、すべてのコンパイルで一貫して使用されている必要があります。これは特に、プロシージャ間でデータ共有またはデータ渡しが行われている場合に必須です。次に例を示します。

- モジュール定義 (ユーザ定義の構造体) または共通ブロック間で受け渡しされるデータあるいは共有されるデータに対しては、同一のデータ・アライメントを使用する必要があります。これには、すべてのコンパイルで `-align` オプションの同一バージョンを使用します。
- モジュール定義または共通ブロック間で受け渡しや共有が行われる種別パラメータあるいはサイズ指定子を持たない、`INTEGER`、`LOGICAL`、`REAL`、`COMPLEX`、または `DOUBLE PRECISION` の宣言が、プログラムに含まれている場合があります。そのような数値データ宣言を制御するオプションは、一貫して使用しなければなりません。

OPTIONS 文を使用してオプションを上書きする

Fortran ソースコードで OPTIONS 文を使用することにより、コマンドラインで指定したいくつかのオプションを上書きすることができます。OPTIONS 文で指定したオプションは、その文を含むプログラム・ユニットにのみ影響を与えます。

オプションに関するヘルプを表示する

すべてのコマンドライン・オプションに関する概要情報を表示するには、コマンドラインで `-help` を入力します。

オプションに関連するコンパイラ・ディレクティブ

いくつかのコンパイラ・ディレクティブはコンパイラ・オプションと同じ効果を持ちます（次表を参照）。しかしながら、コンパイラ・ディレクティブはプログラム中でオン、オフすることができますが、コンパイラ・オプションはコンパイラ・ディレクティブで無効にしない限り、コンパイル全体に対して効果が残ります。

コンパイラ・ディレクティブと等価なコマンドライン・コンパイラ・オプション:

コンパイラ・ディレクティブ	等価なコマンドライン・コンパイラ・オプション
DECLARE	<code>-warn declarations</code>
NODECLARE	<code>-warn nodeclarations</code>
DEFINE <i>symbol</i>	<code>-Dname</code>
FIXEDFORMLINESIZE: <i>option</i>	<code>-extend_source [option]</code>
FREEFORM	<code>-free or -nofixed</code>
NOFREEFORM	<code>-nofree or -fixed</code>
INTEGER: <i>option</i>	<code>-integer_size option</code>
PACK: <i>option</i>	<code>-align [option]</code>
REAL: <i>option</i>	<code>-real_size option</code>
STRICT	<code>-warn stderrs with -stand</code>
NOSTRICT	<code>-warn nostderrors</code>

上述のコンパイラ・ディレクティブはプリフィックス `!DEC$` の後にスペースを付けて指定されます。

例: `!DEC$ NOSTRICT`



注

プリフィックス `!DEC$` は一般的に使用されます。`!DEC$` は、固定形式と自由形式のソースの両方で有効に働きます。固定形式のソースでは、代替プリフィックスとして `cDEC$`、`CDEC$`、`*DEC$`、`cDIR$`、`CDIR$`、`*DIR$`、および `!MS$` を使用することもできます。

コードの生成オプション

コードの生成オプションでは、どのようにコードを生成するかを指定することができます。

コードの生成オプションの説明

`-[no]recursive`

デフォルト: `-norecursive`

すべてのプロシージャ（関数とサブルーチン）を、再帰的な実行が行われる可能性を念頭に置いてコンパイルします。`-recursive` オプションを指定すると、`auto` オプションが設定されます。

`-[no]reentrancy [keyword]`

デフォルト: `-noreentrancy`

マルチスレッド・アプリケーションをサポートする再入力可能コードを生成します。キーワードは以下のいずれかです。

- `none` `-noreentrancy` と同じです。プログラムがスレッド化または非同期の再入可能性に依存しないことをインテル® Fortran ランタイム・ライブラリ (RTL) に通知します。ランタイム・ライブラリは、そうした割り込みからそれ自体のクリティカル領域を防ぐ必要がなくなります。
- `async`
プログラムが RTL をコールできる非同期ハンドラを持っている可能性があることを RTL に通知します。RTL は、非同期割り込みからそれ自体のクリティカル領域を防ぎます。
- `threaded`
プログラムがマルチスレッド化されていることを RTL に通知します。ランタイム・ライブラリは、スレッド・ロッキングによって、それ自体のクリティカル領域を保護します。 `-threads`

を指定すると、マルチスレッド・コードは再入可能でなくてはならないので `-reentrancy threaded` が設定されます。
`-reentrancy` を指定することは `-reentrancy` をスレッド化することと同じです。

`-sox[-]` (IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ)

デフォルト: `-sox-`

コンパイラ・オプションとバージョンの実行ファイルへの保存を有効にします。

このオプションは、Itanium® システムには効果がありません。

互換性オプション

互換性オプションは、ソースファイルおよびデータファイルが以前の Fortran バージョンやその他のオペレーティング・システムにおける、ビッグ・エンディアン形式の書式なしデータファイル、OpenVMS* システムでのランタイム動作、および Microsoft* Fortran PowerStation などの互換性をどのように確保するかを指定します。

関連情報

[データ・オプション](#)

[言語オプション](#)

互換性オプションの説明

`-1`

デフォルト: オフ

代替構文: `-onetrip`

DO ループを少なくとも 1 回実行するようコンパイラに指示します。また、`-[no]f66` も参照してください。

`-assume [no]bscc`

デフォルト: `-assume nobsc`

代替構文: `-nbs` は `-assume nobsc` と同じです。

バックスラッシュ文字 (\) を C 言語の文字リテラルで使用される制御 (エスケープ) 文字として処理するようコンパイラに指示します。デフォルト値の `-assume nobsc` ("assume no BackSlashControlCharacters") を指定した場合、バックスラッシュ文字を文字リテラルで使用される制御文字ではなく、通常の文字として処理するようコンパイラに指示します。

このオプションは、プログラムを UNIX* 以外の環境、例えば OpenVMS* から移行するときに役立ちます。

`-convert`

デフォルト: `-convert native`

数値データが含まれている書式なしファイルの形式を指定します。設定可能な値は以下のとおりです。

- `-convert big_endian`
- `-convert cray`
- `-convert ibm`
- `-convert little_endian`
- `-convert native`
- `-convert vaxg`
- `-convert vaxd`

詳細は「[サポートされるネイティブ数値形式と非ネイティブ数値形式](#)」を参照してください。

`-[no]f77rtl`

デフォルト: `-nof77rtl`

FORTRAN 77 ランタイム動作の使用を指定します。デフォルト値 (`-nof77rtl`) を使用した場合、インテル® Fortran ランタイム動作が使用されます。

このオプションを指定すると、次のランタイム動作を制御します。

- ユニットがファイルに結合されていない場合、一部の INQUIRE 指定子は異なる値を返します。
NUMBER は 0 を返します。ACCESS、BLANK、および FORM は 'UNKNOWN' を返します。

- リスト指定入力の文字列は、アポストロフィまたは引用符で区切る必要があります。区切らなかった場合、エラーが発生します。
- NAMELIST 入力进行处理する場合、各レコードの列 1 は、スキップされます。さらに、グループ名の前に表示される '\$' または '&' は、入力レコードの列 2 に表示する必要があります。

-fpscomp all および -fpscomp none

デフォルト: `-fpscomp libs`

Microsoft* Fortran PowerStation と互換性のあるすべての `-fpscomp` オプションを使用するよう指定します。デフォルトのオプションは、PowerStation 移植ライブラリをリンクに渡すよう指定します。

`-fpscomp none` オプションは、Fortran PowerStation と互換性のあるオプションを一切使用しないよう指定します。

-fpscomp [no]filesfromcmd

デフォルト: `-fpscomp nofilesfromcmd`

OPEN 文のファイル指定子が空の場合に行う Microsoft* Fortran PowerStation の動作を指定します。このオプションは、コマンドライン引数の `OPEN(... FILE='...', ...)` 文で、指定されていないファイル名を検索して、端末コンソールでファイル名の入力を要求します。

-fpscomp [no]general

デフォルト: `-fpscomp nogeneral`

インテル Fortran と PowerStation 間における違いで Microsoft* Fortran PowerStation セマンティクスを使用するよう指定します。

-fpscomp [no]ioformat

デフォルト: `-fpscomp noioformat`

Microsoft* Fortran PowerStation セマンティクス規則、およびリスト指定の書式付き I/O および書式なし I/O のレコード書式を指定します。

-fpscomp [no]ldio_spacing

デフォルト: `-fpscomp noldio_spacing`

リスト出力で、文字値（非区切り文字列）の前で数値の後に実行時に空白を挿入するかどうかを指定します。デフォルトは `-fpscomp noldio_spacing` で、文字値の前で数値の後に空白を挿入します（Fortran 95 規格に準拠）。Fortran PowerStation およびインテル Fortran の 8.0 以前のバージョンとの互換性のために非標準の動作が必要な場合は、`-fpscomp ldio_spacing` または `-fpscomp general (-fpscomp ldio_spacing がセットされる)` のいずれかを指定します。

-fpscomp [no]libs

デフォルト: `-fpscomp libs`

PowerStation 移植ライブラリをリンクに渡すよう指定します。

-fpscomp [no]logicals

デフォルト: `-fpscomp nologicals`

LOGICAL 値の内部バイナリ表現と使用方法を指定します。

`nologicals` が指定された場合、奇数の整数値（最下位ビットが 1）は TRUE、偶数の整数値（最下位ビットが 0）は FALSE として処理されます。リテラル定数 `.TRUE.` は整数値 -1、リテラル定数 `.FALSE.` は整数値 0 になります。

`logicals` が指定された場合、ゼロ以外の整数値は TRUE、ゼロは FALSE として処理されます。リテラル定数 `.TRUE.` は整数値 1、リテラル定数 `.FALSE.` は整数値 0 になります。

デフォルト (`-fpscomp nologicals`) は Compaq* Fortran と互換性があります。インテル Fortran の 8.0 より前のバージョンでは、`-fpscomp logicals` で指定された表現が使用されていました。

LOGICAL 値の内部表現は Fortran 規格では指定されていません。LOGICAL コンテキストで整数値を使用するプログラム、または LOGICAL 値を他の言語で記述されたプロシージャに渡すプログラムは可搬性がなく、正しく動作しません。インテルでは、LOGICAL 値の内部表現に依存するコーディングを行わないことを強く推奨します。

-prof_format_32 (IA-32 および Itanium® ベース・システムのみ)

デフォルト: オフ

32 ビットカウンタ付きのプロファイル・データを生成します。デフォルトでは、64 ビットカウンタ付きのプロファイル・データを生成して、多くのイベントを処理します。

このオプションを使用して、以前のバージョンのコンパイラとの互換性を維持します。

-vms

デフォルト: オフ

ランタイムのシステムで、OpenVMS Alpha システムおよび VAX* システム (VAX FORTRAN*) 上の HP Fortran のように次の動作を行います。

- 一部のデフォルト設定
他のオプションが存在しないため、`-vms` は `-check format` および `-check output_conversion` をデフォルトとして設定します。
- アライメント
`-vms` オプションは、レコード内のフィールドおよび共通ブロックに含まれた項目のアライメントには影響しません。OpenVMS システム上の HP Fortran との互換性を確保するために、`-align norecords` を使用して、次のバイト境界上にあるレコードのフィールドをパックします。
- キャリッジ制御のデフォルト設定
`-vms -ccdefault default` が指定されている場合、ファイルの書式が設定されていて、なおかつユニットが端末に結合されていると、キャリッジ制御は FORTRAN をデフォルトとして設定します。
- INCLUDE 修飾子
コンパイル時に、INCLUDE 文でのファイル名の最後に記述された `/LIST` と `/NOLIST` が認識されます。
INCLUDE 文でのファイル名が完全パスで指定されていない場合、現在のディレクトリがパスとして使用されます。
`-vms` が指定されていない場合、INCLUDE 文を含むファイルが保存されたディレクトリのパスが使用されます。
- 引用符文字
引用符 (") 文字は、文字リテラル ("...") ではなく 8 進定数 ("0..7") として認識されます。
- 相対ファイル内の削除されたレコード
相対ファイル内のレコードが削除されると、そのレコードの最初の 1 バイトは既知の文字 (現在は '@') に設定されます。後でこのレコードの読み取りを試みると、ATTACCNON エラーが発生します。残りのレコード (`-vms` が設定されていない場合はレコード全体) は、書式なしファイルの場合はヌルに、書式付きファイルの場合は空白に設定されます。
- ENDFILE レコード
ENDFILE がシーケンシャル・ユニットに対して実行されると、CTRL/Z を含む 1

バイトのレコードがファイルに書き出されます。-vms が指定されていない場合、内部の ENDFILE フラグが設定され、ファイルは切り捨てられます。

-vms オプションは相対ファイルの ENDFILE には影響しません。これらのファイルは切り捨てられます。

- 暗黙的な論理ユニット番号
-vms オプションを指定することで、ランタイム時の ACCEPT 文、PRINT 文、および TYPE 文、そしてユニット番号を指定しない READ 文および WRITE 文（例えば READ (*,1000)）でインテル Fortran が特定の環境変数を認識します。
- 入力に含まれた空白の処理
-vms オプションは、外部または内部ファイルにおける OPEN 文の BLANK キーワードのデフォルトを明示的な OPEN 文の場合は 'NULL' に、暗黙的な OPEN 文の場合は 'ZERO' として処理します。詳細は、OPEN 文の説明を参照してください。
- OPEN 文の効果
キャリッジ制御は、ファイルの書式が設定されていて、なおかつユニットが端末に結合されている場合 (isatty(3) によって確認されます)、FORTRAN をデフォルトとして設定します。それ以外の場合は、キャリッジ制御は LIST をデフォルトとして設定します。
-vms オプションは、直接アクセスおよび相対編成ファイルのレコード長に影響します。削除されたレコード文字に応じてバッファのサイズが 1 増えます。
- 削除されたレコードおよび ENDFILE レコードの読み取り
ランタイムの直接アクセス READ ルーチンは、取り出したレコードの最初の 1 バイトを確認します。このバイトが '@' またはヌル ("0") の場合、ATTACCNON エラーを返します。
ランタイムのシーケンシャル・アクセス READ ルーチンは、読み取ったレコードに 1 バイトの長さで、CTRL/Z が含まれているかどうかを確認します。そうであれば、ファイル終了 (EOF) を戻します。

コンパイル診断オプション

コンパイル診断オプションは、取得する診断メッセージ（警告とエラー）を指定します。

コンパイル診断オプションの説明

-e90 または -e95

デフォルト: オフ

代替構文: -w90 または -w95

非標準の Fortran 90 (-e90) または非標準の Fortran 95 (-e95) のエラーを報告します。コンパイル時に識別可能な Fortran 言語では標準ではない言語要素を発見すると、コンパイル時エラーを発行します。

-[no]stand も参照してください。

-[no]error_limit n

デフォルト: `-error_limit 30`

1 つのファイルで、許容されるエラーまたは致命的なエラーの数の最大値を指定します。設定された値に達するとコンパイルは中断されます。コマンドラインで `-noerror_limit` を指定する場合、エラーの数に制限はありません。エラーが上限に達した場合、警告メッセージが報告され、コマンド行の次のファイルが（存在していれば）コンパイルされます。

-openmp_report{0|1|2}

デフォルト: オフ `-openmp_report1` は、引数なしで `-openmp_report` が指定されている場合のデフォルトです。

OpenMP* 並列化の診断レベルを指定します。*n* の値は次のとおりです:

- 0 診断情報を表示しません。
- 1 並列化されたループ、領域、およびセクションを示す診断を表示します。
- 2 1 の診断に加えて、主構造、単一構造などを示す診断を表示します。

詳細は、『ユーザーズ・ガイド Vol II: アプリケーションの最適化』の「OpenMP*による並列化の概要」(および関連セクション)を参照してください。

-par_report{0|1|2|3}

デフォルト: オフ `-par_report1` は、引数なしで `-par_report` が指定されている場合のデフォルトです。

自動並列化の診断レベルを指定します。*n* の値は次のとおりです:

- 0 診断情報を表示しません。
- 1 正常に並列化されたループを表示します。
- 2 正常に自動並列化されたループおよび並列化が失敗したループを表示します。
- 3 2 の診断に加えて、依存情報を表示します。

このオプションに関するトピックは、次の Vol II のトピックも参照してください。

自動並列化の概要

自動並列化: 有効、オプション、ディレクティブ、および環境変数

-std, -std90, -std95

デフォルト: オフ (メッセージは表示されません)

代替構文: -[no]stand、-w90 または -stand90 (Fortran 90 の場合)、-w95 または -stand95 (Fortran 95 の場合)

-std、-stand、-std95、-stand95 (等価) は非標準の Fortran 95 に対して警告します。-std90 と -stand90 (等価) は、非標準の Fortran 90 に対して警告します。

このオプションを指定すると、コンパイル時に識別可能な Fortran 言語では標準ではない言語要素を発見すると、コンパイル時メッセージを発行します。

-w90 および -w95 は、それぞれ Fortran 90 と Fortran 95 の 非標準 Fortran の警告をオフにします。

-stand は、-warn stderrors を指定した場合に設定されます。

-vec_report[0|1|2|3|4|5] (IA-32 およびインテル® EM64T システムのみ)

デフォルト: オフ -vec_report1 は、引数なしで -vec_report が指定されている場合のデフォルトです。

ベクトライザの診断レベルを指定します。 n の値は次のとおりです:

- 0 診断情報を表示しません。
- 1 ベクトル化されたループを表示します。
- 2 ベクトル化されたループまたは非ベクトル化ループを表示します。
- 3 ベクトル化されたループと依存情報を表示します。
- 4 非ベクトル化ループを表示します。
- 5 非ベクトル化ループを表示し、ベクトル化されない理由を示します。

詳細は、『ユーザーズ・ガイド Vol II: アプリケーションの最適化』の「ベクトル化の概要」(および関連セクション)を参照してください。

-warn all、-warn none、-nowarn

デフォルト: カスタム (個別に指定)。

コンパイラの診断レベル。次のオプションがあります:

- `-warn all` (すべての診断を表示)
- `-warn none` (診断を非表示)

`-warn all` を指定すると、すべての警告メッセージが表示されますが、`-warn errors` または `-warn stderrs` は設定しません。すべての追加検証を実行し、診断の重大度をオブジェクト・ファイルの生成を妨げるレベルに上げるには、`-warn all -warn errors` または `-warn all -warn stderrs` を指定します。

`-warn` は `-warn all` を指定することと同じです。

`-nowarn` は `-warn none` を指定することと同じです。

`-warn [no]alignments`

デフォルト: `-warn alignments`

自然にアライメントが合っていないデータに対し警告メッセージを発します。

`-warn [no]declarations`

デフォルト: `-warn nodeclarations`

宣言されていない記号に対してエラー・メッセージを表示します。このオプションは、暗黙的な Fortran 規則を使用するのではなく、デフォルトの型を未定義にします (IMPLICIT NONE)。`-u` オプションも参照してください。

`-warn [no]errors`

デフォルト: `-warn noerrors`

標準警告を含むすべての警告診断をエラー診断に変更することで、すべての警告をエラーとして扱います。

`-warn [no]general`

デフォルト: `-warn general`

代替構文: `-W1` (すべての警告を表示)、`-W0`、`-w` (すべての警告を非表示)

コンパイラからのすべての情報レベル診断メッセージおよび警告レベル診断メッセージを表示します。

`-warn nogeneral`、`-nowarn`、`-W0` または `-w` を指定するとすべての警告を非表示にできます。

`-warn [no]ignore_loc`

デフォルト: `-warn noignore_loc`

引数から `%LOC` が削除されると警告メッセージを表示します。

`-warn [no]stderrs`

デフォルト: `-warn nostderrors`

Fortran 標準に違反する警告を警告ではなく、エラーとして扱います。

`-warn stderrs` を指定すると `-stand f95` を設定します。

Fortran 90 標準に対する違反をエラーにする場合には、`-stand f90` とともにこのオプションを指定します。

`-warn [no]truncated_source`

デフォルト: `-warn notruncated_source`

固定形式ソースファイルでカラムの幅が最大値を超えるステートメント・フィールドを持つソース行を読み取る際に、警告メッセージを表示します。固定形式ファイルのカラム幅の最大値は 72、80、または 132 で、`-extend_source` オプションを設定するかしないかにより異なります。`-warn truncated_source` オプションは、切り捨てには影響ありません。最大カラム幅を超える行は常に切り捨てられます。`-warn truncated_source` オプションは、自由形式のソースファイルには適用されません。

`-warn [no]uncalled`

デフォルト: `-warn uncalled`

文関数が呼び出されない場合に警告メッセージを表示します。

`-warn [no]unused`

デフォルト: `-warn nounused`

宣言されているが、使用されていない変数に対して警告メッセージを表示します。

-warn [no]usage

デフォルト: `-warn usage`

代替構文: `-cm (-warn nousage と等価)`

問題のありそうなプログラミングに対するメッセージを非表示にします。

問題のありそうなプログラミングは、許容されるにせよ、プログラミング・エラーの結果であることがよくあります。例えば、`-warn usage` は、継続文字または Hollerith リテラルの最初の部分が、ステートメント・フィールドが終わる前に、末尾の空白で終わっているような場合です。

データ・オプション

データ・オプションは、Fortran データがコンパイル、最適化、およびコードの生成によって扱われる際の規則を指定します。

次のトピックも参照してください。

[互換性オプション](#)

[言語オプション](#)

データ・オプションの説明

-align none

デフォルト: 構造体のみにパディングを追加。(共通ブロックにはパディングは追加されません。)

代替構文: `-noalign`

共通ブロックまたは構造体にパディング・バイトを追加しないようコンパイラに命令します。これは `-noalign` と同じです。

-align [no]commons または -align [no]dcommons

デフォルト: `-align nocommons` または `-align nodcommons`

自然境界上にあるすべての共通ブロックに含まれたデータ項目を、デフォルトのバイト境界ではなく、最大 4 バイト (`-align commons`) または最大 8 バイト (`-align dcommons`) まで、パディング・バイトを追加してアライメントします。

コマンドラインに `-stand オプション` が含まれている場合、コンパイラは `-align dcommons` を無視します。

`-align recnbyte`

デフォルト: `-align rec16byte` (これは、`-Zp16` または `-align records` と同じです。)

代替構文: `-Zp{1|2|4|8|16}`

構造体に、1、2、4、8、または 16 バイトの境界整列条件を指定します。

レコードのフィールドおよび派生型のコンポーネントを、指定された境界 (n には 1、2、4、8、または 16 を指定できます) またはそれらが自然にアライメントされる境界のうち、サイズがより小さくなる境界でアライメントします。

`-align recnbyte` の指定によって、共通ブロックが自然にアライメントされるか、パックされるかを制御することはできません。

このオプションは、次の操作を行います。	同等のオプション
<code>-Zp</code>	<code>-align records</code> または <code>-align rec16byte</code>
<code>-Zp1</code>	<code>-align records</code> または <code>-align rec1byte</code>
<code>-Zp2</code>	<code>-align rec2byte</code>
<code>-Zp4</code>	<code>-align rec4byte</code>
<code>-Zp8</code>	<code>-align rec8byte</code>
<code>-Zp16</code>	<code>-align rec16byte</code> または <code>-align records</code>
<code>/align:none</code>	<code>-noalign</code>
<code>-align all</code>	次のすべてのオプションを指定します。 <code>align nocommons</code> 、 <code>-align dcommons</code> 、 <code>-align records</code> 、 <code>-align nosequence</code>

`-align [no]records`

デフォルト: `-align records`

派生型のコンポーネントおよびレコードのフィールドを最大 8 バイト (派生型に `SEQUENCE` 文を指定した場合) まで、パディングを追加して自然境界上にアライメントするように指定します。また、`-align sequence` も参照してください。

`-align norecords` オプションは、最大 8 バイトの自然境界上ではなく、パディングを追加せずにコンポーネントおよびフィールドを任意のバイト境界上にアライメントするように指定します。

`-align [no]sequence`

デフォルト: `-align nosequence`

SEQUENCE 属性が指定されている派生型のコンポーネントが、現在使用しているアライメント規則に従うことをコンパイラに伝えます。デフォルトのアライメント規則は、不整列のコンポーネントを自然境界上にアライメントします。

デフォルト値の `-align nosequence` を指定した場合、現在使用しているアライメント規則に関わらず、SEQUENCE 属性が指定されている派生型のコンポーネントをパックします。

コマンドラインに `-stand オプション` が含まれている場合、コンパイラは `-align sequence` を無視します。

`-assume [no]byterecl`

デフォルト: `-assume nobyterecl`

書式なしファイルに対するバイトユニットの使用を指定します。このオプションは、次の操作を行います。

- 明示的な OPEN 文の RECL 指定子の値がバイト単位であることを指定します。
- 出力リストで INQUIRE によって返されたレコード長の値を、強制的にバイト単位にします。

デフォルト値の `-assume nobyterecl` は、書式なしファイルの RECL 値が 4 バイト (ロングワード) 単位であることを指定します。

`-assume [no]dummy_aliases`

デフォルト: `-assume nodummy_aliases`

代替構文: `-common_args`

このオプションを使用した場合、コンパイラはプロシージャへの仮引数が他の仮引数と、または参照結合、ホスト結合、または共通ブロックを使用して共有した変数とメモリ位置を共有していることを仮定します。

`-assume dummy_aliases` を指定した場合、呼び出されるサブプログラムのみコ

ンパイルする必要があります。

仮エイリアシングに関連したプログラム・セマンティクスは、標準 Fortran に完全に従っていないため、パフォーマンスが低下します。したがって、デフォルト値の `-assume nodummy_aliases` を指定することで、プログラムのランタイム・パフォーマンスが向上します。しかし、仮エイリアシングに依存するプログラムで `-assume dummy_aliases` を指定しない場合、プログラムはランタイム時に予期せぬ動作を引き起こします。このようなプログラムの結果は、行われた最適化の内容に依存します。場合によっては、通常の結果が生じることもありますが、問題を引き起こすエイリアスが関わる演算に使われた値が変わったために、結果が変わる場合もあります。

仮エイリアシングの仮定に関する詳細については、『ユーザーズ・ガイド Vol II』の「アプリケーションの最適化」を参照してください。

`-assume [no]protect_constants`

デフォルト: `-assume protect_constants`

このオプションは、定数が読み取り専用であることを指定します。

`-assume noprotect_constants` オプションは、定数実引数のコピーを渡すようコンパイラに命令します。その結果、標準 Fortran では行うことができないコピーの変更を、呼び出されたルーチンで変更することができます。呼び出しルーチンは、定数を変更しません。

デフォルト値の `-assume protect_constants` を指定した場合、定数実引数を渡します。この値を変更した場合、エラーが発生する可能性があります。

`-auto_scalar`、`-auto`、および `-save`

デフォルト: `-auto_scalar` (`-recursive` または `-openmp` が指定されていない場合のデフォルトです。指定されている場合、デフォルトは `-auto` です)

`-auto` の代替構文: `-automatic` および `-nosave`

`-save` の代替構文: `-noauto` および `-noautomatic`

デフォルトのローカル記憶領域が不要な場合、変数が格納されている場所を指定します。

`-auto_scalar` オプションは、INTEGER、REAL、COMPLEX、および LOGICAL 組み込み型のスカラ変数をスタックに割り当てます。

`-auto_scalar` は、EQUIVALENCE 文または SAVE 文に含まれている変数、または共通ブロックに含まれている変数には作用しません。`-auto_scalar` を指定すると、プログラムのパフォーマンスが向上することがあります。ただし、プログラムが、最後にルーチンが呼び出されたときと同じ値を持つ変数に依存している場合は、プログラムが正しく機能しないことがあります。複数のサブルーチン呼び出しにわたってその値を保持する必要のある変数は、SAVE 文内になければなりません。

`-auto` は、すべての変数をローカル・スタティック・ストレージではなく、スタックに割り当てます。このオプションは、SAVE 属性が指定されている変数には影響しません。また、EQUIVALENCE 文や共通ブロックに含まれません。

`-save` オプションは、再帰ルーチン内のローカル変数を除き、すべての変数を保存します (静的割り当て)。

`-auto` を指定すると、プログラムのパフォーマンスが向上することがあります。ただし、プログラムが、最後にルーチンが呼び出されたときと同じ値を持つ変数に依存している場合は、プログラムが正しく機能しないことがあります。

`-double_size {64|128}`

デフォルト: `-double_size 64`

DOUBLE PRECISION 型および DOUBLE COMPLEX 型の宣言、定数、関数、および組込み関数を定義します。

`-double_size 64` オプションは、DOUBLE PRECISION 型の宣言、定数、関数、および組込み関数を `REAL*8` として定義し、DOUBLE COMPLEX 型の宣言、関数、および組込み関数を `COMPLEX*16` として定義します。

`-double_size 128` オプションは、DOUBLE PRECISION 型の宣言、定数、関数、および組込み関数を `REAL*16` として定義し、DOUBLE COMPLEX 型の宣言、関数、および組込み関数を `COMPLEX*32` として定義します (Fortran 90/95 用)。

`-dyncom "blk1,blk2,..."`

ランタイムで指定された COMMON ブロックの動的割り当てを有効にします。

例: 共通 ブロック A、B、および、C には次の構文を使用します:

```
-dyncom "a,b,c"
```

-integer_size {16|32|64}

デフォルト: `-integer_size 32`

代替構文: `-i{2|4|8}` (2、4、および 8 は、INTEGER および LOGICAL 変数の KIND を表します)

INTEGER 型または LOGICAL 型の宣言、定数、関数、および組込み関数のデフォルト・サイズ (ビット) を指定します。*n* には 16、32、または 64 を指定することができます。

- *n* が 16 の場合: デフォルトの INTEGER および LOGICAL 変数の長さは 2 バイトです。INTEGER 宣言および LOGICAL 宣言は、(KIND=2) として扱われます。代替構文: `-i2` また、[「INTEGER\(KIND=2\) の表現」](#)も参照してください。
- *n* が 32 の場合: デフォルトの INTEGER および LOGICAL 変数の長さは 4 バイトです。INTEGER 宣言および LOGICAL 宣言は、(KIND=4) として扱われます。代替構文: `-i4` また、[「INTEGER\(KIND=4\) の表現」](#)も参照してください。
- *n* が 64 の場合: デフォルトの INTEGER および LOGICAL 変数の長さは 8 バイトです。INTEGER 宣言および LOGICAL 宣言は、(KIND=8) として扱われます。代替構文: `-i8` また、[「INTEGER\(KIND=8\) の表現」](#)も参照してください。

-pg

デフォルト: オフ

`gprof(1)` を使用して、関数のプロファイリングができるようにコンパイルとリンクを行います。

これは `-p` または `-qp` を指定するのと同じです。

-real_size {32|64|128}

デフォルト: `-real_size 32`

代替構文: `-r{8|16}` (8 および 16 は、実数変数の KIND を表します)

REAL 型または COMPLEX 型の宣言、定数、関数、および組込み関数のデフォルト・サイズ (ビット) を指定します。*n* には 32、64、または 128 を指定することができます。

- *n* が 32 の場合: REAL 型の宣言、定数、関数、および組込み関数を REAL(KIND=4) (SINGLE PRECISION) として定義し、COMPLEX 型の宣言、定数、関数、および組込み関数を COMPLEX(KIND=4) (COMPLEX) として定義します。
代替構文: なし

次のトピックも参照してください。

[REAL\(KIND=4\) \(REAL\) の表現](#)

[COMPLEX\(KIND=4\) \(COMPLEX\) の表現](#)

- n が 64 の場合: REAL 型の宣言、定数、関数、および組込み関数を REAL(KIND=8) (DOUBLE PRECISION) として定義し、COMPLEX 型の宣言、定数、関数、および組込み関数を COMPLEX(KIND=8) (DOUBLE COMPLEX) として定義します。

`-real_size 64` を指定した場合、`CMPLX`、`FLOAT`、`REAL`、`SNGL`、および `AIMAG` を含む組込み関数は、引数が明示的に `REAL(KIND=4)` または `COMPLEX(KIND=4)` として指定されていない限り、`REAL(KIND=4)` または `COMPLEX(KIND=4)` の結果に代わり `REAL(KIND=8)` または `COMPLEX(KIND=8)` の結果を生成するようになります。例えば、`CMPLX` 組込み関数への参照は、`CMPLX` の引数が明示的に `REAL(KIND=4)`、`REAL*4`、`COMPLEX(KIND=4)`、または `COMPLEX*8` として指定されていない限り、`DCMPLX` の結果 (`COMPLEX(KIND=8)`) を生成します。この場合、結果として得られるデータ型は `COMPLEX(KIND=4)` になります。

代替構文: `-r8` または `-autodouble`

次のトピックも参照してください。

[REAL\(KIND=8\) \(DOUBLE PRECISION\) の表現](#)

[COMPLEX\(KIND=8\) \(DOUBLE COMPLEX\) の表現](#)

- n が 128 の場合: REAL 型の宣言、定数、関数、および組込み関数を REAL(KIND=16) として定義し、COMPLEX 型の宣言、定数、関数、および組込み関数を COMPLEX(KIND=16) として定義します。

代替構文: `-r16`

次のトピックも参照してください。

[REAL\(KIND=16\) の表現](#)

[COMPLEX\(KIND=16\) の表現](#)

`-safe_cray_ptr`

デフォルト: オフ (クレイポインタが他の変数のエイリアシングを行わない場合)

クレイポインタが他の変数のエイリアシングを行わない (つまり、他の変数との間で共有するメモリを指定しない) ことを仮定します。

次の例について考えてみます。

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
  b(i) = a(i) + 1
enddo
```

デフォルトでは、コンパイラは `b` と `a` がエイリアスされていると仮定します。このような仮定を回避するには、`-safe_cray_ptr` オプションを指定します。コンパイラは `b(i)` と `a(i)` がお互いに独立しているとして処理します。

しかし、変数をクレイポインタでエイリアスする場合、`-safe_cray_ptr` オプションを使用すると、正確な結果が得られません。下記の例のコードでは、`-safe_cray_ptr` オプションを使用しないでください。

```
pointer (pb, b)
pb = loc(a(2))
  do i=1, n
    b(i) = a(i) +1
  enddo
```

`-zero[-]`

デフォルト: オフ (`-zero-`)

INTEGER、REAL、COMPLEX、または LOGICAL 組込み型のローカルスカラー変数で、保存はされているがまだ初期化されていない変数を、すべてゼロに初期化します。

Use `-save` on the command line to make all local variables specifically marked as SAVE.

外部プロシージャ・オプション

外部プロシージャ・オプションを使用することで、外部プロシージャの呼び出し方法を指定することができます。

外部プロシージャ・オプションの説明

`-assume [no]underscore`

デフォルト: `-assume nounderscore`

代替構文: `-nus`

ユーザ定義の外部名に下線文字を追加します (メイン・プログラム名、名前付き COMMON、BLOCK DATA、MODULE 内のグローバル・データ名、および暗黙的または明示的に EXTERNAL と宣言された名前)。名前のない COMMON は、`_BLNK_` のまま残され、Fortran 組込み名は影響を受けません。

-[no]mixed_str_len_arg

デフォルト: `-nomixed_str_len_arg`

文字引数の隠された文字長を、引数リストの対応する文字引数の直後に置くようにコンパイラに要求します。

デフォルト値では、隠された文字長は、引数リストの終わりにシーケンシャル順で置かれます。文字引数の受け渡しを行う、言語が混在したプログラムを移植するときには、このオプションを正しく指定するか、またはソースコード内で隠された文字長を持つ文字引数の順序を変更する必要があります。

詳細は、「[言語が混在したプログラミングの概要](#)」および関連するセクションを参照してください。

-names keyword

デフォルト: `-names lowercase`

ソースコード識別子と外部名で、どのように大文字・小文字を扱うかを決定します。この命名規則は、名前が定義されているときにも、参照されているときにも適用されます。このオプションは、言語が混在したプログラミングでとても便利です。

keyword には、次のいずれかを代入します:

- `uppercase`: コンパイラは、識別子内の大文字・小文字の違いを無視し、外部名を大文字に変換します。代替構文: `-uppercase`
- `lowercase`: コンパイラは、識別子内の大文字・小文字の違いを無視し、外部名を小文字に変換します。代替構文: `-lowercase`
- `as_is`: コンパイラは、識別子内の大文字・小文字の違いを区別し、外部名の大文字・小文字の違いを保持します。

このオプションの代わりに、必要な名前に対して `ALIAS` ディレクティブを使用することを推奨します。

浮動小数点オプション

浮動小数点オプションでは、浮動小数点データの扱い方を指定することができます。

「[最適化オプション](#)」も参照してください。

浮動小数点オプションの説明

`-assume [no]minus0`

デフォルト: `-assume nominus0`

プロセッサが `-0.0` と `+0.0` を判別でき、書式付き出力に `-0.0` の値を負の符号で書き出すことができる場合、SIGN 組込み関数の IEEE* 浮動小数点値 `-0.0` を扱う際に Fortran 95 の標準セマンティクスを使用するようコンパイラに指示します。

デフォルトは `-assume nominus0` です。これは SIGN 組込み関数で Fortran 90/77 の標準セマンティクスを使用するようコンパイラに指示します。`-0.0` と `+0.0` は `0.0` として扱われ、`-0.0` は負の符号なしで書式付き出力に書き出されます。

`-[no]fltconsistency`

デフォルト: `-nofltconsistency`

代替構文: `-mp`

計算中に浮動小数点の一貫性を向上できます。

`-fltconsistency` は、数値ライブラリ関数のインライン展開を無効にします。

このオプションで、最適化を制限し、宣言された精度を維持します。浮動小数点演算の順序変更はありません。また、各浮動小数点の結果は、次の計算に使用するために浮動小数点プロセッサで保持されずに、ターゲットの変数に格納されます。

例えば、コンパイラでは、分母の逆数によって浮動小数点数の除算計算を乗算に変えられます。ただし、これによって、浮動小数点数の除算計算の結果が多少変わることがあります。

`-fltconsistency` オプションが使用された場合 オプションが使用された場合、浮動小数点の中間結果は 10 バイトの内部精度が完全に保たれます。さらに、X87 浮動小数点レジスタのすべてのスピル/リロードは、制御できないスピル/リロードによって精度が変わらないように、内部フォーマットを使用して行われます。

このオプションは、デフォルトの浮動小数点最適化フラグを使用するとパフォーマンスの低下を引き起こします。この問題を回避するには、`-mp1` オプションを使用します。`-mp1` オプションは、パフォーマンスに影響をほとんど与えることなく、デフォルトの浮動小数点精度に近い精度を実現します。

デフォルト値の `-nofltnconsistency` を使用すると、浮動小数点数結果の一貫性が多少損なわれますが、精度とランタイム・パフォーマンスが向上します。

このオプションを使用すると、実行速度が多少遅くなる場合があります。

『Vol II: アプリケーションの最適化』の「浮動小数点演算の精度の向上と制限」も参照してください。

-fp_port (IA-32 システムのみ)

デフォルト: オフ

浮動小数点演算の後に、浮動小数点の結果を丸めます。そのため、代入時と型変換時にユーザが宣言した精度に丸められます。速度に多少影響があります。

デフォルトでは、浮動小数点演算の結果を高精度で維持します。これにより、浮動小数点演算の結果の一貫性が多少損なわれますが、より良いパフォーマンスを得ることができます。

『Vol II: アプリケーションの最適化』の「IA-32 システムの浮動小数点演算の精度」も参照してください。

-[no]fpconstant

デフォルト: `-nofpconstant`

倍精度変数に代入された単精度定数を倍精度で評価するように要求します。

Fortran 標準では、定数を単精度で評価するように定めています。FORTRAN-77 コンパイラ用に作成された一部のプログラムでは、倍精度変数に代入された単精度定数が倍精度で評価されることを前提としているために、浮動小数点演算の結果が変わることがあります。

次の例では、`-fpconstant` を指定すると、D1 と D2 に同じ値が代入されます。`-fpconstant` オプションを省略すると、コンパイラは標準に基づく動作を行い、D1 に精度の低い値を代入します。

```
REAL (KIND=8) D1, D2
DATA D1 /2.71828182846182/    !REAL (KIND=4) value expanded
to double
DATA D2 /2.71828182846182D0/ !Double value assigned to
double
```


-fpen

デフォルト: -fpe3

メイン・プログラムのランタイム時の浮動小数点例外処理を制御します。これには、例外浮動小数点値が許容されるかどうかや、ランタイム時例外がどの程度正確に報告されるかなどが含まれます。このオプションは、次の例外の処理方法を制御します:

- 浮動小数点演算の結果として、ゼロによる除算、オーバーフロー、または無効な演算が発生した場合。
- 浮動小数点演算の結果として、アンダーフローが発生した場合。
- 正規化されていない数字、またはその他の例外値（正の無限大、負の無限大、または NaN）が算術式中に含まれている場合。

次のオプションを選択できます。

- -fpe0 指定子: アンダーフローは 0.0 です。その他の IEEE 例外ではアボートします。
- -fpe3 指定子: NaN、符号付き無限大、デノーマル結果を生成します。

IA-32 システムで -fpe0 オプションを使用するとランタイム時のパフォーマンスが低下します。

多くのプログラムでは、正規化されていない数値またはその他の例外値を処理する必要はありません。Itanium® ベース・システムで -fpe3 を使用するとランタイム時のパフォーマンスが低下します。

-fpstkchk (IA-32 システムのみ)

デフォルト: オフ

関数呼び出し後に余分なコードを生成し、浮動小数点 (FP) スタックを例外状態にします。

デフォルトでは、チェックは行われません。そのため、FP スタックにオーバーフローが発生すると、NaN 値が FP 計算に加えられ、プログラムの結果に違いが生じます。そのため、オーバーフロー・ポイントが実際のバグの場所からかなり離れていることもあります。-fpstkchk オプションは、不正な呼び出しが発生した直後にアクセス違反を行うコードを配置して、このような問題を見つけやすくします。

-fr32 (Itanium ベース・システムのみ)

デフォルト: オフ

高い精度を持つ浮動小数点レジスタの無効に指定します。

-ftz[-]

デフォルト: オフ (-ftz-)

デノーマル結果を 0 にフラッシュします。このオプションは、メイン・プログラムをコンパイルするときのみ効果があります。

Itanium ベース・システムでは、-O3 オプションを使用すると -ftz オプションが自動的に設定されます。

-IPF_ftl_eval_method0 (Itanium ベース・システムのみ)

デフォルト: オフ

プログラムで宣言された変数型により指定される精度での浮動小数点演算を含む式を評価するようにコンパイラに指示します。デフォルトでは、中間浮動小数点式は高い精度で保たれます。

『Vol II: アプリケーションの最適化』の「Itanium® ベースのシステムに使用する浮動小数点演算オプション」も参照してください。

-IPF_ftlacc[-] (Itanium ベース・システムのみ)

デフォルト: オフ

浮動小数点の精度に影響を与える最適化を無効にします。

デフォルト設定 (-IPF_ftlacc-) が使用された場合、コンパイラは浮動小数点の精度を低下する最適化を適用します。

浮動小数点の精度を高めるために -IPF_ftlacc または [-fltconsistency](#) を使用するには、いくつかの最適化を無効にする必要があります。

『Vol II: アプリケーションの最適化』の「Itanium® ベースのシステムに使用する浮動小数点演算オプション」も参照してください。

-IPF_fma[-] (Itanium ベース・システムのみ)

デフォルト: オフ

浮動小数点積和/積差演算を 1 つの演算命令で行うことを有効にします。

-IPF_fp_relaxed[-] (Itanium ベース・システムのみ)

デフォルト: `-IPF_fp_relaxed-`

`divide` や `sqrt` のような数値演算関数で、より高速な、しかし多少精度が低いコード・シーケンスを使用します。厳密な IEEE* 精度と比較すると、このオプションはこれらの関数で行われる浮動小数点計算の精度を少し下げます (通常は最下位の桁に制限される)。

-IPF_fp_speculationmode (Itanium ベース・システムのみ)

デフォルト: `-IPF_fp_speculationfast`

次のいずれかのモードで、浮動小数点演算を有効にします。

- `fast` 浮動小数点演算をスペキュレーションします。
- `safe` 安全な場合のみ浮動小数点演算のスペキュレーションを有効にします。
- `strict` 浮動小数点演算のスペキュレーションを無効にします。
- `off` `strict` と同じです。

『Vol II: アプリケーションの最適化』の「Itanium® ベースのシステムに使用する浮動小数点演算オプション」も参照してください。

-mp1

デフォルト: オフ

浮動小数点を宣言された精度にできるだけ準拠するように制限します。オプションは、速度に多少影響を与えますが、`-mp` の場合ほどではありません。

-pc{32|64|80} (IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ)

デフォルト: `-pc64`

浮動小数点の仮数部の精度を制御できるようにします。設定可能な値は以下のとおりです。

- `-pc32` 内部 FPU 精度を 24 ビットの仮数に設定します。
- `-pc64` 内部 FPU 精度を 53 ビットの仮数に設定します。
- `-pc80` 内部 FPU 精度を 64 ビットの仮数に設定します。

言語オプション

言語オプションを設定することで、セマンティクス、構文、およびソースファイルの形式を指定できます。

次のトピックも参照してください。

[互換性オプション](#)

[データ・オプション](#)

言語オプションの説明

-[no]altparam

デフォルト: `-altparam` (代替構文を使用可能にします)

代替構文: `-[no] dps`

PARAMETER 文で、代替構文を使用できます (括弧は必要ありません)。

PARAMETER 文の代替構文は以下のとおりです:

```
PARAMETER par1=exp1 [, par2=exp2] ...
```

この形式では、パラメータ名へ定数を代入する際、その前後に括弧をつけません。パラメータの型は、暗黙的な型変換ではなく、割り当てられた式の型により決定されます。

-[no]d_lines

デフォルト: `-nod_lines`

代替構文: `-DD`

第 1 列に D という文字を含む固定形式ファイルの行を、コメント行ではなくソースコードとして処理するように指定します。

-[no]extend_source [size]

デフォルト: `-noextend_source` (72 文字)。 *size* を指定せず、`-extend_source` のみを指定した場合、デフォルトが `-extend_source:132` に変更されます。

代替構文: `-{72|80|132}`

固定形式ソースファイルのステートメント・フィールドの終点を制御するために、列番号を 72、80、または 132 から指定します。size が指定されると、それがステートメント・フィールドの一部として解析される最後の列になります。それ以降の列はすべてコメントとして扱われます。

このオプションは、固定形式のファイルのみに使用できます。

`-[no]f66`

デフォルト: `-nof66` (現在の Fortran 標準セマンティクスを使用します)

代替構文: `-66`

互換性の問題が生じたときに、コンパイラが FORTRAN-66 の解釈を選択するように指定します。このオプションにより、次のような違いが生じます:

- DO ループが、常に少なくとも 1 回実行されます。
- FORTRAN-66 EXTERNAL 文の構文とセマンティクスが使用可能になります。
- OPEN 文の STATUS 指定子が省略された場合、デフォルトが STATUS='UNKNOWN' の代わりに STATUS='NEW' に変更されます。
- OPEN 文の BLANK 指定子が省略された場合、デフォルトが BLANK='ZERO' の代わりに BLANK='NULL' に変更されます。

`-[no]free` または `-[no]fixed`

デフォルト: ファイル拡張子を使用して、ファイル形式を決定します。

代替構文: `-FR` と `-free` は同じです。また、`-FI` も `-fixed` と同じです。

Fortran ソースコードの形式を指定します。このオプションが指定されていない場合、ファイル拡張子によって形式が決定されます。

- `.f90`、`.F90`、または `.i90` の拡張子をもつファイルは、自由形式のソースファイルになります。
- `.f`、`.for`、`.FOR`、`.ftn` または `.i` の拡張子をもつファイルは、固定形式のファイルになります。

`-openmp` または `-openmp_stubs`

デフォルト: オフ (無効)

OpenMP* デイレクティブを処理するように指定します。有効なオプションは以下のとおりです:

- `-openmp` 並列コードを生成します。このオプションが指定されると、マルチスレッド・ライブラリは使用されますが、`fpp` は自動的に実行されません。
`-openmp` が指定された場合は、`-auto` オプションも設定されます。
- `-openmp_stubs` シーケンシャル・コードを生成します。OpenMP デイレクティブが無視され、OpenMP スタブ・ライブラリがリンクされます。

`-[no]pad_source`

デフォルト: `-nopad_source`

ステートメント・フィールドの幅より短い固定形式のソース行に対して、スペースをステートメント・フィールドの終わりに追加するように指定します。このオプションは、複数のソースレコードにわたって続く文字リテラルと Hollerith リテラルの解釈に影響を与えます。

デフォルト設定である `-nopad_source` が使用されると、ステートメント・フィールドの終点よりも前で終わっている文字リテラルまたは Hollerith リテラルが、次のソースレコードにわたって続いている場合、警告メッセージが表示されます。この警告メッセージを抑止するには、`-warn nouseage` オプションを指定します。

`-pad_source` を指定すると、`-warn usage` に関連する警告メッセージは表示されません。

ライブラリ・オプション

ライブラリ・オプションを設定することで、アプリケーションで使用するライブラリを指定できます。

ライブラリ・オプションの説明

`-no_cpprt`

デフォルト: `-cxxlib-icc` (インテル® C++ ライブラリを使用します)

C++ ランタイム・ライブラリをリンクしません。

このオプションは GNU との互換性を維持するために存在し、リンク時の `cpp` ランタイム・ライブラリの使用を無効にします。`-cpprt` および `-yes_cpprt` オプションは存在しません。

`-nodefaultlibs`

デフォルト: オフ (デフォルトのライブラリをインクルードします)

リンク時に標準ライブラリを使用します。

このオプションは GNU との互換性を維持するために存在します。`-defaultlibs` オプションは存在しません。

`-nostdlib` も参照してください。

`-i_dynamic`

デフォルト: オフ

インテルが提供するライブラリへ動的にリンクします。

`-Ldir`

デフォルト: オフ

リンカがライブラリを検索するディレクトリ *dir* を指定します。

`-[no]threads`

デフォルト: `-nothreads`

マルチスレッド・ライブラリをリンクさせるかどうかを指定します。

`-threads` を指定すると、`-reentrancy threaded` オプションが有効になります。

`-nostdlib`

デフォルト: オフ

リンク時に標準ライブラリおよび起動ファイルを使用します。

このオプションは GNU との互換性を維持するために存在します。`-stdlib` は存在しません。

-shared

デフォルト: オフ

実行ファイルの代わりに、動的共用ファイル (DSO) をビルドします。

[「共有ライブラリの作成」](#)も参照してください。

-shared-libcxa

インテルが提供する libcxa C++ ライブラリを動的にリンクします。

デフォルトでは、libcxa ライブラリは動的にリンクされます。(インテルが提供するすべての C++ 関連のライブラリは、デフォルトで動的にリンクされます)。-static オプションを使用し、libcxa ライブラリに対して指定した -static オプションの効果を無効にするときに、このオプションは役立ちます。

このオプションは、[-static-libcxa](#) と逆の効果を持ちます。

-static-libcxa

インテルが提供する libcxa C++ ライブラリを静的にリンクします。

デフォルトでは、libcxa ライブラリは動的にリンクされます。(インテルが提供するすべての C++ 関連のライブラリは、デフォルトで動的にリンクされます)。このオプションを使用することにより、libcxa を静的にリンクできます。このとき、デフォルト動作である標準ライブラリへのリンクも可能です。

[-shared-libcxa](#) [オプション](#)も参照してください。

-static

デフォルト: オフ

代替構文: -non_shared

共有ライブラリ (.so ファイル) とリンクしないようにします。

その他のオプション

アルファベット順に説明します。

-ansi_alias[-]

デフォルト: `-ansi_alias`

コンパイラは、プログラムが Fortran 95 標準の別名規則に準拠していると仮定します。

例えば、実数型のオブジェクトは、整数型としてアクセスできません。データ型およびデータ型定数の規則についての詳細は、『Language Reference』(英語) の「Data Types, Constants, and Variables」を参照してください。

このオプションは、次の点を前提にしてコンパイラに指示します。

- 配列は、配列の外部からアクセスされません。
- ポインタを非ポインタ型にキャストしません。また、その逆もしません。
- 2 つの異なるスカラ型のオブジェクトへの参照にはエイリアスを設定できません。例えば、整数型のオブジェクトは実数型のオブジェクトと、または実数型のオブジェクトは倍精度型のオブジェクトとエイリアスの設定はできません。

プログラムが上記の条件を満たす場合、`-ansi_alias` オプションを設定することでプログラムの最適化が向上します。ただし、プログラムが上記の条件を満たさない場合、コンパイラが誤ったコードを生成する可能性があるため、`-ansi_alias-` でこのオプションを無効にする必要があります。

-assume cc_omp

デフォルト: `-openmp` の指定の有無によりデフォルトは異なります。

OpenMP Fortran API により条件付きコンパイルが有効になります。`"!$space"` が自由形式ソースにある時、または `"!$spaces"` が固定形式のカラム 1 にある時は、残りの行は Fortran 行として受け付けられます。

`-openmp` が指定された場合、デフォルトは `-assume cc_omp` です。指定されていない場合は、デフォルトは `-assume nocc_omp` です。

-assume none

`-assume` オプションをオフにします。

-nobss_init

デフォルト: オフ

ゼロに初期化された変数の BSS 内への配置を無効にします。

デフォルトでは、明示的にゼロに初期化される変数は BSS セクションに配置されます。このオプションを使用すると、明示的にゼロに初期化されるどの変数でも必要に応じて DATA セクションに配置することができます。

-bss_init オプションはありません。

-ccdefault keyword

デフォルト: -ccdefault default

ユニット 6 および * で使用されるキャリッジ制御の種類を指定します。

keyword は以下のいずれかです。

- **default** デフォルトのキャリッジ設定を使用するように指定します。このオプションは **-vms オプション**の影響を受ける場合があります。
-vms -ccdefault default が指定されている場合、ファイルの書式が設定されていて、なおかつユニットが端末に結合されていると、キャリッジ制御は **fortran** をデフォルトとして設定します。
-novms -ccdefault default が指定されている場合、キャリッジ制御は **list** をデフォルトとして設定します。
- **fortran** 最初の文字を標準 Fortran として解釈します。
- **list** レコード間にラインフィードを追加します。
- **none** キャリッジ制御を行いません。

-debug keyword

デフォルト: オフ

拡張デバッグの設定を指定します。このオプションを使用するには、-g オプションも指定する必要があります。

- **keyword variable_locations** は、スカラーローカル変数の検索に役立つ拡張デバッグ情報を出力します。「ロケーション・リスト」として知られる Dwarf オブジェクト・モジュールの機能を使用します。この機能は、ローカルスカラー変数のランタイム・ロケーションがより正確に指定されるようにします。つまり、変数値は、コード中で指定された位置で、メモリ中またはマシンレジスタ

で見つかります。インテル® デバッガ (idb) は、ロケーション・リストを処理して、ローカル変数値をより正確にランタイム時に表示することができます。

-dryrun

デフォルト: オフ

ドライバ・ツール・コマンドを表示します。ツールを実行しません。-v オプションも参照してください。

-dynamic-linkerfile

デフォルト: オフ

デフォルトに代わるダイナミック・リンカ (*file*) を指定します。

-fpic または -fPIC

デフォルト: オフ (-fpic-)

位置に依存しないコードを生成します。Itanium® ベース・システムで共有オブジェクトをビルドする際に必要です。

完全なシンボル・プリエンプションを指定します。グローバル・シンボル定義は、明示的に他の方法で指定されなければ、グローバル・シンボル参照と同様に default 可視属性 (つまり、プリエンプト可能) になります。

[「共有ライブラリの作成」](#)も参照してください。

-fminshared

デフォルト: オフ

コンパイル単位をメイン・プログラムのコンポーネントとして処理し、共有可能オブジェクトの一部としてリンクしないように指示します。

メイン・プログラムで定義されたシンボルはプリエンプトできないため、これによりコンパイラは default 可視属性で宣言されたシンボルを protected 可視属性であるかのように扱います。つまり、fminshared は fvisibility=protected を意味します。

コンパイラは位置に依存しないコードをメイン・プログラム用に生成する必要はありません。グローバル・オフセット・テーブル (GOT) のサイズを減らしてメモリのトラフィックを減らす、絶対アドレス指定を使用することができます。

-fvisibility=keyword と -fvisibility=keyword=file

グローバル・シンボルのデフォルトの可視属性を指定します (`-fvisibility=keyword`)。または、ファイル内のシンボルの可視属性を指定します (`-fvisibility=keyword=file`)。(2 つめの形式は 1 つめの形式を上書きします)。

keyword は、設定される可視属性を指定します。次のいずれかの可視属性を設定できます。

- `default` - 他のコンポーネントはシンボルを参照することができます。また、そのシンボル定義を別のコンポーネントの同じ名前の定義で上書き (プリエンプト) できます。
- `extern` - シンボルは別のコンポーネントで定義されているかのように扱われます。また、シンボルは別のコンポーネントの同じ名前の定義で上書きできます。
- `hidden` - 他のコンポーネントは、直接、シンボルを参照できません。ただし、アドレスを間接的に他のコンポーネントに渡すことができます。
- `internal` - 直接的にも間接的にも定義コンポーネントの外部では参照できません。
- `protected` - 他のコンポーネントはシンボルを参照できますが、別のコンポーネントにある同じ名前の定義でシンボル定義を上書きすることはできません。

file は、設定するシンボルのリストを含むファイルのパス名です。シンボルは、可視属性を空白 (空白、タブ、改行) で区切られます。

-g

デフォルト: オフ

デバッガで使用されるシンボリック・デバッグ情報と行番号をオブジェクト・ファイルに生成します。

-help

すべてのコマンドライン・オプションについて簡単な情報を表示します。

-inline_debug_info

デフォルト: オフ

インライン・コード用に拡張ソース位置情報を出力します。この情報があると、命令のソース位置を報告するときに精度が高くなります。また、関数呼び出しのトレースバックに役立つ拡張デバッグ情報も出力します。インテル・デバッガ (idb) は、この情報を使用してインライン関数の呼び出しフレームのシミュレーションを表示します。

このオプションをデバッグ用に使用するには、-g も指定する必要があります。

-[no]logo

デフォルト: -logo (起動バナーを表示)

起動バナーを表示します。

このオプションは、コマンドラインの任意の場所で指定できます。

起動バナーは次の情報を表示します。

- ID: コンパイラ固有の識別番号
- x.y.z: コンパイラのバージョン
- years: ソフトウェアの著作権が保護される年

-nofor_main

デフォルト: オフ

メイン・プログラムが Fortran で記述されていない場合に指定します。例えば、メイン・プログラムが C で記述されていて、インテル® Fortran サブプログラムを呼び出す場合、ifort コマンドでプログラムをコンパイルする際に -nofor_main を指定します。-nofor_main を指定することで、for_main.o をプログラムにリンクしないようにします。これは、リンク時のスイッチです。

-nofor_main を省略した場合、メイン・プログラムは Fortran プログラムでなければなりません。

-noinclude

デフォルト: オフ

INCLUDE 文で指定されたファイルを /usr/include で検索しないようにします。

このオプションを `-Idir` オプションとともに指定できます。このオプションは、`cpp` (1) 動作に影響を与えません。また、Fortran 95 および Fortran 90 の USE 文とも関係ありません。

`-openmp_profile`

デフォルト: オフ

OpenMP* アプリケーションの解析を有効にします。このオプションを使用するには、あらかじめインテル® スレッド化ツールの 1 つである、スレッド・プロファイラをインストールしておく必要があります。スレッド化ツールがインストールされていない場合、このオプションは効果がありません。評価版の入手方法など、スレッド・プロファイラに関する詳細は、次のインテル® ソフトウェア開発製品 Web サイトを参照してください:
<http://www.intel.co.jp/jp/developer/software/products/>

`-[no]pad`

デフォルト: `-nopad`

変数と配列のメモリ・レイアウトの変更を有効にします。

`-pad` オプションは、構造体と派生型に適用されたときには、事実上、`-align` と同じです。ただし、共通ブロック、派生型、シーケンス形式にも適用されるため、`-pad` の方が有効範囲が広がります。

`-prec_div` (IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ)

デフォルト: オフ

浮動小数点の除算の精度を向上させます。速度に多少影響を与えます。

`-rcd` (IA-32 システムのみ)

デフォルト: オフ

浮動小数点から整数への変換で丸めモードの変更を有効にします。浮動小数点から整数への変換が速くなります。

`-size_lp64` (Itanium ベース・システムのみ)

デフォルト: オフ

long 型と pointer 型のビットサイズを 64 ビットに想定します。

-nostartfiles

デフォルト: オフ

リンク時に使用する標準起動ファイルを指定します。

-startfiles オプションはありません。

-syntax_only

デフォルト: オフ

代替構文: -y および -syntax

ソースファイルの構文のみを検証するように要求します。コードは生成されません。

-T file

デフォルト: オフ

リンクに *file* からリンクコマンドを読み取るように指示します。

-Tf file

デフォルト: オフ

filename を Fortran ソースファイルとしてコンパイルするように指定します。このオプションは、拡張子が非標準 (つまり、.F、.FOR または .F90 のいずれかではないもの) の Fortran ファイルがある場合に使用します。

-tcheck

デフォルト: オフ

マルチスレッド・アプリケーションの解析を有効にします。このオプションを使用するには、あらかじめインテル・スレッド化ツールの 1 つである、インテル® スレッド・チェッカーをインストールしておく必要があります。スレッド化ツールがインストールされていない場合、このオプションは効果がありません。評価版の入手方法など、インテル・スレッド・チェッカーに関する詳細は、次のインテル・ソフトウェア開発製品 Web サイトを参照してください: <http://www.intel.co.jp/jp/developer/software/products/>

-u

デフォルト: オフ

代替構文: `-implicitnone`

IMPLICIT NONE がデフォルトで設定されるように指定します。-warn
[no]declarations も参照してください。

-v

デフォルト: オフ

ドライバ・ツール・コマンドを表示し、ツールを実行します。-dryrun オプションも参照してください。

-V

デフォルト: なし

コンパイラのバージョン情報を表示します。

-what

デフォルト: オフ

Fortran コマンドとコンパイラのバージョン・テキストを出力します。

-Wl,option1[,option2,...]

デフォルト: オフ

オプション (*option1*、*option2*、その他で指定) を処理用にリンカに渡します。

-X

デフォルト: オフ

代替構文: `-nostdinc`

インクルード・ファイルの検索から標準ディレクトリを除外します。コンパイラが `FPATH` 環境変数で指定されたデフォルトのパスを検索しないようにします。

-Xlinker value

デフォルト: オフ

リンクに直接渡される値 (*value*) です。

最適化オプション

最適化オプションでは、アプリケーションの速度およびコードサイズ、そして特定のプロセッサに対しての最適化など、さまざまな最適化をどのように行うのかを指定することができます。

最適化に関する詳細は、「コンパイラ最適化の概要」および『インテル® Fortran ユーザーズ・ガイド Vol II: アプリケーションの最適化』の関連するセクションを参照してください。

「[浮動小数点オプション](#)」も参照してください。

最適化オプションの説明

-arch keyword (IA-32 システムのみ)

デフォルト: `-arch pn4`

コンパイラが命令を生成するアーキテクチャのバージョンを決定します。

以下の `-arch` オプションがあります。

- `-arch pn1`
インテル® Pentium® プロセッサ向けに最適化します。
- `-arch pn2`
インテル Pentium Pro プロセッサ、インテル Pentium II プロセッサ、およびインテル Pentium III プロセッサ向けに最適化します。
- `-arch pn3`
インテル Pentium Pro プロセッサ、インテル Pentium II プロセッサ、およびインテル Pentium III プロセッサ向けに最適化します。`-arch:pn2` オプションと同じです。
- `-arch pn4`
インテル Pentium 4 プロセッサ向けに最適化します。
- `-arch SSE`
ストリーミング SIMD 拡張命令 (SSE) をサポートするインテル Pentium 4 プロセッサ向けに最適化します。

- `-arch SSE2`
ストリーミング SIMD 拡張命令 2 (SSE2) をサポートするインテル Pentium 4 プロセッサ向けに最適化します。

`-assume [no]buffered_io`

デフォルト: `-assume nobuffered_io` (各レコードが書き込まれるたびにバッファはフラッシュされます)

各レコードが書き込まれるたびに、レコードをディスクに書き込む (フラッシュする) か、またはバッファに蓄積するかを指定します。`-assume buffered_io` を指定した場合、レコードはバッファに蓄積されます。

ディスクデバイスの場合、`-assume buffered_io` (または同じ働きをする OPEN 文の `BUFFERED='YES'` 指定子、または `FORT_BUFFERED` ランタイム環境変数) は、Fortran ランタイム・システムによってレコードがディスクに書き込まれるまで、多くのレコード出力文 (WRITE) によって内部バッファが使用されるように要求します。ファイルが直接アクセスで開かれた場合、I/O バッファリングは無視されます。

通常、バッファリング付きの書き込みを使用すると、より大きなデータブロックがディスクに書き込まれ、書き込み回数が減るため、ディスク I/O の効率が向上します。ただし、バッファリング付きの書き込みを指定した場合、システム障害が起こると、ディスクにまだ書き込まれていないレコードが失われる可能性があります。

`FORT_BUFFERED` 環境変数を TRUE に設定しない限り、すべての I/O 操作においてデフォルトの `BUFFERED='NO'` および `-assume nobuffered_io` が使用され、各 WRITE 文 (またはその他のレコード出力文) ごとに Fortran ランタイム・システムは内部バッファを空にします。

OPEN 文の `BUFFERED` 指定子は、指定した論理ユニットに適用されます。これに対して、`-assume [no]buffered_io` オプションおよび `FORT_BUFFERED` 環境変数は、すべての Fortran ユニットの適用されます。

`-auto_ilp32` (Itanium® ベース・システムおよびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ)

デフォルト: オフ

コンパイラは、アプリケーションが 32 ビットのアドレス領域を超えない限り、32 ビットのポインタを使用します。

この最適化は、プログラム全体に対するプロシージャ間の解析を必要とするため、このオプションを使用する際、`-ipo` オプションも使用する必要があります。

32 ビットのアドレス領域 (232) を超えるプログラムにこのオプションを使用した場合、プログラム実行時に予測できない結果を引き起こす原因となる可能性があります。

インテル EM64T システムでは、`-auto_ilp32` は `-xP` または `-axP` が指定されていない限り効果はありません。

`-ax{K|W|N|B|P}` (IA-32 およびインテル EM64T システムのみ)

デフォルト: なし

指定したインテル® プロセッサに対して、機能性が向上する関数を別個に生成可能な箇所を検出するように、コンパイラに指示します。

固有の関数バージョンを生成できることが判明した場合、コンパイラはそれがパフォーマンスの向上につながるかどうかをチェックします。パフォーマンスの向上につながると判明した場合、コンパイラはプロセッサ固有の関数バージョンと汎用バージョンの両方を生成します。汎用バージョンはすべての IA-32 プロセッサ上で実行できます。

プログラムの実行時に、使用されているインテル・プロセッサに応じて、この 2 つのバージョンのどちらを実行するかが選択されます。この方法により、プログラムは従来の IA-32 プロセッサとの互換性を保ちながら、最新のインテル・プロセッサ上でパフォーマンスを大幅に向上することができます。

以下に、指定可能な値および最適化されるプロセッサを示します。

- `-axK` インテル Pentium III プロセッサおよび互換性のあるインテル・プロセッサ
- `-axW` インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ
- `-axN` インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ
メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
- `-axB` インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ
メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
- `-axP` ストリーミング SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサ
メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラ

ー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。

インテル EM64T システムでは、`-axW` および `-axP` オプションのみ有効です。

`-complex_limited_range[-]`

デフォルト: オフ (`-complex_limited_range-`)

複素数のデータ型を使用するいくつかの算術演算で、基本代数展開の使用を有効にします。このオプションを使用すると、複素数算術を多く使用するプログラムにおいてパフォーマンスが向上する場合があります。ただし、指数範囲の極値が正しく計算されない可能性があります。

`-f[no-]alias`

デフォルト: `-falias`

プログラム内のエイリアシングを仮定するよう指定します。

[-f\[no-\]fnalias](#) も参照してください。

`-f[no-]fnalias`

デフォルト: `-ffnalias`

関数内のエイリアシングを仮定するよう指定します。`-fno-fnalias` オプションを指定した場合、関数内のエイリアシングを仮定しませんが、複数の呼び出しにわたるエイリアシングは仮定します。

[-f\[no-\]alias](#) も参照してください。

`-fast`

デフォルト: オフ

ランタイム・パフォーマンスにおけるいくつかの最適化を有効にするショートカット・メソッドを提供します。

Itanium ベース・システムでは、`-fast` オプションを設定すると、パフォーマンスを向上する以下のオプションが設定されます。

- [-O3](#) (最大速度および高レベルの最適化を行います)

- `-ipo` (複数ファイルにわたるプロシージャ間の最適化を有効にします)
- `-static` (共有ライブラリとリンクしないようにします)

IA-32 およびインテル EM64T システムでは、`-fast` オプションは `-xP` も設定します。したがって、`-fast` オプションを設定すると、`-O3 -ipo -static -xP` が設定されます。IA-32 およびインテル EM64T システムで `-xP` を使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。

利用可能な最高のパフォーマンスを得るには、場合によっては、`-xN` などのアーキテクチャ固有のオプションを併用する必要があります。

`-fast` によって設定されたオプションを上書きするには、コマンドラインから `-fast` オプションの後に使用するオプションを指定してください。



注

`-fast` オプションによって設定されたいくつかのオプションは、バージョンによって変更される場合があります。

`-fnsplit[-]` (Itanium ベース・システムのみ)

デフォルト: `-prof_use` が指定されている場合はオン、それ以外の場合はオフ。

`-prof_use` も有効な場合、関数分割を有効にします。(このオプションは、`-prof_use` が有効でない場合、効果はありません)

`-prof_use` を使用する場合、このオプションは自動的に有効になります。

関数分割を無効にするには、`-fnsplit-` を使用してください (ただし、関数のグループ化は無効になりません)。

次の Vol II のトピックも参照してください。

基本的な PGO オプション

プロファイルに基づく最適化の例

`-fp` (IA-32 およびインテル EM64T システムのみ)

デフォルト: オン

`ebp` の汎用レジスタとしての使用を無効にします。

ただし、ほとんどのデバッガは、スタック・フレーム・ポインタとして `ebp` を使用するため、必要な操作をしない限りスタック・バックトレースを生成できません。このオプショ

ンは、フレームポインタを許可し、最適化における ebp レジスタの使用を無効にします。また、デバッガはスタック・バックトレースを生成します。

-gp

デフォルト: オフ

代替構文: -p

gprof ツールを使用して関数のプロファイリングができるようにコンパイルおよびリンクを行います。

-ip

デフォルト: オフ

単一ファイルのプロシージャ間の最適化を有効にします。

関数のインライン展開を実行します。

Vol II: 「-ip と -Qoption 指定子の併用」も参照してください。

-ip_no_inlining

デフォルト: オフ

-ip か -ipo を指定してプロシージャ間の最適化が行われるときに、プロシージャ間のインライン化については無効にします。しかし、他のプロシージャ間の最適化には何の影響も与えません。-ip または -ipo が必要です。

-ip_no_pinning

デフォルト: オフ

部分的なインライン展開を無効にします。-ip または -ipo[n] が必要です。

-ipo[n]

デフォルト: オフ

マルチファイル IPO を有効にします。このオプションを指定すると、別々のファイル内で定義している関数内での呼び出し命令について関数のインライン展開が実行されます。

オプションで、コンパイラが生成するオブジェクト・ファイルの数 n の値を (0 以上の整数で) 指定できます。

n が 0 の場合、オブジェクト・ファイルの予想サイズに基づいて、生成するオブジェクト・ファイルの数をコンパイラが決定します。小規模なアプリケーションでは 1 つのオブジェクト・ファイルを、大規模なアプリケーションでは複数のオブジェクト・ファイルを生成します。

n のデフォルト値は 1 (1 つのオブジェクト・ファイルを生成) です。

関連情報

次の Vol II のトピックも参照してください。

IPO のコンパイル・モデル

xilink を使用したマルチファイル IPO 実行ファイルの作成

-ip と -Qoption 指定子の併用

-ipo_c

デフォルト: オフ

複数のファイルにわたる最適化を行い、マルチファイル・オブジェクト・ファイルを作成します。最後のリンク段階の前に停止して、最適化されたオブジェクト・ファイルを残します。

Vol II: 「IPO 中間出力のキャプチャ」も参照してください。

-ipo_obj

デフォルト: オフ

実際のオブジェクト・ファイルを強制的に生成します。ipo[n] が必要です。-ipo_obj -ipo2 を指定すると、ipo_obj.o および ipo_obj1.o が作成されます。Vol II: 「実際のオブジェクト・ファイルの生成」も参照してください。

-ipo_S

デフォルト: オフ

複数のファイルにわたる最適化を行い、マルチファイル・アセンブリ・ファイルを作成します。-ipo[n] と同じ最適化を実行しますが、最後のリンク段階の前に停止し、最適化されたアセンブリ・ファイルを残します。デフォルトのリスト名は、ipo_out.s です。

Vol II: 「IPO 中間出力のキャプチャ」も参照してください。

-ipo_separate

デフォルト: オフ

ソースファイルごとに 1 つのオブジェクト・ファイルを作成します。このオプションは、`-ipo[n]` で設定されたすべての値よりも優先されます。

-ivdep_parallel (Itanium ベース・システムのみ)

デフォルト: オフ

IVDEP ディレクティブが指定されたループにループ・キャリー・メモリ依存がないことを示します。この手法は、スパース・マトリックス・アプリケーションに役立ちます。

Vol II: 「IVDEP ディレクティブのメモリの依存性」も参照してください。

-nolib_inline

デフォルト: オン

組み込み関数のインライン展開を無効にします。

-On

デフォルト: `-debug` を指定しない限り `-O2`。指定した場合、デフォルトは `-O0`。

アプリケーションの種類におけるコードの最適化を指定します。設定可能な値は以下のとおりです。

- `-O0`
すべての最適化を無効にします。
`-debug` を (キーワードなしで) 指定した場合のデフォルト設定です。
このオプションを指定すると、一部の `-warn` オプションは無視されます。
- `-O1`
IA-32 システムでの代替構文: `-O2` または `-O`
処理速度を最適化します。コードのサイズが増えるだけでそれほど速度が向上しない最適化は無効にします このオプションでは、グローバルな最適化を実行できます。これにはデータフローの解析、コードの移動、ストレングス・レダクションとテストの置換、スプリット・ライフタイムの解析、および命令スケジューリングが含まれます。`-O2` オプションには、`-O1` で実行される最適化も含ま

れています。

IA-32 システムでは、-O1 および -O2 オプションは同じ働きをします。

- -O2
Itanium ベース・システムでの代替構文: -O
コードサイズおよび処理速度を最適化しますが、コードサイズを大きくするだけでさほどの高速化にはつながらない最適化は無効にします。Itanium ベースのコンパイラの -O1 オプションは、コードサイズを小さくするために、ソフトウェアのパイプライン化をオフにします。このオプションは、ソース・プログラム・ユニットにおけるローカルな最適化、共通部分式の認識、およびシフトを使用した整数の乗算と除算の展開を有効にします。
- -O3
処理速度の最適化およびより高度なレベルでの最適化を行います。この最適化オプションには、ループ変換、ソフトウェア・パイプライン処理、およびプリフェッチ (IA-32 のみ) が含まれています。一部のプログラムでは、このオプションを使用してもパフォーマンスが向上しない場合があります。-O3 オプションには、-O2 で実行される最適化も含まれています。このオプションは、処理速度を向上するグローバルな最適化を有効にします (そのため、コードサイズは増加します)。これらの最適化には、次のものが含まれています。
 - 命令スケジューリングを含むループのアンロール
 - コードを反復して、分岐を排除
 - 次元が 2 のべき乗となる配列のサイズをパディングして、キャッシュの使用効率を向上 (Vol II: 「配列の効率的な使用」も参照してください。)
 -O3 を設定すると、-fp も設定されます。

IA-32 システムでは、-O1、-O2、および -O オプションは同じ働きをします。

Itanium ベース・システムでは、-O2 および -O オプションは同じ働きをします。



注

コマンドラインの最後に指定された -On オプションは、その前に指定された他のオプションより優先されます。

-opt_report

デフォルト: オフ

最適化レポートを生成して、stderr に送ります。

Vol II: 「最適化機構レポートの作成」も参照してください。

-opt_report_file file

デフォルト: オフ

最適化レポートを生成し、レポートのファイル名を指定します。このオプションを使用した場合、`-opt_report` を指定する必要はありません。

Vol II: 「最適化機構レポートの作成」も参照してください。

-opt_report_help

デフォルト: オフ

レポートで利用可能な最適化フェーズを表示します。

Vol II: 「最適化機構レポートの作成」も参照してください。

-opt_report_level {min|med|max}

デフォルト: `-opt_report_level min`

最適化レポートの詳細レベルを指定します。

Vol II: 「最適化機構レポートの作成」も参照してください。

-opt_report_phase phase

デフォルト: オフ

レポートを生成する最適化フェーズを指定します。複数の最適化に対して、コマンドライン上で複数回指定することができます。

Vol II: 「最適化機構レポートの作成」も参照してください。

-opt_report_routine [routine]

デフォルト: オフ

名前の一部に部分文字列 (*routine*) を含むすべてのルーチンからレポートを生成します。

この *routine* が指定されていない場合、すべてのルーチンからのレポートが作成されます。

Vol II: 「最適化機構レポートの作成」も参照してください。

-par_threshold[n]

デフォルト: `-par_threshold100`

並列実行が効果的である可能性に基づいてループの自動並列化のしきい値を設定します。*n* には、0 から 100 までの値を設定することができます。

n=0: ループは、計算量に関わらず自動並列化を行います。つまり、常に並列化します。

n=100: ループは並列化実行が有効であることが確実な場合にのみ自動並列化されます。

次の Vol II のトピックも参照してください。

自動並列化のしきい値と診断

自動並列化の概要

自動並列化: 有効、オプション、ディレクティブ、および環境変数

-parallel

デフォルト: オフ

自動パラライザを有効にして、並列で安全に実行できるループのマルチスレッド・コードを生成します。このオプションを使用するには、[-O2](#) または [-O3](#) も指定する必要があります。

次の Vol II のトピックも参照してください。

自動並列化の概要

自動並列化: 有効、オプション、ディレクティブ、および環境変数

-prefetch[-] (IA-32 システムのみ)

デフォルト: `-prefetch (on)`

プリフェッチ挿入による最適化を有効にします。プリフェッチを挿入する目的は、データをキャッシュにロードするタイミングのヒントをプロセッサに知らせることでキャッシュ・ミスが減らすことです。このオプションを使用する場合、[-O3](#) が指定されている必要があります。

プリフェッチ挿入による最適化を無効にするには、`-prefetch-` を使用してください。

-prof_dir dir

デフォルト: 当該プログラムのコンパイル先ディレクトリになります。

プロファイル出力ファイル (.dyn and .dpi) を保存するディレクトリを指定します。既存のディレクトリを指定してください。

次の Vol II のトピックも参照してください。

高度な PGO オプション

IA-32 アーキテクチャのコーディング・ガイドライン

-prof_file file

デフォルト: 拡張子が .dyn および .dpi のソースファイル名

サマリ・ファイルのプロファイリングに使うファイル名を指定します。

次の Vol II のトピックも参照してください。

高度な PGO オプション

IA-32 アーキテクチャのコーディング・ガイドライン

-prof_gen

デフォルト: オフ

各基本ブロックの実行カウントを取得するために、プロファイル用プログラムをインストールメントします。

次の Vol II のトピックも参照してください。

基本的な PGO オプション

プロファイルに基づく最適化の例

-prof_use

デフォルト: オフ

最適化中にプロファイリング情報 (関数分割および関数のグループ化を含む) が使えるようにします。プロファイルによって最適化された実行ファイルを生成し、利用可能なプロファイル出力ファイルをいくつかマージして pgopti.dpi ファイルを 1 つ作成するようにコンパイラに命令するオプションです。

このオプションを使用した場合、[-fnsplit \[-\]](#) オプションは自動的に有効にされます。

このオプションを使用して有効にされた関数のグループ化はオフにできません。

次の Vol II のトピックも参照してください。

基本的な PGO オプション

プロファイルに基づく最適化の例

-scalar_rep[-] (IA-32 システムのみ)

デフォルト: -scalar_rep (on)

ループ変換中に実行されるスカラ置換を有効にします。-O3 が必要です。

-tppn

IA-32 およびインテル EM64T システムのデフォルト値: -tpp7

Itanium ベース・システムのデフォルト値: -tpp2

特定のインテル・プロセッサ向けに最適化します。最適化されたアプリケーションは、その他のプロセッサでも実行することができますが、最適化は以下で示されているプロセッサ向けに行われます。以下に、*n* に指定可能な値を示します。

- 1 Itanium プロセッサ向けに最適化します (Itanium ベース・システムのみ)
- 2 Itanium 2 プロセッサ向けに最適化します (Itanium ベース・システムのみ)
- 5 インテル® Pentium® プロセッサおよび MMX® テクノロジ対応 Pentium プロセッサ向けに最適化します (IA-32 システムのみ)
- 6 インテル Pentium Pro プロセッサ、Pentium II プロセッサ、および Pentium III プロセッサ向けに最適化します (IA-32 システムのみ)
- 7 インテル Pentium 4 プロセッサ、インテル® Xeon™ プロセッサ、インテル Pentium M プロセッサ、およびハイパー・スレッディング (HT) テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 向けに最適化します (IA-32 システムのみ)

インテル EM64T システムでは、-tpp7 オプションのみ利用可能です。

-unroll[n]

デフォルト: -unroll (コンパイラによる判断)

1 回のループを最大何回までアンロールするかを指定します。

設定可能な値は以下のとおりです。

- `-unroll` アンロールを実行するかどうかをコンパイラが判断します。
- `-unroll0` ループのアンロールを無効にします。(注: Itanium ベース・システムでは、この値のみを使用することができます。その他の値は無視されます。)
- `-unrolln` ループのアンロール回数の上限 n を指定します。

`-x[K|W|N|B|P]` (IA-32 およびインテル EM64T システムのみ)

デフォルト: なし

プログラムを特定のインテル・プロセッサ上で動作するようにします。生成されるコードには、他のプロセッサでサポートされない機能の無条件の使用が含まれることがあります。

以下に、指定可能な値および最適化されるプロセッサを示します。

- `-xK` インテル Pentium III プロセッサおよび互換性のあるインテル・プロセッサ
- `-xW` インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ
- `-xN` インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ
メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
- `-xB` インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ
メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
- `-xP` ストリーミング SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサ
メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。

インテル EM64T システムでは、`-xW` および `-xP` オプションのみ有効です。

インテル以外から提供されている x86 プロセッサ上でプログラムを実行する場合、このオプションを指定しないでください。

**警告**

このオプションを使用してコンパイルされたプログラムを、指定された命令セットをサポートしていないプロセッサ上で実行すると、不正命令例外か、他の予期しない動作が発生する場合があります。特に、`-xN`、`-xB`、または `-xP` オプションでコンパイルされたプログラムは、サポートされていないプロセッサ上ではランタイム・エラーが発生します。

出力ファイル・オプション

出力ファイル・オプションを使用することで、コンパイルで出力されるファイルの名前と格納するディレクトリを指定できます。

出力ファイル・オプションの説明

-c

デフォルト: オフ

オブジェクト (.o ファイル) にコンパイルされますが、リンクは行われません。

-fcode-asm

デフォルト: オフ

オプションのコード注釈を含むアセンブリ・ファイルを生成します。このオプションを使用するには、`-S` オプションも指定する必要があります。

-fsource-asm

デフォルト: オフ

オプションのソース注釈を含むアセンブリ・ファイルを生成します。このオプションを使用するには、`-S` オプションも指定する必要があります。

-f[no]verbose-asm

デフォルト: `-S` が指定された場合にオン

コンパイラ注釈を含むアセンブリ・ファイルを作成します。コンパイラ注釈には、オプションやバージョン情報などが含まれています。このオプションを使用するには、`-s` オプションも指定する必要があります。

-module path

モジュール・ファイル (ファイル拡張子は `.mod`) を格納するディレクトリ (*path*) を指定します。詳細は、「[インクルード・ファイルと .mod ファイルの検索](#)」を参照してください。

-ofilename

デフォルト:オフ

出力ファイルの名前を指定します。

`-c` が指定されている場合は、`-o` はオブジェクト名を指定します。

`-s` が指定されている場合は、`-o` はアセンブリ・リスト・ファイル名を指定します。

`-c` および `-s` の両方とも指定されていない場合は、`-o` は実行ファイル名を指定します。

-Qinstall dir

デフォルト:オフ

コンパイラのインストール先のルートとして *dir* を指定します。

-Qlocation,tool,path

デフォルト:オフ

プリプロセッサ、コンパイラ、アセンブラ、およびリンカなどの支援ツールが配置されるディレクトリを指定します。

構文および詳細は、「[-Qlocation を使ってツールの代替場所を指定する](#)」を参照してください。

-Qoption,tool,options

デフォルト:オフ

オプションをツール（プリプロセッサ、コンパイラ、アセンブラ、およびリンカ）に渡します。

構文および詳細は、「[-Qoption を使用してオプションをツールに渡す](#)」を参照してください。

-S

デフォルト: オフ

アセンブリ・ファイル（.s）にコンパイルされますが、リンクは行われません。

-use_asm

デフォルト: オフ

アセンブラを使用してオブジェクトを生成します。

プリプロセッサ・オプション

プリプロセッサ・オプションを使用することで、オプションであるコンパイルの第 1 フェーズで、コンパイラがどのようにファイルを前処理するか、またどのソース・ディレクトリを検索するかを指定できます。

次のトピックも参照してください。

[前処理フェーズ](#)

[定義済みプリプロセッサ・シンボル](#)

プリプロセッサ・オプションの説明

-assume [no]source_include

デフォルト: `-assume source_include`

USE 文で指定したモジュール・ファイルまたは INCLUDE 文で指定したソースファイルを検索するディレクトリを指定します。

設定可能な値は以下のとおりです:

- `-assume source_include` ソースファイル・ディレクトリを検索します
- `-assume nosource_include` 現在のディレクトリを検索します

このオプションは、**-fpp オプション**なしで使用できます。

-Dname[=value]

デフォルト: オフ

条件付きコンパイル・ディレクティブまたは Fortran プリプロセッサの fpp で使用する定義を指定します。定義が 1 つ以上ある場合は、各定義に対して別個の **-D** オプションを使用します。

value には、文字値または整数値を入力します。*value* を入力しなかった場合、*name* は 1 に設定されます。

-Dname の使用例は、「**プリプロセッサ・シンボルの定義**」を参照してください。



注

-DD オプション (**-d_lines** の代替構文) と競合するため、*name* には **D** を使用しないでください。ただし、**-DD=1** のような **-Dname=n** 構文は使用できます。

-[no]fpp

デフォルト: **-nofpp**

代替構文: **-[no]cpp**。fpp 0 は **-nofpp** と同じです。また、fpp *n* (*n* は 1 以上の数値) は、**-fpp** と同じです。

コンパイル前に FPP プリプロセッサ (fpp) を起動し、プリプロセッサ・ディレクティブを有効にします。

-cpp は **-fpp** と同じです (**-fpp** は C プリプロセッサの代わりに fpp を実行します)。

-ldir

デフォルト: Include パス

モジュール・ファイル (USE 文) およびインクルード・ファイル (INCLUDE 文) を検索するディレクトリを、インクルード・パスに追加します。追加するディレクトリが 2 つ以上ある場合は、ディレクトリをセミコロンで区切ります。

現在のディレクトリの代わりに、ソースファイルが置かれているディレクトリを最初に検索させるには、**-assume source_include** を指定します。

すべての USE 文および INCLUDE 文で指定された、ファイル名がデバイス名またはディレクトリ名で始まっていないファイルは、次の順序で検索されます:

1. 最初のソースファイルを含むディレクトリ (デフォルトの `-assume source_include` が指定されている場合)。
2. コンパイルが行われている現在のデフォルト・ディレクトリ。
3. `-I \textit{dir}` オプションで指定されたディレクトリ (指定されている場合)。複数のディレクトリを検索するときは、指定したリストで左から右の順に検索されます。
4. コンパイル時に使用される環境変数 `FPATH` で指定されたディレクトリ。

`-noinclude` も参照してください。

`-preprocess_only`

デフォルト: オフ

代替構文: `-P` および `-F`

Fortran プリプロセッサ (fpp) を実行し、各ソースファイルの結果を、対応する `.i` ファイルまたは `.i90` ファイルに格納します。`.i` ファイルまたは `.i90` ファイルには、行番号 (#) が含まれていません。このオプションでは、ソースファイルはコンパイルされません。

このオプションを使用するには、`-fpp` を指定する必要があります。

`-U name`

デフォルト: オフ

`name` で指定したシンボルの現在有効な定義を無効にします。

`-Wp,option1[,option2,...]`

デフォルト: オフ

オプション (`option1`、`option2` など指定したオプション) をプリプロセッサに渡します。

`-fpp` が fpp を起動する点を除き、このオプションは `-fpp` と同じです。

ランタイム・オプション

ランタイム・オプションでは、エラー・チェックがコンパイル時ではなく、ランタイム時に行われるように設定できます。

ランタイム・オプションの説明

-[no]check [all] または -[no]check [none]

デフォルト: カスタム (オプションは個別に指定されます)。

代替構文: `-C` は `-check all` と同じです。

すべてのランタイム障害をチェックするかまたは無視するかを指定します。`-check all` または `-check none` が指定されている場合、以下に示すランタイム・チェック・オプションは使用できません。

`-nocheck` は `-check none` と同じです。

`-check` は `-check all` と同じです。

-check [no]arg_temp_created

デフォルト: カスタム (オプションは個別に指定されます)。

デフォルト: `-check noarg_temp_created`

ルーチンが呼び出される前に、実引数が一時的な記憶域にコピーされる場合、ランタイム情報を含むメッセージを発行します。

-check [no]args

Default: `-check noargs`

Generates code to check the correspondence between the actual and dummy arguments of a procedure.

-check [no]bounds

デフォルト: `-check nobounds`

代替構文: `-CB`

配列の添字および部分文字列を処理する式に対して、ランタイム・チェックを行うコードを生成します。

デフォルト (`-check nobounds`) では、範囲チェックは実行されません。

配列に対しては、個々の次元で境界がチェックされます。最後の次元が `*` として指定されている配列が仮引数として使用されている場合、その配列に対しては、配列境界のチェックが行われません。また、上位次元と下位次元の両方が `1` の場合にも、配列境界のチェックは行われません。

プログラムのデバッグが終了したら、このオプションを省略することで、実行プログラムのサイズを減らし、実行時の性能を若干向上させることができます。

`-check [no]format`

デフォルト: `-vms` が指定されていない場合は `-check noformat`、`-vms` が指定されている場合は、`-check format` がデフォルトになります。

出力用に書式化される項目のデータ型が `FORMAT` 記述子と一致しないとき、ランタイム・エラー・メッセージを発行します。

`-check [no]output_conversion`

デフォルト: `-vms` が指定されていない場合は `-check nooutput_conversion`、`-vms` が指定されている場合は、`-check output_conversion` がデフォルトになります。

書式の切り捨てが発生した（つまり、数値が大きすぎて、指定された書式フィールドの長さにすべての有効な桁数が収まらない）ときに、ランタイム・エラー・メッセージを発行します。

`-check [no]pointer`

Default: `-check nopointer`

Alternate syntax: `-CA`

Generates code to check that referenced pointers are not null and allocatable arrays are allocated. Requests a run-time error message when referenced pointers are null and allocatable arrays are not allocated.

`-check [no]shape`

Default: `-check noshape`

Alternate syntax: `-CS`

Generates code to check that shapes of arrays conform.

`-[no]traceback`

デフォルト: `-notraceback`

ランタイム時に重大なエラーが発生したとき、ソースファイルのトレースバック情報を表示できるように、オブジェクト・ファイル内に補足情報を生成します。

`-traceback` を指定すると、重大なエラーが発生したときに表示されるコールスタックの 16 進アドレス（プログラム・カウンタのトレース）に、ソースファイル、ルーチン名、および行番号の関係情報が提供されます。`-traceback` が指定されていない場合、この情報は表示されません。また、上級ユーザは、MAP ファイルと、重大なエラーが発生したときに表示されるスタックの 16 進アドレスをもとに、エラーの原因を特定することができます。

`-traceback` を指定すると、実行プログラムのサイズが増えますが、ランタイム時の実行速度には影響がありません。

`-traceback` オプションは、`-debug` オプションとは独立して機能します。

idb を使用したデバッグ

idb を使用したデバッグの概要

See these topics:

[デバッグの概要](#)

[デバッグのためのプログラムの準備](#)

[デバッグコマンドの使用とブレークポイントの設定](#)

[デバッグコマンドのまとめ](#)

[SQUARES サンプル・プログラムのデバッグ](#)

デバッガにおける変数の表示

デバッガコマンドにおける表現

言語が混在したプログラムのデバッグ

信号を生成するプログラムのデバッグ

アライメントされていないデータの検索

デバッグの概要

インテル® デバッガ (idb) は、ソースレベルでデバッグを行うシンボリック・デバッガであり、次のような機能を持ちます:

- プログラムの各ソース行の実行を制御できます。
- 特定のソース行に、またはさまざまな条件を基に、プログラムを停止するポイント (ブレークポイント) を設定できます。
- プログラム内の変数の値を変更できます。
- シンボリック名を使用して、プログラム内の特定の位置を参照できます。デバッガは、インテル® Fortran 言語についての知識を利用し、適切なスコーピング規則と値の評価方法および表示方法を判断します。
- 変数の値を出力し、トレースポイント (トレース) を設定して、変数の値が変更された時に通知します。(トレースポイントは、ウォッチポイントとも呼ばれています)。
- コアファイルの検査、コールスタックの検査、およびレジスタの表示など、その他の機能を実行します。

idb デバッガには、次の 2 つのモードがあります:

- dbx (デフォルト・モード)
- gdb (オプション・モード)

本書のすべての例は、dbx モードで示します。



注

idb に関する詳細は、idb の man ページまたはオンラインの『*Intel Debugger (IDB) Manual*』(英語) を参照してください。

デバッグ・オプション

デバッガを使用するには、`ifort` コマンドで `-g` コマンドライン・オプションを指定します。デバッグには、トレースバック情報およびシンボルテーブル情報の両方が必要です。`-g` を指定した場合、コンパイラは、シンボリック・デバッグに必要なシンボルテーブル情報とトレースバック情報を提供します。`(-notraceback` オプションを指定すると、トレースバック情報を取得しません)。

これらのオプションをプログラム開発のさまざまな工程で使用する例を次に挙げます：

プログラム開発の初期段階では、`-g` オプションを使用して、非最適化コード（最適化レベル `-O0`）を作成します。また、このオプションは、後の段階で、報告された問題をデバッグするためにも使用できます。

トレースバック情報およびシンボルテーブル情報によって、より大きなオブジェクト・ファイルが生成されます。プログラム開発の後期段階では、`-g0` または `-g1` を使用してオブジェクト・ファイルのサイズを最小化します。その結果、最適化コードにより、プログラムの実行に必要なメモリの量も最小化されます。`(-g0` オプションは、トレースバック情報を削除します)。

プログラムのデバッグ終了後に、コンパイルとリンクを再度行うことで、最適化された実行プログラムを作成できます。また、`strip` コマンドを使用することで、トレースバック情報およびシンボルテーブル情報を削除できます。`(strip(1)` を参照)

デバッグを向上する設定についての詳細は、`-debug keyword` オプションを参照してください。



注

インテル® プラットフォームでは、最適化されたコードのデバッグが完全にはサポートされていません。

デバッグのためのプログラムの準備

特定のオプションとともに `ifort` コマンドを使用して、デバッグ用の実行プログラムを作成します。デバッガを起動するには、デバッガのシェルコマンドとプログラムの名前を入力します。

次のコマンド行は、実行プログラムを作成（コンパイルおよびリンク）し、デバッガのインターフェイスを呼び出します：


```
ifort -g -o squares squares.f90
idb squares
Linux Application Debugger for xx-bit applications, Version
x.x, Build xxxx
object file name: squares
reading symbolic information ... done
(idb)
```

この例では、`ifort` コマンドは以下の処理を行います:

- プログラム `squares.f90` のコンパイルとリンクを実行します。
- シンボリック・デバッグと最適化なし (`-g`) に必要なシンボルテーブル情報を要求します。
- 実行ファイルの名前を、`a.out` (`-o squares`) から `squares` にします。

`idb` シェルコマンドは、実行プログラム `squares` を指定して、デバッガを実行します。

デバッガのプロンプト (`idb`) に、デバッガのコマンドを入力できます。

オンライン・マニュアルの『Intel Debugger (IDB) Manual』(英語) も参照してください。

デバッグを向上する設定についての詳細は、`-debug keyword` オプションを参照してください。

デバッガコマンドの使用とブレークポイントの設定

プログラムの重要なポイントで何が起きているのかを明らかにするには、これらのポイントで実行を停止し、プログラム変数の内容を調べ、正しい値が入っているかを確かめる必要があります。デバッガがプログラムを停止するポイントを「ブレークポイント」と呼びます。

ブレークポイントを設定するには、次の `stop` または `stopi` コマンドのいずれかの形式を使用します。

サンプル・プログラムを使用して、次のデバッガコマンドでは、4 行目にブレークポイントを設定し、プログラムを実行し、プログラムを続行し、ブレークポイントを削除してから、プログラムを再実行し、シェルに戻ります:

```
(idb) stop at 4
[#1: stop at "squares.f90":4 ]
(idb) run
[1] stopped at [squares:4 0x120001880]
> 4 OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
```

```
(idb) cont
Process has exited with status 0
(idb) delete 1
(idb) rerun
Process has exited with status 0
(idb) quit
%
```

この例では次の事柄が行われます:

- `stop at 4` コマンドで 4 行目にブレークポイントを設定します。サブプログラム (`calc` など) の先頭にブレークポイントを設定するには、`stop in` コマンド (`stop in calc` など) を使用します。
- `run` コマンドでプログラムの実行を開始し、最初のブレークポイントで停止します。ここでは、プログラムがアクティブになっているので、`print` コマンドを使用して変数の値を表示したり、関連するコマンドを実行できます。
- `cont` コマンドでプログラムの実行を再開 (続行) させます。`cont` コマンド以外にも、`step`、`next`、`run`、または `rerun` コマンドを使用してプログラムの実行を再開できます。
- `delete 1` コマンドは、以前に設定したブレークポイント (イベント番号 1 を持つブレークポイント) を削除します。例えば、`rerun` コマンドを使用する前に、以前に設定したブレークポイントを削除する場合に役立ちます。
- `rerun` コマンドでプログラムを再び実行します。設定したブレークポイントが削除されたので、プログラムは終了まで実行します。
- `quit` コマンドでデバグを終了し、シェルに戻ります。

その他のデバグコマンド

その他のデバグコマンドには次のようなものがあります:

- デバグコマンドに関するヘルプを表示するには、`help` コマンドを入力します。
- 以前に入力したデバグコマンドを表示するには、`history` コマンドを入力します。
- 特定の位置の内容を見るには、`print` または `dump` コマンドを使用します。
- シェルコマンドを実行するには、`sh` コマンド (後ろに実行するシェルコマンドを付けて) を使用します。例えば、関数名を思い出せないときに、次の `grep` シェルコマンドを使用して、「FUNCTION」という文字列を含む行を表示することができます。次の例では、関数名 (SUBSORT) を `grep` コマンドで調べ、`stop in` コマンドで使用しています:

```
(idb) sh grep FUNCTION data.for
INTEGER*4 FUNCTION SUBSORT (A,B)
(idb) stop in subsort
(idb)
```

「デバッガコマンドの概要」も参照してください。

デバッガコマンドの概要

次の表は、`idb` で利用できるデバッグコマンドで頻繁に使用されるもののリストです。これらのコマンドの多くは短縮することができます（例えば、`cont` の代わりに `c`、`step` の代わりに `s` を入力するなど）。`alias` コマンドを使用して、これらの短縮されたコマンドの完全な名前や、独自のエイリアスを作成することもできます。

下記の表は、最もよく使用されるデバッグコマンドの例です。詳細は、オンライン・マニュアルの『Intel® Debugger (IDB) Manual』(英語) を参照してください。

コマンド例	説明
<code>catch</code>	現在デバッガがキャッチするように設定されているすべての信号を表示します。 <code>ignore</code> も参照してください。
<code>catch fpe</code>	デバッガに <code>fpe</code> 信号（または指定された信号）をキャッチするように指示します。これにより、指定された信号がインテル® Fortran ランタイム・ライブラリ (RTL) に到達するのを防ぎます。
<code>catch unaligned</code>	デバッガにアライメントの合っていない信号をキャッチするように指示します。
<code>cont</code>	デバッグ中のプログラムの実行を再開（続行）します。 <code>continue</code> という <code>idb</code> コマンドはないことに注意してください。
<code>delete 2</code>	イベント番号 2 で指定されているブレークポイントやトレースポイントを削除します。 <code>status</code> も参照してください。
<code>delete all</code>	すべてのブレークポイントとトレースポイントを削除します。
<code>help</code>	デバッガのヘルプテキストを表示します。
<code>history 5</code>	最近使用した 5 個のデバッグコマンドを表示します。
<code>ignore</code>	デバッガが現在無視するように設定されている信号を表示します。無視される信号はインテル Fortran RTL に、直接渡すことができます。 <code>catch</code> も参照してください。
<code>ignore fpe</code>	デバッガに <code>fpe</code> 信号（または指定された信号）を無視するように指示します。これにより、指定された信号がインテル Fortran RTL に直接渡され、メッセージが表示されます。
<code>ignore unaligned</code>	デバッガにアライメントの合っていない信号を無視するように指示します（デフォルト）。
<code>kill</code>	プログラムのプロセスを終了させます。デバッガは実行されたままの状態、ブレークポイントおよびトレースポイントは、プログラムが再実行されるまでそのままの状態を保ちます。
<code>list</code>	ソース・プログラム行を表示します。行の範囲をリストするに

	は、 <code>list 1,9</code> などのように、開始行番号、カンマ、終了行番号を追加します。
<code>print k</code>	指定された変数の値を表示します (K など)
<code>printregs</code>	すべてのレジスタとそれらの内容を表示します。
<code>next</code>	1 ソース文ごとに進めますが、コールするサブプログラムの中には入りません。step と比較してください。
<code>quit</code>	デバッグ・セッションを終了します。
<code>run</code>	デバッグするプログラムを実行します。プログラム引数およびリダイレクトを指定できます。
<code>rerun</code>	デバッグ中のプログラムを再実行します。プログラム引数およびリダイレクトを指定できます。
<code>return [routine- name]</code>	関数の実行を関数が呼び出し元に戻るまで続行します。 step コマンドの使用時に、調査の必要がないサブプログラムに入ってしまった場合は、return コマンドを使用して、呼び出し元に戻るまで現在の関数の実行を続行します。return コマンドでルーチンの名前を含めた場合、関数はそのルーチンに制御が戻るまで実行されます。 routine-name はルーチンの名前で、通常 PROGRAM 文、SUBROUTINE 文、または FUNCTION 文で命名されます。PROGRAM 文がない場合、デバッガは main\$ のプリフィックスの後ろにファイル名が付いたメイン・プログラムを参照します。
<code>sh more progout.f90</code>	シェルコマンドの more を実行し、progout.f90 ファイルを表示してデバッグ環境に戻ります。
<code>show thread</code>	デバッガに対し、既知のスレッドをすべてリスト表示します。
<code>status</code>	ブレークポイントとトレースポイントにイベント番号を付けて表示します。delete も参照してください。
<code>step</code>	1 ソース文ごとに進めます。サブプログラム・コールの内部にも入りません。インテル Fortran I/O 文、組込みプロシージャ、ライブラリ・ルーチン、他のサブプログラムについては、step ではなく、next コマンドを使用して、サブプログラム・コールを飛び越してください。next と比較してください。また、return も参照してください。
<code>stop in foo</code>	foo ルーチンの始めで実行を停止します (ブレークポイント)。
<code>stop at 100</code>	現在のソース・ファイルの 100 行目で実行を停止します (ブレークポイント)。
<code>stopi at xxxxxxx</code>	現在のソース・ファイルのアドレス xxxxxxxx で実行を停止します。

<code>thread [n]</code>	現在のスレッドのコンテキストを表示または設定します。
<code>watch location</code>	デバッガ (またはユーザ・プログラム) が指定したメモリ位置にアクセスした時にメッセージを表示します。例: <code>watch 0x140000170</code>
<code>watch variable m</code>	デバッガ (またはユーザ・プログラム) が <i>m</i> に指定した変数にアクセスした時にメッセージを表示します。
<code>whatis symbol</code>	指定されたシンボルのデータ型を表示します。
<code>when at 9 {command}</code>	単一のコマンドまたは複数のコマンドを実行します。 指定された行 (9 行目など) に到達したときに単一の <i>command</i> または複数の <i>command</i> が実行されます。例えば、 <code>when at 9 {print k}</code> は、プログラムがソースコードの 9 行目を実行するときに変数 <i>k</i> の値を出力します。
<code>when in name {command}</code>	単一のコマンドまたは複数のコマンドを実行します。 <i>name</i> に指定したプロシージャに到達したときに、単一の <i>command</i> または複数の <i>command</i> が実行されます。例えば、 <code>when in calc_ave {print k}</code> は、 <i>calc_ave</i> という名前のプロシージャの実行を開始するときに変数 <i>k</i> の値を出力します。
<code>where</code>	コールスタックを表示します。
<code>where thread all</code>	すべてのスレッドのスタクトレースを表示します。

デバッガには、その他の特殊な目的のコマンドもサポートされています。次に例を示します。

- 実行時間が非常に長いプログラムに対し、`attach` コマンドと `detach` コマンドを使用できます。
- `listobj` コマンドは、共有ライブラリに依存したプログラムのデバッグに役立ちます。`listobj` コマンドはデバッガで現在認識されている実行イメージおよび共有ライブラリの名前を表示します。

SQUARES サンプル・プログラムのデバッグ

次の例では、デバッグが必要な SQUARES という名前のプログラムを説明します。このプログラムはコンパイルされ、コンパイラまたはリンカのいずれからも診断メッセージを出力せずにリンクされたものです。ただし、このプログラムの算術式には、論理エラーが含まれています。

コンパイラによって割り当てられた行番号は、この例にも追加されており、説明文が参照しているソース行を識別できるようになっています。

```

PROGRAM SQUARES
  INTEGER INARR(10), OUTARR(10), I, K
!Read the input array from the data file.
  OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
  READ(8,*,END=5) N, (INARR(I), I=1,N)
5   CLOSE (UNIT=8)

!Square all nonzero elements and store in OUTARR.
  K = 0
  DO I = 1, N
    IF (INARR(I) .NE.0) THEN
      K = K + 1      !add this line
      OUTARR(K) = INARR(I)**2
    ENDIF
  END DO

!Print the squared output values.  Then stop.
  PRINT 20, N
20  FORMAT (' Total number of elements read is',I4)
  PRINT 30, K
  0  FORMAT (' Number of nonzero elements is',I4)
  DO, I=1,K
    PRINT 40, I, OUTARR(K)
40  FORMAT(' Element', I4, 'Has value',I6)
  END DO
END PROGRAM SQUARES

```

この SQUARES プログラムは次の関数を実行します。

1. データファイルから一連の整数番号を読み取り、これらの番号を配列 INARR に保存します。datafile.dat ファイルには、整数値 4、3、2、5、および 2 を含む 1 つのレコードが含まれています。最初の番号は、この数値の後に続くデータ項目の数を示します。
2. 各非ゼロ整数値の二乗の値を別の配列 OUTARR にコピーするループを実行します。
3. 元のシーケンスに含まれている非ゼロ要素の数および各要素の二乗の値を出力します。

この例では、(-check bounds コマンドライン・オプションを設定して) プログラムが配列の上下限を確認せずに実行されていることを仮定します。配列の上下限を確認して実行した場合、ランタイム・エラー・メッセージが表示されます。

SQUARES を実行すると、データファイルに含まれた非ゼロ要素の数にかかわらず、次の出力を表示します。

squares

```
Number of nonzero elements is    0
```

OUTARR の現在のインデックスを示す変数 K が 9 行目から 13 行目までのループでインクリメントされなかったため、論理エラーが発生します。K = K + 1 文を 11 行目の前に追加する必要があります。

次の例では、デバッグ・セッションの開始方法および文字セル・インターフェイスを使用して、上記のサンプル・プログラムからエラーを検出する方法を説明します。コールアウトの右に示されている番号は、後述のプログラム解説の番号です。

```
ifort -g -o squares squares.f90    (1)
idb squares                          (2)
Linux Application Debugger for xx-bit applications, Version
x.x, Build xxxx
object file name: squares
reading symbolic information ... done
(idb) list 1,9                       (3)
1    PROGRAM SQUARES
2    INTEGER INARR(20), OUTARR(20)
3 C   !Read the input array from the data file.
> 4    OPEN(UNIT=8, FILE='datafile.dat',
STATUS='OLD')
5    READ(8,*,END=5) N, (INARR(I), I=1,N)
6    5 CLOSE (UNIT=8)
7 C   !Square all nonzero elements and store in OUTARR.
8    K = 0
9    DO 10 I = 1, N
(idb) stop at 8                      (4)
[#1: stop at "squares.f90":8]
(idb) run                            (5)
[1] stopped at ["squares.f90":4 0x120001a88]
> 8    K = 0
(idb) step                           (6)
stopped at [squares:9 0x120001a90]
9    DO 10 I = 1, N
(idb) print n, k                     (7)
4 0
(idb) step                           (8)
stopped at [squares:10 0x120001ab0]]
10   IF(INARR(I) .NE.0) THEN
(idb) s
stopped at [squares:11 0x1200011acc]
11   OUTARR(K) = INARR(I)**2
(idb) print i, k                     (9)
1 0
(idb) assign k = 1                   (10)
(idb) watch variable k               (11)
[#2: watch variable (write) k 0x1400002c0 to 0x1400002c3 ]
(idb) cont                           (12)
Number of nonzero elements is    1
Element    1 has value          4
Process has exited with status 0
```

```

(idb) quit (13)
% vi squares.f90 (14)
.
.
.
10:      IF(INARR(I) .NE.0) THEN
11:          K = K + 1
12:          OUTARR(K) = INARR(I)**2
13:      ENDIF
.
.
.
% ifort -g -o squares squares.f90 (15)
% idb squares
Welcome to the idb Debugger Version x.x-xx
Reading symbolic information ...done
(idb) when at 12 {print k} (16)
[#1: when at "squares.f90":12 { print K} ]
(idb) run (17)
[1] when [squares:12 0x120001ae0]
1
[1] when [squares:12 0x120001ae0]
2
[1] when [squares:12 0x120001ae0]
3
[1] when [squares:12 0x120001ae0]
4
Number of nonzero elements is 4
Element 1 has value 9
Element 2 has value 4
Element 3 has value 25
Element 4 has value 4
Process has exited with status 0
(idb) quit (18)
%
```

1. コマンドラインの `-g` オプションは、SQUARES に関するシンボル情報を、デバッガのオブジェクト・ファイルに出力します。また、このオプションはコンパイラによって処理される多くの最適化を無効にすることで、実行コードがプログラムのソースコードと一致するようにします。
2. シェルコマンド `idb squares` は、デバッガを実行します。画面にデバッガに関する情報とデバッガ・プロンプト (`idb`) が表示されます。このコマンドでは、`squares` という名前の実行プログラムが指定されています。デバッガ・プロンプトが表示された後は、デバッガコマンドを入力することができます。`idb squares` コマンドを実行すると、メイン・プログラム・ユニット (この例では、プログラム SQUARES となります) を開始する前で停止します。
3. `list 1,9` コマンドは、1 行目から 9 行目までを出力します。
4. `stop at 8` コマンドは、8 行目にブレークポイント (1) を設定します。

5. `run` コマンドは、プログラムの実行を開始します。プログラムは、8 行目に設定された最初のブレークポイントで停止するため、プログラムの実行が完了する前に、変数 `N` および `K` を確認することができます。
6. `step` コマンドは、プログラムの実行を 9 行目に移動します。`step` コマンドは、実行コードと関係のないソース行を無視します。またデフォルトでは、デバッガは実行を停止するソース行を識別します。サブプログラムへ移動しないようにするには、`step` コマンドの代わりに `next` コマンドを使用します。
7. `print n, k` コマンドは、変数 `N` および `K` の現在の値を表示します。これらの値は、この時点では正しい値です。
8. `step` コマンドを 2 回実行することで、プログラムの実行をループ (9 行目から 11 行目) まで移動します。ここでは、`INARR` に含まれたすべての非ゼロ要素の値をそれぞれ二乗して、`OUTARR` にコピーします。一部のコマンドは、省略することができます。この例では、`step` コマンドを `s` コマンドとして省略しています。
9. `print i, k` コマンドは、変数 `I` および `K` の現在の値を表示します。予測された変数 `I` の値は 1 です。しかし、変数 `K` の値には、予測された値である 1 の代わりに 0 が含まれています。このエラーを修正するには、11 行目のコードを実行する前に、`K` の値をインクリメントする必要があります。
10. `assign` コマンドは、`K` の値に 1 を割り当てます。
11. `watch variable k` コマンドは、変数 `K` の値が変化するたびに起動されるウォッチポイントを設定します。元のプログラムでは、変数 `K` の値が変更されないため (これは、プログラムのエラーです)、このウォッチポイントは起動されません。
12. このパッチをテストするには、`cont` コマンド (`continue` コマンドの省略コマンドです) を使用して、現在の場所からプログラムの実行を再開します。プログラムの出力は、最初の配列要素のみが正常に動作していること表示します。プログラムのエラーによって `K` の値が変化しないため、ウォッチポイント (`watch variable k` コマンド) が発生しません。プログラムの実行が完了したことを示す `idb` メッセージ "Process has exited with status 0" が表示されます。
13. `quit` コマンドは制御をシェルに戻すため、ソースファイルを修正して、プログラムの再コンパイルおよび再リンクを行うことができます。
14. `vi` シェルコマンドは、テキストエディタを実行します。テキストエディタで、ソースファイルの 10 行目の後に `K = K + 1` 文を追加してください。(コンパイラによって割り当てられた行番号は、この例にも追加されています。)
15. 修正したプログラムをコンパイルおよびリンクします。`idb squares` シェルコマンドにより、修正したプログラムに対してデバッガを開始し、問題が修正されたことを確認することができます。
16. `when at 12 {print k}` コマンドは、ループの各反復における変数 `K` の値を通知します。
17. `run` コマンドは、実行を開始します。
18. 表示された変数 `K` の値で、プログラムが正常に動作していることを確認します。

19. quit コマンドは、デバッグ・セッションを終了し、制御をシェルに戻します。

デバッガにおける変数の表示

変数を参照するには、大文字または小文字のいずれかを使用します。次に例を示します。

```
(idb) print J
(idb) print j
```

コマンド名は、大文字で入力することができます。

```
(idb) print J
```

-names as_is コマンドライン・オプションを使用してプログラムをコンパイルし、大文字・小文字を区別した名前を検証する必要がある場合、\$lang 環境変数を大文字・小文字を区別する言語名に設定することで、idb が大文字・小文字の区別を制御することができます。

モジュール変数

モジュール内で定義された変数を参照するには、一重引用符 (')、モジュール名、もう 1 つの一重引用符 (') を変数名の前に追加してください。例えば、モジュール modfile (MODULE MODFILE 文) で定義された変数 J は、次のコマンドを入力してその値を表示することができます。

```
(idb)      list 5,9
          5      USE MODFILE
          6      INTEGER*4 J
          7      CHARACTER*1 CHR
          8      J = 2**8
(idb)      print 'MODFILE'J
256
```

共通ブロック変数

print または whatis などのデバッガコマンドを使用することで、Fortran 共通ブロック内の変数の値を表示することができます。

共通ブロック全体を表示するには、共通ブロック名を使用します。

共通ブロックの特定の変数を表示するには、変数名を使用します。次に例を示します。

```
(idb)      list 1,11
```

```

1      PROGRAM EXAMPLE
2
3      INTEGER*4 INT4
4      CHARACTER*1 CHR
5      COMMON /COM STRA/ INT4, CHR
6
7      CHR = 'L'
8
9      END
(idb)    print COM STRA
COMMON
          INT4 = 0
          CHR = "L"
(idb)
(idb)    print CHR
"L"

```

共通ブロック内のデータ項目の名前が、共通ブロックの名前と同じ場合は、データ項目がアクセスされます。

派生型変数

Fortran 95/90 の派生型 (TYPE 文) の変数は、`print` または `whatis` などの `idb` コマンドでは、Fortran 95/90 構文形式を使用して表します。

派生型構造体では、派生型変数名、パーセント記号 (%)、およびメンバ名を使用します。次に例を示します。

```

(idb)    list 3,11
3      TYPE X
4          INTEGER A(5)
5      END TYPE X
6
7      TYPE (X) Z
8
9      Z%A = 1
10
11     PRINT *, Z%A
(idb)    print Z%A
(1) 1
(2) 1
(3) 1
(4) 1
(5) 1
(idb)

```

オブジェクト全体を表示するには、`print` コマンドでオブジェクト名を使用します。次に例を示します。

```

(idb)    print Z

```

レコード変数

レコード構造体に含まれたフィールドの値を表示するには、変数名をレコード名、区切り文字（ピリオド（.）またはパーセント記号（%））、そしてフィールド名の順に入力します。

レコード構造体に含まれたすべてのフィールドを表示するには、上記の例で（REC.CHR または REC%CHR ではなく）REC などのレコード構造体の名前を入力します。

ポインタ変数

インテル® Fortran は 2 種類のポインタをサポートします。

- Fortran 95/90 ポインタ（標準準拠）
- 整数ポインタ（Fortran 95/90 標準の拡張）

Fortran 95/90 ポインタ

Fortran 95/90 ポインタは、print コマンドで対応するターゲット・データを表示します。

コールアウトの右に示されている番号は、後述のプログラム解説の番号です。

```
ifort -g point.f90
idb ./a.out
Linux Application Debugger for xx-bit applications, Version
x.x, Build xxxx
object file name: ./a.out
Reading symbolic information ...done
(idb) stop in ptr
[#1: stop in ptr ]
(idb) list 1:13
      1      program ptr
      2
      3      integer, target :: x(3)
      4      integer, pointer :: xp(:)
      5
      6      x = (/ 1, 2, 3/)
      7      xp => x
      8
      9      print *, "x = ", x
     10      print *, "xp = ", xp
     11
     12      end
(idb) run
[1] stopped at [ptr:6 0x120001838]
      6      x = (/ 1, 2, 3/)
(idb) whatis x
```

```

int x(1:3)
(idb) whatis xp                                     (1)
int xp(:)
(idb) s
stopped at [ptr:7 0x120001880]
      7      xp => x
(idb) s
stopped at [ptr:9 0x120001954]
      9      print *, "x = ", x
(idb) s
x =              1              2              3
stopped at [ptr:10 0x1200019c8]
(idb) s
xp =              1              2              3
stopped at [point:12 0x120001ad8]
      12      end
(idb) s
xp =              1              2              3
(idb) whatis xp                                     (2)
int xp(1:3)
(idb) print xp
(1) 1
(2) 2
(3) 3
(idb) quit
%
```

1. 最初の `whatis xp` コマンドでは、`xp` は変数 `x` を指すように割り当てられていない汎用ポインタです。
2. `xp` は、変数 `x` を指すように割り当てられているため、2 回目の `whatis xp` コマンドで、`xp` は変数 `x` と同じサイズ、形状、および値を取得します。

整数ポインタ

Fortran 95/90 ポインタと同様に、[整数ポインタ](#) (Cray* 形式ポインタとも呼ばれます) は、`print` コマンドを使用して、対応するソース形式でターゲット・データを表示します。

```

(idb) stop at 14
[#1: stop at "dfpoint.f90":14 ]
(idb) run
[1] stopped at [dfpoint:14 0x1200017e4]
(idb) list 1,14
      1      program dfpoint
      2
      3      real i(5)
      4      pointer (p,i)
      5
      6      n = 5
      7
      8      p = malloc(sizeof(i(1))*n)
```

```

      9
     10      do j = 1,5
     11          i(j) = 10*j
     12      end do
     13
>     14      end
(idb) what is p
float (1:5) pointer p
(idb) print p
0x140003060 = (1) 10
(2) 20
(3) 30
(4) 40
(5) 50
(idb) quit
%
```

配列変数

配列変数では、Fortran 95/90 のソース文と同様に、括弧の間に添字を追加します。次に例を示します。

```
(idb) assign arrayc(1)=1
```

配列のすべての要素は、その名前を使用して出力することができます。次に例を示します。

```
(idb) print arrayc
(1) 1
(2) 0
(3) 0
(idb)
```

大きい配列のすべての要素を表示することは避けてください。その代わりに、特定の配列要素や配列セクションを表示してください。次に、配列要素 `arrayc(2)` を出力する例を示します。

```
(idb) print arrayc(2)
(2) 0
```

配列セクション

配列セクションは配列の一部であり、セクション自体も配列です。配列セクションは、開始要素、終了要素、およびストライド要素の 3 つの部分から構成されるトリプレット表記の添字を使用することができます。

次の配列宣言について考えてみます。

```
INTEGER, DIMENSION(0:99)      :: arr
INTEGER, DIMENSION(0:4,0:4)   :: FiveByFive
```

例えば $\text{FiveByFive}(4,4) = 44$ 、 $\text{arr}(43) = 43$ のように、各配列の各位置にインデックスの値が入るように初期化されているとします。次に、デバッガで許可された配列式の例を示します。

```
(idb) print arr(2)
2
(idb) print arr(0:9:2)
(0) = 0
(2) = 2
(4) = 4
(6) = 6
(8) = 8
(idb) print FiveByFive(:,3)
(0,3) = 3
(1,3) = 13
(2,3) = 23
(3,3) = 33
(4,3) = 43
(idb)
```

配列セクションに対して許可された操作は、`whatis` と `print` のみに限られます。

配列への代入

配列要素への代入は、`idb` でサポートされています。

配列全体または配列セレクションへの値の代入に関する情報は、『*Intel® Debugger (IDB) Manual*』(英語) の Fortran の章を参照してください。

複素変数

`idb` は、式で `COMPLEX`、`COMPLEX*8`、`COMPLEX*16`、`COMPLEX*32` 変数および定数をサポートします。

次の Fortran プログラムについて考えてみます。

```
PROGRAM complextest
  COMPLEX*8 C8 / (2.0,8.0) /
  COMPLEX*16 C16 / (1.23,-4.56) /
  REAL*4      R4 /2.0/
  REAL*8      R8 /2.0/
  REAL*16     R16 /2.0/

  TYPE *, "C8=", C8
  TYPE *, "C16=", C16
```

```
END PROGRAM
```

idb は、基本算術演算子の他に、COMPLEX 変数および定数の表示と代入をサポートします。次に例を示します。

```
Welcome to the idb Debugger Version x.x-xx
-----
object file name: complex
Reading symbolic information ...done
(idb) stop in complextest
[#1: stop in complextest ]
(idb) run
[1] stopped at [complextest:15 0x1200017b4]
      15      TYPE *, "C8=", C8
(idb) whatis c8
complex c8
(idb) whatis c16
double complex c16
(idb) print c8
(2, 8)
(idb) print c16
(1.23, -4.56)
(idb) assign c16=(-2.3E+10,4.5e-2)
(idb) print c16
(-230000000512, 0.045000000178813934)
(idb)
```

データ型

次の表は、インテル Fortran データ型と対応する組込みデバッガ名を示します。

Fortran 95/90 データ型の宣言	対応するデバッガ名
CHARACTER	character
INTEGER, INTEGER(KIND= <i>n</i>)	integer, integer*n
LOGICAL, LOGICAL (KIND= <i>n</i>)	logical, logical*n
REAL, REAL(KIND=4)	real
DOUBLE PRECISION, REAL(KIND=8)	real*8
REAL(KIND=16)	real*16
COMPLEX, COMPLEX(KIND=4)	complex
DOUBLE COMPLEX, COMPLEX(KIND=8)	double complex
COMPLEX(KIND=16), COMPLEX*32	long double complex

デバグコマンドにおける表現

デバグコマンドにおける表現は、演算子および式に対して Fortran 95/90 のソース言語構文を使用します。

デバグコマンド表現は中括弧 ({}) で囲みます。例えば、`print k` は次のように中括弧 ({}) で囲んで示します。

```
(idb) when at 12 {print k}
```

Fortran 演算子

インテル® Fortran 演算子は以下の演算子を含みます。

- 関係演算子: より小さい (.LT. または <) および 等しい (.EQ. または ==) など。
- 論理演算子: 論理積 (.AND.) および 論理和 (.OR.) など。
- 算術演算子: 加算 (+)、減算 (–)、乗算 (*)、除算 (/) を含む。

演算子の一覧については、『*Intel® Fortran Language Reference*』(英語) マニュアルを参照してください。

プロシージャ

idb デバグは、Fortran 95/90 のソース言語構文を使用して、ユーザ定義された個別のプロシージャの呼び出しをサポートしています。また、デバグは、現在スカラー数に限られているいくつかの Fortran 組込みプロシージャの呼び出しもサポートしています。

関連情報

『*Intel® Fortran Language Reference*』(英語) マニュアル

『*Intel® Debugger (IDB)*』(英語) オンライン・マニュアル

言語が混在したプログラムのデバグ

idb デバグを使用することによって、言語が混在したプログラムをデバグできます。異なる言語で書かれたサブプログラム間のプログラム・フロー制御は透過的に行われます。

デバッガは、実行ファイルに埋め込まれた情報に基づいて、現在のサブプログラムまたはコード・セグメントの言語を自動的に識別します。例えば、プログラムの実行が、Fortran で書かれたサブプログラムで停止した場合、現在の言語は Fortran になります。また、デバッガが C で書かれた関数で停止した場合、現在の言語は C になります。デバッガは現在の言語を使用して、式の評価に使う有効な式構文とセマンティクスを判断します。

デバッガは `$lang` 環境変数に、現在のサブプログラムやコード・セグメントの言語を設定します。`$lang` 環境変数を手動で設定すると、デバッガは、指定された言語の規則とセマンティクスに従って、コマンドで入力される式を解釈します。例えば次のように、`$lang` の現在の設定をチェックし、設定を変更することができます：

```
(idb) print $lang
"C++"
(idb) set $lang = "Fortran"
```

`$lang` 環境変数に「Fortran」が設定されている場合、名前の大文字・小文字は区別されません。`-names as_is` オプションでプログラムをコンパイルした場合に、名前の大文字・小文字を区別するには、`$lang` 環境変数に C などの別の言語を指定して、その変数を表示し、その後に `$lang` 環境変数に「Fortran」を設定します。

信号を生成するプログラムのデバッグ

プログラムがランタイムで信号（例外）に遭遇した場合、そのプログラムのデバッグを簡単にするには、信号の原因をデバッグする前に、以下のコマンドライン・オプションを使用して再コンパイルおよび再リンクする必要があります：

- `-fopen` オプションを使用して、信号の処理を制御します。
- 他のデバッグ作業と同様に、`-g` オプションを使用して、十分なシンボルテーブル情報を生成し、非最適化コードをデバッグします。

要求があれば、`idb` はインテル® Fortran ランタイム・ライブラリ (RTL) よりも先に信号をキャッチし処理します。`idb` コマンドの `catch` および `ignore` を使用して、`idb` によって信号をキャッチするか、または無視するかを制御できます：

- `idb` が信号をキャッチすると、`idb` メッセージが表示され、そのステートメント行で実行が停止します。この場合、RTL が提供するエラー処理ルーチンは呼び出されません。この時点で、変数を調査し、プログラムのどこで信号が発生したかを特定することができます。
- `idb` が信号を無視すると、その信号が RTL に渡されます。これにより、コンパイル時に指定した方法で、ランタイム信号メッセージの処理および表示が可能です。

信号（特にプログラム続行を許容する信号）を生成するプログラムのデバッグ時に、適切なランタイム・エラー・メッセージを取得するには、プログラムの実行前に `ignore` コマンドを使用する必要があります。例えば、次のコマンドを使用して、デバッガに、浮動小数点信号を無視させ、それらを RTL に渡すように指示します。

```
(idb) ignore fpe
```

信号を発生させるプログラムの位置を見つける必要がある場合は、`where` コマンドを使用することを推奨します。

アライメントされていないデータの検索

アライメントされていないデータは、プログラムの実行速度を低下させる場合があります。アライメントされていないデータの原因を判断し、必要に応じてソースコードを修正して、プログラムを再コンパイルおよび再リンクします。

プログラムがランタイムでアライメントされていないデータに遭遇した場合、そのプログラムのデバッグを簡単にするには、`-fopen` オプションを使用して再コンパイルおよび再リンクし、例外の処理方法を制御します。

`idb` を使用してアライメントされていないデータの原因を判断するには、次の手順に従います：

1. アライメントされていないデータがあるプログラムを指定し、デバッガを実行します（次の例では、`testprog` を指定します）：`idb testprog`
2. プログラムを実行する前に `catch unaligned` コマンドを入力します。
(idb) **catch unaligned**
3. プログラムを実行します：
(idb) **run**
Unaligned access pid=28413 <testprog> va=140000154
pc=3ff80805d60
ra=1200017e8 type=stl
Thread received signal BUS
stopped at [oops:13 0x120001834]
13 end
4. `list` コマンドを入力し、12 行目のソースコードを表示します：
(idb) **list 12**
12 i4 = 1
> 13 end
5. `where` コマンドを入力し、アライメントされていない位置を見つけます：
(idb) **where**
6. `up`、`list`、および `down` などの他の適切なデバッガコマンドを使用して、アライメントされていないデータの原因だけを追求します。

7. アライメントされていないデータが報告される他の部分に対しても、これらの手順を繰り返します。rerun コマンドを使用して、デバッグを終了する代わりに、シェル・プロンプトからプログラムを再実行します。
8. アライメントされていないデータの原因を修正した後、プログラムを再度コンパイルリンクします。

データと I/O

データ表現の概要

次のトピックを参照してください。

[組み込みデータ型](#)

[整数データの表現の概要](#)

[論理データの表現](#)

[ネイティブ IEEE 浮動小数点の表現の概要](#)

[文字表現](#)

[Hollerith 表現](#)

組み込みデータ型

インテル® Fortran は、数値データがネイティブ・リトル・エンディアン順になっていると想定します。これは、最下位の最も右側のゼロ・ビット（ビット 0）またはバイトが、最上位の最も左側のビット（またはバイト）よりも低いアドレスを持つ順序です。非ネイティブ・ビッグ・エンディアン形式と VAX* 浮動小数点形式の使用方法については、「[書式なしデータの変換](#)」を参照してください。

図中の :A というシンボルは、ビット 0 を含むバイトのアドレス、すなわち表現されているデータ要素の開始アドレスを示しています。

以下の表に、インテル Fortran が使用している組み込みデータ型、必要な記憶域、および有効な範囲を示します。例えば、INTEGER(4) の宣言は INTEGER(KIND=4) および INTEGER*4 と同じです。

データ型	記憶域	説明
BYTE INTEGER(1)	1 バイト (8 ビット)	BYTE 宣言は、INTEGER(1) と等価な符号付き整数データ型です。
INTEGER	INTEGER(2)、 INTEGER(4)、 および INTEGER(8) を参照。	符号付き整数で、INTEGER(2)、INTEGER(4)、または INTEGER(8)。大きさは <code>-integer_size nn</code> コンパイラ・オプションによって制御されます。デフォルト設定は、 <code>-integer_size 32 (INTEGER(KIND=4))</code> です。
INTEGER(1)	1 バイト (8 ビット)	-128 ~ 127 の範囲の符号付き整数値。
INTEGER(2)	2 バイト (16 ビット)	-32,768 ~ 32,767 の範囲の符号付き整数値。
INTEGER(4)	4 バイト (32 ビット)	-2,147,483,648 ~ 2,147,483,647 の範囲の符号付き整数値。
INTEGER(8)	8 バイト (64 ビット)	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807 の範囲の符号付き整数値。
REAL(4) REAL	4 バイト (32 ビット)	1.17549435E-38 ~ 3.40282347E38 の範囲の IEEE S 浮動小数点形式での単精度実数浮動小数点値。1.17549429E-38 ~ 1.40129846E-45 の範囲の値は正規化されていない値 (サブノーマル) です。
REAL(8) DOUBLE PRECISION	8 バイト (64 ビット)	2.2250738585072013D-308 ~ 1.7976931348623158D308 の範囲の、IEEE T 浮動小数点形式での倍精度実数浮動小数点値。2.2250738585072008D-308 ~ 4.94065645841246544D-324 の範囲の値は正規化されていない値 (サブノーマル) です。
REAL(16) EXTENDED PRECISION	16 バイト (128 ビット)	6.4751751194380251109244389582276465524996Q-4966 ~ 1.189731495357231765085759326628007016196477Q4932 の範囲の、IEEE X 浮動小数点形式での拡張精度実数浮動小数点値。
COMPLEX(4) COMPLEX	8 バイト (64 ビット)	IEEE S 浮動小数点形式の実部と虚部の組として表現される、単精度複素数浮動小数点値。実部と虚部の範囲は 1.17549435E-38 ~ 3.40282347E38 です。1.17549429E-38 ~ 1.40129846E-45 の範囲の値は正規化されていない値 (サブノーマル) です。
REAL(16) EXTENDED PRECISION	16 バイト (128 ビット)	6.4751751194380251109244389582276465524996Q-4966 ~ 1.189731495357231765085759326628007016196477Q4932 の範囲の、IEEE X 浮動小数点形式での拡張精度実数浮動小数点値。
COMPLEX(8) DOUBLE COMPLEX	16 バイト (128 ビット)	IEEE T 浮動小数点形式の実部と虚部の組として表現される、倍精度複素数浮動小数点値。実部と虚部の範囲は 2.2250738585072013D-308 ~ 1.7976931348623158D308 です。2.2250738585072008D-308 ~ 4.94065645841246544D-324 の範囲の値は正規化されていない値 (サブノーマル) です。

COMPLEX(16) EXTENDED PRECISION	32 バイト (256 ビット)	IEEE X 浮動小数点形式の実部と虚部の組として表現される、拡張精度複素数浮動小数点値。実部と虚部の範囲は 6.4751751194380251109244389582276465524996Q-4966 ~ 1.189731495357231765085759326628007016196477Q4932 です。
LOGICAL	LOGICAL(2)、 LOGICAL(4)、 および LOGICAL(8) を参照。	論理値で、LOGICAL(2)、LOGICAL(4)、または LOGICAL(8)。大きさは <code>-integer_size nn</code> コンパイラ・オプションによって制御されます。デフォルト設定は、 <code>-integer_size 32</code> (LOGICAL(4)) です。
LOGICAL(1)	1 バイト (8 ビット)	論理値 <code>.TRUE.</code> または <code>.FALSE.</code> 。
LOGICAL(2)	2 バイト (16 ビット)	論理値 <code>.TRUE.</code> または <code>.FALSE.</code> 。
LOGICAL(4)	4 バイト (32 ビット)	論理値 <code>.TRUE.</code> または <code>.FALSE.</code> 。
LOGICAL(8)	8 バイト (64 ビット)	論理値 <code>.TRUE.</code> または <code>.FALSE.</code> 。
CHARACTER	1 文字につき 1 バイト (8 ビット)	文字コード規約によって表現される文字データ。文字宣言は <code>CHARACTER(LEN=<i>n</i>)</code> または <code>CHARACTER*<i>n</i></code> の形式で指定することができます。 <i>n</i> はバイト数ですが、(*) として指定すれば、文字列長を渡す書式を指定することができます。
HOLLERITH	Hollerith 文 字 1 つにつき 1 バイト (8 ビット)	Hollerith 定数。

『*Intel® Fortran Language Reference*』(英語) で説明しているように、2 進 (ビット) 定数を定義することができます。

次に示すセクションでは、組込みデータ型についてさらに詳しく説明しています。

- [整数データの表現](#)
- [論理データの表現](#)
- [ネイティブ IEEE* 浮動小数点の表現](#)
- [文字表現](#)
- [Hollerith 表現](#)

整数データの表現

整数データの表現の概要

整数データ長は 1、2、4、または 8 バイトです。

INTEGER データ宣言に使用されるデフォルトのデータサイズは、`-integer_size 16` オプションか、`-integer_size 64` オプションが指定されていない限り、`INTEGER(4)` (`INTEGER(KIND=4)` と同等) となります。

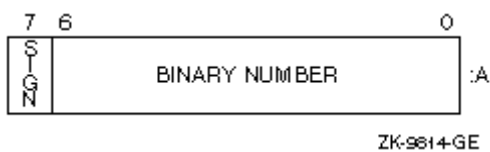
整数データの符号は、符号ビットを正の数値の場合には 0 (ゼロ)、負の数値の場合には 1 に設定することによって指定されます。

次に示すセクションでは整数データについて説明しています。

- [INTEGER\(KIND=1\) の表現](#)
- [INTEGER\(KIND=2\) の表現](#)
- [INTEGER\(KIND=4\) の表現](#)
- [INTEGER\(KIND=8\) の表現](#)

INTEGER(KIND=1) の表現

`INTEGER(1)` 値は -128 ~ 127 までの範囲で、次に示すように 1 つのバイトに格納されます。

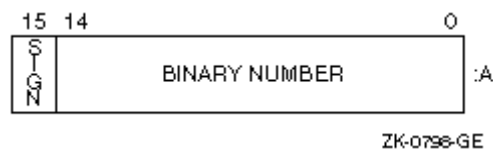


整数は 2 の補数表現で格納されます。次に例を示します。

+22 = 16 (hex)
-7 = F9 (hex)

INTEGER(KIND=2) の表現

`INTEGER(2)` 値は -32,768 ~ 32,767 の範囲で、次に示すように 2 つの連続したバイトに格納されます。



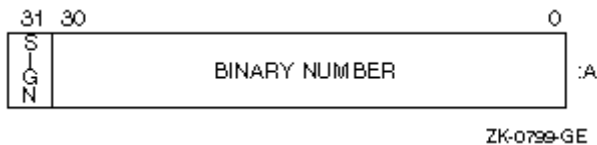
整数は 2 の補数表現で格納されます。以下に例を示します。

+22 = 0016 (hex)

-7 = FFF9 (hex)

INTEGER(KIND=4) の表現

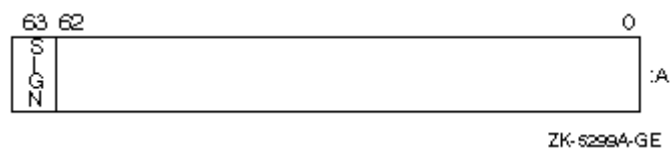
INTEGER(4) 値は -2,147,483,648 ~ 2,147,483,647 の範囲で、次に示すように 4 つの連続したバイトに格納されます。



整数は 2 の補数表現で格納されます。

INTEGER(KIND=8) の表現

INTEGER(8) 値は -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807 の範囲で、次に示すように 8 つの連続したバイトに格納されます。



整数は 2 の補数表現で格納されます。

論理データの表現

論理データ長は、1 バイト、2 バイト、4 バイト、または 8 バイトです。

LOGICAL データ宣言で使用するデフォルトのデータサイズは、`-integer_size 16` または `-integer_size 64` オプションが指定されていない限り、LOGICAL(4) (LOGICAL(KIND=4) と同じ) になります。

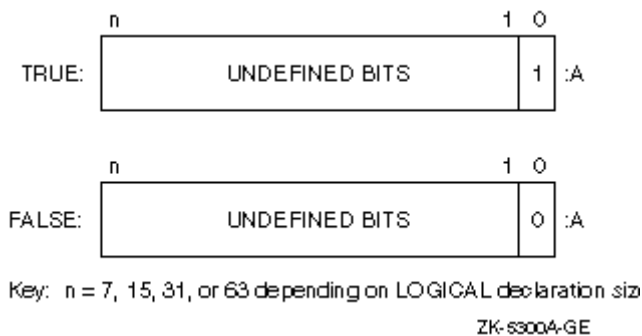
Itanium® ベース・システム向けにパフォーマンスを向上させるには、LOGICAL(2) または LOGICAL(1) の代わりに、LOGICAL(4) (または LOGICAL(8)) を使用してください。

LOGICAL(KIND=1) の値は、1 バイトで格納されます。LOGICAL(1) データは、論理値 .TRUE. と .FALSE. の他にも、-128~127 の範囲の値を持つことができます。また、論理変数は整数データとして解釈することもできます。

LOGICAL(1) に加えて、論理値は任意のバイト境界から始まる 2 つの (LOGICAL(2))、4 つの (LOGICAL(4))、または 8 つの (LOGICAL(8)) 連続したバイトにも格納することができます。

-fpscomp nological コンパイラ・オプションが設定されている場合 (デフォルト設定)、論理値は下位ビットによって TRUE と FALSE のどちらなのかが決定されます。Microsoft* Fortran PowerStation の論理値では、0 (ゼロ) が FALSE で、ゼロ以外の値が TRUE になるように -fpscomp logical を指定します。

LOGICAL(1)、LOGICAL(2)、LOGICAL(4)、および LOGICAL(8) のデータ表現 (-fpscomp nological オプションが設定されている場合) を以下に示します:



ネイティブ IEEE* 浮動小数点の表現

ネイティブ IEEE* 浮動小数点の表現の概要

REAL(4) (IEEE* S_floating)、REAL(8) (IEEE T_floating)、および REAL(16) (IEEE-style X_floating) 形式は、標準リトル・エンディアン IEEE 2 進法浮動小数点表記で格納されます。(IEEE 2 進法浮動小数点の表記に関する詳細は、「IEEE 標準 754」を参照してください)。COMPLEX() 書式は、REAL 値のペアを使用して、データの実部と虚部を示します。

すべての浮動小数点の形式は、分数を符号付き絶対値表記で表します。また、2 進法の基数点は最上位ビットの右に位置します。分数は正規化されているものと仮定されるので、最上位ビットは格納されません（これは「隠れたビットの正規化」と呼ばれます）。このビットは、指数が 0 でない限り 1 と仮定されます。指数が 0 と等しい場合、表現されている値は正規化されていない（サブノーマル）か、プラスまたはマイナスのゼロです。

組込み型 REAL の種別には、4（単精度）、8（倍精度）、および 16（拡張精度）があります。例えば、REAL(KIND=4) では、単精度浮動小数点になります。また、組込み型 COMPLEX の種別にも同様に、4（単精度）、8（倍精度）、および 16（拡張精度）があります。

変数の種別を取得するには、KIND 組込み関数を使用します。REAL*4 などのサイズ指定子も使用できますが、これは Fortran 95 標準を拡張したもののなので注意してください。

特定のコンパイラ・オプションを省略した場合、REAL および COMPLEX のデータ宣言で使用されるデフォルトのサイズは次のとおりです：

- 種別パラメータ（またはサイズ指定子）を指定していない REAL データ宣言では、デフォルト・サイズが REAL (KIND=4) (REAL*4 と同じ) になります。
- 種別パラメータ（またはサイズ指定子）を指定していない COMPLEX データ宣言では、デフォルト・サイズが COMPLEX (KIND=4) (COMPLEX*8 と同じ) になります。

種別パラメータを指定していないすべての REAL または COMPLEX 宣言のサイズを制御するには、`-real_size 64` または `-real_size 128` オプションを使用します。デフォルトでは、`-real_size 32` が使用されます。

種別パラメータを使用して、REAL または COMPLEX 宣言の長さを明示的に宣言するか、DOUBLE PRECISION または DOUBLE COMPLEX を指定します。すべての DOUBLE PRECISION および DOUBLE COMPLEX 宣言のサイズを制御するには、`-double_size 128` オプションを使用します。デフォルトでは、`-double_size 64` が使用されます。

次に示すセクションでは、浮動小数点データについて説明しています：

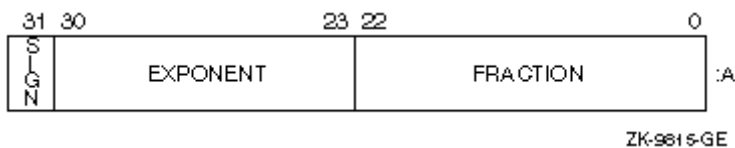
- [REAL\(KIND=4\)（単精度）の表現](#)
- [REAL\(KIND=8\)（DOUBLE PRECISION）の表現](#)
- [REAL\(KIND=16\)（拡張精度）の表現](#)
- [COMPLEX\(KIND=4\)（単精度）の表現](#)
- [COMPLEX\(KIND=8\)（倍精度）の表現](#)
- [COMPLEX\(KIND=16\)の表現](#)

ネイティブ IEEE リトル・エンディアン・データ以外の浮動小数点データの読み書きについては、「[書式なし数値データの変換](#)」を参照してください。

また、「[fordef.for ファイルとその使用](#)」も参照してください。

REAL(KIND=4) (REAL) の表現

REAL(4) (REAL(KIND=4) と同じ) データは、IEEE S 浮動小数点形式で格納された 4 つの連続したバイトを占有します。ビットには下図で示すように、右から 0～31 のラベルが付けられます。

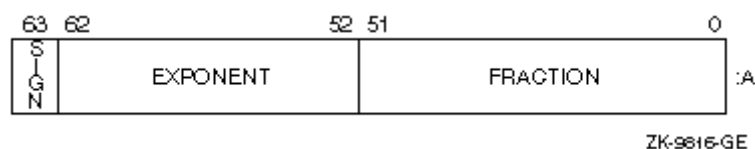


REAL(4) データの形式は符号付き絶対値表現で、ビット 31 が符号ビット(正の数値の場合は 0、負の数値の場合は 1)、ビット 30:23 が超過 127 表記の 2 進指数、ビット 22:0 は表現されていない冗長な最上位小数ビットを含む正規化された 24 ビットの小数です。

データ値は、ほぼ $1.17549435\text{E}-38$ (正規化) から $3.40282347\text{E}38$ の範囲になります。IEEE の正規化されていない (サブノーマル) 制限は $1.40129846\text{E}-45$ です。精度は約 2^{23} 分の 1 で、一般には 10 進で 7 桁です。

REAL(KIND=8) (DOUBLE PRECISION) の表現

REAL(8) (REAL(KIND=8) と同じ) データは、IEEE T 浮動小数点形式で格納された 8 つの連続したバイトを占有します。ビットには下図で示すように、右から 0～63 のラベルが付けられます。

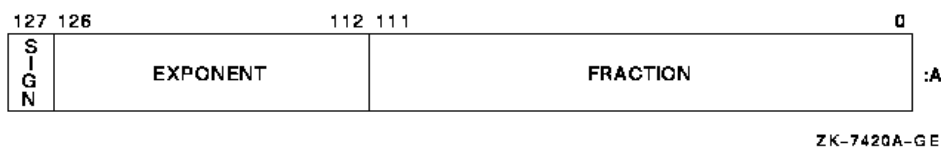


REAL(8)データの形式は符号付き絶対値表現で、ビット 63 が符号ビット(正の数値の場合は 0、負の数値の場合は 1)、ビット 62:52 が超過 1023 表記の 2 進の指数、ビット 51:0 は表現されていない冗長な最上位小数ビットを含む正規化された 53 ビットの小数です。

データ値は、ほぼ 2.2250738585072013D-308 (正規化) から 1.7976931348623158D308 の範囲になります。IEEE の正規化されていない (サブノーマル) 制限は 4.94065645841246544D-324 です。精度は約 2^{52} 分の 1 で、一般には 10 進で 15 桁です。

REAL(KIND=16) (EXTENDED PRECISION) の表現

REAL(16) (REAL(KIND=16) と同じ) データは、IEEE X_floating 形式で格納された 16 の連続したバイトを占有します。ビットには下図で示すように、右から 0~127 のラベルが付けられます。



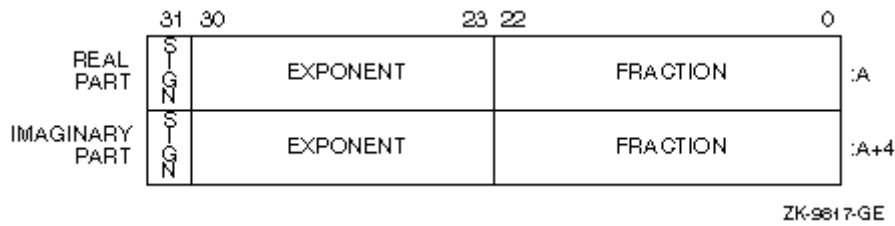
REAL(16) データの形式は符号付き絶対値表現で、ビット 127 が符号ビット (正の数値の場合は 0、負の数値の場合は 1)、ビット 126:112 が超過 16383 表記の 2 進指数、ビット 111:0 は表現されていない冗長な最上位小数ビットを含む正規化された 113 ビットの小数です。

データ値は、ほぼ 6.4751751194380251109244389582276465524996Q-4966 から 1.189731495357231765085759326628007016196477Q4932 の範囲になります。他の浮動小数点形式とは異なり、正規化されていない拡張精度の数字を使用することによる性能低下があっても、それは少しです。これは、正規化されていない REAL (KIND=16) 数字のアクセスが、算術的エラーにならないためです (拡張精度書式は、ソフトウェアでエミュレートされます)。最小の正規化された数字は、3.362103143112093506262677817321753Q-4932 です。

精度は約 2^{112} 分の 1 で、一般には 10 進で 33 桁です。

COMPLEX(KIND=4) (COMPLEX) の表現

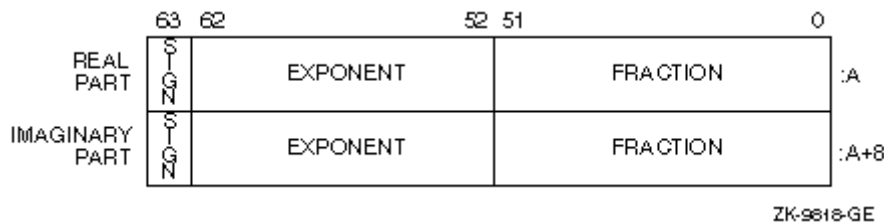
COMPLEX(4) (COMPLEX(KIND=4) および COMPLEX*8 と同じ) データは、IEEE S 浮動小数点形式で格納された REAL(4) 値の組を含んでいる 8 つの連続したバイトです。下位 4 バイトは、複素数の実部を表す REAL(4) データを含んでいます。上位 4 バイトは、複素数の虚部を表す REAL(4) データを含んでいます。この様子を下図に示します。



COMPLEX(4) の数値の実部と虚部のそれぞれが、**REAL(4)** の制限とアンダーフローの特性を持ちます。REAL(4) の数値と同様に、符号ビットの表現は、正の数値では 0(ゼロ)、負の数値では 1 です。

COMPLEX(KIND=8) (DOUBLE COMPLEX) の表現

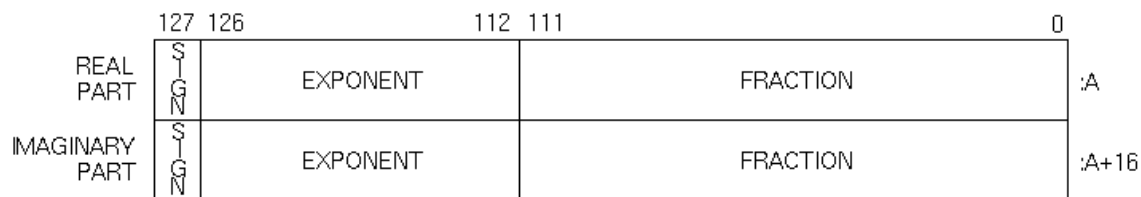
COMPLEX(8) (COMPLEX(KIND=8) および COMPLEX*16 と同じ) のデータは、IEEE T 浮動小数点形式で格納された REAL(8) 値の組を含んでいる 16 個の連続したバイトです。下位 8 バイトは、複素数の実部を表す REAL(8) データを含みます。上位 8 バイトは、複素数の虚部を表す REAL(8) データを含みます。この様子を下図に示します。



COMPLEX(8) の数値の実部と虚部のそれぞれが、**REAL(8)** の制限とアンダーフローの特性を持ちます。REAL(8) の数値と同様に、符号ビットの表現は、正の数値では 0 (ゼロ)、負の数値では 1 です。

COMPLEX(KIND=16) の表現

COMPLEX(16) (COMPLEX(KIND=16) または COMPLEX*32 と同じ) データは、IEEE X 浮動小数点書式で格納された REAL(16) 値の組を含んでいる 32 個の連続したバイトです。下位 16 バイトは、複素数の実部を表す REAL(16) データを含んでいます。上位 16 バイトは、複素数の虚部を表す REAL(8) データを含んでいます。この様子を下図に示します。



LJ-06690

COMPLEX(16) の数値の実部と虚部のそれぞれが、[REAL\(16\)](#) の制限とアンダーフローの特性を持ちます。REAL(16) の数値と同様に、符号ビットの表現は、正の数値では 0 (ゼロ)、負の数値では 1 です。

fordef.for ファイルとその使用

パラメータ・ファイル `/opt/intel_fc_80/include/fordef.for` には、浮動小数点表現のクラスに対応するシンボルおよび `INTEGER*4` の値が含まれています。これらのクラスには、正の正規化されていない数値を示すビットパターンなどの例外も含まれています。

シンボルが含まれたこのファイルと、`FP_CLASS` 組込み関数を使用することで、例外的な数値の識別が可能になります。例えば、これにより、正または負の正規化されていない数値を正のゼロに置換できます。

次は、浮動小数点のビット表現を識別する簡単な例です。

```
include 'fordef.for'

real*4 a
integer*4 class_of_bits
a = 57.0    ! Bit pattern is a finite number
class_of_bits = fp_class(a)
if ( class_of_bits .eq. for_k_fp_pos_norm .or. &
     class_of_bits .eq. for_k_fp_neg_norm ) then
print *, a, ' is a non-zero and non-exceptional value'
else
print *, a, ' is zero or an exceptional value'
end if
end
```

この例では、`/opt/intel_fc_80/include/fordef.for` ファイルのシンボル `for_k_fp_pos_norm` と、`REAL*4` の値 57.0 数とした `FP_CLASS` 組込み関数の戻り値を比べた結果、最初の `print` 文が実行されます。

次の表では、`/opt/intel_fc_80/include/fordef.for` ファイル内のシンボルとその対応する浮動小数点表現について説明します。

fordef.for ファイルのシンボル

シンボル名	浮動小数点のビット表現のクラス
FOR_K_FP_SNaN	シグナル型 NaN
FOR_K_FP_QNaN	クワイエット型 NaN
FOR_K_FP_POS_INF	正の無限大
FOR_K_FP_NEG_INF	負の無限大
FOR_K_FP_POS_NORM	正の正規化された有限数
FOR_K_FP_NEG_NORM	負の正規化された有限数
FOR_K_FP_POS_DENORM	正の正規化されていない数
FOR_K_FP_NEG_DENORM	負の正規化されていない数
FOR_K_FP_POS_ZERO	正のゼロ
FOR_K_FP_NEG_ZERO	負のゼロ

次に、`fordef.for` ファイルと組込み関数 `FP_CLASS` を使ったもう 1 つの例を示します。このプログラムの目的は、例外が発生しても報告を行わず、書式なしファイルから 32 ビットパターンを `REAL*4` に、すばやく読み込み、正規化されていない数値を正のゼロに置換します。

```
include 'fordef.for'
      real*4 a(100)
      integer*4 class_of_bits
!      open an unformatted file as unit 1
!...
      read (1) a
      do i = 1, 100
          class_of_bits = fp_class(a(i))
          if
( class_of_bits .eq. for_k_fp_pos_denorm .or. &
  class_of_bits .eq. for_k_fp_neg_denorm      ) then
              a(i) = 0.0
          end if
      end do
      close (1)
      end
```

このプログラムは、任意の `-fpen` の値でコンパイルできます。組込み関数 `FP_CLASS` は、プログラムが正規化されていない数値で計算を実行しようとする前に、正規化されていない数値を検索して、ゼロに置換します。一方、プログラムが `-fpe0` でコンパイルされていて、ユニット 1 から読み込まれた正規化されていない数値がゼロに置換されない場合、正規化されていない数値を使った最初の計算で浮動小数点例外が発生します。

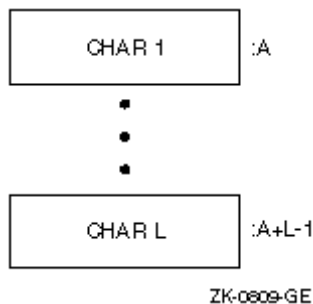
fordef.for ファイルと組み込み関数 FP_CLASS を一緒に使用することで、NaN を識別できます。上記の例のバリエーションとして、IF 文に for_k_fp_snan シンボルと for_k_fp_qnan シンボルを含むこともできます。より簡単にこの方法を行うには、組み込み関数 ISNAN を使用できます。ISNAN を使用した、上記の例の修正は次のようになります。

```
!      The ISNAN function does not need file fordef.for
      real*4 a(100)
!      open an unformatted file as unit 1
!....
      read (1) a
      do i = 1, 100
        if ( isnan (a(i)) ) then
          print *, 'Element ', i, ' contains a NaN'
        end if
      end do
      close (1)
      end
```

このプログラムは、任意の -fopen の値でコンパイルできます。

文字表現

文字列は、次に示すように、メモリ内の連続したバイトの並びです。

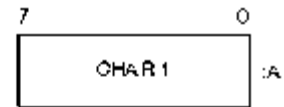


文字列の先頭バイトのアドレス A と文字列長 (バイト数) L の 2 つの属性によって、文字列が指定されます。文字列長 L は 1 ~ 2,147,483,647 (2**31-1) の範囲です。

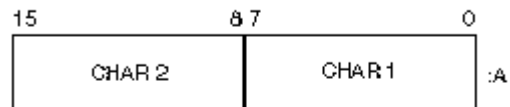
Hollerith 表現

Hollerith 定数は、次の図で示すように、内部的に 1 バイトにつき 1 文字として格納されます。

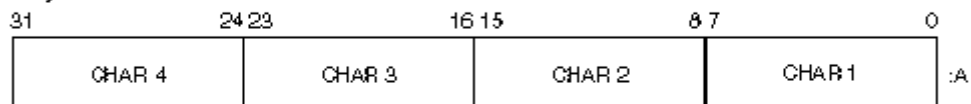
1 Byte



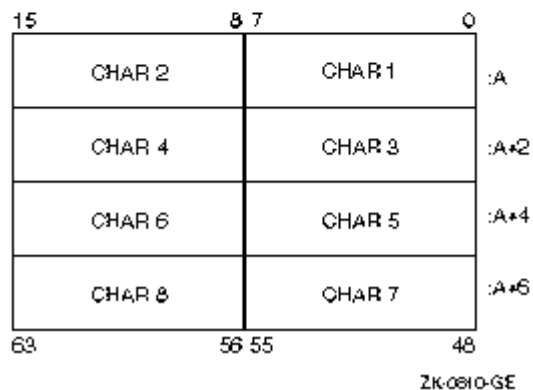
2 Bytes



4 Bytes



8 Bytes



ZK-0810-GE

書式なしデータの変換

書式なしデータの変換の概要

このセクションでは、インテル® Fortran の非ネイティブ書式なし数値データの読み書きを行う方法について説明します。

次のトピックを参照してください。

[サポートされるネイティブ数値形式と非ネイティブ数値形式](#)

[数値変換の制限](#)

[データ書式の指定方法: 概要](#)

[非ネイティブ・データのポータリング](#)

サポートされるネイティブ数値形式と非ネイティブ数値形式

インテル® Fortran は、次のリトル・エンディアン浮動小数点形式をメモリ上でサポートします:

浮動小数点サイズ	メモリ上の書式
REAL(KIND=4) COMPLEX(KIND=4)	IEEE* S_floating
REAL(KIND=8) COMPLEX(KIND=8)	IEEE T_floating
REAL(KIND=16) COMPLEX(KIND=16)	IEEE X_floating

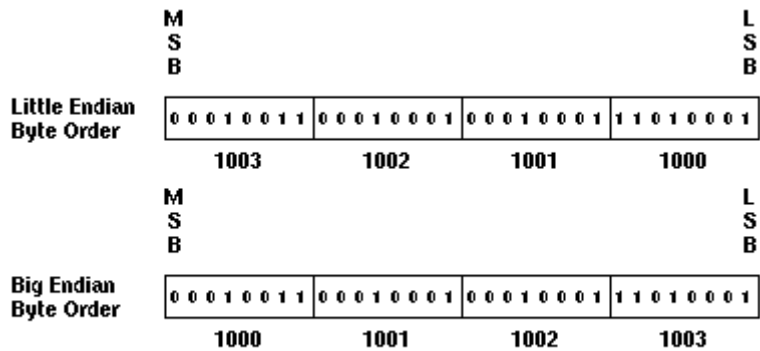
プログラムが、メモリ上の書式とデータサイズが異なる浮動小数点形式を含む書式なしデータファイルの読み書きを行う必要がある場合、その書式なしデータの変換を要求することができます。

各種のコンピュータにおけるデータ格納は、リトル・エンディアンまたはビッグ・エンディアンの格納規則を使用します。格納方法の規則は、一般に、複数のバイトにわたる数値に次のように適用されます:

- リトル・エンディアンでの格納
 - 最下位ビット (LSB) の値は、最も小さいアドレスのバイトに含まれています。
 - 最上位ビット (MSB) の値は、最も大きいアドレスのバイトに含まれています。
 - 数値のアドレスは、LSB を含むバイトです。それ以降の大きいアドレスのバイトには、より上位のビットが含まれています。
- ビッグ・エンディアンでの格納
 - 最下位ビット (LSB) の値は、最も大きいアドレスのバイトに含まれています。
 - 最上位ビット (MSB) の値は、最も小さいアドレスのバイトに含まれています。
 - 数値のアドレスは、MSB を含むバイトです。それ以降の大きいアドレスのバイトには、より下位のビットが含まれています。

次の図は、この 2 つのバイト順方式の違いを示します。

INTEGER 値のリトル・エンディアンとビッグ・エンディアンの格納方法



ZK-6654A-GE

書式なしデータファイルをビッグ・エンディアンとリトル・エンディアンのコンピュータ間で移動するためには、データを変換する必要があります。

インテル Fortran は、プログラムがいくつかの非ネイティブ浮動小数点形式とビッグ・エンディアンの INTEGER または浮動小数点形式で、書式なしデータ（もともと書式なし I/O 文で書き出されたもの）を読み書きする機能を提供しています。サポートされている非ネイティブ浮動小数点形式には、VAX FORTRAN がサポートしている VAX* リトル・エンディアン浮動小数点形式、Sun Microsystems システムと IBM RISC* System/6000 システムの大部分で使用されている標準 IEEE ビッグ・エンディアン浮動小数点形式、IBM 浮動小数点形式（IBM の System/370 とこれに似たシステムで使われているもの）、および CRAY* 浮動小数点形式などがあります。

一般に、書式なしデータの変換は書式付きデータの変換よりも高速で、浮動小数点数の精度が失われる可能性も低くなります。

ネイティブ・メモリ形式には、リトル・エンディアン整数とリトル・エンディアンの IEEE 浮動小数点形式、REAL(KIND=4) 宣言と COMPLEX(KIND=4) 宣言の S_floating、REAL(KIND=8) 宣言と COMPLEX(KIND=8) 宣言の T_floating、および REAL(KIND=16) 宣言と COMPLEX(KIND=16) 宣言の IEEE X_floating があります。

次の表は、サポートされている非ネイティブ書式なしファイル形式のキーワードとそのデータ型のリストです：

非ネイティブ数値形式、キーワード、およびサポートされているデータ型

キーワード	説明
BIG_ENDIAN	適切な INTEGER サイズ（1 バイト、2 バイト、4 バイト、8 バイト）のビッグ・エンディアン整数データと REAL および COMPLEX の拡張単精度数/拡張倍精度数の IEEE 浮動小数点形式。INTEGER (KIND=1) データまたは INTEGER*1 データはリトル・エンディアンとビッグ・エンディアンで同じです。
CRAY	適切な INTEGER サイズ（1 バイト、2 バイト、4 バイト、8 バイト）のビ

	ビッグ・エンディアン整数データと REAL および COMPLEX の単精度数/倍精度数のビッグ・エンディアン CRAY 独自の浮動小数点形式。
FDX	<p>適切な INTEGER サイズ (1 バイト、2 バイト、4 バイト、8 バイト) のネイティブ・リトル・エンディアン整数と次のリトル・エンディアン独自の浮動小数点形式:</p> <ul style="list-style-type: none"> REAL (KIND=4) および COMPLEX (KIND=4) の VAX F_float REAL (KIND=8) および COMPLEX (KIND=8) の VAX D_float REAL (KIND=16) および COMPLEX (KIND=16) の IEEE スタイル X_float
FGX	<p>適切な INTEGER サイズ (1 バイト、2 バイト、4 バイト、8 バイト) のネイティブ・リトル・エンディアン整数と次のリトル・エンディアン独自の浮動小数点形式:</p> <ul style="list-style-type: none"> REAL (KIND=4) および COMPLEX (KIND=4) の VAX F_float REAL (KIND=8) および COMPLEX (KIND=8) の VAX G_float REAL (KIND=16) および COMPLEX (KIND=16) の IEEE スタイル X_float
IBM	適切な INTEGER サイズ (1 バイト、2 バイト、4 バイト、8 バイト) のビッグ・エンディアン整数データと REAL および COMPLEX の単精度数/倍精度数のビッグ・エンディアン IBM 専用 (System/370 とこれに似たシステムで使われているもの) 浮動小数点形式。
LITTLE_ENDIAN	<p>適切な INTEGER サイズ (1 バイト、2 バイト、4 バイト、8 バイト) のネイティブ・リトル・エンディアン整数と次のネイティブ・リトル・エンディアン IEEE 浮動小数点形式:</p> <ul style="list-style-type: none"> REAL (KIND=4) および COMPLEX (KIND=4) の S_float REAL (KIND=8) および COMPLEX (KIND=8) の T_float REAL (KIND=16) および COMPLEX (KIND=16) の IEEE スタイル X_float
NATIVE	メモリとディスクの間で変換は行われません。これは書式なしファイルにおけるデフォルトです。
VAXD	<p>適切な INTEGER サイズ (1 バイト、2 バイト、4 バイト、8 バイト) のネイティブ・リトル・エンディアン整数と次のリトル・エンディアン VAX 独自の浮動小数点形式:</p> <ul style="list-style-type: none"> REAL (KIND=4) および COMPLEX (KIND=4) の VAX F_float REAL (KIND=8) および COMPLEX (KIND=8) の VAX D_float

	<ul style="list-style-type: none"> REAL (KIND=16) および COMPLEX (KIND=16) の VAX H_float
VAXG	<p>適切な INTEGER サイズ (1 バイト、2 バイト、4 バイト、8 バイト) のネイティブ・リトル・エンディアン整数と次のリトル・エンディアン VAX 独自の浮動小数点形式:</p> <ul style="list-style-type: none"> REAL (KIND=4) および COMPLEX (KIND=4) の VAX F_float REAL (KIND=8) および COMPLEX (KIND=8) の VAX G_float REAL (KIND=16) および COMPLEX (KIND=16) の VAX H_float

非ネイティブ形式を読み取ると、ディスク上の非ネイティブ形式はメモリ上のネイティブ形式に変換されます。変換後の非ネイティブ値がネイティブ・データ型の範囲を超えている場合、ランタイム・メッセージが表示されます。

数値変換の制限

インテル® Fortran 浮動小数点変換におけるソリューションは、すべての浮動小数点変換要求に対応できるものではありません。

例えば、レコード構造体変数中のデータフィールド (STRUCTURE 文で指定) および構造型のデータ要素 (TYPE 文) は変換されません。後でプログラムが個々のフィールドとして調べる場合、プログラムを変更しない限り、これらのデータはディスク上に格納されていた 2 進書式のままになっています。EQUIVALENCE 文では、I/O 文で指定された変数のデータ型が使用されます。

プログラムが、複数の形式を持つ浮動小数点フィールドを含んでいる I/O レコードを、それぞれの変数ではなく、1 つの変数 (配列など) に読み取る場合、これらのフィールドは変換されません。後にプログラムが個々のフィールドとして調べる場合、プログラムを変更しない限り、これらのデータはディスク上に格納されていた バイナリ形式のままになっています。

データ書式の指定方法

データ書式の指定方法: 概要

書式なしデータに対して非ネイティブ数値形式を指定する方法は 6 つあります。

- 特定のユニット番号に対応する環境変数を、ファイルを開く前に設定する。環境変数は、`FORT_CONVERT n` という名前を持ちます (n はユニット番号を指します)。詳細は、「[環境変数 FORT_CONVERT \$n\$ を使用する方法](#)」を参照してください。
- 特定のファイル名拡張子に対応する環境変数を、ファイルを開く前に設定する。環境変数は、`FORT_CONVERT.ext` または `FORT_CONVERT_ext` という名前を持ちます (`ext` はファイル名拡張子またはサフィックスを指します)。詳細は、「[環境変数 FORT_CONVERT.ext または FORT_CONVERT_ext を使用する方法](#)」を参照してください。
- 一連のユニットに対応する環境変数を、ファイルを開く前に設定する。環境変数は、`F_UFMTENDIAN` という名前を持ちます。詳細は、「[環境変数 F_UFMTENDIAN を使用する方法](#)」を参照してください。
- 特定のユニット番号に対応する `OPEN` 文の `CONVERT` キーワードを指定する。詳細は、「[OPEN 文 CONVERT を使用する方法](#)」を参照してください。
- `CONVERT=keyword` 指示子を指定する `OPTIONS` 文を使用して、プログラムをコンパイルする。この方法は、プログラムが指定する書式なしデータを使用するすべてのユニット番号に影響を与えます。詳細は、「[OPTIONS 文を使用する方法](#)」を参照してください。
- コマンドライン `-convert keyword` オプションでコンパイルする。この方法は、プログラムが指定する書式なしデータを使用するすべてのユニット番号に影響を与えます。詳細は、「[-convert コンパイラ・オプションを使用する方法](#)」を参照してください。

上記のいずれの方法も指定されていない場合、ネイティブ・リトル・エンディアン形式が使用されます (ディスクとメモリの間で変換は行われません)。

「[サポートされるネイティブ数値形式と非ネイティブ数値形式](#)」に示したキーワードは、リトル・エンディアンおよびビック・エンディアンのみをサポートする環境変数 `F_UFMTENDIAN` を使用する方法を除き、すべて上記の方法とともに使用できます。

複数の方法を指定する場合、書式なしデータを含むファイルを開くと、次の優先順位が適用されます:

1. 特定のユニット番号に対応する環境変数 (`FORT_CONVERT n`) をチェックします。これは、特定のユニットで開かれたすべてのファイルで適用されます。
2. 特定のファイル名拡張子に対応する環境変数 (`FORT_CONVERT.ext` は、`FORT_CONVERT_ext` の前にチェックされます) をチェックします。これは、指定したファイル拡張子を持つすべてのファイルで適用されます。
3. 特定のユニット番号 (またはすべてのユニット) に対応する環境変数 (`F_UFMTENDIAN`) をチェックします。
4. `OPEN` 文の `CONVERT` 指示子をチェックします。

5. プログラムのコンパイル時に `CONVERT=keyword` 指示子を持つ `OPTIONS` 文が指定されていたかどうかをチェックします。
6. プログラムのコンパイル時に、コンパイル・オプション `-convert keyword` が指定されていたかどうかをチェックします。

環境変数 `FORT_CONVERTn` を使用する 方法

この方法を使用することで、指定したそれぞれのユニット番号に対して、非ネイティブな数値形式を指定できます。暗黙的または明示的な `OPEN` が行われる前に、適切な環境変数をユニット番号に設定することで、ランタイムで数値形式を指定できます。

ファイルを開いた時点で適切な環境変数が設定されていると、この方法は他のすべての方法よりも優先されるので、その環境変数が常に使用されます。例えば、この方法を使用することで、プログラムで指定された形式の代わりに、特定の形式をユニット番号に対して使用するように指定できます（1 回限りのファイル変換などに便利です）。

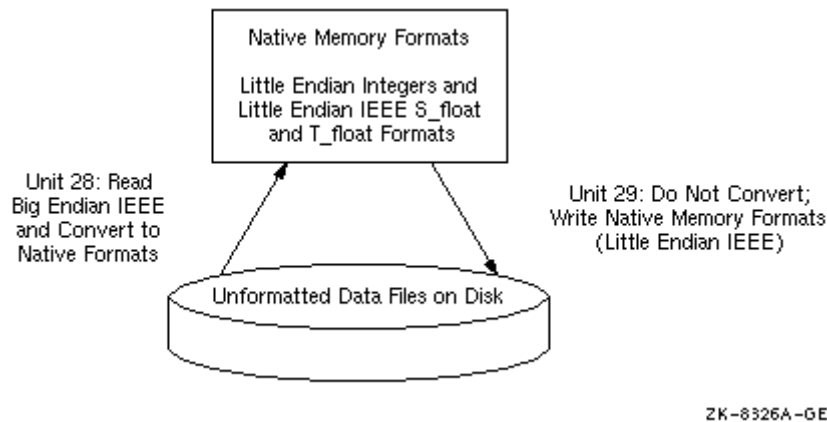
例えば、書式なし I/O 文を使ってユニット 28 から数値データを読み出し、ユニット 29 にそのデータを書き込むコンパイル済みのプログラムがあると仮定します。また、このプログラムを使って、ユニット 28 から非ネイティブ・ビッグ・エンディアン（IEEE 浮動小数点）形式を読み出し、ユニット 29 にそのデータをネイティブ・リトル・エンディアン形式で書き出すとします。この場合、データはユニット 28 からの読み出し時に、ビッグ・エンディアン IEEE 形式からネイティブ・リトル・エンディアン IEEE メモリ形式に変換され、ネイティブ・リトル・エンディアン IEEE 形式のまま、ユニット 29 に書き出されます。

次の一連のコマンドを実行することで、ソースコードを編集したり、プログラムを再びコンパイルすることなく、プログラムを実行する前に適切な環境変数を設定できます（`/usr/users/leslie/convieee`）:

```
setenv FORT_CONVERT28 BIG_ENDIAN
setenv FORT_CONVERT29 NATIVE
/usr/users/leslie/convieee
```

以下の図では、環境変数の設定後にサンプル・ファイル（`/usr/users/leslie/convieee`）を実行したときの、ディスク上とメモリ上で使用されるデータ形式を示します。

書式なしファイルの変換例



この方法は、他のどの方法よりも優先されます。

環境変数 FORT_CONVERT.ext または FORT_CONVERT_ext を使用する方法

この方法を使用することで、指定したそれぞれのファイル拡張子（サフィックス）に対して、非ネイティブな数値形式を指定できます。暗黙的または明示的な OPEN が行われる前に、適切な環境変数を 1 つ以上の書式なしファイルに設定することで、ランタイムで数値形式を指定できます。FORT_CONVERT.ext 書式または FORT_CONVERT_ext 書式が使用可能です（ここで、*ext* はファイル拡張子またはサフィックスを意味します）。環境変数 FORT_CONVERT.ext は、環境変数 FORT_CONVERT_ext の前に検証されます（*ext* が同一の場合）。

例えば、書式なし I/O 文を使って 1 つのファイルから数値データを読み出し、他のファイルにそのデータを書き込むコンパイル済みのプログラムがあると仮定します。また、このプログラムを使って、拡張子が .data であるファイルから非ネイティブ・ビッグ・エンディアン（IEEE 浮動小数点）形式を読み出し、他の .data ファイルにそのデータをネイティブ・リトル・エンディアン形式で書き出すとします。この場合、データは、.data ファイルからの読み出し時に、ビッグ・エンディアン IEEE 形式からネイティブ・リトル・エンディアン IEEE メモリ形式（S_float および T_float）に変換され、ネイティブ・リトル・エンディアン IEEE 形式のまま、.data ファイルに書き出されます（ここでは、ユニット番号に対して、環境変数 FORT_CONVERT.DATA と FORT_CONVERT_n が定義されていないことを前提にしています）。

次の一連のコマンドを実行することで、ソースコードを編集したり、プログラムを再びコンパイルすることなく、プログラムを実行する前に適切な環境変数を設定できます：


```
setenv FORT_CONVERT.DAT BIG_ENDIAN
/usr/users/proj2/cvbigend
```

FORT_CONVERT n は、この方法よりも優先されます。ファイルを開いたときに、適切な環境変数が設定されていると、環境変数 FORT_CONVERT n がユニット番号に対して設定されていない場合、環境変数 FORT_CONVERT.ext または FORT_CONVERT_ext が使用されます。

FORT_CONVERT n 、および FORT_CONVERT.ext または FORT_CONVERT_ext は、環境変数を設定する他の方法よりも優先されます。例えば、この方法を使用することで、プログラムで指定された形式の代わりに、特定の形式をユニット番号に対して使用するように指定できます (1 回限りのファイル変換などに便利です)。

FORT_CONVERT.ext 書式または FORT_CONVERT_ext 書式を使用して適切な環境変数を設定できます。一部の Linux* コマンドシェルでは、環境変数名にドット (.) を使用できないため、FORT_CONVERT_ext 書式を使用することをお勧めします。また、同じ拡張子 (ext) に対して、FORT_CONVERT.ext および FORT_CONVERT_ext の両方を定義した場合、FORT_CONVERT.ext で定義されたファイルが使用されます。

環境変数 F_UFMTENDIAN を使用する方 法

リトル・エンディアン – ビッグ・エンディアンの変換機能は、Fortran 書式なし入力/出力の演算のためのものです。この機能は、ビッグ・エンディアン・データ編成を持つファイルの開発および処理を有効にします。

また、ビッグ・エンディアン・データ形式を受け入れるプロセッサ上で開発されたファイルの処理および IA-32 ベースのリトル・エンディアン・システム上のそのようなプロセッサ用のファイルの作成を有効にします。

リトル・エンディアン – ビッグ・エンディアンへの変換は、次の操作によって実現されます。

- WRITE 操作は、リトル・エンディアン形式をビッグ・エンディアン形式に変換します。
- READ 操作は、ビッグ・エンディアン形式をリトル・エンディアン形式に変換します。

リトル・エンディアン – ビッグ・エンディアンの変換環境変数

リトル・エンディアン – ビッグ・エンディアンの変換機能を使用するためには、`F_UFMTENDIAN` 環境変数を設定して、変換に使用されるユニット数を指定します。その後、このユニット数を使用する `READ/WRITE` 文が対応する変換を実行します。他の `READ/WRITE` 文は、通常どおり動作します。

一般的に、変数はセミコロンで分割される 2 つの部分で構成されます。`F_UFMTENDIAN` 値の中では、スペースは使用できません。この変数の構文は、次のとおりです。

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

各アイテムの意味は次のとおりです。

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- `MODE` は、現在のデータ形式を定義し、ファイルで表現されます。省略できます。
キーワードの `little` は、データがリトル・エンディアン形式で、変換されないことを意味します。これはデフォルトで使用されます。
キーワードの `big` は、データがビッグ・エンディアン形式で、変換されることを意味します。
- `EXCEPTION` は、`MODE` の例外リストの定義に使用します。`EXCEPTION` キーワード (`little` または `big`) は `EXCEPTION` リストからのユニットを接続するファイル内のデータ形式を定義します。この値は、リストされたユニットの `MODE` 値を無効にします。`EXCEPTION` キーワードとコロンは省略できます。キーワードが省略されたときのデフォルトは、`big` です。
- 各リストメンバの `U` とは、シンプルユニット番号またはユニット番号です。リストメンバの上限は、64 です。
`decimal` は、負でない小数で 2^{32} よりも少ない値です。

シェル内の変数設定のコマンドライン:

```
Sh: export F_UFMTENDIAN=MODE;EXCEPTION
```



Note

セミコロンがある場合は、環境変数値が引用符で囲まれていなければなりません。

その他の環境変数設定

また、環境変数には次の構文があります。

```
F_UFMTENDIAN=u[,u] ...
```

シェル内の変数設定のコマンドライン:

```
Sh: export F_UFMTENDIAN=u[,u] ...
```

使用例:

1. `F_UFMTENDIAN=big`

すべての入力/出力操作で `READ` 上では、ビッグ・エンディアンからリトル・エンディアンへの変換、`WRITE` 上ではリトル・エンディアンからビッグ・エンディアンへの変換を実行します。

2. `F_UFMTENDIAN="little;big :10,20"`

または `F_UFMTENDIAN=big :10,20`

または `F_UFMTENDIAN=10,20`

この場合、ユニット番号が 10 および 20 で、入力/出力演算はビッグ・エンディアン - リトル・エンディアンの変換を実行します。

3. `F_UFMTENDIAN="big;little:8"`

この場合、ユニット番号 8 では変換操作は行われません。その他のすべてのユニットで入力/出力演算はビッグ・エンディアン - リトル・エンディアンの変換を実行します。

4. `F_UFMTENDIAN=10-20`

変換目的で、10、11、12 ... 19、20 ユニットを定義します。これらのユニットでは、入力/出力演算は、ビッグ・エンディアン - リトル・エンディアンの変換を実行します。

5. `F_UFMTENDIAN=10,100` を設定して、次のプログラムを実行したとします。

```
integer*4    cc4
integer*8    cc8
integer*4    c4
integer*8    c8
c4 = 456
c8 = 789
```

```
C  prepare little endian representation of data
```

```
open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)
```

C prepare big endian representation of data

```
open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)
```

C read big endian data and operate with them on
C little endian machine

```
open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4
```

C Any operation with data, which have been read

```
C ...
close(100)
stop
end
```

lit.tmp ファイルと big.tmp ファイルを od ユーティリティで比較します。

```
> od -t x4 lit.tmp
0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034
```

```
> od -t x4 big.tmp
0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

これらのファイルでは、バイトの順番が異なることがわかります。

OPEN 文 CONVERT を使用する方法

この方法を使用することで、指定したそれぞれのユニット番号に対して、非ネイティブな数値形式を指定できます。この方法を使用するには、明示的な OPEN 文から、ユニット番号に対してファイルの数値形式を指定する必要があります。

この方法は、OPTIONS 文およびコンパイラ・オプション `-convert keyword` を使用する方法よりも優先されますが、環境変数を設定する方法よりは優先順位が低くなっています。

次のサンプル・ソースコードでは、ユニット 15 から書式なし VAXD 数値データを読み出すための OPEN 文を示します。このデータは、ネイティブ・リトル・エンディアン形式で処理し、ユニット 20 にその形式で書き込むことができます。(CONVERT キーワードまたは環境変数 `FORT_CONVERT20`、`FORT_CONVERT.dat`、`FORT_CONVERT_dat`、または `F_UFMTENDIAN` が指定されていない場合、ユニット 20 では、ネイティブ・リトル・エンディアン・データが使用されます)。

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15
```

```
.
```

```
OPEN (FILE='graph3_t.dat', FORM='UNFORMATTED', UNIT=20)
```

OPEN 文の CONVERT キーワード値をハードコーディングすると、コンパイル後に変更することができません。ただし、OPEN を行う前に、CONVERT キーワード値を変数に設定し、適切な書式を選択できるメニューをユーザに表示することで(メニューの選択項目によって変数を設定します)、ランタイム時に特定の書式を選択することができます。

また、環境変数 (`FORT_CONVERTn`、`FORT_CONVERT.ext`、`FORT_CONVERT_ext`、または `F_UFMTENDIAN`) を設定することで、ランタイム時にユニット番号に対して特定の書式を選択できます。この方法は、OPEN 文の CONVERT キーワードを使用する方法よりも優先されます。

OPTIONS 文を使用する方法

環境変数で設定する方法または OPEN 文の CONVERT *keyword* を併用しない限り、この方法では、すべての書式なしファイルのユニット番号に対して、数値ファイル形式を 1 つしか指定できません。

コンパイル時に数値形式を指定し、すべてのルーチンを、同じ `OPTIONS` 文の `CONVERT keyword` 指示子を使用してコンパイルする必要があります。

環境変数で設定する方法、および `OPEN` 文の `CONVERT` を利用する方法は、この方法よりも優先されます。例えば、環境変数 `FORT_CONVERTn` または `OPEN` 文の `CONVERT` を使用することで、各ユニット番号に、`ifort` オプションで設定した以外の書式を指定できます。

この方法は、`convert keyword` コンパイラ・オプションを使用する方法よりも優先されます。

`OPTIONS` 文を使用することで、対応する `ifort` コマンドの指示子を使用する代わりに、(メモリ上と書式なしファイル内での) 適切な浮動小数点形式を指定することができます。例えば、書式なしファイル形式として、`VAX F_floating` と `G_floating` を使用するには、次の `OPTIONS` 文を指定します：

```
OPTIONS /CONVERT=VAXG
```

この方法は、すべてのユニット番号に影響を与えるため、環境変数を設定する方法または `OPEN` 文の `CONVERT keyword` を併用して特定のユニット番号に異なる形式を指定しない限りは、1 つの形式からデータを読み取り、それを他の形式で書き出すことはできません。

-convert コンパイラ・オプションを使用する方法

コンパイラ・オプション `-convert` を使用する方法では、その他の方法と併用しない限り、すべての書式なしファイルのユニット番号に対して、数値形式を 1 つしか指定できません。

コンパイル時に数値形式を指定し、同じ `-convert keyword` コンパイラ・オプションを使用してすべてのルーチンをコンパイルする必要があります。1 つのソース・プログラムを異なる `ifort` コマンドでコンパイルし、それぞれ特定の書式を読み取る複数の実行プログラムを作成することもできます。

他の方法を指定すると、この方法よりも優先されます。例えば、環境変数または `OPEN` 文の `CONVERT` キーワードを使用することで、`-convert keyword` コンパイラ・オプションで指定した書式とは異なる書式を使用するように、各ユニット番号を指定できます。

次のコマンドは、すべてのユニット番号で VAX D_floating（および F_floating）浮動小数点データを使用するように、プログラム `file.for` をコンパイルします（他の方法が指定されていない場合のみ）。データは、ファイル形式とリトル・エンディアン・メモリ形式（リトル・エンディアン整数、S_float および T_float リトル・エンディアン IEEE 浮動小数点形式）の間で変換されます。作成されたファイル `vconvert.exe` は、次のように実行することができます：

```
ifort file.for -o vconvert.exe -convert vaxd
```

この方法は、すべての書式なしファイルのユニット番号に影響を与えるため、`-convert keyword` コンパイラ・オプションだけでは、1つの書式からデータを読み取り、それを他の書式で書き出すことはできません。環境変数を設定する方法または OPEN 文の CONVERT キーワードを使用する方法と併用すれば、特定のユニット番号に対して、別の書式を指定することができます。

非ネイティブ・データのポータリング

非ネイティブ・データを移植する際の注意事項を示します。

- 書式なしデータとともにソースコードを移植する場合、各ベンダは書式なしファイルのレコード長 (RECL 指定子) を指定するのに異なる単位を使用していることがあります。書式付きファイルは文字数 (バイト) 単位で指定されますが、書式なしファイルはインテル® Fortran (デフォルト設定) とその他の一部のベンダではロングワード単位で指定されます。

ソースファイルの変更なしに、書式なしファイルの RECL の単位 (バイトまたはロングワード) を指定するには、`-assume byterecl` コンパイラ・オプションを使用します。

Fortran 95 標準 (American National Standard Fortran 95、ANSI X3J3/96-007、および International Standards Organization standard ISO/IEC 1539-1:1997) には、次のように記されています：「ファイルが書式なし入出力のために接続された場合、長さはプロセッサ依存の単位で測定される」

- 一部のベンダは、レコード型を決定するために異なる OPEN 文のデフォルト設定を適用します。インテル Fortran のデフォルトのレコード型 (RECORDTYPE) は、OPEN 文の ACCESS および FORM 指定子の値に依存します。
- 一部のベンダは、論理データ型に異なる識別子を使用します。例えば、“true”を表すのに、01 の代わりに 16 進の FF を使用するなどです。
- 移植するソースコードは、ビッグ・エンディアンで使用するよう書かれていることがあります。

Fortran I/O

Fortran I/O の概要

次のトピックを参照してください:

[論理 I/O ユニット](#)

[I/O 文の種類](#)

[I/O 文の形式](#)

[ファイルとファイルの特性の概要](#)

[ファイルのアクセスと割り当て](#)

[デフォルトのパス名とファイル名](#)

[事前結合された標準 I/O ファイルの使用](#)

[ファイルを開く: OPEN 文](#)

[ファイル情報の取得: INQUIRE 文](#)

[ファイルを閉じる: CLOSE 文](#)

[レコード操作の概要](#)

[ユーザが提供する OPEN プロシージャ: USEROPEN 指定子](#)

[レコード型の形式](#)

[Microsoft Fortran Power* Station 互換ファイル](#)

論理 I/O ユニット

インテル® Fortran では、**論理ユニット**とは、プログラムとデバイス間またはプログラムとファイル間のデータ転送で使用されるチャネルを指します。各論理ユニットは、論理ユニット番号で識別されます。論理ユニット番号は、0 から最大 2,147,483,647 ($2^{31}-1$) までの負ではない整数からなります。次に例を示します。


```
READ (2,100) I,X,Y
```

この READ 文では、論理ユニット 2 に対応するデバイスまたはファイルから、100 のラベルが付いた FORMAT 文で指定された形式で、データが入力されます。ファイルを開くときに、UNIT 指定子を使用してユニット番号を指定します。

本来、Fortran プログラムはデバイスに依存しません。論理ユニット番号と物理ファイル間の関連付けは、実行時に行なわれます。ソース・プログラムで指定した論理ユニット番号を変更する代わりに、実行時にこの関連付けを変更して、プログラムの要求に利用可能なリソースを対応させます。例えば、プログラムの実行前にスクリプト・ファイルを使用することで、適切な環境変数を設定したり、端末ユーザがディレクトリ・パス、ファイル名、またはその両方を入力することができます。

OPEN 文で指定した論理ユニット番号を、その開いたファイルに対して実行する他の I/O 文 (READ、WRITE など) で使用します。

OPEN 文は、ユニット番号と外部ファイルを結合し、OPEN 文指定子を使用することで、ファイル属性およびランタイム・オプションを明示的に指定できます (内部ファイル以外のすべてのファイルは、外部ファイルと呼ばれます)。

特定のユニット番号は、標準デバイスに事前に結合されます。ユニット番号 5 は stdin に、ユニット 6 は stdout に、ユニット 0 は stderr に、それぞれ結合されます。ユニット 5 および 6 が OPEN 文によって明示的に開かれていないのに、レコード I/O 文 (READ または WRITE など) で使用する場合は、実行時に、インテル Fortran は暗黙的にユニット 5、6、および 0 を開き、各ユニット番号をオペレーティング・システムの標準 I/O ファイルに結合します (対応する `FORTn` 環境変数が設定されていない場合)。

I/O 文の種類

インテル® Fortran の I/O 文を以下の表に示します:

カテゴリと I/O 文の名前	説明
ファイル結合	
OPEN	ユニット番号を外部ファイルと結合し、ファイル結合の特性を指定します。
CLOSE	ユニット番号と外部ファイル間の結合を無効にします。
ファイル照会	
DEFINE FILE	直接アクセスを行う相対ファイルに対してファイル特性を指定し、OPEN 文と同じように、ユニット番号をそのファイルに結合しま

	す。FORTRAN-77 以前のコンパイラとの互換性のために提供されています。
INQUIRE	指定されたファイル、ユニットへの結合、または出力項目リストの長さに関する情報を返します。
レコード位置	
BACKSPACE	レコード位置を前のレコードの先頭に移動します (シーケンシャル・アクセスのみ)。
DELETE	相対ファイル内の現在のレコード位置にあるレコードを、削除済みとしてマークを付けます (直接アクセスのみ)。
ENDFILE	現在のレコードの後にファイル終了マーク (EOF) を書き込みます (シーケンシャル・アクセスのみ)。
FIND	直接アクセスを行うファイル内のレコード位置を変更します。FORTRAN-77 以前のコンパイラとの互換性のために提供されています。
REWIND	レコードの位置をファイルの先頭に設定します (シーケンシャル・アクセスのみ)。
レコード入力	
READ	データを外部ファイルのレコードまたは内部ファイルから内部記憶域に転送します。
UNLOCK	以前に READ 文でロックした相対ファイル内のレコードまたはシーケンシャル・ファイル内のレコードを解放します。
ACCEPT	入力を stdin から読み込みます。READ とは異なり、ACCEPT は書式付きシーケンシャル入力だけが可能で、ユニット番号は指定しません。
レコード出力	
WRITE	データを内部記憶域から外部ファイルのレコードまたは内部ファイルに転送します。
REWRITE	データを内部記憶域から、外部ファイル内の現在のレコード位置にあるレコードに転送します (直接アクセスの相対ファイルのみ)。
TYPE	レコード出力を stdout に書き出します (PRINT と同じ)。
PRINT	データを内部記憶域から stdout に転送します。WRITE とは異なり、PRINT は、書式付きシーケンシャル出力だけが可能で、ユニット番号は指定しません。

READ 文、WRITE 文、REWRITE 文、TYPE 文、および PRINT 文に加えて、他の I/O レコード関連の文も、特定のファイル編成に限定されます。次に例を示します:

- DELETE 文は、相対ファイルだけに適用されます。(削除したレコードを検出するには、プログラムをコンパイルする際に -vms オプションを指定します)。
- BACKSPACE 文は、シーケンシャル・アクセスを行うために開いたシーケンシャル・ファイルだけに適用されます。
- REWIND 文は、シーケンシャル・アクセスを行うために開いたシーケンシャル・ファイルおよび直接アクセスを行うファイルだけに適用されます。
- ENDFILE 文は、シーケンシャル・アクセスを行うために開いた特定の種類のシーケンシャル・ファイルおよび直接アクセスを行うファイルだけに適用されます。
- UNLOCK 文は、相対ファイル内のレコードまたはシーケンシャル・ファイル内のレコードだけに適用されます。

ファイル関連の文 (OPEN、INQUIRE、および CLOSE) は、すべての相対ファイルまたはシーケンシャル・ファイルに適用されます。

I/O 文の形式

各種のレコード I/O 文は、さまざまな形式でコード化できます。選択する形式は、データの種類と処理方法により異なります。ファイルを開くときに、FORM 指定子を使用して形式を指定します。

I/O 文の形式は次のとおりです:

- **書式付き I/O 文**には、プログラムの内部 (バイナリ) 形式からレコードの外部 (可読文字) 形式へのデータ変換、あるいはその逆のデータ変換の制御に使用される、明示的な書式指定子が含まれます。
- **リスト指定 I/O 文**および **ネームリスト I/O 文**は、機能的に書式付き文と同様です。ただし、これらの文はデータの変換を制御するために使用するメカニズムが異なります: 書式付き I/O 文は明示的な書式指定子を使用しますが、リスト指定 I/O 文および ネームリスト I/O 文はデータ型を使用します。
- **書式なし I/O 文**には、書式指定子が含まれません。そのため、転送されるデータを変換しません (後で読み出すデータを書き込むときに重要です)。

書式付き I/O、リスト指定 I/O、およびネームリスト I/O の各形式では、レコードのプログラムの内部 (バイナリ) 形式から外部 (可読文字) 形式へのデータ変換が必要です。次のような理由があるため、書式なし I/O の使用を検討してください:

- 書式なしデータでは、変換処理を行わないため、I/O が速くなります。
- 書式なしデータでは、出力データをその後で入力データとして使用するとき、浮動小数点数の精度が低下しません。
- 書式なしデータは、ファイル保存領域を浪費しません (バイナリ形式で保存)。

書式付き I/O 文、リスト指定 I/O 文、またはネームリスト I/O 文を使用してファイルにデータを書き込むには、ファイルを開く際に FORM= 'FORMATTED' を指定します。書式なし I/O 文を使用してファイルにデータを書き込むには、ファイルを開く際に FORM= 'UNFORMATTED' を指定します。

書式付き I/O 文、リスト指定 I/O 文、またはネームリスト I/O 文を使用して書き込まれたデータは、書式付きデータとして参照されます。書式なし I/O 文を使用して書き込まれたデータは、書式なしデータとして参照されます。

ファイルからデータを読み出す際には、ファイルにデータを書き込むときと同じ I/O 文形式を使用してください。例えば、書式付き I/O 文でファイルにデータを書き込んだ場合、書式付き I/O 文でそのファイルからデータを読み出さなければなりません。

通常、ファイルで使用する I/O 文の形式は、データの読み出しと書き込みで同じですが、プログラムは書式なしデータを含むファイルを読み出すことも（書式なし入力を使用）、書式付きデータを含む別のファイルへ書き込むこともできます（書式付き出力を使用）。同様に、プログラムは書式付きデータを含むファイルを読み出すことも、書式なしデータを含む別のファイルへ書き込むこともできます。

シーケンシャル・ファイルまたは相対ファイルのレコードをシーケンシャル・アクセスを使用してアクセスできます。相対ファイルおよび特定（固定長）のシーケンシャル・ファイルに対しては、直接アクセスを使用してレコードにアクセスすることもできます。

次の表は、インテル® Fortran プログラムで利用できる主なレコード I/O 文をカテゴリ別に示しています。

ファイルのタイプ、アクセスおよび I/O 形式	使用できる文
外部ファイル、シーケンシャル・アクセス	
書式付き	READ, WRITE, PRINT, ACCEPT, TYPE, REWRITE
リスト指定	READ, WRITE, PRINT, ACCEPT, TYPE
ネームリスト	READ, WRITE, PRINT, ACCEPT, TYPE
書式なし	READ, WRITE, REWRITE
外部ファイル、直接アクセス	
書式付き	READ, WRITE, REWRITE
書式なし	READ, WRITE, REWRITE
内部ファイル	
書式付き	READ, WRITE
リスト指定	READ, WRITE

書式なし

なし



注

REWRITE 文は、相対ファイルに対して直接アクセスを使用する場合のみ使用できます。

ファイルとファイルの特性

ファイルとファイルの特性の概要

次のトピックを参照してください。

[ファイルの編成](#)

[内部ファイルとスクラッチ・ファイル](#)

[レコード型](#)

[レコード・オーバーヘッド](#)

[レコード長](#)

ファイルの編成

*ファイルの編成*とは、レコードが記憶装置上でどのように物理的に配置されるかを示す言葉です。

インテル® Fortran では、2 種類のファイル編成がサポートされています:

- シーケンシャル編成
- 相対編成

デフォルトでは常に OPEN 文で ORGANIZATION= 'SEQUENTIAL' が使用されます。ファイル編成は、OPEN 文の ORGANIZATION 指定子によって指定できます。

シーケンシャル・ファイルは、磁気テープデバイスまたはディスクデバイスに格納できます。また、端末、パイプ、およびラインプリンタなど、その他の周辺デバイスは、シーケンシャル・ファイルとして使用できます。

相対ファイルは、ディスクデバイスに格納しなければなりません。

シーケンシャル編成

シーケンシャルに編成されたファイルは、ファイルに書き出される順番で配置されたレコードから構成されます。例えば、1 番目に書き出されるレコードは、ファイル内で 1 番目のレコードになり、2 番目に書き出されるレコードは、ファイル内で 2 番目のレコードとなります。つまり、レコードは、ファイルの終わりに追加されます。

たいていの場合、シーケンシャル・ファイルは、ファイル内の 1 番目のレコードから順番に読み取られます。また、ディスクに格納されている、固定長のレコード型を持つシーケンシャル・ファイルは、相対レコード番号によっても参照できます（直接アクセス）。

相対編成

相対ファイル内には、セルと呼ばれる番号付きの位置があります。これらのセルは、長さが固定されており、1 を最初のセル、 n をファイル内の最後のセルとして、 $1 \sim n$ の連続した番号が付けられています。個々のセルは、レコードを 1 つ含むか、または空であるかのどちらかです。

相対ファイル内のレコードは、セル番号によって参照されます。セル番号は、レコードの相対レコード番号で、ファイルの先頭を基準とした、そのレコードの相対位置を示します。この相対レコード番号を指定することにより、レコードの位置にかかわらず、直接的にレコードを取得、追加、または削除できます。（削除したレコードを検出するには、プログラムをコンパイルする際に `-vms` オプションを指定します）。

相対ファイルを作成するときに、RECL 値を使用することで、固定長のセルのサイズを指定できます。指定したセルのサイズを超えなければ、セル内に可変長レコードを格納できます。

内部ファイルとスクラッチ・ファイル

インテル® Fortran では、ファイル編成を持たない次の 2 種類のファイルがサポートされています：

- 内部ファイル
- スクラッチ・ファイル

内部ファイル

シーケンシャル・アクセスを使用する場合、内部ファイルを使用してバッファ内の文字データを参照できます。文字変数と文字配列の間で行われる転送と同じように（外部ファイルとは異なり）、内部記憶域の間で転送が行なわれます。

内部ファイルは、次のものから構成されます:

- 文字変数
- 文字配列要素
- 文字配列
- 部分文字列
- ベクトル添字なしの文字配列セクション

READ 文または WRITE 文でユニット番号を指定する代わりに、文字スカラ・メモリ参照または文字配列名参照の形式で、内部ファイル指定子を使用します。

内部ファイルは、指定された文字の内部記憶空間（変数バッファ）であり、固定長レコードのシーケンシャル・ファイルとして扱われます。内部 I/O を実行するには、書式付きおよびリスト指定のシーケンシャル READ 文と WRITE 文を使用します。内部ファイルでは、OPEN や INQUIRE のようなファイル関連の文を使用することはできません（ユニット番号が使用されません）。

内部ファイルが単一の文字変数、配列要素、または部分文字列のみを含む場合、そのファイルは、単一のレコードで構成されています。そのレコードの長さは、レコードに含まれる文字変数、配列要素、または部分文字列の長さに一致します。内部ファイルが文字配列のみを含む場合、そのファイルは一連のレコードで構成されており、各レコードには単一の配列要素が含まれています。内部ファイル内のレコードの順序は、添字の進行順序によって決まります。

内部ファイルのレコードは、レコードを構成する文字変数、配列要素、または部分文字列が定義されている（値がレコードに代入されている）場合のみ、読み出すことができます。

各 READ 文および WRITE 文に先立って、内部ファイルの位置は、常にその第 1 レコードの先頭に位置します。

スクラッチ・ファイル

スクラッチ・ファイルは、OPEN 文で STATUS= ' SCRATCH ' を指定して作成できます。デフォルトでは、これらの一時ファイルは、OPEN 文 の DEFAULTFILE（指定された場合のみ）で指定されたディレクトリに作成されます（後で削除されるときにも、このディレクトリから削除されます）。

レコード型

レコード型は、ファイル内のすべてのレコード長が同じか、異なるか、あるいは他の規則を使用してレコードの終了位置や開始位置を定義しているかを示します。

シーケンシャル・ファイルでは、任意のレコード型が使用できます。相対ファイルには、固定長レコード型を使用する必要があります。

新規のファイルを作成したり、既存のファイルを開く際には、次に説明されているレコード型のいずれかを指定してください。

[「レコード型の形式」](#)も参照してください。

固定長レコード型

ファイル内のレコードは、すべて同じ長さでなければなりません。

ファイルが開かれるときに、レコード長 (RECL) を指定しなければなりません。

[「固定長レコード」](#)も参照してください。

可変長レコード型

ファイル内のレコードの長さは異なってもかまいません。

レコード長情報は、各レコードの先頭と終わりに制御バイトとして格納されます。

[「可変長レコード」](#)も参照してください。

セグメント・レコード型

可変長の書式なしレコードを 1 つまたは複数個含む 1 つの論理レコードです。書式なしシーケンシャル・アクセスでのみ使用できます。

アプリケーションで、Fortran 以外の言語で記述されているプログラムやインテル・プラットフォーム用に記述されていないプログラムに対して同じファイルを使用しなければならない場合には、セグメント・レコード型を使用しないでください。

[「セグメント・レコード」](#)も参照してください。

ストリーム・レコード型

ストリーム・ファイルはレコード単位にはグループ化されておらず、レコード区切り文字を使用しません。

ストリーム・ファイルには、文字またバイナリ・データが含まれ、指定された変数の範囲で読み取りまたは書き込みが行われます。ストリーム・ファイルには、CARRIAGECONTROL= ' NONE ' を指定します。

「ストリーム・ファイル」も参照してください。

Stream_LF レコード型および Stream_CR レコード型

レコードは可変長で、ラインフィード (LF) またはキャリッジ・リターン (CR) がレコード区切り文字となります (Stream_LF ファイルの LF、Stream_CR ファイルの CR)。

Stream_LF ファイルには、埋め込み LF 文字や CARRIAGECONTROL= ' LIST ' を指定することはできません。代わりに、CARRIAGECONTROL= ' NONE ' を指定してください。Stream_CR ファイルには、埋め込み CR 文字を含めることはできません。Stream_LF レコード型は、テキストファイルの通常のレコード型です。

「Stream_LF レコードと Stream_CR レコード」も参照してください。

レコード型の選択

レコード型を選択する前に、アプリケーションで書式なしデータまたは書式付きデータのいずれを使用するかを検討してください。書式付きデータを使用する場合は、セグメント以外のレコード型を選択できます。書式なしデータを使用する場合は、Stream レコード型、Stream_CR レコード型、Stream_LF レコード型は使用しないでください。

セグメント・レコード型は、シーケンシャル・ファイルとともに書式なしシーケンシャル・アクセスに対してのみ使用できます。セグメント・レコードは、インテル® Fortran 以外の言語で記述されたプログラムにより読み取られるファイルに対しては使用できません。

Stream、Stream_CR、Stream_LF、およびセグメントの各レコード型は、シーケンシャル・ファイルでのみ使用できます。

デフォルトのレコード型 (RECORDTYPE) は、OPEN 文に対する ACCESS 指定子および FORM 指定子の値により異なります。

ファイルのレコード型は、ファイル属性としては保持されません。ファイルの作成時に指定したレコード型以外のレコード型を使用した場合、結果は不定です。

I/O レコードは、論理的に関連付けられるフィールド（データ項目）の集合であり、通常は 1 つの単位として処理されます。

ノン・アドバンシング I/O (ADVANCE 指定子) を指定しない限り、各インテル Fortran I/O 文は、最低 1 つのレコードを転送します。

レコード・オーバーヘッド

レコード・オーバーヘッドは、それぞれのレコードに関連付けられたファイル・システムによって内部で使用されるバイトのことです。レコードの読み取り中および書き込み中は使用できません。レコード・オーバーヘッドがわかっていると、アプリケーションの記憶域要件の概算を効率的に行えます。オーバーヘッド・バイトは記憶媒体にあります。OPEN 文で RECL 指定子を使ってレコード長を指定するときには含めないでください。

それぞれのレコードタイプでレコード・オーバーヘッドに必要なバイト数は異なります。これは、次の表で示されています：

レコード型	ファイルの編成	レコード・オーバーヘッド
固定長	シーケンシャル	なし。
固定長	相対	-vms オプションが省略された場合は、0 バイトです（デフォルト）。-vms オプションが指定された場合は、1 バイトです。
可変長	シーケンシャル	各レコードに 8 バイトです。
セグメント	シーケンシャル	各レコードに 4 バイトです。レコードサイズが奇数の場合は、1 つの充填バイト（スペース）が追加されます。
Stream	シーケンシャル	必要なし。
Stream_CR	シーケンシャル	各レコードに 1 バイトです。
Stream_LF	シーケンシャル	各レコードに 1 バイトです。

レコード長

RECL 指定子を使用してレコード長を指定します。

レコード長を指定する単位はデータ形式により異なります。

- 書式付ファイル (FORM = ' FORMATTED ') は、バイト単位でレコード長を指定します。
- 書式なしファイル (FORM = ' UNFORMATTED ') は -assume byterecl を指定して 1 バイト単位を指定しない限り、4 バイト単位でレコード長を指定します。

64 ビットのアドレス可能なシステムでの可変長シーケンシャル・レコード以外では、最大レコード長は 21.4 億バイト (2,147,483,647 からレコード・オーバーヘッドのバイト数を引いた数) になります。64 ビットのアドレス可能なシステムでの可変長シーケンシャル・レコードでは、最大レコード長は、約 17,000 ギガバイトになります。非常に大きなレコードサイズを検討する場合は、システム仮想メモリのような、制限となる要素も考慮してください。

ファイルのアクセスと割り当て

大部分の I/O 操作は、ディスクファイル、キーボード、またはスクリーン・ディスプレイを使用します。その他に次のようなデバイスも使用されます:

- ソケットからの読み出し、またはソケットへの書き込みができます。ただし、ソケットは [USEROPEN ルーチン](#) (通常は C 言語で記述される) を使用して開く必要があります。
- 空のパイプに対して READ を発行すると、読み出しおよび書き込みを操作するために開かれたパイプがブロックされます (データが使用可能になるまで待機します)。
- 空のパイプに対して READ を発行すると、読み出しのみを操作するために開かれたパイプが EOF を返します。

[事前結合ファイル](#)を使用することにより、端末画面やキーボードにアクセスできます。

論理ユニットへのファイルの割り当て

次のいずれかの方法を使用して、論理ユニットへのファイルの割り当て方法を選択することができます:

- 事前結合ユニットなどのデフォルト値を使用する。
- OPEN 文でファイル名を指定する (必要な場合は、ディレクトリも指定する)。
- 環境変数を使用する。

デフォルト値の使用

次の例では、PRINT 文がデフォルトで事前結合ユニット (stdout) と関連付けられています。

```
PRINT *,100
```

次の READ 文は、デフォルトで論理ユニット 7 を `fort.7` ファイルに関連付けます (FILE 指定子が省略されたため)。

```
OPEN (UNIT=7, STATUS='NEW')  
READ (7,100)
```

OPEN 文でファイル名の指定

次に例を示します。

```
OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
```

OPEN 文の FILE 指定子では、通常、ファイル名のみ (`testdata` など) を指定するか、ディレクトリとファイル名の両方 (`/usr/proj/testdata` など) を指定します。

OPEN 文の DEFAULTFILE 指定子では、通常、ディレクトリのみ (`/usr/proj/` など) を含むパス名、またはディレクトリとファイル名の両方 (`/usr/proj/testdata` など) を指定します。

暗黙の OPEN

暗黙の OPEN を実行する場合、FILE および DEFAULTFILE 指定子の値は指定されず、環境変数が使用されます。したがって、暗黙の OPEN を使用した場合、もしくは、OPEN 文の FILE 指定子でファイル名を指定しなかった場合は、環境変数を使用してファイル名またはディレクトリとファイル名の両方を含むパス名を指定できます。

環境変数の使用

シェルコマンドを使用して、適切な環境変数をディレクトリ (必要な場合のみ) とファイル名を示す値に設定することもできます。これによって、ユニットと外部ファイルが関連付けられます。

インテル® Fortran では、各論理 I/O ユニット番号に対する環境変数を `FORTn` (n は論理 I/O ユニット番号) の形式で認識します。ファイル名が OPEN 文で指定されず、対応する `FORTn` 環境変数とそのユニット番号に対して設定されていない場合、`fort.n` (n は論理ユニット番号) の形式でファイル名が生成されます。

暗黙のインテル Fortran 論理ユニット番号

ACCEPT 文、PRINT 文、および TYPE 文を使用する場合と、READ 文および WRITE 文でユニット番号の代わりにアスタリスク (*) を使用する場合には、明示的なユニット番号は指定しません。

これらの各 Fortran 文では、暗黙の内部論理ユニット番号と環境変数が使用されます。デフォルトでは、各環境変数は標準 I/O ファイルと関連付けられている Fortran ファイル名の 1 つと関連付けられます。次の表で、これらの関係を示します:

インテル Fortran 文	-vms が指定された場合の環境変数	-vms が指定されていない場合の環境変数	標準 I/O ファイル名
READ (*,f) iolist	FOR_READ	FORT5	stdin
READ f,iolist	FOR_READ	FORT5	stdin
ACCEPT f,iolist	FOR_ACCEPT	FORT5	stdin
WRITE (*,f) iolist	FOR_PRINT	FORT6	stdout
PRINT f,iolist	FOR_PRINT	FORT6	stdout
TYPE f,iolist	FOR_TYPE	FORT6	stdout

上の表にあるインテル Fortran の環境変数と関連付けられるファイルは変更することができます。変更方法は、他の環境変数の場合と同様で、環境変数割り当てコマンドを使用して行います。次に例を示します。

```
setenv FOR_READ /usr/users/smith/test.dat
```

上記のコマンドを実行すると、アスタリスクを使用した READ 文に対する環境変数が、ディレクトリ /usr/users/smith のファイル test.dat を参照します。

デフォルトのパス名とファイル名

インテル® Fortran は、ファイル指定 (ディレクトリおよびファイル名) をすべてまたは一部を指定して行うことができます (例: /usr/proj/testdata)。次のような指定方法があります。

- OPEN 文の FILE 指定子は、通常ファイル名 (testdata など) のみ、あるいはディレクトリとファイル名の両方 (/usr/proj/testdata など) を指定します。
- OPEN 文の DEFAULTFILE 指定子は、通常ディレクトリ (/usr/proj/ など) のみ、あるいはディレクトリとファイル名の両方を含むパス名 (/usr/proj/testdata など) を指定します。

- **暗黙の OPEN** を使用する場合、または OPEN 文の FILE 指定子がファイル名を指定していない場合、ディレクトリとファイル名の両方を含むファイル名またはパス名を指定する環境変数を使用することができます。

デフォルトのパス名とファイル名の適用例

例えば、ユニット番号 3 の暗黙の OPEN では、インテル Fortran は環境変数 FORT3 をチェックします。環境変数 FORT3 が設定されている場合、その値が使用されます。設定されていない場合、`fort.3` というファイル名を使用します。

次の表では、READ (1,100) 文と同様に現在のディレクトリが `/usr/smith` で、I/O はユニット 1 を使用すると仮定します。

OPEN FILE 値	OPEN DEFAULTFILE 値	FORT1 環境変数値	使用されるパス名
指定なし	指定なし	指定なし	<code>/usr/smith/fort.1</code>
指定なし	指定なし	<code>test.dat</code>	<code>/usr/smith/test.dat</code>
指定なし	チェックなし	<code>/usr/tmp/t.dat</code>	<code>/usr/tmp/t.dat</code>
指定なし	<code>/tmp</code>	指定なし	<code>/tmp/fort.1</code>
指定なし	<code>/tmp</code>	<code>testdata</code>	<code>/tmp/testdata</code>
指定なし	<code>/usr</code>	<code>lib/testdata</code>	<code>/usr/lib/testdata</code>
<code>file.dat</code>	<code>/usr/group</code>	チェックなし	<code>/usr/group/file.dat</code>
<code>/tmp/file.dat</code>	チェックなし	チェックなし	<code>/tmp/file.dat</code>
<code>file.dat</code>	指定なし	チェックなし	<code>/usr/smith/file.dat</code>

結果のパス名がチルド文字 (~) で始まるとき、C シェルススタイルのパス名代入は、最上位のディレクトリ（ルートの下）を使用します。チルドパス名代入の詳細については、`csh(1)` を参照してください。

デフォルトのパス名とファイル名の適用規則

インテル Fortran では、一定の規則に基づいてファイル名とディレクトリ・パスは決定されます。次のようにファイル名の文字列を決定します:

- FILE 指定子が指定されている場合は、その値が使用されます。
- FILE 指定子が指定されていない場合は、インテル Fortran により対応する環境変数が検証されます。対応する環境変数が指定されている場合は、その値が使用されます。対応する環境変数が設定されていない場合は、`fort.n` 形成のファイル名が使用されます。

インテル Fortran により使用されるファイル名文字列が決定されると、次のように（オプションとしてファイル名の前に付く）ディレクトリが決定されます：

- 使用されるファイル名文字列に絶対パス名が含まれる場合は、その値が使用され、DEFAULTFILE 指定子、環境変数、および現在のディレクトリの値は無視されます。
- 使用されるファイル名の文字列に絶対パス名が含まれない場合は、インテル Fortran により DEFAULTFILE 指定子と現在のディレクトリ値が検証されます：対応する環境変数が設定されていて、絶対パス名が指定されている場合はその値が使用されます。それ以外の場合は、DEFAULTFILE 指定子が指定されている、その値が使用されます。DEFAULTFILE 指定子が指定されていない場合は、インテル Fortran は絶対パス名として現在のディレクトリを使用します。

事前結合された標準 I/O ファイルの使用

論理ユニット 5、6、または 0 を開くのに OPEN 文を使用せず、適切な環境変数（FORT n ）を設定していない場合、インテル® Fortran は実行時に（事前結合）ユニット 5、6、または 0 を暗黙的に開き、次に示されているオペレーティング・システム標準 I/O ファイルに関連付けます。

ユニット	環境変数	等価な Linux* 標準 I/O ファイル
5	FORT5	標準入力、stdin
6	FORT6	標準出力、stdout
0	FORT0	標準エラー、stderr

次のいずれかを行って、これらの事前結合ファイルを変更します。

- OPEN 文を使用して、ユニット 5、6、または 0 を開きます。ユニット 5、6、または 0 のファイルを明示的に開く場合、OPEN 文キーワードは、標準の事前結合 I/O ファイルの代わりに使用するファイルに関連した情報を指定します。
- 適切な環境変数（FORT n ）を設定して、I/O を外部ファイルにリダイレクトします。

標準の事前結合ファイルからの入力または出力を実行時にリダイレクトするには、適切な環境変数を設定するかまたはパイプで適切なシェル・リダイレクト文字を使用します（> または < など）。

ファイルを開く: OPEN 文

ファイルを開くには、[事前結合されたファイルを使用](#)（端末出力のためなど）するか、または OPEN 文で明示的に開きます。暗黙的にファイルを開くこともできますが、その場合は、OPEN 文を使用してファイル結合特性やその他の情報を指定することができなくなります。

OPEN 文指定子

OPEN 文により、ユニット番号と外部ファイルが結合されるので、OPEN 文の指定子を使用してファイル属性とランタイム・オプションを明示的に指定することができます。事前結合されたファイルではない限り、一度ファイルを開いたら、再度開く前にファイルを閉じる必要があります。

以前、開いたユニット番号を（一度閉じないで）開くと、次のいずれかが発生します。

- 最初に指定したファイルと異なるファイルを指定すると、インテル® Fortran ランタイム・システムは最初のファイルを閉じてから、新しいファイルを開きます。これにより、2 回目に開いたファイルに対する現在のレコード位置がリセットされます。
- 最初に開いたファイルで指定したファイルと同じファイルを指定すると、そのファイルは再結合されます。このとき、内部的に等価な CLOSE と OPEN は行いません。これにより、レコード位置の前後関係を維持したまま、OPEN 文のランタイム指定子を変更できます。

[INQUIRE 文](#) を使うと、プログラムによりファイルが開かれているかどうかを知ることができます。

特に、OPEN 文を使用して、新しいファイルを作成する場合は、デフォルトを確認するか（『*Intel® Fortran Language Reference*』（英語）マニュアルの OPEN 文の説明を参照してください）、または適切な OPEN 文指定子でファイル属性を明示的に指定してください。

ファイルとユニット情報の指定子

これらの指定子はファイルとユニット情報を識別します。

- UNIT は論理ユニット番号を指定します。
- FILE（または NAME）および DEFAULTFILE は、ディレクトリや外部ファイルの名前を指定します。

- STATUS または TYPE により、新規ファイルの作成、既存ファイルへの上書き、既存ファイルを開く、またはスクラッチ・ファイルを使用するかどうかを示します。
- STATUS または DISPOSE は、CLOSE 後のファイルの存在状態を指定します。

ファイルとレコード特性の指定子

これらの指定子はファイルおよびレコード特性を識別します。

- ORGANIZATION はファイル編成（シーケンシャル・ファイルまたは相対ファイル）を示します。
- RECORDTYPE は、使用するレコード型を示します。
- FORM は、レコードが書式付きか書式なしかを示します。
- CARRIAGECONTROL は、端末制御タイプを示します。
- RECL または RECORDSIZE は、レコードサイズを指定します。

特殊なファイルを開くルーチンの指定子

USEROPEN は、特別なコンテキストを確立するためのファイルを開くルーチンを指定します。これによって、後続のインテル Fortran I/O 文の効果を変更することができます。

ファイルアクセス、処理、および位置の指定子

これらの指定子はファイルのアクセス、処理、および位置を識別します。

- ACCESS はアクセスモード（直接アクセスまたはシーケンシャル・アクセス）を示します。
- SHARED により、他のユーザが同じファイルにアクセスできることを示し、レコードロックをアクティベートにします。インテル Fortran の現在のバージョンでは無視されます。
- POSITION により、ファイルの位置をファイルの先頭、EOF レコードの前、または現在の位置（変更なし）のいずれに配置するかを示します。
- ACTION または READONLY により、ステートメント（文）をレコードの読み出しだけに使用するか、レコードの書き込みだけに使用するか、またはレコードの読み出しと書き込みの両方に使用するかを示します。
- MAXREC により、直接アクセスのための最大レコード数を指定します。
- ASSOCIATEVARIABLE により、直接アクセスのための次のレコード番号を含む変数を指定します。

レコード転送特性の指定子

これらの指定子はレコード転送特性を指定します:

- BLANK により、数値フィールドで空白を無視するかどうかを示します。
- DELIM により、リスト指定出力またはネームリスト出力の中の文字定数に対する区切り文字を指定します。
- PAD は、書式付きレコードを読み出す際に、項目リストと書式仕様で指定したデータよりも、レコードに含まれるデータ量が少ない場合にパディング文字を挿入するかどうかを指定します。
- BLOCKSIZE は、ブロック物理 I/O バッファのサイズを指定します。
- BUFFERCOUNT は、物理 I/O バッファの数を指定します。
- CONVERT は、書式なし数値データの形式を指定します。

エラー処理機能の指定子

これらの指定子はエラー処理に使用されます。

- ERR 指定子では、エラー発生時に分岐するラベルを指定します。
- IOSTAT により、エラーが発生した場合、エラー (IOSTAT) 番号を受け取るための整数変数を指定します。

ファイルを閉じる処理の指定子

DISPOSE は、ファイルが閉じられたときに行う処理を識別します。

OPEN 文でファイルの場所をコード化する

OPEN 文の FILE 指定子および DEFAULTFILE 指定子を使って、論理ユニットで開かれる特定のファイルの完全な定義を指定します。(OPEN 文についての詳細は、『Language Reference』(英語) マニュアルを参照してください。)

次に例を示します。

```
OPEN (UNIT=4, FILE='/usr/users/smith/test.dat',  
      STATUS='OLD')
```

/usr/users/smith ディレクトリの test.dat ファイルが論理ユニット 4 で開かれます。ディレクトリとファイル名が指定されているため、デフォルトは適用されません。FILE 指定子の値は文字定数、変数、または式で表記します。

次の例では、ユーザによりファイル名が指定され、DEFAULTFILE 指定子によりフルのパス名文字列に対するデフォルト値が指定されています。開かれるファイルは /usr/users/smith にあり、ユーザにより入力されたファイル名は変数 DOC に連結されます:

```
CHARACTER (LEN=9) DOC
```

```
WRITE (6,*) 'Type file name '
READ (5,*) DOC
OPEN (UNIT=2, FILE=DOC,
      DEFAULTFILE='/usr/users/smith',STATUS='OLD')
```

スラッシュは、デフォルトのファイル文字列の末尾に追加されます。

ファイル情報の取得: INQUIRE 文

INQUIRE 文は、ファイルに関する情報を返します。INQUIRE 文には、次の 3 つの形式があります:

- ユニットによる照会
- ファイル名による照会
- 出力項目リストによる照会

ユニットによる照会

通常、ユニットによる照会は、開かれた（結合済み）ファイルに対して実行されます。ユニットによる照会によって、インテル® Fortran RTL は、指定されたユニットが結合されているかどうかをチェックします。ユニットの結合状況により、次のいずれかが発生します:

ユニットが結合されている場合:

- EXIST 指定子および OPENED 指定子の変数が TRUE の値を示す
- パス名とファイル名が NAME 指定子変数で返される（ファイルが指定されている場合）
- 以前に結合されたファイルに関して要求された他の情報が返される
- OPEN 文にも関連する INQUIRE 指定子に対して、通常、デフォルトの値が返される
- 結合された書式付きファイルの RECL 値の単位には、常に 1 バイト単位が使用されます。書式なしファイルの場合、-assume byterecl オプションで 1 バイト単位を指定しない限り、RECL には 4 バイト単位が使用されます。

ユニットが結合されていない場合:

- OPENED 指定子は FALSE の値を示す
- ユニット番号 (NUMBER) 指定子変数が、-1 の値として返される
- 返されるその他の情報が、各種指定子の未定義値またはデフォルト値になる

以下の INQUIRE 文の例では次の 3 つが返されます。論理変数 I_OPENED には結合されたファイル (OPENED 指定子) がユニット 3 にあるかどうか、文字変数 I_NAME にはそのファイル名 (大文字・小文字の区別あり) が、文字変数 I_ACTION にはそのファイルが READ、WRITE、または READWRITE 参照で開かれているかどうか、それぞれ返されます。

```
INQUIRE (3, OPENED=I_OPENED, NAME=I_NAME, ACTION=I_ACTION)
```

ファイル名による照会

ファイル名による照会によって、インテル Fortran RTL は、開かれているファイルのリストをスキャンし、一致するファイル名を検索します。一致するファイル名の有無により、次のいずれかが発生します:

ファイル名が一致した場合:

- EXIST 指定子および OPENED 指定子の変数が TRUE の値を示す
- パス名とファイル名が NAME 指定子変数で返される
- UNIT 番号が NUMBER 指定子変数で返される
- 以前に結合されたファイルに関して要求された他の情報が返される
- OPEN 文にも関連する INQUIRE 指定子に対して、通常、デフォルトの値が返される
- 結合された書式付きファイルの RECL 値の単位には、常に 1 バイト単位が使用されます。書式なしファイルの場合、`-assume byterecl` オプションで 1 バイト単位を指定しない限り、RECL には 4 バイト単位が使用されます。

ファイル名が一致しなかった場合:

- OPENED 指定子変数は FALSE の値を示す
- ユニット番号 (NUMBER) 指定子変数が、-1 の値として返される
- EXIST 指定子変数が、指定されたファイル名がデバイスに存在するかどうかを示す (TRUE または FALSE)
- 実際にファイルが存在する場合は、NAME 指定子変数にパス名とファイル名が含まれている
- 返されるその他の情報が、INQUIRE の呼び出し時に指定された情報に基づき、各種指定子のデフォルト値になる

下の INQUIRE 文では次の 3 つが返されます。論理変数 I_OPEN には `log_file` という名前のファイルが結合済みのファイルであるかどうか、論理変数 I_EXIST にはそのファイルが存在するかが、整数変数 I_NUMBER にはユニット番号がそれぞれ返されます。

```
INQUIRE (FILE='log file', OPENED=I OPEN, EXIST=I EXIST,
NUMBER=I _NUMBER)
```

出力項目リストによる照会

ユニットおよび名前による照会とは異なり、出力項目リストによる照会は、外部ファイルにアクセスしません。この照会では、書式なし WRITE 文、READ 文、および REWRITE 文で使用される変数リストに関するレコード長が返されます。以下の INQUIRE 文では、変数リストの最大レコード長を変数 I RECLENGTH に返します。この変数は、その後 OPEN 文で RECL 値を指定するために使用されます。

```
INQUIRE (IOLENGTH=I RECLENGTH) A, B, H
OPEN (FILE='test.dat', FORM='UNFORMATTED', RECL=I RECLENGTH,
UNIT=9)
```

書式なしファイルの場合、-assume byterecl オプションで 1 バイト単位を指定しない限り、IOLENGTH 値は 4 バイト単位として返されます。

ファイルを閉じる: CLOSE 文

通常、開かれた外部ファイルはプログラムの終了前に同じプログラムによって閉じなければなりません。CLOSE 文は、ユニットと外部ファイルの結合を解除します。ユーザは閉じるユニット番号 (UNIT 指定子) を指定する必要があります。

また、次の事項も指定できます。

- そのファイルを削除するかしないか (STATUS 指定子)
- エラー処理情報 (ERR および IOSTAT 指定子)

閉じる時にファイルを削除する方法

- OPEN 文で、ACTION キーワードを指定します (ACTION='READ' など)。READONLY キーワードを使用して開かれたファイルは、閉じる時に削除できないので、READONLY キーワードは使用しません。
- CLOSE 文で、キーワード STATUS='DELETE' を指定します。

外部ファイルを開いて、ユニットによる照会を行った際に、ACCESS 指定子のデフォルト値を使用したくない場合は、そのファイルを閉じた後に、必要な ACCESS 指定子を明示的に指定して、そのファイルを再開することができます。

通常は、事前結合されたユニットを閉じる必要はありません。内部ファイルは、開くことも閉じることも行なわれません。

レコード操作

レコード操作の概要

次のトピックを参照してください。

[レコード I/O 文指定子](#)

[レコードアクセス](#)

[ファイル共有](#)

[開始レコード位置の指定](#)

[アドバンシング・レコード I/O とノン・アドバンシングレコード I/O](#)

[レコード転送](#)

レコード I/O 文指定子

ファイルを開いた後、または事前結合ファイルを使用した後は、次の文を使用できません:

- READ、WRITE、ACCEPT、および PRINT を使用してレコード I/O を実行する。
- BACKSPACE、ENDFILE、および REWIND を使用してファイル内のレコード位置を設定する。
- DELETE、REWRITE、TYPE、および FIND を使用して、さまざまな操作を実行する。

レコード I/O 文では、適切なレコード I/O 書式 (書式付き I/O、リスト指定 I/O、ネームリスト I/O、または書式なし I/O) を使用しなければなりません。

READ レコード I/O 文および WRITE レコード I/O 文では、次の指定子を使用することができます:

- UNIT 指定子では、入力または出力するユニット番号を指定します。
- END 指定子では、ファイル終了マーク (EOF) に到達した時に分岐するラベルを指定します。これは、シーケンシャル・ファイルに対する入力文のみに適用されます。
- ERR 指定子では、エラー発生時に分岐するラベルを指定します。

- IOSTAT 指定子では、エラー発生時にそのエラー番号を格納する整数変数を指定します。
- FMT 指定子では、FORMAT 文のラベルまたは FORMAT を指定する文字データを指定します。
- NML 指定子では、NAMELIST 文の名前を指定します。
- REC 指定子では、直接アクセスするレコード番号を指定します。

ノン・アドバンシング I/O を使用する場合は、ADVANCE、EOR、および SIZE の各指定子を使用します。

REWRITE 文を使用する場合は、UNIT、FMT、ERR および IOSTAT の各指定子を使用できます。

レコードアクセス

レコードアクセスとは、ファイル編成に関係なく、レコードがどのようにファイルから読み取られ、またどのようにファイルへ書き込まれるのかを示す言葉です。レコードアクセスは、ファイルを開くたびに指定され、毎回異なるレコードアクセスが指定される場合もあります。使用可能なレコードアクセスの種類は、ファイル編成とレコード型に依存します。

例えば、レコードアクセスを使用すると次の事柄が可能です:

- ORGANIZATION= ' SEQUENTIAL ' および POSITION= ' APPEND ' (または ACCESS= ' APPEND ') を使用して、シーケンシャル・ファイルにレコードを追加できます。
- 複数の WRITE 文を使用してレコードを順番に追加し、一度ファイルを閉じて、ORGANIZATION= ' SEQUENTIAL ' および ACCESS= ' SEQUENTIAL ' (シーケンシャル・ファイルに固定長のレコードが含まれている場合は、ACCESS= ' DIRECT ') を使用して、再びファイルを開くことができます。

シーケンシャル・アクセス

シーケンシャル・アクセスは、ファイルおよびターミナルなどの I/O デバイスへ、レコードを順番に読み書きします。シーケンシャル I/O は、サポートされたファイル編成とレコード型ならどのような種類でも使用できます。

シーケンシャル編成または相対編成を持つファイルに対して、シーケンシャル・アクセス・モードを指定した場合、レコードは、ファイルの先頭から 1 つずつ読み書きされます。特定のレコードの読み取りは、その前にあるすべてのレコードが読み出された後に行われます。また、新規のレコードは、ファイルの終わりにしか書き込めません。

直接アクセス

直接アクセスは、レコード番号で指定したレコードを、ディスク上にある固定長のレコード型を含むシーケンシャル・ファイルまたは相対ファイルへ読み書きします。

直接アクセスモードを選択すると、レコードを読み書きする順番を決定できます。各 READ 文または WRITE 文には、読み取りまたは書き込みを行うレコードの相対レコード番号が含まれていなければなりません。

シーケンシャル・ディスク・ファイルへの直接的なアクセスは、固定長のレコードが含まれている場合のみ可能です。直接アクセスは、セル番号を使用してレコードを検索するため、以前要求したレコードよりも前または後ろにあるレコードを要求するために、READ 文または WRITE 文を連続的に使用できます。例えば、以下のサンプルコードでは、最初の READ 文でレコード 24 を読み取り、2 番目の READ 文でレコード 10 を読み取っています：

```
READ (12,REC=24) I
READ (12,REC=10) J
```

ファイル編成とレコード型によるレコードアクセスの制限

シーケンシャル・ファイルおよび相対ファイルに対して、両方のアクセスモードを使用することができます。ただし、シーケンシャル編成ファイルへの直接アクセスは、ファイルがディスク上に格納されていて、さらに固定長のレコードを含む場合のみ可能です。

ファイル編成とレコード型の組み合わせで使用可能なアクセスの種類を、以下の表で要約します。

レコード型	シーケンシャル・アクセス	直接アクセス
シーケンシャル・ファイル編成		
固定長	Yes	Yes
可変長	Yes	No
セグメント	Yes	No
ストリーム	Yes	No
Stream_CR	Yes	No
Stream_LF	Yes	No
相対ファイル編成		
固定長	Yes	Yes



注

直接アクセスを行うには、ファイルがディスクデバイスに格納されている必要があります。また、相対ファイルも、アクセスモードの種類に関係なく、ディスクデバイスに格納されている必要があります。

ファイル共有

ユーザのプログラムは、OPEN 文の ACTION (または READONLY) 指示子によって指定された値に従って、レコードの読み取り、書き込み、または読み取りと書き込みの両方を行うために、ファイルを開きます。これにより、そのプログラム自体が指定されたタイプのステートメントを実行することがチェックされます。

パフォーマンス上の理由により、レコードロックおよび共有ファイルのチェックは、インテル® Fortran ではサポートされていません。ファイルを開いている時には、次の場合であってもアクセスは常に認可されます。

- OPEN 文に SHARED 指示子が指定された
- 他のプロセスがすでにそのファイルを開いている

レコードの書き込み（またはレコードの読み取りと書き込み）のためにファイルを開くと、別のプロセスが同時にそのファイルを開いて、レコードに書き込んでいることがわかる場合があります。この場合、各プロセス間でアクセス時間を調整して、同じレコード位置で WRITE 文と REWRITE 文が同時に発生する可能性に対処する必要があります。

開始レコード位置の指定

ディスクファイルを開くときに OPEN 文の POSITION 指定子を使用することで、ファイル内の開始レコード位置として、次のいずれかを指定することができます：

- 第 1 レコードの前の初期位置 (POSITION='REWIND')。シーケンシャル・アクセスの READ 文または WRITE 文は、ファイルの第 1 レコードを読み取りまたは書き込みます。
- ファイル内の最終レコードの後ろ (POSITION='APPEND')。つまり EOF レコードがある場合は、その直前に位置します。新規ファイルでは、この位置が第 1 レコードの前の初期位置になります ('REWIND' と同様)。既存のシーケンシャル・ファイルに対して、シーケンシャル・アクセスを使用する場合、レコードを書き込む前に、'APPEND' を指定することができます。
 - 現在の位置 (POSITION='ASIS')。これは通常、ファイルを再結合する時に、現在のレコード位置を維持するためのみに使用されます。2 回目の

OPEN で、同じユニット番号および同じファイル名を指定することで（ファイル名は省略可）、現在のレコード位置を維持したままで、そのファイルを再び開くことができます。ただし、2 回目の OPEN で同じユニット番号に対して異なるファイル名を指定すると、現在の開いているファイルが閉じられ、指定したファイルが開きます。

次の I/O 文を使用すると、現在のレコード位置を変更することができます。

- REWIND では、レコード位置が第 1 レコードの前の初期位置に設定されます。シーケンシャル・アクセスの READ 文または WRITE 文では、ファイルの第 1 レコードを読み取りまたは書き込みます。
- BACKSPACE では、レコード位置がファイル内の 1 つ前のレコードに設定されます。シーケンシャル・アクセスを使用してレコード 5 を書き込み、そのユニットに対して BACKSPACE を発行した後に、ユニットから読み取りを行なうと、レコード 5 が読み込まれます。
- ENDFILE では、EOF マークが書き込まれます。通常、これは、シーケンシャル・アクセスを使用してレコードを書き込んだ後で、ファイルを閉じる直前に実行します。

ノン・アドバンシング I/O を使用しない限り、通常、レコードの読み取りおよび書き込みでは、現在のレコード位置が 1 レコード単位で前に進みます。単一のレコード I/O 文を使用して、複数のレコードを転送することもできます。

アドバンシング・レコード I/O とノン・アドバンシングレコード I/O

ファイルを開いた後に、READ 文または WRITE 文で ADVANCE 指定子を省略すると（または ADVANCE= 'YES' を指定すると）、アドバンシング I/O（通常の FORTRAN I/O）がレコードアクセスに使用されます。アドバンシング I/O を使用すると次の事柄が行われます:

- レコード I/O 文は、レコード全体（または複数のレコード）を転送します。
- レコード I/O 文は、現在のレコード位置を、次のレコードの位置まで進めます。

READ 文および WRITE 文で ADVANCE= ' NO ' 指定子を指定することで、特定のファイルに対してノン・アドバンシング I/O を要求できます。ノン・アドバンシング I/O は、(リスト指定 I/O およびネームリスト I/O ではない) 書式なし I/O を使用した外部ファイルへのシーケンシャル・アクセスでのみ使用可能です。

ノン・アドバンシング I/O を使用しても、現在のレコード位置は変更されません。また、常にレコード全体を転送するアドバンシング I/O とは違って、レコードの一部を転送することができます。

READ および WRITE レコード I/O 文の ADVANCE 指定子に異なる値（'YES' および 'NO'）を指定することで、アドバンシング I/O とノン・アドバンシング I/O のどちらを使うかを選択できます。

アドバンシング I/O またはノン・アドバンシング I/O のいずれかを使用してレコードを読み取る際に、END 指定子を指定しておくことで、ファイルが終わりまで読み取られたときに、特定のラベルへ分岐することができます。

ノン・アドバンシング I/O はレコード全体を読み取らない場合があるので、レコードが終わりまで読まれたときに特定のラベルへ分岐するさせるための、EOR 指定子もサポートされています。ノン・アドバンシング I/O を使用する際に EOR および IOSTAT 指定子を省略すると、レコードの終わり (end-of-record) が読み取られたときにエラーが発生します。

ノン・アドバンシング入力時に SIZE 指定子を指定することで、読み取られた文字数を返すことができます。例えば、次の READ 文では、SIZE=X (変数 X は整数) が、読み取られた文字数を X に返します。また、レコードが終わりまで読まれた場合は、レコード終了条件に従って、ラベル 700 へ分岐されます：

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700)
    A, F, I
```

レコード転送

I/O 文は、すべてのデータをレコードとして転送します。1 つのレコードに含めることができるデータ量は、次の条件によって異なります：

- 書式付き I/O (固定長レコードを除く) の場合、I/O 文の項目数および I/O 文に関連した形式指定子の組み合わせによって、転送されるデータ量が決定されます。
- ネームリストおよびリスト指定出力の場合、(NAMELIST またはリストによって指定された形式規則とともに) NAMELIST 文または I/O 文のリストで指定された項目を使用し、転送されるデータ量が決定されます。
- 書式なし I/O (固定長レコードを除く) の場合、転送されるデータ量は I/O 文だけで指定されます。

- 固定長レコード (RECORDTYPE='FIXED') を指定すると、すべてのレコードが同じサイズになります。書き込まれる I/O レコードのサイズがレコード長 (RECL) よりも小さい場合は、余分なバイトが追加されます (パディング)。

通常、I/O 文によって転送されるデータは、単一レコードから読み出されたり、単一レコードに書き込まれます。ただし、使用される I/O 形式によっては、単一の I/O 文でも、複数のレコードへのデータ転送が行えます。

レコード転送: 入力

アドバンシング I/O を使用するとき、入力文でレコードに含まれるデータ量よりも小さいデータ・フィールド (データ量) を指定すると、残りのデータ・フィールドは無視されます。

入力文でレコードに含まれるデータ量よりも大きいデータ・フィールドを指定すると、次のいずれかが発生します：

- アドバンシング I/O を使用する書式付き入力では、PAD='YES' を使用してファイルが開かれると、余分なフィールドはスペースとして読み出されます。PAD='NO' を使用してファイルが開かれると、エラーが発生します (入力文では、レコードに含まれるデータ量を超えるデータ・フィールドを指定してはいけません)。
- ノン・アドバンシング I/O (ADVANCE='NO') を使用する書式付き入力では、レコードの終了 (EOR) が返されます。PAD='YES' を使用してファイルが開かれると、余分なフィールドがスペースとして読み出されます。
- リスト指定入力では、別のレコードが読み出されます。
- ネームリスト入力では、別のレコードが読み出されます。
- 書式なし入力では、エラーが発生します。

レコード転送: 出力

出力文で、レコードに含まれるデータ量よりも小さいデータ・フィールド (レコードを埋めるために必要なデータ量よりも少ないデータ量) が指定されると、次のように動作します：

- 固定長レコード (RECORDTYPE='FIXED') では、すべてのレコードのサイズが同一になります。書き込まれる I/O レコードのサイズがレコード長 (RECL) よりも小さい場合、スペース (書式付きレコードの場合) またはゼロ (書式なしレコードの場合) を使って余分なバイトが追加されます (パディング)。
- その他の種類のレコードの場合、フィールドへの書き込みは行われますが、省略された部分には書き込まれません (結果として、レコードが短くなる場合があります)。

出力文で、レコードに含めることができる最大データ量を超えるデータが指定されると、次のようなエラーが発生します:

- 固定長レコードを使用する書式付きまたは書式なし出力の場合、出力文の項目とそれに関連した形式指定子によってバイト数が最大レコード長 (RECL) を超えると、エラーが発生します。
- 固定長レコードを使用しない書式付きまたは書式なし出力の場合、出力文の項目とそれに関連した形式指定子によってバイト数が最大レコード長 (RECL) を超えると、インテル® Fortran RTL は、RECL の値を増やし、より大きなレコードを書き込みます。RECL の値を取得するには、INQUIRE 文を使用します。
- リスト指定出力およびネームリスト出力では、指定されたデータが最大レコード長 (RECL) を超えると、別のレコードが書き込まれます。

ユーザが提供する OPEN プロシージャ: USEROPEN 指定子

インテル® Fortran の OPEN 文で USEROPEN 指定子を使用して、ファイルを直接開くルーチンに制御を渡すことができます。呼び出されたルーチンでは、システムコールまたはライブラリ・ルーチンを使用することで、ファイルを開き、後続のインテル Fortran I/O 文の効果を変更する特別なコンテキストを確立することができます。

インテル Fortran RTL I/O サポートルーチンでは、I/O のためにファイルが最初に関数に開かれたときに通常使用されるシステムコールの代わりに、USEROPEN 関数を呼び出します。OPEN 文の USEROPEN 指定子では、制御を受け取る関数の名前を指定します。呼び出された関数は、ファイル（またはパイプ）を開いて、RTL に制御を戻す際にそのファイルのファイル記述子を返す必要があります。

ファイルを開く際に呼び出された関数では、通常、標準の OPEN 文で指定されるものとは異なるオプションを指定します。

getfd ルーチンを使用して、インテル Fortran RTL から特定のユニット番号のファイル記述子を取得することができます。

呼び出された関数は、C 以外の言語 (Fortran など) で記述することができますが、通常、open または create などのシステムコールの作成には、C 言語が最適です。

USEROPEN 指定子の構文および動作

OPEN 文の USEROPEN 指定子の形式は次のとおりです。

USEROPEN = *function-name*

function-name の値は外部関数の名前を示します。呼び出すプログラム側では、この関数は EXTERNAL 文で宣言する必要があります。例えば、次のインテル Fortran コードを使用して、UOPEN という名前の USEROPEN プロシージャ (リンカでは uopen_ として認識されます) を呼び出します。

```
EXTERNAL    UOPEN
INTEGER     UOPEN
.
.
.
OPEN (UNIT=10, FILE='/usr/test/data', STATUS='NEW',
      USEROPEN=UOPEN)
```

OPEN 文が実行されると、uopen_ 関数は制御を受け取ります。関数はファイルを開き、指定された操作を実行し、そして RTL に制御 (およびファイル記述子) を返します。

USEROPEN 関数が C で記述されている場合、ファイル記述子を格納するために、この関数を 4 バイト整数 (int) の結果を返す C の関数として宣言します。次に例を示します。

```
int    uopen_ (                                (1)
    char    *file_name,                        (2)
    int     *open_flags,                       (3)
    int     *create_mode,                     (4)
    int     *lun,                             (5)
    int     file_length);                     (6)
```

次に、関数の定義およびインテル Fortran RTL から渡される引数を示します。

1. 関数は 4 バイト整数 (int) として宣言する必要があります。
2. 第 1 引数は、開かれるパス名 (ファイル名を含む) です。
3. オープンフラグについては、/usr/include/sys/file.h ヘッダファイルまたは open(2) で説明されています。
4. 作成モード (ファイルの作成時に必要な保護) については、open(2) で説明されています。
5. 第 4 引数は 論理ユニット番号です。
6. 最後の第 5 引数は、パス名の長さ (パス名の長さの隠し引数) です。

open システムコール (open(2) を参照) では、渡されたパス名、オープンフラグ (必要なアクセスタイプやファイルが存在するかどうかなどを定義)、および作成モードが必要です。OPEN 文で指定された論理ユニット番号は、USEROPEN 関数で必要となる場合に備えて渡されます。また、パス名の長さの隠し引数も渡されます。

新しいファイルを作成する場合、open システムコールの代わりに create システムコールを使用することができます (create(2) を参照)。通常は、USEROPEN 関数内でその他の適切なシステムコールまたはライブラリ・ルーチンを使用することができます。

ほとんどの場合、USEROPEN 関数はインテル Fortran RTL によって渡されたオープンフラグ引数を変更するか、open (または create) システムコールの前に新しい値を使用します。関数は、ファイルを開いた後で RTL に制御を返す必要があります。

USEROPEN 関数が Fortran で記述されている場合、その関数を INTEGER (KIND=4) の結果 (場合によっては、インターフェイス・ブロック) を返す FUNCTION として宣言します。呼び出された関数は、RTL に 4 バイト整数のファイル記述子を返す必要があります。

C 言語を使用してファイルの開閉、およびすべてのレコード操作を行う必要があるアプリケーションでは、Fortran の OPEN 文を使用せずに、インテル Fortran プログラムから適切な C プロシージャを呼び出します。

呼び出された USEROPEN 関数の制限事項

インテル Fortran RTL は、1 つの論理ユニットごとにファイル記述子を 1 つだけ使用します。ファイル記述子は、呼び出された関数によって返される必要があります。このため、ファイルを開く際に使用できるのは、特定のシステムコールまたはライブラリ・ルーチンだけです。

ファイル記述子を返さないシステムコールおよびライブラリ・ルーチンには、mknod (mknod(2) を参照) および fopen (fopen(3) を参照) があります。例えば、fopen ルーチンはファイル指定子の代わりにファイルポインタを返します。

USEROPEN プログラムおよび関数の例

次のインテル Fortran コードは、UOPEN という名前の USEROPEN 関数を呼び出します。

```
EXTERNAL    UOPEN
INTEGER     UOPEN

.
.
.

OPEN (UNIT=1, FILE='ex1.dat', STATUS='NEW', USEROPEN=UOPEN,
      ERR=9, IOSTAT=errnum)
```


デフォルトの ifort オプションが使用される場合、外部名には小文字が使用され、その名前の最後に下線 (_) を付加して渡されます。上記の例では、外部関数 UOPEN は、リンクは uopen_ として認識し、C の中では uopen_ として宣言する必要があります。

C およびインテル Fortran プログラムのコンパイルとリンク

icc コマンドを使用して、呼び出される C の uopen_ 関数 (uopen_.c) をコンパイルします。また、ifort コマンドを使用して、インテル Fortran の呼び出しプログラム (ex1.f) をコンパイルします。この ifort コマンドはまた、次のように、適切なライブラリを使用することで、両方のオブジェクト・ファイルをリンクし、a.out ファイルを作成します。

```
icc -c uopen_.c
ifort ex1.f uopen_.c
```

C 関数およびヘッダファイル用のソースコード

次に、uopen_ という名前の C 言語の関数および関連するヘッダファイルを示します。

```
/*

** File: uopen.h -- header file for uopen_.c
**/

#ifndef UOPEN
#define UOPEN 1
/*
**      Function Prototypes
**
*/
int  uopen_ (
    char  *file_name,      /* access read: name of the file
to open. */
    int   *open_flags,     /* access read: READ/WRITE, see
file.h or open(2) */
    int   *create_mode,    /* access read: set if new file
(to be created). */
    int   *lun,            /* access read: logical unit file
opened on. */
    int   file_length);    /* access read: number of
characters in file_name */

#endif

/* End of file uopen.h */

/*
```



```

** File: uopen_.c
*/

/*
** This routine opens a file using data passed by Intel
Fortran RTL.
**
**  INCLUDE FILES
**/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include "uopen.h"/* Include file for this module */

int uopen_ (file_name, open_flags, create_mode, lun,
file_length)

/*
** Open a file using the parameters passed by the calling
Intel
** Fortran program.
**
** Formal Parameters:
**/

char  *file_name;      /* access read: name of the file to
open. */
int   *open_flags;     /* access read: READ/WRITE, see file.h
*/
int   *create_mode;    /* access read: set if new file (to be
created). */
int   *lun;            /* access read: logical unit number
file opened on. */
int   file_length;     /* access read: number of characters
in file_name. */

/*
** Function Value/Completion Code
**
** Whatever is returned by open is immediately returned to
the
** Fortran OPEN.  The returned value is the following:
**   value >= 0 is a valid fd.
**   value < 0 is an error.
**
** Modify open flags (logical OR) to specify the file be
opened for
** write access only, with records appended at the end
(such as
** writing to a shared log file).
**/

{

```

```

        int      result ;                /* Function result value */

        *open_flags =
            O_CREAT |
            O_WRONLY |
            O_APPEND;

        result = open (file_name, *open_flags, *create_mode) ;

        return (result) ;                /* return file descriptor
or error */

    }/* End of routine uopen_ */

/* End of file uopen_.c */

```

インテル Fortran プログラムを呼び出すソースコード

次に、C の `uopen_` 関数を呼び出して、その後 I/O を実行する Fortran プログラムを示します。

```

!
!  Program EX1 opens a file using USEROPEN and writes
!  records to it.
!  It closes and re-opens the file (without USEROPEN) and
!  reads 10 records.
PROGRAM EX1

        EXTERNAL      UOPEN              ! The USEROPEN function.
        INTEGER      ERRNUM, CTR, I

1   FORMAT (I)
        ERRNUM = 0
        WRITE (6,*) 'EX1. Access data using formatted I/O.'
        WRITE (6,*) 'EX1. Open file with USEROPEN and put some
data in it.'

        OPEN (UNIT=1, FILE='ex1.dat', STATUS='NEW',
USEROPEN=UOPEN, ERR=9, IOSTAT=errnum)
        DO CTR=1,10
            WRITE (1,1) CTR
        END DO
        WRITE (6,*) 'EX1. Close and re-open without USEROPEN.'
        CLOSE (UNIT=1)
        OPEN (UNIT=1, FILE='ex1.dat', STATUS='OLD',
FORM='FORMATTED', ERR=99, IOSTAT=errnum)
        WRITE (6,*) 'EX1. Read and display what is in file.'
        DO CTR=1,10
            READ (1,1) i
            WRITE (6,*) i
        END DO
        WRITE (6,*) 'EX1. Successful if 10 records shown.'
        CLOSE (UNIT=1,STATUS='DELETE')

```

```

      STOP
    9 WRITE (6,*) 'EX1.  Error on USEROPEN is ', errnum
      STOP
    99 WRITE (6,*) 'EX1.  Error on 2nd open is ', errnum
END PROGRAM EX1

```

レコード型の形式

固定長レコード

固定長レコードを指定すると、そのファイルの全レコードのバイト数が同一になります。固定長レコードを格納するファイルを開くときには、RECL 指定子を使用し、レコード長を指定する必要があります。シーケンシャル編成ファイルを直接アクセスするために開く場合には、ファイルのレコード位置を正確に計算できるように、固定長レコードを格納することが必要です。

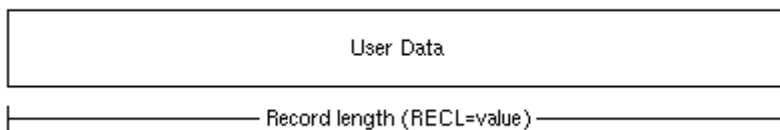
相対ファイルの場合、固定長レコードの構成とオーバーヘッドは、-vms オプションを使用してコンパイルされたか、-vms オプションが省略されたかによって異なります。

-vms オプションが省略された相対ファイルでは（デフォルト）、各レコードには制御情報は含まれません。

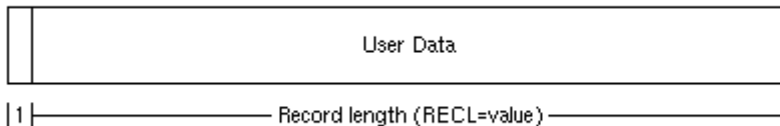
-vms オプションが指定された相対ファイルでは、各レコードの先頭に 1 バイトの制御情報が含まれます。

次の図は、固定長レコードのレコード構成です：

For all sequential files and for relative files where the -vms option was omitted:



For relative files where the -vms option was specified:



ZK-9819-GE

可変長レコード

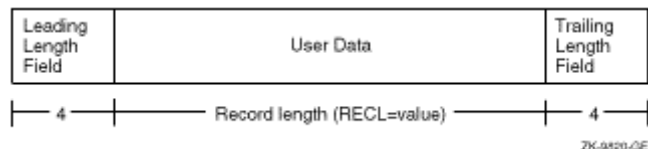
可変長レコードは、指定された最大レコード長まで任意のバイト数を含むことができ、シーケンシャル・ファイルにのみ適用されます。

可変長レコードは、長さフィールドを含む 4 バイトの制御情報が前後に追加されます。終端長さフィールドでは、BACKSPACE で効率的にレコードを戻ることができます。各長さフィールドに格納された 4 バイトの整数値は、その特定の可変長レコード中のデータバイト（オーバーヘッド・バイトを除く）の数を示します。

可変長レコードの文字カウント・フィールドは、Q 書式記述子を指定して READ 文を発行することでレコードの読み取り時に知ることができます。その後、このカウント・フィールドを使用し、I/O リストに何バイトが含まれているかを決定することができます。

2 GB 未満の可変長レコード

次の図は、2 GB 未満の可変長レコードのレコード構成です：



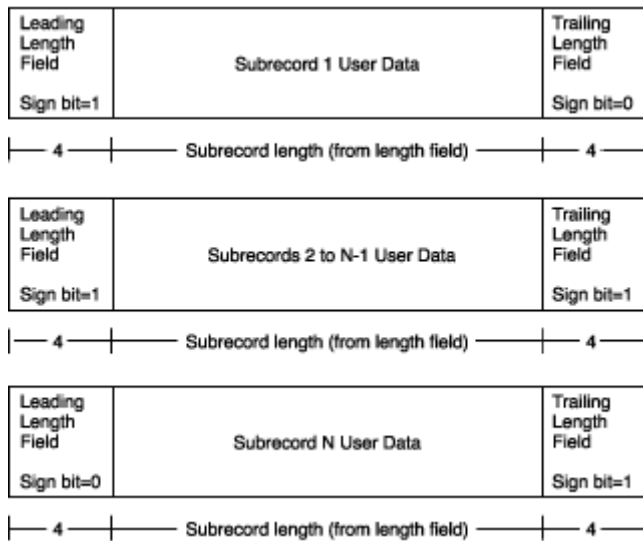
2 GB を超える可変長レコード

2,147,483,639 バイトよりも大きなレコード長では、レコードはサブレコードに分けられます。サブレコードの長さは、1 から 2,147,483,639 までのいずれかです。

先頭長さフィールドの符号ビットは、後続のレコードがあるかないかを示します。終端長さフィールドの符号ビットは、前にサブレコードがあることを示します。符号ビットの位置は、ファイルのエンディアン形式により決定されます。

後続のサブレコードには、符号ビット値が 1 の先頭長さフィールドがあります。レコードを構成する最後のサブレコードには、符号値 0 の先頭長さフィールドがあります。前にサブレコードを持つサブレコードには、符号ビット 1 の終端長さフィールドがあります。レコードを構成する最初のサブレコードには、符号値 0 の終端長さフィールドがあります。符号ビットの値が 1 の場合、レコード長は 2 の補数表現で格納されます。

次の図は、2 GB を越す可変長レコードのレコード構成です：



通常、インテル® Fortran プログラムが可変長レコードを使用して書き出したファイルにはテキストファイルとしてアクセスすることはできません。可変長レコードを含むテキストファイルを出力する場合、代わりに Stream_LF レコード形式を使用してください。

セグメント・レコード

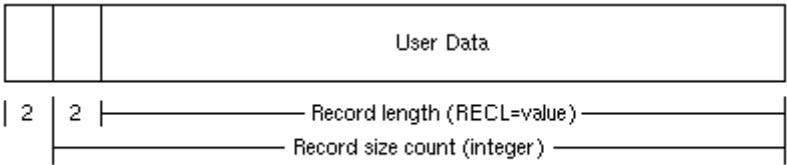
セグメント・レコードは、1 つのシーケンシャル編成ディスクファイル内に、1 つ以上の可変長書式なしレコードから構成される単一の論理レコードです。シーケンシャル・アクセスを使用してシーケンシャル編成ファイルに書き込まれた書式なしデータは、デフォルトではセグメント・レコードとして記憶されます。

セグメント・レコードは、例外的に長いレコードを書き込む必要がある場合で、1 つの長い可変長レコードを定義できない、または定義したくない場合に便利です。例えば、仮想メモリの制限によってプログラムが実行できないような場合です。より小さいセグメント・レコードを使用すれば、プログラムを実行するシステム上の仮想メモリの制限により問題が発生する可能性が減少します。

ディスクファイルでは、セグメント・レコードはセグメントで構成される単一の論理レコードです。各セグメントは、1 つの物理レコードです。セグメント（論理）レコードは、絶対最大レコード長（21.4 億バイト）を超えることができますが、各セグメント（物理レコード）は、最大レコード長を超えることができません。

セグメント・レコードを含む書式なしシーケンシャル・ファイルにアクセスするには、ファイルを開く際に `FORM='UNFORMATTED'` および `RECORDTYPE='SEGMENTED'` を指定します。

次の図のように、セグメント・レコードの構成は、4 バイトの制御情報と、その後のユーザデータで構成されます。



ZK-9821-GE

制御情報は、2 バイト整数のレコード・サイズ・カウント（セグメント識別子に使用される 2 バイトを含む）と、その後の 2 バイトの整数のセグメント識別子で構成されます。セグメント識別子は、このセグメントが次のいずれであるかを示します：

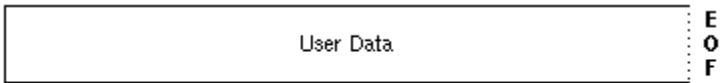
識別子の値	指定セグメント
0	最初と最後のセグメントの間にあるいずれかのセグメント
1	第 1 セグメント
2	最終セグメント
3	単独セグメント

指定されたレコード長が奇数の場合、ユーザデータには空白が 1 つ（1 バイト）埋め込まれますが、この余分なバイトは 2 バイト整数のレコード・サイズ・カウントには加算されません。

ストリーム・ファイル

ストリーム・ファイルは、レコードにグループ化されず、制御情報は含まれません。ストリーム・ファイル、CARRIAGECONTROL='NONE' を指定して使用され、入力文または出力文で指定された変数の範囲でのみ、読み出しされるまたは書き込まれる文字、あるいは 2 進数データを格納します。

次の図は、ストリーム・ファイルの構成です。



ZK-9822-GE

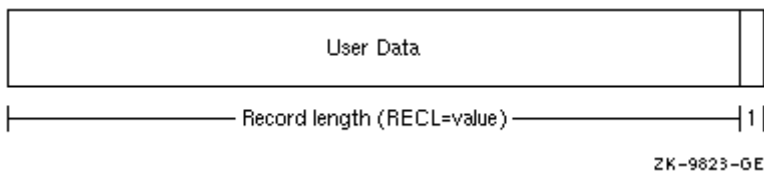
Stream_CR レコードと Stream_LF レコード

Stream_CR および Stream_LF レコードは、可変長レコードです。レコード長は、カウントによってではなく、データに埋め込まれた明示的なレコード区切り文字によって示されます。これらの区切り文字は、ストリーム形式のファイルにレコードを書き出したときに自動的に追加され、レコードを読み取ったときに削除されます。

Stream_CR と Stream_LF では、1 バイトの異なるレコード区切り文字を使用します：

- ストリーム CR ファイルキャリッジ・リターンのみを区切り文字として使用します。したがって、Stream_CR ファイルにはキャリッジ・リターン文字を含めてはなりません。
- Stream_LF ファイルでは、ラインフィードのみをレコード区切り文字として使用します。したがって、Stream_LF にはラインフィード文字を含めてはなりません。Stream_LF レコードは、オペレーティング・システムの通常のテキストファイルのレコード型です。

次の図は、Stream_CR レコードおよび Stream_LF レコードの構成です:



Microsoft* Fortran PowerStation* 互換ファイル

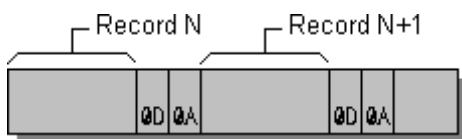
Microsoft* Fortran PowerStation* との互換性のために、[-fpscomp オプション](#)を使用する際には、次の形式のファイルが使用できます:

- 書式付きシーケンシャル・ファイル
- 書式付き直接ファイル
- 書式なしシーケンシャル・ファイル
- 書式なし直接ファイル

書式付きシーケンシャル・ファイル

書式付きシーケンシャル・ファイルは、ファイル内の順序で、順に書き出され、また順に読み取られる一連の書式付きレコードです。レコードの長さはさまざまで、また空の場合もあります。次の図のように、レコードはキャリッジ・リターン (0D) 文字やラインフィード (0A) 文字で区切られます。

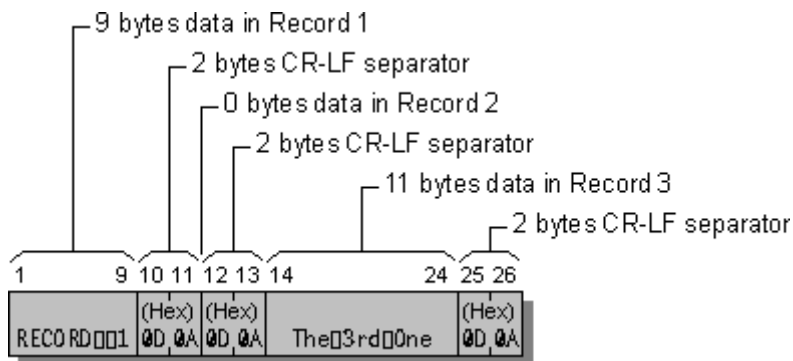
書式付きシーケンシャル・ファイル 内の書式付きレコード



次に、書式付きシーケンシャル・ファイルに 3 つのレコードを書き出すプログラムの例を示します。また、続く図は結果として得られるファイルを示します。

```
OPEN (3, FILE='FSEQ')
! FSEQ is a formatted sequential file by default.
WRITE (3, ' (A, I3)') 'RECORD', 1
WRITE (3, ' ()')
WRITE (3, ' (A11)') 'The 3rd One'
CLOSE (3)
END
```

書式付きシーケンシャル・ファイル



書式付き直接ファイル

書式付き直接ファイルでは、すべてのレコードは同じ長さで、任意の順序で読み取りや書き込みを行なうことができます。レコードのサイズは、OPEN 文の RECL オプションで指定され、最も長いレコードのバイト数と等しいか、またはそれ以上でなくてはなりません。

キャリッジ・リターン (CR) 文字およびラインフィード (LF) 文字は、レコードの区切り子で、RECL 値には含まれません。直接アクセスのレコードがいったん書き出されると、削除はできませんが、書き換えは可能です。

書式付き直接ファイルへの出力中、レコードがデータで満たなかった場合には、コンパイラはレコードの残りの部分に空白を挿入します。この空白の挿入により、ファイルにはすべて同じ長さの、完全に満たされたレコードだけが含まれることになります。また、入力時に、レコードに含まれるデータよりも多くのデータが入力リストと書式に必要な場合、コンパイラはデフォルトで入力レコードに空白を追加します。

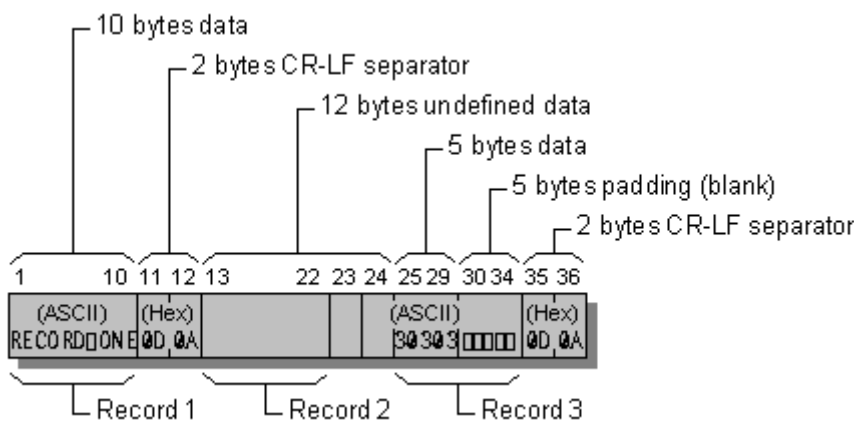
ファイルの OPEN 文で PAD='NO' を設定すると、デフォルトの空白の挿入設定を上書きすることができます。PAD='NO' と設定されている場合、入力レコードには入力リスト

と書式の仕様で指定されているデータ量が含まれなければなりません。データ量が異なる場合には、エラーとなります。PAD='NO' の設定は、出力には影響しません。

次の例は、レコード 1 とレコード 3 の 2 つのレコードを書式付き直接ファイルに書き出すプログラム例です。また、続く図はその結果です。

```
OPEN (3, FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT', RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

書式付き直接ファイル



書式なしシーケンシャル・ファイル

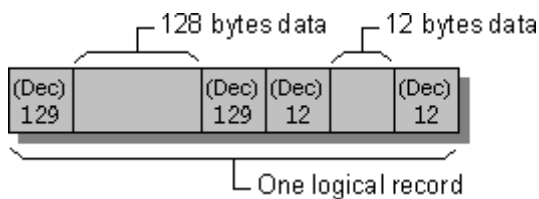
書式なしシーケンシャル・ファイルは、プラットフォームにより、編成が少し異なります。このセクションでは、-fpscomp オプション (-fpscomp ioformat など) が指定された場合に、インテル® Fortran コンパイラにより作成される書式なしシーケンシャル・ファイルについて説明します。異なるファイル編成を行うプラットフォームからファイルにアクセスする場合は、[「書式なしデータの変換の概要」](#)を参照してください。

書式なしシーケンシャル・ファイル内のレコードは長さが異なります。書式なしシーケンシャル・ファイルは、130 バイト以下の物理ブロックと呼ばれる単位として編成されています。個々の物理ブロックは、ファイルに送信された (128 バイトまでの) データとコンパイラが挿入する 2 つの 1 バイトの “長さバイト” から構成されます。長さバイトは、各レコードの開始位置と終了位置を示します。

論理レコード とは、1 つまたは複数の物理ブロックを含む書式なしレコードを示します。(次の図を参照) 論理レコードの大きさは任意です。コンパイラは、必要な数の物理ブロックを使用します。

複数の物理ブロックから構成される論理レコードを作成する際、コンパイラは長さバイトを 129 に設定して、現在の物理ブロックのデータが次の物理ブロックに継続されることを示します。例えば、140 バイトのデータを書き出すと、論理レコードの構造は次の図のようになります。

書式なしシーケンシャル・ファイルの論理レコード

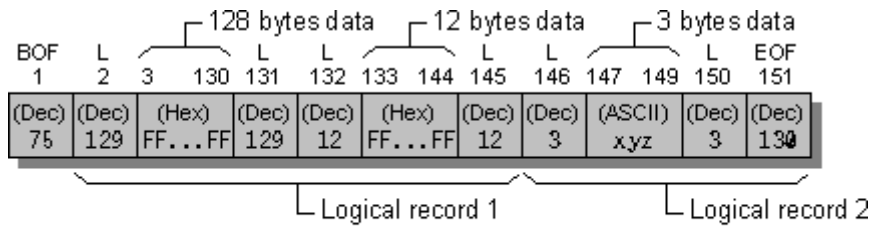


書式なしシーケンシャル・ファイルの最初と最後のバイトは予約されています。最初のバイトには値 75 を、最後のバイトには値 130 が含まれています。Fortran は、これらのバイトをエラーチェックとファイル終了の参照に使用します。

次のプログラムは、続く図で示される書式なしシーケンシャル・ファイルを作成します。

```
! Note: The file is sequential by default
!       -1 is FF FF FF FF hexadecimal.
!
  CHARACTER xyz(3)
  INTEGER(4) idata(35)
  DATA      idata /35 * -1/, xyz /' x', ' y', ' z' /
!
! Open the file and write out a 140-byte record:
! 128 bytes (block) + 12 bytes = 140 for IDATA, then 3 bytes for XYZ.
  OPEN (3, FILE='UFSEQ', FORM='UNFORMATTED')
  WRITE (3) idata
  WRITE (3) xyz
  CLOSE (3)
  END
```

書式なしシーケンシャル・ファイル



BOF Beginning-of-file byte (75 decimal)
 L Physical-block-length byte (0 ≤ L ≤ 129)
 EOF End-of-file byte (130 decimal)

書式なし直接ファイル

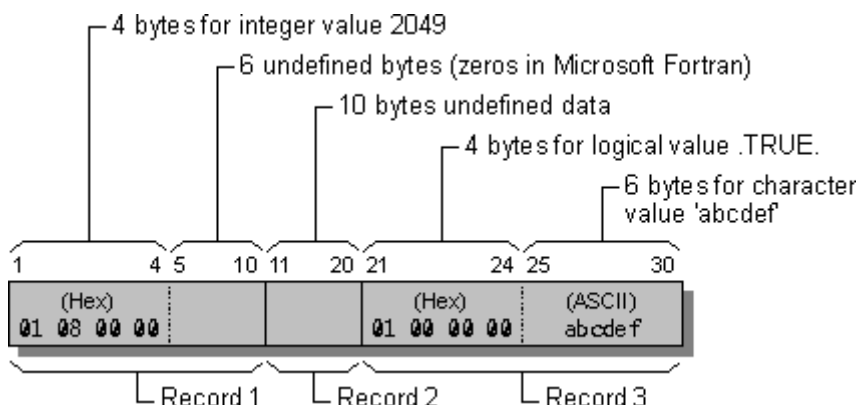
書式なし直接ファイルは、一連の書式なしレコードです。レコードの読み書きは任意の順序で行うことができます。すべてのレコードは、OPEN 文の RECL で指定子で指定された同じ長さです。レコードを区切ったり、レコードの構造を指示する区切りバイトはありません。

書式なし直接ファイルには部分的なレコードを書き出すことができます。インテル Visual Fortran は、ASCII のヌル文字を使用し、これらのレコードを固定レコード長まで埋めます。ファイル内の書き出されなかったレコードには未定義のデータが含まれます。

次のプログラムでは、続く図のようなサンプルの書式なし直接ファイルを作成します。

```
OPEN (3, FILE='UFDIR', RECL=10, &
  & FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
WRITE (3, REC=3) .TRUE., 'abcdef'
WRITE (3, REC=1) 2049
CLOSE (3)
END
```

書式なし直接ファイル



言語が混在したプログラミング

言語が混在したプログラミングの概要

言語が混在したプログラミングとは、ソースコードが複数の言語で書かれているプログラムを作成するプロセスのことを指します。言語が混在したプログラミングでは、次のことが可能です:

- 別の言語で書かれている既存のコードを呼び出す。
- 特定の言語では実装が難しいプロシージャを使用する。
- 処理速度を向上させる。

言語が混在したプログラミングは、インテル® Fortran およびインテル® C++ の間でも可能です。

言語が混在したプログラムを正しく作成するには、変数とプロシージャの命名、スタックの使用、および異なる言語で書かれたルーチン間での引数渡しに関する規則を確立しなくてはなりません。これらの規則をまとめて、*呼び出し規則*といいます。

呼び出し規則には、次の事柄が含まれます:

- スタックの注意事項: ルーチンが受け取る引数の数は、可変なのか、または固定なのか。
- 命名規則
 - 大文字・小文字の違いに特別な意味があるのかどうか。
 - 外部名は変更されているかどうか。
- 引数渡しプロトコル

- 引数は、値によって渡されているのか、または参照によって渡されているのか。
- 言語間で等しいデータ型やデータ構造には、どのようなものがあるのか。

このセクションでは、Fortran、C、C++、およびアセンブリ言語でルーチンを作成するときに使用する呼び出し規則に関する情報を提供します。以下のトピックを参照してください:

[メイン・プログラムからのサブプログラムの呼び出し](#)

[言語が混在したプログラミングのまとめ](#)

[言語が混在したプログラミングにおける呼び出し規則の調整の概要](#)

[言語が混在したプログラミングにおける命名規則の調整の概要](#)

[Fortran でのプロシージャのプロトタイピング](#)

[言語が混在したプログラミングにおけるデータ交換と参照の概要](#)

[言語が混在したプログラミングにおけるデータ型の処理の概要](#)

[インテル® Fortran/C が混在したプログラムの概要](#)

メイン・プログラムからのサブプログラムの呼び出し

メイン・プログラムからの呼び出し

インテル® Fortran のメイン・プログラムから、スタティック・ライブラリおよび共用ライブラリを含むインテル Fortran のサブプログラムを呼び出すことができます。

言語が混在したアプリケーションについては、適切な呼び出し規則が使用されていれば、インテル Fortran メイン・プログラムは インテル® C++ で記述されたサブプログラムを呼び出すことができます ([「インテル® Fortran プログラムからの C プロシージャの呼び出し」](#)を参照してください)。

インテル Fortran のサブプログラムは、インテル C++ のメイン・プログラムによって呼び出せます。

サブプログラムの呼び出し

メイン・プログラムがインテル Fortran またはインテル C++ で記述されている場合は、スタティック・ライブラリのサブプログラムを使用できます。

メイン・プログラムがインテル Fortran またはインテル C++ で記述されている場合は、共用ライブラリのサブプログラムを使用できます。

言語が混在したプログラミングのまとめ

言語が混在したプログラミングとは、ある言語で書かれたルーチンが、別の言語で書かれた関数、プロシージャ、またはサブルーチンを呼び出すことを指します。例えば、Fortran で書かれたメイン・プログラムから、アセンブリ言語でプログラムされた特定のタスクを実行する場合や、既存の共有ライブラリやシステム・プロシージャを呼び出すような場合です。

言語が混在したプログラミングは、インテル® Fortran および インテル® C++ で可能です。これらの言語は、関数、サブルーチン、およびプロシージャをほぼ同じ方法で実装しているからです。次の表に、各言語のさまざまな種類のルーチンの関係を示します。例えば、C のメイン・プログラムから、実際には Fortran のサブルーチンとして実装されている外部の void 関数を呼び出すことができます。

各言語のルーチン呼び出しの対応関係

言語	戻り値付きの呼び出し	戻り値なしの呼び出し
Fortran	FUNCTION	SUBROUTINE
C および C++	function	(void) function

これらの言語がルーチンを実装する方法には、いくつかの重要な違いがあります。プログラムが実行に失敗したり、誤った結果を生成するのを防ぐには、どの 2 つの言語間でも、引数渡し、命名規則、およびその他のインターフェイス関連の問題を慎重に、また、一貫性のある形で調整しなくてはなりません。しかし、言語が混在したプログラミングには、一般にこの余分な作業を行うに足る利点があります。

次に、言語が混在したプログラミングにおける利点と制限をいくつか要約します:

- Fortran/アセンブリ言語

アセンブリ言語のルーチンは、Fortran や C などの高水準言語のように初期化を必要としないため、小さく、きわめて高速に動作します。また、高水準言語

のユーザには使用できないハードウェア命令を参照することができます。
Fortran/アセンブリ言語のプログラムでは、メイン・ルーチンを Fortran でコンパイルすることで、アセンブリ言語のコードから Fortran の高水準プロシージャやライブラリ関数を参照でき、同時にアセンブリ言語のルーチンを速度と効率の点で最大限にチューニングすることができます。また、メイン・プログラムがアセンブリ言語のプログラムであってもかまいません。

- Fortran/C (または C++)

一般に、Fortran と C が混在したプログラムは、一方の言語で書かれた既存のコードを使用することを目的に作成されます。Fortran と C は互いのルーチン呼び出すことができるので、メイン・ルーチンはどちらの言語でも書くことができます。

このセクションでは、Fortran と他の言語の違いを調整するために使用できるキーワード、属性、および技術について説明しています。次に示すセクションでは、呼び出し規則の調整、命令規則の調整、およびインターフェイス・プロシージャの作成方法について説明しています。

- [言語が混在したプログラミングにおける呼び出し規則の調整](#)
- [言語が混在したプログラミングにおける命名規則の調整](#)
- [Fortran でのプロシージャのプロトタイピング](#)

言語が混在したプロシージャ間で一貫性のあるインターフェイスを作成した後は、個々のデータ型（文字列、配列など）の処理方法の違いを調整する必要があります。詳細は、[「言語が混在したプログラミングにおけるデータ交換と参照」](#)を参照してください。



注

このセクションでは、「ルーチン」という言葉を、各言語の関数、サブルーチン、およびプロシージャを指す一般的な用語として使用しています。

言語が混在したプログラミングにおける呼び出し規則の調整

言語が混在したプログラミングにおける呼び出し規則の調整の概要

呼び出し規則は、プログラムがルーチンをどのように呼び出すのか、引数がどのように渡されるのか、そしてルーチンの名前がどのように指定されるのかを決定します。詳細は、「[言語が混在したプログラミングにおける呼び出し規則の調整](#)」を参照してください。

単一言語のプログラムでは、すべてのルーチンでデフォルト設定が 1 つしかなく、また、インターフェイス・ブロックを含むヘッダファイルと Fortran モジュール・ファイルが、呼び出し側と呼び出される側のルーチン間で一貫性を実現するため、呼び出し規則はほぼ常に正しくなります。

言語が混在したプログラムでは、異なる言語が同じヘッダファイルを共有することはできません。このため、異なる呼び出し規則を使用する Fortran と C のルーチンをリンクすると、そのエラーは、実行時に不正な呼び出しが行われるまでは表面化しません。このような不正な呼び出しは、実行時に、プログラム内の実際の原因とは関係のなさそうな場所（呼び出しエラーが原因で発生したメモリやスタックの破壊）で、予期しない結果や致命的なエラーを引き起こします。このため、言語が混在した呼び出しの呼び出し規則を慎重に検証する必要があります。

言語間の呼び出し規則についての説明は、外部プロシージャにのみ適用されます。内部プロシージャを、それを含むプログラム・ユニットの外から呼び出すことはできません。

呼び出し規則は、プログラミングに次の 4 つの形で影響を与えます。

1. 呼び出し側ルーチンは、呼び出し規則を使って、別のルーチンに引数を渡す順序を決定します。呼び出されたルーチンは、呼び出し規則を使って、渡された引数を受け取る順序を決定します。Fortran では、言語が混在したインターフェイス内で INTERFACE 文を使うか、データまたは関数宣言内で指定することで、これらの規則を指定できます。C/C++ および Fortran では、どちらも引数を左から右の順序で渡します。
2. 呼び出し側ルーチンと呼び出されたルーチンは、呼び出し規則を使って、可変長引数を渡すときのオプションを選択します。

3. 呼び出し側ルーチンと呼び出されたルーチンは、呼び出し規則を使って、引数を値で（値渡し）、または参照で（アドレス渡し）渡します。個々の Fortran 引数を、ATTRIBUTES オプションの VALUE または REFERENCE で指定することもできます。
4. 呼び出し側ルーチンと呼び出されたルーチンは、呼び出し規則を使って、プロシージャ名の命名規則を設定します。ALIAS ディレクティブ（または ATTRIBUTES オプションの ALIAS）を使うと、Fortran での名前とは関係なく、任意のプロシージャ名を設定することができます。C では大文字・小文字が区別されるのに対し、Fortran では区別されないので、この機能は有用です。

「ATTRIBUTES プロパティと呼び出し規則」も参照してください。

ATTRIBUTES プロパティと呼び出し規則

ATTRIBUTES プロパティ（またはオプション）の C、REFERENCE、VALUE、および VARYING は、いずれもルーチンの呼び出し規則に影響を与えます。これらのプロパティは、次のように指定することができます：

- ルーチン全体に対して、C プロパティ、REFERENCE プロパティ、および VARYING プロパティを指定できます。
- 各引数に対して、VALUE プロパティおよび REFERENCE プロパティを指定できます。

デフォルトでは、Fortran はすべてのデータを参照で渡します（ただし、例外として文字列の隠された文字長引数は値で渡されます）。C プロパティが使用された場合は、配列以外のほとんどすべてのデータを値で渡すようにデフォルト設定が変更されます。また、呼び出し規則の C プロパティ以外にも、引数プロパティの VALUE および REFERENCE を指定することで、呼び出し規則のプロパティとは無関係に、引数を値または参照で渡すことができます。配列は参照でしか渡せません。

Fortran プロシージャを特定の属性を持つように宣言することにより、複数の呼び出し規則を指定できます。次に例を示します：

```
INTERFACE
SUBROUTINE MY_SUB (I)
    !DEC$ ATTRIBUTES C, ALIAS:'My_Sub_' :: MY_SUB !ia32
systems
    INTEGER I    END SUBROUTINE MY_SUB
END INTERFACE
```

このコードでは、MY_SUB という名前のサブルーチンが C プロパティで宣言され、また、外部名 My_Sub_ が ALIAS プロパティで設定され宣言されています。

次の例では、サブルーチンが C の呼び出し規則で呼び出されるように宣言されています:

```
SUBROUTINE CALLED_FROM_C (A)
  !DEC$ ATTRIBUTES C :: CALLED_FROM_C
  INTEGER A
```

Fortran の一般的な呼び出し規則ディレクティブの効果を以下の表に要約します:

ATTRIBUTES プロパティの呼び出し規則

引数	デフォルト	C	C, REFERENCE
スカラ	参照	値	参照
スカラ [値]	値	値	値
スカラ [参照]	参照	参照	参照
文字列	参照 Len:End または Len:Mixed	文字列(1:1)	参照 Len:End または Len:Mixed
文字列 [値]	エラー	文字列(1:1)	文字列(1:1)
文字列 [参照]	参照 No Len または Len:Mixed	参照 No Len	参照 No Len
配列	参照	参照	参照
配列 [値]	エラー	エラー	エラー
配列 [参照]	参照	参照	参照
派生型	参照	値 サイズに依存	参照
派生型 [値]	値 サイズに依存	値 サイズに依存	値 サイズに依存
派生型 [参照]	参照	参照	参照
F90 ポインタ	記述子	記述子	記述子
F90 ポインタ [値]	エラー	エラー	エラー
F90 ポインタ [参照]	記述子	記述子	記述子

すべての呼び出し規則ではプロシージャ名が小文字になります。

上の表で使用されている用語の意味は以下のとおりです:

[値]	VALUE 属性が割り当てられた引数。
[参照]	REFERENCE 属性が割り当てられた引数。
値	引数の値がスタックにプッシュされます。すべての値が次の 4 バイト境界までパディングされます。
参照	IA-32 システムでは、4 バイトの引数アドレスがスタックにプッシュされます。 Itanium® ベース・システムでは、8 バイトの引数アドレスがスタックにプッシュされます。
Len:End または Len:Mixed	特定の文字列引数について次の意味を持ちます: <ul style="list-style-type: none"> Len:End は、-nomixed_str_len_arg が設定されている場合に適用されます。他のすべての引数の後に、文字列長がスタックに (値で) プッシュされます。これはデフォルトで使用されます。 Len:Mixed は、-mixed_str_len_arg が設定されている場合に適用されます。文字列の先頭アドレスの直後に、文字列長がスタックに (値で) プッシュされます。
No Len または Len:Mixed	特定の文字列引数について次の意味を持ちます: <ul style="list-style-type: none"> No Len は、-nomixed_str_len_arg が設定されている場合に適用されます。呼び出されたプロシージャからは文字列長がわかりません。これはデフォルトで使用されます。 Len:Mixed は、-mixed_str_len_arg が設定されている場合に適用されます。文字列の先頭アドレスの直後に、文字列長がスタックに (値で) プッシュされます。
No Len	文字列引数において、呼び出されたプロシージャからは文字列長がわかりません。
文字列(1:1)	文字列引数において、最初の文字が ICHAR(string(1:1)) のように INTEGER(4) に変換され、スタックに値でプッシュされます。
エラー	コンパイラ・エラーを生成します。

記述子	IA-32 システムでは、配列記述子の 4 バイト・アドレスです。 Itanium ベース・システムでは、配列記述子の 8 バイト・アドレスです。
サイズに依存	IA-32 システムでは、値で指定される派生型引数は次のように渡されます: <ul style="list-style-type: none"> • 1~4 バイトの引数は値で渡されます。 • 5~8 バイトの引数は、2 つのレジスタ (2 つの引数) を使用して値で渡されます。 • 8 バイト以上の引数は一時的な記憶アドレスを値で渡すことにより、値のセマンティックスを提供します。

下の表に、他言語の呼び出し規則と一致する他の ATTRIBUTES プロパティを示します:

他言語の呼び出し規則	一致する ATTRIBUTES プロパティ
C/C++ cdecl (デフォルト)	C

ALIAS プロパティを他の任意の Fortran 呼び出し規則プロパティとともに使用することで、名前に含まれる大文字・小文字をそのまま維持できます。また、DECORATE プロパティを ALIAS プロパティと組み合わせることにより、実際に呼び出しメカニズムと関係付けるために、ALIAS で使用される外部名にプリフィックスとポストフィックスの修飾を持たせるように指定できます。

言語が混在したプログラミングにおける命名規則の調整

言語が混在したプログラミングにおける命名規則の調整の概要

ATTRIBUTES オプションの C は、命名規則および呼び出し規則を決定します。

呼び出し規則は、引数がどのように移動され、格納されるのかを指定します。命名規則は、.o ファイルに格納するときに、シンボル名をどのように変更するかを指定します。名前は、同じプログラムの各部分間および外部のルーチン間で共有される外部

データシンボルにおいて重要になります。シンボル名（サブルーチン名など）は、呼び出すすべてのルーチンの間で同一にする必要があるメモリ位置を識別します。

パラメータ名（プロシージャの定義でそのプロシージャに渡される変数に与えられる名前）は、まったく影響を受けません。

名前は、大文字・小文字の区別あり (C)、大文字・小文字の区別なし (Fortran)、名前の修飾 (C++) などの理由から変更されます。命名規則の調整が行われないと、プログラムはリンクに成功せず、“未解決外部” エラーが発生します。

次のトピックを参照してください。

[C/C++ 命名規則](#)

[Fortran、C、C++ のプロシージャ名](#)

[名前の大文字・小文字の調整](#)

[Fortran モジュール名と ATTRIBUTES](#)

C/C++ 命名規則

C および C++ では、シンボルテーブル中で大文字と小文字の区別が保持されますが、Fortran ではデフォルトで保持されません。この違いには注意を払う必要があります。幸いなことに、Fortran ディレクティブの ATTRIBUTES ALIAS オプションを使用することで、名前の不一致を解消したり、大文字・小文字の区別を保持したり、また、名前がすべて小文字化される Fortran の自動変換を無視できます。

C++ は、C と同じ呼び出し規則と引数渡しの手法を使用しますが、命名規則に関しては、C++ の外部シンボルの修飾のために違いがあります。C++ のコードが .cpp ファイル（統合開発環境で C/C++ のファイルを選択したときに作成されるファイル）に存在する場合、外部名に C++ の名前修飾セマンティクスが適用されるため、リンクエラーが頻繁に発生します。extern "C" の構文を使用することにより、C++ モジュールは名前の修飾を行わず、他の言語とデータおよびルーチンを共有することができます。

次の例では、prn を C の命名規則を使って外部関数として宣言しています。この宣言は C++ のソースコードでよく見られます。

```
extern "C" { void prn(); }
```

Fortran で書かれた関数を呼び出すには、関数を C と同じように宣言し、リンケージ指定として “C” を使用します。例えば、C++ から Fortran 関数の FACT を呼び出すには、次のように宣言します:

```
extern "C" { int FACT( int n ); }
```

extern “C” 構文を使用することにより、C++ から他の言語への呼び出しを調整したり、他の言語から呼び出される C++ ルーチンの命名規則を変更することができます。ただし、extern “C” は C++ からしか使用できません。C++ コードが extern “C” を使用しておらず、コードを変更することもできない場合は、名前の修飾を判断し、他の言語から生成することしか C++ ルーチンを呼び出す手段はありません。修飾方式がバージョン間で変更されないという保証がないので、このようなアプローチは最後の手段としてのみ使用するようにしてください。

extern “C” の使用にはいくつかの制限があります:

- extern “C” でメンバ関数を宣言することはできません。
- 多重定義された関数では、1 つのインスタンスに対してのみ extern “C” 構文を指定できます。多重定義された関数のその他のインスタンスは C++ のリンケージを持ちます。

Fortran、C、C++ のプロシージャ名

Fortran、C、および C++ でのプロシージャ名の処理について次に示します:

言語	属性	変換後の名前	.o ファイル中の名前の大文字・小文字
Fortran	cDEC\$ ATTRIBUTES C	<i>name_</i>	すべて小文字
Fortran	デフォルト	<i>name_</i>	すべて小文字
C	cdecl (デフォルト)	<i>name_</i>	大文字・小文字の区別を保ちます
C	__stdcall	<i>_name@n</i>	大文字・小文字の区別を保ちます
C++	デフォルト	<i>_name@@decoration</i>	大文字・小文字の区別を保ちます

名前の大文字・小文字の調整

次に、言語間での名前の調整方法を簡単に述べます。

- すべてが小文字の名前

C でルーチン名がすべて小文字で表記されている場合、命名規則は自動的に正しくなります。名前はすべて小文字に変換されるので、Fortran ソースコードでは、大文字・小文字の混在を含め、任意の大文字・小文字の組み合わせを使用することができます。

- 大文字・小文字が混在した名前

C でルーチンの名前に大文字・小文字が混在しており、その名前を変更できない場合、Fortran の ATTRIBUTES ALIAS オプションを使用して名前の競合を解決します。ALIAS を指定しないと、Fortran は大文字・小文字が混在した名前を保持しないので、この場合、ALIAS オプションが必要です。

ALIAS オプションを使用するには、.o ファイルで名前を引用符で囲んで指定します。

次は C 関数 `My_Proc` を指定する例です:

```
!DEC$ ATTRIBUTES ALIAS:'My_Proc_' :: My_Proc
```

Fortran モジュール名と ATTRIBUTES

Fortran モジュール・エンティティ（データとプロシージャ）は、他の外部エンティティとは異なる外部名を持ちます。モジュール名は次の規則を使用します:

`MODULENAME`_mp_`ENTITY`_

`MODULENAME` はモジュールの名前です。`ENTITY` は、`MODULENAME` に含まれているモジュール・プロシージャ名またはモジュール・データ名です。`_mp_` はモジュール名とエンティティ名の間に挿入される区切り文字で、常に小文字です。

次に例を示します。

```
MODULE mymod
  INTEGER a
CONTAINS
  SUBROUTINE b (j)
    INTEGER j
```

```
END SUBROUTINE
END MODULE
```

この場合、コンパイル後の .o ファイルには、次のシンボルが定義されます:

```
mymod_mp_a_
mymod_mp_b_
```

コンパイラ・オプションを使用することで、モジュール・データおよびモジュール・プロシージャの名前付けを設定できます。



注

ALIAS 以外の ATTRIBUTES オプションはモジュール名に影響を与えません。

以下の表では、上記のサンプル・モジュールの各サブルーチンに、個々の ATTRIBUTES プロパティがどのように影響を与えるかを示します。

Fortran モジュール名に対する ATTRIBUTES オプションの効果

ルーチン 'b' に対して指定された ATTRIBUTES プロパティ	.o ファイル内のプロシージャ名
なし	mymod_mp_b_
C	mymod_mp_b_
ALIAS	他のすべてを上書きします。エイリアスで指定した名前
VARYING	名前への影響はなし

他の言語から Fortran モジュールを呼び出したり、モジュール・データを参照するコードを書くことができます。他の命名規則と呼び出し規則と同じように、モジュール名は 2 つの言語間で一致していなくてはなりません。一般に、これは、Fortran で C の規則を使用し、また、別の言語でモジュールを定義する場合は ALIAS プロパティを使用して、Fortran での名前を一致させる必要があることを意味します。異なる言語間でのモジュールの使用例は、[「Fortran と C が混在したプログラミングにおけるモジュールの使用」](#)を参照してください。

Fortran でのプロシージャのプロトタイプ グ

プロトタイプ (インターフェイス・ブロック) を Fortran ソースコードで定義することにより、Fortran コンパイラに対して、外部参照用にどの言語規則を使用するかを指示できます。インターフェイス・ブロックは、INTERFACE 文で宣言します。INTERFACE 文に関する詳細は、『Language Reference』(英語) の「Program Units and Procedures」を参照してください。

INTERFACE 文の一般的な形式を次に示します:

```
INTERFACE
  routine statement
  [routine ATTRIBUTE options]
  [argument ATTRIBUTE options]
  formal argument declarations
END routine name
END INTERFACE
```

routine statement では、値が返されるかどうかにより、FUNCTION または SUBROUTINE を定義します。省略可能な *routine ATTRIBUTE options* (C など) は、プロトタイプ文に含まれるルーチンに対して、どの呼び出し規則、命名規則、および引数渡し規則を適用するかを決定します。省略可能な *argument ATTRIBUTE options* (VALUE や REFERENCE など) は、個々の引数に付加されるプロパティです。*formal argument declarations* は、Fortran データ型の宣言を指します。同じ INTERFACE ブロックで複数のプロシージャを指定できます。

例えば、次のプロトタイプを持つ C 関数を呼び出すとします:

```
extern void My_Proc (int i);
```

Fortran からこの関数を呼び出すには、次の INTERFACE ブロックを宣言する必要があります:

```
INTERFACE
  SUBROUTINE my_Proc (I)
    !DEC$ ATTRIBUTES C, ALIAS:'My_Proc_' :: my_Proc
    INTEGER I
  END SUBROUTINE my_Proc
END INTERFACE
```

ALIAS 文字列を除き、Fortran プログラムでは、My_Proc の大文字・小文字が区別されないのでご注意ください。

言語が混在したプログラミングにおけるデータ交換と参照

言語が混在したプログラミングにおけるデータ交換と参照の概要

言語が混在したルーチン間でのデータの共用にはいくつかのアプローチがあります。これらのアプローチは個々の言語中で使用することができます。

一般に、多数のパラメータを使用する場合、またはパラメータ型が多様な場合、モジュールまたは外部データ宣言を使用することを推奨します。これはどの言語を使う場合にも当てはまり、特に言語を混在させるときには重要になります。

[「Fortran と C が混在したプログラミングにおけるモジュールの使用」](#)も参照してください。

次のトピックを参照してください。

[言語が混在したプログラミングにおける引数の渡し方](#)

[言語が混在したプログラミングにおける共通外部データの使用](#)

言語が混在したプログラミングにおける引数の渡し方

Fortran、C、および C++ の間でデータを渡すには、個々の言語で行う場合と同じように、引数の並びを使用します（例えば、CALL MYSUB(a, b, c) では a、b、および c が引数になります）。各引数を渡す方法には 2 通りあります：

- **値で渡す**: 引数の値を渡します。
- **参照で渡す**: 引数のアドレスを渡します。IA-32 システムでは、Fortran、C、および C++ は 4 バイトアドレスを使用します。また、Itanium® ベース・システムでは、これらの言語は 8 バイトアドレスを使用します。

すべての呼び出しに対して、呼び出し側のプログラムと呼び出される側のルーチンで、個々の引数の渡し方が一致していることを確認する必要があります。渡し方が一致していない場合は、呼び出されたルーチンは無効なデータを受け取ることになります。

Fortran での引数の渡し方は、指定した呼び出し規則によって変わります。デフォルトでは、Fortran はすべてのデータを参照で渡します（ただし、例外として文字列の隠された文字長引数は値で渡されます）。

ATTRIBUTES C オプションが使用された場合は、配列以外のすべてのデータを値で渡すようにデフォルト設定が変更されます。C オプションに加えて REFERENCE オプションもプロシージャに対して指定されている場合は、デフォルトですべての引数が参照で渡されます。

Fortran では、呼び出し規則オプションの C を指定する以外にも、引数オプションの VALUE および REFERENCE を指定することで、引数を値または参照で渡すことができます。言語が混在したプログラミングでは、デフォルト設定に頼るよりも、引数の渡し方を明示的に指定することをお勧めします。



注

ATTRIBUTES 以外にも、[コンパイラ・オプションの -\[no\]mixed_str_len_arg](#) を使用することで、デフォルトの引数渡し規則を設定できます（文字列の隠された文字長など）。

C での参照渡しおよび値渡しの例を次に示します。いずれも、次に示すサンプルの Fortran サブルーチン TESTPROC へのインタフェースです。TESTPROC の定義では、各引数の渡し方が宣言されています。この例では、REFERENCE オプションが必ずしも必要ではありませんが、このオプションを使用することで引数渡し規則が明確になります。

```
SUBROUTINE TESTPROC ( VALPARM, REFPARM )
  !DEC$ ATTRIBUTES VALUE :: VALPARM
  !DEC$ ATTRIBUTES REFERENCE :: REFPARM
  INTEGER VALPARM
  INTEGER REFPARM
END SUBROUTINE
```

C および C++ では、配列以外のすべての引数が値で渡されます。配列の場合は、配列の第 1 メンバのアドレスへの参照で渡されます。Fortran とは異なり、C および C++ は各引数の渡し方を指定する呼び出し規則ディレクティブを使用しません。配列以外の C のデータを参照で渡すには、そのデータを指すポインタを渡す必要があります。

また、C の配列を値で渡すには、配列を構造体のメンバとして宣言し、その構造体を渡す必要があります。次の C の宣言では、サンプルの Fortran testproc サブルーチンへの呼び出しを設定しています:

```
extern void testproc( int ValParm, int *RefParm );
```

参照および値で引数を渡す方法を下の表で要約します。配列は通常、参照で渡されるため、C の配列名はその開始アドレスと同等に扱われます。各引数に割り当てられるように、プロシージャにも REFERENCE プロパティを割り当てることができます。

引数の参照渡しおよび値渡し

言語	属性	引数の型	参照渡し	値渡し
Fortran	デフォルト	スカラーおよび派生型	デフォルト	VALUE オプション
	C オプション	スカラーおよび派生型	REFERENCE オプション	デフォルト
	デフォルト	配列	デフォルト	値渡しは不可
	C オプション	配列	デフォルト	値渡しは不可
C/C++		配列以外	argument_name ポインタ	デフォルト
		配列	デフォルト	Struct {type} array_name

この表では、インテル® Fortran における文字列引数および Fortran 95/90 ポインタ引数の渡し方については触れていません。これらの引数は、他の引数とは異なる形式で渡されるからです。Fortran はデフォルトで文字列を文字長とともに参照で渡します。文字長の位置は、コンパイラ・オプション [-mixed_str_len_arg](#) (文字列の先頭アドレスの直後) または [-nomixed_str_len_arg](#) (すべての引数の後) が指定されているかどうかによって異なります。デフォルト設定では、[-nomixed_str_len_arg](#) が使用されます。

Fortran 95/90 の配列ポインタおよび形状引継ぎ配列は、配列記述子のアドレスを渡すことによって渡されます。

Fortran 95/90 のポインタおよび文字列の渡し方に対する属性の影響については、「[配列ポインタと割付け配列の処理](#)」および「[文字列の処理](#)」を参照してください。

言語が混在したプログラミングにおける共通外部データの使用

共通外部データ構造には、Fortran の共通ブロック、およびグローバル変数または外部変数として宣言された C の構造体と変数が含まれます。いずれのデータ指定も、それらを定義するルーチン外のルーチンから使用できる外部変数を作成します。

外部変数では大文字と小文字を区別するため、命名規則のセクションで説明されているように、異なる言語の間で大文字と小文字が一致している必要があります。次のセクションでは、共通外部データの交換について説明します。

- [グローバル変数の使用](#)
- [Fortran の共通ブロックおよび C の構造体の使用](#)

言語が混在したプログラミングにおけるグローバル変数の使用

Fortran と C で変数を共有するには、1 つの言語で変数をグローバル変数（または COMMON）として宣言し、別の言語でその変数を外部変数としてアクセスします。Fortran プログラムの場合、変数は引数として渡す必要があります。

Fortran の変数は、ATTRIBUTES の EXTERN オプションを使用して、グローバル・パラメータにアクセスすることができます。次に例を示します。

```
!DEC$ ATTRIBUTES C, EXTERN :: idata
INTEGER idata (20)
```

EXTERN オプションは、変数が別のソースファイルで定義され、グローバル変数として宣言されていることをコンパイラに伝えます。Fortran が EXTERN オプションを使用して変数を外部変数として宣言する場合、変数を共有している言語はその変数をグローバル変数として宣言する必要があります。

C の場合、変数は次の文を使用してグローバル変数として宣言されます。

```
int idata[20]; // declared as global (outside of any function)
```

Fortran は、変数をグローバル変数（COMMON）として宣言し、他の言語ではその変数を外部変数として参照することができます。

```
! Fortran declaring PI global
REAL PI
COMMON /PI/ PI ! Common Block and variable have the same name
```

C の場合、変数は次の文を使用して外部変数として参照されます。

```
//C code with external reference to PI0
extern float PI;
```

C が参照するグローバル変数名は、Fortran の共通ブロック内の変数名ではなく、共通ブロック名であることに注意してください。したがって、空白の共通ブロックを使用して、C と Fortran の間でデータを参照することはできません。上の例では、共通ブロック名と変数名が同じため、2 つの言語の間における変数を把握するのに役立ちます。共通ブロックが複数の変数を含む場合、すべての変数に同じ共通ブロック名を使用することはできません。(「[Fortran の共通ブロックおよび C の構造体の使用](#)」を参照してください。)

Fortran の共通ブロックおよび C の構造体の使用

Fortran の共通ブロックおよび C の構造体の間で参照を行う場合、メンバ変数をメモリに格納する方法が共通ブロックと構造体で異なるということに注意する必要があります。Fortran は次の規則に従って、共通ブロックの変数を最大限にパックしてメモリに格納します。

- 共通ブロック内にある BYTE、INTEGER(1)、LOGICAL(1)、または CHARACTER 単一変数は、メモリ内の前にある変数または配列の直後から始まります。
- その他すべての型の単一変数は、メモリ内の前にある変数または配列の直後の偶数アドレスから始まります。
- すべての変数配列は、メモリ内の前にある変数または配列の直後の偶数アドレスから始まりますが、CHARACTER 配列は例外で、常に前にある変数または配列の直後から始まります。
- すべての共通ブロックは、4 バイトでアライメントされたアドレスから始まります。

これらのパディング規則があるため、C の構造体要素と Fortran の共通ブロック要素のアライメントを考慮して、両方の言語ですべての変数を同じ型または種類にするか (両方の言語で 4 バイトおよび 8 バイトのデータ型のみ使用することで、簡単にこの作業を行うことができます)、または C のコードで C の構造体の前後に C のパックプラグマを使用して、C のデータを Fortran のデータのようにパックすることで、要素の一致を保証する必要があります。次に例を示します。

```
#pragma pack(2)
struct {
    int N;
    char INFO[30];
} examp;
#pragma pack()
```

元のパック状態に戻すには、構造体の終わりに `#pragma pack()` を追加する必要があります。(注意: Fortran モジュールのデータは、適切な名前を使用することで、C の構造体と直接に共有することができます。)

アライメントおよびパディングを考慮しておけば、C 言語から共通ブロック全体、または複数の共通ブロックにアクセスすることができます。また、Fortran 共通ブロックの個々のメンバは、その他のデータ項目と同じように引数リストで渡すことができます。次のセクションでは、言語が混在したデータ交換における共通ブロックの使用について説明します。

- [共通ブロックおよび C の構造体への直接アクセス](#)
- [共通ブロックのアドレスの受け渡し](#)

共通ブロックおよび C の構造体への直接アクセス

適切なフィールドが含まれた C の外部構造体を定義し、Fortran および C のアライメントとパディングを一致させることで、Fortran の共通ブロックを C から直接アクセスすることができます。ATTRIBUTES オプションの C および ALIAS を使用することで、共通ブロックで大文字と小文字が混在した名前を使用することができます。

例えば、Fortran コードに `Really` という名前の共通ブロックが含まれているとします。

```
!DEC$ ATTRIBUTES ALIAS:'Really' :: Really
REAL(4) x, y, z(6)
REAL(8) ydbl
COMMON / Really / x, y, z(6), ydbl
```

このデータ構造を C のコードからアクセスするには、次の外部データ構造を使用します。

```
#pragma pack(2)
extern struct {
    float x, y, z[6];
    double ydbl;
```



```

} Really;
#pragma pack()

```

また、C の構造体に対応する共通ブロックを作成することで、Fortran から C の構造体にアクセスすることもできます。これは上記に説明したものと逆のケースです。ただし、共通ブロックと構造体の定義、共通のアドレス（名前）の提供、およびメモリ内のアライメントが行われている場合、どちらの言語も変数の同じメモリ位置を共有するため、実装の方法は同じです。

共通ブロックのアドレスの受け渡し

共通ブロックのアドレスを渡すには、ブロックにある最初の変数のアドレスを渡します。つまり、最初の変数を参照によって渡します。受け取り側の C または C++ モジュールは、構造体を参照で受け取ります。

次の例では、C の関数 `initcb` は、最初の変数が `n` という名前の共通ブロックのアドレスを受け取ります。この変数は、3 つのフィールドが含まれた構造体へのポインタとみなされます。

Fortran のソースコード:

```

!
INTERFACE
  SUBROUTINE initcb (BLOCK)
    !DEC$ ATTRIBUTES C :: initcb
    !DEC$ ATTRIBUTES REFERENCE :: BLOCK
    INTEGER BLOCK
  END SUBROUTINE
END INTERFACE
!
INTEGER n
REAL(8) x, y
COMMON /CBLOCK/n, x, y

      . . .
CALL initcb( n )

```

C のソースコード:

```

//
#pragma pack(2)
struct block_type

```



```

{
    int n;
    double x;
    double y;
};
#pragma pack()
//
void initcb( struct block_type *block_hed )
{
    block_hed->n = 1;
    block_hed->x = 10.0;
    block_hed->y = 20.0;
}

```

言語が混在したプログラミングにおけるデータ型の処理

言語が混在したプログラミングにおけるデータ型の処理の概要

呼び出し規則、命名規則、およびデータ交換の方法を解決してても、データ型について配慮する必要があります。これは、個々の言語がデータ型を異なる方法で処理しているためです。

次の表では、Fortran と C の間での等価なデータ型を示します。

等価なデータ型

Fortran データ型	C データ型
INTEGER(1)	char
INTEGER(2)	short
INTEGER(4)	int, long
INTEGER(8)	_int64

REAL(4)	float
REAL(8)	double
REAL(16)	---
CHARACTER(1)	unsigned char
CHARACTER*(*)	「文字列の処理」を参照
COMPLEX(4)	struct complex4 { float real, imag; };
COMPLEX(8)	struct complex8 { double real, imag; };
COMPLEX(16)	---
すべての LOGICAL 型	C では integer 型を使用

次のトピックを参照してください。

[数値、複素数、および論理データ型の処理](#)

[配列ポインタと割付け配列の処理](#)

[整数ポインタの処理](#)

[配列と Fortran 配列記述子の処理](#)

[文字列の処理](#)

[ユーザ定義型の処理](#)

数値、複素数、および論理データ型の処理

通常、数値データの受け渡しに問題は生じません。C プログラムが Fortran ルーチンに符号なしデータ型を渡した場合、ルーチンは引数を等価な符号付きデータ型として受け取ることができます。ただし、値が符号付き型の範囲を超えないように注意する必要があります。

「等価なデータ型」表には、Fortran と C/C++ の等価な数値データ型が要約されています。

C および C++ は、Fortran の COMPLEX(4) 型、COMPLEX(8) 型、および COMPLEX(16) 型を直接的には実装していません。ただし、これと等価な構造体を書くことができます。COMPLEX(4) 型は、それぞれ 4 バイトの浮動小数点数である 2 つのフィールドを持っており、第 1 フィールドは実数部、第 2 フィールドは虚数部を含みます。COMPLEX 型は COMPLEX(4) 型と同じです。COMPLEX(8) 型と COMPLEX(16) 型は、各フィールドに含まれている浮動小数点数のバイト数が異なっていることを除けば同じです (COMPLEX(8) には 8 バイトの浮動小数点数、COMPLEX(16) には 16 バイトの浮動小数点数が各フィールドに含まれています)。



注

IA-32 システムでは、COMPLEX 型の Fortran 関数は、引数リストの先頭に、隠された COMPLEX 引数を追加します。このような Fortran からの呼び出しを実装する C 関数は、この隠された引数を明示的に宣言し、値を返すために使用しなくてはなりません。C の戻り型は void です。

次に、Fortran COMPLEX 型用の C/C++ の構造体定義を示します。

```
struct complex4 {
    float real, imag;
};
struct complex8 {
    double real, imag;
};
```

Fortran LOGICAL(2) は、2 バイトの標識の値として格納されます (0 は FALSE。TRUE の値の処理方法は `-fpscomp [no]logicals` コンパイラ・オプションによって決定されます)。Fortran LOGICAL(4) の値は、4 バイトの標識の値として格納され、

LOGICAL(1) は 1 バイトとして格納されます。LOGICAL 型は LOGICAL(4) と同じで、C の int 型とも等価です。

Fortran では引数リスト、モジュール、共通ブロック、またはグローバル変数で LOGICAL 型の変数を使用でき、C では int 型を同じ引数として使用することができます。共通ブロックで使用する際には、LOGICAL(4) 型の方が、それよりも短いものより優れています。

インテル® C++ のクラス型は、クラスが仮想関数を定義しているか、または基本クラスを持っている場合を除き、対応する C の struct 型と同じ構成を持ちます。これらの機能を持たないクラスは、C の構造体と同じ方法で渡すことができます。

複素数データ型の戻り

Fortran プログラムで、関数から COMPLEX 型または DOUBLE COMPLEX 型の値が返されると予測される場合、Fortran コンパイラは呼び出されたプロシージャの引数リストの先頭に 1 つの引数を追加します。この引数は、呼び出されたプロシージャが結果を格納する位置を示すポインタです。

次に、WBAT という名前の複素数データ型プロシージャを返すための Fortran コードとそれに対応する C ルーチンの例を示します。

C から Fortran への複素数データ型の戻り例

Fortran のコード:

```
COMPLEX BAT, WBAT
REAL X, Y
BAT = WBAT ( X, Y )
```

対応する C ルーチン:

```
struct _mycomplex { float real, imag };
typedef struct _mycomplex _single_complex;

void WBAT (_single_complex location, float *x, float *y)
{
    float realpart;
    float imaginarypart;
    ... program text, producing realpart and imaginarypart...
    *location.real = realpart;
    *location.imag = imaginarypart;
}
```

上記の例では、次の制限および動作が適用されます:

- 引数の位置は、Fortran の呼び出し文で指定するわけではありません。コンパイラによって追加されます。
- C サブルーチンは、結果の実数部と虚数部を `location` に正しくコピーしなければなりません。
- 呼び出されたプロシージャは `void` 型です。

関数から `DOUBLE COMPLEX` 値が返された場合、WBAT の `location` の定義で、`float` 型が `double` 型に置き換えられます。

配列ポインタと割付け配列の処理

Fortran 95/90 の配列ポインタと配列の渡し方は、その時点で有効な ATTRIBUTES プロパティと、渡し先のプロシージャの INTERFACE (存在する場合) の影響を受けます。

INTERFACE で配列ポインタまたは形状無指定配列 (`ARRAY(:)` など) が宣言されている場合、その記述子が渡されます。これは、割付け配列だけでなく、配列ポインタとすべての配列に当てはまります。INTERFACE で配列ポインタまたは形状指定配列が宣言されている場合、またはインターフェイスが存在しない場合、配列ポインタと配列は、連続配列としてベースアドレスによって渡されます。この方法は、連続配列の各部分に対して、配列の最初の要素を渡す方法に似ています。

Fortran 95/90 配列ポインタまたは配列を別の言語に渡すときには、その記述子またはそのベースアドレスを渡すことができます。

次に、各種の属性が有効になっているときに、割付け配列と Fortran 95/90 配列ポインタがどのように渡されるかを示します。

- 配列ポインタまたは配列のプロパティがない場合、またはプロパティが REFERENCE の場合、渡す側のプロシージャのプロパティにかかわらず、記述子によって渡されます (なし; C; または C, REFERENCE)。
- 配列ポインタまたは配列のプロパティが VALUE の場合、渡す側のプロシージャのプロパティにかかわらず、エラーが返されます。

記述子を基本とする配列では、VALUE オプションは使用できないことに注意してください。

Fortran の配列ポインタまたは配列を Fortran 以外のルーチンに記述子によって渡す場合、そのルーチンは記述子を解釈する方法を知らなくてはなりません。記述子の一部は C のポインタのようなアドレス空間へのポインタで、またその一部はポインタ記述または次元数、ストライド、および範囲などの配列プロパティの記述です。

インテル® Fortran の配列記述子の書式についての詳細は、「[配列と Fortran 配列記述子の処理](#)」を参照してください。

スカラデータをポイントする Fortran 95/90 ポインタは、データのアドレスが含まれており、記述子によっては渡されません。

整数ポインタの処理

整数ポインタ (Cray* 形式ポインタとも呼ばれる) は Fortran 90 ポインタとは異なり、むしろ C のポインタに似ています。整数ポインタは、IA-32 システム上では 4 バイトの INTEGER、Itanium® ベース・システム上では 8 バイトの INTEGER です。

整数ポインタの渡し方

整数ポインタを別の言語で書かれたルーチンに渡すには、次の点に注意する必要があります:

- Fortran 以外で書かれたルーチンでは、引数は適切なデータ型のポインタとして宣言します。
- Fortran ルーチンから渡される引数は、被参照オブジェクト (ポインタに指されているオブジェクト) 名ではなく、整数ポインタ名でなくてはなりません。

次に例を示します。

```
! Fortran main program.
INTERFACE
  SUBROUTINE Ptr_Sub (p)
    !DEC$ ATTRIBUTES C, ALIAS:'Ptr_Sub' :: Ptr_Sub
    INTEGER p
  END SUBROUTINE Ptr_Sub
END INTERFACE
REAL A(10), VAR(10)
POINTER (p, VAR) ! VAR is the pointee
                  ! p is the integer pointer
p = LOC(A)
CALL Ptr_Sub (p)
WRITE(*,*) 'A(4) = ', A(4)
END
!
//C subprogram
void Ptr_Sub (float *p)
{
  p[3] = 23.5;
}
```

Itanium ベース・システムでは、INTERFACE ブロック内の `p` は、`INTEGER(8) p` で宣言する必要があります。

Fortran のメイン・プログラムと C の関数をビルドし、実行すると、次の出力が得られます:

```
A(4) = 23.50000
```

ポインタの受け取り方

別の言語で書かれたルーチンからポインタを受け取る際には、次の点に注意する必要があります:

- Fortran 以外で書かれたルーチンでは、引数は適切なデータ型のポインタとして宣言し、通常どおりに渡します。
- Fortran ルーチンが受け取る引数は、整数ポインタ名として宣言し、`POINTER` 文でこれを適切なデータ型の被参照オブジェクトに関連付ける必要があります (渡し側のルーチンとデータ型を一致させます)。Fortran ルーチン内では、被参照オブジェクトを使って、ポインタの指示先を設定し参照します。

次に例を示します。

```
!Fortran subroutine.
SUBROUTINE Iptr_Sub (p)
  !DEC$ ATTRIBUTES C, ALIAS:'Iptr_Sub' :: Iptr_Sub
  integer VAR(10) POINTER (p, VAR) !
  OPEN (8, FILE='STAT.DAT')
  READ (8, *) VAR(4) !Read from file and store
the                                !
                                !fourth element of VAR
END SUBROUTINE Iptr_Sub
!
//C main program
extern void Iptr_Sub(int *p);
main ( void ) { int a[10];
  Iptr_Sub (&a[0]);
  printf("a[3] = %i\n", a[3]);
}
```

C のメイン・プログラムと Fortran のサブルーチンをビルドし、実行すると、`STAT.DAT` ファイルに “4” が含まれている場合、次の出力が得られます:

```
a[3] = 4
```

配列と Fortran 配列記述子の処理

Fortran 95/90 では、配列を配列要素、配列サブセクション、または配列名で参照される配列全体として渡すことができます。Fortran では、配列要素は列優先順に編成されていて、最初の添字（次元）から優先されます。

Fortran と他の言語の間で配列を使用する場合、要素のインデックスと順序を考慮する必要があります。配列要素は個別に参照し、追跡する必要があります。Fortran および C では、配列要素のインデックス付けの方法が異なります。配列のインデックス付けは、ソースレベルでの問題であり、下位のデータに違いはありません。

Fortran および C の配列は、次の 2 つの点で異なります。

- 配列の下限の値が異なります。デフォルトでは、Fortran は配列の先頭の要素のインデックスを 1 とします。C および C++ では、0 とします。したがって、Fortran の添字は 1 つ大きい値に設定しなければなりません。(Fortran には、別の整数で下限を指定するオプションもあります)。
- 多重次元配列の場合、Fortran は最も左のインデックスから優先するのに対し、C では最も右のインデックスから優先します。これは、それぞれ列優先順および行優先順と呼ばれることがあります。

C では、`X[3][3]` として宣言されている配列の最初の 4 つの要素は次のとおりです。

```
X[0][0] X[0][1] X[0][2] X[1][0]
```

一方、Fortran では、最初の 4 つの要素は次のとおりです。

```
X(1, 1) X(2, 1) X(3, 1) X(1, 2)
```

インデックスの順序は、次元をいくつ宣言しても同じです。例えば、次の C の宣言の場合、

```
int arr1[2][10][15][20];
```

これは Fortran の次の宣言に相当します。

```
INTEGER arr1( 20, 15, 10, 2 )
```


C の配列宣言に使用される定数は、他の言語のように上限ではなく、範囲を表しています。このため、`int arr[5][5]` と宣言された C 配列の最後の要素は、`arr[5][5]` ではなく `arr[4][4]` となります。

次の表に、対応する配列宣言を示します。

異なる言語で対応する配列宣言

言語	配列宣言	Fortran からの配列参照
Fortran	<code>DIMENSION x(i, k)</code> または <code>type x(i, k)</code>	<code>x(i, k)</code>
C/C++	<code>type x[k][i]</code>	<code>x(i-1, k-1)</code>

インテル® Fortran の配列記述子の書式

Fortran 95/90 が複数のポインタ・メモリ・アドレスを追跡しなければならない場合に備えて、インテル Fortran コンパイラは、配列の編成に関する詳細情報を格納する *配列記述子* を使用します。

(結合またはプロシージャ・インターフェイス・ブロックによる) 明示的なインターフェイスを使用する場合、インテル Fortran は次の形式の配列引数記述子を生成します。

- 配列へのポインタ (配列ポインタ)
- 形状引継ぎ配列

一部のデータ構造体引数は、適切な明示的なインターフェイスが提供されている場合でも、記述子を使用しません。例えば、明示的な形状配列および大きさ引継ぎ配列は記述子を使用しません。一方、配列ポインタおよび割付け配列は、引数として使用するかどうかに関わらず、記述子を使用します。

インテル Fortran と Fortran 以外の言語 (C など) の間で呼び出しを行う場合、*暗黙的インターフェイス*を使用することで、インテル Fortran の記述子なしで配列を渡すことができます。ただし、呼び出されたルーチンがインテル Fortran 記述子に含まれる情報を必要とする場合、ルーチンを *明示的なインターフェイス*で宣言し、仮配列を形状引継ぎ配列として、またはポインタ属性で指定します。

Fortran 95/90 ポインタは、(配列境界が“矩形”である限り) 任意の形式で編成された任意のメモリ部分と関連付けることができます。また、Fortran 95/90 ポインタを C

などの他の言語に渡し、その言語で記述子を正しく解釈して、必要な情報を取得することができます。

ただし、配列記述子を使用すると、エラーが起こる可能性が高くなります。また、対応するコードの移植性も損なわれます。特に、次のことに注意してください。

- 記述子が正しく定義されていない場合、プログラムは間違ったメモリアドレスを参照し、一般保護違反を発生させることがあります。
- 配列記述子の書式は、各 Fortran コンパイラに対して固有です。配列記述子を使用するコードは、他のコンパイラやプラットフォームへ移植できません。例えば、現在のインテル Fortran の配列記述の書式は、インテル Fortran 7.0 の配列記述の書式と異なります。
- 配列記述子の書式は、将来変更される可能性があります。

次に、IA-32 システム上における現在のインテル Fortran の配列記述子のコンポーネントを示します。

- 最初のロングワード（バイト 0 から 3）には、ベースアドレスが含まれています。ベースアドレスとオフセットで、配列の最初のメモリ位置（開始）を定義します。
- 2 番目のロングワード（バイト 4 から 7）には、配列の 1 つの要素のサイズが含まれています。
- 3 番目のロングワード（バイト 8 から 11）には、オフセットが含まれています。オフセットをベースアドレスに加えて、配列の開始位置を定義します。
- 4 番目のロングワード（バイト 12 から 15）には、配列が定義されている（記憶領域が割り当てられている）場合に設定される下位ビットが含まれています。また、例えば連続した配列を示す場合などで、その他のビットもこのロングワード内でコンパイラによって設定される可能性があります。
- 5 番目のロングワード（バイト 16 から 19）には、配列の次元（次元数）が含まれています。
- 6 番目のロングワード（バイト 20 から 23）は、予約済みです。
- 残りのロングワード（バイト 24 から 107）には、個々の（最大 7 つまでの）次元に関する情報が含まれています。各次元は、さらに 3 つのロングワードによって記述されます。
 - 要素の数（範囲）
 - この次元で連続する 2 つの要素の開始アドレス間の距離（バイト数）
 - 下限

次元数 1 の配列は、3 つのロングワードを追加する必要があるため、合計で 9 つのロングワード（ $6 + 3 \times 1$ ）が含まれ、バイト 35 で終わります。次元数 7 の配列は、合計で 27 のロングワード（ $6 + 3 \times 7$ ）が含まれ、バイト 107 で終わります。

例えば、次のような宣言の場合を考えてみます。

```
integer, target :: a(10,10)
integer, pointer :: p(:, :)
p => a(9:1:-2, 1:9:3)
call f(p)

.
.
.
```

実引数 p の記述子は、次の値が含まれます。

- 最初のロングワード（バイト 0 から 3）には、ベースアドレスが含まれます（ランタイムで割り当てられます）。
- 2 番目のロングワード（バイト 4 から 7）は、4（1 つの要素のサイズ）に設定されます。
- 3 番目のロングワード（バイト 8 から 11）には、オフセットが含まれています（ランタイムで割り当てられます）。
- 4 番目のロングワード（バイト 12 から 15）には、1 が含まれます（下位ビットが設定されます）。
- 5 番目のロングワード（バイト 16 から 19）には、2 が含まれます（次元数）。
- 6 番目のロングワードは、予約済みです。
- 7、8、および 9 番目のロングワード（バイト 24 から 35）には、最初の次元に関する次の情報が含まれています。
 - 5（範囲）
 - -8（要素間の距離）
 - 9（下限）
- 10、11、および 12 番目のロングワード（バイト 36 から 47）には、2 番目の次元に関する次の情報が含まれています。
 - 3（範囲）
 - 120（要素間の距離）
 - 1（下限）
- この例では、バイト 47 が最後のバイトとなります。



注

Itanium[®] ベース・システムの指定子の書式は、IA-32 システムの書式と同じですが、すべてのフィールドが 4 バイトから 8 バイトになります。

文字列の処理

デフォルトでは、インテル® Fortran は、文字列に長さの隠れ引数を渡します。長さの隠れ引数は、符号なしの 4 バイト整数 (IA-32 システム) または 8 バイト整数 (Itanium® ベース・システム) で構成されます。常に、値により渡され、引数リストの最後に追加されます。デフォルトの文字列の渡され方は、属性を使用すると変更できます。

次の表は、渡される文字列におけるさまざまな属性の結果をまとめたものです。

属性として渡される文字列における属性プロパティの効果

引数	デフォルト	C	C, REFERENCE
文字列	長さとともに参照により渡される	最初の文字が INTEGER(4) に変換され、値により渡される	長さとともに参照により渡される
VALUE オプションを持つ文字列	エラー	最初の文字が INTEGER(4) に変換され、値により渡される	最初の文字が INTEGER(4) に変換され、値により渡される
REFERENCE オプションを持つ文字列	たいてい長さとともに参照により渡される	参照により渡される、長さは渡されない	参照により渡される、長さは渡されない

上記の表で重要な点を以下のとおりです:

- C ルーチンに渡される VALUE または REFERENCE の属性は、参照では渡されません。代わりに、最初の文字のみが値によって渡されます。
- C ルーチンに渡される VALUE オプションを持つ文字列は、参照では渡されません。代わりに、最初の文字の値のみが渡されます。
- デフォルトの ATTRIBUTES、ATTRIBUTES C、REFERENCE を持つ文字列引数の場合:
 - `-nomixed_str_len_arg` が設定されると、文字列の長さが (値により)、スタック上のすべての引数の後にプッシュされます。これはデフォルトで使用されます。
 - `-mixed_str_len_arg` が設定されると、文字列の長さが (値により)、スタック上のその文字列の開始アドレスの直後にプッシュされます。
- デフォルトの ATTRIBUTES とともに参照により渡される文字列引数の場合:

- `-nomixed_str_len_arg` が設定されると、呼び出されるプロシージャで文字列の長さは使用できません。これはデフォルトで使用されます。
- `-mixed_str_len_arg` が設定されると、文字列の長さが (値により)、スタック上のその文字列の開始アドレスの直後にプッシュされます。

C のすべての文字列はポインタであるため、C は文字列が文字列の長さを伴わずに参照により渡されることを想定します。また、Fortran 文字列はヌルでは終わらないのに対し、C 文字列はヌルで終わります。Fortran と C 間で文字列を渡す方法は基本的に 2 つあります。Fortran の文字列を C の文字列に変換する方法と Fortran の文字列を受け付ける C のルーチンを書く方法です。

Fortran 文字列を C 文字列に変換するには、参照により文字列を長さを伴わずに渡し、文字列をヌルで終わらせる属性の組み合わせを選択してください。次に例を示します。

```
INTERFACE
  SUBROUTINE Pass_Str (string)
    !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'Pass_Str' :: Pass_Str
    CHARACTER*(*) string
    !DEC$ ATTRIBUTES REFERENCE :: string
  END SUBROUTINE
END INTERFACE
CHARACTER(40) forstring
DATA forstring /'This is a null-terminated string.' C/
```

次の例は、Fortran DATA 文 (『Intel® Fortran Language Reference』(英語) の「C String」を参照) における文字列にヌル終端文字を使用した応用例です:

```
DATA forstring /'This is a null-terminated string.' C/
```

C インターフェイスは次のとおりです:

```
void Pass_Str (char *string)
```

C のルーチンで Fortran の文字列を受け取るには、C は文字列アドレスとともに文字長の引数が渡されることを念頭に置かなくてはなりません。次に例を示します。

```
!Fortran のコード
INTERFACE
  SUBROUTINE Pass_Str (string)
    CHARACTER*(*) string
  END INTERFACE
```

C のルーチンは、2 つの引数が渡されると考えなくてはなりません。

```
void pass_str (char *string, unsigned int length_arg )
```

このインターフェイスは長さの隠れ引数を処理しますが、ヌルで終わる C 文字列とヌルで終わらない Fortran 文字列との調整が必要です。また、Fortran の文字列に割り当てられたデータが宣言された長さよりも短かった場合、Fortran の文字列には空白がパディングされます。

Fortran と C の言語が混在したプログラミングでは、これらの文字列の違いを C のルーチンで処理するよりも、可能な限り C の文字列を採用することをお勧めします。

CHARACTER*(*) 構文を使用する文字列を返す Fortran 関数は、隠れ文字列引数と文字列の長さを引数リストの最初に配置します。

Fortran 関数コールなどを実装する C 関数はこの隠れ文字列引数を明示的に宣言しなければならず、またその引数を使用して値を返さなければなりません。C の戻り型は void です。ただし、文字列を返す関数は使用しないようにする方が、エラーの発生を避けやすいでしょう。できる限りサブルーチンを使用するか、モジュールやグローバル変数に文字列を配置してください。

文字データ型の戻り

Fortran プログラムで関数から CHARACTER 型のデータが返されると予測した場合、Fortran コンパイラは呼び出されたプロシージャの引数リストの先頭に 2 つの引数を追加します。

- 最初の引数は、呼び出されたプロシージャが結果を格納する位置を示すポインタです。
- 2 番目の引数は、返される文字の最大文字数です(必要に応じて空白をパディングします)。

呼び出されたルーチンは、最初の引数で指定したアドレスを介して結果をコピーしなければなりません。次に、MAKECHARS という名前の文字関数を返すための Fortran コードとそれに対応する C ルーチンの例を示します。

C から Fortran への文字型の戻り例

Fortran のコード

```
CHARACTER*10 CHARS, MAKECHARS
DOUBLE PRECISION X, Y
CHARS = MAKECHARS( X, Y )
```

対応する C ルーチン

```
void makechars ( result, length, x, y );
char *result;
int length;
```

```
double *x, *y;
{
...program text, producing returnvalue...
for (i = 0; i < length; i++ ) {
result[i] = returnvalue[i];
}
}
```

上記の例では、次の制限および動作が適用されます:

- 関数の長さや結果は、呼び出し文で指定するものではありません。コンパイラによって追加します。
- 呼び出されたルーチンは、`result` によって指定した位置に結果文字列をコピーしなければなりません。`length` よりも長い文字のコピーはできません。
- `length` よりも短い文字が返された場合、戻り位置の右側に空白がパディングされます。Fortran では、文字を終了するのにゼロは使用しません。
- 呼び出されたプロシージャは `void` 型です。
- 小文字を使用した呼び出しを行うために、C ルーチンまたは `INTERFACE` ブロックには、小文字の名前を使用しなければなりません。

ユーザ定義型の処理

Fortran 95/90 は、ユーザ定義型(C の構造体に似たデータ構造)をサポートしています。ユーザ定義型は他のデータ型と同じように、モジュールおよび共通ブロック内で渡すことができますが、他の言語は型の構造を知っている必要があります。

以下に例を示します:

Fortran のコード:

```
TYPE LOTTA_DATA
  SEQUENCE
  REAL A
  INTEGER B
  CHARACTER(30) INFO
  COMPLEX CX
  CHARACTER(80) MOREINFO
END TYPE LOTTA_DATA
TYPE (LOTTA_DATA) D1, D2
COMMON /T_BLOCK/ D1, D2
```

上の Fortran コードでは、SEQUENCE 文により、構造型定義を格納する順番が保持されます。

C のコード:

```
/* C code accessing D1 and D2 */
extern struct {
    struct {
        float a;
        int b;
        char info[30];
        struct {
            float real, imag;
        } cx;
        char moreinfo[80];
    } d1, d2;
} t_block;
```

インテル® Fortran/C が混在したプログラム

インテル® Fortran/C が混在したプログラムの概要

以下のトピックを参照してください:

[インテル® Fortran/C プログラムのコンパイルとリンク](#)

[Fortran と C が混在したプログラミングにおけるモジュールの使用](#)

[インテル® Fortran プログラムからの C プロシージャの呼び出し](#)

インテル® Fortran/C プログラムのコンパイルとリンク

アプリケーションには、C と Fortran の両方のソースファイルを含めることができます。メイン・プログラムが、C で記述されたルーチン (cfunc.c) を呼び出す Fortran ソー

スファイル (myprog.for) である場合、次のコマンド行を使用してアプリケーションをビルドします。

```
icc -c cfunc.c
ifort -o myprog myprog.for cfunc.o
```

icc (インテル® C++ 用) コマンドは、cfunc.c をコンパイルします。-c オプションは、リンカを呼び出さないことを指定します。このコマンドは cfunc.o を作成します。ifort コマンドは myprog.for をコンパイルし、cfunc.o を myprog.for から作成されたオブジェクト・ファイルにリンクして、myprog を作成します。

C/C++ プログラムでインテル Fortran サブプログラムを呼び出す場合、ifort コマンドラインで -nofor_main オプションを指定します。

```
icc -c cmain.c
ifort -nofor_main cmain.o fsub.f90
```

Fortran と C が混在したプログラミングにおけるモジュールの使用

インテル® Fortran モジュールは C/C++ から直接、アクセスすることができます。そのため、C 言語との間で多くの変数を交換するには、モジュール化が最も簡単な方法です。

次の例では、Fortran でモジュールが宣言されており、C からデータをアクセスできます：

Fortran のコード

```
! F90 Module definition
MODULE EXAMP
  REAL A(3)
  INTEGER I1, I2
  CHARACTER(80) LINE
  TYPE MYDATA
    SEQUENCE
    INTEGER N
    CHARACTER(30) INFO
  END TYPE MYDATA
END MODULE EXAMP
```

C のコード

```
\* C code accessing module data *\nextern float examp_mp_a[3];\nextern int examp_mp_i1, examp_mp_i2;\nextern char examp_mp_line[80];\nextern struct {\n    int n;\n    char info[30];\n} examp_mp_mydata;
```

C++ のコードが .cpp ファイルに存在する場合、C++ 言語のセマンティックスが外部名に適用されるため、リンカエラーが頻繁に発生します。この場合は、extern "C" 構文（詳細は、「[C/C++ 命名規則](#)」を参照してください）を使用します:

```
\* C code accessing module data in .cpp file*\nextern "C" float examp_mp_a[3];\nextern "C" int examp_mp_i1, examp_mp_i2;\nextern "C" char examp_mp_line[80];\nextern "C" struct {\n    int n;\n    char info[30];\n} examp_mp_mydata;
```

また、C 言語でモジュール・プロシージャを定義し、ALIAS ディレクティブを使用することで、Fortran モジュールのルーチン部分を作成できます。C のコードは次のとおりです:

```
// C procedure\nvoid pythagoras (float a, float b, float *c)\n{\n    *c = (float) sqrt(a*a + b*b);\n}
```

C++ のコードが .cpp ファイルに存在する場合は、上記の例のように extern "C" 構文（詳細は、「[C/C++ 命名規則](#)」を参照してください）を使用します:

```
// C procedure\nextern "C" void pythagoras (float a, float b, float *c)\n{\n    *c = (float) sqrt(a*a + b*b);\n}
```

CPROC モジュールを定義する Fortran コード:

```

! Fortran 95/90 Module including procedure
MODULE CPROC
  INTERFACE
    SUBROUTINE PYTHAGORAS (a, b, res)
      !DEC$ ATTRIBUTES C :: PYTHAGORAS
      !DEC$ ATTRIBUTES REFERENCE :: res
! res is passed by REFERENCE because its individual attribute
! overrides the subroutine's C attribute
      REAL a, b, res
! a and b have the VALUE attribute by default because
! the subroutine has the C attribute
    END SUBROUTINE
  END INTERFACE
END MODULE

```

CPROC モジュールを使用して上記のルーチンを呼び出す Fortran コード:

```

! Fortran 95/90 Module including procedure
USE CPROC
CALL PYTHAGORAS (3.0, 4.0, X)
TYPE *, X
END

```

インテル® Fortran プログラムからの C プロシージャの呼び出し

命名規則

デフォルトでは、Fortran コンパイラは関数名とサブプログラム名を大文字に変換します。しかし、C コンパイラは大文字と小文字の変換を行いません。このため、Fortran プログラムから呼び出される C プロシージャには、大文字と小文字を正確に使用して名前を付けなければなりません。例えば、次のような呼び出しの場合を考えてみます。

CALL PROCNAME ()	C プロシージャに、PROCNAME という名前を付けなければなりません。
X=FNNAME ()	C プロシージャに、FNNAME という名前を付けなければなりません。

最初の呼び出しの場合、PROCNAME によって返される値を無視します。2 番目の関数呼び出しでは、FNNAME は値を返さなければなりません。

Fortran と C プロシージャ間での引数の受け渡し

デフォルトでは、Fortran サブプログラムは引数を参照によって渡します。すなわち、Fortran サブプログラムは引数の値ではなく、それぞれの実引数のポインタを渡します。これに対して、C プログラムは値によって引数を渡します。このため、次の点に注意しなければなりません。

- Fortran プログラムから C 関数を呼び出す場合、この C 関数の仮引数はそれぞれ適切なデータ型のポインタとして宣言されていなければなりません。
- C プログラムから Fortran サブプログラムを呼び出す場合、それぞれの実引数はポインタとして明示的に指定されていなければなりません。

エラー処理

エラー処理の概要

次のトピックを参照してください。

[ランタイム・ライブラリのデフォルトのエラー処理](#)

[ランタイム・エラーの処理](#)

[信号処理](#)

[デフォルトのランタイム・ライブラリ例外ハンドラの上書き](#)

[TRACEBACKQQ によるトレースバック情報の取得](#)

ランタイム・ライブラリのデフォルトのエラー処理

プログラムの実行中に、エラーまたは例外条件が発生する場合があります。これらは、以下のような原因で発生します。

- I/O 操作中に発生するエラー

- 無効な入力データ
- 算術ライブラリの呼び出しにおける引数エラー
- 算術エラー
- 他のシステム検出エラー

インテル® Fortran ランタイム・ライブラリ (RTL) は、適切なメッセージを生成し、可能な場合はエラーから回復するための処置を実行します。

Fortran RTL で認識される各エラーに対して、デフォルトの処理が定義されています。明示的にエラー処理方法が指定されない限り、この章で説明されているデフォルトの処理が行なわれます。

Fortran RTL が実際にエラーを処理する方法は、以下の要因によって異なります。

- エラーの重要度。例えば、重要度が “warning” または “info” (情報) のエラー・メッセージが検出された場合、通常、プログラムは実行を続行します。
- I/O 文に関連する特定のエラーに対して、I/O エラー処理指定子が指定された場合と指定されなかった場合。
- 特定のエラーに対して、関連する信号のデフォルトの処理が変更された場合と変更されなかった場合。
- 算術演算に関係する特定のエラー (浮動小数点例外を含む) に対して、コンパイル・オプションによって、エラーを通知するかどうか、また通知するエラーの重要度を決定できます。

算術例外条件の通知および処理方法は、例外の原因とプログラムのコンパイル方法によって異なります。プログラムが例外を処理できるようにコンパイルされなければ、例外条件の原因となった命令の後まで例外は通知されない可能性があります。エラーと例外の処理に関するコンパイラ・オプションは次のとおりです。

- `-check bounds` オプションは、特定の条件を検出するためのコードを生成します。
- `-check noformat` オプションおよび `-check nooutput_conversion` オプションは、ランタイム・エラーの重要度を下げ、プログラムの実行を続行します。
- `-fopen` オプションは、ランタイム時の浮動小数点算術例外の処理方法および通知方法を制御します。
- `-warn xxxx`、`-u`、`-nowarn -w`、および `-w1` オプションは、コンパイル時の警告メッセージを制御します。コンパイル時の警告メッセージは、ランタイム・エラーの原因を判断するのに役立つ場合があります。

ランタイム・メッセージの形式

プログラム実行中 (ランタイム) にエラーが発生すると、Fortran RTL は診断メッセージを発行します。これらのランタイム・メッセージの形式は次のとおりです。

```
forrtl: severity (nnn): message-text
```

各アイテムの意味は次のとおりです。

- `forrtl` は、メッセージの発行元がインテル Fortran RTL であることを示します。
- `severity` は、重要度レベル (`severe`、`error`、`warning`、または `info`) を示します。
- `nnn` は、メッセージ番号を示します。また、I/O 文の `IOSTAT` の値でもあります。
- `message-text` は、メッセージの原因となったイベントの説明です。

次に、ランタイム・メッセージの重要度について、重要度の高いものから説明します。

- メッセージの重要度レベルが `severe` の場合、問題を修正する必要があります。プログラムの I/O 文で `END`、`EOR`、`ERR` 各分岐指定子を使用して、`IOSTAT` 指定子を使用するルーチンなどに制御を渡さない限り、プログラムの実行はエラー検出時に終了します。
- メッセージの重要度レベルが `error` の場合、問題を修正する必要があります。プログラムの実行を続行できても、出力に誤りがある可能性があります。
- メッセージの重要度レベルが `warning` の場合、問題を調査する必要があります。プログラムの実行は続行しますが、出力に誤りがある可能性があります。
- 重要度レベルが `info` のメッセージは、情報提供のみを目的としています。プログラムは続行されます。

重要度レベルが `severe` のエラーでは、環境変数 `FOR_DISABLE_STACK_TRACE` が設定されていない限り、スタックトレース情報がデフォルトで表示されます。-
`traceback` コマンド・ライン・オプションが設定されている場合、スタックトレース情報にはシンボリック情報に対して設定されたプログラム・カウンタが含まれています。それ以外の場合、スタックトレース情報には単なる 16 進数のプログラム・カウンタ情報が含まれます。

場合によっては、スタックトレース情報はコンパイルされたコードによってランタイム時に生成され、配列の一時変数の作成に関する詳細を提供します。

`FOR_DISABLE_STACK_TRACE` が設定されている場合、スタックトレース情報は生成されません。

次のスタックトレース情報に関する例を参照してください。プログラムは、12 行目でエラーを表示します。

```
program ovf
real*4 x(5),y(5)
integer*4 i

x(1) = -1e32
x(2) = 1e38
x(3) = 1e38
x(4) = 1e38
x(5) = -36.0

do i=1,5
  y(i) = 100.0*(x(i))
  print *, 'x = ', x(i), ' x*100.0 = ',y(i)
end do
end
```

```
> ifort -O0 -fpe0 -traceback ovf.f90 -o ovf.exe
> ovf.exe
```

```
x = -1.0000000E+32  x*100.0 = -1.0000000E+34      (1)
forrtl: error (72): floating overflow
Image            PC            Routine          Line        Source
ovf.exe           08049E4A      MAIN__          14         ovf.f90
ovf.exe           08049F08      Unknown         Unknown      Unknown
ovf.exe           400B3507      Unknown         Unknown      Unknown
ovf.exe           08049C51      Unknown         Unknown      Unknown
Abort
```

```
> setenv FOR_DISABLE_STACK_TRACE true
> ovf.exe
```

```
x = -1.0000000E+32  x*100.0 = -1.0000000E+34
forrtl: error (72): floating overflow      (2)
Abort
```

次の情報は、上記の例の右端に記述された番号に関する情報です。

(1) トレースバック情報が存在する場合のスタックトレース情報です。

(2) FOR_DISABLE_STACK_TRACE 環境変数が設定されているため、スタックトレース情報は生成されません。

プログラム終了時にシェルに返される値

インテル Fortran プログラムは、次のいずれかの方法で終了させることができます。

- プログラムが正常終了するまで実行する。ゼロの値がシェルに返されます。

- プログラムが STOP 文または PAUSE 文で中止する。ゼロの値がシェルに返されます。
- 信号が検出され、それがプログラムの続行を許容しないためにプログラムが中止する。1 の値がシェルに返されます。
- プログラムが重要なランタイム・エラーのために中止する。そのランタイム・エラーのエラー番号がシェルに返されます。
- プログラムが CALL EXIT 文で中止する。EXIT に渡された値がシェルに返されます。

重要なエラーのための強制コアダンプ

通常は core ファイルを作成しない重要なエラーに対して、強制的にコアダンプを実行することができます。プログラムを実行する前に、`decfort_dump_flag` 環境変数に True の値 (Y、y、Yes、yEs、True など) を設定して、重要なエラーで core ファイルを作成させます。例えば、次の C シェルコマンドで `decfort_dump_flag` 環境変数を設定します。

```
setenv decfort_dump_flag y
```

core ファイルは、カレント・ディレクトリに書き込まれ、デバッガを使用して調査できます。



注

重要なエラーに対してコアファイルを作成するように要求しているにも関わらず、ファイルが期待どおりに作成されない場合、プロセス制限でコアファイルの利用可能なサイズが低く設定されている (またはゼロに設定されている) 可能性があります。プロセス制限の設定に関する詳細は、シェルのメインページを参照してください。例えば、C シェルコマンド `limit` (引数なし) は、現在の設定を通知します。また、`limit coredumpsize unlimited` はコアファイルの利用可能なサイズを現在のシステムで可能な最大サイズまで許可します。

ランタイム・エラーの処理

インテル® Fortran RTL は、可能な限り適切なメッセージを生成したり、エラーから復旧するために必要な処置を実行するなどの何らかのエラー処理を実行します。次の方法で、デフォルト動作を明示的に補足したり、代えることができます。

- プログラム内のエラー処理コードに制御を移すには、I/O 文中で END、EOR、および ERR の分岐指定子を使用します。

- インテル Fortran RTL エラーコード値に基づいて、Fortran 特有の I/O エラーを識別するには、I/O 文中で I/O 状態指定子 (IOSTAT) を使用します (または ERRSNS サブルーチンを呼び出します)。
- 適切なライブラリ・ルーチンを使用してシステムレベルのエラーコードを取得します。
- 特定のエラー状態の場合、信号処理機能でデフォルト動作を変更してください。

これらのエラー処理方法は同時に使用することができます。プログラマは、同じプログラム内で任意の方法またはすべての方法を使って、インテル Fortran のランタイムのエラーコードと Linux* システムのエラーコードを取得することができます。

END、EOR、および ERR 分岐指定子の使用

インテル Fortran のプログラムの実行中に重大なエラーが起こった場合、デフォルト動作ではエラー・メッセージが表示され、プログラムが終了します。このデフォルト動作を変更するために、I/O 文には、プログラム中の特定の場所に制御を移す 3 つの分岐指定子が用意されています。

- END 分岐指定子は、ファイル終了条件を処理します。
- EOR 分岐指定子は、ノン・アドバンシング読み取りでのレコード終了条件を処理します。
- ERR 分岐指定子は、すべてのエラー状態を処理します。

END、EOR、または ERR 分岐指定子を使用すると、エラー・メッセージは表示されず、実行は指定された文 (通常はエラー処理ルーチン) で続行されます。

エラー処理ルーチンが処理できない予期しないエラーが発生することがあります。この場合、次のいずれかを行います。

- エラー処理ルーチンを変更して、エラー・メッセージ番号を表示するようにします。
- エラーを引き起こす I/O 文から END、EOR、または ERR の各分岐指定子を削除します。

ソースコードを変更したら、プログラムのコンパイル、リンク、実行を行い、エラー・メッセージを表示させます。次に例を示します。

```
READ (8, 50, ERR=400)
```

この文の実行中に重大なエラーが発生した場合、インテル Fortran RTL は制御を文番号 400 の文に移します。同じように、END 指定子を使うと、通常ならエラーとして処理されるファイル終了条件を処理することができます。次に例を示します。

```
READ (12,70,END=550)
```

ノン・アドバンシング I/O を使用しているときには、EOR 指定子を使ってレコード終了条件を処理します。次に例を示します。

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A,
    F, I
```

また、OPEN、CLOSE、または INQUIRE の文で、ERR を指定子として使用することもできます。次に例を示します。

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

この OPEN 文の実行中にエラーが検出されると、制御は文番号 999 の文に移ります。

IOSTAT 指定子の使用

IOSTAT 指定子を使うと、I/O エラーの発生後にプログラムの実行を続け、I/O 操作に関する情報を返すことができます。エラーによっては、IOSTAT に返されないものもあります。

IOSTAT 指定子は、END、EOR、および ERR による分岐転送を補足または置き換えることができます。IOSTAT 指定子を含む I/O 文が実行されると、エラー・メッセージの表示は禁止されます。また、整数変数、配列要素、またはスカラ・フィールド参照が次のいずれかに定義されます:

- ノン・アドバンシング読み取りの際にレコード終了条件が発生した場合、値 -2。
- ファイル終了条件が発生した場合、値 -1。
- 通常の完了の場合、値 0 (エラー状態、ファイル終了条件、レコード終了条件のいずれでもない場合)。
- エラー状態が発生した場合、正の整数値。(この値は、ランタイム時のエラー・メッセージのリストにある Fortran 固有の IOSTAT 番号の 1 つです。詳細は、「[ランタイム時のエラー・メッセージ](#)」を参照してください。

I/O 文の実行と IOSTAT 値の割り当てが行われた後、制御は (指定されている場合) END 文、EOR 文、または ERR 文の文番号に移ります。制御が移らない場合、通常の実行が続けられます。

プログラムに /opt/intel_fc_80/include/for_iosdef.for ファイルをインクルードすると、IOSTAT 値のシンボリック定義を利用できるようになります。

次の例は、IOSTAT 指定子と for_iosdef.for ファイルを使って、OPEN 文のエラー (FILE 指定子の中) を処理しています。

```

CHARACTER (LEN=40) :: FILNM
INCLUDE 'for_iosdef.for'
DO I=1,4
  FILNM = ''WRITE (6,*)

  'Type file name '
  READ (5,*) FILNM
  OPEN (UNIT=1, FILE=FILNM, STATUS='OLD', IOSTAT=IERR,
ERR=100)
  WRITE (6,*) 'Opening file: ', FILNM
! (process the input file) CLOSE (UNIT=1)STOP
100 IF (IERR .EQ.FOR$IOS_FILNOTFOU) THEN
  WRITE (6,*) 'File: ', FILNM, ' does not exist '
ELSE IF (IERR .EQ.FOR$IOS_FILNAMSPE) THEN
  WRITE (6,*) 'File: ', FILNM, ' was bad, enter new
file name'
ELSE
  PRINT *, 'Unrecoverable error, code =',
IERR STOPend if

END DO
WRITE (6,*) 'File not found.Type ls to find file and
run again' END PROGRAM END PROGRAM

```

エラーを取得する別の方法として、ERRSNS サブルーチンを使う方法があります。このサブルーチンは、インテル Fortran RTL エラー（詳細は、『Intel® Fortran Language Reference』（英語）を参照してください。）に関連した最後の I/O システムのエラーコードを取得します。

信号処理

信号は、次に示すような、さまざまな発信元から生成されるアブノーマルなイベントです:

- 端末のユーザ
- プログラムまたはハードウェアのエラー
- 別のプログラムからの要求
- 制御端末にアクセスできるようにプロセスが停止されたとき

例えば、次のような特定のイベントに対して、信号の発行を設定できます。

- プロセスが停止された後に、プロセスを再開するとき
- 子プロセスの状況が変化したとき
- 端末で入力可能な状態になったとき

何も処理が行われないと、受信プロセスを終了する信号（オプションで、core ファイルを作成します）と、プロセスが何か要求しない限り無視されるだけの信号があります。

特定の信号を除いて、`signal` または `sigaction` ルーチン呼び出すことによって、指定した信号を無視するか、またはユーザが記述した信号ハンドラの位置に割り込みを発生させる（制御を渡す）ことができます。

信号に対して `signal` を呼び出し、次のいずれかの処理を設定できます：

- 指定した信号（番号による指定）を無視する。
- 指定した信号に対してデフォルトの処理を行う。これによって、以前に設定した処理をリセットできます。
- 指定した信号から、名前によって指定された信号を受け取るためのプロシージャに制御を移す。

`signal` ルーチン呼び出すことによって、信号に対する処理を変更できます（オペレーティング・システムの信号を遮断して、プロセスの中断を防止するなど）。

プログラムを起動すると、インテル® Fortran RTL は以下の信号を検出するように準備します：

信号	インテル Fortran RTL メッセージ
SIGFPE	Floating-point exception (番号 75)
SIGINT	Process interrupted (number 69)
SIGIOT	IOT trap signal (number 76)
SIGQUIT	Process quit (number 79)
SIGSEGV	Segmentation fault (番号 174)
SIGTERM	Process killed (number 78)

`signal` ルーチン呼び出す（各信号の番号を指定）ことで、インテル Fortran RTL によって設定された信号処理機能が無視されます。デフォルトの処理を復元するには、最初の `signal` の呼び出しから返された値を保存する方法しかありません。

デバッガを使用している場合は、インテル Fortran RTL が適切な信号を受け取り、処理できるコマンドを入力しなければならないことがあります。

デフォルトのランタイム・ライブラリ例外ハンドラの上書き

デフォルトのランタイム・ライブラリ例外ハンドラを上書きするには、アプリケーションで目的の信号の処理を変更する `signal` を呼び出す必要があります。

例えば、信号の処理を `abort()` を呼び出して `core` ファイルを生成するように変更するとします。

次の例では、`clear_signal_` という関数を追加し、`signal()` を呼び出して `SIGABRT` 信号の処理を変更します:

```
#include <signal.h>
void clear_signal_()
{
    signal (SIGABRT, SIG_DFL);
}
int myabort_()
{
    abort();
    return 0;
}
```

ローカルの `clear_signal()` ルーチンの呼び出しをメインに追加する必要があります。この場合、呼び出しがローカルの `myabort()` ルーチンの呼び出しの前に記述されるようにします:

```
program aborts
integer i

call clear_signal()

i = 3
if (i < 5) then
    call myabort()
end if
end
```

TRACEBACKQQ によるトレースバック情報の取得

`TRACEBACKQQ` ルーチンを呼び出すことにより、トレースバック情報を取得できます。

TRACEBACKQQ は、アプリケーションのスタックトレースを開始します。このルーチンを使用することで、検出したアプリケーション・エラーを報告し、この情報を基にデバッグなどを行えます。インテル® Fortran ランタイム・システムの標準スタックトレース・サポートが使用されているので、ランタイム・システムが生成する未処理エラーと例外の出力（重大なエラー・メッセージ）と同一のものが生成されます。TRACEBACKQQ サブルーチンは、TRACEBACKQQ への呼び出しがあった時点までのプログラムのコールスタックを含むスタックトレースを生成します。

ランタイム・サポートから提供されるエラー・メッセージの文字列は、ユーザが指定したメッセージ・テキストに置き換えることができ、また、ユーザが指定しない場合は、省略されます。トレースバック出力は、ランタイム・サポートによって内部でトレースバックが開始されたときと同じように、アプリケーション形式に合った出力先に送られます。

たいていの場合、ユーザは引数なしで TRACEBACKQQ を呼び出し、スタックトレースを生成できます。

```
CALL TRACEBACKQQ()
```

呼び出し場所がどこであっても、この呼び出しにより、ランタイム・ライブラリがヘッダ・メッセージを省略してトレースバックを生成し、実行を終了させます。

ユーザは引数を指定することで、ヘッダとして定義した文字列を使用してスタックトレースを生成したり、アプリケーションの実行を終了する代わりに、呼び出し側へ制御を返して実行を続行させることが可能です。以下に例を示します：

```
CALL TRACEBACKQQ (STRING="Done with pass 1", USER_EXIT_CODE=-1)
```

ユーザ終了コードを -1 に指定すると、呼び出し元のプログラムに制御が返ります。正の値をユーザ終了コードに指定した場合は、その値をオペレーティング・システムに返すように要求します。デフォルト値は 0 で、アプリケーションの実行を終了させます。

ライブラリの作成と使用

ライブラリの使用と作成の概要

次のトピックを参照してください：

[ライブラリの作成](#)

[インテル® Fortran が提供するライブラリ](#)

移植ライブラリの概要

数値演算ライブラリ

ライブラリの作成

ライブラリとは、インデックスの付いたオブジェクト・ファイルのコレクションです。ライブラリは、リンクされたプログラムで必要な場合にインクルードされます。オブジェクト・ファイルとライブラリを組み合わせることで、ソースを公開せずにコードを簡単に配布することができます。また、より少ないコマンドラインのエントリで、プロジェクトをコンパイルすることができます。

スタティック・ライブラリ

スタティック・ライブラリを使用して生成された実行ファイルは、個々のソースまたはオブジェクト・ファイルから生成された実行ファイルと同じです。スタティック・ライブラリは、ランタイムでは必要ないため、実行ファイルを配布する際に含める必要はありません。一般的に、コンパイル時には個々のソースファイルをリンクするより、スタティック・ライブラリをリンクする方が早く処理することができます。

スタティック・ライブラリをビルドするには、次のように操作します。

1. `-c` オプションを使用して、ソースファイルからオブジェクト・ファイルを生成します。
`ifort -c my_source1.f90 my_source2.f90 my_source3.f90`
2. GNU の `ar` ツールを使用して、オブジェクト・ファイルからライブラリを作成します。
`ar rc my_lib.a my_source1.o my_source2.o my_source3.o`
3. プロジェクトをコンパイルして、新しく作成したライブラリをリンクします。
`ifort main.f90 my_lib.a`

ライブラリ・ファイルとソースファイルが異なるディレクトリにある場合、`-Ldir` オプションを使用して、ライブラリのディレクトリを指定します。

```
ifort -L/for/libs main.f90 my_lib.a
```

共有ライブラリ

共有ライブラリは、ダイナミック・ライブラリまたはダイナミック共有オブジェクト (DSO) とも呼ばれ、スタティック・ライブラリとは異なる形でリンクされます。コンパイル時に、リンクはすべての必要なシンボルを実行ファイルにリンクするか、実行時に共有ライブラリからリンクされるようにします。共有ライブラリを使用してコンパイルされた実行

ファイルのサイズは小さくなりますが、共有ライブラリに含まれている関数が正しく動作するように、実行時に共有ライブラリをロードする必要があります。複数のプログラムで同じ共有ライブラリを使用している場合、ライブラリの 1 つのコピーをロードするだけでかまいません。

共有ライブラリをビルドするには、次のように操作します。

1. `-fPIC` および `-c` オプションを使用して、ソースファイルからオブジェクト・ファイルを生成します。
`ifort -fPIC -c my_source1.f90 my_source2.f90 my_source3.f90`
2. `-shared` オプションを使用して、オブジェクト・ファイルからライブラリを作成します。
`ifort -shared my_lib.so my_source1.o my_source2.o my_source3.o`
3. プロジェクトをコンパイルして、新しく作成したライブラリをリンクします。
`ifort main.f90 my_lib.so`

「[共有ライブラリの作成](#)」も参照してください。

インテル® Fortran が提供するライブラリ

インテル® Fortran はスタティックまたは DLL、シングルスレッドまたはマルチスレッドなど異なるタイプのライブラリを特定のライブラリ用に提供します。

次の表では、コンパイラ提供のライブラリを列挙したものです。

ファイル	説明
crtxi.o crtxn.o	C の初期化のサポート
for_main.o	Fortran プログラムのメイン・ルーチン
icrt.internal.map icrt.link	C リンクのサポート
ifcore_msg.cat	Fortran ランタイム・ライブラリ用エラー・メッセージ・カタログ (バージョン 8.1 以降では使用されていません)
libcprts.a libcprts.so libcprts.so.5	C++ 標準言語ライブラリ
libcxa.a libcxa.so libcxa.so.5	I/O データの位置を示す C++ 言語ライブラリ
libcxaguard.a libcxaguard.so libcxaguard.so.5	<code>-cxxlib-gcc</code> オプションとの相互運用に使用

libguide.a libguide.so	パラライザ・ツール用 OpenMP* スタティック・ライブラリ
libguide_stats.a libguide_stats.so	パラライザ・ツールの機能およびプロファイル情報のサポート
libifcore.a libifcore.so libifcore.so.5	インテル固有の Fortran ランタイム・ライブラリ
libifcoremt.a libifcoremt.so libifcoremt.so.5	マルチスレッド・インテル固有 Fortran ランタイム・ライブラリ
libifport.a libifport.so libifport.so.5	移植性および POSIX のサポート
libimf.a libimf.so	数値演算ライブラリ
libirc.a libirc_s.a	インテル固有のライブラリ (最適化)
libompstub.a	OMP の未使用時に OMP サブルーチンの参照を解決するライブラリ
libsvml.a	SVML (Short Vector Mathematical Library)
libunwind.a libunwind.so libunwind.so.5	アンワインドのサポート

移植ライブラリ

移植ライブラリの概要

インテル® Fortran には、各種プラットフォームから PC への移植を容易にしたり、PC で他のプラットフォームとの互換性があるコードを作成できるようにするための関数とサブルーチンが用意されています。移植ライブラリは `libifport.a` と呼ばれます。頻繁に使用される関数は、IFPORT という名前の移植モジュールに含まれています。

次のトピックを参照してください。

[libifport.a 移植ライブラリの使用](#)

[移植ルーチン](#)

libifport.a 移植ライブラリの使用

移植ライブラリ `libifport.a` は次の 2 つの方法で使用できます。

- プログラムに `USE IFPORT` 文を追加します。この文には、移植ライブラリ `libifport.a` が含まれています。
- 正しいパラメータと戻り値を使って、移植ルーチン呼び出します。

デフォルトで、`libifport.a` はリンク時にリンクに渡されます。リンクに `libifport.a` が渡されないようにするには、`-fpscomp nolib` オプションを指定します。

`libifport.a` 移植ライブラリを使用すると、ルーチン用のインターフェイス・ブロックおよびパラメータ定義が提供されます。また、コンパイラが呼び出しの検証を行えるようになります。

このライブラリの一部のルーチンは、異なる変数のセットを呼び出すことができ、サブルーチンとしてではなく関数として呼び出すこともできます。このような場合、引数および呼び出しメカニズムがルーチンの意味を決定します。`libifport.a` 移植ライブラリには、これらのルーチンのプロシージャ定義を提供する汎用のインターフェイス・ブロックが含まれます。

Fortran 95/90 には、多くの移植関数用の組込みプロシージャが含まれています。移植ルーチンは、Fortran 95 標準の拡張機能です。新しいコードを作成する際は、Fortran 95/90 組込みプロシージャを可能な限り使用してください（移植性およびパフォーマンスの理由から）。

移植ルーチン

ここでは、いくつかの移植ルーチンおよびその使用方法について説明します。

ルーチンの一覧は、『インテル® Fortran ライブラリ・リファレンス』マニュアルの概要の章にある「移植ルーチン」の表を参照してください。

情報取得ルーチン

これらのルーチンは、システムのコマンド、コマンドラインからの引数、環境変数、およびプロセスまたはユーザ情報を返します。

グループ、ユーザ、およびプロセス ID は INTEGER(4) の変数です。ログイン名およびホスト名は文字変数です。GETGID および GETUID 関数は、移植性のために提供されていますが、常に 1 を返します。

プロセス制御ルーチン

これらのプロセス制御ルーチンは、プロセスまたはサブプロセスの操作を制御します。SLEEP または ALARM を使用することでサブプロセスの操作が完了するまで待機し、その進行状況を監視して、KILL を経由して信号を送信し、ABORT でプロセスの実行を停止することができます。

KILL は、必ずしもプログラムの実行を停止するわけではありません。その代わりに、信号を確認し、渡されたコードに応じて適切な操作を行うハンドラルーチンを、信号を受信するルーチンに含めることができます。

SYSTEM を使用した場合、コマンドは別のシェルで実行されることに注意してください。SYSTEM 関数で設定された、現在の作業ディレクトリまたは環境変数などのデフォルト設定は、呼び出し側のプログラムが実行されている環境には影響しません。

この移植ライブラリには FORK ルーチンは含まれていません。Linux* システムでは、FORK は親プロセスの複製イメージを作成します。子プロセスおよび親プロセスは、それぞれ固有のリソースのコピーを使用し、互いに独立したプロセスになります。

数値および変換ルーチン

これらの数値および変換ルーチンは、ベッセル関数における演算、データ型の変換、および乱数の生成に利用することができます。

これらの関数のいくつかは、標準の Fortran 95/90 に相当します。データ・オブジェクトの変換は、LONG または SHORT の代わりに INT 組込み関数を使用して行うことができます。RANDOM_NUMBER および RANDOM_SEED 組込みサブルーチンは、数値および変換ルーチンの表に示す乱数関数と同じ関数を実行します。

AND、XOR、OR、LSHIFT、および RSHIFT などの他のビット操作関数は組込み関数です。これらの関数にアクセスするために、IFPORT モジュールを使用する必要はありません。標準の Fortran 95/90 には多くのビット演算ルーチンが含まれています。『Language Reference』(英語) マニュアルの第 9 章「Category Bit」にある表 9-2 に、これらのルーチンのリストが示されています。

入出力ルーチン

移植ライブラリには、ファイルのプロパティを変更し、文字およびバッファの読み書きを行い、ファイルのオフセット位置を変更するルーチンが含まれています。これらの入出力ルーチンは、次の点を考慮した上で READ または WRITE などの標準の Fortran 入出力文で、同じファイルに対して使用することができます。

- 直接ファイルで使用する場合、FSEEK、GETC、または PUTC 操作を行った後のレコード番号は次のレコード全体の番号となります。そのユニットで次に行われる通常の Fortran I/O は、次のレコード全体に対して行われます。例えば、レコード長が 10 のファイルの絶対位置 1 を検索した場合、INQUIRE によって返された NEXTREC の値は 2 となります。絶対位置 10 を検索した場合でも、NEXTREC の値は 2 となります。
- CARRIAGECONTROL='FORTRAN' (デフォルト設定) のユニットでは、PUTC と FPUTC の文字が列 1 に現れる場合、キャリッジ制御文字として処理されます。
- シーケンシャルの書式付きユニットでは、C 言語でキャリッジ・リターン/ラインフィードを表す文字列 "\n" を、ラインフィードまたは CHAR(10) ではなく、CHAR(13) (キャリッジ・リターン) および CHAR(10) (ラインフィード) として記述します。入力時には、10 (ラインフィード) の後に続く 13 (キャリッジ・リターン) は、10 として返されます。(ICHAR('\n') で示される ASCII 値が 10 である文字列 "\n" の長さは 1 文字です。)
- 直接ファイルでは、読み書きはそのままの形式で行われます。レコード間の分離記号は、読み込みおよび上書きすることができます。したがって、ファイルを直接ファイルとして使用する場合は、慎重に操作を行ってください。

これらのルーチンを使用して発生した I/O エラーは、インテル Fortran のランタイム・エラーの原因となります。

移植ファイル I/O ルーチンのいくつかは、標準 Fortran 95/90 に相当しています。例えば、ACCESS 関数を使用して、名前指定されたファイルに対し、モードに従った参照が可能かどうかを確認できます。ACCESS 関数は、ファイルが存在するかどうかを確認するとともに、ファイルに対する読み取り許可、書き込み許可、または実行許可があるかどうかをテストします。この関数は、プログラムの OPEN 文で指定されたファイル属性ではなく、ディスク上に存在するファイル属性を使用します。

ACCESS の代わりに、INQUIRE 文で ACTION パラメータを使用することで、同じ情報を取得することができます。ACCESS 関数は、FAT ファイルの読み取り許可には 0 を返します。これは、すべてのファイルに対して読み込み許可があることを示します。

日付および時刻ルーチン

さまざまな日付および時刻ルーチンを使用して、システム時刻を決定したり、時刻を現地時刻、グリニッジ標準時、日付と時刻要素の配列、または ASCII 文字列に変換することができます。

TIME および DATE は、関数またはサブルーチンのどちらとしても使用できます。名前が重複しているため、プログラムに USE IFPORT 文が含まれていない場合、個別にコンパイルされた各プログラムは、これらのバージョンから 1 つしか使用できません。例えば、プログラムが TIME サブルーチンを 1 度呼び出している場合、TIME を関数として使用することはできません。

標準 Fortran 95/90 には、新しい日付と時刻の組込みサブルーチンが含まれています。

エラー処理ルーチン

エラー処理ルーチンは、エラーを検出して、エラーのレポートを出力します。

IERRNO エラーコードは、UNIX システムの `errno` と類似しています。IFPORT モジュールは、通常 UNIX システムの `errno.h` に含まれている多くの UNIX の `errno` 名のパラメータ定義を提供します。

IERRNO はエラーが発生した場合にのみ更新されます。例えば、GETC 関数の呼び出しによってエラーが発生したが、その後で 2 回呼び出した PUTC が成功した場合、IERRNO を呼び出すと GETC 呼び出しのエラーが返されます。IERRNO は、移植ライブラリ・ルーチンを呼び出した直後に確認してください。その他の標準 Fortran 90 ルーチンによって、値が未定義の値に変更される可能性があります。

アプリケーションがマルチスレッドを使用する場合、IERRNO は各スレッドごとに設定されることに注意してください。

システム、ドライブ、またはディレクトリ制御および照会ルーチン

デバイス、ディレクトリ、およびファイルに関する情報は、これらのルーチンで取得することができます。

標準 Fortran 90 は、ファイル名またはユニット番号でファイルの詳細情報を返す INQUIRE 文を提供します。INQUIRE は、FSTAT、LSTAT、または STAT と同様に

使用してください。LSTAT および STAT は同じ情報を返します。STAT 関数の使用を推奨します。

その他のルーチン

移植ルーチンは、プログラムの呼び出しや制御、キーボードやスピーカ、ファイル管理、配列、浮動小数点での照会や制御、および IEEE* 機能などにも使用することができます。詳細は、『インテル® Fortran ライブラリ・リファレンス』マニュアルの概要の章にある「移植ルーチン」の表を参照してください。

数値演算ライブラリ

libimf.a は、インテルが提供する数値演算ライブラリで libm.a は、gcc* により提供される数値演算ライブラリです。

いずれのライブラリもデフォルトにより、IA-32 および Itanium® コンパイラ上でリンクされます。いずれのライブラリもリンクされるのは、インテルの数値演算ライブラリではない GNU 数値演算ライブラリによりサポートされる数値演算関数のためです。このリンクの処理により GNU ユーザは ifort の使用時にすべての関数を利用でき、サポートされている場合にはインテルの最適化バージョンが有効になります。

libimf.a は、libm.a の前にリンクされます。最初に libm.a をリンクすると、使用される数値演算関数のバージョンが変更されます。

libimf.a は LD_LIBRARY_PATH 変数内で指定した最初のディレクトリに配置することをお勧めします。libimf.a、libm.a ライブラリは、常に Fortran プログラムとリンクされます。

例えば、ライブラリを /perform/ ディレクトリに配置する場合は、他のすべてのライブラリを含むディレクトリの並びをセミコロンで区切って指定するように、LD_LIBRARY_PATH 変数を設定します。

IA-32 コンパイラでの libimf.a

IA-32 コンパイラの場合、libimf.a は汎用数値演算ルーチンおよびインテル® Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサで特別な使用のために最適化される数値演算ルーチンのバージョンの両方を含みます。

Itanium コンパイラでの libimf.a

Itanium コンパイラ の場合、libimf.a は Itanium アーキテクチャで使用するために最適化されます。コンパイラは、インライン化された数値演算ライブラリ・プリミティブを提供し、生成されたコードをスケジューリングします。これにより、一般的な浮動小数点アプリケーションのパフォーマンスを向上することが可能です。

参照情報

コンパイル時の環境変数

コンパイル時の環境変数:

- `FPATH`
インクルード・ファイルおよびモジュール・ファイルのパス
- `IFORTCFG`
デフォルトの設定ファイルの代わりに使用する設定ファイル
- `LD_LIBRARY_PATH`
共有ライブラリ (.so) ファイルのパス
- `PATH`
コンパイラの実行ファイルのパス
- `TMP`、`TMPDIR`、`TEMP`
一時ファイルの格納ディレクトリを指定します。一時ファイルについては、「[コンパイラまたはリンカにより作成される一時ファイル](#)」を参照してください。

ランタイム時の環境変数

インテル® Fortran ランタイム・システムはいくつかの環境変数を認識します。これらの変数は、ランタイム時の診断エラー・レポートをカスタマイズするのに使用できます。

ランタイム環境変数の例:

- `decfort_dump_flag`
この変数が `Y` または `y` に設定される場合、重大なインテル Fortran ランタイム・エラーが発生した際に、コアダンプを行います。
- `F_UFMENDIAN`
この変数は、リトル・エンディアンからビッグ・エンディアンへの変換に使用されるユニット数を指定します。詳細は、「[環境変数 F_UFMENDIAN を使用する方法](#)」を参照してください。
- `FOR_ACCEPT`
`ACCEPT` 文では明示的な論理ユニット番号は指定されません。その代わりに、

ACCEPT 文は暗黙の内部的な論理ユニット番号と FOR_ACCEPT 環境変数が使用されます。FOR_ACCEPT が指定されていない場合、ACCEPT f, iolist というコードは stdin (標準入力) から読み取りを行います。FOR_ACCEPT が定義されている場合は (オプションとしてパスを含むファイル名)、指定されたファイルが読み取られます。

- FOR_DIAGNOSTIC_LOG_FILE
ファイル名が設定されていた場合、指定されたファイルに診断出力を書き出します。
Fortran ランタイム・システムは、そのファイルを開き (アペンド出力)、ファイルにエラー情報 (ASCII テキスト) を書き出そうと試みます。
FOR_DIAGNOSTIC_LOG_FILE の設定は
FOR_DISABLE_DIAGNOSTIC_DISPLAY からは独立しているため、画面上での情報の表示を無効にすると同時に、エラー情報をファイルに収集することができます。ファイル名として割り当てたテキスト文字列はそのままの形で使用されるので、完全な名前を指定する必要があります。ファイルを開くことに失敗すると、エラーは報告されず、ランタイム・システムは診断処理を続行します。
- FOR_DISABLE_DIAGNOSTIC_DISPLAY
すべてのエラー情報の表示を無効にします。
この変数は、プログラムのエラー状態をテストすることだけが目的で、Fortran のランタイム・システムにプログラムの異常終了に関する情報は表示させたくない場合に便利です。
- FOR_DISABLE_STACK_TRACE
この変数を使用すると、重大なエラー・メッセージ・テキストの表示の後に続くコール・スタック・トレース情報が表示されなくなります。
Fortran のランタイム時エラー・メッセージは、
FOR_DISABLE_STACK_TRACE が True に設定されているかどうかとは関係なく表示されます。
- FOR_IGNORE_EXCEPTIONS
True に設定されていると、「Just-in Time」デバッグの許可といったデフォルトのランタイム例外処理が無効になります。ランタイム・システムの例外ハンドラはオペレーティング・システムに対して EXCEPTION_CONTINUE_SEARCH を返し、オペレーティング・システムはこの例外を処理する他のハンドラを探します。
- FOR_NOERROR_DIALOGS
True に設定されていると、特定の例外またはエラーが起こったときに、ダイアログボックスの表示を無効にします。これは、多数のテスト・プログラムをバッチモードで実行するときに、実行が失敗したためにテスト・ストリーム全体が停止することを防ぐ場合に便利です。
- FOR_PRINT
PRINT 文でも WRITE 文でも論理ユニット番号の代わりにアスタリスク (*) を指定すると、明示的な論理ユニット番号は使用されません。その代わりに、どちらの文でも暗黙的な内部論理ユニット番号と FOR_PRINT 環境変数が使用されます。FOR_PRINT が定義されていなければ、PRINT f, iolist また

は `WRITE (*,f) iolist` というコードは、`stdout` (標準出力) に書き出しを行います。`FOR_PRINT` が定義されていれば (オプションとしてパスを含むファイル名)、指定されたファイルが読み取られます。

- `FOR_READ`
`READ` 文で、ユニット番号の代わりにアスタリスク (*) を指定する場合、明示的な論理ユニット番号は使用されません。その代わりに、暗黙的な内部論理ユニット番号と `FOR_READ` 環境変数が使用されます。`FOR_READ` が定義されていない場合、`READ (*,f) iolist` または `READ f,iolist` というコードは `stdin` (標準入力) から読み取りを行います。`FOR_READ` が定義されている場合、(オプションとしてパスを含むファイル名)、指定されたファイルが読み取られます。
- `FOR_TYPE`
`TYPE` 文では明示的な論理ユニット番号は指定されません。その代わりに、暗黙的な内部論理ユニット番号と `FOR_TYPE` 環境変数が使用されます。`FOR_TYPE` が定義されていない場合、`TYPE f,iolist` というコードは `stdout` (標準入力) に書き出しを行います。`FOR_TYPE` が定義されている場合 (オプションとしてパスを含むファイル名)、指定されたファイルへ書き出されます。
- `FORT_BUFFERED`
 端末への出力を除いて、ランタイム時のすべての Fortran I/O ユニットの出力にはバッファ I/O を使用するように要求します。これは、[-assume buffered_io コンパイラ・オプション](#) をサポートするランタイム・メカニズムを提供します。
- `FORT_CONVERTn`
 特定のユニット番号 (*n*) に関連付けられた書式なしファイルのデータ書式を指定することができます。詳細は、「[データ書式の指定方法: 概要](#)」と「[環境変数 FORT_CONVERTn を使用する方法](#)」を参照してください。
- `FORT_CONVERT.ext` and `FORT_CONVERT_ext`
 書式なしファイルに対してデータ書式を特定のファイル拡張子 (接尾辞)(*ext*) で指定します。詳細は、「[データ書式の指定方法: 概要](#)」と「[環境変数 FORT_CONVERT.ext or FORT_CONVERT_ext を使用する方法](#)」を参照してください。
- `FORTn`
 コンパイラ・オプション `-fpscomp filesfromcmd` が指定されておらず、`OPEN` 文でファイル名が指定されていないか、暗黙の `OPEN` が使用されたときに、特定のユニット番号 (*n*) に使用するファイル名を指定することができます。ユニット番号 0、5、および 6 に事前に接続されているファイルは、デフォルトではシステムの標準 I/O ファイルに関連付けられています。
- `NLSPATH`
 インテル Fortran のランタイム・エラー・メッセージ・カタログのパスです。
- `TBK_ENABLE_VERBOSE_STACK_TRACE`
 エラー発生時に、詳細なコールスタック情報を表示します。
 通常は、デフォルトの簡単な出力だけで、エラーの発生場所を特定することが

できます。この簡単な出力には、スタックフレーム 1 つにつき 1 行で 20 個までのスタック・フレームが含まれます。各フレームについて、イメージの名前に続き、PC、ルーチン名、行番号、およびソースファイルが表示されます。

詳細出力を選択すると、簡単な出力で表示される情報に加えて、エラーが機械語例外だった場合には例外コンテキスト・レコードが表示され (マシン・レジスタ・ダンプ)、個々のフレームについてリターンアドレス、フレームポインタとスタックポインタ、およびルーチンに対するパラメータ (存在する場合) が表示されます。この出力はかなり長くなることがあり (ただし 16K バイトに制限されます)、出力を正確に記録したい場合には環境変数

FOR_DIAGNOSTIC_LOG_FILE を使用することを推奨します。多くの場合、詳細出力を使用する必要はありません。

FOR_ENABLE_VERBOSE_STACK_TRACE 変数もまた、Compaq* Fortran との互換性のために認識されます。

- TBK_FULL_SRC_FILE_SPEC

この変数は、トレースバック出力にパスを含む完全なファイル名を表示します。デフォルトでは、トレースバック出力はソース・ファイル・フィールドにファイル名と拡張子しか表示しません。この変数は、詳細を表示する場合に設定します。FOR_FULL_SRC_FILE_SPEC 変数 もまた、Compaq* Fortran との互換性のために認識されます。

- TMP、TMPDIR、および TEMP

一時ファイルの作成場所に、代わりの作業ディレクトリを指定します。詳細は、[「コンパイラまたはリンカにより作成される一時ファイル」](#)を参照してください。

主な IA-32 コンパイラ・ファイルの概要

次の表は、/opt/intel_fc_80/bin 内の IA-32 コンパイラにインストールされるファイルの一覧です。

ファイル	説明
codecov	コード・カバレッジ・ツール用実行ファイル
fortcom	コンパイラによって使用する実行ファイル
fpp	Fortran プリプロセッサ
ifc	以前のバージョンとの互換性用ファイル
ifc.cfg	以前のバージョンとの互換性用ファイル
ifort	インテル® Fortran コンパイラ・バージョン 8
ifortbin	コンパイラによって使用する実行ファイル
ifort.cfg	設定ファイル名
ifortvars.csh	C シェル用のセットアップ・ファイル
ifortvars.sh	bash シェル用のセットアップ・ファイル

profmerge	プロファイルに基づく最適化に使用するユーティリティ
proforder	プロファイルに基づく最適化に使用するユーティリティ
tselect	テスト・プライオリタイゼーション・ツール
uninstall.sh	アンインストール・ユーティリティ
xiar	プロシージャ間の最適化に使用するツール
xild	プロシージャ間の最適化に使用するツール

/lib へインストールされるファイルの一覧は、「[Intel® Fortran が提供するライブラリ](#)」を参照してください。

主な Itanium® コンパイラ・ファイルの概要

次の表は、/opt/intel_fc_80/bin 内の Itanium® コンパイラにインストールされるファイルの一覧です。

ファイル	説明
codecov	コード・カバレッジ・ツール用実行ファイル
efc	以前のバージョンとの互換性用ファイル
efc.cfg	以前のバージョンとの互換性用ファイル
efcbin	以前のバージョンとの互換性用ファイル
fortcom	コンパイラによって使用する実行ファイル
fpp	Fortran プリプロセッサ
ias	アセンブラ
ifort	Intel® Fortran コンパイラ・バージョン 8
ifort.cfg	設定ファイル
ifortbin	コンパイラによって使用する実行ファイル
ifortvars.csh	C シェル用のセットアップ・ファイル
ifortvars.sh	bash シェル用のセットアップ・ファイル
profdcg	プロファイルに基づく最適化に使用するユーティリティ
profmerge	プロファイルに基づく最適化に使用するユーティリティ
proforder	プロファイルに基づく最適化に使用するユーティリティ

tselect	テスト・プライオリタイゼーション・ツール
uninstall.sh	アンインストール・ユーティリティ
xiar	プロシージャ間の最適化に使用するツール
xild	プロシージャ間の最適化に使用するツール

/lib へインストールされるファイルの一覧は、「[インテル® Fortran が提供するライブラリ](#)」を参照してください。

コンパイラの制限

データの記憶容量、配列の大きさ、および実行ファイルの合計サイズは、システム・パラメータによって定められている、使用可能なプロセス仮想アドレス空間の量によってのみ制限されます。

プログラムのサイズ、コードの複雑さなどの、単一のインテル® Fortran プログラム・ユニットがもつ制限と、プログラム内の個々の文が持つ制限を以下の表に示します:

言語要素	制限
CALL または関数参照ごとの引数の実数	メモリの制限に依存
宣言式における関数参照の引数の数	255
配列の次元	7
配列構造のネスト数	20
次元ごとの配列要素数	9,223,372,036,854,775,807 = 2**31-1 (IA-32 システム) 2**63-1 (Itanium® ベース・システム) + メモリ構成による制限
定数: 文字定数および Hollerith 定数	7198 文字
定数: リスト指定 I/O での入力文字数	2048 文字
継続行数	511
データおよび I/O を含む DO のネスト数	7
DO と 整構造 IF 文のネスト (結合された) 数	128
DO ループのインデックス変数	9,223,372,036,854,775,807 = 2**63-1

編集記述子のネスト数	8
書式文の長さ	2048 文字
Fortran ソース行の幅	固定形式: 72 文字 (または - <code>extend_source</code> が有効な場合は 132 文字) 自由形式: 7200 文字
INCLUDE ファイルのネスト数	20 レベル
計算型または割り当て型 GOTO リストのラベル数	メモリの制限に依存
文ごとの字句トークン数	20000
名前付き共通ブロック数	メモリの制限に依存
配列構造を含む DO のネスト数	7
入力/出力を含む DO のネスト数	7
インターフェイス・ブロックの ネスト数	メモリの制限に依存
DO、IF または CASE 構文の ネスト数	メモリの制限に依存
括弧形式のネスト数	メモリの制限に依存
数値定数の桁数	メモリの制限に依存
式中の括弧のネスト数	メモリの制限に依存
構造体のネスト数	30
シンボル名の長さ	63 文字

大きなデータ・オブジェクトを処理する際のメモリ制限に関する詳細は、製品リリース・ノートを参照してください。

16 進、2 進、8 進、および 10 進の間での 変換

次の表は、16 進、2 進、8 進、および 10 進の間での変換を列挙したものです。

16 進	2 進	8 進	10 進
0	0000	00	0
1	0001	01	1
2	0010	02	2

3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
A	1010	12	10
B	1011	13	11
C	1100	14	12
D	1101	15	13
E	1110	16	14
F	1111	17	15

以前のバージョンのインテル® Fortran との互換性

このトピックは、インテル® Fortran バージョン 7.1 またはそれ以前のバージョンに精通し、現在バージョン 8.x を使用している開発者を対象としています。

インテル Fortran では、ISO および ANSI 標準の拡張をサポートしています。これには、次のシステムによって定義されている拡張が含まれます。

- 各種プラットフォーム用のインテル Fortran
- Microsoft* Fortran PowerStation 4.0

インテル Fortran には、Microsoft Fortran PowerStation バージョン 4 がサポートする多数の言語拡張子が追加されています。

インテル Fortran バージョン 7.1 とバージョン 8.x の相違点

次の相違点があります。

- コマンドラインを使用するためのコマンド名が `ifort` に変更されました。インテル Fortran の以前のバージョンでは、コマンド名に `ifc` または `efc` が使用されていました。これらのコマンドはインテル Fortran 8.x でもサポートされていますが、今後リリースされるバージョンでは、`ifort` コマンド名のみがサポートされる予定です。
- デフォルト設定ファイル名が `ifl.cfg` または `efl.cfg` から、`ifort.cfg` に変更されました。
- インテル Fortran コンパイラの事前定義済みシンボル名は、`_INTEL_COMPILER` で、値はインテル Fortran バージョン 8.0 では 800、インテル Fortran バージョン 8.1 では 810 です。
- 書式なしファイルのレコード長 (RECL 指定子) が 32 ビットワードに変更されました。レコード長をバイトで取得するには、`-assume byterecl` オプションを使用します。
- バック・スラッシュ文字 (`\`) は、文字リテラルの制御シーケンスで、エスケープ文字として扱われません。バック・スラッシュをエスケープ文字として扱うには、`-assume bscc` オプションを使用します。
- インテル Fortran バージョン 8.x では、デフォルトで `.TRUE.` の値に整数 -1 を使用しますが、バージョン 7 では、`.TRUE.` の値に整数 1 を使用します。バージョン 8.x で `-fpscomp logicals` オプションを使用すると、コンパイラは、`.TRUE.` の値に整数 1 を使用します。
`.FALSE.` の値には、バージョン 8.x でもバージョン 7 と同様に、整数 0 を使用します。

Fortran または他の言語 (C など) で書かれたルーチンでは、コンパイラが指定する `.TRUE.` および `.FALSE.` の値を使用する必要があります。

- バージョン 8.x で使用されている乱数ジェネレータはバージョン 7 で使用されていた乱数ジェネレータと異なります。

バージョン 8.x では、Park と Miller のアルゴリズムに基づく (Compaq* Fortran で使用されていたものと同じ) 乱数ジェネレータを使用しています。バージョン 7 では、Marsaglia 乱数ジェネレータが使用されていました。

これらの乱数ジェネレータは、Fortran 90 規格に準拠しています。

さらに、バージョン 8.x では、`RANDOM_NUMBER` と `RANDOM_SEED` 組込み関数で (バージョン 7 と) 異なるアルゴリズムを使用します。これらの組込み関数は、IA-32 と Itanium® ベース・システムでは異なるアルゴリズムを使用します。

マニュアル情報

一部のマニュアルで内容が移動しました。特に、次の点に注意してください。

- 『インテル® Fortran ユーザーズ・ガイド』では、アプリケーションのビルドおよびアプリケーションの最適化に関する情報が、Vol I、Vol II に分かれて説明されています。
- 以前、『インテル® Fortran プログラマーズ・リファレンス』で説明されていたインテル Fortran の言語情報（組込みプロシージャやディレクティブなど）が、オンラインの『Intel® Fortran Language Reference』（英語）で説明されています。
- インテル Fortran のすべての言語要素とライブラリ・ルーチンは、参考情報を簡単に検索できるオンライン・ヘルプファイルで説明されています。

インテル Fortran バージョン 8.x で提供されていないバージョン 7.1 の機能

次のインテル Fortran バージョン 7.1 の機能は、バージョン 8.x では利用できません。

- IMPLICIT AUTOMATIC | STATIC 文
- インテル Fortran ランタイム・ライブラリ・システムの Itanium プロセッサ・シミュレータへのサポート

ランタイム時のエラー・メッセージ

次の表に、インテル® Fortran ランタイム・ライブラリ (RTL) によって処理されるエラーを示します。各エラーには、エラー番号、重要度コード、エラー・メッセージ、条件シンボル名、およびエラーの詳細説明が示されています。

条件シンボル値 (PARAMETER 文) をプログラム内で定義するには、次のファイルをインクルードします。

```
/opt/intel_fc_80/include/for_iosdef.f
```

表で説明されているメッセージの重要度は、プログラムの動作を決定します。重要度が `info` および `warning` の場合、プログラムの実行は継続されます。重要度が `error` の場合、プログラムの結果が不正になる可能性があります。重要度が `severe` の場合、プログラムの実行はリカバリ処理が指定されていない限り停止します。severe エラーによるプログラムの終了を回避するには、I/O エラー処理指定子をインクルードして再コンパイルするか、または特定のエラーでは、プログラムを再度実行する前に、信号のデフォルト動作を変更する必要があります。

最初の列には、I/O エラーが検出された際に IOSTAT 変数に返されるエラー番号が示されています。

2 列目の最初の行には、(forrtl: に続いて) 画面に表示される重要度レベル、メッセージ番号、およびメッセージ・テキストが示されています。2 行目には、状況条件シンボル (例えば、FOR\$IOS_INCRECTYP) およびメッセージの説明が示されています。

番号	重要度レベル、番号、メッセージ・テキスト、条件シンボル、および説明
なし ¹	<p>info: Fortran error message number is <i>nnn</i></p> <p>インテル Fortran メッセージ・カタログ・ファイルがこのシステムで見つかりませんでした。このエラーは条件シンボルを持っていません。</p>
なし ¹	<p>warning: Could not open message catalog: for msg.cat</p> <p>インテル Fortran メッセージ・カタログ・ファイルがこのシステムで見つかりませんでした。このエラーは条件シンボルを持っていません。</p>
なし ¹	<p>info: Check environment variable NLSPATH and protection of pathname /for_msg.cat</p> <p>インテル Fortran メッセージ・カタログ・ファイルが見つかりませんでした。このエラーは条件シンボルを持っていません。</p>
なし ¹	<p>Insufficient memory to open Fortran RTL catalog: message 41</p> <p>仮想メモリが不足しているため、インテル Fortran メッセージ・カタログを開けませんでした。この問題を解決するには、プログラムを再度実行する前に、limit (C シェル) または ulimit (Bourne、Korn、および bash シェル) コマンドを使用して、各プロセスあたりのデータ制限を増やしてください。</p> <p>詳細は、エラー 41 を参照してください。このエラーは条件シンボルを持っていません。</p>
1 ¹	<p>severe (1): Not a Fortran-specific error</p> <p>FOR\$IOS_NOTFORSPE。ユーザ・プログラムまたは RTL のエラーがインテル Fortran 固有のエラーではなく、他のどのインテル Fortran ランタイム・メッセージを使っても報告できませんでした。ERRSNS を呼び出すと、この種類のエラーは 1 の値を返します (ERRSNS サブルーチンに関する詳細は、『Intel® Fortran Language Reference Manual』(英語) を参照してください)。</p>
8	<p>severe (8): Internal consistency check failure</p> <p>FOR\$IOS_BUG_CHECK。内部エラーです。プログラムに問題がないことを確認してください。プログラムでエラーが見つかった場合再コンパイルしてください。このエラーが解決されない場合、問題を報告してください。</p>
9	<p>severe (9): Permission to access file denied</p>

	FOR\$IOS_PERACCFIL。指定したファイルのモード（保護）を確認してください。正しいファイルが参照されていることを確認してください。プログラムを再実行する前に、保護、指定したファイル、または使用したプロセスを変更してください。
10	<p>severe (10): Cannot overwrite existing file</p> <p>FOR\$IOS_CAOVEEXI。I/O ユニット x に対して、OPEN 文で STATUS='NEW'（新規ファイルの作成）を指定したとき、指定したファイル xxx は既に存在します。正しいファイル名、ディレクトリ・パスおよびユニットなどがソース・プログラムで指定されていることを確認してください。次の操作を 1 つを行ってください。</p> <ul style="list-style-type: none"> プログラムを再実行する前に、既存のファイルを削除するか名前を変更します。 異なるファイル指定、I/O ユニット、または OPEN 文の STATUS を指定するようにソース・ファイルを変更します。
11	<p>info (11) ¹: Unit not connected</p> <p>FOR\$IOS_UNINOTCON。I/O 操作を行う際に、指定したユニットが開かれていませんでした。正しいユニット番号が指定されていることを確認してください。正しいユニット番号が指定されている場合、OPEN 文を使用してファイルを明示的に開いてください（ファイルをユニット番号に接続します）。</p>
17	<p>severe (17): Syntax error in NAMELIST input</p> <p>FOR\$IOS_SYNERRNAM。ネームリスト指定の READ 文に対する入力構文が間違っています。</p>
18	<p>severe (18): Too many values for NAMELIST variable</p> <p>FOR\$IOS_TOOMANVAL。ネームリスト指定の READ 文で変数に割り当てようとした値が多すぎます。</p>
19	<p>severe (19): Invalid reference to variable in NAMELIST input</p> <p>FOR\$IOS_INVREFVAR。次のいずれかの条件が発生しました。</p> <ul style="list-style-type: none"> 変数がネームリスト・グループのメンバではありません。 スカラー変数を添字として使用しようとしてしました。 配列変数の添字が範囲外です。 配列変数に指定された添字が多すぎるか、または少なすぎます。 非文字変数または配列名の部分文字列を指定しようとしてしました。 文字変数の部分文字列指定子が範囲外です。 変数の添字または部分文字列指定子が整数定数ではありません。

	<ul style="list-style-type: none"> 添字のない配列変数を使用して部分文字列が指定されました。
20	<p>severe (20): REWIND error</p> <p>FOR\$IOS_REWERR。次のいずれかの条件が発生しました。</p> <ul style="list-style-type: none"> ファイルがシーケンシャル・ファイルではありません。 ファイルがシーケンシャル・アクセスまたはアペンドアクセスで開かれていません。 インテル Fortran RTL I/O システムは、REWIND 文の実行時にエラー条件を検出しました。
21	<p>severe (21): Duplicate file specifications</p> <p>FOR\$IOS_DUPFILSPE。閉じる操作を行わずにファイル属性を複数回指定しようとした。DEFINE FILE 文が他の DEFINE FILE 文または OPEN 文の後に指定されました。</p>
22	<p>severe (22): Input record too long</p> <p>FOR\$IOS_INPRECTOO。ファイルが開かれた際に、明示的に指定されたレコード長またはデフォルトのレコード長を超えたレコードが読み出されました。ファイルを読み出すには、RECL= 値 (レコード長) で適切なサイズを指定して OPEN 文を使用してください。</p>
23	<p>severe (23): BACKSPACE error</p> <p>FOR\$IOS_BACERR。インテル Fortran RTL I/O システムは、BACKSPACE 文の実行時にエラー条件を検出しました。</p>
24	<p>severe (24): End-of-file during read</p> <p>FOR\$IOS_ENDDURREA。次のいずれかの条件が発生しました。</p> <ul style="list-style-type: none"> インテル Fortran RTL I/O システムは、END、ERR、または IOSTAT 指定子を含まない READ 文の実行時に、ファイル終了 (EOF) 条件を検出しました。 END、ERR、または IOSTAT 指定子を含まない READ 文の実行時に、ENDFILE 文によって書き込まれたファイル終了 (EOF) レコードを検出しました。 END、ERR、または IOSTAT 指定子を含まない READ 文の実行時に、内部ファイルの終了を示す文字列または配列より先を読み取ろうとしました。 <p>このエラーは、END および ERRSNS によって返されます。</p>

25	<p>severe (25): Record number outside range</p> <p>FOR\$IOS_RECNUMOUT。ファイルを開く際に、直接アクセスの READ、WRITE、または FIND 文が範囲外のレコード番号を指定しました。</p>
26	<p>severe (26): OPEN or DEFINE FILE required</p> <p>FOR\$IOS_OPEDEFREQ。ACCESS = ' DIRECT ' が指定された DEFINE FILE 文または OPEN 文をファイルに対して実行する前に、直接アクセスの READ 文、WRITE 文、または FIND 文を実行しようとしてしました。</p>
27	<p>severe (27): Too many records in I/O statement</p> <p>FOR\$IOS_TOOMANREC。次のいずれかの操作が実行されようとしてしました。</p> <ul style="list-style-type: none"> • ENCODE または DECODE 文による複数のレコードの読み取りまたは書き込み。 • 既存のレコード数を超えるレコードの書き込み。
28	<p>severe (28): CLOSE error</p> <p>FOR\$IOS_CLOERR。インテル Fortran RTL I/O システムは、CLOSE 文の実行時にエラー条件を検出しました。</p>
29	<p>severe (29): File not found</p> <p>FOR\$IOS_FILNOTFOU。ファイルを開く際に、指定された名前のファイルが見つかりませんでした。</p>
30	<p>severe (30): Open failure</p> <p>FOR\$IOS_OPEFAI。インテル Fortran RTL I/O システムは、ファイルを開く際に OPEN、INQUIRE、またはその他の I/O 文でエラーを検出しました。このメッセージは、特定のエラー・メッセージが表示される一般的なエラー条件の 1 つではない場合に発行されます。このエラーは、次のいずれかのファイルに対して OPEN 操作を実行しようとした場合に発生することがあります。</p> <ul style="list-style-type: none"> • ディスク上または生の磁気テープ上に存在しないセグメント・ファイル • 閉じた標準 I/O ファイル
31	<p>severe (31): Mixed file access modes</p> <p>FOR\$IOS_MIXFILACC。次のいずれかの組み合わせを実行しようとしてしました。</p> <ul style="list-style-type: none"> • 同じユニット上での書式付き操作と書式なし操作 • ユニット上での無効なアクセスモードの組み合わせ（直接アクセスと

	<p>シーケンシャル・アクセスの組み合わせなど)</p> <ul style="list-style-type: none"> 他の言語で書かれたプログラムが開いた論理ユニット上に対するインテル Fortran RTL I/O 文
32	<p>severe (32): Invalid logical unit number</p> <p>FOR\$IOS_INVLOGUNI。2,147,483,647 より大きいか、または 0 より小さい論理ユニット番号が I/O 文で使用されました。</p>
33	<p>severe (33): ENDFILE error</p> <p>FOR\$IOS_ENDFILERR。次のいずれかの条件が発生しました。</p> <ul style="list-style-type: none"> ファイルが可変長レコードのシーケンシャル編成ファイルではありません。 ファイルがシーケンシャル・アクセスまたはアペンドアクセスで開かれていません。 書式なしファイルにセグメント・レコードが含まれていません。 インテル Fortran RTL I/O システムは、ENDFILE 文の実行時にエラーを検出しました。
34	<p>severe (34): Unit already open</p> <p>FOR\$IOS_UNIALROPE。DEFINE FILE 文で既に関われている論理ユニットを指定しました。</p>
35	<p>severe (35): Segmented record format error</p> <p>FOR\$IOS_SEGRECFOR。書式なしシーケンシャル・ファイルで無効なセグメント・レコード制御データワードが検出されました。このファイルは、RECORDTYPE= ' FIXED ' または ' VARIABLE ' が有効な状態で作成されたか、または Fortran 以外の言語で記述されたプログラムによって作成された可能性があります。</p>
36	<p>severe (36): Attempt to access non-existent record</p> <p>FOR\$IOS_ATTACCNON。直接アクセスの READ または FIND 文が、相対ファイル (または固定長レコードを含むディスク上のシーケンシャル・ファイル) 終了を超えてアクセスしようとしたか、または相対ファイルから以前に削除されたレコードにアクセスしようとした。</p>
37	<p>severe (37): Inconsistent record length</p> <p>FOR\$IOS_INCRECLEN。レコード長を指定せずに直接アクセスファイルを開こうとしました。</p>
38	<p>severe (38): Error during write</p>

	FOR\$IOS_ERRDURWRI。インテル Fortran RTL I/O システムは、WRITE 文の実行時にエラー条件を検出しました。
39	severe (39): Error during read FOR\$IOS_ERRDURREA。Intel Fortran RTL I/O システムは、READ 文の実行時にエラー条件を検出しました。
40	severe (40): Recursive I/O operation FOR\$IOS_RECIO_OPE。論理ユニットに対して I/O 文を処理している最中に、同じ論理ユニットに別の I/O 操作を実行しようとしてしました。例えば、I/O を実行する関数サブプログラムが、I/O リスト内の式または可変書式の式で参照される同一の論理ユニットに対して I/O 操作を実行しようとした場合に発生します。
41	severe (41): Insufficient virtual memory FOR\$IOS_INSVIRMEM。インテル Fortran RTL は、利用可能な仮想メモリを上回る領域を動的に割り当てようとしてしました。この問題を解決するには、プログラムを再度実行する前に、limit (C シェル) または ulimit (Bourne、Korn、および bash シェル) コマンドを使用して、各プロセスあたりのデータ制限を増やしてください。 各プロセスあたりの最大データサイズが既に割り当てられているかどうかを確認するには、sysconfigtab またはシステム設定ファイルの <i>maxdsiz</i> パラメータの値を確認してください。必要な場合は、この値を増加してください。変更内容は、システムが再起動されるまで有効になりません (sysconfigtab を変更してもカーネルを再構築する必要はありません)。 新しいシステムリソースが有効になった後、このプログラムを再実行してください。
42	severe (42): No such device FOR\$IOS_NO_SUCDEV。OPEN 操作でパス名に無効なまたは不明なデバイス名が含まれています。
43	severe (43): File name specification error FOR\$IOS_FILNAMSPE。OPEN または INQUIRE 文に指定されたパス名またはファイル名をインテル Fortran RTL I/O システムで処理できません。
44	severe (44): Inconsistent record type FOR\$IOS_INCRECTYP。OPEN 文の RECORDTYPE 値が、既存のファイルのレコード形式属性と一致しません。
45	severe (45): Keyword value error in OPEN statement

	FOR\$IOS_KEYVALERR。値を必要とする OPEN または CLOSE 文の指定子に不適切な値が指定されました。
46	<p>severe (46): Inconsistent OPEN/CLOSE parameters</p> <p>FOR\$IOS_INCOPECLO。OPEN または CLOSE 文の指定に矛盾があります。無効な組み合わせの例を次に示します。</p> <ul style="list-style-type: none"> • READONLY または ACTION= ' READ ' と STATUS= ' NEW ' または STATUS= ' SCRATCH ' • READONLY または STATUS= ' REPLACE '、ACTION= ' WRITE '、 または ACTION= ' READWRITE ' • ACCESS= ' APPEND ' と READONLY、ACTION= ' READ '、STATUS= ' NEW '、または STATUS= ' SCRATCH ' • DISPOSE= ' SAVE '、' PRINT '、または ' SUBMIT ' と STATUS= ' SCRATCH ' • DISPOSE= ' DELETE ' と READONLY • CLOSE 文の STATUS= ' DELETE ' と OPEN 文の READONLY • ACCESS= ' APPEND ' と STATUS= ' REPLACE ' • ACCESS= ' DIRECT ' または ' KEYED ' と POSITION= ' APPEND '、' ASIS '、または ' REWIND '
47	<p>severe (47): Write to READONLY file</p> <p>FOR\$IOS_WRIREADFIL。現在有効な OPEN 文で、ACTION= ' READ ' または READONLY が宣言されたファイルに対して書き込み操作を行なおうとしました。</p>
48	<p>severe (48): Invalid argument to Fortran Run-Time Library</p> <p>FOR\$IOS_INVARGFOR。コンパイラは、インテル Fortran RTL に無効なまたは不適切なコードの引数を渡しました。このエラーは、使用中の RTL よりもコンパイラが新しい場合に発生する可能性があります。</p>
51	<p>severe (51): Inconsistent file organization</p> <p>FOR\$IOS_INCFILORG。OPEN 文で指定したファイル編成が、既存のファイル編成と一致しません。</p>
53	<p>severe (53): No current record</p> <p>FOR\$IOS_NO_CURREC。現在のレコードが未定義の状態で、REWRITE 文を実行してレコードを書き換えようとしてしました。現在のレコードを定義するには、READ 文を実行します。READ 文の後で REWRITE 文の前に、論理ユニットに対して INQUIRE 文を実行することができます。READ と REWRITE 文の間</p>

	では、論理ユニットに対して他の操作は実行できません。
55	<p>severe (55): DELETE error</p> <p>FOR\$IOS_DELEERR。インテル Fortran RTL I/O システムは、DELETE 文の実行時にエラー条件を検出しました。</p>
57	<p>severe (57): FIND error</p> <p>FOR\$IOS_FINERR。インテル Fortran RTL I/O システムは、FIND 文の実行時にエラー条件を検出しました。</p>
58 ¹	<p>info (58): Format syntax error at or near xx</p> <p>FOR\$IOS_FMTSYN。書式文字列の部分文字列である xx を含む文で書式構文エラーを確認してください。FORMAT 文に関する詳細は、『Intel® Fortran Language Reference Manual』(英語) を参照してください。</p>
59	<p>severe (59): List-directed I/O syntax error</p> <p>FOR\$IOS_LISIO_SYN²。リスト指定入力レコードのデータ形式が無効か、または定数の型が対応する変数と互換性がありません。変数の値は変更されませんでした。</p>
60	<p>severe (60): Infinite format loop</p> <p>FOR\$IOS_INFFORLOO。I/O リストを含む I/O 文に対応する書式に、これらの値を転送するときに使用するフィールド記述子が指定されていません。</p>
61	<p>severe or info³ (61): Format/variable-type mismatch</p> <p>FOR\$IOS_FORVARMIS²。整数フィールド記述子 (I、L、O、Z、B) を使用して実数変数の読み取りまたは書き込み、あるいは実数フィールド記述子 (D、E、または F) を使用して整数変数または論理変数の読み取りまたは書き込みを実行しようとしていました。</p>
62	<p>severe (62): Syntax error in format</p> <p>FOR\$IOS_SYNNERRFOR。RTL で配列または文字変数に格納された書式を処理している最中に構文エラーが発生しました。</p>
63	<p>error or info³ (63): Output conversion error</p> <p>FOR\$IOS_OUTCONERR²。書式付き出力操作で、有効桁数を失わずに特定の数字の値を、指定されたフィールド長で出力できませんでした。この問題が検出された場合、オーバーフローしたフィールドは、出力レコードにエラーがあることを示すアスタリスクで埋められます。このエラーに対して ERR アドレスが定義されていない場合、エラー・メッセージが表示された後にプログラムの実行は継続します。</p>
64	<p>severe (64): Input conversion error</p>

	FOR\$IOS_INPCONERR ² 。書式付き入力操作で、入力フィールドに無効な文字が検出されたか、または入力値が、入力変数で表現可能な範囲をオーバーフローしました。変数の値は 0 に設定されました。
65	error (65): Floating invalid FOR\$IOS_FLTINV。算術演算で、計算に使用された浮動小数点の値が要求した演算の型に対して不正、または不正な例外値でした。例えば、浮動小数点の値 0.0 または負数の対数を要求した場合に発生します。特定の算術式に対して、-check nopower オプションを指定すると、このメッセージを抑止できます
66	severe (66): Output statement overflows record FOR\$IOS_OUTSTAOVE。出力文で最大レコードサイズを超えるデータを転送しようとしてしました。
67	severe (67): Input statement requires too much data FOR\$IOS_INPSTAREQ。' NO ' の PAD 指定子で開かれたファイルから、書式なし READ 文または書式付きシーケンシャル READ 文を使用して、レコードに存在するデータより多くのデータを読み取ろうとしてしました。
68	severe (68): Variable format expression value error FOR\$IOS_VFEVALERR ² 。可変書式の式での値が、目的の使用に許容される範囲内にありません。例えば、フィールド幅が 0 より小さいか、または 0 の場合に発生します。編集記述子 P (0 の値が仮定される) の場合を除いて、1 の値が仮定されました。
69	¹ error (69): Process interrupted (SIGINT) FOR\$IOS_SIGINT。プロセスが信号 SIGINT を受信しました。この割り込み信号の発信元を確認してください (signal(3) を参照)。
70	¹ severe (70): Integer overflow FOR\$IOS_INTOVF。算術演算中に整数の値がバイト、ワード、またはロングワードの範囲を超えました。この演算結果の下位部は、正しい値です。より大きい整数データサイズを指定することを検討してください (ソース・プログラムを変更するか、または INTEGER 宣言に対して、場合によっては f90 オプション -integer_size <i>nn</i> を使用してください)。
71	¹ severe (71): Integer divide by zero FOR\$IOS_INTDIV。整数算術演算で、ゼロ除算を実行しようとしてしました。この演算の結果として被除数が設定されました。1 で除算を行った場合と同等です。
72	¹ error (72): Floating overflow

	FOR\$IOS_FLTOVF。算術演算で、浮動小数点の値がそのデータ型で表示可能な最大値を超えました。
73	<p><code>error (73): Floating divide by zero</code></p> <p>FOR\$IOS_FLTDIV。浮動小数点算術演算で、ゼロ除算を実行しようとした。</p>
74	<p><code>error (74): Floating underflow</code></p> <p>FOR\$IOS_FLTUND。算術演算で、浮動小数点の値がそのデータ型で表現できる最小有限値よりも小さくなりました。/fpe n オプションの値によって、アンダーフローの結果が 0 に設定される場合と、段階的にアンダーフローさせる場合があります。</p>
75	<p><code>error (75): Floating point exception</code></p> <p>FOR\$IOS_SIGFPE。浮動小数点例外が発生しました。コア・ダンプ・ファイルが作成されました。次のいずれかの原因が考えられます。</p> <ul style="list-style-type: none"> • ゼロ除算 • オーバーフロー • 不正な演算。無限値の減算、ゼロと無限大（符号なし）の乗算、ゼロとゼロの除算、または無限大と無限大の除算などの無効な演算 • オーバーフローによって変換できない場合の、浮動小数点から固定小数点形式への変換
76	<p><code>error (76): IOT trap signal</code></p> <p>FOR\$IOS_SIGIOT。コア・ダンプ・ファイルが作成されました。この IOT 信号の原因について、コアダンプを調べてください。</p>
77	<p><code>severe (77): Subscript out of range</code></p> <p>FOR\$IOS_SUBRNG。宣言した配列の上下限の範囲外で配列参照が検出されました。</p>
78	<p><code>error (78): Process killed (SIGTERM)</code></p> <p>FOR\$IOS_SIGTERM。プロセスが信号 SIGTERM を受信しました。このソフトウェア終了信号の発信元を確認してください (signal (3) を参照)。</p>
79	<p><code>error (79): Process quit (SIGQUIT)</code></p> <p>FOR\$IOS_SIGQUIT。プロセスが信号 SIGQUIT を受信しました。コア・ダンプ・ファイルが作成されました。この終了信号の発信元を確認してください (signal (3) を参照)。</p>
95	<p><code>info (95): Floating-point conversion failed</code></p>

	<p>FOR\$IOS_FLOCONFAL。非ネイティブ浮動小数点データの書式なし読み取りまたは書き込みを実行しようとしたが、浮動小数点の値に以下の問題があったために失敗しました。</p> <ul style="list-style-type: none"> • 対応するネイティブ形式で許容される最大値を超え、値に無限大（正または負）が設定されました。 • 値が無限大（正または負）で、無限大（正または負）に設定されました。 • 無効な値で、非数字の値 (NaN) が設定されました。 <p>非常に小さい数字がゼロに設定されました。このエラーは、指定された非ネイティブ浮動小数点形式が、指定されたファイルに存在する浮動小数点形式に一致しないために発生する場合があります。</p> <p>以下の項目を確認してください。</p> <ul style="list-style-type: none"> • 正しいファイルが指定されている。 • レコード構成が、インテル Fortran が予期する形式に一致している。 • 使用されるデータの範囲。 • 正しい非ネイティブ浮動小数点データ形式が指定されている。
96	<p>info (96): F_UFMTENDIAN environment variable was ignored:erroneous syntax</p> <p>FOR\$IOS_UFMTENDIAN。指定された Fortran ユニットに対して、リトル・エンディアンまたはビッグ・エンディアン形式への変換を指定する構文が間違っています。プログラムは実行されますが、F_UFMTENDIAN の値を変更しなかった場合、誤った結果が生じる可能性があります。正しい構文に関する詳細は、「環境変数 F_UFMTENDIAN を使用する方法」を参照してください。</p>
108	<p>severe (108): Cannot stat file</p> <p>FOR\$IOS_CANSTAFIL。指定されたファイルで stat 操作を実行しようとしたが、失敗しました。正しいファイルとユニットが指定されていることを確認してください。</p>
120	<p>severe (120): Operation requires seek ability</p> <p>FOR\$IOS_OPEREQSEE。シーク操作を実行する機能を必要とするファイルで操作を実行しようとした。正しいユニット、ディレクトリ・パス、およびファイルが指定されていることを確認してください。</p>
138	<p>severe (138): Array index out of bounds (SIGILL)</p> <p>FOR\$IOS_BRK_RANGE。ブレーク例外が SIGTRAP 信号を生成しました</p>

	<p>(signal (3) を参照)。コア・ダンプ・ファイルが作成されました。</p> <p>原因は、配列の次元境界を超える配列添字です。</p> <p>-check bounds オプションを使用して (decfort_dump_flag 環境変数で設定することができます) 再コンパイルするか、またはコア・ダンプ・ファイルを調べて、エラーの原因となったソースコードを確認してください。</p>
139	<p>¹severe (139): Array index out of bounds for index nn (SIGILL)</p> <p>FOR\$IOS_BRK_RANGE2。ブレーク例外が SIGTRAP 信号を生成しました (signal (3) を参照)。コア・ダンプ・ファイルが作成されました。</p> <p>原因は、配列インデックス n の次元境界を超える配列添字です。</p> <p>-check bounds オプションを使用して (decfort_dump_flag 環境変数で設定することができます) 再コンパイルするか、またはコア・ダンプ・ファイルを調べて、エラーの原因となったソースコードを確認してください。</p>
140	<p>¹severe (140): Floating inexact</p> <p>FOR\$IOS_FLTINE。浮動小数点算術演算または変換演算の結果が、数学的に正確な結果と異なります。IEEE 演算の丸められた結果が不正確な場合に、このエラーが通知されます。</p>
144	<p>¹severe (144): reserved operand</p> <p>FOR\$IOS_ROPRAND。インテル Fortran RTL が予約オペランドを検出しました。この問題をインテルに報告してください。</p>
145	<p>¹severe (145): Assertion error</p> <p>FOR\$IOS_ASSERTERR。インテル Fortran RTL がアサーション・エラーを検出しました。この問題をインテルに報告してください。</p>
146	<p>¹severe (146): Null pointer error</p> <p>FOR\$IOS_NULPTRERR。アドレスが設定されていないポインタを使用しようとしました。ソース・プログラムを修正して、再コンパイルおよび再リンクを行ってください。</p>
147	<p>¹severe (147): stack overflow</p> <p>FOR\$IOS_STKOVF。インテル Fortran RTL がプログラムの実行中にスタック・オーバーフローを検出しました。</p>
148	<p>¹severe (148): String length error</p> <p>FOR\$IOS_STRLENERR。文字列操作で、利用可能な文字列長を超える整数</p>

	<p>値がコンテキストに含まれています。</p> <p>-check bounds オプションを使用して (decfort_dump_flag 環境変数で設定することができます) 再コンパイルするか、または core ファイルを調べて、エラーの原因となったソースコードを確認してください。</p>
149	<p>¹severe (149): Substring error</p> <p>FOR\$IOS_SUBSTRERR。配列添字が配列に設定された次元境界を超えています。</p> <p>-check bounds オプションを使用して (decfort_dump_flag 環境変数で設定することができます) 再コンパイルするか、または core ファイルを調べて、エラーの原因となったソースコードを確認してください。</p>
150	<p>¹severe (150): Range error</p> <p>FOR\$IOS_RANGEERR。利用可能な範囲を超える整数値がコンテキストに含まれています。</p>
151	<p>¹severe (151): Allocatable array is already allocated</p> <p>FOR\$IOS_INVREALLOC。既に割り当てられている割付け配列を割り当てることはできません。再び割り当てを可能にするには、配列の割り当てを解除する必要があります。</p>
152	<p>¹severe (152): Unresolved contention for Intel Fortran RTL global resource</p> <p>FOR\$IOS_RESACQFAI。再入可能ルーチンのためのインテル Fortran RTL グローバル・リソースの獲得に失敗しました。</p> <p>マルチスレッド・プログラムの場合、要求されたグローバル・リソースはプログラムの別のスレッドによって使用されています。</p> <p>非同期ハンドラを使用するプログラムの場合、要求されたグローバル・リソースがプログラムの呼出し側（メイン・プログラムなど）によって使用されているにもかかわらず、非同期ハンドラが同じグローバル・リソースを取得しようとして失敗しました。</p>
153	<p>¹severe (153): Allocatable array or pointer is not allocated</p> <p>FOR\$IOS_INVDEALLOC。Fortran-90 割付け配列またはポインタは、割り当てを解除する前に、割り当てられている必要があります。再び割り当てを解除する前に、配列またはポインタを割り当てる必要があります。</p>
173	<p>¹severe (173): A pointer passed to DEALLOCATE points</p>

	<p>to an array that cannot be deallocated</p> <p>FOR\$IOS_INVDEALLOC2。DEALLOCATE に渡されたポインタが、明示的配列、配列スライス、または DEALLOCATE 文で割り当てを解除できなかったメモリの他の型を指しています。以前に ALLOCATE 文で割り当てられた配列全体に対してのみ、DEALLOCATE 文に渡すことができます。</p>
174	<p>¹severe (174): SIGSEGV, message-text</p> <p>FOR\$IOS_SIGSEGV。このエラー番号には、次の 2 種類のメッセージのうち 1 つが発行されます。</p> <ul style="list-style-type: none"> severe (174): SIGSEGV, segmentation fault occurred <p>このメッセージは、プログラムが不正なメモリ参照を実行しようとしたことを示します。プログラムにエラーがないかを確認してください。</p> <ul style="list-style-type: none"> severe (174): SIGSEGV, possible program stack overflow occurred <p>また、次の説明テキストも表示されます。 Program requirements exceed current stacksize resource limit.</p>
175	<p>¹severe (175): DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8</p> <p>FOR\$IOS_SHORTDATEARG。DATE_AND_TIME 組込み関数への DATE 引数に関連付けられた文字数が、要求された長さよりも短すぎました。この引数として渡す文字数を少なくとも 8 文字に増やす必要があります。また、TIME および ZONE 引数の文字数が、最小値を満たしていることを確認してください。</p>
176	<p>¹severe (176): TIME argument to DATE_AND_TIME is too short (LEN=n), required LEN=10</p> <p>FOR\$IOS_SHORTTIMEARG。TIME_AND_TIME 組込み関数への TIME 引数に関連付けられた文字数が、要求された長さよりも短すぎました。この引数として渡す文字数を少なくとも 10 文字に増やす必要があります。また、DATE および ZONE 引数の文字数が、最小値を満たしていることを確認してください。</p>
177	<p>¹severe(177): ZONE argument to DATE_AND_TIME is too short (LEN=n), required LEN=5</p> <p>FOR\$IOS_SHORTZONEARG。ZONE_AND_TIME 組込み関数への ZONE 引数に関連付けられた文字数が、要求された長さよりも短すぎました。この引数として渡す文字数を少なくとも 5 文字に増やす必要があります。また、</p>

	DATE および TIME 引数の文字数が、最小値を満たしていることを確認してください。
178 ¹	<p>severe(178): Divide by zero</p> <p>FOR\$IOS_DIV。浮動小数点または整数のゼロ除算例外が発生しました。</p>
179 ^{1,4}	<p>severe(179): Cannot allocate array---overflow on array size calculation</p> <p>FOR\$IOS_ARRSIZEOVF。配列領域の動的割り当てに失敗しました。要求した領域サイズがアドレス可能なメモリサイズを超えています。</p>
256	<p>severe (256): Unformatted I/O to unit open for formatted transfers</p> <p>FOR\$IOS_UNFIO_FMT。OPEN 文 (FORM 指定子) で書式付きファイルであることを指定したユニットに対して、書式なし I/O を実行しようとしてしました。正しいユニット (ファイル) が指定されていることを確認してください。</p> <p>FORM 指定子が OPEN で指定されておらず、ファイルに書式なしデータが含まれている場合は、OPEN 文で FORM= ' UNFORMATTED ' を指定してください。それ以外の場合は、必要であれば書式付き I/O (リスト指定 I/O またはネームリスト I/O など) を使用します。</p>
257	<p>severe (257): Formatted I/O to unit open for unformatted transfers</p> <p>FOR\$IOS_FMTIO_UNF。OPEN 文 (FORM 指定子) で書式なしファイルであることを指定したユニットに対して、書式付き I/O (リスト指定 I/O またはネームリスト I/O など) を実行しようとしてしました。正しいユニット (ファイル) が指定されていることを確認してください。</p> <p>FORM 指定子が OPEN で指定されておらず、ファイルに書式付きデータが含まれている場合は、OPEN 文で FORM= ' FORMATTED ' を指定してください。それ以外の場合は、必要であれば書式なし I/O を使用します。</p>
264	<p>severe (264): operation requires file to be on disk or tape</p> <p>FOR\$IOS_OPERREQDIS。端末またはパイプなどのデバイスで BACKSPACE 文を実行しようとしてしました。</p>
265	<p>severe (265): operation requires sequential file organization and access</p> <p>FOR\$IOS_OPEREQSEQ。ファイル編成がシーケンシャル編成ではないファイル、またはアクセスがシーケンシャル・アクセスではないファイルで BACKSPACE 文を実行しようとしてしました。BACKSPACE 文はシーケンシャル・アクセスで開かれたシーケンシャル・ファイルでのみ使用することができ</p>

	ます。
266 ¹	error (266): Fortran abort routine called FOR\$IOS_PROABOUSE。プログラムを終了するために abort を呼び出しました。
268 ¹	severe (268): End of record during read FOR\$IOS_ENDRECDUR。EOR 分岐指定子が指定されていないノン・アドバンス I/O READ 文の実行中に、レコード終了 (EOR) 条件が検出されました。
297 ¹	info (297): nn floating invalid traps FOR\$IOS_FLOINVEXC。プログラムの実行中に検出された浮動小数点の不正なデータトラップの総数は、nn でした。この要約メッセージはプログラムの終了時に表示されます。
298 ¹	info (298): nnfloating overflow traps FOR\$IOS_FLOOVFEXC。プログラムの実行中に検出された浮動小数点のオーバーフロー・トラップの総数は、nn でした。この要約メッセージはプログラムの終了時に表示されます。
299 ¹	info (299): nnfloating divide-by-zero traps FOR\$IOS_FLODIV0EXC。プログラムの実行中に検出された浮動小数点のゼロ除算トラップの総数は、nn でした。この要約メッセージはプログラムの終了時に表示されます。
300 ¹	info (300): nnfloating underflow traps FOR\$IOS_FLOUNDEXC。プログラムの実行中に検出された浮動小数点のアンダーフロー・トラップの総数は、nn でした。この要約メッセージはプログラムの終了時に表示されます。

脚注:

1 IOSTAT で返されないエラーを示します。

2 エラー番号 59、61、63、64、および 68 の場合、ERR の転送は I/O 文の終了後に行われます。結果ファイルのステータスおよびレコード位置は、エラーが発生しなかった場合と同じになります。ただし、他の I/O エラーの場合、エラーの検出直後に ERR 転送が行なわれるため、ファイルのステータスおよびレコード位置は定義されません。

エラー 61 および 63 の重要度は、コンパイル中に使用される -check オプションによって異なります。

4 ALLOCATE 文の STAT で返されるエラーを示します。

キーワード

!	_linux_ プリプロセッサ・シンボル.....27
!DEC\$ プリフィックス..... 40	_unix_ プリプロセッサ・シンボル.....27
!MS\$ プリフィックス 40	_unix_ プリプロセッサ・シンボル.....27
*	_FTN_ALLOC() ライブラリ・ルーチン..36
*DEC\$ プリフィックス 40	_OPENMP プリプロセッサ・シンボル.27
*DIR\$ プリフィックス 40	1
/	-1 コンパイラ・オプション42
/bin ファイル.....256, 257	10 進変換 259
/opt/intel_fc_80/include/fordef.for ファイル 132	-132 コンパイラ・オプション66
—	16 進変換 259
ELF プリプロセッサ・シンボル 27	2
_gnu_linux_ プリプロセッサ・シンボル 27	2 進変換..... 259
i386 プリプロセッサ・シンボル 27	6
i386 プリプロセッサ・シンボル 27	-66 コンパイラ・オプション.....66
ia64 プリプロセッサ・シンボル 27	7
ia64 プリプロセッサ・シンボル 27	-72 コンパイラ・オプション.....66
_INTEL_COMPILER プリプロセッサ・シンボル 27	8
_INTEL_COMPILER_BUILD_DATE プリプロセッサ・シンボル 27	8 進変換..... 259
linux プリプロセッサ・シンボル 27	-80 コンパイラ・オプション.....66
	A
	ACCEPT 文..... 151, 153, 161, 172

ACTION 指定子

OPEN 文 174

ADVANCE 指定子

READ 文 176

WRITE 文 176

ALIAS プロパティ 199, 205

-align all コンパイラ・オプション 52

-align none コンパイラ・オプション 52

-align コンパイラ・オプション 52

-altparam コンパイラ・オプション 66

-ansi_alias コンパイラ・オプション 71

-arch コンパイラ・オプション 79

-assume [no]bscc コンパイラ・オプション 42

-assume buffered_io コンパイラ・オプション 79

-assume byterecl コンパイラ・オプション 52

-assume cc_omp コンパイラ・オプション 71

-assume dummy_aliases コンパイラ・オプション 52

-assume minus0 コンパイラ・オプション 60

-assume none コンパイラ・オプション 71

-assume protect_constants コンパイラ・オプション 52

-assume source_include コンパイラ・オプション 95

-assume underscore コンパイラ・オプション 59

ATTRIBUTES プロパティ

言語が混在したプログラミング 199

文字列の処理の影響 226

ATTRIBUTES プロパティ 205, 219

-auto コンパイラ・オプション 52

-auto_ilp32 コンパイラ・オプション 79

-auto_scalar コンパイラ・オプション 52

-automatic コンパイラ・オプション 52

-ax コンパイラ・オプション 79

B

BACKSPACE 文 151, 172

bash_profile 17

bash_profile ファイル 17

BIG_ENDIAN キーワード 136

C

-C run-time コンパイラ・オプション 98

C ソースファイル

コンパイル 19

C プロシージャの呼び出し	-common_args コンパイラ・オプション	52
Fortran プログラム.....		233
C プロパティ	COMPLEX データの表現	130
		199, 205
C 構造	COMPLEX(16) データの表現	131
言語が混在したプログラミングにお ける使用.....	COMPLEX(4) データの表現.....	130
		211
-c 出力ファイル・コンパイラ・オプション	COMPLEX(8) データの表現.....	131
		93
C 変数	COMPLEX(KIND=16) データの表現	131
言語が混在したプログラミングにお ける使用.....	COMPLEX(KIND=4) データの表現 .	130
		211
C/C++ 命名規則	COMPLEX(KIND=8) データの表現 .	131
		203
C/Fortran が混在したプログラム	COMPLEX*16 データの表現.....	131
C プロシージャの呼び出し	COMPLEX*32 データの表現.....	131
		233
引数の渡し方	COMPLEX*8 データの表現	130
		233
概要	-complex_limited_range コンパイラ・オ プション	79
		230
命名規則.....	-convert コンパイラ・オプション	42, 148
		233
-CB コンパイラ・オプション	convert コンパイラ・オプションを使用 する方法.....	148
		98
-ccdefault コンパイラ・オプション	-cpp コンパイラ・オプション	95
		71
cDEC\$ プリフィックス	CRAY キーワード	136
		40
cDIR\$ プリフィックス	Cray 形式ポインタ.....	112, 220
		40
-check コンパイラ・オプション	crtxi.o ファイル.....	246
		98
CLOSE 文.....	crtxn.o ファイル	246
		151, 171
codecov ファイル	-cxxlib-icc コンパイラ・オプション.....	68
		256, 257

D

-D コンパイラ・オプション	95
-d_lines コンパイラ・オプション	66
-DD コンパイラ・オプション	66
-debug コンパイラ・オプション	71
decfort_dump_flag 環境変数	253
DECLARE コンパイラ・ディレクティブ	40
DECORATE プロパティ	199
DEFAULTFILE 指定子	
OPEN 文	161, 163
DEFINE FILE 文	151
DEFINE コンパイラ・ディレクティブ	40
DELETE 文	151, 172
DOUBLE COMPLEX データの表現	131
DOUBLE PRECISION データの表現	129
-double_size コンパイラ・オプション ..	52
-dps コンパイラ・オプション	66
-dryrun コンパイラ・オプション	71
-dynamic-linker コンパイラ・オプション	71
-dyncom コンパイラ・オプション ..	36, 52

E

-e90 コンパイラ・オプション	47
-e95 コンパイラ・オプション	47
ebp レジスタ	79
efc ファイル	257
efc.cfg ファイル	257
efcbn ファイル	257
END 分岐指定子	238
ENDFILE 文	151, 172
EOR 分岐指定子	238
ERR 分岐指定子	238
-error_limit コンパイラ・オプション	47
ERRSNS サブルーチン	238
EXCEPTION_CONTINUE_SEARCH.	253
export コマンド	16
-extend_source コンパイラ・オプション	66
EXTENDED PRECISION データの表現	130

F

-F コンパイラ・オプション	95
F_UFMTENDIAN を使用する方法 ...	143
F_UFMTENDIAN 環境変数	143, 253

-f66 コンパイラ・オプション	66	FOR\$IOS_ATTACCNON エラー・メッセージ	262
-f77rtl コンパイラ・オプション	42	FOR\$IOS_BACERR エラー・メッセージ	262
f90_dyncom ランタイム・ライブラリ・ルーチン	36	FOR\$IOS_BRK_RANGE エラー・メッセージ	262
-falias コンパイラ・オプション	79	FOR\$IOS_BRK_RANGE2 エラー・メッセージ	262
-fast コンパイラ・オプション	79	FOR\$IOS_BUG_CHECK エラー・メッセージ	262
-fcode-asm コンパイラ・オプション ...	93	FOR\$IOS_CANSTAFIL エラー・メッセージ	262
FDX キーワード	136	FOR\$IOS_CAOVEEXI エラー・メッセージ	262
-ffnalias コンパイラ・オプション	79	FOR\$IOS_CLOERR エラー・メッセージ	262
FGX キーワード	136	FOR\$IOS_DELEERR エラー・メッセージ	262
FILE 指定子		FOR\$IOS_DIV エラー・メッセージ	262
OPEN 文	161, 163	FOR\$IOS_DUPFILSPE エラー・メッセージ	262
FIND 文	151, 172	FOR\$IOS_ENDDURREA エラー・メッセージ	262
FIORT_CONVERTn 環境変数	141	FOR\$IOS_ENDFILERR エラー・メッセージ	262
-fixed コンパイラ・オプション	66	FOR\$IOS_ENDRECDUR エラー・メッセージ	262
FIXEDFORMLINESIZE コンパイラ・ディレクティブ	40	FOR\$IOS_ERRDURREA エラー・メッセージ	262
-fltconsistency コンパイラ・オプション	60		
-fminshared コンパイラ・オプション ...	71		
-fnsplit コンパイラ・オプション	79		
FOR\$IOS_ARRSIZEOVF エラー・メッセージ	262		
FOR\$IOS_ASSERTERR エラー・メッセージ	262		

FOR\$IOS_ERRDURWRI エラー・メッセージ262	FOR\$IOS_FMTIO_UNF エラー・メッセージ 262
FOR\$IOS_FILNAMSP E エラー・メッセージ262	FOR\$IOS_FMTSYN エラー・メッセージ 262
FOR\$IOS_FILNOTFOU エラー・メッセージ262	FOR\$IOS_FORVARMIS エラー・メッセージ 262
FOR\$IOS_FINERR エラー・メッセージ262	FOR\$IOS_INCFILOGR エラー・メッセージ 262
FOR\$IOS_FLOCONFAI エラー・メッセージ262	FOR\$IOS_INCOPECLO エラー・メッセージ 262
FOR\$IOS_FLODIV0EXC エラー・メッセージ262	FOR\$IOS_INCRECLEN エラー・メッセージ 262
FOR\$IOS_FLOINVEXC エラー・メッセージ262	FOR\$IOS_INCRECTYP エラー・メッセージ 262
FOR\$IOS_FLOOVFEXC エラー・メッセージ262	FOR\$IOS_INFFORLOO エラー・メッセージ 262
FOR\$IOS_FLOUNDEXC エラー・メッセージ262	FOR\$IOS_INPCONERR エラー・メッセージ 262
FOR\$IOS_FLTDIV エラー・メッセージ262	FOR\$IOS_INPRECTOO エラー・メッセージ 262
FOR\$IOS_FLTINE エラー・メッセージ262	FOR\$IOS_INPSTAREQ エラー・メッセージ 262
FOR\$IOS_FLTINV エラー・メッセージ262	FOR\$IOS_INSVIRMEM エラー・メッセージ 262
FOR\$IOS_FLTOVF エラー・メッセージ262	FOR\$IOS_INTDIV エラー・メッセージ 262
FOR\$IOS_FLTUND エラー・メッセージ262	FOR\$IOS_INTOVF エラー・メッセージ 262

FOR\$IOS_INVARGFOR エラー・メッセージ262	FOR\$IOS_OPEFAI エラー・メッセージ 262
FOR\$IOS_INVDEALLOC エラー・メッセージ262	FOR\$IOS_OPEREQSEE エラー・メッセージ 262
FOR\$IOS_INVDEALLOC2 エラー・メッセージ.....262	FOR\$IOS_OPEREQSEQ エラー・メッセージ 262
FOR\$IOS_INVLOGUNI エラー・メッセージ262	FOR\$IOS_OPERREQDIS エラー・メッセージ 262
FOR\$IOS_INVREALLOC エラー・メッセージ262	FOR\$IOS_OUTCONERR エラー・メッセージ 262
FOR\$IOS_INVREFVAR エラー・メッセージ262	FOR\$IOS_OUTSTAOVE エラー・メッセージ 262
FOR\$IOS_KEYVALERR エラー・メッセージ262	FOR\$IOS_PERACCFIL エラー・メッセージ 262
FOR\$IOS_LISIO_SYN エラー・メッセージ262	FOR\$IOS_PROABOUSE エラー・メッセージ 262
FOR\$IOS_MIXFILACC エラー・メッセージ262	FOR\$IOS_RANGEERR エラー・メッセージ 262
FOR\$IOS_NO_CURREC エラー・メッセージ262	FOR\$IOS_RECIO_OPE エラー・メッセージ 262
FOR\$IOS_NO_SUCDEV エラー・メッセージ262	FOR\$IOS_RECNUMOUT エラー・メッセージ 262
FOR\$IOS_NOTFORSPE エラー・メッセージ262	FOR\$IOS_RESACQFAI エラー・メッセージ 262
FOR\$IOS_NULPTRERR エラー・メッセージ262	FOR\$IOS_REWERR エラー・メッセージ 262
FOR\$IOS_OPEDEFREQ エラー・メッセージ262	FOR\$IOS_ROPRAND エラー・メッセージ 262

FOR\$IOS_SEGRECFOR エラー・メッセージ262	FOR\$IOS_SYNERRFOR エラー・メッセージ 262
FOR\$IOS_SHORTDATEARG エラー・メッセージ262	FOR\$IOS_SYNERRNAM エラー・メッセージ 262
FOR\$IOS_SHORTTIMEARG エラー・メッセージ262	FOR\$IOS_TOOMANREC エラー・メッセージ 262
FOR\$IOS_SHORTZONEARG エラー・メッセージ262	FOR\$IOS_TOOMANVAL エラー・メッセージ 262
FOR\$IOS_SIGFPE エラー・メッセージ262	FOR\$IOS_UFMTENDIAN エラー・メッセージ 262
FOR\$IOS_SIGINT エラー・メッセージ262	FOR\$IOS_UNFIO_FMT エラー・メッセージ 262
FOR\$IOS_SIGIOT エラー・メッセージ262	FOR\$IOS_UNIALROPE エラー・メッセージ 262
FOR\$IOS_SIGQUIT エラー・メッセージ262	FOR\$IOS_UNINOTCON エラー・メッセージ 262
FOR\$IOS_SIGSEGV エラー・メッセージ262	FOR\$IOS_WRIREADFIL エラー・メッセージ 262
FOR\$IOS_SIGTERM エラー・メッセージ262	FOR_ACCEPT 環境変数 253
FOR\$IOS_STKOVF エラー・メッセージ262	FOR_DIAGNOSTIC_LOG_FILE 環境変数 253
FOR\$IOS_STRLENERR エラー・メッセージ262	FOR_DISABLE_DIAGNOSTIC_DISPLAY 環境変数 253
FOR\$IOS_SUBRNG エラー・メッセージ262	FOR_DISABLE_STACK_TRACE 環境変数 253
FOR\$IOS_SUBSTRERR エラー・メッセージ262	FOR_IGNORE_EXCEPTIONS 環境変数 253
	FOR_K_FP_NEG_DENORM シンボル132

FOR_K_FP_NEG_INF シンボル	132	FORT_CONVERT.ext 環境変数	142, 253
FOR_K_FP_NEG_NORM シンボル	132	FORT_CONVERT_ext を使用する方法	142
FOR_K_FP_NEG_ZERO シンボル	132	FORT_CONVERT_ext 環境変数	142, 253
FOR_K_FP_POS_DENORM シンボル	132	FORT_CONVERTn を使用する方法	141
FOR_K_FP_POS_INF シンボル	132	FORT_CONVERTn 環境変数	253
FOR_K_FP_POS_NORM シンボル	132	fortcom	8, 256, 257
FOR_K_FP_POS_ZERO シンボル	132	Fortran	
FOR_K_FP_QNAN シンボル	132	演算子	119
FOR_K_FP_SNAN シンボル	132	Fortran 95/90 ポインタ	112
for_main.o ファイル	246	Fortran I/O	
FOR_NOERROR_DIALOGS 環境変数	253	概要	149
FOR_PRINT 環境変数	253	Fortran PowerStation との互換性..	189
FOR_READ 環境変数	253	Fortran/C 言語が混在したプログラム	
FOR_TYPE 環境変数	253	概要	230
fordef.for ファイル	132	FORTTRAN-66 解釈	66
FORM 指定子		-fp コンパイラ・オプション	79
OPEN 文	153, 160	FP_CLASS 組込み関数	132
FORMAT 文		-fp_port コンパイラ・オプション	60
前処理	9	FPATH 環境変数	23, 253
FORT_BUFFERED 環境変数	253	-fpconstant コンパイラ・オプション	60
FORT_CONVERT.ext を使用する方法	142	-fpe コンパイラ・オプション	60

-fpic コンパイラ・オプション	33, 71	I/O	
fpp	8, 9, 256, 257	レコード I/O 文指定子	172
-fpp コンパイラ・オプション	95	事前結合されたファイル	165
-fpscomp コンパイラ・オプション	42	論理ユニット	150
-fpstkchk コンパイラ・オプション	60	-i_dynamic コンパイラ・オプション	68
-fr32 コンパイラ・オプション	60	i386 プリプロセッサ・シンボル	27
-free コンパイラ・オプション	66	IA64 プリプロセッサ・シンボル	27
FREEFORM コンパイラ・ディレクティブ	40	ias アセンブラ	8, 10
-fsource-asm コンパイラ・オプション	93	ias ファイル	257
-ftz コンパイラ・オプション	60	IBM キーワード	136
-fverbose-asm コンパイラ・オプション	93	icrt.internal.map ファイル	246
-fvisibility コンパイラ・オプション	71	icrt.link ファイル	246
-fvisibility-keyword コンパイラ・オプション	71	idb デバッガ	
G		デバッグを参照	100
-g コンパイラ・オプション	71, 101	IEEE S_floating 形式	136
-gp コンパイラ・オプション	79	IEEE T_floating 形式	136
H		IEEE X_floating 形式	136
-help コンパイラ・オプション	71	ifc ファイル	256
Hollerith データ表現	134	ifc.cfg ファイル	256
I		ifcore_msg.cat ファイル	246
-I コンパイラ・オプション	95	ifort コマンド	
		構文	17

複数の使用	38	INTERFACE	219
例	19	INTERFACE 文	206
ifort ファイル	256, 257	IOSTAT 指定子	238
ifort.cfg	16	-ip コンパイラ・オプション	79
ifort.cfg ファイル	24, 256, 257	-ip_no_inlining コンパイラ・オプション	79
ifortbin ファイル	256, 257	-ip_no_pinlining コンパイラ・オプション	79
IFORTCFG 環境変数	16, 24, 253	-IPF_ft_eval_method0 コンパイラ・オプション	60
ifortvars.csh	16	-IPF_ftacc コンパイラ・オプション	60
ifortvars.csh ファイル	17, 256, 257	-IPF_fma コンパイラ・オプション	60
ifortvars.sh	16	-IPF_fp_relaxed コンパイラ・オプション	60
ifortvars.sh ファイル	17, 256, 257	-IPF_fp_speculation コンパイラ・オプション	60
ifportlib.a ライブラリ	247	-ipo[n] コンパイラ・オプション	79
-implicitnone コンパイラ・オプション	71	-ipo_c コンパイラ・オプション	79
-inline_debug_info コンパイラ・オプション	71	-ipo_obj コンパイラ・オプション	79
INQUIRE 文	151, 168	-ipo_S コンパイラ・オプション	79
INTEGER コンパイラ・ディレクティブ	40	-ipo_separate コンパイラ・オプション	79
INTEGER(IKIND=8) データの表現	126	ISNAN 組込み関数	132
INTEGER(KIND=1) データの表現	125	-ivdep_parallel コンパイラ・オプション	79
INTEGER(KIND=2) データの表現	125	L	
INTEGER(KIND=4) データの表現	126	-L コンパイラ・オプション	68
-integer_size コンパイラ・オプション	52		

ld	
リンクを参照	10
ld 8	
LD_LIBRARY_PATH 環境変数 . 33, 252, 253	
libcprts.a file	246
libcprts.so ファイル	246
libcprts.so.5 ファイル	246
libcxa.a ファイル	246
libcxa.so ファイル	246
libcxa.so.5 ファイル	246
libcxaguard.a ファイル	246
libcxaguard.so ファイル	246
libcxaguard.so.5 ファイル	246
libguide.a ファイル	246
libguide.so ファイル	246
libguide_stats.a ファイル	246
libguide_stats.so ファイル	246
libifcore.a ファイル	246
libifcore.so ファイル	246
libifcore.so.5 ファイル	246
libifcoremt.a ファイル	246
libifcoremt.so ファイル	246
libifcoremt.so.5 ファイル	246
libifport.a ファイル	246
libifport.a ライブラリ	
使用	248
libifport.so ファイル	246
libifport.so.5 ファイル	246
libimf.a ファイル	246
libimf.a ライブラリ	252
libimf.so ファイル	246
libirc.a ファイル	246
libm.a ライブラリ	252
libompstub.a ファイル	246
libsvml.a ファイル	246
libunwind.a ファイル	246
libunwind.so ファイル	246
libunwind.so.5 ファイル	246
linux プリプロセッサ・シンボル	27
LITTLE_ENDIAN キーワード	136
-logo コンパイラ・オプション	71
M	
make コマンド	
使用	17

makefile 17, 20

Microsoft Fortran PowerStation との
互換性 189

Microsoft との互換性 189

-mixed_str_len_arg コンパイラ・オプション 59, 199, 226

-module コンパイラ・オプション 93

-mp1 コンパイラ・オプション 60

N

-names コンパイラ・オプション 59

NATIVE キーワード 136

-nbs コンパイラ・オプション 42

NLSPATH 環境変数 234, 253

-no_cpprt コンパイラ・オプション 68

-noalign コンパイラ・オプション 52

-noauto コンパイラ・オプション 52

-noautomatic コンパイラ・オプション 52

-nobss_init コンパイラ・オプション 71

NODECLARE コンパイラ・ディレクティ
ブ 40

-nodefaultlibs コンパイラ・オプション 68

-nofor_main コンパイラ・オプション 71

NOFREEFORM コンパイラ・ディレク
ティブ 40

-noinclude コンパイラ・オプション 71

-nolib_inline コンパイラ・オプション 79

-nomixed_str_len_arg コンパイラ・オプ
ション 59, 199, 226

-nosave コンパイラ・オプション 52

-nostartfiles コンパイラ・オプション 71

-nostdinc コンパイラ・オプション 71

-nostdlib コンパイラ・オプション 68

NOSTRICT コンパイラ・ディレクティ
ブ 40

-Nso アセンブラ・オプション 10

-nus コンパイラ・オプション 59

O

-O コンパイラ・オプション 79, 93

-onetrip コンパイラ・オプション 42

OPEN

暗黙 161

OPEN 文

DEFAULTFILE 指定子 163

FILE 指定子 163

FORM 指定子 153, 160

ORGANIZATION 指定子 155

POSITION 指定子 175

RECL 指定子	160	-opt_report_help コンパイラ・オプション	79
STATUS 指定子	156	-opt_report_level コンパイラ・オプション	79
USEROPEN 指定子	178	-opt_report_phase コンパイラ・オプション	79
ファイル共有	174	-opt_report_routine コンパイラ・オプション	79
ファイル名の割り当て	161	OPTIONS 文	38
指定子	165	OPTIONS 文を使用する方法	147
事前結合されたファイル	165	ORGANIZATION 指定子	
OPEN 文	151, 161	OPEN 文	155
OPEN 文 CONVERT を使用する方法	146	P	
-openmp コンパイラ・オプション	66	-P コンパイラ・オプション	95
-openmp_profile コンパイラ・オプション	71	-p 最適化コンパイラ・オプション	79
-openmp_report コンパイラ・オプション	47	-p32 アセンブラ・オプション	10
-openmp_stubs コンパイラ・オプション	66	PACK コンパイラ・ディレクティブ	40
opt/intel_fc_80/bin ディレクトリ	16	-pad コンパイラ・オプション	71
opt/intel_fc_80/bin/ifortvars.csh ファイル	17	-pad_source コンパイラ・オプション ...	66
opt/intel_fc_80/bin/ifortvars.sh ファイル	17	-par_report コンパイラ・オプション	47
-opt_report コンパイラ・オプション	79	-par_threshold コンパイラ・オプション	79
-opt_report_file コンパイラ・オプション	79	-parallel コンパイラ・オプション	79
		PATH 環境変数	253
		-pc コンパイラ・オプション	60

-pg コンパイラ・オプション 52

POSITION 指定子

OPEN 文 175

PowerStation との互換性 189

-prec_div コンパイラ・オプション 71

-prefetch コンパイラ・オプション 79

-preprocess_only コンパイラ・オプション 95

PRINT 文 151, 153, 161, 172

-prof_dir コンパイラ・オプション 79

-prof_file コンパイラ・オプション 79

-prof_format_32 コンパイラ・オプション 42

-prof_gen コンパイラ・オプション 79

-prof_use コンパイラ・オプション 79

profdcg ファイル 257

profmerge ファイル 256, 257

proforder ファイル 256, 257

Q

-Qinstall コンパイラ・オプション 93

-Qlocation コンパイラ・オプション 26, 93

-Qoption コンパイラ・オプション 26, 93

R

-r コンパイラ・オプション 52

RANDOM_NUMBER 組込みサブルーチン 248

RANDOM_SEED 組込みサブルーチン 248

-rcd コンパイラ・オプション 71

READ 文

ADVANCE 指定子 176

READ 文 151, 153, 161, 172

READONLY 指定子

OPEN 文 174

REAL コンパイラ・ディレクティブ 40

REAL データの表現 129

REAL(KIND=16) データの表現 130

REAL(KIND=4) データの表現 129

REAL(KIND=8) データの表現 129

-real_size コンパイラ・オプション 52

RECL 指定子

OPEN 文 160

RECL 値 155

-recursive コンパイラ・オプション 41

-reentrancy コンパイラ・オプション 41

REFERENCE プロパティ	199	-stand95 コンパイラ・オプション	47
REWIND 文	151, 172	-static コンパイラ・オプション	68
REWRITE 文	151, 153, 172	-static-libcxa コンパイラ・オプション	68
S		STATUS 指定子	
-S オプション	10	OPEN 文	156
-S コンパイラ・オプション	93	-std コンパイラ・オプション	47
-safe_cray_ptr コンパイラ・オプション	52	-std90 コンパイラ・オプション	47
-save コンパイラ・オプション	52	-std95 コンパイラ・オプション	47
-scalar_rep コンパイラ・オプション	79	Stream レコード型	157, 160
setenv コマンド	16	Stream_CR レコード	185
-shared コンパイラ・オプション	68	Stream_CR レコード型	157, 160
shared-file チェック	174	Stream_LF レコード	185
-shared-libcxa コンパイラ・オプション	68	Stream_LF レコード型	157, 160
sigaction ルーチン		STRICT コンパイラ・ディレクティブ	40
呼び出し	241	-syntax コンパイラ・オプション	71
signal ルーチン		-syntax_only コンパイラ・オプション	71
呼び出し	241	T	
-size_lp64 コンパイラ・オプション	71	-T コンパイラ・オプション	71
-sox コンパイラ・オプション	41	TBK_ENABLE_VERBOSE_STACK_TRACE 環境変数	253
SQUARES サンプル・プログラム	107	TBK_FULL_SRC_FILE_SPEC 環境変数	253
-stand コンパイラ・オプション	47	-tcheck コンパイラ・オプション	71
-stand90 コンパイラ・オプション	47		

TEMP 環境変数.....	14, 253	USE IFPORT 文	248
-Tf コンパイラ・オプション	71	-use_asm コンパイラ・オプション	93
-threads コンパイラ・オプション	68	USEROPEN ルーチン	161
TMP 環境変数	14, 253	USEROPEN 指定子	
TMPDIR 環境変数	14, 253	OPEN 文	178
-tpp コンパイラ・オプション	79	V	
-traceback コンパイラ・オプション	98	-v コンパイラ・オプション	71
TRACEBACKQQ ルーチン	243	VALUE プロパティ	199
tselect.....	256	VARYING プロパティ	199, 205
tselect ファイル	256, 257	VAXD キーワード	136
TYPE 文	151, 153, 161, 172	VAXG キーワード	136
U		-vec_report コンパイラ・オプション	47
-u コンパイラ・オプション	71, 95	-vms コンパイラ・オプション	42
Unicode		W	
文字	11	-w90 コンパイラ・オプション	47
uninstall.sh.....	256	-w95 コンパイラ・オプション	47
uninstall.sh ファイル	256, 257	-warn コンパイラ・オプション	47
unix プリプロセッサ・シンボル	27	-what コンパイラ・オプション	71
UNLOCK 文	151	-WI コンパイラ・オプション	71
-unroll コンパイラ・オプション	79	-Wp コンパイラ・オプション	95
unset コマンド	16	WRITE 文	
unsetenv コマンド	16	ADVANCE 指定子	176

WRITE 文 151, 153, 161, 172

X

-x コンパイラ・オプション 71, 79

xiar ファイル 256, 257

xild ファイル 256, 257

-Xlinker オプション 71

Y

-y コンパイラ・オプション 71

Z

-zero コンパイラ・オプション 52

-Zp コンパイラ・オプション 52

あ

アセンブラ 8, 10

アドバンシング I/O 176

アドレス

共通ブロック、受け渡し 211

アプリケーションのビルド

概要 15

アライメント

コンパイラ・オプション 52

アライメントされていないデータ

検索 121

い

インクルード・ファイル

検索 23

インストール

共用ライブラリ 33

え

エラー・メッセージ

ランタイム 262

エラーの処理 234

エラー処理

ランタイム 238

概要 234

エラー処理 234

エラー処理の機能

OPEN 文指定子 165

エラー処理ルーチン 248

お

オプションに関するヘルプを表示する
..... 38

オプションの上書き 38

か

ガイド

使い方 4

き	コンパイル診断.....47
キャリッジ制御	その他.....71
-ccdefault コンパイラ・オプション . 71	データ.....52
く	プリプロセッサ.....95
グローバル変数	ヘルプ.....38
言語が混在したプログラミングにお ける使用.....211	ライブラリ.....68
こ	ランタイム.....98
コードカバレッジ・ツール.....256, 257	外部プロシージャ.....59
コードの生成オプション.....41	概要.....37
コマンド	形式.....38
デバグ	言語.....66
概要.....105	互換性.....42
表現.....119	最適化.....79
デバグ.....103	出力ファイル.....93
コマンドライン出力	詳細.....38
リダイレクト.....30	コンパイラ・オプション.....60
コンパイラ	コンパイラ・オプションのヘルプ.....38
コンポーネント.....3	コンパイラ・ディレクティブ.....40
デフォルト動作.....11	コンパイラのコンポーネント.....3
起動.....17	コンパイラのデフォルト動作.....11
コンパイラ・オプション	コンパイラのバージョン
code generation.....41	相違点.....260

コンパイラの起動	システム、ドライブ、またはディレクトリ 制御および照会ルーチン	248
デバッグ	101	
概要	6	
コンパイラの制限	258	
コンパイル・フェーズ	8	
コンパイル時の環境変数	253	
コンパイル処理		
制御	16	
コンパイル診断オプション	47	
さ		
サイズ		
実行プログラム	258	
サブプログラム		
メイン・プログラムからの呼び出し	195	
サンプル・プログラム		
SQUARES	107	
し		
シーケンシャル・アクセス		
レコード	172	
シーケンシャル・ファイル編成	155	
シェル・スクリプト		
実行	17	
シンボル		
定義済みプリプロセッサ	27	
シンボルテーブル情報	101	
す		
スクラッチ・ファイル	156	
スタティック・ライブラリ	245	
ストリーミング SIMD 拡張命令 (SSE) .	7	
ストリーミング SIMD 拡張命令 2 (SSE)	7	
ストリーム・ファイル	185	
せ		
セグメント・レコード	185	
セグメント・レコード型	157, 160	
セル番号	155	
そ		
ソケット	161	
その他のオプション	71	
た		
ダイナミック共通ブロック		
メモリの割り当て	36	
使用のガイドライン	36	

つ	対応するデバugga名	112
ツール	文字.....	226
オプションを渡す	データ型	112
場所.....	データ交換	
て	言語が混在したプログラミング	208
ディレクティブ	データ書式	
データ	指定	
アライメントされていない	-convert コンパイラ・オプションを 使用する方法	148
検索	OPEN 文 CONVERT を使用する 方法.....	146
データ・オプション	OPTIONS 文を使用する方法 .	147
データ・プリフェッチ	環境変数 F_UFMTENDIAN を使 用する方法	143
データアクセス	環境変数 FORT_CONVERT.ext を 使用する方法	142
言語が混在したプログラミング	環境変数 FORT_CONVERT_ext を使用する方法	142
データのアライメント	環境変数 FORT_CONVERTn を使 用する方法	141
コンパイラ・オプション	指定.....	139
データの格納	データ書式の指定方法	
データ型	概要.....	139
Fortran と C	データ表現	
言語が混在したプログラミングにお ける処理	COMPLEX.....	130
概要		
書式なしファイルの変換		
組込み		

COMPLEX(16).....	131	デノーマルな数値.....	132
COMPLEX(4).....	130	デバッグ	
COMPLEX(8).....	131	デバッグを参照.....	100
COMPLEX(KIND=16).....	131	デバッグ	
COMPLEX(KIND=4).....	130	SQUARES サンプル・プログラム.....	107
COMPLEX(KIND=8).....	131	アライメントされていないデータの検 索.....	121
COMPLEX*16.....	131	オプション.....	101
COMPLEX*32.....	131	コマンド	
COMPLEX*8.....	130	概要.....	105
DOUBLE COMPLEX.....	131	コマンド.....	103
DOUBLE PRECISION.....	129	プログラムの準備.....	102
EXTENDED PRECISION.....	130	概要.....	100, 101
Hollerith.....	134	言語が混在したプログラム.....	119
REAL.....	129	信号を生成するプログラム.....	120
REAL(KIND=16).....	130	表現.....	119
REAL(KIND=4).....	129	変数の表示.....	112
REAL(KIND=8).....	129	デフォルト	
ネイティブ IEEE 浮動小数点の表現	127	パス名.....	163
概要.....	122	ファイル名.....	163
整数.....	122	と	
文字.....	134	ドキュメント	
論理.....	126	その他.....	4

ドキュメント規則	4	アクセス	161
トレースバック情報		アンロック	151
取得	243	スクラッチ	156
トレースバック情報	101	レコード・オーバーヘッド	160
ね		レコード型	157
ネームリスト I/O 文	153	一時	14
の		開く	165
ノン・アドバンシング I/O	176	概要	155
ノン・アドバンシング・レコード I/O ..	176	割り当て	161
は		事前結合	161
バージョン間の相違点	260	出力	13
パイプ	161	内部	156
パス名		複数	
デフォルト		コンパイルとリンク	19
適用規則	163	ファイルアクセス	
絶対	12	OPEN 文指定子	165
相対	12	ファイルのアクセス	161
パラレライザ	7	ファイルのアンロック	151
ひ		ファイルの割り当て	161
ビッグエンディアン格納	136	ファイルの場所	
ふ		OPEN 文におけるコーディング ...	165
ファイル		ファイルの特性	

OPEN 文指定子.....	165	ファイル名拡張子.....	11
概要.....	155	フェーズ	
ファイルの編成.....	155, 172	コンパイル.....	8
ファイルを閉じる		前処理.....	9
CLOSE 文を参照.....	171	プリプロセッサ・オプション.....	95
ファイルを閉じる動作		プリプロセッサ・シンボル.....	27
OPEN 文指定子.....	165	ブレイクポイント	
ファイル位置		設定.....	103
OPEN 文指定子.....	165	ブレイクポイント.....	103
ファイル共有.....	174	プログラム	
ファイル指定.....	12, 163	作成、実行、デバッグ.....	30
ファイル処理		プロシージャ	
OPEN 文指定子.....	165	プロトタイピング.....	206
ファイル情報		ユーザが提供する OPEN.....	178
OPEN 文指定子.....	165	プロシージャのプロトタイピング.....	206
ファイル情報の取得		プロシージャ名	
INQUIRE 文を参照.....	168	言語が混在したプログラミング....	204
ファイル名		プロセス制御ルーチン.....	248
デフォルト		プロセッサ・ディスパッチ.....	7
適用規則.....	163	プロファイルに基づく最適化.....	7
照会.....	168	ほ	
ファイル名による照会.....	168	ポインタ	

言語が混在したプログラミングにおける受け取り	220	モジュール変数	112
言語が混在したプログラミングにおける渡し方	220	ゆ	
ポインタ変数	112	ユーザーズ・ガイド	
ま		使い方	4
マクロ		ユーザが提供する OPEN プロシージャ	178
プリプロセッサ・シンボルを参照	27	ユーザ定義型	
マップファイル	98	処理	229
マニュアル		ユニット	
その他	4	照会	168
マルチバイト文字	11	ユニットによる照会	168
め		ユニット情報	
メイン・プログラムからのサブプログラムの呼び出し	195	OPEN 文指定子	165
も		ら	
モジュール		ライブラリ	
プログラムのコンパイル	20	共用の作成	33
言語が混在したプログラミングにおける使用	231	作成	245
モジュール (.mod) ファイル		使用	
マルチディレクトリ	20	概要	244
検索	23	ライブラリ	246
使用	20	ライブラリ・オプション	68
		ライブラリの作成	245
		ランタイム	

環境変数.....	253	リンクの防止.....	19
ランタイム・エラー		れ	
処理.....	238	レコード I/O	
ランタイム・オプション	98	アドバンシング	176
ランタイム・チェック	98	ノン・アドバンシング	176
ランタイム・ライブラリ (RTL)		レコード I/O	176
idb	120	レコード I/O 文指定子	172
ランタイム・ライブラリ (RTL) のデフォルト例外ハンドラ.....	243	レコード・オーバーヘッド	160
ランタイム・ライブラリのデフォルトのエラー処理.....	234	レコードアクセス.....	172
ランタイム時のエラー・メッセージ		レコードサイズ	177
リスト.....	262	レコードロック	174
リ		レコード位置	
リスト指定 I/O 文	153	開始の指定	175
リダイレクト		変更.....	175
コマンドライン出力	30	レコード型	
リトルエンディアン格納	136	形式.....	185
リンカ	10	選択.....	157
リンカバイナリ		レコード型	157, 172, 185
指定	19	レコード操作	
リンク		概要.....	171
防止	19	レコード長	160
		レコード転送	177

レコード転送の特性	ファイル.....165, 172
OPEN 文指定子.....165	外部プロシージャ・オプション.....59
レコード特性	概要
OPEN 文指定子.....165	Fortran I/O.....149
レコード変数.....112	Fortran/C が混在したプログラム230
漢字	アプリケーションのビルド.....15
暗黙的な OPEN.....161, 163	エラー処理.....234
以前のバージョンとの互換性.....260	コンパイラ・オプション.....37
移植ライブラリ	コンパイラの起動.....6
概要.....247	データ書式の指定方法.....139
使用.....248	データ表現.....122
移植ライブラリ.....248	デバッグ.....100
移植ルーチン.....248	ネイティブ IEEE 浮動小数点の表現127
引数の渡し方	ファイルとファイルの特性.....155
Fortran と C.....233	ライブラリの使用.....244
言語が混在したプログラミング.....208	レコード操作.....171
演算子	移植ライブラリ.....247
Fortran.....119	言語が混在したプログラミング
応答ファイル.....24	呼び出し規則の調整.....197
可変長レコード.....185	命名規則の調整.....202
可変長レコード型.....157, 160	言語が混在したプログラミング.....194
開く	

言語が混在したプログラミングにおけるデータ型の処理.....	215	環境変数 F_UFMTENDIAN を使用する 方法.....	143
混在言語の問題.....	196	環境変数 FORT_CONVERT.ext を使用 する方法.....	142
書式なしデータの変換	135	環境変数 FORT_CONVERT_ext を使 用する方法.....	142
整数データの表現.....	125	環境変数 FORT_CONVERTn を使用す る方法	141
拡張子		間接コマンドファイル	
ファイル名	11	応答ファイルを参照	24
格納		機能	3
ビッグエンディアン.....	136	規則	
リトルエンディアン	136	デフォルトのパス名.....	163
確認		ドキュメント	4
ランタイム	98	共通ブロック	
割り当て		アドレスを渡す	211
共通ブロック	36	割り当て.....	36
割付け配列		言語が混在したプログラミングにお ける使用.....	211
処理	219	共通ブロック変数.....	112
環境変数		共通外部データ構造.....	211
コンパイル時	253	共用ライブラリ	
シェル・スクリプトの実行による設定	17	インストール	33
ランタイム	253	作成.....	33
設定	16	制限.....	33
表示	16		

共用ライブラリ	245	呼び出し規則	199
型		呼び出し規則の調整の概要	197
I/O 文	151	数値データ型	217
ユーザ定義	229	複素数データ型	217
形式		名前の大文字・小文字の調整	204
レコード型	185	命名規則の調整の概要	202
検索		戻り値	217, 226
インクルード・ファイル	23	問題の概要	196
モジュール (.mod) ファイル	23	論理データ型	217
言語オプション	66	言語が混在したプログラム	
言語が混在したプログラミング		デバッグ	119
ATTRIBUTES プロパティ	199	呼び出し規則	
C/C++ 命名規則	203	ATTRIBUTES プロパティ	199
データアクセス	208	言語が混在したプログラミング	199
データ型の処理	215	説明	194
データ交換	208	呼び出し規則の調整	
プロシージャ名	204	言語が混在したプログラミング	197
メイン・プログラムからのサブプログラムの呼び出し	195	固定形式	
モジュールの使用	231	コンパイラ・オプション	66
引数の渡し方	208	固定形式ファイル	11
外部データの使用	211	固定長レコード	185
概要	194	固定長レコード型	157, 160

互換性	作成、実行、デバッグ.....30
Microsoft Fortran PowerStation .189	主なファイル
互換性オプション.....42	IA-32.....256
最適化オプション.....79	Itanium ベース.....257
作成	出力
共用ライブラリ.....33	リダイレクト.....30
実行プログラム.....30	出力ファイル
参照による引数の受け渡し.....208	名前の変更.....19
使用	出力ファイル.....13
ユーザーズ・ガイド.....4	出力ファイル・オプション.....93
指定	出力ファイル名の変更.....19
データ書式.....139	出力レコード転送.....177
ファイル.....12	出力項目リスト
ファイル名.....163	照会.....168
事前結合されたファイル	出力項目リストによる照会.....168
使用.....172	書式なし I/O 文.....153
事前結合されたファイル.....161, 165	書式なしシーケンシャル・ファイル
自動ベクトライザ.....7	Microsoft Fortran PowerStation ..189
自由形式	書式なしデータ
コンパイラ・オプション.....66	優先順位.....139
自由形式ファイル.....11	書式なしデータの変換
実行プログラム	概要.....135

書式なし直接ファイル

Microsoft Fortran PowerStation ... 189

書式付き I/O 文 153

書式付きシーケンシャル・ファイル

Microsoft Fortran PowerStation ... 189

書式付き直接ファイル

Microsoft Fortran PowerStation ... 189

上書き

デフォルトのランタイム・ライブラリ例
外ハンドラの上書き 243

情報取得ルーチン 248

信号

プログラムのデバッグ 120

説明 241

信号処理 241

新機能の概要 3

数値データ型

処理 217

数値ルーチンと変換ルーチン 248

数値演算ライブラリ 252

数値形式

ネイティブ 136

非ネイティブ 136

数値変換

制限 139

制限

コンパイラ 258

共用ライブラリの作成 33

数値変換 139

制御

コンパイル処理 16

整数データの表現

INTEGER(KIND=1) 125

INTEGER(KIND=2) 125

INTEGER(KIND=4) 126

INTEGER(KIND=8) 126

概要 125

整数ポインタ 112, 220

設定

ブレイクポイント 103

設定ファイル 24

絶対パス名 12

前処理フェーズ 9

組込みデータ型 122

相対パス名 12

相対ファイル編成	155	配列セクション	112
代替ツールの場所		配列への代入	112
指定	26	配列ポインタ	
値による引数の受け渡し	208	処理	219
調整		配列記述子	
名前の大文字・小文字	204	処理	222
直接アクセス		配列記述子の形式	
レコード	172	説明	222
定義済みプリプロセッサ・シンボル ...	27	配列宣言	222
特殊なファイルを開くルーチン		配列代入	112
OPEN 文指定子	165	配列変数	112
内部ファイル	156	非ネイティブ・データ	
日付ルーチンと時刻ルーチン	248	ポーティング	149
入力ファイル	11	非ネイティブ・データのポーティング	149
入力ルーチンと出力ルーチン	248	表記規則	4
入力レコード転送	177	表現	
派生型変数	112	デバッグコマンド	119
配列		表現ルーチン	248
C 222		浮動小数点オプション	60
Fortran	222	浮動小数点の表現	132
処理	222	部分文字列	156
配列サイズ	258	複数ファイル	

コンパイルとリンク.....	19	名前の大文字・小文字	
複素数データ型		調整.....	204
処理	217	命名規則	
複素数変数.....	112	C/C++.....	203
文		C/Fortran が混在したプログラム	233
INTERFACE	206	言語が混在したプログラミングにお ける調整.....	202
文字データ型		免責条項.....	1
戻り.....	226	戻り	
文字データ表現	134	文字データ型.....	226
文字配列	156	戻り値	
文字配列セクション	156	引数リスト上の配置.....	217, 226
文字配列要素.....	156	例外ハンドラ	
文字変数	156	ランタイム・ライブラリ (RTL)	
文字列		上書き.....	243
処理	226	例外的な数値	
変換		識別.....	132
16 進、2 進、8 進、および 10 進	259	論理 I/O ユニット	150
変数		論理データの表現.....	126
デバッガでの表示	112	論理データ型	
変数の表示		処理.....	217
デバッグ	112		