Enterprise PL/I for z/OS
6.2

プログラミング・ガイド

GI13-5620-00

目次

表 xv

図 xvii

はじめに xix
本書について xix
Enterprise PL/I for z/OS のランタイム環境 xix
資料の使用 xix
本書で使用されている表記規則 xx
使用する規則 xx
構文記法の読み方 xxi
表記記号の読み方 xxiii
変更の要約 xxiii
本リリースにおける機能強化 xxiii
からの強化 V6R1 xxiv
V5R3 の機能拡張 xxvi
V5R2 からの機能拡張 xxviii
V5R1 からの機能拡張 xxx
V4R5 からの機能拡張 xxxii
V4R4 からの機能強化 xxxiii
V4R3 からの機能拡張 xxxiii
V4R2 からの機能拡張 xxxv
V4R1 からの機能拡張 xxxvi
V3R9 からの機能拡張 xxxvii
V3R8 からの機能拡張 xxxviii
V3R7 からの機能拡張 xxxix
V3R6 からの機能拡張 xl
V3R5 からの機能拡張 xlii
V3R4 からの機能拡張 xliii
V3R3 からの機能拡張 xliv
V3R2 からの機能拡張 xlv
V3R1 からの機能拡張 xlvii
VisualAge PL/I からの機能拡張 xlvii
ご意見の送付方法 xlviii
アクセシビリティー xlix

第 1 部プログラムのコンパイル 1

第 1 章コンパイラー・オプションと機能の使用 3
コンパイル時オプションの説明 3
AGGREGATE 9
ARCH 9
ASSERT 10
ATTRIBUTES 10
BACKREG 11
BIFPREC 11
BLANK 12
BLKOFF 12
BRACKETS 12

CASE....13
CASERULES....13
CHECK....14
CMPAT....14
CODEPAGE....15
COMMON....16
COMPILE....16
COPYRIGHT....17
CSECT....17
CSECTCUT....18
CURRENCY....18
DBCS....18
DBRMLIB....19
DD....19
DDSQL....20
10 進数....20
DECOMP....23
DEFAULT....23
DEPRECATE....32
DEPRECATENEXT....33
DISPLAY....34
DLL....34
DLLINIT....34
出口....35
EXPORTALL....35
EXTRN....35
FILEREF....35
FLAG....36
FLOAT....36
FLOATINMATH....38
GOFF....38
GONUMBER....39
GRAPHIC....39
HEADER....40
HGPR....40
IGNORE....40
INCAFTER....41
INCDIR....41
INCLUDE....42
INCPDS....42
INITAUTO....42
INITBASED....43
INITCTL....44
INITSTATIC....44
INSOURCE....44
INTERRUPT....45
JSON....45
LANGLVL....47
LIMITS....48
LINEDIR....49
LIST....50
LISTVIEW....50
LP....51
MACRO....51
MAP....52
MARGINI....52
MARGINS....52
MAXBRANCH....53

MAXINIT....54
MAXGEN....54
MAXMEM....54
MAXMSG....55
MAXNEST....55
MAXRUNONIF....56
MAXSTATIC....56
MAXSTMT....56
MAXTEMP....57
MDECK....57
MSGSUMMARY....57
NAME....58
NAMES....58
NATLANG....59
NEST....59
NOT....59
NULLDATE....59
OBJECT....60
OFFSET....60
OFFSETSIZE....60
ONSNAP....60
OPTIMIZE....61
OPTIONS....62
OR....62
PP....62
PPCICS....63
PPINCLUDE....64
PPLIST....64
PPMACRO....64
PPSQL....65
PPTRACE....65
PRECTYPE....65
PREFIX....66
PROCEED....66
プロセス....67
QUOTE....67
REDUCE....67
RENT....68
RESEXP....69
RESPECT....69
RTCHECK....69
RULES....70
SEMANTIC....86
サービス....87
ソース....87
SPILL....87
STATIC....87
STDSYS....88
STMT....88
ストレージ....88
STRINGOFGRAPHIC....89
SYNTAX....89
SYSPARM....89
SYSTEM....90
端末....91
TEST....91
UNROLL....94
USAGE....94

WIDECHAR....96
WINDOW....97
WRITABLE....97
XINFO....98
XML....101
XREF....102
オプションの中のブランク、コメント、およびストリング....102
デフォルト・オプションの変更....103
%PROCESS または *PROCESS ステートメントでオプションを指定する....103
% ステートメントの使用....104
%INCLUDE....105
コンパイラー・リストの使用....106
見出し情報....106
コンパイルに使用するオプション....106
プリプロセッサー入力....107
SOURCE プログラム....107
ステートメントのネスト・レベル....107
ATTRIBUTE および相互参照表....107
集合長さテーブル....108
ステートメント・オフセット・アドレス....109
ストレージ・オフセット・リスト....110
式および属性のリスト....111
ファイル参照テーブル....111
メッセージと戻りコード....112
例....113

第 2 章 PL/I プリプロセッサー....117
インクルード・プリプロセッサー....117
マクロ・プリプロセッサー....118
マクロ・プリプロセッサーのオプション....118
マクロ・プリプロセッサーの例....122
SQL プリプロセッサー....123
プログラミングとコンパイルに関する考慮事項....123
SQL プリプロセッサー・オプション....125
PL/I アプリケーションにおける SQL ステートメントのコーディング....129
LOB データを操作する....140
SQL プリプロセッサ・メッセージの抑制....143
CICS プリプロセッサー....143
プログラミングとコンパイルに関する考慮事項....143
CICS プリプロセッサー・オプション....144
PL/I アプリケーション内での CICS ステートメントのコーディング....144
PL/I を使用した CICS トランザクションの作成....145
エラー処理....145

第 3 章 PL/I カタログ式プロシージャーの使用....147
IBM 提供のカタログ式プロシージャー....147
コンパイルのみ (IBMZC)....148
コンパイルとバインド (IBMZCB)....149
コンパイル、バインド、実行 (IBMZCBG)....150
コンパイルのみ - 64 ビット (IBMQC)....151
コンパイルとバインド - 64-bit (IBMQCB)....152
コンパイル、バインド、実行 - 64 ビット (IBMQCBG)....153
カタログ式プロシージャーの呼び出し....154
複数のカタログ式プロシージャー呼び出しを指定....154
PL/I カタログ式プロシージャーの変更....155
EXEC ステートメント....155
DD ステートメント....156

第 4 章プログラムのコンパイル....157
z/OS UNIX の下でのコンパイラーの呼び出し....157
入力ファイル....157
z/OS UNIX の下でのコンパイル時オプションの指定....158
-qoption_keyword....158
単一フラグおよび複数文字フラグ....159
z/OS 、コンパイラを起動するには JCL....159
EXEC ステートメント....160
標準データ・セット用の DD ステートメント....160
リスト (SYSPRINT)....162
ソース・ステートメント・ライブラリ (SYSLIB)....162
オプションの指定....163
EXEC ステートメントでオプションを指定する....163
EXEC 、オプション・ファイルを使ってオプションを指定する....163

第 5 章 31 ビット・プログラムに対するリンク・エディットおよび実行....165
31 ビット・プログラムに関するリンク・エディットの考慮事項....165
31 ビット・プログラムでバインダーを使用....165
ENTRY カードの使用....165
31 ビット・プログラムに関する実行時の考慮事項....165
PRINT ファイルの書式規則....166
31 ビットプログラム用 PRINT ファイルのフォーマット変更....166
自動プロンプト....167
自動プロンプトの指定変更....167
長い入力行の句読法....168
GET LIST と GET DATA の句読点....168
自動パディング GET EDIT....168
端子入力に SKIP を使用....169
ENDFILE....169
SYSPRINT 31 ビットプログラムへの配慮....169
MSGFILE(SYSPRINT) の使用....170
31 ビット・アプリケーションのルーチンで FETCH を使用する....170
31 ビット・アプリケーションで Enterprise PL/I ルーチンをフェッチ....170
31 ビット・アプリケーションでの PL/I MAIN ルーチンのフェッチ....173
31 ビット・アプリケーションで z/OS C ルーチンをフェッチ....174
31 ビット・アプリケーションでアセンブラー・ルーチンをフェッチ....174
TSO/E で MAIN....174
MAIN 。 z/OS UNIX....175

第 6 章 64 ビット・プログラムに対するリンク・エディットおよび実行....177
64 ビット・プログラムに関するリンク・エディットの考慮事項....177
64 ビット・プログラムでバインダーを使用....177
64 ビットプログラムでの ENTRY カードの使用....177
64 ビット・プログラムに関する実行時の考慮事項....177
SYSPRINT 64 ビットプログラムへの配慮....178
64 ビット・アプリケーションのルーチンで FETCH....178
64 ビット・アプリケーションで Enterprise PL/I ルーチンをフェッチ....178
64 ビット・アプリケーションで PL/I MAIN ルーチンをフェッチ....178
64 ビット・アプリケーションでアセンブラー・ルーチンをフェッチ....179
TSO/E で MAIN....179
MAIN 。 z/OS UNIX....180

第 7 章 64 ビット・アプリケーションを開発する場合の考慮事項....181
コンパイラー・オプションを使用して 64 ビット・アプリケーションをビルド....181
LP(64)で属性 HANDLE 、 POINTER....181
HANDLE 属性....182
POINTER 属性....182

LP(64) で ENTRY 変数を使用 ... 182
LP(64) で組み込み関数を使用 ... 182
SQL プログラムの考慮事項 ... 186
31 ビット・ルーチンとのやり取り ... 187

第 2 部入出力機能の使用 ... 191

第 8 章データ・セットとファイルの使用 ... 193
ファイルの割り振り ... 193
z/OS でのデータ・セットとファイルの関連付け ... 194
複数のファイルと 1 つのデータ・セットの関連付け ... 196
複数のデータ・セットと 1 つのファイルの関連付け ... 196
複数のデータ・セットの連結 ... 197
z/OS での zFS ファイルへのアクセス ... 197
z/OS UNIX でのデータ・セットとファイルの関連付け ... 198
環境変数の使用 ... 199
OPEN ステートメントの TITLE オプションを使用する ... 200
データ・セットに関連付けられていないファイルの使用の試み ... 201
PL/I によるデータ・セットの検索方法 ... 201
DD_DDNAME 環境変数を使った特性の指定 ... 202
データ・セット特性の設定 ... 207
ブロックおよびレコード ... 207
情報交換コード ... 208
レコード・フォーマット ... 208
データ・セットの編成 ... 210
ラベル ... 211
データ定義 (DD) ステートメント ... 211
OPEN ステートメントの TITLE オプションを使用する ... 212
PL/I ファイルとデータ・セットの関連付け ... 212
ENVIRONMENT ... 214

第 9 章ライブラリーの使用 ... 227
ライブラリーのタイプ ... 227
ライブラリーの使用 ... 227
ライブラリーの作成 ... 228
SPACE パラメーター ... 228
ライブラリー・メンバーの作成と更新 ... 228
例: コンパイルされたオブジェクト・モジュール用の新規ライブラリーの作成 ... 229
例: ロード・モジュールの既存ライブラリーへの配置 ... 229
例: ライブラリー・メンバーの更新 ... 230
ライブラリー・ディレクトリーからの情報の取り出し ... 231

第 10 章連続データ・セットの定義と使用 ... 233
ストリーム指向データ伝送の用法 ... 233
ストリーム入出力を用いたファイルの定義 ... 233
PL/I 動的割り振りを使用したストリーム・ファイルの定義 ... 233
ENVIRONMENT オプションの指定 ... 234
ストリーム入出力によるデータ・セットの作成 ... 236
ストリーム入出力によるデータ・セットへのアクセス ... 239
ストリーム入出力による PRINT ファイルの使用 ... 241
31 ビットプログラム用 SYSIN および SYSPRINT ファイルの使用 ... 244
64 ビットプログラム用の SYSIN および SYSPRINT ファイルを使用する ... 245
端末からの入力の制御 ... 245
データのフォーマット ... 246
ストリーム・ファイルおよびレコード・ファイル ... 247
PL/I ダイナミックアロケーションを使用した QSAM ファイルの定義 ... 247
大文字と小文字 ... 247

ファイルの終わり....248
COPY GET ステートメントのオプション....248

第 11 章端末への出力の制御....249
PRINT ファイルのフォーマット....249
ストリーム・ファイルおよびレコード・ファイル....249
PUT EDIT コマンドの出力....249

第 12 章レコード単位データ伝送の使用....251
レコード・フォーマットの指定....252
レコード入出力を使用したファイルの定義....252
ENVIRONMENT オプションの指定....252
CONSECUTIVE....253
ORGANIZATION(CONSECUTIVE)....253
CTLASA|CTL360....254
LEAVE|REREAD....256
レコード入出力によるデータ・セットの作成....257
必須情報....258
レコード入出力によるデータ・セットのアクセスおよび更新....258
必須情報....259
連続データ・セットの例....259

第 13 章領域データ・セットの定義と使用....263
PL/I ダイナミックアロケーションを使用した REGIONAL(1) データセットの定義....265
領域データ・セット用ファイルの定義....265
ENVIRONMENT オプションの指定....266
REGIONAL データセットでキーを使用する....266
REGIONAL(1)のデータセットを使用....266
ダミー・レコード....267
REGIONAL(1) データ・セットの作成....267
REGIONAL(1) データ・セットへのアクセスと更新....268
領域データ・セットの作成時、および領域データ・セットへのアクセス時の必須情報....270

第 14 章 VSAM データセットの定義と使用....273
PL/I ダイナミックアロケーションを使用した VSAM ファイルの定義....273
VSAM データ・セットの使用....273
VSAM データセットでプログラムを実行する....273
代替索引パスとファイルをペア化....274
VSAM 編成....274
VSAM データセットのキー....276
データ・セット・タイプの選択....277
VSAM データセット用ファイルの定義....279
ENVIRONMENT オプションの指定....280
パフォーマンス・オプション....282
代替索引パスのファイルを定義....282
VSAM データ・セットの定義....283
入力順データ・セット....283
ローディング ESDS....284
SEQUENTIAL 。 ESDS....284
キー順および索引付き入力順データ・セット....286
KSDS またはインデックス付き ESDS....288
KSDS 、またはインデックスされたファイルにアクセスするために SEQUENTIAL 。 ESDS....289
KSDS 、またはインデックスされたファイルにアクセスするために DIRECT 。 ESDS....290
を更新する KSDS....291
KSDSs 、またはインデックス付きの代替インデックス ESDSs....292
相対レコード・データ・セット....297
ローディング RRDS....299
SEQUENTIAL 。 RRDS....300

RRDS にアクセスするために DIRECT 301
VSAM 以外のデータセット用に定義されたファイルの使用..... 302
共用データ・セットの使用..... 302

第 3 部プログラムの改良..... 303

第 15 章パフォーマンスの向上..... 305
最適なパフォーマンスのためのコンパイラー・オプションの選択..... 305
OPTIMIZE..... 305
GONUMBER..... 305
ARCH..... 305
REDUCE..... 305
RULES..... 306
PREFIX..... 307
CONVERSION..... 307
FIXEDOVERFLOW..... 308
デフォルト..... 308
パフォーマンスを向上させるコンパイラー・オプションの要約..... 310
パフォーマンス向上のためのコーディング..... 311
DATA ディレクティブ入出力..... 311
入力専用パラメーター..... 311
GOTO ステートメント..... 312
ストリングの割り当て..... 312
ループ制御変数..... 312
PACKAGE 対ネストされた PROCEDURE..... 313
REDUCIBLE 関数..... 314
REPATTERN の使用..... 314
DESCLOCATOR または DESCLIST..... 314
DEFINED 対 UNION..... 314
名前付き定数対静的変数..... 315
ライブラリー・ルーチンの呼び出しの回避..... 316
ライブラリー・ルーチンのプリロード..... 317

第 4 部他の製品に対するインターフェースの使用..... 319

第 16 章ソート・プログラムの使用..... 321
ソート・プログラムの使用準備..... 321
ソート・タイプの選択..... 322
ソート・フィールドの指定..... 324
ソートするレコードの指定..... 325
ソート・プログラムに必要なストレージの決定..... 326
ソート・プログラムの呼び出し..... 327
例 1..... 329
例 2..... 329
例 3..... 329
例 4..... 329
例 5..... 329
ソートが成功したかどうかの判別..... 330
ソート・プログラム用のデータ・セットの確立..... 330
ソート・データの入出力..... 331
データ入出力処理ルーチン..... 331
E15 — 入力処理ルーチン (ソート出口 E15)..... 332
E35 — 出力処理ルーチン (ソート出口 E35)..... 334
PLISRTA の呼び出し例..... 335
PLISRTB の呼び出し例..... 336
PLISRTC の呼び出し例..... 336
PLISRTD の呼び出し例..... 337

可変長レコードのソートの例....338

第 17 章 C との ILC....341
同等なデータ・タイプ....341
単純なタイプの一致....342
struct タイプの一致....342
enum タイプの一致....342
ファイル・タイプの一致....343
C 関数を使用する....343
一致する単純パラメーター・タイプ....344
一致するストリング・パラメーター・タイプ....345
ENTRY を戻す関数....346
リンケージ....347
出力および入力の共有....349
出力の共用....349
入力の共有....349
ATTACH....349
C 標準ストリームのリダイレクト....350
サマリー....350

第 18 章 Java とのインターフェース....351
Java Native Interface (JNI)....351
Java からの PL/I プログラムの呼び出し....352
JNI サンプル・プログラム #1 -「Hello World」....352
ステップ 1: Java プログラムの作成....352
ステップ 2: Java プログラムのコンパイル....353
ステップ 3: PL/I プログラムの作成....353
ステップ 4: PL/I プログラムのコンパイルとリンク....355
ステップ 5: サンプル・プログラムの実行....356
JNI サンプル・プログラム #2 - ストリングを渡す....356
ステップ 1: Java プログラムの作成....356
ステップ 2: Java プログラムのコンパイル....357
ステップ 3: PL/I プログラムの作成....358
ステップ 4: PL/I プログラムのコンパイルとリンク....359
ステップ 5: サンプル・プログラムの実行....360
JNI サンプル・プログラム #3 - 整数の引き渡し....360
ステップ 1: Java プログラムの作成....360
ステップ 2: Java プログラムのコンパイル....361
ステップ 3: PL/I プログラムの作成....362
ステップ 4: PL/I プログラムのコンパイルとリンク....363
ステップ 5: サンプル・プログラムの実行....363
JNI サンプル・プログラム #4 - Java 呼び出し API....364
ステップ 1: Java プログラムの作成....364
ステップ 2: Java プログラムのコンパイル....364
ステップ 3: PL/I プログラムの作成....364
ステップ 4: PL/I プログラムのコンパイルとリンク....367
ステップ 5: サンプル・プログラムの実行....368
既存の Java VM へのプログラムの接続....368
Java および PL/I の同等なデータ・タイプの判別....369

第 5 部特殊プログラミング・タスク....371

第 19 章 AMODE 31 と AMODE 64 コードを混ぜる....373
間の呼び出しに必要な宣言 AMODEs....373
の間の呼び出しに必要なコンパイラとリンカのオプション。 AMODEs....373
間の通話の制限 AMODEs....373
通話パフォーマンスの向上 AMODEs....374

第 20 章 PLISAXA と PLISAXB XML パーサーを使う 377
概説 377
PLISAXA 組み込みサブルーチン 378
PLISAXB 組み込みサブルーチン 378
SAX イベント構成 379
start_of_document 379
version_information 379
encoding_declaration 379
standalone_declaration 379
document_type_declaration 379
end_of_document 380
start_of_element 380
attribute_name 380
attribute_characters 380
attribute_predefined_reference 380
attribute_character_reference 380
end_of_element 380
start_of_CDATA_section 380
end_of_CDATA_section 381
content_characters 381
content_predefined_reference 381
content_character_reference 381
processing_instruction 381
コメント 381
unknown_attribute_reference 382
unknown_content_reference 382
start_of_prefix_mapping 382
end_of_prefix_mapping 382
除外、 382
イベント関数に渡されるパラメーター 382
XML 文書のコード化文字セット 383
対応 EBCDIC コードページ 383
サポートされる ASCII コード・ページ 384
コード・ページの指定 384
例外 385
例 385
継続可能な例外コード 392
例外コードの終了 396

第 21 章 PLISAXC と PLISAXD XML パーサーを使う 401
概説 401
PLISAXC 組み込みサブルーチン 402
PLISAXD 組み込みサブルーチン 402
SAX イベント構成 403
start_of_document 403
version_information 403
encoding_declaration 403
standalone_declaration 403
document_type_declaration 404
end_of_document 404
start_of_element 404
attribute_name 404
attribute_characters 404
end_of_element 404
start_of_CDATA_section 404
end_of_CDATA_section 405
content_characters 405

processing_instruction....405
コメント....405
namespace_declare....405
end_of_input....405
unresolved_reference....406
除外、....406
イベント関数に渡されるパラメーター....406
イベントにおける差異....408
XML 文書のコード化文字セット....409
サポートされるコード・ページ....409
コード・ページの指定....410
例外....410
妥当性検査を伴う XML 文書の構文解析....410
XML スキーマ....411
を作る OSR....412
単純な文書での例....412
PLISAXC 組み込みサブルーチンの使用例....412
PLISAXD 組み込みサブルーチンの使用例....419

第 22 章 PLIDUMP の用法....431
PLIDUMP 使用上の注意....432
PLIDUMP 出力の変数を探す....433
AUTOMATIC 変数を探す....433
STATIC 変数を探す....434
CONTROLLED 変数を探す....435
保存されたコンパイル・データ....438
Copyright....438
タイム・スタンプ....438
保存されたオプション・ストリング....439
PLIDUMP_XCALL 環境変数....439

第 23 章割り込みとアテンションの処理....441
ATTENTION ON-ユニットの使用....442
デバッグ・ツールとの対話....442

第 24 章チェックポイント/再始動機能の使用....443
チェックポイント・レコードの要求....443
チェックポイント・データ・セットの定義....444
再始動の要求....445
システム障害後の自動再始動....445
プログラム内の自動再始動....445
据え置き再始動....445
チェックポイント/再始動活動の変更....445

第 25 章ユーザー出口の用法....447
コンパイラー・ユーザー出口によって実行されるプロシージャー....447
グローバル制御ブロックの構造....448
IBM -supplied compiler exit、 IBMUEXIT....449
コンパイラー・ユーザー出口の活動化....449
コンパイラー・ユーザー出口のカスタマイズ....449
変更 SYSUEXIT....450
独自のコンパイラー出口の作成....450
初期化プロシージャーの作成....451
メッセージ・フィルター操作プロシージャーの作成....451
終了プロシージャーの作成....453
SQL メッセージの抑制例....453

第 26 章 PL/I 記述子....455

引数の引き渡し ... 455
記述子リストによる引数の引き渡し ... 455
ロケーター/記述子で引数を渡す ... 456
CMPAT(V*) 記述子 ... 456
ストリング記述子 ... 456
配列記述子 ... 457
CMPAT(LE) 記述子 ... 458
ストリング記述子 ... 458
配列記述子 ... 459

付録 A SYSADATA メッセージ情報 ... 461
SYSADATA ... 461
サマリー・レコード ... 462
オプション・レコード ... 462
カウンター・レコード ... 463
リテラル・レコード ... 463
ファイル・レコード ... 463
メッセージ・レコード ... 464
SYSADATA シンボル情報を理解する ... 464
序数タイプ・レコード ... 464
序数エレメント・レコード ... 465
シンボル・レコード ... 466
SYSADATA 構文情報を理解する ... 468
ソース・レコード ... 468
トークン・レコード ... 469
構文レコード ... 469

特記事項 ... 475
商標 ... 475
参考文献 ... 477
PL/I 資料 ... 477
関連資料 ... 477
用語集 ... 479

索引 ... 501

表

1. Enterprise PL/I 付属資料の使用方法 xix

2. z/OS 言語環境プログラム 付属資料の使用方法 xx

3. Compile-time options, abbreviations, and IBM-supplied defaults 3

4. SYSTEM オプション・テーブル 90

5. リストされたメッセージの最低重大度を選択するための FLAG オプションの使用 112

6. PL/I エラー・コードと戻りコードの説明 113

7. MACRO プリプロセッサー・オプション、および IBM 提供のデフォルト値 118

8. SQL プリプロセッサー・オプション、および IBM 提供のデフォルト値 126

9. PL/I 宣言から生成される SQL データ・タイプ 135

10. SQL SQL TYPE 宣言から生成されるデータ型 136

11. SQL データ・タイプと PL/I 宣言の対応 137

12. SQL データ・タイプと SQL TYPE 宣言の対応 137

13. z/OS UNIX の下で Enterprise PL/I によりサポートされるコンパイル時オプション・フラグ 159

14. コンパイラーの標準データ・セット 160

15. PL/I ファイル宣言の属性 215

16. レコード単位データ伝送のレコード・フォーマット 218

17. PL/I レコード入出力で使用できるデータ・セット・タイプの比較 223

18. ライブラリー作成時に必要な情報 228

19. レコード・フォーマット 235

20. 連続データ・セットの作成と連続データ・セットへのアクセスで使用できるステートメントとオプション 251

21. IBM マシン・コード印刷制御文字 (CTL360) 255

22. LEAVE および REREAD オプションの影響 256

23. レコード入出力による連続データ・セットの作成: DD ステートメントの必須パラメーター..... 257

24. レコード入出力による連続データ・セットへのアクセス: DD ステートメントの必須パラメーター.. 259

25. 領域データ・セットの作成と領域データ・セットへのアクセスで使用できるステートメントとオプション..... 263

26. 領域データ・セットの作成: DD ステートメントの必須パラメーター..... 271

27. DCB 地域データセットのサブパラメータ..... 272

28. 領域データ・セットへのアクセス: DD ステートメントの必須パラメーター..... 272

29. VSAM データセットの種類と対応する PL/I データセットの構成..... 274

30. VSAM データセットの種類と利点..... 275

31. VSAM データ・セットと使用できるファイル属性..... 278

32. 代替索引パスで実行できる処理..... 278

33. VSAM 入力順データ・セットのロードと入力順データ・セットへのアクセスで使用できるステートメントとオプション..... 283

34. VSAM 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション 286

35. VSAM 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション..... 297

36. PLISRTx (x = A, B, C, または D)へのエントリーポイントと引数..... 327

37. C と PL/I の同等なタイプ..... 341

38. Java 基本タイプと、同等の PL/I ネイティブ・タイプ..... 369

39. 継続可能な例外..... 392

40. 終了例外..... 396

図

1. PAGELENGTH および PAGESIZE.....167

2. 自動プロンプトを使用した出力.....167

3. 自動プロンプトを使用しない出力.....168

4. 固定長レコード..... 209

5. オペレーティング・システムによる DCB への情報の組み込みの方法..... 213

6. 米国標準規格の印刷およびカード穿孔制御文字 (CTLASA)..... 255

7. ソート・プログラムの制御の流れ.....323

8. 入力および出力処理サブルーチンのフローチャート.....333

9. PL/I コンパイラー・ユーザー出口のプロシージャー.....447

はじめに

本書について

本書は、PL/I プログラマーおよびシステム・プログラマーを対象とした 資料です。 PL/I プログラムをコンパイルするための Enterprise PL/I for z/OS® の使用方法を理解するのに役立ちます。 また本書は、プログラム・パフォーマンスを最適化し、エラーに対する処理 を行うのに必要となるオペレーティング・システムの各種機能についても 説明しています。

重要 : 本書では、Enterprise PL/I for z/OS を Enterprise PL/I と呼びます。

Enterprise PL/I for z/OS のランタイム環境

Enterprise PL/I は、ランタイム環境として Language Environment®を使用します。 これは Language Environment 体系に適合し、ランタイム環境を他の Language Environment 適合言語と共用することができます。

Language Environment はランタイム・オプションおよび呼び出し可能サービスの共通セットを提供します。 また、 ILC を呼び出すたびに言語固有の初期化と終了を行う必要がないため、 high-level languages (HLL) とアセンブラの間の interlanguage communication (ILC) も改善される。

資料の使用

Enterprise PL/I に付属の資料は、PL/I を使用してプログラミングを行うときに役立つように設計されています。 Language Environment に付属の資料は、Enterprise PL/I を使用して生成されたアプリケーション用にランタイム環境を管理するときに役立つように設計されています。 それぞれの資料がさまざまな作業に役立ちます。

下の表には、Enterprise PL/I および Language Environment の資料の 使用方法を示しています。 ご使用のコンパイラーとランタイム環境の両方に関する情報を知る必要があります。 これらの資料および関連資料の正式名称および資料番号については、477 ページの『参考文献』を参照してください。

PL/I 情報

表 *1. Enterprise PL/I* 付属資料の使用方法

に..	**使用..**
Enterprise PL/I の評価	Fact Sheet (Data Sheet)
保証情報の理解	Licensed Program Specifications
Enterprise PL/I の計画とインストール	Enterprise PL/I プログラム・ディレクトリー
コンパイラーおよびランタイム変更作業の理解と、プログラムの Enterprise PL/I および Language Environment への適合	コンパイラーおよびランタイム 移行ガイド
プログラムの準備とテスト、およびコンパイラー・オプションに ついての詳細情報の入手	プログラミング・ガイド
PL/I の構文および言語エレメントの仕様についての詳細情報の入手	言語解説書
コンパイラーの問題診断および IBM® への連絡	診断ガイド
コンパイル時メッセージについての詳細情報の入手	メッセージおよびコード

© Copyright IBM Corp. 1999, 2025

Language Environment 情報

表 2. *z/OS* 言語環境プログラム 付属資料の使用方法

に..	使用..
Language Environment の評価	概念
Language Environment の計画	概念 ランタイム・アプリケーション マイグレーション・ガイド
z/OS に Language Environment をインストール	z/OS プログラム・ディレクトリー
z/OS 上で Language Environment をカスタマイズ	カスタマイズ
Language Environment のプログラム・モデルおよび概念の理解	概念 プログラミング・ガイド
Language Environment ランタイム・オプションおよび呼び出し可能サービスの構文の検索	プログラミング・リファレンス
Language Environment で実行されるアプリケーションの開発	プログラミング・ガイドおよび該当する言語の プログラミング・ガイド
Language Environment で実行されるアプリケーションのデバッグ、ランタイム・メッセージに関する詳細情報の入手、Language Environment の問題の診断	デバッグのガイドおよびランタイム・メッセージ
言語間通信 (ILC) アプリケーションの開発	ILC アプリケーションの作成
Language Environment へのアプリケーションの移行	「ランタイム・アプリケーション マイグレーション・ガイド」、および Language Environment で使用できる各言語の移行ガイド

本書で使用されている表記規則

本書では、xx ページの『使用する規則』および xxiii ページの『表記記号の読み方』に示す規則、構文図の書き方、および表記を用いて PL/I および非 PL/I のプログラミング構文を説明しています。

使用する規則

本書における一部のプログラミング構文では、各種エレメントを活字フォントで区別しています。

- 大文字で (UPPERCASE のように) 示した項目は、そのとおりにタイプ する必要がある重要な項目です。
- 小文字で (lowercase のように) 示した項目は、適切な名前 または値に置き換えてタイプする必要があるユーザー提供変数です。 変数は英字で始まり、ハイフン、数字、または下線文字 (_) を 入れることができます。
- 数字 は、数字 (0 から 9 まで) に置き換えなければならないことを示します。
- *DO グループ* は、DO グループに置き換えなければならない ことを示します。
- 下線付きの項目は、デフォルト・オプションです。
- `Examples` は、モノケース・タイプで表示されます。
- 特に指示がなければ、反復可能項目は 1 つ以上のブランクを 使っておのおのを区切ります。

注 : xxiii ページの『表記記号の読み方』の説明にあるような純粋な表記記号ではない記号が示されている場合、それらはすべてプログラミング構文自体の一部です。

これらの規則に従ったプログラミング構文の例については、xxiii ページの『表記例』を参照してください。

構文記法の読み方

本書で使用されている構文図には以下の規則が適用されます。

矢印記号
構文図は、左から右、上から下へと線をたどって読んでください。

►► & boxh; & boxh; & boxh;
ステートメントはここから始まります。

& boxh; & boxh; & boxh; ►
ステートメントの構文は次の行へ続きます。

► & boxh; & boxh; & boxh;
ステートメントは前の行から続いています。

& boxh; & boxh; & boxh; 講習
ステートメントはここで終わります。

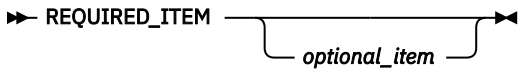
完全なステートメント以外の構文単位の図は、► & boxh; & boxh; & boxh; 記号で始まり、& boxh; & boxh; & boxh; ► 記号で終わります。

規則

- キーワード、許容される同義語、および予約パラメーターは、MVS および OS/2 プラットフォームでは大文字で示され、UNIX プラットフォームでは小文字で示されます。 これらの項目は示されたとおりに入力する必要があります。
- 変数は、小文字のイタリック体で示します (例えば、*column-name*)。 これらはユーザー定義のパラメーターまたはサブオプションを表します。
- コマンドの入力において、パラメーターおよびキーワードを区切る 句読記号がない場合は、少なくとも 1 つのブランクで区切る必要があります。
- 句読記号 (スラッシュ、コンマ、ピリオド、括弧、引用符、等号) と 数字は、示されたとおりに入力する必要があります。
- 脚注は、(1) のように番号を括弧に入れて示します。
- ␣ 記号は 1 つのブランク位置を示します。

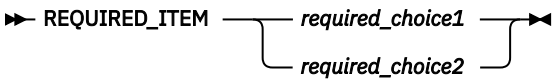
必須項目
必須項目は横線 (メインパス) に示します。

オプション項目
オプション項目は、メインパスの下に示します。

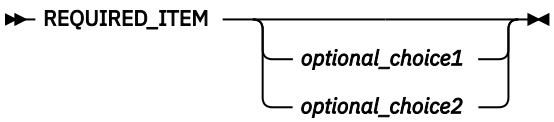


メインパスより上にオプション項目を示すこともあります。これは 読みやすくするためで、ステートメントの実行には影響を及ぼしません。

複数の必須項目またはオプション項目
複数の項目から選択する場合には、それらの項目が縦に重なって、 スタックを形成しています。 複数の項目からいずれか 1 つを選択しなければならない 場合は、 スタック上の項目のうち、1 つがメインパス上に置かれます。

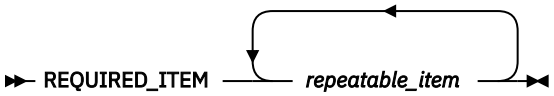


項目がオプショナルである場合、 メインパスの下にある支線上に縦に並んだ項目として示されます。

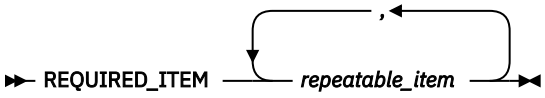


反復可能項目

メインパスの上方を通って左側へ戻る矢印は、項目が反復可能であることを 示します。



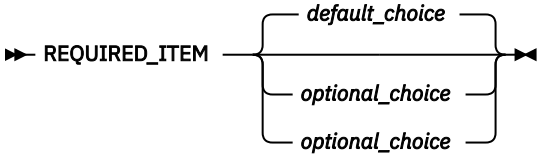
反復矢印の途中にコンマがある場合は、反復される項目をコンマで 区切らなければなりません。



スタックの上の繰り返しを示す矢印は、 スタックから複数の選択項目を指定できることを示しています。

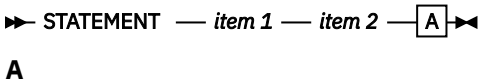
デフォルト・キーワード

IBM 提供のデフォルト・キーワードはメインパスより上に示され、 それ以外の選択項目はメインパスより下に示されます。 構文図の下にあるパラメーター・リストでは、デフォルト選択項目に 下線を付けてあります。



フラグメント

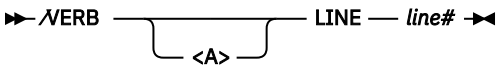
構文図は、フラグメント (部分) に分割する必要がある場合があります。 フラグメントは、文字または フラグメント名で表され、次のように設定されます: | A |。 フラグメントは、メイン・ダイアグラムの終わりの後に続きます。 次の例は、フラグメントの使い方を示したものです。



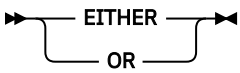
A

置換ブロック

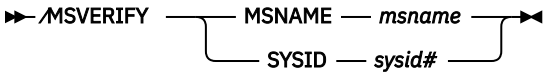
いくつかのパラメーターの集合を <A> のような 置換ブロックで表すことがあります。 例えば、想像上の/VERB コマンドでは、 ,'VERB LINE 1, ,'VERB EITHER LINE 1, または ,'VERB OR LINE 1 を入力することができる。



ここで、<A> は次のようになります。



パラメーターの終わり
数値を持つパラメーターは記号 '#' で終わり、名前であるパラメーターは 'name' で終わり、汎用とすることができるパラメーターは '*' で終わります。



この例の MSNAME キーワードは名前の値をサポートし、SYSID キーワードは数値をサポートします。

表記記号の読み方

本書のプログラミング構文の一部は、表記記号を用いて表されています。 同一構文の他の IBM 資料における記述法に合わせるため、またはテーブルまたは見出し内で 1 行に構文を入れられるようにするためです。

- **中括弧** { } は、選択項目を示します。 項目の 1 つに下線が付いている (デフォルトを示す) 場合、または項目がすべて大括弧で 囲まれている場合を除き、少なくとも 1 つの項目を選択する必要があります。
- 単一の**縦線** | で区切られた項目は、代替項目 です。 単一の縦線で区切られた項目 (または項目のグループ) では、その中の 1 つしか 選択できません。 (2 重縦線 || は、代替項目ではなく、連結演算を指定します。 2 重縦線の詳細については、「*PL/I Language Reference*」 を 参照してください。)
- **大括弧** [ ] に囲まれたものはすべて オプショナルです。 項目が縦に積み重ねられて大括弧で囲まれていれば、1 つの項目しか 指定することはできません。
- **省略符号**, ..., 省略符号の直前にあるタイプの複数の項目が許可されることを示します。

表記例

次の PL/I 構文は、xxiii ページの『表記記号の読み方』の表記記号の実例です。

```
DCL file-reference FILE STREAM
                   {INPUT | OUTPUT [PRINT]}
                   ENVIRONMENT(option ...);
```

この例は、次のように解釈します。

- 最初の行は、*file-reference* を除き、このとおりのスペルで入力しなければなりません。 file-reference は、参照するファイルの名前に置き換えます。
- 2 行目では INPUT または OUTPUT を指定することができますが、 両方を指定することはできません。 OUTPUT を指定した場合は、任意で PRINT も指定できます。 どちらも選択しなかった場合は、デフォルトにより INPUT が 採用されます。
- オプション ...の場合を除き、示されているとおりの最後の行 (括弧とセミコロンを含む) を入力し、スペルを入力する必要があります。 この場合は、1 つ以上のオプションを 1 つ以上のブランクで区切って置換する必要があります。

変更の要約

このセクションでは、バージョン 6 リリース 2 の Enterprise PL/I for z/OS のこの文書に加えられた主な変更点を示します。 本書で解説されている変更には、読者の便宜のため、参照ページが記載されています。 最新の技術変更は、HTML バージョンでは強調表示され、PDF バージョンでは左マージンに縦線 (|) が付けられます。

本リリースにおける機能強化

本リリースでは以下の機能が強化されています。強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- ARCH(15) - ベクトル命令が、インライン・コードによるライブラリ呼び出しの置き換えに使用されるようになり、生成されるコードのパフォーマンスが大幅に向上：
  - HEXENCODE および HEXDECODE 組み込み関数。
  - BASE64ENCODE および BASE64DECODE 組み込み関数。
  - いくつかの BIT 課題
- コンパイラーは、DDD 要素が使用される場合や MMM または MMM が使用される場合を含め、REPATTERN 組み込み関数のより多くのインスタンスに対してインライン・コードを生成する。

使いやすさの向上

- サンプル・ライブラリーのメンバ IBMZLIBD、IBMZLIBP、IBMZLIBS は、 PL/I から ZLIB 圧縮コードを使用する方法を説明しています。
- コンパイラーは、ストレージの問題を引き起こす可能性のあるコードにフラグを立てるようになった。特に、ある変数のアドレスに割り当てられたポインタを、より大きなサイズの変数のベースとして使用するコードを検出し、フラグを立てる。

コンパイラー・オプション機能拡張

変更されたコンパイラー・オプション:

- USAGE オプションの新しい BASE64 サブオプションでは、 base-64 「アルファベット」の最後の 2 文字、標準の「+」と「/」、または URL 「-」と「」のいずれかを選択できる。
- USAGE オプションの新しい HEXDECODE サブオプションは、HEXDECODE 組み込み関数に対して、（妥当性チェックを犠牲にして）はるかに優れたパフォーマンスを求めることができます。
- RULES オプションの新しい NOPUT サブオプションは、プロダクション・コードから不要な PUT 文を排除することを容易にします。
- XINFO オプションの新しい DCLS サブオプションは、すべての外部ルーチンの宣言を含むファイルを生成し、厄介なミスマッチ・インターフェースの問題をキャッチするために、既存の RULES(NOLAXINTERFACE)オプションの使用をより簡単に開始できるようにする。
- 既存の CASE (ASIS) オプションが拡張され、XINFO(SYN) および XINFO(SYM) オプションで生成されるデータも含まれるようになった。

からの強化 V6R1

パフォーマンス向上

- コンパイラーは FIXED DECIMAL 変数と PICTURE 変数の一般的な使用法について、より適切なたコードを生成します。
- このコンパイラーは、いくつかの新しいハードウェア命令を使用します。
- このコンパイラーは、一部のユーザー SELECT ステートメントを内部的に再フォーマットすることにより、コンパイル時と実行時の両方のパフォーマンスを向上させます。
- このコンパイラーは、ORDINALNAME、ORDINALPRED、ORDINALSUCC 組み込み関数用のより優れたコードを生成します。
- このコンパイラーは、INLIST および INARRAY 組み込み関数用のより優れたコードを生成します。

使いやすさの向上

- このコンパイラーは、アプリケーション全体を 64 ビットに変換せずに、拡張 PL/I プログラム・データ処理用アドレッシング・スペースを提供することにより、31 ビット(AMODE 31) PL/I プログラムを拡張して 64 ビット(AMODE 64) PL/I プログラムと連動させることができます。 この新機能により、31 ビット・コードは 64 ビット・コードを呼び出すことができます。 373 ページの『第 19 章 AMODE 31 と AMODE 64 コードを混ぜる』を参照してください。

- このコンパイラーは、より多くの可能性のある STRINGSIZE 条件にフラグを立てます。
- このコンパイラーは、RULES (NOGOTO)の違反として EXEC HANDLE CONDITION および EXEC SQL WHENEVER にフラグを立てます。
- このコンパイラーは、INONLY を INOUT に、INONLY を OUTONLY に、NONASGN を ASGN に渡すように、フラグを立てます。
- このコンパイラーは、IBMZIOP のオプション・ストリングで+ DD:をサポートします。
- OPTIONS (ASSEMBLER)の場合、このコンパイラーはリスクのある INONLY パラメーターにフラグを立てません。
- このコンパイラーは、ランタイム・エラーを回避するために、REPATTERN に新しいコンパイラー・メッセージを発行します。
- このコンパイラーは、コンパイル時に、リスクのあるコードと低性能のコードの両方にフラグを追加します。

コンパイラー・オプション機能拡張

新しいコンパイラー・オプション:

- この 40 ページの『HGPR』コンパイラー・オプションは、コンパイラーが 64 ビット GPR の高位半分を保持、復元するかどうかを指定します。
- この 56 ページの『MAXRUNONIF』コンパイラー・オプションは、おそらく INLIST 組み込み関数を使って書き直さなければならないコードにフラグを立てます。
- 56 ページの『MAXSTATIC』 コンパイラー・オプションは、指定したサイズを超える STATIC 変数にフラグを立てるようにします。

変更されたコンパイラー・オプション:

- この 9 ページの『ARCH』コンパイラー・オプションは、ARCH（14）をサポートして、z/ Architecture モードの 3931-xxx（IBM z16）モデルで使用可能な命令を使用するコードを生成します。
- この 11 ページの『BIFPREC』コンパイラー・オプションのデフォルトが BIFPREC (31) になりました。
- 20 ページの『10 進数』コンパイラー・オプションの CHECKFLOAT サブオプションでは、FLOAT から FIXED DECIMAL への変換のソースが大きすぎる場合に SIZE 条件を上げるコードをコンパイラーが生成する必要があります。
- 20 ページの『10 進数』コンパイラオプションの FOFLONADD および NOFOFLONADD サブオプションでは、コンパイラが FIXEDOVERFLOW 条件を発生させるコードを生成するか、ADD または SUBTRACT 組み込み関数を使用するために切り捨てられた結果を生成する必要があります。
- 20 ページの『10 進数』コンパイラー・オプションの KEEPMINUS サブオプションは、特定の状況下で、コンパイラーが FIXED DECIMAL 割り当てで「負のゼロ」を保持するコードを生成することを請求します。
- 35 ページの『FILEREF』 コンパイラー・オプションの HASH サブオプションは、ファイル参照表の各行の後に、そのファイルの 256 ビット SHA-2 ハッシュの 16 進数値を含む行が続くように指定する。
- 40 ページの『HEADER』 コンパイラー・オプションには、さらに 2 つのサブオプションがある：ALL と FIRST である。 ALL サブオプションはデフォルト設定である。
- INCPDS コンパイラー・オプションが複数指定をサポートするようになり、INCLUDE ファイルが見つかる複数の PDS(E)を指定できるようになりました。
- 45 ページの『JSON』 コンパイラー・オプションの ENCODING サブオプションは、JSONPUT および JSONGET 関数の JSON テキストのコード・ページを指定します。
- 45 ページの『JSON』 コンパイラー・オプションの PARSE(V2) サブオプションを使用すると、ターゲット構造の要素が JSON ソース内で異なる順序で現れる JSON テキストをパースできます。 また、JSON ソースのエラーに対するパーサーの反応をよりコントロールできるようになる。
- 45 ページの『JSON』コンパイラー・オプションの TRIMR と NOTRIMR サブオプションは、JSONPUTVALUE と JSONPUTMEMBER 関数からの出力が右からトリミングされるかどうかを決定します。

- 70 ページの『RULES』 コンパイラー・オプションの IBM サブオプションと ANS サブオプションの違いが縮小され（特に、UNSIGNED の違いはなくなりました）、残りの違いの説明が拡大されました。
- 70 ページの『RULES』 コンパイラー・オプションの UNREFVALUE と NOUNREFVALUE サブオプションは、コンパイラーが参照されない VALUE 定数にフラグを立てるかどうかを制御します。
- 89 ページの『SYSPARM』コンパイラー・オプションの最大長が 1023 になりました。
- 91 ページの『TEST』 コンパイラー・オプションの SOURCE サブオプションは、GOFF オプションも指定されている場合、コンパイラーは、TEST(SEPARATE) オプションで使用されたのと同じ圧縮形式のすべてのソース・ファイルを、生成されたオブジェクト・ファイルに無負荷クラスとして含めることを指定します。
- 94 ページの『USAGE』 コンパイラー・オプションの UUID サブオプションは、UUID と UUID4 組み込み関数の出力が小文字か大文字かを決定します。
- 94 ページの『USAGE』コンパイラー・オプションの VALIDDATE サブオプションは、1582 の日付が有効かどうかをコンパイラーが認識すべきかどうかを決定します。そうでない場合、コンパイラーは、YYYY を使用して、多くのパターンに対して VALIDDATE 組み込み関数のより適切で高速なインラインコードを生成できます。
- XINFO(XML) コンパイラ・オプションの HASH サブオプションは、FILENAME 行ごとに、生成される XML ファイルの後に、そのファイルの 256 ビット SHA-2 ハッシュの 16 進数値を <CONTENTHASH> と </CONTENTHASH> タグで囲んだ行が続くように指定します。

新規マクロ・プリプロセッサー・オプション

- この 120 ページの『EOLCOMM (E)』オプションは、マクロプリプロセッサが//マクロ呼び出しの一部であるテキストであっても行末コメントマーカーとして認識する必要があることを指定します。

V5R3 の機能拡張

本リリースでは以下の機能が強化されています。強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

使いやすさの向上

- コンパイラーは、=+ および =- が現れる位置に、タイプミスの可能性が高いとしてフラグを立てます。
- コンパイラーは、リストの 1 ページの最大行数として 65535 行をサポートします。
- コンパイラーは、EXIT に渡されるストリングの最大長を 1023 まで増やしました。 35 ページの『出口』を参照してください。
- MAIN のパラメーターが (CHAR(*) VAR と指定された不確定の長さではなく) 特定の長さの CHAR VARYING と宣言されていると、コンパイラーは警告メッセージを出します。
- コンパイラーは、HANDLE、OFFSET、ENTRY、および AREA 変数への ('' または ''b と指定された) ヌル・ストリングの割り当ての使用をサポートします。
- コンパイラーは、POINTER に使用される NULLINIT 値の制御をサポートします。

コンパイラー・オプション機能拡張

新しいコンパイラー・オプション:

- 54 ページの『MAXINIT』 コンパイラー・オプションは、生成されるオブジェクトのサイズが著しく大きくなるコードを見つけるのに役立ちます。
- MACRO プリプロセッサーは、INCLUDE プリプロセッサーと同じ 120 ページの『ID』 オプションをサポートします。
- MACRO プリプロセッサーの 121 ページの『IGNORE』 オプションは、MACRO プリプロセッサーが特定のステートメントを無視するかどうかを指定します。
- SQL プリプロセッサーの 128 ページの『LINEFILE』 オプションを使用すると、ステートメント番号にファイル番号と行番号を含めることができます。

変更されたコンパイラー・オプション:

- 23 ページの『DEFAULT』 コンパイラー・オプションには、以下の新しいサブオプションがあります。

  NULLINIT (SYSNULL | NULL)
  DEFAULT コンパイラー・オプションの NULLINIT サブオプションは、NULLINIT 属性を持つ POINTER に割り当てられた値を判別するのに役立ちます。

  LAXQUAL | NOLAXQUAL

  DEFAULT (NOLAXQUAL) オプションは、コンパイラーがドット修飾なしで名前への参照を解決するときに、最初に宣言されたブロック内で名前があいまいであるが、親ブロック内のスカラーへの参照である場合に、コンパイラーがその親ブロック参照になるように参照を解決することを指定します。
- 23 ページの『DEFAULT』 コンパイラー・オプションの NOLAXINOUT サブオプションは、2 組のサブオプションをサポートします。
- 9 ページの『ARCH』 コンパイラー・オプションは、 z/Architecture® モードの 8561-xxx (IBM z15™) モデルで使用可能な命令を使用するコードを生成するために ARCH (13) をサポートします。
- 20 ページの『10 進数』 コンパイラー・オプションの FOFLONDIV および NOFOFLONDIV サブオプションは、DIVIDE 組み込み関数が使用されている場合は FIXEDOVERFLOW 条件を発生させるコードか、切り捨てた結果を生成するコードを生成するようコンパイラーに要求します。
- 23 ページの『DEFAULT』 コンパイラー・オプションの PADDING および NOPADDING サブオプションは、定義された構造体にコンパイラーが埋め込みを行うかどうかを指定します。
- 94 ページの『USAGE』 コンパイラー・オプションには、以下の新しいサブオプションがあります。

  REGEX(RESET | NORESET)
  USAGE コンパイラー・オプションの REGEX サブオプションを使用すると、アプリケーション内の他のどの関数が C ロケールに依存しているかが分かっている場合に、反復 REGEX 呼び出しのパフォーマンスを向上させることを選択できます。
- 70 ページの『RULES』 コンパイラー・オプションには、以下の新しいサブオプションがあります。

  COMPLEX | NOCOMPLEX
  RULES コンパイラー・オプションの NOCOMPLEX サブオプションを使用すると、COMPLEX 属性、組み込み関数、または定数の使用に対してフラグを立てるという要求を強制できます。

  GLOBAL | NOGLOBAL
  RULES コンパイラー・オプションの NOGLOBAL サブオプションを使用すると、ブロックのネスト・サブプロシージャーで使用されていて、そのブロックで AUTOMATIC、CONTROLLED、DEFINED、PARAMETER、および ASSIGNABLE STATIC と宣言されている変数にフラグを立てることができます。

  LAXEXPORTS | NOLAXEXPORTS
  RULES コンパイラー・オプションの NOLAXEXPORTS サブオプションを使用すると、どの PACKAGE もそれがエクスポートするすべてのルーチンを明示的に指定するという要求を強制できます。

  LAXFIELDS | NOLAXFIELDS
  RULES コンパイラー・オプションの NOLAXFIELDS サブオプションを使用すると、SQL SELECT または INSERT 内でフィールドを明示的に指定する必要があるという要求を強制できます。

  LAXOPTIONAL | NOLAXOPTIONAL
  RULES コンパイラー・オプションの NOLAXOPTIONAL サブオプションを使用すると、パラメーターが OPTIONAL と宣言されていない OMITTED および PRESENT 組み込み関数の使用に対してコンパイラーがフラグを立てるようになります。

  LAXPACKAGE | NOLAXPACKAGE
  RULES コンパイラー・オプションの NOLAXPACKAGE サブオプションを使用すると、すべてのコンパイル単位に明示的な PACKAGE ステートメントが含まれるという要求を強制できます。

  LAXPARMS | NOLAXPARMS
  RULES コンパイラー・オプションの NOLAXPARMS サブオプションを使用すると、すべてのパラメーターで INOUT/INONLY/OUTONLY 属性のいずれかを指定するという要求を強制できます。

  LAXSCALE | NOLAXSCALE
  RULES コンパイラー・オプションの NOLAXSCALE サブオプションを使用すると、スケール係数がゼロ以外の FIXED BIN を不許可にし、そのフラグ設定を 1 次ソース・ファイルのみに限定することができます。

UNREFSTATIC | NOUNREFSTATIC
RULES コンパイラー・オプションの NOUNREFSTATIC サブオプションは、PLIXOPT および PLITABS にフラグを立てなくなります。

PADDING | NOPADDING
RULES(NOPADDING) コンパイラー・オプションの LOOSE/STRICT サブオプションを使用すると、前後に埋め込みのある構造体を見つけることもできます。

- 「CEE」、「DFH」、「DSN」、「EYU」、「IBM」、「PLI」、および「SQL」で始まる名前の参照は、RULES (NOLAXENTRY) および RULES (NOLAXQUAL) 検査から除外されます。
- 45 ページの『JSON』 コンパイラー・オプションには、以下の新しい変更があります。

CASE(LOWER)
JSON(CASE) コンパイラー・オプションの LOWER サブオプションは、JSONPUT 組み込み関数によって生成されて JSONGET 組み込み関数によって予期される JSON テキスト内の名前はすべて小文字になることを指定します。

GET(HEEDCASE | IGNORECASE)
JSON(GET) コンパイラー・オプションの HEEDCASE サブオプションは、JSONGET 組み込み関数は完全一致の名前を必要とすることを指定します。 JSON(GET) コンパイラー・オプションの IGNORECASE サブオプションは、JSONGET 組み込み関数は大/小文字を無視して名前を一致と見なすことを指定します。

- LIMITS(STRING) オプションの 64K サブオプションを使用すると、ストリングの対応する制限を 65535 に設定できます。 48 ページの『LIMITS』を参照してください。
- USAGE(HEX) コンパイラー・オプションでは、省略形の CSTG および STG は CURRENTSIZE および SIZE の同義語として受け入れられます。 94 ページの『USAGE』を参照してください。
- MACRO プリプロセッサーの 119 ページの『CASE』 オプションには、デフォルトとして CASE(ASIS) オプションがあります。

SQL 機能拡張

- SQL プリプロセッサーは、DEFINE STRUCTURE ステートメントのエレメントを使用する参照をサポートします。 132 ページの『ホスト変数の宣言』を参照してください。
- SQL プリプロセッサーは、スカラー・ホスト変数で ORDINAL 属性をサポートします。 133 ページの『スカラー・ホスト変数の宣言』を参照してください。

CICS 機能強化

- CICS プリプロセッサー出力に、そのプリプロセッサーの稼働中に有効な CICS オプションすべてのリストが組み込まれるようになりました。

V5R2 からの機能拡張

本リリースでは以下の機能が強化されています。強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- 最初の引数が 1 < = *N* < = 4 の CHAR (*N*) で、他のすべての引数が長さ < = *N* の CHAR である場合に、INLIST のコードが改善されました。
- SELECT(*x*) の *x* が CHAR である場合に、そのコードが強化されました。 これに該当する場合、コンパイラーは二分探索によって SELECT を実行します。

使いやすさの向上

- コンパイラーは、到達不能な ASSERT UNREACHABLE ステートメントに、他の到達不能なステートメントにフラグを立てるときとは異なるメッセージでフラグを立てます。

- コンパイラーは、より速い MEMCU12 または MEMCU21 に変換できる可能性がある、MEMCONVERT 組み込み参照にフラグを立てます。
- コンパイラーは、REPATTERN に変換しなければならない可能性がある、TRANSLATE 組み込み参照にフラグを立てます。
- コンパイラーは、他の構造のメンバーである AGGREGATE リスト・タイプの構造において、その機能が強化されました。
- コンパイラーは、PROCEDURE ブロックまたは BEGIN ブロックの INLINE をオフにするシチュエーションの多くで、通知メッセージを出します。
- コンパイラーは、ネストされた PROCEDURE および BEGIN ブロックで、INLINE オプションを受け入れます。
- 属性リストに、長さ 256 以下の定数 CHARACTER および BIT における VALUE 属性、また数値 PICTURE 定数における VALUE 属性の内容が表示されます。

コンパイラー・オプション機能拡張

- 10 ページの『ASSERT』 コンパイラー・オプションは、ASSERT ステートメントが、ASSERTION 条件を発生させるデフォルト・ライブラリー・ルーチンを呼び出すか、またはユーザーによって指定されたルーチンを呼び出すかを制御します。
- 13 ページの『CASE』 コンパイラー・オプションは、一部の名前が大文字で表示されるか、またはソース・プログラムでの表示と同じフォーマットで表示されるかを制御します。
- 14 ページの『CMPAT』コンパイラー・オプションでサブオプション CMPAT(LE) および CMPAT(V1) が復活しました。
- 19 ページの『DBRMLIB』コンパイラー・オプションは、SQL プリプロセッサーが UNIX システム・サービスで動作しているときにその SQL プリプロセッサーの DBRM データ・セットを作成する場所を制御します。
- 70 ページの『RULES』 コンパイラー・オプションには、以下のサブオプションがあります。

  LAXCONV | NOLAXCONV
  RULES コンパイラー・オプションの NOLAXCONV サブオプションを使用すると、オペランドに算術型がない演算式にフラグを立てることができます。

  LAXINTERFACE | NOLAXINTERFACE
  RULES コンパイラー・オプションの NOLAXINTERFACE サブオプションを使用すると、外部 PROCEDURE のそれぞれに有効な明示宣言を含まないコードにフラグを立てることができます。

  MULTIENTRY | NOMULTIENTRY
  RULES コンパイラー・オプションの NOMULTIENTRY サブオプションを使用すると、ENTRY ステートメントを含むコードにフラグを立てることができます。

  MULTIEXIT | NOMULTIEXIT
  RULES コンパイラー・オプションの NOMULTIEXIT サブオプションを使用すると、複数の RETURN ステートメントを含むコードにフラグを立てることができます。

  MULTISEMI | NOMULTISEMI
  RULES コンパイラー・オプションの NOMULTISEMI サブオプションを使用すると、複数のセミコロンを含むソース行にフラグを立てることができます。

  UNREFCTL | NOUNREFCTL
  RULES コンパイラー・オプションの NOUNREFCTL サブオプションを使用すると、非参照 CTL 変数にフラグを立てることができます。

  UNREFDEFINED | NOUNREFDEFINED
  RULES コンパイラー・オプションの NOUNREFDEFINED サブオプションを使用すると、非参照 DEFINED 変数にフラグを立てることができます。

  UNREFENTRY | NOUNREFENTRY
  RULES コンパイラー・オプションの NOUNREFENTRY サブオプションを使用すると、非参照 ENTRY 定数にフラグを立てることができます。

UNREFFILE | NOUNREFFILE
RULES コンパイラー・オプションの NOUNREFFILE サブオプションを使用すると、非参照 FILE 定数にフラグを立てることができます。

UNREFSTATIC | NOUNREFSTATIC
RULES コンパイラー・オプションの NOUNREFSTATIC サブオプションを使用すると、非参照 STATIC 変数にフラグを立てることができます。

YY | NOYY
RULES コンパイラー・オプションの NOYY サブオプションを使用すると、2 桁の年の使用に対してフラグを立てることができます。

- 70 ページの『RULES』 コンパイラー・オプションの NOELSEIF サブオプションは、単純な DO-END で囲まれた IF ステートメントの直後に続く ELSE ステートメントにフラグを立てます。
- 70 ページの『RULES』 コンパイラー・オプションの NOGOTO オプションは、LOOSEFORWARD サブオプションを受け入れるため、前方に分岐する GOTO ステートメントには NOGOTO フラグ設定が適用されません。

CICS 機能強化

- CICS プリプロセッサー出力に、そのプリプロセッサーの稼働中に有効な CICS オプションすべてのリストが組み込まれるようになりました。

V5R1 からの機能拡張

本リリースでは以下の機能が強化されています。強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- Enterprise PL/I は、ベクトル機構をさらに活用するようになりました。
- 桁数の多い固定小数点の除算のいくつかが、10 進浮動小数点 (DFP) 機能を使用して行われるようになりました。 これにより、一部の ZERODIVIDE 例外は INVALIDOP として報告される可能性があります。

使いやすさの向上

- Enterprise PL/I は、64 ビット・アプリケーションのコードをコンパイルするためのサポートを提供するようになりました。 詳しくは、 181 ページの『第 7 章 64 ビット・アプリケーションを開発する場合の考慮事項』を参照してください。
- コンパイラーは、IBMZIOP におけるオプション・ストリング同士が相互に同じ場合、および指定変更できないいずれかのオプションと矛盾するオプションをユーザーが指定した場合に、警告メッセージを発行するようになりました。
- コンパイラーは、コンパイル時に使用するコンパイラー・オプションに関する情報も、生成する SYSADATA ファイルに組み込むようになりました。 SYSADATA レコードの宣言はすべて、サンプル・データ・セット SIBMZSAM において組み込みファイルで提供されるようになりました。

コンパイラー・オプション機能拡張

- 新しい 12 ページの『BRACKETS』 コンパイラー・オプションを使用すれば、SQL プリプロセッサーが SQL 配列参照において左右の括弧として受け入れる記号を指定できます。 このオプションを使用すれば、そのような言語をさらに利用しやすくなります。
- コンパイラーは、新しい 23 ページの『DECOMP』 コンパイラー・オプションを使用してリスト・セクションを生成します。 このリスト・セクションには、ソース・プログラムで使用されるすべての式に対するすべての中間式とその属性が表示されます。
- 20 ページの『10 進数』 コンパイラー・オプションには、新しいサブオプション TRUNCFLOAT があります。 このサブオプションを使用すれば、切り捨てが行われる可能性があるときに固定小数点への浮動小数点の代入をコンパイラーが処理する方法を制御できます。

- 新しい 35 ページの『EXPORTALL』 コンパイラー・オプションは、外部で定義されたすべてのプロシージャーと変数を、DLL アプリケーションで使用できるようにエクスポートするかどうかを制御します。
- 新しい 40 ページの『HEADER』 コンパイラー・オプションを使用すれば、コンパイラー・リスト内の各ヘッダー行の中央に表示されるものを制御できます。
- 44 ページの『INSOURCE』 コンパイラー・オプションには、FIRST および ALL という 2 つの新しいサブオプションがあります。 これらは、リスト・ファイルに表示されるソース・リストの数を制御します。
- 新しい 51 ページの『LP』 コンパイラー・オプションを使用すれば、コンパイラーが 31 ビット・コードを生成するのか 64 ビット・コードを生成するのかを指定できます。
- 57 ページの『MDECK』 コンパイラー・オプションには、AFTERALL および AFTERMACRO という 2 つの新しいサブオプションがあります。 これらは、MDECK が生成されるタイミングを制御します。
- 新しい 59 ページの『NULLDATE』 コンパイラー・オプションを使用すれば、ご使用のプログラムにおいて SQL ヌル日付を有効な日付として使用できます。
- 新しい 60 ページの『OFFSETSIZE』 コンパイラー・オプションは、64 ビット・アプリケーションにおける OFFSET 変数のサイズを判別します。
- 70 ページの『RULES』 コンパイラー・オプションには、以下の新しいサブオプションがあります。

  LAXSTMT | NOLAXSTMT
  RULES(NOLAXSTMT) を使用して、複数のステートメントを持つすべての行にフラグを立てることができます。

  UNREFBASED | NOUNREFBASED
  RULES(NOUNREFBASED) を使用して、BASED ストレージにある非参照 BASED 変数にフラグを立てることができます。
- 70 ページの『RULES』 コンパイラー・オプションの NOPROCENDONLY サブオプションには、ALL および SOURCE という 2 つの新しいサブオプションがあります。 これらのサブオプションを使用すれば、各プロシージャーの END ステートメントにそのプロシージャーの名前が組み込まれていなければならないという規則をさらに実施しやすくなります。
- 102 ページの『XREF』 コンパイラー・オプションには、EXPLICIT および IMPLICIT という 2 つの新しいサブオプションがあります。 これらのサブオプションを使用すれば、暗黙的変数参照が XREF リストに組み込まれているのかどうかを判別できます。

注 :

あるリリースのすべてのコンパイラー・オプションのデフォルト設定は、 SIBMZSAM データセットの IBMXO*vrm*, where *v* がバージョン番号、 *r* がリリース番号、 *m* が修正番号で、通常は 0 です（たとえば、V5R3 は IBMXO530 、 V5R1 は IBMXO510 ）。 このデータセットには、サポートされているすべての PL/I リリースの IBMXO*vrm* メンバが含まれている。 リリースごとにデフォルト設定のどこが変更され、どのような新しいオプションが追加されたかを確認するには、2 つのリリースの IBMXO*vrm* 。 また、 IBMXO*vrm* ファイルをテンプレートとして使用し、お好みの設定で独自のオプションファイルを作成することもできます。

SQL 機能拡張

- SQL プリプロセッサーは、DEFINE ALIAS ステートメント、DEFINE ORDINAL ステートメント、および DEFINE STRUCTURE ステートメントを構文解析するようになりました。 SQL ステートメントで使用できる PL/I タイプが DEFINE ALIAS ステートメントで定義される場合、そのタイプで宣言された変数も SQL ステートメントで使用できます。
- 59 ページの『NULLDATE』 コンパイラー・オプションが指定されている場合、*year*、*month*、および *day* がすべて 1 の値になっている SQL ヌル日付は、一部の日次処理組み込み関数で、有効な日付として受け入れられます。
- INDFOR 属性を使用すれば、別の PL/I 構造体と一致するように標識変数の構造体をより宣言しやすくなります。
- DSNH030 SQL プリプロセッサー・メッセージの重大度をより変更しやすくするために、固有のプリプロセッサー・メッセージ IBM3317 が指定されるようになりました。

V4R5 からの機能拡張

本リリースは、本書および他の IBM PL/I ブックに記載されている次の機能強化を提供します。

パフォーマンス向上

- MVC が 1 つ除去されたため、EXEC CICS ステートメント用に生成されたコードの実行速度が向上しました。
- FIXED DEC の MOD および REM に関して 15 を超える精度で非常に高速なコードが生成されるようになりました。
- ARCH オプションで最大値として 11 が受け入れられるようになりました。 ARCH(11) が指定されると、コンパイラーは、対象の z システム上で新規ハードウェア命令を使用するコードを生成します。 このコードにより、特に、SEARCH 組み込み関数および VERIFY 組み込み関数の一部インスタンスのパフォーマンスが向上します。

SQL 機能拡張

- 最初の無効ホスト変数が見つかった時点では EXEC SQL ステートメントの妥当性検査は停止せず、代わりにすべてのホスト変数参照が検査されます。
- 新規 SQL プリプロセッサー・オプション (NO)CODEPAGE により、SQL プリプロセッサーが コンパイラー CODEPAGE オプションをどのように指定するのかが決まります。
- 新規 SQL プリプロセッサー・オプション (NO)WARNDECP により、SQL プリプロセッサーによって生成される「ノイズ」の量を減らすことができます。
- ホスト変数として使用される構造体は、標識変数として使用される構造体も持つようになりました。
- 名前付き定数 (すなわち、VALUE 属性を持つ PL/I 変数) をホスト変数として使用できるようになりました (その設定の定数が SQL で許可される場合)。

使いやすさの向上

- 新規 MAXBRANCH コンパイラー・オプションにより、過度に複雑なコードを見つけやすくなりました。
- 新規コンパイラー・オプション FILEREF および NOFILEREF は、コンパイラーがファイル参照テーブルを生成するのかどうかを制御します。
- 新規 JSON コンパイラー・オプションにより、JSONPUT 組み込み関数によって 生成されたり JSONGET 組み込み関数によって期待されたりする JSON テキスト内の名前の大/小文字を選択できるようになりました。
- LIMITS コンパイラー・オプションで、BIT 変数、CHARACTER 変数、または WIDECHAR 変数の長さのしきい値として値 32K、512K、8M、および 128M を持つ サブオプションとして STRING が受け入れられるようになりました。
- XML コンパイラー・オプションにより、XMLCHAR 組み込み関数によって生成される XML 属性をアポストロフィで囲むのか引用符で囲むのかを 選択できるようになりました。
- コードがプロシージャーにおいて RETURNS 属性を伴って END ステートメントに到達した場合に エラーになるように RULES(NOLAXRETURN) コンパイラー・オプションが拡張されました。
- コンパイラー・オプション RULES(NOLAXNESTED) および RULES(NOPADDING) に対する新規サブオプション ALL | SOURCE は、疑わしいコーディングにコンパイラーがフラグを立てるタイミングをより詳細に制御します。
- ユーザーは、新規 FORCE(NOLAXQUAL) 属性と、RULES(NOLAXQUAL) オプションの新規 FORCE サブオプションを使用すれば、 構造体ごとに NOLAXQUAL 規則を強制できます。
- INITIAL 属性を持つ REFER オブジェクトに E レベル・メッセージのフラグが立てられるようになりました。
- マクロ・プリプロセッサーが DEPRECATE オプションと DEPRECATENEXT オプションをサポートするようになりました。 これらのオプションを使用すれば、計画した段階にわたって、選択したマクロ・プロシージャーを除去できます。

- FIXED 変数や FLOAT 変数に VALUE 属性がある場合、コンパイラーはその値を ATTRIBUTES リストにおいて文字ストリングとしてリストします。
- 配列メンバーと配列メンバーの間に埋め込みが含まれる (構造体に CHAR(31) VARYING ALIGNED の配列が含まれるときなど) VARYING の配列に マークを付けるために集合リストが拡張されました。
- SUBSCRIPTRANGE が使用可能になっていて、複数の非定数境界を持つ配列が配列の割り当てに含まれる場合に、 その境界同士が一致しなければ SUBSCRIPTRANGE 条件を発生させるコードをコンパイラーが生成するようになりました。 境界がすべて定数の場合、コンパイラーは、その境界同士が一致することをコンパイル時に検査し続けます。
- いずれかの PLISAX イベント構造体におけるイベントが (NULLENTRY 組み込み関数または UNSPEC 疑似変数を使用して) ヌルに設定されている 場合、XML パーサーはそのイベントを呼び出しません。 これにより、関心のあるイベントのみに XML 構文解析コードを制限することができ、 同時に構文解析全体のパフォーマンスを向上させることができます。
- コンパイラーは、現在コンパイル中のプロシージャーに対して FETCH や RELEASE を実行する試みにフラグを立てるようになりました。

V4R4 からの機能強化

本リリースは、本書および他の IBM PL/I ブックに記載されている次の機能強化を提供します。

パフォーマンス向上

- 疑似アセンブラー・リストの生成が、前のリリースよりも早くなりました。
- コンパイラーは、ARCH(7) でコンパイルされるようになりました。
- コンパイラーは、STATIC 属性を持たず、100 個を超える INITIAL 項目を持つ変数の宣言にフラグを立てるようになりました。

使いやすさの向上

- プログラムに間違った構文が含まれているときに SQL プリプロセッサーから発行されるメッセージが増加しました。 これらのメッセージにより、エラーがあるソース・ステートメントを識別しやすくなります。
- 新しい NOINCLUDE オプションを使用して、マクロ・プリプロセッサーの外部での %INCLUDE ステートメントおよび %XINCLUDE ステートメントの使用を禁止することができます。
- %INCLUDE ステートメントが最後のコンパイラー・パスによって処理されるとき、組み込みファイルがソース・リスト内で大括弧に囲まれるようになりました。
- DEFAULT オプションの NULLSTRPTR サブオプションが持つ新しい STRICT サブオプションによって、POINTERS への '' の割り当ておよび比較に対して無効のフラグが立てられます。
- STMT オプションを指定すると、ソース・リストおよびメッセージ・リストには論理ステートメント番号とソース・ファイル番号の両方が入ります。

V4R3 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- ARCH(10) オプションを使用すると、IBM zEnterprise® EC12 System 命令を活用できます。
- VERIFY および SEARCH 用のインライン・コードがコンパイラーによって生成されるようになりました (VERIFY および SEARCH の引数が 3 つあって、その 2 番目の引数が 1 文字の場合)。
- PICTURE から DFP への追加変換用のインライン・コードがコンパイラーによって生成されるようになりました。

- BIT から CHAR への追加変換用のインライン・コードがコンパイラーによって生成されるようになりました。
- BIT から WIDECHAR への変換用のインライン・コードがコンパイラーによって生成されるようになりました。
- FIXED DEC の TRIM 用に生成されるコードが向上しました。

使いやすさの向上

- SQL プリプロセッサーにおいて以下の点が向上しました。
  - ONEPASS オプションがサポートされます。
  - ホスト変数宣言で一部の制限付き式を使用できます。
  - EXEC SQL ステートメントのリストが、元のソースによく似た読みやすいフォーマットで表示されます。
  - LIKE 属性を使用して宣言されたホスト変数を使用できます。
  - 新規 DEPRECATE オプションがサポートされます。このオプションを使用すると、プリプロセッサーが非推奨のステートメントのリストにフラグを立てます。
- ADATA ファイルに、以下の属性、組み込み関数、およびステートメントの使用が記録されるようになりました。
  - INONLY、INOUT、および OUTONLY 属性
  - XMLATTR および XMLOMIT 属性
  - ALLCOMPARE、UTF8、UTF8TOCHAR、および UTF8TOWCHAR 組み込み関数
  - ASSERT ステートメント
- 新規 CASERULES オプションを使用すれば、PL/I キーワードに対して大/小文字の規則を指定できます。例えば、すべてのキーワードは大文字でなければならないという規則を指定できます。
- DEPRECATE オプションで新規 STMT サブオプションを使用できます。 このサブオプションを使用すると、コンパイラーが非推奨のステートメントのリストにフラグを立てます。
- 新規 DEPRECATENEXT オプションを使用すれば、関数を段階的に非推奨にできます。
- IGNORE オプションで新規 ASSERT サブオプションを使用できます。 このサブオプションを使用すると、コンパイラーがすべての ASSERT ステートメントを無視します。
- 新規 (NO)MSGSUMMARY オプションは、コンパイル中に発行されたすべてのメッセージの要約をコンパイラーがリストに追加するかどうかを制御します。
- RTCHECK オプションで新規 NULL370 サブオプションを使用できます。 このサブオプションは、旧 NULL() 値に相当するポインターが逆参照されるかどうかを検査します。 旧 NULL() 値に相当するポインターとは、16 進値 'FF000000'x を持つポインターのことです。
- RULES オプションで、CONTROLLED 属性の使用に対してフラグを立てるかどうかを制御する (NO)CONTROLLED がサブオプションとして受け入れられるようになりました。
- RULES オプションで、実行可能コードのセクション間にネスト・プロシージャーが存在するプログラムにフラグを立てるかどうかを制御する (NO)LAXNESTED がサブオプションとして受け入れられるようになりました。
- RULES オプションで、RECURSIVE 属性の使用、または自分自身を直接呼び出すプロシージャーの使用に対してフラグを立てるかどうかを決定する (NO)RECURSIVE がサブオプションとして受け入れられるようになりました。
- RULES(NOUNREF) オプションで、すべての未参照変数にフラグを立てるかどうかを決定する SOURCE | ALL がサブオプションとして受け入れられるようになりました。
- RULES(NOLAXIF) では、x = y = z の形式の割り当てに対してコンパイラーがフラグを立てるようにもなりました。
- RULES (NOLAXSCALE) により、コンパイラーは p < 0 のときに ROUND (x, p) にフラグを立てるようになりました。

V4R2 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- ARCH(9) オプションを使用すると、IBM zEnterprise 196 (z196) System 命令 (high-word facility、floating-point extension facility、および population count facility) をさらに活用することができます。
- 新しい UNROLL コンパイラー・オプションによって、ループのアンロールを制御できるようになりました。
- コンパイラーによってインライン・コードが生成され、 ULENGTH および USUBSTR 組み込み関数の文字ストリングが解決できるようになりました。
- コンパイラーによって MEMINDEX(*p*, *n*, *x*) 用のインライン・コードが 生成されるようになりました。*x* は WCHAR(1) です。
- コンパイラーによって STG(*x*) 用のインライン・コードが生成されるようになりました。 この *x* は、REFER を使用する BASED 変数であり、以下の条件の両方を満たします。
  - x 内の NONVARYING BIT はすべて、ALIGNED 属性で指定される。
  - x 内のその他のエレメントはすべて、UNALIGNED で指定される。

デバッグの向上

- コンパイラーが、デバッグ・ツール内のタイプ付き構造体をサポートするようになりました。

SQL サポートの強化

- SQL プリプロセッサーは、次のように大幅に変更されました。
  - 現時点で、ブロック・スコーピングを完全にサポートしています。
  - プリプロセッサー・ロード・モジュールが、8 分の 1 よりも小さくなりました。
  - プリプロセッサーの実行速度が速くなりました。
  - PL/I データ・タイプを指定できる任意の場所で、SQL TYPE 属性が使用できるようになりました。
  - ホスト変数の宣言を処理する際に、 以下のコンパイラー・オプションをサポートするようになりました。 デフォルトが正しく適用され、不適切なホスト変数は適宜拒否されます。
    - DEFAULT(ANS | IBM)
    - DEFAULT(ASCII | EBCDIC)
    - DEFAULT((NO)EVENDEC)
    - DEFAULT((NON)NATIVE)
    - DEFAUT(SHORT(HEX | IEEE))
    - RULES((NO)LAXCTL)
  - PRECISION 属性を正しく処理するようになりました。
  - UNSIGNED 属性と COMPLEX 属性を認識し、 任意のホスト変数でこれらの属性の使用を拒否するようになりました。
  - DSNHMLTR を必要とするコードが含まれる最外部のプロシージャーで、 DSNHMLTR が確実に宣言されるようになりました。
  - パッケージを正しく処理するようになりました。
  - ソース・コード内の文字が、SQLAVDAID を設定するコードが生成されるときに 確実に印刷できるようになりました。
  - 標識配列に下限 1 を設定する必要がなくなりました。

– SQL パラメーター・リスト構造体を生成するようになりました。 この構造体では共用体や init 節の数が少なくなり、構造体のエレメントに基づく追加の宣言はなくなりました。

使いやすさの向上

- 新しい PPLIST コンパイラー・オプションは、メッセージを提供しないプリプロセッサー・フェーズで生成されるリスト部分を条件付きで削除します。
- 構造体宣言にコンマが欠落している場合、コンパイラーは、より良いメッセージを出します。
- ソース・コードに無効のシフトイン・バイトおよびシフトアウト・バイトが含まれている場合、コンパイラーは新規メッセージを出します。
- コンパイラーは、INONLY 属性で宣言される任意のパラメーターに NONASSIGNABLE 属性を適用します。INONLY として宣言されるパラメーターの割り当てにはフラグを立てます。
- RULES(NOLAXENTRY) を指定すると、コンパイラーは DSN で始まる名前には フラグを立てません。
- RULES(NOUNREF) を指定すると、コンパイラーは DSN または SQL で始まる名前には フラグを立てません。
- RULES コンパイラー・オプションの新しいサブオプション NOSELFASSIGN を指定すると、 コンパイラーは変数の自己への割り当てにフラグを立てます。
- RULES コンパイラー・オプションの新しいサブオプション NOLAXRETURN を指定すると、RETURN ステートメントが何らかの無効な方法で使用された場合に、コンパイラーがエラー条件を引き起こすコードを生成します。

V4R1 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

デバッグの向上

- TEST(SEPARATE) が指定されている場合、コンパイラーはオプションでステートメント番号テーブルをデバッグ・ファイルに組み込むため、生成されるオブジェクト・デックのサイズが削減されます。
- TEST(SEPARATE) が指定されている場合、コンパイラーは、宣言、参照、および割り当てのソース行を特定する情報を組み込みます。
- TEST(SEPARATE) が指定されている場合、BASED 変数が配列エレメントの ADDR またはその他の複合参照を基にしていると、コンパイラーは暗黙的ロケーター参照を特定するための情報を生成します。

パフォーマンス向上

- ARCH(9) オプションを使用すると、IBM zEnterprise System 命令を活用できます。
- REFER を使用する構造体のすべてのエレメントがバイト整合である場合、コンパイラーは、ライブラリー呼び出しを通じてではなく、コードのインライン化により、これらのエレメントに対する参照を解決します。

使いやすさの向上

- 新しい DEPRECATE コンパイラー・オプションは、非推奨の変数名および組み込みファイル名にフラグを立てます。
- GONUMBER オプションの新しい SEPARATE サブオプションが提供され、生成されたステートメント番号テーブルを別個のデバッグ・ファイルに組み込むことができます。
- RULES オプションの新しい NOGLOBALDO サブオプションは、親ブロックで宣言されているすべての DO ループ制御変数にフラグを立てます。
- RULES オプションの新しい NOPADDING サブオプションは、埋め込みを含むすべての構造体にフラグを立てます。
- SQL プリプロセッサーが XREF オプションをサポートするようになりました。

- 新しい PLISAXD 組み込みサブルーチンにより、XML System Services パーサーを使用して、スキーマに対する妥当性検査を伴う XML 文書の解析が可能になりました。

保守容易性の向上

- GOSTMT オプションまたは IMPRECISE オプションの使用には、非サポートのフラグが立てられるようになりました。
- コンパイラーは、有効なすべての SQL プリプロセッサー・オプションを常にリストするようになりました。

V3R9 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- 長さが 256 以下のストリングに対する UVALID はインライン化されるようになりました。
- ARCH(7) 以上を指定すると、UTF-8、UFT-16、および UTF-32 相互間の高速変換に対応するために CU12、CU14、CU21、CU24、CU41、および CU42 命令が使用されます。
- ARCH(7) 以上を指定すると、1 バイト・バッファーと 2 バイト・バッファー間の高速変換に対応するために TRTT、TROT、TRTO、および TROO 命令が使用されます。
- スカラーの類似配列の割り当てはストレージ・コピー操作として扱われるようになりました。
- BIT VARYING から BIT VARYING への割り当てはすべてインライン化されるようになりました。
- バイト整合の BIT NONVARYING から BIT VARYING への割り当てはすべてインライン化されるようになりました。
- ROUND および ROUNDDEC 組み込み関数は、丸める引数が DFP であるときインライン化されるようになりました。

  注 : PL/I for z/OS V5.1 以降、ROUNDDEC 組み込み関数は ROUNDAWAYFROMZERO に名前変更されています。
- 最高のパフォーマンスを得るためのオプション選択を簡略化するために、以下のことが行われました。
  - COMPACT オプションが除去されました。
  - DEFAULT(REORDER | ORDER) のデフォルト設定が DEFAULT(REORDER) に変更されました。
  - TUNE オプションが除去されました。
- NULL ポインターの逆参照の検出では、ARCH(8) のもとで新しい比較トラップ命令が活用されます。
- コンパイラーはパフォーマンスの向上のために ARCH(6) でビルドされました。

使いやすさの向上

- CICS プリプロセッサーは、ブロック・スコーピングをサポートするようになったため、必要なローカル CICS 宣言をすべてのネストなしプロシージャーに追加するようになりました。
- SQL プリプロセッサーは、新しい SCOPE オプションをによってホスト変数参照を解決するとき、宣言のスコープに関する PL/I 規則をサポートするようになりました。 NOSCOPE は前のリリースとの互換性を保つためのデフォルトです。
- マクロ・プリプロセッサーは、コンパイラー・リスト内の %include、%xinclude、%inscan、および %xinscan ステートメントをコメントとして残すことになりました。
- マクロ・プリプロセッサーは、%DO SKIP; ステートメントを介して、コンパイルからコードのセクションを省略する簡単かつ明瞭な方法を提供するようになりました。
- マクロ・プリプロセッサーは NAMEPREFIX というオプションをサポートするようになりました。ユーザーは、このオプションを使用して、マクロ・プロシージャーおよび変数を、指定した文字から強制的に開始することができます。

- IGNORE コンパイラー・オプションにより、PUT FILE または DISPLAY ステートメント、あるいはその両方を抑制することができます (いずれか一方のステートメントがデバッグ目的で使用されていたことがありますが、その場合は実動バージョンからコンパイルされました)。
- DEFAULT コンパイラー・オプションの NULLSTRPTR サブオプションにより、割り当てのソースがヌル・ストリングであるとき、ポインターに sysnull または null のどちらを割り当てるかをユーザーが制御できます。
- 新しい MAXGEN オプションは、任意の 1 つのユーザー・ステートメントに対して生成される中間言語ステートメントの最大数を指定するもので、コンパイラーがこの最大数を超える場合、ステートメントにフラグを立てるようにします。
- 新しい ONSNAP オプションにより、ユーザーは、MAIN または FROMALIEN プロシージャーのプロローグに ON STRINGRANGE SNAP; または ON STRINGSIZE SNAP; ステートメントを挿入するようにコンパイラーに要求できるようになりました。
- INITAUTO オプションの新しい SHORT サブオプションは、INITAUTO オプションがランタイム STORAGE オプションの全部を複写するのではなく、レジスターに最適化される可能性がある変数を初期化するように、INITAUTO オプションを制限します。
- 新しい RTCHECK オプションは、NULL ポインターの逆参照をテストするコードを生成します。
- 次のような、リスクを伴う可能性がある各種ステートメントに、コンパイラーがフラグを立てるようになりました。
  - FIXED 演算の結果がゼロより小さいスケール係数を持つコード
  - 関数として使用されたが RETURNS 属性なしで宣言された ENTRY
  - 不適切であるにも関わらず BYVALUE として宣言されたパラメーター (例えば、FIXED DEC パラメーターが BYVALUE であると宣言するなど)
  - FIXEDOVERFLOW を起こす可能性がある FIXED DECIMAL 加算および乗算演算
- 不良コードの制御強化およびフラグ設定を可能にするように RULES オプションが拡張されました。
  - NOPROCENDONLY は、終了する PROC を指定していない、PROC の END ステートメントにフラグを立てます。
  - NOSTOP は STOP および EXIT の使用にフラグを立てます。
  - NOLAXQUAL(STRICT) は、レベル 1 名で修飾されていない変数にフラグを立てます。
  - NOLAXSCALE は、FIXED DEC(p,q)、および FIXED BIN(p,q) の宣言にフラグを立てます (ただし q < 0 または p < q の場合)。
  - NOGOTO(LOOSE) は、同じブロック内にある場合にのみ GOTO を許可します。
  - 複数の DELAY STATEMENT を別々のプロシージャーで並行して実行することができます。

保守容易性の向上

- コンパイラーがファイルを開けないとき、可能であれば、コンパイラーが関連する C ランタイム・メッセージをリスト内のメッセージにも組み込むようになりました。
- ユーザー・コードがコンパイル時に DFP 変換を必要とするが、DFP ハードウェアを備えていないマシンでコンパイルが実行されている場合は、このエラーがトラップされ、意味のあるエラーが出されるようになりました。
- SQL プリプロセッサーがサブオプションとして INCONLY を指定しないで複数回呼び出された場合は、コンパイラーによって作成された DBRM ライブラリーが空になり、ユーザーにこの問題を警告するため E レベル・メッセージが出されるようになりました。

V3R8 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

パフォーマンス向上

- ARCH(8) および TUNE(8) オプションを指定すると、z/HE 命令が使用されます。
- HGPR オプションによって、32 ビット・コードでの 64 ビット・レジスターの使用がサポートされます。
- GOFF オプションによって、GOFF オブジェクトの生成がサポートされます。
- PFPO 命令は、異なる浮動フォーマットの変換で使用されます。
- SRSTU は、UTF-16 INDEX で生成されたコードで使用されます。
- ヌルの内部プロシージャーの呼び出しは、完全に除去されるようになりました。

使いやすさの向上

- PLISAXC によって、SAX インターフェースを使用して XML System Services パーサーにアクセスできます。
- INCDIR オプションは、バッチでサポートされます。
- LISTVIEW オプションによって、TEST の AFTERMACRO などのサブオプションで以前は提供されていたサポートが提供されるようになりました。
- RULES オプションの NOLAXENTRY サブオプションによって、非プロトタイプ ENTRY のフラグを立てることができます。
- DECIMAL オプションの (NO)FOFLONMULT サブオプションによって、FIXED DECIMAL の MULTIPLY で FOFL を発生させるかどうかを制御できます。
- USAGE オプションの HEX および SUBSTR サブオプションによって、ユーザーは、対応する組み込み関数の振る舞いをさらに制御できます。
- DDSQL コンパイラー・オプションによって、EXEC SQL INCLUDE で使用される代替 DD 名を指定できます。
- MACRO および SQL プリプロセッサーによって提供される INCONLY サブオプションを使用して、当該プリプロセッサーが INCLUDE のみを実行するように要求できます。
- LOB(DB2®) SQL プリプロセッサー・オプションが選択されている場合に、統合された SQL プリプロセッサーでは、既にサポート済みの BLOB、CLOB、および DBCLOB SQL 型以外にも、すべての *LOB_FILE、*LOCATOR、ROWID、BINARY、および VARBINARY SQL 型について、Db2 プリコンパイラー・スタイル宣言が生成されるようになりました。

V3R7 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

デバッグの向上

- TEST オプションが機能拡張され、ユーザーが指定したプリプロセッサーが実行された後 (または、すべてのプリプロセッサーが実行された後) にソースが表示されるのと同様に、ソースをリストおよび「デバッグ・ツール・ソース (Debug Tool source)」ウィンドウに表示することを選択できます。

パフォーマンス向上

- BASR 命令が、BALR 命令の代わりに使用されるようになりました。
- 仲介として FIXED BIN(63) を使用することにより、精度の高い FIXED DEC から FLOAT への変換が、インライン化され、高速化されました。
- CHAR 組み込み関数は、CHAR 式に適用される際には常にインライン化されるようになりました。
- FIXED BIN(p,q) からスケールのない FIXED DEC への変換用の生成コードが大幅に改良されました。
- ARCH(7) のもとで TRTR は、TRT が SEARCH および VERIFY に対して使用されたのと同じ状態で、SEARCHR および VERIFYR に対して使用されます。

- UNPKU は、一部の PICTURE を WIDECHAR に変換する (ライブラリー呼び出しをするのではなく) ために使用されます。

使いやすさの向上

- IEEE 10 進浮動小数点 (DFP) がサポートされます。
- 新規の MEMCONVERT 組み込み関数を使用すると、任意のコード・ページ間で任意の長さのデータを変換できます。
- 新規の ONOFFSET 組み込み関数を使用すると、以前は実行時のエラー・メッセージまたはダンプ、つまりある条件が発生したユーザー・プロシージャーのオフセットでのみ使用可能だった他の情報に簡単にアクセスすることができます。
- 新規の STACKADDR 組み込み関数は、現在の動的保存域 (z/OS 上のレジスター 13) のアドレスを戻し、ユーザー独自の診断コードの作成を容易にします。
- アセンブラー・リスト内の簡略記号フィールドの長さが拡大され、長い簡略記号を持つ新規 z/OS 命令のサポートが改善されました。
- 属性、相互参照、およびメッセージ・リストで、使用可能な右マージンが拡張されました。
- CODEPAGE オプションが、1026 (トルコ語コード・ページ) および 1155 (1026 コード・ページに加え、ユーロ記号) に対応できるようになりました。
- 新規の MAXNEST オプションを使用すると、BEGIN、DO、IF、および PROC ステートメントの過度なネストにフラグを立てることができます。
- RULES オプションの新規 (かつ、デフォルトでない) のサブオプション NOELSEIF を指定すると、直後に IF ステートメントが続く ELSE ステートメントにコンパイラーがフラグを立て、SELECT ステートメントとして書き直すように提案します。
- RULES オプションの新規 (かつ、デフォルトでない) のサブオプション NOLAXSTG を指定すると、コンパイラーは、変数 A が ADDR(B) および STG(A) > STG(B) の BASED として宣言されている場所にフラグを立てます。以前のように、B が定数の存在期間を指定した AUTOMATIC、BASED、または STATIC である場合のみでなく、B が定数の存在期間を指定して宣言されるパラメーターである場合にもフラグが立てられます。
- 新規の QUOTE オプションを使用すると、引用符 (") 記号に大体コード・ポイントを指定することができます。これは、この記号がコード・ページ・インバリアントではないためです。
- 新規の XML コンパイラー・オプションを使用すると、XMLCHAR 組み込み関数の出力内のタグを、すべて大文字にするか、大/小文字を宣言で使用した とおりに指定できます。
- メッセージを生成しないコンパイルであっても、コンパイラー・メッセージがリストされるはずの行に、「コンパイラー・メッセージはありません (no compiler messages)」というメッセージが表示されるようになりました。
- マクロ・プリプロセッサーは、%INCLUDE ステートメントのみを処理するのか、すべてのマクロ・ステートメントを処理する必要があるかを指定することのできる新規サブオプションをサポートします。
- LOB(Db2) SQL プリプロセッサー・オプションが選択されている場合に、統合された SQL プリプロセッサーでは、既にサポート済みの BLOB、CLOB、および DBCLOB SQL 型以外にも、すべての *LOB_FILE、*LOCATOR、ROWID、BINARY、および VARBINARY SQL 型について、Db2 プリコンパイラー・スタイル宣言が生成されるようになりました。

V3R6 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

Db2 V9 サポート

- STDSQL(YES/NO) のサポート
- CREATE TRIGGER (複数 SQL ステートメント) のサポート
- FETCH CONTINUE のサポート

- SQL ステートメントに組み込まれた SQL スタイルのコメント ('--') のサポート
- 以下の追加 SQL TYPE のサポート
  - SQL TYPE IS BLOB_FILE
  - SQL TYPE IS CLOB_FILE
  - SQL TYPE IS DBCLOB_FILE
  - SQL TYPE IS XML AS
  - SQL TYPE IS BIGINT
  - SQL TYPE IS BINARY
  - SQL TYPE IS VARBINARY
- SQL プリプロセッサーが Db2 コプロセッサー・オプションもリストするようになりました。

デバッグの向上

- TEST(NOSEPNAME) を指定すると、デバッグ・サイドのファイルの名前がオブジェクト・デックに保存されません。

パフォーマンス向上

- ARCH(7) での z/OS 拡張即値機能のサポート
- ARCH(6) での CLCLU、MVCLU、PKA、TP、および UNPKA 命令の活用
- ARCH(5) での CVBG および CVDG 命令の活用
- CLCLE の拡張使用
- Db2 日時パターンに関する変換をインライン化
- ALLOCATION 組み込み関数をインライン化
- 浮動 $ を含んだ変換をインライン化
- PIC'(n)Z' への割り当てから条件付きコードを除去
- FIXED BIN からスケール因数を指定する PICTURE への変換をインライン化
- 次元を継承したが 8 で割り切れるストライドを持つ BIT 変数への割り当てをインライン化

使いやすさの向上

- ブロックが使用する、AUTOMATIC ストレージのストレージ・オフセット (ブロックごと) 順のリストも、MAP 出力に含まれるようになりました。
- 規格合致検査が拡張されて構造体が含められました。
- リストには、ファイル内の行番号用に 7 カラムが含まれるようになります。
- z/OS 環境では THREADID 組み込み関数がサポートされるようになりました。
- PICSPEC 組み込み関数がサポートされるようになりました。
- 新規 CEESTART オプションにより、オブジェクト・デックの始めまたは終わりに CEESTART csect を配置できるようになりました。
- 新規 PPCICS、PPMACRO、および PPSQL オプションにより、対応するプリプロセッサーで使用されるデフォルト・オプションを指定できるようになりました。
- ATTRIBUTES リストに ENVIRONMENT オプションが組み込まれるようになりました。
- DISPLAY オプションがサポートする新規サブオプションによって、REPLY を指定した DISPLAY か REPLY を指定しない DISPLAY が、さまざまな DESC コードで可能になりました。
- コメント内のセミコロンに対するフラグ・メッセージに、セミコロンのある行の行番号が組み込まれるようになりました。
- REFER 項目が変更される可能性がある割り当てに対してフラグが立ちます。
- KEY/KEYFROM 文節が含まれない KEYED DIRECT ファイルの使用に対してフラグが立ちます。

- PICTURE をループ制御変数として使用することに対してフラグが立ちます。

V3R5 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

デバッグの向上

- TEST(SEPARATE) を指定すると、デバッグ情報の大部分が別個のデバッグ・ファイルに書き込まれます。
- AUTOMONITOR に割り当てのターゲットが含まれます。
- AUTOMATIC の初期化後に AT ENTRY フックが配置されるようになりました。それによって、変数を調べる前のブロックへのステップイントゥが必要なくなりました。

パフォーマンス向上

- 分岐相対命令の生成により、基底レジスターと制御権移動ベクトルの必要性が大幅に削減されます。
- ARCH(6) での z/OS 長変位機能のサポート。
- REFER を使用する単純構造体が、ライブラリー呼び出しを介さずにインラインでマップされます。
- REFER を使用する構造体のうち、これまでのようにライブラリー呼び出しを介してマップされるものについては、REFER が副構造体の配列の境界を指定していれば、生成されるコードが少なくなります。
- 重複 INCLUDE が高速で処理されるようになりました。
- 最終位置が I または R の PICTURE 変数への変換が、インライン化されました (最終文字が T の PICTURE 変数への変換は既にインライン化されていました)。
- B を含まないピクチャーがインライン化された場合、そのピクチャーに対応する、1 つ以上の B で終わる PICTURE 変数への変換がインライン化されるようになりました。
- CHARACTER 変数から X のみで構成される PICTURE 変数への変換が、インライン化されるようになりました。

使いやすさの向上

- リストなどのすべての部分で、ソース・ファイルがファイル 0、最初のインクルード・ファイルがファイル 1、2 番目の (固有の) インクルード・ファイルがファイル 2 のように数えられます。
- 規格合致検査が拡張されて配列が含められました。
- 呼び出されたプリプロセッサーのビルドの日付がリストに組み込まれます。
- COBOL との ILC を簡単にするために、1 バイト FIXED BINARY 引数を抑止できます。
- SYSADATA、SYSXMLSD、および SYSDEBUG の代替 DD 名を指定できます。
- XNUMERIC を指定した場合、RULES(NOLAXMARGINS) はシーケンス番号を許容します。
- RULES(NOUNREF) は、参照されない AUTOMATIC 変数に対してフラグを立てます。
- 変数への割り当てがライブラリー呼び出しを介して行われる場合、ライブラリー呼び出しに対するフラグ・メッセージには、ターゲット変数の名前が含まれます。
- 一回限りの DO ループに対してフラグが立ちます。
- 引数として使用されるラベルに対してフラグが立ちます。
- PRV が使用される場合、FETCHABLE の PARAMETER CONTROLLED 以外の ALLOCATE と FREE に対してフラグが立ちます。
- DEFINED および BASED がそれぞれの基数より大きい場合、基数が後で宣言されるとしても、それぞれに対してフラグが立ちます。
- FIXED DEC から 8 バイト整数への暗黙的型変換に対してフラグが立ちます。

V3R4 からの機能拡張

本リリースでは以下の機能が強化されています。 強化された機能については、本書のほか、他の IBM PL/I ブックにも説明があります。

マイグレーションの強化

- 旧コードとの CONTROLLED の共用のサポート
- デフォルト初期化の向上
- ADD、DIVIDE、および MULTIPLY での小数部指定の容易化
- GRAPHIC の STRING に対する旧セマンティクスのサポート
- DEFAULT ステートメントに対する旧セマンティクスのサポート
- ストレージ・オーバーレイを使用したフラグ宣言
- 開始 内の リターンに対するリフトの制限
- コメント内のセミコロンに対するフラグ付け (オプション)
- アセンブラーで初期化される EXT STATIC のサポート
- 無効な紙送り制御文字に対するフラグ付け
- 言語の誤用 (特に RETURN) に対するフラグ付けの強化
- REPLACEBY2 組み込み関数のサポート
- SIZE を発生させる 10 進代入における FOFL の抑止 (オプション)
- 言語で旧コンパイラーとは異なる処理をしたことに対するフラグ付けの強化

パフォーマンス向上

- INDEX および TRANSLATE でのコード生成の向上
- ピクチャーに対する代入のインライン化の向上
- ソースが FIXED DEC であった場合に、変換がインラインで行われるときの PICTURE に対する CHARACTER の変換で生成されるコードの向上
- パック 10 進数変換で生成されるコードの向上
- REFER の一部の使用法で生成されるコードの向上
- 長さが不明な文字ストリングの比較のインライン化
- 連結に使用されるスタック・ストレージの量の削減
- GET/PUT STRING EDIT ステートメントのインライン化の向上
- LE 条件処理の短縮の強化
- BIN FIXED の OR および AND のインライン化の向上
- ALIGNED BIT(8) に対する SIGNED FIXED BIN(8) のインライン化
- コンパイラーが実行時に構造にマップするためのライブラリー・ルーチン呼び出し を生成するステートメントに対するフラグ付け
- リスト生成に使用される I/O の量の削減

使いやすさの向上

- オプションで、16 進数のアグリゲート リストにオフセットを指定します
- LIMITS(FIXEDDEC(15,31)) オプションによる、必要な場合のみの DEC(31) のサポート
- オプション内でのコメントの許可
- オプションで、偶数精度の FIXED DEC 宣言にフラグを立てる
- SIZE を発生させる可能性がある DEC 割り当てに対して、オプションで DEC にフラグを立てる
- SIZE を発生させる可能性のある PIC 代入に対する DEC/PIC のフラグ付け

- NOINIT 属性による、INIT なしの LIKE のサポート
- z/OS UNIX での PDS からの組み込みの容易化
- マクロ・プリプロセッサー内での LOWERCASE、MACNAME、TRIM および LOWERCASE 組み込み関数のサポート
- PTF による、オプションの紹介の容易化
- オプションで *PROCESS の使用を許可しない
- MDECK 内での *PROCESS の保持 (オプション)
- 一回限りの INCLUDE のサポート
- マクロで決定する INCLUDE 名のサポート
- 実行時のストリング・パラメーター検査のサポート
- 未初期化変数である可能性があることに対するフラグ付けの強化
- コーディング・エラーの可能性がある、異常な比較に対するフラグ付け
- STORAGE オプションの出力のフォーマットが使いやすくなり、LIST オプションの出力には、コンパイル単位から各ブロックまでの 16 進オフセット が組み込まれるようになりました。

デバッグの向上

- オーバーレイ・フックに対するより良いサポート
- LE ダンプ内の CONTROLLED 変数の解決の容易性の強化
- リストへのユーザー指定のオプションの常時組み込み

V3R3 からの機能拡張

このリリースには、以下を含む Enterprise PL/I V3R3 で拡張された機能 がすべて備わっています。

XML サポートの強化

XMLCHAR 組み込み関数が、参照される構造体のエレメントの名前と値で XML をバッファーに書き込み、書き込まれたバイト数を戻します。 次にこの XML は、PL/I SAX パーサーを使用したコードとともに他のアプリケーションに渡され、実行されます。

パフォーマンスの改善

- OPT(2) でのコンパイル時間は、Enterprise PL/I V3R2 の場合よりも、特に大規模なプログラムで大幅に少なくなります。
- コンパイラーは、ED および EDMK 命令を使用して、PICTURE および CHARACTER へのインライン化された数値変換を行います。 これにより、より早く、短いコード・シーケンスおよびコンパイルの高速化が実現しました。
- コンパイラーは、ストリング比較をさらに効率よく行うコードを生成するようになりました。 このことも、より早く、短いコード・シーケンスという結果をもたらしています。
- コンパイラーは、より短く高速なコードを生成し、FIXED DECIMAL から、末尾に overpunch という文字のついた PICTURE への変換を行います。
- ARCH および TUNE コンパイラー・オプションは、有効なサブオプションとして 5 つを受け入れます。 ARCH(5) のもとでは、コンパイラーが適切なときに、NILL、NILH、OILL、OILH、LLILL、および LLILH などの新規 z/Architecture 命令を生成します。

容易なマイグレーション

- 新規の BIFPREC コンパイラー・オプションは、さまざまな組み込み関数によって戻された FIXED BIN の結果の精度を制御し、これにより OS PL/I コンパイラーとのよりよい互換性を提供します。

- 新規の BACKREG コンパイラー・オプションは、コンパイラーが、逆チェーン・レジスターとしてどのレジスターを使用するかを制御し、これにより、古いオブジェクト・コードと新しいオブジェクト・コードの混合を容易にします。
- 新規の RESEXP コンパイラー・オプションは、コードの中の制限つき式の評価を制御し、OS PL/I コンパイラーとのよりよい互換性を提供します。
- 新規の BLKOFF コンパイラー・オプションは、コンパイラーの疑似アセンブラー・リストにおけるオフセットの計算方法を制御します。
- STORAGE コンパイラー・オプションは、それぞれのプロシージャーおよび開始ブロックによって使用されるストレージの要約をコンパイラーに作成させます。これは、リストの一部として作成され、OS PL/I コンパイラーで作成されたものと似ています。

使いやすさの向上

- RULES コンパイラー・オプションの新規の LAXDEF サブオプションにより、いわゆる無許可定義を使用できるようになり、この際、コンパイラーが E レベル・メッセージを生成することはありません。
- 新規の FLOATINMATH コンパイラー・オプションにより、数学関数の評価に関する精度の制御が容易になりました。
- 新規の MEMINDEX、MEMSEARCH(R) および MEMVERIFY(R) 組み込み関数により、32K より大きいストリングの検索が可能です。
- DISPLAY(WTO) コンパイラー・オプションの、新規の ROUTCDE および DESC サブオプションは、対応する WTO の要素の制御を提供します。
- コンパイラーは、それぞれのオブジェクトの中に短ストリングを保管し、それが関連するコードが実行されている間もストレージにあり、そのオブジェクトを作成するために使用されるオプションをすべて記録します。 これにより、さまざまなツールによるより良い診断が可能になりました。
- コンパイラーは、ステートメントがマージまたは削除された場所をさらに識別するメッセージを発行します。
- PLIDUMP 出力は、静的ユーザーの 16 進ダンプをインクルードするようになりました。
- PLIDUMP 出力は、言語環境プログラムのトレースバックで、それぞれのプログラムをコンパイルするために使用するオプションをインクルードするようになりました。
- PLIDUMP 出力は、PL/I ファイルに関する情報をインクルードするようになりました。

デバッグ・サポートの向上

- REFER を使用した BASED 構造体は、DebugTool および データ指示 I/O ステートメントでサポートされるようになりました (制限は他のすべての BASED 変数上と同じです)。
- 他の構造体のスカラー・メンバーから (順番に BASED されるなどで) BASED された BASED 構造体は、DebugTool およびデータ指示 I/O ステートメントでサポートされるようになりました (制限は他のすべての BASED 変数上と同じです)。

V3R2 からの機能拡張

本リリースでは、次を含む Enterprise PL/I V3R2 での機能強化もすべて提供されます。

パフォーマンスの改善

- このコンパイラーでは、インライン・コードを生成してより多くの型変換を処理できるようになりました。 これにより、型変換が以前に比べ、格段に早く行われるようになります。 また、ライブラリー呼び出しによって行われるすべての型変換は、コンパイラーによってフラグが立てられるようになりました。
- コンパイラー生成コードが、 さまざまな状況において使用するスタック・ストレージの容量が減少しました。
- コンパイラーが、TRANSLATE 組み込み関数を 参照するために生成するコードが改善されました。
- SUBSCRIPTRANGE 検査用のコンパイラー生成コードは、既知の境界を持つ配列の場合は、 処理速度が以前の倍になりました。

- ARCH と TUNE オプションは、サブオプション 4 をサポートするように なり、zSeries マシンで新たに命令の開発が行えるようになりました。
- ARCH(2)、FLOAT(AFP) および TUNE(3) がデフォルトになりました。

容易なマイグレーション

- マイグレーションを容易にし、互換性を持たせるために、コンパイラーのデフォルト値が 変更されました。 変更されたデフォルト値は、次のとおりです。
  - CSECT
  - CMPAT(V2)
  - LIMITS(EXTNAME(7))
  - NORENT
- コンパイラーは、OPTIONS(COBOL) を指定した PROC および ENTRY で、NOMAP、NOMAPIN および NOMAP 属性 を指定できるようになりました。
- コンパイラーは、複数の ENTRY ステートメントを指定した PROC をサポートするようになりました。 この ENTRY ステートメントは、前リリースのホスト・コンパイラーと同様、それぞれが異なる RETURNS 属性を 保持することができます。
- コンパイラーは、OPTIONS(RETCODE) において、PROC と ENTRY には OPTIONS(COBOL) が指定されていることを 想定しています。
- 未処理の場合、SIZE 条件は ERROR にプロモートされません。
- コンパイル時間とストレージ要件を減らすために、さまざまな変更が行われました。
- OFFSET オプションは、前リリースの PL/I コンパイラーで生成されたものとよく似たステートメント・オフセット・テーブルを 生成します。
- FLAG オプションは前リリースのコンパイラーの場合と意味はまったく同じですが、 新規の MAXMSG オプションは、 指定された重大度において、メッセージが指定した回数発生した後に コンパイラーを終了すべきかどうかを、ユーザーが決定できるようになりました。 例えば、FLAG(I) MAXMSG(E,10) を指定すると、I レベル・メッセージはすべて確認し、E レベル・メッセージは 10 回発生したらコンパイルを終了するように指定できます。
- AGGREGATE リストには、調節可能エクステントを備えた構造を組み込めるようになりました。
- STMT オプションは、リストのいくつかのセクションをサポートするようになりました。
- LINESIZE で使用できる最大値は、F フォーマット・ファイルでは 32759、V フォーマット・ファイルでは 32751 に 変更されました。

使いやすさの向上

- コンパイラー・オプションのデフォルトは、インストール時に変更できるようになりました。
- 内蔵 SQL プリプロセッサーは、Db2 Unicode をサポートするようになりました。
- コンパイラーは、デバッグ・ツールが Auto Monitor をサポートできるようにする情報を 生成するようになりました。それで、各ステートメントが実行される直前に、ステートメントで使用されるすべての変数のすべての値が表示されます。
- 新規の NOWRITABLE コンパイラー・オプションを使用すると、NORENT を指定した場合に、 最適なパフォーマンスを犠牲にしても、コンパイラーが FILE と CONTROLLED を 操作するコードを生成する際に、書き込み可能な静的値を使用しないように指定できます。
- 新規の USAGE コンパイラー・オプションを使用すると、RULES(IBM|ANS) オプションの他の影響を受けずに、ROUND および UNSPEC 組み込み関数の IBM または ANS 動作を完全に制御できます。
- 新規の STDSYS コンパイラー・オプションは、 コンパイラーに SYSPRINT ファイルと C stdout ファイルを同一にするように指定します。
- 新規の COMPACT コンパイラー・オプションが使用されると、 コードが大きくなることを制限する最適化を使用するようにコンパイラーに指示します。
- SYSPRINT の LRECL は 137 に変更され、C/C++ コンパイラーの LRECL と 一致するようになりました。

- PUT LIST と PUT EDIT ステートメントで POINTER が使用できるように なりました。8 バイトの 16 進値が出力されます。
- ABNORMAL 属性を STATIC 変数で指定すると、STATIC 変数が 使用されていなくてもこの変数は保存されます。

V3R1 からの機能拡張

本リリースでは、次を含む Enterprise PL/I V3R1 での機能強化もすべて提供されます。

- z/OS でのマルチスレッド化のサポート
- z/OS での IEEE 浮動小数点のサポート
- マクロ・プリプロセッサーでの ANSWER ステートメントのサポート
- PLISAXA および PLISAXB 組み込みサブルーチンを介した SAX 形式 XML 構文解析
- 追加の組み込み関数
  - CS
  - CDS
  - ISMAIN
  - LOWERCASE
  - UPPERCASE

VisualAge PL/I からの機能拡張

本リリースでは、次を含む VisualAge® PL/I V2R2 での機能強化もすべて提供します。

- WIDECHAR 属性を介した初期 UTF-16 サポート

  以下については、まだサポートされていません。

  - ソース・ファイル内の WIDECHAR 文字
  - W ストリング定数
  - ストリーム入出力内の WIDECHAR 式の使用
  - レコード入出力での WIDECHAR からの暗黙の型変換または WIDECHAR への暗黙の型変換
  - レコード入出力での暗黙の endianness フラグ

    WIDECHAR ファイルを作成する場合は、 ファイルの最初の 2 バイトとして endianness フラグ ('fe_ff'wx) を書き込んでください。
- DEFAULT ステートメントでサポートされる DESCRIPTORS オプションと VALUE オプション
- PUT DATA 機能強化
  - POINTER、OFFSET およびサポートされているその他の非計算変数
  - タイプ 3 DO 仕様が使用可能
  - 添え字が使用可能
- DEFINE ステートメントの機能強化
  - 指定されていない構造体の定義
  - CAST および RESPEC タイプ付き関数
- 追加の組み込み関数
  - CHARVAL
  - ISIGNED
  - IUNSIGNED
  - ONWCHAR
  - ONWSOURCE

  - WCHAR
  - WCHARVAL
  - WHIGH
  - WIDECHAR
  - WLOW
- プリプロセッサー機能強化
  - プリプロセッサー・プロシージャーでの配列のサポート
  - %DO ステートメントでの WHILE、UNTIL および LOOP キーワードのサポート
  - %ITERATE ステートメントのサポート
  - %LEAVE ステートメントのサポート
  - %REPLACE ステートメントのサポート
  - %SELECT ステートメントのサポート
  - 追加の組み込み関数
    - COLLATE
    - COMMENT
    - COMPILEDATE
    - COMPILETIME
    - COPY
    - COUNTER
    - DIMENSION
    - HBOUND
    - 索引
    - LBOUND
    - 長さ
    - MACCOL
    - MACLMAR
    - MACRMAR
    - 最大
    - MIN
    - PARMSET
    - QUOTE
    - REPEAT
    - SUBSTR
    - SYSPARM
    - システム
    - SYSVERSION
    - TRANSLATE
    - VERIFY

ご意見の送付方法

本書または PL/I の他のマニュアルについてご意見がありましたら、IBM 発行のマニュアルに関する情報の Web ページ (http://www.ibm.com/jp/manuals/) よりお送りください。今後の参考にさせていただきます。(URL は、変更になる場合があります)

この文書または他の PL/I 文書についてご意見がある場合は、次の宛先に電子メールをお送りください。 Compilers@ibm.com.

資料名、資料番号、PL/I のバージョン、および該当する場合はご意見の対象であるテキストの具体的な場所 (例: ページ番号) を必ず記載してください。

お客様が IBM に情報を送信する場合、お客様は、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、当該情報を使用または配布する非独占的権利を IBM に付与します。

アクセシビリティー

アクセシビリティー機能は、運動や視覚などに障害を持つユーザーが情報技術製品を快適に使用できるように支援します。 z/OS のアクセシビリティー機能には、Enterprise PL/I に対するアクセシビリティーが備わっています。

アクセシビリティー機能

z/OS には、以下の主要アクセシビリティー機能が含まれています。

- スクリーン・リーダーおよび画面拡大ソフトウェアで一般的に使用されるインターフェース
- キーボードのみによるナビゲーション
- 色、コントラスト、フォント・サイズなどの表示属性をカスタマイズする機能

z/OS は、最新の W3C 標準である WAI-AL1.0 (http://www.w3.org/TR/wai-aria/)を使用して、 US セクション 508 (https://www.access-board.gov/ict/) および Web Content Accessibility Guidelines (WCAG) 2.0 (http://www.w3.org/TR/WCAG20/)に確実に準拠します。 アクセシビリティー機能を利用するには、最新リリースのスクリーン・リーダーを、この製品でサポートされる最新の Web ブラウザーと併用してください。

キーボード・ナビゲーション

ユーザーは z/OS ユーザーは、 TSO/E または ISPF を使ってユーザー・インターフェースにアクセスすることができる。

また、ユーザーは IBM Developer for z/OS を使用して z/OS サービスにアクセスすることもできます。

これらのインターフェースへのアクセスに関する情報については、以下の資料を参照してください。

- *z/OS TSO/E* 入門
- *z/OS TSO/E* ユーザーズ・ガイド
- *z/OS ISPF* ユーザーズガイド第 *1* 巻
- *IBM Developer for z/OS* の資料

上記の資料には、 キーボード・ショートカットまたはファンクション・キー (PF キー) の使用方法を 含む TSO/E および ISPF の使用方法が記載されています。 それぞれの資料では、PF キーのデフォルトの設定値とそれらの機能の変更方法についても説明しています。

インターフェース情報

エンタープライズ PL/I オンライン製品資料は、標準の Web ブラウザーで表示できる IBM 資料で入手できます。

PDF ファイルでのアクセシビリティー・サポートは限定的です。 PDF 資料では、オプションのフォント拡大機能およびハイコントラスト表示設定を使用でき、キーボードのみでナビゲートできます。

スクリーン・リーダーが構文図やソース・コード例を正確に読み上げたり、ピリオドやコンマの PICTURE 記号を含むテキストを正確に読み上げたりできるようにするには、スクリーン・リーダーがすべての句読文字を読み上げるように設定する必要があります。

支援技術製品は、z/OS にあるユーザー・インターフェースと連携して機能します。 ガイダンス情報については、z/OS インターフェースへのアクセスに使用する支援技術製品の資料を参照してください。

関連アクセシビリティー情報

IBM では、標準の IBM ヘルプ・デスクおよびサポート Web サイトに加えて、耳が不自由なお客様に販売およびサポート・サービスをご利用いただけるように TTY 電話サービスを提供します。

TTY サービス
800-IBM-3383 (800-426-3383)
(北アメリカ内)

IBM のアクセシビリティーへの取り組み

IBM のアクセシビリティーへの取り組みについては、IBM Accessibility (www.ibm.com/able) を参照してください。

第 1 部 プログラムのコンパイル

© Copyright IBM Corp. 1999, 2025

第 1 章 コンパイラー・オプションと機能の使用

この章では、コンパイラーに使用できるオプションと、その省略形 および IBM 提供のデフォルトについて説明します。

重要 : PL/I はコンパイル時に Language Environment ランタイムへのアクセスを要求します。

Enterprise PL/I を使用すれば、31 ビット・アプリケーションと 64 ビット・アプリケーションの両方を開発できます。 コードをコンパイルするときは、31 ビット・アプリケーションには LP(32) オプションを使用し、64 ビット・アプリケーションには LP(64) オプションを使用します。 ただし、LP(64) でのコンパイラーの動作は LP(32) でのコンパイラーの動作とは異なります。 詳しくは、『181 ページの『第 7 章 64 ビット・アプリケーションを開発する場合の考慮事項』』および『177 ページの『第 6 章 64 ビット・プログラムに対するリンク・エディットおよび実行』』を参照してください。

大部分のデフォルトは、PL/I プログラムのコンパイル時に指定変更することができます。 コンパイラーのインストール時には、デフォルトを指定変更することもできます。

あるリリースのすべてのコンパイラー・オプションのデフォルト設定は、 SIBMZSAM データセットの IBMXO*vrm*, where *v* がバージョン番号、 *r* がリリース番号、 *m* が修正番号で、通常は 0 です（たとえば、V5R3 は IBMXO530 、 V5R1 は IBMXO510 ）。 このデータセットには、サポートされているすべての PL/I リリースの IBMXO*vrm* メンバが含まれている。 リリースごとにデフォルト設定のどこが変更され、どのような新しいオプションが追加されたかを確認するには、2 つのリリースの IBMXO*vrm* 。 また、 IBMXO*vrm* ファイルをテンプレートとして使用し、お好みの設定で独自のオプションファイルを作成することもできます。

コンパイル時オプションの説明

多くのコンパイラー・オプションに肯定形式と否定形式があります。 否定形式は、肯定形式の初めに NO を付け加えたものです (例えば TEST および NOTEST)。 オプションによっては、肯定形式のみを持つものもあります (例: SYSTEM)。

コンパイラー・オプションには以下の 3 つのタイプがあります。

1. キーワードの単純な組み合わせ:機能を要求する肯定形式、およびその機能を禁止する代替否定形式 (例えば、NEST および NONEST)
2. オプションを修飾する値リストを提供するためのキーワード (例えば、FLAG(W))
3. 上記の 1 と 2 を組み合わせたもの (例えば、NOCOMPILE(E))

3 ページの表 3 に、すべてのコンパイラー・オプションと、その省略形 (存在する場合)、および IBM 提供のデフォルト値をリストします。 省略できるサブオプションがオプションに含まれている場合、その省略形については、オプションの詳細説明のところで説明します。

簡潔にするため、オプションのいくつかは、表では緩やかに記述されている（例えば、 LANGLVL のサブオプションの 1 つだけが必須であり、同様に、 TEST のサブオプションの 1 つを指定すれば、もう 1 つを指定する必要はない)。 完全に正確な構文全体については、後続のトピックで説明します。

表 *3. Compile-time options, abbreviations, and IBM-supplied defaults*

Compile-time option	Abbreviated name	z/OS default
AGGREGATE[(DEC|HEX)] | NOAGGREGATE	AG | NAG	NOAGGREGATE
ARCH(n)	-	ARCH(12)
ASSERT(ENTRY | CONDITION)	-	ASSERT(ENTRY)
ATTRIBUTES[(FULL|SHORT)] | NOATTRIBUTES	A | NA	NA [(FULL)][1]
BACKREG(5 | 11)	-	BACKREG(5)

© Copyright IBM Corp. 1999, 2025

表 3. *Compile-time options, abbreviations, and IBM-supplied defaults (続き)*

Compile-time option	Abbreviated name	z/OS default
BIFPREC(15 | 31)	-	BIFPREC(31)
BLANK('c')	-	BLANK('t')[2]
BLKOFF | NOBLKOFF	-	BLKOFF
BRACKETS('*symbol_1symbol_2*')	-	BRACKETS(‘[]’)
CASE(UPPER | ASIS)	-	CASE(UPPER)
CASERULES(KEYWORD(LOWER | MIXED | START | UPPER))	-	CASERULES(KEYWORD (MIXED))
CHECK(STORAGE | NOSTORAGE)	-	CHECK(NSTG)
CMPAT(LE | V1 | V2 | V3)	CMP	CMPAT(V2)
CODEPAGE(n)	CP	CODEPAGE(1140)
COMMON | NOCOMMON	-	NOCOMMON
COMPILE | NOCOMPILE[(W | E | S)]	C | NC	NOCOMPILE(S)
COPYRIGHT(’string’) | NOCOPYRIGHT	-	NOCOPYRIGHT
CSECT | NOCSECT	CSE | NOCSE	CSECT
CSECTCUT(n)	-	CSECTCUT(4)
CURRENCY('c')	CURR	CURRENCY($)
DBCS | NODBCS	-	NODBCS
DBRMLIB(*'data-set-name'*) | NODBRMLIB	-	NODBRMLIB
DD(ddname-list)	-	DD(SYSPRINT,SYSIN, SYSLIB,SYSPUNCH, SYSLIN,SYSADATA, SYSXMLSD,SYSDEBUG)
DDSQL(ddname)	-	DDSQL('')
DECIMAL(CHECKFLOAT | NOCHECKFLOAT, FOFLONADD | NOFOFLONADD, FOFLONASGN | NOFOFLONASGN, FOFLONDIV | NOFOFLONDIV, FOFLONMULT | NOFOFLONMULT, FORCEDSIGN | NOFORCEDSIGN, KEEPMINUS | NOKEEPMINUS, TRUNCFLOAT | NOTRUNCFLOAT)	DEC	DEC(NOCHECKFLOAT, NOFOFLONADD, FOFLONASGN, NOFOFLONDIV, NOFOFLONMULT, NOFORCEDSIGN, NOKEEPMINUS, NOTRUNCFLOAT)
DECOMP | NODECOMP	-	NODECOMP
DEFAULT(*attribute* | *option*)	DFT	See DEFAULT

表 3. *Compile-time options, abbreviations, and IBM-supplied defaults (続き)*

Compile-time option	Abbreviated name	z/OS default
DEPRECATE(BUILTIN(*built-in-name*) | ENTRY(*entry-name*) | INCLUDE(*filename*) | STMT(*statement-name*) | VARIABLE(*variable-name*))	-	DEPRECATE(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())
DEPRECATENEXT(BUILTIN(*built-in-name*) | ENTRY(*entry-name*) | INCLUDE(*filename*) | STMT(*statement-name*) | VARIABLE(*variable-name*))	-	DEPRECATENEXT(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())
DISPLAY (STD | WTO(ROUTCDE(x) DESC(y) REPLY(z)))	-	DISPLAY(WTO)
DLL | NODLL		NODLL
DLLINIT | NODLLINIT	-	NODLLINIT
EXIT | NOEXIT	-	NOEXIT
EXTRN(FULL | SHORT)	-	EXTRN(FULL)
EXPORTALL	-	EXPORTALL
FILEREF | NOFILEREF	-	FILEREF
FLAG[(I | W | E | S)]	F	FLAG(W)
FLOAT(DFP | NODFP)	-	FLOAT(NODFP)
FLOATINMATH(ASIS | LONG | EXTENDED)	-	FLOATINMATH(ASIS)
GOFF | NOGOFF	-	NOGOFF
GONUMBER(SEPARATE | NOSEPARATE) | NOGONUMBER	GN | NGN	NOGONUMBER
GRAPHIC | NOGRAPHIC	GR | NGR	NOGRAPHIC
IGNORE(ASSERT | DISPLAY | PUT) | NOIGNORE	-	NOIGNORE
HEADER(SOURCE | FILE)	-	SOURCE
HGPR(NOPRESERVE | PRESERVE)	-	NOPRESERVE
INCAFTER([PROCESS(*filename*)])	-	INCAFTER()
INCDIR('*directory name*') | NOINCDIR	-	NOINCDIR
INCLUDE | NOINCLUDE	-	INCLUDE
INCPDS('*PDS name*') | NOINCPDS	-	NOINCPDS
INITAUTO([SHORT | FULL]) | NOINITAUTO	-	NOINITAUTO
INITBASED | NOINITBASED	-	NOINITBASED
INITCTL | NOINITCTL	-	NOINITCTL

表 3. *Compile-time options, abbreviations, and IBM-supplied defaults (続き)*

Compile-time option	Abbreviated name	z/OS default
INITSTATIC | NOINITSTATIC	-	NOINITSTATIC
INSOURCE[(FULL|SHORT)(ALL|FIRST)] | NOINSOURCE	IS | NIS	NOINSOURCE
INTERRUPT | NOINTERRUPT	INT | NINT	NOINTERRUPT
JSON(CASE(UPPER | ASIS | LOWER), GET(HEEDCASE | IGNORECASE),TRIMR | NOTRIMR)	-	JSON(CASE(UPPER), GET(HEEDCASE), TRIMR)
LANGLVL(NOEXT | OS)	-	LANGLVL(OS)
LIMITS(*options*)	-	See 48 ページの『LIMITS』
LINEDIR | NOLINEDIR	-	NOLINEDIR
LIST | NOLIST	-	NOLIST
LISTVIEW(SOURCE | AFTERMACRO | AFTERCICS | AFTERSQL | AFTERALL)	-	LISTVIEW(SOURCE)
LP(32 | 64)	-	LP(32)
MACRO | NOMACRO	M | NM	NOMACRO
MAP | NOMAP	-	NOMAP
MARGINI('c') | NOMARGINI	MI | NMI	NOMARGINI
MARGINS(m,n[,c])| NOMARGINS	MAR(m,n)	MARGINS F-format: (2,72) V-format: (10,100)
MAXBRANCH(max)	-	MAXBRANCH(2000)
MAXINIT	-	MAXINIT(64K)
MAXGEN(n)	-	MAXGEN(100000)
MAXMEM(n)	MAXM	MAXMEM(1048576)
MAXMSG(I | W | E | S,n)	-	MAXMSG(W,250)
MAXNEST(BLOCK(x) DO(y) IF(z))	-	MAXNEST(BLOCK(17) DO(17) IF(17))
MAXRUNONIF(n)	-	MAXRUNONIF(10)
MAXSTMT(n)	-	MAXSTMT(4096)
MAXTEMP(n)	-	MAXTEMP(50000)
MDECK | NOMDECK	MD | NMD	NOMDECK
MSGSUMMARY[(XREF | NOXREF)] | NOMSGSUMMARY	-	NOMSGSUMMARY
NAME[('*external-name*')] | NONAME	N	NONAME

表 3. *Compile-time options, abbreviations, and IBM-supplied defaults (続き)*

Compile-time option	Abbreviated name	z/OS default
NAMES('*lower*'[,*upper*])	-	NAMES('#@$','#@$')
NATLANG(ENU | UEN)	-	NATLANG(ENU)
NEST | NONEST	-	NONEST
NOT	-	NOT('¬')
NULLDATE | NONULLDATE	-	NONULLDATE
OBJECT | NOOBJECT	OBJ | NOBJ	OBJECT
OFFSET | NOOFFSET	OF | NOF	NOOFFSET
OFFSETSIZE(n)[3]	-	OFFSETSIZE(4)
ONSNAP(STRINGRANGE, STRINGSIZE) | NOONSNAP	-	NOONSNAP
OPTIMIZE(0 | 2 | 3) | NOOPTIMIZE	OPT | NOPT	OPT(0)
OPTIONS[(ALL|DOC)] | NOOPTIONS	OP | NOP	NOOPTIONS
OR('c')	-	OR(' | ')
PP(*pp-name*) | NOPP	-	NOPP
PPCICS('string') | NOPPCICS	-	NOPPCICS
PPINCLUDE('string') | NOPPINCLUDE	-	NOPPINCLUDE
PPLIST(KEEP | ERASE)	-	PPLIST(KEEP)
PPMACRO('string') | NOPPMACRO	-	NOPPMACRO
PPSQL('string') | NOPPSQL	-	NOPPSQL
PPTRACE | NOPPTRACE	-	NOPPTRACE
PREFIX(*condition*)	-	See PREFIX
PRECTYPE(ANS | DECDIGIT | DECRESULT)	-	PRECTYPE(ANS)
PROCEED | NOPROCEED[(W | E | S)]	PRO | NPRO	NOPROCEED(S)
PROCESS[(KEEP | DELETE)] | NOPROCESS	-	PROCESS(DELETE)
QUOTE("")	-	QUOTE("")
REDUCE | NOREDUCE	-	REDUCE
RENT | NORENT	-	NORENT
RESEXP | NORESEXP	-	RESEXP
RESPECT([DATE])	-	RESPECT()

表 3. *Compile-time options, abbreviations, and IBM-supplied defaults (続き)*

Compile-time option	Abbreviated name	z/OS default
RTCHECK(NULLPTR | NONULLPTR | NULL370)	-	RTCHECK(NONULLPTR)
RULES(*options*)	-	See 70 ページの『RULES』
SEMANTIC | NOSEMANTIC[(W | E | S)]	SEM | NSEM	NOSEMANTIC(S)
SERVICE('*service string*') | NOSERVICE	SERV | NOSERV	NOSERVICE
SOURCE | NOSOURCE	S | NS	NOSOURCE
SPILL(n)	SP	SPILL(512)
STATIC(FULL | SHORT)	-	STATIC(SHORT)
STDSYS | NOSTDSYS	-	NOSTDSYS
STMT | NOSTMT	-	NOSTMT
STORAGE | NOSTORAGE	STG | NSTG	NOSTORAGE
STRINGOFGRAPHIC(CHAR | GRAPHIC)	-	STRINGOFGRAPHIC (GRAPHIC)
SYNTAX | NOSYNTAX[(W | E | S)]	SYN | NSYN	NOSYNTAX(S)
SYSPARM('string')	-	SYSPARM('')
SYSTEM(MVS | CICS | IMS | TSO | OS)	-	SYSTEM(MVS)
TERMINAL | NOTERMINAL	TERM | NTERM	
TEST(*options*) | NOTEST	-	See 91 ページの『TEST』[4]
UNROLL(AUTO | NO)		UNROLL(AUTO)
USAGE(*options*)	-	See 94 ページの『USAGE』
WIDECHAR(BIGENDIAN | LITTLEENDIAN)	WCHAR	WIDECHAR(BIGENDIAN)
WINDOW(w)	-	WINDOW(1950)
WRITABLE | NOWRITABLE[(FWS|PRV)]	-	WRITABLE
XINFO(*options*)	-	XINFO(NODEF,NOMSG, NOSYM,NOSYN,NOXML)
XML(CASE(UPPER | ASIS))	-	XML(CASE(UPPER))
XREF[(FULL | SHORT)(EXPLICIT | IMPLICIT)] | NOXREF	X | NX	NX [(FULL)][1]

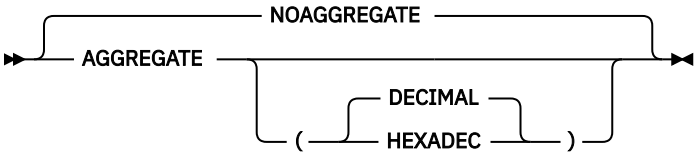
Notes :

1. FULL is the default suboption if the suboption is omitted with ATTRIBUTES or XREF.
2. The default value for the BLANK character is the tab character with value '05'x.
3. The OFFSETSIZE option is ignored if the LP(32) option is in effect.
4. (ALL,SYM) is the default suboption if the suboption is omitted with TEST.

以下のトピックでオプションをアルファベット順に説明します。 コンパイラーが情報をリストするように指定するオプションについては、簡単な説明のみが含まれています。 生成されるリストについて詳しくは、106 ページの『コンパイラー・リストの使用』を参照してください。

AGGREGATE

AGGREGATE オプションは、ソース・プログラム中の配列と主要な構造体の長さをコンパイラ・リストで示す集約長テーブルを作成する。



省略形: AG、NAG

AGGREGATE オプションのサブオプションは、サブ要素のオフセットを集約長テーブルにどのように表示するかを決定する：

DECIMAL
すべてのオフセットが 10 進数で表示されます。

HEXADEC
すべてのオフセットが 16 進数で表示されます。

Aggregate Length Table では、ディメンションなしのメジャー構造またはマイナー構造の長さは常にバイト単位で表されるが、メジャー構造またはマイナー構造にアラインされていないビット要素が含まれる場合、その長さは正確ではないかもしれない。

Aggregate Length Table には構造体は含まれるが、エクステントが一定でない配列は含まれない。 しかし、この構造体は非固定エクステントを保持し、構造体内部のエレメントのサイズとオフセットは、不正確であるか、または * として指定されます。

ARCH

ARCH オプションは、実行プログラムの命令を生成するアーキテクチャを指定する。 このオプションにより、最適化プログラムは特定のハードウェア命令セットの 利点を利用できます。

ARCH レベルには以下の値を指定できる：

12
z/Architecture モードの 3906-xxxx (IBM z14™) モデルで使用可能な命令を使用するコードを生成します。

具体的には、これらの ARCH(12)マシンとその後続マシンは、ベクトルパック 10 進数機能をサポートする命令を追加した。この機能は、中間結果をメモリではなくベクトルレジスタに格納することで、パック 10 進数計算とゾーン 10 進数計算を高速化するものである。

※13
z/Architecture モードの 8561-xxx (IBM z15™) モデルで使用可能な命令を使用するコードを生成します。

具体的には、これらの ARCH（13）機とその後続機は、以下の設備でサポートされる指示を追加する：

- ベクトル・パック 10 進数拡張機能
- ベクトル拡張機能 2
- その他の命令拡張機能 3
- 調整済みベクトル・ロード/保管のヒント

14

IBM z/Architecture モードの 3931-xxxx (IBM z16) モデルで使用可能な命令を使用するコードを生成します。

具体的には、このレベル 14 マシンとその後継マシンは、新しいベクトル・パック 10 進数拡張機能 2 によってサポートされる命令を追加します。 この新しい機能により、数字編集データ項目を使用するステートメントを含む PL/I のパフォーマンスが向上します。

15

IBM z/Architecture mode の 9175-xxxx (IBM z17) モデルで利用可能な命令を使用するコードを生成する。

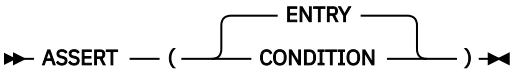
具体的には、この ARCH(15)マシンとその後続マシンは、新しいベクター・エンハンスメント・ファシリティ 3 とベクター・パックド・デシマル・エンハンスメント・ファシリティ 3 でサポートされる命令を追加する。 ベクトル拡張機能 3 は、16 バイト整数演算のインライン化を可能にする新しい命令を追加した。 Vector Packed Decimal Enhancement Facility 3 は、大きな FIXED BIN と FIXED DEC 値間の変換のパフォーマンスを改善します。

ノート：

1. ARCH に 12 未満の値を指定すると、コンパイラーはそれを 12 にリセットする。
2. 上記のモデル番号における x (2084-xxx など) はワイルドカードであり、そのタイプを持つ任意のマシン (英数字からなる) を表します。
3. 高い ARCH レベルには、低い方の ARCH レベルの機能が含まれます。 例えば、 ARCH(14)は、 ARCH の下位レベルのすべての施設を含んでいる。
4. 異なる ARCH レベルでコンパイルされたコードを、何の制限もなく混在させることができる。

ASSERT

ASSERT オプションは、 ASSERT ステートメントが、 ASSERTION 条件を発生させるデフォルト・ライブラリ・ルーチンを呼び出すか、ユーザーが提供するルーチンを呼び出すかを制御する。



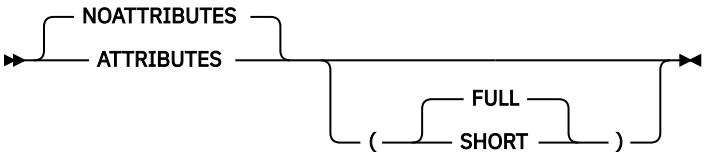
デフォルトは ASSERT （ENTRY)。

エントリー
ASSERT ステートメントが、 *Enterprise PL/I for z/OS Language Reference* に記述されているインターフェイスを持つユーザー提供のルーチンを呼び出すことを指定する。

条件
ASSERT ステートメントが、適切な ONCODE 組み込み関数で ASSERTION 条件を上げるライブラリ・ルーチンを呼び出すことを指定する。

ATTRIBUTES

ATTRIBUTES オプションは、コンパイラーがソース・プログラムの識別子とその属性の表をコンパイラー・リスティングに含めるように指定します。



省略形: A、NA、F、S

FULL
すべての ID と属性がコンパイラー・リストに組み込まれます。 FULL がデフォルトです。

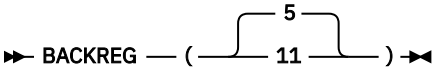
SHORT
非参照 ID が省かれ、リストの取り扱いが簡単になります。

ATTRIBUTES 、 XREF （相互参照表が作成される）の両方を含めると、2 つの表が結合される。 ただし、SHORT と FULL サブオプションが競合する場合は、最後に指定されたオプションが使用される。 例えば、ATTRIBUTES(SHORT) XREF(FULL) を指定すると、 FULL が複合リストに適用される。

BACKREG

BACKREG オプションは、バックチェーン・レジスタを制御します。バックチェーン・レジスタは、ネストされたルーチンが呼び出されたときに、親ルーチンの自動記憶装置のアドレスを渡すために使用されるレジスタです。

注 : LP(64)オプションでは、 BACKREG オプションは無視される。



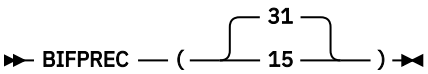
PL/I (MVS とは & VM 版)、 OS PL/I V2R3、およびそれ以前のコンパイラーとの互換性を高めるには、BACKREG(5) を使用すること。

ENTRY VARIABLE を共有するすべてのルーチンは、同じ BACKREG オプションでコンパイルされなければならない。アプリケーションのすべてのコードは、同じ BACKREG オプションでコンパイルされることが強く推奨される。

VisualAge PL/I for OS/390® でコンパイルされたコードは、実質的に BACKREG(11) オプションを使用することに注意。 Enterprise PL/I V3R1 または V3R2 でコンパイルされたコードも、デフォルトで BACKREG(11) オプションを使用する。

BIFPREC

BIFPREC オプションは、各種組み込み関数によって返される FIXED BIN 結果の精度を制御します。



PL/I (MVS とは & VM 版)、OS PL/I V2R3、およびそれ以前のコンパイラーとの最良の互換性を得るには、BIFPREC(15) を使用します。

BIFPREC は次の組み込み関数に影響します。

- COUNT
- 索引
- 長さ
- LINENO
- ONCOUNT
- PAGENO
- 検索
- SEARCHR
- SIGN
- VERIFY
- VERIFYR

BIFPREC コンパイラー・オプションの影響が最も明らかに見えるのは、上記の組み込み関数の結果の 1 つが、パラメーター・リストなしに宣言された外部関数に受け渡されるときです。 例えば、次のような部分コードがあるとします。

```
dcl parm char(40) var;
dcl funky ext entry( pointer, fixed bin(15) );
dcl beans ext entry;
call beans( addr(parm), verify(parm), ' ' );
```

実際に関数 beans が当該パラメーターを POINTER および FIXED BIN(15) として宣言するとします。 上記のコードがオプション BIFPREC(31) でコンパイルされている場合、および上記のコードが z/OS のようなビッグ・エンディアン・システム上で実行される場合、コンパイラーは 2 番目の引数として 4 バイトの整数を渡し、2 番目のパラメーターはゼロとなります。

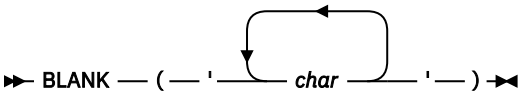
関数 funky は、すべてのシステム上でどちらのオプションでも機能することに注意してください。

BIFPREC オプションは、組み込み関数 DIM、HBOUND および LBOUND には影響しません。 CMPAT オプションは、これら 3 つの組み込み関数によって返される FIXED BIN 結果の精度を決定します。

- CMPAT(V1) では、これらの配列処理関数によって FIXED BIN(15) の結果が返されます。
- CMPAT(V2) および CMPAT(LE) では、FIXED BIN(31) の結果が返されます。
- CMPAT(V3) では、FIXED BIN(63) の結果が返されます。

BLANK

BLANK オプションは、空白文字の代替記号を 10 個まで指定する。



注 : 引用符と引用符の間にブランクをコーディングしないでください。

BLANK シンボルに対する IBM -supplied default code point は '05'X である。

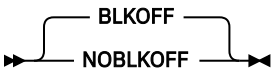
char
単一の SBCS 文字

英字も、数字も、「*PL/I Language Reference*」に定義されている特殊文字も指定できません。

BLANK オプションを指定しても、標準の空白記号は空白として認識される。

BLKOFF

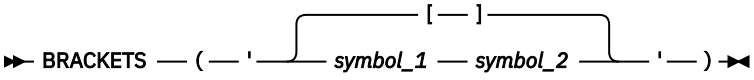
BLKOFF オプションは、疑似アセンブラー・リスト (LIST オプションによって生成される) とステートメント・オフセット・リスト (OFFSET オプションによって生成される) に示されるオフセットを、現行モジュールの開始位置を基準とするか、現行プロシージャーの開始位置を基準とするかを制御します。



疑似アセンブラー・リストには、現行モジュールの開始位置 からの各ブロックのオフセットも含まれています (それぞれのステートメントに表示されるオフセット を、ブロックまたはモジュールのオフセットに変換できるようにするため)。

BRACKETS

BRACKETS オプションは、SQL プリプロセッサが SQL 配列参照の左括弧と右括弧として受け付ける記号を指定します。



symbol_1
SQL 配列参照において左括弧として認識される記号を指定します。

symbol_2
SQL 配列参照において右括弧として認識される記号を指定します。

注 : 指定する 2 つの値は互いに異なるものでなければならず、 PL/I 文字セットや、 NAMES, NOT、 OR などの他の PL/I オプションで使用されている文字であってはならない。

デフォルトは BRACKETS('[]')。

関連情報

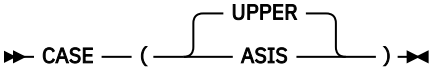
58 ページの『NAMES』
NAMES オプションは、識別子で許可される外国語文字を指定する。

59 ページの『NOT』
NOT オプションは、論理演算子 NOT として使える代替記号を 7 つまで指定する。

62 ページの『OR』
OR オプションは、論理演算子 OR として最大 7 つの代替記号を指定する。 これらの記号は連結演算子としても使われ、連続する 2 つの論理 OR 記号として定義されます。

CASE

CASE オプションは、一部の名前を大文字で表示するか、ソース・プログラムに表示されるのと同じ形式で表示するかを制御する。



デフォルトは CASE （UPPER)。

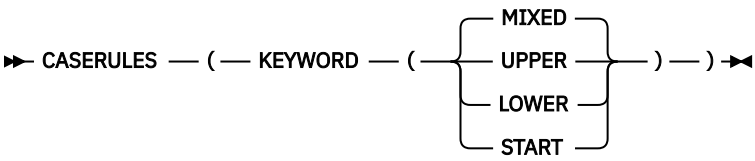
UPPER
すべての名前が大文字で表示されることを指定します。

ASIS
以下の対象において、ソースにおける大/小文字で名前が表示されることを指定します。

- リスト AGGREGATE、ATTRIBUTES、および XREF
- 組み込み関数 PACKAGENAME、PROCNAME、および ORDINALNAME によって返される値
- PUT DATA 出力 (TEST オプションが使用されていない場合、および GET DATA ステートメントで名前が使用されていない場合)

CASERULES

CASERULES オプションは、キーワードの大文字小文字ルールの適用を制御する。



LOWER
小文字になっていないキーワードにフラグを立てるようにコンパイラーに指示します。

MIXED
すべてのキーワードをコーディングされたままの状態で受け入れるようにコンパイラーに指示します。 MIXED がデフォルトです。

START
先頭文字が大文字になっていなかったり後続の文字が小文字になっていなかったりするキーワードにフラグを立てるようにコンパイラーに指示します。

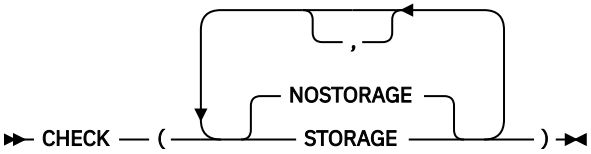
UPPER
大文字になっていないキーワードにフラグを立てるようにコンパイラーに指示します。

ノート:

1. CASERULES オプションは、 OPTIONS と ENVIRONMENT 属性の要素には適用されません。
2. CASERULES オプションは、どのプリプロセッサにも適用されない。

CHECK

CHECK オプションは、コンパイラがさまざまなプログラミング・エラーを検出するための特別なコードを生成するかどうかを指定する。



省略形: STG、NSTG

STORAGE | NOSTORAGE
CHECK(STORAGE) を指定すると、コンパイラーは ALLOCATE ステートメントと FREE ステートメントに対して多少異なるライブラリー・ルーチンを呼び出します (これらのステートメントが AREA 内で使用される場合を除く)。

注 : LP(64) オプションでは STORAGE サブオプションはサポートされていません。

「*PL/I Language Reference*」で説明されている次の組み込み関数は、CHECK(STORAGE) を指定した場合だけ使用できます。

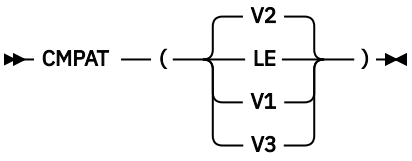
- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

AMODE(24) は、 Enterprise PL/I の用途には推奨されない。 CHECK(STORAGE) オプションでコンパイルしたコードで、AMODE(24) を使用しなければならない場合は、 HEAP(,,BELOW) ランタイム・オプションも指定しなければならない。

CMPAT

CMPAT オプションは、文字列、 AREAs、配列、または構造体を共有するプログラムに対して、 OS PL/I バージョン 1、 OS PL/I バージョン 2、 PL/I (MVS とは & VM 版)、または Enterprise PL/I for z/OS とのオブジェクト互換性を維持するかどうかを指定する。

注 : LP(64) オプションでは、 CMPAT オプションは無視される。事実上、 CMPAT(V3) は常にオンである。



ABBREVIATIONS: CMP

LE
CMPAT(LE) では、 VisualAge PL/I for OS/390 または Enterprise PL/I for z/OS でコンパイルされたプログラムとのみ、また、 CMPAT(V1) および CMPAT(V2) オプションがコンパイル時に使用されていない

場合に限り、プログラムは文字列、 AREAs、配列、または構造体を共有できます。 Db2 ストアド・プロシージャは (LE) でコンパイルしてはならない。 CMPAT

V1
CMPAT(V1) では、 CMPAT(V1) オプションが使用されている限り、 OS PL/I コンパイラでコンパイルされたプログラムや、それ以降の PL/I コンパイラでコンパイルされたプログラムと、文字列、 AREAs、配列、構造体を共有することができます。

V2
CMPAT(V2) では、 CMPAT(V2) オプションが使用されている限り、 OS PL/I コンパイラでコンパイルされたプログラムや、それ以降の PL/I コンパイラでコンパイルされたプログラムと、文字列、 AREAs、配列、構造体を共有することができます。

V3
CMPAT(V3) では、いずれかの CMPAT(V*) オプションが使用されている間は、OS PL/I コンパイラーでコンパイルされたプログラムや、それ以降の PL/I コンパイラーでコンパイルされたプログラムと、ストリングを共用できます。 ただし、CMPAT(V3) でコンパイルされなかったコードとは、AREA、配列、構造体を共用できません。

アプリケーション内のすべてのモジュールは、同じ CMPAT オプションでコンパイルされなければならない。

新旧のコードを混合する場合は、次の制限があります。 次の制限については、「*Enterprise PL/I for z/OS* コンパイラーおよびランタイム 移行ガイド」を参照してください。

DFT(DESCLIST)オプションは、 CMPAT(V*)オプションと競合する。 CMPAT(V*)オプションと共に指定された場合、メッセージが発行され、DFT(DESCLOCATOR)オプションが仮定される。

CMPAT(V3) では、配列は 8 バイト整数が取り得る任意の値で宣言できる。 しかし、LP(64)オプションが使用されない限り、現在のところ、配列の合計サイズは、 CMPAT(V2) で宣言された配列と同じ制限があります。

CMPAT(V3) では、以下の組み込み関数は常に FIXED BIN(63) の結果を返す：

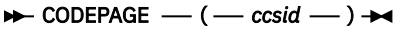
- CURRENTSIZE/CSTG
- DIMENSION
- HBOUND
- LBOUND
- LOCATION
- SIZE/STG

これらの関数は 8 バイトの整数値を返すので、CMPAT(V3) の下では、LIMITS オプションの FIXEDBIN サブオプションの 2 番目のオプションは 63 でなければならない。

ただし、 CMPAT(V3) の場合でも、ステートメントとフォーマット・ラベルの定数は 4 バイト整数で指定しなければならない。

CODEPAGE

CODEPAGE オプションは、CHARACTER と WIDECHAR 間、CHARACTER と UCHAR 間、EBCDIC と ASCII 間の変換に使われるコード・ページを指定する。



以下の表において、最初の数字は Euro Country Extended Code Page (ECECP) を、2 番目の数字は Country Extended Code Page (CECP) を表している。

サポート対象の CCSID	説明
01047	Latin 1/オープン・システム
01140、00037	米国、カナダなど
01141、00273	オーストリア、ドイツ

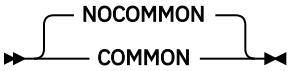
サポート対象の CCSID	説明
01142、00277	デンマーク、ノルウェー
01143、00278	フィンランド、スウェーデン
01144、00280	イタリア
01145、00284	スペイン、ラテンアメリカ (スペイン語)
01146、00285	英国
01147、00297	フランス
01148、00500	国際
01149、00871	アイスランド
01155, 01026	トルコ
00819	ISO 8859-1 ラテン 1
00813	ISO 8859-7 ギリシャ語
00920	ISO 8859-9 Latin 5

デフォルトの CCSID 1140 は、 CCSID 37（EBCDIC Latin-1, USA）と同等だが、ユーロ記号が含まれている。

COMMON

COMMON オプションは、EXTERNAL STATIC 変数用の CM リンケージ・レコード を生成するようコンパイラーに指示します。

注 : LP(64) オプションでは、COMMON オプションは無視されます。



COMMON オプションが指定されると、NORENT オプションが適用される場合に、INITIAL 値を含まない RESERVED 以外の EXTERNAL STATIC 変数に対して CM リンケージ・レコードが生成されます。 これは、OS PL/I コンパイラーが行う処理と一致します。

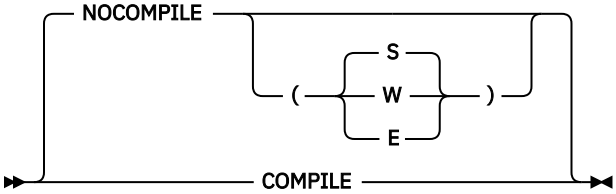
NOCOMMON オプションを指定すると、Enterprise PL/I の初期のリリースではそうであったように、SD レコードが書き込まれます。

共通オプションは、 *N* > 7 の場合、RENT オプションまたは LIMITS (EXTNAME (*N*)) と一緒に使用してはなりません。

COMPILE

COMPILE オプションを指定すると、前処理またはセマンティック・チェック中に指定された重大度のメッセージが生成された場合、ソース・プログラムのセマンティック・チェックがすべて終了した時点でコンパイラーがコンパイルを停止する。

コンパイラが続行するかどうかは、検出されたエラーの深刻度（以下のリストの NOCOMPILE オプションで指定）に依存する。 NOCOMPILE オプションは、セマンティック・チェックの後、無条件に処理を停止することを指定する。



省略形: C、NC

COMPILE
重大エラーまたは回復不能エラーが検出されない限り、コードを生成します。 このサブオプションは NOCOMPILE(S) と同等です。

NOCOMPILE
コンパイルはセマンティック検査後に停止されます。

NOCOMPILE(W)
警告、エラー、重大エラー、または回復不能エラーが検出された場合、コードは生成されません。

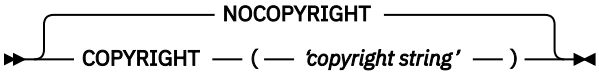
NOCOMPILE(E)
エラー、重大エラー、または回復不能エラーが検出された場合は、コードを生成しません。

NOCOMPILE(S)
重大エラーまたは回復不能エラーが検出された場合は、コードを生成しません。

NOCOMPILE オプションでコンパイルを終了すると、相互参照リストと属性リストが作成されます。

COPYRIGHT

COPYRIGHT オプションは、オブジェクト・モジュール内にストリングを 配置します。 このストリングは、オブジェクトのリンク先であるロード・モジュールとともに メモリーにロードされます。



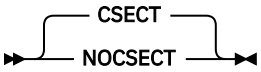
ストリングの長さは 1000 文字に制限されます。 ただし、ストリングは、100 文字を超える場合、オプション・リストに表示されません。

ロケールが異なっていてもストリングを読み取ることができるように、 インバリアント文字セットからの文字だけを使用する必要があります。

CSECT

CSECT オプションは、オブジェクトモジュールが生成された場合、名前付き CSECTs を含むことを保証する。

製品のサービスやプログラムのデバッグに SMP/E を使用する場合は、このオプションを使用してください。



省略形: CSE、NOCSE

NOCSECT オプションでは、オブジェクト・モジュールのコード・セクションとスタティック・セクションにデフォルトの名前が付けられます。

- NOCSECT オプションは、同じ静的セクション名やコード・セクション名を持つ可能性のあるオブジェクトをリンクする際に、バインダーのコンフリクトを防ぐためのものである。 これを行う唯一の方法は、バインダーが不平を言わないブランク名に置き換えることです。 オブジェクトを amblist で検査すると、静的セクション名とコード・セクション名の名前項目もブランクになるため、これは意図された動作です。

CSECT オプションでは、オブジェクト・モジュールのコード・セクションと静的セクションに、次のように定義される「パッケージ名」に依存する名前が付けられます：

- パッケージ・ステートメントが使用された場合は、「パッケージ名」はパッケージ・ステートメントの一番左のラベルになります。
- そうでない場合は、最初のプロシージャー・ステートメントにある一番左のラベルが「パッケージ名」になります。

長さ 7 の「変更パッケージ名」は、次のように形成されます。

- パッケージ名が 7 文字より短い場合は、アスタリスク (*) が前に追加されて、7 文字の長さの変更パッケージ名が作成されます。
- パッケージ名が 7 文字を超える場合、最初の *n* 文字と最後の *7 - n* 文字を使用して変更されたパッケージ名が作成されます。ここで、 *n* は CSECTCUT オプション。
- それ以外の場合は、パッケージ名がそのまま変更後のパッケージ名になります。

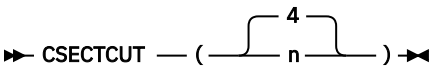
コード csect 名は、変更パッケージ名に 1 を追加して作成されます。

静的 csect 名は、変更パッケージ名に 2 を追加して作成されます。

そのため、SAMPLE というパッケージの場合、コード csect 名は *SAMPLE1 となり、静的 csect 名は *SAMPLE2 となります。

CSECTCUT

CSECTCUT オプションは、CSECT オプションの処理のときに、コンパイラーが、どのようにロング・ネームを処理するかを制御します。



CSECTCUT オプションは、CSECT オプションを指定しない限り影響しません。 CSECT オプションで使用される「パッケージ名」が 7 文字以下の場合も影響しません。

CSECTCUT オプションの中の値 *n* は、0 および 7 の間でなければいけません。

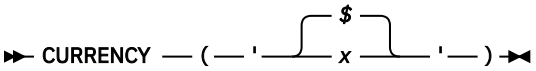
CSECT オプションで使用される「パッケージ名」が 7 文字を超える場合、コンパイラーはその名前を 7 文字に (最初の *n* 文字と最後の *7 - n* 文字を使用することによって) 縮小します。

例えば、BEISPIEL という名前を持つ 1 つのプロシージャーから成るコンパイルについて考えてみてください。

- CSECTCUT(3) では、コンパイラーはその名前を BEIPIEL に縮小します。
- CSECTCUT(4) では、コンパイラーはその名前を BEISIEL に縮小します。

CURRENCY

CURRENCY オプションでは、ドル記号の代わりに画像文字列で使用する代替文字を指定できる。

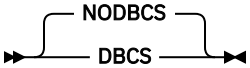


省略形: CURR

x
コンパイラーおよびランタイムがピクチャー・ストリング内でドル記号として認識し受け入れる必要がある文字

DBCS

DBCS オプションを指定すると、 GRAPHIC オプションが指定されていなくても、リストが生成される場合、DBCS が存在する可能性を考慮するようになる。



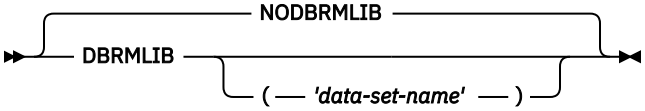
UNIX システムサービス z/OS UNIX システムサービスでは、 NODBCS オプションを指定すると、リストが作成された場合、 DBCS のシフトコードがすべて". "として表示される。

バッチおよび z/OS UNIX システム・サービスでは、 NODBCS オプションを使用すると、リスト・ページのヘッダー・テキストに一致しないシフト・コードが含まれないようになります。

GRAPHIC オプションも指定されている場合は、 NODBCS オプションを指定してはならない。

DBRMLIB

DBRMLIB オプションは、SQL プリプロセッサが z/OS UNIX System Services の下で実行するときに DBRM データセットを書き込む場所を指定します。



データセット名を指定せずに DBRMLIB オプションを指定すると、DBRM データセットが、ソース .pli ファイルと同じ名前で、拡張子 .dbrm を付けて、現在の UNIX System Services ディレクトリに書き込まれる。 例えば、次のコマンドを実行すると、

```
pli -c -qpp=sql -qdbrmlib sample.pli
```

DBRM データ・セットは sample.dbrm で作成されます。

DBRMLIB オプションがデータセット名とともに指定された場合、データセット名は Db2 で要求される属性を持つ PDS(E) の名前を指定しなければならない。 DBRM データ・セットは、ソース .pli ファイルと同じ名前を持つメンバーとしてのデータ・セットで作成されます。 例えば、次のコマンドを実行すると、

```
pli -c -qpp=sql -qdbrmlib="USER.TEST.DBRM" sample.pli
```

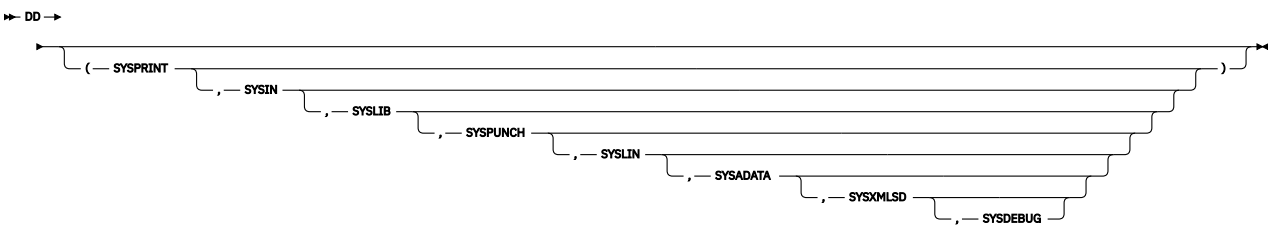
DBRM データ・セットは USER.TEST.DBRM(SAMPLE) で作成されます。

コンパイラが UNIX システム・サービスの SQL プリプロセッサを起動するときに、 DBRMLIB オプションを指定する必要があります。 このオプションは、コンパイラーが z/OS UNIX システム・サービスの下で SQL プリプロセッサーを呼び出す場合にのみ有効です。

デフォルトは NODBRMLIB です。

DD

DD オプションを使用すると、コンパイラが使用するさまざまなデータセットに代替の DD 名を指定することができる。



DD 名は 8 つまで指定できます。 その DD 名は順番に以下のデータ・セットに対して代替 DD 名を指定します。

- SYSPRINT
- SYSIN

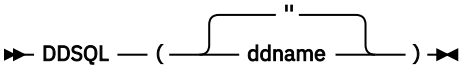
- SYSLIB
- SYSPUNCH
- SYSLIN
- SYSADATA
- SYSXMLSD
- SYSDEBUG

1 次コンパイラー・ソース・ファイルの DD 名として ALTIN を使用する場合は、DD(SYSPRINT,ALTIN) を指定する必要があります。 DD(ALTIN) を指定した場合は、SYSIN が 1 次コンパイラー・ソース・ファイルの DD 名として使用され、ALTIN がコンパイラー・リストの DD 名として使用されます。

アスタリスク (*) を使用して、デフォルト DD 名が使用されるように指示することもできます。 したがって、DD(*,ALTIN) は DD(SYSPRINT,ALTIN) と同等です。

DDSQL

DDSQL オプションを使用すると、SQL プリプロセッサーが EXEC SQL INCLUDE 文を解決する際に使用するデータ・セットの代替 DD 名を指定できます。

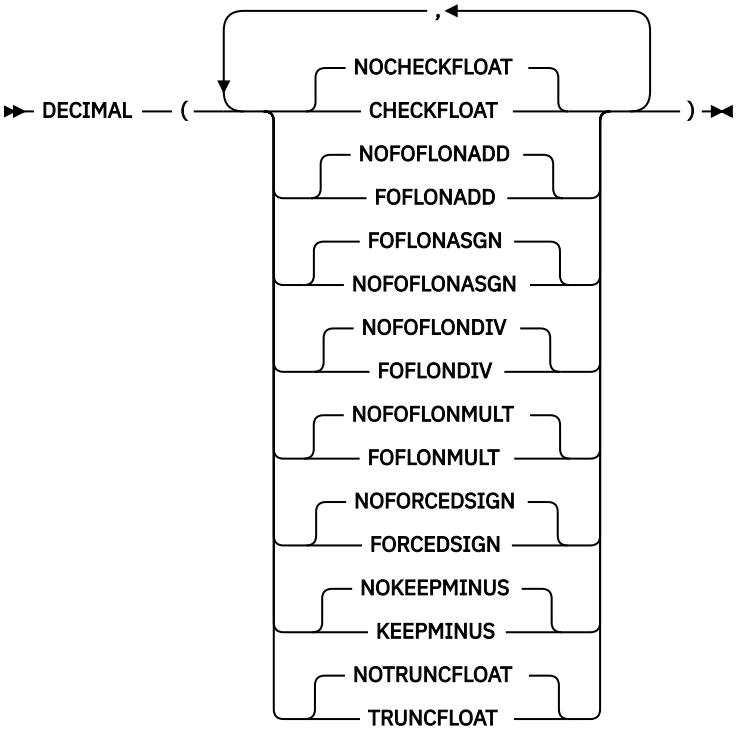


DDSQL('') オプションでは、 EXEC SQL INCLUDE 文の解決に使用される DD 名は DD コンパイラ・オプションの SYSLIB の DD 名です。

このオプションは、SQL プリコンパイラーから統合 SQL プリプロセッサーに移行する場合に役に立つ可能性があります。

10 進数

DECIMAL オプションは、特定の FIXED DECIMAL 演算および代入をコンパイラー が処理する方法を指定します。



CHECKFLOAT | NOCHECKFLOAT

DECIMAL (CHECKFLOAT) を指定すると、FLOAT が FIXED DECIMAL に変換されるときにソースが大きすぎないかどうかを検査するコードがコンパイラーによって生成されます。 大きすぎる場合は、SIZE が発生します。

以下のすべてが該当する場合は、プログラムにエラーがあり、結果が予測不能であることに注意してください。

- FLOAT から FIXED DECIMAL への変換で、ソースが大きすぎます。
- SIZE は有効ではありません。
- デシマル (チェックフロート) が有効でない

NOCHECKFLOAT がデフォルトです。

FOFLONADD | NOFOFLONADD

フロンアッド オプションは、組み込み関数で指定された精度に対して大きすぎる FIXED DEC 結果を生成する ADD 組み込み関数の使用に対して フィックスドオーバーフロー 条件を発生させるコードをコンパイラーが生成することを必要とします。

逆に、ノフロンアドオプションを指定すると、コンパイラーは、ADD 組み込み関数を使用した場合に切り捨てられた結果を生成するコードを生成します。

注 : フロンアッド オプションを使用すると、デフォルトの言語セマンティクスが変更されます (つまり、SIZE 条件が有効になっていない限り、FIXED DEC に適用された ADD 組み込み関数の結果が大きすぎます)。

デフォルトは NOFOFLONADD です。

FOFLONASGN | NOFOFLONASGN

FOFLONASGN オプションが有効になっていて SIZE 条件が無効になっている場合に、FIXED DECIMAL 式が FIXED DECIMAL ターゲットに代入されて有効数字が失われるときは必ず FIXEDOVERFLOW 条件を発生させるコードをコンパイラーが生成することを FOFLONASGN オプションは要求します。

反対に、NOFOFLONASGN オプションを指定すると、コンパイラーは、このような代入で有効数字 が失われたときに FIXEDOVERFLOW 条件を発生しないコードを生成します。

FIXED DEC ソース値を FIXED DEC ターゲットに割り当てるときに SIZE または FOFL が発生するかどうか (またはどちらも発生しないか) は、このオプションの設定、および SIZE 条件と FOFL 条件の有効化によって異なります。

FIXED DEC の割り当ての場合(p2,q2) FIXED DEC(p1,q1):

- (p1 - q1)> = (p2 - q2) の場合、SIZE は発生せず、FOFL は発生しません。
- (p1 - q1) の場合 < (p2 - q2)
  - SIZE が有効になっている場合、有効数字が失われると SIZE が発生し、FOFL が発生することはありません。
  - SIZE が使用不可の場合、SIZE は決して起こりません。
    - FEFL が有効で、オプション DEC (FOFLONASGN) が有効な場合、有効数字が失われると FOFL が発生します。
    - そうしないと、FOFL も発生しません。

KEEPMINUS オプションを使用すると、上記のロジックをオーバーライドできます。

デフォルトは FOFLONASGN です。

FOFLONDIV | NOFOFLONDIV

DIVIDE 組み込み関数が使用されていて、その組み込み関数で生成される FIXED DEC 結果が、DIVIDE 組み込み関数で指定された精度には大きすぎる場合に必ず FIXEDOVERFLOW 条件を発生させるコードをコンパイラーが生成することを FOFLONDIV オプションは要求します。

逆に、NOFOFLONDIV オプションでは、コンパイラーは、そのような DIVIDE 組み込み関数が使用されている場合に切り捨てた結果を生成するコードを生成します。

注 :

1. FOFLONDIV オプションを使用すると、デフォルト言語のセマンティクスが変更されます (このため、SIZE 条件が有効にされていない限り、FIXED DEC に適用された DIVIDE 組み込み関数の大きすぎる結果は切り捨てられます)。
2. NOFOFLONDIV オプションを指定した場合でも、LIMITS オプションの FIXEDDEC サブオプションで許可されるよりも桁数が多くなる商を生成する DIVIDE によって、例外が発生する可能性があります。

   例えば、FIXED DEC(5) と宣言されて値が 31415 の変数 A と、FIXED DEC(5) と宣言されて値が 10 の変数 B があり、LIMITS オプションの FIXEDDEC サブオプションが最大精度を 15 と指定しているとします。この場合は、以下のようになります。

   a. DIVIDE(A, B, 13, 11) は結果 41.5 を生成します。これは、FIXED DEC(15,11) の商 3141.5 が切り捨てられた値です。
   b. DIVIDE(A, B, 13, 12) は FOFL を発生させます。商 3141.5 が大きすぎて FIXED DEC(15,12) に収まらないためです。

デフォルトは NOFOFLONDIV です。

FOFLONMULT | NOFOFLONMULT

MULTIPLY 組み込み関数が使用されていて、その組み込み関数で生成される FIXED DEC 結果が、MULTIPLY 組み込み関数で指定された精度には大きすぎる場合に必ず FIXEDOVERFLOW 条件を発生させるコードをコンパイラーが生成することを FOFLONMULT オプションは要求します。

逆に、NOFOFLONMULT オプションでは、コンパイラーは、そのような MULTIPLY 組み込み関数が使用されている場合に切り捨てた結果を生成するコードを生成します。

注 : FOFLONMULT オプションを使用すると、デフォルト言語のセマンティクスが変更されます (このため、SIZE 条件が有効にされていない限り、FIXED DEC に適用された MULTIPLY 組み込み関数の大きすぎる結果は切り捨てられます)。

デフォルトは NOFOFLONMULT です。

FORCEDSIGN | NOFORCEDSIGN

FORCEDSIGN オプションが指定された場合、コンパイラーは強制的に追加コードを生成して、値ゼロの FIXED DECIMAL 結果が生成されるときは必ず結果の符号ニブルの値が 'C'X となるようにします。このオプションを指定すると、コンパイラーが生成するコードは、NOFORCEDSIGN サブオプション で生成されるコードよりもパフォーマンスが大幅に低下します。

また、FORCEDSIGN オプションが有効であると、コードを実行するときに、データ例外がより発生しやすくなります。 例えば、1 つの FIXED DEC(5) 変数を別の FIXED DEC(5) 変数に代入する場合、通常、コンパイラーは移動を実行するために MVC 命令を生成します。 ただし、FORCEDSIGN オプションが有効になっている場合は、結果の符号が望ましいものになるように、コンパイラーは移動を実行する ZAP 命令を生成します。 ソースに無効なパック 10 進データが含まれていると、MVC 命令ではなく、ZAP 命令が 10 進データ例外 を発生させます。

DECIMAL(NOFORCEDSIGN) を指定すると、ある種の計算では「負のゼロ」が生成されることがあります。

ただし、プログラマーは、DEC (KEEPMINUS) オプションを指定した場合にのみ、負のゼロを取得する必要があります。

デフォルトは NOFORCEDSIGN です。

KEEPMINUS | NOKEEPMINUS (キープマイナス)

KEEPMINUS を指定すると、コンパイラーは、場合によっては負のゼロを保持するための特別なコードを生成します。

特に、以下のすべての条件が満たされた場合、FOFL は決して発生せず、有効数字が失われる可能性があり、ソース内の負のゼロが保持されます。

- オプション DEC (KEEPMINUS) が有効である。
- オプション DEC (NOFORCEDSIGN) が有効である。

- ソースは参照であり、割り当てのターゲットである可能性があります
- ソースとターゲットは REAL です。
- 0 < = q < = p (ソースとターゲット)
- q1 <> q2
- SIZE が有効でないか、SIZE が発生できない。

このような状況では、上記の FOFLONASGN オプションの下で説明した規則は、この一連のテストがすべて真である場合には適用されません。

NOKEEPMINUS を指定すると、コンパイラーは負のゼロを保持するための特別なコードを生成しません。上記の FOFLONASGN サブオプションで説明されている規則は、代入でソースがターゲットに対して大きすぎる場合に SIZE、FOFL、または条件が発生しないかどうかを決定します。

KEEPMINUS オプションの使用は強く推奨されません。このオプションのロジックと効果を理解するのが難しく、コンパイラーがパフォーマンスの悪いコードを生成する可能性があるためです。 負のゼロを取得することに依存するコードを作成することは、非常に不適切なプログラミング手法です。

NOKEEPMINUS はデフォルトです。

TRUNCFLOAT | NOTRUNCFLOAT

トランスクフロートサブオプションは、切り捨てが発生する可能性がある場合に、16 進浮動小数点から固定小数点への割り当てを処理するようにコンパイラーに指示します。 このサブオプションは、IEEE FLOAT BINARY または IEEE FLOAT DECIMAL には適用されません。

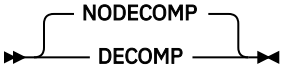
トランスクフロートでは、16 進浮動小数点値が FIXED DEC (p, q) (p < =18 および abs (q) < = p) に変換され、ソース値がターゲットに対して大きすぎる場合、ソース値は切り捨てられ、オーバーフローは発生しません。

NOTRUNCFLOAT のもとでは、ソースが大きすぎると、コードは無効になります。 固定小数点 (15,15) のターゲットの場合、ソースの絶対値が 1000 より小さいと、結果は OK になります。

デフォルトは NOTRUNCFLOAT です。

DECOMP

DECOMP オプションは、コンパイラに、コンパイルで使われた式の分解を与えるリストセクションを生成するように指示する。



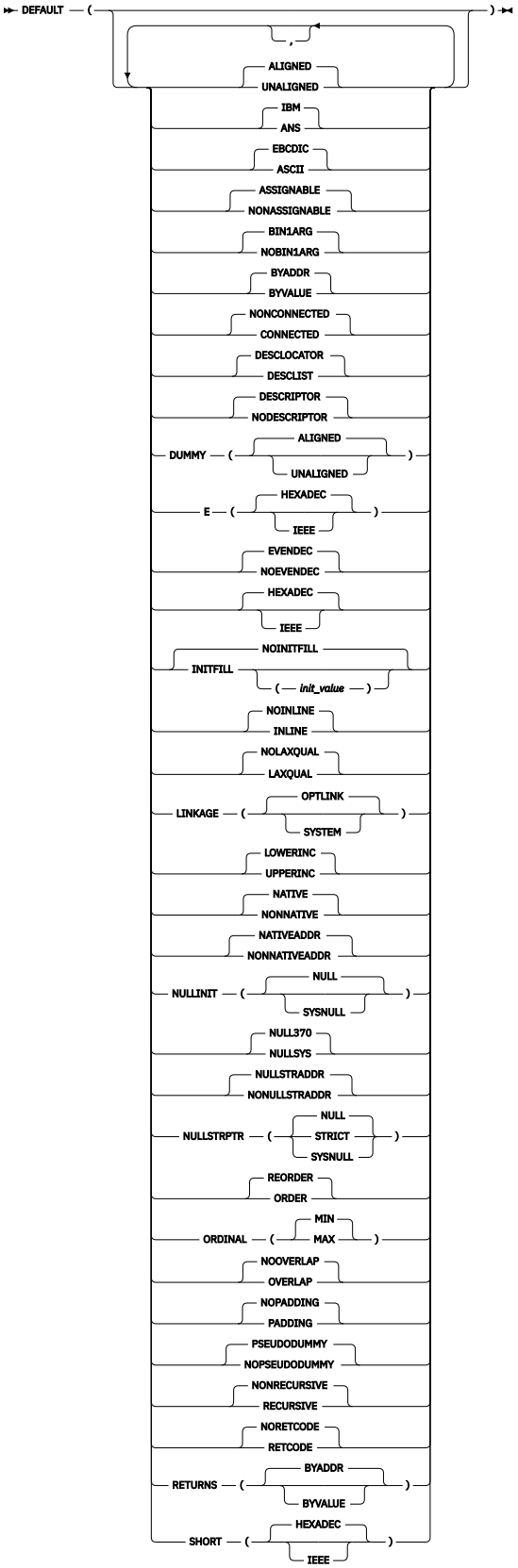
DECOMP オプションを指定すると、コンパイラーはソース・プログラムで使用されているすべての式の中間式とその属性を示すリスト・セクションを生成する。

NODECOMP オプションでは、コンパイラーはこのリスト・セクションを生成しない。

デフォルトは NODECOMP です。

DEFAULT

DEFAULT オプションは、属性とオプションのデフォルトを指定します。 これらのデフォルトは、属性またはオプションがソース・コードで明示的または 暗黙に指定されていない場合だけ適用されます。



省略形: DFT、ASGN、NONASGN、NONCONN、CONN、INL、NOINL

ALIGNED | UNALIGNED

このサブオプションは、ご使用のすべての変数に対してバイト位置合わせを強制します。

ALIGNED を指定すると、明示的に (おそらく親構造体で) または DEFAULT ステートメントにより暗黙に UNALIGNED 属性が指定されていない限り、 ピクチャー、文字、ビット、グラフィック、uchar、およびワイド文字 を除くすべての変数 に ALIGNED 属性が与えられます。

UNALIGNED を指定すると、明示的に (おそらく親構造体で) または DEFAULT ステートメントに より暗黙に ALIGNED 属性が指定されていない限り、すべての 変数に UNALIGNED 属性が与えられます。

ALIGNED がデフォルトです。

IBM | ANS
このサブオプションは、IBM のデフォルトを使用するのか ANS SYSTEM のデフォルトを使用するのかを指定します。 IBM および ANS の演算デフォルトを次の表に示します。

属性	デフォルト(IBM)	DEFAULT(ANS)
FIXED DECIMAL	(5,0)	(10,0)
FIXED BINARY	(15,0)	(31,0)
FLOAT DECIMAL	(6)	(6)
FLOAT BINARY	(21)	(21)

サブオプション IBM サブオプションでは、I から N で始まる名前の変数のデフォルトは FIXED BINARY で、それ以外の変数のデフォルトは FLOAT DECIMAL です。 ANS サブオプションを選択すると、すべての変数のデフォルトは FIXED BINARY になります。

IBM がデフォルトです。

ASCII | EBCDIC
このサブオプションは、問題プログラム文字データの内部表現に使用される文字セットのデフォルトを設定します。

ASCII 文字セットの照合順序に依存するプログラムをコンパイルする場合のみ、ASCII を指定してください。 例えば、プログラムが数字のソート・シーケンスまたは小文字および大文字の アルファベット順を使用している場合に、このような依存関係が存在します。 また、高位ビットの状態を変えて大文字の英字を作成するプログラムにも、 このような依存関係が存在します。

ASCII と EBCDIC の間のコンパイル時の変換では、 ASCII の 16 進値は、'20'x と ' 7e 'x の間でなければならない。

注 : コンパイラーは、A および E を文字ストリングの 接尾部としてサポートします。 A という接尾辞は、EBCDIC コンパイラー・オプションが有効な場合でも、ASCII データを表す文字列であることを示す。 あるいは、DEFAULT(ASCII)を選択した場合でも、 E という接尾辞は、文字列が EBCDIC であることを示す。

```
'123'A is the same as '313233'X
'123'E is the same as 'F1F2F3'X
```

EBCDIC がデフォルトです。

ASSIGNABLE | NONASSIGNABLE
このオプションを指定すると、コンパイラーは、ASSIGNABLE 属性 または NONASSIGNABLE 属性を持つものとして宣言されていないすべての 静的変数に、指定の属性を適用します。 コンパイラーは、NONASSIGNABLE 変数が割り当てのターゲットである ステートメントにフラグを付けます。

ASSIGNABLE がデフォルトです。

非譲渡性 サブオプションは、明示的な アソシエイト 属性または 非譲渡性 属性を持つ変数、あるいは親から アソシエイト 属性または非譲渡性属性を継承する構造メンバーには影響しません。

BIN1ARG | NOBIN1ARG
このサブオプションは、プロトタイプ化されていない関数に渡される 1 バイトの REAL FIXED BIN 引数を コンパイラーが処理する方法を制御します。

BIN1ARG が指定された場合、コンパイラーは、プロトタイプ化されていない関数に FIXED BIN 引数をそのまま渡します。

ただし、NOBIN1ARG の場合、コンパイラーは、プロトタイプ化されていない関数に 渡された 1 バイトの REAL FIXED BIN 引数を 2 バイトの一時的な FIXED BIN 引数に代入して、 代わりにその一時的な引数を渡します。

次の例を考えてください。

```
dcl f1 ext entry;
dcl f2 ext entry( fixed bin(15) );

call f1( 1b );
call f2( 1b );
```

DEFAULT(BIN1ARG) を指定した場合、コンパイラーは 1 バイトの FIXED BIN(1) 引数のアドレスをルーチン `f1` に渡し、2 バイトの FIXED BIN(15) 引数のアドレスをルーチン `f2` に渡します。 ただし、DEFAULT(NOBIN1ARG) を指定した場合、コンパイラーは 2 バイトの FIXED BIN(15) 引数のアドレスを両方のルーチンに渡します。

ルーチン `f1` が COBOL ルーチンの場合、COBOL では 1 バイトの整数をサポートしていないため、このルーチンに 1 バイトの整数の引数を渡すと問題が発生する可能性があることに注意してください。 この場合、DEFAULT(NOBIN1ARG) の使用が有用となることもありますが、エントリー宣言ステートメントで引数属性を指定する方がより良いと考えられます。

BIN1ARG がデフォルトです。

BYADDR | BYVALUE
このサブオプションは、引数やパラメーターを参照で渡すのか値で渡すのかについてのデフォルトを設定します。 BYVALUE は、いくつかの特定の引数およびパラメーターだけに 適用されます。 詳しくは、「*PL/I* 言語解説書」を参照してください。

BYADDR はデフォルトです。

CONNECTED | NONCONNECTED
このサブオプションは、パラメーターの接続または非接続に関するデフォルトを設定します。 CONNECTED を指定すると、パラメーターをレコード単位の入出力の ターゲットまたはソースとして、あるいはストリング・オーバーレイ定義の 基礎として使用できます。

NONCONNECTED がデフォルトです。

DESCLIST | DESCLOCATOR
DEFAULT(DESCLIST) を指定すると、コンパイラーはリストの中のすべての 記述子を「隠れた」最後のパラメーターとして渡します。

DEFAULT(DESCLOCATOR) を指定すると、以前のリリースの PL/I の場合と 同様に、記述子を必要とするパラメーターが記述子またはロケーターを使用して 渡されます。 これにより、古いコードは、あるルーチンから、ポインターを受け取ることを想定しているルーチンに構造体を渡す場合でも、引き続き機能できます。

DFT(DESCLIST) オプションは、CMPAT(V*) オプションと対立します。 DFT(DESCLIST) がいずれかの CMPAT(V*) オプションと一緒に指定されると、メッセージが発行されて DFT(DESCLOCATOR) オプションが想定されます。

DESCLOCATOR がデフォルトです。

DESCRIPTOR | NODESCRIPTOR
PROCEDURE ステートメントで DESCRIPTOR を使用すると、記述子リストが 渡されたことを示します。 ENTRY ステートメントで DESCRIPTOR を使用すると、 記述子リストを渡す必要があることを示します。 NODESCRIPTOR を使用すると、より効率的なコードになりますが、 次の制限があります。

- PROCEDURE ステートメントに関して、いずれかのパラメーターに次の項目が含まれている場合、NODESCRIPTOR は無効です。
  - 配列の境界、ストリングの長さ、または配列のサイズ (NONASSIGNABLE 属性 を持つ VARYING ストリングまたは VARYINGZ ストリングの場合を除く) に指定された アスタリスク (*)

  - NONCONNECTED 属性
  - UNALIGNED BIT 属性
- ENTRY 宣言では、ENTRY 記述リストの中で配列の境界、ストリングの長さ、 またはエリアのサイズにアスタリスク (*) が指定されている場合は、 NODESCRIPTOR は無効です。

DESCRIPTOR がデフォルトです。

DUMMY(ALIGNED | UNALIGNED)
このサブオプションは、仮引数が作成される状態の数を減らします。

DUMMY(ALIGNED) は、引数が位置合わせにおいてのみパラメーター と相違している場合にも、仮引数を作成するべきであることを示します。 DUMMY(UNALIGNED) は、スカラー (不変ビットを除く) または スカラーの配列が位置合わせにおいてのみパラメーターと相違している場合に、 そのスカラーまたはスカラーの配列には仮引数を作成してはならないことを示します。

次の例を考えてください。

```
dcl
  1 a1 unaligned,
    2 b1   fixed bin(31),
    2 b2   fixed bin(15),
    2 b3   fixed bin(31),
    2 b4   fixed bin(15);

dcl x entry( fixed bin(31) );

call x( b3 );
```

DEFAULT(DUMMY(ALIGNED)) を指定すると仮引数が作成されますが、DEFAULT(DUMMY(UNALIGNED)) を指定すると仮引数は作成されません。

DUMMY(ALIGNED) がデフォルトです。

E (HEXADEC | IEEE)
E サブオプションは、E フォーマット項目の指数として使用する数字の桁数を 指定します。

E(IEEE) を指定すると、E フォーマット項目の指数として 4 桁の数字が使用されます。

E(HEXADEC) を指定すると、E フォーマット項目の指数として 2 桁の数字が使用されます。

DFT(E(HEXADEC)) を 指定した場合に、99 より大きい絶対値を持つ指数が含まれる式を使用しようとすると、SIZE 条件が発生します。

コンパイラー・オプション DFT (IEEE) が有効な場合、通常はオプション DFT (E (IEEE)) も使用する必要があります。 ただし、このオプションのもとでは、DFT (E (HEXADEC)) のもとで有効な E フォーマット項目がいくつかあります。 は無効です。 例えば、DFT(E(IEEE)) を指定すると、E フォーマット項目が無効であるため、ステートメント `put skip edit(x) (e(15,8));` にフラグが立てられます。

E(HEXADEC) がデフォルトです。

EVENDEC | NOEVENDEC
このサブオプションは、偶数精度を指定して宣言された固定小数点変数 に関するコンパイラーの許容度を制御します。

NOEVENDEC のもとでは、固定小数点変数の精度は次の最大の奇数に 切り上げられます。 これにより、意図しない結果が生じる可能性があります。例えば、Db2 SQL は、時刻期間変数が FIXED DECIMAL (6) であることを予期し、DECIMAL (7) への NOEVENDEC プロモーションを拒否します。

EVENDEC を指定して FIXED DEC(2) 変数に 123 を割り当てると、SIZE 条件 が発生します。
NOEVENDEC を指定した場合は、SIZE 条件は発生しません。

EVENDEC がデフォルトです。

HEXADEC | IEEE
このサブオプションは、すべての FLOAT 変数およびすべての浮動小数点中間結果を保持するために使用されるデフォルト表現を指定します。 また、このサブオプションは、コンパイラーが浮動小数点の式を評価するときに、16 進の命令と数学ルーチンを使用するか、IEEE 浮動小数点の命令と数学ルーチンを使用するかを決定します。

JAVA とやり取りするプログラムに加え、浮動小数点データのデフォルト表現として IEEE を使用するプラットフォームとデータの受け渡しをするプログラムにも、IEEE オプションを使用することをお勧めします。

HEXADEC がデフォルトです。

INITFILL | NOINITFILL
このサブオプションは、自動変数のデフォルト初期化を制御します。

16 進値 (nn) を使用して INITFILL を指定した場合は、ブロックに入るたびに、 ブロック内のすべての自動変数によって使用されるストレージを初期化するために、その値が使用されます。 16 進値を入力しない場合、デフォルトは '00' です。

16 進値は引用符を付けても付けなくても指定できますが、引用符を付けて 16 進値を指定する場合は、そのストリングの末尾に X を付けてはならないことに注意してください。

NOINITFILL を指定した場合は、変数が明示的に初期化されない限り、 自動変数によって使用されるストレージには任意のビット・パターンを保持できます。

INITFILL は、プログラムの実行速度を低下させる可能性があるため、実動プログラムでは指定しないでください。 ただし、INITFILL オプションの生成するコードは、LE STORAGE オプション よりも高速に実行されます。 また、このオプションは、プログラム開発時に、未初期化自動変数を検出するために役立ちます。 DFT(INITFILL('00')) および DFT(INITFILL('ff')) を指定してプログラムが正しく実行されれば、未初期化自動変数は存在しないと考えられます。

NOINITFILL がデフォルトです。

INLINE | NOINLINE
このオプションは、インライン・プロシージャー・オプションの デフォルトを設定します。

INLINE を指定すると、コードの実行が高速になりますが、場合によっては 実行可能ファイルも大きくなります。 インライン化によるパフォーマンスの改善方法の詳細については、305 ページの『第 15 章 パフォーマンスの向上』を参照してください。

NOINLINE がデフォルトです。

LAXQUAL | NOLAXQUAL
このオプションは、コンパイラーが非修飾参照を解決する方法のデフォルトを設定します。

DEFAULT (NOLAXQUAL) options は、コンパイラーがドット修飾なしで名前への参照を解決するときに、最初に宣言されたブロック内で名前があいまいであるが、親ブロック内のスカラーへの参照である場合に、コンパイラーがその親ブロック参照になるように参照を解決することを指定します。

NOLAXQUAL オプションの使用は、どの PL/I 標準にも準拠していないため、推奨されません。

LAXQUAL がデフォルトです。

LINKAGE
プロシージャー呼び出しのためのリンケージ規則は次のとおりです。

OPTLINK
Enterprise PL/I のデフォルトのリンケージ規則。 このリンケージにより最良のパフォーマンスが得られます。

SYSTEM
システム API の標準リンケージ規則。

LINKAGE(OPTLINK) は、JAVA によって呼び出されるすべてのルーチンや、JAVA に対する呼び出しを行うすべてのルーチンに使用します。 また、LINKAGE(OPTLINK) は、C によって呼び出されるすべてのルーチンや、C に対して呼び出しを行うすべてのルーチンにも使用します (C コードがデフォルト以外のリンケージを使用してコンパイルされた場合を除く)。

LINKAGE(SYSTEM) は、最後 (および最後のみ) のパラメーターのアドレスにおいて高位ビットがオンになっていることを予期するすべての非 PL/I ルーチンに使用します。

PROCEDURE または ENTRY に OPTIONS(ASSEMBLER) を指定すると、このオプションの設定値に関係なく LINKAGE(SYSTEM) が強制されるので、注意してください。

LINKAGE(OPTLINK) がデフォルトです。

注 : LINKAGE サブオプションは、LP(64) オプションでは無視されます。

LOWERINC | UPPERINC
LOWERINC を指定した場合は、コンパイラーは INCLUDE ファイルの実際のファイル名が 小文字であることを要求します。 UPPERINC を指定した場合は、コンパイラーはこの名前が 大文字であることを要求します。

注 : このサブオプションは、z/OS UNIX 環境でのコンパイルにだけ適用されます。

z/OS UNIX では、インクルード名は拡張子 `.inc` で作成されます。 例えば、DFT (LOWERINC) オプションでは、ステートメント `%INCLUDE STANDARD;` を指定すると、コンパイラーは `standard.inc` を組み込もうとします。 しかし、DFT (UPPERINC) オプションを指定すると、ステートメント `%INCLUDE STANDARD;` によって、コンパイラーは `STANDARD.INC` を組み込もうとします。

LOWERINC はデフォルト値です。

NATIVE | NONNATIVE
このサブオプションは、固定 2 進数、序数、オフセット、エリア、および可変ストリング・データの内部表現だけに影響します。 NONNATIVE サブオプションが有効な場合、NONNATIVE 属性は、 NATIVE 属性を指定して宣言されていないこの種のすべての変数に適用されます。

NONNATIVE は、非ネイティブ・フォーマットに依存してこの種の変数を保持する プログラムをコンパイルする場合だけ指定してください。

ご使用のプログラムで固定 2 進変数がポインター変数またはオフセット変数に基づいている (あるいは、それとは反対にポインター変数またはオフセット変数が固定 2 進変数に基づいている) 場合は、次のサブオプションの組み合わせをどちらか指定します。

- NATIVE サブオプションと NATIVEADDR サブオプションの両方
- NONNATIVE サブオプションと NONNATIVEADDR サブオプションの両方

それ以外の組み合わせを指定すると、結果は予測できません。

NATIVE がデフォルトです。

NATIVEADDR | NONNATIVEADDR
このサブオプションが影響するのはポインターの内部表現のみです。 NONNATIVEADDR サブオプションが有効な場合、NONNATIVE 属性は、 NATIVE 属性を指定して宣言されていないすべてのポインター変数に適用されます。

ご使用のプログラムで固定 2 進変数がポインター変数またはオフセット変数に基づいている (あるいは、それとは反対にポインター変数またはオフセット変数が固定 2 進変数に基づいている) 場合は、次のサブオプションの組み合わせをどちらか指定します。

- NATIVE サブオプションと NATIVEADDR サブオプションの両方
- NONNATIVE サブオプションと NONNATIVEADDR サブオプションの両方

それ以外の組み合わせを指定すると、結果は予測できません。

NATIVEADDR がデフォルトです。

NULLINIT(SYSNULL | NULL)
DEFAULT (NULLINIT) オプションは、NULLINIT 属性を持つ POINTER に割り当てられた値を判別するのに役立ちます。

デフォルト(ヌルイニット(ヌル)), (ヌルイニット(ヌル))デフォルト (ヌルストラプトル (ヌル)), (ヌルストラプトル (ヌル))および DFT (NULL370) がすべて有効な場合、LP (32) では POINTER に 16 進値 'FF_00_00_00' が割り当てられ、LP (64) では '00000000_7FFFFBAD' が割り当てられます。

それ以外の場合、POINTER には値 SYSNULL () が割り当てられます。

デフォルトは NULLINIT (NULL) です。

NULLSYS | NULL370
このサブオプションは、NULL 組み込み関数によって戻される値を決定します。 NULLSYS を指定すると、binvalue(null()) は 0 に等しくなります。 NULL370 を指定した場合、LP (32) では binvalue (null ()) は 'FF_00_00_00' xn に等しく、LP (64) では '00000000_7FFFFBAD' xn に等しくなります。

NULL370 がデフォルトです。

NULLSTRADDR | NONULLSTRADDR
このサブオプションは、引数として渡されたヌル・ストリングをコンパイラーがどのように処理するのかを制御します。

NULLSTRADDR では、ヌル・ストリングが入り口呼び出しで引数として指定されていると、コンパイラーは自動ストレージの初期化部分のアドレスを渡します。 これは、OS PL/I および PL/I for MVS コンパイラーの動作と互換性があります。

ただし、NONULLSTRADDR を 指定した場合は、入り口の呼び出し時に引数としてヌル・ストリングが指定されると、コンパイラーによってヌル・ポインターが引数のアドレスとして渡されます。 これは、Enterprise PL/I コンパイラーの早期リリースの動作と互換性があります。

NULLSTRADDR がデフォルトです。

NULLSTRPTR
このサブオプションは、POINTER、HANDLE、および OFFSET に割り当てられたヌル・ストリングをコンパイラーがどのように処理するかを制御します。

NULLSTRPTR(SYSNULL) を指定すると、POINTER、HANDLE、または OFFSET に '' を割り当てた結果はポインターに SYSNULL() を割り当てたときと同じになります。

NULLSTRPTR(NULL) を指定すると、POINTER、HANDLE、または OFFSET に '' を割り当てた結果はポインターに NULL() を割り当てたときと同じになります。

NULLSTRPTR(STRICT) を指定した場合、POINTER、HANDLE、および OFFSET に対して '' を割り当てたり比較したりすると、無効のフラグが立てられます。

NULLSTRPTR(NULL) がデフォルトです。

ORDER | REORDER
このサブオプションは、オブジェクト・コードの最適化に影響します。 REORDER を指定すると、コードをより最適化できます。 詳しくは、305 ページの『第 15 章 パフォーマンスの向上』を参照してください。

REORDER がデフォルトです。

ORDINAL(MIN | MAX)
ORDINAL(MAX) を指定すると、定義に PRECISION 属性が含まれていない すべての順序数に、属性 PREC(31) が与えられます。 これを指定しない場合は、これらの順序数には、その値の範囲を カバーする最小の精度が与えられます。

ORDINAL(MIN) がデフォルトです。

OVERLAP | NOOVERLAP
OVERLAP を指定すると、コンパイラーは、割り当てでソースとターゲットのオーバーラップが起こることがあると想定し、割り当ての結果が正しくなるように、必要に応じて追加のコードを生成します。

OVERLAP サブオプションは、ストリング変数のみに適用されます。 これは FIXED DECIMAL またはその他の変数タイプの割り当てには効果がありません。 これらの割り当てでは、ソースとターゲットはオーバーラップしてはいけません。

NOOVERLAP を使用すると、パフォーマンスの優れたコードが生成されます。 ただし、NOOVERLAP を使用する場合は、ソースとターゲットがオーバーラップしないようにする必要があります。

NOOVERLAP がデフォルトです。

PADDING | NOPADDING
このサブオプションは、定義された構造体に埋め込みを行うかどうかを決定します。

DEFAULT(PADDING) を指定した場合、コンパイラーは DEFINE STRUCT ステートメント内のすべての構造体のサイズを切り上げて、構造体がその位置合わせの倍数を占めるようにします。

DEFAULT(NOPADDING) を指定した場合、コンパイラーはこの丸めを行いませんが、サイズが位置合わせの倍数でない定義された構造体にフラグを立てます。

DEFAULT(PADDING) をお勧めします。定義された構造体が C ルーチンに渡されるときは、これを使用する必要があります。

NOPADDING がデフォルトです。

PSEUDODUMMY | NOPSEUDODUMMY
このサブオプションは、プロトタイプ化されていない関数に対する引数として SUBSTR 参照が指定されたときに仮引数を作成するのかどうかを決定します。

PSEUDODUMMY を指定すると、プロトタイプ化されていない関数に対する引数として SUBSTR 参照が指定されたときに仮引数が作成されます。

NOPSEUDODUMMY を指定すると、プロトタイプ化されていない関数に対する引数として SUBSTR 参照が指定されたときに仮引数は作成されません。

PSEUDODUMMY がデフォルトです。

RECURSIVE | NONRECURSIVE
DEFAULT(RECURSIVE) を指定すると、コンパイラーはすべてのプロシージャー に RECURSIVE 属性を適用します。 DEFAULT(NONRECURSIVE) を指定すると、RECURSIVE 属性を持つプロシージャー 以外のすべてのプロシージャーが非再帰的になります。

NONRECURSIVE がデフォルトです。

RETCODE | NORETCODE
RETCODE が指定されると、コンパイラーは追加のコードを生成して、RETURNS 属性を持たない外部プロシージャーが PLIRETV 組み込み関数を呼び出すことにより取得した整数値を返してから、プログラムがそのプロシージャーから戻るようにします。

NORETCODE を指定すると、RETURNS 属性を持たないプロシージャーに対して 特別なコードは生成されません。

NORETCODE がデフォルトです。

RETURNS (BYVALUE | BYADDR)
このサブオプションは、関数が値を返す方法に関するデフォルトを設定します。 詳しくは、「*PL/I Language Reference*」を参照してください。

アプリケーションに ENTRY ステートメントが含まれていて、その ENTRY ステートメントまたはそのステートメントを含むプロシージャー・ステートメントに RETURNS オプションが指定されている場合は、RETURNS(BYADDR) を指定してください。 また、それらのエントリーのエントリー宣言でも、RETURNS(BYADDR) を 指定する必要があります。

RETURNS(BYADDR) がデフォルトです。

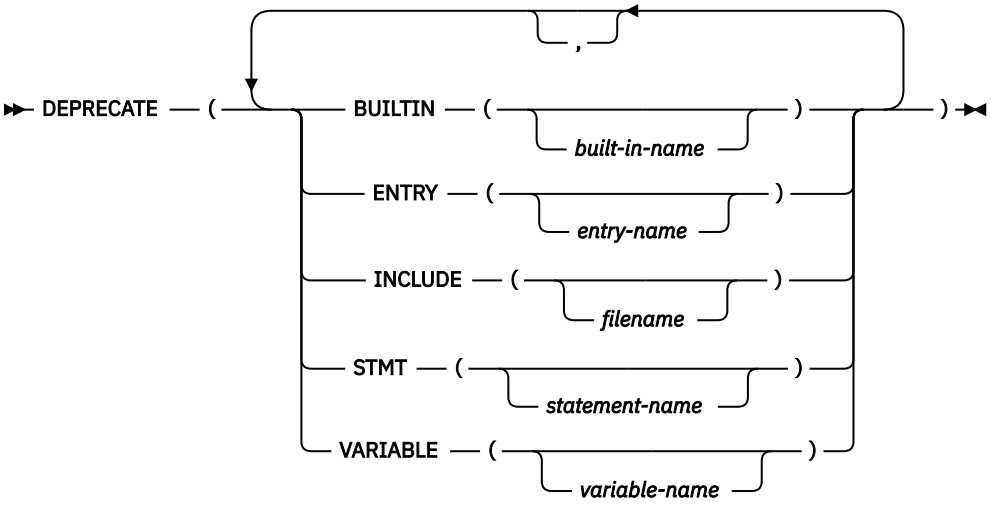
SHORT (HEXADEC | IEEE)
このサブオプションは、IBM 以外の UNIX コンパイラーとの互換性を 向上させます。 短数（16 進数）は、 *P* < = 21 の場合、FLOAT BIN (*P*) を短 (4 バイト) 浮動小数点数にマップします。 *P* < = 24 の場合、SHORT (IEEE) は FLOAT BIN (*P*) を短 (4 バイト) 浮動小数点数にマップします。

SHORT (HEXADEC) がデフォルトです。

デフォルト: DEFAULT(ALIGNED IBM EBCDIC ASSIGNABLE BIN1ARG BYADDR NONCONNECTED DESCLOCATOR DESCRIPTOR DUMMY(ALIGNED) E(HEXADEC) EVENDEC HEXADEC NOINITFILL NOINLINE LINKAGE(OPTLINK) LOWERINC NATIVE NATIVEADDR NULL370 NULLSTRPTR(NULL) NULLSTRADDR REORDER ORDINAL(MIN) NOOVERLAP NOPADDING PSEUDODUMMY NONRECURSIVE NORETCODE RETURNS(BYADDR) SHORT(HEXADEC))

DEPRECATE

このオプションは、非推奨の変数名、組み込みファイル名、およびステートメント名にフラグを立てて、エラー・メッセージを出力します。



BUILTIN
BUILTIN 属性を持つすべての *built-in-name* 宣言にフラグを立てます。

built-in-name
BUILTIN 変数の名前

ENTRY
ENTRY 属性を持つすべての *entry-name* 宣言にフラグを立てます。

entry-name
レベル 1 名

INCLUDE
filename を含むすべての %INCLUDE ステートメントにフラグを立てます。

filename
ファイルの名前

STMT
名前が *statement-name* であるすべてのステートメントにフラグを立てます。

statement-name
ステートメントの名前

この名前は、PL/I ステートメントの最初のキーワードによって識別されます。 STMT オプションは以下のキーワードを受け付ける：

- allocate
- assert
- attach
- begin
- call
- close
- delay
- delete
- detach
- display
- exit
- fetch

- flush
- free
- get
- goto
- iterate
- leave
- locate
- on
- open
- put
- read
- resignal
- revert
- rewrite
- signal
- stop
- wait
- write
- release

VARIABLE
BUILTIN または ENTRY 属性を持たない変数名の宣言にフラグを立てる。

variable-name
レベル 1 名

DEPRECATE オプションを指定するには、これらのサブオプションのうち少なくとも 1 つを、空のサブオプションリストに指定する必要がある。 例えば、次の 2 つの指定はどちらも無効です。

- DEPRECATE
- DEPRECATE(BUILTIN)

サブオプションの 1 つを指定しても、その前に指定された他のサブオプションの設定は変更されない。

サブオプションを複数回指定すると、前の指定が置き換えられます。

いずれの場合も、サブオプション・リストの検査は行われません。

デフォルト ： DEPRECATE(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())

例

- 以下の指定は同等です。

```
DEPRECATE(ENTRY(old)) DEPRECATE(BUILTIN(acos))
DEPRECATE(ENTRY(old) BUILTIN(acos))
```

- 次の例では、最初の指定の x は y に置き換えられます。

```
DEPRECATE(BUILTIN(x)) DEPRECATE(BUILTIN(y))
DEPRECATE(BUILTIN(y))
```

DEPRECATENEXT

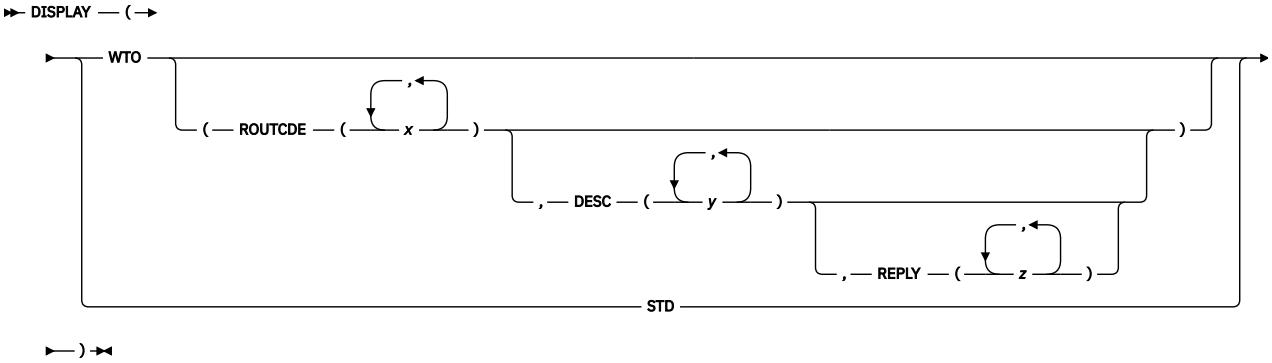
DEPRECATENEXT オプションの目的と使い方は、 DEPRECATE オプションと同じですが、将来の開発段階で非推奨となる項目については、コンパイラーがエラー・メッセージではなく警告メッセージを発行する点が異なります。

デフォルト ： DEPRECATENEXT(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())

詳しくは、を参照してください。

DISPLAY

DISPLAY オプションは、 DISPLAY ステートメントがどのように I/O を実行するかを決定する。



STD
DISPLAY ステートメントは、すべてテキストを stdout に 書き出し、REPLY テキストを stdin から読み込んで完了します。

WTO
REPLY が指定されない DISPLAY ステートメントはすべて WTO を使用して完了し、REPLY が指定された DISPLAY ステートメントはすべて WTOR を使用して完了します。 これがデフォルト値です。

次のサブオプションがサポートされています。

ROUTCDE
WTO で ROUTCDE として使用される 1 つ以上の値を指定します。 デフォルトの ROUTCDE は 2 です。

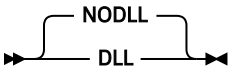
DESC
WTO で DESC として使用される 1 つ以上の値を指定します。 デフォルトの DESC は 3 です。

REPLY
WTOR で DESC として使用される 1 つ以上の値を指定します。 省略した場合は、DESC オプション (またはデフォルト) の値が使用されます。

ROUTCDE、DESC、および REPLY に指定される値はすべて 1 から 16 までの間の値でなければなりません。

DLL

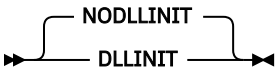
DLL オプションが有効な場合、コンパイラーは DLL 。 DLL 、関数や外部変数をエクスポートまたはインポートすることができる。 DLL オプションが指定されている場合は、 RENT オプションも指定しなければならない。



デフォルトは NODLL です。

DLLINIT

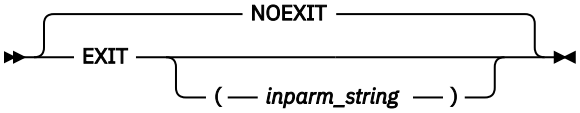
DLLINIT オプションは、 MAIN 以外のすべての外部手続きに OPTIONS(FETCHABLE) を適用する。 このオプションは、外部プロシージャーを 1 つだけ含むコンパイル単位でのみ使用し、そのプロシージャーは DLL としてリンクする。



NODLLINIT はあなたのプログラムに影響を与えない。

出口

EXIT オプションにより、コンパイラー・ユーザー出口を呼び出す ことができます。



inparm_string
初期化中にコンパイラー・ユーザー出口ルーチンに渡されるストリング。 ストリングの長さは最大 1023 文字です。

このオプションの使用方法について詳しくは、447 ページの『第 25 章 ユーザー出口の用法』を参照してください。

EXPORTALL

EXPORTALL オプションは、外部で定義されたすべてのプロシージャーと変数を DLL アプリケーションが使用できるようにエクスポートするかどうかを制御します。

EXPORTALL の下では、OPTION(DLLINTERNAL) を指定するものを除き、外部で定義されたすべての手続きと変数がエクスポートされます。

NOEXPORTALL の下では、OPTION(DLLEXTERNAL) または OPTIONS(FETCHABLE) を指定するものを除き、外部で定義された手続きや変数はエクスポートされません。

モジュールを（DLL 関数を通してアクセスするのではなく）単にフェッチされる DLL としてリンクする場合は、 NOEXPORTALL オプションを使用する必要があります。

デフォルトは EXPORTALL であるが、 NOEXPORTALL の方がパフォーマンス的には良い。

注 : XINFO(DEF) オプションがより有用な情報を生成するためには、 EXPORTALL オプションが有効でなければならない。

EXTRN

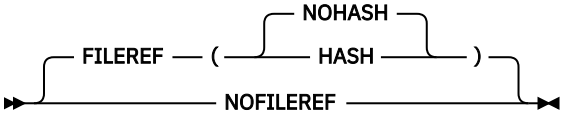
EXTRN オプションは、外部入り口定数の EXTRN を発行する 時期を制御します。

FULL
宣言されたすべての外部入り口定数に対して EXTRN を発行します。 これがデフォルト値です。

SHORT
参照されている定数に対してのみ EXTRN を発行します。

FILEREF

(NO)FILEREF オプションは、コンパイラーがファイル参照表を作成するかどうかを制御する。 NOFILEREF オプションは、指定された FLAG 設定に対してコンパイラがメッセージを発しない場合、リストからファイル参照テーブルを削除する。



FILEREF オプションの HASH サブオプションは、ファイル参照表の各行の後に、そのファイルの 256 ビット SHA-2 ハッシュの 16 進値を含む行が続くように指定する。

コンパイラーは、ファイルをオープンするときに SHA2INIT256 組み込み関数を呼び出して、このハッシュを実行する。 そして、その関数が返すトークンを使って、ファイルの各行を読み込んだ後に SHA2UPDATE256 組み込み関数を起動し、ファイルを閉じた後に SHA2FINAL256 組み込み関数を起動する。

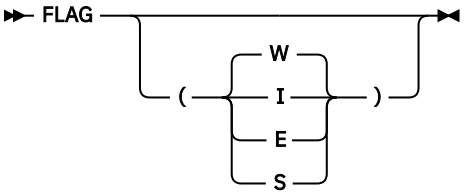
ファイルをハッシュ化する際、コンパイラーは右マージンより短い行を空白で埋め、幅を広げる。

このハッシュ値は、XINFO(XML(HASH))オプションが指定された場合に生成される xml ファイルに表示される値と同じである。

デフォルトは FILEREF(NOHASH)です。

FLAG

FLAG オプションは、コンパイラ・リストにメッセージを表示する必要があるエラーの最小重大度を指定する。



省略形: F

I
すべてのメッセージをリストします。

W
通知メッセージを除くすべてのメッセージをリストします。

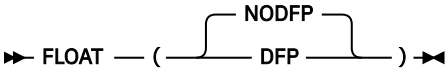
E
警告メッセージと通知メッセージを除くすべてのメッセージを リストします。

S
重大エラー・メッセージおよび回復不能エラー・メッセージだけを リストします。

指定した重大度を下回っているメッセージ、またはコンパイラー出口ルーチンにより フィルターに掛けられて取り除かれたメッセージは、リストされません。

FLOAT

FLOAT オプションは、追加の浮動小数点レジスタの使用と、10 進浮動小数点をサポートするかどうかを制御する。



DFP
DFP すべての DECIMAL FLOAT データは、マニュアルに記載されている DFP フォーマットで保持され、DECIMAL FLOAT を使用した操作は、マニュアルに記載されている DFP ハードウェア命令を使用して実行される。 操作の *z*/アーキテクチャー 原則 DECIMAL FLOAT データはすべて、マニュアルに記載されている DFP フォーマットで保持され、DECIMAL FLOAT を使用する操作は、マニュアルに記載されている DFP ハードウェア命令を使用して実行されます。

NODFP
DFP 。

FLOAT(DFP) では、以下のようになる：

- 拡張 DECIMAL FLOAT の最大精度は 34 (16 進浮動小数点と同様に 33 ではない) になります。
- 短精度 DECIMAL FLOAT の最大精度は 7 (16 進浮動小数点と同様に 6 ではない) になります。
- 以下のビルトインの DECIMAL FLOAT の値は、すべて適切に変更されます：
  - EPSILON
  - HUGE
  - MAXEXP
  - MINEXP
  - PLACES
  - RADIX
  - TINY
- 次の組み込み関数はすべて DECIMAL FLOAT の適切な値を返します (また、人間がはるかに容易に理解できる値を返します。例えば、SUCC(1D0) は 1.000_000_000_000_001 になり、ROUND 関数は小数点以下の桁で丸めます)。
  - EXPONENT
  - PRED
  - ROUND
  - SCALE
  - SUCC
- 10 進浮動小数点リテラルは、ゼロ以外の数字が小数点の後に続かないように、必要に応じて「右側ユニット表示」に変換された場合、つまり、指数が調整された場合 (例えば、3.1415E0 を 31415E-4 として表示する場合に行われるように)、当該リテラルの精度に対する正常な数値の範囲内にある指数を持つ必要があります。 これらの境界は、MINEXP-1 と MAXEXP-1 の値によって与えられる。 特に、次が保持される必要があります。
  - 短精度の浮動小数点の場合、 -95 <= 指数 <= 90
  - 長精度の浮動小数点の場合、 -383 <= 指数 <= 369
  - 拡張精度の浮動小数点の場合、 -6143 <= 指数 <= 6111
- DECIMAL FLOAT が CHARACTER に変換されると、文字列は指数として 4 桁を保持することになる（16 進数の浮動小数点では 2 桁が使用される）。
- IEEE および HEXADEC 属性は、FLOAT BIN に適用される場合のみ受け入れられ、DEFAULT(IEEE/HEXADEC) オプションは FLOAT BIN にのみ適用されます。
- 数学組み込み関数 (ACOS、COS、SQRT など) は、DECIMAL FLOAT 引数を受け入れ、対応する Language Environment 機能を使用してこの引数を評価します。 DECIMAL FLOAT 指数も同様に処理されます。
- 演算において一方のオペランドが FLOAT DECIMAL であり他方が 2 進 (つまり、FIXED BINARY、FLOAT BINARY、または BIT) である場合に DFP を使用するのであれば、その演算で行われる変換に留意する必要があります。 このような操作では、 PL/I の言語規則により、FLOAT DECIMAL オペランドを FLOAT BINARY に変換する必要があり、その変換にはライブラリ呼び出しが必要になる。 例えば、*A = A + B;* という形式の代入があるとします。 *A* は FLOAT DECIMAL であり、*B* は FIXED BINARY です。 この場合は、3 つの変換が行われ、そのうち 2 つはライブラリー呼び出しとなります。

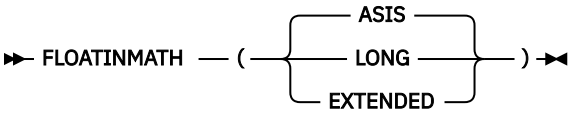
  1. *A* は、ライブラリー呼び出しにより、FLOAT DECIMAL から FLOAT BINARY に変換されます。
  2. *B* は、インライン・コードにより、FIXED BINARY から FLOAT BINARY に変換されます。
  3. *A + B* の和は、ライブラリー呼び出しにより、FLOAT BINARY から FLOAT DECIMAL に変換されます。

  DECIMAL 組み込み関数を使用することが有効であると考えられる状況: ステートメントが *A = A + DEC(B);* に変更された場合は、ライブラリー呼び出しが除去されます。 また、ライブラリー呼び出しは、*B* を FLOAT DECIMAL 一時変数に代入してから、その一時変数を *A* に追加しても、除去できます。

- 組み込み関数 SQRTF は、DECIMAL FLOAT 引数には対応していない（マッピングできるハードウェア命令がないため）。
- DFP は 型関数ではサポートされていない。 CAST

FLOATINMATH

FLOATINMATH オプションは、数学組み込み関数を呼び出すときに、コンパイラーが使用する精度を指定します。



ASIS
数学組み込み関数に対する引数が、long 型または拡張型浮動小数点精度を持つように強制されなくなります。

LONG
short 型浮動小数点精度の数学組み込み関数に対する引数が、最大 long 型浮動小数点精度に変換され、同じ最大 long 型浮動小数点精度の結果を出します。

EXTENDED
short または long 型浮動小数点精度の数学組み込み関数に対する引数が、最大拡張型浮動小数点精度に変換され、同じ最大拡張型浮動小数点精度の結果を生成します。

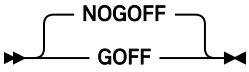
精度 *P* を持つ FLOAT DEC 式は、*P* < = 6 の場合は短精度浮動小数点精度を持ち、6 < *P* < = 16 の場合は長精度浮動小数点精度を持ち、 *P* > 16 の場合は拡張浮動小数点精度を持ちます。

精度 *P* を持つ FLOAT BIN 式は、*P* < = 21 の場合は短精度浮動小数点、21 < *P* < = 53 の場合は長精度浮動小数点精度、 *P* > 53 の場合は拡張浮動小数点精度を持ちます。

最大拡張型浮動小数点精度は、プラットフォームによって決まります。

GOFF

GOFF オプションは、一般オブジェクト・ファイル・フォーマット (GOFF) でオブジェクト・ファイルを生成するようにコンパイラーに指示します。



GOFF および OBJECT オプションが有効になっている場合、コンパイラーは、オブジェクト・ファイルを GOFF フォーマットで生成します。

NOGOFF および OBJECT オプションが有効になっている場合、コンパイラーは、オブジェクト・ファイルを XOBJ フォーマットで生成します。

GOFF フォーマットは、S/370 オブジェクト・モジュール・フォーマットおよび拡張オブジェクト・モジュール・フォーマットを置き換えるものです。 このフォーマットによって、以前のフォーマットのさまざまな制限 (例: 16 MB のセクション・サイズ) が除去され、ロング・ネームおよび長い属性のネイティブ z/OS サポートなど、多くの便利な拡張機能が提供されるようになりました。 GOFF は、XCOFF および ELF などの業界標準の一部の側面を取り込んでいます。

GOFF オプションを指定する場合は、バインダーを使用して出力を PDSE にバインドする必要があります。 GOFF は AMODE (24) プログラムと互換性がありません。

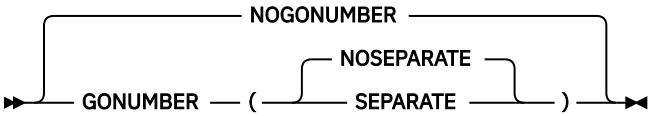
GOFF オプションでは、以下のオプションはサポートされていません。

- COMMON
- NOWRITABLE(PRV)

注 : GOFF およびファイル名が重複したソース・ファイルを使用した場合、リンカーは、エラーを出してコード・セクションの 1 つを破棄する可能性があります。 このような場合には、NOCSECT を指定して CSECT オプションをオフにしてください。

GONUMBER

GONUMBER オプションは、ソース・プログラムの行番号を実行時メッセージに含めるための追加情報をコンパイラーが生成することを指定する。



省略形: GN、NGN

SEPARATE
TEST(SEPARATE) オプションが指定されている場合、生成されたステートメント番号テーブルを別のデバッグ・ファイルに置きます。 GONUMBER(SEPARATE) を使用すると、ステートメント番号を実行時メッセージに含めることができなくなります。

NOSEPARATE
生成されたステートメント番号テーブルがオブジェクト・デックに入れられます。

あるいは、オフセット・アドレスを使用して行番号を派生させることもできます。 オフセット・アドレスは、ランタイム・メッセージにも、OFFSET オプションで生成されるテーブルまたは LIST オプションで生成されるアセンブラー・リストにも、常に含まれています。

GONUMBER の使用は TEST オプションの ALL および STMT サブオプション により強制されます。

なお、GOSTMT オプションは存在しない。 エラーが発生した場所を特定する情報を実行時に生成する唯一のオプションは、 GONUMBER 。 GONUMBER オプションを使用すると、ランタイム・エラー・メッセージに含まれる「ステートメント (statement)」という語は、STMT オプションが有効な場合でも、NUMBER コンパイラー・オプションで使用される行番号を参照します。

TEST(SEPARATE) を指定せずに GONUMBER(SEPARATE) オプションを指定すると、GONUMBER(NOSEPARATE) に変更される。

TEST と NOGONUMBER の両方を指定すると、NOGONUMBER オプションは GONUMBER(NOSEPARATE) に変更されます。

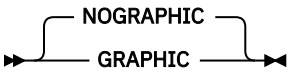
デフォルトは NOGONUMBER です。

互換性のため、GONUMBER オプションを指定した場合のデフォルト・サブオプションは SEPARATE になります。

GRAPHIC

GRAPHIC オプションは、ソース・プログラムが全角文字を含むことができることを指定する。

16 進コード「0E」はシフトアウト制御コードとして扱われ、「0F」はシフトイン制御コードとして扱われます (コメント内やストリング定数内も含めて、ソース・プログラム内のどこにこれらのコードが現れるのかは関係ありません)。



省略形: GR、NGR

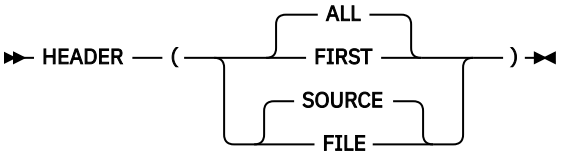
GRAPHIC オプションをつけると、コンパイルに使われた STREAM ファイルにも GRAPHIC ENVIRONMENT オプションが適用される。

ソース・プログラムが以下のいずれかを使用している場合、GRAPHIC オプションを指定しなければならない :

- DBCS ID
- 漢字ストリング定数
- 混合ストリング定数
- ソース内のどこか別の場所にあるシフト・コード

HEADER

HEADER オプションを使うと、コンパイラー・リストの各ヘッダー行の真ん中 のどこに、どのような種類の情報（ソース・ファイル名または *PROCESS ステートメ ントの後の最初の行）を表示するかを制御できる。



コンパイラ・リストの各ヘッダー行の中央に表示される情報のタイプ：

- HEADER(FILE)を指定すると、コンパイラーはソース・ファイル名を使用する。
- HEADER(SOURCE)を指定すると、コンパイラは、 *PROCESS ステートメントの後の最初のソース行のテキストを使用するようになる。

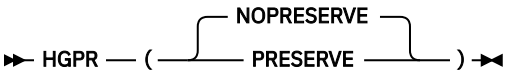
どのヘッダー行に、コンパイラー・リスティングに表示する情報を記述するか：

- HEADER(ALL)を指定すると、コンパイラーはそのテキストをコンパイラー・リストのすべてのヘッダー行に配置する。
- HEADER(FIRST)を指定すると、コンパイラーはそのテキストを、コンパイラー・リストの最初のヘッダー行だけに入れる。

デフォルトは HEADER(SOURCE ALL)。

HGPR

HGPR オプションは、コンパイラが関数で使用する 64 ビット GPR の上位半分を保存し、復元するかどうかを指定します。

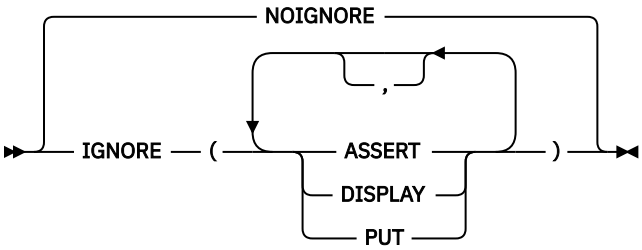


PRESERVE
PRESERVE は、関数のプロローグに保存し、エピローグに復元することにより、関数用 64 ビット GPR の上位半分を保持するようにコンパイラーに指示します。 PRESERVE サブオプションは、呼び出し元が Enterprise PL/I および/または z/OS XL C /C++コンパイラーに生成されたコードであることがわからない場合にのみ必要です。

NOPRESERVE
NOPRESERVE を使用すると、コンパイラーは、関数用 64 ビット GPR の上位半分の保持を省略できます。 パフォーマンスの考慮から、HGPR のデフォルトのサブオプションは、NOPRESERVE になっています。 HGPR は、「64 ビット GPR 上位半分」を意味します。これは、ネイティブ 64 ビット命令の使用を意味します。 特に、アプリケーションが 8 バイト整数を使う場合、ネイティブの 64 ビット命令から受益するべきです。

IGNORE

IGNORE オプションは、ASSERT ステートメント、DISPLAY ステートメント、および PUT ステートメントを無視するかどうかを制御します。 ステートメントを無視するというのは、ステートメントをセミコロンで置き換えるようなものです。



ASSERT
コンパイラーはすべての ASSERT ステートメント (そのステートメントに含まれているすべての関数参照を含む) を無視します。

表示
コンパイラーは、ステートメントに組み込まれているすべての関数参照を含め、すべての DISPLAY ステートメントを無視します。

PUT
コンパイラーはすべての PUT FILE ステートメントを無視します。

INCAFTER

INCAFTER オプションでは、ソース・プログラムの中で特定のステートメント の後に組み込むファイルを指定します。

filename
最後の PROCESS ステートメントの後に組み込むファイルの名前

現在、PROCESS は唯一のサブオプションであり、最後の PROCESS ステートメント の後に組み込むファイルの名前を指定します。

次の例を考えてください。

```
INCAFTER(PROCESS(DFTS))
```

この例は、ソースの最後の PROCESS ステートメントの後にステートメント %INCLUDE DFTS; を指定するのと同等です。

INCDIR

INCDIR コンパイラー・オプションは、組み込みファイルの検索に使用される検索パスに追加されるディレクトリーを指定します。

directory name
組み込みファイルを検索するディレクトリーの名前。 INCDIR オプションを複数回指定できます。その場合、ディレクトリーは 指定された順序で検索されます。

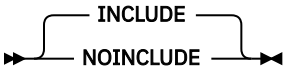
バッチの場合を除いて、コンパイラーは次の順序で INCLUDE ファイルを探します。

1. 現行ディレクトリー
2. –I フラグまたは INCDIR コンパイラー・オプションで指定された ディレクトリー
3. /usr/include ディレクトリー
4. INCPDS コンパイラー・オプションで指定された PDS

バッチ環境では、このオプションは DFT(LOWERINC) オプションと併用されることが最良と考えられ、このオプションは %include x; 形式の組み込みステートメントにのみ影響します。 このような組み込みステートメントの場合、x.inc という名前の zFS ファイルは、このオプションで指定されたディレクトリーで最初に検索されます。 zFS ファイルが見つからない場合、x は SYSLIB DD で指定された PDS(E) のメンバーであるはずです。 %include dd(x);の形式の組み込みステートメントの場合、zFS ファイルは組み込まれません。メンバー x は常に、指定された dd によって指定された PDS のメンバーでなければなりません。

INCLUDE

INCLUDE コンパイラー・オプションは、コンパイラーの最終パスが %INCLUDE ステートメントと %XINCLUDE ステートメントを処理するかどうかを制御します。



INCLUDE
マクロ・プリプロセッサーと、コンパイラーの最終パスの両方が、%INCLUDE ステートメントと %XINCLUDE ステートメントを処理します。

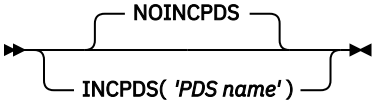
NOINCLUDE
マクロ・プリプロセッサーのみが %INCLUDE ステートメントと %XINCLUDE ステートメントを処理します。

INCLUDE がデフォルトです。

INCPDS

INCPDS オプションは、z/OS UNIX においてプログラムをコンパイルするときにコンパイラーが組み込むファイルがある PDS を指定します。

注 : このオプションは、z/OS UNIX 環境でのコンパイルにだけ適用されます。



PDS name
組み込まれるファイルがある PDS の名前。

INCPDS オプションを複数回指定することもでき、その場合、コンパイラーは INCPDS オプションを指定した順番に各名称の PDS を検索する。

例えば、SOURCE.PLI という PDS からプログラム TEST をコンパイルする場合に、PDS SOURCE.INC に含まれる組み込みファイルを使用するときは、次のコマンドを指定します。

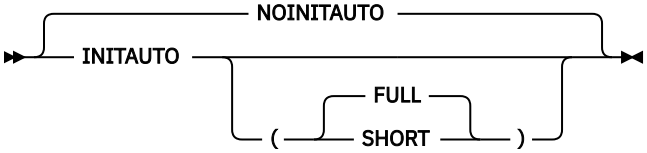
```
pli -c -qincpds="SOURCE.INC" "//'SOURCE.PLI(TEST)'"
```

コンパイラーは次の順序で組み込みファイルを検索します。

1. 現行ディレクトリー
2. –I フラグまたは INCDIR コンパイラー・オプションで指定された ディレクトリー
3. /usr/include Directory
4. INCPDS コンパイラー・オプションで指定された PDS

INITAUTO

INITAUTO オプションは、INITIAL 属性を指定せずに宣言されたすべての AUTOMATIC 変数に、INITIAL 属性を追加するようコンパイラーに指示します。



INITAUTO(FULL) を指定すると、コンパイラーは、INITIAL 属性を持たないすべての AUTOMATIC 変数に、そのデータ属性に従って以下の INITIAL 属性を追加します。

- INIT((*) 0) - データ属性が FIXED または FLOAT の場合
- INIT((*) '') - データ属性が PICTURE、CHAR、BIT、GRAPHIC、UCHAR または WIDECHAR の場合
- INIT((*) SYSNULL()) - データ属性が POINTER または OFFSET の場合
- INIT((*) NULLENTRY()) - データ属性が ENTRY の場合

コンパイラーは、その他の属性を持つ変数には INITIAL 属性を追加しません。

完全に初期化されていない AUTOMATIC 変数 (ただし、DFT(INITFILL) オプションと違って、これらの変数は意味のある初期値を持つようになりました) が入っている各ブロックごとに、プロローグの中により多くのコードが INITAUTO により生成されるよう になり、パフォーマンスに悪い影響を与えることになります。

INITAUTO オプションは、NOINIT 属性を指定して宣言された変数に対しては、INITIAL 属性を適用しません。

INITAUTO(SHORT) が指定されると、コンパイラーは、INITIAL 属性を持たない AUTOMATIC 変数に INITIAL 属性を追加します (ただし、この変数もスカラーであり、次のいずれかの属性を持つ場合に限られます)。

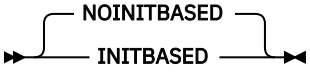
- POINTER
- OFFSET
- FIXED BIN
- FLOAT
- NONVARYING BIT
- NONVARYING CHAR(1)
- NONVARYING UCHAR(1)
- NONVARYING WCHAR(1)

INITAUTO(FULL) の場合と同様、追加された INITIAL 属性はデータ・タイプに適合します。

ランタイム STORAGE オプションを使用してすべてのストレージをゼロ設定しても、最適化プログラムが未初期化の AUTOMATIC 変数、特に、レジスターに合せて最適化されている同変数に対して不要なコードを生成することがあります。 上にリストしたデータ・タイプを持つスカラー変数はレジスターに最適化される可能性がある変数です。したがって、INITAUTO(SHORT) を使用した方が DFT(INITFILL) を使用するよりパフォーマンスへの影響が少ない場合があります。ただし、後者は最適化プログラムにあいまいなコードを残します。 さらに、コードが正しいのは、すべての変数が使用前に明示的に初期化された場合に限られます。

INITBASED

INITBASED オプションは、INITIAL 属性を指定せずに宣言されたすべての BASED 変数に、INITIAL 属性を追加するようコンパイラーに指示します。



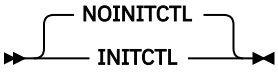
このオプションは、BASED 変数に対してであることを除き、INITAUTO と同じ機能を実行します。

INITBASED オプションにより、完全に初期化されていない BASED 変数の ALLOCATE に対して、より多くのコードが生成されるようになり、パフォーマンスに悪い影響を与える ことになります。

INITBASED オプションは、NOINIT 属性を指定して宣言された変数に対しては、INITIAL 属性を適用しません。

INITCTL

INITCTL オプションは、INITIAL 属性を指定せずに宣言されたすべての CONTROLLED 変数に、INITIAL 属性を追加するようコンパイラーに指示します。



このオプションは、CONTROLLED 変数に対してであることを除き、INITAUTO と同じ機能を実行します。

INITCTL オプションにより、完全に初期化されていない CONTROLLED 変数の ALLOCATE に対して、より多くのコードが生成されるようになり、パフォーマンスに悪い影響を与える ことになります。

INITCTL オプションは、NOINIT 属性を指定して宣言された変数に対しては、INITIAL 属性を適用しません。

INITSTATIC

INITSTATIC オプションは、INITIAL 属性を指定せずに宣言されたすべての STATIC 変数に、INITIAL 属性を追加するようコンパイラーに指示します。

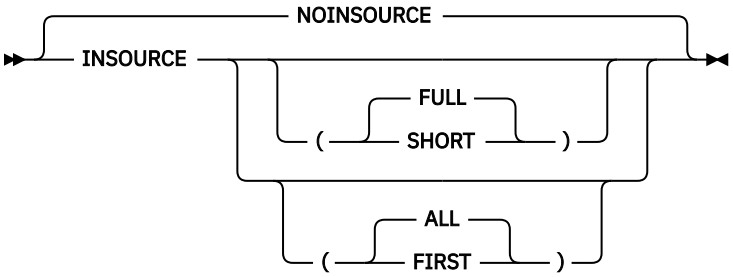
このオプションは、STATIC 変数に対してであることを除き、INITAUTO と同じ機能を実行します。

INITSTATIC オプションでは、一部に大きいオブジェクトを作成したり、長いコンパイルを作成する 可能性はありますが、それ以外はパフォーマンスに影響を与えることはありません。

INITSTATIC オプションは、NOINIT 属性を指定して宣言された変数に対しては、INITIAL 属性を適用しません。

INSOURCE

INSOURCE オプションは、コンパイラが PL/I マクロの前にソース・プログラムのリストを含めることを指定します、 CICS または SQL プリプロセッサがそれを翻訳する前に、コンパイラがソース・プログラムのリストを含めることを指定します。



省略形: IS、NIS

FULL
INSOURCE のリストは%NOPRINT 文を無視し、プリプロセッサが翻訳する前のすべてのソースを含む。

FULL がデフォルトです。

SHORT
INSOURCE リストは、%PRINT と %NOPRINT ステートメントに従う。

ALL
INSOURCE リストには、各プリプロセッサーとコンパイラー自身によって生成されるソースリストが含まれる。 ALL はデフォルトです。

FIRST
INSOURCE リストには、最初のプリプロセッサによって生成されたソース・リストのみが含まれる。

INSOURCE オプションでは、プログラムのロジックに従うのではなく、各ファイルが読み込まれるたびにテキストがリストに含まれる。 例えば、次のような単純なプログラムについて検討してみます。 このプログラムでは、PROC ステートメントと END ステートメントの間に %INCLUDE ステートメントがあります。

```
insource: proc options(main);
  %include member;
end;
```

INSOURCE のリストには、member に含まれるテキストの前に、メイン・プログラムがすべて含まれる (そして、 に含まれるテキストの前に、そのファイルがすべて含まれる)。

INSOURCE(SHORT) オプションでは、%INCLUDE ステートメントによって含まれるテキストは、%INCLUDE ステートメントが実行されたときに有効だった印刷/非印刷ステータスを継承しますが、その印刷/非印刷ステータスは、含まれるテキストの最後に復元されます (ただし、 SOURCE リストでは、印刷/非印刷ステータスは、含まれるテキストの最後には復元されません)。

INTERRUPT

INTERRUPT オプションを指定すると、コンパイル済みプログラムが アテンション要求 (割り込み) に応答します。

省略形: INT、NINT

このオプションにより、コンパイル済み PL/I プログラムが対話式システム の下で実行されるときのアテンション割り込みの効果が決まります。 このオプションは、TSO の下で実行されるプログラムに対してだけ効果があります。 ATTENTION 条件の発生に依存するプログラムを作成した場合、 そのプログラムは INTERRUPT オプションを使って コンパイルしなければなりません。 このオプションを使用すると、アテンション割り込みをプログラミングの 不可欠な部分にすることができます。 この方法により、プログラムを対話式で大幅に制御できます。

INTERRUPT オプションを指定すると、アテンション割り込みが生じた場合、 確立されている ATTENTION ON ユニット が制御を得ます。 ATTENTION ON ユニットの実行が完了すると、GOTO ステートメントで別の場所が指定されていない限り、制御は割り込み点に戻ります。 ATTENTION ON ユニットを確立していないと、 アテンション割り込みは 無視されます。

NOINTERRUPT を指定すると、プログラムの実行時のアテンション割り込みで、ATTENTION ON ユニットに制御が渡ることはありません。

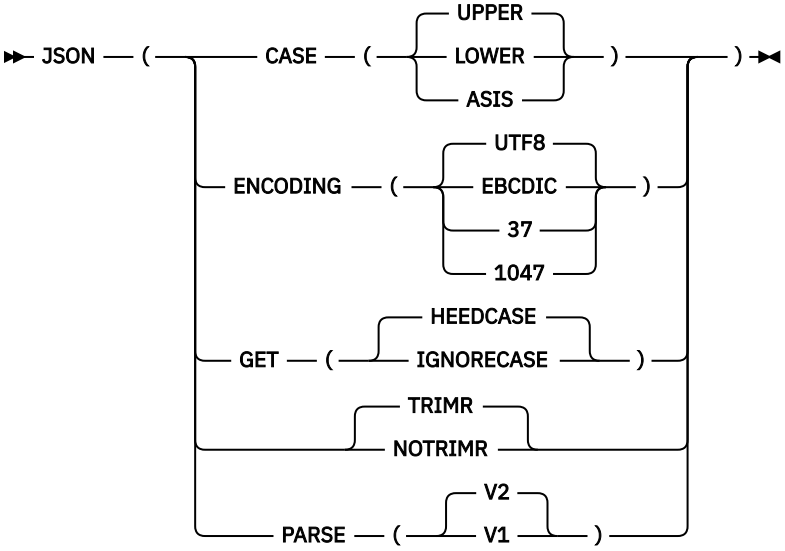
テストの目的だけにアテンション割り込み機能が必要な 場合は、INTERRUPT オプションの代わりに TEST オプションを使用してください。

関連情報
91 ページの『TEST』
TEST オプションは、コンパイラがオブジェクト・コードの一部として生成するテスト機能のレベルを指定します。 このオプションを使用すれば、テスト・フックの位置を制御したり、記号テーブルを生成するかどうかを制御したりできます。

441 ページの『割り込みとアテンションの処理』

JSON

JSON オプションを使用すれば、JSONPUT 組み込み関数によって生成されたり JSONGET 組み込み関数によって予期されたりする JSON テキスト内の名前の大/小文字を選択できます。



CASE(UPPER | LOWER | ASIS)

CASE (UPPER)サブオプションでは、 JSONPUT 組み込み関数によって生成され、JSONGET 組み込み関数によって期待される JSON テキスト内の名前は、すべて大文字になります。

CASE (LOWER)サブオプションでは、 JSONPUT 組み込み関数によって生成され、JSONGET 組み込み関数によって期待される JSON テキスト内の名前は、すべて小文字になります。

CASE (ASIS)サブオプションでは、 JSONPUT 組み込み関数によって生成され、JSONGET 組み込み関数によって期待される JSON テキスト内の名前は、その宣言で使用される大文字と小文字になります。 マクロ・プリプロセッサー・オプション CASE(ASIS) を使用せずに MACRO プリプロセッサーを使用すると、コンパイラーによって参照されるソースでは名前がすべて大文字になるため、JSON(CASE(ASIS)) オプションを指定しても無意味です。

デフォルトは CASE(UPPER) です。

ENCODING(UTF8 | EBCDIC | 37 | 1047)

ENCODING(UTF8) サブオプションの下で、 JSONPUT と JSONGET 関数の JSON テキストのコードページは UTF-8 です。 これはデフォルトのサブオプションであり、標準準拠の JSON に対してのみ有効な選択肢である。

ENCODING(EBCDIC)サブオプションの下では、JSONPUT と JSONGET 関数の JSON テキストのコード・ページは、CODEPAGE コンパイラー・オプションで指定されたものと同じです。 このサブオプションは、CODEPAGE オプションで EBCDIC コードページを指定した場合のみ有効である。

ENCODING(37)サブオプションで、 JSONPUT 、JSONGET 関数の JSON テキストのコード・ページは 37 になる。 このサブオプションは、CODEPAGE オプションがコードページ 1047 を指定している場合にのみ有効である。

ENCODING(1047)サブオプションでは、JSONPUT と JSONGET 関数の JSON テキストのコード・ページは 1047 です。 このサブオプションは、CODEPAGE オプションがコードページ 37 を指定している場合にのみ有効である。

GET(HEEDCASE | IGNORECASE)

GET(HEEDCASE)サブオプションの下では、 JSONGET 組み込み関数は、正確に一致する名前を要求する。

GET(IGNORECASE)サブオプションの下で、 JSONGET 組み込み関数は、大文字と小文字が無視される場合、一致する名前と見なす。

デフォルトは GET(HEEDCASE) です。

PARSE(V1 | V2)

オプションの JSON(PARSE(V1)) :

- JSON ソースがターゲット宣言よりも多くの値を配列に指定した場合、ERROR 条件が発生します（予想されたときに閉じ括弧] が見つからなかったことを報告します）
- 第 3 引数が構造体の場合、JSON テキスト内の名前は構造体内の名前と一致しなければならない。 そうでない場合は、ERROR 条件が発生します。

注 : このオプションでは、構造体内のすべての要素に名前と値のペアを指定する必要はないが、指定される名前は構造体内の順番と同じでなければならない。

コンパイラー・オプションの JSON(PARSE(V2))：

- JSON ソースがターゲット宣言よりも多くの値を配列に指定した場合：
  - SUBSCRIPTRANGE が有効な場合、 SUBSCRIPTRANGE コンディションが発生する
  - そうでなければ、余分な値は無視される
- 第 3 引数が構造体で、JSON テキストの名前が構造体のどの名前とも一致しない場合：
  - CONFORMANCE が有効な場合、CONFORMANCE 条件が発生し、ONJSONNAME 組み込み関数はマッチしない名前を返す
  - そうでない場合、名前とその JSON 値は無視されます
- JSON の null は、構造体や配列の値として受け入れられる。

注 : このオプションでは、構造体内のすべての要素に名前と値のペアを指定する必要はなく、構造体内の順序と同じ順序で名前を指定する必要もない。

デフォルトは PARSE(V2)。

TRIMR | NOTRIMR

JSON(TRIMR) オプションの下では、 JSONPUTVALUE および JSONPUTMEMBER 関数からの出力は、右からトリミングされます。

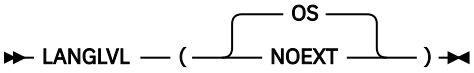
JSON(NOTRIMR) オプションの下では、 JSONPUTVALUE および JSONPUTMEMBER 関数からの出力は、右からトリミングされない。

構造内のフィールドに JSONTRIMR 属性が指定されている場合、その属性はそのフィールドの JSON (NOTRIMR) オプションをオーバーライドします。

デフォルトは TRIMR。

LANGLVL

LANGLVL オプションは、コンパイラに受け入れさせたい PL/I 言語定義のレベルを指定する。



NOEXT
以下の ENVIRONMENT オプションのみ受け付ける：

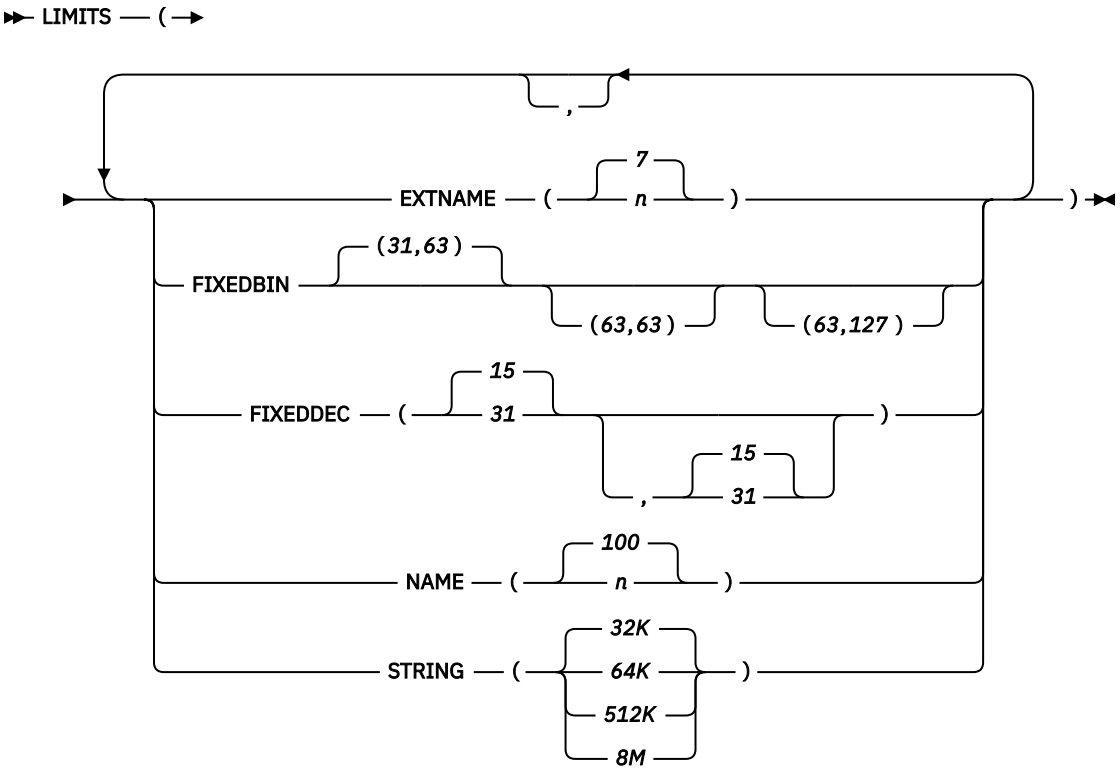
- Bkwd
- Consecutive
- Ctlasa
- Deblock
- Genkey
- Graphic
- Indexed
- Keylength
- Keyloc
- Organization

- Recsize
- Regional
- Relative
- Scalarvarying
- Vsam

OS
すべての ENVIRONMENT オプションが使用できます。 ENVIRONMENT オプションの全リストは 215 ページの表 15 を参照。

LIMITS

LIMITS オプションでは、各種のインプリメンテーションの制限を指定します。



EXTNAME
EXTERNAL 名の最大長を指定します。 *n* の最大値は 100、最小値は 7 です。

FIXEDDEC
FIXED DECIMAL の最大精度として 15 または 31 のいずれかを指定します。 デフォルトは FIXEDDEC(15,31) です。

FIXEDDEC(15,31) を指定した場合は 15 桁を超える桁数で FIXED DECIMAL 変数を宣言できますが、15 桁を超える桁数のオペランドが式に含まれていない限り、コンパイラーはすべての算術演算に対して最大精度として 15 を使用します。

FIXEDDEC(15,31) が指定された場合は、FIXEDDEC(31) が指定された場合よりもパフォーマンスがはるかに向上します。

FIXEDDEC(15) と FIXEDDEC(15,15) は等価であり、同様に FIXEDDEC(31) と FIXEDDEC(31,31) も等価です。

FIXEDDEC(31,15) は指定できません。

FIXEDBIN
SIGNED FIXED BINARY の最大精度を 31、63、127 のいずれかに指定する。 デフォルトは (31,63) です。

指定できるオプションは 3 つある：

- **fixedbin(31,63)：** これを指定すると、8 バイト整数を宣言できるが、式に 8 バイト整数が含まれていない限り、コンパイラーはすべての整数演算に 4 バイト整数を使用する。
- **固定ビン(63)**
- **fixedbin(63,127)：** これが指定されている場合、16 バイト整数を宣言することができるが、式に 16 バイト整数が含まれていない限り、コンパイラーはすべての整数演算に 8 バイト整数を使用する。

注 : FIXEDBIN(31,63)は、FIXEDBIN(63)や FIXEDBIN(63,127)よりもはるかに良いパフォーマンスを提供する。

注 : FIXEDBIN(31,63)または FIXEDBIN(63)オプションを指定すると、コンパイラはデータ型が混在する式に対して 8 バイト整数演算を使用する可能性がある。 例えば FIXED BIN(31) の値が FIXED DEC(13) の値に加算される場合、コンパイラーは FIXED BIN の結果を生成し、LIMITS(FIXEDBIN(31,63)) が指定されていると、その結果の精度は 31 より大きくなります (FIXED DEC の精度が 9 より大きいため)。 この状況が発生すると、コンパイラーは通知メッセージ IBM2809 を発行します。

注 : FIXEDBIN(63,31)、FIXEDBIN(31,31)、FIXEDBIN(31,127) は使用できません。

16 バイト整数演算の性能は、ARCH レベルが 15 未満では著しく悪化する。

UNSIGNED FIXED BINARY の最大精度は 1 つ大きい、つまり 32、64、128 である。

名前
プログラムの中の変数名の最大長を指定します。 *n* の最大値は 100、最小値は 31 です。

STRING
BIT、CHARACTER、GRAPHIC、UCHAR、または WIDECHAR 変数の長さのしきい値として、これらの値を受け入れる： 32K 64K,,. 512K 8M 次の表は、これらの値の対応する制限と、有効になっていなければならない必須オプションを示しています。

値	対応する制限 (しきい値)	有効になっていなければならない必須オプション
32k	32767	
64k	65535	BIFPREC(31)
512k	524287	BIFPREC(31) および CMPAT(V3)
8M	8388607	BIFPREC(31) および CMPAT(V3)

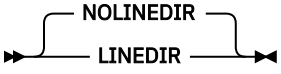
注 :

- デフォルト値は 32K です。
- VARYING に指定できる最大値は 64K です。
- STRING(64K) を使用するには BIFPREC(31) オプションが有効になっていなければなりませんが、CMPAT(V3) オプションは不要です。

STRING(64K) オプションは、接頭部が 2 バイト長のままになるにしても、(接頭部が 4 バイト長の VARYING4 ストリングとは異なり) VARYING ストリングがより長いことを許可します。 ただし、STRING 制限が 64K より大きい場合は、長さが 64K より小さい CHAR VARYING 変数のみ SQL VARCHAR として使用できます。

LINEDIR

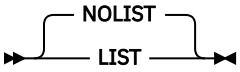
LINEDIR オプションは、コンパイラが %LINE ディレクティブを受け付けるように指定します。



LINEDIR オプションが指定された場合、コンパイラーは %INCLUDE ステートメントをすべて拒否する。 LINEDIR オプションが指定された場合、コンパイラーは TEST オプションの SEPARATE サブオプションの使用も拒否する。

LIST

LIST オプションは、コンパイラが擬似アセンブラ・リストを生成するように指定する。



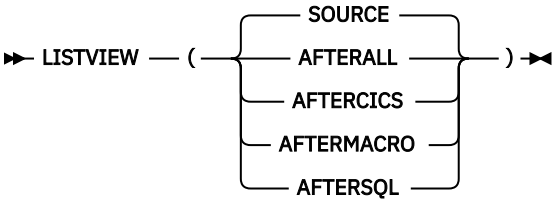
LIST オプションを指定すると、コンパイルに必要な時間と領域が増える。 OFFSET 、 MAP のオプションは、より安価に必要な情報を提供できる。

各ブロックごとに、疑似アセンブラー・リストも、リストの終わりに、 全コンパイル単位の開始位置からそのブロック内の最初の命令までのオフセットで組み込まれます。

LISTVIEW

LISTVIEW オプションは、コンパイラがソースをソース一覧に表示するか、1 つ以上のプリプロセッサで処理された後のソースを表示するかを指定します。

NOSOURCE オプションが有効な場合、LISTVIEW オプションは無視される。 プリプロセッサの 1 つに深刻なエラーがある場合、 LISTVIEW オプションは無視される。



SOURCE
ソース・リストで生のソースを表示します。また、さらに重要なこととして、IBM Debug Tool でこれをソース・ビューとして表示します。

AFTERALL
最後のプリプロセッサーの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツール でこれをソース・ビューとして立ち上げます。

AFTERALL の省略形として AALL を使用できます。

AFTERCICS
CICS プリプロセッサーの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツール でこれをソース・ビューとして立ち上げます。

ACICS は、AFTERCICS の省略形として使用されることがあります。

AFTERMACRO
MACRO プリプロセッサーの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツール でこれをソース・ビューとして立ち上げます。

AMACRO は、AFTERMACRO の省略形として使用されることがあります。

AFTERSQL
SQL プリプロセッサーの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツール でこれをソース・ビューとして立ち上げます。

ASQL は、AFTERSQL の省略形として使用されることがあります。

TEST オプションを指定して、LISTVIEW に SOURCE 以外のサブオプションを指定した場合には、TEST オプションに SEPARATE サブオプションも指定する必要があります。

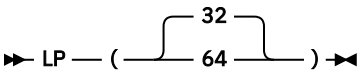
以下の例は、AFTERMACRO、AFTERSQL、AFTERALL サブオプションの効果の違いを示している。

PP オプションが PP(MACRO('INCONLY'), SQL, MACRO) であるとします。

- LISTVIEW(AFTERMACRO) では、TEST(SEP) が指定された場合にリストおよび デバッグ・ツール ソース・ウィンドウに示される「ソース」は、MACRO プリプロセッサーの 2 番目の呼び出しによって生成された MDECK から得られたかのように表示されます。
- LISTVIEW(AFTERSQL) では、TEST(SEP) が指定された場合にリストおよび デバッグ・ツール ソース・ウィンドウに示される「ソース」は、SQL プリプロセッサーの呼び出しによって生成された MDECK から得られたかのように表示されます (したがって、%DCL およびその他のマクロ・ステートメントは依然として表示されます)。
- LISTVIEW(AFTERALL) では、マクロ・プリプロセッサーが PP オプションの最後であるため、「ソース」は LISTVIEW(AFTERMACRO) オプションの下にきます。

LP

LP オプションは、コンパイラが 31 ビット・コードと 64 ビット・コードのどちらを生成するかを指定する。 また、 POINTER 、 HANDLE 、関連する変数のデフォルトサイズも決定する。



32
LP(32) では、コンパイラーは 31 ビット・コードを生成します。 さらに、タイプ *size_t* は FIXED BIN(31) に解決されます。 POINTER および HANDLE のデフォルト・サイズは 4 バイトです。

64
LP(64) では、コンパイラーは 64 ビット・コードを生成します。 さらに、タイプ *size_t* は FIXED BIN(63) に解決されます。 POINTER および HANDLE のデフォルト・サイズは 8 バイトです。

注 : LP(64) では、一部のコンパイラー・オプションは適用されません。 詳しくは、181 ページの『コンパイラー・オプションを使用して 64 ビット・アプリケーションをビルド』 を参照してください。

デフォルトは LP(32) です。

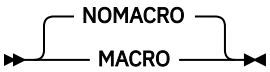
関連情報

181 ページの『64 ビット・アプリケーションを開発する場合の考慮事項』
Enterprise PL/I を使用すれば、31 ビット/64 ビット・アプリケーションを開発できます。 ご使用のアプリケーションで 64 ビット環境がサポートされるようにするには、必要に応じてコードを調整しなければならない場合があります。 このセクションでは、開発時およびコンパイル時に考慮すべき事項について説明します。

177 ページの『64 ビット・プログラムに対するリンク・エディットおよび実行』
LP(64)でコンパイルした後、64 ビット・プログラムは、 Language Environment ランタイム・ライブラリへの参照と同様に、互いに未解決の参照を含む 1 つまたは複数のオブジェクト・モジュールで構成されます。 これらの参照は、リンク・エディット時 (静的) または実行時 (動的) に 解決されます。

MACRO

MACRO オプションは MACRO プリプロセッサを起動する。



省略形: M、NM

また、PP(MACRO) オプションを使って MACRO プリプロセッサを呼び出すこともできる。 しかし、同じコンパイルで MACRO オプションと PP(MACRO) オプションの両方を使うことは推奨されない。

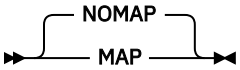
関連情報

62 ページの『PP』
PP オプションは、コンパイル前に呼び出すプリプロセッサー (およびそれらの順序) を指定します。

118 ページの『マクロ・プリプロセッサー』
マクロを使用すれば、インプリメンテーションの詳細と処理対象のデータを隠して演算のみを表す方法で、通常使用される PL/I コードを作成できます。 汎用のサブルーチンとは対照的に、マクロでは、個別用途ごとに必要となるコードのみを生成できます。 マクロ・プリプロセッサは、 MACRO オプションまたは PP(MACRO) オプションを指定して呼び出すことができる。

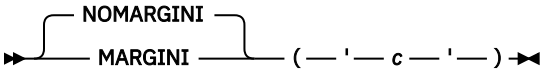
MAP

MAP オプションは、ダンプで静的変数や自動変数を見つけるために使用できる追加情報をコンパイラーが生成するように指定します。



MARGINI

MARGINI オプションは、INSOURCE オプションと SOURCE オプションによって生成されるリストの左側のマージンの前の列と、右側のマージンの後の列に、コンパイラが配置する文字を指定する。



省略形: MI、NMI

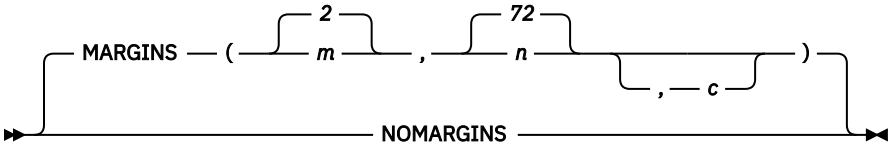
c
マージン標識として印刷される文字。

注 : MARGINI は (' ')と同じ。 MARGINI

MARGINS

SOURCE オプション、INSOURCE オプション、またはその両方が適用される場合、 MARGINS オプションは、各コンパイラ入力レコードのどの部分に PL/I 文が含まれるかを指定し、リストをフォーマットする ANS 制御文字の位置を指定する。 コンパイラーは、これらの限界外にあるデータは処理しませんが、 ソース・リストには入れます。

PL/I ソースがソース入力レコードから取り出される際、レコードの最初の データ・バイトが、その直前のレコードの最後のデータ・バイトのすぐ後にくる ように取り出されます。 変数レコードの場合、ブランクが必要であれば、必ず各レコードのマージン間に明示的にブランクを挿入するようにしなければなりません。



省略形: MAR

m
コンパイラーによって処理される左端の文字 (最初のデータ・バイト) の 桁番号。 これは、100 を超えてはなりません。

n
コンパイラーによって処理される右端の文字 (最後のデータ・バイト) の 桁番号。 これはmより大きくなければなりませんが、 200 を超えてはなりません。

可変長レコードは、最大レコード長になるように効果的にブランクが 埋め込まれます。

c
ANS プリンター制御文字の桁番号。 200 を超えてはなりません。また、分 および *N* に指定された値の範囲外でなければなりません。 *c* の値が 0 の場合は、ANS 制御文字が存在しないことを示します。 次の制御文字だけを使用できます。

(ブランク)
1 行スキップしてから印刷する。

0
2 行スキップしてから印刷する。

–
3 行スキップしてから印刷する。

+
スキップしないで印刷する。

1
改ページする

これ以外の文字を使用するとエラーになり、ブランクに置き換えられます。

c の値がソース・レコードの最大長より大きいと、リストの形式が予測できなくなるため、使用しないでくださ い。 この問題を避けるには、可変長レコードのソース・マージンの左側に 紙送り制御文字を置きます。

%PAGE ステートメントや %SKIP ステートメントを使用する代わりに MARGINS(,,c) を指定できます (「*PL/I Language Reference*」を参照してください)。

IBM -固定長レコードのデフォルトは MARGINS(2,72)。 可変長レコードと不定長レコードの IBM 提供のデフォルトは MARGINS(10,100) です。 このデフォルトは、プリンター制御文字が**ない**ことを指定します。

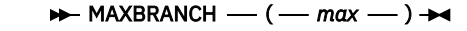
MARGINS オプションを使用して、プログラムの主入力のデフォルトを上書きする。 2 次入力のマージンは 1 次入力の場合と同じでなければなりません。

NOMARGINS オプションは、以前に遭遇した MARGINS オプションのインスタンスを抑制する。 このオプションの目的は、ご使用のシステムのコンパイル時間オプションをデフォルト設定にできるようにすることで、このコンパイル時間オプションは、変数ソース・フォーマット・ファイルが使用可能になっている間、固定フォーマット・ソース設定用に調整された MARGINS オプションを使用します。

通常、NOMARGINS オプションを使用する場合は、コンパイラに渡されるパラメータ文字列の一部として指定することになります。 コンパイラは、%PROCESS 文でこのオプションを見つけると、 NOMARGINS を無視します。

MAXBRANCH

MAXBRANCH オプションは、分岐が多すぎるブロックにフラグを立てる。 ブランチには、すべての条件付きジャンプと、ブランチテーブルに変換できる SELECT ステートメントの各 WHEN が含まれる。



max
ブロックのサイクロマティックな複雑さや条件付きの複雑さを測定する限度。 デフォルトは 2000 です。

「`if a then ...; else ...`」という形式のステートメント 収容ブロック内のブランチの総数に 1 を加算し、「`if a = 0 | b = 0 then ...`」という形式のステートメントを追加します。 2 を加えます。

MAXINIT

MAXINIT オプションは、VALUE 型関数を型付き構造体に適用したときに、コンパイラが大きなオブジェクト・ファイルを生成するコードにフラグを立てるかどうかを決定します。

max
型付き構造体に対して VALUE 型関数を適用する際に使用できる最大バイト数を指定する。 コンパイラーは、*max* で指定された量を超えるバイトを使用するすべてのステートメントにフラグを立てます。

このオプションでフラグが立てられたステートメントを調べてください。 異なるコーディングをするか、型付き構造体の定義を変更すると、コンパイラーによって生成されるオブジェクト・ファイルのサイズを縮小できる可能性があります。

デフォルトは MAXINIT(64K)です。

MAXGEN

MAXGEN オプションは、任意のユーザー文に対して生成されるべき中間言語文の最大数を指定する。 このオプションが指定されると、この最大数を超えたすべてのステートメントにコンパイラーがフラグを立てます。

任意のユーザー・ステートメントに対して生成される中間言語ステートメントの数は、コンパイラー・リリース、コンパイラー保守レベル、および有効になっているコンパイラー・オプションによって異なる可能性があります。 このオプションの唯一の目的は、過剰な量のコードが生成される (つまり、コーディングが正常に行われていない可能性がある) ステートメントを検出できるように支援することです。

ただし、プリプロセッサーが使用されると、一部のステートメントに対して生成される中間言語ステートメントの数が非常に多くなる可能性があることに注意してください。 このような状況では、MAXGEN のしきい値を大きく設定するか、 LISTVIEW(AFTERALL) オプションを使用する方がよいだろう。

デフォルトは MAXGEN (100000)。

MAXMEM

OPTIMIZE を指定してコンパイルする場合、 MAXMEM オプションを指定すると、特定の、メモリを大量に消費する最適化のローカル・テーブルに使用するメモリ量を、指定したキロバイト数に制限することができます。

MAXMEM に指定できるメモリの範囲は 1 ～ 2097152 です。 デフォルトは 1048576 です。

最大値の 2097152 を指定した場合、 コンパイラーは無制限のメモリーが使用可能であることを想定します。 MAXMEM にこれより小さい値を指定すると、コンパイラは、特に OPT(2) オプションが有効な場合に、最適化が阻害されているので、MAXMEM にこれより大きい値を使用してみるようにというメッセージを出すかもしれません。

利用可能なメモリーがデフォルト値より少ない（または多い）ことがわかっている場合は、 MAXMEM 。

MAXMEM オプションで指定されたメモリが特定の最適化に対して不足する場合、最適化の質が低下するようにコンパイルが完了し、警告メッセージが発行される。

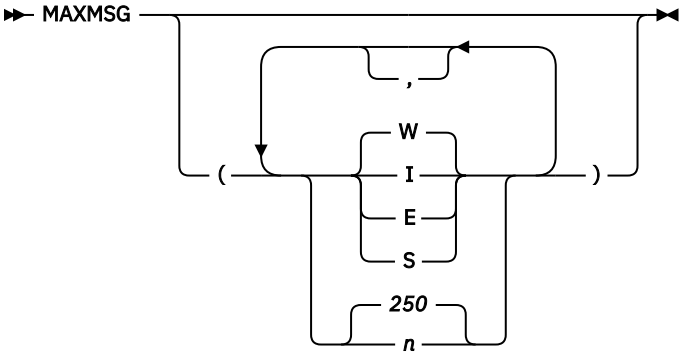
省略形: MAXMEM

MAXMEM に大きなサイズが指定された場合、コンパイルされるソース・ファイル、ソース中のサブプログラムのサイズ、コンパイルに利用可能な仮想ストレージによっては、仮想ストレージが不足しているためにコンパイルが中断されることがある。

MAXMEM オプションを使用する利点は、大規模で複雑なアプリケーショ ンの場合、コンパイラはエラー・メッセージ insufficient virtual storage でコンパイルを 終了する代わりに、最適化度が若干低いオブジェクト・モジュールを生成 し、警告メッセージを生成することである。

MAXMSG

MAXMSG オプションは、コンパイルが生成すべき、指定された重大度（またはそれ以上）のメッセージの最大数を指定する。



I
すべてのメッセージを数えます。

W
情報メッセージを除くすべてのメッセージを数えます。

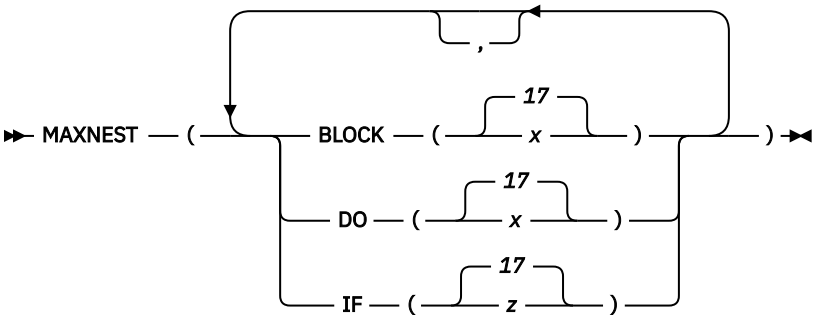
E
警告メッセージと通知メッセージを除くすべてのメッセージを 数えます。

S
重大エラー・メッセージおよび回復不能エラー・メッセージだけを 数えます。

n
メッセージの数がこの値を超えた場合、コンパイルを終了します。 指定の重大度より低いメッセージ、またはコンパイラー出口ルーチンにより フィルターに掛けられて取り除かれたメッセージは、カウントされません。 値は 0 から 32767 までの範囲で入力します。 0 を 指定した場合、指定された重大度の最初のエラーが検出されると コンパイルは終了します。

MAXNEST

MAXNEST オプションは、コンパイラがプログラムを複雑すぎると判断するまでに許容される、さまざまな種類の文の最大入れ子数を指定する。



BLOCK
BEGIN および PROCEDURE ステートメントの最大ネストを指定します。

DO
DO ステートメントの最大ネストを指定します。

IF
IF ステートメントの最大ネストを指定します。

ネスト限度の値範囲は 1 から 50 までです。

デフォルトは MAXNEST(BLOCK(17) DO(17) IF(17))です。

MAXRUNONIF

このオプションは、(a = y1 | a = y2 |... | a = yn)という形の条件式を持つ IF 文（y はすべて定数、n は MAXRUNONIF しきい値以上）のフラグを立てるメッセージを出すようにコンパイラに指示するしきい値を指定する。

IF-式の比較数に応じて、コンパイラ・メッセージは、式を INLIST 参照に変換するか、 IF ステートメントを SELECT ステートメントに変換することを提案する。 例えば、 MAXRUNONIF(4)は、コンパイラにフラグを立てることになる：

```
if code = 'DE' | code = 'FR' | code = 'ES' | code = 'IT' then
```

これを以下のように変換することを推奨します。

```
if inlist( code, 'DE', 'FR', 'ES', 'IT' ) then
```

このオプションのデフォルトは MAXRUNONIF(10)。

MAXSTATIC

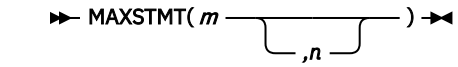
MAXSTATIC オプションは、指定したサイズを超える STATIC 変数にフラグを立てます。

MAX
STATIC 変数の最大サイズを指定する。 コンパイラーは、*max* で指定された量以上のバイトを使用するすべての STATIC 変数にフラグを立てます。

デフォルトは MAXSTATIC(1M)。

MAXSTMT

MAXSTMT オプションを指定すると、実行可能なステートメントの数が指定した数より多いブロックにフラグを立てるようになる。 optimize の下では、コンパイラーは、指定された数以上の実行ステートメントを持つブロックの最適化もオフにする。



m
OPT(2)のカットオフ値を指定する。 デフォルトは 4096 です。

n
オプションです。 これは、OPT(3) のカットオフ値を指定します。 デフォルトは 8192 です。

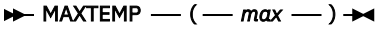
MAXSTMT(*m*)が指定された場合、 *n* は *m* に設定される。 つまり、 MAXSTMT(4096)を指定することは、MAXSTMT(4096,4096)と等価である。

MAXSTMT に大きな *m* を指定した場合、実行可能なステートメントが大量にあるブロックがあると、利用可能な仮想ストレージが不足してコンパイルが中断される可能性がある。

注 : 実行可能文とは、コンパイラがコードを生成する必要のある文のことで、STMT オプションで特定されるソース言語文とは 1 対 1 には対応しない。 例えば、 DECLARE/DEFINE ステートメントは実行可能コードを生成しないので、 MAXSTMT の目的にはカウントされない。

MAXTEMP

MAXTEMP オプションは、コンパイラー生成一時ステートメント用のストレージの量を非常に多く使用しているステートメントについて、コンパイラーがいつフラグを立てるかを判断します。

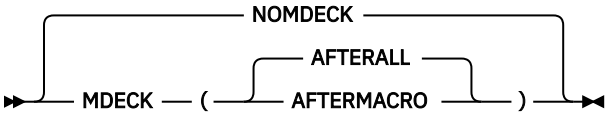


MAX
コンパイラー生成一時ステートメントに使用できるバイト数の限度。 *max* で指定されたバイト数より多くのバイトを使用するすべてのステートメントにコンパイラーがフラグを立てます。*max* のデフォルトは 50000 です。

このオプションでフラグが立てられたステートメントを調べてください。 そのステートメントを別の方法でコーディングすれば、コードで必要となるスタック・ストレージの量を減らすことができる可能性があります。

MDECK

MDECK オプションは、z/OS の SYSPUNCH DD 文で定義されたファイルか、z/OS UNIX の.dek ファイルのいずれかに、プリプロセッサがその出力のコピーを生成することを指定する。



省略形: MD、NMD

MDECK オプションを使うと、プリプロセッサからの出力を 80 列のレコードのファイルとして保持することができる。 このオプションは、MACRO オプションの使用中にだけ 使用することができます。

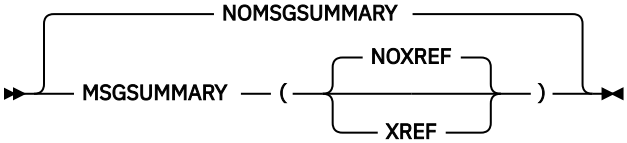
AFTERALL
最後のプリプロセッサーが呼び出された後でファイルが生成されます。

AFTERMACRO
マクロ・プリプロセッサーが最後に呼び出された後でファイルが生成されます (呼び出しがあった場合)。

関連情報
51 ページの『MACRO』
MACRO オプションは MACRO プリプロセッサを起動する。

MSGSUMMARY

MSGSUMMARY オプションは、コンパイラがコンパイル中に発行されたすべてのメッセージの要約をリストに追加するかどうかを決定する。



MSGSUMMARY(NOXREF)
コンパイラーがメッセージの要約をリストに追加します。 この要約はリスト内のファイル参照テーブルの後に配置されます。 この要約はコンパイラー・コンポーネント別にソートされ、さらに各コンポーネント内で重大度別にソートされ、続いてメッセージ番号でソートされます。

この要約には以下の情報が含まれます。

- コンパイルで生成された各メッセージ (同じメッセージが複数ある場合でも示されるメッセージは 1 つのみ)
- 各メッセージが生成された回数

MSGSUMMARY(XREF)
コンパイラーがメッセージの要約をリストに追加します。 この要約は、MSGSUMMARY(NOXREF) が指定されたときに追加されるものと同じです。 ただし、この要約では、各メッセージの後ろに、メッセージが発行された行またはステートメントの番号がすべてリストされるという点が異なります。

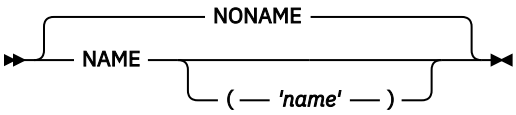
NOMSGSUMMARY
メッセージの要約は生成されません。

NOMSGSUMMARY はデフォルトです。 MSGSUMMARY が指定されている場合、 MSGSUMMARY(NOXREF) がデフォルトとなる。

MSGSUMMARY を使って生成されたメッセージ・サマリー付きのコンパイラー・リス トの例については、113 ページの『#unique_145/unique_145_Connect_42_complistxm』 を参照のこと。

NAME

NAME オプションは、コンパイラが作成する TEXT ファイルに NAME レコードが含まれることを指定する。



省略形: N

NAME オプションのサブオプションとして名前が指定されていない場合、使用される名前は以下のように決定される：

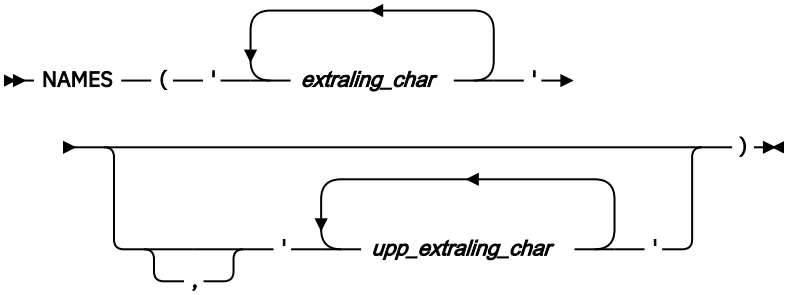
- PACKAGE 、一番左の名前が使われる。
- そうでない場合は、最初の PROCEDURE 。

LIMITS(EXTNAME(*n*)) オプションを *n* <= 8 で使用する場合、名前の長さは 8 文字を超えてはならない。

NAMES

NAMES オプションは、識別子で許可される外国語文字を指定する。

言語外文字とは、...で定義されているアルファベット 26 文字、数字 10 文字、特殊文字以外の文字のことである。 *PL/I Language Reference*.



extralingual_char
特別言語文字。

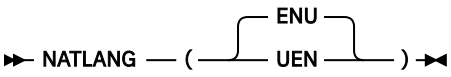
upp_extraling_char
最初のサブオプションで指定した文字に対応する大文字として解釈させる特別言語文字

2 番目のサブオプションを省略すると、PL/I は最初のサブオプションで 指定された文字を小文字と大文字の両方として使用します。 2 番目のサブオプションを指定する場合は、最初のサブオプションで 指定したのと同じ数の文字を指定しなければなりません。

デフォルトは NAMES('@#$' '@#$')です。

NATLANG

NATLANG オプションは、コンパイラー・メッセージやヘッダーなどの言語を指定する。

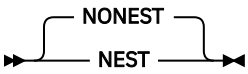


ENU
コンパイラーのメッセージやヘッダーなどはすべて大/小文字混合の英語になります。

UEN
コンパイラーのメッセージやヘッダーなどはすべて大文字の英語になります。

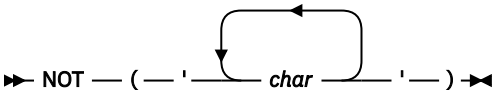
NEST

NEST オプションは、SOURCE オプションから得られるリストが、各ステートメントのブロック・レベルと DO グループ・レベルを示すように指定する。



NOT

NOT オプションは、論理演算子 NOT として使える代替記号を 7 つまで指定する。



char
単一の SBCS 文字

標準の論理否定記号 (¬) を除き、 *PL/I Language Reference* で定義されている英字、数字、および特殊文字を指定することはできません。 少なくとも 1 つの有効な文字を指定する必要があります。

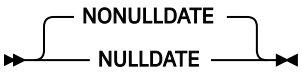
NOT オプションを指定すると、標準 NOT 記号は、文字ストリング内の 1 つ の文字として指定しない限り認識されなくなります。

例えば、NOT('~') を指定すると、波形記号 'A1'X が論理 NOT 演算子として認識され、標準 NOT 記号 ('¬') '5F'X は認識されません。 同様に、NOT('~¬') は波形記号または標準 NOT 記号のどちらかが論理否定演算子として認識されることを意味します。

NOT 記号用の IBM 提供のデフォルト・コード・ポイント は、'5F'X です。 論理否定記号は、キーボード上では論理否定記号 (¬) または 脱字記号 (^) として表記されていることがあります。

NULLDATE

NULLDATE オプションは、SQL の NULL 日付を、一部の datetime 処理組み込み関数で有効な日付として受け入れるようにコンパイラに指示します。

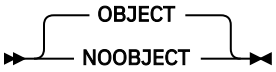


NULLDATE オプションでは、 VALIDDATE および REPATTERN 組み込み関数は、SQL の NULL 日付（ 年、日、 月がすべて 1 に等しい）を有効な日付として受け付けます。

デフォルトは NONULLDATE です。

OBJECT

OBJECT オプションは、コンパイラーがオブジェクト・モジュールを作成することを指定する。 バッチ z/OS では、コンパイラーはオブジェクトを SYSLIN DD で定義されたデータセットに格納し、z/OS UNIX では、コンパイラーは.o ファイルを作成する。



省略形: OBJ、NOBJ

NOOBJECT オプションでは、コンパイラーはオブジェクト・モジュールを作成しない。 しかし、NOOBJECT オプションの下では、コンパイラーは構文解析と意味解析の全フェーズを完了し、初期化されていない変数の検出もすべて完了するため、NOCOMPILE, NOSEMANTIC, または NOSYNTAX オプションの場合よりも多くのメッセージを生成する可能性があります。

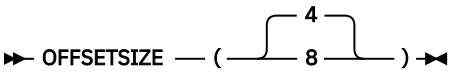
NOOBJECT オプションでは、 LIST, MAP, OFFSET、 STORAGE オプションは無視される。

OFFSET

OFFSET オプションは、各プロシージャーと BEGIN ブロックの行番号と、そのプロシージャーのプライマリー・エントリー・ポイントからのオフセット・アドレスの表をコンパイラーが表示することを指定する。 このテーブルは、 GONUMBER オプションが使用されていない場合、実行時エラーメッセージからステートメントを特定するために使用することができます。

OFFSETSIZE

OFFSETSIZE オプションは、64 ビット・アプリケーションの OFFSETSIZE 変数のサイズを決定する。



4
OFFSETSIZE(4)では、OFFSETSIZE 変数のサイズはすべて 4 バイトである。 これがデフォルト値です。

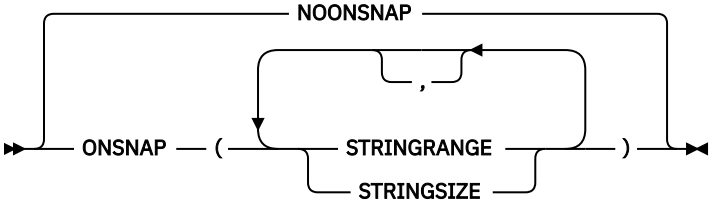
8
OFFSETSIZE(8)では、 OFFSETSIZE 変数のサイズはすべて 8 バイトである。

OFFSETSIZE または AREA 変数を共有するコードは、 OFFSETSIZE オプションに同じ値を指定してコンパイルしなければならない。

LP(32) オプションが有効な場合、 OFFSETSIZE オプションは無視される。

ONSNAP

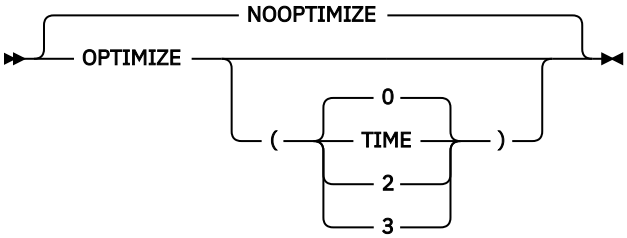
OPTIONS(MAIN) または OPTIONS(FROMALIEN) 属性を持つ PROCEDURE に対して、ONSNAP オプションは、コンパイラがその PROCEDURE のプロローグ・コードに `ON STRINGRANGE SNAP;` 文、 `ON STRINGSIZE SNAP;` 文、またはその両方を挿入することを指定します。 これにより、このような PROCEDURE から呼び出された他のルーチンで対応する条件が発生した場合に呼び出しチェーンを突き止めやすくなります。



ONSNAP オプションは、これらの属性のいずれかを持たない PROCEDURE には影響しない。

OPTIMIZE

OPTIMIZE オプションは、必要な最適化のタイプを指定する。



省略形: OPT、NOPT

NOOPTIMIZE
OPTIMIZE(0) と同等。

OPTIMIZE(0)
高速コンパイルを指定しますが、最適化は禁止します。

OPTIMIZE(2) or OPTIMIZE(TIME)
OPTIMIZE(3) と同等。 OPTIONS リストには OPTIMIZE(3)と表示される。

OPTIMIZE(3)
さらに効率のよいオブジェクト・プログラムが作成されるために、 生成された機械命令を最適化します。 この種の最適化は、オブジェクト・コードに必要なストレージの量も減らすことができる。

DFT(REORDER) オプションは、OPTIMIZE オプションと併用することを強く推奨する。 実際、以下の条件がすべて当てはまる PROCEDURE または BEGIN-ブロックでは、OPTIMIZE の効果は厳しく制限される：

- ORDER オプションはブロックに適用される。
- このブロックには、ハードウェアで検出された状態（ ZERODIVIDE など）用の ON ユニットが含まれる。
- ブロックに、これらの ON ユニットからの分岐のターゲットになる (可能性 がある) ラベルがある。

OPTIMIZE(3)を使用すると、 NOOPTIMIZE よりもコンパイル時間が大幅に増加し、必要なスペースも大幅に増加する可能性があります。 例えば、OPTIMIZE(3) で大きなプログラムをコンパイルする場合、数分かかるかもしれないし、 100M 以上の領域を必要とするかもしれない。

最適化中、コンパイラーは実行時効率を高めるために、コードを移動することができます。 その結果、プログラム・リスト中のステートメント番号が、ランタイム・メッセージで使用されるステートメント番号と対応しなくなることがあります。 また、ステートメント番号テーブルを使用する Fault Analyzer や CEEDUMP などの診断製品によって報告されたステートメントが正しくない場合もあります。

OPTIMIZE(3)の使用は、以下のように TEST オプションの機能を著しく制限することに注意：

- TEST の HOOK サブオプションが有効になっている場合は、ブロック・フックのみが生成されます。
- TEST の NOHOOK サブオプションが有効な場合、変数をリストまたは変更しようとすると失敗することがあります（変数がレジスタに最適化されている可能性があるため)。また、特定のステートメントで停止しようとすると、デバッガが何度も停止することがあります（ステートメントがいくつかに分割されている可能性があるため)。

1 つ以上のチェックアウト条件（SIZE, STRINGRANGE, STRINGSIZE、 SUBSCRIPTRANGE ）とともに PREFIX オプションを使用すると、コンパイルに必要な時間とスペースを大幅に増やすことができる。

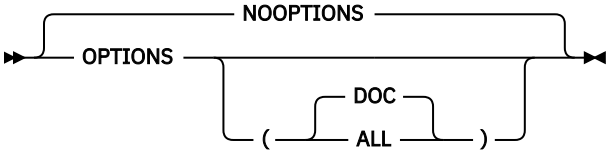
関連情報

305 ページの『パフォーマンスの向上』

ユーザーのプログラムの速度を向上することに関する多数の考慮事項は、 使用するコンパイラーとそれを実行するプラットフォームには関係ありません。 しかし、この章では、考慮事項の中でも PL/I コンパイラーとそれによって生成される コードに特有な考慮事項を識別して説明します。

OPTIONS

OPTIONS オプションは、コンパイラーがコンパイラー・リスティングに、コンパイル時に使用するコンパイラー・オプションのリストを含めるように指定します。



省略形: OP、NOP

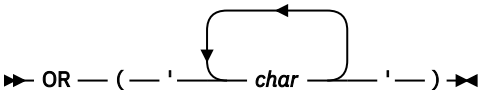
このリストには、デフォルトで適用されるすべてのオプション、EXEC 文の **PARM** パラメータまたは起動コマンド (pli) で指定されるオプション、%PROCESS 文で指定されるオプション、 z/OS の IBM_OPTIONS 環境変数で指定されるオプション、および任意のオプション・ファイルから組み込まれるすべてのオプションが含まれます。

OPTIONS(DOC) の場合、 OPTIONS のリストには、コンパイラーがリリースされた時点で、この文書に文書化されているオプション（およびサブオプション）のみが含まれます。

OPTIONS(ALL) の場合、OPTIONS リストには、コンパイラのリリース後に PTF によって追加されたオプションも含まれる。

OR

OR オプションは、論理演算子 OR として最大 7 つの代替記号を指定する。 これらの記号は連結演算子としても使われ、連続する 2 つの論理 OR 記号として定義されます。



注 : 引用符と引用符の間にブランクをコーディングしないでください。

OR シンボル (|) の IBM -supplied default code point は '4F'X。

char
単一の SBCS 文字

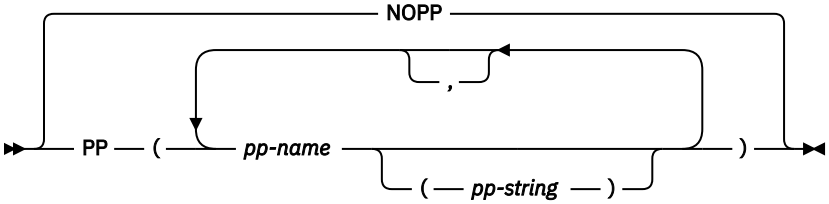
で定義されているアルファベット文字、数字、特殊文字は指定できない。 *PL/I Language Reference*OR で定義されているアルファベット、数字、特殊文字は指定できない。少なくとも 1 つの有効な文字を指定する必要があります。

OR オプションを指定した場合、標準の OR 記号は、文字列の中の 1 文字として指定しない限り、認識されなくなります。

例えば、 OR （'□'）は、バックスラッシュ文字「' E0 'X」が論理 OR 演算子として認識され、連続する 2 つのバックスラッシュが連結演算子として認識されることを意味する。 標準的な OR シンボルである'|', '4F'X, は、どちらの演算子としても認識されない。 同様に、OR(' \|') は、バックスラッシュまたは標準 OR シンボルのいずれかが論理 OR 演算子として認識され、いずれかのシンボルまたは両方のシンボルを使用して連結演算子を形成できることを意味する。

PP

PP オプションは、コンパイル前に呼び出すプリプロセッサー (およびそれらの順序) を指定します。



pp-name
特定のプリプロセッサーに与えられた名前。 現在サポートされているプリプロセッサーは、CICS、INCLUDE、MACRO、 および SQL だけです。 未定義の名前を使用すると診断エラーの原因となります。

pp-string
対応するプリプロセッサーのオプションを表す、引用符で区切られた 100 文字以内のストリング。 例えば、 PP(MACRO('CASE(ASIS)')) は、 CASE (ASIS)というオプションをつけて MACRO プリプロセッサを起動する。

プリプロセッサー・オプションは左から右へ処理されます。2 つのオプションが対立する場合は、最後の (右端の) オプションが 使用されます。 例えば、オプション文字列 'CASE(ASIS) CASE(UPPER)' を指定して MACRO プリプロセッサを起動すると、オプション CASE (UPPER) が使用される。

最大 31 のプリプロセッサー・ステップを指定でき、同じプリプロセッサーを複数回指定できます (CICS および SQL のプリプロセッサーを除く)。 CICS プリプロセッサーを呼び出すことができるのは 1 回まで、SQL プリプロセッサーは 2 回までです。 SQL プリプロセッサーは、最初の指定で INCONLY がオプションとして指定された場合にのみ、2 回呼び出すことができます。

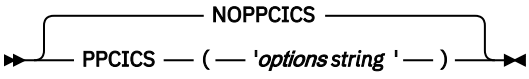
PP オプションとともに MACRO オプションが指定された場合は、MACRO プリプロセッサーが PP オプションにおけるプリプロセッサーのリストの先頭に追加されます (まだリストの先頭に置かれていない場合)。 そのため、MACRO と PP(SQL MACRO) を指定すると、PP オプションは PP(MACRO SQL MACRO) になり、MACRO プリプロセッサが 2 回呼び出されることになる。 ただし、MACRO と PP(MACRO SQL) が指定された場合は、PP オプションは変更されずに、MACRO プリプロセッサーが 1 回のみ呼び出されます。 ただし、同じコンパイルで MACRO オプションと PP(MACRO) オプションを両方とも使用することは勧められません。

PP オプションを複数回指定した場合、コンパイラーはそれらを実質的に連結します。 したがって、PP(SQL) PP(CICS) を指定することは、PP(SQL CICS) を指定するのと同じことになります。 これはまた、PP(MACRO SQL('CCSID0')) と PP(MACRO SQL('CCSID0 DATE(ISO)')) を指定した場合、結果の PP オプションは PP(MACRO SQL('CCSID0') MACRO SQL('CCSID0 DATE(ISO)')) となり、MACRO プリプロセッサと SQL プリプロセッサの両方が 2 回起動され、SQL プリプロセッサの 2 回目の起動はエラーになることを意味する。 これを先の SQL オプションを上書きするために行うのであれば、PP オプションでプリプロセッサ・オプションを指定するのではなく、PPSQL オプションを通して指定する、つまり、 PP(MACRO SQL) PPSQL('CCSID0 DATE(ISO)').

関連情報
117 ページの『PL/I プリプロセッサー』
PL/I コンパイラーを使用するときは、ご使用のプログラムにおいて組み込みプリプロセッサーを 1 つ以上指定できます。 組み込みプリプロセッサー、マクロ・プリプロセッサー、SQL プリプロセッサー、または CICS プリプロセッサーを指定できます。 また、これらのプリプロセッサーを呼び出す順序を指定できます。

PPCICS

PPCICS オプションは、プリプロセッサが呼び出されたときに CICS オプションを指定する。



PPCICS('EDF') PP(CICS) を指定することは、 PPCICS('EDF') を指定することと同じである。

このオプションは、PP(CICS) オプションが指定されなければ有効ではありません。 ただし、CICS プリプロセッサーが呼び出されるときに使用される CICS プリプロセッサー・オプションのセットを指定する場合

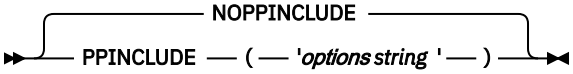
は、インストール・オプションの出口でこのオプションを指定することができます。 そして、PP(CICS)を指定するたびに、 PPCICS オプションで指定されたオプションのセットが使われる。

また、プリプロセッサを起動するときに指定されたオプションは、 PPCICS オプションで指定されたものを上書きする。 つまり、 PPCICS('EDF') PP(CICS('NOEDF')) を指定することは、 PP(CICS('EDF NOEDF')) や、さらに単純な PP(CICS('NOEDF')) を指定することと同じである。

オプション・ストリングの長さは 1000 文字に制限されます。 ただし、ストリングは、100 文字を超える場合、オプション・リストに表示されません。

PPINCLUDE

PPINCLUDE オプションは、 INCLUDE プリプロセッサーが起動されたときに渡すオプションを指定する。



PPINCLUDE('ID(-inc)') PP(INCLUDE) を指定することは、 PP(INCLUDE('ID(-inc)')) を指定することと同じである。

このオプションは、 PP(INCLUDE) オプションが指定されない限り、何の効果もない。 ただし、 INCLUDE プリプロセッサーが起動されたときに使用される INCLUDE プリプロセッサー・オプションのセットを指定したい場合は、インストール・オプションの終了時にこのオプションを指定できます。 その後、 PP(INCLUDE) を指定するたびに、PPINCLUDE オプションで指定されたオプションのセットが使用される。

また、プリプロセッサを起動するときに指定されたオプションは、PPINCLUDE オプションで指定されたものを上書きする。 つまり、 PPINCLUDE('ID(-inc)') PP(INCLUDE('ID(+include)')) を指定することは、 PP(INCLUDE('ID(-inc) ID(+include)')) や、さらに単純な PP(INCLUDE('ID(+include)')) を指定することと同じである。

オプション・ストリングの長さは 1000 文字に制限されます。 ただし、ストリングは、100 文字を超える場合、オプション・リストに表示されません。

PPLIST

PPLIST オプションは、コンパイラが各プリプロセッサー・フェーズで生成されたリストの一部を保持するか消去するかを制御する。

PPLIST(KEEP) を指定すると、コンパイラーは各プリプロセッサー・フェーズで生成されたリストの一部を保持する。

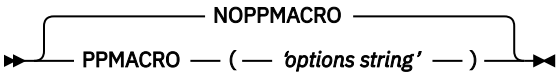
PPLIST(ERASE) を指定すると、コンパイラは、メッセージを生成しないプリプロセッサー・フェーズによって生成されたリストの部分を消去します。

コンパイラーは、 EXIT と FLAG オプションによって抑制されたメッセージを数えない。 したがって、 FLAG(W) と PPLIST(ERASE) の両方を指定すると、警告、エラー、または厳しいメッセージを生成しないプリプロセッサからのすべての出力をコンパイラが抑制することになる。

PPLIST(KEEP)がデフォルトです。

PPMACRO

PPMACRO オプションは、 PPMACRO プリプロセッサーが起動されたときに渡すオプションを指定する。



オプション・ストリング
118 ページの『マクロ・プリプロセッサーのオプション』 を参照してください。

PPMACRO('CASE(ASIS)') PP(MACRO) を指定することは、 PP(MACRO('CASE(ASIS)')) を指定することと同じである。

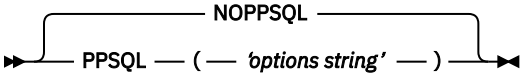
このオプションは、 PP(MACRO) オプションが指定されない限り、何の効果もない。 ただし、MACRO プリプロセッサが起動したときに使用する MACRO プリプロセッサ・オプションのセットを指定したい場合は、インストール・オプションの終了時にこのオプションを指定できます。 その後、MACRO または PP(MACRO) オプションを指定すると、 PPMACRO オプションで指定されたオプションのセットが使用される。

また、PPMACRO オプションで指定されたオプションは、プリプロセッサーが呼び出されるときに指定されるオプションの形に変更されます。 つまり、PPMACRO('CASE(ASIS)') PP(MACRO('CASE(UPPER)')) を指定することは、PP(MACRO('CASE(ASIS) CASE(UPPER)')) や、さらに単純な PP(MACRO('CASE(UPPER)')) を指定することと同じである。

オプション・ストリングの長さは 1000 文字に制限されます。 ただし、ストリングは、100 文字を超える場合、オプション・リストに表示されません。

PPSQL

PPSQL オプションは、その SQL プリプロセッサーに渡されるオプションを指定します。



オプション・ストリング
125 ページの『SQL プリプロセッサー・オプション』 を参照してください。

PPSQL('APOSTSQL') PP(SQL) を指定することは、 PP(SQL('APOSTSQL')) を指定することと同じである。

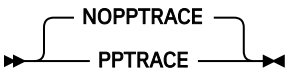
このオプションは、 PP(SQL) オプションが指定されない限り、何の効果もない。 ただし、SQL プリプロセッサーが呼び出される場合に使用されるオプションのセットを指定する場合は、インストール・オプションの出口でこのオプションを指定することができます。 そして、PP(SQL) を指定するたびに、PPSQL のオプションのセットが使われる。

また、プリプロセッサの起動時に指定されたオプションは、 PPSQL オプションで指定されたものを上書きする。 したがって、 PPSQL('APOSTSQL') PP(SQL('QUOTESQL')) を指定することは、 PP(SQL('APOSTSQL QUOTESQL')) や、さらに単純な PP(SQL('QUOTESQL')) を指定することと同じである。

オプション・ストリングの長さは 1000 文字に制限されます。 ただし、ストリングは、100 文字を超える場合、オプション・リストに表示されません。

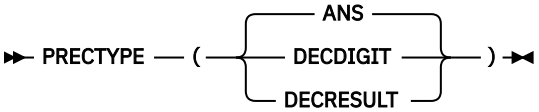
PPTRACE

PPTRACE オプションは、デッキ・ファイルがプリプロセッサのために書かれるとき、そのファイルのすべての空白でない行が %LINE ディレクティブを含む行で先行されることを指定します。 このディレクティブは、その非ブランク行が帰属するオリジナルの ソース・ファイルと行を示します。



PRECTYPE

PRECTYPE オプションは、オペランドが FIXED であり、少なくとも 1 つが FIXED BIN である場合に、MULTIPLY、DIVIDE、ADD、および SUBTRACT 組み込み関数の属性をコンパイラーが取り出す方法を決定します。



ANS
PRECTYPE(ANS) では、BIF(*x*,*y*,*p*) および BIF(*x*,*y*,*p*,0) の値 *p* は、2 進数字を指定するものとして解釈されます。 演算は 2 項演算として行われます。 その結果には属性 FIXED BIN(*p*,0) が付与されます。

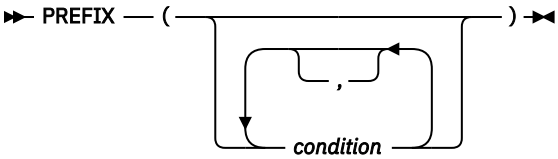
ただし、BIF(*x*,*y*,*p*,*q*) で *q* が ゼロではない場合、演算は 10 進演算として行われ、その結果には属性 FIXED DEC(*t*,*u*) が付与されます。 *t* および *u* は *p* および *q* の 10 進数に相当するもの (すなわち、*t* = 1 + ceil(*p* / 3.32) および *u* = ceil(*q* / 3.32)) です。 この場合、*x*、*y*、*p*、および *q* は、実際にすべて 10 進数に変換されます (これとは対照的に、DECDIGIT サブオプションでは、*x* と *y* のみが 10 進数に変換されます (*q* がゼロであっても 10 進数に変換されます))。 この状態では、コンパイラーは通知メッセージ 1BM1053 を出します。

DECDIGIT
PRECTYPE(DECDIGIT) では、BIF(*x*,*y*,*p*) および BIF(*x*,*y*,*p*,0) の値 *p* は、10 進数字を指定するものとして解釈されます。 演算は 2 項演算として行われます。 その結果には属性 FIXED BIN(*s*) が付与されます。 *s* は *p* に対応する 2 進数に相当します (すなわち *s* = ceil(3.32***p*))。 BIF(*x*,*y*,*p*,*q*) の場合 (*q* はゼロ以外)、PRECTYPE(DECDIGIT) での結果は、下で説明する PRECTYPE(DECRESULT) の結果と同じです。

DECRESULT
PRECTYPE(DECRESULT) では、BIF(*x*,*y*,*p*) の値 *p*、および BIF(*x*,*y*,*p*,*q*) の 値 *p* と *q* は 10 進数字を指定するものとして解釈されます。 演算は 10 進演算として行われます。 その結果には属性 FIXED DEC(*p*,0) または FIXED DEC(*p*,*q*) がそれぞれ付与されます。 結果は、DECIMAL 組み込み関数が *x* と *y* に適用された場合に生成されるものと同じになります。

PREFIX

PREFIX オプションは、ソース・プログラムを変更することなく、コンパイル中のコンパイル・ユニットで指定された PL/I 条件を有効または無効にします。 指定された条件プレフィックスは、最初の PACKAGE または PROCEDURE ステートメントの先頭に論理的に付加される。



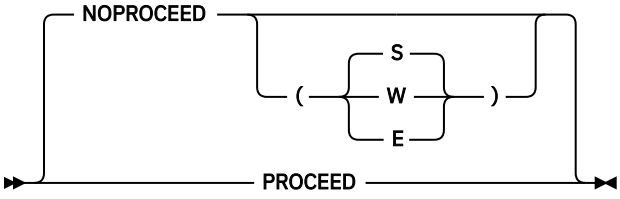
condition
PL/I プログラムにおいて有効/無効にできる任意の条件 (「*PL/I Language Reference*」を参照)。

1 つ以上のチェックアウト条件（CONFORMANCE, SIZE, STRINGRANGE, STRINGSIZE、SUBSCRIPTRANGE ）とともに PREFIX オプションを使用すると、コンパイルに必要な時間とスペースを大幅に増やすことができる。

デフォルト ： PREFIX(NOCONFORMANCE CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE UNDERFLOW ZERODIVIDE)

PROCEED

PROCEED オプションは、以前のプリプロセッサによって発行されたメッセージの重要度に応じて、プリプロセッサによる処理が完了した後にコンパイラを停止する。



省略形: PRO、NPRO

PROCEED
NOPROCEED(S) と同等です。

NOPROCEED
プリプロセッサーがコンパイルを終えた後、処理を停止します。

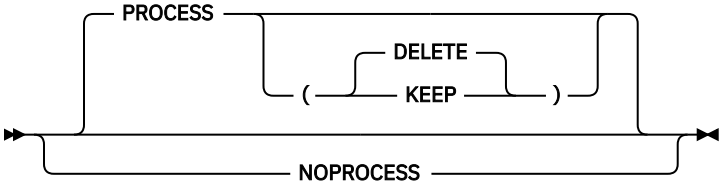
NOPROCEED(S)
このプリプロセスの段階で重大エラーまたは回復不能エラーが検出された場合は、プリプロセッサーおよびコンパイラーの呼び出しは継続されません。

NOPROCEED(E)
このプリプロセスの段階でエラー、重大エラー、または回復不能エラーが検出された場合は、プリプロセッサーおよびコンパイラーの呼び出しは継続されません。

NOPROCEED(W)
このプリプロセスの段階で警告、エラー、重大エラー、または回復不能エラーが検出された場合は、プリプロセッサーやコンパイラーの呼び出しは継続されません。

プロセス

PROCESS オプションは、*PROCESS ステートメントが許可されるかどうか、および許可される場合に、それらのステートメントが MDECK ファイルに書き込まれるかどうかを決定します。



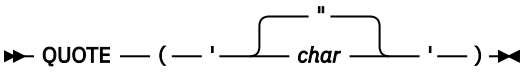
NOPROCESS オプションを指定すると、コンパイラーは、すべての *PROCESS ステートメントに E レベル・メッセージのフラグを立てます。

PROCESS(KEEP) オプションを指定すると、コンパイラーは *PROCESS ステートメントにフラグを立てることをせず、すべての *PROCESS ステートメントが コンパイラーによって MDECK 出力に保存されます。

PROCESS(DELETE) オプションを指定すると、コンパイラーは *PROCESS ステートメントにフラグを立てることをせず、どの *PROCESS ステートメントも コンパイラーによって MDECK 出力に保存されません。

QUOTE

QUOTE オプションは、引用符として使える代替記号を指定する。



注 : 引用符と引用符の間にブランクをコーディングしないでください。

IBM QUOTE シンボルのデフォルト・コードポイントは「」である。"

char
単一の SBCS 文字

で定義されている英字、数字、特殊文字は指定できない。 *PL/I Language Reference*QUOTE" で定義されているアルファベット、数字、特殊文字は指定できない。

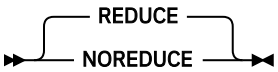
有効な文字を 1 つ指定する必要があります。

GRAPHIC オプションも指定されている場合、 QUOTE オプションは無視される。

REDUCE

REDUCE オプションは、たとえパディング・バイトが上書きされる可能性があるとしても、構造体への NULL 文字列の代入をより単純な操作に減らすことをコンパイラに許可することを指定する。

また、 REDUCE オプションを使うと、コンパイラーは、構造体に POINTER フィールドが含まれていても、マッチする構造体の割り当てを単純な集約移動に減らすことができる。



NOREDUCE オプションは、構造体への NULL 文字列の代入を、構造体の基本メンバへの一連の NULL 文字列の代入に分解しなければならないことを指定する。

NOREDUCE オプションでは、一緒に移動される要素が AREA または VARYING(Z) 属性を持つ場合、集約移動に削減できる BY NAME 割り当ては削減されない。

REDUCE オプションを使うと、構造体への NULL 文字列の代入に対して生成されるコード行数が少なくなり、通常、コンパイルが速くなり、コードの実行がずっと速くなる。 しかし、埋め込みバイトはゼロにリセットされることがあります。

例えば次の構造体では、*field12* と *field13* の間に 1 バイトの埋め込み があります。

```
dcl
 1 sample ext,
   5  field10          bin fixed(31),
   5  field11          bin fixed(15),
   5  field12          bit(8),
   5  field13          bin fixed(31);
```

次に、割り当て `sample = '';`について考えてみます。

NOREDUCE オプションでは、4 つの代入が生成され、パディングバイトは変更されない。

しかし、 REDUCE では、代入は 1 回に減るが、パディングバイトはゼロになる。

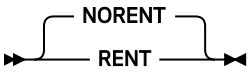
NOREDUCE オプションは、コンパイラの動作を OS PL/I や PL/I for MVS コンパイラに近づけます。 これらのコンパイラーは、構造体に POINTER フィールドが含まれていない限り、マッチする構造体の割り当てを単純な集約移動に減らしてしまう。 NOREDUCE オプションを付けると、このコンパイラーは同じように動作する。

RENT

コードが静的変数を変更しなければ、そのコードは「本質的に再入可能」です。 RENT オプションは、本来再入可能でないコードを再入可能にすることを指定する。

再入可能性について詳しくは、「*z/OS Language Environment Programming Guide*」を参照してください。 RENT オプションを使用する場合、リンケージ・エディターは生成されたオブジェクト・モジュールを直接処理することはできません。

注 : LP(64)オプションでは、 RENT オプションは無視される。事実上、 RENT は常にオンである。



NORENT オプションは、再入可能でないコードから再入可能なコードを生成しないことを指定する。 本質的に再入可能なコードは、そのまま再入可能です。

RENT オプションでコンパイルされた 1 つ以上のプログラムを含むモジュール（MAIN または FETCHABLE のいずれか）をリンクする場合、リンク・ステップで DYNAM=DLL と REUS=RENT を指定する必要があります。

NORENT 、LIMITS(EXTNAME (*n*)) (*n* <= 7)のオプションを指定すると、コンパイラが生成するテキスト・デッキは、古い PL/I コンパイラが生成するものと同じ形式になる。 他のオプションを使用する場合は、PDSE を使用する必要があります。

NORENT オプションで生成されるコードは、NOWRITABLE オプションも指定しないと、リエントラントにならないことがある。

NORENT を使用すると、コンパイラの一部の機能が使用できなくなる。 特に、以下の考慮事項に注意してください。

- DLL は構築できません。

- 再入可能な書き込み可能静的変数はサポートされません。
- STATIC ENTRY VARIABLE は INITIAL 値を持つことはできません。

RENT 、NORENT コードを混在させることができるが、以下の制限がある：

- RENT でコンパイルされたコードと NORENT でコンパイルされたコードは、EXTERNAL STATIC 変数を共有している場合、混在させることはできません。
- RENT でコンパイルされたコードは、 NORENT でコンパイルされたコード内の ENTRY VARIABLE セットを呼び出すことはできません。
- RENT でコンパイルされたコードは、NORENT でコンパイルされたコードを含むモジュールをフェッチできる。
- NORENT コードでコンパイルされたコードは、そのモジュールの ENTRY が OPTIONS(RENT) で宣言されていない限り、RENT でコンパイルされたコードを含むモジュールをフェッチすることはできない。
- NORENT WRITABLE でコンパイルされたコードは、 NORENT WRITABLE でコンパイルされたコードと外部変数 CONTROLLED を共有している場合、混在させることはできません。

上記の制約事項に従えば、以下を行うこともできます。

- `mnorent` と呼ばれる NORENT ルーチンは、静的にリンクされ、 `mrent` と呼ばれる RENT ルーチンを呼び出します。
- 次に、 RENT ルーチン `mrent` 、RENT でコンパイルされたエントリーポイントを持つ、別個にリンクされたモジュールをフェッチして呼び出す。

RESEXP

RESEXP オプションは、これによってある条件が発生してコンパイルが S レベル・メッセージで終了するとしても、コンパイラーがコンパイル時にすべての制限つき式を評価できるように指定します。

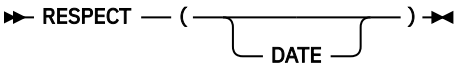
NORESEXP コンパイラー・オプションでは、コンパイラーは、INITIAL 値文節の式も含めて宣言されたすべての制限付き式を評価します。

例えば、NORESEXP オプションでは、コンパイラーは次のステートメントにフラグを立てません (ZERODIVIDE 例外が実行時に発生します)。

```
if preconditions_not_met then
   x = 1 / 0;
```

RESPECT

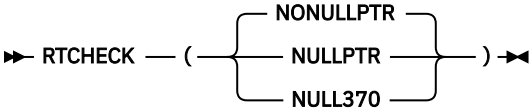
RESPECT オプションを指定すると、コンパイラーは DATE 属性の指定を尊重し、 DATE 組み込み関数の結果に DATE 属性を適用します。



デフォルトの RESPECT( ) を使用すると、コンパイラは DATE 属性の指定を無視し、 DATE 組み込み関数の結果に DATE 属性を適用しないようにします。

RTCHECK

RTCHECK オプションは、NULL ポインタが参照解除された場合、つまりポインタが変数値の変更または取得に使用された場合に、 ERROR 条件が強制的に発生するように追加コードを生成することを指定します。



NULLPTR
ヌル・ポインタ、つまり SYSNULL()に等しいポインタが参照解除された場合、ERROR 条件が強制的に発生するよう、余分なコードが生成される。

NULL370
DEFAULT(NULL370) 値に等しいポインタが参照解除された場合、ERROR 条件が強制的に発生するよう、余分なコードが生成される。 この値は、LP(32) では 'FF000000'x の 16 進値であり、LP(64) では '00000000_7FFFFBAD'x の 16 進値です。

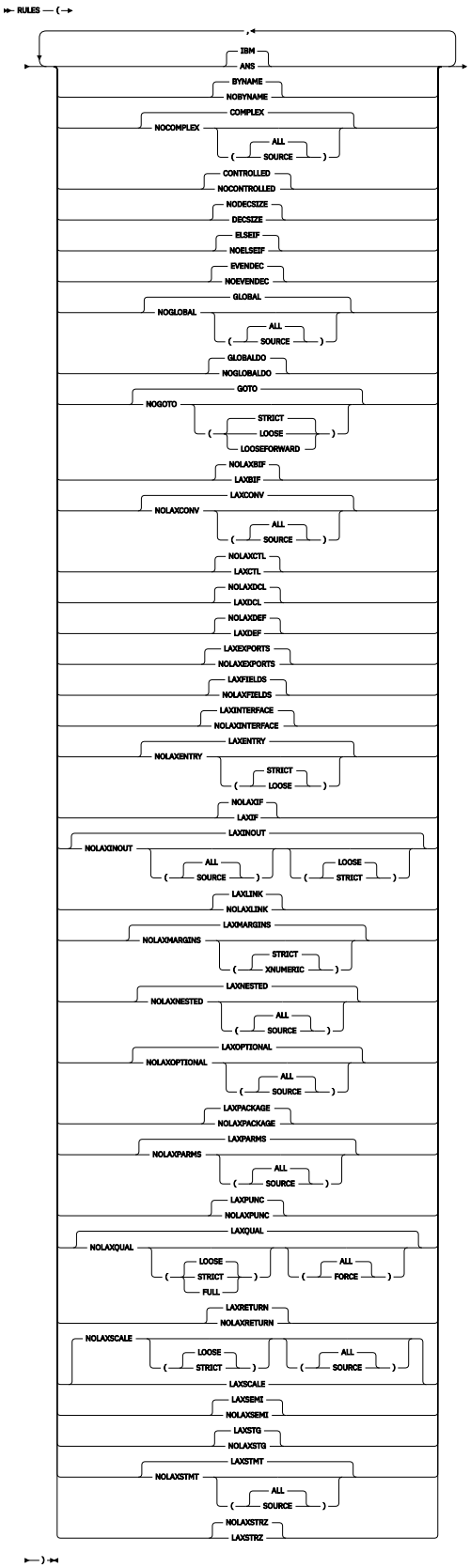
NONULLPTR
NULL ポインタが参照解除された場合に ERROR 条件を発生させるための余分なコードは生成されない。

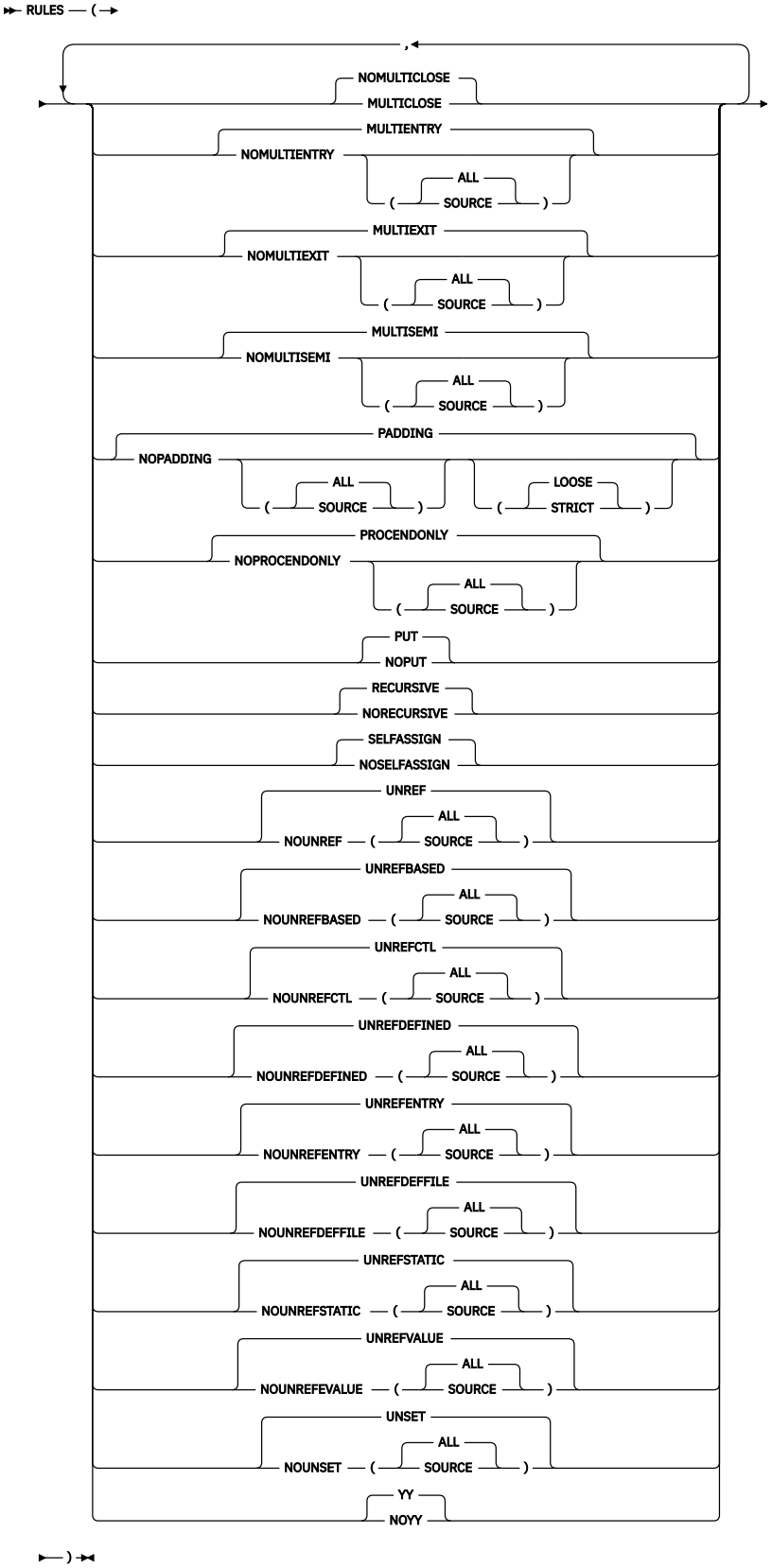
デフォルトは RTCHECK(NONULLPTR)です。

注 : ヌル・ポインタが参照解除されると、コンペア・アンド・トラップ・データ例外が発生する。

RULES

RULES オプションは、特定の言語機能を許可または不許可にし、代替手段が利用可能な場合にセマンティクスを選択できるようにします。 これは一般プログラミング・エラーの診断に役立ちます。


RULES
(
,
IBM
ANS
BYNAME
NOBYNAME
COMPLEX
NOCOMPLEX
ALL
SOURCE
CONTROLLED
NOCONTROLLED
NODECSIZE
DECSIZE
ELSEIF
NOELSEIF
EVENDEC
NOEVENDEC
GLOBAL
NOGLOBAL
GLOBALDO
NOGLOBALDO
GOTO
NOGOTO
STRICT
LOOSE
LOOSEFORWARD
NOLAXBIF
LAXBIF
LAXCONV
NOLAXCONV
NOLAXCTL
LAXCTL
NOLAXDCL
LAXDCL
NOLAXDEF
LAXDEF
LAXEXPORTS
NOLAXEXPORTS
LAXFIELDS
NOLAXFIELDS
LAXINTERFACE
NOLAXINTERFACE
LAXENTRY
NOLAXENTRY
NOLAXIF
LAXIF
LAXINOUT
NOLAXINOUT
LOOSE
STRICT
LAXLINK
NOLAXLINK
LAXMARGINS
NOLAXMARGINS
XNUMERIC
LAXNESTED
NOLAXNESTED
LAXOPTIONAL
NOLAXOPTIONAL
LAXPACKAGE
NOLAXPACKAGE
LAXPARMS
NOLAXPARMS
LAXPUNC
NOLAXPUNC
LAXQUAL
NOLAXQUAL
FULL
FORCE
LAXRETURN
NOLAXRETURN
NOLAXSCALE
LAXSCALE
LAXSEMI
NOLAXSEMI
LAXSTG
NOLAXSTG
LAXSTMT
NOLAXSTMT
NOLAXSTRZ
LAXSTRZ
)



IBM | ANS

IBM サブオプションの場合:

- 文字列データを必要とする操作では、 BINARY 属性を持つデータは BIT に変換される。
- OPTIONS 属性で変数を宣言すると、 ENTRY 属性が指定されます。

- FIXED BIN 、ゼロ以外のスケールファクターが許される。
- 組み込み関数（ADD, BINARY、 FIXED BIN など）の結果が FIXED BIN 属性を持っている場合、指定された、または暗示されたスケールファクターはゼロ以外である可能性があります。
- 算術演算については、以下のルールが適用される：

  1. オペランドの 1 つが FIXED BIN 、1 つが FIXED DEC の場合、 FIXED DEC オペランドのスケールファクターが 0 より大きくても、結果は FIXED BIN となる。

     例えば、 x が FIXED BIN(15) で、 y が FIXED BIN(31)の場合、次のようになる

     x + 2 は (16,0)の属性を持つ。 FIXED BIN

     x + は (20,4) の属性を持つ。 2.1 FIXED BIN

     y + 2 の属性は (31,0) FIXED BIN

     y + は (31,4) の属性を持つ。 2.1 FIXED BIN

  2. 両方のオペランドがスケールされていない場合（ FIXED BIN ）、結果はスケールされていない（ FIXED BIN ）。ただし、いくつかの割り算では、スケールファクターがゼロでない結果が得られることがある。

     例えば、 x が FIXED BIN(15) で、 y が FIXED BIN(31)の場合、次のようになる

     x / 2 は FIXED BIN(31,16)の属性を持つ

     y / 2 は FIXED BIN(31,0)の属性を持つ

ANS サブオプションの場合:

- 文字列データを必要とする操作では、 BINARY 属性を持つデータは CHARACTER に変換される。
- OPTIONS 属性を持つ変数を宣言すると、ENTRY 属性は暗黙指定されません。
- ゼロ以外のスケール因数は、FIXED BIN 宣言では **使用できません**。
- 組み込み関数（ADD, BINARY, FIXEDBIN など）の結果が FIXED BIN 属性を持つ場合、指定された、または暗示されたスケールファクターはゼロでなければなりません。
- 算術演算については、以下のルールが適用される：

  1. オペランドの 1 つが FIXED BIN 、1 つが FIXED DEC の場合、結果は次のようになる

     – FIXED DEC オペランドのスケールファクターがゼロの場合の FIXED BIN

     – FIXED DEC オペランドが 0 より大きいスケールファクタを持つ場合、FIXED DEC

     例えば、 x が FIXED BIN(15)で、 y が FIXED BIN(31)の場合、次のようになる

     x + 2 は FIXED BIN(16,0)の属性を持つ

     x + は FIXED DEC(7,1) の属性を持つ。 2.1

     y + 2 は FIXED BIN(31,0)の属性を持つ

     y + は FIXED DEC(13,1) の属性を持つ。 2.1

  2. 両オペランドが未補正の FIXED BIN の場合、除算でも結果は未補正の FIXED BIN となる。

     例えば、 x が FIXED BIN(15)で、 y が FIXED BIN(31)の場合、次のようになる

     x / 2 は FIXED BIN(15,0)の属性を持つ

     y / 2 は FIXED BIN(31,0)の属性を持つ

デフォルトは RULES(IBM) です。

BYNAME | NOBYNAME

NOBYNAME を指定すると、コンパイラーは E レベル・メッセージを 伴うすべての BYNAME 割り当てにフラグを付けます。

デフォルトは RULES(BYNAME) です。

COMPLEX | NOCOMPLEX

RULES(NOCOMPLEX) を指定すると、コンパイラーは、COMPLEX 属性または組み込み関数の使用、および接尾部が I の数値の使用に対してフラグを立てるようになります。

RULES(COMPLEX) を指定した場合、コンパイラーはそのようなステートメントにフラグを立てません。

すべて

ALL が指定された場合は、RULES(NOCOMPLEX) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース

SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(COMPLEX) です。 RULES(NOCOMPLEX) を指定した場合のデフォルトは ALL です。

CONTROLLED | NOCONTROLLED

NOCONTROLLED を指定した場合、コンパイラーが CONTROLLED 属性のすべての使用に対してフラグを立てます。

CONTROLLED を指定した場合、コンパイラーは CONTROLLED 属性の使用に対してフラグを立てません。

デフォルトは RULES(CONTROLLED) です。

DECSIZE | NODECSIZE

DECSIZE を指定すると、代入によって SIZE 条件が発生した場合に SIZE 条件が使用不可であると、コンパイラーは FIXED DECIMAL 変数に対する FIXED DECIMAL 式のすべての代入にフラグを立てます。

RULES(DECSIZE) が指定されると、コンパイラーは多くのメッセージを出力することがあります。 これは、SIZE が無効になっていると X = X + 1 形式のすべてのステートメントにフラグが立てられるためです (X が FIXED DECIMAL の場合)。

デフォルトは RULES(NODECSIZE) です。

ELSEIF | NOELSEIF

NOELSEIF を指定すると、直後に IF ステートメントが続く ELSE ステートメントに対しコンパイラーがフラグを立て、SELECT ステートメントとして書き直すように提案します。

一連のネストされた IF-THEN-ELSE ステートメントではなく SELECT ステートメントの使用を実施する場合に、このオプションを使用すると便利です。

また RULES(NOELSEIF) コンパイラー・オプションは、単純な DO-END で囲まれた IF ステートメントの直後に続く ELSE ステートメントにフラグを立てるようになりました。

デフォルトは RULES(ELSEIF) です。

EVENDEC | NOEVENDEC

NOEVENDEC を指定すると、コンパイラーは、偶数精度を指定するすべての FIXED DECIMAL 宣言にフラグを立てます。

デフォルトは RULES(EVENDEC) です。

GLOBAL | NOGLOBAL

RULES(NOGLOBAL) を使用すると、コンパイラーは、ブロックのネスト・サブプロシージャーで使用されていて、そのブロックで以下のいずれかのストレージ属性を使用して宣言されている変数にフラグを立てるようになります。

AUTOMATIC
CONTROLLED
DEFINED
PARAMETER
ASSIGNABLE STATIC

ルール(グローバル) を指定すると、コンパイラーはそのような使用にフラグを立てません。

すべて
ALL が指定された場合は、RULES(NOGLOBAL) のすべての違反にフラグが立てられます。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(GLOBAL) です。 RULES(NOGLOBAL) を指定した場合のデフォルトは ALL です。

GLOBALDO | NOGLOBALDO
NOGLOBALDO の指定は、コンパイラーに、親ブロックで宣言された制御変数を持つすべての DO ループにフラグを立てるように指示します。

親に DO ループ制御変数として宣言されている変数を使用すると、その変数が DO ループ制御変数として使用されているブロックで宣言されている場合よりも、ループのパフォーマンスが大幅に低下します。

デフォルトは RULES(GLOBALDO) です。

GOTO | NOGOTO
NOGOTO(STRICT) を指定すると、コンパイラーはラベル定数への GOTO ステートメントにフラグを立てます。ただし、その GOTO が ON ユニットを終了する場合を除きます。

NOGOTO(LOOSE) を指定すると、コンパイラーはラベル定数への GOTO ステートメントにフラグを立てます。ただし、その GOTO が ON ユニットを終了する場合や、ターゲットのラベル定数がその GOTO ステートメントと同じブロック内にある場合を除きます。

NOGOTO(LOOSEFORWARD) を指定すると、コンパイラーはラベル定数への GOTO ステートメントにフラグを立てます。ただし、その GOTO が ON ユニットを終了する場合や、ターゲットのラベル定数がその GOTO ステートメントと同じブロック内にあり、GOTO ステートメントの後ろに配置されている場合を除きます。

RULES (NOGOTO) を指定すると、コンパイラーは EXEC HANDLE CONDITION および EXEC SQL WHENEVER ステートメントの使用にエラーのフラグを立てます。

デフォルトは RULES(GOTO) です。 RULES(NOGOTO) を指定した場合のデフォルトは STRICT です。

LAXBIF | NOLAXBIF
LAXBIF を指定すると、空のパラメーター・リストを使用しない場合でも、 コンパイラーは NULL のようなコンテキスト宣言を組み込み関数のために 作成します。

デフォルトは RULES(NOLAXBIF) です。

LAXCONV | NOLAXCONV
RULES(LAXCONV) を指定した場合、コンパイラーはオペランドに算術型がない演算式にフラグを立てません。

RULES(NOLAXCONV) を指定した場合、コンパイラーはオペランドに算術型がない演算式にフラグを立てます。

すべて
ALL が指定された場合は、RULES(NOLAXCONV) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(LAXCONV) です。 RULES(NOLAXCONV) を指定した場合のデフォルトは ALL です。

LAXCTL | NOLAXCTL
LAXCTL を指定すると、固定エクステントを使用して CONTROLLED 変数を 宣言しても、CONTROLLED 変数を異なるエクステントに割り当てることができます。 NOLAXCTL を指定すると、CONTROLLED 変数を異なるエクステントに割り当てる場合に、そのエクステントをアスタリスクとして指定するか、非定数式として指定する必要があります。

NOLAXCTL が指定されていると、次のコードは不正になります。

```
dcl a bit(8) ctl;
alloc a;
alloc a bit(16);
```

ただし、次のコードは NOLAXCTL が指定されていても有効です。

```
dcl b bit(n) ctl;
dcl n fixed bin(31) init(8);
alloc b;
alloc b bit(16);
```

デフォルトは RULES(NOLAXCTL) です。

LAXDCL | NOLAXDCL
LAXDCL を指定すると、暗黙宣言が可能になります。 NOLAXDCL を指定すると、BUILTIN の場合と SYSIN および SYSPRINT ファイル の場合を除いて、暗黙宣言およびコンテキスト宣言はすべて禁止になります。

デフォルトは RULES(NOLAXDCL) です。

LAXDEF | NOLAXDEF
LAXDEF を指定すると、いわゆる無許可の定義が、コンパイラーのメッセージなしに受け入れられます (コンパイラーが通常作成する E レベル・メッセージも出ません)。

デフォルトは RULES(NOLAXDEF) です。

LAXENTRY | NOLAXENTRY
LAXENTRY を指定すると、非プロトタイプ・エントリー宣言が許可されます。 ノラクゼントリ を指定すると、コンパイラーはすべてのプロトタイプ化されていないエントリー宣言にフラグを立てます。 ただし、「CEE」、「DFH」、「DSN」、「EYU」、「IBM」、「PLI」、および「SQL」で始まる名前の宣言は除きます。

注 : ENTRY にパラメーターを指定しない場合は、単に ENTRY と宣言するのではなく、ENTRY() と宣言する必要があります。

STRICT
RULES(NOLAXENTRY(STRICT)) が指定されていると、コンパイラーは、OPTIONS(ASM) 属性を持っていてプロトタイプ化されていない エントリー宣言にフラグを立てます。

LOOSE
RULES(NOLAXENTRY(LOOSE)) が指定されていると、コンパイラーは、OPTIONS(ASM) 属性を持っていてプロトタイプ化されていない エントリー宣言にフラグを立てません。

デフォルトは RULES(LAXENTRY) です。 RULES(NOLAXENTRY) を指定した場合のデフォルトは STRICT です。

LAXEXPORTS | NOLAXEXPORTS
NOLAXEXPORTS を指定すると、コンパイラーは、エクスポートされるルーチンのリストを含む EXPORTS 文節を PACKAGE ステートメントで指定していない EXTERNAL ルーチンが含まれるコンパイルにフラグを立てるようになります。

LAXEXPORTS を指定した場合、コンパイラーはそのようなコンパイル単位にフラグを立てません。

デフォルトは RULES(LAXEXPORTS) です。

LAXFIELDS | NOLAXFIELDS
RULES(NOLAXFIELDS) を指定すると、コンパイラーは以下の SQL ステートメントにフラグを立てるようになります。

- EXEC SQL SELECT*
- EXEC SQL INSERT INTO <name> (この後に括弧で囲まれたフィールド名のリストが続かないもの)

RULES(LAXFIELDS) を指定すると、コンパイラーはこのようなステートメントにフラグを立てなくなります。

デフォルトは RULES(LAXFIELDS) です。

LAXIF | NOLAXIF

RULES(NOLAXIF) を指定すると、コンパイラーは、BIT(1) NONVARYING 属性を持たないすべての IF、WHILE、UNTIL、および WHEN 文節にフラグを立てます。 また、この指定によって、コンパイラーは x=y=z 形式の割り当てに対してフラグを立てますが、x=(y=z) 形式の割り当てに対してはフラグを立てません。

NOLAXIF の場合、以下のすべてのコードにフラグが立てられます。

```
dcl i fixed bin;
dcl b bit(8);
   .
   .
   .
if i then …
if b then …
```

デフォルトは RULES(NOLAXIF) です。

LAXINOUT | NOLAXINOUT

ノーラックスインアウト を指定すると、コンパイラーは、明示的な INOUT 属性を持つすべてのパラメーターが入力 (および場合によっては出力) パラメーターであると想定し、そのようなパラメーターが設定されていないとコンパイラーが判断した場合に警告を発行します。

ルール・オブ・ノラクシンアウトのサブオプションとして STRICT を指定すると、コンパイラーは アドルで割り当て可能 パラメーターが INOUT であると想定し、設定されていない可能性がある場合はその使用にフラグを立てます。

すべて
すべてを指定すると、ルール (ノラクシンアウト) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(LAXINOUT) です。 ルール (ノラクシンアウト) を指定すると、ルースと すすべてがデフォルトになります。

LAXINTERFACE | NOLAXINTERFACE

NOLAXINTERFACE を指定すると、コンパイラーは、外部の非 MAIN PROCEDURE のそれぞれの一致する明示宣言を含まないコードにフラグを立てるようになります。

NOLAXINTERFACE を指定した場合は、以下のようにする必要があります。

- PACKAGE ステートメントがある場合、エクスポートされるどの非 MAIN プロシージャーにもパッケージ・スコープの明示宣言が必要であり、その明示宣言はプロシージャー・ステートメントから派生する暗黙宣言と一致しなければなりません。
- PACKAGE ステートメントがない場合、最外部プロシージャーは、MAIN でなければその明示宣言を含んでいなければならず、その明示宣言は対応するプロシージャー・ステートメントから派生する暗黙宣言と一致しなければなりません。

デフォルトは RULES(LAXINTERFACE) です。

LAXLINK | NOLAXLINK

NOLAXLINK を指定すると、以下のいずれかが一致 しない場合に、コンパイラーは、2 つの ENTRY 変数または定数の代入または比較 のすべてにフラグを立てます。

- パラメーター記述リスト

  例えば、A1 が ENTRY(CHAR(8)) と宣言され、A2 が ENTRY(POINTER) VARIABLE と宣言されている場合に、RULES(NOLAXLINK) が指定されると、コンパイラーは A1 を A2 に代入しようとする処理にフラグを立てます。

- RETURNS 属性

  例えば、A3 が ENTRY RETURNS(FIXED BIN(31)) と宣言されていて、A4 が RETURNS 属性なしで ENTRY VARIABLE と宣言されている場合に、RULES(NOLAXLINK) が指定されると、コンパイラーは A3 を A4 に代入しようとする処理にフラグを立てます。

- LINKAGE および他の OPTIONS サブオプション

  例えば、A5 が ENTRY OPTIONS(ASM) と宣言されていて、A6 が OPTIONS 属性なしで ENTRY VARIABLE と宣言されている場合に、RULES(NOLAXLINK) が指定されると、コンパイラーは A5 を A6 に代入しようとする処理にフラグを立てます。 これは、A5 の宣言における OPTIONS(ASM) では、A5 が LINKAGE(SYSTEM)) を持つことが暗黙指定されるのに対して、A6 には OPTIONS 属性がないため、デフォルトで A6 が LINKAGE(OPTLINK) を持つことになるためです。

デフォルトは RULES(LAXLINK) です。

LAXMARGINS | NOLAXMARGINS
NOLAXMARGINS を指定すると、STRICT および XNUMERIC サブオプションの設定に応じて、右マージンの後に非ブランク文字がある行にコンパイラーがフラグを立てます。 このオプションは、誤って右マージンへ押し出された終了コメントなどのコードの検出に役立ちます。

いずれかのプリプロセッサーと共に NOLAXMARGINS および STMT オプションを使用すると、NOLAXMARGINS オプションによってフラグが立てられるステートメントは、ステートメント番号ゼロとして報告されます (ステートメントの番号付けはすべてのプリプロセッサーが終了した後でのみ行われますが、マージンの外側のテキストの検出はソースが読み取られるとすぐに行われるためです)。

STRICT
STRICT が指定された場合、コンパイラーは、右マージンの後に非ブランク文字を含むすべての行にフラグを立てます。

XNUMERIC
XNUMERIC が指定された場合、コンパイラーは、右マージンの後に非ブランク文字を含むすべての行にフラグを立てます (ただし、右マージンが 72 桁目であり、73 桁目から 80 桁目までのすべてに数字が含まれる場合を除く)。

デフォルトは RULES(LAXMARGINS) です。 RULES(NOLAXMARGINS) を指定した場合のデフォルトは STRICT です。

LAXNESTED | NOLAXNESTED
RULES(LAXNESTED) を指定した場合、サブプロシージャーの後にあるプロシージャー内の実行可能コードにコンパイラーはフラグを立てません。

RULES(NOLAXNESTED) を指定した場合、プロシージャー内のサブプロシージャーの後にあるすべての実行可能コードにコンパイラーがフラグを立てます。

すべて
ALL が指定された場合は、RULES(NOLAXNESTED) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(LAXNESTED) です。 RULES(NOLAXNESTED) を指定した場合のデフォルトは ALL です。

LAXOPTIONAL | NOLAXOPTIONAL
RULES(NOLAXOPTIONAL) を指定すると、コンパイラーは、パラメーターが OPTIONAL と宣言されていない OMITTED および PRESENT 組み込み関数の使用に対してフラグを立てるようになります。

すべて
ALL が指定された場合は、RULES(NOLAXOPTIONAL) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

例えば、以下のように指定します。

```
Test: proc( a, b );
    declare a fixed bin(31);
    declare b fixed bin(31) optional;
```

RULES(NOLAXOPTIONAL) が指定された場合、コンパイラーは OMITTED(a) にフラグを立てます。

デフォルトは RULES(LAXOPTIONAL) です。 RULES(NOLAXOPTIONAL) を指定した場合のデフォルトは ALL です。

LAXPACKAGE | NOLAXPACKAGE
NOLAXPACKAGE を指定すると、コンパイラーは、明示的 PACKAGE ステートメントを含まないコンパイルにフラグを立てるようになります。

LAXPACKAGE を指定した場合、コンパイラーはそのようなコンパイル単位にフラグを立てません。

デフォルトは RULES(LAXPACKAGE) です。

LAXPARMS | NOLAXPARMS
RULES(NOLAXPARMS) を指定すると、コンパイラーは、INOUT/INONLY/OUTONLY 属性のいずれも明示的に指定されておらず、別の属性 (BYVALUE など) によって暗黙的にも指定されていないパラメーターにフラグを立てるようになります。

RULES(LAXPARMS) を指定した場合、コンパイラーはそのようなステートメントにフラグを立てません。

すべて
ALL が指定された場合は、RULES(NOLAXPARMS) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(LAXPARMS) です。 RULES(NOLAXPARMS) を指定した場合のデフォルトは ALL です。

LAXPUNC | NOLAXPUNC
NOLAXPUNC を指定すると、コンパイラーは想定される句読点が欠落している場所に E レベル・メッセージのフラグを立てます。

例えば、ステートメント `I = (1 * (2);` があるとします。 この場合、コンパイラーはセミコロンの前に右括弧があるものと想定します。 RULES(NOLAXPUNC) が指定されると、このステートメントに E レベル・メッセージのフラグが立てられます。 それ以外の場合は W レベル・メッセージのフラグが立てられます。

デフォルトは RULES(LAXPUNC) です。

LAXQUAL | NOLAXQUAL
NOLAXQUAL を指定すると、コンパイラーは、レベル 1 以外の構造体メンバーへのすべての参照にフラグを立てます。 「CEE」、「DFH」、「DSN」、「EYU」、「IBM」、「PLI」、および「SQL」 で始まる名前の参照は、NOLAXQUAL 検査から除外されます。

LOOSE
RULES(NOLAXQUAL(LOOSE)) を指定すると、コンパイラーは、レベル 1 でなくドット修飾のない構造体メンバーへのすべての参照にフラグを立てるようになります。 次の例を考えてください。

```
dcl
  1 a,
    2 b,
      3 b fixed bin,
      3 c fixed bin;

c   = 11;   /* would be flagged */
b.c = 13;   /* would not be flagged */
a.c = 17;   /* would not be flagged */
```

STRICT
RULES(NOLAXQUAL(STRICT)) を指定すると、コンパイラーは、レベル 1 の名前を含まない構造体メンバーへのすべての参照にフラグを立てるようになります。 次の例を考えてください。

```
dcl
  1 a,
    2 b,
      3 b fixed bin,
      3 c fixed bin;
```

```
c   = 11;   /* would be flagged */
b.c = 13;   /* would be flagged */
a.c = 17;   /* would not be flagged */
```

FULL
RULES(NOLAXQUAL(FULL)) を指定すると、コンパイラーは、親の名前のすべては含まない構造体メンバーへのすべての参照にフラグを立てるようになります。

すべて
ALL が指定された場合は、RULES(NOLAXQUAL) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

FORCE
FORCE が指定された場合は、FORCE(NOLAXQUAL) 属性を持つ構造体で発生した違反にのみフラグが立てられます。

デフォルトは RULES(LAXQUAL) です。 RULES(NOLAXQUAL) を指定した場合のデフォルトは LOOSE および ALL です。

LAXRETURN | NOLAXRETURN
NOLAXRETURN が指定されている場合、RETURN ステートメントが以下のいずれかのように使用されているときに、 エラー状態を引き起こすコードをコンパイラーが生成します。

- RETURNS オプションを指定せずにコーディングされたプロシージャー内に式を持つ場合
- RETURNS オプションを指定せずにコーディングされたプロシージャー内に式を持たない場合

コードが プロシージャーにおいて RETURNS 属性を伴って END ステートメントに到達した場合にエラーが発生します。

デフォルトは RULES(LAXRETURN) です。

LAXSCALE | NOLAXSCALE
NOLAXSCALE を指定すると、コンパイラーは、 $q < 0$ または $p < q$ である FIXED DEC (*p,q*)宣言にフラグを立てる。

また、 $P < 0$ の場合、コンパイラーは丸め (*X*,*P*) にフラグを立てます。

コンパイラが ROUND (*x,p*)のフラグを立てたときに出力されるメッセージは、FIXED DEC (*p,q*)宣言のフラグを立てたときに出力されるメッセージとは異なる。 このため、EXIT オプションを使用すれば、ROUND(*x*,*p*) にフラグが立てられているときに発行されるメッセージを抑止して、他の疑わしい宣言に対するメッセージを維持できます。

RULES(NOLAXSCALE) のサブオプションとして STRICT を指定した場合、コンパイラーは FIXED BIN(p,q) (ここで q > 0) にもフラグを立てます。 LOOSE がデフォルトです。

すべて
ALL が指定された場合は、RULES(NOLAXSCALE) のすべての違反にフラグが立てられます。 ALL はデフォルトです。

ソース
SOURCE が指定された場合は、1 次ソース・ファイルで発生した違反にのみフラグが立てられます。

デフォルトは RULES(NOLAXSCALE) です。 RULES(NOLAXSCALE)を指定すると、ALL がデフォルトになる。

LAXSEMI | NOLAXSEMI
NOLAXSEMI を指定すると、コンパイラーは、コメント内に現れるすべてのセミコロン にフラグを立てます。

デフォルトは RULES(LAXSEMI) です。

LAXSTG | NOLAXSTG
ノーラックスティージーを指定すると、コンパイラーは、B がパラメーターである場合でも、変数 A が ADDR (B) および STG (A)> STG (B) で BASED として宣言されている宣言にフラグを立てます。

NOLAXSTG が指定されているときでも、B に添え字があれば、IBM2402I E レベル・メッセージは生成されません。

B が AUTOMATIC ストレージまたは STATIC ストレージにあるとしたらコンパイラーはこの種の問題に既にフラグを立てているはずですが、B がパラメーターの場合、コンパイラーはデフォルトではこれにフラグを立てません (プログラマーによっては、実引数を記述しないでプレースホルダー属性で B を宣言するためです)。 パラメーターと引数の宣言が合致する (または合致しなければならない) 場合に RULES(NOLAXSTG) を指定すると、より多くのストレージ・オーバーレイ問題を検出できるようになります。

デフォルトは RULES(LAXSTG) です。

LAXSTMT | NOLAXSTMT

NOLAXSTMT が指定された場合、コンパイラーは、複数のステートメントを持つすべての行にフラグを立てます。

すべて

RULES(NOLAXSTMT(ALL)) が指定された場合、コンパイラーはすべての NOLAXSTMT 違反にフラグを立てます。 ALL はデフォルトです。

ソース

RULES(SOURCE) が指定された場合、コンパイラーは 1 次ソース・ファイルに含まれる違反にのみフラグを立てます。

また、NOLAXSTMT は、キーワードのリスト (空の場合もある) が指定された EXCEPT を受け入れます。そのキーワードは、行の 2 番目のステートメントがそのキーワードのいずれかで始まる場合にフラグが立てられないものです。 これにより、例えば、IF … THEN ステートメントと同じ行で DO; を使用できるようになります。

EXCEPT では以下のキーワードを使用できます。

- allocate
  assert
  attach
  begin
  呼び出し
  キャンセル
  close
  declare
  定義
  遅延
  削除
  detach
  表示
  do
  else
  終了
  終了
  fetch
  flush
  空き
  取得する
  go
  goto
- halt
  の場合
  iterate
  leave
  locate
  オン
  ・オープン
  otherwise
  put
  読み取り

reinit
リリース
resignal
返す
revert
rewrite
select
シグナル
停止
unlock
待ち
いつ
書き込み

デフォルトは RULES(LAXSTMT) です。 RULES(NOLAXSTMT) を指定した場合のデフォルトは ALL です。

LAXSTRZ | NOLAXSTRZ
LAXSTRZ を指定すると、コンパイラーは、余分のビットがすべてゼロである (または、余分の文字がすべてブランクで ある) 場合に、長すぎる定数値に初期化された、または割り当てられた ビット変数または文字変数にフラグを立てません。

デフォルトは RULES(NOLAXSTRZ) です。

MULTICLOSE | NOMULTICLOSE
NOMULTICLOSE を指定すると、コンパイラーは、E レベル・メッセージを伴うステートメントの複数のグループを強制的に閉じるステートメントすべてにフラグを立てます。

デフォルトは RULES(NOMULTICLOSE) です。

MULTIENTRY | NOMULTIENTRY
NOMULTIENTRY が指定された場合、コンパイラーは、複数の ENTRY ステートメントを含むコードにフラグを立てます。

すべて
NOMULTIENTRY(ALL) が指定された場合、コンパイラーは RULES(NOMULTIENTRY) のすべての違反にフラグを立てます。

ソース
NOMULTIENTRY(SOURCE) が指定された場合、コンパイラーは 1 次ソース・ファイルで発生した違反にのみフラグを立てます。

デフォルトは RULES(MULTIENTRY) です。 RULES(NOMULTIENTRY) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

MULTIEXIT | NOMULTIEXIT
NOMULTIEXIT が指定された場合、コンパイラーは、複数の RETURN ステートメントを含むコードにフラグを立てます。

すべて
NOMULTIEXIT(ALL) が指定された場合、コンパイラーは RULES(NOMULTIEXIT) のすべての違反にフラグを立てます。

ソース
NOMULTIEXIT(SOURCE) が指定された場合、コンパイラーは 1 次ソース・ファイルで発生した違反にのみフラグを立てます。

デフォルトは RULES(MULTIEXIT) です。 RULES(NOMULTIEXIT) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

MULTISEMI | NOMULTISEMI
NOMULTISEMI が指定された場合、コンパイラーはコメントやストリング以外での複数のセミコロンを含むすべての行にフラグを立てます。

すべて
NOMULTISEMI(ALL) が指定された場合、コンパイラーは RULES(NOMULTISEMI) のすべての違反にフラグを立てます。

ソース
NOMULTISEMI(SOURCE) が指定された場合、コンパイラーは 1 次ソース・ファイルで発生した違反にのみフラグを立てます。

デフォルトは RULES(MULTISEMI) です。 RULES(NOMULTISEMI) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

PADDING | NOPADDING
NOPADDING を指定すると、コンパイラーは、埋め込みを含む構造体にフラグを立てます。

すべて
NOPADDING(ALL) を指定すると、コンパイラーは、LOOSE/STRICT サブオプションの設定に応じて、すべての RULES(NOPADDING) 違反にフラグを立てるようになります。

ソース
NOPADDING(SOURCE) が指定された場合、コンパイラーは 1 次ソース・ファイルで発生した違反にのみフラグを立てます。

LOOSE
NOPADDING(LOOSE) を指定すると、コンパイラーは、内部埋め込みが含まれる構造体にフラグを立てるようになります。

STRICT
NOPADDING(STRICT) を指定すると、コンパイラーは、以下のいずれかが含まれる構造体にフラグを立てるようになります。

- 内部埋め込み
- 構造体の前の埋め込み (いわゆるハング・バイトまたはハング・ビット)。 これは、構造体は n バイト位置合わせだが、n バイト位置合わせでの最初のエレメントの前が n バイトの倍数になっていない場合、または、構造体はバイト (以上の) 位置合わせでの最初のエレメントの前が 8 ビットの倍数になっていない場合に行われます。
- 構造体の後の埋め込み。 これは、構造体のサイズがその位置合わせの倍数でない場合、または、構造体はバイト (以上の) 位置合わせでの最後のエレメントの後が 8 ビットの倍数になっていない場合に行われます。

デフォルトは RULES(PADDING) です。 RULES(NOPADDING) を指定した場合のデフォルトのサブオプションのサブオプションは ALL および LOOSE です。

PROCENDONLY | NOPROCENDONLY
NOPROCENDONLY を指定すると、PROCEDURE を終了するすべての END ステートメントにフラグが立てられます。ただし、その END ステートメントが PROCEDURE を指定していない場合、つまり END キーワードの直後にセミコロンがある場合に限られます。

すべて
RULES(NOPROCENDONLY(ALL)) が指定された場合、コンパイラーはすべての NOPROCENDONLY 違反にフラグを立てます。 ALL はデフォルトです。

ソース
RULES(NOPROCENDONLY(SOURCE)) が指定された場合、コンパイラーは 1 次ソース・ファイルに含まれる違反にのみフラグを立てます。

デフォルトは RULES(PROCENDONLY) です。 RULES(NOPROCENDONLY) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

PUT | NOPUT

NOPUT を指定すると、コンパイラーは PUT FILE 文にフラグを立てる。 PUT 文が DO-block や IF-THEN などでネストされているか、ネストされていないかに応じて、2 つのメッセージのいずれかを発行する。

デフォルトは RULES(PUT)。

RECURSIVE | NORECURSIVE
NORECURSIVE を指定した場合、RECURSIVE 属性のすべての使用に対して、 または自分自身を直接呼び出すプロシージャーの使用に対して、コンパイラーがフラグを立てます。

RECURSIVE を指定した場合、RECURSIVE 属性の使用に対しても、自分自身を直接呼び出すプロシージャーの使用に対しても、コンパイラーはフラグを立てません。

注 : RULES(NORECURSIVE) と DFT(RECURSIVE) を同時に使用しないでください。

デフォルトは RULES(RECURSIVE) です。

SELFASSIGN | NOSELFASSIGN
NOSELFASSIGN が指定されている場合、コンパイラーは、ソースとターゲットが同じになっているすべての割り当てにフラグを立てます。

デフォルトは RULES(SELFASSIGN) です。

UNREF | NOUNREF
NOUNREF を指定すると、参照されないレベル 1 AUTOMATIC 変数、および参照されるサブエレメントを含まない構造体または共用体であるレベル 1 AUTOMATIC 変数に、コンパイラーがフラグを立てます。 NOUNREF が指定された場合は、接頭部 DSN、DFH、EYU、および SQL のいずれかで始まる名前を持つ変数は無視されます。

すべて
RULES(NOUNREF(ALL)) を指定した場合、すべての非参照変数にコンパイラーがフラグを立てます。 NOUNREF が指定されるときは、ALL がデフォルトです。

ソース
RULES(NOUNREF(SOURCE)) を指定した場合、INCLUDE ファイル内で宣言されていない非参照変数にコンパイラーがフラグを立てます。

デフォルトは RULES(UNREF) です。 RULES(NOUNREF) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

UNREFBASED | NOUNREFBASED
NOUNREFBASED を指定すると、コンパイラーは BASED ストレージにある参照されない BASED 変数にフラグを立てる。

すべて
RULES(NOUNREFBASED(ALL)) が指定された場合、コンパイラーはすべての非参照 BASED 変数にフラグを立てます。

ソース
RULES(NOUNREFBASED(SOURCE)) が指定された場合、コンパイラーは INCLUDE ファイル内で宣言されていない非参照 BASED 変数にフラグを立てます。

デフォルトは RULES(UNREFBASED) です。 RULES(NOUNREFBASED) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

UNREFCTL | NOUNREFCTL
NOUNREFCTL を指定すると、コンパイラーは参照されない CTL 変数にフラグを立てる。

すべて
RULES(NOUNREFCTL(ALL)) が指定された場合、コンパイラーはすべての非参照 CTL 変数にフラグを立てます。

ソース
RULES(NOUNREFCTL(SOURCE)) が指定された場合、コンパイラーは INCLUDE ファイル内で宣言されていない非参照 CTL 変数にフラグを立てます。

デフォルトは RULES(UNREFCTL) です。 RULES(NOUNREFCTL) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

UNREFDEFINED | NOUNREFDEFINED
NOUNREFDEFINED を指定すると、コンパイラーは参照されない DEFINED 変数にフラグを立てます。

すべて
RULES(NOUNREFDEFINED(ALL)) が指定された場合、コンパイラーはすべての非参照 DEFINED 変数にフラグを立てます。

ソース
RULES(NOUNREFDEFINED(SOURCE)) が指定された場合、コンパイラーは INCLUDE ファイル内で宣言されていない非参照 DEFINED 変数にフラグを立てます。

デフォルトは RULES(UNREFDEFINED) です。 RULES(UNREFDEFINED) を指定した場合のデフォルトのサブオプションのサブオプションは ALL です。

UNREFENTRY | NOUNREFENTRY
NOUNREFENTRY を指定すると、コンパイラーは参照されない ENTRY 定数にフラグを立てる。

すべて
RULES(NOUNREFENTRY(ALL)) が指定された場合、コンパイラーはすべての非参照 ENTRY 定数にフラグを立てます。

ソース
RULES(NOUNREFENTRY(SOURCE)) が指定された場合、コンパイラーは INCLUDE ファイル内で宣言されていない非参照 ENTRY 定数にフラグを立てます。

デフォルトは RULES(UNREFENTRY) です。 RULES(NOUNREFENTRY)を指定すると、デフォルトのサブオプションは ALL になる。

UNREFFILE | NOUNREFFILE
NOUNREFFILE を指定すると、コンパイラーは参照されない FILE 定数にフラグを立てる。

すべて
RULES(NOUNREFFILE(ALL)) が指定された場合、コンパイラーはすべての非参照 FILE 定数にフラグを立てます。

ソース
RULES(NOUNREFFILE(SOURCE)) が指定された場合、コンパイラーは INCLUDE ファイル内で宣言されていない非参照 FILE 定数にフラグを立てます。

デフォルトは RULES(UNREFFILE) です。 RULES(NOUNREFFILE)を指定すると、デフォルトのサブオプションは ALL になる。

UNREFSTATIC | NOUNREFSTATIC
NOUNREFSTATIC を指定すると、コンパイラーは、PLIXOPT と PLITAB を除くすべての非参照 STATIC 変数にフラグを立てるようになります。

すべて
RULES(NOUNREFSTATIC(ALL)) が指定された場合、コンパイラーはすべての非参照 STATIC 変数にフラグを立てます。

ソース
RULES(NOUNREFSTATIC(SOURCE)) が指定された場合、コンパイラーは INCLUDE ファイル内で宣言されていない非参照 STATIC 変数にフラグを立てます。

デフォルトは RULES(UNREFSTATIC) です。 RULES(NOUNREFSTATIC)を指定すると、デフォルトのサブオプションは ALL になる。

非参照値｜名詞参照値
NOUNREFVALUE を指定すると、コンパイラーは参照されない VALUE 定数にフラグを立てる。

すべて
RULES(NOUNREFVALUE(ALL))を指定すると、コンパイラーは参照されないすべての VALUE 定数にフラグを立てる。

ソース
RULES(NOUNREFVALUE(SOURCE))を指定すると、コンパイラーは INCLUDE ファイルで宣言されていない参照されない VALUE 定数にフラグを立てる。

デフォルトは RULES(UNREFVALUE)。 RULES(NOUNREFVALUE)を指定すると、デフォルトのサブオプションは ALL になる。

アンセット｜名詞セット

NOUNSET を指定すると、コンパイラーは、値が明示的に割り当てられていない可能性のある自動変数の使用にフラグを立てます。

コンパイラーは、以下の可能性のある問題にフラグを立てるタイミングを決定する際に、このロジックに従います。

- 自動変数に値が割り当てられていない可能性がある場合
  - 初期化されている場合
    - ルール（未設定）を指定した場合、その値の使用にフラグは立てられません。
    - ルール　(名詞) を指定すると、その値の使用にフラグが立てられます。
  - 初期化されていない場合は、その値の使用にフラグが立てられます。
- それ以外の場合は、その値の使用にフラグは立てられません。

このオプションは、ユーザーが INIT 値をすべてに無意識に適用しないようにすることを目的としています。

デフォルトは RULES (UNSET) です。 ルール (名詞) を指定した場合、デフォルトは ALL です。

YY | NOYY
NOYY を指定した場合、コンパイラーは以下のような 2 桁の年の使用に対してフラグを立てます。

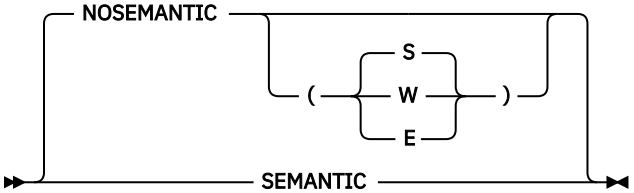
- 年が YY または ZY で指定された日時パターン
- 日時パターンを持たない DATE 属性 (パターン YYMMDD が暗黙指定されるため)
- 組み込み関数 Y4DATE、Y4JULIAN、および Y4YEAR
- DATE 組み込み関数
- 時間帯引数を持つ日時関数

デフォルトは RULES(YY) です。

デフォルト: RULES (IBM BYNAME COMPLEX CONTROLLED NODECSIZE EVENDEC ELSEIF GLOBAL GLOBALDO GOTO NOLAXBIF LAXCONV NOLAXCTL NOLAXDCL NOLAXDEF LAXENTRY LAXEXPORTS NOLAXFIELDS NOLAXIF LAXINOUT LAXINTERFACE LAXLINK LAXNESTED LAXOPTIONAL LAXPACKAGE LAXPARMS LAXPUNC LAXMARGINS(STRICT) LAXQUAL LAXRETURN NOLAXSCALE LAXSEMI LAXSTG LAXSTMT NOLAXSTRZ NOMULTICLOSE MULTIENTRY MULTIEXIT MULTISEMI PADDING PROCENDONLY RECURSIVE SELFASSIGN UNREF UNREFBASED UNREFCTL UNREFDEFINED UNREFENTRY UNREFFILE UNREFSTATIC YY)

SEMANTIC

SEMANTIC オプションは、セマンティック・チェック・ステージの実行が、このステー ジの前に発行されたメッセージの重大度に依存することを指定する。



省略形: SEM、NSEM

SEMANTIC
NOSEMANTIC(S) と同等。

NOSEMANTIC
処理は構文検査の後で停止されます。 セマンティック検査は実行されません。

NOSEMANTIC (S)
重大エラーまたは回復不能エラーが検出された場合は、 セマンティック検査は行われません。

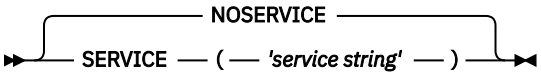
NOSEMANTIC (E)
エラー、重大エラー、または回復不能エラーが検出された場合は、 セマンティック検査は行われません。

NOSEMANTIC (W)
警告、エラー、重大エラー、または回復不能エラーが検出された場合は、 セマンティック検査は行われません。

ある種の重大エラーが見つかった場合は、セマンティック検査は実行されません。 すべての参照が正しく解決されることをコンパイラーが確認できない 場合 (例えば、組み込み関数またはエントリーの参照に引数が少なすぎる 場合)、組み込み関数またはエントリーの参照の中の引数の妥当性は検査されません。

サービス

SERVICE オプションは、オブジェクト・モジュール内にストリングを 配置します。 このストリングは、オブジェクトのリンク先であるロード・モジュールとともに メモリーにロードされ、また LE ダンプにトレースバックが組み込まれている 場合、このストリングはそのトレースバックにも組み込まれます。



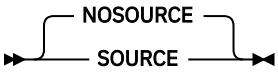
省略形: SERV、NOSERV

ストリングの長さは 64 文字に制限されます。

ロケールが異なっていてもストリングを読み取ることができるように、 インバリアント文字セットからの文字だけを使用する必要があります。

ソース

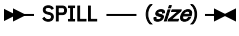
SOURCE オプションは、コンパイラーが、コンパイラー・リスト内にソース・プログラムのリストを 組み込むことを指定します。 リストされるソース・プログラムは、オリジナルのソース入力か、 あるいは (任意のプリプロセッサーが使用される場合は) プリプロセッサーの出力です。 ただし、LISTVIEW オプションは、このオプションが有効な場合にリストに表示される内容を変更することができます。



省略形: S、NS

SPILL

SPILL オプションは、コンパイルに使用する予備域のサイズを指定します。 一度に使用されるレジスターの数が多すぎる場合、コンパイラーは レジスターの一部を予備域と呼ばれる一時記憶域にダンプします。

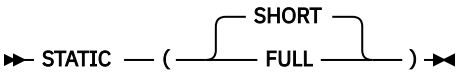


省略形: SP

予備域を拡張する必要がある場合、どのサイズまで増やす必要が あるかを示すコンパイラー・メッセージが出されます。 ソース・プログラムに必要な予備域が分かったら、必要なサイズ (バイト) を上記の構文図のとおりに指定できます。 最大予備域サイズは 3900 です。 通常、このオプションを指定する必要があるのは、OPTIMIZE を指定して非常に大規模なプログラムをコンパイルする場合だけです。

STATIC

STATIC オプションは、INTERNAL STATIC 変数を (その変数が参照されないとしても) オブジェクト・モジュールに保持するかどうかを制御します。



SHORT
INTERNAL STATIC は、使用される場合のみオブジェクト・モジュールに保管されます。

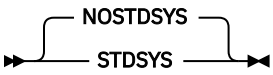
FULL
INITIAL の All INTERNAL STATIC は、オブジェクト・モジュールに保管されます。

INTERNAL STATIC 変数が「eyecatcher」として使用される場合は、STATIC(FULL) オプションを指定して、生成されたオブジェクト・モジュールにその変数を含める必要があります。

STDSYS

STDSYS オプションは、コンパイラーが SYSPRINT ファイルを C 標準出力ファイルと、 SYSIN ファイルを C 標準入力ファイルと等しくするように指定する。

注 : LP(64)オプションでは、 STDSYS オプションは無視される。事実上、 STDSYS は常にオンである。

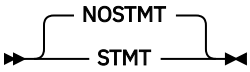


STDSYS オプションを使えば、 PL/I と C が混在したアプリケーションの開発とデバッグが容易になるかもしれない。

SYSPRINT が標準出力と等号化されるとき、その LINESIZE は 132（C で許容される最大値）より大きくすることはできない。

STMT

STMT オプションは、ソース・プログラム内のステートメントをカウントし、このステートメント番号を使用して、AGGREGATE、ATTRIBUTES、SOURCE、および XREF オプションを用いて作成されたコンパイラー・リスト内のステートメントを識別することを指定します。



デフォルトは NOSTMT です。

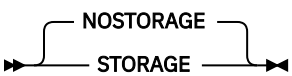
STMT オプションが指定されている場合、リストは NOSTMT の場合と同じ形式になります。ただし、リストの特定の部分では、ソース行とファイル番号を示す桁のすぐ右側に追加の桁があります。 この追加の列には、もしあれば、論理 PL/I ステートメント番号 が表示されます。

特に、SOURCE リスト、AGGREGATE リスト、ATTRIBUTES/XREF リスト、GENERIC リスト、DECOMP リスト、およびすべてのコンパイラー・メッセージを表示するリストには、そのような列があります。「参照:」または「セット:」情報が表示される場合、表示される番号は常にソース行とファイル番号になります。

GOSTMT オプションは存在しないことに注意してください。 実行時にエラーの発生個所を特定するための情報を生成するオプションは、GONUMBER オプションのみです。 ゴナンバー オプションを使用すると、ランタイム・エラー・メッセージの ステートメント という用語は、行番号を指します。

ストレージ

STORAGE オプションは、各プロシージャーおよび各開始ブロックによって使用されるストレージの要約をリストの一部として生成するようにコンパイラーに指示します。

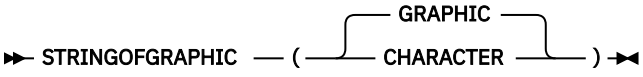


省略形: STG、NSTG

STORAGE 出力には、コンパイルの内部統計に使用されたストレージの量も含まれます。

STRINGOFGRAPHIC

STRINGOFGRAPHIC オプションは、GRAPHIC 集合に適用されたときの STRING 組み込み関数の結果が、CHARACTER または GRAPHIC のいずれの属性を持つかを決定します。



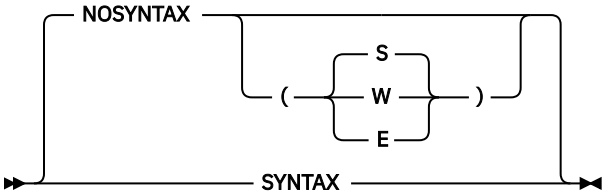
省略形: CHAR、 G

文字
STRINGOFGRAPHIC(CHAR) を指定すると、STRING 組み込み関数が UNALIGNED NONVARYING GRAPHIC 変数の配列または構造体に適用される場合、結果は CHARACTER 属性を持ちます。

GRAPHIC
STRINGOFGRAPHIC(GRAPHIC) を指定すると、STRING 組み込み関数が GRAPHIC 変数の配列または構造体に適用される場合、結果は GRAPHIC 属性を持ちます。

SYNTAX

SYNTAX オプションは、MACRO オプションを指定したときに、回復不能なエラーが発生しない限り、コンパイラが前処理の後に構文チェックを続行することを指定する。 コンパイラがコンパイルを続行するかどうかは、 NOSYNTAX オプションで指定されたエラーの深刻度によって決まる。



省略形: SYN、NSYN

SYNTAX
重大エラーまたは回復不能エラーが起こらない限り、プリプロセス後に 構文検査が継続されます。SYNTAX は NOSYNTAX(S)と同じ。

NOSYNTAX
プリプロセス後、処理は無条件に停止されます。

NOSYNTAX(W)
警告、エラー、重大エラー、または回復不能エラーが検出された場合は、構文検査は行われません。

NOSYNTAX(E)
コンパイラーはエラー、重大エラー、または回復不能エラーを検出すると、構文検査を行いません。

NOSYNTAX(S)
コンパイラーは重大エラーまたは回復不能エラーを検出すると、構文検査を行いません。

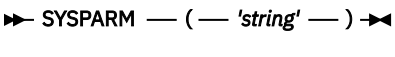
NOSYNTAX オプションでコンパイルを終了すると、相互参照リスト、属性リスト、その他ソース・プログラムに続くリストは作成されない。

このオプションを使用すると、プリプロセッサーを使用する PL/I プログラムを デバッグするときに無駄な操作を省略することができます。

NOSYNTAX オプションが有効な場合、プリプロセッサの指定はすべて CICS プリプロセッサの指定は CICS XOPT または XOPTS オプションによるプリプロセッサの指定は無視される。 これにより、コンパイラーが CICS 変換プログラムを呼び出す前に MACRO プリプロセッサーを呼び出すことができます。

SYSPARM

SYSPARM オプションは、マクロ機能組み込み関数 SYSPARM が返す文字列の値を指定する。

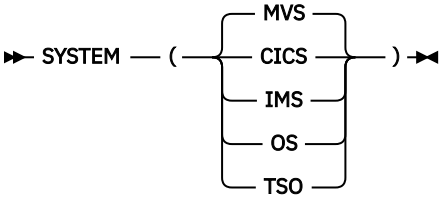


string
最大 1023 文字の長さにすることができます。 デフォルトはヌル・ストリングです。

マクロ機能の詳細については、「*PL/I Language Reference*」を参照してください。

SYSTEM

SYSTEM オプションは、 MAIN PL/I プロシージャにパラメータを渡すために使われる書式を指定するもので、一般に、プログラムが実行されるホスト・システムを示す。



90 ページの表 4 は、使用できるパラメーター・リストのタイプと、 指定したホスト・システムの下でどのようにプログラムが 実行されるかを示しています。 また、 NOEXECOPS の暗黙の設定も表示される。 MAIN プロシージャは、この表で有効であると示されているタイプのパラメータ・リストのみを受け取らなければならない。 SYSTEM オプションに関する追加の実行時情報は *z/OS Language Environment Programming Guide*.

表 *4. SYSTEM* オプション・テーブル

SYSTEM オプション	**パラメーター・リストのタイプ**	**プログラムの 稼働タイプ**	**暗黙の NOEXECOPS**
SYSTEM(MVS)	シングル CHARACTER 文字列またはパラメータなし	z/OS アプリケーション・プログラム	いいえ
	または、任意のパラメーター・リスト		はい
SYSTEM(CICS)	ポインター	CICS トランザクション	はい
SYSTEM(IMS)	ポインター	IMS アプリケーション・プログラム	はい
SYSTEM(OS)	z/OS UNIX パラメーター・リスト (parameter list)	z/OS UNIX アプリケーション・プログラム	はい
SYSTEM(TSO)	へのポインタ CPPL[1]	TSO コマンド・プロセッサー (command processor)	はい

注:
1. TSO 下で MAIN プロシージャーを呼び出す方法について詳しくは、174 ページの『TSO/E で MAIN 』を参照してください。

SYSTEM(IMS) では、すべてのポインターが値 (BYVALUE) で渡されると想定されますが、SYSTEM(MVS) では、すべてのポインターがアドレス (BYADDR) で渡されると想定されます。

CICS 環境で実行される MAIN プロシージャーは、SYSTEM(CICS) または SYSTEM(MVS) を指定してコンパイルする必要があります。

SYSTEM(MVS) を指定してコンパイルされ、ランタイム・オプションが渡されずに実行される、Db2 ストアード・プロシージャーなどのコードの MAIN プロシージャーの OPTIONS オプションには、NOEXECOPS を指定することを強くお勧めします。

コンパイラーは、複数のパラメーターを持つか、CHARACTER VARYING 以外のパラメーターを 1 つだけ持つ、SYSTEM(MVS) でコンパイルされたすべての MAIN プログラムにフラグを立てます。 ライブラリーは単にパラメーター・リストを MAIN に渡すだけでオプションのスキャンやその他のパラメーター・リストのメッセージングを行わないため、このような MAIN プログラムは SYSTEM(OS) でコンパイルした方が恐らくよいと思われます。

端末

TERMINAL オプションは、コンパイル時に作成された診断メッセージおよび通知メッセージを端末に表示するかどうかを決めます。

注 : このオプションは、z/OS UNIX 環境でのコンパイルにだけ適用されます。

省略形: TERM、NTERM

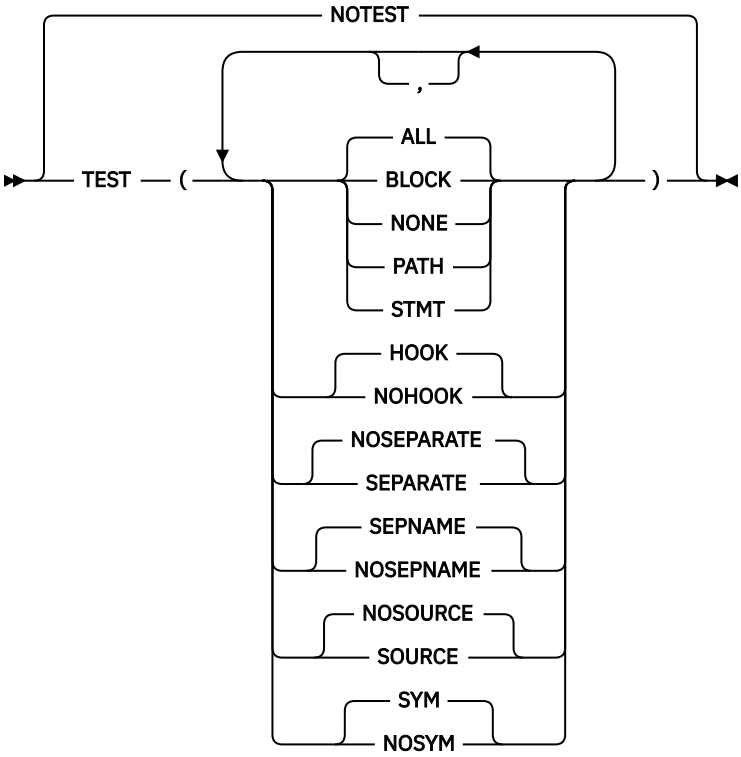
端末
メッセージは端末に表示されます。

NOTERMINAL
通知コンパイラー・メッセージまたは診断コンパイラー・メッセージ は端末に表示されません。

TEST

TEST オプションは、コンパイラがオブジェクト・コードの一部として生成するテスト機能のレベルを指定します。 このオプションを使用すれば、テスト・フックの位置を制御したり、記号テーブルを生成するかどうかを制御したりできます。

NOTEST オプションは、すべての検査情報の生成を抑制する。



ALL
コンパイラーはステートメント・テーブルを生成します。HOOK サブオプションが有効になっている場合は、可能なすべての場所にフックを挿入してステートメント・テーブルを生成します。

注 : opt(2) と opt(3) を指定すると、フックはブロック境界にのみ設定されます。

BLOCK
コンパイラーはステートメント・テーブルを生成します。HOOK サブオプションが有効になっている場合は、ブロック境界 (ブロック入り口とブロック出口) にフックを挿入します。

NONE
プログラムにフックを入れません。

PATH
コンパイラにステートメント・テーブルを生成させ、 HOOK サブオプションが有効な場合は、これらの場所にフックを挿入するように指示する：

- 反復 DO ステートメントに囲まれた最初のステートメントの前
- IF ステートメントの真の部分に含まれる最初のステートメントの前
- IF ステートメントの偽の部分に含まれる最初のステートメントの前
- SELECT グループの真の WHEN または OTHERWISE ステートメント に含まれる最初のステートメントの前
- ユーザー・ラベルに続くステートメントの前 (ラベルの付いた FORMAT ステートメントを除く)

  ステートメントに複数のラベルがある場合は、フックが 1 つのみ挿入されます。
- CALL または関数参照 (ルーチンに制御が渡される前と後の両方)
- ブロック境界

STMT
コンパイラにステートメント・テーブルを生成させ、 HOOK サブオプションが有効な場合は、ステートメント境界とブロック境界にフックを挿入する。

HOOK
TEST サブオプション ALL、STMT、BLOCK、または PATH のいずれかが有効である場合に、コンパイラーによって、生成されたコードにフックが挿入されます。

NOHOOK
コンパイラーによって、生成されたコードにフックは挿入されません。

IBM Debug Tool がオーバーレイ・フックを生成する場合、サブオプション ALL、PATH、STMT、または BLOCK の 1 つを指定しなければなりませんが、HOOK は指定する必要はなく、事実上 NOHOOK を指定することをお勧めします。

NOHOOK を指定すると、ENTRY および EXIT のブレークポイントは、デバッグ・ツール が停止する PATH ブレークポイントにすぎなくなります。

SEPARATE
コンパイラーは、生成するデバッグ情報のほとんどを別個のデバッグ・ファイルに書き込みます。 このオプションを使用する場合、TEST オプションを有効にすると、コンパイラーが作成するオブジェクト・デックのサイズがかなり小さくなります。

プログラムに GET または PUT DATA ステートメントが含まれる場合、別個のデバッグ・ファイルに含まれるデバッグ情報の量が少なくなります。これらのステートメントについては、シンボル・テーブル情報をオブジェクト・デックに書き込む必要があるためです。

生成されるデバッグ情報には、コンパイラーに渡されたソースの圧縮版が常に含まれています。 つまり、ソースは SYSIN DD * を使用して指定されていたり、前のジョブ・ステップで作成された一時データ・セットであったりする可能性があります (例えば、ソースは古い SQL/CICS プリコンパイラーの出力である可能性があります)。 LISTVIEW オプションに指定するサブオプションは、 ソースの内容を制御します。

SEPARATE がバッチ・コンパイルで使用される場合、そのコンパイルの JCL には SYSDEBUG の DD カードが含まれていなければなりません。 その DD カードでは、RECFM=FB および 80 <= LRECL <= 1024 のデータ・セットが指定されなければなりません。

このサブオプションは LINEDIR コンパイラー・オプションと併用できません。

NOSEPARATE
コンパイラーは、生成するデバッグ情報のすべてをオブジェクト・デックに書き込みます。

このオプションを指定すると、生成されるデバッグ情報には、コンパイラーに渡されたソースの圧縮版は含まれません。 これは、プログラムをデバッグしようとするときに デバッグ・ツール で見つけることができるデータ・セットにソースがなければならないことを意味します。

SEPNAME
コンパイラーは、別個のデバッグ・ファイルの名前をオブジェクト・デックに書き込みます。

SEPARATE オプションが有効になっていない場合、このオプションは無視されます。

NOSEPNAME
コンパイラーは、別個のデバッグ・ファイルの名前をオブジェクト・デックに書き込みません。

SEPARATE オプションが有効になっていない場合、このオプションは無視されます。

SOURCE
GOFF オプションも指定されている場合、TEST(SEPARATE)オプションで使用されたのと同じ圧縮形式のソース・ファイルをすべて、生成されるオブジェクト・ファイルに無負荷クラスとして含めるようにします。

NOSOURCE
GOFF オプションも指定されている場合、TEST(SEPARATE)オプションで使用されたのと同じ圧縮形式のすべてのソース・ファイルを、生成されるオブジェクト・ファイルのノー・ロード・クラスに含めないようにします。

SYM
変数を名前で検査するためのシンボル・テーブルを作成します。

NOSYM
シンボル・テーブルは生成されません。

ノート：

- LP(64) では、フックは生成されず、 SEP または NOSEP コンパイラオプションは無視されます。
- opt(2) または opt(3) を指定すると、フックはブロック境界にのみ設定されます。 つまり、最適化されたコードのデバッグは、事実上、 PROCEDUREs と BEGIN ブロックへの出入りをトレースすることに限定される。
- SEPARATE コンパイラー・オプションでコンパイルしたコードをデバッグするには、 デバッグ・ツール バージョン 6（またはそれ以降）を使用する必要があります。
- 範囲が連結データ・セットに及ぶ入力ファイルは、サポートされていません。

TEST(NONE,NOSYM) を指定すると、コンパイラーはオプションを NOTEST に設定する。

TEST(NONE,NOSYM) の使用は強く推奨されておらず、これらの設定を指定したときに何を意図しているのかは不明である。 TEST(ALL,SYM,NOHOOK) 、 TEST(STMT,SYM,NOHOOK)。

NOTEST と TEST(NONE,NOSYM) 以外の TEST オプションは、プログラム・テスト用のアテンション割り込み機能を自動的に提供する。

プログラムに ATTENTION ON-unit があり、それを呼び出したい場合は、以下のいずれかのオプションをつけてコンパイルする必要がある：

- INTERRUPT オプション
- NOTEST または TEST 以外のオプション TEST(NONE,NOSYM)

注 : ATTENTION は でのみサポートされている。 TSO

TEST オプションは GONUMBER を意味する。

TEST オプションはオブジェクト・コードのサイズを大きくし、パフォーマンスに影響する可能性があるため、フックの数や配置を制限したい場合がある。

TEST オプションを指定すると、インライン化は行われない。

REFER の構造体はシンボル・テーブルでサポートされています。

TEST(SYM) が有効な場合、コンパイラーは デバッグ・ツール のオートモニター機能を有効にするためのテーブルを生成する。 これらのテーブルは、 TEST(SEPARATE) オプションが有効でない限り、オブジェクト・モジュールのサイズを大幅に増大させる可能性がある。 デバッグ・ツール のオートモニター機能が有効になっている場合、これらのテーブルは、ステートメントが実行される前に、ステートメントで使用される変数の値を表示するために使用されます - 変数が計算タイプを持っているか、属性 POINTER, OFFSET または HANDLE を持っている限り。 ステートメントが割り当てステートメントである場合は、ターゲット

の値も表示されます。 ただし、ターゲットの初期設定や割り当てがあらかじめ行われていない場合は、その値に意味がありません。

名前に * を使用して宣言された変数はすべて、デバッグ・ツール の使用時には表示されません。 また、* が親構造体または副構造体の名前として使用されている場合は、そのいずれの子も表示されません。 したがって、「名前を付けない」でおく構造体エレメントの名前には、単一の下線を使用することをお勧めします。

UNROLL

UNROLL オプションは、最適化におけるループのアンロールを制御する。 Loop アンローリングは、ループ本体を複数回複製し、それに応じてループ制御コードを調整する最適化である。

AUTO
コンパイラーは、アンロールの適切な対象になると判断したループをアンロールすることを許可されています。

UNROLL オプションを指定すると、生成されるオブジェクト・コードのサイズが大きくなることがある。

NO
コンパイラーはループのアンロールを許可されていません。

NOOPTIMIZE オプションが有効な場合、 UNROLL オプションは無視される。

Loop unrolling は、命令スケジューリングとソフトウェア・パイプラインのために命令レベルの並列性を公開することにより、プログラムの性能を向上させる。 また、これにより新しいループ本体の中のコードが大きくなり、 それがレジスター割り振りへのプレッシャー増大につながるために、 パフォーマンスが低下することがあります。

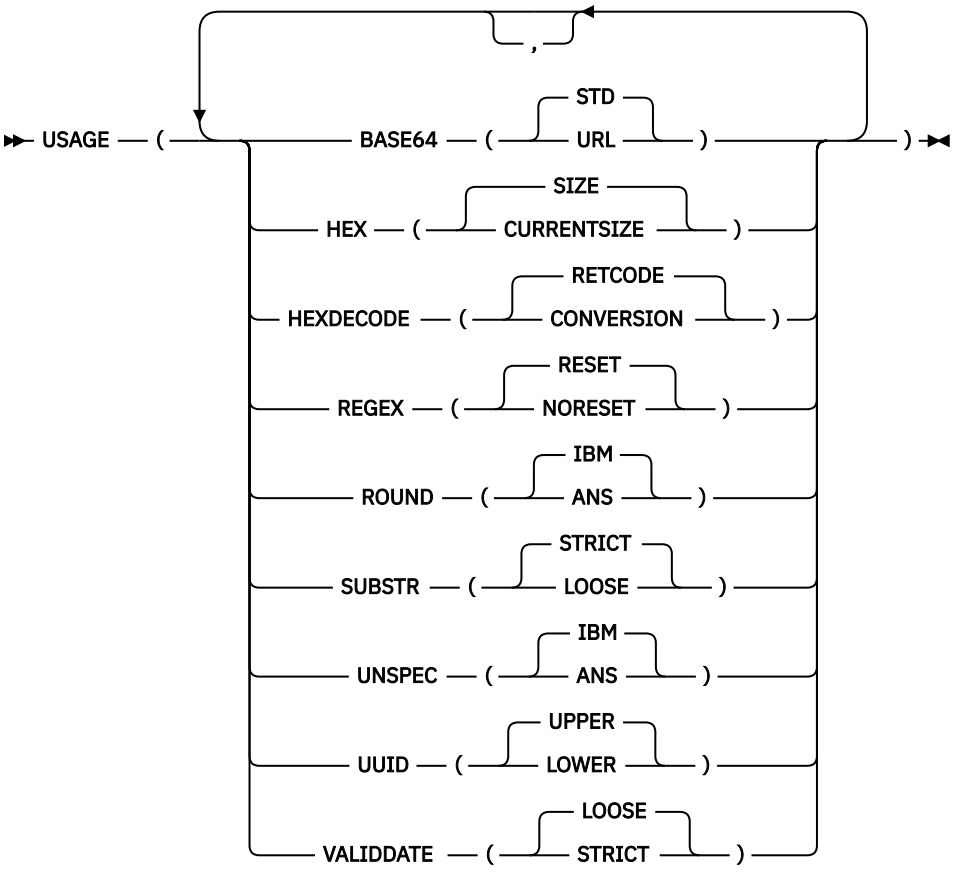
したがって、ループをアンロールする前に、以下の手順を踏んで、 UNROLL オプションが特定のアプリケーションのパフォーマンスを向上させるかどうかを確認する：

1. 通常のオプションを指定してプログラムをコンパイルします。
2. 標準的なワークロードでプログラムを実行します。
3. UNROLL オプションを指定してプログラムを再コンパイルする。
4. 同じ条件でプログラムを再実行します。

UNROLL(AUTO）がデフォルトである。

USAGE

USAGE オプションを使用すると、選択した組み込み関数に対して異なるセマンティクスを選択することができます。



BASE64(STD | URL)

BASE64(STD) サブオプションでは、 base64 エンコーディングの最後の 2 桁は + と / である。

BASE64(URL) サブオプションでは、 base64 エンコードの最後の 2 桁は-と_である。

HEX(SIZE | CURRENTSIZE)

HEX(SIZE) サブオプションで、HEX が VARYING または VARYINGZ ストリングに適用された場合、ストリングで使用されているストレージの最大容量を示す 16 進ストリングが返されます。

HEX(CURRENTSIZE) サブオプションで、HEX が VARYING または VARYINGZ ストリングに適用された場合、ストリングで使用されているストレージの現在容量を示す 16 進ストリングが返されます。

USAGE(HEX) オプションでは、CSTG および STG はそれぞれ CURRENTSIZE および SIZE の同義語として受け入れられます。

HEXDECODE(RETCODE | CONVERSION)

HEXDECODE(CONV)サブオプションの下では、HEXDECODE と HEXDECODE8 組み込み関数は、無効なソース・データが見つかった場合に CONVERSION 条件を発生させます。

HEXDECODE(RETCODE) サブオプションの下では、HEXDECODE および HEXDECODE8 組み込み関数は、ソースが偶数桁であるかどうかをチェックする以外は、無効なソース・データをチェックしません (無効な場合、関数は値 -2) を返します)。

REGEX(RESET | NORESET)

REGEX 組み込み関数の各呼び出しで REGEX (RESET) サブオプションが指定された場合、REGEX 関数のコード・ページ引数が現行ロケールに対応するコード・ページと異なっていれば、そのロケール値が保存され、REGEX 関数を終了する前に復元されます。 現行ロケールに対応しないコード・ページを指定した REGEX 関数が繰り返し呼び出されると、パフォーマンスが大幅に低下することがあります。

REGEX (NORESET) サブオプションが指定された場合は、コード・ページ引数と一致させるためにロケールを変更する必要があるとして、ロケール値の保存と復元は行われません。 これはパフォーマンス

の点ではるかに有利ですが、元のそのロケール設定に依存する他のコードがある場合、そのコードは予期したとおりに機能しない可能性があるということにもなります。

ROUND(IBM | ANS)
ROUND(IBM) サブオプションを指定すると、ROUND 組み込み関数の最初の引数が FLOAT 属性を持っている場合、2 番目の引数は無視されます。

ROUND(ANS) サブオプションを指定すると、ROUND 組み込み関数は、「*PL/I Language Reference*」で説明されているようにインプリメントされます。

SUBSTR(STRICT | LOOSE)
SUBSTR(STRICT) サブオプションでは、x が CHARACTER タイプである場合は、長さ MIN(z, MAXLENGTH(x)) または長さ z を SUBSTR(x,y,z) 組み込み関数参照に割り当てる許可をコンパイラーに付与します。

SUBSTR(LOOSE) サブオプションでは、同じ参照で、長さが z であるストリングが返されます。

SUBSTR(LOOSE) サブオプションは、SUBSTR(x,y,z) 参照 (x は CHAR(1) BASED 変数) が指定されている場合に役立つ可能性があります。

STRINGRANGE が有効になっている場合に、z > MAXLENGTH(x) であれば、このオプションのいずれかの設定で STRINGRANGE が適用されます。

UNSPEC(IBM | ANS)
UNSPEC(IBM) サブオプションを指定すると、UNSPEC を構造体に適用することはできません。配列に適用すると、ビット・ストリングの配列が戻されます。

UNSPEC(ANS) サブオプションを指定すると、UNSPEC を構造体に適用できます。構造体または配列に適用すると、UNSPEC はシングル・ビット・ストリングを戻します。

UUID(UPPER | LOWER)

USAGE (UUID (LOWER)) コンパイラー・オプションが有効な場合、UUID および UUID4 によって返されるストリング内のすべての英字は小文字になります。 このオプションを使用すると、国際標準に準拠する UUID 値が生成されます。

USAGE (UUID (UPPER)) コンパイラー・オプションが有効な場合、UUID および UUID4 によって返されるストリング内のすべての英字は大文字になります。

VALIDDATE(LOOSE | STRICT)

USAGE (VALIDDATE (LOOSE)) オプションが有効になっている場合、コンパイラーは、1582 年のどの日付も有効ではない (1582 年 10 月 15 日より前の日付のみではない) と想定します。そのため、コンパイラーは、YYYY を使用する多くのパターンに対して、VALIDDATE 組み込み関数のより優しく高速なインライン・コードを生成できます。

USAGE (VALIDDATE (STRICT)) オプションが有効な場合、コンパイラーは、1582 の年の値を持つリリアン・カレンダーからの日付を VALIDDATE 関数で有効として受け入れます。

Default: USAGE(HEX(SIZE) REGEX(RESET) ROUND(IBM) SUBSTR(STRICT) UNSPEC(IBM) UUID(UPPER)) VALIDDATE(LOOSE)

WIDECHAR

WIDECHAR オプションは、WIDECHAR データが保管されるフォーマットを指定します。

BIGENDIAN
WIDECHAR データをビッグ・エンディアン・フォーマットで保管するように指示します。 例えば、UTF-16 文字 1 の WIDECHAR 値は '0031'x として保管されます。

LITTLEENDIAN
WIDECHAR データをリトル・エンディアン・フォーマットで保管するように指示します。 例えば、UTF-16 文字 1 の WIDECHAR 値は '3100'x として保管されます。

WX 定数は、常にビッグ・エンディアン・フォーマットで指定する必要があります。 したがって、WIDECHAR(LITTLEENDIAN) オプションを指定した 場合、値 '1' は '3100'x として保管されますが、この値は 常に '0031'wx として指定する必要があります。

WINDOW

WINDOW オプションは、各種の日付関連の組み込み関数で使用 される w ウィンドウ引数を設定します。

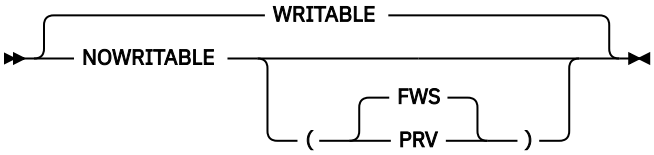
w
固定ウィンドウの開始を表す符号なし整数、または "スライディング" ウィンドウを指定する負の整数のいずれか。 例えば、`WINDOW(-20)` は、プログラムを実行する 20 年前に開始されるウィンドウを示します。

WRITABLE

WRITABLE オプションは、コンパイラーが静的ストレージを書き込み可能なストレージとして扱うことができることを指定します (また、そうする場合は、その結果のコードが再入不可になります)。

このオプションは、RENT オプションを指定してコンパイルされたプログラムには効果がありません。

注 : LP(64) オプションでは、WRITABLE オプションは無視されます。



NORENT WRITABLE オプションを使用すると、コンパイラーは、 以下の定数または変数を実装するために静的ストレージ上で書き込みできるようになります。

- CONTROLLED 変数
- FETCHABLE ENTRY 定数
- FILE 定数

NORENT WRITABLE オプションを指定すると、CONTROLLED 変数を 使用するモジュール、入出力を実行するモジュール、または FETCH を使用するモジュールは、再入可能にはなりません。

NORENT NOWRITABLE オプションでは、コンパイラーが、以下の定数または変数の静的ストレージ上に書き込まないようにする必要があります。

- CONTROLLED 変数
- FETCHABLE ENTRY 定数
- FILE 定数

NORENT NOWRITABLE オプションを指定すると、 CONTROLLED 変数を使用するモジュール、入出力を実行するモジュール、 または FETCH を使用するモジュールは、再入可能になります。

FWS および PRV サブオプションは、以下のように、コンパイラーが CONTROLLED 変数を扱う方法を決定します。

FWS
EXTERNAL プロシージャーに入るときに、コンパイラーはライブラリー呼び出し を行って、プロシージャー (およびすべてのサブプロシージャー) 内で CONTROLLED 変数をアドレス指定するために使用できるストレージを見つけます。

PRV
コンパイラーは、CONTROLLED 変数をアドレス指定するために、旧 OS PL/I コンパイラーで使用していたものと同じ、疑似レジスター変数メカニズムを使用します。

このため、NORENT NOWRITABLE(PRV) オプションを指定すると、旧コードと新規コードで CONTROLLED 変数を共用することができます。

ただし、NORENT NOWRITABLE(PRV) オプションを指定すると、CONTROLLED 変数の使用が、 旧コンパイラーと同じすべての制約を受けることも意味しています。

NORENT NOWRITABLE(FWS) オプションを指定すると、 以下のアプリケーションは、RENT または WRITABLE オプションを指定してコンパイルされた場合と 同じようには実行されないことがあります。

- CONTROLLED 変数を使用するアプリケーション
- FILE CONSTANT を FILE VARIABLE に割り当てるアプリケーション

NORENT NOWRITABLE(FWS) を指定した場合のアプリケーションのパフォーマンスは、多くの CONTROLLED 変数を多くの PROCEDURE で使用していると特に悪くなる場合があります。

NOWRITABLE オプションを指定すると、PROCEDURE の外側で PACKAGE において以下の変数や定数を宣言できません。

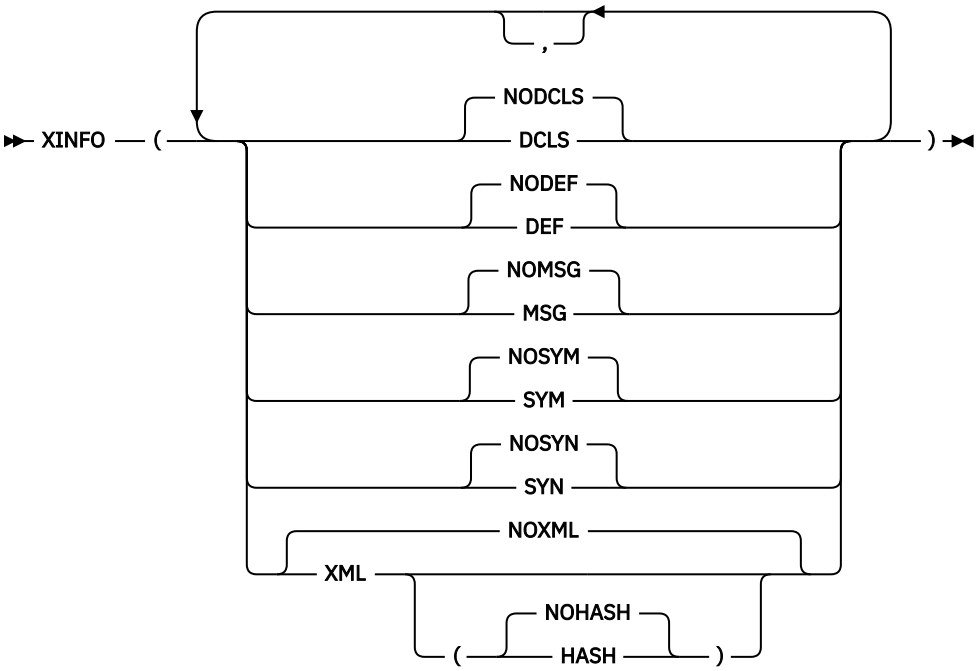
- CONTROLLED 変数
- FETCHABLE ENTRY 定数
- FILE 定数

NORENT WRITABLE を指定してコンパイルされたコードは、 任意の外部 CONTROLLED 変数を共用する、NORENT NOWRITABLE を指定して コンパイルされたコードと混在できません。 一般に、WRITABLE でコンパイルされたコードと NOWRITABLE でコンパイルされたコードを 混在しないでください。

関連情報
68 ページの『RENT』
コードが静的変数を変更しなければ、そのコードは「本質的に再入可能」です。 RENT オプションは、本来再入可能でないコードを再入可能にすることを指定する。

XINFO

XINFO オプションは、現行コンパイル単位に関する追加の情報が入った ファイルを追加して生成するようにコンパイラーに指定します。



DCLS
コンパイル単位でエクスポートされた各プロシージャー名の PL/I declare 文を含むテキストファイルを作成する。 そして、このファイルを使って、これらの手続きを呼び出す他のコードにインクルードされるべきインクルード・ファイルを作成することができる。 これは、RULES(NOLAXINTERFACE)を強制したり、呼び出し元と呼び出し先のミスマッチを避けるのに非常に役立つ。 バッチでは、このファイルは SYSDCLSD DD 文で指定されたファイルに書き込まれる。 z/OS UNIX System Services では、このファイルはオブジェクトデッキと同じディレクトリに書き込まれ、拡張子は dcl となる。

NODCLS
コンパイラーはそのようなテキスト・ファイルは作成しない。

DEF
定義 SIDEDECK ファイルが作成されます。 このファイルには、コンパイル単位に関する以下の情報がリストされます。

- 定義されているすべての EXTERNAL プロシージャー
- 定義されているすべての EXTERNAL 変数
- 静的に参照されるすべての EXTERNAL ルーチンおよび変数
- 動的に呼び出されるフェッチ済みモジュール

バッチ環境では、このファイルは SYSDEFSD DD ステートメントによって指定された ファイルに書き込まれます。 z/OS UNIX システム・サービスでは、このファイルはオブジェクト・デックと同じディレクトリーに作成されて拡張子 def が付けられます。

例えば、次のプログラムがあるとします。

```
defs: proc;
   dcl (b,c) ext entry;
   dcl x ext fixed bin(31) init(1729);
   dcl y ext fixed bin(31) reserved;
   call b(y);
   fetch c;
   call c;
end;
```

この場合、次の def ファイルが生成されます。

```
EXPORTS CODE
  DEFS
EXPORTS DATA
  X
```

```
  IMPORTS
    B
    Y
  FETCH
    C
```

def ファイルを使用して、アプリケーションの 依存性グラフを作成したり、相互参照分析を行ったりすることができます。

注 : 上記の情報を生成するには、XINFO (DEF) オプションに対して EXPORTALL オプションが有効になっている必要があります。

NODEF
定義 SIDEDECK ファイルは作成されません。

MSG
メッセージ情報が ADATA ファイルに生成されます。

バッチでは、ADATA ファイルは、SYSADATA DD ステートメントによって指定されたファイルに生成されます。 z/OS UNIX では、ADATA はオブジェクト・ファイルと同じディレクトリーに生成され、拡張子は adt になります。

NOMSG
メッセージ情報は ADATA ファイルに生成されません。 MSG も SYM も指定しないと、ADATA ファイルは生成されません。

SYM
シンボル情報が ADATA ファイルに生成されます。

バッチでは、ADATA ファイルは、SYSADATA DD ステートメントによって指定されたファイルに生成されます。 z/OS UNIX では、ADATA ファイルはオブジェクト・ファイルと同じディレクトリーに生成され、拡張子は adt になります。

NOSYM
シンボル情報は ADATA ファイルに生成されません。

SYN
構文情報が ADATA ファイルに生成されます。 XINFO(SYN) オプションを指定すると、コンパイラーが必要とするストレージ量 (メモリーと生成されるファイルの両方) が大幅に増えることがあります。

バッチでは、ADATA ファイルは、SYSADATA DD ステートメントによって指定されたファイルに生成されます。 z/OS UNIX では、ADATA ファイルはオブジェクト・ファイルと同じディレクトリーに生成され、拡張子は adt になります。

NOSYN
構文情報は ADATA ファイルに生成されません。

xml(ハッシュ | ノーハッシュ)
XML サイド・ファイルが作成されます。 この XML ファイルには以下の項目が含まれます。

- コンパイル用のファイル参照テーブル
- コンパイルされたプログラムのブロック構造
- コンパイル時に作成されたメッセージ

バッチ環境では、このファイルは SYSXMLSD DD ステートメントによって指定される ファイルに書き込まれます。 z/OS UNIX システム・サービスでは、このファイルはオブジェクト・デックと同じディレクトリーに作成されて拡張子 xml が付けられます。

作成された XML の DTD ファイルは次のようになります。

```
<?xml encoding="UTF-8"?>

<!ELEMENT PACKAGE ((PROCEDURE)*,(MESSAGE)*,FILEREFERNCETABLE)>
<!ELEMENT PROCEDURE (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT BEGINBLOCK (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT MESSAGE (MSGNUMBER,MSGLINE?,MSGFILE?,MSGTEXT)>
<!ELEMENT FILE (FILENUMBER,INCLUDEDFROMFILE?,INCLUDEDONLINE?,FILENAME)>
<!ELEMENT FILEREFERENCETABLE (FILECOUNT,FILE+)>
```

```
<!ELEMENT BLOCKFILE (#PCDATA)>
<!ELEMENT BLOCKLINE (#PCDATA)>
<!ELEMENT MSGNUMBER (#PCDATA)>
<!ELEMENT MSGLINE (#PCDATA)>
<!ELEMENT MSGFILE (#PCDATA)>
<!ELEMENT MSGTEXT (#PCDATA)>
<!ELEMENT FILECOUNT (#PCDATA)>
<!ELEMENT FILENUMBER (#PCDATA)>
<!ELEMENT FILENAME (#PCDATA)>
<!ELEMENT INCLUDEFROMFILE (#PCDATA)>
<!ELEMENT INCLUDEDONLINE (#PCDATA)>
```

XINFO(XML)オプションの HASH サブオプションは、FILENAME 行ごとに、生成された XML ファイルの後に、そのファイルの 256 ビットの SHA-2 ハッシュの 16 進数値を <CONTENTHASH> と </CONTENTHASH> タグで囲んだ行が続くように指定します。

コンパイラーは、ファイルをオープンする際に SHA2INIT256 組み込み関数を呼び出すことで、このハッシュを実行する。 そして、その関数が返すトークンを使って、ファイルの各行を読み込んだ後に SHA2UPDATE256 組み込み関数を起動し、ファイルを閉じた後に SHA2FINAL256 組み込み関数を起動する。

ファイルをハッシュ化する際、コンパイラーは右マージンより短い行を空白で埋め、幅を広げる。

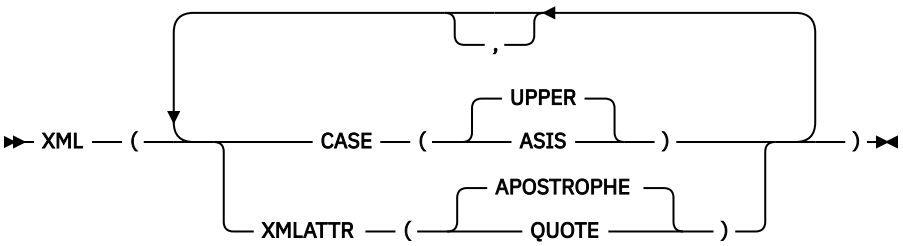
このハッシュ値は、FILEREF(HASH)オプションが指定された場合にリストに表示される値と同じである。

NOXML
XML サイド・ファイルは作成されません。

関連情報
461 ページの『SYSADATA メッセージ情報』
XINFO コンパイル時オプションの MSG サブオプションを指定すると、コンパイラーは SYSADATA ファイルを生成する。

XML

XML オプションは、XMLCHAR 組み込み関数によって生成される XML 中の名前の大文字と小文字を指定します。



CASE(UPPER | ASIS)
CASE(UPPER) サブオプションでは、 XMLCHAR 組み込み関数が生成する XML の名前はすべて大文字になる。

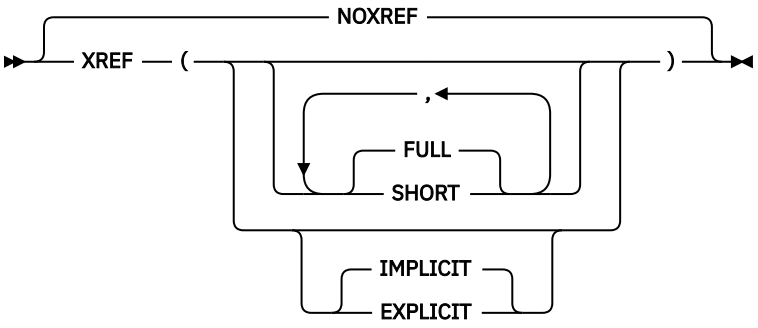
CASE(UPPER) サブオプションでは、 XMLCHAR 組み込み関数が生成する XML の名前は、宣言で使われている大文字と小文字になります。 マクロ・プリプロセッサ・オプション CASE(ASIS) を使用せずに MACRO プリプロセッサを使用した場合、コンパイラが見るソースはすべて大文字の名前になり、XML(CASE (ASIS))オプションの指定が無意味になることに注意してください。

XMLATTR(APOSTROPHE | QUOTE)
XMLATTR(APOSTROPHE) サブオプションでは、XMLCHAR 組み込み関数によって生成される XML 属性はすべてアポストロフィで囲まれます。

XMLATTR(QUOTE) サブオプションでは、XMLCHAR 組み込み関数によって生成された XML 属性はすべて引用符で囲まれます。 CODEPAGE(1026) および CODEPAGE(1155) では、引用符の 16 進値は「FC'x」であり、その他のすべての EBCDIC コードページでは「 '7F'x 」である。

XREF

XREF オプションは、プログラムで使用されている名前と、コンパイラ・リストで宣言または参照されているステートメントの番号の相互参照表を提供します。



省略形: X、NX

FULL
XREF(FULL) では、すべての ID と属性がコンパイラー・リストに組み込まれます。 FULL がデフォルトです。

SHORT
XREF(SHORT) では、非参照 ID はコンパイラー・リストから除外されます。

EXPLICIT
XREF(EXPLICIT) では、構造が参照されると、その構造のみがコンパイラー・リストに組み込まれます。

IMPLICIT
XREF(IMPLICIT) の場合、構造が参照されると、その構造と、そのすべてのメンバーがコンパイラー・リストに組み込まれます。 IMPLICIT がデフォルトです。

XREF オプションを使用したときに作成されるクロス・リファレンス・リストに含まれない名前は、END ステートメントのラベル参照のみです。 例えば、プロシージャー PROC1 のステートメント番号 20 が `END PROC1;` であると想定します。 この場合、ステートメント番号 20 は PROC1 用の相互参照リストには入りません。

XREF と ATTRIBUTES オプションの両方を指定すると、2 つのリストが組み合わされる。 SHORT と FULL が対立する場合は、最後に指定したオプションで使用法が 決まります。 例えば、 ATTRIBUTES(SHORT) XREF(FULL)とすると、組み合わされたリストに FULL オプションが表示される。

関連情報
108 ページの『相互参照テーブル』
ATTRIBUTES と XREF を指定すると、相互参照テーブルと属性テーブルが結合されます。 名前に対する属性のリストは、ファイル番号と行番号で識別されます。

オプションの中のブランク、コメント、およびストリング

オプションを指定するときにブランクを使用できる場所では常に、必要な数のブランクやコメントも指定できます。 ただし、いくつかの規則に注意する必要があります。

%PROCESS 、またはオプション・ファイル内の行でコメントが指定された場合、そのコメントは開始行と同じ行で終わらなければならない。

同様に、コメントがコマンドラインや PARM= specification で始まる場合、コメントもそこで終わらなければならない。

同じルールが文字列にも適用される。 %PROCESS 行またはオプション・ファイル内の行で文字列を指定する場合、その文字列は、開始行と同じ行で終わらなければならない。 同様に、文字列がコマンドラインや PARM= specification で始まる場合は、そこで終わる必要がある。

デフォルト・オプションの変更

提供されているデフォルトのコンパイラー・オプションを変更したい場合は、コンパイラーのインストール中に、サンプルのジョブ IBMZWIOP を編集して送信してください。

このジョブで、あらかじめ適用しておくオプションを指定し、これら以外のオプションは後で追加するという形にすることで、デフォルト・オプションを実質的に変更することができます。 またこのジョブでは、後から追加して最終的に適用するオプションを指定することで、デフォルト・オプションを実質的に変更することができます。この方法を使用すると、このジョブで指定したオプションが指定変更されることがありません。

マクロのプリプロセッサ・オプションのデフォルトを変更したい場合は、このジョブの一部として適切な PPMACRO オプションを指定することで、インストール時に変更することもできます。 PPCICS と PPSQL オプションは、それぞれ CICS と SQL プリプロセッサにそれぞれ対応する変更を加えることができます。

詳しくは、サンプル・ジョブに記述されている説明を参照してください。

%PROCESS または *PROCESS ステートメントでオプションを指定する

%PROCESS または *PROCESS 文は、各外部プロシージャの開始を特定し、各コンパイルにコンパイラー・オプションを指定できるようにする。 あなたのプログラムでは、 %PROCESS または *PROCESS のどちらを使ってもよい。

注 : 一貫性と読みやすさのため、 %PROCESS または *PROCESS のステートメントは常に %PROCESS と呼ばれる。

%PROCESS ステートメントで指定したオプションは、ソース・ステートメントの入力終了または次の %PROCESS ステートメントまでのコンパイルに適用される。

%PROCESS ステートメントでオプションを指定するには、以下のようにコーディングする：

```
%PROCESS options;
```

ここで *options* はコンパイラー・オプションのリストです。

オプション・リストの終わりにはセミコロンを入れなければならず、 また、オプション・リストはデフォルトの右側ソース・マージンを超えてはなりません。 パーセント記号 (%) またはアスタリスク (*) が レコードの最初の列になければなりません。 キーワード PROCESS は、次のバイト（列）または任意の数の空白の後に続くことができる。 オプション・キーワードは、 コンマまたは少なくとも 1 つのブランクを使って区切らなければなりません。

文字数を制限するものはレコードの長さだけです。 オプションをどれも指定したくない場合は、次のようにコーディングします。

```
%PROCESS;
```

%PROCESS ステートメントを次のレコードに続ける必要がある場合は、区切り文字の後でリストの最初の部分を終了し、次のレコードに続ける。 キーワードやキーワード引数を複数のレコードに分割することはできません。 %PROCESS ステートメントを数行続けることも、新しい %PROCESS ステートメントを開始することもできる。 次の例は、隣接する複数の %PROCESS ：

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST TEST ;
```

コンパイル時オプション、その省略形構文、および IBM 提供のデフォルト は、3 ページの表 3 に示してあります。

PROCINC ステートメントを使用して、追加の PROCESS ステートメントを含むファイルをインクルードすることができます。

インクルードするファイルを指定するには、 %PROCINC ：

```
%PROCINC dataset-name;
```

ここで、dataset-name は組み込むファイルの名前です。

また、 %PROCESS ステートメントをセミコロンで終わらせ、デフォルトの右側のソース・マージンを超えてはならない。 パーセント記号 (%) またはアスタリスク (*) が レコードの最初の列になければなりません。 キーワード %PROCESS は、次のバイト（列）または任意の数の空白の後に続くことができる。

%PROCESS キーワードの後には、少なくとも 1 つの空白がなければならない。

含まれるデータセットには、 %PROCESS と %PROCINC ステートメント（および任意の数の空白行）のみが含まれていなければならない。

%PROCESS 、 %PROCINC ステートメントがあるかどうかをコンパイラがどのように判断するかは、最初のソース・ファイルの形式に依存する：

F または FB フォーマット
レコードの先頭文字がアスタリスク (*) またはパーセント記号 (%) の場合、コンパイラーは次の非ブランク文字が PROCESS であるか PROCINC であるかを検査します。

V または VB フォーマット
レコードの最初の文字が数値の場合、コンパイラーは最初の 8 文字がシーケンス番号であると想定し、9 番目の文字がアスタリスク (*) またはパーセント記号 (%) の場合は、次の非ブランク文字が PROCESS または PROCINC であるかどうかを検査します。 ただし、先頭文字が数値ではなく、アスタリスク (*) またはパーセント記号 (%) である場合、コンパイラーは、次の非ブランク文字が PROCESS または PROCINC であるかどうかを検査します。

U フォーマット
レコードの先頭文字がアスタリスク (*) またはパーセント記号 (%) の場合、コンパイラーは次の非ブランク文字が PROCESS または PROCINC であるかどうかを検査します。

% ステートメントの使用

コンパイラーの操作を指示するステートメントは、パーセント (%) 記号で 始まります。 % ステートメントを使用して、ソース・プログラムのリストを制御し、ソース・プログラムに外部ストリングを組み込むことができます。% ステートメントは、ラベルまたは条件接頭語を持つことはできません。また、コンパウンド・ステートメントの単位であってはなりません。 % ステートメントはおのおの、1 行内に単独で入っていなければなりません。

各 % 制御ステートメントの使用法を以下に示します。 これらのステートメントについて詳しくは、「*PL/I Language Reference*」を参照してください。

%INCLUDE
外部テキストをソース・プログラムに組み込むようコンパイラーに指示します。

%XINCLUDE
外部テキストがまだ組み込まれていない場合、外部テキストをソース・プログラムに組み込むようコンパイラーに指示します。

%PRINT
SOURCE および INSOURCE のリストの印刷を 再開するよう、コンパイラーに指示します。

%NOPRINT
%PRINT ステートメントに出会うまで、ソースとインソースリストの印刷を中断するようにコンパイラに指示する。

%PAGE
プログラムリストの %PAGE ステートメントの直後にあるステートメントを、次のページの最初の行に表示するようにコンパイラに指示する。

%POP
最新の %PUSH により保存された %PRINT および %NOPRINT の状況を復元するように、コンパイラーに指示します。

%PUSH
%PRINT 、%NOPRINT の現在の状態を、先入れ先出しでプッシュ・ダウン・ スタックに保存する。

%SKIP
スキップしたい行数を指定します。

%INCLUDE

%INCLUDE ステートメントは、コンパイル単位内の指定された点に 追加の PL/I ファイルを組み込むために使用します。

%INCLUDE ステートメントを使ってライブラリのソース・テキストを PL/I プログラムに組み込む方法については *PL/I Language Reference*.

バッチ・コンパイルの場合
ライブラリー とは、メンバーと呼ばれるその他のデータ・セットを保管するのに使用できる z/OS 区分データ・セットです。 %INCLUDE ステートメントを使用して、 PL/I プログラムに挿入するソース・テキストは、ライブラリ内のメンバとして存在する必要があります。 ソース・ステートメント・ライブラリーをコンパイラーに対して定義するプロセスについて詳しくは、162 ページの『ソース・ステートメント・ライブラリ (SYSLIB)』を参照してください。

ステートメント `%INCLUDE DD1 (INVERT);` は、DD ステートメント名 DD1 で定義されたライブラリのメンバ INVERT のソース・ステートメントを、ソース・プログラムに連続して挿入することを指定する。 コンパイル・ジョブ・ステップには、適切な DD 。

ddname を省略すると、ddname SYSLIB 。 その場合、SYSLIB という名前で、DD ステートメントを含める必要がある。 (IBM 提供のカタログ式プロシージャーの場合は、 コンパイル・プロシージャー・ステップにこの名前の DD ステートメントは 組み込まれていません。)

z/OS UNIX コンパイルの場合
UPPERINC を指定しない限り、実際のインクルード・ファイル名は小文字でなければならない。 例えば、組み込みステートメント `%include sample` を使用すると、コンパイラーはファイル `sample.inc` を検出できますが、ファイル SAMPLE.inc は検出できません。 組み込みステートメント `%include SAMPLE` を使用しても、コンパイラーは `sample.inc` を検索します。

コンパイラーは INCLUDE ファイルを以下の順番で探す：

1. 現行ディレクトリー
2. I フラグまたは INCDIR コンパイラー・オプションで指定されたディレクトリ
3. `/usr/include` Directory
4. PDS INCPDS コンパイラー・オプション

コンパイラーが検出した最初のファイルがソースに組み込まれます。

%INCLUDE ステートメントに含まれるソース・テキストに %PROCESS ステートメントがあると、コンパイルでエラーが発生する。

ライブラリからのソース・ステートメントのインクルードは、 %INCLUDE ステートメントを使用して、FUN のソース・ステートメントをプロシージャ TEST にインクルードすることを示しています。 図書館 HPU8。NEWLIB は、 という修飾名を持つ ステートメントで定義され、このジョブのカタログ化されたプロシージャのステートメントに追加される。 PLI.SYSLIB DD ソース・ステートメント・ライブラリは、SYSLIB という名前の DD ステートメントによって定義されているため、 %INCLUDE ステートメントに ddname を含める必要はありません。

ソース・プログラムや、入れようとするテキストにマクロ・ステートメントがまったく入っていなければ、プリプロセッサーを呼び出す必要はありません。

ライブラリからのソース文を含む：

```
//OPT4#9     JOB
//STEP3      EXEC IBMZCBG,PARM.PLI='INC,S,A,X,NEST'
//PLI.SYSLIB DD DSN=HPU8.NEWLIB,DISP=OLD
//PLI.SYSIN  DD *
   TEST: PROC OPTIONS(MAIN) REORDER;
     DCL ZIP PIC '99999';          /* ZIP CODE                    */
```

```
      DCL EOF BIT INIT('0'B);
      ON ENDFILE(SYSIN) EOF = '1'B;
      GET EDIT(ZIP) (COL(1), P'99999');
      DO WHILE(¬EOF);
        PUT SKIP EDIT(ZIP, CITYFUN(ZIP)) (P'99999', A(16));
        GET EDIT(ZIP) (COL(1), P'99999');
      END;
      %PAGE;
      %INCLUDE FUN;
   END;                              /* TEST                          */
//GO.SYSIN DD *
95141
95030
94101
//
```

コンパイラー・リストの使用

コンパイルのときにコンパイラーは、大半がオプショナルのリストを生成 しますが、そのリストには、ソース・プログラム、コンパイル、および オブジェクト・モジュールに関する情報が入っています。

次のリストの説明は、印刷ページ上の外観について述べています。

注 : コンパイラー・リストは利用可能ですが、これはプログラミング・インターフェースではないため、変更される可能性があります。

特定の処理段階に達する前にコンパイルが終了する場合、対応するリストは表示されません。

見出し情報

リストの最初のページは、製品番号、コンパイラー・バージョン番号、コンパイラー・ビルド日時を指定するストリング、およびコンパイル開始日時によって識別されます。 このページおよび以降のページには番号が付けられます。

次にリストには、このコンパイルに指定されたすべてのオプションが表示されます。 これらのオプションは、 NOOPTIONS オプションが指定されていても表示され、順に以下のオプションが含まれる：

- 初期インストール・オプション (インストール時のオプション・セットで、あらかじめ適用されるオプションです。これら以外のオプションは、後で追加されます)。
- z/OS UNIX では、 *IBM_OPTIONS* 環境変数で指定されたオプションはすべて、以下のようになる
- コンパイラーに渡されるパラメーター・ストリング (すなわち、z/OS UNIX におけるコマンド行、またはバッチにおける PARM=) で指定されたオプション。
- コンパイラー・パラメーター・ストリングで指定されたオプション・ファイルで指定されたオプション。

  これには、各オプション・ファイルの名前とその内容が、コンパイラーが読み取ったそのままの形式で含まれます。
- ソースの *PROCESS または %PROCESS 行で指定されたオプション
- 最終インストール・オプション (インストールで設定されたオプション、および他の何らかのオプションの後に適用されるオプション)。

リストの終わり付近には、コンパイル時にエラーや警告状態がなにも 検出されなかった旨のステートメントか、または 1 つ以上のエラー が検出された旨のメッセージが入ります。 メッセージのフォーマットについては、112 ページの『メッセージと戻りコード』を参照してください。 リストの最後から 2 番目の行は、コンパイルに要した時間を示します。 リストの最後の行は `END OF COMPILATION OF` *xxxx* です (*xxxx* は外部プロシージャー名です)。 NOSYNTAX コンパイラー・オプションを指定した場合、またはコンパイルの初期段階でコンパイラーが異常終了した場合、外部プロシージャー名 *xxxx* は組み込まれずに切り捨てられて、この行は `END OF COMPILATION` となります。

以下のトピックでは、リストのオプション部分を指定順に説明します。

コンパイルに使用するオプション

OPTIONS オプションを指定すると、デフォルトのオプションを含め、コンパイルに指定されたオプションの完全なリストが次のページに表示される。

コンパイル時に最終的に有効になるすべてのオプションの設定値がリストに示されます。 オプションの設定値が初期インストール・オプションの適用後にデフォルト設定値と異なる場合は、その行に正符号 (+) のマークが付けられます。

プリプロセッサー入力

MACRO と INSOURCE オプションの両方を指定した場合、コンパイラーはプリプロセッサーへの入力を 1 行に 1 レコードずつ、各行に左から連番をつけてリストアップする。

プリプロセッサーがエラーまたはエラーの可能性を検出すると、そのページまたは入力リストに続くページにメッセージが印刷されます。 このメッセージのフォーマットは、112 ページの『メッセージと戻りコード』の説明にあるコンパイラー・メッセージのフォーマットと同じです。

SOURCE プログラム

SOURCE オプションを指定すると、コンパイラーは 1 行に 1 レコードをリストアップする。 これらのレコードには、ソース行番号とソース・ファイル番号が必ず含まれます。 ただし、ファイルに含まれる行の数が 999999 行以上になる場合、コンパイラーは、ファイルが大きすぎることを示すフラグをファイルに立て、そのファイルのソース行番号に含まれる下位 6 桁のみをリストします。

入力レコードにプリンター制御文字や %SKIP 、 %PAGE ステートメントが含まれている場合は、それに応じて行間が空けられる。

リストの印刷を停止するには、 %NOPRINT 文を使用します。

リストの印刷を再開するには、 %PRINT 文を使用します。

MACRO オプションを指定すると、ソース・リストには、一次入力データ・セット内の %INCLUDE ステートメ ントの代わりに、含まれるテキストが表示されます。

ステートメントのネスト・レベル

NEST オプションを指定すると、ステートメント番号または行番号の右側に、ブロックレベルと DO レベルが、それぞれ LEV 、 NT という見出しで表示される。

次の例を参照してください。

```
Line.File LV NT
   1.0             A: PROC OPTIONS(MAIN);
   2.0     1       B: PROC;
   3.0     2          DCL K(10,10) FIXED BIN (15);
   4.0     2          DCL Y FIXED BIN (15) INIT (6);
   5.0     2          DO I=1 TO 10;
   6.0     2  1          DO J=1 TO 10;
   7.0     2  2             K(I,J) = N;
   8.0     2  2          END;
   9.0     2  1       BEGIN;
  10.0     3  1          K(1,1)=Y;
  11.0     3  1       END;
  12.0     2  1    END B;
  13.0     1       END A;
```

ATTRIBUTE および相互参照表

ATTRIBUTES オプションを指定すると、コンパイラーはソース・プログラム中の識別子のリストと、宣言された属性およびデフォルトの属性を含む属性表を表示します。 XREF オプションを指定すると、コンパイラーはソース・プログラム中の識別子のリストと、その識別子が出現する文のファイル番号と行番号を含む相互参照表を表示します。

ATTRIBUTES と XREF の両方を指定すると、2 つのテーブルが組み合わされる。 これらの表では、識別子を明示的に宣言した場合、コンパイラーはその DECLARE のファイル番号と行番号を列挙する。 コンテキストで宣言された変数は +++++ とマークされ、 それ以外の暗黙的に宣言された変数は ***** とマークされます。

属性テーブル

属性テーブルには、ソース・プログラムにおける ID のリストが、宣言された属性およびデフォルト属性とともに含まれます。

コンパイラーは、属性 INTERNAL と REAL を決して含めない。 それぞれの相反する属性、 EXTERNAL と COMPLEX が表示されない限り、それらを仮定することができる。

ファイル識別子の場合、属性 FILE は常に表示され、属性 EXTERNAL は該当する場合に表示されます。それ以外の場合、コンパイラーは明示的に宣言された属性のみをリストアップします。

ENTRY 属性が適用されない限り、 OPTIONS 属性は表示されず、その場合は以下のオプションのみが適切に表示されます：

- ASSEMBLER
- COBOL
- FETCHABLE
- NODESCRIPTOR
- RETCODE

コンパイラーは、配列用の次元属性をまず印刷します。 境界は配列宣言のとおりに印刷されますが、式はアスタリスクに 置き換えられます。ただし、式がコンパイラーにより定数に還元された場合は、 定数の値が印刷されます。

文字ストリング、ビット・ストリング、漢字ストリング、または区域変数 の場合、コンパイラーは宣言のとおりに長さを印刷しますが、式はアスタリスク に置き換えられます。ただし、式がコンパイラーにより定数に還元された場合は、 定数の値が印刷されます。

相互参照テーブル

ATTRIBUTES と XREF を指定すると、相互参照テーブルと属性テーブルが結合されます。 名前に対する属性のリストは、ファイル番号と行番号で識別されます。

以下の条件下では、相互参照テーブルの `Sets:` 部分に ID が表示されます。

- その ID が代入ステートメントのターゲットである。
- DO 、ループ制御変数として使用される。
- これは、 ALLOCATE または LOCATE ステートメントの SET オプションで使用される。
- これは、 DISPLAY ステートメントの REPLY オプションで使用される。

未参照の ID がある場合、それらは別個のテーブルに示されます。

集合長さテーブル

AGGREGATE オプションにより、長さの集計表が得られる。 このテーブルには、配列ではなく構造体が組み込まれています。この構造体は非固定エクステントを保持し、構造体内部のエレメントのサイズとオフセットは、不正確であるか、または * として指定されます。

リストされた集合の場合、テーブルには次の情報が入っています。

- 宣言された集合のロケーション
- 集合名と集合内の各エレメント
- 集合の先頭からの各エレメントのバイト・オフセット
- 各エレメントの長さ
- 各集合、構造体、および副構造体の全長
- 各エレメントの次元の合計数

データ長テーブルに示されているデータ・オフセットを解釈する場合は注意が必要です。 奇数オフセットは、ハーフワード位置合わせ、フルワード位置合わせ、または ダブルワード位置合わせが行われていないデータ・エレメントを必ずしも表しません。 ある構造体または そのエレメント用に位置合わせ済みの属性

を指定したり暗示したりすると、 テーブルの初めに対して適切な位置合わせが行われていることを テーブルが示さなくても、適切な位置合わせ要件はその 構造体内の他のエレメントに合致します。

2 つの構造要素の間にパディングがある場合、適切な診断情報とともに /*PADDING*/ のコメントが表示される。

ステートメント・オフセット・アドレス

LIST コンパイル・オプションを使用すると、コンパイラーはコンパイラー・ リストに疑似アセンブラー・リストを組み込みます。 このリストには、BLKOFF コンパイラー・オプションの設定によって異なる意味を持つオフセットが命令ごとに含まれています。

- BLKOFF オプションでは、このオフセットは、その命令が属する関数またはサブルーチンのプライマリエントリポイントからのオフセットである。 つまり、このオプションのもとでは、オフセットはそれぞれの新規ブロックごとに再設定されます。
- NOBLKOFF オプションでは、このオフセットはコンパイル・ユニットの開始点からの命令のオフセットとなる。 つまり、このオプションのもとではオフセットは累積されます。

疑似アセンブラー・リストには、各ブロックのコードの終わりに現行モジュールの開始位置 からのブロックのオフセットも含まれています (それぞれのステートメントに表示されるオフセット を、ブロックまたはモジュールのオフセットに変換できるようにするため)。

これらのオフセットを、ランタイム・エラー・メッセージに示されたオフセットと共に使用して、このメッセージが該当するステートメントを判別できます。

OFFSET オプションは、各ステートメントについて、そのステートメントに属する最初の命令のオフセットを示すテーブルを生成する。

Finding statement number（コンパイラ・リストの例 ）に示した例では、 SUB1 エントリからのオフセット +98 で条件が発生したことを示すメッセージが表示されている。 コンパイラー・リストの抜粋は、行番号 8 に関連付けられたこのオフセットを示しています。 ステートメント番号の検索（ランタイムメッセージの例）の場合、この誤ったステートメントからのランタイム出力が表示されます。

ステートメント番号の検索 (コンパイラー・リストの例)

```
Compiler Source
    Line.File
       1.0
       2.0      TheMain: proc options( main );
       3.0        call sub1();
       4.0        Sub1: proc;
       5.0          dcl (i, j) fixed bin(31);
       6.0
       7.0          i = 0; j = 0;
       8.0          j = j / i;
       9.0          put skip data( j );
      10.0        end Sub1;
      11.0      end TheMain;

 . . .

OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

000000                    000002 |        THEMAIN  DS       0D
000000  47F0  F024        000002 |                 B        36(,r15)
000004  01C3C5C5                                           CEE eyecatcher
000008  000000B0                                           DSA size
00000C  000001F8                                           =A(PPA1-THEMAIN)
000010  47F0  F001        000002 |                 B        1(,r15)

 . . .

000000                    000004 |        SUB1     DS       0D
000000  47F0  F024        000004 |                 B        36(,r15)
000004  01C3C5C5                                           CEE eyecatcher
000008  00000140                                           DSA size
00000C  00000190                                           =A(PPA1-SUB1)
000010  47F0  F001        000004 |                 B        1(,r15)

...

000086  5020  D0B8        000007 |                 ST       r2,I(,r13,184)
```

```
00008A  1842             000007 |                 LR       r4,r2
00008C  5040  D0BC       000007 |                 ST       r4,J(,r13,188)
000090  5800  D0B8       000008 |                 L        r0,I(,r13,184)
000094  8E40  0020       000008 |                 SRDA     r4,32
000098  1D40             000008 |                 DR       r4,r0
00009A  1805             000008 |                 LR       r0,r5
00009C  5000  D0BC       000008 |                 ST       r0,J(,r13,188)
0000A0  4100  D0C0       000009 |                 LA       r0,_temp1(,r13,192)
0000A4  5000  D130       000009 |                 ST       r0,_temp2(,r13,304)
0000A8  A708  5A88       000009 |                 LHI      r0,H'23176'
0000AC  4000  D0EC       000009 |                 STH      r0,_temp1(,r13,236)
0000B0  5800  6004       000009 |                 L        r0,SYSPRINT(,r6,4)
0000B4  5000  D12C       000009 |                 ST       r0,_temp2(,r13,300)
0000B8  4100  0001       000009 |                 LA       r0,1
0000BC  5000  D0C0       000009 |                 ST       r0,_temp1(,r13,192)
0000C0  4100  D128       000009 |                 LA       r0,_temp2(,r13,296)

...
```

ステートメント番号の検索 (ランタイム・メッセージの例)

```
Message :

IBM0301S ONCODE=320  The ZERODIVIDE condition was raised.
         From entry point SUB1 at compile unit offset +00000098
         at entry offset +00000098 at address 0EB00938.
```

ダンプと ON ユニット SNAP エラー・メッセージ内に示された項目オフセット をこのテーブルと対比すれば、誤りのあるステートメントを見つけ出すことが できます。 このステートメントを識別するには、メッセージ内で指名された ブロックに関連したテーブルのセクションを探し出してから、メッセージ 内のオフセット以下かまたはそれと等しい最大オフセットを見つけ出し ます。 このオフセットに関連したステートメント番号が求める番号です。

ストレージ・オフセット・リスト

MAP コンパイル・オプションを使用すると、コンパイラーはコンパイラー・ リストにストレージ・オフセット・リストを組み込みます。

ストレージ・オフセット・リストは、以下のレベル 1 変数がプログラムで使用された場合に、 これらのストレージ内のロケーションを示します。

- AUTOMATIC
- CONTROLLED ただし PARAMETERs
- STATIC ただし、。 ENTRY CONSTANTs FETCHABLE

このリストには、コンパイラーが生成した一時データの一部も組み込まれます。

エクステントを調整できる AUTOMATIC 変数の場合、このテーブルには 2 つのエントリーがある：

- 変数名の前に `_addr` が付いた項目 - 変数のアドレスのロケーションを示します
- 変数名の前に `_desc` が付いた項目 - 変数の記述子のアドレスのロケーションを示します

STATIC と CONTROLLED 変数の場合、格納場所は RENT/NORENT コンパイラー・オプションに依存し、NORENT オプションが有効な場合は、CONTROLLED 変数の格納場所も WRITABLE/NOWRITABLE コンパイラー・オプションに依存します。

ストレージ・オフセット・リストの最初の列には **IDENTIFIER** というラベルが付けられています。 この列には、4 列目に示されているロケーションが代入される変数の名前が含まれています。

ストレージ・オフセット・リストの 2 番目の列は、**DEFINITION** というラベルが付けられ、*"B-F:N"* という形式のストリングを含んでいます。

- ここで、*B* は変数が宣言されたブロックの番号です。

  このブロック番号に応じてブロック名をブロック名リストで検索できます。このブロック名リストはストレージ・オフセット・リスト (および、存在する場合は、疑似アセンブリー・リスト) の前にあります。
- ここで、*F* は変数が宣言されたソース・ファイルの番号です。

このファイル番号に対応するファイル名を、ファイル参照テーブル (コンパイル・リスト全体の終わりのあたりにある) で検索できます。

- ここで、*N* は変数がソース・ファイル内で宣言されたソース行の番号です。

ストレージ・オフセット・リストの 3 番目の列は、**ATTRIBUTES** というラベルが付けられ、変数のストレージ・クラスを示します。

ストレージ・オフセット・リストの 4 番目の列は、 ラベルがなく、変数のロケーションを検索する方法を示します。

このストレージ・オフセット・リストはブロック別および変数名別にソートされ、ユーザー変数のみも含まれます。 また、MAP オプションを指定すると、コンパイラーは以下のマップも生成します。

- すべての STATIC 変数を 16 進数オフセット順に並べた「静的マップ」
- 各ブロックについて、すべての AUTOMATIC 変数を 16 進オフセット順に並べた「自動マップ」

PL/I 言語のマッピング規則で、構造体を、構造体が開始していると思われる場所から最大で 8 バイト分オフセットする必要がある場合があります。 例えば、次のように宣言された AUTOMATIC 構造体 A を考えてみます。

```
         dcl
           1 A,
             2 B char(2),
             2 C fixed bin(31);
```

C は、4 バイト境界に位置合わせする必要があり、この構造体には 2 バイトの埋め込みが必要です。 ただし、 PL/I は、これらの 2 バイトを B の後ではなく、 B の前に置きます。 構造体の先頭の前にあるこの 2 バイトの「埋め込み」は、構造体のハング・バイト と呼ばれます。

これらのハング・バイトは、コンパイラーによって生成された「自動マップ」にも反映されます。 「ストレージ・オフセット・リスト」には、ハング・バイトが含まれない A のオフセットと長さが示されます。

```
   A     Class = automatic,   Location = 186 : 0xBA(r13),     Length = 6
```

一方、「自動マップ」には、ハング・バイトが含まれた A のオフセットと長さが示されます。

```
    OFFSET (HEX)   LENGTH (HEX)   NAME

         98              8        #MX_TEMP1
         A0             18        _Sfi
         B8              8        A
```

式および属性のリスト

DECOMP コンパイラー・オプションを使用すると、コンパイラーはコンパイラー・リストに、ソース・プログラムで使用されているすべての式の中間式とその属性を示すセ クションを追加します。

関連情報

23 ページの『DECOMP』

DECOMP オプションは、コンパイラに、コンパイルで使われた式の分解を与えるリストセクションを生成するように指示する。

ファイル参照テーブル

ファイル参照テーブルには、コンパイル時に読み取られるファイルに関する情報が登録されています。

ファイル参照テーブルは、以下のファイル情報がリストされた 3 つの列で構成されます。

- コンパイラーによってファイルに割り当てられた番号
- ファイルの組み込み元データ
- ファイルの名前

最初にリストされるファイルはソース・ファイルなので、 組み込み元の列にある最初の項目はブランクです。 この列にある後続のエントリーには組み込みステートメントの行番号が示され、その後にピリオドと、組み込みステートメントが含まれるソース・ファイルのファイル番号が続きます。

ファイルが PDS または PDSE のメンバーである場合、ファイル名は完全修飾データ・セット名とメンバー名を示します。

ファイルがサブシステム (ライブラリアンなど) を介して組み込まれた場合、ファイル名の形式は DD:*ddname*(*member*) になります。ただし、

- *ddname* は %INCLUDE （ddname が指定されていない場合は SYSLIB ）で指定された ddname である。
- *member* は、 %INCLUDE 文で指定されたメンバ名である。

メッセージと戻りコード

プリプロセッサーまたはコンパイラーがエラーまたはエラーの可能性を 検出すると、メッセージが生成されます。 コンパイラーは、コンパイル・ジョブまたはジョブ・ステップごとに、成功または失敗の程度を示す戻りコードを生成します。

メッセージ

プリプロセッサーが生成したメッセージは、プリプロセッサーが処理した ステートメントのリストの直後のリストに印刷されます。 NOTE ステートメントを使えば、前処理段階で独自のメッセージを生成できる。このようなメッセージは、特定の置換が何回行われたかを示すのに 使用できます。 コンパイラーが生成したメッセージはリストの最後に印刷されます。

メッセージが生成されないコンパイルの場合、コンパイラーは、コンパイラー・メッセージがリストされるはずの行に「コンパイラー・メッセージはありません （no compiler messages)」というメッセージを組み込みます。

メッセージは次のフォーマットで示されます。

```
PPPnnnnI X
```

PPP
メッセージの発信元を識別する接頭部です (例えば、IBM は PL/I コンパイラーを示します)。

nnnn
4 桁のメッセージ番号です。

X
重大度コードを識別するものです。

すべてのメッセージはその重大度によって等級付けされ、重大度コードは I, W, E, S と U である。

コンパイラーは、 112 ページの表 5 に示すように、 FLAG オプションで指定された重大度以上のメッセージだけをリストアップします。

表 *5.* リストされたメッセージの最低重大度を選択するための *FLAG* オプションの使用

Type of message	Option
Information	FLAG(I)
Warning	FLAG(W)
Error	FLAG(E)
Severe Error	FLAG(S)
Unrecoverable Error	Always listed

各メッセージのテキスト、説明、および推奨プログラマー応答については、「*Enterprise PL/I* メッセージおよびコード」を参照してください。

戻りコード

各コンパイル・ジョブまたはジョブ・ステップごとにコンパイラーは、 操作がどの程度成功または失敗したかをオペレーティング・システムに 示すための戻りコードを生成します。 z/OS の場合、このコードはステップの終わり メッセージに 現れますが、その前には各ステップ別のジョブ制御ステートメントと ジョブ・スケジューラー・メッセージのリストが入っています。

各重大度コードとそれに対応する戻りコードについて、113 ページの表 6 で説明します。

表 *6. PL/I* エラー・コードと戻りコードの説明

Severity code	Return code	Message type	Description
I	0000	Informational	The compiled program should run correctly. The compiler might inform you of a possible inefficiency in your code or some other condition of interest.
W	0004	Warning	A statement might be in error (warning) even though it is syntactically valid. The compiled program should run correctly, but it might produce different results than expected or be significantly inefficient.
E	0008	Error	A simple error fixed by the compiler. The compiled program should run correctly, but it might produce different results than expected.
S	0012	Severe	An error not fixed by the compiler. If the program is compiled and an object module is produced, it should not be used.
U	0016	Unrecoverable	An error that forces termination of the compilation. An object module is not successfully created.

注 : Compiler messages are printed in groups according to these severity levels.

例

このコンパイラ・リストの例は、コンパイラが以下の msgsumm プログラムを以下のオプションでコンパイルしたときに生成されます： PP(SQL,MACRO,CICS), SOURCE, FLAG(I), INSOURCE, MSGSUMMARY(XREF).

```
msgsumm: proc;

  exec sql include sqlca;

  exec cics what now;

  exec cics not this;

  %dcl z0 fixed bin;
  %dcl z1 fixed dec;
end;
```

注 : このプログラムには意図的な誤りがあります。 MSGSUMMARY オプションが指定されているので、コンパイラーはリストの最後に Summary of Messages セクションを含める。 MSGSUMMARY オプションの XREF サブオプションが指定されているため、このセクションには、サマリーの各メッセージに関連する行番号も含まれる。

コンパイラー・リスト例

```
5655-PL6  IBM(R) Enterprise PL/I for z/OS  6.1      (Built:20220429)
2022.05.09 12:58:42   Page     1
                    Options Specified
  Install:
  Command: +DD:OPTIONS
  File: DD:OPTIONS
  PP(SQL,MACRO,CICS),S,F(I),IS,MSGSUMMARY(XREF)
  Install:
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     2
 SQL (Built:20220407) Preprocessor Source
    Line.File
       1.0      msgsumm: proc;
```

```
       2.0
       3.0        exec sql include sqlca;
       4.0
       5.0        exec cics what now;
       6.0        exec cics not this;
       7.0
       8.0        %dcl z0 fixed bin;
       9.0        %dcl z1 fixed dec;
      10.0
      11.0      end;
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     3
 SQL Preprocessor Options Used
     CCSID0
   NOCODEPAGE
     DEPRECATE( STMT() )
   NOEMPTYDBRM
   NOINCONLY
     LINEONLY
   NOWARNDECP
 DB2 for z/OS Coprocessor Options Used
   APOST
   APOSTSQL
   ATTACH(TSO)
   CCSID(500)
   CONNECT(2)
   DEC(15)
   DECP(DSNHDECP)
   FLOAT(S390)
   NEWFUN(V11)
   TWOPASS
   PERIOD
   STDSQL(NO)
   SQL(DB2)
   NOXREF
   NO SOURCE
   DSNHDECP LOADED FROM - (DSN.DB2VB10.SDSNLOAD(DSNHDECP))
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     4
 SQL Preprocessor Messages
 Message       Line.File Message Description
 IBM3250I W       3.0    DSNH053I DSNHPSRV  NO SQL STATEMENTS WERE FOUND
```

```
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     5
 MACRO (Built:20220315) Preprocessor Source
    Line.File
       1.0      msgsumm: proc;
       1.0
       2.0
       3.0
       3.0      /*$*$*$
       3.0        exec sql include sqlca
       3.0       $*$*$*/
       3.0      DCL
       3.0        1 SQLCA  ,
       3.0          2 SQLCAID     CHAR(8),
       3.0          2 SQLCABC     FIXED BIN(31),
       3.0          2 SQLCODE     FIXED BIN(31),
       3.0          2 SQLERRM     CHAR(70) VAR,
       3.0          2 SQLERRP     CHAR(8),
       3.0          2 SQLERRD(6)  FIXED BIN(31),
       3.0          2 SQLWARN,
       3.0            3 SQLWARN0  CHAR(1),
       3.0            3 SQLWARN1  CHAR(1),
       3.0            3 SQLWARN2  CHAR(1),
       3.0            3 SQLWARN3  CHAR(1),
       3.0            3 SQLWARN4  CHAR(1),
       3.0            3 SQLWARN5  CHAR(1),
       3.0            3 SQLWARN6  CHAR(1),
       3.0            3 SQLWARN7  CHAR(1),
       3.0          2 SQLEXT,
       3.0            3 SQLWARN8  CHAR(1),
       3.0            3 SQLWARN9  CHAR(1),
       3.0            3 SQLWARNA  CHAR(1),
       3.0            3 SQLSTATE  CHAR(5);
       3.0
       4.0
       5.0        exec cics what now;
       6.0        exec cics not this;
       7.0
```

```
       8.0        %dcl z0 fixed bin;
       9.0        %dcl z1 fixed dec;
      10.0
      11.0      end;
```

Compiler listing example (continued)

```
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     6
 MACRO Messages
 Message       Line.File Message Description
 IBM3552I E       8.0    The statement element BIN is invalid. The statement will be ignored.
 IBM3552I E       9.0    The statement element DEC is invalid. The statement will be ignored.
 IBM3258I W       8.0    Missing ; assumed before BIN.
 IBM3258I W       9.0    Missing ; assumed before DEC.
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     7
 CICS (Built:20220315) Preprocessor Source
    Line.File
       1.0      msgsumm: proc;
       1.0
       2.0
       3.0
       3.0      /*$*$*$
       3.0        exec sql include sqlca
       3.0       $*$*$*/
       3.0      DCL
       3.0        1 SQLCA  ,
       3.0          2 SQLCAID     CHAR(8),
       3.0          2 SQLCABC     FIXED BIN(31),
       3.0          2 SQLCODE     FIXED BIN(31),
       3.0          2 SQLERRM     CHAR(70) VAR,
       3.0          2 SQLERRP     CHAR(8),
       3.0          2 SQLERRD(6)  FIXED BIN(31),
       3.0          2 SQLWARN,
       3.0            3 SQLWARN0  CHAR(1),
       3.0            3 SQLWARN1  CHAR(1),
       3.0            3 SQLWARN2  CHAR(1),
       3.0            3 SQLWARN3  CHAR(1),
       3.0            3 SQLWARN4  CHAR(1),
       3.0            3 SQLWARN5  CHAR(1),
       3.0            3 SQLWARN6  CHAR(1),
       3.0            3 SQLWARN7  CHAR(1),
       3.0          2 SQLEXT,
       3.0            3 SQLWARN8  CHAR(1),
       3.0            3 SQLWARN9  CHAR(1),
       3.0            3 SQLWARNA  CHAR(1),
       3.0            3 SQLSTATE  CHAR(5);
       3.0
       4.0
       5.0        exec cics what now;
       6.0        exec cics not this;
       7.0
      10.0
      11.0      end;
```

Compiler listing example (continued)

```
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     8
 CICS Messages
 Message       Line.File Message Description
 IBM3750I S       5.0    DFH7059I S  WHAT COMMAND IS NOT VALID AND IS NOT TRANSLATED.
 IBM3750I S       6.0    DFH7059I S  NOT COMMAND IS NOT VALID AND IS NOT TRANSLATED.
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page     9
 No Compiler Messages
 File Reference Table
   File    Included From  Name
      0                   DD:SYSIN
5655-PL6  IBM(R) Enterprise PL/I for z/OS  msgsumm: proc;
2022.05.09 12:58:42   Page    10
 Summary of Messages
 Component Message     Total  Default Message Description
 SQL       IBM3250I W      1  DSNH053I DSNHPSRV  NO SQL STATEMENTS WERE FOUND
                              Refs: 3.0
 MACRO     IBM3552I E      2  The statement element %1 is invalid. The statement will be
ignored.
                              Refs: 8.0 9.0
 MACRO     IBM3258I W      2  Missing %1 assumed before %2.
```

```
                              Refs: 8.0 9.0
 CICS      IBM3750I S      2  DFH7059I S  WHAT COMMAND IS NOT VALID AND IS NOT TRANSLATED.
                              Refs: 5.0 6.0
 Compiler  <none>

 Component    Return Code    Messages (Total/Suppressed)    Time
 SQL              4               1  /  0                   0 secs
 MACRO            8               4  /  0                   0 secs
 CICS            12               2  /  0                   0 secs
 Compiler         0               0  /  0                   0 secs
 End of compilation
```

Compiler listing example (continued)

第 2 章 PL/I プリプロセッサー

PL/I コンパイラーを使用するときは、ご使用のプログラムにおいて組み込みプリプロセッサーを 1 つ以上指定できます。 組み込みプリプロセッサー、マクロ・プリプロセッサー、SQL プリプロセッサー、または CICS プリプロセッサーを指定できます。 また、これらのプリプロセッサーを呼び出す順序を指定できます。

- インクルード・プリプロセッサーは、特殊なインクルード・ディレクティブを 処理し、外部ソース・ファイルを取り込みます。
- マクロ・プリプロセッサーは、% ステートメントとマクロに基づいていて、ソース・プログラムを変更します。
- SQL プリプロセッサーは、ソース・プログラムを変更し、EXEC SQL ステートメントを PL/I ステートメントに変換します。
- CICS プリプロセッサーは、ソース・プログラムを変更し、EXEC CICS ステートメントを PL/I ステートメントに変換します。

各プリプロセッサーは、ニーズに合わせて処理を調整するために使用できるオプションをいくつかサポートしています。

コンパイル時のオプション MDECK, INSOURCE、および SYNTAX は、PP オプションも指定した場合にのみ意味を持つ。

関連情報

57 ページの『MDECK』
MDECK オプションは、z/OS の SYSPUNCH DD 文で定義されたファイルか、 z/OS UNIX の.dek ファイルのいずれかに、プリプロセッサがその出力のコピーを生成することを指定する。

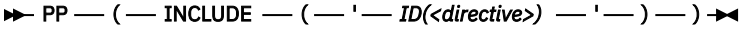
44 ページの『INSOURCE』
INSOURCE オプションは、コンパイラが PL/I マクロの前にソース・プログラムのリストを含めることを指定します、 CICS または SQL プリプロセッサがそれを翻訳する前に、コンパイラがソース・プログラムのリストを含めることを指定します。

89 ページの『SYNTAX』
SYNTAX オプションは、MACRO オプションを指定したときに、回復不能なエラーが発生しない限り、コンパイラが前処理の後に構文チェックを続行することを指定する。 コンパイラがコンパイルを続行するかどうかは、 NOSYNTAX オプションで指定されたエラーの深刻度によって決まる。

インクルード・プリプロセッサー

インクルード・プリプロセッサーは、 PL/I ディレクティブ %INCLUDE 以外のインクルード・ディレクティブを使って、外部ソース・ファイルをプログラムに組み込むことができます。

次の構文図は、 INCLUDE プリプロセッサーがサポートするオプションを示している：



ID

インクルード・ディレクティブの名前を指定します。 最初の一続きの非ブランク文字としてのこのディレクティブで始まる行は、インクルード・ディレクティブとして扱われます。

指定するディレクティブの後に、1 つ以上のブランク、およびインクルード・メンバー名が必要で、最後にオプションでセミコロンを付けることができます。 *ddname(membername)* の構文はサポートされません。

次の例では、1 つ目のインクルード・ディレクティブは有効で、2 つ目のものは無効です。

```
++include payroll
++include syslib(payroll)
```

© Copyright IBM Corp. 1999, 2025

例 1

次の例では、-INC (および場合によっては先行ブランク) から始まる行がすべて、インクルード・ディレクティブとして扱われます。

```
pp( include( 'id(-inc)'))
```

例 2

次の例では、++INCLUDE (および場合によっては先行ブランク) から始まる行がインクルード・ディレクティブとして扱われます。

```
pp( include( 'id(++include)'))
```

マクロ・プリプロセッサー

マクロを使用すれば、インプリメンテーションの詳細と処理対象のデータを隠して演算のみを表す方法で、通常使用される PL/I コードを作成できます。 汎用のサブルーチンとは対照的に、マクロでは、個別用途ごとに必要となるコードのみを生成できます。 マクロ・プリプロセッサは、 MACRO オプションまたは PP(MACRO) オプションを指定して呼び出すことができる。

PP(MACRO) はオプションなしで指定することも、118 ページの『マクロ・プリプロセッサーのオプション』で説明されているオプションとともに指定することもできます。

これらすべてのオプションに対してデフォルトが指定された場合のマクロ・プリプロセッサーの動作は OS PL/I V2R3 マクロ・プリプロセッサーの動作と同じです。

オプションを指定する場合、リストは引用符で囲む必要がある（一重でも二重でも、一致すればよい）。例えば、FIXED(BINARY) オプションを指定するには、 PP(MACRO('FIXED(BINARY)')) を指定しなければならない。

複数のオプションを指定する場合は、コンマまたは 1 つ以上のブランクでオプションを分離する必要があります。 たとえば、CASE(ASIS) と RESCAN(UPPER) オプションを指定するには、PP(MACRO('CASE(ASIS) RESCAN(UPPER)')) または PP(MACRO("CASE(ASIS),RESCAN(UPPER)")) を指定する。 オプションは任意の順序で指定できます。

コンパイラーのマクロ・プリプロセス機能については、「*PL/I Language Reference*」を参照してください。

マクロ・プリプロセッサーのオプション

このセクションでは、マクロ・プリプロセッサーがサポートするオプションについて説明します。

表 *7. MACRO* プリプロセッサー・オプション、および *IBM* 提供のデフォルト値

MACRO Preprocessor option	Abbreviated name	z/OS default
CASE(UPPER | ASIS)	-	CASE(ASIS)
DBCS(EXACT | INEXACT)	-	DBCS(INEXACT)
DEPRECATE (ENTRY(*entry name*))	-	-
DEPRECATENEXT (ENTRY(*entry name*))	-	-
EOLCOMM | NOEOLCOMM	-	EOLCOMM
FIXED(DECIMAL | BINARY)	-	FIXED(DECIMAL)
ID('*directive*')	-	-

表 7. *MACRO* プリプロセッサー・オプション、および *IBM* 提供のデフォルト値 *(*続き*)*

MACRO Preprocessor option	Abbreviated name	z/OS default
IGNORE(NOPRINT) | NOIGNORE	-	NOIGNORE
INCONLY | NOINCONLY	-	NOINCONLY
NAMEPREFIX(*character*) | NONAMEPREFIX	-	NONAMEPREFIX
RESCAN(ASIS | UPPER)	-	RESCAN(ASIS)

CASE

このオプションは、プリプロセッサーが入力テキストを大文字に変換する必要があるかどうかを指定します。

UPPER
入力テキストを大文字に変換します。

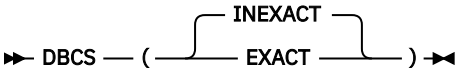
ASIS
入力テキストは「現状のまま」です。

CASE(ASIS)がデフォルトです。

GRAPHIC 、 CASE(ASIS) 。

DBCS

このオプションは、テキストの置換時にプリプロセッサが DBCS を正規化するかどうかを指定します。

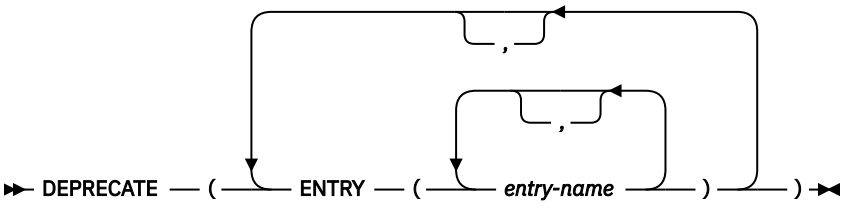


EXACT
入力テキストは「現状のまま」です。プリプロセッサーは <kk.B> と <kk>B を別の名前として扱います。

INEXACT
入力テキストは「正規化」されます。プリプロセッサーは <kk.B> と <kk>B を同じ名前の 2 つのバージョンとして扱います。

DEPRECATE

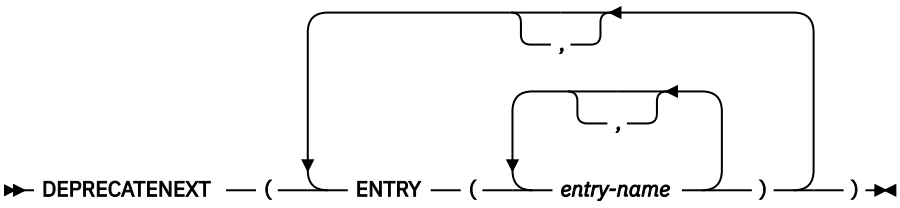
このオプションは、エラー・メッセージを発行して非推奨にしたいマクロ・プロシージャーの使用箇所にフラグを立てます。



ENTRY
これは、*entry-name* という名前のマクロ・プロシージャーのすべての使用箇所にフラグを立てます。

DEPRECATENEXT

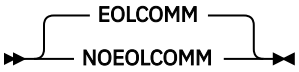
このオプションは、警告メッセージを発行して非推奨にしたいマクロ・プロシージャーの使用箇所にフラグを立てます。



ENTRY
これは、*entry-name* という名前のマクロ・プロシージャーのすべての使用箇所にフラグを立てます。

EOLCOMM (E)

EOLCOMM は、マクロ呼び出しの一部であるテキストでも、マクロプリプロセッサが // を行末コメントマーカーとして認識するように指定します。



次のマクロ呼び出しを考えてみます。

```
%dcl trace entry;
Trace( level // 8 );
```

EOLCOMM では、これは事実上マクロの引数として "level "を渡していることになり、次の行にはマクロの呼び出しを閉じる") "が必要である。

しかし、 NOEOLCOMM では、"level // 8 "というテキストがマクロの引数として渡され、") "がマクロの呼び出しを閉じる。

EOLCOMM がデフォルトです。

FIXED

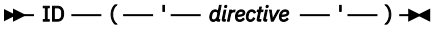
このオプションは、 FIXED 変数のデフォルト・ベースを指定する。

DECIMAL
FIXED 変数は という属性を持つ。 REAL FIXED DEC(5)

BINARY
FIXED 変数は という属性を持つ。 REAL SIGNED FIXED BIN(31)

ID

このオプションは、インクルード・ディレクティブの名前を指定します。 最初の一続きの非ブランク文字としてのこのディレクティブで始まる行は、インクルード・ディレクティブとして扱われます。



指定するディレクティブの後に、1 つ以上のブランク、 およびインクルード・メンバー名が必要で、最後にオプションでセミコロンを付けることができます。 *ddname(membername)* の構文はサポートされません。

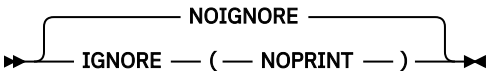
次の場合、

```
PP(MACRO("ID('++INCLUDE')"))
```

MACRO プリプロセッサーは、最初の非ブランク文字として ++INCLUDE があり、その後にブランクと ID が続く行を、あたかも ++INCLUDE X が実際に %INCLUDE X; (元のソースで X の後に何もない場合のみセミコロンが追加される) であるかのように扱います。

IGNORE

このオプションは、MACRO プリプロセッサーが特定のステートメントを無視するかどうかを指定します。



IGNORE
プリプロセッサーが特定のステートメントを無視することを指定します。

IGNORE NOPRINT. IGNORE(NOPRINT) は、 プリプロセッサが ステートメントを無視することを指定します。 MACRO %NOPRINT

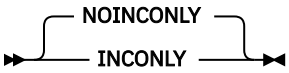
NOIGNORE
プリプロセッサが %NOPRINT ステートメントを無視しないように指定する。

デフォルトは NOIGNORE です。

INCONLY

INCONLY オプションは、プリプロセッサーで %INCLUDE および %XINCLUDE ステートメントのみを処理する必要があることを指定します。

NOINCONLY オプションは、 %INCLUDE と %XINCLUDE ステートメントだけでなく、プリプロセッサがすべてのプリプロセッサ・ステートメントを処理することを指定する。



INCONLY オプションが有効な場合、 INCLUDE も XINCLUDE もマクロとしては使えない：

- プロシージャー名
- ステートメント・ラベル
- 変数名

INCONLY オプションと NOINCONLY オプションは互いに排他的である。

互換性のため、デフォルトは NOINCONLY。

NAMEPREFIX

NAMEPREFIX オプションは、プリプロセッサー・プロシージャーと変数の名前が指定された文字で始まらなければならないことを指定します。

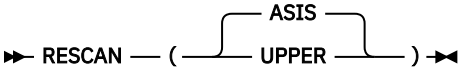
NONAMEPREFIX オプションは、プリプロセッサー・プロシージャーと変数の名前が特定の文字で始まる必要はないことを指定する。

文字は「現状のまま」で指定し、引用符で囲まないようにしてください。

デフォルトは NONAMEPREFIX です。

RESCAN

このオプションは、テキストの再スキャンのとき、プリプロセッサーが ID の大/小文字をどのように処理するかを指定します。



UPPER
再スキャンは大文字小文字を区別しません。

ASIS
再スキャンは大文字小文字を区別します。

このオプションの影響を見るため、次のコード・フラグメントについて考えてみましょう。

```
%dcl eins char ext;
%dcl text char ext;

%eins = 'zwei';

%text = 'EINS';
display( text );

%text = 'eins';
display( text );
```

PP(MACRO('RESCAN(ASIS)')) でコンパイルすると、2 番目の表示文の中で、 text の値が eins に置き換えられるが、それ以上の置き換えは起こらない。 これは、 RESCAN(ASIS)、 eins がマクロ変数 eins と一致しないからである。前者はそのままで、後者は大文字である。 そのため、次のテキストが生成されます。

```
DISPLAY( zwei );

DISPLAY( eins );
```

しかし、 PP(MACRO('RESCAN(UPPER)')) を使ってコンパイルした場合、2 つ目の表示文では、 text の値は eins に置き換えられるが、 RESCAN(UPPER) の下で、 eins がマクロ変数 eins と一致するため、さらに置き換えが行われる（両方とも大文字)。 そのため、次のテキストが生成されます。

```
DISPLAY( zwei );

DISPLAY( zwei );
```

まとめると、 RESCAN(UPPER) は大文字小文字を無視するが、 RESCAN(ASIS) は無視しない。

マクロ・プリプロセッサーの例

プリプロセッサーを使用してソース・デックを作成する方法を以下に例示します。

マクロ・プリプロセッサを使用してソース・デッキを作成する 」で示した例では、プリプロセッサ変数 USE に割り当てられた値に従って、ソース文はサブルーチン (CITYSUB) または関数 (CITYFUN) のいずれかを表します。

SYSPUNCH で使われる DSNAME は、プリプロセッサ出力が置かれるソース・プログラム・ライブラリを指定する。 通常、コンパイルが続行され、プリプロセッサー出力がコンパイルされます。

ソース・デックを作成するためのマクロ・プリプロセッサーの使用

```
//OPT4#8  JOB
 //STEP2 EXEC  IBMZC,PARM.PLI='MACRO,MDECK,NOCOMPILE,NOSYNTAX'
 //PLI.SYSPUNCH DD  DSNAME=HPU8.NEWLIB(FUN),DISP=(NEW,CATLG),UNIT=SYSDA,
 //          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
 //PLI.SYSIN  DD *
  /* GIVEN ZIP CODE, FINDS CITY                                      */
  %DCL USE CHAR;
  %USE = 'FUN'                  /* FOR SUBROUTINE, %USE = 'SUB' */ ;
  %IF USE = 'FUN' %THEN %DO;
  CITYFUN: PROC(ZIPIN) RETURNS(CHAR(16)) REORDER; /* FUNCTION         */
                       %END;
```

```
                %ELSE %DO;
CITYSUB: PROC(ZIPIN, CITYOUT) REORDER;     /* SUBROUTINE            */
  DCL CITYOUT CHAR(16);          /* CITY NAME                        */
                      %END;
  DCL (LBOUND, HBOUND) BUILTIN;
  DCL ZIPIN PIC '99999';        /* ZIP CODE                          */
  DCL 1 ZIP_CITY(7) STATIC,     /* ZIP CODE - CITY NAME TABLE        */
        2 ZIP PIC '99999' INIT(
                    95141, 95014, 95030,
                    95051, 95070, 95008,
                    0),           /* WILL NOT LOOK AT LAST ONE       */
        2 CITY CHAR(16) INIT(
                    'SAN JOSE', 'CUPERTINO', 'LOS GATOS',
                    'SANTA CLARA', 'SARATOGA', 'CAMPBELL',
                    'UNKNOWN CITY');  /* WILL NOT LOOK AT LAST ONE  */
  DCL I FIXED BIN(31);
  DO I = LBOUND(ZIP,1) TO             /* SEARCH FOR ZIP IN TABLE    */
         HBOUND(ZIP,1)-1              /* DON'T LOOK AT LAST ELEMENT */
         WHILE(ZIPIN ¬= ZIP(I));
  END;
%IF USE = 'FUN' %THEN %DO;
  RETURN(CITY(I));                    /* RETURN CITY NAME           */
                      %END;
                %ELSE %DO;
  CITYOUT=CITY(I);                    /* RETURN CITY NAME           */
                      %END;
END;
```

SQL プリプロセッサー

通常、PL/I プログラムのコーディングは、プログラムが Db2 データベースにアクセスする場合でもアクセスしない場合でも同じです。 しかし、データの取得、更新、挿入、削除を行うために Db2 データの取得、更新、挿入、削除 Db2 その他のサービスを使用するには、 SQL ステートメントを使用する必要があります。 PL/I アプリケーションでは、動的および静的な EXEC SQL ステートメントを使用できます。

Db2 とやり取りするには、次の作業を行う必要があります。

- 必要な SQL ステートメントをコーディングし、**EXEC SQL** で区切る。
- プリコンパイラーを使用する。 Db2 プリコンパイラーを使うか、Db2® for z/OS バージョン 9 リリース 1 以降を使う場合は、 PL/I PP(SQL()) コンパイラー・オプションを使ってコンパイルする。

EXEC SQL サポートを利用する前に、システムにアクセスする権限が必要です。 Db2 システムにアクセスする権限が必要です。 権限については、担当の Db2 データベース管理者に お問い合わせください。

プログラミングとコンパイルに関する考慮事項

PL/I SQL プリプロセッサを使用すると、PL/I コンパイラは、SQL ステートメントが埋め込まれたソース・プログラムをコンパイル時に処理します。 独立したプリコンパイルステップの使用は引き続きサポートされるが、 PL/I SQL プリプロセッサの使用を推奨する。

プリプロセッサを使用すると、デバッグ中に SQL ステートメントのみが表示される（生成された PL/I ソースは表示されない）ため、IBM Debug Tool を使用したインタラクティブなデバッグが強化されます。 ただし、 SQL プリプロセッサを使用するには、 Db2 for z/OS バージョン 9 リリース 1 以降が必要です。

プリプロセッサを使用すると、プログラムに対するプリコンパイラの制限の一部が解除される。 Db2 プリコンパイラを使うと、SQL プログラムに対するプリコンパイラの制限がいくつか解除される。 プリプロセッサで SQL ：

- SQL INCLUDE 。
- 構造化ホスト変数の完全修飾名を使用する。
- ネストされた PL/I プログラムのどのレベルでも SQL ステートメントをインクルードできる。
- PL/I データ型を指定できるところならどこでも、 SQL TYPE 属性を使用してください。 このような属性はすべてファクタリングすることができ、構造体要素や配列、 BASED のようなストレージクラスで使用することができます。
- LIKE 属性で宣言された変数をホスト変数として使用する。

SQL プリプロセッサは、 EXEC SQL ステートメント、 DECLARE ステートメント、および宣言のブロックを区切るステートメントを探してソースをスキャンするため、 PL/I ステートメントはすべて構文的に正しくなければならない。 ステートメントが正しくコーディングされていないと、プリプロセッサが BEGIN, DO, PACKAGE, PROCEDURE、または SELECT ステートメントに END ステートメントを検索するときに、プリプロセッサを誤解させる可能性がある。 このような不正確なコードを識別するために、 SQL プリプロセッサーは次のようにエラーフラグを立てる：

- 対応する右括弧がない左括弧
- SELECT セミコロンで終わらない文
- THEN キーワードを含まない IF 文
- 無効な記号で始まるステートメント

ソースに MACRO と SQL の両方の記述がある場合：

- SQL プリプロセッサの INCONLY オプションを使わずに SQL と MACRO プリプロセッサを起動したい場合は、まず MACRO プリプロセッサを起動しなければならない。 その場合、実行可能な PL/I ステートメントが許されるところならどこでも、 EXEC SQL ステートメントが許される。
- SQL プリプロセッサの INCONLY オプションで SQL と MACRO プリプロセッサを呼び出したい場合は、MACRO プリプロセッサの前に SQL プリプロセッサを呼び出せばよい。 その場合、EXEC SQL ステートメントが許されるのは、その直前にセミコロンがある場合のみである（もちろん、空白やコメントが介在する場合もある）。

SQL プリプロセッサは、 PL/I コンパイラと同じ方法で DBCS をサポートしている。 GRAPHIC PL/I コンパイラ・オプションが有効な場合、ソース言語要素の一部は DBCS および SBCS 文字で記述できます。 特に、ソース・プログラムでは、以下の場所で DBCS ：

- コメント内
- ステートメント・ラベルおよび ID の一部として
- G または M リテラルで

SQL ステートメントをプログラムおよびコンパイルする際、 PL/I 組み込み関数、コンパイラ・オプション、ステートメントの使用には以下の制限が適用されます：

- EXEC SQL ステートメントが PL/I に変換されるとき､以下の組み込み関数が生成されたコードに含まれることがある。 以下の組み込み関数のいずれかを構造体でエレメント名として使用する場合は、 これらを BUILTIN として明示的に宣言することも必要です。
  - ADDR
  - LENGTH
  - MAXLENGTH
  - PTRVALUE
  - SYSNULL
- BIND(: *t,p* :)のような PL/I 型関数を EXEC SQL ステートメント内で使用してはならない。
- プリプロセッサーを使用してコンパイルを行う場合、以下のコンパイラー・オプションは使用できません。
  - DFT(ASCII)
  - DFT(IEEE)
- PL/I プリプロセッサは、これらのステートメントを常に無視するので、 SQL クエリで DECLARE STATEMENT ステートメントを使用しないでください。

SQL プリプロセッサー・オプションでコンパイルすると、オブジェクト・モジュールやリストといった通常の コンパイラー出力とともに、データベース・リクエスト・モジュール ( ) が生成される。 Db2PL/I データベース・リクエスト・モジュール (DBRM) が生成されます。 バインド処理の入力として Db2 バインド処理への入力として、 DBRM データセットには、プログラム中の SQL ステートメントとホスト変数に関する情報が含まれている。 しかし、バインドやランタイム処理において、 DBRM の情報のすべてが重要なわけではない。 例えば、 DBRM の HOST 値が PL/I 以外の言語を指定している場合、心配する必要はない。 こ

れは、HOST 値のインストール・デフォルトとして他の言語が選択されていることを意味するだけであって、このことがプログラムのバインド処理やランタイム処理に影響することはありません。

EMPTYDBRM オプションが有効で、ソースが以下の条件のいずれかを満たす場合、プリプロセッサは翻訳を必要とする文がないことを示すメッセージを発行し、 DBRM を作成しません：

- ソースには EXEC SQL の記述はない。
- ソースには、 EXEC SQL INCLUDE SQLCA と EXEC SQL INCLUDE SQLDA 以外の EXEC SQL INCLUDE ステートメントのみが含まれています。

PL/I コンパイラ・リストには、プリプロセッサが生成するエラー診断（ SQL ステートメントの構文エラーなど）が含まれています。 EXEC SQL ステートメントのリストは、オリジナルのソースに似た読みやすいフォーマットで表示される。

プリプロセッサーを使用するには、以下の操作を行う必要があります。

- プログラムのコンパイル時に次のオプションを指定する。

```
PP(SQL('options'))
```

このコンパイラー・オプションは、コンパイラーが統合 SQL プリプロセッサーを起動することを示す。 SQL キーワードの後の括弧内に、 SQL 処理オプションのリストを指定する。 これらのオプションはコンマまたはスペースで区切ることができ、オプションのリストは引用符 で囲む必要があります (単一引用符か二重引用符を使用し、同じ種類の引用符で囲む必要があります)。

例えば、PP(SQL('DATE(USA),TIME(USA)') は、DATE と TIME の両方のデータ型に USA フォーマットを使うようにプリプロセッサに指示する。

また、 LOB をサポートするには、以下のオプションを指定する必要があります：

```
LIMITS( FIXEDBIN(31,63)  FIXEDDEC(15,31) )
```

SQL プリプロセッサー・オプションを指定する別の方法は、 PPSQL コンパイラー・オプションを使うことである。 この使用方法について詳しくは、65 ページの『PPSQL』を参照してください。

- コンパイル・ステップの JCL に、以下のデータ・セットの DD ステートメントを含める：
  - Db2 ロード・ライブラリー (*prefix*.SDSNLOAD)

    SQL プリプロセッサーは Db2SQL モジュールを呼び出す。 このため、Db2 ロード・ライブラリーのデータ・セット名を、コンパイル・ステップ用の STEPLIB 連結に組み込む必要があります。
  - SQL INCLUDE ステートメント用ライブラリ

    プログラムに、ソース・プログラムへの二次入力を指定する SQL INCLUDE *member-name* ステートメントが含まれている場合、コンパイル・ステップの SYSLIB 連結に、 *member-name* を含むデータ・セットの名前を含める必要がある。
  - DBRM・ライブラリー

    PL/I 。 Db2 データベース要求モジュール(DBRM)を生成し、 DBRM が書き込まれるデータセットを指定するために、 DBRMLIB DD ステートメントが必要である。

    DBRMLIB DD 、コンパイル中に複数回開閉できるデータセットを指定しなければならない。

  例えば、 JCL に次のような行があるかもしれない：

```
//STEPLIB  DD  DSN=DSNA10.SDSNLOAD,DISP=SHR
//SYSLIB   DD  DSN=PAYROLL.MONTHLY.INCLUDE,DISP=SHR
//DBRMLIB  DD  DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
```

SQL プリプロセッサー・オプション

このセクションでは、SQL プリプロセッサーがサポートするオプションについて説明します。

SQL プリプロセッサに渡すことができるオプションは、 PL/I SQL プリプロセッサで処理されるものと、SQL プリプロセッサのコプロセッサで処理されるものの 2 種類です。 Db2 コプロセッサによって処理さ

れるものです。 これらのオプションは、 PP(SQL('*option-list*')) オプションのオプション文字列で指定する必要があり、オプション文字列の中でオプションを混在させることができる。

SQL プリプロセッサ・オプションを指定する場合、オプションのリストは引用符で囲む必要がある。 例えば、 CCSID0 オプションを指定するには、 PP(SQL('CCSID0')) を指定しなければならない。

126 ページの表 8 は、プリプロセッサ・オプションと、(もしあれば) その省略形、および -supplied default values を一覧表示します。 PL/I SQL IBM この表では、相互排他的なオプションは縦棒 (|) によって分離されています。 また括弧 ([ ]) は、囲まれたオプションが省略可能であることを示しています。

Db2 コプロセッサー・オプションについて詳しくは、「*Db2 for* アプリケーション・プログラミングと *SQL* ガイド *z/OS*」を参照してください。

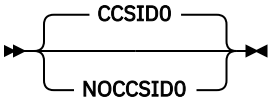
表 *8. SQL* プリプロセッサー・オプション、および *IBM* 提供のデフォルト値

SQL Preprocessor option	Abbreviated name	z/OS default
CCSID0 | NOCCSID0	-	CCSID0
CODEPAGE | NOCODEPAGE	-	NOCODEPAGE
DEPRECATE(STMT([EXPLAIN | GRANT | REVOKE | SET_CURRENT_SQLID]))	-	DEPRECATE(STMT())
EMPTYDBRM | NOENTRYDBRM	-	NOENTRYDBRM
HOSTCOPY | NOHOSTCOPY	-	HOSTCOPY
INCONLY | NOINCONLY	-	NOINCONLY
LINEFILE | LINEONLY	-	LINEONLY
WARNDECP | NOWARNDECP	-	NOWARNDECP

CCSID0

CCSID0 オプションは、 WIDECHAR 以外のホスト変数に、 PL/I SQL プリプロセッサが CCSID 値を割り当てないことを指定する。

NOCCSID0 オプションを使うと、ホスト変数に PL/I SQL プリプロセッサで CCSID の値を割り当てることができる。



プログラムが BIT データではないデータ型で FOR BIT DATA 列を更新する場合、CCSID0. CCSID0 を選択すると、ホスト変数が に関連付けられていないことが通知される。 Db2 を選択すると、ホスト変数が CCSID に関連付けられていないことを通知します。 そうでない場合、BIT データでない CCSID に関連付けられたホスト変数が FOR BIT DATA 列に代入されると、エラーが発生する。 Db2 エラーが発生する。

WIDECHAR は常に 1200 の値が割り当てられている。 CCSID

プリコンパイラを使用していた古い PL/I プログラムとの互換性のために Db2 プリコンパイラーを使用した古いプログラムとの互換性を保つには、 CCSID0 を有効にしてください。

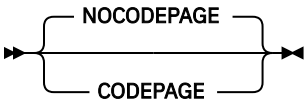
CCSID0 と は互いに排他的なオプションである。 NOCCSID0

デフォルトは CCSID0 です。

CODEPAGE

CODEPAGE オプションが有効な場合、文字型の SQL ホスト変数の CCSID には常にコンパイラの CODEPAGE オプションが使用されます。

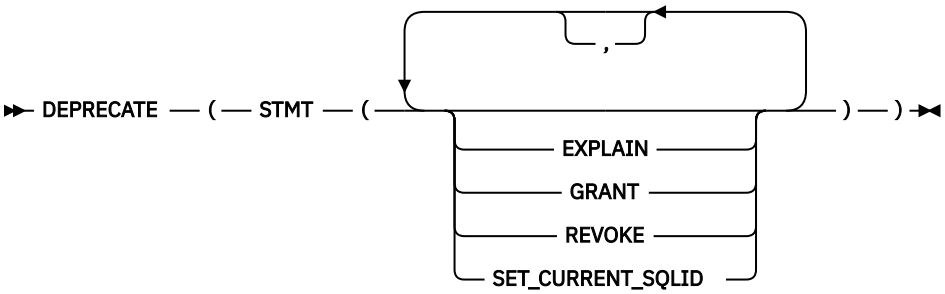
NOCODEPAGE オプションが有効な場合、 SQL プリプロセッサー・オプション NOCCSID0 も有効な場合に限り、コンパイラー CODEPAGE オプションが文字型の SQL ホスト変数の CCSID として使用される。



デフォルトは NOCODEPAGE です。

DEPRECATE

DEPRECATE オプションは、プリプロセッサが指定された文を非推奨とすることを示す。



STMT
プリプロセッサーによって非推奨のフラグが立てられるステートメントのリストを指定します。 このリストは空でもかまいません。

EXPLAIN
EXPLAIN SQL 。

GRANT
GRANT SQL 。

REVOKE
REVOKE SQL 。

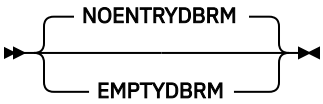
SET_CURRENT_SQLID
SET CURRENT SQLID SQL 。

デフォルトは DEPRECATE(STMT())です。

EMPTYDBRM

EMPTYDBRM オプションは、 SQL プリプロセッサーが、コードに EXEC SQL ステートメントが含まれていなくても、常に DBRM を作成するように指定する。 しかし、 SQL プリプロセッサーは、 INCONLY オプションが呼び出されても、 DBRM を作成しない。

NOENTRYDBRM オプションは、 SQL プリプロセッサが DBRM を作成しないことを指定する。

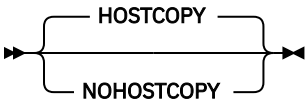


デフォルトは NOENTRYDBRM です。

HOSTCOPY

HOSTCOPY オプションは、SQL プリプロセッサーが LP(64) の下で実行されるときに、各 EXEC SQL 文の周囲でホスト変数をバー下のストレージにコピーしたり、バー下のストレージからコピーしたりするコードを生成するかどうかを決定する。

NOHOSTCOPY が指定された場合、 SQL プリプロセッサはこのコードを生成しない。 ただし、すべてのホスト変数がバー以下のストレージに存在するようにする（例えば、 ALLOC31 組み込み関数でベース・ポインタを取得する BASED 変数にする）のは、ユーザーの責任です。



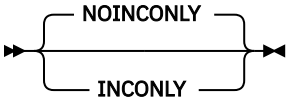
デフォルトは HOSTCOPY です。

LP(32) では HOSTCOPY オプションは無視される。

INCONLY

INCONLY オプションは、SQL 通信領域 (SQLCA) と SQL 記述子領域 (SQLDA) の `includes` を除き、SQL プリプロセッサが EXEC SQL INCLUDE ステートメントのみを処理することを指定する。 このオプションが有効な場合、 SQL プリプロセッサによって生成されるコードはない。

NOINCONLY オプションは、 EXEC SQL INCLUDE ステートメントだけでなく、 SQL プリプロセッサがすべてのステートメントを処理することを指定する。



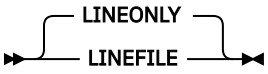
INCONLY オプションを指定すると、コンパイラーは SQL オプション・リストを作成しない。なぜなら、INCONLY の下では他のオプションはすべて無視されるからである。

INCONLY オプションと INCONLY オプションは互いに排他的である。

互換性のため、デフォルトは INCONLY。

LINEFILE

LINEFILE オプションは、 DBRM ステートメント番号フィールドが、 SQL ステートメントごとにファイル番号と行番号の両方をエンコードするように指定する。



LINEFILE オプションが有効な場合、 DBRM ステートメント番号フィールドには、各 SQL ステートメントのファイル番号と行番号の両方がエンコードされる。 具体的には、ステートメント番号の最初の 12 ビットにファイル番号が入り、残りの 20 桁に行番号が入ります。

LINEONLY オプションが有効な場合、DBRM ステートメント番号フィールドは SQL ステートメントごとに行番号をエンコードするだけである。

LINEONLY オプションが有効な場合、 DBRM ステートメント番号フィールドは、各 SQL ステートメントの行番号をエンコードするだけである。

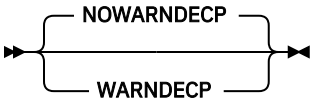
残りのコンパイル処理と同様に、コンパイラーは番号 0 をプライマリソースファイルに割り当て、番号 1 を最初のインクルードファイルに割り当てます。 従って、 SQL のすべてのステートメントがプライマリ・ソース・ファイルにある場合、 LINEFILE と LINEONLY の間に違いはない。

互換性のため、 LINEONLY がデフォルト。

WARNDECP

WARNDECP オプションが有効な場合、コンパイル時に Db2 で提供される DSNHDECP モジュールを使用すると、プリプロセッサは警告メッセージを表示します。

NOWARNDECP オプションが有効な場合、警告メッセージは表示されない。



デフォルトは NOWARNDECP です。

PL/I- SQL プロセッサー・オプションに関する特記事項

このトピックでは、 SQL プロセッサのオプション FLOAT, ONEPASS および STDSQL を指定する際に従わなければならないいくつかのルールについて説明します。

PL/I コンパイラに以下の SQL プロセッサ・オプションを指定すると、以下のルールが適用される：

FLOAT
FLOAT オプションと PL/I DEFAULT(HEXADEC | IEEE) オプションが異なる場合、エラーメッセージが表示される。

ONEPASS | TWOPASS
ONEPASS オプションの下では、 SQL 文で使用する前に、すべてのホスト変数を宣言する必要があります。

STDSQL
STDSQL(YES) というオプションの下では、 SQL BEGIN DECLARE SECTION と SQL END DECLARE SECTION の間ですべてのホスト変数を宣言しなければならない。

PL/I アプリケーションにおける SQL ステートメントのコーディング

で定義された言語を使用することで、PL/I アプリケーションで SQL ステートメントをコーディングすることができます。 *Db2 UDB for SQL* リファレンス *z/OS*. このセクションでは、 SQL コードに関する具体的な要件について説明します。

SQL 通信エリアの定義

SQL ステートメントを含む PL/I プログラムは、 SQLCODE 変数（ STDSQL(86) プリプロセッサーオプションが使用されている場合）または SQL 通信領域 (SQLCA) のいずれかを含める必要があります。

129 ページの『#unique_231/unique_231_Connect_42_sqlca』 に示されているように、SQLCA の一部は SQLCODE 変数と SQLSTATE 変数から構成されています。

- SQLCODE の値は、 SQL ステートメントが実行されるたびに、データベース・サービスによって設定される。 アプリケーションは SQLCODE 値をチェックして、最後の SQL ステートメントが成功したかどうかを判断することができる。
- SQLSTATE 変数は、 SQL 文の結果をコンパイラが解析するときに、SQLCODE 変数の代わりとして使うことができる。 SQLCODE 変数と同様に、 SQL 文が実行されるたびに、 SQLSTATE 変数がデータベース・サービスによって設定される。

SQLCA 、 EXEC SQL INCLUDE ：

```
  exec sql include sqlca;
```

SQLCA 構造は SQL 宣言セクション内で定義してはならない。 SQLCODE 、 SQLSTATE の宣言のスコープには、プログラム内のすべての SQL 文のスコープが含まれていなければならない。

The PL/I declaration of SQLCA

```
  Dcl
    1 Sqlca,
```

```
    2 sqlcaid      char(8),           /* Eyecatcher = 'SQLCA   '   */
    2 sqlcabc      fixed binary(31),  /* SQLCA size in bytes = 136 */
    2 sqlcode      fixed binary(31),  /* SQL return code           */
    2 sqlerrmc     char(70) var,      /* Error message tokens      */
    2 sqlerrp      char(8),           /* Diagnostic information    */
    2 sqlerrd(0:5) fixed binary(31),  /* Diagnostic information    */
    2 sqlwarn,                        /* Warning flags             */
      3 sqlwarn0   char(1),
      3 sqlwarn1   char(1),
      3 sqlwarn2   char(1),
      3 sqlwarn3   char(1),
      3 sqlwarn4   char(1),
      3 sqlwarn5   char(1),
      3 sqlwarn6   char(1),
      3 sqlwarn7   char(1),
    2 sqlext,
      3 sqlwarn8   char(1),
      3 sqlwarn9   char(1),
      3 sqlwarna   char(1),
      3 sqlstate   char(5);       /* State corresponding to SQLCODE */
```

SQL ディスクリプタ領域の定義

SQLCA とは異なり、 SQL 記述子領域 (SQLDA) はプログラム中に複数存在することができ、 SQLDA は任意の有効な名前を持つことができる。

以下の文は SQLDA を必要とする：

PREPARE *statement-name* INTO *descriptor-name* FROM *host-variable*
EXECUTE...USING DESCRIPTOR *descriptor-name*
FETCH...USING DESCRIPTOR *descriptor-name*
OPEN...USING DESCRIPTOR *descriptor-name*
DESCRIBE *statement-name* INTO *descriptor-name*

SQLDA を含めるには、 EXEC SQL INCLUDE ステートメントを使用する：

```
exec sql include sqlda;
```

SQLDA 、 SQL 宣言セクション内で定義してはならない。

SQL 記述子領域の PL/I 宣言

```
Dcl
  1 Sqlda based(Sqldaptr),
    2 sqldaid     char(8),          /* Eye catcher = 'SQLDA   ' */
    2 sqldabc     fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
    2 sqln        fixed binary(15), /* Number of SQLVAR elements*/
    2 sqld        fixed binary(15), /* # of used SQLVAR elements*/
    2 sqlvar(Sqlsize refer(sqln)),  /* Variable Description     */
      3 sqltype   fixed binary(15), /* Variable data type       */
      3 sqllen    fixed binary(15), /* Variable data length     */
      3 sqldata   pointer,          /* Pointer to variable data value*/
      3 sqlind    pointer,          /* Pointer to Null indicator*/
      3 sqlname   char(30) var ;    /* Variable Name            */

Dcl
  1 Sqlda2 based(Sqldaptr),
    2 sqldaid2     char(8),          /* Eye catcher = 'SQLDA   ' */
    2 sqldabc2     fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
    2 sqln2        fixed binary(15), /* Number of SQLVAR elements*/
    2 sqld2        fixed binary(15), /* # of used SQLVAR elements*/
    2 sqlvar2(Sqlsize refer(sqln2)), /* Variable Description     */
      3 sqlbiglen,
        4 sqllongl fixed binary(31),
        4 sqlrsvdl fixed binary(31),
      3 sqldatal   pointer,
      3 sqltname   char(30) var;

dcl Sqlsize    fixed binary(15);     /* number of sqlvars (sqln) */
dcl Sqldaptr   pointer;
dcl Sqltripled char(1) value('3');
dcl Sqldoubled char(1) value('2');
dcl Sqlsingled char(1) value(' ');
```

SQL ステートメントを埋め込む

プログラムの最初の文は、 PROCEDURE または PACKAGE 。 実行可能なステートメントが表示される場所であれば、 SQL ステートメントをプログラムに追加することができます。

また、 PACKAGE 、プロシージャの外側に以下の SQL ステートメントを追加することもできる：

- EXEC SQL BEGIN DECLARE SECTION
- EXEC SQL END DECLARE SECTION
- EXEC SQL DECLARE if no executable code needs to be generated for the statement
- EXEC SQL INCLUDE

各 SQL 文は、 EXEC （or EXECUTE） SQL で始まり、セミコロン（;）で終わらなければならない。

例えば、 UPDATE ：

```
           exec sql update DSN8A10.DEPT
           set   Mgrno  = :Mgr_Num
           where Deptno = :Int_Dept;
```

コメント

SQL ・ステートメントに加えて、 SQL ・ステートメントを埋め込む際にも、空白が許される場所であればコメントを含めることができる。

コメントが SQL 文の中にある場合、コメントを閉じるスラッシュ（/）は、リストでは大なり記号（>）として表示される。 次の例は、コンパイラがソース・リストで指定された SQL ステートメントをどのように表示するかを示している。

次の例 SQL ：

```
    exec sql insert into table /* some text */ values(:data);
```

コンパイラーは、このステートメントをソース・リストにおいて 次のように表示します。

```
    /*$*$*$
     exec sql insert into table /* some text *> values(:data)
     $*$*$*/
```

SQL スタイルのコメント（'--'）は、 ステートメントに埋め込まれた場合にサポートされます。 SQL

SQL

SQL ステートメントの行継続ルールは、他の PL/I ステートメントと同じです。

コードの組み込み

ソースコード中に以下の SQL ステートメントを記述することで、 SQL ステートメントまたは PL/I ホスト変数宣言ステートメントを含めることができます。 これを、ステートメントを組み込む場所に置いてください。

```
  exec sql include member;
```

マージン

コーディングする必要があります SQL 列 *m* から *n* までのステートメント。ここで、 *m* と *n* は MARGINS(*m*, *n*) コンパイラオプション。

名称

ホスト変数には、任意の有効な PL/I 変数名を使用できます。 ホスト変数名の長さは、 LIMITS(NAME(*n*)) コンパイラー・オプションで指定された値 *n* を超えてはならない。

ステートメント・ラベル

END DECLARE SECTION ステートメントと INCLUDE text-file-name ステートメントを除き、実行可能な SQL ステートメントは、 PL/I ステートメントと同様に、ラベル接頭辞を付けることができる。

WHENEVER ステートメント

SQL WHENEVER ステートメント内の GOTO 句のターゲットは、 PL/I ソースコード内のラベルでなければならず、 WHENEVER ステートメントの影響を受ける SQL ステートメントの範囲内になければならない。

ホスト変数の使用

SQL 文で使用されるすべてのホスト変数は明示的に宣言されなければならず、 SQL 文内のすべてのホスト変数の前にはコロン（:）を付けなければならない。

ホスト変数参照では添え字は使用できません。

以下のトピックでは、 ホスト変数の使用方法について詳しく説明します。

- 132 ページの『配列をホスト変数として使用』
- 132 ページの『ホスト変数の宣言』
- 133 ページの『スカラー・ホスト変数の宣言』
- 135 ページの『同等の SQL と PL/I データ型の決定』
- 138 ページの『SQL と PL/I データ型の互換性の判断』

配列をホスト変数として使用

次の 2 つの方法でのみ、配列をホスト変数として使用できます。

- ホスト構造の標識変数の配列として
- 次のいずれかのステートメントで使用される場合、ホスト変数の配列として
  - 複数行フェッチ用の FETCH ステートメント
  - 複数行挿入の INSERT ステートメント
  - 複数行の MERGE ステートメント

このような配列はすべて一次元で、 CONNECTED 属性を持ち、一定の境界を持たなければならない。

配列をホスト変数として使用するこれ以外の方法は、すべて無効です。

ホスト変数の宣言

ホスト変数の宣言は、通常の PL/I 変数宣言と同じ場所で行うことができます。

有効な PL/I 宣言のサブセットだけが、有効なホスト変数宣言として 認識されます。

SQL プリプロセッサは、以下の制限付きで DEFINE ALIAS, DEFINE ORDINAL、および DEFINE STRUCTURE ステートメントをサポートしています：

- DEFINE ALIAS 型で宣言された変数は、SQL 文でその基本型が許可されていれば、SQL 文で使用することができます。
- DEFINE ORDINAL 型で宣言された変数は、序数が NATIVE 、サイズが 2 バイトまたは 4 バイトであれば、SQL 文で使用することができる。
- DEFINE STRUCTURE 型の要素を含む参照はホスト変数として使用できるが、指示変数としては使用できない。

プリプロセッサは、 PL/I DEFAULT 文で指定されたデータ属性のデフォルトを使用しない。 変数の宣言が認識されない場合は、ステートメントがその変数を参照すると、次のようなメッセージが出されることがあります。

```
'The host variable token ID is not valid'
```

LIKE を使用して宣言された構造体、またはこの構造体の 1 つのエレメントを、ホスト変数として使用する場合は、LIKE オブジェクトの宣言が SQL プリプロセッサーによって認識されなければなりません。 例えば、インクルードされていない %INCLUDE ファイルに LIKE オブジェクトが含まれていてはなりません。

ホスト変数宣言内で、制限付きの式を使用すれば、配列の境界や、ストリングの長さを定義できます。ただし、この式には、以下のいずれかの形式が必要です。

- 式に適用された接頭演算子 (この式が整数に評価できる場合)
- 2 つの式に適用された加算演算子または減算演算子 (いずれの式も整数に評価できる場合)
- 2 つの式に適用された乗算演算子 (いずれの式も整数に評価できる場合)
- 名前付き定数に対する参照 (この参照が整数に評価できる場合)
- これらの組み込み関数の 1 つ、 INDICATORS, HBOUND, HBOUNDACROSS, LENGTH, と MAXLENGTH
- 整数の数値

名前付き定数を使用してホスト変数の境界や長さを定義できますが、名前付き定数自体をホスト変数として使用することはできません (ただし、以下の両方の条件が適用される場合を除く)。

- DB2 では、 EXEC SQL ステートメント内のその場所に、単純な無名の定数を置くことができる。
- 名前付き定数に以下のいずれかの属性がある。
  - CHARACTER この場合、名前付き定数の VALUE 属性には文字列を指定しなければならない。
  - FIXED この場合、名前付き定数の VALUE 属性は、10 進数か、上記と同じ制限で整数定数に還元できる式を指定しなければならない。

変数の名前とデータ属性だけがプリプロセッサーによって使用され、 位置合わせ、スコープ、およびストレージの属性は無視されます。

スカラー・ホスト変数の宣言

以下のデータ属性のいずれかを使用して、 スカラー・ホスト変数を宣言する必要があります。

CHARACTER, GRAPHIC, or WIDECHAR
CHARACTER, GRAPHIC、または WIDECHAR 属性で宣言されたホスト変数は、文字列ホスト変数と呼ばれます。 ストリング・ホスト変数には以下の制限が適用されます。

- NONVARYING 、または VARYING 属性のいずれかを持たなければならない。
- VARYING 、 NATIVE 属性がなければならない。

FIXED BINARY, FIXED DECIMAL, or FLOAT
FIXED および BINARY、FIXED および DECIMAL、または FLOAT 属性を使用して宣言されたホスト変数は、数値ホスト変数と呼ばれます。 数値ホスト変数には以下の制限が適用されます。

- REAL 属性が必要である。
- FIXED 属性および BINARY 属性がある場合は、SIGNED 属性、NATIVE 属性、ゼロ・スケール係数、および 7 を超える精度が必要である。
- FIXED 属性および DECIMAL 属性がある場合は、精度よりも小さく負数ではないスケール係数が必要である。
- FLOAT 属性および DECIMAL 属性がある場合は、FLOAT(DFP) オプションが有効になっていない限り、17 未満の精度が必要である。
- FLOAT 属性および BINARY 属性がある場合は、54 未満の精度が必要である。

ORDINAL
ORDINAL 属性で宣言されたホスト変数は、数値ホスト変数とも呼ばれます。 数値ホスト変数には以下の制限が適用されます。

- SIGNED の場合、精度は 7 より大きくなければなりません。UNSIGNED の場合、精度は 8 より大きくなければなりません。
- NATIVE 属性が必要である。

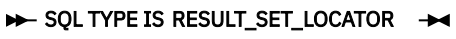
SQL TYPE

SQL TYPE 属性を使用して宣言されるホスト変数は、 SQL TYPE ホスト変数と呼ばれます。 この属性の仕様は、 以下の構文図のいずれかに準拠している必要があります。

BINARY

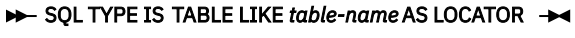
VARBINARY

Result set locator

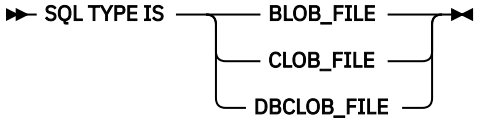


ROWID

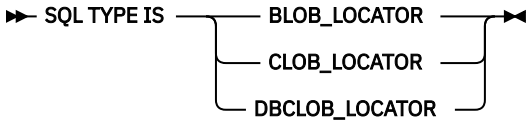
Table locator



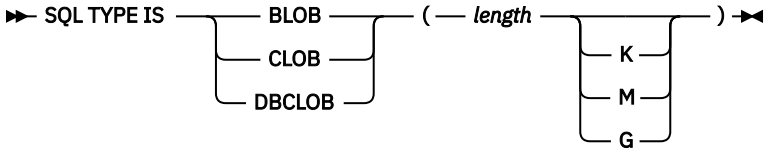
LOB file reference



LOB locator



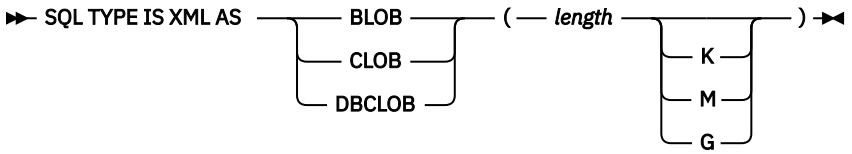
LOB variable



BLOB
You can also use **BINARY LARGE OBJECT** as an alternative to **BLOB**.

CLOB
You can also use either **CHARACTER LARGE OBJECT** or **CHAR LARGE OBJECT** as an alternative to **CLOB**.

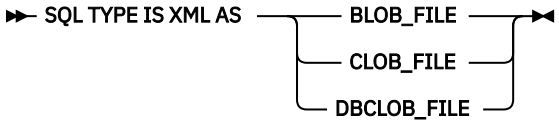
XML LOB variable



BLOB
You can also use **BINARY LARGE OBJECT** as an alternative to **BLOB**.

CLOB
You can also use either **CHARACTER LARGE OBJECT** or **CHAR LARGE OBJECT** as an alternative to **CLOB**.

XML file reference



以下の定数宣言は、 SQL プリプロセッサによって生成される。 これらの宣言を使用すると、ファイル参照ホスト変数を使用する際に ファイル・オプション変数を設定できます。

```
DCL SQL_FILE_READ      FIXED BIN(31) VALUE(2);
DCL SQL_FILE_CREATE    FIXED BIN(31) VALUE(8);
DCL SQL_FILE_OVERWRITE FIXED BIN(31) VALUE(16);
DCL SQL_FILE_APPEND    FIXED BIN(31) VALUE(32);
```

同等の *SQL* と *PL/I* データ型の決定

ホスト変数のベース SQLTYPE と SQLLEN は 135 ページの表 9 と 136 ページの表 10 に従って決定される。 ホスト変数が指標変数と一緒に現れる場合、 SQLTYPE はベース SQLTYPE に 1 を足したものになる。

所定の SQL データ・タイプと同等の PL/I データ・タイプを判別するには、 137 ページの表 11 および 137 ページの表 12 を使用できます。

表 *9. PL/I* 宣言から生成される *SQL* データ・タイプ. PL/I 宣言から生成される SQL データ・タイプ

PL/I data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
BIN FIXED(p), 7 < p <= 15	500	2	SMALLINT
BIN FIXED(p), 15 < p <= 31	496	4	INTEGER
BIN FIXED(p), 31 < p <= 63	492	8	BIGINT
DEC FIXED(p,s), 0<=p<=15 and 0<=s<=p	484	p (byte 1) s (byte 2)	DECIMAL(p,s)
BIN FLOAT(p), 1 ≤ p ≤ 21	480	4	REAL or FLOAT(n) 1<=n<=21
BIN FLOAT(p), 22 ≤ p ≤ 53	480	8	DOUBLE PRECISION or FLOAT(n), 22<=n<=53
Under FLOAT(NODFP):			
DEC FLOAT(p), 1 ≤ p ≤ 6	480	4	FLOAT (single precision)
DEC FLOAT(p), 7 ≤ p ≤ 16	480	8	FLOAT (double precision)
Under FLOAT(DFP):			
DEC FLOAT(p), 1 ≤ p ≤ 7	996	4	DECFLOAT (single precision)
DEC FLOAT(p), 8 ≤ p ≤ 16	996	8	DECFLOAT (double precision)
DEC FLOAT(p), 17 ≤ p ≤ 34	996	16	DECFLOAT (extended precision)
CHAR(n)	452	n	CHAR(n)
CHAR(n) VARYING	448	n	VARCHAR(n)

表 *9. PL/I* 宣言から生成される *SQL* データ・タイプ. PL/I 宣言から生成される SQL データ・タイプ *(*続き*)*

PL/I data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
GRAPHIC(n), 1 ≤ n ≤ 127	468	n	GRAPHIC(n)
GRAPHIC(n) VARYING	464	n	VARGRAPHIC(n)
SIGNED ORDINAL PREC(p), 7 < p <= 15	500	2	SMALLINT
SIGNED ORDINAL PREC(p), 15 < p <= 31	496	4	INTEGER
UNSIGNED ORDINAL PREC(p), 8 < p <= 16	500	2	SMALLINT
UNSIGNED ORDINAL PREC(p), 16 < p <= 32	496	4	INTEGER

表 *10. SQL SQL TYPE* 宣言から生成されるデータ型

PL/I data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
SQL TYPE IS BLOB(*n*) 1<*n*<2147483647	404	*n*	BLOB(*n*)
SQL TYPE IS CLOB(*n*) 1<*n*<2147483647	408	*n*	CLOB(*n*)
SQL TYPE IS DBCLOB(*n*) 1<*n*<1073741823 (2)	412	*n*	DBCLOB(*n*) (2)
SQL TYPE IS ROWID	904	40	ROWID
SQL TYPE IS VARBINARY(*n*) 1<*n*<32704	908	*n*	VARBINARY(*n*)
SQL TYPE IS BINARY(*n*) 1<*n*<255	912	*n*	BINARY(*n*)
SQL TYPE IS BLOB_FILE	916	267	BLOB File Reference (1)
SQL TYPE IS CLOB_FILE	920	267	CLOB File Reference (1)
SQL TYPE IS DBCLOB_FILE	924	267	DBCLOB File Reference (1)
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB Locator (1)
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB Locator (1)
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB Locator (1)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result Set Locator
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table Locator (1)

注 :

1. このデータ・タイプを列タイプとして使用しないでください。
2. *n* は 2 バイト文字の数です。

表 *11. SQL* データ・タイプと *PL/I* 宣言の対応

SQL データ・タイプ	**PL/I 同等**	**備考**
SMALLINT	BIN FIXED(15)	
整数	BIN FIXED(31)	
BIGINT	BIN FIXED(63)	
DECIMAL(p,s)	DEC FIXED(p) または DEC FIXED(p,s)	p = precision および s = scale; 1 ≤ p ≤ 31 および 0 ≤ s ≤ p
FLOAT(NODFP) を指定した場合:		
REAL または FLOAT(n)	BIN FLOAT(p) または DEC FLOAT(m)	1 ≤ p ≤ 21 および 1 ≤ m ≤ 6
DOUBLE PRECISION、DOUBLE、または FLOAT(n)	BIN FLOAT(p) または DEC FLOAT(m)	22 ≤ p ≤ 53 および 7 ≤ m ≤ 16
FLOAT(DFP) を指定した場合:		
DECFLOAT	DEC FLOAT(m)	短精度 10 進浮動小数 1 ≤ m ≤ 7
DECFLOAT	DEC FLOAT(m)	Long Decimal Float 8 ≤ m ≤ 16
DECFLOAT	DEC FLOAT(m)	拡張 10 進浮動小数 17 ≤ m ≤ 34
CHAR(n)	CHAR(n)	1 ≤ n ≤ 32767
VARCHAR(n)	CHAR(n) VAR	
GRAPHIC(n)	GRAPHIC(n)	*n* は、2 バイト文字の数を示す (バイト数ではない) 正整数。1 ≤ n ≤ 16383
VARGRAPHIC(n)	GRAPHIC(n) VAR	*n* は、2 バイト文字の数を示す (バイト数ではない) 正整数。1 ≤ n ≤ 16383
日付	CHAR(n)	*n* の最小値は 10。
時刻	CHAR(n)	*n* の最小値は 8。
TIMESTAMP	CHAR(n)	*n* の最小値は 26。

表 *12. SQL* データ・タイプと *SQL TYPE* 宣言の対応

SQL データ・タイプ	**PL/I と同等**	**備考**
結果セット・ロケーター	SQL TYPE IS RESULT_SET_LOCATOR	このデータ・タイプは、結果セットの受け取りにだけ使用します。 このデータ・タイプを列タイプとして使用しないでください。
テーブル・ロケーター	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	このデータ・タイプは、ユーザー定義の関数または ストアード・プロシージャー内だけで、変位テーブルの行を受け取るために 使用します。 このデータ・タイプを列タイプとして使用しないでください。
BLOB ロケーター	SQL TYPE IS BLOB_LOCATOR	このデータ・タイプは、BLOB 列のデータを操作するためにだけ使用します。 このデータ・タイプを列タイプとして使用しないでください。

表 *12. SQL* データ・タイプと *SQL TYPE* 宣言の対応 *(*続き*)*

SQL データ・タイプ	PL/I と同等	備考
CLOB ロケーター	SQL TYPE IS CLOB_LOCATOR	このデータ・タイプは、CLOB 列のデータを操作するためにだけ使用します。 このデータ・タイプを列タイプとして使用しないでください。
DBCLOB ロケーター	SQL TYPE IS DBCLOB_LOCATOR	このデータ・タイプは、DBCLOB 列のデータを操作するためにだけ使用します。 このデータ・タイプを列タイプとして使用しないでください。
BLOB ファイル参照	SQL TYPE IS BLOB_FILE	このデータ・タイプは、BLOB ファイルの参照としてのみ使用します。 このデータ・タイプを列タイプとして使用しないでください。
CLOB ファイル参照	SQL TYPE IS CLOB_FILE	このデータ・タイプは、CLOB ファイルの参照としてのみ使用します。 このデータ・タイプを列タイプとして使用しないでください。
DBCLOB ファイル参照	SQL TYPE IS DBCLOB_FILE	このデータ・タイプは、DBCLOB ファイルの参照としてのみ使用します。 このデータ・タイプを列タイプとして使用しないでください。
BLOB(*n*)	SQL TYPE IS BLOB(*n*)	*1<n<2147483647*
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	*1<n<2147483647*
DBCLOB(*n*)	SQL TYPE IS DBCLOB(*n*)	*n* は 2 バイト文字の数です。 *1<n<1073741823*
ROWID	SQL TYPE IS ROWID	
XML AS	SQL TYPE IS XML AS ...	XML バージョンの BLOB、CLOB、DBCLOB、BLOB_FILE、CLOB_FILE、または DBCLOB_FILE を記述するのに使用します。

SQL と *PL/I* データ型の互換性の判断

SQL ステートメント内の PL/I ホスト変数は、その変数が使用される列と互換性があるタイプでなければなりません。

- 数値データ・タイプは、相互に互換性があります。 SMALLINT, INTEGER, DECIMAL、または FLOAT 列は、 BIN FIXED(15), BIN FIXED(31), DECIMAL*(p,s*)、BIN FLOAT *(n*) (*n* は 22～53 の範囲)、または DEC FLOAT *(m*) (*m* は 7～16 の範囲) の PL/I ホスト変数と互換性がある。
- 文字データ・タイプは、相互に互換性があります。 CHAR または VARCHAR 列は、固定長または可変長の PL/I 文字ホスト変数と互換性がある。
- Datetime データ・タイプは、文字ホスト変数と互換性があります。 DATE, TIME、または TIMESTAMP 列は、固定長または可変長の PL/I 文字ホスト変数と互換性がある。

  必要に応じて、データベース・マネージャーは自動的に固定長文字ストリングを 可変長ストリングに変換したり、可変長ストリングを固定長文字ストリングに変換したりします。

ホスト構造体の使用

構造体または共用体でないメンバーを持つ構造体の名前を、PL/I ホスト構造体の名前にすることができます。

次の例で、B はスカラー C1 と C2 からなるホスト構造体の名前です。

```
dcl 1 A,
      2 B,
```

```
          3 C1 char(...),
          3 C2 char(...);
```

ホスト構造体は 2 レベルに制限されます。 ホスト構造体は、ホスト変数の名前付き集合と考えることができます。

ホスト構造体のリーフ要素にはそれぞれ、以下の有効なホスト・データ属性のいずれかが必要です (132 ページの『ホスト変数の宣言』での説明を参照)。

- CHARACTER, GRAPHIC, or WIDECHAR
- FIXED BINARY, FIXED DECIMAL, or FLOAT
- SQL TYPE

標識変数の使用

標識変数は、2 バイトの整数 (BIN FIXED(15)) であるか、2 バイトの整数の配列であるか、 または 2 バイトの整数 (もしくはその配列) のみを含む構造体です。

検索時には、関連したホスト変数にヌル値が割り当てられている かどうかを示すために、標識変数が使用されます。 列への割り当て時には、 ヌル値を割り当てる必要があるかどうかを示すために、負の標識変数が使用されます。

構造体は、関連ホスト変数も構造体である場合に限り、標識変数として使用できます。

標識変数はホスト変数と同じ方法で宣言され、両変数の宣言は プログラマーの裁量でどのように組み合わせることもできます。

SQL プリプロセッサーでは、標識配列の下限が 1 である必要はありません。

指標変数は、属性 REAL NATIVE SIGNED FIXED BIN(15) を持たなければならない。

以下の例では、 指示変数を含む SQL 文の 変数を宣言する方法を示しています。

SQL 指示変数を含むステートメント

```
exec sql fetch Cls_Cursor into :Cls_Cd,
                               :Day :Day_Ind,
                               :Bgn :Bgn_Ind,
                               :End :End_Ind;
```

変数は以下のように宣言できます。

```
exec sql begin declare section;
dcl Cls_Cd    char(7);
dcl Day       bin fixed(15);
dcl Bgn       char(8);
dcl End       char(8);
dcl (Day_Ind, Bgn_Ind, End_Ind)   bin fixed(15);
exec sql end declare section;
```

ホスト構造体の例

この例では、ホスト構造体とインジケータ配列の宣言に続いて、ホスト構造体にデータを取得するために使用できる SQL ステートメントを示しています。

```
dcl 1 games,
      5 sunday,
        10 opponents char(30),
        10 gtime     char(10),
        10 tv        char(6),
        10 comments  char(120) var;
dcl indicator(4) fixed bin (15);


exec sql
     fetch cursor_a
     into :games.sunday:indicator;
```

LOB データを操作する

LOBS, CLOBSBLOBS は 2,147,483,647 バイト（2 ギガバイト）にもなる。 ダブルバイト CLOBS 、1,073,741,823 文字（1 ギガバイト）の長さが可能。 Db2 テーブルのラージ・オブジェクト（LOB ）データを使用するには、LOB ロケータや LOB ファイル参照などのテクニックを使用して、データがまだデータベースにある間にデータを操作します。

LOB 、すべてのデータを保持するためにホスト変数を宣言することは、非効率的または非現実的である。 Db2 大量のデータを移動する必要があるからです。 したがって、LOB データを操作するには、以下のテクニックを使用することをお勧めします：

- LOB ロケータ

  LOB ロケータを使えば、LOB データをホスト変数に移さずに、LOB データを操作できる。 LOB ロケータを使うことで、プログラムに必要なメモリはずっと少なくてすむ。

- LOB ファイル参照

  LOB ファイル参照変数を使用して、 LOB 列とシステム外の外部ファイルとの間でデータをインポートまたはエクスポートできます。 Db2 ファイル参照変数を使うことができます。

大きなデータ断片の移動を最小限にするための手法について詳しくは、「*Db2 for* アプリケーション・プログラミングと *SQL* ガイド *z/OS*」、または IBM Redbooks® 資料「*Db2 for z/OS* の *LOB*：より強く、より速く」を参照してください。

PL/I および Db2 環境で CLOB を扱う方法の例については、pliclob サンプル・プログラムを参照してください。

LOB ロケータ

LOB データとそれに関連するすべての基本的なアクティビティの実体化を避けるために、 LOB ロケータを使用することができます。

LOB ロケーターを使用する利点を以下に挙げる：

- LOB ロケータで LOBs を操作する際のストレージの節約
- データベースから取得することなくデータを操作できる
- を保持するために大容量のストレージを使用することを避ける。 LOB
- 大きなデータ断片を移動するための時間やリソースが不要になるためパフォーマンスが向上する

特に、LOB ロケーターは以下の状況で役に立ちます。

- の一部しか必要ない場合 LOB
- メモリが足りない場合 LOB
- パフォーマンスが重要な場合
- クライアント環境またはサーバー環境において、ネットワークを使用してシステム間でデータを移動しない場合

以下のコード例は pliclob サンプル・プログラムからのものです。 サンプルプログラムでは、 LOB ロケータを使って、 `dsn8a10.emp_photo_resume` テーブルから `resume` CLOB のセクションを特定し、操作している。 Db2V10 テーブルから (各行の前にある番号はプログラムの一部ではなく、プログラムの後にある説明で使用します。)

```
1.   dcl hv_loc_resume  sql type is clob_locator;
2.    exec sql
3.       select resume into :hv_loc_resume
4.       from dsn8a10.emp_photo_resume
5.       where empno = :hv_empno;
6.
7.    exec sql
8.       set :start_resume = (posstr(:hv_loc_resume, 'Resume:'));
```

2 行目から 5 行目では、LOB ロケーター `hv_loc_resume` を `emp_photo_resume` 表内の 従業員番号 `hv_empno` の履歴書 (resume) の位置に設定しています。 7 行目から 8 行目では、*start_resume* ホスト変数

を resume の 'Resume:' セクションの先頭に 設定しています。 これで、履歴書 (resume) をデータベース内に置いたまま、resume データの操作を開始できます。

LOB ファイル参照変数

LOB ファイル参照変数を使用して、 LOB 列とシステム外の外部ファイルとの間でデータをインポートまたはエクスポートできます。 Db2 ファイル参照変数を使うことができます。

LOB ファイル参照変数を使用する利点を以下に挙げる：

- LOB のデータをホスト変数で移動するよりも処理時間が短い。 データの移動が Db2 処理時間やネットワーク転送時間とオーバーラップすることはありません。
- アプリケーション・ストレージの使用量が少ない。 LOB データは Db2 から直接ファイルに移動され、アプリケーションのメモリーでは実体化されない。

以下のコード例は pliclob サンプル・プログラムからのものです。 サンプル・プログラムは、 LOB のファイル参照を使用して、外部ファイルに resume の切り詰めた新しいバージョンを作成する。 (各行の前にある番号はプログラムの一部ではなく、プログラムの後にある説明で使用します。)

```
1.   dcl hv_clob_file   sql type is clob_file;
2.   name_string = '/SYSTEM/tmp/pliclob2.txt';
3.   hv_clob_file.sql_lob_file_name_len = length(name_string);
4.   hv_clob_file.sql_lob_file_name     = name_string;
5.   hv_clob_file.sql_lob_file_options  = ior(sql_file_overwrite);
6.
7.   exec sql
8.    values ( substr(:hv_loc_resume,:start_resume,
9.                    :start_pers_info-:start_resume)
10.          || substr(:hv_loc_resume,:start_work_hist,
11.                    :start_interests-:start_work_hist)
12.          )
13.    into :hv_clob_file;
```

ホスト変数 *hv_clob_file* は、 LOB ファイル参照として宣言されている。 2～4 行目では、 LOB ファイル・リファレンスのファイル名フィールドに完全修飾ファイル名が設定され、ファイル名の長さにその長さが設定される。 既存のファイルがすべて上書きされるように、overwrite オプションを設定しています (5 行目)。 このようなオプションや他のファイル・オプションについて詳しくは、「*Db2 for* アプリケーション・プログラミングと *SQL* ガイド *z/OS*」を参照してください。

次に、 SQL VALUES ステートメントを使って、 resume の resume 名前と職歴のセクションを、8 行目から 13 行目のように、 hv_clob_file LOB ファイル参照に直接連結する。

例： pliclob サンプルプログラム

このサンプル・プログラム（ PL/I ）は、 PL/I とその環境で CLOBs を操作する方法を示している。 Db2 環境で操作する方法を示しています。

このプログラムを正しく実行するには、Db2 に付属のサンプル・データベースがインストールされていなければなりません。 このサンプルでは、Db2 V10 および表 dsn8a10.emp_photo_resume が前提となります。 別のバージョンの Db2 を使用している場合は、表の参照を変更する必要があります。

ノート：

- LOB ロケータと LOB ファイル参照変数を使用する場合、 resume CLOB はまだデータベース内にあり、メモリやストレージにはありません。
- データベース内の resume CLOB のフォーマットは変更されず、 resume の再フォーマットは、書き出された 2 番目のファイルでのみ行われる。

pliclob サンプルプログラム

```
pliclob: procedure options(main);
 display('begin pliclob');
 exec sql include sqlca;

 dcl hv_empno       char(06);
 dcl name_string    char(256) var;
 dcl hv_resume      sql type is clob(50k);
```

```
dcl hv_clob_file   sql type is clob_file;
dcl hv_loc_resume  sql type is clob_locator;

dcl start_resume     fixed bin(31);
dcl start_pers_info  fixed bin(31);
dcl start_dept_info  fixed bin(31);
dcl start_education  fixed bin(31);
dcl start_work_hist  fixed bin(31);
dcl start_interests  fixed bin(31);

/* Extract resume CLOB for employee '000130' into a file in z/OS  */
/* UNIX file system. The contents of this file shows the initial  */
/* format of the resume CLOB in the data base.                    */
/* Note: this program must have 'write' access to the directory   */
/*       designated in the 'name_string' variable.                */
name_string = '/SYSTEM/tmp/pliclob1.txt';
hv_clob_file.sql_lob_file_name_len = length(name_string);
hv_clob_file.sql_lob_file_name     = name_string;
hv_clob_file.sql_lob_file_options  = ior(sql_file_overwrite);

hv_empno = '000130';
exec sql
   select resume into :hv_clob_file
   from dsn8a10.emp_photo_resume
   where empno = :hv_empno;
display('file1   sqlca.sqlcode = ' || sqlca.sqlcode );

/* Next, a CLOB locator is used to locate the resume CLOB for     */
/* employee number '000130' in the data base.  Then a series of   */
/* Db2 SET statements using the posstr Db2 function finds the     */
/* beginning position of each section within the resume.          */
exec sql
   select resume into :hv_loc_resume
   from dsn8a10.emp_photo_resume
   where empno = :hv_empno;
display('select resume sqlcode = '|| sqlca.sqlcode);

exec sql set :start_resume =
         (posstr(:hv_loc_resume, 'Resume:'));
display('first set sqlcode     = '|| sqlca.sqlcode);

exec sql set :start_pers_info =
         (posstr(:hv_loc_resume, 'Personal Information'));
display('second set sqlcode    = '|| sqlca.sqlcode);
```

```
exec sql set :start_dept_info =
         (posstr(:hv_loc_resume, 'Department Information'));
display('third set sqlcode     = '|| sqlca.sqlcode);

exec sql set :start_education =
         (posstr(:hv_loc_resume, 'Education'));
display('fourth set sqlcode    = '|| sqlca.sqlcode);

exec sql set :start_work_hist =
         (posstr(:hv_loc_resume, 'Work History'));
display('fifth set sqlcode     = '|| sqlca.sqlcode);

exec sql set :start_interests =
         (posstr(:hv_loc_resume, 'Interests'));
display('sixth set sqlcode     = '|| sqlca.sqlcode);

/* Finally, by using the CLOB locator and the start references    */
/* of each section in the resume, along with the Db2 substr and   */
/* concatenate (||) functions, the resume CLOB is written out to  */
/* a second file in a slightly different format:                  */
/*   1. the Personal Information section is omitted due to        */
/*      privacy concerns.                                         */
/*   2. the sections within the resume are written out in this    */
/*      order: Resume, Work History, Education then Department    */
/*      Information.                                              */
/*                                                                */
/* After the second file is written out, the changes to the       */
/* resume CLOB can be verified by comparing the contents of the   */
/* two files pliclob1.txt and pliclob2.txt.                       */
/*                                                                */
/* Note: this program must have 'write' access to the directory   */
/*       designated in the 'name_string' variable.                */
name_string = '/SYSTEM/tmp/pliclob2.txt';
hv_clob_file.sql_lob_file_name_len = length(name_string);
hv_clob_file.sql_lob_file_name     = name_string;
hv_clob_file.sql_lob_file_options  = ior(sql_file_overwrite);
```

```
  exec sql
   values ( substr(:hv_loc_resume,:start_resume,
                   :start_pers_info-:start_resume)
          || substr(:hv_loc_resume,:start_work_hist,
                   :start_interests-:start_work_hist)
          || substr(:hv_loc_resume,:start_education,
                   :start_work_hist-:start_education)
          || substr(:hv_loc_resume,:start_dept_info,
                   :start_education-:start_dept_info)
          )
    into :hv_clob_file;

   display('file2   sqlca.sqlcode = ' || sqlca.sqlcode );
   display('End   pliclob');

 end;
```

pliclob sample program (continued)

SQL プリプロセッサ・メッセージの抑制

IBM-supplied compiler user exit (IBMUEXIT) 、メッセージを表示しないようにしたり、メッセージの重大度を変更したりすることができる。

ユーザー出口の変更によるプリプロセッサー・メッセージの抑止例については、 453 ページの『SQL メッセージの抑制例』を参照してください。

CICS プリプロセッサー

EXEC 。 CICSPL/I ステートメントを使用することができます。 CICS.

PP(CICS)オプションを指定しない場合、 EXEC CICS ステートメントが解析され、その中の変数参照が検証されます。 それらが正しければ、 NOCOMPILE オプションが有効である限り、メッセージは発行されない。 トランスレータを起動せず CICS トランスレータを起動せず、 COMPILE オプションが有効な場合、コンパイラは S レベルのメッセージを発行します。

PP オプションの CICS サブオプションを指定すると、コンパイラーは、CICS プリプロセッサーを呼び出します。 互換性を保つため、コンパイラーは以下のオプションが有効な場合にも CICS 互換性のため、コンパイラーは以下のオプションが有効な場合にもプリプロセッサを起動します： CICSXOPT または XOPTS。ただし、これらのオプションはいずれも PP(CICS) オプションと一緒に指定してはなりません。

プログラミングとコンパイルに関する考慮事項

で実行するプログラムを開発する場合 CICS で実行するプログラムを開発する場合、 EXEC CICS コマンドを翻訳する必要がある：

- によって提供されるコマンド言語トランスレータによって CICSPL/I によって提供されるコマンド言語トランスレータによって行われる
- PL/ICICSPL/I のコンパイルの一部としてプリプロセッサによって実行される（これには CICS TS 2.2 またはそれ以降)

プリプロセッサを使用するには CICS プリプロセッサを使用するには、PP(CICS) と DFT(EBCDIC) コンパイル時オプションも指定しなければならない。 に渡されるデータはすべて CICS に渡されるデータはすべて NATIVE 形式でなければならない。

PP()のサブオプションに CICS)オプションのサブ・オプションの 1 つとして指定しない限り、コンパイラは のフラグを立てます。 CICS)オプションのサブ・オプションとして指定しない限り、コンパイラはソース中の EXEC CICS ステートメントにフラグを立てます。 同様に、PP()オプションのサブオプショ ンに CPSM または DLI を指定しなかった場合、 EXEC CPSM または EXEC DLI ステートメントにフラグが付けられる。 CICS)オプションのサブオプションとして CPSM または DLI を指定しなければ、フラグを立てる。

あなたのプログラムが CICS プログラムが MAIN プロシージャの場合は、 SYSTEM(CICS)または SYSTEM(MVS)オプションを付けてコンパイルしなければならない。 SYSTEM(MVS) を指定してコンパイルする場合は、ランタイム APAR PQ91318 に対応する PTF を適用する必要があります。 NOEXECOPS このオ

プションでは、MAIN プロシージャに渡されるパラメータはすべて POINTER でなければならない。
SYSTEM コンパイル時オプションについては、90 ページの『SYSTEM』 を参照。

プログラムを再入可能にしたい場合 CICS プログラムをリエントラントにし、 FILEs または CONTROLLED 変数を使用する場合は、 NOWRITABLE 変数も使用してコンパイルする必要があります。

あなたの CICS プログラムに EXEC ステートメントを含むファイルやマクロを使用している場合は、コードを翻訳する前に プリプロセッサーを実行する必要があります。 CICS ステートメントを含むファイルやマクロを使用している場合は、コードが翻訳される前に MACRO プリプロセッサを実行する必要があります (上記のいずれかの方法で)。 CICS プリプロセッサーを使用する場合、次の例に示すよう な PP オプション 1 つを使用してこのプリプロセッサーを指定できます。

```
    pp (macro(…)  cics(…)  )
```

CICS プリプロセッサーは、CICS 変数および API の宣言のセットをすべてのネストなしプロシージャーに追加します。 その結果、END ステートメントと一致するステートメントを必要とするすべてのステートメントを追跡し、これらのステートメントのいくつかが欠けていたり、間違っていたりすると、プリプロセッサは惑わされて、これらの宣言を挿入しないかもしれません。 また、このプリプロセッサーは、このようなステートメントのネストの深さが 150 を超える場合は、重大メッセージで終了します。

最後に CICS プリプロセッサを使うには CICSSDFHLOAD データセットが PL/I コンパイラの STEPLIB DD の一部になっていなければならない。

CICS プリプロセッサー・オプション

CICS 変換プログラムは、数多くのオプションをサポートしています。

これらのオプションについては、「*CICS Application Programming Guide*」を参照してください。

これらのオプションは引用符 (左右の引用符が同じであれば単一引用符でも二重引用符でもかまわない) で囲む必要があることに注意してください。 たとえば CICS 例えば、 EDF オプションを指定してプリプロセッサを起動するには、PP(CICS ('EDF')) オプションを指定する必要があります。

PL/I アプリケーション内での CICS ステートメントのコーディング

アプリケーションで CICSPL/I ステートメントを記述することができます。 *CICS on Open Systems Application Programming Guide* で定義されている言語を使用します。 CICS コード固有の要件については、以下のセクションで説明します。

CICS ステートメントの組み込み

統合プリプロセッサではなく CICS 統合プリプロセッサではなくトランスレータを使用する場合、 PL/I プログラムの最初の文は PROCEDURE 文でなければなりません。 実行可能ステートメントを置くことができる任意の場所で、プログラムに CICS ステートメントを追加できます。 各 CICS 文は、 EXEC （または EXECUTE）で始まり、セミコロン（;）で終わらなければならない。 CICS で始まり、セミコロン（;）で終わっていなければならない。

例えば、 GETMAIN ：

```
   EXEC CICS GETMAIN SET(BLK_PTR) LENGTH(STG(BLK));
```

コメント

CICS ステートメントのほかに、ブランクを入力できる場所では 組み込み CICS ステートメントに PL/I コメントを組み込むことができます。

CICS ステートメントの継続

CICS ステートメントの行継続規則は、他の PL/I ステートメントと同じです。

コードの組み込み

含まれるコードに EXEC CICS ステートメントが含まれている場合、またはプログラムが EXEC ステートメントを生成する PL/I マクロを使用している場合は、次のいずれかのオプションを使用する必要があります。

CICS ステートメントを生成するマクロを使用している場合は、次のいずれかのオプションを使用する必要があります：

- MACRO コンパイル時オプション
- PP オプションの MACRO オプション（PP オプションの CICS オプションの前）

マージン
CICS ステートメントは、 MARGINS コンパイル時オプションで指定されたカラム内でコード化されなければならない。

ステートメント・ラベル
EXECCICS ステートメントは、 PL/I ステートメントと同様に、ラベルの接頭辞を持つことができます。

PL/I を使用した CICS トランザクションの作成

PL/I を CICS 機能と組み合わせて使用して、CICS サブシステム用のアプリケーション・プログラム (トランザクション) を作成することができます。 これを行う場合、PL/I プログラムには、通常はオペレーティング・システムから直接提供される機能が CICS から提供されます。 これらの機能には、ほとんどのデータ管理機能や、 ジョブとタスクの管理機能すべてが含まれます。

次の PL/I CICS プログラムの制限を順守する必要があります。

- マクロ・レベル CICS はサポートされません。
- PL/I の入力および出力は、以下の場合を除いて使用できません。
  - PUT FILE(SYSPRINT)
  - DISPLAY
  - CALL PLIDUMP
- PLISRTx 組み込みサブルーチンは使用できません。
- PL/I 以外の言語で書かれたルーチンに EXEC CICS ステートメントが含まれている場合、そのルーチンを PL/I CICS プログラムから呼び出すことはできません。 EXEC CICS ステートメントを含む非 PL/I プログラムとやり取りする場合は、EXEC CICS LINK または EXEC CICS XCTL を使用する必要があります。

PUT FILE(SYSPRINT) 。 CICS で許可されているが、パフォーマンスを低下させる可能性があるため、通常、本番用プログラムでは使用すべきではない。

なぜなら CICS EIB アドレスは CICS トランスレータまたは PL/I CICS OPTIONS(FETCHABLE)ルーチンの EIB に対するアドレス指定は、以下のいずれかの方法で行う必要があります：

- 次のコマンドを使用します。

```
EXEC CICS ADDRESS EIB(DFHEIPTR)
```

- 外部プロシージャを呼び出す CALL 文の引数として、 EIB アドレスを渡す。

エラー処理

Language Environment では、一部の EXEC CICS コマンドは、いずれの PL/I ON ユニットでも、PL/I ON ユニットから呼び出されるいずれのコードでも使用できません。

以下の EXEC CICS コマンドは ON ユニットでは使用できません。

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

他のすべての EXEC CICS コマンドは ON ユニット内で許可されています。 ただし、 NOHANDLE オプション、 RESP オプション、 RESP2 オプションを使用してコーディングする必要がある。

第 3 章 PL/I カタログ式プロシージャーの使用

本章では IBM Enterprise PL/I for z/OS コンパイラーで使用する IBM 提供の標準カタログ式プロシージャーについて説明します。 また、プロシージャーの呼び出し方、一時的または永続的な変更方法に ついても説明します。

Language Environment SCEERUN2 データセットは、 STEPLIB にあり、カタログ化されたプロシージャのいずれかを使用するときにコンパイラがアクセスできなければならない。

カタログ式プロシージャーは、ライブラリーに保管された ジョブ制御ステートメントのセットです。カタログ式プロシージャーには、1 つ以上の EXEC ステートメントがあり、さらに 1 つ以上の DD ステートメントが続く場合があります。 ステートメントを検索するには、入力ストリーム内の EXEC ステートメント の PROC パラメーターの中で、カタログ式プロシージャーの名前を指定します。

カタログ化されたプロシージャーを使用することで、時間を節約し、Job Control Language (JCL) エラーを減らすことができます。 カタログ式プロシージャー内のステートメントがユーザー要件と正しく 一致していなくても、ジョブが終わるまでの間、簡単にステートメントを 変更したり、新たにステートメントを付け加えることができます。 こうしたプロシージャーは見直しを行って変更を加えることにより、使用できる各種機能を最も効率よく利用できるよう、また、ユーザー固有の規則を守れるようにしてください。

IBM 提供のカタログ式プロシージャー

このセクションでは、Enterprise PL/I for z/OS で使用するために提供されている PL/I カタログ式プロシージャーについて説明します。

コンパイルおよびリンク・エディットのための個々のステートメントについては、159 ページの『z/OS 、コンパイラを起動するには JCL』 および「*z/OS Language Environment Programming Guide*」を参照してください。

以下の PL/I カタログ式プロシージャーは、Enterprise PL/I for z/OS で使用するために提供されています。

IBMZC
Compile only

IBMZCB
Compile and bind

IBMZCBG
Compile, bind, and run

IBMQC
Compile only (64-bit)

IBMQCB
Compile and bind (64-bit)

IBMQCBG
Compile, bind, and run (64-bit)

カタログ手順 IBMZCB, IBMZCBG, IBMQCB および IBMQCBG は、 DFSMS/MVS 1.4 で紹介したプログラム管理バインダーの機能を使用している。 これらの手続きは、 PDSE にプログラムオブジェクトを生成します。

上記のカタログ式プロシージャーには、入力データ・セット用の DD ステートメントは入っていません。ユーザーが必ず提供しなければなりません。 カタログ化されたプロシージャーを呼び出す 」に示した例は、カタログ化されたプロシージャー IBMZCBG を呼び出して、 PL/I プログラムをコンパイル、バインド、実行するために使用する JCL ステートメントを示しています。

Enterprise PL/I REGION 32M の最小サイズが必要である。 大きいプログラムにはより多くのストレージが必要です。 実行中のカタログ化されたプロシージャを呼び出す EXEC 文で REGION を指定しなかった場合、コンパイラはサイトのデフォルトの REGION サイズを使用します。 このデフォルト・サイズは、PL/I プログラムのサイズにより適切な場合と 適切でない場合があります。

© Copyright IBM Corp. 1999, 2025

最適化をオンにしてプログラムをコンパイルした場合、 REGION のサイズ（と時間）はずっとずっと大きくなるかもしれない。 EXEC ステートメントで REGION を指定する例については、 148 ページの『#unique_267/unique_267_Connect_42_fig111』 を参照のこと。

例: カタログ式プロシージャーの呼び出し

カタログ式プロシージャーの呼び出し

```
//COLEGO    JOB
//STEP1     EXEC IBMZCBG, REGION.PLI=32M
//PLI.SYSIN DD *
                .
                .
                .
    (insert PL/I program to be compiled here)
                .
                .
                .
/*
```

コンパイルのみ (IBMZC)

カタログド・プロシージャー IBMZC に示されている IBMZC カタログド・プロシージャーには、プロシージャー・ステップが 1 つだけ含まれており、コンパイルに指定されるオプションは OBJECT と OPTIONS。(IBMZPLI はコンパイラーのシンボル名である。) PLI.SYSIN コンパイル・プロシージャ・ステップを含む他のカタログ・プロシージャと同様に、 IBMZC は入力データ・セット用の DD ステートメントを含まない。

OBJECT コンパイル時オプションを指定すると、コンパイラーはオブジェクト・モジュールを、リンケージ・エディターへの入力に適した構文で、SYSLIN という名前の DD 文で定義された標準データセットに配置します。 &&LOADSET ジョブの終了後もオブジェクト・モジュールを保持したい場合は、 &&LOADSET（つまり、&&で始まらない名前）に永続的な名前を代入し、データセットを使用した最後のプロシージャ・ステップの適切な DISP パラメータに KEEP を指定する必要があります。 以下のように、あなた自身の SYSLIN DD 。 このステートメントのデータセット名と処分パラメータは、 IBMZC プロシージャ SYSLIN DD ステートメントのものを上書きします。 この例では、コンパイル・ステップが唯一のジョブ・ステップです。

```
//PLICOMP EXEC IBMZC
//PLI.SYSLIN  DD  DSN=MYPROG,DISP=SHR
//PLI.SYSIN   DD ...
```

148 ページの『#unique_269/unique_269_Connect_42_fig113』 の **DISP** パラメータにある MOD という用語は、コンパイラがデータセットに複数のオブジェクト・モジュールを配置できるようにするもので、PASS は、対応する DD ステートメントがそこに含まれていれば、データセットが後の手続きステップで利用できるようにするものである。

SYSLIN SPACE パラメーターは、1 シリンダーの初期割り当てと、必要に応じてさらに 15 シリンダー（合計 16 シリンダー）の割り当てを可能にする。

カタログ手順 IBMZC

```
//IBMZC   PROC LNGPRFX='IBMZ.V6R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200
//*
//*********************************************************************
//* Licensed Materials - Property of IBM                              *
//* 5655-PL6                                                          *
//* Copyright IBM Corp. 1999, 2022 All Rights Reserved                *
//*                                                                   *
//* US Government Users Restricted Rights - Use, duplication or       *
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. *
//*                                                                   *
//*********************************************************************
//*
//*       IBM Enterprise PL/I for z/OS
//*       Version 6 Release 1 Modification 0
//*
//*  COMPILE A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
```

```
//*   LNGPRFX   IBMZ.V6R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*
//*  USER MUST SUPPLY //PLI.SYSIN DD STATEMENT THAT IDENTIFIES
//*   LOCATION OF COMPILER INPUT
//*
//*********************************************************************
//* COMPILE STEP
//*********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

コンパイルとバインド (IBMZCB)

149 ページの『#unique_270/unique_270_Connect_42_fig114a』 に示す IBMZCB カタログ手順には、2 つの手順ステップが含まれる：PLI はカタログ手順 IBMZC と同じであり、 BIND はプログラム管理バインダー（シンボリック名 IEWBLINK ）を呼び出して、最初の手順ステップで生成されたオブジェクト・モジュールをバインドする。

コンパイル手順ステップの入力データには、修飾子 ddname PLI.SYSIN が必要である。 EXEC ステートメント BIND の **COND** パラメータは、コンパイラが生成するリターンコードが 8 より大きい場合（つまり、コンパイル中に深刻なエラーや回復不能なエラーが発生した場合）、この手続きステップをバイパスすることを指定する。

カタログ手順 IBMZCB

```
//IBMZCB  PROC LNGPRFX='IBMZ.V6R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200,GOPGM=GO
//*
//*********************************************************************
//* Licensed Materials - Property of IBM                              *
//* 5655-PL6                                                          *
//* Copyright IBM Corp. 1999, 2022 All Rights Reserved                *
//*                                                                   *
//* US Government Users Restricted Rights - Use, duplication or       *
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. *
//*                                                                   *
//*********************************************************************
//*
//*       IBM Enterprise PL/I for z/OS
//*       Version 6 Release 1 Modification 0
//*
//*  COMPILE AND BIND A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V6R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*********************************************************************
//* COMPILE STEP
//*********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*********************************************************************
//* BIND STEP
//*********************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI)
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
```

```
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
```

プログラム・マネジメント・バインダーは、作成したプログラム・オブジェクトを常に、DD ステートメントで定義された標準データセットに、 SYSLMOD という名前で配置する。 カタログ・プロシージャのこのステートメントは、プログラム・オブジェクトが置かれ、メンバー名 GO が与えられる新しい一時ライブラリー &&GOSET を指定する。 テンポラリ・ライブラリの指定において、カタログ化されたプロシージャは、同じジョブでプログラム・オブジェクトを実行することを前提としています。プログラム・オブジェクトを保持したい場合は、 SYSLMOD という名前の DD 文の代わりに、独自の文を指定する必要があります。

コンパイル、バインド、実行 (IBMZCBG)

150 ページの『#unique_271/unique_271_Connect_42_fig115a』 に示される IBMZCBG カタログ手順は、3 つの手順ステップを含む： PLI, BIND そして、 GO. PLI と BIND は、 IBMZCBG の 2 つの手続きステップと同じであり、GO は、ステップ BIND で作成されたプログラム・オブジェクトを実行する。 GO ステップは、前のプロシージャー・ステップで 重大エラーまたは回復不能エラーが発生しなかった場合にのみ実行されます。

コンパイル手順ステップの入力データは、 PLI.SYSIN という名前の DD ステートメントで指定し、GO ステップは、 GO.SYSIN という名前の DD ステートメントで指定する。

カタログ式プロシージャー IBMZCBG

```
//IBMZCBG  PROC LNGPRFX='IBMZ.V6R1M0',LIBPRFX='CEE',
//              SYSLBLK=3200,GOPGM=GO
//*
//*******************************************************************
//* Licensed Materials - Property of IBM                            *
//* 5655-PL6                                                        *
//* Copyright IBM Corp. 1999, 2022 All Rights Reserved              *
//*                                                                 *
//* US Government Users Restricted Rights - Use, duplication or     *
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. *
//*                                                                 *
//*******************************************************************
//*
//*       IBM Enterprise PL/I for z/OS
//*       Version 6 Release 1 Modification 0
//*
//*  COMPILE, BIND, AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V6R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*******************************************************************
//* COMPILE STEP
//*******************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*******************************************************************
//* BIND STEP
//*******************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI)
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
```

```
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
//*******************************************************************
//* RUN STEP
//*******************************************************************
//GO       EXEC PGM=*.BIND.SYSLMOD,COND=((8,LT,PLI),(8,LE,BIND))
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

コンパイルのみ - 64 ビット (IBMQC)

カタログ手順 IBMQC (64 ビット) に示されている IBMQC カタログ手順は、1 つの手順ステップのみを含み、コンパイルに指定されるオプションは OBJECT と OPTIONS である。(IBMZPLI はコンパイラのシンボル名である) PLI.SYSIN コンパイル・プロシージャ・ステップを含む他のカタログ・プロシージャと同様に、 IBMQC は入力データ・セット用の DD ステートメントを含まない。

LP(64) コンパイル時オプションによりコンパイラーは 64 ビット・コードを生成します。 OBJECT コンパイル時オプションを指定すると、コンパイラーは、リンケージ・エディターへの入力に適した構文で、DD 文で定義された標準データセットに、 SYSLIN という名前でオブジェクト・モジュールを配置します。 &&LOADSET ジョブの終了後もオブジェクト・モジュールを保持したい場合は、 &&LOADSET （つまり、 &&で始まらない名前）に永続的な名前を代入し、データセットを使用した最後のプロシージャ・ステップの適切な DISP パラメータに KEEP を指定する必要があります。 以下のように、あなた自身の SYSLIN DD 。このステートメントのデータセット名と disposition パラメータは、 IBMQC プロシージャ SYSLIN DD ステートメントのものを上書きします。 この例では、コンパイル・ステップが唯一のジョブ・ステップです。

```
//PLICOMP EXEC IBMQC
//PLI.SYSLIN DD DSN=MYPROG,DISP=SHR
//PLI.SYSIN DD ...
```

カタログド・プロシージャー IBMQC（64 ビット ）の **DISP** パラメーターの MOD という用語は、コンパイラーがデータセットに複数のオブジェクト・モジュールを配置することを可能にし、PASS は、対応する DD 文がそこに含まれていれば、データセットが後のプロシージャー・ステップで利用可能であることを保証する。

SYSLIN SPACE パラメーターは、1 シリンダーの初期割り当てと、必要に応じてさらに 15 シリンダー（合計 16 シリンダー）の割り当てを可能にする。

カタログ式プロシージャー IBMQC (64 ビット)

```
//IBMQC   PROC LNGPRFX='IBMZ.V6R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200
//*
//*******************************************************************
//* Licensed Materials - Property of IBM                            *
//* 5655-PL6                                                        *
//* Copyright IBM Corp. 1999, 2022 All Rights Reserved              *
//*                                                                 *
//* US Government Users Restricted Rights - Use, duplication or     *
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. *
//*                                                                 *
//*******************************************************************
//*
//*       IBM Enterprise PL/I for z/OS
//*       Version 6 Release 1 Modification 0
//*
//*  COMPILE A 64-BIT PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V6R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*
//*  USER MUST SUPPLY //PLI.SYSIN DD STATEMENT THAT IDENTIFIES
//*   LOCATION OF COMPILER INPUT
//*
//*******************************************************************
//* COMPILE STEP
//*******************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='LP(64),OBJECT,OPTIONS'
```

```
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUT2   DD  DSN=&&SYSUT2,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUT3   DD  DSN=&&SYSUT3,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

コンパイルとバインド - 64-bit (IBMQCB)

152 ページの『#unique_273/unique_273_Connect_42_figibmqcb』 に示す IBMQCB カタログ手順には、2 つの手順ステップが含まれる：PLI はカタログ手順 IBMQCB と同じであり、BIND はプログラム管理バインダー（シンボリック名 IEWBLINK ）を呼び出して、最初の手順ステップで生成されたオブジェクト・モジュールをバインドする。

コンパイル手順ステップの入力データには、修飾子 ddname PLI.SYSIN が必要である。 EXEC ステートメント BIND の **COND** パラメータは、コンパイラが生成するリターンコードが 8 より大きい場合（つまり、コンパイル中に深刻なエラーや回復不能なエラーが発生した場合）、この手続きステップをバイパスすることを指定する。

64 ビット・プログラムのバインディングには、 DYNAM(DLL) binder オプションが必要で、 CASE(MIXED), AMODE(64)、 RMODE(ANY) オプションを強く推奨する。 さらに、 PL/I サイドデッキ IBMPQV11 と C サイドデッキファイル CELQS003 をバインダーの入力として含める必要がある。

カタログ式プロシージャー IBMQCB (64 ビット)

```
//IBMQCB  PROC LNGPRFX='IBMZ.V6R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200,GOPGM=GO
//*
//*******************************************************************
//* Licensed Materials - Property of IBM                            *
//* 5655-PL6                                                        *
//* Copyright IBM Corp. 1999, 2022 All Rights Reserved              *
//*                                                                 *
//* US Government Users Restricted Rights - Use, duplication or     *
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. *
//*                                                                 *
//*******************************************************************
//*
//*       IBM Enterprise PL/I for z/OS
//*       Version 6 Release 1 Modification 0
//*
//*  COMPILE AND BIND A 64-BIT PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V6R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*******************************************************************
//* COMPILE STEP
//*******************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='LP(64),OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUT2   DD  DSN=&&SYSUT2,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUT3   DD  DSN=&&SYSUT3,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*******************************************************************
//* BIND STEP
//*******************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
```

```
// PARM='CASE(MIXED),DYNAM(DLL),AMODE=64,RMODE=ANY,LIST,MAP,XREF'
//SYSLIB   DD  DSN=&LIBPRFX..SCEEBND2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DSN=&LIBPRFX..SCEELIB(CELQS003),DISP=SHR
//         DD  DSN=&LIBPRFX..SCEELIB(IBMPQV11),DISP=SHR
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
```

プログラム・マネジメント・バインダーは、作成したプログラム・オブジェクトを常に、DD ステートメントで定義された標準データセットに、 SYSLMOD という名前で配置する。 カタログ・プロシージャのこのステートメントは、プログラム・オブジェクトが配置され、メンバ名 GO が与えられる新しい一時ライブラリ &&GOSET を指定する。 テンポラリ・ライブラリの指定において、カタログ化されたプロシージャは、同じジョブでプログラム・オブジェクトを実行することを前提としています。プログラム・オブジェクトを保持したい場合は、 SYSLMOD という名前の DD 文の代わりに、独自の文を指定する必要があります。

コンパイル、バインド、実行 - 64 ビット (IBMQCBG)

153 ページの『#unique_274/unique_274_Connect_42_figibmqcbg』 に示す IBMQCBG のカタログ手順は、3 つの手順ステップを含む： PLI, BIND と GO。 PLI と は、 の 2 つの手続きステップと同一であり、GO はステップ で作成されたプログラムオブジェクトを実行する。 BIND IBMQCBG BIND GO ステップは、前のプロシージャー・ステップで 重大エラーまたは回復不能エラーが発生しなかった場合にのみ実行されます。

コンパイル手順ステップの入力データは、 PLI.SYSIN という名前の DD ステートメントで指定し、GO ステップは、 GO.SYSIN という名前の DD ステートメントで指定する。

カタログ手順 IBMQCBG (64 ビット)

```
//IBMQCBG  PROC LNGPRFX='IBMZ.V6R1M0',LIBPRFX='CEE',
//              SYSLBLK=3200,GOPGM=GO
//*
//*******************************************************************
//* Licensed Materials - Property of IBM                            *
//* 5655-PL6                                                        *
//* Copyright IBM Corp. 1999, 2022 All Rights Reserved              *
//*                                                                 *
//* US Government Users Restricted Rights - Use, duplication or     *
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp. *
//*                                                                 *
//*******************************************************************
//*
//*       IBM Enterprise PL/I for z/OS
//*       Version 6 Release 1 Modification 0
//*
//*  COMPILE, BIND, AND RUN A 64-BIT PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V6R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*******************************************************************
//* COMPILE STEP
//*******************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='LP(64),OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUT2   DD  DSN=&&SYSUT2,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSUT3   DD  DSN=&&SYSUT3,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*******************************************************************
//* BIND STEP
//*******************************************************************
```

```
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
// PARM='CASE(MIXED),DYNAM(DLL),AMODE=64,RMODE=ANY,LIST,MAP,XREF'
//SYSLIB   DD  DSN=&LIBPRFX..SCEEBND2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DSN=&LIBPRFX..SCEELIB(CELQS003),DISP=SHR
//         DD  DSN=&LIBPRFX..SCEELIB(IBMPQV11),DISP=SHR
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
//*******************************************************************
//* RUN STEP
//*******************************************************************
//GO       EXEC PGM=*.BIND.SYSLMOD,COND=((8,LT,PLI),(8,LE,BIND))
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

カタログ式プロシージャーの呼び出し

カタログ化されたプロシージャを呼び出すには、 EXEC 文の **PROC** パラメータにその名前を指定する。

例えば、カタログ化された手続き IBMZC を使用するには、入力ストリームの他のジョブ制御文の中の適切な位置に、以下の文を含めることができる：

```
//stepname EXEC PROC=IBMZC
```

キーワード **PROC** をコード化する必要はない。 EXEC 文の最初のオペランドが PGM= or PROC= で始まらない場合、ジョブスケジューラはそれをカタログ化されたプロシージャの名前として解釈する。 次のステートメントは上記のステートメントと同等です。

```
//stepname EXEC IBMZC
```

JOB ステートメントにパラメータ MSGLEVEL=1 を含めると、オペレーティングシステムはオリジナルの EXEC ステートメントをリストに含め、カタログ化されたプロシージャのステートメントを追加します。 このリストでは、カタログ式プロシージャーのステートメントは 最初の 2 文字の XX あるいは X/ によって識別されます。X/ は、カタログ式プロシージャーの現在の呼び出しで 変更されたステートメントを意味します。

DD ステートメントを追加するか、 EXEC または DD ステートメントの 1 つまたは複数のパラメータをオーバーライドすることによって、カタログ化されたプロシージャのステートメントを、それが呼び出されるジョブ・ステップの期間中変更する必要があるかもしれません。 例えば、コンパイラを呼び出すカタログ・プロシージャでは、 SYSIN という名前の DD 文を追加して、ソース文を含むデータ・セットを定義する必要があります。 また、1 つのジョブで複数の標準リンク・エディットのプロシージャー・ステップを使用する場合、 複数のロード・モジュールを実行したければ、呼び出す カタログ式プロシージャーのうち、最初のものを除くすべてのものを変更しなければ なりません。

複数のカタログ式プロシージャー呼び出しを指定

同じジョブ内で、複数のカタログ式プロシージャーを呼び出す ことができます。また、同じジョブ内で同じカタログ式プロシージャーを複数回呼び出す こともできます。

これらのカタログ式プロシージャーの 2 つ以上に同一の リンク・エディット・プロシージャー・ステップが関与しない限り、 特別な問題が起こることはないと思われます。 複数のカタログ式プロシージャーが同一のリンク・エディット・プロシージャー・ステップ に関与する場合には、すべてのロード・モジュールを実行できるようにするために、 以下に示す予防策をとる必要があります。

リンケージ・エディターがロード・モジュールを作成するとき、ロード・モジュールは DD 文で定義された標準データセットに SYSLMOD という名前で配置される。 バインダーがプログラム・オブジェクトを作成するとき、DD ステートメントで定義された PDSE に、 SYSLMOD という名前でプログラム・オブジェクトを配置します。 リンケージ・エディタ NAME ステートメントがない場合、リンケージ・エディタまたはバ

インダは、DSNAME パラメータで指定されたメンバ名をモジュール名として使用する。 標準カタログ手順では、 SYSLMOD という名前の DD 文は常に、メンバー名 GO の一時ライブラリ &&GOSET を指定する。

カタログ化されたプロシージャ IBMZCBG を同じジョブ内で 2 回使用して、2 つの PL/I プログラムをコンパイル、バインド、実行し、バインダが作成する 2 つのプログラム・オブジェクトにそれぞれ名前を付けなかった場合、最初のプログラム・オブジェクトは 2 回実行され、2 番目のプログラム・オブジェクトはまったく実行されません。

これを防ぐには、次に挙げる方法のうちどれか 1 つを使用します。

- GO ステップの最後にライブラリ &&GOSET を削除する。 GO ステップの終わりでの、カタログ式プロシージャーの最初の呼び出しで、 次の構文の DD ステートメントを追加します。

```
//GO.SYSLMOD DD DSN=&&GOSET,
//   DISP=(OLD,DELETE)
```

- `//BIND.SYSLMOD DD DSN=&&GOSET(GO1)`カタログ化されたプロシージャの 2 回目以降の呼び出しで、 SYSLMOD という名前の DD ステートメントを、ロード・モジュールの名前を変えるように修正する。
- NAME linkage editor オプションを使用して、各プログラムオブジェクトに異なる名前を付け、各ジョブステップ EXEC ステートメントを変更して、そのジョブステップの名前を持つプログラムオブジェクトの実行を指定します。

プログラム・オブジェクトにメンバ名を割り当てるには、 SYSLMOD DD 文の DSNAME パラメータでリンケージ・エディタ NAME オプションを使用します。 このプロシージャを使用する場合、プログラムを実行する EXEC ステートメントが、実行するモジュールの名前について SYSLMOD DD ステートメントを参照する場合は、membername は NAME オプションの名前と同じで**なければなりません**。

もう一つの方法は、次の例のように、EXEC プロシージャー文に GOPGM を使って、各プログラムに異なる名前をつけることである：

```
//  EXEC IBMZCBG,GOPGM=GO2
```

PL/I カタログ式プロシージャーの変更

カタログ化されたプロシージャを呼び出す EXEC ステートメントにパラメータを含めるか、EXEC ステートメントの後に追加の DD ステートメントを配置することで、カタログ化されたプロシージャを一時的に変更できます。

一時変更は、プロシージャーが呼び出されるジョブ・ステップの期間だけ 適用されます。 プロシージャー・ライブラリー内のカタログ式プロシージャーのメイン・コピーには影響しません。

一時的な修正は、カタログ化されたプロシージャー内の EXEC または DD ステートメントに適用できる。 EXEC ステートメントのパラメータを変更するには、カタログ化されたプロシージャーを呼び出す EXEC ステートメントに対応するパラメータを含める必要があります。 DD 文の 1 つ以上のパラメータを変更するには、カタログ化されたプロシージャを呼び出す EXEC 文の後に、対応する DD 文を含める必要があります。 カタログ化されたプロシージャーに新しい EXEC ステートメントを追加することはできませんが、常に追加の DD ステートメントを含めることができます。

EXEC ステートメント

カタログ化されたプロシージャを呼び出す EXEC ステートメントにパラメータを含めることで、カタログ化されたプロシージャを一時的に変更することができます。

カタログ化されたプロシージャを呼び出す EXEC ステートメントのパラメータに修飾されていない名前がある場合、そのパラメータはカタログ化されたプロシージャ内のすべての EXEC ステートメントに適用されます。 カタログ式プロシージャーに対する影響は、次のようにパラメーターに よって異なります。

- PARM は最初の手続きステップに適用され、他の パラメータは無効になる。 PARM
- COND および、すべての手順ステップに適用される。 ACCT
- TIME および は、すべての手順ステップに適用され、既存の値を上書きする。 REGION

例えば、次のステートメントが実行されると、その下に示されていることが行われます。

```
//stepname EXEC IBMZCBG,PARM='OFFSET',REGION=32M
```

- カタログに掲載されている手続き IBMZCBG を呼び出す。
- 手続きステップ PLI の EXEC ステートメントの OBJECT と OPTIONS のオプション OFFSET を置き換える。
- 手続きステップ BIND の EXEC 文の PARM パラメータを NULL にする。
- 3 つのプロシージャー・ステップすべてに、領域サイズ 32M を指定します。

カタログ化されたプロシージャの EXEC ステートメント 1 つだけで、パラメータ値を変更する場合、または EXEC ステートメント 1 つに新しいパラメータを追加する場合は、パラメータ名をプロシージャのステップ名で修飾して、EXEC ステートメントを特定する必要があります。 例えば、前の例でプロシージャー・ステップ PLI の領域サイズだけを変更するには、次のようにコーディングします。

```
//stepname EXEC PROC=IBMZCBG,PARM='OFFSET',REGION.PLI=90M
```

呼び出す EXEC 文で指定された新しいパラメータは、プロシージャ EXEC 文の対応するパラメータを完全に上書きする。

値を指定せずにキーワードと等号をコーディングすると、 パラメーターに指定されたオプションをすべて無効にすることができます。 例えば、カタログに載っている手続き IBMZCBG を呼び出すときに、リンケージ・エディターのリストの大部分を表示しないようにするには、次のようにコードする：

```
//stepname EXEC IBMZCBG,PARM.BIND=
```

DD ステートメント

EXEC ステートメントの後に追加の DD ステートメントを記述することで、カタログ化されたプロシージャーを一時的に変更することができます。

カタログ式プロシージャーに DD ステートメントを追加したり、 既存 DD ステートメントの 1 つ以上のパラメーターを変更するには、 入力ストリーム内の適切な場所に procstepname.ddname の形の DD ステートメントを入れなければなりません。 ddname が procstepname として 識別されているプロシージャー・ステップに既に存在する DD ステートメントの 名前である場合に、新しい DD ステートメント内のパラメーターは既存の DD ステートメントの 対応パラメーターをすべて指定変更します。そうでない場合は、新しい DD ステートメントがプロシージャー・ステップに追加されます。 例えば、以下のステートメントは、カタログ化されたプロシージャ IBMZC のプロシージャ・ステップ PLI に DD ステートメントを追加する：

```
//PLI.SYSIN DD *
```

以下の文は、既存の DD 文 SYSPRINT （コンパイラ・リストをクラス C のシステム出力デバイスに送信させる）を修正する。

```
//PLI.SYSPRINT DD SYSOUT=C
```

指定変更を行う DD ステートメントは、プロシージャー呼び出しの後で、 カタログ式プロシージャーに出現するとおりの順序で出現しなければなりません。 該当ステップに指定変更を行う DD ステートメントを指定した後に、DD ステートメント を追加することができます。

DD ステートメントのパラメーターを指定変更するには、そのパラメーターを改訂した形でコーディングするか、同様の機能を実行する 置換パラメーターをコーディングします (例えば、SPLIT を SPACE に 置き換えます)。 パラメーターを無効にするには、値を指定せずにキーワードと等号を コーディングします。
DCB サブパラメーターは、変更したいものだけをコーディングして 指定変更できます。 つまり、指定変更を行う DD ステートメントの DCB パラメーターは、 カタログ式プロシージャー内の対応ステートメントの DCB パラメーター全体を 指定変更するとは限りません。

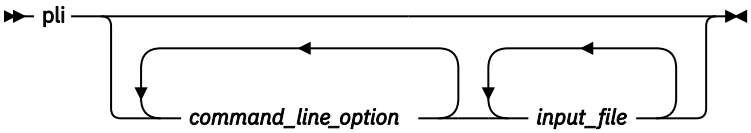
第 4 章 プログラムのコンパイル

この章では、z/OS UNIX システム・サービス (z/OS UNIX) の下でコンパイラーを呼び出す方法と、z/OS の下でコンパイルするのに使用するジョブ制御ステートメントについて説明します。

プログラムのコンパイル時に、 Language Environment SCEERUN のデータセットにコンパイラがアクセスできなければならない。

z/OS UNIX の下でのコンパイラーの呼び出し

z/OS UNIX 環境でプログラムをコンパイルするには、 **pli** コマンドを使用します。



command_line_option
次のようにして **command_line_option** を指定できます。

- **-q***option*
- オプション・フラグ (通常は前に - を付けた単一文字)

コマンド・ラインでコンパイル時オプションを指定した場合、ソース・ファイルで %PROCESS ステートメントを使って設定した場合とは書式が異なります。 158 ページの『z/OS UNIX の下でのコンパイル時オプションの指定』を参照してください。

input_file
プログラム・ファイル用の z/OS UNIX ファイル指定。

ファイル指定から拡張子が省略された場合、コンパイラーは拡張子 .pli を想定します。 パス全体が省略された場合、コンパイラーは現行ディレクトリーを想定します。

入力ファイル

pli コマンドは、 PL/I のソース・ファイルをコンパイルし、できあがったオブジェクト・ファイルを、コマンド・ラインで指定されたオブジェクト・ファイルやライブラリと、指定された順序でリンクし、1 つの実行可能ファイルを生成する。

pli コマンドが受け入れるファイルのタイプは次のとおりです。

ソース・ファイル – .pli
.pli ファイルはすべてコンパイル用のソース・ファイル です。 **pli** コマンドは、リストされた順にソース・ファイルを コンパイラーに送ります。 コンパイラーは、指定されたソース・ファイルを検出できない場合はエラー・メッセージを生成します。 その場合、**pli** コマンドは、次のファイルがあればそのファイルを処理します。

zFS のソースファイルはすべて行区切りで、 EBCDIC でエンコードされていなければならない。

PDS(E) にあるソースファイルをコンパイルすることもできます。 例えば、 PDS ファイル USER.SOURCE.PLI からメンバー SAMPLE をコンパイルしたい場合、次のコマンドで実行できる

```
pli -c "//'USER.SOURCE.PLI(SAMPLE)'"
```

で、データ・セット名 'USER.SOURCE.PLI(SAMPLE)' は、先行する "// "で囲まれています。

オブジェクト・ファイル – .o
.o ファイルはすべてオブジェクト・ファイルです。 **pli** コマンドは、 **-c** オプションを指定しない限り、リンク編集時にライブラリ・ファイルとともにすべてのオブジェクト・ファイルをリンク編集者に送る。 すべてのソース・ファイルをコンパイルした後、コンパイラーは リンケージ・エディターを呼

© Copyright IBM Corp. 1999, 2025

び出して、結果のオブジェクト・ファイルを、 入力ファイル・リストに指定されているすべてのオブジェクト・ファイルに リンク・エディットし、単一の実行可能出力ファイルを作成します。

ライブラリー・ファイル ー .a
pli コマンドは、リンク・エディット時にすべてのライブラリー・ファイル (.a ファイル) をリンケージ・エディターに送信します。

z/OS UNIX の下でのコンパイル時オプションの指定

Enterprise PL/I には、コンパイラーのデフォルト設定を変更するためのコンパイル時オプションが備わっています。 オプションはコマンド・ラインで指定でき、ソース・プログラム内の %PROCESS ステートメントがそれを上書きしない限り、ファイル内のすべてのコンパイル・ユニットに対して有効です。

これらのオプションの詳細については、3 ページの『コンパイル時オプションの説明』を参照してください。

コマンド行で指定したオプションは、オプションのデフォルト設定を 指定変更します。 指定したオプションは、ソース・ファイルで設定されているオプションにより 指定変更されます。

コマンド行でコンパイル時オプションを指定するには、次の 3 つの方法が あります。

- **-q***option_keyword* (コンパイラー特有)
- 単一および複数文字の旗
- -q+/u/myopts.txt

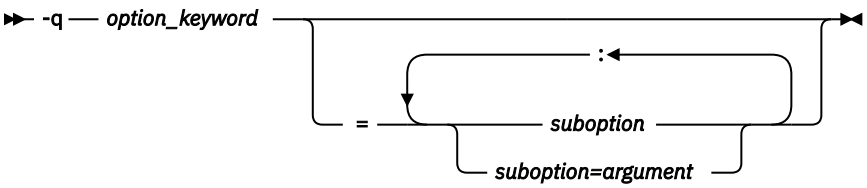
z/OS UNIX では、ユーザー出口入力ファイルが以下のような場所にあるとします。

```
/a/b/ibmuexit.inf
```

-qexit オプションを指定してコンパイルする前に `export DD_SYSUEXIT=/a/b/ibmuexit.inf` を指定するか、// シンボルが必要な `-qexit=//a/b/ibmuexit.inf` を指定してコンパイルすることができます。

-qoption_keyword

-q*option* のフォーマットを使用して、コマンド行でオプションを指定できます。



同じコマンド行で複数の **-q***option* を指定できます。 ただし、オプションをブランクで区切る必要があります。 オプション・キーワードは大文字または小文字のどちらでも指定できます。 ただし、**-q** は小文字で指定しなければなりません。

コンパイル時のオプションの中には、サブ・オプションを指定できるものもある。 これらのサブオプションは、コマンド ライン上で **-q** *option_keyword* の後に等号を付けて示されます。 複数のサブオプションはコロン(:)で区切り、空白を挟まないこと。

例えば、複数のサブオプションを含むオプションは RULES。 コマンドラインで RULES(LAXDCL) を指定するには、以下のコマンドを入力する：

```
-qrules=ibm:laxdcl
```

LIMITS オプションは、各サブオプションにも引数があるため、少し複雑である。 以下の例のように、コマンドラインで LIMITS(EXTNAME(31),FIXEDDEC(15)) ：

```
-qlimits=extname=31:fixeddec=15
```

関連情報

70 ページの『RULES』
RULES オプションは、特定の言語機能を許可または不許可にし、代替手段が利用可能な場合にセマンティクスを選択できるようにします。 これは一般プログラミング・エラーの診断に役立ちます。

48 ページの『LIMITS』
LIMITS オプションでは、各種のインプリメンテーションの制限を指定します。

単一フラグおよび複数文字フラグ

z/OS UNIX ファミリーのコンパイラーでは、共通の標準フラグをいくつも 使用します。 各言語には、独自の追加フラグのセットがあります。

フラグ・オプションによっては、フラグの一部を形成する引数があります。 以下の例では、/home/test3/include は、 INCLUDE ファイルを検索するインクルード・ディレクトリーである。

```
pli samp.pli -I/home/test3/include
```

それぞれのフラグ・オプションは、別々の引数 として指定してください。

表 *13. z/OS UNIX* の下で *Enterprise PL/I* によりサポートされるコンパイル時オプション・フラグ

オプション	説明
-c	コンパイルのみ。
-e	フェッチ可能なロード・モジュールに名前とエントリーを作成します。
-I< ディレクトリー >*	INCLUDE ファイルを検索するディレクトリにパス *<dir>* を追加する。 -I の後にはパス名が必要です。1 つの -I オプション には 1 つのパスしか指定できません。 複数のパスを追加するには、複数の -I オプションを使用します。 -I とパス名の間にスペースは入れないでください。
-0, -02	生成されるコードを最適化します。 このオプションは -qOPT=2 と同等です。
-q< オプション>*	それをコンパイラーに渡します。*<option>* はコンパイル時オプションです。 各オプションはコンマで区切り、各サブオプションは等号またはコロンで区切ってください。 **-Q (Q)** と*<option>*の間にスペースを入れることはできません。
-v	コンパイルおよびリンクのステップを表示し、それらのステップを実行します。
-#	コンパイルおよびリンクのステップを表示しますが、それらのステップを 実行しません。

注: * 指示のある個所に引数を指定しなければなりません。 そうしないと、予測できない結果となります。

z/OS 、コンパイラを起動するには JCL

コンパイラを呼び出すジョブ・ステップに必要なすべての JCL ステートメントを提供するよりも、カタログ化されたプロシージャを使用することが多いでしょうが、コンパイラを最大限に活用し、必要に応じてカタログ化されたプロシージャのステートメントをオーバーライドできるように、これらのステートメントを熟知しておく必要があります。

いわゆる「バッチ・コンパイル」と呼ばれているもの (複数のオブジェクト・デックが 1 回で生成されるコンパイル) はサポートされていません。

BPXBATCH によるコンパイラーの起動もサポートされていない。

次のセクションでは、コンパイルに必要な JCL について説明する。 IBM 提供のカタログ式プロシージャー (147 ページの『IBM 提供のカタログ式プロシージャー』) には、これら のステートメントが入っています。 カタログ式プロシージャーを使用しない場合だけ、これらのステートメント を自分でコーディングする必要があります。

EXEC ステートメント

基本的な EXEC ステートメントは //stepname EXEC PGM。

512K は、このステートメントの パラメーターに必要である。 **REGION**

最適化をオンにしてプログラムをコンパイルする場合は、必要な REGION サイズ (および時間) がはるかに大きくなることがあります。

EXEC ステートメントの **PARM** パラメータは、コンパイラが提供するオプション機能を 1 つ以上指定するために使用できる。 これらの機能は 163 ページの『EXEC ステートメントでオプションを指定する』で説明されています。 各種オプションの説明については、3 ページの『第 1 章 コンパイラー・オプションと機能の使用』を参照してください。

標準データ・セット用の DD ステートメント

コンパイラーは標準データ・セットをいくつか要求します。 データ・セットの数は、指定されたオプション機能によって異なります。 そのようなデータ・セットは、DD ステートメントで定義する必要があります。

これらのデータ・セットは、標準 DD 名を持つ DD ステートメントに 定義する必要があります。標準 DD 名は、他のデータ・セット特性 と共に、160 ページの表 14 に示されています。 DD ステートメント SYSIN、SYSUT1、および SYSPRINT は常に必要です。

標準データ・セットはどれも直接アクセス装置に保管できますが、DD ステートメント に SPACE パラメーターを組み込む必要があります。 このパラメーターは、必要な補助記憶域の量を指定するために データ・セットを定義します。 IBM 提供のカタログ式プロシージャーに割り振られる補助記憶域の量は、大部分のアプリケーションにとって十分と思われます。

表 *14.* コンパイラーの標準データ・セット

標準 DD 名	データ・セットの内容	使用可能な装置クラス [1]	レコード・フォーマット (RECFM)	レコード・サイズ (LRECL)
SYSDCLSD	XINFO(DCLS)出力	SYSDA	VB	16383
SYSDEBUG	TEST(SEPARATE) 出力	SYSDA	F,FB	> =80 かつ < =1024
SYSDEFSD	XINFO(DEF) 出力	SYSDA	F,FB	128
SYSIN	コンパイラーへの入力	SYSSQ	F,FB,U	<101(100)
			VB,V	<105(104)
syslib	INCLUDE ファイル用ソース・ステートメント	SYSDA	F,FB,U	<101
			V,VB	<105
SYSLIN	オブジェクト・モジュール	SYSSQ	FB	80
SYSPRINT。	リスト (メッセージを含む)	SYSSQ	VBA	137 (MARGINS <= 100 の場合) 255 (MARGINS > 100 の場合)
SYSPUNCH	プリプロセッサー出力、コンパイラー出力	SYSSQ SYSCP	FB	80 または MARGINS() 値
SYSUT1	一時作業ファイル	SYSDA	F	4051
SYSUT2	一時作業ファイル	SYSDA	FB	3200

表 *14.* コンパイラーの標準データ・セット *(*続き*)*

標準 DD 名	データ・セットの内容	使用可能な装置クラス [1]	レコード・フォーマット (RECFM)	レコード・サイズ (LRECL)
SYSUT3	一時作業ファイル	SYSDA	FB	3200
SYSXMLSD	XINFO(XML) 出力	SYSDA	VB	16383
SYSADATA	XINFO(MSG) 出力	SYSDA	U	1024

1. **装置クラスの説明**

   SYSSQ
   順次装置

   SYSDA
   直接アクセス装置

   SYSUT1 以外はブロック・サイズを指定できます。 SYSUT1 のブロック・サイズと論理レコード長はコンパイラーが選択します。

ノート:

1. 指定変更できるコンパイル時 SYSPRINT の値は BLKSIZE だけです。
2. SYSUT2 と SYSUT3 は、GONUMBER オプションまたは TEST オプションが有効になっている場合に限り、LP(64) でのみ必要となります。

入力 (SYSIN)

コンパイラへの入力は、 SYSIN という名前の DD ステートメントで定義されたデータセットでなければならない。

このデータセットは CONSECUTIVE 。 入力は 1 つ以上の外部 PL/I プロシージャーでなければなりません。 1 つのジョブまたはジョブ・ステップで複数の外部プロシージャをコンパイルする場合は、最初のプロシージャを除き、各プロシージャの前に %PROCESS 文を記述してください。

PL/I ソース・プログラムの入力メディアとして、80 バイト・レコードが 一般に使用されます。 入力データ・セットは直接アクセス装置または他の順次メディアに記録できます。 入力データ・セットには、固定長レコード (ブロック化または非ブロック 化のもの)、可変長レコード (コード化または非コード化のもの)、または不定長レコードのいずれでも入れることができます。 最大レコード・サイズは 100 バイトです。

入力ファイルの最大行数は 999999 です。

コンパイラーへの入力のためにデータ・セットを連結する場合は、 連結されたデータ・セットは同じような特性 (例えば、ブロック・サイズや レコード・フォーマット) を持っていなければなりません。

出力 (SYSLIN, SYSPUNCH)

OBJECT コンパイル・タイム・オプションを指定すると、コンパイラーからの 1 つ以上のオブジェクト・モジュール形式の出力がデータ・セット SYSLIN に格納される。 このデータセットは DD 。

オブジェクト・モジュールは、常に、ブロック化または非ブロック化の 80 バイトの 固定長レコードの形式になります。 SYSLIN に BLKSIZE が指定され、80 でない場合、 LRECL には 80 を指定しなければならない。

SYSLIN DD 、一時的なデータ・セットか恒久的なデータ・セットのどちらかを指定しなければならない。 いかなるタイプのデータ・セットも連結して指定することはできない。

SYSLIN DD は、 PDS や PDSE ではなく、シーケンシャルなデータセットを指定しなければならない。

MDECK コンパイル時オプションを指定すると、SYSPUNCH という名前で DD ステートメントによって定義されたデータセットも、プリプロセッサからの出力を格納するために使用される。

一時的なワークファイル(SYSUT1)

コンパイラーは、一時作業ファイルとして使用するためのデータ・セットを 必要とします。 これは、SYSUT1 という名前の DD ステートメントによって定義され、 スピルファイルとして知られている。 これは直接アクセス装置上にある必要があり、マルチボリューム・データ・セットとして割り振ってはなりません。

予備ファイルは、主記憶域の論理拡張部分として使用され、 テキストと辞書情報を入れるためにコンパイラーとプリプロセッサーに よって使用されます。 SYSUT1 の LRECL と BLKSIZE は、スピルファイルページに利用可能なストレージ量に基づいてコンパイラが選択する。

本書および SYSUT1 のカタログ手順に記載されている DD の記述は、1024 バイトのブロック単位での領域割り当てを要求している。 これは直接アクセス・ストレージ・スペースの適切な 2 次割り振りが取得されるようにするために行われます。

一時的なワークファイル (SYSUT2, SYSUT3)

LP(64)オプションでプログラムをコンパイルし、GONUMBER または TEST オプションが有効な場合、コンパイラーは SYSUT2 と SYSUT3 をテンポラリ・データ・セットとして必要とします。

大規模なプログラムの場合、 SYSUT2 または SYSUT3 データセットに十分な空き容量がないと、コンパイラが異常終了することがある。

リスト (SYSPRINT)

コンパイラーは、処理したすべてのソース・ステートメント、オブジェクト・モジュールに関連した情報、および、必要に応じて メッセージの入ったリストを生成します。

リストに入れられる情報の大半はオプショナル情報であり、適切なコンパイル 時オプションを入れれば、ユーザーが必要とする情報部分を指定することが できます。 表示される可能性がある情報、および関連コンパイル時オプションについては、106 ページの『コンパイラー・リストの使用』を参照してください。

コンパイラにリストを保存させたいデータセットは、 SYSPRINT という名前の DD ステートメントで定義する必要があります。 このデータセットは CONSECUTIVE 。 リストは通常印刷されますが、任意の順次装置または直接アクセス装置に 保管することもできます。 ご使用のシステムで出力クラス A がプリンターを指すきまりに なっている場合は、印刷出力には次のステートメントで十分です。

```
//SYSPRINT DD SYSOUT=A
```

ソース・ステートメント・ライブラリ (SYSLIB)

INCLUDE 文を使用して、ライブラリからソース文を PL/I プログラムに導入する場合は、 DD 文でライブラリを SYSLIB という名前で定義するか、独自の ddname（または ddnames）を選択し、各 %INCLUDE 文で ddname を指定します。

DD ステートメントでは、 PDS または PDSE を指定するが、実際のメンバーは指定しない。 たとえば、データ・セット INCLUDE.PLI を使用して、ライブラリ SYSLIB からファイル HEADER をインクルードするには、次の %INCLUDE ステートメントのいずれかを使用します：

- `%INCLUDE HEADER;`
- `%INCLUDE SYSLIB( HEADER );`

DD ：

```
SYSLIB  DD DISP=SHR,DSN=INCLUDE.PLI
```

ただし、次のステートメントは無効です。

```
SYSLIB  DD DISP=SHR,DSN=INCLUDE.PLI(HEADER)
```

すべての%INCLUDE ファイルは、 SYSIN ソース・ファイルと同じレコード・フォーマット（固定、可変、または未定義）、同じ論理レコード長、同じ左右マージンを持つ必要があります。

ライブラリの BLOCKSIZE は 32760 バイト以下でなければならない。

どのインクルード・ファイルであれ、1 ファイル当たりの最大行数は 999999 です。

オプションの指定

各コンパイルでは、PROCESS 文または EXEC 文の PARM パラメータでオプションを指定して上書きしない限り、コンパイル時オプションの IBM -supplied またはインストール時のデフォルトが適用される。

PARM パラメータで指定されたオプションはデフォルト値を上書きし、%PROCESS 文で指定されたオプションは PARM パラメータで指定された値とデフォルト値の両方を上書きする。

注 : 矛盾する属性が他のオプションの指定により明示的または暗黙的に 指定された場合は、最新の暗黙的または明示的なオプションが受け入れられます。 このように指定変更されるオプションについては、診断メッセージは 出されません。

EXEC ステートメントでオプションを指定する

EXEC ステートメントでオプションを指定するには、 PARM= の後にオプションのリストを記述する。 オプションは任意の順序でリストできます。 オプションはコンマで区切り、リストは単一引用符で囲む必要があります。

次の例を参照してください。

```
//STEP1 EXEC PGM=IBMZPLI,PARM='OBJECT,LIST'
```

MARGINI('c') のように引用符を伴うオプションの場合は、引用符を二重にする必要があります。 オプション・リストの長さは、分離文字コンマを含めて 100 文字を 超えてはなりません。 ただし、使用可能であれば、オプションの省略構文を使用してスペースを節約できます。 ステートメントが次行へ続くときには、オプション・リストを括弧 (引用符ではなく) で囲み、各行内のオプション・リストを引用符で囲み、そして、最終行を除く各行の最後のコンマが引用符の外側にくるようにしなければなりません。 これらの点をすべて以下に例示します。

```
//STEP1 EXEC PGM=IBMZPLI,PARM=('AG,A',
//       'C,F(I)',
// 'M,MI(''X''),NEST,STG,X')
```

カタログ化されたプロシージャを使用していて、オプションを明示的に指定したい場合は、次の例のように、コンパイラを呼び出すプロシージャ・ステップの名前でキーワード PARM を修飾して、そのプロシージャを呼び出す EXEC 文に PARM パラメータを含める必要があります：

```
//STEP1 EXEC nnnnnnn,PARM.PLI='A,LIST'
```

EXEC 、オプション・ファイルを使ってオプションを指定する

EXEC ステートメントでオプションを指定するもうひとつの方法は、オプション・ファイルですべてのオプションを宣言し、次のようにコーディングすることである：

```
//STEP1 EXEC PGM=IBMZPLI,PARM='+DD:OPTIONS'
```

この方法を使うと、よく使用するオプションの整合性のあるセットを 作成できます。 これは他のプログラマーに共通のオプション・セットを使ってもらいたい 場合に特に効果的です。 また、100 文字の制限もなくなります。

MARGINS オプションはオプション・ファイルには適用されない：1 列目のデータはオプションの一部として読み込まれる。 また、ファイルが F フォーマットの場合、72 桁目より後のデータは無視されます。

parm ストリングには「通常」のオプションを使用でき、 複数のオプション・ファイルを指定することもできます。 例えば、 LIST のオプションと、 GROUP DD のファイルと PROJECT DD のファイルの両方のオプションを指定するには、次のように指定する：

```
  PARM='LIST +DD:GROUP +DD:PROJECT'
```

PROJECT ファイルのオプションは、 GROUP ファイルのオプションより優先される。

またこの場合、 LIST オプションは、いずれかのオプションファイルで指定された NOLIST オプションによってオフにされるかもしれない。 LIST オプションがオンになっていることを確認するには、以下のように指定する：

```
PARM='+DD:GROUP +DD:PROJECT LIST'
```

z/OS UNIX 環境では、オプション・ファイルも使用できます。 例えば z/OS UNIX で、ファイル /u/pli/group.opt に含まれるオプションを使用して sample.pli をコンパイルするには、次のコマンドを使用します。

```
pli -q+/u/pli/group.opt  sample.pli
```

コンパイラーの旧リリースでは、オプション・ファイルの指定の前に付ける トリガー文字として、文字 '@' を使用していました。 この文字は、 EBCDIC コードポイントの不変セットの一部ではない。そのため、不変である文字「+」が好まれる。 ただし、「@」文字は 16 進値 '7C'x を使用して指定される限りは引き続き使用できます。

第 5 章 31 ビット・プログラムに対するリンク・エディットおよび実行

LP(32)でコンパイルした後、31 ビット・プログラムは、 Language Environment ランタイム・ライブラリへの参照と同様に、互いに未解決の参照を含む 1 つ以上のオブジェクト・モジュールで構成されます。 これらの参照は、リンク・エディット時 (静的) または実行時 (動的) に 解決されます。

したがって、PL/I プログラムのコンパイルが終わった後で次にとるべきステップは、そのプログラムをテスト・データを用いてリンクして実行し、予想どおりの結果が出るかどうか確認することです。

Language Environment は、ユーザーがプログラムを実行するのに必要なランタイム環境とサービスを提供します。 PL/I およびその他すべての Language Environment に準拠した言語プログラムのリンクと実行については、「*z/OS Language Environment Programming Guide*」を参照してください。 既存の PL/I プログラムから Language Environment への移行については、「*Enterprise PL/I for z/OS* コンパイラーおよびランタイム 移行ガイド」を参照してください。

31 ビット・プログラムに関するリンク・エディットの考慮事項

オプション RENT またはオプション LIMITS(EXTNAME(*n*)) で *n* > 8 でコンパイルした場合、リンカ出力に PDSE を使用する必要があります。

31 ビット・プログラムでバインダーを使用

バインダー出力は PDSE に入れる必要があります。

DLL をリンクする場合、バインドステップ中に必要な定義サイドデッキを指定する必要がある。

ENTRY カードの使用

フェッチされるモジュールで、 Enterprise PL/I ルーチンをエントリー・ポイントとして持つモジュールをビルドする場合、 ENTRY カードには、その PL/I エントリー・ポイントの名前を指定する必要がある。

モジュールを Enterprise PL/I からフェッチする場合、 ENTRY カードに CEESTART を指定することができるが、これは強く推奨されない。 しかし、モジュールが COBOL またはアセンブラからフェッチされる場合、 ENTRY カードには、 CEESTART ではなく、 PL/I モジュールへのエントリー・ポイントの名前を絶対に指定しなければならない。

31 ビット・プログラムに関する実行時の考慮事項

プログラム初期化ルーチンに渡されるパラメーターとしてランタイム・オプションを指定できます。 また、PLIXOPT 変数でランタイム・オプションを指定することもできる。 また、 PLIXOPT 変数を使ってランタイム・オプションを指定し、プログラムを再コンパイルすることで、既存のプログラムを変更すれば、パフォーマンスの観点からも有益であることがわかるかもしれない。

PLIXOPT の使い方については *z/OS Language Environment Programming Guide*.

端末の入出力を簡素化するため、端末に割り当てられるストリーム・ファイル には各種規則が取り入れられています。 次の 3 つの領域が影響を受けます。

1. PRINT ファイルのフォーマット
2. 自動プロンプト機能
3. 入力のスペーシングと句読法の規則

注 : 端末でのレコード入出力にはプロンプトその他の機能は 提供されていません。したがって、端末から、または端末への伝送には ストリーム入出力を使用することを強くお勧めします。

© Copyright IBM Corp. 1999, 2025

PRINT ファイルの書式規則

PRINT ファイルがターミナルに割り当てられた場合、印刷されながら読み込まれることが想定される。 したがって、印刷時間を短縮するために、スペーシングは最小限に 行われます。

PAGE, SKIP、および ENDPAGE キーワードには以下のルールが適用される：

- PAGE オプションや書式項目は 3 行スキップされる。
- SKIP オプションや、 (2)より大きな書式項目は、3 行スキップされる。 SKIP SKIP (2)以下は通常の扱い。
- ENDPAGE 。

31 ビットプログラム用 PRINT ファイルのフォーマット変更

ターミナルで PRINT ファイルからの出力に通常のスペーシングを適用したい場合は、PL/I 用に独自のタブテーブルを用意する必要がある。

以下の手順に従います。

1. メイン・プログラムまたはメイン・プログラムにリンクされたプログラム内で、 PLITABS という外部構造体を宣言する。
2. 要素を初期化する PAGELENGTH ページに収まる行数に。 この値は PAGESIZE とは異なり、 ENDPAGE が立ち上がる前にページに印刷したい行数を定義するものである（ 167 ページの図 1 参照）。

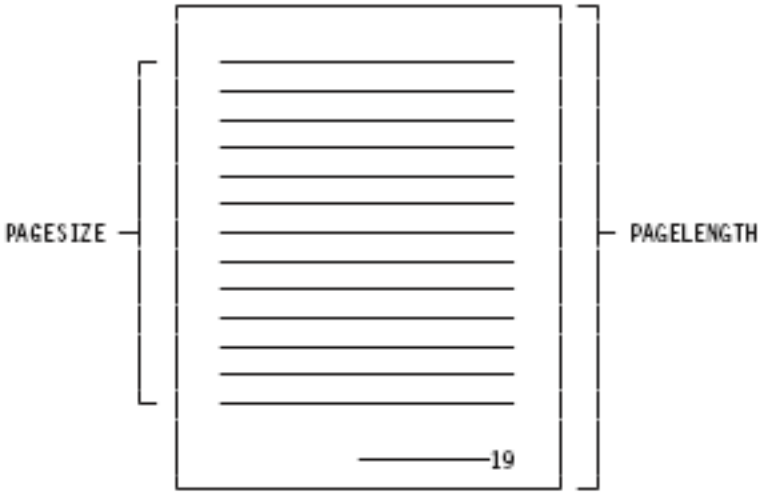
64 行の PAGELENGTH が必要な場合は、 166 ページの『#unique_302/unique_302_Connect_42_tabsrh』のように PLITABS を宣言する。 タブ・テーブルの指定変更の詳細については、243 ページの『タブ制御テーブルの指定変更』を参照してください。

コードに PLITABS の宣言が含まれている場合、PLITABS 構造体の値と最初のフィールドがすべて有効でなければならない。 このフィールドには、構造体が設定されるタブの数を指定しているフィールドへのオフセットが入っている必要があります。そうでない場合、Enterprise PL/I ライブラリー・コードは正しく機能しません。

RENT オプションでコンパイルする場合、 PLITABS は NONASGN 属性で宣言しなければならない。 NONASGN 属性は NORENT でコンパイルする際にも指定できるため、 PLITABS は常に NONASGN 属性と一緒に宣言することをお勧めします。

Declaration of PLITABS- この宣言は、標準的なページサイズ、行サイズ、および集計位置を示す。

```
DCL 1 PLITABS STATIC EXTERNAL NONASGN,
  ( 2   OFFSET INIT (14),
    2    PAGESIZE INIT (60),
    2    LINESIZE INIT (120),
    2    PAGELENGTH INIT (64),
    2    FILL1 INIT (0),
    2    FILL2 INIT (0),
    2    FILL3 INIT (0),
    2    NUMBER_OF_TABS INIT (5),
    2    TAB1 INIT (25),
    2    TAB2 INIT (49),
    2    TAB3 INIT (73),
    2    TAB4 INIT (97),
    2    TAB5 INIT (121)) FIXED BIN (15,0);
```



PAGELENGTH: the number of lines that can be printed on a page

PAGESIZE: the number of lines that will be printed on a page
before the ENDPAGE condition is raised

図 *1. PAGELENGTH* および *PAGESIZE*

自動プロンプト

プログラムは、端末に関連したファイルからの入力が必要になると、 プロンプトを出します。 このプロンプトは、次の行にコロンを印刷してから、 コロンの次の行の 1 桁目にスキップするという形をとります。 これにより、行全体を入力に使用できます。

次の例を参照してください。

```
:
(space for entry of your data)
```

この種のプロンプトを 1 次プロンプトと呼びます。

自動プロンプトの指定変更

1 次プロンプトを指定変更するには、データ要求の最後の項目をコロンに します。 2 次プロンプトは指定変更できません。

例えば、次の 2 つの PL/I ステートメントが実行されると、167 ページの図 2 に示されている出力が端末に表示されます。

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));
```

```
ENTER TIME OF PERIHELION
:         (automatic prompt)
(space for entry of data)
```

図 *2.* 自動プロンプトを使用した出力

ただし、次のように、最初のステートメントで出力の終わりにコロンがある場合、自動プロンプトは指定変更されます。 端末に表示されるシーケンスを 168 ページの図 3 に示します。

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

図 *3.* 自動プロンプトを使用しない出力

注 : 指定変更は 1 つのプロンプトにのみ有効です。 自動プロンプトを指定変更しない限り、次の項目に関するプロンプトが 自動的に出されます。

長い入力行の句読法

端末で 2 行以上のスペースを必要とするデータを 1 つのデータ項目として送信するには、 SBCS ハイフンを行継続文字として使用する必要があります。 最終行を除く各行の最後の文字として、 SBCS ハイフンを入力する。

例えば、`this data must be transmitted as one unit.` という文を伝送するには、次のようにデータを入力する必要があります。

```
:'this data must be transmitted -
+:as one unit.'
```

`unit.'` の後で ENTER を押すまで伝送は行われません。 ハイフンは除去されます。 送信された項目は論理行 と呼ばれます。

注 : 最後のデータ文字がハイフンまたは PL/I 負符号である行を送 るには、行の終わりに 2 つのハイフンを入力し、次の行をヌル行にします。 次の例を参照してください。

```
xyz--
(press ENTER only on this line)
```

GET LIST と GET DATA の句読点

GET LIST 、 GET DATA ステートメントでは、プログラマーがカンマを省略した場合、ターミナルから送信される各論理行の末尾にカンマが追加される。 したがって、項目を別々の論理行に入力する場合は、項目を区切るための ブランクまたはコンマを入力する必要はありません。

PL/I ステートメント `GET LIST(A,B,C);` が指定された場合、端末で入力できるデータは次のとおりです。

```
:1
+:2
+:3
```

この規則は、文字ストリング・データの入力にも適用されます。 したがって、1 つの文字ストリングは 1 つの論理行として送信する 必要があります。 そうしないと、ブレークポイントにコンマが置かれます。 例えば、以下のデータを入力するとします。

```
:'COMMAS SHOULD NOT BREAK
+:UP A CLAUSE.'
```

結果のストリングは `COMMAS SHOULD NOT BREAK, UP A CLAUSE.` となります。

行継続文字としてハイフンが使用された場合、コンマは追加されません。

自動パディング GET EDIT

GET EDIT 、行末に空白を入れる必要はない。 データは指定された長さまで埋め込まれます。

例えば、 PL/I ステートメント GET EDIT (NAME) (A(15));の場合、5 文字の SMITH を入力すると、プログラムが 15 文字を受け取るようにデータに 10 個のブランクが埋め込まれます。

```
'SMITH          '
```

注 : 単一のデータ項目は 1 つの論理行として送信する必要があります。 そうしないと、送信される最初の行に必要なブランクが埋め込まれ、 完了したデータ項目と見なされます。

端子入力に SKIP を使用

入力に対する SKIP の使用はすべて、ファイルが端末に割り当てられた時点で SKIP(1) と解釈される。 SKIP(1)は、現在使用可能な論理行の未使用データをすべて無視する命令として扱われる。

ENDFILE

端末でファイルの終わりを入力できます。 それには、2 つの文字 /* からなる論理行を入力します。

それ以上、ファイルを閉じずに使用しようとすると、 ENDFILE の条件が発生する。

SYSPRINT 31 ビットプログラムへの配慮

PL/I 標準 SYSPRINT ファイルは、アプリケーション内の複数のエンクレーブで共有される。 同じエンクレーブまたは異なるエンクレーブから、例えば STREAM PUT のような I/O リクエストを発行できる。 これらのリクエストは、アプリケーション全体に共通するファイルとして、標準の PL/I SYSPRINT 。 SYSPRINT ファイルが暗黙的にクローズされるのは、アプリケーションが終了するときだけであり、エンクレーブが終了するときではない。

標準の PL/I SYSPRINT ファイルには、 STREAM PUTs のような、ユーザー主導の出力のみが含まれる。 ランタイムライブラリメッセージやその他の同様の診断出力は、 Language Environment MSGFILE。 を参照のこと。 *z/OS Language Environment Programming Guide* SYSPRINT ファイルの出力を Language Environment MSGFILE にリダイレクトする方法の詳細については、こちらを参照してください。

アプリケーション内の複数のエンクレーブで共有するには、 PL/I SYSPRINT ファイルを SYSPRINT というファイル名で EXTERNAL FILE 定数として宣言し、STREAM 属性と OUTPUT 属性、および OPEN 時の PRINT 属性（暗示的）を持たなければならない。 これは、コンパイラがデフォルトで指定する標準の SYSPRINT 。

アプリケーション内には標準的な PL/I SYSPRINT FILE が 1 つだけ存在し、このファイルはアプリケーション内のすべてのエンクレーブで共有される。 例えば、 SYSPRINT ファイルは、アプリケーション内の複数のネストされたエンクレーブや、Language Environment プリイニシャライゼーション関数によってアプリケーション内で作成および終了される一連のエンクレーブによって共有することができる。 アプリケーション内のエンクレーブで共有するには、 PL/I SYSPRINT ファイルをそのエンクレーブで宣言する必要があります。 標準 SYSPRINT ファイルは、エンクレーブ間でファイル引数として渡して共有することはできない。 標準 SYSPRINT ファイルで宣言された属性は、EXTERNAL で宣言された定数と同様に、アプリケーション全体で同じでなければならない。 PL/I はこの規則を強制しません。 TITLE オプションと MSGFILE(SYSPRINT) オプションの両方が、 SYSPRINT を別のデータセットにルーティングしようとする。 そのため、2 つのオプションが一緒に使われた場合、競合が発生し、 TITLE のオプションは無視される。

アプリケーション内に共通の SYSPRINT ファイルを持つことは、密接に結びついた飛び地を利用するアプリケーションにとって利点となる。 しかし、アプリケーション内のすべてのエンクレーブは、同じ 共用データ・セットに書き込みを行うので、各エンクレーブ間での 何らかの調整が必要となります。

SYSPRINT ファイルは、アプリケーションのエンクレーブ内で最初に参照されたときに（暗黙的または明示的に）オープンされる。 SYSPRINT ファイルが CLOSE されると、ファイル・リソースは解放され（ファイルが開かれていなかったかのように）、すべてのエンクレーブがクローズされたステータスを反映するように更新される。

複数の飛び地のアプリケーションで SYSPRINT 、LINENO 組み込み関数は、飛び地の最初の PUT または OPEN が発行されるまで、現在の行番号を返すだけである。 これは旧プログラムとの完全な互換性を維持するために 必要です。

COUNT 内蔵機能は飛び地レベルで維持されている。 これは常に、エンクレーブ内の最初の PUT が出されるまで ゼロの値を返します。 ネストされた子エンクレーブが親エンクレーブから呼び出された場合、親エンクレーブが子エンクレーブから制御を取り戻すと、 COUNT 組み込み関数の値は未定義になる。

TITLE オプションを使用して、標準 SYSPRINT ファイルを 異なるオペレーティング・システム・データ・セットと関連付けることができますが、特定のオープン 関連付けは、別のものをオープンする前にクローズする必要があることに留意してください。 この関連付けはオープン状態が続く間は各エンクレーブ間で保持されます。

PL/I 標準的な ファイルに関連する条件処理は、現在のセマンティクスとスコープを維持する。 PL/I SYSPRINT 例えば、子エンクレーブ内で生じた ENDPAGE 条件は、その子エンクレーブ 内で、設定された ON ユニットを呼び出すだけです。 これは親エンクレーブ内の ON ユニットの呼び出しは行いません。

標準 PL/I SYSPRINT ファイルのタブは、異なるエンクレーブから PUT が行われる場合、エンクレーブにユーザー PLITABS テーブルが含まれていれば、異なることがある。

PL/I SYSPRINT 、RECORD ファイルまたは STREAM INPUT ファイルとして使用される場合、 PL/I 、個々のエンクレーブまたはタスク・レベルでサポートされるが、エンクレーブ間で共有可能なファイルとしてはサポートされない。 同じアプリケーションの異なるエンクレーブで、異なるファイル属性（例えば RECORD と STREAM）で PL/I SYSPRINT ファイルを同時に開いた場合、結果は予測不可能となる。

SYSPRINT は、 でコンパイルされたコードと、 の古いコンパイラでコンパイルされたコードとの間でも共有することができるが、以下の条件をすべて満たす必要がある： Enterprise PL/I PL/I

- SYSPRINT は として宣言しなければならない。 STREAM OUTPUT
- アプリケーションは TSO の下で実行されてはならない。
- ランタイム・オプション MSGFILE(SYSPRINT) が有効な場合、アプリケーション内に初期化前のプログラムとストアド・プロシージャがあってはならない。

MSGFILE(SYSPRINT) の使用

SYSPRINT STREAM PRINT 、ファイル宣言の ENVIRONMENT オプションで指定されたファイル属性はすべて無視される。

SYSPRINT の OPEN ステートメントで指定された属性はすべて無視される。

OPEN ステートメントを使用して PL/I SYSPRINT STREAM PRINT ファイルを開くと、そのファイルは PL/I 制御ブロック内で opened としてマークされますが、実際には言語環境によって開かれます。

CLOSE ステートメントを使用して PL/I SYSPRINT STREAM PRINT ファイルを閉じると、そのファイルは PL/I 制御ブロックで closed としてマークされますが、 言語環境はそのファイルを開いたままにします。

Language Environment メッセージと PL/I ユーザー指定出力の間で同期化は行われないため、出力の順序は予測不能です。

MSGFILE(SYSPRINT) を使うと、LINESIZE オプションで指定された行のサイズが最大 225 文字に制限される。

31 ビット・アプリケーションのルーチンで FETCH を使用する

Enterprise PL/I では、PL/I、C、COBOL、またはアセンブラーによってコンパイルされたルーチンをフェッチできます。

31 ビット・アプリケーションで Enterprise PL/I ルーチンをフェッチ

旧 PL/I コンパイラーによって課せられた、フェッチされたモジュールの制限は、 ほぼすべて除去されました。 そのため、現在、フェッチされたモジュールは以下の操作を実行できます。

- 他のモジュールをフェッチする。
- 任意の PL/I ファイルに入出力操作を実行する。 このファイルは、フェッチされたモジュール、メイン・モジュール、または他のフェッチされたモジュールによりオープンできます。

- ALLOCATE そして、 自身の 変数。 FREE CONTROLLED

しかし、フェッチの対象である Enterprise PL/I モジュールに対するいくつかの制限があります。

1. OPTIONS(FETCHABLE) リンク編集ステップで カードが提供されない場合は、フェッチされたルーチンの ステートメントで指定する必要がある。 ENTRY PROCEDURE
2. ENTRY 、その PL/I エントリーポイントの名前を指定する。
   - モジュールを Enterprise PL/I から取得する場合、 ENTRY カードに CEESTART を指定することができるが、これは強く推奨されない。
   - しかし、モジュールが COBOL またはアセンブラからフェッチされる場合、 ENTRY カードは絶対に、CEESTART ではなく、 PL/I モジュールへのエントリ・ポイントの名前を指定しなければならない。
3. フェッチされる側のコードのいずれかをコンパイルするために RENT コンパイラー・オプションが使用された場合、そのモジュールは DLL としてリンクする必要があります。
4. NORENT コンパイラー・オプションがフェッチ・コードのコンパイルに使用された場合、そのモジュールの ENTRY が OPTIONS(RENT) で宣言されていなければ、フェッチされたモジュールは以下の条件の少なくとも 1 つを満たさなければなりません：
   - MAIN モジュールである。
   - NORENT コードのみで構成されている。
   - これは、 NORENT オプションを有効にして C または Enterprise PL/I コンパイラでコンパイルされたコードをエントリーポイントとしている。 この場合は、RENT オプションでコンパイルされたコードもモジュールに含まれている可能性がありますが、そのコードの呼び出しはサポートされていません。
5. フェッチコードのコンパイルに RENT コンパイラーオプションが使用された場合、フェッチされる ENTRY は、フェッチモジュール内で OPTIONS(COBOL) または OPTIONS(ASM) として宣言されてはならない。 このような状況でディスクリプタの受け渡しを避けたい場合は、 ENTRY の宣言で OPTIONS(NODESCRIPTOR) 属性を指定する必要がある。
6. Enterprise PL/I ルーチンは自らをフェッチすることはできません。

   複数回ロードされる再入可能で再利用不可能なモジュールの場合、処理の順番は後入れ先出しになる。

   例えば、プログラム A がモジュール LOADMODA をロードし、次にプログラム B を呼び出し、そのプログラム B も LOADMODA をロードし、次に LOADMODA に対して DELETE を発行する場合、削除される LOADMODA のコピーはプログラム B に関連付けられたものです。この時点では、プログラム A に関連付けられた LOADMODA のコピーはまだ存在します。

   言い換えれば、 LOADMODA に対して要求された DELETE は、どのプログラムが要求を出したかに関係なく、最後にロードされたコピーをリリースする。

NORENT WRITABLE そのため、ルーチンのプロローグ・コードでは、定数を表すポインタがゼロになります。 NORENT WRITABLE FETCHABLE これは、正しい PL/I セマンティクスを提供しながら、コードの連続的な再利用性を保証する一方で、 NORENT WRITABLE コードにおける TITLE 付き FETCH の使用に制限を課します。 FETCH A TITLE('B') を実行したルーチンが終了して再入力された場合、CALL A ステートメントを実行する前に、 FETCH A TITLE('B') を再実行する必要があります（そうでない場合、 CALL を実行する前に、A の暗黙的な FETCH （ただし、TITLE なし）を実行することになります)。

これらの制約事項の一部を例示するために、以下のサンプル・プログラムを検討してください。

```
*process s list;
  Mainrtn:  Proc Options( main nodescriptor );

    Dcl Fetrtn entry( char(5), char(5) ) ext
               options( fetchable nodescriptor );
    Dcl string1 char(5);
    Dcl string2 char(5);

    on error begin;
      on error system;
      call plidump('H');
    end;

    string1 = 'hello';
    string2 = 'there';
```

```
    put skip list( procname() || ' Before fetch of FETRTN routine' );
    Fetch Fetrtn;

    put skip list( procname() || ' Before call of FETRTN routine' );
    Call Fetrtn( string1, string2 );

    put skip list( procname() || ' Before release of FETRTN routine' );
    Release Fetrtn;

  End ;
```

```
*process list s map options;
  Fetrtn:  proc( parm1, parm2 ) options( fetchable nodescriptor );

   dcl parm1 char(5) byaddr;
   dcl parm2 char(5) byaddr;

   put skip list( procname() || ' ' || parm1 );
   put skip list( procname() || ' ' || parm2 );

  end Fetrtn;
```

上記項目 1 に従い、フェッチされたルーチン FETRTN の PROCEDURE ステートメントには、OPTIONS(FETCHABLE) が指定されている。

上記の項目 3 に従い、ルーチンを DLL にリンクする場合、リンカ・オプション DYNAM=DLL を指定しなければならない。 DLL は、 PDSE または一時データセットにリンクされなければならない（この場合、DSNTYPE=LIBRARY は、 SYSLMOD DD ステートメントで指定されなければならない)。

上記のサンプル・プログラムをコンパイル、リンク、実行するためのサンプル JCL ：

サンプル・フェッチ・プログラムをコンパイル、リンク、実行するためのサンプル JCL

```
//*********************************************************************
//* compile the fetched program
//*********************************************************************
//PLIF  EXEC PGM=IBMZPLI
//STEPLIB  DD  DSN=IBMZ.V6R1M0.SIBMZCMP,DISP=SHR
//         DD  DSN=CEE.SCEERUN2,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*process list s map options;
  Fetrtn:  proc( parm1, parm2 ) options( fetchable nodescriptor );

   dcl parm1 char(5) byaddr;
   dcl parm2 char(5) byaddr;

   put skip list( procname() || ' ' || parm1 );
   put skip list( procname() || ' ' || parm2 );

  end Fetrtn;
//*********************************************************************
//* link the fetched program
//*********************************************************************
//LKEDF EXEC PGM=IEWL,PARM='XREF,LIST,LET,DYNAM=DLL',
//    COND=(7,LT,PLIF),REGION=5000K
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD  DD DSN=my.pdse.dataset(FETRTN),DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(3,1)),
//            DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=*
//SYSDEFSD DD DUMMY
//SYSLIN   DD DSN=&&LOADSET,DISP=SHR
//         DD DDNAME=SYSIN
//SYSIN    DD *
  ENTRY FETRTN
```

```
//*********************************************************************
//* compile the MAIN program
//*********************************************************************
//PLIM EXEC PGM=IBMZPLI,PARM='SOURCE,LIST',REGION=256K
//STEPLIB   DD  DSN=IBMZ.V6R1M0.SIBMZCMP,DISP=SHR
//          DD  DSN=CEE.SCEERUN2,DISP=SHR
```

```
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET2,DISP=(MOD,PASS),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*process s list;
  Mainrtn:  Proc Options( main nodescriptor );

    Dcl Fetrtn entry( char(5) byaddr,
                      char(5) byaddr )
               ext
               options( fetchable nodescriptor );

    Dcl string1 char(5);
    Dcl string2 char(5);

    on error begin;
      on error system;
      call plidump('H');
    end;

    string1 = 'hello';
    string2 = 'there';

    put skip list( procname() || ' Before fetch of FETRTN routine' );
    Fetch Fetrtn;

    put skip list( procname() || ' Before call of FETRTN routine' );
    Call Fetrtn( string1, string2 );

    put skip list( procname() || ' Before release of FETRTN routine' );
    Release Fetrtn;

  End ;
//*********************************************************************
//* link the MAIN program
//*********************************************************************
//LKEDM EXEC PGM=IEWL,PARM='XREF,LIST,LET,DYNAM=DLL',
//    COND=(9,LT,PLIM),REGION=5000K
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD  DD DSN=my.pdse.dataset(MAINRTN),DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(3,1)),
//            DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=*
//SYSDEFSD DD DUMMY
//SYSLIN    DD DSN=&&LOADSET2,DISP=SHR
//         DD DDNAME=SYSIN
//*********************************************************************
//* invoke the MAIN program
//*********************************************************************
//GO      EXEC PGM=MAINRTN,COND=(7,LT),PARM='XPLINK(ON)/'
//STEPLIB  DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DSN=my.pdse.dataset,DISP=SHR
//SYSPRINT DD  SYSOUT=*
```

Sample JCL to compile, link, and run sample fetch program (continued)

31 ビット・アプリケーションでの PL/I MAIN ルーチンのフェッチ

Enterprise PL/I アプリケーションでは、 PL/I MAIN プログラムをフェッチすることもできる。 PL/I MAIN プログラムの FETCH が発生すると、子エンクレーブが作成される。

以下の規則に従う必要があります。

- 取得した MAIN プログラムに、パラメータ文字列でランタイム・オプションを渡すことはできない。
- SYSTEM(MVS) コンパイラ・オプションが指定されている場合、任意のパラメー タ・リストを渡すことができますが、パラメータが単一の CHAR VARYING 文字列以外の場合、コンパイラは MAIN ルーチンに警告メッセージを表示します。
- フェッチ・プログラムのフェッチされた MAIN ルーチンの ENTRY 宣言では、 OPTIONS(ASM) または OPTIONS(NODESCRIPTOR) を指定してはならない。
- ランタイム・オプションを解析しようとすると、無効なランタイム・オプションに関する LE 。 フェッチされた MAIN ルーチンで NOEXECOPS が指定された場合、渡された文字列は実行時オプションのために解析されない。

- フェッチされた MAIN プログラムにパラメータが渡されない場合、フェッチされたプログラムは、フェッチされた MAIN ルーチンの ENTRY 宣言で OPTIONS(LINKAGE(SYSTEM)) を指定するか、DEFAULT(LINKAGE(SYSTEM)) でコンパイルされなければならない。

例

以下は、 PL/I フェッチされた MAIN プログラムのサンプルである：

```
FMAIN: proc(parm) options(main,noexecops );
  DCL parm char(*) var;
  DCL SYSPRINT print;
  DCL PLIXOPT CHAR(11) VAR INIT('RPTOPTS(ON)')
      STATIC EXTERNAL;
  Put skip list("FMAIN parm: "|| parm);
  Put skip list("FMAIN finished ");
End FMAIN;
```

以下は、別の PL/I MAIN プログラムをフェッチする PL/I MAIN プログラムのサンプルである：

```
MainFet: Proc Options(main);
    Dcl Parm char(1000) var;
    Dcl SYSPRINT print;
    Dcl Fmain entry(char(*) var) ;
    Put skip list("MainFet: start ");
    Parm = 'local-parm';
    Put skip list("MainFet parm: "|| Parm);
    Fetch Fmain;
    Call Fmain(Parm);
    Release Fmain;
    Put skip list("MainFet:testcase finished ");
End;
```

31 ビット・アプリケーションで z/OS C ルーチンをフェッチ

NORENT オプションが指定されていない限り、 z/OS C ルーチンをフェッチするルーチンの ENTRY 宣言では、 OPTIONS(COBOL) または OPTIONS(ASM)を指定してはならない。これらは、 DLLs としてリンクされていない COBOL または ASM ルーチンのみに指定されるべきである。

z/OS C のドキュメントには、 z/OS C DLL をコンパイルしてリンクする方法が記載されています。

31 ビット・アプリケーションでアセンブラー・ルーチンをフェッチ

NORENT オプションが指定されていない限り、アセンブラ・ルーチンをフェッチするルーチンの ENTRY 宣言には OPTIONS(ASM) を指定しなければならない。

データのみのアセンブラ・モジュールをフェッチする場合は、 FETCH A SET(P) を使用して、ポインタ P が正しく設定されていることを確認する必要があります。 しかし、もしフェッチするモジュールとフェッチされるモジュールの AMODEs が両方とも AMODE(31)でない場合、ポインタ P はモジュールそのものではなく、AMODE 変更するグルーコードを指すことになることに注意してください。 AMODE の違いを考慮する場合は、アセンブラ・モジュールにテーブルのアドレスを返させることを推奨する。

TSO/E で MAIN

SYSTEM(MVS)オプションで MAIN プログラムをコンパイルすると、 TSO CALL コマンドを使用するか、TSO コマンドプロセッサとしてプログラムを呼び出すことができる。 ランタイム・オプションとパラメーターの両方を、MVS バッチ配下と同じ方法で渡すことができます。

例えば、 MAIN プログラム TSOARG1 が、 `userid.TEST.load` というデータセットのメンバーとしてリンク編集された場合、次のように呼び出すことができる：

```
   CALL TEST(TSOARG1) 'RPTSTG(ON),TRAP(ON)/THIS IS MY ARGUMENT'
or
   CALL TEST(TSOARG1) '/THIS IS MY ARGMENT'
or
   TSOARG1 TRAP(ON)/THIS IS MY ARGUMENT
```

```
or
  TSOARG1 /THIS IS MY ARGUMENT
```

注 : CALL ステートメントを使用せずにプログラムを実行するには、TSOARG1 (`userid.TEST.load`) を含むデータセットが標準の TSO プログラム検索リストになければならない。 これは TSO ：

```
    TSOLIB ACTIVATE DSN('userid.TEST.load')
```

しかし、 SYSTEM(TSO) オプションをつけて MAIN プログラムをコンパイルすると、 Command Processor Parameter List (CPPL) へのポインタが渡される。 この場合、 NOEXECOPS 。 あなたのプログラムは TSO コマンドとして呼び出すことができるが、 TSO CALL ステートメントによって呼び出すことはできない。 次の例を参照してください。

```
    TSOARG2 This is my argument
```

175 ページの『#unique_183/unique_183_Connect_42_cpplxmp』 のプログラムは、 SYSTEM(TSO) インターフェイスを使用して、 CPPL からプログラムの引数をアドレス指定し、表示する。

SYSTEM(STD) オプション使用時に TSO の下にある CPPL からプログラム引数を表示するサンプルプログラム

```
*process system(tso);
 tsoarg2: proc (cppl_ptr) options(main);
   dcl cppl_ptr pointer;
   dcl 1 cppl based(cppl_ptr),
     2 cpplcbuf   pointer,
     2 cpplupt    pointer,
     2 cpplpscb   pointer,
     2 cpplect    pointer;
   dcl 1 cpplbuf based(cpplcbuf),
     2 len    fixed bin(15),
     2 offset fixed bin(15),
     2 argstr char(1000);
   dcl my_argument char(1000) varying;
   dcl my_argument_len fixed bin(31);
   dcl length builtin;

   my_argument_len = len - offset - 4;
   if my_argument_len = 0 then
     my_argument = '';
   else
     my_argument =  substr(argstr,offset + 1, my_argument_len);
   display('Program args: ' || my_argument);
 end tsoarg2;
```

MAIN プログラムがコマンドとして呼び出される場合でも、 CALL によって呼び出される場合でも、 PLIXOPT 文字列を通して常に実行時オプションを指定することができる。

MAIN 。 z/OS UNIX

z/OS UNIX では、 SYSTEM(MVS) または SYSTEM(OS) オプションを使用して、 MAIN プログラムをコンパイルすることができます。ただし、プログラムに渡されるパラメータの数と形式は、使用するオプションによって異なります。

SYSTEM(MVS)オプションで MAIN プログラムをコンパイルすると、プログラムは通常通り、起動時に指定されたパラメータを含む 1 つの CHARACTER VARYING 文字列を渡される。

ただし、 SYSTEM(OS) オプションを指定して MAIN プログラムをコンパイルすると、 z/OS UNIX マニュアルで指定されている 7 つのパラメータが渡されます。 これら 7 つのパラメーターには、以下が含まれます。

- 引数カウント (第 1 引数として実行可能な名前を含む)
- 複数の引数のアドレスを含む、単一の配列のアドレス
- ヌル終了文字ストリングである複数の引数のアドレスを含む、単一の配列のアドレス
- 環境変数セットのカウント
- 複数の環境変数の長さのアドレスを含む、単一の配列のアドレス
- ヌル終了文字ストリングである複数の環境変数のアドレスを含む、単一の配列のアドレス

176 ページの『#unique_318/unique_318_Connect_42_ussmain』 のプログラムは、 SYSTEM(OS) インターフェイスを使用して、個々の引数と環境変数をアドレス指定し、表示する。

z/OS UNIX 引数と環境変数を表示するサンプルプログラム

```
*process display(std) system(os);

 sayargs:
  proc(argc, pArgLen, pArgStr, envc, pEnvLen, pEnvStr, pParmSelf)
  options( main, noexecops );

   dcl argc                fixed bin(31) nonasgn byaddr;
   dcl pArgLen             pointer nonasgn byvalue;
   dcl pArgStr             pointer nonasgn byvalue;
   dcl envc                fixed bin(31) nonasgn byaddr;
   dcl pEnvLen             pointer nonasgn byvalue;
   dcl pEnvStr             pointer nonasgn byvalue;
   dcl pParmSelf           pointer nonasgn byvalue;

   dcl q(4095)             pointer based;
   dcl bxb                 fixed bin(31) based;
   dcl bcz                 char(31) varz based;

   display( 'argc = ' || argc );
   do jx = 1 to argc;
     display( 'pargStr(jx) =' || pArgStr->q(jx)->bcz );
   end;
   display( 'envc = ' || envc );
   do jx = 1 to envc;
     display( 'pEnvStr(jx) =' || pEnvStr->q(jx)->bcz );
   end;

 end;
```

第 6 章 64 ビット・プログラムに対するリンク・エディットおよび実行

LP(64)でコンパイルした後、64 ビット・プログラムは、 Language Environment ランタイム・ライブラリへの参照と同様に、互いに未解決の参照を含む 1 つまたは複数のオブジェクト・モジュールで構成されます。 これらの参照は、リンク・エディット時 (静的) または実行時 (動的) に 解決されます。

したがって、PL/I プログラムのコンパイルが終わった後で次にとるべきステップは、そのプログラムをテスト・データを用いてリンクして実行し、予想どおりの結果が出るかどうか確認することです。

Language Environment は、ユーザーがプログラムを実行するのに必要なランタイム環境とサービスを提供します。 PL/I およびその他すべての Language Environment に準拠した言語プログラムのリンクと実行については、「*z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*」を参照してください。 既存の PL/I プログラムから Language Environment への移行については、「*Enterprise PL/I for z/OS* コンパイラーおよびランタイム 移行ガイド」を参照してください。

64 ビット・プログラムに関するリンク・エディットの考慮事項

64 ビットコードのバインド手順は、31 ビットコードの従来のバインド手順とはかなり違うことに注意してください。

64 ビット・プログラムでバインダーを使用

バインダーの出力を PDSE データセットに入れる必要があります。

DYNAM(DLL) 、 AMODE(64) をバインドステップのパラメータとして指定する必要がある。 以下のオプションも強くお勧めします。

```
CASE(MIXED), RMODE(ANY)
```

LE 2.4 （またはそれ以降）のデータセットの接頭辞が CEE の場合、バインドステップでは次のようにする

- SYSLIB には以下を含める必要があります

```
//          DD DISP=SHR,DSN=CEE.SCEEBND2
```

- SYSLIB には、SCEELKED または 31 ビット・リンク・エディット・データ・セットを含めることはできません。
- SYSLIN には以下を含める必要があります

```
//          DD DISP=SHR,DSN=CEE.SCEELIB(IBMPQV11)
//          DD DISP=SHR,DSN=CEE.SCEELIB(CELQS003)
```

SIBMZPRC の PROC IBMQCB または IBMQCBG は、64 ビット・プログラムの完全なバインド・ステップを含む例として使用できます。

64 ビットプログラムでの ENTRY カードの使用

フェッチされるモジュールで、 Enterprise PL/I ルーチンをエントリー・ポイントに持つモジュールをビルドする場合、 ENTRY カードは、 CELQSTRT を指定しなければならない。

64 ビット・プログラムに関する実行時の考慮事項

プログラム初期化ルーチンに渡されるパラメーターとしてランタイム・オプションを指定できます。 また、PLIXOPT 変数でランタイム・オプションを指定することもできる。 また、 PLIXOPT 変数を使ってランタイム・オプションを指定し、プログラムを再コンパイルすることで、既存のプログラムを変更すれば、パフォーマンスの観点からも有益であることがわかるかもしれない。

© Copyright IBM Corp. 1999, 2025

PLIXOPT の使い方については *z/OS Language Environment Programming Guide*.

端末の入出力を簡素化するため、端末に割り当てられるストリーム・ファイル には各種規則が取り入れられています。 次の 3 つの領域が影響を受けます。

1. PRINT ファイルのフォーマット
2. 自動プロンプト機能
3. 入力のスペーシングと句読法の規則

注 : 端末でのレコード入出力にはプロンプトその他の機能は 提供されていません。したがって、端末から、または端末への伝送には ストリーム入出力を使用することを強くお勧めします。

SYSPRINT 64 ビットプログラムへの配慮

64 ビット・プログラムの場合、 SYSPRINT は C 標準出力ファイルと同じである。 また、共有 SYSPRINT 。

SYSPRINT の LINESIZE は、132（C 言語で許される最大の値）より大きくすることはできない。

LP(64)でコンパイルすると、 STDSYS オプションが有効になります。 これは、64 ビット・ストリーム・ファイルの場合、下位 16 進値が '00'x、'0C'x から '0F'x まで、および '15'x のデータは、'4B'x に変換されることを意味します。

STDSYS オプションに関する詳細および制限については、以下を参照してください。

- 88 ページの『STDSYS』
- 3 ページの『第 1 章 コンパイラー・オプションと機能の使用』
- *Enterprise PL/I for z/OS Compiler and Run-time Migration Guide (DB2* コンパイラーおよびランタイムマイグレーション・ガイド*)* の 印刷不能文字を伴う STREAM 入出力

64 ビット・アプリケーションのルーチンで FETCH

Enterprise PL/I では、PL/I、C、またはアセンブラーによってコンパイルされたルーチンをフェッチできます。

64 ビット・アプリケーションで Enterprise PL/I ルーチンをフェッチ

フェッチされたモジュールは、 FETCH, I/O, ALLOCATE、 FREE 操作を実行できる。 ただし、依然として制限がいくつか適用されます。

フェッチされたモジュールは次の操作を実行できます。

- 他のモジュールをフェッチする。
- 任意の PL/I ファイルに入出力操作を実行する。 このファイルは、フェッチされたモジュール、メイン・モジュール、または他のフェッチされたモジュールによりオープンできます。
- ALLOCATE そして、 自身の 変数。 FREE CONTROLLED

しかし、フェッチの対象である Enterprise PL/I モジュールに対するいくつかの制限があります。

1. リンク編集ステップで ENTRY カードが提供されない場合は、フェッチされたルーチンの PROCEDURE ステートメントで OPTIONS(FETCHABLE) を指定する必要がある。
2. ENTRY カードは、 CELQSTRT を指定しなければならない。
3. LP(64)では、RENT は事実上常にオンであるため、フェッチされる ENTRY は、フェッチモジュール内で OPTIONS(COBOL) または OPTIONS(ASM) として宣言されてはならない。 このような状況で記述子の受け渡しを避けたい場合は、 ENTRY 宣言で OPTIONS(NODESCRIPTOR) 属性を指定する必要があります。
4. Enterprise PL/I ルーチンは自らをフェッチすることはできません。

64 ビット・アプリケーションで PL/I MAIN ルーチンをフェッチ

64 ビット PL/I MAIN ルーチンは 64 ビット PL/I MAIN ルーチンをフェッチできません。

64 ビット・アプリケーションでアセンブラー・ルーチンをフェッチ

アセンブラ・ルーチンをフェッチするルーチンの ENTRY 宣言には、 OPTIONS(ASM) を指定しなければならない。 アセンブラ・ルーチンは AMODE=64 でリンクされなければならない。 LINKAGE(SYSTEM) を指定する必要がある。

データのみのアセンブラ・モジュールをフェッチする場合は、 FETCH A SET(P) を使用して、ポインタ P が正しく設定されていることを確認する必要があります。

TSO/E で MAIN

SYSTEM(MVS)オプションで MAIN プログラムをコンパイルすると、 TSO CALL コマンドを使用するか、 TSO コマンドプロセッサとしてプログラムを呼び出すことができる。 ランタイム・オプションとパラメーターの両方を、MVS バッチ配下と同じ方法で渡すことができます。

例えば、 MAIN プログラム TSOARG1 が、 `userid.TEST.load` というデータセットのメンバーとしてリンク編集された場合、次のように呼び出すことができる：

```
    CALL TEST(TSOARG1) 'RPTSTG(ON),TRAP(ON)/THIS IS MY ARGUMENT'
  or
    CALL TEST(TSOARG1) '/THIS IS MY ARGMENT'
  or
    TSOARG1 TRAP(ON)/THIS IS MY ARGUMENT
  or
    TSOARG1 /THIS IS MY ARGUMENT
```

注 : CALL ステートメントを使用せずにプログラムを実行するには、TSOARG1 (`userid.TEST.load`) を含むデータセットが標準の TSO プログラム検索リストになければならない。 これは TSO ：

```
    TSOLIB ACTIVATE DSN('userid.TEST.load')
```

ただし、 SYSTEM(TSO) オプションを付けて MAIN プログラムをコンパイルすると、 Command Processor Parameter List (CPPL) へのポインタが渡されます。 この場合、 NOEXECOPS 。 あなたのプログラムは、 TSO コマンドとして呼び出すことはできるが、 TSO CALL ステートメントによって呼び出すことはできない。 パラメーター・リスト内のポインターは 31 ビット・ポインターであることに注意してください。 次の例を参照してください。

```
    TSOARG2 This is my argument
```

SYSTEM(STD)オプション使用時に TSO 下で CPPL からプログラム引数を表示するサンプルプログラムのプログラムは、SYSTEM(TSO) インターフェイスを使用して、CPPL からのプログラム引数をアドレス指定し、表示します。

SYSTEM(STD) オプションを使用したときに TSO のもとで CPPL からプログラム実引数を表示するサンプル・プログラム

```
*process system(tso);
 tsoarg2: proc (cppl_ptr) options(main);
   dcl cppl_ptr pointer;
   dcl 1 cppl based(cppl_ptr),
     2 cpplcbuf   pointer(32),
     2 cpplupt    pointer(32),
     2 cpplpscb   pointer(32),
     2 cpplect    pointer(32);
   dcl 1 cpplbuf based(cpplcbuf),
     2 len    fixed bin(15),
     2 offset fixed bin(15),
     2 argstr char(1000);
   dcl my_argument char(1000) varying;
   dcl my_argument_len fixed bin(31);
   dcl length builtin;

   my_argument_len = len - offset - 4;
   if my_argument_len = 0 then
     my_argument = '';
   else
     my_argument =  substr(argstr,offset + 1, my_argument_len);
```

```
    display('Program args: ' || my_argument);
  end tsoarg2;
```

MAIN プログラムがコマンドとして呼び出される場合でも、 CALL によって呼び出される場合でも、 PLIXOPT 文字列を通して常に実行時オプションを指定することができる。

MAIN 。 z/OS UNIX

z/OS UNIX では、 SYSTEM（MVS ）または SYSTEM(OS) オプションを使用して、 MAIN プログラムをコンパイルできます。ただし、プログラムに渡されるパラメータの数と形式は、使用するオプションによって異なります。

SYSTEM(MVS)オプションを使って MAIN プログラムをコンパイルすると、通常通り、起動時に指定されたパラメータを含む CHARACTER VARYING 文字列が渡される。

ただし、 MAIN のプログラムを SYSTEM(OS) オプションでコンパイルすると、 z/OS UNIX のマニュアルで指定されている 7 つのパラメータが渡される。 これら 7 つのパラメーターには、以下が含まれます。

- 引数カウント (第 1 引数として実行可能な名前を含む)
- 複数の引数のアドレスを含む、単一の配列のアドレス
- ヌル終了文字ストリングである複数の引数のアドレスを含む、単一の配列のアドレス
- 環境変数セットのカウント
- 複数の環境変数の長さのアドレスを含む、単一の配列のアドレス
- ヌル終了文字ストリングである複数の環境変数のアドレスを含む、単一の配列のアドレス

180 ページの『#unique_329/unique_329_Connect_42_pg64ussmain』 のプログラムは、SYSTEM(OS) インターフェイスを使用して、個々の引数と環境変数をアドレス指定し、表示する。

z/OS UNIX 引数と環境変数を表示するサンプルプログラム：

```
*process display(std) system(os);

 sayargs:
  proc(argc, pArgLen, pArgStr, envc, pEnvLen, pEnvStr, pParmSelf)
  options( main, noexecops );

    dcl argc                fixed bin(31) nonasgn byaddr;
    dcl pArgLen             pointer nonasgn byvalue;
    dcl pArgStr             pointer nonasgn byvalue;
    dcl envc                fixed bin(31) nonasgn byaddr;
    dcl pEnvLen             pointer nonasgn byvalue;
    dcl pEnvStr             pointer nonasgn byvalue;
    dcl pParmSelf           pointer nonasgn byvalue;

    dcl q(4095)             pointer based;
    dcl bxb                 fixed bin(31) based;
    dcl bcz                 char(31) varz based;

    display( 'argc = ' || argc );
    do jx = 1 to argc;
      display( 'pargStr(jx) =' || pArgStr->q(jx)->bcz );
    end;
    display( 'envc = ' || envc );
    do jx = 1 to envc;
      display( 'pEnvStr(jx) =' || pEnvStr->q(jx)->bcz );
    end;

  end;
```

第 7 章 64 ビット・アプリケーションを開発する場合の考慮事項

Enterprise PL/I を使用すれば、31 ビット/64 ビット・アプリケーションを開発できます。 ご使用のアプリケーションで 64 ビット環境がサポートされるようにするには、必要に応じてコードを調整しなければならない場合があります。 このセクションでは、開発時およびコンパイル時に考慮すべき事項について説明します。

コンパイラー・オプションを使用して 64 ビット・アプリケーションをビルド

64 ビット・アプリケーション用にコードをコンパイルするには、LP(64) コンパイラー・オプションを使用する必要があります。 LP(64)では、いくつかのコンパイラー・オプションやサブ・オプションがサポートされておらず、コンパイル時に無視されるオプションやサブ・オプションがあることに注意しなければならない。

LP(64)でコードをコンパイルする場合、以下のコンパイラー・オプションに注意してください：

11 ページの『BACKREG』
このオプションは無視されます。

14 ページの『CHECK』
STORAGE サブオプションは、LP(64)ではサポートされていない。 つまり、以下の組み込み関数は使用できません。

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

14 ページの『CMPAT』
このオプションは無視される。事実上、 LP(64)では、 CMPAT(V3) は常にオンである。

16 ページの『COMMON』
このオプションは無視されます。

23 ページの『DEFAULT』
LP(64)では、以下のサブオプションは無視される：

- LINKAGE

68 ページの『RENT』
このオプションは無視される。事実上、 LP(64)では、 RENT は常にオンである。

88 ページの『STDSYS』
このオプションは無視される。事実上、 LP(64)では、 STDSYS は常にオンである。

97 ページの『WRITABLE』
このオプションは無視されます。

関連情報
51 ページの『LP』
LP オプションは、コンパイラが 31 ビット・コードと 64 ビット・コードのどちらを生成するかを指定する。 また、 POINTER 、 HANDLE 、関連する変数のデフォルトサイズも決定する。

LP(64)で属性 HANDLE 、 POINTER

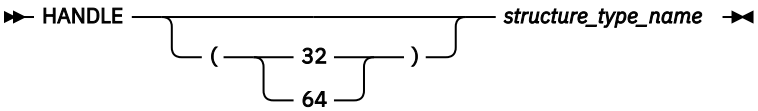
LP(64)下の HANDLE と POINTER のデフォルトのサイズとアライメントは、 LP(32)下のデフォルトとは異なる。

© Copyright IBM Corp. 1999, 2025

HANDLE 属性

LP(32)の場合、デフォルトは HANDLE(32)。 LP(64)の場合、デフォルトは HANDLE(64)。

Syntax



HANDLE(32)は 4 バイトの大きさで、デフォルトではフルワード整列されている。

HANDLE(64)は 8 バイトサイズで、デフォルトではダブルワードでアラインされている。

HANDLE(64)は LP(64)の下でのみ有効である。

HANDLE （32） を HANDLE （64） に割り当てることは常に有効である。その逆は、 HANDLE （64） の最初の 4 バイトがゼロの場合のみ有効である。

ハンドルのサイズと位置合わせが変更されたために、ハンドルを含む構造体で埋め込みバイトが使用されることがあります。

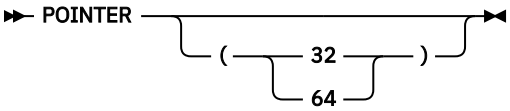
関連情報

51 ページの『LP』
LP オプションは、コンパイラが 31 ビット・コードと 64 ビット・コードのどちらを生成するかを指定する。 また、 POINTER 、 HANDLE 、関連する変数のデフォルトサイズも決定する。

POINTER 属性

LP(32) では、デフォルトは POINTER(32) である。LP(64) では、デフォルトは POINTER(64) である。

Syntax



POINTER(32)は 4 バイトの大きさで、デフォルトではフルワード整列されている。

POINTER(64)は 8 バイトサイズで、デフォルトではダブルワードでアラインされている。

POINTER(64)は LP(64)の下でのみ有効である。

POINTER （32） を POINTER （64） に割り当てることは常に有効である。その逆は、 POINTER （64） の最初の 4 バイトがゼロの場合のみ有効である。

ポインターのサイズと位置合わせが変更されたために、ポインターを含む構造体で埋め込みバイトが使用されることがあります。

関連情報

51 ページの『LP』
LP オプションは、コンパイラが 31 ビット・コードと 64 ビット・コードのどちらを生成するかを指定する。 また、 POINTER 、 HANDLE 、関連する変数のデフォルトサイズも決定する。

LP(64) で ENTRY 変数を使用

LP(64) では、 LIMITED 属性の有無にかかわらず、すべての ENTRY 変数のサイズは 8 バイトで、デフォルトではダブルワードにアラインされています。 したがって、 ENTRY 変数を含む構造体は、パディング・バイトを持つことになる。

LP(64) で組み込み関数を使用

64 ビット・アプリケーションを開発するときは、一部の組み込み関数では引数の型と戻りの型が異なることを理解しておく必要があります。

LP(64)の下で FIXED BIN(63)の値を返す組み込み関数

以下の組み込み関数は、LP(64)の下では FIXED BIN(63)の値を返しますが、LP(32)の下では FIXED BIN(31)の値を返します。

- AUTOMATIC
- AVAILABLEAREA
- BASE64DECODE
- BASE64DECODE8
- BASE64DECODE16
- BASE64ENCODE
- BASE64ENCODE8
- BASE64ENCODE16
- CURRENTSTORAGE
- FILEID
- FILEREAD
- FILESEEK
- FILETELL
- HEXENCODE
- HEXENCODE8
- HEXDECODE
- HEXDECODE8
- JSONGETARRAYEND
- JSONGETARRAYSTART
- JSONGETCOLON
- JSONGETCOMMA
- JSONGETMEMBER
- JSONGETOBJECTEND
- JSONGETOBJECTSTART
- JSONGETVALUE
- JSONPUTARRAYEND
- JSONPUTARRAYSTART
- JSONPUTCOLON
- JSONPUTCOMMA
- JSONPUTMEMBER
- JSONPUTOBJECTEND
- JSONPUTOBJECTSTART
- JSONPUTVALUE
- JSONVALID
- MEMCONVERT
- MEMCU12
- MEMCU14
- MEMCU21
- MEMCU24
- MEMCU12

- MEMCU41
- MEMCU42
- MEMINDEX
- MEMSEARCH
- MEMSEARCHR
- MEMVERIFY
- MEMVERIFYR
- LOCATION
- LOCSTG
- OFFSETDIFF
- POINTERDIFF
- STORAGE
- WSCOLLAPSE
- WSCOLLAPSE16
- WSREPLACE
- WSREPLACE16
- XMLCHAR
- XMLSCRUB
- XMLSCRUB16

整数引数が LP(64) の下で FIXED BIN(63) に変換される組込み関数

以下の組み込み関数には、ストレージの断片のサイズを表す 1 つ以上の引数があります。 これらの引数は、必要であれば LP(64)の下で FIXED BIN(63)に変換される。

- ALLOCATE
- BASE64DECODE
- BASE64DECODE8
- BASE64DECODE16
- BASE64ENCODE
- BASE64ENCODE8
- BASE64ENCODE16
- CHECKSUM
- COMPARE
- FILEWRITE
- HEXENCODE
- HEXENCODE8
- HEXDECODE
- HEXDECODE8
- JSONGETARRAYEND
- JSONGETARRAYSTART
- JSONGETCOLON
- JSONGETCOMMA
- JSONGETMEMBER
- JSONGETOBJECTEND

- JSONGETOBJECTSTART
- JSONGETVALUE
- JSONPUTARRAYEND
- JSONPUTARRAYSTART
- JSONPUTCOLON
- JSONPUTCOMMA
- JSONPUTMEMBER
- JSONPUTOBJECTEND
- JSONPUTOBJECTSTART
- JSONPUTVALUE
- JSONVALID
- MEMCONVERT
- MEMCU12
- MEMCU14
- MEMCU21
- MEMCU24
- MEMCU41
- MEMCU42
- MEMINDEX
- MEMSEARCH
- MEMSEARCHR
- MEMVERIFY
- MEMVERIFYR
- PLIASCII
- PLIEBCDIC
- PLIFILL
- PLIMOVE
- PLIOVER
- PLISAXA
- PLISAXB
- PLISAXC
- PLISAXD
- PLITRAN11
- PLITRAN12
- PLITRAN21
- PLITRAN22
- WSCOLLAPSE
- WSCOLLAPSE16
- WSREPLACE
- WSREPLACE16
- XMLCHAR
- XMLSCRUB
- XMLSCRUB16

LP(64) でサポートされない組み込み関数

LP(64) では、 CHECK コンパイラ・オプションの STORAGE サブオプションがサポートされていないため、以下の組込み関数は使用できません：

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

注 : 64 ビットでは、 PLISRT の組み込み関数は、LE V2R2 以降でのみサポートされています。

SQL プログラムの考慮事項

同じソースを使用して 32 ビット・アプリケーションおよび 64 ビットのアプリケーションを開発するには、EXEC SQL INCLUDE ステートメントを使用して sqlda または sqlda2 を宣言すること、および sqlvar のサイズとして stg(sqlvar(1)) を使用するように sqldabc フィールドを設定することをお勧めします。 ご使用の SQL プログラムがこの推奨事項に従っていない場合は、64 ビット・アプリケーションを開発するためにコードを変更しなければならない可能性があります。

必要に応じて、既存の 32 ビット SQL プログラムを、その振る舞いを変えずに変更できます。 こうした変更により、プログラムの移植がいっそう簡単になる可能性があります。

sqlda の宣言

EXEC SQL INCLUDE sqlda ステートメントで sqlda が宣言されていない場合は、sqllen フィールドの後に次のフィールドを追加して sqlda の宣言を変更する必要があります。

```
       char( length(hex(sysnull()))/2 - 4 ),
```

例えば、プログラムで sqlda 構造体が以下のように宣言されているとします。

```
dcl
      1 fsqlda based(fsqldaptr),
        2 sqldaid      char(8),
        2 sqldabc      fixed bin(31),
        2 sqln         fixed bin(15),
        2 sqld         fixed bin(15),
        2 sqlvar(fsqlsize refer(sqln)),
          3 sqltype    fixed bin(15),
          3 sqllen     fixed bin(15),
          3 sqldata    pointer,
          3 sqlind     pointer,
          3 sqlname    char(30) var;
```

この宣言を以下のように変更する必要があります。

```
dcl
      1 fsqlda based(fsqldaptr),
        2 sqldaid      char(8),
        2 sqldabc      fixed bin(31),
        2 sqln         fixed bin(15),
        2 sqld         fixed bin(15),
        2 sqlvar(fsqlsize refer(sqln)),
          3 sqltype    fixed bin(15),
          3 sqllen     fixed bin(15),
          3 *          char( length(hex(sysnull()))/2 - 4 ),
          3 sqldata    pointer,
          3 sqlind     pointer,
          3 sqlname    char(30) var;
```

sqlda2 の宣言

EXEC SQL INCLUDE sqlda2 ステートメントで sqlda2 が宣言されていない場合は、sqlrsvdl フィールドの後に次のフィールドを追加して sqlda2 の宣言を変更する必要があります。

```
       char( length(hex(sysnull())) - 8 ),
```

例えば、プログラムで sqlda2 構造体が以下のように宣言されているとします。

```
dcl
      1 fsqlda2 based(fsqldaptr),
        2 sqldaid2     char(8),
        2 sqldabc2     fixed bin(31),
        2 sqln2        fixed bin(15),
        2 sqld2        fixed bin(15),
        2 sqlvar2(fsqlsize refer(sqln2)),
          3 sqlbiglen,
            4 sqllongl fixed bin(31),
            4 sqlrsvdl fixed bin(31),
          3 sqldatal   pointer,
          3 sqltname   char(30) var;
```

この宣言を以下のように変更する必要があります。

```
dcl
      1 fsqlda2 based(fsqldaptr),
        2 sqldaid2     char(8),
        2 sqldabc2     fixed bin(31),
        2 sqln2        fixed bin(15),
        2 sqld2        fixed bin(15),
        2 sqlvar2(fsqlsize refer(sqln2)),
          3 sqlbiglen,
            4 sqllongl fixed bin(31),
            4 sqlrsvdl fixed bin(31),
          3 *          char( length(hex(sysnull())) - 8 ),
          3 sqldatal   pointer,
          3 sqltname   char(30) var;
```

設定 sqldabc

sqldabc フィールドが sqlvar のサイズとして 44 に設定されている場合は、代わりに stg(sqlvar(1)) を使用するようにそのフィールドを設定する必要があります。

例えば、ステートメント sqlda.sqldabc = 16 + (44 * sqlda.sqld); は sqlda.sqldabc = 16 + (stg(sqlda.sqlvar(1)) * sqlda.sqld);に変更する必要があります。

次のようにコーディングします。

```
SQLDABC = LEN_SQLDA + SQLN * LEN_SQLVAR;
```

LEN_SQLVAR 、以下のように宣言を変更しなければならない：

- **オリジナル宣言** DCL LEN_SQLVAR FIXED BIN(15) VALUE(44);
- **更新宣言** DCL LEN_SQLVAR FIXED BIN(15) VALUE(STG(SQLDA.SQLVAR(1)));

DCL LEN_SQLVAR FIXED BIN(15) VALUE(44);

31 ビット・ルーチンとのやり取り

64 ビット LE 対応コードから 31 ビット LE 対応コードを呼び出すことができ、その逆も可能です。 詳細は、 373 ページの『第 19 章 AMODE 31 と AMODE 64 コードを混ぜる』を参照してください。

非言語環境プログラム・ベース の 31 ビット・ルーチンとのやり取りを簡単に行えるように、31 ビット・ルーチンのロード、呼び出し、および解放を行うための汎用インターフェースが提供されています。 ロードされたモジュールのエントリー・ポイント・アドレスを取得するための関数も提供されています。

IBMPC32I、31 ビット・ルーチンをロードする。 これは、モジュールの名前を入力として保持する 8 バイトの文字ストリングへのポインターを取り、呼び出しおよび解放関数に使用されるファイル・ハンドルを戻します。

```
   dcl <load32>   ext( "_IBMPC32I" )
                 entry( char(8) byaddr
inonly )
                 returns( pointer byvalue )
                 options( nodescriptor linkage(optlink) );
```

IBMPC32I、31 ビット・ルーチンを呼び出す。 最初のパラメータは、IBMPC32I の呼び出しから返されたファイル・ハンドルであり、2 番目のパラメータは、31 ビット・ルーチンのユーザー・パラメータ・リストである。 ユーザー・パラメーター・リストは 2 GB 境界より下のストレージに存在していなければならないことに注意してください。 IBMPC32I は、呼び出された 31 ビット・ルーチンからレジスタ 15 の値を返す。

```
   dcl <call32>  ext( "_IBMPC32C" )
                entry( pointer byvalue, pointer byvalue )
                returns( fixed bin(31) byvalue )
                options( nodescriptor linkage(optlink) );
```

IBMPC32T を呼び出して 31 ビット・ルーチンを公開します。 これは、ファイル・ハンドルをパラメーターとして使用します。

```
   dcl <rel32>    ext( "_IBMPC32T" )
                 entry( pointer byvalue )
                 options( nodescriptor linkage(optlink) );
```

IBMPC32I、ロードされたモジュールのエントリーポイントアドレスを取得する。 これは、ファイル・ハンドルをパラメーターとして使用し、ロードされたモジュールのエントリー・ポイント・アドレスを返します。 このアドレスに直接分岐することは無効であることに注意してください。

```
   dcl <epa32>   ext( "_IBMPC32E" )
                entry( pointer byvalue )
                returns( pointer byvalue )
                options( nodescriptor linkage(optlink) );
```

次のコード・フラグメント例では、上記のインターフェースの使用方法を示したものです。 DSNALI は 31 ビット Db2 機能です。

```
    ....

    dcl load32      ext( "_IBMPC32I" )
                         entry( char(8) byaddr inonly )
                         returns( pointer byvalue )
                         options( nodescriptor );

    dcl call32     ext( "_IBMPC32C" )
                       entry( pointer byvalue, pointer byvalue )
                       returns( fixed bin(31) byvalue )
                       options( nodescriptor );

    dcl  file_pointer   pointer;
    dcl  plist_pointer pointer;
    dcl 1 dsnali_plist based(plist_pointer),
    3 args_list(10)        pointer(32),
    3 content union,
    5 connect,
    7 functioncode  char(12),
    7 subsystemid  char(4),
    7 tecb              fixed bin(31),
    7 secb             fixed bin(31),
    7 ribpointer       ptr(32),
    7 returncode     fixed bin(31),
    7 reasoncode    fixed bin(31),
    5 open,
    7 functioncode  char(12),
    7 subsystemid   char(4),
    7 planname       char(8),
    7 returncode     fixed bin(31),
    7 reasoncode   fixed bin(31);
```

```
 dcl lastargflag bit(1) based;

 ....

plist_pointer = alloc31( stg(dsnali_plist) );
file_pointer = load32( "DSNALI" );

/* set up argument list for CONNECT function for DSNALI call */
args_list( 1 ) = addr( connect.functioncode );
args_list( 2 ) = addr( connect.subsystemid );
args_list( 3 ) = addr( connect.tecb );
args_list( 4 ) = addr( connect.secb );
args_list( 5 ) = addr( connect.ribpointer );
args_list( 6 ) = addr( connect.returncode ) ;
args_list( 7 ) = addr( connect.reasoncode ) ;

/* mark last argument */
addr( args_list( 7 ) ).lastargflag = '1'b;

/* set up values for CONNECT function parameters */
connect.functioncode = 'CONNECT';
connect.subsystemid = 'DB2S';
connect.tecb = 0;
connect.secb = 0;
connect.ribpointer = sysnull;
connect.returncode = 0;
connect.reasoncode = 0;

/* invoke DSNALI with CONNECT function */
rc = call32( file_pointer, plist_pointer );

...

/* set up argument list for OPEN function for DSNALI call */
args_list( 1 ) = addr( open.functioncode );
args_list( 2 ) = addr( open.subsystemid );
args_list( 3 ) = addr( open.planname );
args_list( 4 ) = addr( open.returncode );
args_list( 5 ) = addr( open.reasoncode );

/* mark last argument */
addr( args_list( 5 ) ).lastargflag = '1'b;

/* set up values for OPEN function parameters */
open.functioncode = 'OPEN';
open.subsystemid = 'DB2S';
open.planname = 'TESTCASE';
open.returncode = 0;
open.reasoncode = 0;

/* invoke DSNALI with OPEN function */
rc = call32( file_pointer, plist_pointer ) ;

 ....
```

第 2 部 入出力機能の使用

© Copyright IBM Corp. 1999, 2025

第 8 章 データ・セットとファイルの使用

本章では、ファイルを割り振り、ご使用のプログラム内で既知のファイルにデータ・セットを関連付ける方法について説明します。 また、主要な 5 つのタイプのデータ・セットを概説し、そのデータ・セットをどのように編成するのか、およびそのデータ・セットにどのようにアクセスするのかについて説明します。この章は、ファイルやデータ・セットの特性をいくつか指定する方法を学習するときに役立ちます。

PL/I プログラムは、レコード と呼ばれる情報単位の 処理と送信を行います。 レコードの集まりをデータ・セット と呼びます。 データ・セットは、PL/I プログラムの外部にある情報の物理的な 集まりです。 つまり、データ・セットは、PL/I またはその他の言語で書かれたプログラムや、 オペレーティング・システムのユーティリティー・プログラムによって、 作成、アクセス、または変更することができます。

PL/I プログラムは、ファイル と呼ばれるデータ・セットの シンボルによる表現または論理表現を使って、データ・セット内の情報を認識 したり処理したりします。

注 : INDEXED は を意味し、バッチでのみサポートされる。 VSAM

注 : 領域データ・セットは、64 ビット・プログラムではサポートされません。

ファイルの割り振り

シーケンシャル、 VSAM, REGIONAL(1)、 zFS など、どのタイプのファイルでも、以下の方法で外部名を定義できる：

- MVS または TSO ：
  - JCL (ジョブ制御言語) の *DD* 名
  - 環境変数名
  - OPEN 文の TITLE オプション
- z/OS UNIX システム・サービス 環境の場合:
  - 環境変数名
  - OPEN 文の TITLE オプション

バッチ環境の zFS ファイルには、次の規則が適用されます。

- FILEDATA が指定されない場合、デフォルト設定は TEXT となる。
- レコード・フォーマットまたは ENVIRONMENT オプションが指定されていない場合、デフォルトは LF タイプの V となる。
- FILEDATA が BINARY の場合、固定長ファイルのみ有効。 エラーメッセージ MSGIBM0210S は、FILEDATA=BINARY が可変長ファイルで指定された場合に発行される。
- 可変長ファイルの場合、ファイルは TYPE=LF であると仮定される。つまり、レコードは LF (x'15') 文字で区切られる。
- 区切り文字を含むデータファイルの場合は、 PL/I ライブラリが区切り文字を適切に処理できるように、FILEDATA=TEXT を指定する。

動的割り振り

PL/I プログラムは、環境変数または TITLE オプションによって指定された属性を使用して、ファイルを動的に割り振ります。

PL/I 動的割り振りを使用するには、DSN() 形式 (MVS データ・セットの場合)、または PATH() 形式 (zFS ファイルの場合) を使用してファイル名を指定する必要があります。 大/小文字が区別される PATH オプションの **pathname** サブオプションを除き、すべてのオプションと属性は大文字でなければなりません。

© Copyright IBM Corp. 1999, 2025

DSN() オプションに一時データ・セット名を使用しないでください。 ファイル名を指定する以下の例を参照してください。

```
OPEN FILE(FILEIN) TITLE('DSN(USER.FILE.EXT),SHR');
OPEN FILE(FILEIN) TITLE('PATH(/usr/FILE.EXT)');

EXPORT DD_FILE="DSN(USER.FILE.EXT),SHR"
EXPORT DD_FILE="PATH(/usr/FILE.EXT)"
```

いつ動的割り振りを行うかを決定する際には、次の優先規則が適用されます。

1. ファイル用に以下の DD ステートメントの 1 つがある場合は、それが使用されます。 この規則は z/OS UNIX システム・サービス 環境では無効です。
   - JCL DD
   - TSO ALLOCATE
   - ユーザー開始の動的割り振り
2. ファイル用の DD ステートメントがなく、TITLE オプションが OPEN ステートメントに指定されている場合は、ファイルの外部名に関連付けることによって TITLE オプションが使用されます。
3. ファイル用の DD ステートメントがなく、TITLE オプションが指定されていないが、ファイル用の環境変数はある場合は、ファイルの外部名に関連付けることによって環境変数が使用されます。

MVS データ・セットの場合、Enterprise PL/I ランタイムは、OPEN ステートメントごとに環境変数または TITLE オプションの内容を検査します。

- 同じ外部名を持つファイルが、先行する OPEN ステートメントによって動的に割り振られ、その OPEN ステートメント以後に環境変数または TITLE オプションの内容が変更された場合、ランタイムは前の割り振りの解除を動的に行い、現在、環境変数に設定されているか、TITLE オプションに指定されているオプションを使用してファイルを再割り振りします。
- 環境変数または TITLE オプションの内容が変更されなかった場合、ランタイムは割り振り解除または再割り振りを行わずに、現在の割り振りを使用します。

zFS ファイルの場合、DD ステートメントは割り振り解除され、後続の各 OPEN ステートメントで再割り振りされます。

注 :

1. PL/I 動的割り振りに DSN() または PATH() 形式の指定を使用する際、DD ステートメントと、OPEN ステートメントの TITLE オプションの両方がファイルに指定されている場合は、DD ステートメントが使用され、TITLE オプションは無視されます。
2. z/OS UNIX システム・サービス 環境では、ユーザー開始の動的割り振りはサポートされません。 この動的割り振りが試行されると、TITLE オプションまたは環境変数が使用されていても、外部名が使用中であるため、UNDEFINEDFILE 条件が発生します。

z/OS でのデータ・セットとファイルの関連付け

PL/I プログラム内で使用されるファイルには、PL/I ファイル名が付きます。 プログラムの外部で存在する物理データ・セットの名前は、 オペレーティング・システムが認識できる名前、すなわち、データ・セット名 または *dsname* をとります。 名前のないデータ・セットもあります。その場合、 データ・セットはそのデータ・セットが設定されている装置によってシステムに認識されます。

オペレーティング・システムには、 使用しているプログラムが参照する物理データ・セット を識別する手段が必要なので、PL/I ファイル名を dsname に関連付ける、 プログラムにとっては外部のデータ定義 つまり *DD* ステートメント を記述する必要があります。

例えば、ご使用のプログラムに次のファイル宣言がある場合は、PL/I ファイルの名前に一致するデータ定義名 (*dd* 名) を指定して DD ステートメントを作成しなければなりません。

```
DCL STOCK FILE STREAM INPUT;
```

つまり、DD ステートメントにより物理データ・セット名 (dsname) が指定され、 その特性が指定されます。

```
//GO.STOCK  DD DSN=PARTS.INSTOCK, . . .
```

DD ステートメントの作成方法について詳しくは、ご使用のシステムのジョブ制御言語 (JCL) 資料を参照してください。

データ・セットを PL/I ファイルに関連付ける方法は、いくつかあります。 データ・セットを PL/I ファイルに関連付けるには、 データ・セットを定義する DD ステートメントの dd 名を、 次のいずれかと同じになるように指定します。

- 宣言された PL/I ファイル名
- 関連する OPEN ステートメントの TITLE オプションに指定されている式の文字ストリング値

また、対応する dd 名が次の条件を満たすように、PL/I ファイル名を選択する必要があります。

- ファイルが暗黙的にオープンされる場合、またはファイルを明示的にオープンする OPEN ステートメントに TITLE オプションが組み込まれていない場合、dd 名にはデフォルトでファイル名が使用されます。 ファイル名が 8 文字を超える場合、デフォルトの dd 名はそのファイル名の最初の 8 文字から構成されます。
- JCL の文字セットに区切り文字 (_) が含まれていません。 したがって、この文字を DD 名に使用することはできません。 有効な dd 名を式として使用する TITLE オプションにより ファイルをオープンする場合を除き、ファイル名の最初の 8 文字に 区切り文字は使用しないでください。 英字の拡張文字 $、@、および # も dd 名に使用することができますが、dd 名の最初の文字には A から Z までの英字でなければなりません。

外部名は 7 文字までに限定されているため、7 文字を超える外部ファイル名は、そのファイル名の最初の 4 文字と最後の 3 文字を連結したものに短縮されます。 しかしこのような短縮名は、関連 DD ステートメント内で dd 名 として使用される名前では**ありません**。

以下のステートメントについて考察します。

1. `OPEN FILE(SAMPLE);`
2. `OPEN FILE(OLDSAMPLE);`
3. `READ FILE(DETAIL) ...;`

ステートメント番号 1 が実行されると、ファイル名 SAMPLE は、現行ジョブ・ステップの DD ステートメントの DD 名と同じになります。 ステートメント番号 2 が実行されると、OLDSAMPL という名前は、現行ジョブ・ステップの DD ステートメントの DD 名と同じになります。 (ファイル名の最初の 8 文字が dd 名になります。 OLDSAMPLE が外部名である場合は、プログラム内で使用するためにコンパイラーによって OLDSPLE に短縮されます。) ステートメント番号 3 を使ってファイル DETAIL を暗黙的にオープンする場合、 ファイル名 DETAIL は現行ジョブ・ステップの DD ステートメントの dd 名と同じであると見なされます。

上記の場合はいずれも、対応する DD ステートメントが ジョブ・ストリーム内に存在しなければなりません。 それが存在しないと、UNDEFINEDFILE 条件が発生します。 上記 3 つの DD ステートメントは、次のように始まります。

1. `//SAMPLE DD ...`
2. `//OLDSAMPL DD ...`
3. `//DETAIL DD ...`

ファイルを明示的あるいは暗黙的にオープンするステートメント内のファイル参照がファイル定数でない場合は、その DD ステートメント名はファイル参照の値と同じ**でなければなりません**。 次の例は、DD ステートメントをどのようにファイル変数の値と関連付けるかを示します。

```
DCL PRICES FILE VARIABLE,
    RPRICE FILE;
      PRICES = RPRICE;
      OPEN FILE(PRICES);
```

この DD ステートメントでは、データ・セットをファイル定数 RPRICE と関連付ける必要があります。 RPRICE はファイル変数 PRICES の値です。

```
//RPRICE DD DSNAME=...
```

また、ファイル変数を使えば、1 つのステートメントで、いくつもの ファイルを何回も処理することができます。 次の例を参照してください。

```
DECLARE F FILE VARIABLE,
        A FILE,
        B FILE,
        C FILE;
          .
          .
          .
     DO F=A,B,C;
        READ FILE (F) ...;
          .
          .
          .
     END;
```

READ 。3 つのファイル A、B、C を読み込み、それぞれを異なるデータセットに関連付けることができる。 ファイル A、B、C は、それぞれの場合にこの READ ステートメントが実行されたあとも、 オープンの状態に保たれます。

次の OPEN ステートメントは、TITLE オプションの使用を示したものです。

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

このステートメントが正常に実行されるためには、現在のジョブ・ステップに DETAIL1 を ddname とする DD ステートメントがなければなりません。 このステートメントは、次のように始まります。

```
//DETAIL1 DD DSNAME=DETAILA,...
```

このように、ddname DETAIL1 を通して、データセット DETAILA とファイル DETAIL を関連付ける。

複数のファイルと 1 つのデータ・セットの関連付け

TITLE オプションを使用すると、以下の条件が満たされている場合に、複数の PL/I ファイルを同じ外部データ・セットに関連付けることができます。

- 最初のファイル関連付けは、同じ TITLE 名に対して 2 番目のファイル関連付けが開かれる前に閉じられます。
- 2 つのファイルは同じファイル属性を持ちます。

複数のファイルに同じ DD ステートメントを使用したい場合は、最初のファイルにファイル属性を指定することはできず、同じ DD ステートメントを使用して他のファイルにデフォルトのファイル属性を使用することもできません。 これは、入力ファイルと出力ファイルの両方に適用されます。 別々の DD ステートメントを使用する必要があります。

詳しくは、 *DFSMS* データ・セットの使用 の資料を参照してください。

次の例では、INVNTRY は 2 つのファイルに関連付けられるデータ・セットを定義する DD ステートメントの名前です。

```
    OPEN FILE (FILE1) TITLE('INVNTRY');
    .....
    CLOSE FILE (FILE1);
    .....
    OPEN FILE (FILE2) TITLE('INVNTRY');
```

複数のデータ・セットと 1 つのファイルの関連付け

TITLE オプションを使用すれば、複数のデータ・セットを 1 つのファイルに関連付けることができます。

ファイル名は、時点が異なれば、まったく別のデータ・セットを表すことができます。 次の OPEN ステートメントを見てください。

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

このステートメントを正常に実行するには、 現行ジョブ・ステップにおいて dd 名として DETAIL1 を使用して DD ステートメントを指定する必要があります。

```
//DETAIL1 DD DSNAME=DETAILA,...
```

ファイル DETAIL1 が DD ステートメント DETAIL1 の DSNAME パラメーターで指定されているデータ・セットに関連付けられます。 このファイルをクローズしてオープンし直せば、TITLE オプションで別の dd 名を指定して、そのファイルを別のデータ・セットに関連付けることができます。

TITLE オプションを使用すれば、オープン時に、特定のファイル名に関連付けるデータ・セットを複数のデータ・セットから 1 つ動的に選択できます。 次の例を考えてください。

```
DO IDENT='A','B','C';
   OPEN FILE(SAMPLE)
        TITLE('SAMPLE1'||IDENT);
    .
    .
    .
   CLOSE FILE(SAMPLE);
END;
```

この例では、DO グループの最初の反復中に SAMPLE がオープンされると、関連する DD 名は SAMPLE1A と見なされます。 処理が終了すると、そのファイルはクローズされ、 ファイル名と dd 名との関連付けも解除されます。 DO グループの 2 回目の反復時に、SAMPLE が再びオープンされます。 今回は、SAMPLE が DD 名 SAMPLE1B に関連付けられています。 同様に、DO グループの最後の反復時に、SAMPLE は DD 名 SAMPLE1C に関連付けられます。

複数のデータ・セットの連結

入力に関してのみ、複数の順次データ・セットや領域データ・セットを連結できます (すなわち、連続する 1 つのデータ・セットとして処理されるように複数のデータ・セットをリンクできます。 それには、そのようなデータ・セットを記述する最初の DD ステートメントを除くすべての DD ステートメントから dd 名を削除します。

例えば、次の DD ステートメントを定義すると、そのステートメントが 出てくるジョブ・ステップの実行の間中、データ・セット LIST1、LIST2、LIST3 は 1 つのデー タ・セットとして処理されます。

```
//GO.LIST DD DSNAME=LIST1,DISP=OLD
//        DD DSNAME=LIST2,DISP=OLD
//        DD DSNAME=LIST3,DISP=OLD
```

PL/I プログラムからの読み込みの場合は、 連結データ・セットは同じボリューム上にある必要はありません。

連結データ・セットを逆方向に処理することはできません。

z/OS での zFS ファイルへのアクセス

バッチ・プログラムから zFS ファイルにアクセスするには、DD ステートメント内、または OPEN ステートメントの TITLE オプション 内に zFS ファイル名を指定します。

例えば、DD zFS を使用して zFS ファイル /u/USER/sample.txt にアクセスするには、次のように DD ステートメントをコーディングします。

```
   //zFS DD PATH='/u/USER/sample.txt',PATHOPTS=ORDONLY,DSNTYPE=zFS
```

OPEN ステートメントの TITLE オプションを使用して同じファイルにアクセスするには、次のようにコーディングします。

```
   OPEN FILE(zFS) TITLE('//u/USER/sample.txt');
```

TITLE オプションにある 2 つのスラッシュに注意してください。 最初のスラッシュはファイル名 (DD 名でなく) が後に続くことを示し、2 番目は完全修飾 zFS ファイル名の先頭を示します。 バッチ環境で zFS ファイルが参照されるときは、ファイルを指定するために使用できる現行ディレクトリーが存在しないため、完全修飾名を使用する必要があります。

次のいずれかの方法で zFS ファイル名を指定することにより、PL/I 動的割り振りを使用してバッチ・プログラムから zFS ファイルにアクセスできます。

- DD ステートメント
- OPEN ステートメントの TITLE オプション
- PUTENV 組み込み関数
- ENVAR オプションを使用した PLIXOPT ストリング

次の例は、上記の方法を使用して zFS ファイルにアクセスする方法を示しています。

```
//zFS DD PATH='/u/USER/sample.txt',PATHOPTS=ORDONLY...

OPEN FILE(zFS) TITLE('PATH(/u/USER/sample.txt)');

xx = putenv('DD_zFS=/u/USER/sample.txt');

Dcl plixopt char(100) var ext static
    init('ENVAR("DD_zFS=PATH(/u/USER/sample.txt)")');
```

注 : PL/I 動的割り振りを使用するには、DSN() 形式 (MVS データ・セットの場合)、または PATH() 形式 (zFS ファイルの場合) を使用してファイル名を指定してください。

バッチ環境では、PL/I は zFS ファイルの取り扱い方法を以下の順序で決定します。

1. ファイル宣言で ENV(F) が指定されている場合は、ファイルが固定長レコードで構成されていると想定します。
2. ファイル宣言で ENV(V) が指定されている場合は、ファイルが lf 区切りレコードで構成されていると想定します。
3. ファイルの DD ステートメントで FILEDATA=BINARY が指定されている場合は、ファイルが固定長レコードで構成されていると想定します。
4. 上記以外の場合は、ファイルが lf 区切りレコードで構成されていると想定します。

UNIX から固定長 z/OS ファイルにアクセスするには、ファイルのレコード・サイズをファイルの ENVIRONMENT 属性で、または OPEN ステートメントの TITLE オプションで指定する必要があります。

- ファイル宣言に ENV(F RECSIZE(...)) が含まれていない場合は、データ・セット名とそれらの属性を TITLE オプションで指定する必要があります。 例えば、固定長 80 バイト・レコードのファイルの場合は、TITLE オプションを次のように指定することができます。

  ```
     '/dataset.name,type(fixed),recsize(80)'
  ```

- ENVIRONMENT 属性が F または RECSIZE の一方のみを指定している場合は、データ・セット名と省略された属性を TITLE オプションで指定する必要があります。
- ENVIRONMENT 属性が F と RECSIZE の両方を指定している場合は、データ・セット名のみを TITLE オプションで指定する必要があります。

z/OS UNIX でのデータ・セットとファイルの関連付け

PL/I プログラム内で使用されるファイルには、PL/I ファイル名が付きます。 また、データ・セットには、オペレーティング・システムにより識別される名前が付きます。

PL/I では、ご使用のプログラムで PL/I ファイルが指すデータ・セットを認識する手段が必要であるため、使用するデータ・セットに ID を指定するか、PL/I によるデフォルト ID の使用を許可する必要があります。

環境変数、または OPEN ステートメントの TITLE オプションを使用することにより、 明示的にデータ・セットを識別することができます。

PL/I 動的割り振りを使用するには、DSN() 形式 (MVS データ・セットの場合)、または PATH() 形式 (zFS ファイルの場合) を使用してファイル名を指定する必要があります。

PL/I 動的割り振りを使用して z/OS UNIX プログラムから zFS ファイルにアクセスするには、次のいずれかの方法で zFS ファイル名を指定できます。

- OPEN ステートメントの TITLE オプション
- PUTENV 組み込み関数
- ENVAR オプションを使用した PLIXOPT ストリング
- EXPORT ステートメント

環境変数の使用

export コマンドを使用して、PL/I ファイルに関連付けるデータ・セットを識別する環境変数を設定し、オプションでデータ・セットの特性を指定します。 環境変数によって提供される情報は、データ定義（または DD ）情報と呼ばれる。

これらの環境変数名の形式は DD_*DDNAME* です。ここで、*DD* 名 は PL/I ファイル定数 (または以下に定義する 代替 *DDNAME*) の名前です。 ファイル名が zFS ファイルを参照する場合は、ファイル名を適切に修飾する必要があります。 これを行わないと、PL/I ライブラリーは、そのファイル名が MVS データ・セットであると想定します。

例

- `declare MyFile stream output;`
  `export DD_MYFILE=/datapath/mydata.dat`

  `/datapath/mydata.dat` は zFS ファイルです。 ファイル名は完全修飾されています。
- `export DD_MYFILE=./mydata.dat`

  `./mydata.dat` は、現行ディレクトリーにある zFS ファイルを指します。
- `export DD_MYFILE=mydata.dat`

  `mydata.dat` は MVS データ・セットです。

次の例は、PL/I 動的割り振りを使用した zFS ファイルへのアクセスを示しています。

```
export DD_zFS="PATH(/u/USER/sample.txt)"
export DD_FILE="DSN(USER.FILE.EXT),SHR"
```

IBM のメインフレーム環境に精通していれば、この環境変数は以下のステートメントと同様に考えることができます。

z/OS での DD ステートメント
ALLOCATE ステートメント TSO

DD_DDNAME 環境変数を併用した構文およびオプションについて詳しくは、202 ページの『DD_DDNAME 環境変数を使った特性の指定』を参照してください。

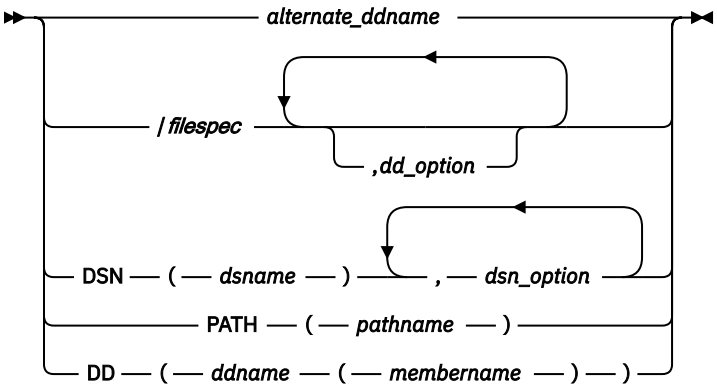
z/OS UNIX 環境では、バッチ環境よりも多くの可変長 zFS ファイル・タイプがサポートされるため、PL/I は zFS ファイルを以下のように扱います。

1. ファイル宣言で ENV(F) が指定された場合、ファイルは固定長レコードで構成されると仮定される。
2. ファイルの EXPORT ステートメントに TYPE が指定されている場合、そのファイルはそのタイプのレコードで構成されていると見なされる。
3. 上記以外の場合は、ファイルが lf 区切りレコードで構成されていると想定します。

OPEN ステートメントの TITLE オプションを使用する

OPEN ステートメントの TITLE オプションを使用して、PL/I ファイルに関連付けるデータセットを特定し、オプションでそのデータセットの追加特性を指定することができる。

expression は、次の構文をとる文字ストリングを与えられなければなりません。



注 : DD オプションは、64 ビットでのみ使用可能です。

代替 *dd* 名
代替 DD_DDNAME 環境変数の名前

代替環境変数 DD_DDNAME は、ファイル定数にちなんだ名前ではない。 例えば、 INVENTRY という名前のファイルがプログラムにあり、 DD_DDNAME という 2 つの環境変数を設定し、1 つ目を INVENTRY 、2 つ目を PARTS という名前にした場合、このステートメントを使うことで 2 つ目の環境変数にファイルを関連付けることができる：

```
open file(Inventry) title('PARTS');
```

filespec
使用しているシステムでの任意の有効なファイル指定

filespec の最大長は、1023 文字です。

dd_option
DD_DDNAME 環境変数で許可される 1 つ以上のオプション

環境変数 DD_DDNAME のオプションの詳細については、 202 ページの『DD_DDNAME 環境変数を使った特性の指定』 を参照のこと。

dsname
完全修飾 MVS データ・セット名。

dsn_option
1 つ以上の DSN オプション。

DSN オプションについて詳しくは、247 ページの『PL/I ダイナミックアロケーションを使用した QSAM ファイルの定義』、265 ページの『PL/I ダイナミックアロケーションを使用した REGIONAL(1) データセットの定義』、および 273 ページの『PL/I ダイナミックアロケーションを使用した VSAM ファイルの定義』 を参照してください。

pathname
完全修飾 zFS パス名。

***DD* 名**
PDS 、DD ステートメントに関連する DD 名。 PDSE

membername
PDS 、または PDSE

以下は、 z/OS DSN で OPEN ステートメントをこのように使う例である：

```
open file(Payroll) title('/June.Dat,append(n),recsize(52)');
```

TITLE オプションにある必須の先行スラッシュに注意してください。 この先行スラッシュは、ファイル名 (DD 名ではなく) が後に続くことを示しています。 この場合、June.Dat は MVS データ・セットです。

June.Dat が zFS ファイルの場合、 OPEN の文例は以下のようになる：

```
open file(Payroll) title('//u/USER/June.Dat,append(n),recsize(52)');
```

TITLE 、2 つのスラッシュがあることに注意。1 つ目のスラッシュは（DD 名ではなく）ファイル名であることを示し、2 つ目のスラッシュは完全修飾された zFS ファイル名の始まりであることを示す。

完全修飾名の代わりに相対 zFS ファイル名を指定することもできます。 次の例を参照してください。

```
open file(Payroll) title('./June.Dat,append(n),recsize(52)');
```

データ・セット名 June.Dat には、現行 z/OS UNIX ディレクトリーのパス名が接頭部として付きます。

DD_DDNAME この形式では、 PL/I はすべての DD 情報を TITLE 式またはファイル宣言の ENVIRONMENT 属性から取得する。

64 ビット用の DD オプションを使用する場合は、DD 名とともに PDS または PDSE メンバー名を指定する必要がある。 この OPEN に関連する DD 文では、DSN フィールドに PDS または PDSE 名を指定しなければならない。

次の組み合わせは、OPEN ステートメントを、DD 名と PDS または PDSE メンバー名の組み合わせで使用する方法を示しています：

```
OPEN FILE(QSAM01) title('DD(QSAM(MEMFILE))');

//QSAM DD DSN=USER.PDS.DATA,DISP=SHR
```

注 : PDS または PDSE メンバ内の 1 つまたは複数のレコードを追加または削除するには、ライブラリの別の部分でメンバ全体を書き換える必要があります。 サンプル・プログラムは 230 ページの『例: ライブラリー・メンバーの更新』 を参照。 PDS または PDSE メンバーの処理の詳細については、 227 ページの『第 9 章 ライブラリーの使用』 を参照のこと。

以下の例は、 PL/I ダイナミック・アロケーションをサポートする OPEN ステートメントの使用方法を示している：

- z/OS データ・セットの場合:

  ```
  OPEN FILE(QSAM01) title('DSN(USER.FILE.EXT),SHR');
  ```

- zFS ファイルの場合:

  ```
  OPEN FILE(QSAM01) title('PATH(/u/USER/sample.txt)');
  ```

データ・セットに関連付けられていないファイルの使用の試み

データ・セットに関連付けされていないファイルを (OPEN ステートメントの TITLE オプションを使って、あるいは、DD_DDNAME 環境変数を設定して) 使用しよ うとすると、UNDEFINEDFILE 条件が発生します。

SYSIN ファイルと SYSPRINT ファイルだけは例外です。 つまり、この 2 つのファイルはデフォルトでそれぞれ stdin および stdout になります。

PL/I によるデータ・セットの検索方法

PL/I により、新規データ・セットを作成するためのパス、 または既存データ・セットへアクセスするためのパスが、次のいずれかの方法で設定されます。

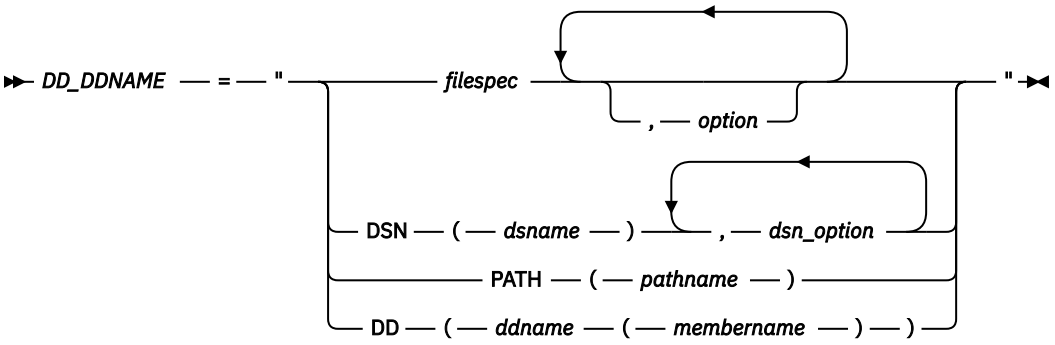
- 現行ディレクトリー。

- *export DD_DDNAME* 環境変数により定義されているパス。

DD_DDNAME 環境変数を使った特性の指定

export コマンドを使用すると、PL/I ファイルに 関連付けられるデータ・セットを識別する環境変数を設定することができます。 また、オプショナルでデータ・セットの特性も指定することができます。 環境変数から得られるこのような情報を、 データ定義 (または、dd) 情報と呼びます。

DD_DDNAME syntax



注 : DD オプションは、64 ビットでのみ使用可能です。

この構文ではブランクを使用することができます。 また、このステートメントの構文は、コマンド入力時にはチェックされません。 データ・セットのオープン時に、このステートメントの構文が検証されます。 構文が間違っていると、oncode 96 で UNDEFINEDFILE 。

DD_DDNAME
環境変数の名前

The DDNAME must be in uppercase and can be either the name of a file constant or an alternate DDNAME that you specify in the TITLE option of your OPEN statement. For more information, see 200 ページの『OPEN ステートメントの TITLE オプションを使用する』.

31 文字を超える代替 DDNAME を使用する場合、環境変数名の形成に使用される文字は最初の 31 文字のみです。

filespec
PL/I ファイルに関連付けるファイルまたは装置名の指定。

オプション
DD 情報として指定できるオプション。

DSNAME
完全修飾 MVS データ・セット名。

pathname
完全修飾 zFS パス名。

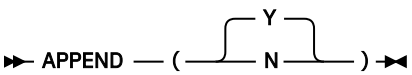
***DD* 名**
環境変数の名前

membername
PDS または PDSE 内のメンバーの名前

The following topics describe the options that you can specify as DD information. Note that these options do not apply to the DSN(), PATH(), or DD(*ddname*(*membername*)) formats.

APPEND

APPEND オプションは、既存のデータセットを拡張するか再作成するかを指定する。



Y
新規レコードを、順次データ・セットの終わりに追加する、あるいは相対データ・ セットまたは索引付きデータ・セットに挿入することを指定します。

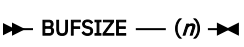
N
ファイルが存在する場合、そのファイルを再作成することを指定します。

APPEND オプションは OUTPUT ファイルにのみ適用される。 APPEND は以下の条件では無視される：

- 指定ファイルが存在しない場合
- ファイルには OUTPUT 属性がありません。
- 組織は REGIONAL(1)。

BUFSIZE

BUFSIZE オプションは、バッファーのバイト数を指定します。



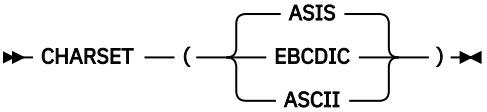
RECORD 出力はデフォルト設定でバッファーに入り、BUFSIZE のデフォルト値 64k です。 STREAM 出力はバッファーに入れられますが、デフォルトでは入れられません。 STREAM 出力のデフォルト BUFSIZE 値はゼロです。

BUFSIZE の値がゼロの場合、バッファーのバイト数は、RECSIZE オプションまたは LRECL オプションで指定された値と同じです。

BUFSIZE オプションが有効なのは、連続バイナリー・ファイルだけです。 ファイルが端末入力に使用されている場合は、 効率を上げるために、BUFSIZE に値をゼロを割り当てるべきです。

レコード入出力用の CHARSET

このバージョンの CHARSET オプションは、レコード入出力を使用する連続ファイルにのみ適用されます。 ユーザーは、このオプションを使用することにより、ASCII データ・ファイルを入力ファイルとして使用したり、出力ファイルの文字セットを指定したりできます。

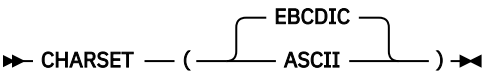


入力ファイルの形式、または出力ファイルにとらせたい形式 に基づいて、CHARSET のサブオプションを選択してください。

ストリーム入出力用の CHARSET

CHARSET オプションのこのバージョンは、ストリーム入力ファイルおよびストリーム出力ファイルにだけ適用されます。 ユーザーは、このオプションを使用することにより、ASCII データ・ファイルを入力ファイルとして使用したり、出力ファイルの文字セットを指定したりできます。

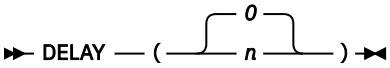
ストリーム入出力を使用しているときに ASIS を指定しようとしても、 エラーは出されず、文字セットは EBCDIC として扱われます。



入力ファイルの形式、または出力ファイルにとらせたい形式 に基づいて、CHARSET のサブオプションを選択してください。

DELAY

DELAY オプションは、システムでファイルやレコードのロックを取得できない場合に、失敗した操作をコンパイラーが再試行するのを遅延させる時間 (ミリ秒) を指定します。

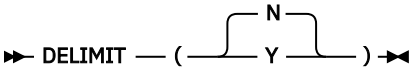


このオプションを適用できるのは、VSAM ファイルだけです。

DELIMIT

DELIMIT オプションは、入力ファイルにフィールド区切り文字が含まれているかを指定します。

フィールド区切り文字は、レコードのフィールドを分離するブランクかまたはユーザー定義文字です。 このオプションを適用できるのは、ソート入力ファイルだけです。



ソート・ユーティリティー・プログラムは、 フィールド区切り文字の有無により、 テキスト・ファイルとバイナリー・ファイルを区別します。 フィールド区切り文字が含まれている入力ファイルは、 テキスト・ファイルとして処理され、区切り文字がない入力ファイルはバイナリー・ファイルと見なされます。 この情報は、ライブラリーが正しいパラメーターを ソート・ユーティリティー・プログラムに渡すために必要です。

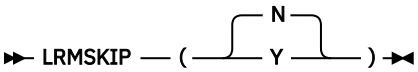
LRECL

LRECL オプションは RECSIZE オプションと同じです。

LRECL が指定されておらず LINESIZE 値による暗黙指定 もされていない場合 (ただし TYPE(FIXED) ファイルを除く)、 デフォルトは 1024 です。

LRMSKIP

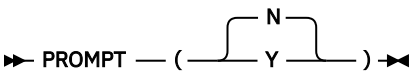
LRMSKIP オプションを使用すれば、ファイルがオープンされた後で最初に実行される SKIP フォーマット項目の先頭ページの *n* 行目で出力を開始できます。 *n* は、PUT ステートメントまたは GET ステートメントの SKIP オプションで指定された値を指します。



n がゼロまたは 1 の場合、出力は先頭ページの 1 行目から開始されます。

PROMPT

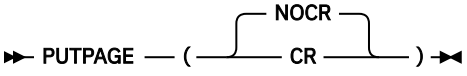
PROMPT オプションは、端末からのストリーム入力のプロンプトとしてコロンを表示するかどうかを指定する。



PUTPAGE

PUTPAGE オプションは、用紙送り文字の後ろに復帰文字を入れるかどうかを指定します。 このオプションが適用されるのは、プリンター向けファイルだけです。

プリンター向けファイルは、PRINT 属性を指定して宣言された ストリーム出力ファイル、あるいは、CTLASA 環境オプションを 指定して宣言されたレコード出力ファイルです。



NOCR
用紙送り文字 ('0C'x) の後ろに復帰文字 ('0D'x) を入れな いことを指します。

CR
用紙送り文字の後ろに復帰文字を追加することを指します。 このオプションは、出力が IBM 以外のプリンターに送られる場合に指定する必要があります。

RECCOUNT

RECCOUNT オプションは、PL/I ファイルのオープン・プロセス中に作成される、 相対データ・セットまたは領域データ・セットにロードできる最大のレコード数を指定します。

注 : RECCOUNT オプションは、64 ビット・プログラムではサポートされません。

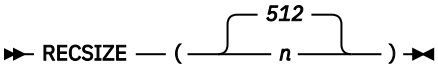
PL/I がデータ・セットを作成も再作成もしない場合、RECCOUNT オプションは無視されます。

RECCOUNT オプションのデフォルトは 50 です。

注 : z/OS において REGIONAL(1) データ・セットの機能性とパフォーマンスを向上させるには、/filespec パラメーターと RECCOUNT パラメーターの両方が指定された TITLE オプションを省略することをお勧めします。 この場合、ファイルにロードされるレコードの数は、データ・セットの最初のエクステントに割り振られたスペースによって決まります。 詳しくは、263 ページの『第 13 章 領域データ・セットの定義と使用』を参照してください。

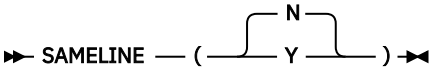
RECSIZE

RECSIZE オプションは、データ・セット内のレコードの長さを指定します。 領域データ・セットおよび固定長データ・セットの場合は、RECSIZE はデータ・セットの各レコードの長さを指定します。その他のデータ・セット・タイプの場合は、RECSIZE はレコードがとれる最大の長さを指定します。



SAMELINE

SAMELINE オプションは、入力を求めるプロンプトのステートメントと同じ行で、 システム・プロンプトを行わせるかどうかを指定します。



以下の例は、PROMPT オプションと SAMELINE オプションのいくつかの組み合わせの結果を示しています。

例 1

ステートメント PUT SKIP LIST('ENTER:');を指定すると、出力は次のようになります。

```
prompt(y) sameline(y)      ENTER: (cursor)
prompt(y) sameline(n)      ENTER:
                           (カーソル)
prompt(n) sameline(y)      ENTER: (cursor)
prompt(n) sameline(n)      ENTER:
                           (カーソル)
```

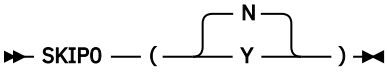
例 2

ステートメント PUT SKIP LIST('ENTER');を指定すると、出力は次のようになります。

```
prompt(y) sameline(y)      ENTER: (cursor)
prompt(y) sameline(n)      ENTER
                           :
                           (カーソル)
prompt(n) sameline(y)      ENTER (cursor)
prompt(n) sameline(n)      ENTER
                           (カーソル)
```

SKIP0

SKIP0 オプションは、ソース・プログラムに SKIP(0) ステートメント がコーディングされた場合、ライン・カーソルをどこに移動するかを指定します。 SKIP0 オプションは、PM アプリケーションとしてリンクされていない 端末ファイルに適用されます。



SKIP0(N)
カーソルを次の行の先頭に移動することを指定します。

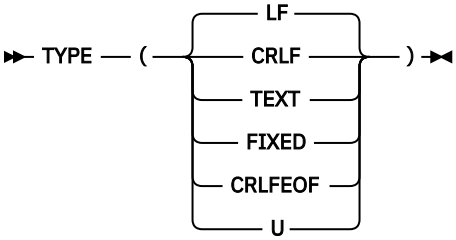
SKIP0(Y)
カーソルを現在行の先頭に移動することを指定します。

次の例は、現在の出力行の先頭にカーソルが移動するように、 出力を端末スキップ・ゼロ行で行う方法を示しています。

```
export DD_SYSPRINT='stdout:,SKIP0(Y)'
```

TYPE

TYPE オプションは、ネイティブ・ファイルのレコードのフォーマットを指定する。



CRLF
レコードを文字の組み合わせ CR - LF で区切ることを指定します。 ('CR' と 'LF' は、 それぞれ復帰と改行の ASCII 値である '0D'x と '0A'x を表します。)出力ファイルの場合、PL/I は各レコードの終わりにこれらの文字を挿入します。 入力ファイルの場合、PL/I はこれらの文字を破棄します。 入力、出力のいずれの場合も、これらの文字は RECSIZE の考慮事項には入りません。

データ・セットのレコード長として決められている値よりも長いレコードを データ・セットに入れることはできません。

LF
レコードが LF 文字組み合わせで区切られることを指定します。 ('LF' は、ASCII コードの用紙送り、つまり '0A'x を表します。)出力ファイルの場合、PL/I は各レコードの終わりにこれらの文字を挿入します。 入力ファイルの場合、PL/I はこれらの文字を破棄します。 入力、出力のいずれの場合も、これらの文字は RECSIZE の考慮事項には入りません。

データ・セットのレコード長として決められている値よりも長いレコードを データ・セットに入れることはできません。

TEXT
前述の LF と同じです。

FIXED
データ・セット内の各レコードの長さが同じであることを指定します。 データ・セット内のレコード長として指定されている値は、レコードの境界を認識す る場合に使用されます。

TYPE(FIXED) ファイルに含まれるすべての文字は、制御文字が存在する場合はそれも含めてデータとみなされる。 指定したレコード長が、存在している文字を反映していること、あるいは、指定したレコード長がレコード内の全文字を扱えることを確認してください。

CRLFEOF
出力ファイルを除き、このサブオプションは CRLF と同じ情報を指定する。 ファイルの 1 つが出力についてクローズされるときに、 ファイルの終わりマーカーが最後のレコードに追加されます。

U
レコードが不定形式であることを表します。 これらの不定形式ファイルは、OPEN および CLOSE を除いて、 どのレコード入出力またはストリーム入出力のステートメントでも使用できません。 TYPE(U) ファイルからの読み取りは、FILEREAD 組み込み関数を使用すること のみにより行うことができます。 また、TYPE(U) ファイルへの 書き込みは、FILEWRITE 組み込み関数を使用することのみにより行うことができます。

ASA(N) オプションを指定したプリンター向けファイルの場合 このオプションが無視されるということを除き、TYPE オプションを 適用できるのは CONSECUTIVE ファイルだけです。

使用しているプログラムが TYPE(FIXED) が有効である既存のデータ・セット にアクセスしようとしており、かつそのデータ・セット長が ユーザーが指定した複数の論理レコード長の倍数でない 場合は、PL/I は UNDEFINEDFILE 条件を発生させます。

TYPE(FIXED)属性を持つ nonprint ファイルを使用する場合、 SKIP は行末までの空白で置き換えられる。 TYPE(LF) が使われている場合、 SKIP は末尾の空白を除いた LF に置き換えられる。

データ・セット特性の設定

データ・セットは、オペレーティング・システムのデータ管理ルーチンが理解できる特定のフォーマットで保管されているレコードから構成されています。 ユーザーのプログラムでファイルの宣言またはオープンを行うときに、ユーザーはそのファイルに入っているレコードの特性を PL/I およびオペレーティング・システム に記述します。

また、 JCL 、または OPEN ステートメントの TITLE オプションの式を使用して、オペレーティングシステムにデータセットまたはデータセットに関連する PL/I ファイル内のデータの特性を記述することもできます。

必ずしも、プログラムの内と外の両方で、自分のデータを 記述する必要はありません。多くの場合、1 回の記述が、データ・セットとその 関連 PL/I ファイルの両方に役立ちます。 実際、データの特性は一箇所のみに記述する方が有利です。

使用するプログラム・データおよびデータ・セットを効率よく記述するには、オペレーティング・システムがどのようにデータを移動および保管するのかを理解する必要があります。

ブロックおよびレコード

データ・セット内のデータ項目は、 ブロック間ギャップ (IBG) で区切られているブロックに配置されます。 (これをレコード間ギャップと呼んでいるマニュアルもあります。)

ブロック は、データ・セットに送られてくる、 あるいはデータ・セットから送り出されるデータの単位です。 各ブロックには、1 つのレコード、レコードの一部分、または複数のレコード が入っています。 ブロック・サイズは、DD ステートメントの BLKSIZE パラメーター内、あるいは ENVIRONMENT 属性の BLKSIZE オプション内で指定することができます。

レコード は、プログラムに送られてくる、 またはプログラムから送り出されるデータの単位です。 レコード長は、DD ステートメントの LRECL パラメーター 内、OPEN ステートメントの TITLE オプション内、または ENVIRONMENT 属性 の RECSIZE オプション内で指定することができます。

PL/I プログラムを作成する場合は、読み取りあるいは書き込みを行うレコードだけを考慮すれば済みます。 しかし、自分のプログラムが作成あるいはアクセスするデータ・セットを記述する場合は、 ブロックおよびレコード間の関係を知っている必要があります。

ブロック化によって、磁気ストレージ・ボリューム内のストレージ・スペースを 節約することができます。ブロック化が、ブロック間ギャップ数を 減らし、データ・セットを処理するのに必要な入出力操作回数 を減らすことにより効率を上げるためです。 また、レコードは、データ管理ルーチンにより、ブロック化およびブロック化 解除されます。

情報交換コード

データが記録される通常のコードは拡張 2 進化 10 進コード (EBCDIC) です。

ASCII コードの各文字は 7 ビット・パターンで表されるので、 このようなパターンが 128 通りあります。また、ASCII セットには、有効な ASCII コードがない EBCDIC 文字 を表すために使用される置換文字 (SUB 制御文字) があります。 ASCII 置換文字は、00111111 というビット・パターンを持つ EBCDIC SUB 文字に変換されます。

レコード・フォーマット

データ・セット内のレコードは、次のいずれかのフォーマットを持っています。

固定長
可変長
不定長

レコードは、必要に応じて、ブロック化することができます。 オペレーティング・システムは、 固定長レコードと可変長レコードを非ブロック化しますが、 不定長レコードを非ブロック化するには、 ユーザーのプログラム内にコードを提供する必要があります。

レコード・フォーマットは、DD ステートメント の RECFM パラメーター内、OPEN ステートメントの TITLE オプション内、 または ENVIRONMENT 属性のオプションとして指定します。

固定長レコード

固定長レコードには、次に挙げるフォーマットを指定することがでます。

F
固定長、非ブロック化

FB
固定長、ブロック化

FS
固定長、非ブロック化、標準

FBS
固定長、ブロック化、標準

固定長レコードを持っているデータ・セットの 場合は、209 ページの図 4 に示すように、すべてのレコードの長さが等しくなります。 レコードがブロック化されていると、通常、 各ブロックには同じ数の固定長レコードが入っています (ただし、 ブロックは切り捨てられる場合もあります)。 レコードがブロック化されていない場合は、 各レコードがブロックを構成します。

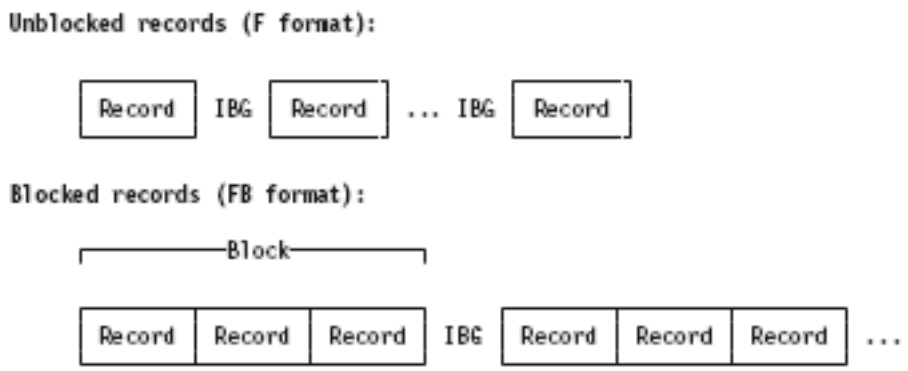


図 *4.* 固定長レコード

オペレーティング・システムでは、一定のレコード長に基づいてブロック化、 非ブロック化が行われるため、可変長レコードより固定長レコードの方が速く処理されます。

可変長レコード

指定できる可変長レコードのフォーマットは、次のとおりです。

V
可変長、非ブロック化

VB
可変長、ブロック化

VS
可変長、非ブロック化、スパン

VBS
可変長、ブロック化、スパン

V フォーマットでは、可変長レコードと可変長ブロックの両方を使用することができます。 各レコードの 4 バイトの接頭部と、 各ブロックの最初の 4 バイトには、オペレーティング・システムが使用 する場合の制御情報が入ります (レコードまたはブロックのバイト単位の 長さを含む)。 このような制御フィールドのため、可変長レコードは逆方向に読み込むことができません。

V フォーマットは、非ブロック化可変長レコードを表します。 各レコードは、レコードが 1 つだけを持つブロックとして扱われます。 ブロックの最初の 4 バイトにはブロック制御情報が入り、次の 4 バイトにはレコード制御情報が入ります。

VB フォーマットは、ブロック化可変長レコードを表します。 各ブロックには、そのブロックに収容可能なレコード数と同じ数のレコードが入ります。 ブロックの最初の 4 バイトにはブロック制御情報が入り、各レコードの 4 バイト接頭部にはレコード制御情報が入ります。

スパン・レコード: スパン・レコードは可変長レコードで、 レコードの長さがブロック・サイズを超えることもできます。 超えた場合は、レコード・フォーマットを VS または VBS のいずれかで指定して、 レコードを複数のセグメントに分割し、2 つ以上の連続ブロックに置きます。 セグメンテーションおよび組み立ては、オペレーティング・システムによって処理されます。 スパン・レコードを使用すると、レコードの長さを問わず、 ブロック・サイズが選択でき、補助記憶域を最大限に活用し、伝送の効果を最大限に高めます。

VS フォーマットは、V フォーマットに似ています。 それぞれのブロックは、1 つのレコードまたは レコードのセグメントのみを含みます。 ブロックの最初の 4 バイトにはブロック制御情報が入り、次の 4 バイトにはレコードまたはセグメント制御情報 (レコードが単体のものか、 または最初、中間、最後のセグメントであるかということを示す情報を含む) が入ります。

VBS フォーマットでは、各ブロックが、単体のレコードまたはセグメントをできるだけ多く保持することができるという点において、VS フォーマットと異なっています。したがって、各ブロックのサイズがほとんど同じになります (ただし、各セグメントは最低 1 バイトのデータを含まなければならないため、最大で 4 バイトの変化幅があります)。

不定長レコード

U フォーマットでは、F フォーマットにも V フォーマットにも 当てはまらないレコードを処理することができます。 オペレーティング・システムおよびコンパイラーは各ブロックをレコード として扱います。 ユーザー・プログラムで必要に応じてブロック化あるいは非ブロック化を 行わなくてはなりません。

データ・セットの編成

オペレーティング・システムのデータ管理ルーチンは、いくつかのタイプのデータ・セットを処理できます。 これらのデータ・セットは、その内部でのデータ保管方法、およびデータにアクセスするために許可される方法において異なります。

次の表は、VSAM 以外のデータセットの 3 つの主な種類と、その PL/I 構成を説明する対応するキーワードを示している [1]。

データ・セットのタイプ	PL/I 編成
順次	CONSECUTIVE or ORGANIZATION(consecutive)
索引付き	INDEXED or ORGANIZATION(indexed)
直接	REGIONAL or ORGANIZATION(relative)

データ・セットの 4 つ目のタイプ 区分 には、対応する PL/I 編成はありません。

また、PL/I では、3 つのタイプの VSAM データ編成 (*ESDS*、*KSDS*、および *RRDS*) もサポートされています。 VSAM データ・セットの詳細については、273 ページの『第 14 章 VSAM データセットの定義と使用』を参照してください。

シーケンシャル（または CONSECUTIVE ）データセットでは、レコードは物理的な順序で配置される。 あるレコードが与えられた場合、そのレコードの次のレコード位置は、 そのレコードが物理的にデータ・セットのどこにあるかによって決まります。 直接アクセス装置に対して順次編成を選択することができます。

インデックス付きシーケンシャル （または INDEXED ）データセットは、ダイレクト・アクセス・ボリューム上に存在しなければならない。 オペレーティング・システムが維持する索引や索引のセットにより、特定の基本レコードの位置が与えられます。 これによって、順次処理だけでなく、レコードの直接検索、置換、追加、および削除を行うことができます。

ダイレクト （または REGIONAL ）データセットは、ダイレクト・アクセス・ボリューム上に存在しなければならない。 データ・セットは複数の領域に分割され、各領域には 1 つ以上のレコードが入っています。 領域番号を指定するキーを使えば、任意のレコードに直接アクセスすることができます。 順次処理を行うことも可能です。

区分 データ・セットでは、順次編成されるデータから成り、 それぞれがメンバーと呼ばれる独立したグループは、直接アクセス・ データ・セット内に存在します。 このタイプのデータ・セットには、 各メンバーの位置をリストするディレクトリーが 1 つ含まれています。 区分データ・セットはしばしば ライブラリーと呼ばれます。 このコンパイラーには、区分データ・セットを作成したり、 区分データ・セットにアクセスするための特殊機能はありません。 各メンバーは PL/I プログラムによって CONSECUTIVE データセットとして処理できる。 区分データ・セットをライブラリーとして使用する方法について詳しくは、227 ページの『第 9 章 ライブラリーの使用』を参照してください。

[1] 順次 および 直接 という用語を、PL/I ファイル属性 シーケンシャル および ダイレクト と混同しないでください。 属性は、対応するデータ・セットの編成方法ではなく、ファイルの処理方法を示します。

ラベル

オペレーティング・システムは内部ラベルを使って、直接アクセス・ボリューム を識別したり、データ・セット属性 (例えば、レコード長やブロック・サイズ) を保管します。 これらの属性情報は、最初は DD ステートメントまたはユーザー・プログラムから入手する必要があります。

IBM 標準ラベルには、初期ボリューム・ラベルとヘッダー・ラベルという 2 つの部分があります。 初期ボリューム・ラベルは、特定のボリュームとその所有者を識別します。 一方、ヘッダー・ラベルはボリューム上の各データ・セットの前後にあります。 ヘッダー・ラベルには、システム情報、装置依存情報 (記録手法など)、およびデータ・セット特性が含まれています。

直接アクセス・ボリュームには、IBM 標準ラベルが付きます。 ボリューム・ラベルで各ボリュームが識別されます。 このラベルには、ボリュームのシリアル番号とボリューム目次（VTOC ）のアドレスが含まれている。目次には、ボリュームに格納された各データセットのデータセット・コントロール・ブロック（DSCB）と呼ばれるラベルが含まれる。

データ定義 (DD) ステートメント

データ定義 (DD) ステートメントは、オペレーティング・システムに 対してデータ・セットを定義するためのジョブ制御ステートメントであり、 入出力リソースの割り振りをオペレーティング・システムに要求するものです。 データ・セットを動的に割り振らない場合は、 各ジョブ・ステップに、そのジョブ・ステップで処理する データ・セットの DD ステートメントをすべて組み込んでおく必要があります。

ジョブ制御ステートメントの構文については、「*z/OS MVS JCL* ユーザーズ・ガイド」を参照してください。 DD ステートメントのオペランド・フィールドには、データ・セットの 位置 (例えば、ボリューム通し番号や、ボリュームをマウントする 装置の識別名) を記述したキーワード・パラメーター、および データそのものの属性 (例えば、レコード・フォーマット) を記述した キーワード・パラメーターを入れることができます。

DD ステートメントを使用すると、使用するデータ・セットおよび入出力装置 から独立した PL/I ソース・プログラムを作成することができます。 また、ユーザー・プログラムを再コンパイルしなくても、データ・セットの パラメーターを変更したり、さまざまなデータ・セットを処理したり することができます。

ENVIRONMENT 属性の LEAVE および REREAD オプションを使用すると、磁気テープ・ボリュームの終わりに達したとき、または磁気テープのデータ・セットが閉じたときにとるアクションを制御する DISP パラメーターを使用できます。 LEAVE オプションおよび REREAD オプションについては、256 ページの『LEAVE|REREAD』 を参照してください。

PL/I バージョン 1 の標準機能であった書き込み妥当性検査は、本バージョンでは実行さ れません。 書き込み妥当性検査を実行するには、JCL DD ステートメントの DCB パラメーター の OPTCD サブパラメーターを使って要求できます。 資料「*OS/VS2 Job Control Language*」を参照してください。

条件付きサブパラメーターの用法

PL/I プログラムにより処理されるデータ・セットの DISP パラメーターの条件 付きサブパラメーターを使用する場合は、ステップ異常終了機能を使う必要があります。

ステップ異常終了機能は、次のようなにして入手します。

1. アプリケーションの条件付きサブパラメーターを適用する必要がある障害が発生したために プログラムの実行を終了する場合は常に、ERROR 条件を発生させるか、 または ERROR 信号を出す。
2. PL/I のユーザー出口を ABEND 要求に変更する必要がある。

データ・セット特性

DD ステートメントの DCB (データ制御ブロック) パラメーターを使用すると、データ・セット内のデータの特性を記述したり、実行時のデータの処理方法を記述することができます。 DD ステートメントの他のパラメーターは、主としてデータ・セットの識別、位置、 処置を扱うのに対し、DCB パラメーターはレコード自体の処理に必要な情報を指定します。

DCB パラメーターのサブパラメーターについては、「*OS/VS2 Job Control Language*」を参照してください。

DCB パラメーターには、以下のデータ特性を記述するサブパラメーターが含まれます。

- データ・セットの編成とそのアクセス 方法 (サブパラメーター CYLOFL、 DSORG、LIMCT、NTM、および OPTCD)
- プリンターの行送りなどの装置依存情報 (サブパラメーター CODE、FUNC、 MODE、OPTCD=J、 PRTSP、STACK、および STACK)
- レコード・フォーマット (サブパラメーター BLKSIZE、KEYLEN、LRECL、および RECFM)
- 各レコードの最初のバイト (RECFM サブパラメーター) に挿入される ASA 制御文字 (ある場合)

BLKSIZE、LRECL、KEYLEN、および RECFM (またはそれらと同等のもの) は、DCB パラメーター ではなく、ユーザーの PL/I プログラムの ファイル宣言の ENVIRONMENT 属性を使って指定することができます。

DCB パラメーターを使って、PL/I プログラム内でデータ・セット用に (宣言されたファイル属性と、その属性で暗黙指定されたその他の属性を使って) 既に 設定されている情報を指定変更することはできません。 既に指定されている情報を変更しようとする DCB サブパラメーターは無視されます。

新規データ・セットに関しては、DD ステートメントとの競合が発生する場合は、プログラムで定義されたファイルの属性が使用されます。

PDS ファイルをクローズするときに、RC=4 を伴うメッセージ IEC225I が出される場合があります。 このメッセージは安全であり、無視できます。

次の DCB パラメーターの例は、長さ 40 バイトの固定長レコードが 400 バイトの長さのブロックにグループ化されることを指定しています。

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

OPEN ステートメントの TITLE オプションを使用する

OPEN ステートメントの TITLE オプションを使用して、PL/I ファイルに関連付けるデータセットを特定し、オプションでデータセットの追加特性を指定することができます。

関連情報

200 ページの『OPEN ステートメントの TITLE オプションを使用する』
OPEN ステートメントの TITLE オプションを使用して、PL/I ファイルに関連付けるデータセットを特定し、オプションでそのデータセットの追加特性を指定することができる。

PL/I ファイルとデータ・セットの関連付け

PL/I OPEN ステートメントを実行することにより、 ファイルとデータ・セットを関連付けることができます。 PL/I CLOSE ステートメントを実行することにより、関連付けられていたデータ・セットからファイルが切り離されます。

ファイルのオープン

PL/I OPEN ステートメントを実行することにより、 ファイルとデータ・セットを関連付けることができます。 これを行うには、ファイルを記述している情報とデータ・セットを記述している 情報をマージする必要があります。 ファイル属性とデータ・セット特性の間に矛盾が検出されると、UNDEFINEDFILE 条件が発生します。

PL/I ライブラリーのサブルーチンは、データ・セットの骨組みデータ制御ブロックが作成されます。 このサブルーチンは、DECLARE ステートメントおよび OPEN ステートメントから得られるファイル属性と、宣言された属性で暗黙指定される属性を使用して、可能な限りデータ制御ブロックを完成させます。 (213 ページの図 5 を参照してください。) その後で、このサブルーチンは OPEN マクロ命令を発行します。 このマクロ命令は、データ管理ルーチンを呼び出して、正しいボリュームがマウントされていることを検査し、データ制御ブロックを完成させます。

データ管理ルーチンは、さらに必要な情報がないか調べるためにデータ制御ブロックを検査し、 次に、まず DD ステートメント内からその情報を検索し、最後に、 データ・セットが存在するかどうか、かつデータ・セット・ラベル内に標準ラベル が付いているかどうかを調べます。 新規データ・セットの場合は、データ管理ルーチンは、ラベル (必 要な場合) を作成し、そのラベルにデータ制御ブロックからの情報を入れます。

INPUT データ・セットの場合、属性が矛盾しない限り、PL/I プログラムは DCB 属性を指定変更できます。 OUTPUT データ・セットの場合、PL/I プログラムは DCB 属性を指定変更できません。 ただし、データ・セットのオープン時に欠落 DCB 属性があれば、その属性は PL/I プログラムから取得されます (プログラムでその属性が指定されている場合)。

DCB フィールドがこれらのソースの情報で埋められると、制御は PL/I ライブラリー・サブルーチンに戻ります。 依然として値が入力されていないフィールドには、PL/I OPEN サブルーチン がデフォルトの情報を提供します。 例えば、LRECL が指定されていない場合は、BLKSIZE に与えられた値が提供されます。

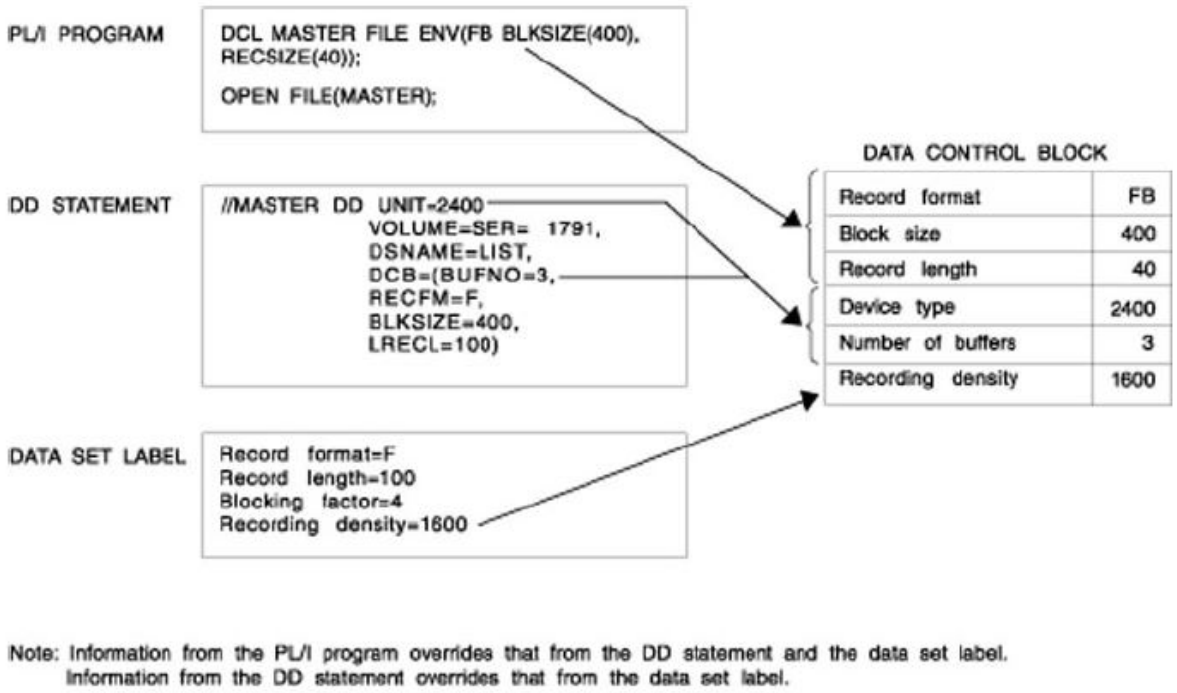


図 *5.* オペレーティング・システムによる *DCB* への情報の組み込みの方法

システム決定ブロック・サイズの使用

データ機能プロダクト (DFP) のシステム決定ブロック・サイズ機能を z/OS 上で使用する場合は、割り当てられている装置タイプに最適なブロック・サイズが DFP により算出されます。

新しい DASD データセットを作成する際、以下の条件がすべて真である場合、システムは最適なブロックサイズを導き出し、データセットのラベルに保存します：

- ブロック・サイズが利用できないか、ソースから指定されていません。 BLKSIZE=0 指示します。
- LRECL を指定するか、データ・クラス内にある。 データセットは SMS 、管理する必要はない。
- RECFM を指定するか、データ・クラス内にある。 固定または可変である必要があります。
- PS または PO として DSORG を指定するか、DSORG を省略すると、データ・クラスでは PS または PO となる。

システム決定ブロック・サイズがアクティブであれば、DFP がブロック・サイズを決定し、それをデータ・セット・ラベルに置いてから、PL/I でファイルを開きます。 したがって、 PL/I プログラムが ENVIRONMENT オプションでブロックサイズを指定する場合、システムで決定されたブロックサイズの値と矛盾してはならない。

システム決定ブロック・サイズについて詳しくは、「*z/OS DFSMS* データ・セットの使用法」を参照してください。

ファイルのクローズ

PL/I CLOSE ステートメントを実行すると、ファイルが関連付けられていたデータセットから切り離される。

PL/I ライブラリサブルーチンは、まず CLOSE マクロ命令を発行する。データ管理ルーチンから制御が戻ると、サブルーチンはファイルがオープンされたときに作成されたデータ制御ブロックを解放する。 データ管理ルーチンは、新規データ・セットのラベルの作成を完了し、既存データ・セットのラベルを更新します。

ENVIRONMENT

ENVIRONMENT 属性の様々なオプションを使って、データの特性を指定することができます。 ファイルのタイプごとに、さまざまな属性および環境オプションがあります。

ENVIRONMENT 属性

PL/I ファイル宣言ファイルの ENVIRONMENT 属性を使用すれば、ファイルに関連付けられているデータ・セットの物理編成に関する情報を指定したり他の関連情報を記述したりできます。 この情報のフォーマットは、括弧で囲んだオプション・リストでなければなりません。

省略形: ENV

ブランクまたはコンマで区切ったオプションを、任意の順序で指定することができます。

次の例は、完全なファイル宣言のコンテキストにおける ENVIRONMENT 属性の構文を示したものです (指定されているオプションは VSAM 用です)。

```
DCL FILENAME FILE RECORD SEQUENTIAL
    INPUT ENV(VSAM GENKEY);
```

次の 215 ページの表 15 は、ENVIRONMENT オプションとファイル属性をまとめたものである。 使用に関する特定の制限については、表に含まれる注記およびコメントを参照してください。

表 *15. PL/I* ファイル宣言の属性

データ・セット・タイプ / ファイル・タイプ	Stream	RECORD									凡例:
			連続	順次				直接			C VSAM では検査 D デフォルト I 指定または暗黙指定が必要 N VSAM では無視 O オプション S 指定が必要 - 無効
	Consecutive	Buffered	Unbuffered	Regional	Teleprocessing	Indexed	VSAM	Regional	Indexed	VSAM	

表 *15. PL/I* ファイル宣言の属性

ファイル属性 [1]											暗黙指定属性
ファイル	I	I	I	I	I	I	I	I	I	I	
入力 [1]	D	D	D	D	D	D	D	D	D	D	ファイル
出力	O	O	O	O	O	O	O	O	O	O	ファイル
環境	I	I	I	S	S	S	S	S	S	S	ファイル
ストリーム	D	-	-	-	-	-	-	-	-	-	ファイル
印刷 [1]	O	-	-	-	-	-	-	-	-	-	ファイル・ストリーム出力
レコード	-	I	I	I	I	I	I	I	I	I	ファイル
更新日	-	O	O	O	-	O	O	O	O	O	ファイル・レコード
順次	-	D	D	D	-	D	D	-	-	D	ファイル・レコード
バッファー付き	-	D	-	-	I	D	D	-	-	S	ファイル・レコード
キー順 [2]	-	-	-	O	I	O	O	I	I	O	ファイル・レコード
直接	-	-	-	-	-	-	S	S	S	S	キー付きファイル・レコード

ENVIRONMENT オプション											**コメント**
F|FB|FS|FBS|V|VB|VS|VBS||U	I	S	S	-	-	-	N	-	-	N	VS と VBS が無効です ストリームあり
F|FB|U	S	S	-	-	-	-	N	-	-	N	ASCII データ・セットのみ
F|V|U	-	-	-	S	-	-	N	S	-	N	REGIONAL(1) の場合は F のみ
F|FB|V|VB	-	-	-	-	-	S	N	-	S	N	
RECSIZE(n)	I	I	I	I	S	I	C	I	I	C	RECSIZE または BLKSIZE (あるいはその両方) 指定する必要があります 連続
BLKSIZE(n)	I	I	I	I	-	I	N	I	I	N	両方を指定する必要がある
SCALARVARYING	-	O	O	O	-	O	O	O	O	O	ASCII データ・セットの場合は無効
CONSECUTIVE	D	D	D	-	-	-	O	-	-	O	VSAM ESDS の場合は指定可能
LEAVE|REREAD	O	O	O	-	-	-	-	-	-	-	
CTLASA|CTL360	-	O	O	-	-	-	-	-	-	-	ASCII データ・セットの場合は無効
GRAPHIC	O	-	-	-	-	-	-	-	-	-	
INDEXED	-	-	-	-	-	S	O	-	S	O	VSAM ESDS の場合は指定可能
KEYLOC(n)	-	-	-	-	-	O	-	-	O	-	
ORGANIZATION	D	-	-	-	-	-	-	-	-	-	
GENKEY	-	-	-	-	-	O	O	-	O	O	INPUT または UPDATE ファイルのみ (KEYED が必要)
REGIONAL(1)	-	-	-	S	-	-	-	S	-	-	
VSAM	-	-	-	-	-	-	S	-	-	S	
BKWD	-	-	-	-	-	-	O	-	-	O	
REUSE	-	-	-	-	-	-	O	-	-	O	OUTPUT ファイルのみ

ENVIRONMENT オプション											コメント

ノート：

1. INPUT 属性が指定されているファイルは PRINT 属性をとることはできません。
2. INDEXED 出力および REGIONAL 出力にはキーが必要です。

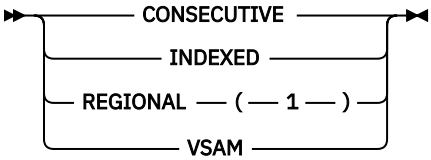
複数のデータ・セット編成に適用される ENVIRONMENT オプションについては、後続のトピックで説明します。 また、各オプションについては、オプションが適用される各データ・セット編成とともに後続のセクションで説明します。

関連情報

273 ページの『VSAM データセットの定義と使用』
この章では、レコード指向のデータ伝送のための VSAM （ Virtual Storage Access Method ）編成、 VSAM ENVIRONMENT オプション、他の PL/I データセット編成との互換性、および PL/I がサポートする 3 種類の VSAM データセット（エントリ・シーケンス、キー・シーケンス、相対レコード）のロードとアクセスに使用するステートメントについて説明します。

データ・セット編成オプション

以下のオプションは、データ・セット編成を指定します。



各オプションについては、適用されるデータ・セット編成の 項目で解説されています。

その他の *ENVIRONMENT* オプション

ブロック・サイズやレコード長など、整数引数を必要とする ENVIRONMENT オプションでは、定数または変数を使用できます。 変数は、添え字や修飾子があってはならず、FIXED BINARY(31,0) と STATIC の属性を持たなければならない。

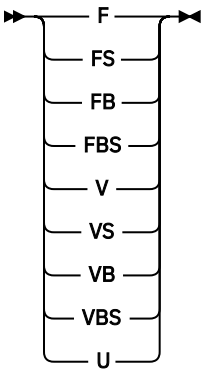
以下のリストは、 ENVIRONMENT のオプションと、それに相当する DCB のパラメーターを示している。

ENVIRONMENT option	DCB subparameter
Record format	RECFM[1]
RECSIZE	LRECL
BLKSIZE	BLKSIZE
CTLASA|CTL360	RECFM
KEYLENGTH	KEYLEN

注： [1]VS は DCB ではなく、 ENVIRONMENT オプションとして指定しなければならない。

レコード単位データ伝送のレコード・フォーマット

サポートされるレコード・フォーマットは、データ・セット編成により異なります。



レコードのフォーマットは、次のいずれかです。

表 *16.* レコード単位データ伝送のレコード・フォーマット

長さ	フォーマット	
固定長	F	非ブロック化
	FB	ブロック化
	FS	非ブロック化、標準
	FBS	ブロック化、標準
可変長	V	非ブロック化
	VB	ブロック化
	VS	スパン
	VBS	ブロック化、スパン
不定長	U	(ブロック化できない)

U フォーマット・レコードが可変長ストリングに読み込まれる 場合は、PL/I はストリングの長さを検索されたデータのブロック長に設定します。

上記のレコード・フォーマット・オプションは、VSAM データ・セットには適用されません。 VSAM データ・セットに関連付けられている ファイルにレコード・フォーマット・オプションを指定すると、そのオプションは無視されます。

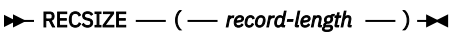
VS フォーマットのレコードは、連続編成のデータ・セットにのみ指定できます。

ストリーム指向データ伝送のレコード・フォーマット

ストリーム指向データ伝送のレコード・フォーマット・オプションについては、233 ページの『ストリーム指向データ伝送の用法』を参照してください。

RECSIZE オプション

RECSIZE オプションは、レコード長を指定します。



VSAM データ・セットに関連付けられているファイルに関しては、*record-length* は次の値の合計です。

1. データに必要な長さ

   可変長レコードおよび不定長レコードの場合は、この長さが最大長になります。

2. 必要な制御バイト

   可変長レコードには 4 バイトが必要です (レコード長 の接頭部用)。固定長レコードおよび不定長レコードでは 必要ありません。

VSAM データ・セットの場合、データ・セットが定義されるときに、 レコードの最大長と平均長がアクセス方式サービス・プログラム・ユーティリティーに指定されます。 検査の目的で RECSIZE オプションをファイル宣言に組み込む場合は、 レコードの最大長を指定する必要があります。 指定した RECSIZE がデータ・セットに定義されている値と 矛盾する場合は、UNDEFINEDFILE 条件が発生します。

record-length は、整数として、または属性 FIXED BINARY(31,0) STATIC を持つ変数として指定できます。

レコード長の値には、次のような規則があります。

最大長:
固定長、および不定長 (ASCII データ・セットを除く): 32760

UPDATE ファイルを持つ V フォーマット、ならびに VS および VBS フォーマット: 32756

入力ファイルおよび出力ファイルを持つ VS および VBS フォーマット: 16777215

ASCII データ・セット: 9999

VSAM データ・セット: 32761

注 : 32,756 バイトより長い VS 形式および VBS 形式のレコードの場合は、ENVIRONMENT の RECSIZE オプションに長さを指定し、DD ステートメントの DCB サブパラメーターに LRECL=X を指定する必要があります。 RECSIZE が INPUT または OUTPUT の許容最大値を超えると、レコード条件が発生するか、レコードが切り捨てられます。 UPDATE ファイルは LRECL=X ではサポートされていません。

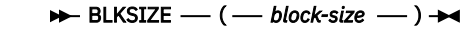
ゼロ値:
有効な値は、最初に、ファイルに関連付けられているデータ・セットの DD ステートメントで検索され、次にデータ・セット・ラベルで検索されます。

上記のいずれかで値を入手できない場合は、デフォルト・アクションが行われます (220 ページの『レコードフォーマット、 BLKSIZE、 RECSIZE デフォルト』参照)。

負の値:
UNDEFINEDFILE 条件が発生します。

BLKSIZE オプション

BLKSIZE オプションは、データ・セット上の最大ブロック・サイズを指定します。



block-size (ブロック・サイズ) は、次の値の合計です。

1. 次のいずれかの全長
   - 1 つのレコード
   - 1 つのレコードと、1 つまたは 2 つのレコード・セグメント
   - 複数のレコード
   - 複数のレコードと、1 つまたは 2 つのレコード・セグメント
   - 2 つのレコード・セグメント
   - 1 つのレコード・セグメント

   可変長レコードの場合、各レコードの長さまたは各レコード・セグメントの長さ には、レコードやレコード・セグメント用の 4 制御バイトが含まれています。

   前述のリストは、レコードおよびレコード・セグメントのオプション (固定長または可変長、ブロック化または非ブロック化、スパンまたは非スパン) の考えられるすべての組み合わせをまとめたものです。スパン・レコードのブロック・サイズを指定する場合、各レコードおよび各レコード・セグメントにはレコード長のための 4 制御バイトが必要であり、これらの数量は各ブロックに必要な 4 制御バイトに加算されるものであることに注意してください。

2. 必要なその他の制御バイト
   - 可変長ブロック・レコードには、4 バイトが必要です (ブロック・サイズの場合)。
   - 固定長レコードおよび不定長レコードには、追加の制御バイトは必要ありません。
3. 必要なブロック・プレフィックス・バイト（ASCII データセットのみ）

block-size は整数として、または属性 FIXED BINARY(31,0) STATIC を持つ変数として指定できる。

レコード長の値には、次のような規則があります。

最大長:
32760

ゼロ値:
z/OS 上で BLKSIZE を 0 (ゼロ) に設定すると、データ機能プロダクトがブロック・サイズを設定します。 詳しくは、220 ページの『レコードフォーマット、 BLKSIZE、 RECSIZE デフォルト』を参照してください。

負の値:
UNDEFINEDFILE 。

次のように、ブロック・サイズとレコード長の関係は、レコードのフォーマットによって 異なります。

FB-format or FBS-format
このブロック・サイズは、レコード長の倍数でなければなりません。

VB-format:
このブロック・サイズは、次の値の合計値以上でなければなりません。

1. 任意のレコードの最大長
2. 4 制御バイト

VS-format or VBS-format:
ブロック・サイズは、レコード長より小、等しい、またはより大きくすることができます。

ノート :

- BLKSIZE 、ブロックされていない（F または V フォーマットの）レコードを以下のいずれかの方法で使用する：
  - BLKSIZE オプションを指定し、 RECSIZE オプションは指定しない。 レコード長をブロック・サイズ (制御バイトまたは接頭語バイトを差し引く) と同じ値に設定し、 レコード・フォーマットを未変更のままにする。
  - BLKSIZE と RECSIZE の両方を指定し、2 つの値の関係が、使用するレコード・フォーマットのブロッキングに適合していることを確認する。 レコード・フォーマットを FB、VB に設定する (どちらでも該当する方)。
- FB フォーマットまたは FBS フォーマット・レコードでブロック・サイズが レコード長と等しい場合は、レコード・フォーマットは F に設定されます。
- BLKSIZE オプションは VSAM データセットには適用されず、指定しても無視される。

レコードフォーマット、 BLKSIZE、 RECSIZE デフォルト

非 VSAM データ・セットに対してレコード・フォーマット、ブロック・サイズ、レコード長を指定しないと、デフォルト・アクションが実行されます。

レコード形式:
関連付けられている DD ステートメントまたはデータ・セット・ラベル内で 検索が行われます。 値が検出されない場合は、UNDEFINEDFILE 条件が発生します。 ただし、ダミー・データ・セットまたはフォアグラウンド端末に 関連付けられているファイルの場合を除きます。その場合は、 レコード・フォーマットが U に設定されます。

ブロック・サイズあるいはレコード長:
ブロック・サイズまたはレコード長のいずれかが指定されている場合は、関連付けられている DD ステートメントまたはデータ・セット・ラベル内で、 もう一方の検索が行われます。 その検索により値が検出されても、値が指定オプションの値と矛盾している 場合は、UNDEFINEDFILE 条件が発生します。

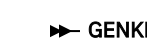
検索が成功しなかった場合は、指定オプションから値が導き出されます (制御バイトまたは 接頭部バイトを追加あるいは除去して)。

ブロック・サイズもレコード長も指定されていない場合は、UNDEFINEDFILE 条件 が発生します。ただし、ダミー・データ・ セットと関連付けられているファイルの場合は除きます。 その場合、BLKSIZE は 121 (F フォーマット・レコードや U フォーマット・レコード の場合)、または 129 (V フォーマット・レコードの場合) に設定されます。 フォアグラウンド端末に関連付けられているファイルの場合 は、RECSIZE は 120 に設定されます。

z/OS 上でデータ機能プロダクト (DFP) のシステム決定ブロック・サイズ機能を使用している場合は、割り当てられている装置タイプに最適なブロック・サイズが DFP により算出されます。 DD 割り当てまたは ENVIRONMENT ステートメント内で BLKSIZE(0) を指定する場合は、レコード長、レコード・フォーマット、および装置タイプを使って BLKSIZE が DFP により算出されます。

GENKEY オプション – キーの分類

GENKEY (総称キー) オプションは、INDEXED キー順データ・セット および VSAM キー順データ・セットにのみ適用されます。 このオプションを使用すれば、データ・セットに記録されているキーを分類したり、SEQUENTIAL KEYED INPUT ファイルや SEQUENTIAL KEYED UPDATE ファイルでキー・クラスに従ってレコードにアクセスしたりできます。



総称キーはキーのクラスを識別する文字ストリングです。このストリングで始まる すべてのキーはそのクラスのメンバーです。 例えば、記録済みキー "ABCD"、"ABCE"、および "ABDF" はすべて、総称キー "A" および "AB" によって識別されるクラスのメンバーです。 最初の 2 つは、クラス "ABC" のメンバーでもあります。 これら 3 つの記録済みキーは、それぞれクラス "ABCD"、"ABCE"、"ABDF" の固有メンバーであると見なすことができます。

GENKEY オプションを使用すると、 特定クラスのキーを持つ最初のレコードから、VSAM データ・セットを順次に読み取ることも更新することもできます。 また、INDEXED データ・セットの場合は、このオプションを使用すると、 特定のクラスのキーを持つ最初の非ダミー・レコードを順次に 読み取ることも更新することもできます。 READ ステートメントの KEY オプションに総称キーを 入れることにより、クラスを識別することができます。 KEY オプションを指定せずに、READ ステートメントで後続のレコードを読み とることができます。 キー・クラスの終わりに到達したときに、その旨の指摘は行われません。

KEY オプションを指定した READ ステートメントを使用することによって、特定クラスのキーを持つ最初のレコードを検索することができますが、KEYTO オプションは KEY オプションと同じステートメントで使用することができないため、レコードに組み込みキーがない限り、実際のキーを得ることはできません。

次の例では、3 バイトを超えているキーの長さが想定されます。

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (GENKEY);
          .
          .
          .
       READ FILE(IND) INTO(INFIELD)
                      KEY ('ABC');
          .
          .
          .
NEXT:  READ FILE (IND) INTO (INFIELD);
          .
          .
          .
       GO TO NEXT;
```

最初の READ ステートメントによって、「ABC」で始まるキー を持ったデータ・セット内の最初の非ダミー・レコードが、INFIELD に 読み込まれることになります。 第二の READ ステートメントが実行されるごとに、 次に高位のキーを持つ非ダミー・レコードが取り出されます。 2 番目の READ ステートメントが繰り返し実行されると、より高位のキー・クラスからレコードが読み取られます。 これは、キー・クラスの終わりに到達したときに標識が指定されていないためです。 特定クラスのキーを超えて読み取りを続けたくない場合は、各自の責任で各キーを検査してください。 最初の READ ステートメントをもう一度実行すると、 ファイルがキー・クラス「ABC」の最初のレコードの位置に再配置されます。

指定クラス内のキーを持つレコードがデータ・セットにない場合、 または指定クラスのキーを持つレコードがすべてダミー・レコード の場合は、KEY 条件が発生します。 そのあと、データ・セットは上位のキーを持つ次のレコードに、 あるいはファイルの終わりに位置付けられます。

GENKEY オプションの有無によって、KEYLEN サブパラメーターで 指定されているキー長より短いソース・キーを 提供する READ ステートメントの実行が影響を受けます。 KEYLEN サブパラメーターは、索引付きデータ・セットを 定義する DD ステートメント内にあります。 GENKEY オプションを指定すると、それによってソース・キーが総称キーと解釈され、 キーがソース・キーで始まるデータ・セット内の最初の非ダミー・レコードに、 そのデータ・セットが位置付けられることになります。 GENKEY オプションを指定しないと、 指定したキー長になるよう、READ ステートメントの短いソース・キーの 右側にブランクが埋め込まれ、データ・セットは、この埋め込まれたキーを 持ったレコード (このようなレコードが存在する場合) に位置付けられます。 WRITE ステートメントの場合は、短いソース・キーには常に ブランクで埋め込まれます。

GENKEY オプションを使用しても、キー長が指定キー長以上であるソース・キーを提供 してもその結果に影響はありません。 必要に応じて右側が切り捨てられるソース・キーは、 特定のレコード (キーがそのクラスの唯一の メンバーだと考えられるレコード) を識別します。

SCALARVARYING オプション - 可変長ストリング

SCALARVARYING オプションは 可変長ストリングの入出力について使用します。 このオプションではどのフォーマットのレコードで 使用することもできます。

ストレージを可変長ストリング用に割り当てるとき、コンパイラーは、 ストリングの現行長を指定するための 2 バイト接頭部を組み込みます。 エレメント可変長ストリングの場合は、そのファイルに SCALARVARYING が指定されている 場合にだけ、この接頭部が出力時に組み込まれるか、入力時に認識されます。

位置指定モード・ステートメント (LOCATE および READ SET) を使って、エレメント可変長ストリングを持つデータ・セットを作成し読み取る場合は、SCALARVARYING を指定して長さ接頭部の存在を認識させる必要があります。これは、バッファーの位置を指定するポインターは、常に長さ接頭部の開始位置を指すと想定されるからです。

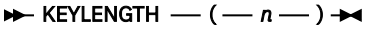
SCALARVARYING を指定して、エレメント可変長ストリングが送信される場合は、 長さ接頭部を組み込むためにレコード長に 2 バイトを与える必要があります。

SCALARVARYING を使用して作成されるデータ・セットは、SCALARVARYING を指定しているファイルだけがアクセスするようにします。

同じファイルに SCALARVARYING と CTLASA/CTL360 を指定することはできません。それを行うと、最初のデータ・バイトがあいまいになるからです。

KEYLENGTH オプション

KEYLENGTH オプションは、KEYED ファイルに対して記録されるキーの長さを指定します。 INDEXED ファイルにも KEYLENGTH オプションを指定することができます。

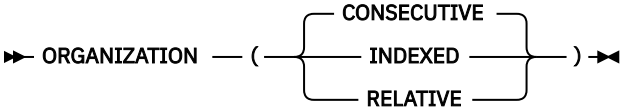


n
これは、KEYED ファイルに対して記録されるキーの長さを指定します。

検査目的で VSAM ファイル宣言に KEYLENGTH オプションを組み込む場合に、そのオプションで指定したキー長が、データ・セットに対して定義されている値と矛盾すると、UNDEFINEDFILE 条件が発生します。

ORGANIZATION オプション

ORGANIZATION オプションは、 PL/I ファイルに関連するデータセットの構成を指定する。



CONSECUTIVE
該当のファイルが連続データ・セットに関連付けられていることを表します。 連続ファイルは、ネイティブ・データセットか、 VSAM, ESDS, RRDS、または KSDS データセットのどちらかになる。

RELATIVE
該当のファイルが相対データ・セットに関連付けられていることを表します。 RELATIVE は、データセットにキーが記録されていないレコードが含まれていることを指定する。 相対ファイルは VSAM 直接データセットである。 相対キーの範囲は、1 から nnnn までです。

PL/I レコード入出力で使用されるデータ・セットのタイプ

RECORD 属性を持ったデータ・セットは、データがプログラム変数に入っているとおりに 補助記憶域との間で送受信されるレコード単位データ伝送によって 処理されます。データ変換は行われません。 データ・セット内のレコードは、プログラム内の各変数に対応しています。

223 ページの表 17 に、PL/I レコード入出力で使用できるさまざまなデータ・セット・タイプで使用可能な機能を示します。

表 *17. PL/I* レコード入出力で使用できるデータ・セット・タイプの比較

	VSAM KSDS	**VSAM ESDS**	**VSAM RRDS**	**INDEXED**	**CONSECUTIVE**	**REGIONAL (1)**
順序	キー オーダー	エントリー オーダー	NUM- 番号付き	キー オーダー	エントリー オーダー	担当 地域
DEVICES	DASD	DASD	DASD	DASD	DASD、 カード等	DASD
ACCESS						
1 キー	123	123	123	12	2	12
2 順次						
3 逆方向						
Alternate (代替) 索引 アクセス 同上	123	123	いいえ	いいえ	いいえ	いいえ
どのように 拡張	で 新規 キー	時刻 終了	内 空 スロット	で 新規 キー	時刻 終了	内 空 スロット

表 17. *PL/I* レコード入出力で使用できるデータ・セット・タイプの比較 (続き)

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
DELETION						
	はい、1	いいえ	はい、1	はい、2	いいえ	はい、1

1
スペースは再使用可能

2
スペースは再使用できない

以下の各セクションは、各種データ・セットでのレコード入出力データ・セットの用法を説明しています。

- 233 ページの『第 10 章 連続データ・セットの定義と使用』
- 263 ページの『第 13 章 領域データ・セットの定義と使用』
- 273 ページの『第 14 章 VSAM データセットの定義と使用』

z/OS UNIX での環境変数の設定

いくつかの環境変数は、z/OS UNIX で使用するために設定したりエクスポートしたりできます。 すべてのユーザーがアクセスできるように環境変数をシステム全体用に設定するには、 サブセクションで推奨されている行をファイル /etc/profile に追加します。 ある特定ユーザーだけに環境変数を設定するには、該当するユーザーのホーム・ディレクトリーにあるファイル .profile に その環境変数を追加します。

変数は、次にユーザーがログオンしたときに設定されます。

次の例に、環境変数の設定方法を示します。

```
LANG=ja_JP
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
LIBPATH=/home/joe/usr/lib:/home/joe/mylib:/usr/lib
export LANG NLSPATH LIBPATH
```

前の例の最後のステートメントを使用する代わりに、上記の各行 (export LANG=ja_JP...) に export を追加することができます。

環境変数の現在の設定を調べるには、 ECHO コマンドを使う。 BYPASS の値を定義するには、以下の 2 つの例のどちらかを使うことができる：

```
echo $LANG

echo $LIBPATH
```

z/OS UNIX での PL/I 標準ファイル (SYSPRINT および SYSIN)

デフォルトにより、SYSIN は stdin から 読み取られ、SYSPRINT は stdout に送られます。 関連付けを変更したい場合は、OPEN ステートメントの TITLE オプションを 使うか、データ・セットまたは別の装置に名前を指定する DD_DDNAME 環境変数を 設定する必要があります。

関連情報

224 ページの『z/OS UNIX での環境変数の設定』
いくつかの環境変数は、z/OS UNIX で使用するために設定したりエクスポートしたりできます。 すべてのユーザーがアクセスできるように環境変数をシステム全体用に設定するには、 サブセクションで推奨されている行をファイル /etc/profile に追加します。 ある特定ユーザーだけに環境変数を設定するには、該当するユーザーのホーム・ディレクトリーにあるファイル .profile に その環境変数を追加します。

z/OS UNIX での標準入力装置、標準出力装置、および標準エラー装置のリダイレクト

標準入力装置、標準出力装置、および標準エラー装置をファイルにリダイレクトできます。

例えば、次のようなプログラムでリダイレクトを使用できます。

```
Hello2: proc options(main);
  put list('Hello!');
end;
```

プログラムをコンパイルしてリンクした後で、コマンド行で次のコマンドを入力して、そのプログラムを呼び出すことができます。

```
hello2 > hello2.out
```

stdout と stderr を 1 つのファイルにまとめる場合は、次のコマンドを入力します。

```
hello2 > hello2.out 2>&1
```

表示ステートメントの場合と同様に、(より大) 記号を使用すると、 その後ろに指定されているファイル、この場合は hello2.out に出力がリダイレクトされます。 これは、'Hello' という語がファイル hello2.out に書き込まれることを意味します。 PRINT 属性はデフォルトで SYSPRINT に適用されるため、出力にはプリンター制御文字も組み込まれるということに注意してください。

READ ステートメントは stdin からのデータにもアクセスできます。

を活用する LBI

large block interface (LBI) を大きなブロック・サイズに使用することで、テープ・データ・セットのパフォーマンスを向上させることができる。 LBI をサポートする磁気テープ装置の場合、PL/I は、ブロック・サイズが 32760 を超えることができるこの機能を使用します。

LBI は、スパンファイルまたは UPDATE 。 さらに、PL/I は U フォーマット・ファイルに対する LBI サポートを提供しません。 LBI は 31 ビット・モードでのみサポートされます。

32760 を超えるブロック・サイズを使用すると、別のシステムでテープを読み取れなくなる場合があります。 32760 より大きなブロック・サイズで作成したテープは、32760 より大きなブロック・サイズをサポートする磁気テープ装置を持つシステムでしか読み取ることができません。 ファイル、装置、またはオペレーティング・システム・レベルに関して、指定したブロック・サイズが大きすぎると、ランタイム・メッセージが出されます。

第 9 章 ライブラリーの使用

z/OS オペレーティング・システムでは、区分データ・セット、区分データ・セット/拡張、およびライブラリー の 3 つの用語は同義であり、他のデータ・セット (通常、ソース・モジュール、オブジェクト・モジュール、またはロード・モジュールの形のプログラム) の保管に使用できるタイプのデータ・セットを指します。

ライブラリーは、直接アクセス・ストレージに保管する必要があり、 全体が 1 つのボリューム内に入っていなければなりません。 ライブラリーには、連続して編成された、メンバーと呼ばれる独立した データ・セットが入っています。 各メンバーには、ライブラリーの一部であるディレクトリーに 保管される 8 文字を超えない長さの固有名が付いています。 1 つのデータ・セット・ラベルだけが維持されるので、1 つのライブラリーに属するすべてのメンバーは同じデータ特性を持っている必要があります。

ディレクトリー内に新規項目を入れる十分なスペースが残っていないか、または メンバーそのもののスペースが足りなくなるまでは、メンバーを 個々に作成することができます。 メンバーの名前を指定すれば、メンバーを個別にアクセスすることができます。

DD ステートメントまたはそれらに対応する会話型での同等機能を使用することにより、メンバーの作成およびアクセスを行うことができます。

メンバーの削除は、 IBM ユーティリティ・プログラム IEHPROGM を使って行うことができます。 このプログラムは、ディレクトリからメンバ名を削除し、メンバにアクセスできないようにします。しかし、ライブラリを再作成するか、 IBM ユーティリティ・プログラム IEBCOPY などを使用して未使用領域を圧縮しない限り、メンバが占有している領域を再び使用することはできません。 DD 文の **DISP** パラメータを使用してメンバを削除しようとすると、データ・セット全体が削除されます。

ライブラリーのタイプ

ライブラリーのタイプには、システム・プログラム・ライブラリー、システム・プロシージャー・ライブラリー、専用プログラム・ライブラリーなどがあります。

PL/I プログラムでは、 次のタイプのライブラリーを使用することができます。

- システムプログラムライブラリ SYS1.LINKLIB または同等のもの

  このライブラリーには、コンパイラーやリンケージ・エディターなどのすべてのシステム処理プログラムを収容することができます。
- 専用プログラム・ライブラリー

  これらのライブラリーには、通常、ユーザーが作成したプログラムが収容されます。 多くの場合、一時専用ライブラリーを作成すれば、 リンケージ・エディターからのロード・モジュール出力を、 同一ジョブ内の後半のジョブ・ステップで実行するまでの間 保管しておけるので便利です。 一時ライブラリーは、ジョブの終了時に削除されます。 専用ライブラリーは、自動ライブラリー呼び出し用にも リンケージ・エディターやローダーによって使用されます。
- システムプロシージャライブラリ SYS1.PROCLIB または同等のもの

  このライブラリーには、ユーザーのシステム用としてカタログされているジョブ制御プロシージャーが入っています。

ライブラリーの使用

PL/I プログラムは、ライブラリーを直接使用することができます。

ユーザーが新たにメンバーをライブラリーに追加しようとする と、オペレーティング・システムは、その関連ファイルがクローズ されるときに、データ・セット名の一部として指定されているメンバー名を使って 新規メンバーのディレクトリー項目を作成します。

ユーザーがライブラリーのメンバーにアクセスする 場合、オペレーティング・システムは、 ユーザーがデータ・セット名の一部として指定したメンバー名から そのディレクトリー項目を見つけ出すことができます。

© Copyright IBM Corp. 1999, 2025

同一ライブラリーの複数のメンバーを 1 つの PL/I プログラムで処理することはできますが、このようなファイルは同時に 1 つしか出力としてオープンすることはできません。 DD ステートメントに入っているメンバー名を指定することにより、 異なるメンバーにアクセスすることができます。

ライブラリーの作成

ライブラリーを作成するには、ライブラリーの作成に必要となる情報が含まれている DD ステートメントをジョブ・ステップに組み込みます。

ライブラリーの作成時に必要となる情報については、228 ページの表 18 を参照してください。 ライブラリーの作成に必要な情報は、SPACE パラメーターを除けば、 連続して編成されたデータ・セット (252 ページの『レコード入出力を使用したファイルの定義』参照) の場合とほとんど同じです。

表 *18.* ライブラリー作成時に必要な情報

必須情報	**DD ステートメントのパラメーター**
使用する装置のタイプ	UNIT=
ライブラリーを入れるボリュームの通し番号	VOLUME=SER
ライブラリー名	DSNAME=
ライブラリーのスペース必須容量	SPACE=
ライブラリーの後処理	DISP=

SPACE パラメーター

DD 文の SPACE パラメータを使って、作成するライブラリに必要な容量を指定することができます。

ライブラリを定義する DD 文の SPACE パラメータは、必ずこの形式でなければならない：

```
SPACE=(units,(quantity,increment,directory))
```

3 番目の項目 (increment) はオプションであり、省略する場合はコンマを残すことで、その項目が省略されていることを示すことができます。 最後の項目 (割り振られるディレクトリー・ブロックの数を指定する) は必須です。

ライブラリーに必要な補助記憶域の大きさは、 保管するメンバーの数とサイズと、 メンバーがどのくらい頻繁に追加または置換されるかによって異なります。 (削除されたメンバーのスペースは解放されません。) 必要なディレクトリー・ブロック数は、メンバー数および別名数によって異なります。 SPACE パラメータにインクリメンタルな数量を指定することで、データセットの作成時または新メンバーの追加時に必要な場合、オペレーティングシステムがより多くのスペースを取得できるようにすることができる。ただし、ディレクトリブロック数は作成時に固定され、増やすことはできない。

例

次の例では、DD 文がジョブ・スケジューラに、ボリューム・シリアル番号 3412 の DASD の 5 シリンダーを新しいライブラリ名 ALIB に割り当て、この名前をシステム・カタログに入力するよう要求している。 SPACE パラメーターの最後の項は、データセットに割り当てられたスペースの一部を、10 個のディレクトリブロック用に確保するよう要求する。

```
//   PDS DD UNIT=SYSDA,VOL=SER=3412,
//   DSNAME=ALIB,
//   SPACE=(CYL,(5,,10)),
//   DISP=(,CATLG)
```

ライブラリー・メンバーの作成と更新

ライブラリー・メンバーを作成および更新するときは、このトピックにあるガイドラインに従う必要があります。

各ライブラリー・メンバーは、同じ特性を持っていなければなりません。 同じ特性メンバーを持っていない場合は、あとのメンバー検索 が難しくなります。 ボリューム目次(VTOC)には、各メンバーに 1 つではなく、ライブラリのデータセット制御ブロック(DSCB)が 1 つだけ含まれるため、同一の特性が必要である。 PL/I プログラムを使ってメンバーを作成する場合は、オペレーティング・システムによりディレクトリー項目が作成されます。つまり、ユーザーはユーザー・データ・フィールドに情報を入力することはできません。

ライブラリとメンバを同時に作成する場合、DD 文には 228 ページの『ライブラリーの作成』 にリストされているパラメータをすべて含める必要があります（ただし、データ・セットを一時的に使用する場合は **DISP** パラメータを省略できます)。 **DSNAME** パラメータには、括弧内にメンバー名を含める必要がある。 例えば、 DSNAME=ALIB(MEM1) は、データセット ALIB のメンバー MEM1 に名前を付けます。 メンバがリンケージ・エディタによってライブラリに配置される場合は、 DSNAME パラメータにメンバ名を含める代わりに、リンケージ・エディタの NAME 文または NAME コンパイル時オプションを使用できます。 また、メンバーの特性 (レコード・フォーマットなど) を DCB パラメーターまたは PL/I プログラムに記述する必要があります。 これらの特性は、そのデータ・セットに追加される他のメンバーにも適用されま す。

既存のライブラリに追加するメンバを作成する場合、SPACE パラメータは必要ありません。 元のスペース割り振りは、個々のメンバーに 対してではなく、ライブラリー全体に対して適用されるからです。 さらに、メンバーの特徴を記述する必要はない。なぜなら、それらはすでにライブラリーの DSCB に記録されているからである。

1 つのジョブ・ステップで複数のメンバーをライブラリーに 追加するには、各メンバーごとに DD ステートメントを組み込み、そして そのライブラリーを参照するファイルをクローズしてから 次のファイルをオープンするようにしなければなりません。

例: コンパイルされたオブジェクト・モジュール用の新規ライブラリーの作成

この例では、カタログ化されたプロシージャー IBMZC を使用して、単純な PL/I プログラムをコンパイルし、オブジェクト・モジュールを EXLIB という新しいライブラリに配置します。 新しいライブラリを定義し、オブジェクト・モジュールに名前を付ける DD 文は、カタログ化されたプロシージャの DD 文 SYSLIN 。

PL/I プログラムは、 TIME 組み込み関数によって生成される文字列の形式で 2 つの値が与えられると、その差をミリ秒単位で返す関数手続きである。

コンパイルされたオブジェクト・モジュール用の新規ライブラリーの作成

```
//OPT10#1 JOB
//TR        EXEC  IBMZC
//PLI.SYSLIN  DD  UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//      SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSIN   DD  *
   ELAPSE:  PROC(TIME1,TIME2);
     DCL (TIME1,TIME2) CHAR(9),
         H1 PIC '99' DEF TIME1,
         M1 PIC '99' DEF TIME1 POS(3),
         MS1 PIC '99999' DEF TIME1 POS(5),
         H2 PIC '99' DEF TIME2,
         M2 PIC '99' DEF TIME2 POS(3),
         MS2 PIC '99999' DEF TIME2 POS(5),
         ETIME FIXED DEC(7);
     IF H2<H1 THEN H2=H2+24;
     ETIME=((H2*60+M2)*60000+MS2)-((H1*60+M1)*60000+MS1);
     RETURN(ETIME);
   END ELAPSE;
/*
```

例: ロード・モジュールの既存ライブラリーへの配置

この例では、 PL/I プログラムをコンパイルしてリンク編集し、既存のライブラリ HPU8.CCLM にロード・モジュールを配置するために、カタログ化された手続き IBMZCL を使用しています。

ロード・モジュールの既存ライブラリーへの配置

```
//OPT10#2  JOB
//TRLE      EXEC  IBMZCL
//PLI.SYSIN   DD  *
   MNAME:  PROC  OPTIONS(MAIN);
     .
```

```
      .
      .
      program
      .
      .
      .

   END MNAME;
/*
//LKED.SYSLMOD  DD  DSNAME=HPU8.CCLM(DIRLIST),DISP=OLD
```

例: ライブラリー・メンバーの更新

PL/I プログラムを使って、ライブラリーのメンバー内の 1 つ以上の レコードを追加または削除するには、そのライブラリー内の別の部分でそのメンバーを全部作成し直さなければなりません。 メンバーがそれまで占有していたスペースは再度使用することができないため、これはあまり経済的な提案とは言えません。ユーザーの PL/I プログラム内でファイルを 2 つ使う必要が ありますが、2 つとも同じ DD ステートメントと 関連付けることができます。

ライブラリ・メンバの更新」に示すプログラムは、「PL/I プログラムでライブラリ・メンバを作成する」に示すプログラムによって作成されたメンバを更新する。 このプログラムは、ブランクだけのレコードを除き、 元のメンバーのレコードをすべてコピーします。

PL/I プログラムでライブラリ・メンバーを作成する

```
//OPT10#3  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  NMEM:  PROC OPTIONS(MAIN);
     DCL IN FILE RECORD SEQUENTIAL INPUT,
         OUT FILE RECORD SEQUENTIAL OUTPUT,
         P POINTER,
         IOFIELD CHAR(80) BASED(P),
         EOF BIT(1) INIT('0'B);
     OPEN FILE(IN),FILE (OUT);
     ON ENDFILE(IN) EOF='1'B;
     READ FILE(IN) SET(P);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
     WRITE FILE(OUT) FROM(IOFIELD);
     READ FILE(IN) SET(P);
     END;
     CLOSE FILE(IN),FILE(OUT);
  END NMEM;
/*
//GO.OUT   DD  UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//     DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//     DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
  MEM:  PROC OPTIONS(MAIN);
        /* this is an incomplete dummy library member */
```

ライブラリー・メンバーの更新

```
//OPT10#4  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  UPDTM: PROC OPTIONS(MAIN);
     DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
         EOF BIT(1) INIT('0'B),
         DATA CHAR(80);
     ON ENDFILE(OLD)  EOF = '1'B;
     OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');
     READ FILE(OLD) INTO(DATA);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
     IF DATA=' ' THEN ;
     ELSE WRITE FILE(NEW) FROM(DATA);
     READ FILE(OLD) INTO(DATA);
     END;
   CLOSE FILE(OLD),FILE(NEW);
  END UPDTM;
/*
//GO.OLD DD  DSNAME=HPU8.ALIB(NMEM),DISP=(OLD,KEEP)
```

ライブラリー・ディレクトリーからの情報の取り出し

ライブラリー・ディレクトリーは、 データ・セットの先頭に置かれる一連のレコード (項目) です。 1 つ以上のディレクトリー項目が各メンバーに存在します。 各項目には、メンバー名、ライブラリー内の メンバーの相対アドレス、および可変量のユーザー・データが入っています。

ユーザー・データは、メンバーを作成したプログラムによって挿入される情報です。 リンケージ・エディターで作成されたメンバー (ロード・モジュール) を参照する項目は、 システムのマニュアルに記載されている標準形式のユーザー・データを組み込みます。

PL/I プログラムを使ってメンバーを作成するときには、オペレーティング ・システムがユーザーに代わってディレクトリー項目を作成するので、 ユーザーはユーザー・データを書き込むことはできません。 ただし、アセンブラー言語マクロ命令を使用すると、メンバーを作成したり、独自の ユーザー・データを作成することができます。 このためにマクロ命令を使用する方法は、データ管理資料に 説明されています。

第 10 章 連続データ・セットの定義と使用

この章では、連続データセットの構成と、ストリームおよびレコード指向のデータ伝送のための連続データセットを定義する ENVIRONMENT オプションについて説明する。 その次に、伝送タイプごとに、連続データ・セットの作成、 アクセスおよび更新方法についても説明します。

連続編成のデータ・セット内では、各レコードは連続する 物理的な位置のみに基づいて編成されます。 データ・セットが作成されるときは、レコードは提示される順番で 連続的に書き込まれます。 なお、レコードは、レコードが書き込まれた順序でのみ検索できます。 連続するデータセットに有効なファイル属性と ENVIRONMENT オプションについては、 215 ページの表 15 を参照のこと。

ストリーム指向データ伝送の用法

このセクションでは、STREAM 属性を持つ PL/I ファイルで使用するデータセットの定義方法について説明します。 本書では、ENVIRONMENT のオプションと、データセットの作成およびアクセス方法について説明します。 また、データ・セットの作成、アクセス時に使用する DD ステートメント の必須パラメーターを表にまとめ、文章による記述を 説明するために PL/I プログラムの例もいくつか掲載しています。

STREAM 属性を持つデータ・セットは、ストリーム指向のデータ伝送によって処理される。これにより、PL/I プログラムは、ブロックやレコードの境界を無視し、データ・セットを文字やグラフィック形式のデータ値の連続ストリームとして扱うことができる。

ストリーム指向データ伝送用のデータ・セットを作成し、それにアクセスするには、「*PL/I Language Reference*」で説明しているリスト指示、データ指示、および編集指示の入出力ステートメントを使用します。

出力の場合、PL/I は必要に応じてデータ項目をプログラム変数から文字の形に変換し、 文字またはグラフィックスのストリームを、 データ・セットに伝送するためにレコードに構築します。

入力の場合、PL/I はデータ・セットからレコードを取り出し、 ユーザー・プログラムが要求した複数のデータ項目に分割し、さらに それをプログラム変数に割り当てるのに適した形に変換します。

ストリーム指向データ伝送は、グラフィック・データの読み取りと書き込みに使用できます。 適切なプログラミング・サポートがあれば、グラフィックスを表示し、印刷し、 入力することができる端末、プリンター、およびデータ入力装置があります。 ユーザーのデータが使用する装置または印刷ユーティリティー・プログラム で受け入れられているフォーマットになっているかどうか確認する必要があります。

ストリーム入出力を用いたファイルの定義

ストリーム指向データ伝送用のファイルはファイル宣言で定義できます。

```
DCL filename FILE STREAM
             INPUT | {OUTPUT [PRINT]}
             ENVIRONMENT(options);
```

デフォルト・ファイル属性については、215 ページの表 15 を参照してください。 FILE 属性については、「*PL/I Language Reference*」を参照してください。 PRINT 属性について詳しくは、241 ページの『ストリーム入出力による PRINT ファイルの使用』を参照してください。 ENVIRONMENT 属性のオプションについては、234 ページの『ENVIRONMENT オプションの指定』を参照してください。

PL/I 動的割り振りを使用したストリーム・ファイルの定義

ストリーム・ファイルを定義するには、 DD 文、環境変数、または OPEN 文の TITLE オプションを使用する。

環境変数または TITLE オプションを使用する場合、名前は大文字でなければならない。 次のいずれかの方法で MVS データ・セットを指定してください。

- DSN(データセット名)
- DSN(データセット名 (メンバー名))

© Copyright IBM Corp. 1999, 2025

data-set-name は完全修飾でなければなりません。一時データセットに設置できません。 たとえば、 &から開始できません。

次のように zFS ファイルを指定してください。

```
PATH(absolute-path-name)
```

以下の属性は、 DSN キーワードの後に任意の順序で指定できます：

```
NEW, OLD, SHR, or MOD
TRACKS or CYL
SPACE(n,m)
VOL(volser)
UNIT(type)
KEEP, DELETE, CATALOG, or UNCATALOG
STORCLAS(storageclass)
MGMTCLAS(managementclass)
DATACLAS(dataclass)
CLOSE
NORECALL
RELEASE
```

注 : 環境変数または OPEN ステートメントの TITLE オプションを使用して PDS または PDSE を作成することはできませんが、既存の PDS または PDSE に新しいメンバーを作成することはできます。

CLOSE, NORECALL、および RELEASE 属性は、 BPXWDYN インターフェースの同じ属性にマッピングされる。 *z/OS* と *SVC 99* がサポートするサービスの詳細については、 *Using REXX* and *z/OS UNIX System Services*BPXWDYN を参照してください。

ENVIRONMENT オプションの指定

このトピックでは、ストリーム指向データ伝送およびレコード単位データ伝送に対して連続データ・セットを定義する ENVIRONMENT オプションについて説明します。

215 ページの表 15 に、ENVIRONMENT オプションの要約があります。 以下のオプションをストリーム指向データ伝送に適用できます。

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
F|FB|FS|FBS|V|VB|VS|VBS|U
RECSIZE(record-length)
BLKSIZE(block-size)
GRAPHIC
LEAVE|REREAD
```

上記オプションをストリーム指向データ伝送に対して指定する方法については、以下のトピックを参照してください。

オプション	参照先のトピック
CONSECUTIVE	234 ページの『CONSECUTIVE』
F|FB|FS|FBS|V|VB|VS|VBS|U	235 ページの『レコード・フォーマット・オプション』
RECSIZE	235 ページの『RECSIZE』
BLKSIZE	219 ページの『BLKSIZE オプション』
GRAPHIC	236 ページの『GRAPHIC』
LEAVE|REREAD	256 ページの『LEAVE|REREAD』

CONSECUTIVE

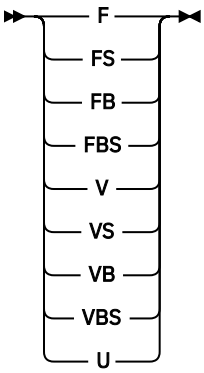
STREAM ファイルは CONSECUTIVE データ・セット編成を持っていなくてはなりませんが、CONSECUTIVE はデフォルトのデータ・セット編成なので、これを ENVIRONMENT オプション内に指定する必要はありません。

STREAM ファイルの CONSECUTIVE オプションは、210 ページの『データ・セットの編成』に記載 されているものとまったく同じです。

レコード・フォーマット・オプション

ストリーム指向データ伝送では、レコード境界は無視されますが、データ・セットを作成する場合、レコード・フォーマットは重要な意味を持ちます。 これは、レコード・フォーマットがデータ・セットが占有するストレージの容量と、 データを処理するプログラムの効率に影響するためだけでなく、データ・セットが後で、 レコード単位データ伝送でも処理できるようにするためです。

いったんレコード・フォーマットを指定したならば、 ストリーム指向データ伝送を使う限り、レコードおよびブロックを 意識する必要はありません。 データ・セットは、行に配置された一連の文字またはグラフィックスと見なすことができま す。また、SKIP オプションまたはフォーマット項目 (PRINT ファイルでは、PAGE オプション、LINE オプションおよびフォーマット項目) を使って、新しい行を選択する ことができます。



レコードのフォーマットは、以下のいずれかにすることができます。 詳しくは、208 ページの『レコード・フォーマット』を参照してください。

表 *19.* レコード・フォーマット

レコード長	レコード	ブロック/ブロック解除オプション
固定長	F FB FS FBS	非ブロック化 ブロック化 非ブロック化、標準 ブロック化、標準
可変長	V VB VS VBS	非ブロック化 ブロック化
不定長	U	(ブロック化できない)

なお、レコードのブロック化も非ブロック化も自動的に行われます。

RECSIZE

ストリーム指向データ伝送の場合の RECSIZE は 214 ページの『ENVIRONMENT 』に 記載されているものと同じです。 また、OPEN ステートメントの LINESIZE オプションで指定された値 は、RECSIZE オプショ

ンで指定された値を指定変更します。 LINESIZE については、「*PL/I Language Reference*」で説明されています。

グラフィックスのリスト指示伝送およびデータ指示伝送についてのレコード・サイズに関する 追加の考慮事項は、「*PL/I Language Reference*」に記載されています。

レコード・フォーマット、BLKSIZE および RECSIZE のデフォルト値

ENVIRONMENT 属性や関連 DD ステートメント/データ・セット・ラベルでレコード・フォーマット、BLKSIZE、RECSIZE オプションのいずれも指定しないと、PL/I がデフォルト値を決定します。

- 入力ファイル:

  デフォルトは、220 ページの『レコードフォーマット、 BLKSIZE、 RECSIZE デフォルト』で説明されているレコード単位データ伝送 と同様に適用されます。

- 出力ファイル

  レコード・フォーマット
  VB フォーマットに設定されます。

  レコード長
  指定した、またはデフォルトの LINESIZE の値が使用されます。

  – PRINT ファイル:

  - F、FB、FBS、または U: 行サイズ + 1
  - V または VB: 行サイズ + 5

  – 非 PRINT ファイル:

  - F、FB、FBS、または U: linesize
  - V または VB: linesize + 4

  ブロック・サイズ
  SYSOUT に関連付けられたファイル:

  – F、FB、または FBS: レコード長

  – V または VB: レコード長 + 4

  新規/一時データ・セット:

  – DFP によって決定される最適ブロック・サイズ

GRAPHIC

編集指示入出力の GRAPHIC オプションを指定します。

入力データや出力データにグラフィックスが含まれているのに、GRAPHIC オプションが 指定されていなければ、リスト指示入出力およびデータ指示入出力の ERROR 条件が発生します。

編集指示入出力で GRAPHIC オプションを指定すると、出力時に DBCS 変数と定 数の左右に区切り文字が追加され、グラフィック入力時にも左右に区切り文字が付け 加えられます。 GRAPHIC オプションを指定しない場合は、 出力データの左右に区切り文字は追加されず、 また、入力のグラフィックスにも左右の区切り文字は必要ありません。 また、GRAPHIC オプションを指定した場合は、入力データの左右に区切り文字が ないと、ERROR 条件が発生します。

グラフィック・データ・タイプについて、また編集指示入出力の G フォーマット項目については、「*PL/I Language Reference*」を参照してください。

ストリーム入出力によるデータ・セットの作成

データ・セットを作成するには、ユーザーの PL/I プログラム内で、 またはデータ・セットを定義する DD ステートメント内で、特定の情報を オペレーティング・システムに与える必要があります。

z/OS UNIX の場合は、以下のいずれかの方法を使用して追加情報を指定します。

- TITLE OPEN
- DD_DDNAME 環境変数
- ENVIRONMENT 属性

以下のトピックでは、データ・セットを作成するために指定しなければならない必須情報と、指定可能な一部のオプション情報について説明します。

必須情報

ご使用のアプリケーションで STREAM ファイルを作成する場合、PL/I は、そのファイルの行サイズ値を導き出します。

PL/I は、次のいずれかのソースから優先順位順 (降順) に行サイズ値を導き出します。

- OPEN ステートメントの LINESIZE オプション
- ENVIRONMENT 属性の RECSIZE オプション
- OPEN ステートメントの TITLE オプションの RECSIZE オプション
- DD_DDNAME 環境変数の RECSIZE オプション
- PL/I 提供のデフォルト値

LINESIZE の値が与えられていて、RECSIZE の値が与えられていない場合は、PL/I は、次のようにレコード長を導き出します。

- V フォーマット PRINT ファイルの場合、値は LINESIZE + 5。
- V フォーマット非 PRINT ファイルの場合、値は LINESIZE + 4。
- F フォーマット PRINT ファイルの場合、値は LINESIZE + 1。
- 上記以外の場合はすべて、値は LINESIZE。

LINESIZE の値が与えられておらず、RECSIZE の値が与えられている場合は、PL/I は、次のように RECSIZE から行サイズの値を導き出します。

- V フォーマット PRINT ファイルの場合、値は RECSIZE - 5。
- V フォーマット非 PRINT ファイルの場合、値は RECSIZE - 4。
- F フォーマット PRINT ファイルの場合、値は RECSIZE - 1。
- 上記以外の場合はすべて、値は RECSIZE。

LINESIZE も RECSIZE も与えられていない場合は、PL/I は、ファイル属性および関連付けられたデータ・セットのタイプに基づいて、デフォルトの行サイズの値を決定します。 PL/I が、適切なデフォルト行サイズを提供できない場合は、UNDEFINEDFILE 条件が発生します。

以下の条件下では、OUTPUT ファイルに対してデフォルト行サイズ値が付与されます。

- ファイルは PRINT 属性を持っている。 この場合、値はタブ制御テーブルから得られます。
- 関連データ・セットが端末 (stdout または stderr) である。 この場合、値は 120 です。

なお、LINESIZE オプションが (OPEN ステートメントで) 指定されていて、(ENVIRONMENT 属性、TITLE オプション、または DD ステートメントで) RECSIZE も指定されていて、(レコード・フォーマットおよび適切な制御バイト・オーバーヘッドを考慮に入れて) レコード・サイズ値が小さ過ぎて LINESIZE を保持できない場合は、以下のことが行われます。

- Language Environment for z/OS 1.9 以前のリリースを使用している場合、DD SYSOUT= ファイルに関しては、指定された LINESIZE に一致する新規レコード・サイズを決定するために LINESIZE オプションが使用され、DD DSN= ファイルおよび他のすべてのファイルに関しては、UNDEFINEDFILE 条件が発生します。
- Language Environment for z/OS 1.9 より後のリリースを使用している場合は、すべてのファイルに関して UNDEFINEDFILE 条件が発生します。

例: ストリーム指向データ伝送によるデータ・セットの作成

以下の例では、編集指示ストリーム指向データ伝送を使用して直接アクセス・ストレージ装置上にデータ・セットを作成する方法が示されています。

ファイル SYSIN によって入力ストリームから読み取られるデータには、名前の付いていない 7 文字サブフィールドが入っているフィールド VREC が含まれており、フィールド NUM は情報が入っているこれらのサブフィールドの数を定義します。 出力ファイル WORK からは、フィールド FREC 全体と、情報が含まれている VREC のサブフィールドのみがデータ・セットに伝送されます。

ストリーム指向データ伝送によるデータ・セットの作成

```
//EX7#2  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM OUTPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 PAD CHAR(25),
               2 VREC CHAR(35),
             EOF BIT(1) INIT('0'B),
             IN CHAR(80) DEF REC;
         ON ENDFILE(SYSIN) EOF='1'B;
         OPEN FILE(WORK) LINESIZE(400);
         GET FILE(SYSIN) EDIT(IN)(A(80));
         DO WHILE (¬EOF);
         PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
         GET FILE(SYSIN) EDIT(IN)(A(80));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(NEW,CATLG),UNIT=SYSDA,
//           SPACE=(TRK,(1,1))
//GO.SYSIN DD *
R.C.ANDERSON       0 202848 DOCTOR
B.F.BENNETT        2 771239 PLUMBER          VICTOR HAZEL
R.E.COLE           5 698635 COOK             ELLEN  VICTOR JOAN   ANN    OTTO
J.F.COOPER         5 418915 LAWYER           FRANK  CAROL  DONALD NORMAN BRENDA
A.J.CORNELL        3 237837 BARBER           ALBERT ERIC   JANET
E.F.FERRIS         4 158636 CARPENTER        GERALD ANNA   MARY   HAROLD
/*
```

例: グラフィック・データのストリーム・ファイルへの書き込み

以下の例では、リスト指示出力を使用してグラフィックスをストリーム・ファイルに書き込むプログラムが示されています。

この例では、グラフィック・データを印刷できる出力装置があることを想定しています。 このプログラムは、従業員レコードを読み取り、特定の地域に在住する従業員を選択します。 このプログラムはさらに、住所フィールドを編集して、 各住所項目間にグラフィック・ブランクを挿入し、 従業員番号、氏名、および住所を印刷します。

グラフィック・データのストリーム・ファイルへの書き込み

```
//EX7#3  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN      DD *
% PROCESS GRAPHIC;
  XAMPLE1:  PROC OPTIONS(MAIN);
            DCL  INFILE FILE INPUT RECORD,
                 OUTFILE FILE OUTPUT STREAM ENV(GRAPHIC);
 /* GRAPHIC OPTION MEANS DELIMITERS WILL BE INSERTED ON OUTPUT FILES. */
           DCL
                1 IN,
                   3 EMPNO CHAR(6),
                   3 SHIFT1 CHAR(1),
                   3 NAME,
                      5 LAST G(7),
                      5 FIRST G(7),
                   3 SHIFT2 CHAR(1),
```

```
                  3 ADDRESS,
                     5 ZIP CHAR(6),
                     5 SHIFT3 CHAR(1),
                     5 DISTRICT G(5),
                     5 CITY G(5),
                     5 OTHER G(8),
                     5 SHIFT4 CHAR(1);
           DCL EOF BIT(1) INIT('0'B);
           DCL ADDRWK G(20);
     ON ENDFILE (INFILE) EOF = '1'B;
     READ FILE(INFILE) INTO(IN);
     DO WHILE(¬EOF);
             DO;
                 IF SUBSTR(ZIP,1,3)¬='300'
                    THEN LEAVE;
                 L=0;
                 ADDRWK=DISTRICT;
                 DO I=1 TO 5;
                 IF SUBSTR(DISTRICT,I,1)= <   >
                    THEN LEAVE;               /* SUBSTR BIF PICKS 3P  */
                 END;                         /* THE ITH GRAPHIC CHAR */
                 L=L+I+1;                     /* IN DISTRICT          */
                 SUBSTR(ADDRWK,L,5)=CITY;
                 DO I=1 TO 5;
                 IF SUBSTR(CITY,I,1)= <   >
                    THEN LEAVE;
                 END;
                 L=L+I;
                 SUBSTR(ADDRWK,L,8)=OTHER;
                 PUT FILE(OUTFILE) SKIP           /* THIS DATA SET    */
                 EDIT(EMPNO,IN.LAST,FIRST,ADDRWK) /* REQUIRES UTILITY */
                     (A(8),G(7),G(7),X(4),G(20)); /* TO PRINT GRAPHIC */
                                                  /* DATA             */
                 END;                       /* END OF NON-ITERATIVE DO */
         READ FILE(INFILE) INTO (IN);
         END;                                 /* END OF DO WHILE(¬EOF) */
     END XAMPLE1;
 /*
 //GO.OUTFILE   DD  SYSOUT=A,DCB=(RECFM=VB,LRECL=121,BLKSIZE=129)
 //GO.INFILE DD *
 ABCDEF<
>300099<           3 3 3 3 3 3 3   >
 ABCD  <
>300011<           3 3 3 3       >
 /*
```

ストリーム入出力によるデータ・セットへのアクセス

ストリーム指向データ伝送を使用してアクセスされるデータ・セットは、ストリーム指向データ伝送で作成する必要はありませんが、そのようなデータ・セットには CONSECUTIVE 編成が必要であり、そのようなデータ・セットに含まれるデータはすべて文字またはグラフィックの形式になっていなければなりません。 入力のための関連ファイルをオープンし、 データ・セットに入っているレコードを読み取るか、あるいは 出力のためにファイルをオープンし、 終わりにレコードを追加してデータ・セットを拡張することができます。

データ・セットにアクセスするには、次のいずれかの方法でデータ・セットを識別する必要があります。

- ENVIRONMENT 属性
- DD_DDNAME 環境変数
- OPEN ステートメントの TITLE オプション

以下のトピックでは、DD ステートメントに組み込まなければならない必須情報について説明します。 また、指定可能なオプション情報についてもいくつか説明します。 ただし、ここで述べる内容は、入力ストリームのデータ・セットには当てはまりません。

必須情報

ユーザー・アプリケーションで既存の STREAM ファイルにアクセスするには 、PL/I はそのファイルのレコード長を入手する必要があります。

データ・セットがレコード長を持っていない場合は、次のいずれかのソースから値を取得できます。

- OPEN ステートメントの LINESIZE オプション

- ENVIRONMENT 属性の RECSIZE オプション
- DD_DDNAME 環境変数の RECSIZE オプション
- OPEN ステートメントの TITLE オプションの RECSIZE オプション
- PL/I 提供のデフォルト値

既存の OUTPUT ファイルを使用する場合、および RECSIZE の値を 提供する場合、PL/I は、236 ページの『ストリーム入出力によるデータ・セットの作成』に説明されているように レコード長を決定します。

以下の条件では、PL/I は INPUT ファイルに対してデフォルトのレコード長の値を使用します。

- ファイルが SYSIN で、値が 80 の場合
- ファイルが端末 (stdout: または stderr:) に関連付けられており、値が 120 の場合

レコード・フォーマット

ストリーム指向データ伝送を使ってデータ・セットにアクセスするときには、 そのデータ・セットのレコード・フォーマットを知っていなくてもかまいません (ブロック・サイズを指定しなければならない場合を除く)。 各 GET ステートメントがそれぞれ個別の数の文字あるいはグラフィックスを データ・ストリームからプログラムへ転送します。

ユーザーがレコード・フォーマットの情報を与える場合は、 その情報はデータ・セットの実際の構造と矛盾しないものである必要があります。 例えば、F フォーマット・レコード、600 バイトのレコード・サイズ、およ び 3600 バイトのブロック・サイズを使って作成した データ・セットの場合、3600 バイトの最大ブロック・サイズを持った U フォーマット・レコードの 場合と同様にそのレコードにアクセスすることができます。 ただし、ブロック・サイズ 3500 を指定すると、データは切り捨てられます。

例: ストリーム指向データ伝送によるデータ・セットへのアクセス

以下の例では、ストリーム指向データ伝送を使用してデータ・セットにアクセスするプログラムが示されています。

240 ページの『#unique_435/unique_435_Connect_42_f75』 にあるプログラムは、238 ページの『#unique_430/unique_430_Connect_42_f7777』 のプログラムで作成したデータ・セットを読み取って、ファイル SYSPRINT を使ってその中に入っているデータをリストします。

データのセットはそれぞれ GET ステートメントによって 2 つの変数 (FREC および VREC) に読み込まれます。 FREC には常に 45 文字が含まれ、VREC には常に 35 文字が含まれます。 GET ステートメントが実行されるたびに、VREC には、7*NUM という式で算出される数の文字が含まれます。 35 文字に満たない場合はブランクが埋め込まれます。 DD ステートメントの DISP パラメーターは単に DISP=OLD とすることもできま す。DELETE を省略すると既存のデータ・セットは削除されません。

ストリーム指向データ伝送によるデータ・セットへのアクセス

```
//EX7#5  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM INPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 SERNO CHAR(7),
                 3 PROF CHAR(18),
               2 VREC CHAR(35),
             IN CHAR(80) DEF REC,
             EOF BIT(1) INIT('0'B);
         ON ENDFILE(WORK) EOF='1'B;
         OPEN FILE(WORK);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
```

```
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)
```

ストリーム入出力による PRINT ファイルの使用

オペレーティング・システムにも PL/I 言語にも、 出力データのフォーマット設定を簡単に行えるようにするための機能がいくつかあります。

オペレーティング・システムで、ユーザーが各レコードの最初のバイトを印刷制御文字として 使用することができるようにしています。 制御文字は、印刷されずにプリンターに改行や改ページを行わせます。 (印刷制御文字については、255 ページの図 6 および 255 ページの表 21 を参照してください。)

PL/I プログラムでは、 PRINT ファイルを使用することで、ストリーム指向のデータ送信から印刷出力のレイアウトを制御する便利な手段が提供される。 コンパイラーは、PAGE、 SKIP、LINE オプション、およびフォーマット項目に対応 して印刷制御文字を自動的に挿入します。

関連するデータセットを直接印刷するつもりがなくても、STREAM OUTPUT ファイルに PRINT 属性を適用することができます。 PRINT ファイルが直接アクセス・データ・セットに関連付けられていると、 印刷制御文字はこのデータ・セットのレイアウトには効力がありませんが、 レコード内のデータの一部として現れます。

PRINT ファイルを FB または VB で開くと、UNDEFINEDFILE 条件が発生する。 PRINT ファイルは "A "オプション、つまり FBA または VBA で開く必要があります。

コンパイラーは、PRINT ファイルで伝送される各レコードの最初の バイトが米国標準規格 (ANS) の印刷制御文字用に予約し、自動的に適切な文字を 挿入します。

PRINT ファイルは、以下の 5 つの印刷制御文字のみを使用する：

文字
アクション

1 行空けて (ブランク文字) から印刷する

0
2 行空けてから印刷する

-
3 行空けてから印刷する

+
1 行目から印刷する

1
改ページする

コンパイラは、 PAGE, SKIP, と LINE のオプションや書式項目を、現在のレコードの残りを空白でパディングし、次のレコードに適切な制御文字を挿入することで処理する。 SKIP または LINE に 3 行を超えるスペースが指定されている場合は、 コンパイラーは、適切な制御文字を使って必要な数のブランク・レコードを挿入し、 必要なスペーシングを行います。 印刷制御オプションやフォーマット項目がない場合は、レコードがフルになると、コンパイラーは、次のレコードの最初のバイトにブランク文字 (1 行のスペース) を挿入します。

PRINT ファイルの伝送先が端末の場合は、出力フォーマットを指定しない限り、PAGE、SKIP、LINE の各オプションに 3 行を超えるスキップをさせることはありません。

印刷する行の長さの制御

PL/I プログラム (ENVIRONMENT 属性) または DD 文でレコード長を指定するか、OPEN 文 (LINESIZE オプション) で行サイズを指定することで、 PRINT ファイルが生成する印刷行の長さを制限することができます。

レコード長には印刷制御文字用の余分のバイト数を含める必要があります。 すなわち、レコード長は印刷される行の長さ より 1 バイト (V フォーマット・レコードの場合は、5 バイト) 長くなければなりません。

LINESIZE オプションで指定する値は、印字される行の文字数を指し、コンパイラは印字制御文字を追加する。

レコードをブロック化しても、PRINT ファイルで 生成される出力の外観に影響はありませんが、 直接アクセス装置上でファイルがデータ・セットと関連付けられていれば、 補助記憶域をより効率よく使えるようになります。 LINESIZE オプションを使用する場合は、行のサイズがブロックサイズに適合していることを確認してください。 F フォーマットのレコードの場合、ブロック・サイズは正確に (行サイズ +1) の倍数でなくてはならず、V フォーマットのレコードの場合、 ブロック・サイズは行サイズよりも最低 9 バイト大きくなければなりません。

ファイルをいったんクローズし、新しい行サイズでもう一度オープンすれば、実 行中に PRINT ファイルの行サイズを変更できますが、PRINT ファイルを使って直接 アクセス装置上にデータ・セットを作成する場合は、注意が必要です。 ファイルを初めてオープンしたとき、 データ・セットに設定したレコード・フォーマットは変更できません。 OPEN ステートメントに指定した行サイズと、 既に設定されているレコード長が矛盾すると、UNDEFINEDFILE 条件が発生します。 このような矛盾が生じないように、使用する予定の最大行サイズより最低 9 バイト大き いブロック・サイズを指定して V フォーマット・レコードを指定するか、最初の OPEN ステートメントで最大行サイズを必ず指定するようにしてください。 (プリンターに送る予定の出力は、直接プリンターへ送ろうとする 場合でも、UNIT= を使ってプリンターを指定しない限り、一時的に直接アクセス装置に 保管することができます。)

PRINT ファイルは 120 文字のデフォルトの行サイズを持っています。したがって、PRINT ファイルのレコード・フォーマットを指定する必要はありません。 また、他の情報がない場合には、コンパイラーは V フォーマットのレコードであると見なします。 完全なデフォルト値は、次のとおりです。

```
BLKSIZE=129
LRECL=125
RECFM=VBA
```

例

ストリーム・データ伝送による印刷ファイルの作成は、 PRINT ファイルと、ストリーム指向のデータ伝送文の印刷オプションを使用して、表をフォーマットし、後で印刷するためにダイレクト・アクセス・デバイスに書き込む方法を説明します。 このテーブルは、6' 間隔の、0° から 359° 54' までの角度の正弦から成ります。

ENDPAGE ON-ユニットのステートメントは、各ページの下にページ番号を挿入し、次のページの見出しを設定する。

このプログラムで作成されたデータ・セットを定義する DD ステートメントには、 レコード・フォーマットに関する情報は含まれていません。 ファイル TABLE をオープンするステートメントに指定されたファイル宣言およ び行サイズから、コンパイラーは次のように推測します。

レコード・フォーマット =
V（ PRINT ファイルのデフォルト）

レコード・サイズ =
98 (行サイズ + 1 バイト (印刷制御文字用) + 4 バイト (レコード制御フィールド用))

ブロック・サイズ =
102 (レコード長 + 4 バイト (ブロック制御フィールド用))

レコード指向データ伝送の印刷 」のプログラムは、「 ストリーム・データ伝送による印刷ファイルの作成 」のプログラムによって作成された表を印刷するために、レコード指向データ伝送を使用する。

ストリーム・データ伝送による印刷ファイルの作成

レコード指向のデータ送信の印刷の例では、結果のファイルを印刷する。

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 SINE: PROC OPTIONS(MAIN);
   DCL TABLE       FILE STREAM OUTPUT PRINT;
   DCL DEG         FIXED DEC(5,1) INIT(0);  /* INIT(0) FOR ENDPAGE  */
   DCL MIN         FIXED DEC(3,1);
   DCL PGNO        FIXED DEC(2)   INIT(0);
   DCL ONCODE      BUILTIN;
```

```
   ON ERROR
     BEGIN;
       ON ERROR SYSTEM;
       DISPLAY ('ONCODE = '|| ONCODE);
     END;

   ON ENDPAGE(TABLE)
     BEGIN;
       DCL I;
       IF PGNO ¬= 0 THEN
         PUT FILE(TABLE) EDIT ('PAGE',PGNO)
             (LINE(55),COL(80),A,F(3));
       IF DEG ¬= 360 THEN
         DO;
           PUT FILE(TABLE) PAGE EDIT ('NATURAL SINES') (A);
           IF PGNO ¬= 0 THEN
             PUT FILE(TABLE) EDIT ((I DO I = 0 TO 54 BY 6))
                                  (SKIP(3),10 F(9));
           PGNO = PGNO + 1;
         END;
       ELSE
         PUT FILE(TABLE) PAGE;
     END;

   OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
   SIGNAL ENDPAGE(TABLE);

   PUT FILE(TABLE) EDIT
    ((DEG,(SIND(DEG+MIN) DO MIN = 0 TO .9 BY .1) DO DEG = 0 TO 359))
    (SKIP(2), 5 (COL(1), F(3), 10 F(9,4) ));
   PUT FILE(TABLE) SKIP(52);
 END SINE;
```

タブ制御テーブルの指定変更

PRINT ファイルへのデータ・ディレクションとリスト・ディレクションの出力は、あらかじめ設定されたタビュレーターの位置にアライメントされる。 31 ビット・プログラムに関しては、タブ制御テーブルをカスタマイズできます。 ただし、ユーザー定義 PLITAB は 64 ビット・プログラムではサポートされていないため、タブ・コントロール・テーブルを上書きすることはできない。

タブ・テーブルの宣言方法の例については、166 ページの『#unique_302/unique_302_Connect_42_tabsrh』 および 244 ページの『#unique_303/unique_303_Connect_42_f77』 の例を参照してください。 テーブル内のフィールドの定義は以下のとおりです。

OFFSET OF TAB COUNT:
Tab count のオフセットを示すハーフワード 2 進整数。これは、使用されるタブの数を示すフィールドです。

PAGESIZE:
デフォルトのページ・サイズを定義するハーフワード 2 進整数

ページ・サイズは、ストリーム出力時、および PLIDUMP データ・セットへのダ ンプ出力時に使われる値です。

LINESIZE:
デフォルトの行サイズを定義するハーフワード 2 進整数

PAGELENGTH:
端末での印刷で使われるデフォルトのページ長を定義するハーフワード 2 進整数

FILLERS:
3 ハーフワードの 2 進整数。将来の利用のために予約済み

TAB COUNT:
テーブル内のタブ位置の数 (最大 255) を定義するハーフワード 2 進整数

タブ・カウント = 0 の場合は、指定されたタブ位置はすべて無視されます。

Tab1–Tabn:
印刷行内のタブ位置を定義する n 個のハーフワード 2 進整数

最初のタブ位置には 1 という番号がつき、最大値は 255 です。 各タブの値は、テーブル内でその前にあるタブの値より大きく なければなりません。 さもなければその値は無視されます。 印刷される出力の最初のデータ・フィールドは、次の有効なタブ位置から始まります。

リンケージ・エディターで外部参照を PLITABS に解決させることで、プログラムのデフォルト PL/I タブ設定をオーバーライドできる。 参照を解決させるには、PLITABS という名前のテーブルを上記のフォーマットで指定する。

RENT オプションでコンパイルする場合、 PLITABS は NONASGN 属性で宣言しなければならない。 NONASGN 属性は NORENT でコンパイルする際にも指定できるため、 PLITABS は常に NONASGN 属性と一緒に宣言することをお勧めします。

このタブ・テーブルを提供するには、 PLITABS という名前の PL/I 構造体をソース・プログラムに含めます。この構造体は、MAIN プロシージャまたは MAIN プロシージャにリンクされたプログラムの中で、STATIC EXTERNAL であることを宣言しなければなりません。 PL/I の構造例を 244 ページの『#unique_303/unique_303_Connect_42_f77』 に示す。 この例では、3 つの位置 (30、60、および 90) にタブが設定され、ページ・サイズおよび行サイズにデフォルト値が使用されています。 TAB1 は行に印刷される 2 番目の項目の位置を識別するものであることに注意してください。 行の最初の項目は常に左マージンから始まります。 構造体の第 1 項目は NO_OF_TABS フィールドへのオフセットです。 FILL フィールドは省略できません。

PL/I 構造 プリセットタブの設定を変更するためのものです。 PLITABS

```
DCL 1 PLITABS STATIC EXT NONASGN,
    2 (OFFSET INIT(14),
      PAGESIZE INIT(60),
      LINESIZE INIT(120),
      PAGELENGTH INIT(0),
      FILL1 INIT(0),
      FILL2 INIT(0),
      FILL3 INIT(0),
      NO_OF_TABS INIT(3),
      TAB1 INIT(30),
      TAB2 INIT(60),
      TAB3 INIT(90)) FIXED BIN(15,0);
```

31 ビットプログラム用 SYSIN および SYSPRINT ファイルの使用

プログラム中に FILE オプションを付けずに GET ステートメントをコーディングすると、コンパイラはファイル名 SYSIN を挿入します。 PUT 文を FILE オプションなしでコーディングすると、コンパイラーは SYSPRINT という名前を挿入する。

SYSPRINT を宣言しない場合、コンパイラーは通常のデフォルト属性に加え、 PRINT 属性をファイルに与えます。 属性の完全セットは次のとおりです。

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

SYSPRINT は PRINT ファイルなので、コンパイラーはデフォルトの行サイズ 120 文字と V-format レコードも提供する。 ユーザーは最小限の情報のみを該当する DD ステートメントに与えるだけで済みます。 クラス A のシステム出力装置がプリンターであるという通常の規則を使用する ユーザーのシステムの場合には、次のステートメントで十分です。

```
 //SYSPRINT DD SYSOUT=A
```

注 : SYSIN と は、初期化時にユーザー・エグジットで確立される。 SYSPRINT IBM - SYSIN 、 SYSPRINT のデフォルトが端末に送られる。

ファイルを明示的に宣言またはオープンすることで、コンパイラが SYSPRINT に与える属性を上書きすることができる。 SYSPRINT 、 z/OS 言語環境プログラム メッセージ・ファイル・オプションとの相互作用については *z/OS Language Environment Programming Guide*.

コンパイラーは、入力ファイル SYSIN に特別な属性を与えません。もし宣言しなければ、デフォルトの属性だけを受け取ります。 SYSIN 入力ストリームにない場合は、完全な DD 情報を提供する必要がある。

関連情報
169 ページの『SYSPRINT 31 ビットプログラムへの配慮』

64 ビットプログラム用の SYSIN および SYSPRINT ファイルを使用する

64 ビット・プログラムの場合、 SYSPRINT は C の標準出力ファイルと等しく、 SYSIN は C の標準入力ファイルと等しい。

関連情報
178 ページの『SYSPRINT 64 ビットプログラムへの配慮』
64 ビット・プログラムの場合、 SYSPRINT は C 標準出力ファイルと同じである。 また、共有 SYSPRINT 。

端末からの入力の制御

次に挙げることを行えば、ユーザーの PL/I プログラムで、入力ファイル へのデータを端末から入力することができます。

1. 入力ファイルを明示的または暗黙的に、 CONSECUTIVE 環境オプションで宣言する（すべてのストリームファイルがこの条件を満たす）。
2. 入力ファイルを端末に割り振る

標準のデフォルト入力ファイル SYSIN はストリーム・ファイルであり、ターミナルに割り当てることができるので、通常はこれを使うことができる。

ストリームファイルへの入力は、コロン（:）で促される。プログラム内で GET ステートメントが実行されるたびに、コロンが表示されます。 GET 、システムは次の行に進む。 ユーザーはそこに必要なデータを入力することができます。 GET ステートメントの実行を完了するのに十分なデータを含まない行を入力すると、プラス記号の後にコロン（+:）が続くプロンプトが表示される。

継続させたい行の最後にハイフンを付け加えると、2 行以上 のデータが入力されるまで、ユーザー・プログラムへのデータ伝送を遅らせることができます。

ユーザーのプログラムで、ユーザーに入力を求めるプロンプトを出す出力ステートメント を組み込んだ場合、ユーザー自身のプロンプトをコロンで終了して、 初期システム・プロンプトが出ないようにすることができます。 たとえば、GET ステートメントの前には、PUT ステートメントを次のように記述することができる：

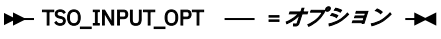
```
PUT SKIP LIST('ENTER NEXT ITEM:');
```

次の GET ステートメントのシステム・プロンプトを禁止するには、自分のプロンプトが以下の条件を満たしている必要がある：

1. リスト指示か編集指示のどちらかでなければならず、リスト指示の場合は PRINT 。
2. プロンプトを伝送するファイルは、端末に割り振る必要があります。 端末でファイルをコピーするだけの場合は、 システム・プロンプトが表示されないようにすることはできません。

TSO では、 TSO_INPUT_OPT という環境変数がサポートされており、上記のデフォルトの方法よりも柔軟な方法で端末からの入力を制御することができる。

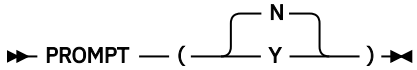
環境変数 TSO_INPUT_OPT の構文は以下の通り。大文字でなければならない：



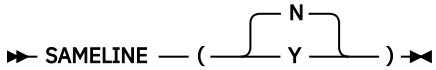
大文字にも小文字にもできるオプションを指定する場合は、ブランクを使用できません。 複数のオプションを指定する場合は、それらのオプションをコンマで区切る必要があります。 また、このステートメントの構文は、コマンド入力時にはチェックされません。 データ・セットのオープン時に、このステートメントの構文が検証されます。 構文が間違っていると、oncode 96 で UNDEFINEDFILE 。

指定できるオプションは以下のとおりです。

PROMPT
PROMPT オプションは、コロンを端末からのストリーム入力のプロンプトとして表示するかどうかを指定します。



SAMELINE
SAMELINE オプションは、入力を求めるプロンプトのステートメントと同じ行で、 システム・プロンプトを行わせるかどうかを指定します。



これらのオプションが端末に対する入出力にどのように影響するのかについて詳しくは、205 ページの『SAMELINE』にある例を参照してください。

TSO 、この環境変数を指定するひとつの方法は、PLIXOPT の文字列を使うことである。 次の例を参照してください。

```
   DCL PLIXOPT char(50) var ext static
       init('ENVAR("TSO_INPUT_OPT=PROMPT(Y),SAMELINE(Y)")');
```

データのフォーマット

端末で入力するデータのフォーマットは、バッチ・モードのストリーム入力データと完全に同じでなければなりません。 ただし、例外がいくつかあります。

- 入力のために簡略化された句読法

  別々の入力項目を別々の行に入力する場合に、間にブランクやコンマを入力する必要はありません。 コンパイラーは各行の末尾にコンマを挿入します。

  例えば、次のステートメントに応答するとします。

  ```
  GET LIST(I,J,K);
  ```

  端末での対話は、次のようになります。

  ```
  :
  1
  +:2
  +:3
  ```

  各項目の後ろに改行が付きます。 これは、次に示すものとまったく同じです。

  ```
  :
  1,2,3
  ```

  ある項目を次の行まで続けたい場合は、1 行目の最後に継続文字を入力します。 継続文字がなければ、GET LIST ステートメントまたは GET DATA ステートメントで はコンマが挿入され、GET EDIT ステートメントでは埋め込みが行われま す (以下参照)。

- GET EDIT での自動埋め込み

  GET EDIT ステートメントに関しては、入力行の末尾にブランクを入力する必要はありません。 ユーザーが入力する項目には、正しい長さになるまで埋め込みが行われます。

  例えば、次の PL/I ステートメントを考えてみてください。

  ```
  GET EDIT(NAME)(A(15));
  ```

  SMITH という 5 文字を入力して即時に改行を行うことができます。 プログラムが 15 文字からなるストリングを受け取れるように、 この項目には 10 個のブランクが埋め込まれます。 ある項目を 2 番目の行または後続行に続けたければ、最終行を除くすべての行の終わりに連結文字を入れなければなりません。 そうでなければ伝送される最初の行には、埋め込み処理が行われ、完結したデータ項目として扱われます。

- SKIP オプションまたはフォーマット項目

  GET ステートメントにおける SKIP は、まだ入力されていないデータを無視するようにプログラムに要求します。 SKIP(*n*) の *n* が 1 より大きい場合、その SKIP(n) はすべて SKIP(1) とみなされます。 SKIP(1) は、現在の行にある未使用データはすべて無視されることを意味します。

ストリーム・ファイルおよびレコード・ファイル

ストリーム・ファイルとレコード・ファイルを両方とも端末に割り振ることができます。 しかし、この場合、レコード・ファイル用のプロンプトは表示されません。

端末に複数のファイルを割り振ったとき、そのうちの 1 つ以上がレコード・ファイルであると、ファイル出力は必ずしも同期化されるとは限りません。 なお、端末へのデータ伝送および端末からのデータ伝送の順序が、対応する PL/I の 入出力ステートメントの実行順序と等しくなる保証はありません。

また、端末からのレコード・ファイルの入力は、TCAM の制限により、大文字で受け取られます。 問題を避けるため、可能であればストリーム・ファイルをお使いください。

PL/I ダイナミックアロケーションを使用した QSAM ファイルの定義

DD ステートメント、環境変数、または OPEN ステートメントの TITLE オプションを使用して、 QSAM または zFS ファイルを定義できます。

環境変数または TITLE オプションを使用する場合、名前は大文字でなければならない。 次のいずれかの方法で MVS データ・セットを指定してください。

- DSN(*data-set-name*)
- DSN(*data-set-name* (*member-name*))

data-set-name は完全修飾でなければなりません。一時データセットに設置できません。 たとえば、 &から開始できません。

次のように zFS ファイルを指定してください。

PATH (*absolute-path-name*)

DSN キーワードの後に、以下の属性を任意の順序で指定できます。

```
NEW, OLD, SHR, or MOD
TRACKS or CYL
SPACE(n,m)
VOL(volser)
UNIT(type)
KEEP, DELETE, CATALOG, or UNCATALOG
STORCLAS(storageclass)
MGMTCLAS(managementclass)
DATACLAS(dataclass)
CLOSE
NORECALL
RELEASE
```

注 : 環境変数や OPEN 文の TITLE オプションを使用して、 PDS または PDSE を作成することはできませんが、既存の PDS または PDSE に新しいメンバーを作成することはできます。

CLOSE, NORECALL、および RELEASE 属性は、 BPXWDYN インターフェースの同じ属性にマッピングされる。 *z/OS* と *SVC 99* がサポートするサービスの詳細については、 *Using REXX* and *z/OS UNIX System Services*BPXWDYN を参照してください。

大文字と小文字

ストリーム・ファイルでは、文字ストリングは小文字または大文字で入力した通りにプログラム に送られます。 一方、レコード・ファイルでは、文字はすべて大文字になります。

ファイルの終わり

桁 1 および桁 2 に /* があり他の文字はない行の /* は、ファイルの終わりマーカーとして扱われます。すなわち、これらの文字は ENDFILE 条件を発生させます。

UNIX システム・サービス環境では、キー・シーケンス ESC-D もファイルの終わりマーカーとして使用できます。 ESC は特定の文字として事前定義されており、この事前定義された文字をシーケンス内で使用する必要があります。

COPY GET ステートメントのオプション

GET ステートメントでは、 COPY オプションを指定することができる。しかし、入力ファイルと同様に COPY ファイルがターミナルに割り当てられている場合、データのコピーは印刷されない。

第 11 章 端末への出力の制御

端末では、以下の条件を満たす PL/I ファイルからデータを取得できます。

1. ファイルは、CONSECUTIVE 環境オプションで明示的または暗黙的に宣言される。 ストリーム・ファイルはすべてこの条件を満たしている。
2. ファイルが端末に割り振られている。

標準的なプリントファイル SYSPRINT は、一般的にこの両方の条件を満たしている。

PRINT ファイルのフォーマット

SYSPRINT または他の PRINT ファイルからのデータは、一般的に、 端末でページの形にフォーマットされません。 PAGE オプション、LINE オプション、およびフォーマット項目については、 常に、3 行がスキップされます。 通常、ENDPAGE 条件が発生することはありません。 SKIP(*n*) の *n* が 3 より大きい場合も、3 行のみがスキップされます。 SKIP(0) はバックスペースによってインプリメントされるので、 バックスペース機能の備わっていない端末で使用してはなりません。

PRINT ファイルは、ユーザー・プログラムにタブ制御テーブルを挿入する ことによって、ページの形にフォーマット設定することができます。 このテーブルは PLITABS という名前でなければなりません。 また、そのテーブルの内容については、243 ページの『タブ制御テーブルの指定変更』に説明があります。 ユーザーは、エレメント PAGELENGTH を必要なページ長、すなわち印刷可能な最大行数で表された、各ページを印刷する用紙の長さに初期化しなければなりません。 エレメント PAGESIZE は、各ページに印刷しようとする実際の行数に 初期化しなければなりません。 PAGESIZE に指定した行数をページに印刷し終わると、ENDPAGE 条件が生じますが、 これに対する標準システム動作は、PAGELENGTH から PAGESIZE を引いた値に等しい行数を スキップし、その後で次ページの印刷を開始することです。 標準レイアウト以外の場合は、PLITABS の他の要素を PLITABS の宣言に示す値に初期化する必要があります。 PLITABS を使って、リスト指示およびデータ指示出力の タブ位置を変更することもできます。 ILC アプリケーションで改ページをフォーマット設定する 必要があるとき、SYSPRINT の代わりに PLITABS を使用することができます。 改ページを制御するには、PAGESIZE を 32767 に 設定して、PUT PAGE ステートメントを使用します。

端末の中にはタブ設定機能を備えたものもありますが、 リスト指示およびデータ指示出力では、常に、 ブランク文字を伝送することによってタブが設定されます。

ストリーム・ファイルおよびレコード・ファイル

ストリーム・ファイルとレコード・ファイルを両方とも端末に割り振ることができます。 ただし、端末に複数のファイルを割り振り、それらのファイルの 1 つが SYSPRINT またはレコード・ファイルである場合、ファイルの出力は必ずしも同期化されるとは限りません。

プログラムと端末との間でやりとりされる時のデータの順序が、それに対応する PL/I 出力ステートメントが実行されるのと同じ順序になるという保証はありません。

PUT EDIT コマンドの出力

PUT EDIT コマンドからターミナルへの出力のフォーマットは、行モード TPUTs で、 フィールドの開始と終了の文字はスクリーン上の空白として表示される。

© Copyright IBM Corp. 1999, 2025

第 12 章 レコード単位データ伝送の使用

PL/I は、RECORD 属性を持つさまざまなタイプのデータ・セットをサポートして います。 このセクションでは、連続データ・セットの使用法について説明します。

251 ページの表 20 は、レコード単位データ伝送を使って、連続データ・セット を作成したり、連続データ・セットにアクセスする場合に 使用できるステートメントとオプションをリストしています。

表 *20.* 連続データ・セットの作成と連続データ・セットへのアクセスで使用できるステートメントとオプション

File declaration[1]	**Valid statements**[2] **with options you must specify**	**Other options you can specify**
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference); LOCATE based-variable FILE(file-reference);	 SET(pointer-reference)
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference);	 FROM(reference)

© Copyright IBM Corp. 1999, 2025

表 20. 連続データ・セットの作成と連続データ・セットへのアクセスで使用できるステートメントとオプション (続き)

File declaration[1]	Valid statements[2] with options you must specify	Other options you can specify
SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	

Notes :

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT.
2. The statement `READ FILE (file-reference);` is a valid statement and is equivalent to `READ FILE(file-reference) IGNORE (1);`.

関連情報

257 ページの『レコード入出力によるデータ・セットの作成』
連続データ・セットを作成するには、SEQUENTIAL OUTPUT の 関連ファイルをオープンする必要があります。 WRITE あるいは LOCATE ステートメントを使用してレコードを書くことができます。

レコード・フォーマットの指定

レコード・フォーマット情報を与える場合には、 その情報はデータ・セットの実際の構造に矛盾しないものである必要があります。

例えば、レコード・フォーマット FB、レコード・サイズ 600 バイト、およびブロック・サイズ 3600 バイトでデータ・セットを作成した場合は、最大ブロック・サイズ 3600 バイトの U フォーマット・レコードであるかのようにレコードにアクセスできます。 ブロック・サイズを 3500 バイトに指定すると、データが切り捨てられます。

レコード入出力を使用したファイルの定義

ファイル宣言を使用すれば、レコード単位データ伝送用のファイルを定義できます。

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
             ENVIRONMENT(options);
```

デフォルト・ファイル属性については、215 ページの表 15 を参照してください。 ファイル属性については、「*PL/I Language Reference*」で説明されています。 ENVIRONMENT 属性のオプションについては、252 ページの『ENVIRONMENT オプションの指定』 を参照のこと。

ENVIRONMENT オプションの指定

このセクションでは、連続データ・セットに適用できる ENVIRONMENT オプションについて説明します。

連続データ・セットに適用できる ENVIRONMENT オプションは以下のとおりです。

```
F|FB|FS|FBS|V|VB|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
```

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
CTLASA|CTL360
LEAVE|REREAD
```

これらのオプションについては、以下のトピックを参照してください。

ENVIRONMENT オプション	参照先のトピック
F|FB|FS|FBS|V|VB|U	• 214 ページの『ENVIRONMENT 属性』 • 217 ページの『レコード単位データ伝送のレコード・フォーマット』
RECSIZE(record-length)	• 214 ページの『ENVIRONMENT 属性』 • 218 ページの『RECSIZE オプション』
BLKSIZE(block-size)	• 214 ページの『ENVIRONMENT 属性』 • 219 ページの『BLKSIZE オプション』
SCALARVARYING	• 214 ページの『ENVIRONMENT 属性』 • 222 ページの『SCALARVARYING オプション - 可変長ストリング』
CONSECUTIVE	253 ページの『CONSECUTIVE』
ORGANIZATION(CONSECUTIVE)	253 ページの『ORGANIZATION(CONSECUTIVE)』
CTLASA|CTL360	254 ページの『CTLASA|CTL360』
LEAVE|REREAD	256 ページの『LEAVE|REREAD』

どのオプションを指定する必要があるか、どれがオプショナルで、どれが デフォルト・オプションであるかを知るには、215 ページの表 15 を参照してください。

CONSECUTIVE

CONSECUTIVE オプションは、連続したデータセット構成のファイルを定義する。

CONSECUTIVE がデフォルトです。

関連情報

210 ページの『データ・セットの編成』
オペレーティング・システムのデータ管理ルーチンは、いくつかのタイプのデータ・セットを処理できます。 これらのデータ・セットは、その内部でのデータ保管方法、およびデータにアクセスするために許可される方法において異なります。

ORGANIZATION(CONSECUTIVE)

ORGANIZATION(CONSECUTIVE) オプションは、ファイルが連続したデータセットに関連付けられていることを指定する。

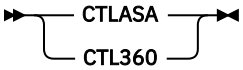
ファイルは、ネイティブ・データセットか、 VSAM データセットのどちらかである。

関連情報

222 ページの『ORGANIZATION オプション』
ORGANIZATION オプションは、 PL/I ファイルに関連するデータセットの構成を指定する。

CTLASA|CTL360

プリンタ制御オプション CTLASA と CTL360 は、連続したデータセットに関連する OUTPUT ファイルにのみ適用される。 この 2 つのオプションは、レコードの最初の文字を制御文字として解釈 するように指定します。



CTLASA オプションは American National Standard Vertical Carriage Positioning Characters or American National Standard Pocket Select Characters (Level 1) を指定する。 また、CTL360 オプションは、IBM のマシン・コード制御文字を指定するためのオプションです。

255 ページの図 6 にリストされている米国標準規格の制御文字は、関連付けら れたレコードが印刷、穿孔される前に、指定の処置を実行するための文字です。

なお、マシン・コード制御文字は、装置タイプによって異なります。 プリンター用の IBM マシン・コード制御文字は、255 ページの表 21 にリストしてあります。

コード
アクション

1 行空けて (ブランク・コード) から印刷する

0
2 行空けてから印刷する

-
3 行空けてから印刷する

+
1 行目から印刷する

1
チャネル 1 にスキップする

2
チャネル 2 にスキップする

3
チャネル 3 にスキップする

4
チャネル 4 にスキップする

5
チャネル 5 にスキップする

6
チャネル 6 にスキップする

7
チャネル 7 にスキップする

8
チャネル 8 にスキップする

9
チャネル 9 にスキップする

A
チャネル 10 にスキップする

B
チャネル 11 にスキップする

C
チャネル 12 にスキップする

V
スタッカー 1 を選択する

W
スタッカー 2 を選択する

図 *6.* 米国標準規格の印刷およびカード穿孔制御文字 *(CTLASA)*

表 *21. IBM* マシン・コード印刷制御文字 *(CTL360)*

印刷してから実行 **コード・バイト**	**アクション**	**直ちに処置 (印刷は行わない)** **コード・バイト**
00000001	印刷のみ (スペースなし)	—

表 *21. IBM* マシン・コード印刷制御文字 *(CTL360)* (続き)

印刷してから実行 コード・バイト	アクション	直ちに処置 (印刷は行わない) コード・バイト
00001001	スペース 1 行	00001011
00010001	スペース 2 行	00010011
00011001	スペース 3 行	00011011
10001001	チャネル 1 にスキップする	10001011
10010001	チャネル 2 にスキップする	10010011
10011001	チャネル 3 にスキップする	10011011
10100001	チャネル 4 にスキップする	10100011
10101001	チャネル 5 にスキップする	10101011
10110001	チャネル 6 にスキップする	10110011
10111001	チャネル 7 にスキップする	10111011
11000001	チャネル 8 にスキップする	11000011
11001001	チャネル 9 にスキップする	11001011
11010001	チャネル 10 にスキップする	11010011
11011001	チャネル 11 にスキップする	11011011
11100001	チャネル 12 にスキップする	11100011

LEAVE|REREAD

磁気テープ処理オプション LEAVE および REREAD は、磁気テープ・ボリュームの終端に達したとき、または磁気テープ・ボリューム上のデータ・セットがクローズされたときに実行される動作を指定します。

LEAVE オプションはテープの巻き戻しを防ぐ。 REREAD オプションは、テープを巻き戻して、データ・セットの再処理を可能にします。 これらのいずれかを指定しない場合、ボリュームの終わりまたはデータ・セットのクローズ時のアクションは、関連 DD ステートメントの DISP パラメーターによって制御されます。

データセットが同じプログラム内で最初に前方に読み込まれ、または書き込まれた後、後方に読み込まれる場合、 LEAVE オプションを指定して、ファイルが閉じられたとき（またはマルチボリュームデータセットの場合、ボリューム切り替えが発生したとき）に巻き戻されないようにする。

256 ページの表 22 は、 と REREAD オプションの効果をまとめたものである。 LEAVE

表 *22. LEAVE* および *REREAD* オプションの影響

ENVIRONMENT オプション	DISP パラメーター	アクション
REREAD	—	データ・セットを再処理する現行ボリュームの位置を決めます。
LEAVE	—	現行ボリュームをデータ・セットの論理終了に位置決めします。

表 *22. LEAVE* および *REREAD* オプションの影響 (続き)

ENVIRONMENT オプション	DISP パラメーター	アクション
Neither REREAD nor LEAVE	PASS	データ・セットの終わりに ボリュームの位置を決める。
	DELETE	現行ボリュームを巻き戻す。
	KEEP CATLG UNCATLG	現行ボリュームを 巻き戻し・アンロードする。

レコード入出力によるデータ・セットの作成

連続データ・セットを作成するには、SEQUENTIAL OUTPUT の 関連ファイルをオープンする必要があります。 WRITE あるいは LOCATE ステートメントを使用してレコードを書くことができます。

251 ページの表 20 は、連続データ・セットを作成するためのステートメントとオプションを示しています。

データ・セットを作成する際、DD ステートメント内で、 オペレーティング・システムに対してそのデータ・セットを識別しなければ なりません。 DD ステートメントに組み込まなければならない必須情報、および指定可能なオプション情報を 257 ページの表 23 にまとめます。

表 *23.* レコード入出力による連続データ・セットの作成*: DD* ステートメントの必須パラメーター

ストレージ装置	必要な場合	指定すべき事項	パラメーター
すべて	常時	出力装置	UNIT= または SYSOUT= または VOLUME=REF=
		ブロック・サイズ [1]	DCB=(BLKSIZE=...
直接アクセスのみ	常時	必要なストレージ・スペース	SPACE=
直接アクセス	データ・セットが別のジョブ・ステップによって使用されるが、ジョブ終了時に必要ではない場合	後処理	DISP=
	ジョブ終了後も保持されるデータ・セット	後処理	DISP=
		データ・セットの名前	DSNAME=
	特定装置上に配置されるデータ・セット	ボリューム通し番号	VOLUME=SER= または VOLUME=REF=

注 :

1. または、ENVIRONMENT 属性を用いれば、PL/I プログラム内でブロック・サイズを指定することもできます。

必須情報

データ・セットを作成する際、DD ステートメント内で、 オペレーティング・システムに対してそのデータ・セットを識別しなければ なりません。 このトピックでは、ステートメントに組み込む必要がある必須情報について説明します。

連続データ・セットを作成する場合は、以下の必須情報を DD ステートメントに指定する必要があります。

- PL/I ファイルと関連付けるデータ・セットの名前

  なお、連続編成のデータ・セットは、どのタイプの装置上にも存在できます。

- レコード長

  レコード長は、ENVIRONMENT 属性、DD_DDNAME 環境変数、または OPEN ステートメントの TITLE オプションの RECSIZE オプションを使って指定することができます。

  端末装置 (stdout: または stderr:) に関連付けられたファイルでは、RECSIZE オプションが指定されていなければ、PL/I はデフォルトのレコード長 の 120 を使用します。

レコード入出力によるデータ・セットのアクセスおよび更新

連続データ・セットを作成し終えれば、順次入力、順次出力、または、直接アクセス装置上のデータ・セットの場合は、更新を行うために、その連続データ・セットにアクセスするためのファイルをオープンすることができます。

259 ページの『#unique_438/unique_438_Connect_42_f88』 は、連続データ・セットにアクセスし、 それを更新するプログラムの例です。 出力用にファイルを開き、最後にレコードを追加してデータセットを拡張する場合は、DD 文で DISP=MOD 。 それを指定しないと、そのデータ・セットは上書きされます。 更新用のファイルをオープンしても、その既存順にしかレコードを 更新することはできず、レコードを挿入したければ、新たにデータ・セットを 作成しなければなりません。 連続データ・セットにアクセスしたり連続データ・セットを更新したりするための ステートメントとオプションを 251 ページの表 20 に示します。

SEQUENTIAL UPDATE ファイルで連続したデータセットにアクセスする場合、REWRITE ステートメントで更新する前に、 READ ステートメントでレコードを取得する必要があります。ただし、取得したすべてのレコードを書き換える必要はありません。 REWRITE 、常に最後に読み込まれたレコードが更新される。

次の ステートメントを見てください。

```
READ FILE(F) INTO(A);
     .
     .
     .
READ FILE(F) INTO(B);
     .
     .
     .
REWRITE FILE(F) FROM(A);
```

REWRITE 、2 番目の READ 。 最初のステートメントで読み取られたレコードは、2 番目の READ ステートメントが実行された後は書き換えることができない。

磁気テープ上の連続データ・セットの場合、末尾へのレコードの追加を除き、更新はできません。 レコードを置換または挿入するには、データ・セットを読み取り、更新したレコードを新規データ・セットに書き込む必要があります。

磁気テープ上の連続データ・セットの読み取りは、順方向にのみ行うことができます。 逆方向読み取りはサポートされていません。

データ・セットにアクセスするには、DD ステートメント でそのデータ・セットをオペレーティング・システムに識別する必要があります。 259 ページの表 24 は、連続データ・セットに アクセスするのに必要な DD ステートメントのパラメーターを要約しています。

表 *24.* レコード入出力による連続データ・セットへのアクセス*: DD* ステートメントの必須パラメーター

パラメーター	指定すべき事項	必要な場合
DSNAME=	データ・セットの名前	常時
DISP=	データ・セットの後処理	常時
UNIT= または VOLUME=REF=	入力装置	データ･セットがカタログされていない場合 (すべての装置)
VOLUME=SER=	ボリューム通し番号	データ･セットがカタログされていない場合 (直接アクセス)
DCB=(BLKSIZE=	ブロック・サイズ [1]	データ・セットに標準ラベルが付いていない場合

注:

1. または、ENVIRONMENT 属性を用いれば、PL/I プログラム内でブロック・サイズを指定することもできます。

以下のトピックでは、DD ステートメントに組み込まなければならない必須情報、および指定可能なオプション情報について説明します。 ただし、ここで述べる内容は、入力ストリームのデータ・セットには当てはまりません。

必須情報

データ・セットがカタログされている場合は、DD ステートメントに次の情報だけを指定する必要があります。

- データセットの名前 (DSNAME パラメータ)

  オペレーティング・システムは、システム・カタログ内でそのデータ・セ ットを記述した情報を検索し、また、必要があれば、オペレーターに、その データ・セットが入っているボリュームをマウントするよう要求します。

- データセットが存在することの確認 (DISP パラメータ)

  末尾にレコードを追加してデータセットを拡張するつもりで出力用にデータセットを開く場合は、DISP=MOD。そうでない場合は、出力用にデータセットを開くと上書きされる。

データセットがカタログ化されていない場合は、さらにデータセットを読み込むデバイスを指定し、ダイレクトアクセスデバイスの場合は、データセットを含むボリュームのシリアル番号を指定する必要があります（UNIT および VOLUME パラメータ）。

連続データ・セットの例

例: 連続データ・セットのマージ、ソート、作成と連続データ・セットへのアクセス

259 ページの『#unique_438/unique_438_Connect_42_f88』 は、連続データ・セットの作成、および連続データ・セットへのアクセスを示したものです。 このプログラムは、入力ストリーム中の 2 つのデータセットの内容をマージし、新しいデータセット、 &&TEMP に書き込む。元のデータセットのそれぞれは、EBCDIC 照合順序で並べられた 15 バイトの固定長レコードを含んでいる。 2 つの入力ファイル、 INPUT1 と INPUT2 は、デフォルトの属性 BUFFERED を持ち、ロケート・モードを使用して、関連するデータセットからそれぞれのバッファにレコードを読み込む。 バッファー内の基底付き変数へのアクセスは、 ファイルがクローズされた後は行わないでください。

連続データ・セットのマージ、ソート、作成と連続データ・セットへのアクセス

```
//EXAMPLE  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN   DD *
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;
```

```
 MERGE: PROC OPTIONS(MAIN);
   DCL (INPUT1,                             /* FIRST INPUT FILE     */
        INPUT2,                             /* SECOND INPUT FILE    */
        OUT )      FILE RECORD SEQUENTIAL;  /* RESULTING MERGED FILE*/
   DCL SYSPRINT    FILE PRINT;              /* NORMAL PRINT FILE    */

   DCL INPUT1_EOF  BIT(1) INIT('0'B);       /* EOF FLAG FOR INPUT1  */
   DCL INPUT2_EOF  BIT(1) INIT('0'B);       /* EOF FLAG FOR INPUT2  */
   DCL OUT_EOF     BIT(1) INIT('0'B);       /* EOF FLAG FOR OUT     */
   DCL TRUE        BIT(1) INIT('1'B);       /* CONSTANT TRUE        */
   DCL FALSE       BIT(1) INIT('0'B);       /* CONSTANT FALSE       */

   DCL ITEM1       CHAR(15) BASED(A);       /* ITEM FROM INPUT1     */
   DCL ITEM2       CHAR(15) BASED(B);       /* ITEM FROM INPUT2     */
   DCL INPUT_LINE  CHAR(15);                /* INPUT FOR READ INTO  */
   DCL A           POINTER;                 /* POINTER VAR          */
   DCL B           POINTER;                 /* POINTER VAR          */

   ON ENDFILE(INPUT1) INPUT1_EOF = TRUE;
   ON ENDFILE(INPUT2) INPUT2_EOF = TRUE;
   ON ENDFILE(OUT)    OUT_EOF    = TRUE;

   OPEN FILE(INPUT1) INPUT,
        FILE(INPUT2) INPUT,
        FILE(OUT)    OUTPUT;

   READ FILE(INPUT1) SET(A);                /* PRIMING READ         */
   READ FILE(INPUT2) SET(B);

   DO WHILE ((INPUT1_EOF = FALSE) & (INPUT2_EOF = FALSE));
     IF ITEM1 > ITEM2 THEN
       DO;
         WRITE FILE(OUT) FROM(ITEM2);
         PUT FILE(SYSPRINT) SKIP EDIT('1>2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT2) SET(B);
       END;
     ELSE
       DO;
         WRITE FILE(OUT) FROM(ITEM1);
         PUT FILE(SYSPRINT) SKIP EDIT('1<2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT1) SET(A);
       END;
   END;
```

```
   DO WHILE (INPUT1_EOF = FALSE);           /* INPUT2 IS EXHAUSTED  */
     WRITE FILE(OUT) FROM(ITEM1);
     PUT FILE(SYSPRINT) SKIP EDIT('1', ITEM1) (A(2),A);
     READ FILE(INPUT1) SET(A);
   END;

   DO WHILE (INPUT2_EOF = FALSE);           /* INPUT1 IS EXHAUSTED  */
     WRITE FILE(OUT) FROM(ITEM2);
     PUT FILE(SYSPRINT) SKIP EDIT('2', ITEM2) (A(2),A);
     READ FILE(INPUT2) SET(B);
   END;

   CLOSE FILE(INPUT1), FILE(INPUT2), FILE(OUT);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(OUT) SEQUENTIAL INPUT;

   READ FILE(OUT) INTO(INPUT_LINE);         /* DISPLAY OUT FILE     */
   DO WHILE (OUT_EOF = FALSE);
     PUT FILE(SYSPRINT) SKIP EDIT(INPUT_LINE) (A);
     READ FILE(OUT) INTO(INPUT_LINE);
   END;
   CLOSE FILE(OUT);

 END MERGE;
/*
//GO.INPUT1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.INPUT2 DD *
BBBBBB
DDDDDD
```

```
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,
//          DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))
```

Merge Sort—creating and accessing a consecutive data set (continued)

例: レコード単位データ伝送の印刷

261 ページの『#unique_438/unique_438_Connect_42_f814』 のプログラムは、レコード単位データ伝送を使用して、242 ページの『#unique_437/unique_437_Connect_42_f76』 のプログラムで作成されたテーブルを出力します。

レコード単位データ伝送の印刷

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 PRT: PROC OPTIONS(MAIN);
   DCL TABLE       FILE RECORD INPUT SEQUENTIAL;
   DCL PRINTER     FILE RECORD OUTPUT SEQL
                        ENV(V BLKSIZE(102) CTLASA);
   DCL LINE        CHAR(94) VAR;

   DCL TABLE_EOF   BIT(1) INIT('0'B);       /* EOF FLAG FOR TABLE   */
   DCL TRUE        BIT(1) INIT('1'B);       /* CONSTANT TRUE        */
   DCL FALSE       BIT(1) INIT('0'B);       /* CONSTANT FALSE       */


   ON ENDFILE(TABLE) TABLE_EOF = TRUE;

   OPEN FILE(TABLE),
        FILE(PRINTER);

   READ FILE(TABLE) INTO(LINE);             /* PRIMING READ         */

   DO WHILE (TABLE_EOF = FALSE);
     WRITE FILE(PRINTER) FROM(LINE);
     READ FILE(TABLE) INTO(LINE);
   END;

   CLOSE FILE(TABLE),
         FILE(PRINTER);
 END PRT;
```

第 13 章 領域データ・セットの定義と使用

この章では、領域データ・セットの編成、データ伝送ステートメント、および 領域データ・セットを定義する ENVIRONMENT オプションについて述べます。 また、領域編成のタイプごとに、領域データ・セットを作成したり領域データ・セットにアクセスしたりする方法についても説明します。

注 : 領域データ・セットは、64 ビット・プログラムではサポートされません。

領域編成のデータ・セットは 2 つの領域に分かれますが、それぞれの領域 は領域番号で識別され、またそのおのおのに、領域編成のタイプに 応じて、単数または複数のレコードを入れることができます。 これらの領域には、ゼロから始まる連続番号が付けられ、レコードは データ伝送ステートメント内に領域番号と一緒にキーを指定することによってアクセスすることができます。

領域データ・セットは、直接アクセス装置に限られます。

データ・セットを領域編成にすれば、データ・セット内でのレコードの 物理配置を制御することができ、また、特定アプリケーションへのアクセス 時間を最適化することができます。 このような最適化は、連続編成または索引編成では使用すること はできません。それは、これらの編成では、昇順キー値に応じて、連続レコードが厳密な物理順序または論理順序で書き込まれるからです。 これらの方式のいずれも直接アクセス・ストレージ・デバイスの特性を十分には利用しません。

領域データ・セットは、連続データ・セットまたは索引付きデータ・セットと 似た方法で、昇順の領域番号順にレコードを提示することによって作成することができます。 別の方法として、直接アクセスを用いることができ、その場合、 レコードはランダム順序で提示し、それらを事前にフォーマット設定された領域に直接挿入します。 領域データ・セットを作成した後は、INPUT または UPDATE だけでなく SEQUENTIAL または DIRECT 属性を持ったファイルを使用してそのデータ・セットにアクセスすることができます。 データ・セットが SEQUENTIAL INPUT ファイルまたは SEQUENTIAL UPDATE ファイルと関連付けて いれば、領域番号またはキーを指定する必要はありません。 ファイルに DIRECT 属性があれば、無作為にレコードを検索、追加、 削除、および置換することができます。

領域データ・セット内のレコードは、有効なデータが入っている実際のレコードであるか、またはダミー・レコードのいずれかです。

領域編成での、他のタイプのデータ・セット編成よりも大きな利点は、 ユーザーがレコードの相対配置を制御できることにあります。 適切なプログラミングにより、装置の能力および アプリケーションの要件に合わせレコード・アクセスを 最適化することができます。

領域データ・セットの直接アクセスは、索引付きデータ・セットのアクセス より早く行うことができますが、領域データ・セットには、順次処理は レコードをランダム順に提示することがあるという欠点があります。 順次検索の順序は必ずしもレコードが提示される順序ではなく、 また必ずしも相対キー値と関連する必要もありません。

263 ページの表 25 は、領域データ・セットを作成し、 また領域データ・セットへアクセスできるための データ伝送ステートメント とオプションをリストしています。

表 *25.* 領域データ・セットの作成と領域データ・セットへのアクセスで使用できるステートメントとオプション

ファイル宣言 [1]	有効ステートメント [2] および必須オプション	含めることができる その他のオプション
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE 基底付き変数 FROM(file-reference) KEYFROM(expression);	SET(pointer-reference)

© Copyright IBM Corp. 1999, 2025

表 25. 領域データ・セットの作成と領域データ・セットへのアクセスで使用できるステートメントとオプション (続き)

ファイル宣言 [1]	有効ステートメント [2] および必須オプション	含めることができる その他のオプション
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE[3]	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	
DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	DELETE FILE(file-reference) KEY(expression);	

表 25. 領域データ・セットの作成と領域データ・セットへのアクセスで使用できるステートメントとオプション (続き)

ファイル宣言 [1]	有効ステートメント [2] および必須オプション	含めることができる その他のオプション

ノート:

1. 完全なファイル宣言には属性 FILE、 RECORD、および ENVIRONMENT が含まれています。 オプション KEY、KEYFROM、あるいは KEYTO のいずれかを使用する場合は、 属性 KEYED も含めなくてはなりません。
2. ステートメント `READ FILE(file-reference);` は、ステートメント `READ FILE(file-reference) IGNORE(1);` と同等です。
3. 新たにデータ・セットを作成する場合は、ファイルに UPDATE 属性が含まれていてはなりません。

PL/I ダイナミックアロケーションを使用した REGIONAL(1) データセットの定義

DD 文、環境変数、または OPEN 文の TITLE オプションを使用して、 REGIONAL(1) データセットを定義できます。

環境変数または TITLE オプションを使用する場合、名前は大文字でなければならない。 次のように MVS データ・セットを指定してください。

```
DSN(data-set-name)
```

data-set-name は完全修飾でなければなりません。一時データセットに設置できません。 たとえば、 &から開始できません。

DSN キーワードの後に、以下の属性のいずれかを指定する必要がある：

```
OLD
SHR
```

領域データ・セット用ファイルの定義

ファイル宣言を使用すれば、領域データ・セットを定義できます。

順次領域データ・セットの定義

順次領域データ・セットを定義するには、次の属性が指定されたファイル宣言を使用します。

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

BUFFERED と UNBUFFERED は、 REGIONAL(1) データセットでは同じように扱われるため、ENVIRONMENT オプションでどちらのオプションを指定してもよい。 例えば、 SEQUENTIAL UNBUFFERED ファイルの REWRITE では FROM オプションは不要であり、 UNBUFFERED が指定されていても OUTPUT SEQUENTIAL データセットでは LOCATE ステートメントが許可される。

直接領域データ・セットの定義

直接領域データ・セットを定義するには、次の属性が指定されたファイル宣言を使用します。

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
```

```
              DIRECT
              ENVIRONMENT(options);
```

デフォルト・ファイル属性については、215 ページの表 15 を参照してください。 ファイル属性について詳しくは、「*PL/I Language Reference*」を参照してください。 ENVIRONMENT オプションのサブオプションについては、 266 ページの『ENVIRONMENT オプションの指定』 を参照。

ENVIRONMENT オプションの指定

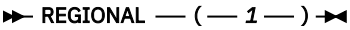
このセクションでは、領域データ・セットに適用できる ENVIRONMENT オプションについて説明します。

領域データ・セットに適用できる ENVIRONMENT オプションは以下のとおりです。

```
REGIONAL({1})
F
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
```

REGIONAL

REGIONAL オプションを使用すれば、領域編成のファイルを定義できます。



1
REGIONAL(1) を指定します。

REGIONAL(1)
データ・セットに、記録済みキーのない F フォーマットのレコードが入っていることを表します。 データ・セット内の各領域にはただ 1 つのレコードが入っており、 したがって、各領域番号はデータ・セット内の相対レコードに 対応しています (すなわち、領域番号はデータ・セットの始めから 0 で始まります)。

REGIONAL(1) データ・セットには記録済みキーが 1 つもありませんが、 REGIONAL(1) DIRECT INPUT ファイルまたは REGIONAL(1) DIRECT UPDATE ファイルを 使えば、記録済みキーのないデータ・セットも処理できます。

RECSIZE(record-length)
BLKSIZE(block-size)
RECSIZE と BLKSIZE の両方を指定する場合は、それぞれに同じ値を指定する必要があります。

重複した領域番号がなく、また大半の領域がいっぱいになる (データ・セット 内の無駄なスペースが削減される) アプリケーションの場合には、REGIONAL(1) 編成が最適です。

REGIONAL データセットでキーを使用する

キーには、記録済みキー とソース・キー の 2 種類があります。

記録済みキー は、レコードを識別するためにデータ・セット内の各キーの直前に付く 文字ストリングです。その長さは 255 文字を超えることはできません。 ソース・キーとは、データ送信ステートメントの KEY または KEYFROM オプションに表示される、ステートメントが参照するレコードを識別するための式の文字値である。 領域データ・セット内のレコードにアクセスする場合は、ソース・キーは 領域番号を与え、同時に、記録済みキーも与えることができます。

索引付きデータ・セット用のキーと異なり、 領域データ・セットの記録済みキーはレコード内に埋め込まれることはありません。

REGIONAL(1)のデータセットを使用

REGIONAL(1)のデータセットでは、記録されたキーがないため、地域番号が特定のレコードの唯一の識別となる。

ソース・キーの文字値は、16777215 を超えてはならない符号なし 10 進整数を表さなければなりません (許容される実際のレコード数は、レコード・サイズ、装置容量、および個々のアクセス方式での制限をどのように組み合わせるかによって、これより小さくなることがあります。 固定フォーマット・レコードを持つ直接 regional(1) ファイルに関しては、相対トラック・アドレッシングでアドレスできるトラックの最大数は 65536 です。 領域番号がこの数値を超える場合、領域番号はモジュロ 16777216 として扱われ、例えば、16777226 は 10 として扱われます。 0 から 9 までの文字とブランク文字だけが、ソース・キー内では有効です。 先行ブランクはゼロとして解釈されます。 領域番号に埋め込まれたブランクを使用することはできません。したがって、 埋め込まれたブランクが最初に見つかった時点で、その領域番号は終了します。 ソース・キー中に 8 文字を超えて存在する場合は、 右端の 8 文字のみが領域番号として使用され、8 文字未満の場合は、 左側にブランク (ゼロとして解釈される) が挿入されます。

ダミー・レコード

REGIONAL(1)データセットのレコードは、有効なデータを含む実際のレコードか、ダミーレコードのいずれかである。 REGIONAL(1)データセットのダミーレコードは、その最初のバイトの定数(8)'1'B によって識別される。

このようなダミー・レコードは、データ・セットの作成時かまたはレコードの 削除時にデータ・セット内に挿入されますが、データ・セットが読み取られる ときに無視されません。 ユーザーの PL/I プログラムはそれらを認識するように作成されなくてはなりません。 ダミー・レコードは、有効データで置き換えることができます。 ダミー・レコードの最初のバイトに (8)'1'B を挿入した場合は 、検索されないダミー・レコードを持っているデータ・セットに ファイルをコピーすると、そのレコードは失われることがあります。

REGIONAL(1) データ・セットの作成

REGIONAL(1) データ・セットは、順次に作成することも直接アクセスによって作成することもできます。

263 ページの表 25 は、 領域データ・セットを作成するためのステートメントとオプションを示しています。

SEQUENTIAL OUTPUT ファイルを使ってデータ・セットを作成するとき、 ファイルをオープンすると、データ・セット上のすべてのトラックが 消去され、各トラックの先頭に、そのトラック上で使用できるスペースの 大きさを記録する容量レコードが書き込まれることになります。 レコードは領域番号の昇順で提示しなくてはならず、そのシーケンスから省略された領域は ダミー・レコードによって埋められます。 このシーケンスにエラーがあると、あるいは、重複キーを提示すると、KEY 条件が発生します。 ファイルがクローズされるときに、現行エクステントの終わりのスペース にはダミー・レコードが埋め込まれます。

DIRECT OUTPUT ファイルを使ってデータ・セットを作成すると、 そのデータ・セットに割り振られた 1 次エクステント全体が、 ファイルのオープン時にダミー・レコードで埋められます。 レコードをランダム順で提示することができますが、 レコードを重複して提示した場合、既存レコードは上書きされます。

順次作成の場合、データ・セットは最大 15 までのエクステントを 持つことができ、また複数のボリューム上に またがっていてもかまいません。 直接作成の場合、データ・セットは 1 つしかエクステントを持つことはできず、したがって 1 つのボリューム上にしか存在することは できません。

例

REGIONAL(1) データ・セットの作成方法を 267 ページの『#unique_466/unique_466_Connect_42_f830』に示します。 この例のデータ・セットは、電話番号とその電話番号を割り当てる加入者の氏名リストです。電話番号は領域データ・セット内の領域番号と対応しており、各領域番号が占め る領域には加入者名のデータが入っています。

REGIONAL(1) データセットの作成：

```
//EX9    JOB
//STEP1  EXEC IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 CRR1:   PROC OPTIONS(MAIN);
  /*  CREATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY     */

   DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
   DCL  SYSIN FILE INPUT RECORD;
   DCL  SYSIN_REC BIT(1) INIT('1'B);
   DCL 1   CARD,
```

```
         2   NAME   CHAR(20),
         2   NUMBER CHAR( 2),
         2   CARD_1 CHAR(58);
    DCL IOFIELD CHAR(20);

       ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
       OPEN FILE(NOS);
       READ FILE(SYSIN) INTO(CARD);

       DO WHILE(SYSIN_REC);
          IOFIELD = NAME;
          WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
          PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
          READ FILE(SYSIN) INTO(CARD);
       END;

       CLOSE FILE(NOS);
  END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS     DD DSN=MYID.NOS,UNIT=SYSDA,SPACE=(20,100),
//           DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G.           12
BAKER,R.            13
BRAMLEY,O.H.        28
CHEESNAME,L.        11
CORY,G.             36
ELLIOTT,D.          85
FIGGINS,E.S.        43
HARVEY,C.D.W.       25
HASTINGS,G.M.       31
KENDALL,J.G.        24
LANCASTER,W.R.      64
MILES,R.            23
NEWMAN,M.W.         40
PITT,W.H.           55
ROLF,D.E.           14
SHEERS,C.D.         21
SURCLIFFE,M.        42
TAYLOR,G.C.         47
WILTON,L.W.         44
WINSTONE,E.M.       37
/*
```

REGIONAL(1) データ・セットへのアクセスと更新

いったん REGIONAL(1) データ・セットを作成すると、SEQUENTIAL INPUT および SEQUENTIAL UPDATE、または DIRECT INPUT および DIRECT UPDATE のためにそのデータ・セットにアクセスするファイルをオープンすることができます。 そのファイルは、既存のデータ・セットを上書きする場合にのみ OUTPUT に関してオープンできます。

263 ページの表 25 は、領域データ・セットにアクセスするためのステートメントとオプションを示しています。

順次アクセス

REGIONAL(1) データ・セットを処理する SEQUENTIAL ファイルをオープンするには、INPUT 属性または UPDATE 属性を使用します。

データ伝送ステートメントには KEY オプションを指定してはなりませんが、KEYTO オプションは使えるため、ファイルは KEYED 属性を持つことができます。 KEYTO オプションで参照されるターゲット文字ストリングが 8 文字を超えると、返される値 (8 文字の領域番号) の左側にブランクが埋め込まれます。 ターゲット・ストリングが 8 文字より短い場合は、返された値の左側が切り捨てられます。

順次アクセスは、領域番号の昇順で行われます。 ダミー・レコードか実レコードかに関係なく、レコードはすべて検索されるため、PL/I プログラムにダミー・レコードを認識させておく必要があります。

REGIONAL(1) データ・セットで順次入力を使用すれば、領域番号の昇順ですべてのレコードを読み取ることができます。 また、順次更新では、順番に各レコードを読み取り、もう一度書き込むことが可能です。

REGIONAL(1) データ・セットにアクセスする SEQUENTIAL UPDATE ファイル に対する READ ステートメントと REWRITE ステートメント間の関係を決める規則は、 連続データ・セットの場合と同じです。

関連情報

233 ページの『連続データ・セットの定義と使用』
この章では、連続データセットの構成と、ストリームおよびレコード指向のデータ伝送のための連続データセットを定義する ENVIRONMENT オプションについて説明する。 その次に、伝送タイプごとに、連続データ・セットの作成、 アクセスおよび更新方法についても説明します。

直接アクセス

REGIONAL(1) データ・セットの処理に使用される DIRECT ファイルをオープンするには、INPUT 属性または UPDATE 属性を使用します。 すべてのデータ伝送ステートメントはソース・キーを 持っていなければならず、DIRECT 属性は KEYED 属性を暗黙指定します。

次の規則に従って REGIONAL(1) データ・セット内のレコードを 検索、追加、削除、または置換するには、DIRECT UPDATE ファイルを 使用します。

検索
ダミー・レコードも実際のレコードもすべて検索されます。 したがって、ユーザー・プログラムがダミー・レコードを認識できなくてはなりません。

加算
WRITE ステートメントは、ソース・キーで指定された 領域の既存レコード (実際のレコードまたはダミー・レコード) を新規レコードで置き換えます。

削除
DELETE ステートメントでソース・キーを使って指定したレコードは、 ダミー・レコードに変換されます。

置換
REWRITE ステートメントでソース・キーを使って指定したレコードは、ダミ ー・レコードであれ実際のレコードであれ変換されます。

例

ここでは、REGIONAL(1) データ・セットの更新方法について例を交えて説明します。

269 ページの『#unique_470/unique_470_Connect_42_f831』に示されているプログラムは、データ・セットを更新し、その内容をリストします。 別のレコードや更新済みレコードを書き込む前に、その領域内の 既存レコードをテストし、それがダミー・レコードであるかどうかを 確認します。これは、たとえダミーでなくても WRITE ステートメント は REGIONAL(1) データ・セット中の既存レコードを上書きできるためです。 同様に、データ・セットの内容が連続して読み取られたり印刷されたりするときは、各レコードがテストされ、ダミー・レコードは印刷されません。

REGIONAL(1) データ・セットの更新

```
//EX10    JOB
//STEP2   EXEC  IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 ACR1: PROC OPTIONS(MAIN);
   /*  UPDATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY      */
    DCL NOS FILE RECORD  KEYED ENV(REGIONAL(1));
    DCL  SYSIN FILE INPUT RECORD;
    DCL  (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
    DCL 1   CARD,
         2   NAME   CHAR(20),
         2   (NEWNO,OLDNO)  CHAR( 2),
         2   CARD_1 CHAR( 1),
         2   CODE   CHAR( 1),
         2   CARD_2 CHAR(54);
    DCL IOFIELD CHAR(20);
    DCL BYTE    CHAR(1) DEF IOFIELD;

   ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
   OPEN FILE (NOS) DIRECT UPDATE;
   READ FILE(SYSIN) INTO(CARD);

   DO WHILE(SYSIN_REC);
      SELECT(CODE);
         WHEN('A','C') DO;
             IF CODE = 'C' THEN
                DELETE FILE(NOS) KEY(OLDNO);
```

```
            READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
            IF UNSPEC(BYTE) = (8)'1'B
               THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
            ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
         END;
         WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
      OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
      END;
      READ FILE(SYSIN) INTO(CARD);
   END;

   CLOSE FILE(SYSIN),FILE(NOS);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(NOS) SEQUENTIAL INPUT;
   ON ENDFILE(NOS) NOS_REC = '0'B;
   READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   DO WHILE(NOS_REC);
      IF UNSPEC(BYTE) ¬= (8)'1'B
         THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD)(A(2),X(3),A);
      PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   END;
  CLOSE FILE(NOS);
  END ACR1;
/*
```

```
//GO.NOS   DD DSN=J44PLI.NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.          5640 C
GOODFELLOW,D.T.      89   A
MILES,R.               23 D
HARVEY,C.D.W.        29   A
BARTLETT,S.G.        13   A
CORY,G.                36 D
READ,K.M.            01   A
PITT,W.H.              55
ROLF,D.F.              14 D
ELLIOTT,D.           4285 C
HASTINGS,G.M.          31 D
BRAMLEY,O.H.         4928 C
/*
```

Updating a REGIONAL(1) data set (continued)

領域データ・セットの作成時、および領域データ・セットへのアクセス時の必須情報

領域データ・セットを作成するには、PL/I プログラムまたは DD ステートメントにおいて、データ・セットを定義する特定の情報をオペレーティング・システムに渡す必要があります。

領域データ・セットを作成するには、次の情報を与える必要があります。

- データ・セットを書き込む装置 (DD ステートメントの UNIT パラメーターまたは VOLUME パラメーター)
- ブロック・サイズ

  ブロック・サイズは、 PL/I プログラム（ ENVIRONMENT 属性の BLKSIZE オプション）か、DD ステートメント（BLKSIZE サブパラメーター）で指定できます。 レコード長を指定しないと、非ブロック化レコードがデフォルトになり、 レコード長はそのブロック・サイズから決められます。 レコード長を指定する場合は、ブロック・サイズと等しくなければなりません。

データ・セットを保持したい場合（つまり、ジョブの終了時にオペレーティング・システムに削除させたくない場合)、DD 文はデータ・セットに名前を付け、それをどのように処分するかを示さなければならない（DSNAME と DISP パラメータ)。 後のステップでデータセットを使用したいが、ジョブ終了後には必要ない場合は、 DISP パラメーターだけで十分です。

データ・セットを特定のダイレクト・アクセス・デバイスに保存したい場合は、DD 文（ **VOLUME** パラメータの SER または REF サブパラメータ）でボリューム・シリアル番号を指定する必要があります。 とっておきたいデータ・セット用の通し番号を指定しないと、 オペレーティング・システムが番号を割り振り、それをオペレーターに 通知してから、ユーザーのプログラム・リスト上にその番号を印刷します。

271 ページの表 26 には、領域データ・セットの作成に DD ステートメントで必要となるすべての必須パラメーターが要約されています。 272 ページの表 27 には、必要な DCB サブパラメーターがリストされています。 DCB サブパラメーターについては、「*z/OS MVS JCL* ユーザーズ・ガイド」を参照してください。

領域データ・セットは、システム出力 (SYSOUT) 装置上に置くことは できません。

DCB パラメータで、 **DSORG** パラメータを指定する場合は、 DSORG=DA をコード化して、データセット組織を直接指定する必要がある。 地域データセットの DD 文では、 **DUMMY** または **DSN=NULLFILE** パラメータを指定することはできません。 **DSORG=DA** を使うと、メッセージ IEC225I が発行されるかもしれない。 このメッセージは安全であり、無視できます。

表 *26.* 領域データ・セットの作成*: DD* ステートメントの必須パラメーター

パラメーター	指定すべき事項	必要な場合
UNIT= or VOLUME=REF=	出力装置 [1]	常時
SPACE=	必要なストレージ・スペース [2]	常時
DCB=	データ制御ブロック情報 272 ページの表 27 を参照してください。	常時
DISP=	後処理	データ・セットが別のジョブ・ステップで使われるが、別のジョブでは必要ない場合。
DISP=	後処理	ジョブ終了後も保持されるデータ・セット
DSNAME=	データ・セットの名前	
VOLUME=SER= or VOLUME=REF=	ボリューム通し番号	特定のボリューム上に存在するデータ・セット

ノート:

1. 領域データ・セットは、直接アクセス装置に限られます。
2. 順次アクセスの場合、データ・セットには最大 15 個のエクステントを指定できます。 そのエクステントは複数のボリューム上に配置できます。 直接アクセスによる作成の場合、データ・セットはエクステントを 1 つ だけ持つことができます。

領域データ・セットにアクセスするには、DD ステートメントを使って そのデータ・セットをオペレーティング・システムに識別する必要があります。 次項に、DD ステートメントに入れなければならない最低限の情報 を示します。この情報は 272 ページの表 28 に要約されています。

データ・セットがカタログされている場合は、DD ステートメントに次の情報だけを指定する必要があります。

- データセットの名前 (DSNAME パラメータ)

  オペレーティング・システムは、システム・カタログ内で該当のデータ・セットを 記述する情報を検出し、また、必要があれば、オペレーターに、その データ・セットが入っているボリュームをマウントするよう要求します。
- データ・セットが存在することの確認 (DISP パラメーター)

データ・セットがカタログされていない場合は、さらに データ・セットを読み取る装置、およびデータ・セットが入っているボリュームの 通し番号 (UNIT パラメーターおよび VOLUME パラメーター) を指定する必要があります。

順次更新用にマルチボリューム領域データ・セットをオープンすると、 最初のボリューム終了時に ENDFILE 条件が発生します。

表 *27. DCB* 地域データセットのサブパラメータ

サブパラメーター	指定するもの	必要な場合
RECFM=F	レコード・フォーマット [1]	常時
BLKSIZE=	ブロック・サイズ [1]	
DSORG=DA	データ・セットの編成	

[1]Or you can specify the block size in the ENVIRONMENT attribute.

表 *28.* 領域データ・セットへのアクセス*: DD* ステートメントの必須パラメーター

パラメーター	指定すべき事項	必要な場合
DSNAME=	データ・セットの名前	常時
DISP=	データ・セットの後処理	
UNIT= or VOLUME=REF=	入力装置	データ・セットがカタログ されていない場合
VOLUME=SER=	ボリューム通し番号	

第 14 章 VSAM データセットの定義と使用

この章では、レコード指向のデータ伝送のための VSAM （ Virtual Storage Access Method ）編成、 VSAM ENVIRONMENT オプション、他の PL/I データセット編成との互換性、および PL/I がサポートする 3 種類の VSAM データセット（エントリ・シーケンス、キー・シーケンス、相対レコード）のロードとアクセスに使用するステートメントについて説明します。

そしてこの章の終わりには、VSAM データ・セットを作成して それにアクセスするのに必要な、PL/I ステートメント、 アクセス方式サービス・コマンド、および JCL ステートメントの一連の例を示しています。

Enterprise PL/I は、ISAM データ・セットをサポートしていません。

VSAM の機能、 VSAM データ・セットとインデックスの構造、アクセス・メソッド・サービスによって定義される方法、および必要な JCL ステートメントに関する追加情報については、ご使用のシステムの VSAM 出版物を参照してください。

PL/I ダイナミックアロケーションを使用した VSAM ファイルの定義

DD ステートメント、環境変数、または OPEN ステートメントの TITLE オプションを使用して、VSAM データセットを定義できます。

環境変数または TITLE オプションを使用する場合、名前は大文字でなければならない。 次のように MVS データ・セットを指定してください。

```
DSN(data-set-name)
```

data-set-name は完全修飾でなければなりません。一時データセットに設置できません。 たとえば、 &から開始できません。

DSN キーワードの後に、以下の属性のいずれかを指定する必要がある：

```
OLD
SHR
```

VSAM データ・セットの使用

プログラムが VSAM データセットを使用する必要がある場合は、 DD ステートメントを指定して、プログラムがデータセットにアクセスできるようにする必要があります。 また、ご使用のプログラムは、代替索引パスを使用してキー順データ・セットおよび入力順データ・セットにアクセスすることもできます。

VSAM データセットでプログラムを実行する

あなたのプログラムが VSAM のデータセットにアクセスできるようにするには、DD ステートメントで特定の情報を提供する必要があります。

VSAM データセットにアクセスするプログラムを実行する前に、以下の情報を知っておく必要がある：

- VSAM データセットの名前
- PL/I ファイルの名前
- データ・セットを他のユーザーと共用するかどうか

その後で、そのデータ・セットにアクセスするのに必要な DD ステートメントを 作成することができます。

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

例えば、ファイルの名前が PL1FILE で、データセットの名前が VSAMDS で、データセットの排他制御を行いたい場合は、以下のように入力する：

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

© Copyright IBM Corp. 1999, 2025

データセットを共有するには、 DISP=SHR を使用してください。

Enterprise PL/I は、ISAM データ・セットをサポートしていません。

データセットに使用される VSAM バッファの数を制御して VSAM のパフォーマンスを最適化するには、VSAM の出版物を参照のこと。

代替索引パスとファイルをペア化

代替インデックスを使用する場合、PL/I ファイルにベースデータセット/代替インデックスのペアを関連付ける DD 文の DSNAME パラメータにパス名を指定するだけです。

代替索引を使用する前に、処理についての制約事項に気を付ける必要があります。制限事項は 278 ページの表 32 にまとめてあります。

PL1FILE という PL/I ファイルと PERSALPH という代替インデックス・パスがある場合、必要な DD 文は以下のようになる：

```
//PL1FILE DD DSNAME=PERSALPH,DISP=OLD
```

VSAM 編成

VSAM 、3 種類のデータセットがある。 各タイプは、大体、PL/I データ・セット編成に対応しています。 VSAM 、3 種類のデータセットはすべて順序付けされ、レコードに関連するキーを持つことができる。 3 つのタイプ全部において、順次アクセスとキーによるアクセスの両方を行うことができます。

表 *29. VSAM* データセットの種類と対応する *PL/I* データセットの構成

VSAM データ・セット・タイプ	**対応する PL/I データ・セット編成**
キー順データ・セット (KSDS)	索引付きデータ・セット
入力順データ・セット (ESDS)	連続データ・セット
相対レコード・データ・セット (RRDS)	領域データ・セット

キー順データ・セットだけがその論理レコードの一部としてキーを持つことができますが、入力順データ・セット (相対バイト・アドレスを使用) および相対レコード・データ・ セット (相対レコード番号を使用) でもキー順アクセスを行うことができます。

すべての VSAM データセットはダイレクトアクセスストレージデバイスに保持され、それらを使用するには仮想ストレージオペレーティングシステムが必要である。

VSAM データセットの物理的な構成は、他のアクセス方法で使われるものとは異なる。 VSAM はブロッキングの概念を使わず、相対レコードのデータセットを除けば、レコードは固定長である必要はない。 VSAM 編成のデータ・セットでは、 データ項目は制御インターバル に並べられ、 さらにそれが制御域 内に並べられます。 処理に備えて、制御インターバル内のデータ項目は論理レコード内に並べられます。 制御インターバルは、1 つ以上の論理レコードを収容することができ、1 つの論理 レコードが複数の制御インターバルにスパンしてもかまいません。 ブロッキング要因やレコードの長さに関する懸念は、 VSAM によってほぼ取り除かれるが、レコードは指定された最大サイズを超えることはできない。 VSAM は制御インターバルへのアクセスを許可するが、この種のアクセスは ではサポートされていない。 PL/I

VSAM データセットには、プライムとオルタネートの 2 種類のインデックスがあります。 プライム・インデックスとは、データセットを定義するときに確立される KSDS へのインデックスのことで、常に存在し、KSDS に対する唯一のインデックスとなり得る。 KSDS 、または ESDS に 1 つ以上の代替インデックスを持つことができます。 ESDS に代替インデックスを定義することで、ESDS を一般的に KSDS。 KSDS の代替インデックスは、プライム・インデックスとは異なる論理レコードのフィールドをキー・フィールドとして使用できるようにする。 代替索引は重複キーが使用できる非固有、重複キーが使用できない固有 のいずれかにすることができます。 基本索引には、重複キーがあってはなりません。

代替索引を持つデータ・セット内の変更は、今後も使用するすべての索引に反映されなければなりません。この活動はインデックス・アップグレードと呼ばれ、 VSAM 、データセットのインデックス・アップグレード・セット内の任意のインデックスに対して行われる。 (KSDS の場合、素数のインデックスは常にイン

デックス・アップグレード・セットのメンバーである) しかし、基本索引または固有代替索引に重複キーを作成する、データ・セット内の変更は避けてください。

初めて VSAM データセットを使用する前に、Access Method Services の DEFINE コマンドを使用して、そのデータセットをシステムに定義する必要があります。DEFINE コマンドは、データセットのタイプ、構造、必要なスペースを完全に定義するために使用できます。 このコマンドはまた、データセットのインデックス（キーの長さと位置とともに)、およびデータセットが KSDS 、または 1 つ以上の代替インデックスを持つ場合のインデックス・アップグレード・セットも定義します。 こうして、アクセス・メソッド・サービスによって、 VSAM データセットが"作成"される。

新たに作成された VSAM データセットに初期データを書き込む操作を、本書ではロードと呼ぶ。

3 つのタイプのデータ・セットは、次の各目的別に使用します。

- 入力順データ・セット は、主として作成された順序 (またはその逆の順序) でアクセスするデータの場合に使用します。
- キー順データ・セット は、通常どおりにレコード内の キーを介してレコードにアクセスする場合に使用します (例えば、レコードに アクセスするのに部品番号が使用される在庫制御ファイル)。
- 相対レコード・データ・セット は、各項目が特定の番号を持っており、 その番号によって相対レコードに通常アクセスするようなデータに 使用します (例えば、各番号に関連したレコードを持った電話システム)。

すべてのタイプの VSAM データセットのレコードに、キーで直接アクセスすることも、順次（バックワードまたはフォワード）アクセスすることもできます。 2 方向の組み合わせを使用することもできます。 キーで開始点を選択し、その点から順方向あるいは逆方向に 読み取りを行います。

キー順および入力順データ・セットに代替索引 を作成できます。 作成後は、 多くのシーケンスを使用して、または多くのキーのうちのいずれかを使用して、ユーザー・データにアクセスできます。 例えば、 従業員番号順に保持されているか、索引を付けられているデータ・セットを使用して、 代替索引 内に名前でそれに索引を付けることができます。 それから、英字順、逆英字順、または名前を直接キーとして使用して、そのデータ・セットにアクセスできます。 従業員番号の同じ種類の組み合わせによって、データ・セットにアクセスすることもできます。

275 ページの表 30 に、同一データをどのように この 3 つの異なったタイプの VSAM データ・セット内に保持できるかを示し、 それぞれの利点と欠点を示しています。

表 *30. VSAM* データセットの種類と利点

データ・セット・タイプ	ロード方式	読み取り方式	更新方式	利点と欠点
キー順	順次に並んだ索引または固有でなければならない基本索引。	KEYED プライム・インデックスにレコードのキーを指定する SEQUENTIAL インデックスの順番を前後させる キーで位置決めした後に、逆方向または順方向の順次読み取り。	KEYED 任意のインデックスで一意なキーを指定する SEQUENTIAL 一意のキーによる次のポジショニング レコードの削除が許可される。 レコードの追加が許可される。	**利点** : 完全アクセスおよび更新。 **欠点** : ロード前にレコードが基本索引順にソートされていなければならない。 **使用** : アクセスがキーと関連する場合に使用。

表 30. VSAM データセットの種類と利点 (続き)

データ・セット・タイプ	ロード方式	読み取り方式	更新方式	利点と欠点
入力順	順次 (順方向のみ)。 各レコードの RBA を入手して、キーとして使用可能。	SEQUENTIAL 前後 KEYED RBA 使用 キーで位置決めをした後で、逆方向または順方向に順次。	新しいレコードは終わりにのみ。 既存レコード長は変更不可。 レコードの削除不可。	**利点** : 簡単迅速に作成。 固有索引は不要。 **欠点** : 限定された更新機能。 **使用** : データが主として順次式でアクセスされる場合。
相対レコード	スロット 1 から順次。 スロット番号を指定して KEYED。 キーで位置決めした後で、順次書き込み。	KEYED 数字をキーに指定 空きレコードを省いて、順方向または逆方向へ順次。	指定したスロットから開始し、次のスロットへ順次。 番号をキーとして指定して KEYED。 レコードの削除が許可される。 空きスロットへのレコード追加が許可される。	**利点** : 番号によるレコードへの高速アクセス。 **欠点** : 構造が番号付けに拘束される。 固定長レコード。 **使用** : レコードが番号によってアクセスされる場合に使用。

VSAM データセットのキー

すべての VSAM データセットは、そのレコードに関連するキーを持つことができる。

キー順データ・セット、および代替索引 を介して入力順データ・セットにアクセスする場合は、そのキーは論理レコード内の定義されたフィールドになります。 エントリー・シーケンス・データセットの場合、キーはレコードの *relative byte address* (RBA) である。 相対レコード・データ・セットの場合、キーは、相対レコード 番号 になります。

インデックス付き VSAM データセットのキー

キー順データ・セットのキー、および代替索引 を使用してアクセスされる入力順データ・セットのキーは、データ・セットに記録されている論理レコードの一部です。 データ・セットの作成時に、キーの長さと位置を定義します。

KEY, KEYFROM、KEYTO オプションのキーを参照する方法については、KEY（式）オプション、KEYFROM（式）オプション、 KEYTO （参照）オプションのトピックを参照してください。 *PL/I Language Reference*.

相対バイトアドレス (RBA)

相対バイト・アドレスは、 KEYED SEQUENTIAL ファイルに関連付けられた ESDS でキー・アクセスを使用することを可能にする。

RBAs（キー）は長さ 4 の文字列で、その値は VSAM で定義される。 RBA PL/I しかし、データセット内のレコードの相対的な位置を決定するために、その値を比較することはできる。 RBA は通常は印刷できません。

KEYTO オプションを使用して、レコードの RBA を取得することができます。 オプションは、データセットのロードまたは拡張時に WRITE ステートメントで使用するか、データセットの読み取り時に READ ステートメントで使用します。 その後、 READ または REWRITE 文の KEY オプションで、これらのいずれかの方法で取得した RBA を使用することができる。

WRITE 文の KEYFROM オプションに RBA を使用しないこと。

VSAM は、相対バイトアドレスを のキーとして使用できるが、この使用は ではサポートされていない。
KSDS PL/I

相対レコード番号

RRDS のレコードは、1 から始まり、続くレコードごとに 1 ずつ増える相対レコード番号で識別される。 この相対レコード番号は、データ・セットへのキー順アクセスで、キーとして使用することができます。

相対レコード番号として使用されるキーは、長さ 8 の文字ストリングです。 KEY または KEYFROM オプションで使用するソース・キーの文字値は、符号なし整数を表す必要があります。 ソース・キーが 8 文字の長さでなければ、左側で切り捨てられるか、 あるいは、ブランク (ゼロと解釈される) が埋められます。 KEYTO オプションが返す値は、先頭のゼロを抑えた長さ 8 の文字列である。

データ・セット・タイプの選択

プログラムを計画するときは、使用するデータ・セットのタイプを最初に決定しなければなりません。 3 種類の VSAM データセットと、5 種類の VSAM 以外のデータセットが用意されている。

VSAM データセットは、他のタイプのデータセットのすべての機能に加え、 でのみ利用可能な追加機能を提供することができる。 VSAM VSAM は通常、パフォーマンスにおいて他のデータセット・タイプに匹敵し、しばしばそれを上回ることができる。 しかし、 VSAM 、機能の誤用による性能劣化の可能性が高くなる。

8 つのタイプのデータ・セットをすべて比較するには、223 ページの表 17 を参照してください。 ただし、大規模インストール・システムでデータ・セット・タイプを選択するときに影響する要因の多くは、本書で説明できる範囲を超えています。

VSAM 、データセットの種類を選択する際には、データを必要とする最も一般的なシーケンスに基づいて選択する必要があります。 次の手順は、データ・セットと索引を組み合わせて必要な機能を得るときに役立ちます。

1. データ・タイプを決めて、それにアクセスする方法を決めます。
   a. 主として順次の場合 - ESDS を優先
   b. 主としてキーによる場合 - KSDS を優先
   c. 主として番号による場合 - RRDS を優先
2. データ・セットのロード方法を決めます。 KSDS 従って、アプリケーションによっては、 別のインデックス・ パスを持つ ESDS の方が、より実用的な代替手段となりうる。
3. 代替索引 パスを使用してアクセスする必要があるかどうかを決めます。 これらは KSDS と KSDS でのみサポートされている。 代替索引 パスが必要な場合は、 代替索引 が固有キーを持つか、非固有キーを持つかを決めます。 非固有キーを使用する場合は、 キー処理の制限が生じます。 しかし、 今後のレコードすべてに固有キーを使用するということは実用的ではありません。 固有キー用に作成した索引に非固有キーを持つレコードを挿入しようとすると、 エラーが発生します。
4. ユーザーが必要とするデータ・セットとパスが決まったら、ユーザーが想定している操作がサポート されているかどうか確認してください。 278 ページの表 31 を参考にしてください。

ダミーの VSAM データ・セットにはアクセスしないでください。 未定義ファイルがあることを示すエラー・メッセージを受け取るためです。

278 ページの表 31 は、VSAM データ・セットのタイプに基づく、矛盾のないファイル属性の組み合わせを示しています。

表 *31. VSAM* データ・セットと使用できるファイル属性

	SEQUENTIAL	**KEYED SEQUENTIAL**	**DIRECT**
INPUT	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	
OUTPUT	ESDS	ESDS	KSDS
	RRDS	KSDS	RRDS
		RRDS	Path(U)
UPDATE	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	

Key :

ESDS
Entry-sequenced data set

KSDS
Key-sequenced data set

RRDS
Relative record data set

Path(N)
Alternate index path with nonunique keys

Path(U)
Alternate index path with unique keys

Notes :

- You can combine the attributes on the left with those at the top of the figure for the data sets and paths shown. For example, only an ESDS and an RRDS can be SEQUENTIAL OUTPUT.
- PL/I does not support dummy VSAM data sets.

表 *32.* 代替索引パスで実行できる処理

基本クラスター・タイプ	**代替索引キー・タイプ**	**処理中**	**制約事項**
KSDS	固有キー	通常の KSDS として	アクセスのキーは変更できません。
	非固有キー	制限つきキー・アクセス	アクセスのキーは変更できません。

表 32. 代替索引パスで実行できる処理 (続き)

基本クラスター・タイプ	代替索引キー・タイプ	処理中	制約事項
ESDS	固有キー	As KSDS	削除なし。 アクセスのキーは変更できません。
	非固有キー	制限つきキー・アクセス	削除なし。 アクセスのキーは変更できません。

関連情報

283 ページの『入力順データ・セット』
このトピックでは、入力順データ・セット (ESDS) に関連付けられているファイルに対して使用できるステートメントとオプションについて説明します。

286 ページの『キー順および索引付き入力順データ・セット』
インデックス付きデータセットは、プライムインデックスを持つキー配列データセット(KSDS)とすることができる。また、 代替インデックスを持つ KSDS またはエントリー配列データセット(ESDS)とすることもできる。 このトピックでは、インデックスされた VSAM データセットに関連するファイルで許されるステートメントとオプションについて説明します。

297 ページの『相対レコード・データ・セット』
このトピックでは、VSAM 相対レコード・データ・セット (RRDS) に関連するファイルで使用できるステートメントとオプションについて説明します。

VSAM データセット用ファイルの定義

このトピックでは、シーケンシャルまたはダイレクト VSAM データセットを定義するために使用できるファイル宣言について説明します。

シーケンシャル VSAM データセットの定義

シーケンシャル VSAM データ・セットを定義するには、以下の属性を持つファイル宣言を使用する：

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

ダイレクト VSAM データセットの定義

ダイレクト VSAM データセットは、以下の属性を持つファイル宣言を使用して定義する：

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
            [KEYED]
             ENVIRONMENT(options);
```

215 ページの表 15 はデフォルト属性を示しています。 ファイル属性については、「*PL/I Language Reference*」で説明されています。 ENVIRONMENT 属性のオプションについては、 280 ページの『ENVIRONMENT オプションの指定』 を参照のこと。

ファイル属性 INPUT, OUTPUT、または UPDATE と DIRECT, SEQUENTIAL、または KEYED SEQUENTIAL のいくつかの組み合わせは、 VSAM データセットの特定のタイプにのみ許可されます。 278 ページの表 31 に互換性のある組み合わせを示しています。

ENVIRONMENT オプションの指定

このセクションでは、 VSAM データセットに適用される ENVIRONMENT オプションについて説明します。

データセットの構造に影響を与える ENVIRONMENT 属性のオプションの多くは、VSAM データセットには必要ありません。 このオプションを指定しても、無視されるか、または検査の目的で使用されるだけです。チェックされた値がデータセットに定義された値と矛盾する場合、ファイルを開こうとすると UNDEFINEDFILE 。

VSAM データセットには、以下の ENVIRONMENT オプションを使用できます：

```
BKWD
BUFND (n)
BUFNI (n)
BUFSP (n)
GENKEY
PASSWORD (password-specification)
REUSE
SCALARVARYING
SKIP
VSAM
```

GENKEY オプションと SCALARVARYING オプションは、VSAM 以外のデータセットに使用した場合と同じ効果を持つ。 VSAM RLS では、オプション BUFND, BUFNI、および BUFSP は無視されることに注意。

VSAM データセットでチェックされるオプションは RECSIZE 、キー配列データセットの場合は KEYLENGTH 、KEYLOC である。 215 ページの表 15 は、VSAM の場合に無視されるオプションを示しています。 215 ページの表 15 には、必須オプションとデフォルト・オプションも示されています。

VSAM データセットの場合、データセットを定義するときに、レコードの最大長と平均長を Access Method Services ユーティリティに指定します。 チェックのためにファイル宣言に RECSIZE オプションを含める場合は、最大レコードサイズを指定する。 指定した RECSIZE 値がデータセットに定義された値と矛盾する場合、 UNDEFINEDFILE 。

BKWD

BKWD オプションは、 VSAM データセットに関連する SEQUENTIAL INPUT または SEQUENTIAL UPDATE ファイルの後方処理を指定する。

順次読み取り (すなわち、KEY オプションなしの読み取り) では、直前の順序にあるレコ ードが検索されます。 索引付きデータ・セットの場合は、直前のレコードは、 一般的に、その次の低位キーを持つレコードのことです。 しかし、非固有な代替索引 を介してデータ・セットにアクセスしている場合は、同じキーを持つレコードは通常のシーケンスでリカバリーされます。 例えば、次のレコードを考えてみます。 C1、C2、および C3® のキーは同じです。

```
A B C1 C2 C3 D E
```

レコードは次の順序でリカバリーされます。

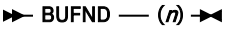
```
E D C1 C2 C3 B A
```

BKWD オプションを指定したファイルを開くと、データセットは最後のレコードに配置される。 ENDFILE は、データセットの先頭に達したときに通常の方法で発生する。

REUSE オプションまたは GENKEY オプションと一緒に BKWD オプションを指定しないでください。 また、WRITE 文は、 BKWD オプションで宣言されたファイルには使用できない。

BUFND

BUFND オプションは、 VSAM データセットに必要なデータバッファの数を指定する。



n
整数、または属性 FIXED BINARY(31) STATIC を持つ変数を指定する。

ファイルに SEQUENTIAL 属性があり、連続した長いレコード群を順次処理する場合、複数のデータ・バッファがパフォーマンスを向上させる。

BUFNI

BUFNI オプションは、 VSAM キー・シーケンス・データセットに必要なインデックス・バッファの数を指定する。

n
整数、または属性 FIXED BINARY(31) STATIC を持つ変数を指定する。

ファイルに KEYED 属性がある場合、複数のインデックス・バッファがパフォーマンスを向上させます。 少なくとも、索引のレベル数と同数の索引バッファー数を指定してください。

BUFSP

BUFSP オプションは、 VSAM データセットに必要な（データとインデックスの両方のコンポーネントの）合計バッファスペースをバイト単位で指定します。

n
整数、または属性 FIXED BINARY(31) STATIC を持つ変数を指定する。

通常、 BUFSP ではなく、 BUFNI と BUFND オプションを指定するのが望ましい。

GENKEY

GENKEY (総称キー) オプションは、INDEXED キー順データ・セット および VSAM キー順データ・セットにのみ適用されます。 このオプションを使用すれば、データ・セットに記録されているキーを分類したり、SEQUENTIAL KEYED INPUT ファイルや SEQUENTIAL KEYED UPDATE ファイルでキー・クラスに従ってレコードにアクセスしたりできます。

詳しくは、221 ページの『GENKEY オプション ― キーの分類』を参照してください。

パスワード

システムに VSAM データセットを定義するとき（Access Method Services の DEFINE コマンドを使用する）、READ と UPDATE パスワードを関連付けることができる。 その時点から、データ・セットへのアクセスに使用する PL/I ファイルの宣言には、該当するパスワードを組み込む必要があります。

password-specification
プログラムで必要とされるアクセスのタイプに対するパスワードを指定する文字定数または文字変数を指定します。 定数を指定する場合は、反復因数を含めることはできません。変数を指定する場合は、レベル 1、エレメント、静的、および添え字なしでなければなりません。

文字ストリングは埋め込みまたは切り捨てが行われて 8 文字にされた後、VSAM に渡されて検査されます。誤ったパスワードだった場合、システム・オペレーターには、正しいパスワードを指定するための機会が何回か与えられます。 その許容回数は、データ・セットを定義するときに指定します。 この回数トライに失敗すると、 UNDEFINEDFILE 。

REUSE

REUSE オプションは、VSAM データセットに関連付けられた OUTPUT ファイルをワークファイルとして使用することを指定する。

データ・セットは、ファイルをオープンするたびに、空のデータ・セットとして扱われます。 そのデータ・セット用のすべての 2 次割り振りが解放され、 データ・セットは初めてオープンされたときと 同様に取り扱われます。

REUSE オプションのあるファイルは、 代替インデックスや BKWD オプションのあるデータセットと 関連づけず、 INPUT または UPDATE 用に開かないでください。

REUSE オプションは、Access Method Services DEFINE CLUSTER コマンドで REUSE を指定した場合にのみ有効になります。

SKIP

SKIP オプションは、可能であれば常に VSAM OPTCD の「SKP」を使用することを指定します。 このオプションは、KEYED SEQUENTIAL INPUT または UPDATE ファイルを使用してアクセスするキー順データ・セットに適用できます。

このオプションは、データ・セット全体に分散している個々のレコードにプログラムがアクセスし、そのアクセスが主に昇順のキー順で行われる場合に、ファイルに指定します。

プログラムが KEY オプションを使用せずに多数のレコードを順次読み取る場合、またはプログラムがデータ・セット内の特定のポイントに多数のレコードを挿入する場合は (大量順次挿入)、このオプションを省略してください。

SKIP オプションを指定 (または省略) することは、決して間違った使い方ではありません。このオプションがパフォーマンスに大きく影響するのは、前述の場合のみです。

VSAM

VSAM データセットには VSAM オプションを指定する必要があります。

パフォーマンス・オプション

DD ステートメントの AMP パラメーター内にバッファー・オプションを指定するこ ともできます。これらは、アクセス方式サービスの資料に解説があります。

VSAM のパフォーマンスの最適化について詳しくは、DFSMS データ・セットの使用法の資料を参照してください。 共有リソースについて詳しくは、MVS Batch Local Shared Resources の資料を参照してください。

代替索引パスのファイルを定義

VSAM を使用すると、キー配列およびエントリー配列のデータセットに代替インデックスを定義できます。

代替索引を使用すれば、基本索引以外のいくつかの方法でキー順データ・セットにアクセスできます。 また、入力順データ・セットに索引を付け、キーでアクセスすることも、キーの順番にアクセスすることもできます。 その結果、1 つの形式で作成されたデータに、さまざまな方法で アクセスできるようになります。 例えば、ある従業員ファイルが、個人番号、名前、および部門番号によって 索引付けられていたとします。

代替索引が作成されると、ユーザーは、実際は代替索引パス (代替索引とデータ・セット間の接続として機能する) と された 3 番目のオブジェクトを通じてデータ・セットにアクセスします。

ここでは、2 つのタイプの代替索引が使用できます。固有キーと非固有キーです。 固有キー代替索引の場合は、それぞれのレコードが 別の代替キーを持っていなければなりません。 非固有キー代替索引の場合は、 任意の数のレコードが同じ代替キーを持つことができます。 上記の例では、名前が使用されている代替索引は固有キー代替索引にすることができます (各人の名前が異なる場合)。 部門番号を使用している代替索引は、 各部門に複数の人が所属することができるので非固有キー代替索引になります。

ほとんどの点で、ユニークキー・オルタネートインデックスパスを介してアクセスされるデータセットは、プライムインデックスを介してアクセスされる KSDS のように扱うことができます。 レコードはキーによって、または順次にアクセスでき、ユーザーはレコードを更新し、さらに新規のレコードを追加できます。データセットが KSDS の場合、レコードを削除したり、更新されたレコードの長さを変更したりすることができる。 制約事項と許可されている処理については、278 ページの表 32 で示しています。 レコードを追加または削除したら、 データ・セットに関連したすべての索引が、新しい状況を反映するために変更されます (デフォルト)。

非固有キー代替索引パスを使用してデータ・セットにアクセスする場合、 アクセスされるレコードはキーとシーケンスによって判別されます。 キーを使用して位置決めを行えば、順次アクセスを続けることができます。 キーを使用して、最初のレコードにアクセスできます。 データ・セットを逆方向に読み取る場合は、 キーの順番のみが逆になります。 同じキーを持つレコードの順番は、 データ・セットをいずれの方向で読み取るにしても、同じままです。

VSAM データ・セットの定義

Access Method Services の DEFINE CLUSTER コマンドを使用して、 VSAM データセットを定義し、カタログ化することができる。

DEFINE コマンドを使用するには、以下の情報が必要です：

- マスター・カタログがパスワード保護されていれば、マスター・カタログの名前とパスワード。
- マスターカタログを使用しない場合に使用する VSAM プライベートカタログの名前とパスワード
- データセット用の VSAM スペースが利用可能かどうか
- 作成する VSAM データセットのタイプ
- データ・セットを入れる先のボリューム。
- データ・セット内の平均および最大レコード・サイズ。
- 索引付きデータ・セット用のキーの位置と長さ。
- データ・セットに割り振られるスペース。
- DEFINE
- Access Method Services プログラムの使用方法

情報が得られたら、 DEFINE コマンドをコーディングし、 Access Method Services を使ってデータセットを定義し、カタログを作成することができる。

入力順データ・セット

このトピックでは、入力順データ・セット (ESDS) に関連付けられているファイルに対して使用できるステートメントとオプションについて説明します。

表 *33. VSAM* 入力順データ・セットのロードと入力順データ・セットへのアクセスで使用できるステートメントとオプション

ファイル宣言 [1]	有効ステートメントおよび	含めることができる その他のオプション
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	LOCATE 基底付き変数 FILE(file-reference);	SET(pointer-reference)

表 33. *VSAM* 入力順データ・セットのロードと入力順データ・セットへのアクセスで使用できるステートメントとオプション (続き)

ファイル宣言[1]	有効ステートメントおよび	含めることができる その他のオプション
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) または KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) または KEY(expression)[3]
	READ FILE(file-reference);	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) または KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) または KEY(expression)[3]
	READ FILE(file-reference)[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) および/または KEY(expression)[3]

ノート:

1. 完全なファイル宣言には 属性 FILE、RECORD、および ENVIRONMENT が含まれます。 オプションの KEY あるいは KEYTO のいずれかを使用する場合は、属性 KEYED も 含めなくてはなりません。
2. ステートメント `READ FILE(file-reference);` は、ステートメント `READ FILE(file-reference) IGNORE (1);` と同等です。
3. KEY オプションにおいて使用される式は、あらかじめ KEYTO オプションで取得された相対バイト・アドレスでなければなりません。

ローディング ESDS

ESDS がロードされるとき、関連ファイルは SEQUENTIAL OUTPUT のために開かれなければならない。 レコードは、提示された順序で保持されます。

KEYTO オプションを使用すると、各レコードが書き込まれたときの相対バイト・アドレスを取得できる。 後でこれらのキーを使用すれば、このデータ・セットにキーによるアクセスを行うことができます。

SEQUENTIAL 。 ESDS

ESDS へのアクセスに使用される SEQUENTIAL ファイルは、INPUT 属性または UPDATE 属性のいずれかで開くことができます。 KEY または KEYTO のいずれかのオプションを使用する場合、ファイルにも KEYED 属性が必要です。

順次アクセスの順序は、レコードをデータ・セットに初めにロードしたときと同じです。 READ 文で KEYTO オプションを使用すると、読み込まれたレコードの RBA を回復することができる。 KEY オプションを使用すると、復元されるレコードは指定した RBA を持つレコードとなる。 次の順次アクセスは、データ・セットの新しい場所から開始されます。

UPDATE ファイルの場合、WRITE 文はデータセットの最後に新しいレコードを追加する。 REWRITE ステートメントでは、KEY オプションを使用した場合、書き換えられるレコードは指定された RBA を持つレコードになります。それ以外の場合は、前の READ でアクセスされたレコードになります。 REWRITE 、置き換えられるレコードの長さを変更しようとしてはならない。

エントリー配列のデータセットでは、 DELETE ステートメントは使用できない。

を定義してロードする。 ESDS

このトピックでは、エントリー配列データセット(ESDS)を定義し、ロードする PL/I プログラムの例を示す。このプログラムは SEQUENTIAL OUTPUT ファイルを使ってデータセットに書き込む。

285 ページの『#unique_500/unique_500_Connect_42_f97b』 に示すプログラムでは、データセットは DEFINE CLUSTER コマンドで定義され、 PLIVSAM.AJC1.BASE という名前が与えられている。 NONINDEXED キーワードを指定すると、 ESDS が定義される。

PL/I プログラムは、 SEQUENTIAL OUTPUT ファイルと WRITE FROM ステートメントを使ってデータセットに書き込む。 ファイルの DD 文には、DEFINE CLUSTER コマンドの NAME パラメータで指定されたデータセットの DSNAME が含まれる。

レコードの RBA を KEYED ファイルのキーとして取得し、以後使用することができる。 そのためには、キーを保持する適切な変数を宣言し、WRITE...KEYTO ステートメントを使用しなければならない。 次の例を参照してください。

```
DCL CHARS CHAR(4);
WRITE FILE(FAMFILE) FROM (STRING)
 KEYTO(CHARS);
```

通常、キーは印刷できませんが、 後で使用するときのためにとっておくことができることに注意してください。

カタログの手順 IBMZCBG 。 同じプログラム (285 ページの『#unique_500/unique_500_Connect_42_f97b』 参照) を使ってデータ・セットにレコードを 追加できるため、このプログラムはライブラリーに保持されています。 レコードを追加する手順について詳しくは、286 ページの『更新 ESDS』 にある例を参照してください。

ESDS の定義とロード

```
//OPT9#7  JOB
//STEP1    EXEC   PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD    *
     DEFINE CLUSTER -
      (NAME(PLIVSAM.AJC1.BASE) -
      VOLUMES(nnnnnn) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2))
/*
//STEP2  EXEC IBMZCLG
//PLI.SYSIN      DD *
   CREATE: PROC OPTIONS(MAIN);

      DCL
        FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
        IN FILE RECORD INPUT,
        STRING CHAR(80),
        EOF BIT(1) INIT('0'B);

      ON ENDFILE(IN) EOF='1'B;

      READ FILE(IN) INTO (STRING);
      DO I=1 BY 1 WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
        WRITE FILE(FAMFILE) FROM (STRING);
        READ FILE(IN) INTO (STRING);
      END;

      PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
    END;
/*
//LKED.SYSLMOD  DD  DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
```

```
//         UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.FAMFILE DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=OLD
//GO.IN      DD     *
FRED                    69          M
ANDY                    70          M
SUZAN                   72          F
/*
```

更新 ESDS

このトピックでは、 ESDS の最後に新しいレコードを追加する例を示す。

この例は、PL/I 表示されるプログラム 285 ページの『#unique_500/unique_500_Connect_42_f97b』。 このプログラムは、レコードをデータ・セットに追加する場合にも使用できます。 プログラムでは、データセット PLIVSAM.AJC1.BASE は DEFINE CLUSTER コマンドで定義される。

286 ページの『#unique_501/unique_501_Connect_42_f98』 は、 の最後に新しいレコードが追加されたことを示している。 ESDS SEQUENTIAL OUTPUT ファイルが使用され、それに関連するデータセット PLIVSAM.AJC1.BASE が **DSNAME** パラメーターで指定される。

ESDS の更新

```
//OPT9#8   JOB
//STEP1    EXEC  PGM=PGMA
//STEPLIB   DD   DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP)
//           DD   DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//FAMFILE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//IN        DD   *
JANE                    75          F
//
```

ESDS、レコードの長さを変えない限り、既存のレコードを書き換えることができる。 これには、SEQUENTIAL または KEYED SEQUENTIAL 更新ファイルを使用することができます。 キーを使用する場合は、RBA または 代替索引 パスのキーを使用することができます。

Delete ステートメントは ESDS では使用できません。

キー順および索引付き入力順データ・セット

インデックス付きデータセットは、プライムインデックスを持つキー配列データセット(KSDS)とすることができる。また、 代替インデックスを持つ KSDS またはエントリー配列データセット(ESDS)とすることもできる。 このトピックでは、インデックスされた VSAM データセットに関連するファイルで許されるステートメントとオプションについて説明します。

特に断り書きがなければ、286 ページの表 34 における説明はすべての索引付き VSAM データ・セットに当てはまります。

表 *34. VSAM* 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)

表 *34. VSAM* 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション (続き)

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference)	KEY(expression)
DIRECT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

表 *34. VSAM* 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション (続き)

File declaration[1]	Valid statements with options you must include	Other options you can also include
DIRECT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); READ FILE(file-reference) SET(pointer-reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); DELETE FILE(file-reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Notes :

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the declaration.
2. The statement `READ FILE(file-reference);` is equivalent to the statement `READ FILE(file-reference) IGNORE(1);`.
3. Do not associate a SEQUENTIAL OUTPUT file with a data set accessed through an *alternate index*.
4. Do not associate a DIRECT file with a data set accessed through a nonunique *alternate index*.
5. DELETE statements are not allowed for a file associated with an ESDS accessed through an *alternate index*.

KSDS またはインデックス付き ESDS

KSDS がロードされている場合、関連するファイルを開いて KEYED SEQUENTIAL OUTPUT を参照する必要がある。 レコードをキーの昇順で表示し、 KEYFROM オプションを使用する必要があります。

データ・セットをロードするためには基本索引を使用しなければならないことに注意してください。代替索引 を介して VSAM データ・セットをロードすることはできません。

KSDS 、すでにいくつかのレコードが含まれており、SEQUENTIAL 属性と OUTPUT 属性で関連ファイルを開いた場合、データセットの最後にのみレコードを追加することができます。 前項のルールが適用されます。 特に、指定する最初のレコードはデータ・セット上に存在する 最高位のキーより大きいキーを持っていなくてはなりません。

289 ページの『#unique_502/unique_502_Connect_42_f912』 は、 を定義するために使用される DEFINE コマンドを示している。 KSDS このデータセットには PLIVSAM.AJC2.BASE という名前が与えられ、 INDEXED オペランドが使用されているため、 KSDS として定義されている。 レコード内のキーの位置は、 KEYS オペランド内で定義されます。

PL/I プログラムでは、 KEYED SEQUENTIAL OUTPUT ファイルと WRITE...FROM...KEYFROM ステートメントが使用される。 データは、昇順のキー順で示されます。 KSDS 。

ファイルは、DEFINE コマンドで指定された名前を DSNAME パラメータとして使用する DD ステートメントによってデータセットと関連付けられる。

キー順データ・セット (KSDS) の定義とロード

```
//OPT9#12  JOB
//   EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT  DD  SYSOUT=A
//SYSIN     DD  *
   DEFINE CLUSTER -
     (NAME(PLIVSAM.AJC2.BASE) -
     VOLUMES(nnnnnn) -
     INDEXED -
     TRACKS(3 1) -
     KEYS(20 0) -
     RECORDSIZE(23 80))
/*
//   EXEC IBMZCBG
//PLI.SYSIN      DD *
 TELNOS: PROC OPTIONS(MAIN);

        DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD POS(1),
            NUMBER CHAR(3) DEF CARD POS(21),
            OUTREC CHAR(23) DEF CARD POS(1),
            EOF BIT(1) INIT('0'B);

        ON ENDFILE(SYSIN) EOF='1'B;

        OPEN FILE(DIREC) OUTPUT;

        GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
        WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;

       CLOSE FILE(DIREC);

        END TELNOS;
/*
//GO.DIREC  DD  DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN  DD  *
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,D.        141
CORY,G.             336
ELLIOTT,D.          875
FIGGINS,S.          413
HARVEY,C.D.W.       205
HASTINGS,G.M.       391
KENDALL,J.G.        294
LANCASTER,W.R.      624
MILES,R.            233
NEWMAN,M.W.         450
PITT,W.H.           515
ROLF,D.E.           114
SHEERS,C.D.         241
SUTCLIFFE,M.        472
TAYLOR,G.C.         407
WILTON,L.W.         404
WINSTONE,E.M.       307
//
```

KSDS 、またはインデックスされたファイルにアクセスするために SEQUENTIAL 。 ESDS

INPUT または UPDATE 属性で、KSDS にアクセスするために使用される SEQUENTIAL ファイルを開くことができます。

KEY オプションのない READ ステートメントでは、レコードはキーの昇順（ BKWD オプションが使用されている場合はキーの降順）で回復される。 KEYTO オプションを使用することによって、このように回復されたレコードのキーを得ることができます。

KEY オプションを使用すると、READ ステートメントで回復されるレコードは、 指定したキーを持つレコードになります。 このような READ ステートメントはデータ・セットを指定されたレコードに 位置付け、その後の順次読み取りは後続のレコードを順番に回復します。

KEYFROM オプションを指定した WRITE ステートメントは、KEYED SEQUENTIAL UPDATE ファイルに対して許可される。 前回行ったアクセスの位置に関係なく、データ・セット内の任意の位置に 挿入を行うことができます。 固有索引によってデータ・セットにアクセスするときは、そのデータ・セットに既にあるレコードと同じキーを持つレコードを挿入しようとすると、KEY 条件が生じます。 非固有索引の場合には、同一キーを持つレコードを後になって検索すると、 レコードがデータ・セットに追加された順に、検索が行われます。

UPDATE ファイルでは、REWRITE ステートメントは、KEY オプションのあるなしに 関係なく使用できます。 KEY オプションを使用した場合、再書き込みされるレコードは、指定されたキーを持つ 最初のレコードであり、それ以外の場合は直前の READ ステートメントによってアクセスされた レコードです。 代替索引 を使用してレコードを再書き込みする場合は、レコードの基本キーを変更しないでください。

KSDS 、またはインデックスされたファイルにアクセスするために DIRECT 。 ESDS

INPUT, OUTPUT, または UPDATE 属性を持つインデックス付き VSAM データセットにアクセスするために使用される DIRECT ファイルを開くことができます。 一意でないインデックスを通してデータセットにアクセスするために、 DIRECT ファイルを使用しないでください。

データセットにレコードを追加するために DIRECT OUTPUT ファイルを使用し、すでに存在するレコードと同じキーを持つレコードを挿入しようとすると、KEY 条件が発生する。

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

KSDS の更新は、プライム・インデックスを使って KSDS 。

KSDS の更新

```
//OPT9#13  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 DIRUPDT: PROC OPTIONS(MAIN);

         DCL DIREC FILE RECORD KEYED ENV(VSAM),
             ONCODE BUILTIN,
             OUTREC CHAR(23),
             NUMBER CHAR(3) DEF OUTREC POS(21),
             NAME CHAR(20) DEF OUTREC,
             CODE CHAR(1),
             EOF BIT(1) INIT('0'B);

         ON ENDFILE(SYSIN) EOF='1'B;

         ON KEY(DIREC) BEGIN;
          IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                            ('NOT FOUND: ',NAME)(A(15),A);
          IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                           ('DUPLICATE: ',NAME)(A(15),A);
          END;

         OPEN FILE(DIREC) DIRECT UPDATE;

        GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
          (COLUMN(1),A(20),A(3),A(1));
        DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
        (A(1),A(20),A(1),A(3),A(1),A(1));
        SELECT (CODE);
           WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
           WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
           WHEN('D') DELETE FILE(DIREC) KEY(NAME);
           OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
              ('INVALID CODE: ',NAME) (A(15),A);
        END;
        GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
          (COLUMN(1),A(20),A(3),A(1));
        END;
```

```
         CLOSE FILE(DIREC);
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(DIREC) SEQUENTIAL INPUT;

         EOF='0'B;
         ON ENDFILE(DIREC) EOF='1'B;

         READ FILE(DIREC) INTO(OUTREC);
         DO WHILE(¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
         READ FILE(DIREC) INTO(OUTREC);
         END;
         CLOSE FILE(DIREC);
   END DIRUPDT;
```

```
/*
//GO.DIREC DD DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.         516C
GOODFELLOW,D.T.     889A
MILES,R.               D
HARVEY,C.D.W.       209A
BARTLETT,S.G.       183A
CORY,G.                D
READ,K.M.           001A
PITT,W.H.
ROLF,D.F.              D
ELLIOTT,D.          291C
HASTINGS,G.M.          D
BRAMLEY,O.H.        439C
/*
```

Updating a KSDS (continued)

DIRECT 更新ファイルが使用され、SYSIN ファイルのレコードに渡されるコードに従ってデータが変更される：

A
新しいレコードの追加

C
既存名の番号の変更

D
レコードの削除

ラベル NEXT、名前、番号、コードが読み込まれ、コードの値に応じてアクションが取られる。 KEY ON-unit は、不正確なキーを処理するために使用される。 更新が終了すると（ラベル PRINT で）、ファイル DIREC は閉じられ、属性 SEQUENTIAL INPUT で再び開かれる。 次に、このファイルは順次読み取られ、印刷されます。

このファイルは、「キー配列データセット（KSDS）の定義とロード」（Access Method Services DEFINE CLUSTER ）コマンドで定義された DSNAME PLIVSAM.AJC2.BASE を使用する DD ステートメントによってデータセットに関連付けられます。

を更新する KSDS

KSDS を更新する方法はいくつかある。 大量順次挿入には、 KEYED SEQUENTIAL UPDATE ファイルを使用する。

この方法では、データは、各 WRITE ステートメントの後ではなく本当に必要な場合にのみデータ・セットに書き込まれるため、またデータ・セット内のフリー・スペースの平衡が保たれるため、パフォーマンスが向上します。

以下のステートメントを使用すれば、効率的な大量順次挿入を行うことができます。

```
DCL DIREC KEYED SEQUENTIAL UPDATE
    ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
 KEYFROM(NAME);
```

PL/I 入出力ルーチンは、キーが順番に並んでいることを検出し、 VSAM に正しい要求を行う。 キーが順々になっていない場合には、そのことが検出され、パフォーマンスの利点は失われますがエラーにはなりません。

KSDS の更新で示した例では、DIRECT ファイルを使って KSDS を更新している。 この方法は、この例に示されているようにデータに適しています。

KSDSs 、またはインデックス付きの代替インデックス ESDSs

代替インデックスを使用すると、一意または非一意キーを使用して、 KSDSs またはインデックス付き ESDSs にさまざまな方法でアクセスできます。

の一意キー代替インデックスパスの作成 ESDS

このトピックでは、ESDS の固有キー代替索引パスを作成する方法を説明する例を示します。

292 ページの『#unique_507/unique_507_Connect_42_f97a』 では、285 ページの『#unique_500/unique_500_Connect_42_f97b』 に定義されロードされた ESDS の固有キー代替索引パスを作成する方法を説明しています。 このパスを使用すると、レコードの最初の 15 バイトの子の名前によって、データ・セットへの索引 付けが行われます。

以下のアクセス方式サービス・コマンドが使用されます。

DEFINE ALTERNATEINDEX
代替索引をデータ・セットとして VSAM に定義します。

BLDINDEX
関連レコードへのポインターを代替索引に入れます。

DEFINE PATH
PL/I ファイルと関連付けるエンティティーを、DD ステートメントで定義します。

DD ステートメントは、 の と オペランド、およびソートファイルに必要である。 BLDINDEX INFILE OUTFILE 各所で正しい名前が指定されていることを確認してください。

ESDS の固有キー代替索引パスを作成

```
//OPT9#9    JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC1.ALPHIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 1) -
     KEYS(15 0) -
     RECORDSIZE(20 40) -
     UNIQUEKEY -
    RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC1.ALPHIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC1.ALPHPATH) -
     PATHENTRY(PLIVSAM.AJC1.ALPHIND))
//
```

ESDS の非固有キー代替索引パスを作成

このトピックでは、ESDS の非固有キー代替索引パスを作成する方法を説明する例を示します。

ESDS の一意でないキー代替インデックス・パスの作成 ESDS の一意でないキー代替インデックス・パスの作成を示します。 代替索引を使用すれば、子の性によってデータを選択できます。 これにより少女または少年のデータに別々にアクセスでき、 キーを使用して、それぞれのグループの各メンバーにアクセスできます。

以下のアクセス方式サービス・コマンドが使用されます。

DEFINE ALTERNATEINDEX
代替索引をデータ・セットとして VSAM に定義します。

BLDINDEX
関連レコードへのポインターを代替索引に入れます。

DEFINE PATH
PL/I ファイルと関連付けるエンティティーを、DD ステートメントで定義します。

DD 文は、 BLDINDEX の INFILE と OUTFILE のオペランドと、ソートファイルに必要である。 各所で正しい名前が指定されていることを確認してください。

この例では、 NONUNIQUEKEY オペランドは、インデックスが一意でないキーを持つことを指定しています。 一意でないキーを持つインデックスを作成する場合、指定する RECORDSIZE が十分な大きさであることを確認してください。 非固有代替索引では、それぞれの代替索引レコードが、関連した索引キーを持つすべての レコードへのポインターを含みます。 ポインターは、ESDS の場合は RBA の形をとり、 KSDS の場合はプライムキーの形をとる。 多くのレコードが同じキーを持つ場合は、 大きなレコードが必要です。

ESDS の非固有キー代替索引パスを作成

```
//OPT9#10   JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   /* care must be taken with recordsize */
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC1.SEXIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 1) -
     KEYS(1 37) -
     RECORDSIZE(20 400) -
     NONUNIQUEKEY -
    RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC1.SEXIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC1.SEXPATH) -
     PATHENTRY(PLIVSAM.AJC1.SEXIND))
//
```

の一意キー代替インデックスパスの作成 KSDS

このトピックでは、 KSDS の一意キー代替インデックスパスを作成する方法を例示します。

294 ページの『#unique_509/unique_509_Connect_42_f97d』 は、 に対するユニークキー・オルタネートインデックスパスの作成を示している。 KSDS 電話番号でデータ・セットに索引が付けられます。 これにより、電話番号をキーとして使用して、電話番号の持ち主の名前を検索できるようになります。 また、どの番号が使用されていないかを示すためにデータ・セットを番号順にリストすることができます。

この例では、 UNIQUEKEY オペランドで、キーが一意であることを指定している。

以下の Access Method Services ：

DEFINE ALTERNATEINDEX
代替索引データを保持するデータ・セットを定義します。

BLDINDEX
関連レコードへのポインターを代替索引に入れます。

DEFINE PATH
PL/I ファイルと関連付けるエンティティーを、DD ステートメントで定義します。

DD 文は、 BLDINDEX の INFILE と OUTFILE 、およびソート・ファイルに必要である。 組み込まれる名前を混同することがないようにしてください。

KSDS の固有キー代替索引パスを作成

```
//OPT9#14   JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC2.NUMIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 4) -
     KEYS(3 20) -
     RECORDSIZE(24 48) -
     UNIQUEKEY -
    RELATE(PLIVSAM.AJC2.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC2.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC2.NUMIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC2.NUMPATH) -
     PATHENTRY(PLIVSAM.AJC2.NUMIND))
//
```

固有キーを使用して代替索引を作成する場合は、同じ代替キーでさらにレコードが組み込まれることがないようにしてください。 実際、固有キー代替索引は、電話帳として使用するには、完全に満足のいくものとはいえません。 なぜなら、固有キー代替索引では、2 人の人間が同じ番号を持つことができないためです。 同様に、基本キーでは 1 人の人間が 2 つの電話番号を持てません。 解決策としては、 ESDS 、2 つの一意でないキーの代替インデックスを持つか、データフォーマットを再構築して 1 人につき 1 つ以上の番号を許可し、番号に対して一意でないキーの代替インデックスを持つことである。

一意でない代替インデックス・キーの検出

VSAM 、代替インデックスパスを通してデータセットにアクセスしている場合、SAMEKEY 組み込み関数によって一意でないキーの存在を検出することができます。

各検索の後、SAMEKEY 、今検索したレコードと同じ代替インデックス・キーを持つレコードがさらに存在するかどうかを示す。 したがって、最後のレコードから先を読むことなく、一意でないキーを持つ一連のレコードの最後のレコードで停止することが可能である。 SAMEKEY (file-reference)は、入出力ステートメントが正常に完了し、アクセスされたレコードに同じキーを持つ別のレコードが続く場合、 '1'B を返し、そうでない場合は '0'B を返す。

で代替インデックスを使用する。 ESDSs

295 ページの『#unique_512/unique_512_Connect_42_f97e』 は、ESDS での代替索引の使用と、逆方向読み取りについて説明します。 このプログラムは以下の 4 つのファイルを使用します。

BASEFLE
基本データ・セットを順方向に読み取る。

BACKFLE
基本データ・セットを逆方向に読み取る。

ALPHFLE
名前によって子の索引付けを行う英字代替索引パス。

SEXFILE
子の性別に対応する代替索引パス。

すべてのファイルに DD ステートメントがあります。 DSNAME パラメーターに基本データ・セット名を 指定して、BASEFLE と BACKFLE を基本データ・セットに接続し、292 ページの『#unique_507/unique_507_Connect_42_f97a』 と 293 ページの『#unique_508/unique_508_Connect_42_f97c』 で示したパス名を指定して ALPHFLE と SEXFILE を接続します。

プログラムではデータにアクセスするために SEQUENTIAL ファイルを使用し、最初はそれを通常の順で書き込みし、それから逆順で書き込みします。 AGEQUERY ラベルでは、固有代替索引の 代替索引キーに関連付けられたデータを読み取るために、DIRECT ファイルが使用されます。

最後に、SPRINT ラベルでは、非固有キー代替索引パスを使用して、 ファミリー内の女性のリストを書き込むために KEYED SEQUENTIAL ファイルが使用されます。 同じキーを持つすべてのレコードを読み取るために、SAMEKEY 組み込み関数が使用されます。 女性名は もともと入力された順でアクセスされます。これが行われるのは ファイルが順方向または逆方向に読み取られる場合です。 非固有キー・パスの場合は、BKWD オプションはキーの読み取り順にのみ影響します。 同じキーを持つ項目の順番はファイルが順方向に読み取られる場合と同じです。

例の終わりでは、アクセス方式サービスの DELETE コマンドが基本データ・セットを削除するために使用されています。 これが行われると、関連する代替索引とパスも削除されます。

で代替インデックスを使用する。 KSDSs

296 ページの『#unique_512/unique_512_Connect_42_f97f』 では、KSDS を更新するための、固有な代替索引キーを持つパスの使用法と 代替索引の順にそれにアクセスし出力する方法を説明します。

代替インデックス・パスは、パス名（PLIVSAM.AJC2.NUMPATH、 294 ページの『#unique_509/unique_509_Connect_42_f97d』の DEFINE PATH コマンドで指定される）を DSNAME として指定する DD ステートメントによって PL/I ファイルに関連付けられる。

プログラムの最初のセクションでは、 DIRECT OUTPUT ファイルを使用して、代替インデックス・キーを使用して新しいレコードを挿入する。 代替索引を用いて変更を加える場合は、既存のレコードへのアクセス用の 基本キーや代替索引キーを変更しないでください。 また、変更の際に、基本索引や固有キー代替索引に、重複するキーを追加しないでください。

プログラムの 2 番目のセクションでは (PRINTIT ラベルで)、データ・セットが SEQUENTIAL INPUT ファイルを使用して代替索引キーの順で読み取られます。 その後、SYSPRINT に書き込まれます。

ESDS での代替索引パスと逆方向読み取り

```
//OPT9#15  JOB
//STEP1  EXEC IBMZCLG
//PLI.SYSIN      DD *
  READIT: PROC OPTIONS(MAIN);
    DCL   BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
                 /*File to read base data set forward */
          BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
                 /*File to read base data set backward */
          ALPHFLE FILE DIRECT INPUT ENV(VSAM),
          /*File to access via unique alternate index path */
          SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
           /*File to access via nonunique alternate index path */
          STRING CHAR(80),  /*String to be read into */
          1 STRUC DEF (STRING),
            2 NAME CHAR(25),
            2 DATE_OF_BIRTH CHAR(2),
            2 FILL CHAR(10),
            2 SEX CHAR(1);
            DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;
            DCL EOF BIT(1) INIT('0'B);

       /*Print out the family eldest first*/

       ON ENDFILE(BASEFLE) EOF='1'B;
       PUT EDIT('FAMILY ELDEST FIRST')(A);
       READ FILE(BASEFLE) INTO (STRING);
       DO WHILE(¬EOF);
          PUT SKIP EDIT(STRING)(A);
          READ FILE(BASEFLE) INTO (STRING);
       END;
       CLOSE FILE(BASEFLE);
       PUT SKIP(2);
        /*Close before using data set from other file not
          necessary but good practice to prevent potential
          problems*/

       EOF='0'B;
       ON ENDFILE(BACKFLE) EOF='1'B;
       PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);
```

```
          READ FILE(BACKFLE) INTO(STRING);
          DO WHILE(¬EOF);
            PUT SKIP EDIT(STRING)(A);
            READ FILE(BACKFLE) INTO (STRING);
          END;

          CLOSE FILE(BACKFLE);
          PUT SKIP(2);

          /*Print date of birth of child specified in the file
            SYSIN*/
          ON KEY(ALPHFLE) BEGIN;
             PUT SKIP EDIT
                (NAMEHOLD,' NOT A MEMBER OF THE SMITH FAMILY') (A);
             GO TO SPRINT;
          END;
```

```
       AGEQUERY:
          EOF='0'B;
          ON ENDFILE(SYSIN) EOF='1'B;
          GET SKIP EDIT(NAMEHOLD)(A(25));
          DO WHILE(¬EOF);
                READ FILE(ALPHFLE) INTO (STRING) KEY(NAMEHOLD);
                PUT SKIP (2) EDIT(NAMEHOLD,' WAS BORN IN ',
                    DATE_OF_BIRTH)(A,X(1),A,X(1),A);
                GET SKIP EDIT(NAMEHOLD)(A(25));
          END;
       SPRINT:
          CLOSE FILE(ALPHFLE);
          PUT SKIP(1);

         /*Use the alternate index to print out all the females in the
           family*/
             ON ENDFILE(SEXFILE) GOTO FINITO;
             PUT SKIP(2) EDIT('ALL THE FEMALES')(A);
             READ FILE(SEXFILE) INTO (STRING) KEY('F');
             PUT SKIP EDIT(STRING)(A);
             DO WHILE(SAMEKEY(SEXFILE));
                READ FILE(SEXFILE) INTO (STRING);
                PUT SKIP EDIT(STRING)(A);
             END;

       FINITO:
           END;
      /*
      //GO.BASEFLE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
      //GO.BACKFLE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
      //GO.ALPHFLE   DD   DSN=PLIVSAM.AJC1.ALPHPATH,DISP=SHR
      //GO.SEXFILE   DD   DSN=PLIVSAM.AJC1.SEXPATH,DISP=SHR
      //GO.SYSIN     DD    *
      ANDY
      /*
      //STEP2    EXEC PGM=IDCAMS,REGION=512K
      //SYSPRINT    DD    SYSOUT=A
      //SYSIN       DD     *
           DELETE -
                PLIVSAM.AJC1.BASE
      //
```

Alternate Index Paths and Backward Reading with an ESDS (continued)

KSDS アクセス用の固有代替索引パスの使用

```
 //OPT9#16   JOB
 //STEP1     EXEC IBMZCLG,REGION.GO=256K
 //PLI.SYSIN DD   *
    ALTER: PROC OPTIONS(MAIN);
      DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
          NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
          IN FILE RECORD,
          STRING CHAR(80),
          NAME CHAR(20) DEF STRING,
          NUMBER CHAR(3) DEF STRING POS(21),
          DATA CHAR(23) DEF STRING,
          EOF BIT(1) INIT('0'B);

      ON KEY (NUMFLE1) BEGIN;
        PUT SKIP EDIT('DUPLICATE NUMBER')(A);
      END;
```

```
     ON ENDFILE(IN) EOF='1'B;

     READ FILE(IN) INTO (STRING);
     DO WHILE(¬EOF);
       PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
       WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
       READ FILE(IN) INTO (STRING);
     END;

     CLOSE FILE(NUMFLE1);

     EOF='0'B;
     ON ENDFILE(NUMFLE2) EOF='1'B;

     READ FILE(NUMFLE2) INTO (STRING);
     DO WHILE(¬EOF);
       PUT SKIP EDIT(DATA)(A);
       READ FILE(NUMFLE2) INTO (STRING);
     END;

     PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
  END;
/*
//GO.IN    DD   *
RIERA L            123
/*
//NUMFLE1    DD   DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//NUMFLE2    DD   DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//STEP2    EXEC PGM=IDCAMS,COND=EVEN
//SYSPRINT   DD  SYSOUT=A
//SYSIN    DD    *
     DELETE -
     PLIVSAM.AJC2.BASE
//
```

相対レコード・データ・セット

このトピックでは、VSAM 相対レコード・データ・セット (RRDS) に関連するファイルで使用できるステートメントとオプションについて説明します。

表 *35. VSAM* 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)

表 *35. VSAM* 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション *(*続き*)*

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	DELETE FILE(file-reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

表 *35. VSAM* 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション *(*続き*)*

File declaration[1]	Valid statements with options you must include	Other options you can also include

Notes :

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, your declaration must also include the attribute KEYED.

   The UNLOCK statement for DIRECT UPDATE files is ignored if you use it for files associated with a VSAM RRDS.
2. The statement `READ FILE(file-reference);` is equivalent to the statement `READ FILE(file-reference) IGNORE(1);`.

ローディング RRDS

RRDS を読み込む場合は、関連するファイルを開いて OUTPUT を参照する必要がある。 DIRECT 、または SEQUENTIAL 。

DIRECT OUTPUT ファイルの場合、各レコードは WRITE 文の KEYFROM オプションで相対レコード番号(またはキー）で指定された位置に配置される（ 276 ページの『VSAM データセットのキー』 を参照)。

SEQUENTIAL OUTPUT ファイルの場合は、KEYFROM オプションの有無にかかわらず、WRITE ステートメントを使用する。 KEYFROM オプションを指定すると、 レコードは指定されたスロット内に置かれ、 省略した場合は、レコードは 現在位置に続くスロット内に 置かれます。 レコードを昇順の相対レコード番号順に並べる必要はありません。 KEYFROM オプションを省略しても、KEYTO オプションを使用することにより、書き込まれたレコードの相対レコード番号を取得することができます。

KEYFROM または KEYTO オプションを使用せずに RRDS を順次ロードする場合、ファイルに KEYED 属性は必要ありません。

既にレコードが存在する位置にレコードをロードしようとすることは誤りです。 KEYFROM オプションを使用する場合は、KEY 条件が生じ、 それを省略する場合は ERROR 条件が発生します。

相対レコードデータセット(RRDS)の定義と読み込みでは、データセットは DEFINE CLUSTER コマンドで定義され、 PLIVSAM.AJC3.BASE という名前が与えられます。 それが RRDS であるという事実は、NUMBERED キーワードによって判断されます。 PL/I 、DIRECT OUTPUT ファイルがロードされ、WRITE...FROM...KEYFROM 。

データが順に並んでいてキーが順序どおりになっていれば、SEQUENTIAL ファイル を使用して、先頭からデータ・セットに書き込むことができたはずです。 この後レコードは、次に使用できるスロットに入れられ、該当する番号が付けられます。 各レコード用のキーの番号が、KEYTO オプションを使って戻されたはずです。

PL/I ファイルは、DEFINE CLUSTER コマンドで 指定された名前を DSNAME として使用する DD ステートメントによってデータ・セットへ関連付けられます。

相対レコード・データ・セット (RRDS) の定義とロード

```
//OPT9#17  JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD *
         DEFINE CLUSTER -
             (NAME(PLIVSAM.AJC3.BASE) -
             VOLUMES(nnnnnn) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20))
/*
//STEP2  EXEC IBMZCBG
//PLI.SYSIN      DD *
 CRR1:   PROC OPTIONS(MAIN);
         DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
             CARD CHAR(80),
```

```
             NAME CHAR(20) DEF CARD,
             NUMBER CHAR(2) DEF CARD POS(21),
             IOFIELD CHAR(20),
             EOF BIT(1) INIT('0'B);
          ON ENDFILE (SYSIN) EOF='1'B;
          OPEN FILE(NOS);
          GET FILE(SYSIN) EDIT(CARD)(A(80));
         DO WHILE (¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
          IOFIELD=NAME;
          WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
          GET FILE(SYSIN) EDIT(CARD)(A(80));
          END;
          CLOSE FILE(NOS);
 END CRR1;
/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.           12
BAKER,R.            13
BRAMLEY,O.H.        28
CHEESNAME,L.        11
CORY,G.             36
ELLIOTT,D.          85
FIGGINS.E.S.        43
HARVEY,C.D.W.       25
HASTINGS,G.M.       31
KENDALL,J.G.        24
LANCASTER,W.R.      64
MILES,R.            23
NEWMAN,M.W.         40
PITT,W.H.           55
ROLF,D.E.           14
SHEERS,C.D.         21
SURCLIFFE,M.        42
TAYLOR,G.C.         47
WILTON,L.W.         44
WINSTONE,E.M.       37
//
```

SEQUENTIAL。 RRDS

INPUT 、または UPDATE 属性を持つ RRDS へのアクセスに使用される SEQUENTIAL ファイルを開くことができます。 KEY、KEYTO、KEYFROM オプションのいずれかを使用する場合は、 属性 KEYED も指定する必要があります。

KEY オプションが指定されていない READ ステートメントの場合、レコードは、昇順の相対 レコード番号順に回復されます。 データ・セット内に空きスロットがあれば、すべてスキップされます。

KEY オプションを指定すると、READ ステートメントで回復されるレコードは、 指定した相対レコード番号を持つレコードになります。 このような READ ステートメントはデータ・セットを指定されたレコードに 位置付け、その後の順次読み取りは後続のレコードを順番に回復します。

KEYFROM オプションが指定されていても いなくても、WRITE ステートメントは、KEYED SEQUENTIAL UPDATE ファイル に使用することができます。 前回行ったアクセスの位置に関係なく、データ・セット内の任意の位置に 挿入を行うことができます。 KEYFROM オプションが指定された WRITE ステートメントの場合、 そのデータ・セットに既に存在するレコードと同じ相対レコード番号 を持つレコードを挿入しようとすると、KEY 条件が発生します。 KEYFROM オプションを省略すると、 現在位置から見て次のスロットにレコードが書き込まれます。 このスロットが空いていなければ、ERROR 条件が発生します。

KEYTO オプションを使えば、KEYFROM オプションを 指定していない WRITE ステートメントにより追加されるレコードのキーを回復することができます。

REWRITE ステートメントは、KEY オプションのあるなしに 関係なく、UPDATE ファイルで使用できます。 KEY オプションを使用した場合、 再書き込みされるレコードは指定された相対レコード番号を持つ レコードであり、そうでない場合は直前の READ ステートメントによってアクセスされた レコードです。

KEY オプションの有無に関係なく DELETE ステートメントを使用すれば、データ・セットからレコードを削除できます。

RRDS にアクセスするために DIRECT

RRDS にアクセスするために使われる DIRECT ファイルは、 OUTPUT, INPUT、または UPDATE 属性を持つことができる。 KEYED SEQUENTIAL ファイルを使用するのとまったく同じように、レコードの読み取り、書き込み、書き換え、削除ができる。

301 ページの『#unique_515/unique_515_Connect_42_f918a』 は。 RRDS DIRECT UPDATE ファイルが使われ、新しいレコードはキーごとに書き込まれる。 空のレコードは VSAM で利用できないので、レコードが空かどうかをチェックする必要はない。

プログラムの後半では、ラベル PRINT から、更新されたファイルがプリントアウトされる。 REGIONAL(1) のように空のレコードをチェックする必要はない。

PL/I ファイルは、 DSNAME PLIVSAM.AJC3.BASE を指定する DD ステートメントによってデータセットに関連付けられ、 DEFINE CLUSTER コマンドで指定された名前は、 301 ページの『#unique_515/unique_515_Connect_42_f918a』 で指定される。

例の最後では、 DELETE コマンドを使ってデータセットを削除している。

RRDS の更新

```
//* NOTE: WITH A WRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A DUPLICATE MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO CORRESPONDS TO AN EXISTING NUMBER IN THE LIST,
//*       FOR EXAMPLE, ELLIOT.
//*       WITH A REWRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A NOT FOUND MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO DOES NOT CORRESPOND TO AN EXISTING NUMBER IN
//*       THE LIST, FOR EXAMPLE, NEWMAN AND BRAMLEY.
//OPT9#18  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 ACR1:  PROC OPTIONS(MAIN);
                DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
                  (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
                  BYTE CHAR(1) DEF IOFIELD, EOF BIT(1) INIT('0'B),
                  ONCODE BUILTIN;
        ON ENDFILE(SYSIN) EOF='1'B;
        OPEN FILE(NOS) DIRECT UPDATE;
        ON KEY(NOS) BEGIN;
           IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                    ('NOT FOUND:',NAME)(A(15),A);
           IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                    ('DUPLICATE:',NAME)(A(15),A);
        END;
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
           (A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
        SELECT(CODE);
           WHEN('A') WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           WHEN('C') DO;
            DELETE FILE(NOS) KEY(OLDNO);
            WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           END;
           WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
           OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME)(A(15),A);
        END;
```

```
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        END;
         CLOSE FILE(NOS);
            PRINT:
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(NOS) SEQUENTIAL INPUT;
         EOF='0'B;
         ON ENDFILE(NOS) EOF='1'B;
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          DO WHILE (¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD)(A(5),A);
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          END;
       CLOSE FILE(NOS);
```

```
 END ACR1;

/*
//GO.NOS     DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN   DD  *
NEWMAN,M.W.          5640C
GOODFELLOW,D.T.      89  A
MILES,R.               23D
HARVEY,C.D.W.        29  A
BARTLETT,S.G.        13  A
CORY,G.                36D
READ,K.M.            01  A
PITT,W.H.              55
ROLF,D.F.              14D
ELLIOTT,D.           4285C
HASTINGS,G.M.          31D
BRAMLEY,O.H.         4928C
//STEP3    EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN    DD   *
      DELETE -
          PLIVSAM.AJC3.BASE
//
```

Updating an RRDS (continued)

VSAM 以外のデータセット用に定義されたファイルの使用

共用データ・セットの使用

PL/I では、データ・セットの領域間またはシステム間での共用が許されます。 この種の共有のサポートは VSAM によって提供される。

サポートの詳細については、以下の DFSMS マニュアルを参照してください：

- *DFSMS:* データ・セットの使用法
- *DFSMS:* カタログのためのアクセス方式サービス・プログラム

第 3 部 プログラムの改良

© Copyright IBM Corp. 1999, 2025

第 15 章 パフォーマンスの向上

ユーザーのプログラムの速度を向上することに関する多数の考慮事項は、 使用するコンパイラーとそれを実行するプラットフォームには関係ありません。 しかし、この章では、考慮事項の中でも PL/I コンパイラーとそれによって生成される コードに特有な考慮事項を識別して説明します。

最適なパフォーマンスのためのコンパイラー・オプションの選択

選択するコンパイラー・オプションに応じて、コンパイラーによって生成される コードのパフォーマンスを大幅に向上できることがあります。 ただし、ほとんどのパフォーマンスの考慮事項と同様に、 オプションの選択にはトレードオフがあります。

幸いなことに、コンパイラー・オプションはコマンド行または 構成ファイルで指定できるため、ソース・コードを編集しなくても、 コンパイル時オプションに伴うトレードオフについて比較検討することができます。

詳細を省きたい場合は、生成されるコードのパフォーマンスを向上させる最も簡単な方法として、次の (非デフォルトの) コンパイラー・オプションを指定する方法があります。

```
OPT(2) or OPT(3)
DFT(REORDER)
```

次のトピックでは、パフォーマンスの向上と、特定のコンパイラー・オプションに伴うトレードオフについて、より詳しく説明します。

OPTIMIZE

OPTIMIZE オプションを指定すると、プログラムの速度を上げることができます。 このオプションを指定しないと、コンパイラーは基本的な最適化のみを実行します。

OPTIMIZE(2) を選択すると、コンパイラーは、パフォーマンスを改善するコードを生成するように指示されます。 通常、結果として生成されるコードは、プログラムが NOOPTIMIZE を 使ってコンパイルされるときより短くなります。 しかし、場合によっては、長い命令シーケンスが、 短い命令シーケンスよりも実行時間が短いこともあります。 例えば、WHEN 文節の値にギャップが指定されている SELECT ステートメント 用にブランチ・テーブルが作成された場合などがこれに該当します。 生成される命令の数が増えた分は、通常、ほかの場所での命令の実行数が減ることによって相殺されます。

GONUMBER

GONUMBER オプションを指定すれば、デバッグに使用されるステートメント番号表を生成できます。

この追加情報は、デバッグ時に非常に役立つことがありますが、ステートメント番号表を含めると、実行可能ファイルのサイズが大きくなります。 実行可能ファイルが大きくなると、ロードに時間がかかります。

ARCH

ARCH オプションで最高値を使用すれば、z/OS において使用できる最も幅広い命令セットから命令を選択するようにコンパイラーに指示することができます。 これにより、コンパイラーは最適なコードを生成できます。

REDUCE

REDUCE オプションは、コンパイラーが、構造体に対する ヌル・ストリングの割り当てを縮小して、 単純なコピー操作にすることを認められることを指定します。 その操作が、埋め込みバイトの上書きを意味するにすぎない場合もあります。

REDUCE オプションが指定されると、ヌル・ストリングを構造体に割り当てるために生成されるコードの行の数が少なくなり、その結果として通常はコンパイルが高速になり、コードの実行速度が大きく向上します。 しかし、埋め込みバイトはゼロにリセットされることがあります。

© Copyright IBM Corp. 1999, 2025

例えば次の構造体では、field11 と field12 の間に 1 バイトの埋め込み があります。

```
dcl
 1 sample ext,
   5  field10          bin fixed(31),
   5  field11          dec fixed(13),
   5  field12          bin fixed(31),
   5  field13          bin fixed(31),
   5  field14          bit(32),
   5  field15          bin fixed(31),
   5  field16          bit(32),
   5  field17          bin fixed(31);
```

次に、割り当て sample = '';について考えてみます。

NOREDUCE オプションでは、8 つの割り当てが生成されますが、埋め込みバイトは変更されません。

ただし、REDUCE では、割り当ては 3 つの演算に削減されます。

RULES

RULES サブオプションのほとんどは、特定のコーディング方法 (変数を宣言 しないことなど) にフラグを立てる基準となる重大度だけに影響し、 パフォーマンスには影響しません。 ただし、次のサブオプションはパフォーマンスに影響します。

IBM/ANS

RULES(IBM) オプションを使うと、コンパイラーはスケールされた FIXED BINARY をサポートします。パフォーマンスについてより重大なことは、いくつかの操作により、コンパイラーは、スケールされた FIXED BINARY の結果を生成します。

RULES(ANS) を指定すると、スケールされた FIXED BINARY はサポートされず、 スケールされた FIXED BINARY の結果は生成されません。 つまり、RULES(ANS) を指定して生成されたコードは、常に RULES(IBM) を指定して生成されたコードと少なくとも同じ速さで実行され、場合によっては実行速度が速くなります。

例えば、次のような部分コードがあるとします。

```
dcl (i,j,k) fixed bin(15);
   .
   .
   .
i = j / k;
```

RULES(IBM) を指定した場合は、割り算の結果に属性 FIXED BIN(31,16) が含まれます。 つまり、割り算の前にはシフト命令が必要で、 割り当てを実行するためにさらにいくつかの命令が必要になります。

RULES(ANS) のもとでは、割り算の結果は属性 FIXED BIN(15,0) を持ちます。 つまり、割り算の前にシフトは必要なく、割り当てを実行するために追加の命令は必要ありません。

(NO)GLOBALDO

NOGLOBALDO を指定すると、親ブロックで宣言された制御変数を持つすべての DO ループにフラグを立てるようにコンパイラに指示する。 これは、パフォーマンスにとって非常に悪い、推奨されないプログラミング手法のインスタンスをキャッチするための良い方法です。

例えば、コードを介して配列 B の正のエレメントを配列 A に割り当てるコードがあるとします。

```
        do jx = 1 to hbound(a);
          if b(jx) <= 0 then;
          else
            a(jx) = b(jx);
        end;
```

JX は、このループを含む同じ PROCEDURE 。

RULES(NOGLOBALDO) を指定すると、これが真でないコードにフラグを立てる。

(NO)LAXCTL

RULES(LAXCTL) のもとでは、CONTROLLED 変数を固定エクステントによって宣言しても、異なるエクステントに割り振ることができます。 ただし、このコーディング方法はパフォーマンスに重大な影響を及ぼします。 RULES(NOLAXCTL) オプションを使用して、そのような方法を許可しないようにすることをお勧めします。

例えば、RULES(LAXCTL) を指定すると、次のような構造体を宣言できます。

```
dcl
  1 a controlled,
    2 b char(17),
    2 c char(29);
```

しかし、その後でこの構造体を次のように割り振ることができます。

```
allocate
  1 a,
    2 b char(170),
    2 c char(290);
```

コンパイラーは構造体 A に対する参照や、その構造体のメンバーに対する参照を見つけると必ず、構造体におけるフィールドの長さ、次元、またはオフセットに関して何の情報もないと強制的に想定します。 このため、パフォーマンスが著しく低下する可能性があります。

RULES(NOLAXCTL) では、可変エクステントを使用して CONTROLLED 変数を割り振る場合は、そのエクステントを、アスタリスクまたは非定数式を使用して宣言する必要があります。 したがって、RULES(NOLAXCTL) を指定し、固定エクステントを使用して CONTROLLED 変数を宣言すれば、コンパイラーはその変数を参照するためのコードをはるかに適切に生成できます。

PREFIX

PREFIX オプションは、選択された PL/I 条件がデフォルトで使用可能になっているかどうかを判別します。 PREFIX のデフォルトのサブオプションは、PL/I 言語定義に適合するように 設定されますが、デフォルトを指定変更すると、ユーザーのプログラムの パフォーマンスに大きい影響を与えることがあります。

デフォルトのサブオプションを以下に示します。

MANCE がありません
CONVERSION
INVALIDOP
FIXEDOVERFLOW
OVERFLOW
INVALIDOP
NOSIZE
NOSTRINGRANGE
NOSTRINGSIZE
NOSUBSCRIPTRANGE
UNDERFLOW
ZERODIVIDE

SIZE、STRINGRANGE、STRINGSIZE、または SUBSCRIPTRANGE サブオプションを指定すると、コンパイラーによってエクストラ・コードが生成されます。このコードは、他の方法では見つけるのが難しいソース内のいろいろな問題領域の位置を正確に示すのに役立ちます。 ただし、この追加のコードにより、プログラムのパフォーマンスが大幅に低下することがあります。

CONVERSION

CONVERSION 条件を使用不可にすると、いくつかの文字から数値への変換がインラインで、ソースの妥当性検査が行われずに実行されます。 したがって、NOCONVERSION を指定した場合も、プログラムのパフォーマンスに影響があります。

FIXEDOVERFLOW

プラットフォームによっては、ハードウェアによって FIXEDOVERFLOW 条件が生じるため、コンパイラーはその検出に追加のコードを生成しなくてもよい場合があります。

デフォルト

DEFAULT オプションを使うと、属性のデフォルトを選択できます。 PREFIX オプションの場合と 同様に、DEFAULT のサブオプションは PL/I 言語定義に従うように 設定されます。 デフォルトを変更すると、パフォーマンスに影響が及ぶ場合があります。

IBM/ANS および ASSIGNABLE/NONASSIGNABLE など、サブオプションのいくつかはプログラムのパフォーマンスに影響しません。 ただし、ほかのすべてのサブオプションは、多かれ少なかれパフォーマンス に影響を与えることがあり、不適切に適用された場合は、プログラムが無効 になることもあります。 以下のトピックでは、より重要ないくつかのサブオプションについて詳しく説明します。

BYADDR または BYVALUE

DEFAULT(BYADDR) オプションが有効な場合は、 入り口宣言の属性でほかのものが指定されていない限り、参照によって (PL/I の必要性に応じて) 引数が渡されます。 参照によって引数が渡されると、変数そのものが渡されるときに、 引数のアドレスが、あるルーチン (呼び出しルーチン) から別 のルーチン (呼び出されたルーチン) に渡されます。 呼び出されたルーチン側に制御があるときに引数が変更されると、 呼び出しルーチンの実行が再開されたときに呼び出しルーチンにそれが反映されます。

多くの場合、プログラム・ロジックは、参照によって渡される変数に応じて異なります。 しかし、参照によって変数を渡すと、 次の 2 つのようなパフォーマンスの低下が起きる場合があります。

1. そのパラメーターに対するすべての参照に、追加の命令が必要になる。
2. 変数のアドレスが別のルーチンに渡されると、コンパイラーは、その変数が変更されるときを想定して、その変数の参照用のコードとして非常に保守的なコードを生成することを強制される。

したがって、プログラム・ロジック上で認められるときは いつでも、BYVALUE サブオプションを使い、値によってパラメーターを渡す必要があります。 BYADDR 属性を使って、1 つのパラメーターが参照によって 渡されるように指示するときでも、DEFAULT(BYVALUE) オプションを使って、ほかのすべてのパラメーターが値によって渡されるように設定できます。

BYVALUE 引数は、正当にレジスターで受け渡しできるものである必要がある。 したがって、そのタイプは以下のいずれかでなければなりません。

- REAL FIXED BIN
- REAL FLOAT
- POINTER
- OFFSET
- HANDLE
- LIMITED ENTRY
- ファイル
- ORDINAL
- CHAR(1)
- WCHAR(1)
- ALIGNED BIT(*n*) (*n* は 8 以下)

さらに、BYVALUE パラメーターをアセンブラー・コードと一緒に使用するときは、パラメーター・リストで 4 バイトが使用されるように、4 の倍数バイトを必要としない BYVALUE パラメーターが拡張されることに注意する必要があります。

プロシージャーが BYADDR によって渡される 1 つのパラメーターだけを受け取り、 それを変更する場合は、値によってそのパラメーターを受け取る関数 にプロシージャーを変換することを考慮してください。 その関数は、パラメーターの更新値が含まれた RETURN ステートメントで終了します。

BYADDR パラメーターが指定されたプロシージャー

```
a: proc( parm1, parm2, ..., parmN );

  dcl parm1 byaddr  ...;
  dcl parm2 byvalue ...;
   .
   .
   .
  dcl parmN byvalue ...;

  /* program logic */

end;
```

BYVALUE パラメーターが指定された、より高速の同等な関数

```
a: proc( parm1, parm2, ..., parmN )
   returns( ... /* attributes of parm1 */ );

  dcl parm1 byvalue  ...;
  dcl parm2 byvalue ...;
   .
   .
   .
  dcl parmN byvalue ...;

  /* program logic */

  return( parm1 );

end;
```

(NON)CONNECTED

DEFAULT(NONCONNECTED) オプションは、すべての集合体パラメーターが NONCONNECTED であるとコンパイラーが想定するように指示します。 NONCONNECTED 集合体パラメーターのエレメントに対する参照では、コンパイラーがパラメーターの記述子にアクセスするためのコードを生成する必要があります。 これは、集合体が固定エクステントを使って宣言されている場合にも同様です。

集合体パラメーターに固定エクステントが指定 されており、CONNECTED が指定されている場合、コンパイラーはこれらの命令を生成しません。 したがって、アプリケーションが NONCONNECTED パラメーターを渡さない 場合は、DEFAULT(CONNECTED) オプションを使うとコードがより最適化されます。

(NO)DESCRIPTOR

DEFAULT(DESCRIPTOR) オプションは、デフォルトでは、 ストリング、領域、または集合体パラメーター用に記述子が渡されることを示します。 しかし、記述子は、パラメーターに非固定エクステントが指定されている場合、 またはパラメーターが NONCONNECTED 属性を持つ配列である場合にだけ、使われます。

この場合は、記述子を渡すために必要となる命令とスペースは効果がなく、かなりのコストがかかります (多くの場合、構造化記述子のサイズは構造体そのもののサイズよりも大きくなります)。 したがって、PROCEDURE ステートメントと ENTRY 宣言で 必要なときにだけ DEFAULT(NODESCRIPTOR) を指定し、OPTIONS(DESCRIPTOR) を使うと、 コードの実行がより最適化されます。

(NO)INLINE

NOINLINE サブオプションは、プロシージャーと開始ブロックがインライン化されないように指示します。 インライン化は、最適化を指定するときにだけ発生します。

ユーザー・コードをインライン化すると、関数呼び出しとリンケージのオーバーヘッドが除去され、関数のコードが最適化プログラムに公開されて、結果としてコード・パフォーマンスが向上します。 インライン化は、関数のオーバーヘッドが無視できるようなものではないとき、 例えば、関数がネストされたループ内で呼び出されるときなどに、最高の結果をもたらします。 また、インライン化は、インライン化された関数によってさらに最適化の機会が与えられるとき、 例えば、定数引数が使われるときなどにも、有益です。

ネストされていない多くのプロシージャーが含まれているプログラムに関しては、以下の情報を考慮してください。

- プロシージャーの規模が小さく、ほんのいくつかの場所からしか呼び出されない場合は、INLINE を 指定することによってパフォーマンスを向上させることができます。
- プロシージャーの規模が大きく、複数の場所から呼び出される場合は、 インライン化によって、プログラム全体にわたってコードの重複が起こります。 このようなプログラム・サイズの増大は、速度の増大を相殺します。 この場合は、NOINLINE をデフォルトのままにして、 個別に選択したプロシージャーだけで OPTIONS(INLINE) を指定することをお勧めします。

インライン化を使う場合は、スタック・スペースを拡大する必要があります。 関数が呼び出されると、そのローカル・ストレージが呼び出し時に割り振られ、 呼び出し関数に戻るときに解放されます。 その関数がインライン化されると、そのストレージは、それを呼び出す関数に入ったときに 割り振られ、呼び出し関数が終了するまで解放されません。 インライン化した関数のローカル・ストレージ用に、 十分なスタック・スペースがあることを確認してください。

LINKAGE

この LINKAGE サブオプションは、OPTIONS 属性の LINKAGE サブオプション、または入力のオプションが指定されなかったときにコンパイラーが使用しなければならないデフォルト・リンケージを指定します。 コンパイラーは、それぞれが固有のパフォーマンス特性を 持つ各種のリンケージをサポートします。

外部エンティティー (例えば、オペレーティング・システムなど) によって提供される ENTRY を呼び出すときには、その ENTRY に対してあらかじめ定義されているリンケージを使用する必要があります。

ただし、独自のアプリケーションを作成するときには、リンケージ規則を選択できます。 OPTLINK リンケージは、ほかのリンケージ規則よりも 大幅にパフォーマンスを向上させるので、これを選択することをお勧めします。

(RE)ORDER

DEFAULT(ORDER) オプションは、ORDER オプションが すべてのブロックに適用されることを示します。 つまり、ON ユニットで参照されるそのブロック (または ON ユニットから 動的に派生するブロック) 内の変数に最新の値が適用されることを示します。 これにより、そのような変数でのほとんどすべての最適化が実際上禁止されます。

したがって、プログラム・ロジックによって認められる 場合は、DEFAULT(REORDER) を使って、上位コードを生成してください。

NOOVERLAP

DEFAULT(NOOVERALP) オプションを指定すると、コンパイラーは 割り当て時にソースとターゲットがオーバーラップしていないと想定するため、 より小さく速いコードを生成することができます。

ただし、このオプションを使用する場合は、割り当て時にソースとターゲットがオーバーラップしないようにする必要があります。 例えば、DEFAULT(NOOVERLAP) オプションでは、次の例にある割り当ては無効です。

```
dcl c char(20);
substr(c,2,5) = substr(c,1,5);
```

RETURNS(BYVALUE) または RETURNS(BYADDR)

DEFAULT(RETURNS(BYVALUE)) オプションが有効な場合は、BYADDR が指定されないすべての RETURNS 記述リストに BYVALUE 属性が適用されます。 つまり、これらの関数は、最適なコードを生成するために、可能なときにはレジスターに値を戻します。

パフォーマンスを向上させるコンパイラー・オプションの要約

要約すると、次に挙げるオプション (ユーザーのアプリケーションに適用できる場合) は パフォーマンスを向上させることができます。

- OPTIMIZE(3)
- ARCH(14)
- REDUCE
- RULES(ANS NOLAXCTL NOGLOBALDO)
- DEFAULT with the following suboptions

  BYVALUE
  CONNECTED
  NODESCRIPTOR
  INLINE
  LINKAGE(OPTLINK)
  REORDER
  NOOVERLAP
  RETURNS(BYVALUE)

パフォーマンス向上のためのコーディング

コードを作成するときには、指定されたタスクを実行するために適する方法が複数あるのが普通です。 多くの重要な要素、例えば、読み易さや保守容易性などによって、 コーディングするスタイルの選択は変わってきます。 次のトピックでは、コーディングを行うときにプログラムのパフォーマンスに影響を及ぼす可能性がある選択肢について説明します。

DATA ディレクティブ入出力

デバッグに GET DATA ステートメントと PUT DATA ステートメントを使うと、非常に有効であることがあります。 ただし、これらのステートメントを使うと、一般的にはパフォーマンスが低下という犠牲が伴います。 このパフォーマンスの低下は、変数リストを 使わずに GET DATA または PUT DATA を使うと、非常に大きくなります。

多くのプログラマーは、次の例に示す ように、ON ERROR コード内で PUT DATA ステートメントを使います。

```
on error
  begin;
    on error system;
   .
   .
   .
    put data;
   .
   .
   .
  end;
```

この場合、PUT DATA ステートメントで、 選択された変数のリストを組み込むことにより、プログラムがより最適化されます。

上記の例の ON ERROR ブロックには、PUT DATA ステートメントの 前に ON ERROR システム・ステートメントが含まれています。 これにより、PUT DATA ステートメントでエラーが起きても (このエラーは、リストされる変数に無効な FIXED DECIMAL 値が含まれている場合に 起きる可能性がある)、ON ERROR ブロックのほかの場所でエラーが起きても、 プログラムが無限ループに入ることが回避されます。

入力専用パラメーター

プロシージャーに、入力専用に使われる BYADDR パラメーターが含まれている場合は、そのパラメーターを NONASSIGNABLE として宣言するのが (ASSIGNABLE のデフォルト属性を取得させるのではなく) 最良の方法です。 プロシージャーがあとからそのパラメーターの定数を使って呼び出された場合、 コンパイラーは静的ストレージに定数を入れ、その静的領域のアドレスを渡します。

この方法は、レジスターに渡すことができないストリングや その他のパラメーターに特に役立ちます (レジスターに渡すことができる 入力専用パラメーターは、BYVALUE として宣言するのが最良です)。

例えば次の宣言では、getenv に対する 最初のパラメーターが、入力専用の CHAR VARYINGZ ストリングです。

```
dcl getenv     entry( char(*) varyingz nonasgn byaddr,
                      pointer byaddr )
               returns( native fixed bin(31) optional )
               options( nodescriptor );
```

ストリング IBM_OPTIONS が指定されてこの関数が呼び出されると、コンパイラーは、コンパイラー生成一時記憶域にそのストリングを割り当て、その領域のアドレスを渡すのではなく、ストリングのアドレスを渡すことができます。

GOTO ステートメント

別のブロック内のラベルまたはラベル変数を使う GOTO ステートメントは、 コンパイラーが実行する最適化を厳しく制限します。

ラベル配列が初期化され、暗黙的または明示的 に AUTOMATIC と宣言された場合、その配列のエレメントへの GOTO は最適化の妨げとなります。 しかし、配列が STATIC と宣言された場合は、 コンパイラーがその配列に対して CONSTANT 属性を想定し、最適化は妨げられません。

ストリングの割り当て

あるストリングが別のストリングに割り当てられているとき、コンパイラーは、ソースとターゲットがオーバーラップする場合でもターゲットの値を正しい値とし、ソース・ストリングがターゲットよりも長い場合はソース・ストリングを切り捨てます。 これには、いくつかの追加命令が必要になります。

コンパイラーは、これらの追加の命令を必要なときにだけ生成しようとしますが、 コンパイラーが必要でないということに確信を持てない場合、ユーザーは、 プログラマーとして、その追加の命令が必要ないということを知っていることが多いのです。 例えば、ソースとターゲットが基底付き文字ストリングであって、 ユーザーはそれらがオーバーラップすることがあり得ないことを知っている場合、 コンパイラーであれば生成するように強制される ところを、PLIMOVE 組み込み関数を使ってその追加のコードを除去することができます。

次の例では、2 番目の割り当てステートメント用に、より速いコードが生成されます。

```
dcl based_Str   char(64) based;
dcl target_Addr pointer;
dcl source_Addr pointer;

target_Addr->based_Str = source_Addr->based_Str;

call plimove( target_Addr, source_Addr, stg(based_Str) );
```

ユーザーがソースとターゲットがオーバーラップするのではないかと疑いがある場合、または ターゲットがソースを収容できる大きさであるかどうかが疑われる 場合は、ユーザーは PLIMOVE 組み込み関数を使ってはなりません。

ループ制御変数

ループ制御変数を適切に定義すれば、プログラムのパフォーマンスが向上します。

プログラムのパフォーマンスを向上させるには、ループ制御変数に対して以下のいずれかのタイプを使用します。 ユーザーは、まれに必要になる場合を除き、これ以外のタイプの変数を使うべきではありません。

ゼロのスケール因数を持つ FIXED BINARY
FLOAT
ORDINAL
HANDLE
POINTER
OFFSET

また、ループ制御変数が配列、構造体、 または共用体のメンバーでない場合にも、パフォーマンスは向上します。 ループ制御変数がこのようなメンバーである場合、コンパイラーは警告メッセージを出します。 AUTOMATIC であり、ほかの目的で使われないループ制御変数は、コードの生成を最適化します。

ループ制御変数が FIXED BINARY である場合は、精度が 31 で SIGNED である場合に、 パフォーマンスが最高に達します。

プログラムがループ制御変数の値だけでなく、 そのアドレスにも依存している場合は、パフォーマンスが低下します。 例えば、ADDR 組み込み関数が変数に適用される場合、または変数が参照 (BYADDR) によって別のルーチンに渡される場合は、パフォーマンスが低下します。

PACKAGE 対ネストされた PROCEDURE

呼び出し側のネストされたプロシージャーは、 追加の隠しパラメーター (逆チェーン・ポインター) が渡されるように求めます。 結果として、アプリケーションに含まれている ネストされたプロシージャーが少なくなればなるほど、実行速度は速くなります。

アプリケーションのパフォーマンスを向上させるには、 ネストされたプロシージャーの母娘の対を、 パッケージ内部のレベル 1 の姉妹プロシージャーに変換します。 この変換は、ネストされたプロシージャーが、その親プロシージャーで宣言された自動かつ内部の静的変数に依存していない場合に可能です。

313 ページの『ネストされたプロシージャーの例』にあるプロシージャー b に、a で宣言された変数が使用されていない場合は、両方のプロシージャーを再編成して、313 ページの『パッケージ化されたプロシージャーの例』に示されているパッケージに組み込むことで、両方のプロシージャーのパフォーマンスを向上させることができます。

ネストされたプロシージャーの例

```
a: proc;

  dcl (i,j,k) fixed bin;
  dcl ib      based fixed bin;
   .
   .
   .
  call b( addr(i) );
   .
   .
   .
  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;
end;
```

パッケージ化されたプロシージャーの例

```
p: package exports( a );

  dcl ib      based fixed bin;

  a: proc;

    dcl (i,j,k) fixed bin;
   .
   .
   .
    call b( addr(i) );
   .
   .
   .
  end;

  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;

end p;
```

REDUCIBLE 関数

REDUCIBLE は、引数 (1 つ以上) が変更されない限り、プロシージャーまたは入り口を複数回呼び出す必要がないこと、およびプロシージャーの呼び出しに副次作用がないことを示します。

例えば、変更されないデータに基づいて結果を計算するユーザー作成の 関数には、REDUCIBLE が宣言されなければなりません。 乱数や時刻などの、変更されるデータに基づいて結果を計算する 関数は、IRREDUCIBLE として宣言する必要があります。

次の例では、REDUCIBLE が宣言の一部になっているため、f が一度だけ呼び出されます。 宣言に IRREDUCIBLE が使われていると、f が 2 度呼び出されます。

```
dcl (f) entry options( reducible ) returns( fixed bin );

select;
  when( f(x) < 0 )
    .
    .
    .
  when( f(x) > 0 )
    .
    .
    .
  otherwise
    .
    .
    .
end;
```

REPATTERN の使用

REPATTERN 組み込み関数を使って、ある日付/時刻パターンから別の日付/時刻パターンにデータを変換するコードを書くことができます。

例えば、「YYYY-MM-DD HH:MI:SS.999999 '」というパターンで今日の日付と時刻を取得する最も速い方法は、 TIMESTAMP と REPATTERN 組み込み関数を次のように使うことです：

```
repattern( timestamp(),
                'YYYY-MM-DD HH:MI:SS.999999',
                'YYYY-MM-DD-HH.MI.SS.999999' )
```

DESCLOCATOR または DESCLIST

DEFAULT(DESCLOCATOR) オプションが有効になっている場合、コンパイラーは従来のコンパイラーとほぼ同じように、記述子 (ストリングや構造体など) を必要とする引数を渡すために記述子ロケーターを使用します。

このオプションを使用すれば、エントリー・ポイントが宣言している引数をすべて 渡さなくても、エントリー・ポイントを呼び出すことができます。

またこのオプションを使用すると、構造体を渡してからそれをポインターとして 受け取るといった、あまり賢明ではないプログラミング方法を従来どおり使うこと ができます。

ただし、DEFAULT(DESCLOCATOR) を指定した場合にコンパイラーによって生成されるコードは、状況によっては DEFAULT(DESCLIST) の場合のコードよりもパフォーマンスが悪くなることがあります。

関連情報

455 ページの『PL/I 記述子』
この章では、実行時の PL/I ルーチン間での PL/I パラメーターの引き渡しの規則について説明します。

DEFINED 対 UNION

UNION 属性は、DEFINED 属性よりも強力で、より多くの機能を提供します。 さらにコンパイラーは、共用体参照の場合、より優れたコードを生成します。

次の例では、変数の対 b3 と b4 が、b1 および b2 と同じ機能を実行しますが、 コンパイラーは共用体の対の場合に、より最適化されたコードを生成します。

```
dcl b1 bit(32);
dcl b2 bit(16) def b1;

dcl
  1 * union,
    2 b3 bit(32),
    2 b4 bit(16);
```

DEFINED 属性ではなく UNION を使うコードは、誤って解釈されることが少なくなります。 共用体の中の変数宣言は 1 個所にあるので、共用体のメンバーが変更されたり、 ほかのすべてのメンバーが変更されても、それを理解しやすくなっています。 この動的変更は、DEFINED 変数を使う宣言では宣言ステートメントが複数行離れていることがあるので、認識が難しくなります。

名前付き定数対静的変数

名前付き定数は、VALUE 属性を使って変数を宣言することによって定義できます。 INITIAL 属性を指定して静的変数を使い、変数を変更しない場合は、VALUE 属性を使って変数を名前付き定数として宣言する必要があります。 コンパイラーは NONASSIGNABLE のスカラー STATIC 変数を、真の名前付き定数として扱いません。

コンパイラーは、コンパイル時に式が評価される時にはいつでも、 より最適化されたコードを生成するので、 名前付き定数を使って、読み易さを低下させずに効果的なコードを生成できます。 例えば次の例では、VERIFY 組み込み関数の 2 通りの使用法を用いて、 同一のオブジェクト・コードが生成されています。

```
dcl numeric char  value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );
```

次の例は、VALUE 属性を使って、読み易さを低下させずに 最適なコードを取得できる方法を示しています。

意味のある名前が付けられていない最適なコードの例

```
dcl  x  bit(8) aligned;

select( x );
  when( '01'b4 )
    .
    .
    .
  when( '02'b4 )
    .
    .
    .
  when( '03'b4 )
    .
    .
    .
end;
```

意味のある名前が付けられた最適でないコードの例

```
dcl (  a1  init( '01'b4)
      ,a2  init( '02'b4)
      ,a3  init( '03'b4)
      ,a4  init( '04'b4)
      ,a5  init( '05'b4)
    ) bit(8) aligned static nonassignable;

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
    .
```

```
    .
    .
  when( a2 )
    .
    .
    .
  when( a3 )
    .
    .
    .
end;
```

意味のある名前が付けられた最適なコードの例

```
dcl (  a1  value( '01'b4)
      ,a2  value( '02'b4)
      ,a3  value( '03'b4)
      ,a4  value( '04'b4)
      ,a5  value( '05'b4)
    ) bit(8);

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
    .
    .
    .
  when( a2 )
    .
    .
    .
  when( a3 )
    .
    .
    .
end;
```

ライブラリー・ルーチンの呼び出しの回避

ビット単位操作 (接頭部 NOT、2 項演算子 AND、2 項演算子 OR、およ び 2 項演算子 EXCLUSIVE OR) は、多くの場合、 ライブラリー・ルーチンを呼び出すと評価されます。

ただし、これらの操作は、次のいずれかの条件が真である場合に、 ライブラリーが呼び出されずに処理されます。

- 両方のオペランドが bit(1) である
- 両方のオペランドの位置が合い、同じ固定長である

特定の割り当て、式、および組み込み関数参照の場合は、 コンパイラーがライブラリー・ルーチンの呼び出しを生成します。 これらの呼び出しを回避すると、一般的にコードの実行速度が速くなります。

コンパイラーがこのような呼び出しを生成する時点を判断の助けとして、ライブラリー・ルーチンを使って変換が行われるときにはいつでも、コンパイラーがメッセージを生成します。

コードが BASED 構造体のメンバを REFER で参照する場合、コンパイラは多くの場合、実行時に構造体をマップするためにライブラリ・ルーチンへの呼び出しを 1 つ以上生成しなければならない。 これらの呼び出しは効率を低下させる可能性があるため、コンパイラー がこれらの呼び出しを行う場合には、メッセージを出して、 コード内の問題となる可能性のある場所を見つけることができるようにしています。

REFER もし、BASED の構造体を使用するコードがあり、コンパイラがこのメッセージでフラグを立てている場合は、対応するエクステントを持つ構造体を宣言するサブルーチンに構造体を渡すことで、パフォーマンスが向上する可能性があります。 これにより、 CALL ステートメントで構造体が一度マッピングされるが、コールされたサブルーチンで構造体にアクセスする際には、それ以上再マッピングされることはない。

ライブラリー・ルーチンのプリロード

PL/I ライブラリには、 RMODE 24 ルーチン IBMPOIOA が 1 つ含まれており、低レベルのシステム i/o 機能に使用される。 あなたのコードが RECORD i/o 、または SYSPRINT を STREAM OUTPUT （ STDSYS オプションでコンパイルせずに）ファイルとして使用する場合、このルーチンをプリロードするか、 (E)LPA。

第 4 部 他の製品に対するインターフェースの使用

© Copyright IBM Corp. 1999, 2025

第 16 章 ソート・プログラムの使用

コンパイラは、 PLISRT*x* (*x* = A, B, C, or D)と呼ばれるインターフェイスを提供し、 IBM-提供されたソート・プログラムを利用できるようにしている。

ソート・プログラムを PLISRT*x* で使用するには、次のようにしなければならない：

1. PLISRT*x*、ソートするフィールドの情報を渡す。 この情報には、レコード長、使用ストレージの最大値、戻りコードとして 使用する変数の名前、およびソートを実行するのに必要なその他の情報が含まれます。
2. ソートプログラムが必要とするデータセットを JCL DD ステートメントで指定する。

ソート・プログラムは、PL/I から使用されると、 大量のソート・フィールド上の正常な長さのレコードすべてをソートします。 ほとんどのタイプのデータは、昇順または降順でソートできます。 ソートされるデータのソースは、データ・セットである場合と、 ソートにレコードが必要になるたびにソート・プログラム によって呼び出されるユーザー作成の PL/I プロシージャーである場合とがあります。 同様に、ソートの宛先も、データ・セットでも、ソートされたレコードを処理す る PL/I プロシージャーでもかまいません。

PL/I プロシージャーを使えば、 ソート自体の前でも後でも処理を行えるので、1 回のソート・インターフェースの 呼び出しで、ソート操作をすべて完了することができます。 入出力を処理する PL/I プロシージャーは、ソート・プログラム自身から呼び出されるため、 事実上ソート・プログラムの一部となることを理解しておくことが大切です。

PL/I は、DFSORT と一緒に、または同じインターフェースを持ったプログラムと一緒に稼働することができます。 DFSORT はプログラム製品 のリリースである。 5740-SM1 DFSORT には、プログラム・ロジックを記述する必要をなくすために使用できる多くの組み込み機能があります（例えば、 INCLUDE, OMIT, OUTREC, と SUM ステートメント、さらに多くの ICETOOL 演算子)。 詳細については「*DFSORT Application Programming Guide*」を、ま たチュートリアルとしては「*Getting Started with DFSORT*」を参照してください。

以下の情報は DFSORT に適用されます。 DFSORT 以外のプログラムを使用することもできるため、実際の機能や制約事項はさまざまです。 このような機能や制約事項に関しては、「*DFSORT Application Programming Guide*」または、ご使用になるソート製品用のこれに相当する資料を参照してください。

ソート・プログラムを使うには、ソース・プログラム内に正しい PL/I ステートメントを入れなければならず、また JCL 内で正しいデータ・セットを指定しなければなりません。

ソート・プログラムの使用準備

ソート・プログラムを使用するには、 まず必要とするソートのタイプ、データ内のソート・フィールドの長さとフォーマット、 データ・レコード長、ソートに使う補助記憶域と主記憶域の大きさを決めます。

使用する PLISRT*x* エントリーポイントを決めるには、ソートされていないデータのソースと、ソートされたデータの行き先を決めなければならない。 データ・セットと PL/I サブルーチンのどちらかを選択します。 データ・セットを使用する方がより分かりやすく、パフォーマンスも速くなります。 PL/I サブルーチンを使用すると、より高い柔軟性と機能性が得られ るため、データをソートする前に処理したり印刷することができ、ソート 済みの形のまま直ちにデータを使用することができます。 入力または出力処理のサブルーチンを使用する場合は、331 ページの『データ入出力処理ルーチン』を参照してください。

エントリー・ポイント、データのソース、宛先を次に示します。

Entry point	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine
PLISRTD	Subroutine	Subroutine

© Copyright IBM Corp. 1999, 2025

使用するエントリー・ポイントを決定したら、今度はデータ・セットに関して以下の事項を決定する必要があります。

- ソート・フィールドの位置。 これらは 1 つのレコード全体あるいはその任意の一部 (複数の部分も可能) となります。
- これらのフィールドが表すデータのタイプ (例えば文字または 2 進数など)。
- 各フィールドでのソートを昇順にするか、降順にするか。
- 同じレコードは、入力されたとおりの順序で保持するか、 またはソート時に順序を変更することができるのか。

これらのオプションは、 PLISRT*x* の最初の引数である SORT ステートメントで指定する。 次に、ソートされるレコードに関して以下の事項を決定する必要があります。

- レコード・フォーマットが固定フォーマットであるか、可変フォーマットであるか。
- レコード長。これは可変フォーマットの最大長です。

これらは PLISRT*x* の第 2 引数である RECORD ステートメントで指定する。

最後に、ソート・プログラム用に確保しようとする主記憶域 と補助記憶域の大きさを決めなければなりません。 詳細については、326 ページの『ソート・プログラムに必要なストレージの決定』を参照してください。

ソート・タイプの選択

PL/I プログラム内で、ソート・インターフェース・サブルーチン PLISRT*x* に対する CALL ステートメントを使用することによって、ソートを指定します。 このサブルーチンには、 *X*= A、B、C、および D の 4 つのエントリー・ポイントがあります。 それぞれが、ソートされていないデータの異なるソースと、ソートされているデータの宛先を指定します。

例えば、PLISRTA の呼び出しは、未ソート・データ (ソートへの入力) が、ある 1 つのデータ・セット上にあることを指定し、ソート済みデータ (ソートからの出力) を別のデータ・セットに置くことを指定します。 CALL PLISRT*x* ステートメントに含めなければならないものは、ソートしようとするデータ・セットに関するソート・プログラム情報を示した引数リスト、ソートを行うフィールド、使用可能なスペースの大きさ、ソートが成功したかまたは失敗したかを示すための戻りコードを ソート・プログラムが入れる変数の名前、および使用可能な出力または入力の処理プロシージャーがあればその名前です。

ソート・インターフェース・ルーチンは、ソート・プログラムの引数リストを、PLISRT*x* 引数リストが提供する情報と、選択した PLISRT*x* エントリー・ポイントから作成します。 次に、制御はソート・プログラムに移されます。 出力または入力の処理ルーチンを指定していれば、それぞれの未ソートまたは ソート済みレコードを処理するのに必要な回数だけ、処理ルーチンがソート・プログラムによって呼び出されます。 ソート操作が完了するとソート・プログラムは、戻りコードで ソートが成功したか失敗したかを知らせて、PL/I 呼び出しプロシージャーに戻ります。 なお、その戻りコードは、インターフェース・ルーチンに 渡される引数のうちの 1 つに入れられています。 次に戻りコードは、処理を継続すべきかどうかを 判別するために、PL/I ルーチン中でテストすることができます。 323 ページの図 7 は、この操作を示す単純化されたフローチャートです。

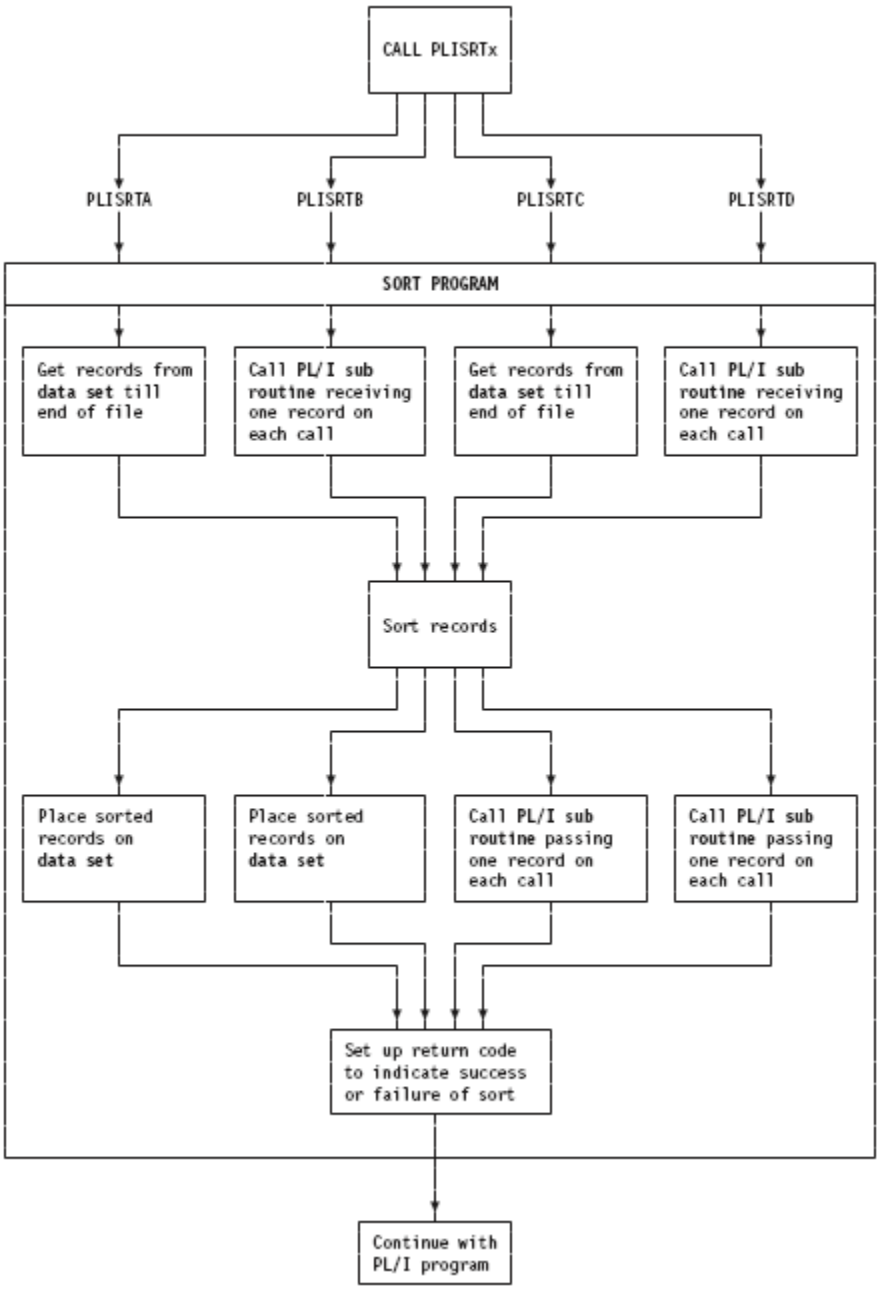


図 7. ソート・プログラムの制御の流れ

ソート・プログラムそのものにおいては、 ソート・プログラムと入力および出力処理ルーチン との間の制御の流れは、 戻りコードで制御されます。 ソート・プログラムは、その処理の途中で、適切な時点でこれらのルーチンを呼び出します。 (ソート・プログラムとそれに関連した資料では、これらのルーチンは ユーザー出口 と呼ばれます。 ソートされる入力を渡すルーチンは、E15 ソート・ユーザー出口です。 ソート済み出力を処理する ルーチンは、E35 ソート・ユーザー出口です。) これらのルーチンから、ソート・プログラムは、そのルーチンをもう一度呼び出すべきか、 または次の処理段階に進むべきかを示す戻りコードがくることを予期します。

ソート・プログラムに関して以下の重要なポイントを覚えておく必要があります。

- ソート・プログラムは、完全なソート命令を処理する自己完結型プログラムです。

- ソート・プログラムは戻りコードを使用して、呼び出し側と通信したり、自分で呼び出したユーザー出口と通信したりします。

ソート・フィールドの指定

このトピックでは、PL/I からソート・プログラムを使用するために指定しなければならない必須の PL/I ステートメントについて説明します。 SORT 文は PLISRT*x* の第一引数である。

SORT ・ステートメントの構文は、文字列式でなければならない：

```
'bSORTbFIELDS=(start1,length1,form1,seq1,
...startn,lengthn,formn,seqn)[,other options]b'
```

次の例を参照してください。

```
' SORT FIELDS=(1,10,CH,A) '
```

b
1 つ以上のブランクを表します。 ここに示されているブランクは必須です。 これ以外のブランクは認められません。

start,length,form,seq
ソート・フィールドを定義します。 指定するソート・フィールドの数は任意 ですが、フィールドの全長には限度があります。 複数のフィールドがソートされる場合は、レコードはまず 最初のフィールドに従ってソートされ、次に、等しい値を持つレコードが 2 番目のフィールドに従ってソートされ、というようになります。 すべてのソート値が等しい場合、 EQUALS オプションを使用しない限り、等しいレコードの順序は任意となる。 (この定義リストの後半を参照してください。) フィールドは互いにオーバーレイしてもかまいません。

DFSORT (5740-SM1) の場合、ソート・フィールドの最大合計長は 4092 バイトに制限され、すべてのソート・フィールドはレコードの先頭から 4092 バイト以内になければならない。 他のソート製品は異なる制限を持つ場合があります。

開始
レコード内の開始位置を指定します。 "バイト. ビット" 表記を使用できるバイナリー・データを除き、値はバイト単位で指定します。 ストリング内の最初のバイトはバイト 1 と見なされ、最初のビットはビット 0 と見なされます。 (したがって、バイト 2 の 2 番目のビットは 2.1 と呼ばれます。) 可変長レコードの場合、4 バイト長の接頭部を入れて、5 をデータの最初のバイトにします。

長さ
ソート・フィールドの長さを指定します。 "バイト. ビット" 表記を使用できる 2 進数を除き、値はバイト単位で指定します。 ソート・フィールドの長さには、それぞれのデータ・タイプ別に制約があります。

フォーム
データのフォーマットを指定します。 これは、ソートの目的で用いられるフォーマットです。 PL/I ルーチンとソート・プログラム間でやり取りされるデータはすべて、 文字ストリングの形式でなければなりません。 主なデータ・タイプとその長さに対する制約事項を次に示します。 これ以外のデータ・タイプも特殊目的のソート用に用意されています。 これに関しては、「*DFSORT Application Programming Guide*」または、ご使用のソート製品用の資料を参照してください。

コード	データ・タイプと長さ
CH	文字 1 から 4096 まで
ZD	ゾーン 10 進数符号付き 1 から 32 まで
PD	パック 10 進符号付き 1 から 32 まで
FI	固定小数点、符号付き 1 から 256 まで
BI	2 進数、符号なし 1 ビットから 4092 バイトまで
FL	浮動小数点、符号付き 1 から 256 まで

全フィールドの合計長は、4092 バイトを超えてはなりません。

seq
データがソートされる順序を指定します。

A
昇順 (つまり、1,2,3,...)

D
降順 (つまり、...,3,2,1)

注 : E を指定することはできません。 その理由は、PL/I は、ユーザーが提供した順序を渡す手段を備えていないからです。

他のオプション
使用するソート・プログラムに応じて、ほかにいくつかのオプションを指定できます。 それらのオプションは FIELDS オペランドとの間、およびオプション同士の間をコンマで区切らなければなりません。 オペランドとオペランドの間にブランクを入れないでください。

FILSZ=y
ソートのレコード数を指定し、ソート・プログラムによる最適化を可能にします。 y が単に近似値であれば、y の前に E を付けなければなりません。

SKIPREC=y
ソート・プログラムが入力ファイルの先頭から y 個のレコードを無視して残りのレコードをソートし始めるように指定します。

CKPT または CHKPT
チェックポイントをとることを指定します。 このオプションを使用する場合は、SORTCKPT データ・セットを指定する必要があります。 さらに、DFSORT をインストールする場合は、16NCKPT=NO インストール・オプションを指定する必要があります。

EQUALS|NOEQUALS
等しいレコードの順序を、入力したときと同じままにするか (EQUALS)、または任意にするか (NOEQUALS) を指定します。 NOEQUALS を使えば、ソート・パフォーマンスを上げることができます。 デフォルト・オプションは、ソート・プログラムがインストールされるときに選択されます。 IBM 提供のデフォルトは、NOEQUALS です。

DYNALLOC=(d,n)
(OS/VS ソート・プログラムの場合のみ) プログラムは中間記憶装置を動的に割り振ることを指定します。

d
装置タイプ (3380 など) を指定します。

n
作業域数を指定します。

ソートするレコードの指定

このトピックでは、PL/I からソート・プログラムを使用するために指定しなければならない必須の PL/I ステートメントについて説明します。 RECORD ステートメントは *PLISRTx* の第 2 引数である。

RECORD 文の構文は文字列式でなければならず、評価されると以下の構文になる：

```
'bRECORDbTYPE=rectype[,LENGTH=(L1,[,,L4,L5])]b'
```

次の例を参照してください。

```
' RECORD TYPE=F,LENGTH=(80) '
```

b
1 つ以上のブランクを表します。 ここに示されているブランクは必須です。 これ以外のブランクは認められません。

タイプ
次のように、レコード・タイプを指定します。

F
固定長

V
可変長 EBCDIC

D
可変長 ASCII

未ソート・データとソート済みデータを処理するのに 入力ルーチンと出力ルーチンを使用するときでも、 ソート・プログラムが使用する作業データ・セットに 適用されるレコード・タイプを指定しなければなりません。

可変長ストリングを入力ルーチン (E15 出口) からソート・プログラムに渡すときには、 通常はレコード・フォーマットとして V を指定すべきです。 ただし、F を指定すると、最大長になるまでレコードにブランクが埋め込まれます。

長さ
ソートするレコードの長さを指定します。 PLISRTA または PLISRTC を使用する場合、長さは入力データセットから取得されるため、 LENGTH を省略することができる。 ソートできるレコードの最大長と最小長には制限があることに注意してください。 可変長のレコードの場合は、4 バイトの接頭部を含める必要があります。

L1
ソートするレコードの長さを指定します。 VSAM 、変化するレコードとしてソートされたデータセットの場合、最大レコードサイズ＋ 4 となる。

,,
PL/I からの呼び出し時にソート・プログラムに適用されない 2 つの引数を表します。 後続の引数を使用する場合は、コンマを組み込む必要があります。

L4
可変長レコードの使用時の最小レコード長を指定します。 これを指定すると、この値は、ソート・プログラムによって最適化のために使用されます。

L5
可変長レコードの使用時の形式指定上の (最も一般的な) レコード長を指定します。 これを指定すると、この値は、ソート・プログラムによって最適化のために使用されます。

最大レコード長 : レコードの長さは、ユーザーが指定する最大長を超えることはありません。 最大レコード長は、可変長レコードの場合は 32756 バイトであり、固定長レコードの場合は 32760 バイトです。

ソート・プログラムに必要なストレージの決定

ソートには、主記憶域と補助記憶域の両方が必要です。

主記憶域

DFSORT 、メインストレージの最小値は 88K バイトだが、最高のパフォーマンスを得るためには、より多くのストレージ（1 メガバイト以上）を推奨する。 DFSORT は、 仮想または拡張アーキテクチャー・プロセッサーの上のストレージを利用することができる。 16M z/OS DFSORT では、拡張ストレージも利用できる。 次のようにストレージ・パラメーターを渡せば、ソート・プログラムが使用可能 なストレージを最大限使用するように指定することができます。

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B||UNSPEC('MAX');
CALL PLISRTA
    (' SORT FIELDS=(1,80,CH,A) ',
     ' RECORD TYPE=F,LENGTH=(80) ',
       MAXSTOR,
       RETCODE,
       'TASK');
```

ファイルを E15 または E35 出口ルーチン内でオープンする場合には、ファイルを正常にオープンできるよう、十分な残余ストレージを確保しておいてください。

補助記憶域

ある特定のソート操作の最小補助記憶域を計算するのは、複雑な作業です。 補助記憶域によって最大限に効率を上げるには、 できるかぎり直接アクセス記憶装置 (DASD) を使用します。 プログラムの効率化に関する詳細は、『*DFSORT アプリケーション・プログラミング・ガイド*』、特に DFSORT が必要な補助記憶域を判断して割り当ててくれるワークスペースの動的割り当てに関する情報を参照してください。

ソートが確実に機能するのに十分なストレージを提供することだけに関心があるのであれば、SORTWK データセットの合計サイズを、ソートされるレコードの 3 セットを保持するのに十分な大きさにする。 (3 つのデータ・セット内に十分なスペースがあれば、3 より大きい数を指定してもとくに利益は得られません。)

ただし、この推奨値は近似値であるため、うまくいかないこともあるので、 その場合には、ソートに関する資料を参照してください。 この推奨値でうまくいった場合でも、スペースを無駄に使っている可能性はあります。

ソート・プログラムの呼び出し

CALL PLISRT*x*。 このトピックでは、使用可能なエントリー・ポイントと引数をリストします。

表 *36. PLISRTx (x = A, B, C,* または *D)*へのエントリーポイントと引数

エントリー・ポイント	引数
PLISRT ソート入力: データ・セット ソート出力: データ・セット	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード[、データ・セット接頭部、メッセージ・レベル、ソート手法])
PLISRTB ソート入力: PL/I サブルーチン ソート出力: データ・セット	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード、入力ルーチン [、データ・セット接頭部、メッセージ・レベル、ソート手法])
PLISRTC ソート入力: データ・セット ソート出力: PL/I サブルーチン	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード、出力ルーチン [、データ・セット接頭部、メッセージ・レベル、ソート手法])
PLISRTD ソート入力: PL/I サブルーチン ソート出力: PL/I サブルーチン	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード、入力ルーチン、出力ルーチン[、データ・セット接頭部、メッセージ・レベル、ソート手法])
SORT ステートメント	ソートプログラム SORT ステートメントを含む文字列式。 ソート・フィールドとフォーマットを記述する。 324 ページの『ソート・フィールドの指定』を参照してください。
RECORD ステートメント	ソートプログラム RECORD ステートメントを含む文字列式。 データの長さとレコード・フォーマットを記述する。 325 ページの『ソートするレコードの指定』を参照してください。
ストレージ	ソート・プログラムで使用される主記憶域の最大値を示す 固定 2 進式。 DFSORT の場合は >88K バイトでなければならない。 326 ページの『ソート・プログラムに必要なストレージの決定』も参照してください。

表 *36. PLISRTx (x = A, B, C,* または *D)*へのエントリーポイントと引数 *(*続き*)*

エントリー・ポイント	引数
戻りコード	精度 (31,0) の固定 2 進変数。 ソート・プログラムが完了するとこの中に 戻りコードが入る。 戻りコードの意味は次のとおり。 0= ソート・プログラムは正常に完了 16= ソート・プログラムは失敗 20= ソート・プログラム・メッセージ・データ・セットが欠落
入力ルーチン	(PLISRTB と PLISRTD のみ) ソート・プログラムにレコードをソート出口 E15 で渡すのに 使用する PL/I 外部または内部プロシージャーの名前。
出力ルーチン	(PLISRTC および PLISRTD の場合のみ。) ソートがソート出口 E35 でソート済みレコード を渡す PL/I 外部または内部プロシージャーの名前。
DD 名の接頭部	ソートデータセットの ddnames のデフォルト接頭辞'SORT' を置き換える 4 文字の文字列表現 SORTIN, SORTOUT と SORTWKnn、および使用されている場合は SORTCNTL。 したがって、引数が "TASK "の場合、ddnames TASKIN、TASKOUT, TASKWKnn, and TASKCNTL が使用できる。 この機能により、同一ジョブ・ステップ内でソート・プログラムを 複数呼び出すことができる。 4 文字はアルファベットで始まり、予約名 PEER, BALN, CRCX, OSCL, POLY, DIAG, SYSC, or LIST のいずれでもないこと。 後続引数のうちのいずれかが必要だが、この引数は必要でない場合、 この引数にはヌル・ストリングをコーディングしなければならない。
メッセージ・レベル	ソート・プログラムの診断メッセージの処理方法を示す 次のような 2 文字の文字ストリング式。 NO メッセージを SYSOUT へ出さない AP すべてのメッセージを SYSOUT へ送る CP 重大メッセージを SYSOUT へ送る SYSOUT は通常、プリンターに割り振られるため、簡略記号文字 「P」を使用。ソート・プログラムによっては、他のコードを使用することもできる。 このようなコードに関する詳細は、「*DFSORT Application Programming Guide*」を参照。後続引数が必要だが、この引数は必要でない場合、 この引数にはヌル・ストリングをコーディングしなければならない。
ソート手法	(これは DFSORT では使用されない。互換性の ためのみに存在。) 次のように、行おうとするソートのタイプを示す長さ 4 の文字ストリング。 PEER 対等機能ソート BALN 平衡 CRCX 交差ソート OSCL 振動 POLY 多相ソート 通常、ソート・プログラムは、ユーザーの処置を必要としないで、 使用可能なスペースの大きさを分析して最も効率のよい手法を選択する。 この引数を使用するのは、ソート上の問題で迂回を行うときか、または、 別の手法を使えばパフォーマンスが向上するのが確実なときに限ら なければならない。 詳細については「*DFSORT Application Programming Guide*」を 参照。

次の例は、CALL PLISRT*x* ステートメントが通常とる形式を示しています。

例 1

以下の例では PLISRTA に対する呼び出しが示されています。 PLISRTA では、1048576 (1 メガバイト) のストレージと、FIXED BINARY (31,0) として宣言された戻りコード RETCODE が使用され、80 バイトのレコードが SORTIN から SORTOUT にソートされます。

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE);
```

例 2

この例は、入力、出力、および作業データ・セットが TASKIN、TASKOUT、TASKWK01 などという名前に変わっている点を除けば、例 1 と同じです。

ソート・プログラムを 1 つのジョブ・ステップの中 で 2 回呼び出したときに、このようになることがあります。

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE,
                'TASK');
```

例 3

次の例は、2 つのフィールド上でソートが行われることを除いて例 1 と同じです。 最初に文字である 1 バイト目から 10 バイト目でソートされ、それらの文字が同じ場合は、2 進数フィールドである 11 バイト目と 12 バイト目でソートされます。

いずれのフィールドも、昇順でソートされます。

```
CALL PLISRTA (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE);
```

例 4

この例は、PLISRTB の呼び出しを示しています。

PL/I ルーチン PUTIN によって入力がソート・プログラムに渡されます。 80 バイト固定長レコードの 1 文字目から 10 文字目までに対してソートが実行されます。

```
CALL PLISRTB (' SORT FIELDS=(1,10,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE,
                PUTIN);
```

例 5

この例では、 PLISRTD。

入力は PL/I ルーチン PUTIN によって供給され、出力は PL/I ルーチン PUTOUT に渡される。 ソートされるレコードは 84 バイト可変 (長さの接頭部を含む) です。 データのバイト 1 から 5 までを昇順でソートし、次にこれらのフィールドが等しい場合は、バイト 6 から 10 までを降順でソートします。 (4 バイト長さ接頭部が含まれているので、 実際に開始点で使う値は 5 と 10 であることに注意してください。) 両方のフィールドが同じである場合は、入力の順序は保持されます。 (EQUALS)

```
CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(84) ',
                1048576,
                RETCODE,
```

```
                PUTIN,       /*input routine (sort exit E15)*/
                PUTOUT);     /*output routine (sort exit E35)*/
```

ソートが成功したかどうかの判別

ソート・プログラムはソートが完了すると、 PLISRT*x* の呼び出しの 4 番目の引数内で指定した変数内に、戻りコードを設定します。

次に、 CALL PLISRT*x* ステートメントの後のステートメントに制御を戻します。 戻された値は、次のようにソートが成功または失敗のどちらであったかを示します。

0
ソート・プログラムは正常に完了した

16
ソート・プログラムは失敗した

20
ソート・メッセージ・データ・セットが欠落している

戻りコードを渡す先の変数は FIXED BINARY (31,0) と宣言する必要があります。 通常の作業では、CALL PLISRT*x* ステートメントの後で戻りコードの値をテストし、操作が成功したか失敗したかに応じて適切な処置をとります。

次の例を参照してください (戻りコードが RETCODE の場合):

```
IF RETCODE¬=0 THEN DO;
  PUT DATA(RETCODE);
  SIGNAL ERROR;
END;
```

ソート後のジョブ・ステップが、ソートの成功、失敗に依存している場合は、 ソート・プログラム内で戻された値を、PL/I プログラムからの戻りコードとして 設定する必要があります。 これで、その戻りコードを次のジョブ・ステップに使用することができます。 PL/I 戻りコードは、PLIRETC への呼び出しで設定されます。 PLIRETC を呼び出すには、ソート・プログラムから返される値を使用します。

```
CALL PLIRETC(RETCODE);
```

この PLIRETC の呼び出しを、ソート・プログラムへ制御情報を渡すのに 戻りコードが使用される入出力ルーチン内での呼び出しと混同しないでください。

ソート・プログラム用のデータ・セットの確立

システムに認識されていないライブラリーに DFSORT がインストールされている場合は、JOBLIB または STEPLIB DD ステートメントで DFSORT ライブラリーを指定する必要があります。

ソート・プログラムを呼び出すときは、特定のソート・データ・セットがオープンされていてはなりません。

SYSOUT
ソート・プログラムからのメッセージが書き込まれるデータ・セット (通常はプリンター)

ソート作業データ・セット: SORTWK01–SORTWK32

注 : 32 より多いソート作業データ・セットを指定した場合、DFSORT は 最初の 32 個しか使用しません。

******WK01–****WK32**
ソート処理で使用される 1 から 32 個までのデータ・セット。

これらは直接アクセスでなければなりません。 必要なスペースとデータ・セットの数については、326 ページの『ソート・プログラムに必要なストレージの決定』を参照してください。

******** は、 PLISRT*x* への呼び出しにおいてデータ・セット接頭部引数 として指定できる 4 文字を表します。 この文字を使えば、SORT 以外の接頭部を使ってデータ・セットを使用する こともできま

す。 これは英字で始まらなければならず、 名前は PEER、BALN、CRCX、OSCL、POLY、SYSC、LIST、または DIAG であってはなりません。

入力データ・セット: SORTIN

******IN**
PLISRTA および PLISRTC を呼び出すときに使用する入力データ・セット

詳しくは、****WK01–****WK32 を 参照してください。

出力データ・セット: SORTOUT

******アウト**
PLISRTA および PLISRTB を呼び出すときに使用する出力データ・セット

詳しくは、****WK01–****WK32 を 参照してください。

チェックポイント・データ・セット: SORTCKPT
チェックポイント・データを保持するために使用されるデータ・セット (CKPT オプションまたは CHKPT オプションが SORT ステートメント引数で使用されていて DFSORT 16NCKPT=NO インストール・オプションが指定されている場合)

このプログラム DD ステートメントに関する詳細は、「*DFSORT Application Programming Guide*」を参照してください。

DFSPARM SORTCNTL
追加の制御ステートメントまたは変更された制御ステートメントが入った (オプショナルの) データ・セット

このプログラム DD ステートメントに関する詳細は、「*DFSORT Application Programming Guide*」を参照してください。

詳しくは、****WK01–****WK32 を 参照してください。

ソート・データの入出力

ソートするデータのソースは、データ・セットから直接提供される場合と、 ユーザーが作成したルーチン (ソート出口 E15) によって間接的に提供される場合とがあります。 同様に、ソートされた出力の宛先も、データ・セットである場合と、 ユーザー提供のルーチン (ソート出口 E35) である場合とがあります。

PLISRTA は、データ・セットからデータ・セットにソートするものであるため、 すべてのインターフェースの中で最も単純なインターフェースです。 PLISRTA プログラムの例については、335 ページの『#unique_573/unique_573_Connect_42_fig148』 を参照してください。 他のインターフェースには、入力処理ルーチンと出力処理ルーチンの一方または両方が必要です。

データ入出力処理ルーチン

入力処理ルーチンと出力処理ルーチンは、 PLISRTB, PLISRTC、または PLISRTD が使用されると、ソートによって呼び出される。

これらのルーチンは、PL/I で作成する必要があり、内部プロシージャーであっても外部プロシージャーであってもかまいません。 PLISRT*x* を呼び出すルーチンの内部手続きである場合、名前のスコープに関しては、通常の内部手続きと同じように動作する。 入力手続き名と出力手続き名は、 PLISRT*x* を呼び出す手続きの中で知っていなければならない。

これらのルーチンは、各レコードが、ソート・プログラムで必要になるか、 ソート・プログラムから渡されるたびに個別に呼び出されます。 したがって、各ルーチンは一度に 1 つのレコードを処理できるように作成しなければなりません。 プロシージャ内で AUTOMATIC として宣言された変数は、呼び出しの間、その値を保持しません。 従って、カウンターのように、ある呼び出しから次の呼び出しまで保持する必要がある項目は、 STATIC として宣言するか、含むブロックで宣言する必要がある。

E15、 E35 ソート出口は、 MAIN 手続きであってはならない。

E15 — 入力処理ルーチン (ソート出口 E15)

入力ルーチンは、通常、データがソートされる前に、データに対して何らかの 処理を加えるのに使用されます。

入力ルーチンは、336 ページの『#unique_575/unique_575_Connect_42_fig149』 および 337 ページの『#unique_576/unique_576_Connect_42_fig147』 に示してあるとおり、データを印刷するのに使用するか、正しい結果を出すためにソート・フィールドを生成または操作するのに使用することができます。

入力処理ルーチンは、PLISRTB または PLISRTD を呼び出すときに ソート・プログラムが使用します。 ソート・プログラムはレコードが必要な場合は、文字ストリング・フォーマットでレコードを返す入力ルーチンを呼び出して、戻りコード 12 を出力します。 この戻りコードの意味は、渡したレコードはソートの対象になるという ことです。 ソート・プログラムは、戻りコード 8 が渡されるまで、このルーチンを呼び出し続けます。 戻りコード 8 の意味は、すべてのレコードが既に渡されていて、ソート・プログラムがそのルーチンを再び呼び出す必要がないということです。 戻りコードが 8 のときにレコードが戻された場合、そのレコードはソート・プログラム によって無視されます。

ルーチンによって戻されるデータは、文字ストリングでなければなりません。 この文字ストリングは、固定でも可変でも構いません。 可変の場合、PLISRT*x* への呼び出し内の 2 番目の引数である RECORD ステートメント内のレコード・フォーマットとして、通常は、V を指定する必要があります。 しかし、F を指定することも可能で、その場合は、 ストリングが最大長になるようブランクが埋め込まれます。 レコードは RETURN ステートメントを使って戻されるため、PROCEDURE ステートメント 内で RETURNS 属性を指定しなければなりません。 戻りコードは、PLIRETC の呼び出し内で設定されます。 代表的な入力ルーチンのフローチャートを、 以下の図 333 ページの図 8 に示します。

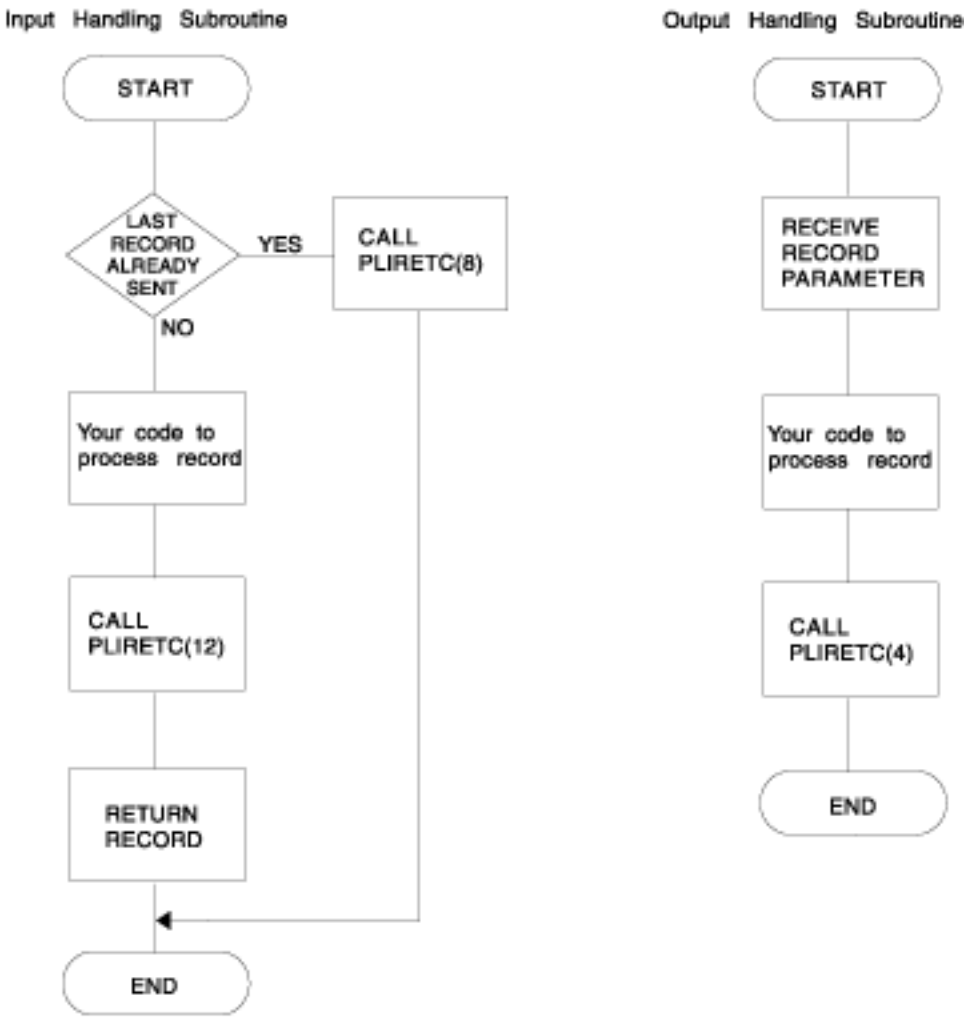


図 *8.* 入力および出力処理サブルーチンのフローチャート

代表的な入力ルーチンの骨組みコードを、333 ページの『#unique_574/unique_574_Connect_42_fig142』に示します。

入力プロシージャー用の骨組みコード

```
E15: PROC RETURNS (CHAR(80));
     /*----------------------------------------------------------------*/
     /*RETURNS attribute must be used specifying length of data to be */
     /* sorted, maximum length if varying strings are passed to Sort. */
     /*----------------------------------------------------------------*/
   DCL STRING CHAR(80); /*-------------------------------------------*/
                        /*A character string variable will normally be*/
                        /* required to return the data to Sort       */
                        /*-------------------------------------------*/

   IF LAST_RECORD_SENT THEN
     DO;
      /*----------------------------------------------------------------*/
      /*A test must be made to see if all the records have been sent,  */
      /*if they have, a return code of 8 is set up and control returned*/
      /*to Sort                                                         */
      /*----------------------------------------------------------------*/

        CALL PLIRETC(8);  /*-----------------------------------------*/
                          /* Set return code of 8, meaning last record */
                          /* already sent.                           */
                          /*-----------------------------------------*/
        RETURN('');
     END;
```

```
   ELSE
     DO;
      /*-----------------------------------------------*/
      /* If another record is to be sent to Sort, do the*/
      /* necessary processing, set a return code of 12  */
      /* by calling PLIRETC, and return the data as a   */
      /* character string to Sort                       */
      /*-----------------------------------------------*/

   ****(The code to do your processing goes here)

       CALL PLIRETC (12);  /*--------------------------------------*/
                           /* Set return code of 12, meaning this  */
                           /* record is to be included in the sort */
                           /*--------------------------------------*/
      RETURN (STRING);  /*Return data with RETURN statement*/
     END;

  END;   /*End of the input procedure*/
```

入力ルーチンの例については、336 ページの『#unique_575/unique_575_Connect_42_fig149』 および 337 ページの『#unique_576/unique_576_Connect_42_fig147』 を参照してください。

ソート・プログラムでは、戻りコード 12 (ソートに現行レコードを組み込む) と戻りコード 8 (すべてのレコードを送信済み) 以外に戻りコード 16 (ソート・プログラム失敗) が使用されます。 戻りコード 16 が発生すると、ソートは終了し、ソート・プログラムは制御を PL/I プログラムに返します。

注 : PLIRETC への呼び出しは、PL/I プログラムから渡され、それ以降の任意の ジョブ・ステップで使用することができる戻りコードを設定します。 出力処理ルーチンを使用した後 は、PLISRT*x* を呼び出してから PLIRETC を呼び出し、 戻りコードをリセットして、ゼロ以外の完了コードが出ないようにすることをお勧めします。 ソート・プログラムからの戻りコードを引数として 使用して PLIRETC を呼び出せば、PL/I 戻りコードにソートの成功または 失敗を反映させることができます。 この方法は、337 ページの『#unique_577/unique_577_Connect_42_fig1410』 に示してあります。

E35 – 出力処理ルーチン (ソート出口 E35)

出力処理ルーチンは通常、ソート後に必要なすべての処理に使われます。

例えば、出力処理ルーチンは、337 ページの『#unique_577/unique_577_Connect_42_fig1410』 や 337 ページの『#unique_576/unique_576_Connect_42_fig147』 にあるように、ソートされたデータを印刷したり、ソートされたデータで詳細情報を生成したりする場合に使用できます。 出力処理ルーチンは、ソート・プログラムが PLISRTC または PLISRTD を呼び出すときに使用します。 レコードのソートが終われば、ソート・プログラムはそれを一度に 1 つずつ、出力処理ルーチンに渡します。 次に、出力ルーチンは、必要に応じてそれを処理します。 すべてのレコードを渡し終えると、ソート・プログラムはその戻りコードをセットアップし、 CALL PLISRT*x* ステートメントの後のステートメントに戻ります。 ソート・プログラムは、最終レコードに達したことを出力処理ルーチンに示しません。 したがって、データ終了処理は、PLISRT*x* を呼び出すプロシージャーで行わなければなりません。

レコードはソート・プログラムから出力ルーチンへ文字ストリングとして 渡されるため、そのデータを受け取るには、出力処理サブルーチン内で 文字ストリング・パラメーターを宣言しなければなりません。 また、出力処理サブルーチンはソート・プログラムに戻りコード 4 を渡して、別のレコードを処理する準備ができたことを示す必要があります。 この戻りコードは、PLIRETC への呼び出しによって設定します。

ソートを停止するには、ソート・プログラムに戻りコード 16 を渡します。 この場合、ソート・プログラムは戻りコード 16 (ソート失敗) を出力して呼び出し側プログラムに制御を返します。

ソート・プログラムからルーチンに渡されるレコードは、文字ストリング・パラメーターです。 PLISRT*x* に対する呼び出しにおいて 2 番目の引数でレコード・タイプを F として指定した場合は、レコード長を指定してこのパラメーターを宣言する必要があります。 レコード・タイプを V として指定する場合は、次の例にあるように、パラメーターを調整可能なパラメーターとして宣言する必要があります。

```
DCL STRING CHAR(*);
```

338 ページの『#unique_579/unique_579_Connect_42_rahvar』 は、可変長レコードをソートするためのプログラムを示しています。

典型的な出力処理ルーチンのフローチャートを、333 ページの図 8 に示してあります。 典型的な出力処理ルーチンの骨組みコードは、335 ページの『#unique_578/unique_578_Connect_42_fig144』 に示してあります。

出力処理プロシージャー用の骨組みコード

```
E35: PROC(STRING);      /*The procedure must have a character string
                          parameter to receive the record from Sort*/

  DCL STRING CHAR(80);  /*Declaration of parameter*/

 (Your code goes here)

  CALL PLIRETC(4);   /*Pass return code to Sort indicating that the next
                       sorted record is to be passed to this procedure.*/
   END E35;          /*End of procedure returns control to Sort*/
```

PLIRETC を呼び出すと、PL/I プログラムから渡され、 その後の任意のジョブ・ステップで使用できる戻りコードが 設定されることに注意しなければなりません。 出力処理ルーチンを使用したときは、PLISRT*x* を呼び出してから PLIRETC を呼び出して戻りコードをリセットし、ゼロ以外の完了コードが発生しないようにすることをお勧めします。 ソート・プログラムからの戻りコードを引数として 使用して PLIRETC を呼び出せば、PL/I 戻りコードにソートの成功または 失敗を反映させることができます。 この方法は、このトピックに続く例で示されています。

PLISRTA の呼び出し例

このトピックでは、PLISRTA プログラムの例を示します。

PL/I の入力処理ルーチンおよび出力処理ルーチンが、戻りコード情報をソート・プログラムに伝達するたびに、戻りコード・フィールドがゼロにリセットされます。 そのため、そのような戻りコードは、ソート・プログラムの特定用途以外の通常の戻りコードとしては使用されません。

処理条件に関する詳細、特に入力および出力処理ルーチン時に生じる条件 に関しては、「*z/OS Language Environment Programming Guide*」を参照してください。

PLISRTA-入力データセットから出力データセットへのソート：

```
//OPT14#7  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX106: PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                     1048576
                     RETURN_CODE);
     SELECT (RETURN_CODE);
       WHEN(0) PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT (
            'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
       CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/
       END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*
```

PLISRTB の呼び出し例

このトピックでは、PLISRTB プログラムの例を示します。 このプログラムは、入力ルーチンを呼び出してデータを取得したり、ソート済みレコードをデータ・セット上に配置したりします。

PLISRTB - 入力処理ルーチンから出力データ・セットへのソート

```
//OPT14#8  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX107:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E15X);
     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER   PUT SKIP EDIT
           ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
     END /* select */;
     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E15X:   /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT
            STREAM AND PUTS THEM BEFORE THEY ARE SORTED*/
     PROC RETURNS (CHAR(80));
          DCL SYSIN FILE RECORD INPUT,
              INFIELD CHAR(80);

          ON ENDFILE(SYSIN) BEGIN;
             PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
             CALL PLIRETC(8);  /* signal that last record has
                                  already been sent to sort*/
             INFIELD = '';
             GOTO ENDE15;
             END;

           READ FILE (SYSIN) INTO (INFIELD);
           PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
           CALL PLIRETC(12);  /* request sort to include current
                                 record and return for more*/
    ENDE15:
          RETURN(INFIELD);
          END E15X;
  END EX107;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT  DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

PLISRTC の呼び出し例

このトピックでは、PLISRTC プログラムの例を示します。 このプログラムは、入力データ・セット内のレコードをソートしたり、出力処理ルーチンを呼び出してソート済みデータを出力したりします。

PLISRTC - 入力データ・セットから出力処理ルーチンへのソート

```
//OPT14#9  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX108:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E35X);
        SELECT(RETURN_CODE);
          WHEN(0)  PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
          WHEN(16) PUT SKIP EDIT
              ('SORT FAILED, RETURN_CODE 16') (A);
          WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
          OTHER    PUT SKIP EDIT
              ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
        END /* select */;
     CALL PLIRETC (RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E35X:   /* output handling routine prints sorted records*/
       PROC (INREC);
          DCL INREC CHAR(80);
          PUT SKIP EDIT (INREC) (A);
          CALL PLIRETC(4); /*request next record from sort*/
       END E35X;
    END EX108;
/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT   DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

PLISRTD の呼び出し例

このトピックでは、PLISRTD プログラムの例を示します。 このプログラムは、入力処理ルーチンを呼び出して未ソート・データを出力したり、出力処理ルーチンを呼び出してソート済みデータを出力したりします。

PLISRTD-入力処理ルーチンから出力処理ルーチンへのソート：

```
//OPT14#10  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX109:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);
     CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E15X,
                  E35X);

     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT
           ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
```

```
     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

  E15X:   /* Input handling routine prints input before sorting*/
        PROC RETURNS(CHAR(80));
             DCL INFIELD CHAR(80);

             ON ENDFILE(SYSIN) BEGIN;
                PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT.  ',
                    'SORTED OUTPUT SHOULD FOLLOW')(A);
                CALL PLIRETC(8);  /* Signal end of input to sort*/
                INFIELD = '';
                GOTO ENDE15;
             END;

             GET FILE (SYSIN) EDIT (INFIELD) (A(80));
             PUT SKIP EDIT (INFIELD)(A);
             CALL PLIRETC(12);  /*Input to sort continues*/
   ENDE15:
            RETURN(INFIELD);
             END E15X;

   E35X:   /* Output handling routine prints the sorted records*/
        PROC (INREC);

             DCL INREC CHAR(80);
             PUT SKIP EDIT (INREC) (A);
      NEXT:  CALL PLIRETC(4); /* Request next record from sort*/
             END E35X;
  END EX109;
```

```
 /*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
```

可変長レコードのソートの例

以下の例では、可変長レコードをソートするためのプログラムが示されています。

入出力処理ルーチンを使った可変長レコードのソート

```
 //OPT14#11 JOB ...
 //STEP1 EXEC IBMZCBG
 //PLI.SYSIN DD *
   /* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH
      RECORDS */

   EX1306:  PROC OPTIONS(MAIN);
            DCL RETURN_CODE FIXED BIN(31,0);
            CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
                         ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                         /*NOTE THAT LENGTH IS MAX AND INCLUDES
                           4 BYTE LENGTH PREFIX*/
                         1048576
                         RETURN_CODE,
                         PUTIN,
                         PUTOUT);

                SELECT(RETURN_CODE);
                  WHEN(0)  PUT SKIP EDIT (
                          'SORT COMPLETE RETURN_CODE 0') (A);
                  WHEN(16) PUT SKIP EDIT (
                          'SORT FAILED, RETURN_CODE 16') (A);
                  WHEN(20) PUT SKIP EDIT (
                          'SORT MESSAGE DATASET MISSING ') (A);
                  OTHER    PUT SKIP EDIT (
```

```
                       'INVALID RETURN_CODE = ', RETURN_CODE)
                           (A,F(2));
               END /* SELECT */;

      CALL PLIRETC(RETURN_CODE);
       /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
       PUTIN: PROC RETURNS (CHAR(80) VARYING);
         /*OUTPUT HANDLING ROUTINE*/
         /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE
           WHEN USING VARYING LENGTH RECORDS*/
           DCL STRING CHAR(80) VAR;

           ON ENDFILE(SYSIN) BEGIN;
                PUT SKIP EDIT ('END OF INPUT')(A);
                CALL PLIRETC(8);
                STRING = '';
                GOTO ENDPUT;
                END;

           GET EDIT(STRING)(A(80));
           I=INDEX(STRING||'  ','  ')-1;/*RESET LENGTH OF THE*/
           STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO */
                                        /* LENGTH OF TEXT IN */
                                        /* EACH INPUT RECORD.*/
```

```
           PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
           CALL PLIRETC(12);
  ENDPUT:  RETURN(STRING);
           END;
  PUTOUT:PROC(STRING);
          /*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
           DCL STRING CHAR (*);
            /*NOTE THAT FOR VARYING RECORDS THE STRING
              PARAMETER FOR THE OUTPUT-HANDLING ROUTINE
              SHOULD BE DECLARED ADJUSTABLE BUT CANNOT BE
              DECLARED VARYING*/
           PUT SKIP EDIT(STRING)(A);  /*PRINT THE SORTED DATA*/
           CALL PLIRETC(4);
           END;  /*ENDS PUTOUT*/
          END;
 /*
 //GO.SYSIN DD *
 003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
 002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
 003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
 059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
 073872HOME TAVERN, WESTLEIGH
 000931FOREST, IVER, BUCKS
 /*
 //GO.SYSPRINT DD SYSOUT=A
 //GO.SORTOUT DD SYSOUT=A
 //GO.SYSOUT DD SYSOUT=A
 //GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
 //GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
 //*
```

Sorting varying-length records using input and output handling routines (continued)

第 17 章 C との ILC

この章では、 PL/I と C の間の InterLanguage Communication (ILC) のいくつかの側面について説明する。 ここでは、両方の言語に共通する多くのデータ型を使用する方法について例を交えて説明しています。 また、その例は、C を呼び出したり C に呼び出されたりする PL/I コードを作成するときに役立ちます。

同等なデータ・タイプ

このトピックは、C と PL/I に共通する同等なデータ・タイプの一覧です。

表 *37. C* と *PL/I* の同等なタイプ

C のタイプ	**適合する PL/I タイプ**
char[...]	char (...) varyingz
wchar[...]	wchar (...) varyingz
符号付き文字	fixed bin(7)
符号なし char	unsigned fixed bin(8)
短い	fixed bin(15)
符号なし short	unsigned fixed bin(16)
int	fixed bin(31)
unsigned int	unsigned fixed bin(32)
長い長い	fixed bin(63)
符号なしロング・ロング	unsigned fixed bin(64)
浮動小数点	float bin(21)
倍精度	float bin(53)
long double	浮動ビン (p) (p> = 54)
enum	ordinal
typedef	define alias
struct	define struct
union	define union

© Copyright IBM Corp. 1999, 2025

表 37. C と PL/I の同等なタイプ (続き)

C のタイプ	適合する PL/I タイプ
struct *	handle

単純なタイプの一致

次の例は、C ヘッダー・ファイル time.h から抜粋した、単純な time_t の typedef の変換を示しています。

単純なタイプの一致

```
typedef long time_t;

define alias time_t fixed bin(31);
```

struct タイプの一致

次の例は、C ヘッダー・ファイル time.h から抜粋した、単純な tm の struct の変換を示しています。

struct タイプの一致の例

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};

define structure
  1 tm
     ,2 tm_sec   fixed bin(31)
     ,2 tm_min   fixed bin(31)
     ,2 tm_hour  fixed bin(31)
     ,2 tm_mday  fixed bin(31)
     ,2 tm_mon   fixed bin(31)
     ,2 tm_year  fixed bin(31)
     ,2 tm_wday  fixed bin(31)
     ,2 tm_yday  fixed bin(31)
     ,2 tm_isdst fixed bin(31)
   ;
```

enum タイプの一致

以下の例は、C ヘッダー・ファイル stdio.h にある単純な enum __device_t を変換する方法を示したものです。

列挙型の等価性のサンプル：

```
typedef enum {
   __disk       =  0,
   __terminal   =  1,
   __printer    =  2,
   __tape       =  3,
   __tdq        =  5,
   __dummy      =  6,
   __memory     =  8,
   __hfs        =  9,
   __hiperspace = 10
} __device_t;

define ordinal __device_t (
        __disk       value(0)
      , __terminal   value(1)
```

```
          , __printer    value(2)
          , __tape       value(3)
          , __tdq        value(4)
          , __dummy      value(5)
          , __memory     value(8)
          , __zfs       value(9)
          , __hiperspace value(10)
       );
```

ファイル・タイプの一致

C のファイル宣言はプラットフォームによって異なりますが、 通常は次のように始まります。

FILE タイプの C 宣言の開始

```
    struct __file {
        unsigned char  *__bufPtr;
     ... } FILE;
```

必要なものはファイルのポインター (トークン) です。 そのため、この変換は次のようにうまく解決できます。

PL/I C ファイルに相当

```
    define struct    1 file;
    define alias     file_Handle  handle file;
```

C 関数を使用する

C 関数 **fopen** と **fread** を使用して、ファイルを読み取ってフォーマット付き 16 進数としてダンプするプログラムをプログラマーが作成するとします。

このプログラムのコードは、次のように簡単なものです。

fopen と fread を使ってファイルをダンプするサンプルコード：

```
  filedump:
    proc(fn) options(noexecops main);

    dcl fn             char(*) var;

    %include filedump;

    file = fopen( fn, 'rb' );

    if file = sysnull() then
      do;
        display( 'file could not be opened' );
        return;
      end;

    do forever;
      unspec(buffer) = ''b;

      read_In = fread( addr(buffer), 1, stg(buffer), file );

      if read_In = 0 then
        leave;

      display(    heximage(addr(buffer),16,' ')  ||  '  '
               || translate(buffer,(32)'.',unprintable)  );

      if read_In < stg(buffer) then
        leave;
    end;

    call fclose( file );
  end filedump;
```

INCLUDE 、 filedump ：

ファイルダンプ・プログラムの宣言

```
    define struct      1 file;
    define alias       file_Handle  handle file;

    define alias       size_t unsigned fixed bin(32);
    define alias       int signed fixed bin(31);

    dcl file           type(file_Handle);
    dcl read_In        fixed bin(31);
    dcl buffer         char(16);

    dcl unprintable    char(32) value( substr(collate(),1,32) );
```

一致する単純パラメーター・タイプ

C 関数の宣言の変換は間違いを起こしやすいかもしれません。 例えば、344 ページの『#unique_587/unique_587_Connect_42_ilc012』に示されている C 関数 **fread** の宣言は、344 ページの『#unique_587/unique_587_Connect_42_ilc013』 に示されている宣言に変換できます。

fread の C 宣言

```
  size_t   fread(  void *,
                   size_t,
                   size_t,
                   FILE *);
```

fread の誤った宣言 (その 1)

```
   dcl fread     ext
                 entry( pointer,
                        type size_t,
                        type size_t,
                        type file_Handle )
                 returns( type size_t );
```

プラットフォームによっては、C の名前に大文字小文字の区別があるため、 これではリンクが正常に行われない場合があります。 この種のリンカーの問題を防ぐために最適な方法は、external 属性の拡張形式を使用して、大文字小文字混合の名前を指定することです。 したがって、例えば **fread** の宣言は次のように改良できます。

fread の誤った宣言 (その 2)

```
   dcl fread     ext('fread')
                 entry( pointer,
                        type size_t,
                        type size_t,
                        type file_Handle )
                 returns( type size_t );
```

ただし、これは正しく実行されません。 PL/I パラメーターはデフォルトで byaddr ですが、C パラメーターはデフォルトで byvalue であるためです。 この問題を修正するには、byvalue 属性をパラメーターに追加します。

fread の誤った宣言 (その 3)

```
   dcl fread     ext('fread')
                 entry( pointer byvalue,
                        type size_t byvalue,
                        type size_t byvalue,
                        type file_Handle byvalue )
                 returns( type size_t );
```

345 ページの『#unique_587/unique_587_Connect_42_ilc016』第 4 のパラメーター（テンポラリ _temp5 のアドレス）が関数 **fread** に渡され、そのアドレスの整数の中にリターンコードが格納される。 これは、returns に byaddr 属性が適用されている 場合に値を戻す方法の規則で、PL/I はデフォルトでこの規則を使用します。

RETURNS BYADDR に対して生成されるコード

```
      *      read_In = fread( addr(buffer), 1, stg(buffer), file );
      *
             L     r4,FILE(,r13,176)
             L     r1,fread(,r5,12)
             LA    r2,_temp5(,r13,420)
             LA    r8,BUFFER(,r13,184)
             L     r15,&EPA_&WSA(,r1,8)
             L     r0,&EPA_&WSA(,r1,12)
             ST    r0,_CEECAA_(,r12,500)
             LA    r1,#MX_TEMP1(,r13,152)
             ST    r8,#MX_TEMP1(,r13,152)
             LA    r8,1
             ST    r8,#MX_TEMP1(,r13,156)
             ST    r7,#MX_TEMP1(,r13,160)
             ST    r4,#MX_TEMP1(,r13,164)
             ST    r2,#MX_TEMP1(,r13,168)
             BALR  r14,r15
             L     r0,_temp5(,r13,420)
             ST    r0,READ_IN(,r13,180)
```

C の戻り値は byvalue なので、これは正しく実行されません。 このエラーを修正するには、byvalue 属性をもう 1 つ追加します。

fread の正しい宣言

```
   dcl fread     ext('fread')
                 entry( pointer byvalue,
                        type size_t byvalue,
                        type size_t byvalue,
                        type file_Handle byvalue )
                 returns( type size_t byvalue );
```

345 ページの『#unique_587/unique_587_Connect_42_ilc018』それ以上のアドレスは渡されず、戻り値は単にレジスタ 15 に戻される。

RETURNS BYVALUE に対して生成されるコード

```
      *      read_In = fread( addr(buffer), 1, stg(buffer), file );
      *
             L     r2,FILE(,r13,176)
             L     r1,fread(,r5,12)
             LA    r7,BUFFER(,r13,184)
             L     r15,&EPA_&WSA(,r1,8)
             L     r0,&EPA_&WSA(,r1,12)
             ST    r0,_CEECAA_(,r12,500)
             LA    r1,#MX_TEMP1(,r13,152)
             ST    r7,#MX_TEMP1(,r13,152)
             LA    r7,1
             ST    r7,#MX_TEMP1(,r13,156)
             ST    r4,#MX_TEMP1(,r13,160)
             ST    r2,#MX_TEMP1(,r13,164)
             BALR  r14,r15
             LR    r0,r15
             ST    r0,READ_IN(,r13,180)
```

一致するストリング・パラメーター・タイプ

現在、**fread** は正しく変換されるため、プログラマーがこの変換を **fopen** に対して試すとします。

fopen の最初の誤った宣言：

```
  dcl fopen    ext('fopen')
               entry( char(*) varyingz byvalue,
                      char(*) varyingz byvalue )
               returns( byvalue type file_handle );
```

しかし、C にストリングはなく、ポインターのみがあります。 そのポインターは値 (byvalue) で渡されます。 そのため、ストリングは参照 (byaddr) によって渡されなければなりません。

fopen の 2 つ目の誤った宣言：

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
             returns( byvalue type file_handle );
```

しかし、PL/I はストリングとともに記述子を渡しますが、C は記述子を解釈できないので、記述子は抑止する必要があります。 この宣言を抑止するには、options(nodescriptor) を宣言に追加します。

fopen の宣言が正しい：

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
             returns( byvalue type file_handle )
             options ( nodescriptor );
```

これは正しく機能しますが最適ではありません (パラメーターが入力専用であるため)。 パラメーターが定数の場合、nonassignable 属性を指定すれば、 コピーが作成されて渡されることはありません。 したがって、**fopen** の宣言の最適な変換は次のとおりです。

fopen の最適で正しい宣言：

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz nonasgn byaddr,
                    char(*) varyingz nonasgn byaddr )
             returns( byvalue type file_handle )
             options ( nodescriptor );
```

この時点で、**fclose** 関数の宣言には、おそらく returns 指定の optional 属性を除けば、意外なものはほとんどありません。 この属性を指定すれば、CALL ステートメントを使用して **fclose** 関数を呼び出すことができるようになり、戻りコードを処理する必要がなくなります。 ただし、ファイルが出力ファイルだった場合は、**fclose** の戻りコードを必ず確認する必要があることに注意してください。 これは、ファイルがクローズされるときにのみ最終バッファーが書き出されるが、その書き出しは、スペース不足が原因で失敗する可能性があるためです。

fclose 宣言：

```
dcl fclose   ext('fclose')
             entry( type file_handle byvalue )
             returns( optional type int byvalue )
             options ( nodescriptor );
```

これで、z/OS UNIX において次のコマンドでプログラムをコンパイルして実行できるようになりました。

filedump のコンパイルと実行のためのコマンド：

```
pli -qdisplay=std filedump.pli

filedump filedump.pli
```

この結果、次の出力が生成されます。

実行中の filedump の出力：

```
15408689 938584A4 94977A40 97999683  . filedump: proc
4D86955D 409697A3 899695A2 4D959685  (fn) options(noe
A7858396 97A24094 8189955D 5E151540  xecops main);..
```

ENTRY を戻す関数

C のクイック・ソート関数 **qsort** は、比較ルーチンを使用します。 例えば、整数の配列をソートするには、次の関数 (byvalue 属性が 2 回使用される) を使用できます。

C qsort 関数の比較ルーチンの例

```
comp2:
  proc( key, element )
  returns( fixed bin(31) byvalue );

  dcl (key, element) pointer byvalue;
  dcl word based fixed bin(31);

  select;
    when( key->word < element->word )
      return( -1 );
    when( key->word = element->word )
      return( 0 );
    otherwise
      return( +1 );
  end;
end;
```

次のコード・フラグメントに示すように、C **qsort** 関数とこの比較ルーチンを組み合わせて使用して、整数の配列をソートできます。

C qsort 関数を使用するためのコード例

```
  dcl a(1:4) fixed bin(31) init(19,17,13,11);

  put skip data( a );

  call qsort( addr(a), dim(a), stg(a(1)), comp2 );

  put skip data( a );
```

しかし、C の関数ポインターは PL/I ENTRY 変数とは違うので、C の **qsort** 関数は単純に次のように宣言してはならない：

qsort の誤った宣言

```
dcl qsort    ext('qsort')
             entry( pointer,
                    fixed bin(31),
                    fixed bin(31),
                    entry returns( byvalue fixed bin(31) )
                  )
             options( byvalue nodescriptor );
```

PL/I ENTRY 変数がネスト関数を指す場合があることに注意してください (このため、エントリー・ポイント・アドレスだけでなく逆チェーン・アドレスが必要です)。 しかし、C 関数ポインターが指す先は非ネスト関数だけに限定されるので、PL/I ENTRY 変数と C 関数ポインターはストレージ容量を使用することすらありません。

ただし、C 関数ポインターは PL/I タイプ LIMITED ENTRY と同等です。 このため、C **qsort** 関数は次のように宣言できます。

qsort の正しい宣言

```
dcl qsort    ext('qsort')
             entry( pointer,
                    fixed bin(31),
                    fixed bin(31),
                    limited entry
                    returns( byvalue fixed bin(31) )
                  )
             options( byvalue nodescriptor );
```

リンケージ

z/OS 上では、リンケージに関して次の 2 つの重要な事実があります。

- IBM C、JAVA、および Enterprise PL/I は、デフォルトで同じリンケージを使用します。
- このリンケージはシステム・リンケージではありません。

すべてのパラメーターが byaddr である従来の PL/I アプリケーションでは、関数にデフォルト・リンケージがある場合に生成されるコードと、関数にシステム・リンケージがある場合に生成されるコードとの違いは、通常は問題になりません。 ただし、パラメーターが byvalue である場合 (C および JAVA では これが通常) は、この違いによってコードが使えなくなる可能性があります。

実際には、パラメーターが byaddr である場合の違いは ほんの小さなものです。 348 ページの『#unique_590/unique_590_Connect_42_ilc027』 では、デフォルト・リンケージを使用する 関数に対して生成されたコードと、 システム・リンケージを使用する関数に対して生成されたコードの違いは、 システム・リンケージ呼び出しの最終パラメーターの高位ビットがオン になっていることだけです。

この違いは多くのプログラムには透過的です。

パラメーターが BYADDR である場合のコード

```
      dcl dfta  ext entry( fixed bin(31) byaddr
                         ,fixed bin(31) byaddr );
      dcl sysa  ext entry( fixed bin(31) byaddr
                         ,fixed bin(31) byaddr )
                 options( linkage(system) );

      *      call dfta( n, m );
      *
                LA    r0,M(,r13,172)
                LA    r2,N(,r13,168)
                L     r15,=V(DFTV)(,r3,126)
                LA    r1,#MX_TEMP1(,r13,152)
                ST    r2,#MX_TEMP1(,r13,152)
                ST    r0,#MX_TEMP1(,r13,156)
                BALR  r14,r15
      *
      *      call sysa( n, m );
      *
                LA    r0,M(,r13,172)
                LA    r2,N(,r13,168)
                O     r0,=X'80000000'
                L     r15,=V(SYSV)(,r3,130)
                LA    r1,#MX_TEMP1(,r13,152)
                ST    r2,#MX_TEMP1(,r13,152)
                ST    r0,#MX_TEMP1(,r13,156)
                BALR  r14,r15
```

ただし、パラメーターが byaddr でなく byvalue の 場合は、大きな違いがあります。 348 ページの『#unique_590/unique_590_Connect_42_ilc028』 で、デフォルト・リンケージを使用する関数 の場合は、レジスター 1 は渡される整数の値を指します。一方、 システム・リンケージを使用する関数の場合は、 レジスター 1 はこれらの値のアドレスを指します。

この違いは多くのプログラムには透過的ではありません。

パラメーターが BYVALUE である場合のコード

```
      dcl dftv  ext entry( fixed bin(31) byvalue
                         ,fixed bin(31) byvalue );
      dcl sysv  ext entry( fixed bin(31) byvalue
                         ,fixed bin(31) byvalue )
                  options( linkage(system) );

      *      call dftv( n, m );
      *
                L     r2,N(,r13,168)
                L     r0,M(,r13,172)
                L     r15,=V(DFTV)(,r3,174)
                LA    r1,#MX_TEMP1(,r13,152)
                ST    r2,#MX_TEMP1(,r13,152)
                ST    r0,#MX_TEMP1(,r13,156)
                BALR  r14,r15
      *
      *      call sysv( n, m );
      *
                L     r1,N(,r13,168)
                L     r0,M(,r13,172)
                ST    r0,#wtemp_1(,r13,176)
                LA    r0,#wtemp_1(,r13,176)
                ST    r1,#wtemp_2(,r13,180)
                LA    r2,#wtemp_2(,r13,180)
                O     r0,=X'80000000'
```

```
L     r15,=V(SYSV)(,r3,178)
LA    r1,#MX_TEMP1(,r13,152)
ST    r2,#MX_TEMP1(,r13,152)
ST    r0,#MX_TEMP1(,r13,156)
BALR  r14,r15
```

出力および入力の共有

このセクションでは、STDSYS オプションを指定することによって入出力を C プログラムと共用する方法について説明します。

STDSYS オプションの詳細および制限については、以下の情報を参照してください。

- 3 ページの『第 1 章 コンパイラー・オプションと機能の使用』の 88 ページの『STDSYS』。
- 「*Enterprise PL/I for z/OS* コンパイラーおよびランタイム 移行ガイド」の『印刷不能な文字が含まれた STREAM 入出力』。

出力の共用

C プログラムと SYSPRINT を共有したい場合は、 PL/I のコードを STDSYS オプションでコンパイルする必要がある。

デフォルトでは、 DISPLAY ステートメントは、その出力を表示するために WTO を使用します。 DISPLAY(STD) コンパイラー・オプションを指定すると、DISPLAY 文は C の **puts** 関数を使用して出力を表示する。 これは、z/OS UNIX 環境では特に便利です。

MVS バッチ、TSO バッチ、IMS バッチ、および IMS インタラクティブで出力を共有するための標準 C ストリームの動作は、以下のとおりです。

1. stdout はまず DD へ。。SYSPRINT
2. DD:SYSPRINT が存在しない場合、 stdout は DD:SYSTERM を探す。
3. DD:SYSTERM も DD:SYSERR も存在しない場合、ライブラリは DD SYSPRINT を使って sysout=*データセットをオープンし、 stdout ストリームをそこに送る。

入力の共有

SYSIN を C プログラムと共用するには、STDSYS オプションを指定してアプリケーションをコンパイルし、入力ストリーム・ファイルとして SYSIN をオープンする必要があります。 C ライブラリーで予約されている DD 名は使用しないでください。

また、 SYSIN を前のジョブ・ステップの一時データ・セットにコピーし、それを PL/I ジョブ・ステップの SYSIN として使用することもできます。これは、インストリーム・ファイルに割り振られていない場合に共用することができます。

入力ストリーム・ファイルは、JCL 内の SYSIN DD への割り振り時に一度だけ開くことができます。

MVS バッチ、TSO バッチ、IMS バッチ、および IMS インタラクティブで入力を共有するための標準 C ストリームの動作は、以下のとおりです。

1. stdin は DD:SYSIN に送られます。
2. DD:SYSIN が存在しない場合、stdin からの読み取り操作はすべて失敗します。

ATTACH

PL/I プログラムの ATTACH ステートメントは、基礎となる pthread ライブラリーを使用してスレッド管理を実行する。 ATTACH 文が発行されると、pthread ライブラリは標準 C ストリーム・ファイルを割り当てようとする。

ATTACH ステートメントが発行された後、SYSIN が JCL で SYSIN DD に割り当てられているインストリーム・ファイル（例えば、//'SYSIN DD *' や //'SYSIN DD DATA'）である場合、アプリケーションが SYSIN を開こうとすると失敗する。 SYSIN OPEN FAILED エラーが出されます。

C 標準ストリームのリダイレクト

PL/I と C が混在するアプリケーション、およびマルチスレッド・アプリケーションでは、すべて STDSYS オプションを使用する。

NOSTDSYS オプションを付けてプログラムをコンパイルすると、 SYSIN DD 名と SYSPRINT DD 名を C で使用する際に衝突する可能性があります。

例えば、プログラムに ATTACH ステートメントが含まれていると、 C 環境が直接始動します。 C 環境を開始すると、 PL/I アプリケーション・プログラムに関係なく、 SYSIN および SYSPRINT ストリームがオープンおよびクローズされます。 場合によっては、SYSIN が開けなくなったり、SYSPRINT データ・セットが上書きされたりすることがあります。

サマリー

このトピックでは、このセクションで説明されているキーポイントを要約します。

- C は大/小文字の区別がある。
- パラメータは BYVALUE であるべきだ。
- 戻り値は BYVALUE でなければならない。
- 文字列パラメータは BYADDR でなければならない。
- 配列と構造体も BYADDR であるべきだ。
- 記述子を渡してはならない。
- 入力専用の文字列は、 NONASSIGNABLE。
- C の関数ポインターは LIMITED ENTRYs にマップされる。
- IBM C コンパイラーと IBM PL/I コンパイラーは、同じデフォルト・リンケージを使用する (このことが問題になる)。

第 18 章 Java とのインターフェース

この章では、Java™ と Java Native Interface (JNI) の概要を示し、JNI を PL/I と組み合わせて使用する場合の利点について説明します。

この章では、シンプルな Java アプリケーション（PL/I ）について説明し、2 つの言語の互換性に関する情報を提供する。 Java-PL/I サンプルアプリケーションのビルドと実行方法に関する説明は、 z/OS の z/OS UNIX システム・サービス 環境で作業を行うことを前提としています。

PL/I から Java と通信する前に、 z/OS システムに Java がインストールされている必要があります。 z/OS Java 環境のセットアップ方法の詳細については、該当のシステム管理者に問い合わせてください。 詳細については、 IBM Java のサポートウェブサイト https://www.ibm.com/support/pages/java-sdk-products-zos をご参照ください。

これらのサンプルプログラムは IBM SDK for z/OS Java Technology Edition バージョン 8.0 および IBMSemeru Runtime Certified Edition でコンパイルされています。 z/OS バージョン 11.0 でテストされています。 z/OS UNIX システム・サービス 環境の Java のレベルを調べるには、コマンドラインから次のコマンドを入力する：

```
java -version
```

次の例にあるようにアクティブ Java バージョンが表示されます。

```
java version "11.0.15" 2022-04-19
IBM Semeru Runtime Certified Edition for z/OS 11.0.15.0 (build 11.0.15+10)
IBM J9 VM 11.0.15.0 (build z/OS-Release-11.0.15.0-b01, JRE 11 z/OS s390x-64-Bit
```

IBM Java バージョン、31 ビットと 64 ビットが利用可能です。 8.0 IBM Semeru は 64 ビットのみ。 あなたの環境で適切なライブラリにアクセスできることを確認する。

64 ビット Java との通信では、64 ビット PL/I。

64 ビット Java と 31 ビット PL/I 間の通信では、64 ビット Java プログラムが 64 ビット PL/I インタフェース・ルーチンを呼び出し、そのインタフェース・ルーチンが 31 ビット PL/I ルーチンを呼び出せるようにすることを推奨する。 PL/I 31 ビットおよび 64 ビットの相互運用性を確保するには、 Enterprise PL/I for z/OS 6.1 コンパイラが必要です。 この機能の詳細については、 373 ページの『第 19 章 AMODE 31 と AMODE 64 コードを混ぜる』 を参照してください。

Java Native Interface (JNI)

Java は、高い堅牢性、拡張性、信頼性を備えた最新のエンタープライズ・アプリケーションを構築するためのプラットフォームを提供する。 開発者は Java の API、ライブラリ、フレームワークを使用することで、バッチアプリやトランザクションアプリ、マイクロサービスなどを構築できる。 Java Native Interface (JNI) は、 ネイティブ・プログラミング言語に対する Java インターフェース で、Java Development Kit の一部です。

JNI を使用するプログラムを書くことは、あなたのコードが多くのプラットフォームでポータブルであることを保証する一つの方法である。

JNI によって、Java 仮想マシン (JVM) 内で稼働する Java コードは、PL/I などの他言語で書かれたアプリケーションやライブラリーと相互運用できます。 さらに、*Invocation API* を使用すれば、Java 仮想マシンを PL/I アプリケーションに組み込むことができます。

Java は完成度の高いプログラム言語ですが、状況によっては 他のプログラミング言語で書かれたプログラムを呼び出す必要も生じます。 Java からこれを行うには、ネイティブ言語に対するメソッド呼び出し (ネイティブ・メソッド と呼ばれる) を使用します。

ネイティブ・メソッドを使用する理由をいくつか示します。

- アプリケーションのニーズを満たす Java クラス・ライブラリーには ない特殊な機能がネイティブ言語に備わっている。

© Copyright IBM Corp. 1999, 2025

- ネイティブ言語で書かれたアプリケーションが既に多数存在し、 Java アプリケーションからこれらにアクセスできるようにしたい。
- ネイティブ言語で一連の複雑な計算を集中的にインプリメントし、これらの関数を Java アプリケーションから呼び出したい。
- ユーザーまたはプログラマーがネイティブ言語の幅広いスキル を持っていて、この利点を失いたくない。

JNI を使用したプログラミングでは、ネイティブ・メソッドを使用して多種多様な操作を実行できます。

- ネイティブ・メソッドでは、Java メソッドでオブジェクトが使用される方法と同じ方法で Java オブジェクトを使用できます。
- ネイティブ・メソッドでは、Java オブジェクト (配列やストリングなど) を作成して検査し、そのオブジェクトを使用してタスクを実行できます。
- ネイティブ・メソッドでは、Java アプリケーション・コードによって作成されたオブジェクトを検査して使用できます。
- ネイティブ・メソッドでは、ネイティブ・メソッドで作成されたりネイティブ・メソッドに渡されたりした Java オブジェクトを更新できます。 更新されたオブジェクトは、Java アプリケーションで使用できるようにすることが可能です。

最後に、ネイティブ・メソッドは Java プログラミング・フレームワーク に既に組み込まれている機能を利用して、既存の Java メソッドを 呼び出すことも簡単にできます。 このように、アプリケーションのネイティブ言語側と Java 側の 両方で Java オブジェクトを作成、更新、および使用でき、 さらにこれらのオブジェクトを相互間で共用できます。

Java では、直接バイト・バッファを使用することで、ネイティブ・アプリケーションが所有するメモリに格納されたデータにアクセスできる。 このオフヒープ（Java のヒープから外れた）メモリは、言語境界を越えてデータをマーシャリングする 1 つの方法であり、パフォーマンスに役立つ。 より詳細な情報については、 IBM Java マニュアルを参照してください。

Java からの PL/I プログラムの呼び出し

Java プログラムから PL/I プログラムを呼び出す場合、PL/I で開いたファイルはすべて、PL/I が最終的に Java に制御を返す前に、明示的に PL/I で閉じる必要があります。

同様に、 PL/I プログラムでフェッチされたモジュールはすべて、解放する必要があります。

JNI サンプル・プログラム #1 -「Hello World」

最初のサンプル・プログラムは、「Hello World!」の別のバリエーションです。 "Hello World!" プログラムには、1 つの Java クラス `callingPLI.java` があります。 PL/I で作成されたネイティブ・メソッドは、`hiFromPLI.pli` に含まれています。

このサンプル・プログラムを作成するための手順を簡単に概説します。

1. ネイティブ・メソッドを含むクラスを定義し、ネイティブ・ロード・ライブラリー をロードし、ネイティブ・メソッドを呼び出す Java プログラムを作成する。
2. Java プログラムをコンパイルして Java クラスを作成する。
3. ネイティブ・メソッドを実装し、「Hello!」を表示する PL/I プログラムを作成します。 text.
4. PL/I プログラムをコンパイルしてリンクする。
5. PL/I プログラム内のネイティブ・メソッドを呼び出す Java プログラムを実行する。

ステップ 1: Java プログラムの作成

手順

1. ネイティブ・メソッドを宣言します。

Java メソッドであるかネイティブ・メソッドであるかに関係なく、 メソッドはすべて Java クラス内で宣言する必要があります。 Java メソッドとネイティブ・メソッドの宣言の違いは、キーワード **native** だけです。 **native** キーワードは、このメソッドのインプリメンテーション のある場所が、プログラムの実行時にロードされるネイティブ・ライブラリー内 であることを Java に指示します。 ネイティブ・メソッドは次のように宣言できます。

```
public native void callToPLI();
```

上記のステートメントの中で、void はこのネイティブ・メソッド呼び出し から予期される戻り値がないことを示しています。 メソッド名 callToPLI( ) の空括弧は、ネイティブ・メソッドの呼び出し時に渡すパラメーターがないことを示しています。

2. ネイティブ・ライブラリーが実行時にロードされるようにネイティブ・ライブラリーをロードします。

   以下の Java ステートメントを使用すれば、ネイティブ・ライブラリーをロードできます。

   ```
   static {
     System.loadLibrary("hiFromPLI");
   }
   ```

   上記のステートメントでは、ネイティブ・ライブラリーを検索して ロードするために、Java システム・メソッド System.loadLibrary(...) が 呼び出されています。 PL/I 共有ライブラリー libhiFromPLI.so は、 PL/I プログラムをコンパイルしてリンクするステップ中に作成されます。

3. Java main メソッドの作成

   callingPLI クラスには、クラスのインスタンスを生成して ネイティブ・メソッドを呼び出す main メソッドも含まれ ています。 main メソッドは callingPLI のインスタンスを生成し、 callToPLI() ネイティブ・メソッドを呼び出します。

   このトピックで前述した点をすべて含む callingPLI クラスの完全な定義は次のようになります。

   ```
   public class callingPLI {
     public native void callToPLI();
     static {
        System.loadLibrary("hiFromPLI");
     }
     public static void main(String[] argv) {
        callingPLI callPLI = new callingPLI();
        callPLI.callToPLI();
        System.out.println("And Hello from Java, too!");
     }
   }
   ```

ステップ 2: Java プログラムのコンパイル

手順

Java コンパイラーを使用して callingPLI クラスをコンパイルし、実行可能形式にします。
以下のコマンドを使用できます。

```
javac callingPLI.java
```

ステップ 3: PL/I プログラムの作成

ネイティブ・メソッドの PL/I インプリメンテーションは、 他の PL/I サブルーチンとほぼ同じようなものです。

便利な PL/I コンパイラー・オプション

サンプル・プログラムには、重要なコンパイラー・オプションを定義する一連の *PROCESS 。

- 31 ビット

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
```

- 64 ビット

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process LP(64) Default( ASCII IEEE );
```

次に、これらのオプションの概要と利点を説明します。

Extname(100)
Java スタイルの長い外部名を許可します。

Margins(1,100)
マージンを拡張して、Java スタイルの名前と ID が入る場所を 確保します。

Display(Std)
WTO を介さずに、"Hello World" テキストを stdout に書き込みます。 z/OS UNIX 環境では、WTO はユーザーによって意識されることはありません。

Dllinit
DLL の作成に必要な初期化コードをインクルードします。

Extrn(Short)
EXTRN は、参照対象の定数に対してのみ発行されます。

Default(ASCII IEEE)
ASCII は、 と のデータが JAVA で保持される形式である ASCII で保持されることを指定します。 CHARACTER PICTURE

IEEE は、 データが IEEE 形式（JAVA が保持する形式）で保持されることを指定する。 FLOAT

RENT
コードが静的変数に対して書き込みを行う場合でもコードを再入可能にします。

注 : 64 ビットの場合は、 LP(64)コンパイラー・オプションを追加する。 LP(64)は RENT を意味するため、RENT オプションを削除することもできる。

PL/I プロシージャー名とプロシージャー・ステートメントの正しい形式

PL/I プロシージャー名は、実行時に Java クラス・ローダーによって検出されるために、Java 命名規則に準拠している必要があります。 Java 命名体系は 3 つの部分で構成されます。 最初の部分は Java 環境に対してルーチンを識別し、2 番目の部分はネイティブ・メソッドを定義する Java クラスの名前、 3 番目の部分はネイティブ・メソッド自体の名前です。

以下に、サンプル・プログラムの PL/I プロシージャー名 `Java_callingPLI_callToPLI` の詳細を示します。

JAVA
動的ライブラリー内にあるネイティブ・メソッドはすべて、Java を最初に指定する必要があります。

_callingPLI
ネイティブ・メソッドを宣言する Java クラスの名前。

_callToPLI
ネイティブ・メソッド自体の名前。

注 : PL/I と C とでは、ネイティブ・メソッドのコーディングに重要な違いがあります。 JDK に付属する **javah** ツールは、C プログラムに必要となる外部参照の形式を生成します。 ネイティブ・メソッドを PL/I で記述し、PL/I 外部参照の命名について上記のルールに従っている場合、PL/I ネイティブ・メソッドに対して **javah** のステップを実行する必要はありません。

さらに、 PROCEDURE ステートメントの OPTIONS オプションで以下のオプションを指定しなければならない：

- FromAlien
- NoDescriptor
- ByValue

サンプル・プログラムの完全なプロシージャー・ステートメントは次のとおりです。

```
Java_callingPLI_callToPLI:
Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI インクルード・ファイル

Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、ibmzjni.inc です。これには、ibmzjnim.inc が含まれます。 これらの組み込みファイルは、次のステートメントを使用して組み込まれます。

```
 %include ibmzjni;
```

ibmzjni と ibmzjnim のインクルードファイルは、 PL/I SIBMZSAM データセットで提供されている。

完全な PL/I プロシージャー

念のため、ネイティブ・メソッドを定義している PL/I プログラム全体を以下に示す：

- 31 ビット

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
PliJava_Demo: Package Exports(*);

   Java_callingPLI_callToPLI:
   Proc( JNIEnv , MyJObject )
     External( "Java_callingPLI_callToPLI" )
     Options( FromAlien NoDescriptor ByValue );

   %include ibmzjni;
   Dcl myJObject Type jObject;

   Display('Hello from Enterprise PL/I!');

End;
```

- 64 ビット

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process LP(64) Default( ASCII IEEE );
PliJava_Demo: Package Exports(*);

   Java_callingPLI_callToPLI:
   Proc( JNIEnv , MyJObject )
     External( "Java_callingPLI_callToPLI" )
     Options( FromAlien NoDescriptor ByValue );

     %include ibmzjni;
     Dcl myJObject Type jObject;

     Display('Hello from Enterprise PL/I!');

End;
```

ステップ 4: PL/I プログラムのコンパイルとリンク

手順

1. 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

```
   pli -c hiFromPLI.pli
```

2. 出来上がった PL/I オブジェクトデッキを以下のコマンドで共有ライブラリにリンクする：
   - 31 ビット

```
c89 -o libhiFromPLI.so hiFromPLI.o
```

   - 64 ビット

```
c89 -Wl,"LP64,DLL" -o libhiFromPLI.so hiFromPLI.o
```

   lib PL/I そうしないと、 Java クラス・ローダーはそのライブラリを見つけることができません。

ステップ 5: サンプル・プログラムの実行

手順

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
   java callingPLI
```

サンプル・プログラムの出力は以下のとおりです。

```
   Hello from Enterprise PL/I!
   And Hello from Java, too!
```

最初の行は PL/I ネイティブ・メソッドから書き込まれたものです。 2 行目は、PL/I ネイティブ・メソッド呼び出しから戻った後の呼び出し元 Java クラスのものである。

共有ライブラリの場所が LIBPATH に追加されているか、 java.library.path プロパティで参照されていることを確認する。

JNI サンプル・プログラム #2 - ストリングを渡す

このサンプル・プログラムは、Java と PL/I の間で双方向にストリングの受け渡しを行います。

jPassString.java プログラムの完全なリストについては、357 ページの『#unique_607/unique_607_Connect_42_jsamp2』 を参照してください。 Java 部分には、1 つの Java クラス jPassString.java があります。 PL/I で作成されたネイティブ・メソッドは、 passString.pli に含まれています。 352 ページの『JNI サンプル・プログラム #1 - 「Hello World」』 に記載されている内容の多くが、このサンプル・プログラムにも当てはまります。 以下のトピックでは、このサンプル・プログラムの新しい面または異なる面についてのみ説明します。

ステップ 1: Java プログラムの作成

手順

1. ネイティブ・メソッドを宣言します。

```
public native void pliShowString();
```

2. ネイティブ・ライブラリーをロードします。

```
static {
  System.loadLibrary("passString");
}
```

3. Java main メソッドの作成

   jPassString クラスには、クラスのインスタンスを生成して ネイティブ・メソッドを呼び出す main メソッドも含まれ ています。 main メソッドは jPassString のインスタンスを生成し、 pliShowString() ネイティブ・メソッドを呼び出します。

   このサンプル・プログラムは、ストリングの入力をユーザーに促し、 コマンド行からその値を読み込みます。 この作業は、357 ページの『#unique_607/unique_607_Connect_42_jsamp2』 に 示す try/catch ステートメント内で行われます。

   Java サンプルプログラム#2 - 文字列の受け渡し

```
// Read a string, call PL/I, display new string upon return
import java.io.*;

public class jPassString{

 /* Field to hold Java string                          */
 String myString;

 /* Load the PL/I native library                       */
 static {
        System.loadLibrary("passString");
 }

 /* Declare the PL/I native method                     */
 public native void pliShowString();

 /* Main Java class                                    */
 public static void main(String[] arg) {

   System.out.println(" ");

   /* Instantiate Java class and initialize string     */
   jPassString myPassString = new jPassString();
   myPassString.myString = " ";

   /* Prompt user for a string                         */
   try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));

        /* Process until 'quit' received               */
        while (!myPassString.myString.equalsIgnoreCase("quit")) {
          System.out.println(
            "From Java: Enter a string or 'quit' to quit.");
          System.out.print("Java Prompt > ");
          /* Get string from command line              */
          myPassString.myString = in.readLine();
          if (!myPassString.myString.equalsIgnoreCase("quit"))
            {
             /* Call PL/I native method                */
             myPassString.pliShowString();
             /* Return from PL/I and display new string */
             System.out.println(" ");
             System.out.println(
                 "From Java: String set by PL/I is: "
               + myPassString.myString );
            }
        }
        }  catch (IOException e) {
        }
 }
}
```

ステップ 2: Java プログラムのコンパイル

手順

Java コンパイラーを使用して、Java コードをコンパイルします。
以下のコマンドを使用できます。

```
javac jPassString.java
```

ステップ 3: PL/I プログラムの作成

353 ページの『ステップ 3: PL/I プログラムの作成』で説明されている PL/I の「Hello World」サンプル・プログラムを作成する方法に関するすべての情報は、このプログラムにも当てはまります。

PL/I プロシージャー名とプロシージャー・ステートメントの正しい形式

このプログラムの PL/I プロシージャー名は Java_jPassString_pliShowString です。

サンプル・プログラムの完全なプロシージャー・ステートメントは次のとおりです。

```
Java_jPassString_pliShowString:
Proc( JNIEnv , myjobject )
  external( "Java_jPassString_pliShowString" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI インクルード・ファイル

Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、ibmzjni.inc です。これには、ibmzjnim.inc が含まれます。 これらの組み込みファイルは、次のステートメントを使用して組み込まれます。

```
 %include ibmzjni;
```

ibmzjni および ibmzjnim 組み込みファイルは、 PL/I SIBMZSAM データ・セットで提供されます。

完全な PL/I プロシージャー

完全な PL/I プログラムは、 サンプル・プログラム#2 - 文字列の受け渡し に示されている。 このサンプル PL/I プログラムは、JNI を介していくつかの呼び出しを行います。

入り口で、呼び出し側 Java オブジェクト myObject への参照が PL/I プロシージャーに渡されます。 PL/I プログラムはこの参照を使用して、呼び出し側からの情報を取得します。 最初の情報は、**GetObjectClass** JNI 関数を使用して検索される呼び出し側オブジェクトのクラスです。 このクラス値は、Java オブジェクト内の Java ストリング・フィールドの ID を取得するために、**GetFieldID** JNI 関数によって使用されます。 この Java フィールドは、フィールド名 myString、および JNI フィールド記述子 Ljava/lang/String; (これによりフィールドは Java ストリング・フィールドとみなされる) によってさらに詳細に識別されます。 その後、Java ストリング・フィールドの値が **GetObjectField** JNI 関数を使用して検索されます。 PL/I が Java ストリング値を使用するには、事前にこの値をアンパックして PL/I が解釈できる形式にする必要があります。 **GetStringUTFChars** JNI 関数は、 Java 文字列を PL/I varyingz 文字列に変換し、 PL/I プログラムによって表示される。

取得した Java ストリングを表示した後、PL/I プログラムは、呼び出し側 Java オブジェクト内のストリング・フィールドの更新に使用する PL/I ストリングの入力をユーザーに促します。 PL/I の文字列値は、**NewString** JNI 関数によって Java 文字列に変換される。 この新しい Java ストリングを使用して、**SetObjectField** JNI 関数によって呼び出し側 Java オブジェクト内のストリング・フィールドが更新されます。

PL/I プログラムが終了すると Java に制御が戻され、新しく更新された Java ストリングが Java プログラムによって表示されます。

PL/I **サンプルプログラム #2 - 文字列を渡す**

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
plijava_demo: package exports(*);

   Java_passString_pliShowString:
```

```
   Proc( JNIEnv , myJObject )
     external( "Java_jPassString_pliShowString" )
     Options( FromAlien NoDescriptor ByValue );

   %include ibmzjni;

   Dcl myBool                          Type jBoolean;
   Dcl myClazz                         Type jClass;
   Dcl myFID                           Type jFieldID;
   Dcl myJObject                       Type jObject;
   Dcl myJString                       Type jString;
   Dcl newJString                      Type jString;
   Dcl myID                            Char(9) Varz static init( 'myString' );
   Dcl mySig                           Char(18) Varz static
                                       init( 'Ljava/lang/String;' );
   Dcl pliStr                          Char(132) Varz Based(pliStrPtr);
   Dcl ( pliReply, pliReplyA )         Char(132) Varz;
   Dcl pliStrPtr                       Pointer;
   Dcl nullPtr                         Pointer;

   Display(' ');

   /* Get information about the calling Class */
   myClazz = GetObjectClass(JNIEnv, myJObject);

   /* Get Field ID for String field from Java */
   myFID = GetFieldID(JNIEnv, myClazz, myID, mySig );

   /* Get the Java String in the string field */
   myJString = GetObjectField(JNIEnv, myJObject, myFID );

   /* Convert the Java String to a PL/I string */
   pliStrPtr = GetStringUTFChars(JNIEnv, myJString, myBool );

   Display('From PLI: String retrieved from Java is: ' || pliStr );
   Display('From PLI: Enter a string to be returned to Java:' )
          reply(pliReplyA);

   /* Convert to EBCDIC */
   Call PLIEBCDIC( addr( pliReply ), addr( pliReplyA ), len( pliReplyA ) );

   /* Convert the new PL/I string to a Java String */
   newJString = NewString(JNIEnv, trim(pliReply), length(pliReply) );

   /* Change the Java String field to the new string value */
   nullPtr = SetObjectField(JNIEnv, myJObject, myFID, newJString);

 End;

end;
```

注 : この例は 31 ビットのものである。 64 ビットの場合は、 LP(64)コンパイラー・オプションを追加する。

ステップ 4: PL/I プログラムのコンパイルとリンク

手順

1. 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

   ```
   pli -c passString.pli
   ```

2. 出来上がった PL/I オブジェクトデッキを以下のコマンドで共有ライブラリにリンクする：
   - 31 ビット

     ```
     c89 -o libpassString.so passString.o
     ```

   - 64 ビット

     ```
     c89 -Wl,”LP64,DLL” -o libpassString.so passString.o
     ```

名前には必ず lib 接頭部を付けてください。 そうしないと、PL/I 共用ライブラリーも Java クラス・ローダーも名前を検出できません。

ステップ 5: サンプル・プログラムの実行

手順

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
java jPassString
```

サンプル・プログラムの出力 (Java と PL/I の両方からユーザー入力を求める要求を含む) は次のようになります。

```
>java jPassString

From Java: Enter a string or 'quit' to quit.
Java Prompt > A string entered in Java

From  PLI: String retrieved from Java is: A string entered in Java
From  PLI: Enter a string to be returned to Java:
A string entered in PL/I

From Java: String set by PL/I is: A string entered in PL/I
From Java: Enter a string or 'quit' to quit.
Java Prompt > quit
>
```

共有ライブラリの場所が LIBPATH に追加されているか、 java.library.path プロパティで参照されていることを確認する。

JNI サンプル・プログラム #3 - 整数の引き渡し

このサンプル・プログラムは、Java と PL/I の間で双方向に整数の受け渡しを行います。

jPassInt.java プログラムの完全なリストについては、361 ページの『#unique_613/unique_613_Connect_42_jsamp3』 を参照してください。 Java 部分には、1 つの Java クラス jPassInt.java があります。 PL/I で作成されたネイティブ・メソッドは、 passInt.pli に含まれています。 352 ページの『JNI サンプル・プログラム #1 - 「Hello World」』 に記載されている内容の多くが、このサンプル・プログラムにも当てはまります。 以下のトピックでは、このサンプル・プログラムの新しい面または異なる面についてのみ説明します。

ステップ 1: Java プログラムの作成

手順

1. ネイティブ・メソッドを宣言します。

   ```
   public native void pliShowInt();
   ```

2. ネイティブ・ライブラリーをロードします。

   ```
   static {
     System.loadLibrary("passInt");
   }
   ```

3. Java main メソッドの作成

   jPassInt クラスには、クラスのインスタンスを生成してネイティブ・メソッドを呼び出す main メソッドも含まれています。 main メソッドは jPassInt のインスタンスを生成し、pliShowInt() ネイティブ・メソッドを呼び出します。

このサンプル・プログラムは、整数の入力をユーザーに促し、コマンド行からその値を読み込みます。この作業は、361 ページの『#unique_613/unique_613_Connect_42_jsamp3』 に 示す try/catch ステートメント内で行われます。

Java サンプル・プログラム #3 - 整数を渡す

```
// Read an integer, call PL/I, display new integer upon return
import java.io.*;
import java.lang.*;

public class jPassInt{

 /* Fields to hold Java string and int                  */
 int myInt;
 String myString;

 /* Load the PL/I native library                        */
 static {
         System.loadLibrary("passInt");
 }

 /* Declare the PL/I native method                      */
 public native void pliShowInt();

 /* Main Java class                                     */
 public static void main(String[] arg) {

   System.out.println(" ");

   /* Instantiate Java class and initialize string      */
   jPassInt pInt = new jPassInt();
   pInt.myInt = 1024;
   pInt.myString = " ";

   /* Prompt user for an integer                        */
   try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));

        /* Process until 'quit' received                */
        while (!pInt.myString.equalsIgnoreCase("quit")) {
          System.out.println
            ("From Java: Enter an Integer or 'quit' to quit.");
          System.out.print("Java Prompt > ");
          /* Get string from command line               */
          pInt.myString = in.readLine();
          if (!pInt.myString.equalsIgnoreCase("quit"))
            {
             /* Set int to integer value of String      */
             pInt.myInt = Integer.parseInt( pInt.myString );
             /* Call PL/I native method                 */
             pInt.pliShowInt();
             /* Return from PL/I and display new string */
             System.out.println(" ");
             System.out.println
               ("From Java: Integer set by PL/I is: " + pInt.myInt );
            }
        }
        }  catch (IOException e) {
        }
 }

}
```

ステップ 2: Java プログラムのコンパイル

手順

Java コンパイラーを使用して、Java コードをコンパイルします。
以下のコマンドを使用できます。

```
javac jPassInt.java
```

ステップ 3: PL/I プログラムの作成

353 ページの『ステップ 3: PL/I プログラムの作成』で説明されている PL/I の「Hello World」サンプル・プログラムを作成する方法に関するすべての情報は、このプログラムにも当てはまります。

PL/I プロシージャー名とプロシージャー・ステートメントの正しい形式

このプログラムの PL/I プロシージャー名は Java_jPassInt_pliShowInt です。

サンプル・プログラムの完全なプロシージャー・ステートメントは次のとおりです。

```
Java_passNum_pliShowInt:
Proc( JNIEnv , myjobject )
  external( "Java_jPassInt_pliShowInt" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI インクルード・ファイル

Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、ibmzjni.inc です。これには、ibmzjnim.inc が含まれます。 これらの組み込みファイルは、次のステートメントを使用して組み込まれます。

```
 %include ibmzjni;
```

ibmzjni と ibmzjnim のインクルードファイルは、 PL/I SIBMZSAM データセットで提供されている。

完全な PL/I プロシージャー

完全な PL/I プログラムは、362 ページの『#unique_615/unique_615_Connect_42_psamp3』 に示してあります。 このサンプル PL/I プログラムは、JNI を介していくつかの呼び出しを 行います。

入り口で、呼び出し側 Java オブジェクト myObject への参照が PL/I プロシージャーに渡されます。 PL/I プログラムはこの参照を使用して、呼び出し側からの情報を取得します。 最初の情報は、**GetObjectClass** JNI 関数を使用して検索される呼び出し側オブジェクトのクラスです。 このクラス値は、Java オブジェクト内の Java 整数フィールドの ID を取得するために、**GetFieldID** JNI 関数によって使用されます。 この Java フィールドは、フィールド名 myInt および JNI フィールド記述子 I (これによりフィールドは整数フィールドとみなされる) によってさらに詳細に識別されます。 Java 整数フィールドの値は、 **GetIntField** JNI 関数によって取得され、 PL/I プログラムによって表示される。

取得した Java 整数を表示した後、PL/I プログラムは、呼び出し側 Java オブジェクト内の整数フィールドの更新に使用する PL/I 整数の入力をユーザーに促します。 PL/I の整数値は、**SetIntField** JNI 関数によって、呼び出し元の Java オブジェクトの整数フィールドを更新するために使われる。

PL/I プログラムが終了すると Java に制御が戻され、新しく更新された Java 整数が Java プログラムによって表示されます。

PL/I サンプル・プログラム #3 - 整数を渡す

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
plijava_demo: package exports(*);

 Java_passNum_pliShowInt:
 Proc( JNIEnv , myjobject )
   external( "Java_jPassInt_pliShowInt" )
   Options( FromAlien NoDescriptor ByValue );

  %include ibmzjni;

   Dcl myClazz          Type jClass;
   Dcl myFID            Type jFieldID;
   Dcl myJInt           Type jInt;
   dcl rtnJInt          Type jInt;
   Dcl myJObject        Type jObject;
```

```
   Dcl pliReply         Char(132) Varz;
   Dcl nullPtr          Pointer;

   Display('  ');

   /* Get information about the calling Class             */
   myClazz = GetObjectClass(JNIEnv, myJObject);

   /* Get Field ID for int field from Java                */
   myFID = GetFieldID(JNIEnv, myClazz, "myInt", "I");

   /* Get Integer value from Java                         */
   myJInt = GetIntField(JNIEnv, myJObject, myFID);

   display('From  PLI: Integer retrieved from Java is: ' || trim(myJInt) );
   display('From  PLI: Enter an integer to be returned to Java:' )
          reply(pliReply);

   rtnJInt = pliReply;

   /* Set Integer value in Java from PL/I                 */
   nullPtr = SetIntField(JNIEnv, myJObject, myFID, rtnJInt);

 End;

end;
```

注 : この例は 31 ビットのものである。 64 ビットの場合は、 LP(64)コンパイラー・オプションを追加する。

ステップ 4: PL/I プログラムのコンパイルとリンク

手順

1. 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

   ```
   pli -c passInt.pli
   ```

2. 出来上がった PL/I オブジェクトデッキを以下のコマンドで共有ライブラリにリンクする：

   - 31 ビット

     ```
     c89 -o libpassInt.so passInt.o
     ```

   - 64 ビット

     ```
     c89 -Wl,”LP64,DLL” -o libpassInt.so passInt.o
     ```

   名前には必ず lib 接頭部を付けてください。 そうしないと、PL/I 共用ライブラリーも Java クラス・ローダーも名前を検出できません。

ステップ 5: サンプル・プログラムの実行

手順

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
   java jPassInt
```

サンプル・プログラムの出力 (Java と PL/I の両方からユーザー入力を求める要求を含む) は次のようになります。

```
>java jPassInt

From Java: Enter an Integer or 'quit' to quit.
Java Prompt > 12345

From  PLI: Integer retrieved from Java is: 12345
```

```
From  PLI: Enter an integer to be returned to Java:
54321

From Java: Integer set by PL/I is: 54321
From Java: Enter an Integer or 'quit' to quit.
Java Prompt > quit
>
```

共有ライブラリの場所が LIBPATH に追加されているか、 `java.library.path` プロパティで参照されていることを確認する。

JNI サンプル・プログラム #4 - Java 呼び出し API

このサンプル・プログラムは、今までのサンプルと若干異なります。 このサンプルでは、最初に PL/I が Java 呼び出し API を使用して Java を呼び出して、組み込み Java 仮想マシン (JVM) を作成します。 次に、PL/I は Java メソッドを呼び出して、 Java メソッドが表示するストリングをそのメソッドに渡します。

PL/I サンプル・プログラムは `createJVM.pli` という名前で、このサンプル・プログラムによって呼び出される Java メソッドは `javaPart.java` に含まれています。

ステップ 1: Java プログラムの作成

このタスクについて

このサンプルは PL/I ネイティブ・メソッドを使用しないため、それを宣言する必要がありません。 その代わり、このサンプルの Java 部分には単純な Java メソッドが 1 つ含まれています。

手順

Java main メソッドの作成

`javaPart` クラスには、ステートメントが 1 つだけ含まれます。 このステートメントは、 Java から短い `'Hello World...'` を出力し、 PL/I プログラムから渡された文字列を追加する。 クラス全体は、364 ページの『#unique_619/unique_619_Connect_42_jsamp4』 に示されています。

Java サンプル・プログラム#4 - 文字列の受信と表示

```
// Receive a string from PL/I then display it after saying "Hello"
public class javaPart {
   public static void main(String[] args) {
      System.out.println("From Java  -  Hello World... " + args[0]);
   }
}
```

ステップ 2: Java プログラムのコンパイル

手順

Java コンパイラーを使用して、Java コードをコンパイルします。
以下のコマンドを使用できます。

```
javac javaPart.java
```

ステップ 3: PL/I プログラムの作成

353 ページの『ステップ 3: PL/I プログラムの作成』で説明されている PL/I の「Hello World」サンプル・プログラムを作成する方法に関する多くの情報は、このプログラムにも当てはまります。 しかし、このサンプルでは PL/I は Java を呼び出すため、考慮する必要のある点が他にもあります。

PL/I プロシージャー名とプロシージャー・ステートメントの正しい形式

このサンプルでは PL/I プログラムが Java を呼び出すため、PL/I プログラムは MAIN になります。 この PL/I プログラムの外部名は参照されないため、考慮する必要はありません。

サンプル・プログラムの完全なプロシージャー・ステートメントは次のとおりです。

```
createJVM: Proc Options(Main);
```

JNI インクルード・ファイル

Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、`ibmzjni.inc` です。これには、`ibmzjnim.inc` が含まれます。 このサンプルでは PL/I が Java を呼び出しているとしても、これらの組み込みファイルは必要です。 これらの組み込みファイルは、次のステートメントを使用して組み込まれます。

```
   %include ibmzjni;
```

`ibmzjni` と `ibmzjnim` のインクルードファイルは、 PL/I SIBMZSAM データセットで提供されている。

PL/I プログラムと Java ライブラリーとのリンク

注 : このセクションは 31 ビットにのみ適用される。

この PL/I サンプル・プログラムは Java を呼び出すため、Java ライブラリーにリンクする必要があります。 Java ライブラリーは XPLINK とリンクされているが、 PL/I モジュールはリンクされていない。 PL/I XPLINK ただし、変数を使用して、ランタイム・オプションを指定する必要がある。 PLIXOPT **XPLINK=ON** 変数 PLIXOPT は次のように宣言できる：

```
  Dcl PLIXOPT     Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );
```

PLIXOPT については *z/OS Language Environment Programming Guide*.

Java 呼び出し API の使用

この PL/I サンプル・プログラムは、Java 呼び出し API **JNI_CreateJavaVM** を呼び出して、独自の組み込み JVM を作成します。 この API には、366 ページの『#unique_621/unique_621_Connect_42_psamp4』に示されているような、セットアップおよび初期化を正しく行うための特定の構造体が必要です。

1. **JNI_GetDefaultJavaVMInitArgs** は、デフォルトの初期化オプションを取得するために呼び出される。
2. これらのデフォルト・オプションが、`java.class.path` 情報の追加により変更されます。
3. **JNI_CreateJavaVM** が組み込み JVM を作成するために呼び出されます。

完全な PL/I プログラム

31 ビット用と 64 ビット用の完全な PL/I プログラムをそれぞれ 366 ページの『#unique_621/unique_621_Connect_42_psamp4』 と 367 ページの『#unique_621/unique_621_Connect_42_table_jw4_fj2_mvb』 に示す。 サンプルの PL/I ・プログラムは、JNI を通じていくつかの呼び出しを行う。

このサンプルでは、Java オブジェクト (この場合は新しく作成された JVM) への参照は渡されませんが、代わりに **JNI_CreateJavaVM** API への呼び出しから戻されます。 PL/I プログラムはこの参照を使用して、JVM から情報を取得します。 最初の情報部分は、呼び出される Java メソッドが含まれたクラスです。 このクラスは、**FindClass** JNI 関数を使用して検出されます。 次に、クラス値が **GetStaticMethodID** JNI 関数によって使用され、呼び出される Java メソッドの ID が取得されます。

この Java メソッドを呼び出す前に、Java で認識されるフォーマットに PL/I ストリングを変換します。 PL/I プログラムでは、ストリングは ASCII フォーマットで保持されます。 Java ストリングは UTF フォー

マットで保管されます。 さらに、Java ストリングは、PL/I プログラマーが認識するような意味では真にストリングではありませんが、それ自体 Java クラスであり、メソッドを通じてのみ変更できます。 **NewStringUTF** JNI 関数を使用して Java ストリングを作成します。 この関数は、UTF に変換された PL/I ストリングを含む myJString という Java オブジェクトを返します。 次に、 **NewObjectArray** JNI 関数を呼び出して Java オブジェクト配列を作成し、 myJString オブジェクトへの参照を渡す。 この関数は、Java メソッドで表示されるストリングを含む Java オブジェクト配列に対する参照を返します。

これで、**CallStaticVoidMethod** JNI 関数で Java メソッドを呼び出せるようになりました。 また、このメソッドにより、その関数に渡されるストリングが表示されるようになります。 文字列を表示した後、PL/I ・プログラムは、 **DestroyJavaVM** JNI 関数を使用して組み込み JVM を破棄し、 PL/I ・プログラムが完了する。

完全なソースは 366 ページの『#unique_621/unique_621_Connect_42_psamp4』 と 367 ページの『#unique_621/unique_621_Connect_42_table_jw4_fj2_mvb』 に示されている。

PL/I サンプル・プログラム#4 - Java 呼び出し API (31 ビット) の呼び出し

```
*Process Margins( 1, 100 ) ;
*Process Display(STD) Rent;
*Process Default( ASCII ) Or('|');
createJVM: Proc Options(Main);

  %include ibmzjni;

  Dcl myJObjArray      Type jobjectArray;
  Dcl myClass          Type jclass;
  Dcl myMethodID       Type jmethodID;
  Dcl myJString        Type jstring;
  Dcl myRC             Fixed Bin(31) Init(0);
  Dcl myPLIStr         Char(50) Varz
                       Init('... a PLI string in a Java Virtual Machine!');
  Dcl OptStr1          char(1024) varz;
  Dcl OptStr2          char(1024) varz;
  Dcl myNull           Pointer;
  Dcl VM_Args          Like JavaVMInitArgs;
  Dcl myOptions        Like JavaVMOption;
  Dcl PLIXOPT          Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );

  Display('From PL/I - Beginning execution of createJVM...');
  VM_Args.version = JNI_VERSION_10;

  myRC = JNI_GetDefaultJavaVMInitArgs( addr(VM_Args) );
  OptStr1 = "-Djava.class.path=.:";
  OptStr2 = "-Djava.compiler=NONE";

  myOptions(1).theOptions = addr(OptStr1);
  myOptions(2).theOptions = addr(OptStr2);
  VM_Args.nOptions = 2;
  VM_Args.JavaVMOption = addr(myOptions);

  /* Create the Java VM */
  myrc = JNI_CreateJavaVM(
             addr(jvm_ptr),
             addr(JNIEnv),
             addr(VM_Args) );

  /* Get the Java Class for the javaPart class */
  myClass = FindClass(JNIEnv, "javaPart");

  /* Get static method ID */
  myMethodID = GetStaticMethodID(JNIEnv, myClass,"main",
               "([Ljava/lang/String;)V" );

  /* Create a Java String Object from the PL/I string. */
  myJString = NewStringUTF(JNIenv, myPLIStr);
  myJObjArray = NewObjectArray(JNIEnv, 1,
                FindClass(JNIEnv,"java/lang/String"), myJString);

  Display('From PL/I - Calling Java method in new JVM from PL/I...');
  Display(' ');
  myNull = CallStaticVoidMethod(JNIEnv, myClass,
               myMethodID, myJObjArray );

  /* destroy the Java VM */
  Display(' ');
  Display('From PL/I - Destroying the new JVM from PL/I...');
```

```
  myRC = DestroyJavaVM( JavaVM );

  end;
```

PL/I サンプル・プログラム#4 - Java 呼び出し API (64 ビット) の呼び出し

```
*Process Margins( 1, 100 ) ;
*Process Display(STD) LP(64);
*Process Default( ASCII ) Or('|');
createJVM: Proc Options(Main);

  %include ibmzjni;

  Dcl myJObjArray      Type jobjectArray;
  Dcl myClass          Type jclass;
  Dcl myMethodID       Type jmethodID;
  Dcl myJString        Type jstring;
  Dcl myRC             Fixed Bin(31) Init(0);
  Dcl myPLIStr         Char(50) Varz
                       Init('... a PLI string in a Java Virtual Machine!');
  Dcl OptStr1          char(1024) varz;
  Dcl OptStr2          char(1024) varz;
  Dcl myNull           Pointer;
  Dcl VM_Args          Like JavaVMInitArgs;
  Dcl myOptions        Like JavaVMOption;

  Display('From PL/I - Beginning execution of createJVM...');
  VM_Args.version = JNI_VERSION_10;

  myRC = JNI_GetDefaultJavaVMInitArgs( addr(VM_Args) );
  OptStr1 = "-Djava.class.path=.:";
  OptStr2 = "-Djava.compiler=NONE";

  myOptions(1).theOptions = addr(OptStr1);
  myOptions(2).theOptions = addr(OptStr2);
  VM_Args.nOptions = 2;
  VM_Args.JavaVMOption = addr(myOptions);

  /* Create the Java VM */
  myrc = JNI_CreateJavaVM(
             addr(jvm_ptr),
             addr(JNIEnv),
             addr(VM_Args) );

  /* Get the Java Class for the javaPart class */
  myClass = FindClass(JNIEnv, "javaPart");

  /* Get static method ID */
  myMethodID = GetStaticMethodID(JNIEnv, myClass,"main",
               "([Ljava/lang/String;)V" );

  /* Create a Java String Object from the PL/I string. */
  myJString = NewStringUTF(JNIenv, myPLIStr);
  myJObjArray = NewObjectArray(JNIEnv, 1,
                FindClass(JNIEnv,"java/lang/String"), myJString);

  Display('From PL/I - Calling Java method in new JVM from PL/I...');
  Display(' ');
  myNull = CallStaticVoidMethod(JNIEnv, myClass,
               myMethodID, myJObjArray );

  /* destroy the Java VM */
  Display(' ');
  Display('From PL/I - Destroying the new JVM from PL/I...');
  myRC = DestroyJavaVM( JavaVM );

  end;
```

ステップ 4: PL/I プログラムのコンパイルとリンク

手順

1. 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

```
    pli -c createJVM.pli
```

2. 出来上がった PL/I オブジェクトデッキを以下のコマンドで共有ライブラリにリンクする：
   - 31 ビット

     ```
     c89 -o createJVM createJVM.o $JAVA_HOME/lib/j9vm/libjvm.x
     ```

   - 64 ビット

     ```
     c89 -Wl,"LP64,DLL" -o createJVM createJVM.o $JAVA_HOME/lib/j9vm/libjvm.x
     ```

   $JAVA_HOME 環境変数への参照に注意してください。 この変数は、Java 製品がインストールされているディレクトリを指すべきである。 例えば、ご使用の環境でこの変数をセットアップするには、次のコマンドを使用します。

   ```
   export JAVA_HOME="/usr/lpp/java/J11.0_64"
   ```

   この場合、Java 製品は /usr/lpp/java/J11.0_64 にインストールされているものとする。

ステップ 5: サンプル・プログラムの実行

手順

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
createJVM
```

サンプル・プログラムの出力は以下のとおりです。

```
From PL/I  - Beginning execution of createJVM...
From PL/I  - Calling Java method in new JVM from PL/I...
From Java  - Hello World... ... a PLI string in a Java Virtual Machine!
From PL/I  - Destroying the new JVM from PL/I...
```

既存の Java VM へのプログラムの接続

Java 呼び出し API を使用すれば、ご使用のプログラムを、そのプログラムによって作成されたものではない既存の Java VM に接続できます。

このタスクについて

PL/I アプリケーションが IMS JMP (Java メッセージ処理) 領域内で実行されていれば、Java VM は IMS によって既に作成されています。 以下の手順を使用すれば、ご使用のプログラムを Java VM に接続できます。 その後で、Java Native Interface を使用して、必要な関数を呼び出すことができます。

手順

1. **JNI_GetCreatedJavaVMs** API を使用して、プログラムの接続先となる Java VM インスタンスを見つけます。

   この IMS 環境では、作成される Java VM が 1 つのみであるため、Java VM ポインターが 1 つのみ返されるように要求します。 この呼び出しは次のようにコーディングできます。

   ```
   rc = JNI_GetCreatedJavaVMs( jvm_ptr, 1, nVMs );
   ```

   jvm_ptr
   Java VM へのポインターです。IBMZJNI インクルード・ファイル内で宣言されます。 呼び出しが正常終了して戻ってくると、この変数内の Java VM のアドレスが Java から返されます。

   nVMs
   Java が戻るときに Java VM アドレスの数で更新する固定ビン (31) 変数です。 呼び出しが成功終了した場合、*nVMs* は 1 です。
2. Java VM 用の JNI 環境ポインターを取得します。

GetEnv 関数は、IBMZJNI インクルード・ファイルで宣言されている **JNIInvokeInterface_ structure** で使用できる：

```
rc = GetEnv( jvm_ptr, JNIEnv, JNI_VERSION_10 );
```

jvm_ptr
Java VM インスタンスへのポインターです。

JNIEnv
Java VM インスタンスの JNI 環境ポインターに対して Java によって設定されるポインターです。

JNI_VERSION_10
インターフェース・バージョン番号の値 (これも IBMZJNI インクルード・ファイル内で宣言されます) を保持する定数です。

タスクの結果

これで、Java VM の JNI 環境ポインターが設定されたため、Java Native Interface を使用して任意の JNI 関数を呼び出すことができます。

Java および PL/I の同等なデータ・タイプの判別

PL/I から Java とのやり取りを行う際には、2 つのプログラム言語間でデータ・タイプを一致させる必要があります。

次の表に、Java の基本的なタイプと PL/I の同等なタイプを示します。

表 *38. Java* 基本タイプと、同等の *PL/I* ネイティブ・タイプ

Java のタイプ	**PL/I タイプ**	**サイズ (ビット)**
Boolean	jboolean	8、符号なし
バイト	jbyte	8
char	jchar	16、符号なし
短い	jshort	16
int	jint	32
長い	jlong	64
浮動小数点	jfloat	21
倍精度	jdouble	53
無効	jvoid	該当なし

第 5 部 特殊プログラミング・タスク

© Copyright IBM Corp. 1999, 2025

第 19 章 AMODE 31 と AMODE 64 コードを混ぜる

64 ビット LE 対応コードから 31 ビット LE 対応コードを呼び出すことができ、その逆も可能です。 これは、FETCHABLE ルーチンに限られ、そのようなルーチンが逆の AMODE で実行されることを指定する必要がある。 これを指定するには、 ENTRY OPTIONS 属性 AMODE と AMODE を使用する必要があります。

間の呼び出しに必要な宣言 AMODEs

31 ビットが 64 ビットを呼び出す場合

以下の属性を使用して、64 ビット・ルーチンを宣言します。

```
OPTIONS(FETCHABLE AMODE64 NODESCRIPTOR) ENTRY(...)
```

ここで、(...)は、少なくとも渡されるパラメーターの数を指定します。 例えば、パラメータを渡さない場合は、 ENTRY() と指定するが、2 つのパラメータを渡す場合は、少なくとも ENTRY(*, *) と指定する。

次に、以下の制限に従う限り、他のルーチンを呼び出すのと同じようにルーチンを呼び出すことができます。

64 ビットが 31 ビットを呼び出す場合

以下の属性で 31 ビット・ルーチンを宣言します。

```
OPTIONS(FETCHABLE AMODE31 NODESCRIPTOR CMPAT(xx) LINKAGE(yy)) ENTRY(...)
```

ここで、(...)は、少なくとも渡されるパラメーターの数を指定します。 例えば、パラメータを渡さない場合は、 ENTRY() と指定するが、2 つのパラメータを渡す場合は、少なくとも ENTRY(*, *) と指定する。

CMPAT(xx) の xx は、 V1、 V2、 V3 のいずれかでなければならない。

LINKAGE(yy) の yy は OPTLINK または SYSTEM のいずれかでなければならない。

次に、以下の制限に従う限り、他のルーチンを呼び出すのと同じようにルーチンを呼び出すことができます。

の間の呼び出しに必要なコンパイラとリンカのオプション。 AMODEs

64 ビットが 31 ビットを呼び出す場合

64 ビットから呼び出される 31 ビット・プロシージャは、 RENT オプションを付けてコンパイルしなければならない（ただし、 NORENT でコンパイルされたコードとリンクしたり、フェッチしたりすることはできる）。

31 ビット・フェッチャブル・モジュールは、 DYNAM=DLL オプションでリンクされなければならない。

間の通話の制限 AMODEs

異なる AMODEs で実行されるルーチン間の呼び出しには、以下の制限が適用される：

- ENTRY または PROCEDURE と OPTIONS(AMODE31) または OPTIONS(AMODE64) が BYVALUE パラメータを持つ場合、そのパラメータの型は POINTER, HANDLE, ORDINAL, または REAL FIXED BIN のいずれかでなければならない。
- ENTRY または PROCEDURE と OPTIONS(AMODE31) または OPTIONS(AMODE64) に RETURNS(BYVALUE...) がある場合 属性が指定されている場合、 RETURNS 属性は、ORDINAL タイプか、4 バイト以下のスケールされていない REAL FIXED BIN のどちらかを指定しなければなりません。

© Copyright IBM Corp. 1999, 2025

- OPTIONS(AMODE31)または OPTIONS(AMODE64)のエントリー定数には、64 個以下のパラメー タを括弧で括ったリスト（空の場合もある）を指定しなければならない。 引数がない場合は、ENTRY ()として宣言しなければなりません。
- OPTIONS(AMODE31) または OPTIONS(AMODE64) を持つエントリ定数は、暗黙的にも明示的にも、別の ENTRY に割り当てることはできない。
- ある AMODE で発生した例外は、その AMODE で処理されなければならない。
- ある AMODE の GOTO 文は、反対側の AMODE のルーチンを越えてはならない。
- OPTIONS(AMODE31)で ENTRY に渡される配列と構造体の引数は、すべて参照でなければならない（配列や構造体の式ではない）。
- パラメータが POINTER、HANDLE、または POINTERs または HANDLEs を含む集約である場合、これらの POINTER と HANDLE は POINTER(32)と HANDLE(32)の属性を持たなければならない。
- RELEASE 関数を使用して、フェッチされたルーチンと関連するストレージを解放することを推奨します。

例えば、整数の単一リンク・リストにポインターを渡すには、そのリストの構造体を次のように宣言しなければなりません。

```
dcl
  1 list_element   based,
    2 list_next    pointer(32),
    2 list_int     fixed bin(31);
```

同様に、変数 X を 64 ビットから 31 ビットに渡す場合、X が AMODE 31 ストレージにあることがわかっている場合を除き、そのパラメーターを ADDR（X）として指定しないでください。

代わりに、パラメーターを単に X と指定する必要があります。 例えば、A31 が CHAR (100) VARYING 変数のアドレスを受け取る 31 ビット・ルーチンである場合は、次のようにコーディングします。

```
DCL A31 ENTRY( POINTER BYVALUE )
        OPTIONS( FETCHABLE AMODE31 NODESCRIPTOR CMPAT(V2) LINKAGE(OPTLINK) );

DCL X CHAR(100) VARYING;

CALL A31( ADDR(X) );
```

コーディングする必要があります。

```
DCL A31 ENTRY( CHAR(100) VARYING BYADDR )
        OPTIONS( FETCHABLE AMODE31 NODESCRIPTOR CMPAT(V2) LINKAGE(OPTLINK) );

DCL X CHAR(100) VARYING;

CALL A31( X );
```

通話パフォーマンスの向上 AMODEs

純粋に AMODE 31 または純粋に AMODE 64 であるコードは、 AMODEs が混在しているコードよりも、ほとんどの場合、パフォーマンスが向上する。 AMODE を混ぜるのは、どうしても必要な場合に限るべきである。

AMODE31 および AMODE64 キーワードは、反対側の AMODE から呼び出されるプロシージャーにのみ指定すべきである。

このようなプロシージャが終了すると、生成されたエピローグ・コードは開いているファイルをすべてフラッシュしてしまうので、 AMODEs を混ぜていないアプリケーションではパフォーマンスが低下する可能性がある。

64 ビット・コードが 31 ビット・コードを呼び出すと、このコンパイラーは以下のコードを生成します。

1. 全パラメーターに AMODE 31 のストレージを割り当てる
2. AMODE 31 にパラメータをコピーする

3. 呼び出しを行います。
4. AMODE 31 のストレージをパラメータにコピーする
5. 割り振りストレージを解放します。

上記の例では、生成されたコードは、 AMODE 31 ストレージに 102 バイト（ CHR(100) VARYING 変数を保持するのに必要なストレージ）を割り当て、そのストレージに X をコピーし、 A31 を呼び出し、制御が AMODE 64 に戻ると、そのストレージを X にコピーし直す。

このような呼び出しがあるコードで、パラメータが INONLY または OUTONLY であることが分かっている場合は、その属性で宣言した方がパフォーマンス上はるかに優れています。

上記の例では、A31 が渡されたパラメーターを使うが設定されていないことがわかった場合、次のように A31 を宣言する必要があります。

```
      DCL A31 ENTRY( CHAR(100) VARYING BYADDR INONLY )
              OPTIONS( FETCHABLE AMODE31 NODESCRIPTOR CMPAT(V2) LINKAGE(OPTLINK) );
```

この場合、コンパイラーは、呼び出しの前に X を AMODE 31 のストレージにコピーするコードを生成するが、呼び出しの後にそのストレージを X にコピーし直すコードは生成しない。 上記の手順 4 はスキップされます。

A31 が渡されたパラメーターを設定するが、使用しないことが分かった場合、次のように A31 を宣言する必要があります。

```
      DCL A31 ENTRY( CHAR(100) VARYING BYADDR OUTONLY )
              OPTIONS( FETCHABLE AMODE31 NODESCRIPTOR CMPAT(V2) LINKAGE(OPTLINK) );
```

この場合、コンパイラーは、呼び出しが行われる前に、パラメータ用に AMODE 31 ストレージを割り当てるコードを生成するが、この時点では、X をそのストレージにコピーするコードは生成しない。 コンパイラーは、呼び出しの後、AMODE 31 のストレージを X にコピーするコードを生成する。 上記手順 2 はスキップされます。

しかし、INONLY と OUTONLY を指定するのは、あなたのコードの動作にマッチするときだけにするよう注意しなければならない。 例えば、64 ビットコードから 31 ビットコードに渡されるパラメータが構造体の場合、31 ビットコードによって構造体の要素が変更されない場合にのみ、 INONLY を指定する必要がある。

第 20 章 PLISAXA と PLISAXB XML パーサーを使う

PLISAXx (x = A or B) 組み込みサブルーチンは、基本的な XML 解析機能を提供し、プログラムが受信 XML ドキュメントを受け取り、整形式かどうかをチェックし、その内容に反応することを可能にする。

これらのサブルーチンは XML を生成しません。XML の生成は、 PL/I プログラムロジックで行うか、 XMLCHAR 組み込み関数を使用する必要があります。

PLISAXA および、特別な環境要件はない。 PLISAXB CICS、IMS、MQ Series、z/OS バッチ、および TSO を含め、主要なすべてのランタイム環境で実行できます。

PLISAXA と、いくつかの重要な限界がある： PLISAXB

- XML 名前空間はサポートされていません。
- ユニコード UTF-8 文書はサポートされていません。
- XML 文書の構文解析前に、その文書全体を (バッファーまたはファイルで) 渡す必要があります。

PLISAXC 、 PLISAXD 組み込みサブルーチンにはこのような制限はない。

関連情報

401 ページの『PLISAXC と PLISAXD XML パーサーを使う』
PLISAXC および PLISAXD 組み込みサブルーチンは、基本的な XML 構文解析機能を提供します。 これによりプログラムは、インバウンド XML 文書を取り込み、適格性を検査し、その内容を処理できます。

概説

XML 構文解析用のインターフェースには、大きく分けてイベント・ベース とツリー・ベースの 2 種類があります。

イベント・ベース API の場合、パーサーはコールバックによって アプリケーションにイベントを報告します。 このようなイベントには、文書の開始、エレメントの開始などがあります。 アプリケーションは、パーサーから報告されたイベントを処理するためのハンドラーを備えています。 Simple API for XML (SAX) は、業界標準のイベントベース API の例である。

Document Object Model (DOM) のようなツリーベースの API の場合、パーサーは XML を内部のツリーベースの表現に変換する。 ツリーをナビゲートするためのインターフェースが提供されています。

IBM PL/I は、XML 文書をパースするための -ライクなイベントベースのインタフェースを提供する。 SAX パーサーは、対応する文書フラグメントへの参照を渡して、 アプリケーション提供のパーサー・イベント用のハンドラーを呼び出します。

パーサーには次の特性があります。

- 高性能であるが非標準のインターフェースを提供します。
- ユニコード UTF-16、または 383 ページの『XML 文書のコード化文字セット』で示されたいずれかの 1 バイト・コード・ページでエンコードされた XML ファイルをサポートします。
- このパーサーは妥当性検査を行いませんが、整形式であるかどうかを部分的に検査し、何かを検出すると例外イベントを生成します。

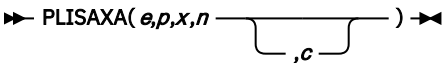
XML 文書の準拠レベルには適格性と有効性の 2 つがあり、 どちらのレベルも XML 標準に定義されています。 XML 標準は、http://www.w3c.org/XML/ に掲載されています。 これらの定義を要約すると、XML 文書が基本的な XML 文法と、 いくつかの特定の規則 (開始エレメントと終了エレメントの タグが一致していることなどの要件) に準拠していれば、 XML 文書は整形式です。 また、整形式の XML 文書は、関連する文書型宣言(DTD)があり、 DTD で表現されている制約に適合していれば有効です。

各パーサー・イベントに対して、 379 ページの『#unique_635/unique_635_Connect_42_figxml』に示すコード例のように、適切なパラメーターを受け取り、適切な戻り値を返す PL/I 関数を提供する必要があります。 特に、戻り値は value（BYVALUE ）で返さなければならないことに注意。また、これらの関数はすべて OPTLINK 。 DEFAULT(LINKAGE(OPTLINK))オプションを使ってこのリンケージを指定することもできますし、 OPTIONS(LINKAGE(OPTLINK)) 属性を通して個々の PROCEDUREs 、 ENTRYs 。

© Copyright IBM Corp. 1999, 2025

PLISAXA 組み込みサブルーチン

PLISAXA 組み込みサブルーチンを使って、プログラム内のバッファにある XML ドキュメントに対して XML パーサーを呼び出すことができます。



e
An event structure

p
A pointer value or "token" that the parser will pass back to the event functions

x
The address of the buffer containing the input XML

n
The number of bytes of data in that buffer

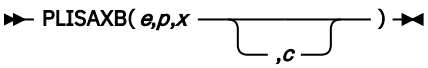
c
A numeric expression specifying the purported codepage of that XML

ノート:

- XML が CHARACTER VARYING または WIDECHAR VARYING 文字列に含まれている場合、 ADDRDATA 組み込み関数を使用して最初のデータ・バイトのアドレスを取得する必要があります。
- XML が WIDECHAR 文字列に含まれている場合、バイト数の値は LENGTH 組み込み関数が返す値の 2 倍になる。

PLISAXB 組み込みサブルーチン

PLISAXB 組み込みサブルーチンを使用して、ファイル内の XML ドキュメントに対して XML パーサーを呼び出すことができます。



e
An event structure

p
A pointer value or "token" that the parser will pass back to the event functions

x
A character string expression specifying the input file

c
A numeric expression specifying the purported codepage of that XML

バッチのもとでは、入力ファイルを指定する文字ストリングは `file://dd:`*ddname* のフォームであり、*ddname* は、そのファイルを指定している DD ステートメントの名前です。

z/OS UNIX では、入力ファイルを指定する文字ストリングは `file://`*filename* の形式になっていなければなりません。 *filename* は z/OS UNIX ファイルの名前です。

バッチと z/OS UNIX の両方の環境とも、入力ファイルを指定する文字ストリングには、先行ブランクも末尾ブランクも含めません。

入力 XML ファイルのサイズは、2G 未満でなくてはなりません。 さらに、パーサーはファイル全体をメモリに読み込むので、あなたのプログラムの REGION は、パーサーがドキュメントのすべてを格納できるストレージの一部を取得するのに十分な大きさでなければなりません。

SAX イベント構成

イベント構造は、パーサーがさまざまな「イベント」に対して呼び出す関数を指す、 24 LIMITED ENTRY 変数からなる構造である。

これらの ENTRY はすべて、 OPTLINK のリンケージを使用しなければならない。

XML パーサーをサポートするライブラリルーチンは、PLISAX イベント構造体のイベントが（ NULLENTRY 組み込み関数または UNSPEC の使用によって）NULL に設定されているかどうかを認識し、その場合は呼び出さない。 これにより、関心のあるイベントのみに XML 構文解析コードを制限することができ、同時に構文解析全体のパフォーマンスを向上させることができます。

サンプル XML 文書

```
xmlDocument =
   '<?xml version="1.0" standalone="yes"?>'
|| '<!--This document is just an example-->'
|| '<sandwich>'
|| '<bread type="baker&quot;s best"/>'
|| '<?spread please use real mayonnaise ?>'
|| '<meat>Ham &amp; turkey</meat>'
|| '<filling>Cheese, lettuce, tomato, etc.</filling>'
|| '<![CDATA[We should add a <relish> element in future!]]>'
|| '</sandwich>'
|| 'junk';
```

この構造体での出現順に、パーサーは次のイベントを認識することができます。

注 : 各イベントの説明は、379 ページの『#unique_635/unique_635_Connect_42_figxml』 の XML 文書の例に対応しています。 この説明にある *XML* テキスト という用語は、イベントに渡されるポインターと長さに基づくストリングを意味しています。

start_of_document

このイベントは、文書の構文解析が開始されるときに 1 回発生します。 パーサーは、 LF (Line Feed) や NL (New Line) のような行制御文字を含む、文書全体のアドレスと長さを渡します。 379 ページの『#unique_635/unique_635_Connect_42_figxml』 で示されている例の場合、文書の長さは 305 文字です。

version_information

このイベントは、オプショナルの XML 宣言内のバージョン情報に対して 発生します。 パーサーは、バージョン値 (379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されている例では 1.0) を含むテキストのアドレスと長さを渡します。

encoding_declaration

このイベントは、XML 宣言内でオプショナルのエンコード宣言に対して 発生します。 パーサーは、エンコード方式値が入ったテキストのアドレスと長さを 渡します。

standalone_declaration

このイベントは、XML 宣言内でオプショナルのスタンドアロン宣言に 対して発生します。 パーサーは、スタンドアロン値を含むテキストのアドレスと長さを渡す (サンプル XML ドキュメントに示されている例では yes)。

document_type_declaration

このイベントは、パーサーが文書タイプ宣言を検出したときに発生します。 文書タイプ宣言は文字シーケンス <!DOCTYPE で始まり > 文字で終わります。 その間には、かなり複雑な文法規則で記述された内容が含まれます。 パーサーは、開始と終了の文字シーケンスを含む宣言全体が含まれるテキストのアドレスと長さを渡します。 これは、XML テキストに区切り文字が含まれる唯一のイベントです。 379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されている例には、文書タイプ宣言が含まれていません。

end_of_document

このイベントは、文書の構文解析が完了したときに 1 回発生します。

start_of_element

このイベントは、エレメント開始タグ、または空エレメント・タグ ごとに 1 回発生します。 パーサーは、エレメント名が入ったテキストのアドレスと長さを 渡します。 379 ページの『#unique_635/unique_635_Connect_42_figxml』 で示した例のパース中の最初の start_of_element イベントでは、これは文字列 sandwich となる。

attribute_name

このイベントは、有効な名前をパーサーが認識した後で、エレメント開始タグまたは空エレメント・タグにおける属性ごとに発生します。 パーサーは、属性名が入ったテキストのアドレスと長さを 渡します。 379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されている例における唯一の属性名は type です。

attribute_characters

このイベントは、属性値のフラグメントごとに発生します。 パーサーは、フラグメントが入ったテキストのアドレスと長さを 渡します。 属性値は通常、次のように複数の行に分割されている場合でも ただ 1 つのストリングで構成されます。

```
<element attribute="This attribute value is
split across two lines"/>
```

ただし、属性値は複数の部分で構成されている場合があります。 例えば、379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されている例にある type 属性の値は、ストリング baker、単一文字 '、およびストリング s best という 3 つのフラグメントで構成されています。 パーサーは、これらのフラグメントを 3 つの別々のイベントとして 渡します。 各ストリング (baker および s best) は attribute_characters イベントとして渡され、単一文字 ' は attribute_predefined_reference イベントとして渡されます。

関連情報

380 ページの『attribute_predefined_reference』

attribute_predefined_reference

このイベントは、5 つの定義済みエンティティー参照 &、'、>、<、および " について属性値において発生します。 パーサーは、それぞれ &、'、>、<、または " のいずれか 1 つを含む CHAR(1) 値または WIDECHAR(1) 値を渡します。

attribute_character_reference

このイベントは、&#dd; または &#xhh; という形式の数字参照 (ユニコード・コード・ポイントまたは「スカラー値」) について、属性値において発生します。 *d* は 10 進数字を表します。 *h* は 16 進数字を表します。 パーサーは、対応する整数値を含む FIXED BIN(31)値を渡す。

end_of_element

このイベントは、エレメント終了タグ、または空エレメント・タグ ごとに、パーサーがタグの終了不等号括弧を認識したときに 1 回発生します。 パーサーは、エレメント名が入ったテキストのアドレスと長さを渡します。

start_of_CDATA_section

このイベントは、 CDATA セクションの開始時に発生する。 CDATA セクションは、文字列 で始まり、文字列 で終わる。そして、そうでなければ XML マークアップとして認識される文字を含むテキストブロックを

「エスケープ」するために使用される。 <![CDATA[ ]]> パーサーは、開始文字 <![CDATA[が含まれているテキストのアドレスと長さを渡します。 パーサーは、これらの区切り文字の間にある CDATA セクションの内容を、1 つの content-characters イベントとして渡します。 サンプル XML 文書に示した例では、content-characters イベントはテキスト `We should add a <relish> element in future!` を渡される。

end_of_CDATA_section

このイベントは、パーサーが CDATA セクションの終わりを認識したときに発生する。 パーサーは、終了文字シーケンス]]> を含むテキストのアドレスと長さを渡します。

content_characters

このイベントは、XML 文書の「本体」である、エレメントの開始タグと 終了タグの間にある文字データを表します。 パーサーは、データを含むテキストのアドレスおよび長さを渡します。 データは通常、次のように複数の行に分割されている場合でも、単一のストリングのみで構成されます。

```
<element1>This character content is
split across two lines</element1>
```

エレメント内容に参照や他のエレメントが含まれている場合、内容全体が複数のセグメントで構成されることがあります。 例えば、 379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示す例の `meat` 要素の内容は、文字列 `Ham ` 、文字 &、文字列 `turkey` で構成される。 これら 2 つのストリング・フラグメントのそれぞれ先頭と末尾にある スペースに注意してください。 パーサーは、これら 3 つの内容フラグメントを別々のイベントとして 渡します。 ストリング・コンテンツ・フラグメント `Ham ` および `turkey` を content_characters イベントとして渡し、単一の & 文字を content_predefined_reference イベントとして渡します。 パーサーはまた、content_characters イベントを使って、CDATA セクションのテキストをアプリケーションに渡す。

content_predefined_reference

このイベントは、5 つの定義済みエンティティー参照 &、'、>、<、および " についてエレメント内容において発生します。 パーサーは、それぞれ &、'、>、<、または " のいずれか 1 つを含む CHAR(1) 値または WIDECHAR(1) 値を渡します。

content_character_reference

このイベントは、&#dd; または &#xhh; という形式の数字参照 (ユニコード・コード・ポイントまたは「スカラー値」) について、エレメント内容において発生します。 *d* は 10 進数字を表します。 *h* は 16 進数字を表します。 パーサーは、対応する整数値を含む FIXED BIN(31)値を渡す。

processing_instruction

処理命令 (PI) を使用すると、XML 文書にアプリケーション用の特別な命令を 含めることができます。 このイベントは、パーサーが PI 開始文字シーケンス <? に続く名前を認識したときに発生します。 さらにこのイベントは、処理命令 (PI) ターゲットに続く、PI 終了文字シーケンス ?> の直前までのデータを対象とします。 データの末尾にある空白文字は含まれますが、先頭にある空白文字は含まれません。 パーサーは、ターゲットが含まれているテキスト (379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されている例では `spread`) のアドレスと長さ、およびデータが含まれているテキスト (例では `please use real mayonnaise`) のアドレスと長さを渡します。

コメント

このイベントは、XML 文書内のコメントに対して発生します。 パーサーは、開始コメント区切り文字 <!-- と終了コメント区切り文字 --> の間にあるテキストのアドレスと長さを渡します。 379 ページの『#unique_635/unique_635_Connect_42_figxml』に示されている例では、`This document is just an example` が唯一のコメントのテキストです。

unknown_attribute_reference

このイベントは、イベント attribute_predefined_character に記載されている 5 つの定義済みエンティティ参照以外のエンティティ参照の属性値内で発生します。 パーサーは、エンティティー名が入ったテキストのアドレスと長さを 渡します。

unknown_content_reference

このイベントは、 content_predefined_character イベントでリストされている 5 つの定義済みエンティティ参照以外のエンティティ参照のエレメントコンテンツ内で発生する。 パーサーは、エンティティー名が入ったテキストのアドレスと長さを 渡します。

start_of_prefix_mapping

このイベントは、現在は生成されません。

end_of_prefix_mapping

このイベントは、現在は生成されません。

除外、

XML 文書の処理中にエラーを検出すると、パーサーはこのイベントを 生成します。

イベント関数に渡されるパラメーター

これらの関数はすべて、パーサーへのリターンコードである BYVALUE FIXED BIN(31)値を返さなければならない。 パーサーを正常に継続するためには、 この値がゼロでなければなりません。

これらの関数はすべて、組み込み関数の第 2 引数として渡されたトークン値である BYVALUE POINTER を第 1 引数として渡される。

以下の例外を除き、すべての関数には、イベントのテキスト要素のアドレスと長さを与える BYVALUE POINTER と BYVALUE FIXED BIN(31) も渡される。 以下の関数とイベントは例外です。

end_of_document
ユーザー・トークン以外の引数は渡されません。

attribute_predefined_reference
ユーザートークンに加えて、1 つの追加引数が渡される。 BYVALUE CHAR(1)、または、 UTF-16 ドキュメントの場合は、 BYVALUE WIDECHAR(1)で、定義済みの文字の値を保持する。

content_predefined_reference
ユーザートークンに加えて、1 つの追加引数が渡される。 BYVALUE CHAR(1)、または、 UTF-16 ドキュメントの場合は、 BYVALUE WIDECHAR(1)で、定義済みの文字の値を保持する。

attribute_character_reference
ユーザートークンに加えて、数値参照の値を保持する BYVALUE FIXED BIN(31)が渡される。

content_character_reference
ユーザートークンに加えて、数値参照の値を保持する BYVALUE FIXED BIN(31)が渡される。

processing_instruction
ユーザー・トークンに加えて、次の 4 つの引数が渡されます。

1. BYVALUE POINTER ターゲットテキストのアドレス
2. BYVALUE FIXED BIN(31)はターゲット・テキストの長さである
3. データ・テキストのアドレスである BYVALUE POINTER
4. データ・テキストの長さである BYVALUE FIXED BIN(31)

除外、
ユーザー・トークンに加えて、次の 3 つの引数が渡されます。

1. BYVALUE POINTER 、問題のあるテキストのアドレス
2. BYVALUE FIXED BIN （31） ： 文書内の問題のあるテキストのバイトオフセット
3. 例外コードの値である BYVALUE FIXED BIN(31)

XML 文書のコード化文字セット

PLISAX 組み込みサブルーチンは、Unicode でエンコードされた WIDECHAR の XML 文書 UTF-16、またはこのセクションで明示的にサポートされているシングルバイト文字セットのいずれかでエンコードされた CHARACTER の XML 文書のみをサポートします。

パーサーは、XML 文書のエンコード方式に関する情報ソースを 3 つまで使用し、 これらのソース間で矛盾を検出した場合は、次のように例外 XML イベントを シグナル通知します。

1. パーサーは、文書の最初の文字を検査することによって 文書の基本エンコードを判別します。
2. ステップ 1 が正常に完了した場合、パーサーはエンコード宣言を 検索します。
3. 最後に、 PLISAX 組み込みサブルーチン呼び出しのコードページ値を参照する。 このパラメータが省略された場合、デフォルトは、明示的に指定した、またはデフォルトで指定された CODEPAGE コンパイラ・オプション値によって提供される値になる。

XML ドキュメントが、このセクションでリストされているサポートされているコードページのいずれかを指定するエンコーディング宣言を含む XML 宣言で始まっている場合、パーサは、基本的なドキュメントエンコーディングまたは PLISAX 組み込みサブルーチンからのエンコーディング情報のいずれとも衝突しなければ、そのエンコーディング宣言を尊重します。 XML 文書に XML 宣言が全くない場合や、XML 宣言にエンコーディング宣言が省略されている場合、パーサは、基本的な文書のエンコーディングと衝突しない限り、 PLISAX 組み込みサブルーチンからのエンコーディング情報を使用して文書を処理します。

対応 EBCDIC コードページ

以下の表において、最初の数字は Euro Country Extended Code Page (ECECP) を、2 番目の数字は Country Extended Code Page (CECP) を表している。

CCSID	説明
01047	Latin 1/オープン・システム
01140、00037	米国、カナダなど
01141、00273	オーストリア、ドイツ
01142、00277	デンマーク、ノルウェー
01143、00278	フィンランド、スウェーデン
01144、00280	イタリア
01145、00284	スペイン、ラテンアメリカ (スペイン語)
01146、00285	英国
01147、00297	フランス
01148、00500	国際

CCSID	説明
01149、00871	アイスランド

サポートされる ASCII コード・ページ

CCSID	説明
00813	ISO 8859-7 ギリシャ語/ラテン語
00819	ISO 8859-1 Latin 1/オープン・システム
00920	ISO 8859-9 Latin 5 (ECMA-128、トルコ TS-5881)

コード・ページの指定

ドキュメントが XML 宣言の中にエンコーディング宣言を含んでいない場合、あるいは XML 宣言を全く持っていない場合、パーサーは PLISAX 組み込みサブルーチン呼び出しによって提供されたエンコーディング情報をドキュメントの基本エンコーディングと共に使用します。

ほとんどの XML 文書の最初にある XML 宣言の中で、文書のエンコード情報を 指定することもできます。エンコード宣言を含む XML 宣言の例を以下に示します。

```
<?xml version="1.0" encoding="ibm-1140"?>
```

XML 文書にエンコーディング宣言が含まれている場合は、 PLISAX 組み込みサブルーチンが提供するエンコーディング情報と、文書の基本エンコーディングが一致していることを確認してください。 エンコーディング宣言、PLISAX 組み込みサブルーチンによって提供されるエンコーディング情報、ドキュメントの基本エンコーディングの間に矛盾がある場合、パーサーは例外 XML イベントを通知します。

エンコード宣言は、番号または別名を使用して指定できます。

番号を使用してエンコード宣言を指定

CCSID （先頭のゼロの有無は問わない）番号の前に、以下のいずれかの文字（大文字・小文字の混在可）を指定することができる。

- IBM_ IBM-
- CP CP_ CP-
- CCSID_ CCSID-

別名を使用してエンコード宣言を指定

以下のサポート対象の別名 (大文字と小文字の組み合わせは自由) を任意に使用できます。

コード・ページ	サポートされる別名
037	EBCDIC-CP-US、EBCDIC-CP-CA、EBCDIC-CP-WT、 EBCDIC-CP-NL
500	EBCDIC-CP-BE、EBCDIC-CP-CH
813	ISO-8859-7, ISO_8859-7
819	ISO-8859-1, ISO_8859-1
920	ISO-8859-9, ISO_8859-9

コード・ページ	サポートされる別名
1200	UTF-16

例外

ほとんどの例外の場合、XML テキストには、文書の、例外が検出されたポイントまで (そのポイントを含む) の構文解析済み部分が入ります。 構文解析の開始前にシグナル通知されるエンコード対立例外に関しては、XML テキストの長さはゼロになるか、または文書からのエンコード宣言値のみが XML テキストに含まれます。

379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されている例には、例外イベントを引き起こす項目が 1 つあります。 それは、sandwich エレメント終了タグの後ろにある余分な junk です。

例外には次の 2 種類があります。

1. 構文解析を任意で継続できる例外

   継続可能な例外の例外コードの範囲は 1 から 99 まで、100001 から 165535 まで、または 200001 から 265535 までです。 例に含まれる例外イベントの例外番号は 1 であるため、その例外イベントは継続可能です。

2. 継続が可能でない致命的な例外

   致命的な例外の例外コードは、99 より大きい (ただし 100000 より小さい) 値です。

例外イベント関数からゼロ以外のリターン・コードでリターンすると、通常、パーサーは文書の処理を停止し、 PLISAXA または PLISAXB 組み込みサブルーチンを呼び出したプログラムに制御を戻します。

継続可能な例外の場合は、ゼロの戻りコードを指定して例外イベント関数から 戻ることにより、パーサーに文書の処理継続を要求します。 ただし、その後でさらに例外が発生する場合もあります。 継続を要求したときにパーサーが行う処置について 詳しくは、392 ページの『継続可能な例外コード』 を参照してください。

範囲 100001 から 165535 まで、および 200001 から 265535 までの例外番号の例外については、特殊なケースが適用されます。 これらの例外コードの範囲は、 PLISAXA または PLISAXB 組み込みサブルーチンによって（明示的または暗黙的に）提供される CCSID の値と、文書の CCSID（エンコーディング宣言を含む文書の先頭を調べることによって決定される）が同一でないことを示しています。

これらの例外の場合、例外イベントに渡される例外コードには、ドキュメントの CCSID に、EBCDIC CCSID の場合は 100000 を加えたもの、ASCII CCSID の場合は 200000 を加えたものが含まれる。 例えば、例外コードに 101140 が含まれる場合、文書の CCSID は 01140 となる。 PLISAXA または PLISAXB 組み込みサブルーチンによって提供される CCSID 値は、呼び出しの最後の引数として明示的に設定されるか、最後の引数が省略され CODEPAGE コンパイラー・オプションの値が使用される場合に暗黙的に設定される。

これらの CCSID 衝突例外の例外イベント関数から戻った後のリターンコードの値によって、パーサーは 3 つのアクションのうちの 1 つを取る：

1. リターンコードが 0 の場合、パーサーは組み込みサブルーチンが提供する CCSID 。
2. 戻りコードに文書の CCSID （つまり、元の例外コード値から 100000 または 200000 を引いた値）が含まれている場合、パーサーは文書の CCSID を使って処理を進めます。 これは、構文解析イベントのいずれかから非ゼロの値が戻された後、パーサーが処理を継続する唯一のケースです。
3. そうでない場合、パーサーは文書の処理を停止し、 PLISAXA または PLISAXB 組み込みサブルーチンに制御を戻し、 ERROR 条件を発生させる。

例

この例では、 PLISAXA 組み込みサブルーチンの使い方を説明する。

次の例では、379 ページの『#unique_635/unique_635_Connect_42_figxml』 に示されているサンプル XML 文書が使用されています。

PLISAXA コーディングの例 - 型宣言：

```
saxtest: package exports(saxtest);

  define alias event
      limited entry( pointer, pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_end_of_document
      limited entry( pointer )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_predefined_ref
      limited entry( pointer, char(1) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) nodescriptor );

  define alias event_character_ref
      limited entry( pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_pi
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_exception
      limited entry( pointer, pointer, fixed bin(31),
                     fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );
```

PLISAXA コーディング例 - イベント構造：

```
saxtest: proc options( main );

  dcl
    1 eventHandler static

      ,2 e01 type event
             init( start_of_document )
      ,2 e02 type event
             init( version_information )
      ,2 e03 type event
             init( encoding_declaration )
      ,2 e04 type event
             init( standalone_declaration )
      ,2 e05 type event
             init( document_type_declaration )
      ,2 e06 type event_end_of_document
             init( end_of_document )
      ,2 e07 type event
             init( start_of_element )
      ,2 e08 type event
             init( attribute_name )
      ,2 e09 type event
             init( attribute_characters )
      ,2 e10 type event_predefined_ref
             init( attribute_predefined_reference )
      ,2 e11 type event_character_ref
             init( attribute_character_reference )
      ,2 e12 type event
             init( end_of_element )
      ,2 e13 type event
             init( start_of_CDATA )
      ,2 e14 type event
             init( end_of_CDATA )
      ,2 e15 type event
             init( content_characters )
      ,2 e16 type event_predefined_ref
             init( content_predefined_reference )
      ,2 e17 type event_character_ref
             init( content_character_reference )
      ,2 e18 type event_pi
             init( processing_instruction )
      ,2 e19 type event
```

```
               init( comment )
        ,2 e20 type event
               init( unknown_attribute_reference )
        ,2 e21 type event
               init( unknown_content_reference )
        ,2 e22 type event
               init( start_of_prefix_mapping )
        ,2 e23 type event
               init( end_of_prefix_mapping )
        ,2 e24 type event_exception
               init( exception )
        ;
```

PLISAXA コーディング例 - メインルーチン：

```
    dcl token       char(8);

    dcl xmlDocument char(4000) var;

    xmlDocument =
       '<?xml version="1.0" standalone="yes"?>'
    || '<!--This document is just an example-->'
    || '<sandwich>'
    || '<bread type="baker&quot;s best"/>'
    || '<?spread please use real mayonnaise ?>'
    || '<meat>Ham &amp; turkey</meat>'
    || '<filling>Cheese, lettuce, tomato, etc.</filling>'
    || '<![CDATA[We should add a <relish> element in future!]]>'.
    || '</sandwich>'
    || 'junk';


    call plisaxa( eventHandler,
                  addr(token),
                  addrdata(xmlDocument),
                  length(xmlDocument) );

  end;
```

PLISAXA コーディング例 - イベントルーチン：

```
  dcl chars char(32000) based;

  start_of_document:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' length=' || tokenlength );

    return(0);
  end;

  version_information:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  encoding_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );
```

```
    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

standalone_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  document_type_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  end_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;


start_of_element:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  attribute_name:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
```

```
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  attribute_characters:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

attribute_predefined_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) nodescriptor );

    dcl userToken      pointer;
    dcl reference   char(1);

    put skip list( lowercase( procname() )
     || ' ' || hex(reference ) );

    return(0);
  end;

  attribute_character_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl reference      fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || hex(reference ) );

    return(0);
  end;

  end_of_element:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

start_of_CDATA:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;
```

```
  end_of_CDATA:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  content_characters:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

content_predefined_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) nodescriptor );

    dcl userToken      pointer;
    dcl reference   char(1);

    put skip list( lowercase( procname() )
     || ' ' || hex(reference ) );

    return(0);
  end;

  content_character_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl reference      fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || hex(reference ) );

    return(0);
  end;

  processing_instruction:
    proc( userToken, piTarget, piTargetLength,
                             piData, piDataLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl piTarget       pointer;
    dcl piTargetLength fixed bin(31);
    dcl piData         pointer;
    dcl piDataLength   fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(piTarget->chars,1,piTargetLength ) ||
'>' );

    return(0);
  end;


comment:
    proc( userToken, xmlToken, TokenLength )
```

```
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  unknown_attribute_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  unknown_content_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

start_of_prefix_mapping:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  end_of_prefix_mapping:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>'
);

    return(0);
  end;

  exception:
    proc( userToken, xmlToken, currentOffset, errorID )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );
```

```
    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl currentOffset  fixed bin(31);
    dcl errorID        fixed bin(31);

    put skip list( lowercase( procname() )
     || ' errorid =' || errorid );

    return(0);
  end;
end;
```

前のプログラムで生成される出力は次のとおりです。

PLISAXA コーディング例 - プログラム出力：

```
  start_of_dcoument length=
     305
  version_information <1.0>
  standalone_declaration <yes>
  comment <This document is just an example>
  start_of_element <sandwich>
  start_of_element <bread>
  attribute_name <type>
  attribute_characters <baker>
  attribute_predefined_reference 7D
  attribute_characters <s best>
  end_of_element <bread>
  processing_instruction <spread>
  start_of_element <meat>
  content_characters <Ham >
  content_predefined_reference 50
  content_characters < turkey>
  end_of_element <meat>
  start_of_element <filling>
  content_characters <Cheese, lettuce, tomato, etc.>
  end_of_element <filling>
  start_of_cdata <<![CDATA[>
  content_characters <We should add a <relish> element in future!>
  end_of_cdata <]]>
  end_of_element <sandwich>
  exception errorid =             1
  content_characters <j>
  exception errorid =             1
  content_characters <u>
  exception errorid =             1
  content_characters <n>
  exception errorid =             1
  content_characters <k>
  end_of_document
```

継続可能な例外コード

このトピックでは、例外コードについて説明します。 また、例外が発生した後で処理を続行するようにパーサーに要求したときにパーサーが行うアクションについても説明します。

次の表では、例外イベントに渡される例外コード・パラメーターの各値は「**番号**」列にリストされています。 例外コードごとに説明があります。 また、例外が発生した後で処理を続行するようにパーサーに要求したときにパーサーが行うアクションも示されています。 この説明にある *XML* テキスト という用語は、イベントに渡されるポインターと長さに基づくストリングを意味しています。

表 *39.* 継続可能な例外

数値	説明	継続時のパーサーの処置
1	エレメント内容の外側で、空白文字をスキャン中にパーサーが無効文字を検出した。	パーサーは content_characters イベントを生成し、XML テキストには (単一の) 無効文字が入る。 構文解析は無効文字の後の文字から継続する。

表 39. 継続可能な例外 (続き)

数値	説明	継続時のパーサーの処置
2	エレメント内容の外側で、処理命令、エレメント、コメント、または文書タイプ宣言の無効な開始をパーサーが検出した。	パーサーは content_characters イベントを生成し、XML テキストには 2 から 3 文字まで の無効な先頭文字シーケンスが入る。 構文解析は無効シーケンスの後の文字から継続する。
3	属性名の重複をパーサーが検出した。	パーサーは attribute_name イベントを生成し、XML テキストには 重複した属性名が入る。
4	属性値の中にマークアップ文字 < をパーサーが検出した。	パーサーは例外イベントを生成する前に、< 文字の前にある属性値の部分に対して attribute_characters イベントを生成する。 例外イベントの後で、パーサーは attribute_characters イベントを生成し、< を XML テキストに含める。 構文解析は < の後の文字から継続する。
5	エレメントの開始タグと終了タグの名前が一致しない。	パーサーは end_of_element イベントを生成し、XML テキストには一致しなかった終了名が入る。
6	エレメント内容の中で無効文字をパーサーが検出した。	パーサーは、後続の content_characters イベントの XML テキストに無効文字を入れる。
7	エレメント内容の中で、エレメント、コメント、処理命令、または CDATA セクションの無効な開始をパーサーが検出した。	例外イベントを生成する前に、< マークアップ文字の前にある内容の部分に対して、パーサーは content_characters イベントを生成する。 例外イベントの後、パーサーは content_characters イベントを生成し、XML テキストには < と無効文字の 2 つの文字が入る。 構文解析は無効文字の後の文字から継続する。
8	エレメント内容の中で、対になる CDATA 開始文字シーケンス <![CDATA[がない終了文字シーケンス]]> をパーサーが検出した。	パーサーは例外イベントを生成する前に、]]> 文字シーケンスの前にある内容の部分に対して content_characters イベントを生成する。 パーサーは、例外イベントの後で、content_characters イベントを生成し、3 文字のシーケンス]]> を XML テキストに含める。 構文解析はこのシーケンスの後の文字から継続する。
9	コメントの中で無効文字をパーサーが検出した。	パーサーは、後続の comment イベントの XML テキストに無効文字を入れる。
10	コメントの中で、文字シーケンス -- の後に > が付いていないことをパーサーが検出した。	パーサーは -- 文字シーケンスによってコメントが終了したと想定し、comment イベントを生成する。 構文解析は -- シーケンスの後の文字から継続する。
11	処理命令データ・セグメントの中で、無効文字をパーサーが検出した。	パーサーは、後続の processing_instruction イベントに対して XML テキストに無効文字を組み込む。
12	処理命令ターゲット名が、小文字、大文字、または大/小文字混合の xml だった。	パーサーは processing_instruction イベントを生成し、元のとおりの大/小文字で xml を XML テキストに含める。

表 *39.* 継続可能な例外 *(*続き*)*

数値	説明	継続時のパーサーの処置
※13	16 進文字参照 (形式 &#xdddd;) の中で、無効数字をパーサーが検出した。	パーサーは attribute_characters イベントまたは content_characters イベントを生成し、XML テキストには無効数字が入る。 参照の構文解析は、この無効数字の後から継続する。
14	10 進文字参照 (形式 &#dddd;) の中で、無効数字をパーサーが検出した。	パーサーは attribute_characters イベントまたは content_characters イベントを生成し、XML テキストには無効数字が入る。 参照の構文解析は、この無効数字の後から継続する。
15	XML 宣言におけるエンコード宣言値が、小文字大文字に関係なく A から Z までの文字で始まっていなかった。	パーサーは encoding イベントを生成し、XML テキストには指定されたとおりのエンコード宣言値が入る。
16	文字参照が正しい XML 文字を参照していない。	パーサーは、attribute_character_reference イベントまたは content_character_reference イベントを生成し、文字参照で指定された単一ユニコード文字を XML-NTEXT に含める。
17	エンティティー参照名の中で、無効文字をパーサーが検出した。	パーサーは、後続の unknown_attribute_reference イベント、または unknown_content_reference イベントの XML テキストに 無効文字を入れる。
18	属性値の中で、無効文字をパーサーが検出した。	パーサーは、後続の attribute_characters イベントの XML テキストに無効文字を入れる。 **注 :** PL/I XML パーサーは標準からはずれて、属性ストリングにおいて「<」を有効なものとして受け入れます。
50	文書は EBCDIC でエンコードされており、CODEPAGE コンパイラオプションはサポートされている EBCDIC コードページを指定していましたが、文書のエンコーディング宣言は認識可能なエンコーディングを指定していませんでした。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
51	文書は EBCDIC でエンコードされており、文書エンコード宣言はサポートされている EBCDIC エンコードを指定していますが、パーサは CODEPAGE コンパイラオプションで指定されたコードページをサポートしていません。	パーサーは、文書のエンコード宣言に指定されたエンコード方式を使用する。
52	文書は EBCDIC でエンコードされており、CODEPAGE コンパイラ・オプションはサポートされている EBCDIC コード・ページを指定していたが、文書のエンコーディング宣言は ASCII エンコーディングを指定していた。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
53	文書は EBCDIC でエンコードされており、CODEPAGE コンパイラオプションはサポートされている EBCDIC コードページを指定していましたが、文書エンコーディング宣言はサポートされている Unicode エンコーディングを指定していました。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。

表 39. 継続可能な例外 (続き)

数値	説明	継続時のパーサーの処置
54	文書は EBCDIC でエンコードされており、CODEPAGE コンパイラ・オプションはサポートされている EBCDIC コード・ページを指定していますが、文書のエンコード宣言はパーサがサポートしていない Unicode エンコードを指定しています。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
55	文書は EBCDIC でエンコードされており、CODEPAGE コンパイラ・オプションはサポートされている EBCDIC コード・ページを指定したが、文書のエンコーディング宣言はパーサーがサポートしていないエンコーディングを指定した。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
56	文書は ASCII でエンコードされており、CODEPAGE コンパイラー・オプションはサポートされている ASCII コード・ページを指定していたが、文書のエンコーディング宣言は認識可能なエンコーディングを指定していなかった。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
57	ドキュメントは ASCII でエンコードされ、ドキュメントのエンコーディング宣言はサポートされている ASCII エンコーディングを指定したが、パーサは CODEPAGE コンパイラオプションで指定されたコードページをサポートしていない。	パーサーは、文書のエンコード宣言に指定されたエンコード方式を使用する。
58	文書は ASCII でエンコードされており、CODEPAGE コンパイラ・オプションはサポートされている ASCII コード・ページを指定していたが、文書のエンコーディング宣言はサポートされている EBCDIC エンコーディングを指定していた。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
59	文書は ASCII でエンコードされ、 CODEPAGE コンパイラ・オプションはサポートされる ASCII コード・ページを指定したが、文書のエンコーディング宣言はサポートされる Unicode エンコーディングを指定した。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
60	文書は ASCII でエンコードされており、CODEPAGE コンパイラ・オプションはサポートされている ASCII コード・ページを指定していたが、文書のエンコード宣言はパーサがサポートしていない Unicode エンコードを指定していた。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。
61	ドキュメントは ASCII でエンコードされており、 CODEPAGE コンパイラー・オプションはサポートされている ASCII コード・ページを指定したが、ドキュメントのエンコーディング宣言はパーサーがサポートしていないエンコーディングを指定した。	パーサーは、 CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。

表 39. 継続可能な例外 (続き)

数値	説明	継続時のパーサーの処置
100001 から 165535 まで	EBCDIC CODEPAGE コンパイラオプションで指定されたエンコーディングと、文書エンコーディング宣言で指定されたエンコーディングは、どちらもサポートされている EBCDIC コードページですが、同じではありません。 例外コードには、エンコード宣言の CCSID に 100000 を加算した値が入る。	例外イベントからゼロを返した場合、パーサーは CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。 文書のエンコード宣言からの CCSID を戻した場合 (例外コードから 100000 を減算して)、パーサーはこのエンコード方式を使用する。
200001 から 265535 まで	CODEPAGE コンパイラオプションで指定されたエンコーディングと、文書エンコーディング宣言で指定されたエンコーディングは、どちらもサポートされている ASCII コードページですが、同じではありません。 例外コードには、エンコード宣言の CCSID に 200000 を加算した値が入る。	例外イベントからゼロを返した場合、パーサーは CODEPAGE コンパイラー・オプションで指定されたエンコーディングを使用します。 文書のエンコード宣言からの CCSID を戻した場合 (例外コードから 200000 を減算して)、パーサーはこのエンコード方式を使用する。

例外コードの終了

このトピックでは、終了例外コードについて説明します。

表 40. 終了例外

数値	説明
100	XML 宣言の開始のスキャン中に、パーサーが文書の終わりに達した。
101	XML 宣言の終了の検索中に、パーサーが文書の終わりに達した。
102	ルート・エレメントの検索中に、パーサーが文書の終わりに達した。
103	XML 宣言のバージョン情報の検索中に、パーサーが文書の終わりに達した。
104	XML 宣言のバージョン情報値の検索中に、パーサーが文書の終わりに達した。
106	XML 宣言のエンコード宣言値の検索中に、パーサーが文書の終わりに達した。
108	XML 宣言のスタンドアロン宣言値の検索中に、パーサーが文書の終わりに達した。
109	属性名のスキャン中に、パーサーが文書の終わりに達した。
110	属性値のスキャン中に、パーサーが文書の終わりに達した。
111	属性値の文字参照またはエンティティー参照のスキャン中に、パーサーが文書の終わりに達した。
112	空エレメント・タグのスキャン中に、パーサーが文書の終わりに達した。
113	ルート・エレメント名のスキャン中に、パーサーが文書の終わりに達した。
114	エレメント名のスキャン中に、パーサーが文書の終わりに達した。
115	エレメント内容の文字データのスキャン中に、パーサーが 文書の終わりに達した。
116	エレメント内容の処理命令のスキャン中に、パーサーが 文書の終わりに達した。
117	エレメント内容のコメントまたは CDATA セクションの スキャン中に、パーサーが文書の終わりに達した。
118	エレメント内容のコメントのスキャン中に、パーサーが文書の終わりに達 した。
119	エレメント内容の CDATA セクションの スキャン中に、パーサーが文書の終わりに達した。

表 *40.* 終了例外 *(*続き*)*

数値	説明
120	エレメント内容の文字参照またはエンティティー参照のスキャン中に、パーサーが文書の終わりに達した。
121	ルート・エレメントの終了後のスキャン中に、 パーサーが文書の終わりに達した。
122	文書タイプ宣言の開始が無効である可能性があることを パーサーが検出した。
123	2 番目の文書タイプ宣言をパーサーが検出した。
124	ルート・エレメント名の先頭文字が、文字、_、または：でない。
125	エレメントの最初の属性名の先頭文字が、文字、_、または：でない。
126	エレメント名の内部または後に、パーサーが無効文字を検出した。
127	属性名の後に = 以外の文字が続いていることをパーサーが検出した。
128	無効な属性値区切り文字をパーサーが検出した。
130	属性名の先頭文字が、文字、_、：のいずれでもなかった。
131	属性名の内部または後に無効文字をパーサーが検出した。
132	空エレメント・タグが、> とそれに続く / で終わっていなかった。
133	エレメント終了タグ名の先頭文字が、文字、_、：のいずれでもなかった。
134	エレメント終了タグ名が > で終わっていない。
135	エレメント名の先頭文字が、文字、_、または：でない。
136	エレメント内容の中で、コメントまたは CDATA セクションの無効な 開始をパーサーが検出した。
137	コメントの無効な開始をパーサーが検出した。
138	処理命令の先頭文字が、文字、_、または：でない。
139	処理命令ターゲット名の内部または後に、無効文字をパーサーが検出した。
140	処理命令が終了文字シーケンス ?> で終わっていない。
141	文字参照またはエンティティー参照名の中で、& の後に無効文字をパーサーが検出した。
142	XML 宣言の中にバージョン情報がない。
143	XML 宣言の中で、'version' の後に = が付いていない。
144	XML 宣言の中で、バージョン宣言値が欠落しているか、 誤って区切られている。
145	XML 宣言の中で、バージョン情報値に不正な文字が指定されているか、 または開始と終了の区切り文字が一致しない。
146	XML 宣言の中で、バージョン情報値の終了区切り文字の後に 無効文字をパーサーが検出した。
147	XML 宣言の中で、オプショナルのエンコード宣言があるべき個所に 無効な属性をパーサーが検出した。
148	XML 宣言の中で、encoding の後に = が付いていない。
149	XML 宣言の中で、エンコード宣言値が欠落しているか、 誤って区切られている。

表 *40.* 終了例外 *(*続き*)*

数値	説明
150	XML 宣言の中で、エンコード宣言値に不正な文字が指定されているか、 または開始と終了の区切り文字が一致しない。
151	XML 宣言の中で、エンコード宣言値の終了区切り文字の後に 無効文字をパーサーが検出した。
152	XML 宣言の中で、オプショナルのスタンドアロン宣言があるべき個所に 無効な属性をパーサーが検出した。
153	XML 宣言の中で、'standalone' の後に = が付いていない。
154	XML 宣言の中で、スタンドアロン宣言値が欠落しているか、 誤って区切られている。
155	スタンドアロン宣言値が yes または no のどちらでもない。
156	XML 宣言の中で、スタンドアロン宣言値に不正な文字が指定されているか、 または開始と終了の区切り文字が一致しない。
157	XML 宣言の中で、スタンドアロン宣言値の終了区切り文字の後に 無効文字をパーサーが検出した。
158	XML 宣言が正しい文字シーケンス ?> で終わっていないか、無効な属性を含んでいる。
159	ルート・エレメントの終了後、文書タイプ宣言の開始をパーサーが 検出した。
160	ルート・エレメントの終了後、エレメントの開始をパーサーが 検出した。
300	この文書は EBCDIC でエンコードされていたが、 CODEPAGE コンパイラ・オプションでサポートされている ASCII コード・ページが指定されていた。
301	この文書は EBCDIC でエンコードされていたが、 CODEPAGE コンパイラ・オプションで Unicode が指定されていた。
302	この文書は EBCDIC でエンコードされているが、 CODEPAGE コンパイラ・オプションでサポートされていないコード・ページが指定されている。
303	ドキュメントは EBCDIC でエンコードされていましたが、 CODEPAGE コンパイラオプションはサポートされておらず、ドキュメントエンコーディング宣言は空か、サポートされていないアルファベットエンコーディングエイリアスが含まれていました。
304	この文書は EBCDIC でエンコードされていますが、 CODEPAGE コンパイラオプションはサポートされておらず、文書にエンコード宣言が含まれていません。
305	ドキュメントは EBCDIC でエンコードされていますが、 CODEPAGE コンパイラオプションはサポートされておらず、ドキュメントエンコーディング宣言はサポートされている EBCDIC エンコーディングを指定していません。
306	ドキュメントは ASCII でエンコードされていたが、 CODEPAGE コンパイラ・オプションでサポートされている EBCDIC コード・ページが指定されていた。
307	文書は ASCII でエンコードされていたが、 CODEPAGE コンパイラ・オプションで Unicode が指定されていた。
308	ドキュメントは ASCII でエンコードされていたが、 CODEPAGE コンパイラオプションが、サポートされている EBCDIC コードページ、ASCII、または Unicode を指定していなかった。
309	CODEPAGE コンパイラ・オプションはサポートされている ASCII コード・ページを指定したが、ドキュメントは Unicode でエンコードされていた。

表 *40.* 終了例外 *(*続き*)*

数値	説明
310	CODEPAGE コンパイラ・オプションはサポートされている EBCDIC コード・ページを指定したが、ドキュメントは Unicode でエンコードされていた。
311	CODEPAGE コンパイラ・オプションでサポートされていないコード・ページが指定されましたが、ドキュメントは Unicode でエンコードされていました。
312	文書は ASCII でエンコードされていたが、外部指定されたエンコード方式と、文書のエンコード宣言で指定されたエンコード方式が両方ともサポートされていなかった。
313	文書は ASCII でエンコードされているが、 CODEPAGE コンパイラオプションがサポートされておらず、文書にエンコード宣言が含まれていない。
314	ドキュメントは ASCII でエンコードされていますが、 CODEPAGE コンパイラオプションがサポートされておらず、ドキュメントのエンコーディング宣言でサポートされている ASCII エンコーディングが指定されていません。
315	文書は UTF-16 リトル・エンディアンでエンコードされているが、パーサーはこのプラットフォーム上でその方式をサポートしない。
316	文書は UCS4 でエンコードされているが、パーサーは その方式をサポートしない。
317	文書のエンコード方式をパーサーが判別できない。 文書は損傷している可能性がある。
318	文書は UTF-8 でエンコードされているが、パーサーは その方式をサポートしない。
319	文書は UTF-16 ビッグ・エンディアンでエンコードされているが、パーサーはこのプラットフォーム上でその方式をサポートしない。
501, 502, 503	PLISAX(A|B)で内部エラーが発生した。 IBM サポートにお問い合わせください。
500	PLISAXA 内部データ構造のメモリ割り当てに失敗しました。 アプリケーション・プログラムが使用できるストレージ量を増やしてください。
520	PLISAXB 内部データ構造のメモリ割り当てに失敗しました。 アプリケーション・プログラムが使用できるストレージ量を増やしてください。
521	PLISAX(A|B)で内部エラーが発生した。 IBM サポートにお問い合わせください。
523	PLISAXB ファイル I/O エラーが発生しました。
524	ファイルシステムから XML ドキュメントをキャッシュしようとして、 PLISAXB 、メモリ割り当てに失敗しました。 アプリケーション・プログラムが使用できるストレージ量を増やしてください。
525	サポートされていない URI スキームが PLISAXB に指定された。
526	PLISAXB 、提供された XML ドキュメントが最小の 4 文字に満たないか、大きすぎた。
527, 560	PLISAX(A|B)で内部エラーが発生した。 IBM サポートにお問い合わせください。
561	PLISAX(A|B)のいずれにもイベントハンドラが指定されていない。
562, 563, 580, 581	PLISAX(A|B)で内部エラーが発生した。 IBM サポートにお問い合わせください。
600 から 99999 まで	内部エラー。 このエラーをサービス担当員に報告してください。

第 21 章 PLISAXC と PLISAXD XML パーサーを使う

PLISAXC および PLISAXD 組み込みサブルーチンは、基本的な XML 構文解析機能を提供します。 これによりプログラムは、インバウンド XML 文書を取り込み、適格性を検査し、その内容を処理できます。

PLISAXC で使用される XML パーサーは検証を行わないが、部分的に整形式のエラーをチェックし、エラーを発見した場合は例外イベントを生成する。

PLISAXD 組み込みサブルーチンは、検証機能を備えた XML 解析機能を提供する。 これにより、インバウンド XML 文書がインバウンド XML スキーマで指定された一連の規則に従っているかどうかが判別されます。

PLISAXC 、PLISAXD のサブルーチンでは XML の生成は行わず、PL/I のプログラムロジックか、XMLCHAR の組み込み関数で行う必要がある。

PLISAXC と は、 でサポートされていないことを除けば、特別な環境要件はない。 PLISAXD AMODE 24 CICS、IMS、MQ Series、z/OS バッチ、および TSO を含め、主要なすべてのランタイム環境で実行できます。

PLISAXC 、 PLISAXD 組み込みサブルーチン、 PLISAXA 、 PLISAXB 組み込みサブルーチンには多くの類似点があるため、このセクションの一部の情報は、 377 ページの『第 20 章 PLISAXA と PLISAXB XML パーサーを使う』 の情報と重複しています。

概説

XML 構文解析用のインターフェースには、大きく分けてイベント・ベース とツリー・ベースの 2 種類があります。

イベント・ベース API の場合、パーサーはコールバックによって アプリケーションにイベントを報告します。 このようなイベントには、文書の開始、エレメントの開始などがあります。 アプリケーションは、パーサーから報告されたイベントを処理するためのハンドラーを備えています。 Simple API for XML (SAX) は、業界標準のイベントベース API の例である。

Document Object Model (DOM) のようなツリーベースの API の場合、パーサーは XML を内部のツリーベースの表現に変換する。 ツリーをナビゲートするためのインターフェースが提供されています。

IBM PL/I コンパイラは、 または を使用して、 のようなイベントベースのインタフェースを XML ドキュメントの解析に提供します。 PLISAXC PLISAXD SAX パーサーは、対応する文書フラグメントへの参照を渡して、 アプリケーション提供のパーサー・イベント用のハンドラーを呼び出します。

パーサーには次の特性があります。

- 高性能であるが非標準のインターフェースを提供します。
- ユニコード UTF-16、UTF-8、または 409 ページの『XML 文書のコード化文字セット』 で示されたいずれかの 1 バイト・コード・ページでエンコードされた XML ファイルをサポートします。

XML 文書の準拠レベルには適格性と有効性の 2 つがあり、 どちらのレベルも XML 標準に定義されています。 XML 標準は、http://www.w3c.org/XML/ に掲載されています。 これらの定義を要約すると、XML 文書が基本的な XML 文法と、 いくつかの特定の規則 (開始エレメントと終了エレメントの タグが一致していることなどの要件) に準拠していれば、 XML 文書は整形式です。 さらに、整形式 XML 文書に文書タイプ宣言 (DTD) が 関連していて、文書が DTD に表された制約に準拠している場合、 その文書は有効です。

各パーサー・イベントに対して、 403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示すコード例のように、適切なパラメーターを受け取り、適切な戻り値を返す PL/I 関数を提供する必要があります。

ノート：

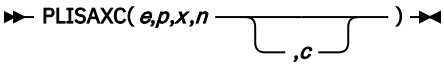
- これらの関数では、返り値は value (BYVALUE) で返さなければならない。

© Copyright IBM Corp. 1999, 2025

- これらの機能については、使用するリンケージは OPTLINK のリンケージでなければならない。DEFAULT(LINKAGE(OPTLINK))オプションを使ってこの連結を指定することもできるし、OPTIONS(LINKAGE(OPTLINK)) 属性を使って個々の PROCEDUREs と ENTRY で指定することもできる。

PLISAXC 組み込みサブルーチン

PLISAXC 組み込みサブルーチンでは、プログラム内の 1 つ以上のバッファにある XML ドキュメントに対して XML パーサーを呼び出すことができます。



e
イベント構造体。

p
パーサーがイベント関数に戻すポインター値または「トークン」。

x
入力 XML が入っている初期バッファーのアドレス。

n
そのバッファーにあるデータのバイト数。

XML が UCHAR ストリングに含まれている場合、バイト数の値は BYTELENGTH 組み込み関数から返されます。

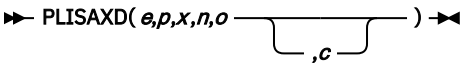
c
その XML のコード・ページを指定する数値表現。

ノート:

- XML が VARYING or VARYING4 string に含まれている場合は、 ADDRDATA 組み込み関数を使用して、最初のデータ・バイトのアドレスを取得する必要があります。
- XML が WIDECHAR 文字列に含まれている場合、バイト数の値は LENGTH 組み込み関数が返す値の 2 倍になる。

PLISAXD 組み込みサブルーチン

PLISAXD 組み込みサブルーチンにより、検証機能を持つ XML パーサーを呼び出すことができます。 XML 文書と Optimized Schema Representation (OSR) ファイルの両方が、プログラム内の 1 つ以上のバッファー内にあります。



e
イベント構造体。

p
パーサーがイベント関数に戻すポインター値または「トークン」。

x
入力 XML が入っている初期バッファーのアドレス。

n
そのバッファーにあるデータのバイト数。

XML が UCHAR ストリングに含まれている場合、バイト数の値は BYTELENGTH 組み込み関数から返されます。

o
入力 OSR が入っているバッファーのアドレス。

c
XML 文書のコード・ページを指定する数値表現。

注:

- XML が VARYING or VARYING4 string に含まれている場合は、 ADDRDATA 組み込み関数を使用して、最初のデータ・バイトのアドレスを取得する必要があります。
- XML が WIDECHAR 文字列に含まれている場合、バイト数の値は LENGTH 組み込み関数が返す値の 2 倍になる。
- OSR は、プリプロセス版のスキーマです。 OSR について詳しくは、 *XML System Services* ユーザーズ・ガイドおよび解説書を参照してください。

SAX イベント構成

イベント構造は、 19 LIMITED ENTRY 変数からなる構造で、パーサーがさまざまな「イベント」に対して呼び出す関数を指している。

これらの ENTRYs はすべて、 OPTLINK のリンケージを使用しなければならない。

XML パーサーをサポートするライブラリ・ルーチンは、PLISAX イベント構造体のイベントが (NULLENTRY 組み込み関数または UNSPEC の使用によって) NULL に設定されていることを認識し、その場合はそのイベントを呼び出さない。 これにより、関心のあるイベントのみに XML 構文解析コードを制限することができ、 同時に構文解析全体のパフォーマンスを向上させることができます。

ENTRYs PLISAXC と にプログラムから渡されるユーザートークンである。 PLISAXD

このセクションでは、 Sample XML 文書の XML 文書の例を参照して、これら 19 のイベントを説明する。 この説明にある *XML テキスト* という用語は、イベントに渡されるポインターと長さに基づくストリングを意味しています。

サンプル XML 文書

```
xmlDocument =
   '<?xml version="1.0" standalone="yes"?>'
|| '<!--This document is just an example-->'
|| '<sandwich>'
|| '<bread type="baker&quot;s best"/>'
|| '<?spread please use real mayonnaise ?>'
|| '<meat>Ham &amp; turkey</meat>'
|| '<filling>Cheese, lettuce, tomato, etc.</filling>'
|| '<![CDATA[We should add a <relish> element in future!]]>'
|| '</sandwich>';
```

XML 文書の内容に応じて、パーサーは以下のイベントを認識します。

start_of_document

このイベントは、文書の構文解析が開始されるときに 1 回発生します。 パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。

version_information

このイベントは、オプショナルの XML 宣言内のバージョン情報に対して 発生します。 パーサーは、バージョン値 (403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例では `1.0`) を含むテキストのアドレスと長さを渡します。

encoding_declaration

このイベントは、XML 宣言内でオプショナルのエンコード宣言に対して 発生します。 パーサーは、エンコード方式値が入ったテキストのアドレスと長さを 渡します。

standalone_declaration

このイベントは、XML 宣言内でオプショナルのスタンドアロン宣言に 対して発生します。 パーサーは、スタンドアロン値 (403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例では yes) を含むテキストのアドレスと長さを渡します。

document_type_declaration

このイベントは、パーサーが文書タイプ宣言を検出したときに発生します。 文書タイプ宣言は文字シーケンス <!DOCTYPE で始まり > 文字で終わります。 その間には、かなり複雑な文法規則で記述された内容が含まれます。 パーサーは、開始と終了の文字シーケンスを含む宣言全体が含まれるテキストのアドレスと長さを渡します。 これは、XML テキストに区切り文字が含まれる唯一のイベントです。 403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例には、文書タイプ宣言が含まれていません。

end_of_document

このイベントは、文書の構文解析が完了したときに 1 回発生します。 パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。

start_of_element

このイベントは、エレメント開始タグ、または空エレメント・タグ ごとに 1 回発生します。 パーサーは、エレメント名が入ったテキストのアドレスと長さ、および適用される名前空間情報を 渡します。 403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例において構文解析中に最初に発生する start_of_element イベントの場合、このストリングは sandwich です。

attribute_name

このイベントは、有効な名前をパーサーが認識した後で、エレメント開始タグまたは空エレメント・タグにおける属性ごとに発生します。 パーサーは、属性名が入ったテキストのアドレスと長さ、および適用される名前空間情報を 渡します。 403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例における唯一の属性名は type です。

attribute_characters

このイベントは、属性値ごとに発生します。 パーサーは、フラグメントが入ったテキストのアドレスと長さを 渡します。 属性値は通常、次のように複数の行に分割されている場合でも ただ 1 つのストリングで構成されます。

```
<element attribute="This attribute value is
split across two lines"/>
```

また、パーサーはフラグ・バイト (内容の一部を形成する追加文字を次のイベントが提供するのかどうかを示す) も渡します。 これは、開始タグと終了タグの間に多くのデータが存在する場合に当てはまります。

end_of_element

このイベントは、エレメント終了タグ、または空エレメント・タグ ごとに、パーサーがタグの終了不等号括弧を認識したときに 1 回発生します。 パーサーは、エレメント名が入ったテキストのアドレスと長さ、および適用される名前空間情報を 渡します。

start_of_CDATA_section

このイベントは、 CDATA セクションの開始時に発生する。 CDATA セクションは、文字列 で始まり、文字列 で終わる。そして、そうでなければ XML マークアップとして認識される文字を含むテキストブロックを「エスケープ」するために使われる。 <![CDATA[ ]]> パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。 このイベントの後、パーサーはこれらの区切り文字の間にある CDATA セクションの内容を、1 つ以上の content-characters イベントとして渡す。 403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示す例では、content-characters イベントはテキスト We should add a <relish> element in future! を渡される。

end_of_CDATA_section

このイベントは、パーサーが CDATA セクションの終わりを認識したときに発生する。 パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。

content_characters

このイベントは、XML 文書の「本体」である、エレメントの開始タグと 終了タグの間にある文字データを表します。 パーサーは、データを含むテキストのアドレスおよび長さを渡します。 データは通常、次のように複数の行に分割されている場合でも、単一のストリングのみで構成されます。

```
<element1>This character content is
split across two lines</element1>
```

また、パーサーはフラグ・バイト (内容の一部を形成する追加文字を次のイベントが提供するのかどうかを示す) も渡します。 これは、開始タグと終了タグの間に多くのデータが存在する場合に当てはまります。

パーサーはまた、content_characters イベントを使って、 CDATA セクションのテキストをアプリケーションに渡す。

processing_instruction

処理命令 (PI) を使用すると、XML 文書にアプリケーション用の特別な命令を 含めることができます。 このイベントは、パーサーが PI 開始文字シーケンス <? に続く名前を認識したときに発生します。 さらにこのイベントは、処理命令 (PI) ターゲットに続く、PI 終了文字シーケンス ?> の直前までのデータを対象とします。 データの末尾にある空白文字は含まれますが、先頭にある空白文字は含まれません。 パーサーは、ターゲットが含まれているテキスト (403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例では spread) のアドレスと長さ、およびデータが含まれているテキスト (例では `please use real mayonnaise`) のアドレスと長さを渡します。

また、パーサーはフラグ・バイト (内容の一部を形成する追加文字を次のイベントが提供するのかどうかを示す) も渡します。 これは、開始タグと終了タグの間に多くのデータが存在する場合に当てはまります。

コメント

このイベントは、XML 文書内のコメントに対して発生します。 パーサーは、開始コメント区切り文字 <!-- と終了コメント区切り文字 --> の間にあるテキストのアドレスと長さを渡します。 403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されている例では、`This document is just an example` が唯一のコメントのテキストです。

また、パーサーはフラグ・バイト (内容の一部を形成する追加文字を次のイベントが提供するのかどうかを示す) も渡します。 これは、開始タグと終了タグの間に多くのデータが存在する場合に当てはまります。

namespace_declare

このイベントは、XML 文書内の名前空間宣言に対して発生します。 パーサーは、名前空間接頭部のアドレスと長さ (存在する場合)、および名前空間 URI のアドレスと長さを渡します。 名前空間接頭部が存在しない場合には、渡される長さはゼロであり、アドレスの値は使用されません。 PLIXSAXA 、 PLISAXB 組み込みサブルーチンには対応するイベントはない。

end_of_input

このイベントは、パーサーが現在の入力バッファーの最後に到達すると常に発生します。 パーサーは (BYVALUE ユーザートークンとともに) 2 つの BYADDR パラメータを渡す。 このイベントとコンテンツ文字イベントは、 BYADDR パラメータを持つ唯一のイベントであるが、呼び出されたイベントが変更すべきパラメータを持つ唯一のイベントであることに注意されたい。 PLIXSAXA 、 PLISAXB の組み込みサブルーチンには対応するイベントはなく、 PLISAXC 、 PLISAXD が任意のサイズの XML 文書を解析できるのはこのイベントのおかげである。

unresolved_reference

このイベントは、XML 文書内の未解決の参照に対して発生します。 パーサーは、未解決の参照のアドレスおよび長さを渡します。

除外、

XML 文書の処理中にエラーを検出すると、パーサーはこのイベントを 生成します。

イベント関数に渡されるパラメーター

これらの関数はすべて、パーサーへのリターンコードである BYVALUE FIXED BIN(31)値を返さなければならない。 ゼロ以外の値が返された場合は、パーサーは終了します。

これらの関数はすべて、第一引数として BYVALUE POINTER 。 このポインターは本来、組み込み関数に 2 番目の引数として渡されるトークン値です。

以下の例外を除き、すべての関数には、イベントのテキスト要素のアドレスと長さを与える BYVALUE POINTER と BYVALUE FIXED BIN(31) も渡される。 以下の関数とイベントは例外です。

start_of_document
ユーザー・トークン以外の引数は渡されません。

end_of_document
ユーザー・トークン以外の引数は渡されません。

start_of_CDATA
ユーザー・トークン以外の引数は渡されません。

end_of_CDATA
ユーザー・トークン以外の引数は渡されません。

start_of_element
通常の 3 つのパラメーターに加えて、次の 4 つの引数が渡されます。

1. 名前空間プレフィックスのアドレスである BYVALUE POINTER
2. BYVALUE FIXED BIN(31)。名前空間プレフィックスの長さ
3. 名前空間 uri のアドレスである BYVALUE POINTER
4. BYVALUE FIXED BIN(31)。これは名前空間 uri の長さである

end_of_element
通常の 3 つのパラメーターに加えて、次の 4 つの引数が渡されます。

1. 名前空間プレフィックスのアドレスである BYVALUE POINTER
2. BYVALUE FIXED BIN(31)。名前空間プレフィックスの長さ
3. 名前空間 uri のアドレスである BYVALUE POINTER
4. BYVALUE FIXED BIN(31)。これは名前空間 uri の長さである

attribute_name
通常の 3 つのパラメーターに加えて、次の 4 つの引数が渡されます。

1. 名前空間プレフィックスのアドレスである BYVALUE POINTER
2. BYVALUE FIXED BIN(31)。名前空間プレフィックスの長さ
3. 名前空間 uri のアドレスである BYVALUE POINTER
4. BYVALUE FIXED BIN(31)。これは名前空間 uri の長さである

attribute_characters
通常の 3 つのパラメーターに加えて、次の 1 つの引数が渡されます。

- 以下の情報を示す BYVALUE ALIGNED BIT(8)フラグバイト：

- 次のイベントに content-charcaters がさらに提供されるかどうか。これは、最初のビットがオンである場合は真です。つまり、このフィールドと '80'BX との AND 演算の結果が非ヌルである場合、これは真です。
- XML に再び変換するときにエスケープしなければならない文字が存在しないかどうか。これは、2 番目のビットがオンの場合は真です。つまり、このフィールドと '40'BX との AND 演算の結果が非ヌルである場合、これは真です。

このエントリーも OPTIONS(NODESCRIPTOR) で宣言しなければならないことに注意。

namespace_declare
ユーザー・トークンに加えて、次の 4 つの引数が渡されます。

1. 名前空間プレフィックスのアドレスである BYVALUE POINTER
2. BYVALUE FIXED BIN(31)。名前空間プレフィックスの長さ
3. 名前空間 uri のアドレスである BYVALUE POINTER
4. BYVALUE FIXED BIN(31)。これは名前空間 uri の長さである

content_characters
通常の 3 つのパラメーターに加えて、次の 1 つの引数が渡されます。

- BYVALUE ALIGNED BIT(8)フラグバイトで、以下の情報を示す：
  - 次のイベントに content-charcaters がさらに提供されるかどうか。これは、最初のビットがオンである場合は真です。つまり、このフィールドと '80'BX との AND 演算の結果が非ヌルである場合、これは真です。
  - XML に再び変換するときにエスケープしなければならない文字が存在しないかどうか。これは、2 番目のビットがオンの場合は真です。つまり、このフィールドと '40'BX との AND 演算の結果が非ヌルである場合、これは真です。

このエントリーも OPTIONS(NODESCRIPTOR) で宣言しなければならないことに注意。

end_of_input
ユーザー・トークンに加えて、次の 2 つの引数が渡されます。

1. 次の入力バッファーのアドレスを示す BYADDR POINTER
2. 次の入力バッファーの長さを示す BYADDR FIXED BIN(31)

processing_instruction
ユーザー・トークンに加えて、次の 5 つの追加引数が渡されます。

1. BYVALUE POINTER ターゲットテキストのアドレス
2. BYVALUE FIXED BIN(31)はターゲット・テキストの長さである
3. データ・テキストのアドレスである BYVALUE POINTER
4. データ・テキストの長さである BYVALUE FIXED BIN(31)
5. BYVALUE ALIGNED BIT(8) 以下の情報を示すフラグバイト：
   - 次のイベントに content-charcaters がさらに提供されるかどうか。これは、最初のビットがオンである場合は真です。つまり、このフィールドと '80'BX との AND 演算の結果が非ヌルである場合、これは真です。
   - XML に再び変換するときにエスケープしなければならない文字が存在しないかどうか。これは、2 番目のビットがオンの場合は真です。つまり、このフィールドと '40'BX との AND 演算の結果が非ヌルである場合、これは真です。

   このエントリーも OPTIONS(NODESCRIPTOR) で宣言しなければならないことに注意。

comment
通常の 3 つのパラメーターに加えて、次の 1 つの引数が渡されます。

- BYVALUE ALIGNED BIT(8)フラグバイトで、以下の情報を示す：

– 次のイベントに content-charcaters がさらに提供されるかどうか。これは、最初のビットがオンである場合は真です。つまり、このフィールドと '80'BX との AND 演算の結果が非ヌルである場合、これは真です。
– XML に再び変換するときにエスケープしなければならない文字が存在しないかどうか。これは、2 番目のビットがオンの場合は真です。つまり、このフィールドと '40'BX との AND 演算の結果が非ヌルである場合、これは真です。

このエントリーも OPTIONS(NODESCRIPTOR) で宣言しなければならないことに注意。

exception
ユーザー・トークンに加えて、次の 3 つの引数が渡されます。

1. BYVALUE FIXED BIN （31） ： 文書内の問題のあるテキストのバイトオフセット
2. 例外のリターンコードである BYVALUE FIXED BIN(31)
3. 例外の理由コードである BYVALUE FIXED BIN(31)

イベントにおける差異

以下のイベントは PLISAXA と PLISAXB のイベント構造の一部であるが、PLISAXC と PLISAXD にはない：

- attribute_predefined_reference
- attribute_character_reference
- content_predefined_reference
- content_character_reference
- unknown_attribute_reference
- unknown_content_reference
- start_of_prefix_mapping
- end_of_prefix_mapping

以下のイベントは、PLISAXA と PLISAXB のイベント構造には含まれないが、PLISAXC と PLISAXD に含まれる：

- namespace_declare
- unresolved_reference
- end_of_input

PLISAXA 、 PLISAXB、 PLISAXC 、 PLISAXD に共通するイベントのいくつかは、（遍在するユーザートークンを除けば）異なるパラメータが渡される：

start_of_document
パラメーターが渡されない

start_of_element
名前空間データも渡される

end_of_element
名前空間データも渡される

attribute_name
名前空間データも渡される

attribute_characters
フラグ・バイトも渡される

start_of_cdata
パラメーターが渡されない

end_of_cdata
パラメーターが渡されない

content_characters
フラグ・バイトも渡される

processing_instruction
フラグ・バイトも渡される

コメント
フラグ・バイトも渡される

除外、
エラー ID の代わりに戻りコードおよび理由コードが渡される

XML 文書のコード化文字セット

その PLISAXC そして PLISAXD 組み込みサブルーチンは XML 文書のみをサポートします。 WIDECHAR UTF-16 でエンコードされた UCHAR、 UTF-8 でエンコードされた UCHAR、または UTF-8 もしくはこのセクションにリストされている明示的にサポートされているシングルバイト文字セットのいずれかでエンコードされた CHARACTER です。

パーサーは、XML 文書のエンコード方式に関する情報ソースを 3 つまで使用し、 これらのソース間で矛盾を検出した場合は、次のように例外 XML イベントを シグナル通知します。

1. パーサーは、文書の最初の文字を検査することによって 文書の基本エンコードを判別します。
2. ステップ 1 が正常に完了した場合、パーサーはエンコード宣言を 検索します。
3. 最後に、 PLISAXC または PLISAXD 組み込みサブルーチン呼び出しのコードページ値を参照する。 このパラメータが省略された場合、デフォルトは、明示的に指定した、またはデフォルトで指定された CODEPAGE コンパイラ・オプション値によって提供される値になる。

XML 文書が、サポートされているコードページのいずれかを指定するエンコーディング宣言を含む XML 宣言で始まる場合、パーサは、基本的な文書エンコーディング、または PLISAXC または PLISAXD 組み込みサブルーチンからのエンコーディング情報のいずれとも衝突しなければ、そのエンコーディング宣言を尊重します。 XML 文書に XML 宣言が全くない場合、または XML 宣言にエンコーディング宣言が省略されている場合、パーサは、基本的な文書のエンコーディングと衝突しない限り、 PLISAXC または PLISAXD 組み込みサブルーチンからのエンコーディング情報を使用して文書を処理します。

サポートされるコード・ページ

以下の表において、最初の数字は Euro Country Extended Code Page (ECECP) を、2 番目の数字は Country Extended Code Page (CECP) を表している。

CCSID	説明
01208	ユニコード UTF-8
01047	Latin 1/オープン・システム
01140、00037	米国、カナダなど
01141、00273	オーストリア、ドイツ
01142、00277	デンマーク、ノルウェー
01143、00278	フィンランド、スウェーデン
01144、00280	イタリア
01145、00284	スペイン、ラテンアメリカ (スペイン語)
01146、00285	英国
01147、00297	フランス
01148、00500	国際
01149、00871	アイスランド

コード・ページの指定

ドキュメントが XML 宣言の中にエンコーディング宣言を含んでいない場合、あるいは XML 宣言を全く持っていない場合、パーサーは PLISAXC または PLISAXD 組み込みサブルーチン呼び出しによって提供されたエンコーディング情報をドキュメントの基本エンコーディングと共に使用します。

ほとんどの XML 文書の最初にある XML 宣言の中で、文書のエンコード情報を 指定することもできます。エンコード宣言を含む XML 宣言の例を以下に示します。

```
<?xml version="1.0" encoding="ibm-1140"?>
```

XML ドキュメントにエンコーディング宣言が含まれている場合は、 PLISAXC または PLISAXD 組み込みサブルーチンが提供するエンコーディング情報、およびドキュメントの基本エンコーディングと一致していることを確認してください。 エンコーディング宣言、 PLISAXC または PLISAXD 組み込みサブルーチンによって提供されるエンコーディング情報、ドキュメントの基本エンコーディングの間に矛盾がある場合、パーサーは例外 XML イベントを通知します。

エンコード宣言は、番号または別名を使用して指定できます。

番号を使用してエンコード宣言を指定

CCSID （先頭のゼロの有無は問わない）番号の前に、以下のいずれかの文字（大文字・小文字の混在可）を指定することができる：

- IBM_
  IBM-
- CP
  CP_
  CP-
- CCSID_
  CCSID-

別名を使用してエンコード宣言を指定

以下のサポート対象の別名 (大文字と小文字の組み合わせは自由) を任意に使用できます。

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
813	ISO-8859-7, ISO_8859-7
819	ISO-8859-1, ISO_8859-1
920	ISO-8859-9, ISO_8859-9
1200	UTF-16

例外

例外イベントが発生した場合、そのイベントに渡される理由コードおよび戻りコードは、XML System Services パーサーからのものです。 これらの戻りコードおよび理由コードの意味については、このパーサーに付属の資料に説明があります。

妥当性検査を伴う XML 文書の構文解析

PLISAXD 組み込みサブルーチンは、 PLISAXC と同じ方法で XML 文書を解析するだけでなく、受信 XML 文書が受信 XML スキーマで指定された一連のルールに適合しているかどうかも判定する。

PLISAXD 組み込みサブルーチンを使用する場合、XML 検証に使用される受信スキーマは、 Optimized Schema Representation (OSR) として知られる前処理済みの形式でなければなりません。

以下のトピックでは、XML スキーマについて、および XML スキーマに対して OSR を構築する方法について説明します。 PLISAXD 組み込みサブルーチンを使って XML 文書を検証付きで解析する例については、419 ページの『PLISAXD 組み込みサブルーチンの使用例』 を参照。

XML スキーマ

XML スキーマは、W3C により定義された、XML 文書の構造と内容を記述および制約するためのメカニズムです。

データ型および名前空間のサポートを通じて、XML スキーマは、XML エレメントおよび属性の標準構造を提供することができます。 そのため、XML スキーマ (それ自体が XML で表される) は、指定されたタイプ (在庫品目など) の XML 文書のクラスを効率的に定義できます。

次の XML 文書の例では、在庫管理の目的で品目を記述しています。

```
      '<?xml version="1.0" standalone="yes"?>'
   || '<!--Document for stock keeping example-->'
   || '<stockItem itemNumber="453-SR">'
   || '<itemName>Stainless steel rope thimbles</itemName>'
   || '<quantityOnHand>23</quantityOnHand>'
   || '<stockItem>';
```

在庫管理の文書例は、適切な形式であり、stock.xsd という次のスキーマに従って有効です。 (各行の前にある番号はスキーマの一部ではなく、スキーマの後にある説明で使用します。)

```
1. <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2.
3. <xsd:element name="stockItem" type="stockItemType"/>
4.
5. <xsd:complexType name="stockItemType">
6.  <xsd:sequence>
7.   <xsd:element name="itemName" type="xsd:string" minOccurs="0"/>
8.   <xsd:element name="quantityOnHand">
9.    <xsd:simpleType>
10.    <xsd:restriction base="xsd:nonNegativeInteger">
11.      <xsd:maxExclusive value="100"/>
12.     </xsd:restriction>
13.    </xsd:simpleType>
14.   </xsd:element>
15.  </xsd:sequence>
16.  <xsd:attribute name="itemNumber" type="SKU" use="required"/>
17. </xsd:complexType>
18.
19. <xsd:simpleType name="SKU">
20.  <xsd:restriction base="xsd:string">
21.   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
22.  </xsd:restriction>
23. </xsd:simpleType>
24.
25. </xsd:schema>
```

スキーマは、ルート・エレメントが stockItem であることを宣言しています (行 3)。 このルート・エレメントには、SKU 型の必須属性 itemNumber があり (行 16)、次のような他のエレメントのシーケンスが含まれています (行 6 から 15)。

- オプションのストリング型の itemName エレメント (行 7)
- nonNegativeInteger 型に基づいた 1 から 99 までに制約された範囲内の必須エレメント quantityOnHand (行 8 から 14)

型宣言はインライン化し、無名化することができます (行 9 から 13)。これには、quantityOnHand エレメントに正しい値を指定するための maxExclusive ファセットが含まれています。

一方、itemNumber 属性については、名前付き型 SKU が別途宣言されています (行 19 から 23)。これには、正規表現構文を使用して、その型の正しい値が 3 桁の数字、ハイフン (負符号)、2 つの大文字がこの順序で構成されるように指定する、パターン・ファセットが含まれています。

を作る OSR

テキスト形式のスキーマから OSR 形式のスキーマを生成するには、 z/OS UNIX コマンド **xsdosrg** を使用する。このコマンドは、 z/OS UNIX システム・サービス で提供されている OSR ジェネレーターを呼び出す。

例えば、stock.xsd ファイル内のテキスト形式スキーマを stock.osr ファイル内のプリプロセス形式のスキーマに変換するには、以下の z/OS UNIX コマンドを使用できます。

```
xsdosrg -v –o /u/HLQ/xml/stock.osr  /u/HLQ/xml/stock.xsd
```

/u/HLQ/xml/ は、stock.osr ファイルおよび stock.xsd ファイルがあるディレクトリーです。

生成された OSR ファイルを PDS にコピーしたい場合は、 z/OS UNIX **cp** 。 MVS とは データ・セット名を指定するには、名前の前にダブルスラッシュ (//) を付けます。 例えば、zFS ファイル stock.osr を、レコード長 80 の HLQ.XML.OSR という名前の固定ブロック・レコード・フォーマット PDS にコピーするには、次のコマンドを使用できます。

```
cp –p /u/HLQ/xml/stock.osr “//’HLQ.XML.OSR(STOCK)’”
```

PDS の完全修飾名を省略するには、次のように cp ステートメント内のシングルクォーテーションを削除する：

```
cp –p /u/HLQ/xml/stock.osr “//XML.OSR(STOCK)”
```

詳しくは、「*XML System Services* ユーザーズ・ガイドおよび解説書」を参照してください。

単純な文書での例

このセクションには、 PLISAXC と PLISAXD 組み込みサブルーチンの使い方を説明する 2 つの例が含まれています。

PLISAXC 組み込みサブルーチンの使用例

この例では、 PLISAXC 組み込みサブルーチンの使い方を説明する。

次の例では、403 ページの『#unique_674/unique_674_Connect_42_figxmlc』 に示されているサンプル XML 文書が使用されています。 この例では名前空間を使用しておらず、すべての入力は PLISAXC が最初に呼び出されたときに渡される（その結果、end_of_input イベントは呼び出されないはずである）。

PLISAXC コーディング例 - 型宣言

```
saxtest: package exports(saxtest);

  define alias event
      limited entry( pointer, pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_with_flag
      limited entry( pointer, pointer, fixed bin(31),
                     bit(8) aligned )
      returns( byvalue fixed bin(31) )
      options( nodescriptor byvalue linkage(optlink) );

  define alias event_with_namespace
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_without_data
      limited entry( pointer )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_pi
```

```
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_namespace_dcl
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_exception
        limited entry( pointer, fixed bin(31),
                       fixed bin(31),
                       fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_end_of_input
        limited entry( pointer,
                       pointer byaddr,
                       fixed bin(31) byaddr )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );
```

PLISAXC コーディング例 - イベント構造

```
  saxtest: proc options( main );


    dcl
      1 eventHandler static

        ,2 e01 type event_without_data
               init( start_of_document )

        ,2 e02 type event
               init( version_information )

        ,2 e03 type event
               init( encoding_declaration )

        ,2 e04 type event
               init( standalone_declaration )

        ,2 e05 type event
               init( document_type_declaration )

        ,2 e06 type event_without_data
               init( end_of_document )

        ,2 e07 type event_with_namespace
               init( start_of_element )

        ,2 e08 type event_with_namespace
               init( attribute_name )

        ,2 e09 type event_with_flag
               init( attribute_characters )

        ,2 e10 type event_with_namespace
               init( end_of_element )

        ,2 e11 type event_without_data
               init( start_of_CDATA )

        ,2 e12 type event_without_data
               init( end_of_CDATA )

        ,2 e13 type event_with_flag
               init( content_characters )

        ,2 e14 type event_pi
               init( processing_instruction )

        ,2 e15 type event_with_flag
               init( comment )

        ,2 e16 type event_namespace_dcl
               init( namespace_declare )
```

```
        ,2 e17 type event_end_of_input
               init( end_of_input )

        ,2 e18 type event
               init( unresolved_reference )

        ,2 e19 type event_exception
               init( exception )
        ;
```

PLISAXC コーディング例 - メインルーチン

```
    dcl token       char(8);

    dcl xmlDocument char(4000) var;

    xmlDocument =
       '<?xml version="1.0" standalone="yes"?>'
    || '<!--This document is just an example-->'
    || '<sandwich>'
    || '<baker type="baker"s best"/>
    || '<?spread please use real mayonnaise ?>'
    || '<meat>Ham &amp; turkey</meat>'
    || '<filling>Cheese, lettuce, tomato, etc.</filling>'
    || '<![CDATA[We should add a <relish> element in future!]]>'.
    || '</sandwich>'
    || ' ';


    call plisaxc( eventHandler,
                  addr(token),
                  addrdata(xmlDocument),
                  length(xmlDocument) );

  end;
```

PLISAXC コーディング例 - イベントルーチン

```
  dcl chars char(32000) based;

  start_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' length=' || tokenlength );

    return(0);
  end;

  version_information:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  encoding_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
```

```
    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  standalone_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

```
  document_type_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  namespace_declare:
    proc( userToken, nsPrefix, nsPrefixLength,
                     nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() ) );
    put skip list( 'prefix = '
     || ' <' || substr(nsPrefix->chars,1,nsPrefixlength ) || '>' );
    put skip list( 'Uri = '
     || ' <' || substr(nsUri->chars,1,nsUrilength ) || '>' );

    return(0);
  end;

  end_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  start_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
```

```
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  attribute_name:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  end_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  content_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  attribute_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );
```

```
    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  start_of_CDATA:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

  end_of_CDATA:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  processing_instruction:
    proc( userToken,
          piTarget, piTargetLength,
          piData, piDataLength,
          flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl piTarget       pointer;
    dcl piTargetLength fixed bin(31);
    dcl piData         pointer;
    dcl piDataLength   fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  comment:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

if flags = ''b then;
```

```
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  unresolved_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  exception:
    proc( userToken, currentOffset, return_code, reason_code )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl currentOffset  fixed bin(31);
    dcl return_code    fixed bin(31);
    dcl reason_code    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' return_code =' || hex(return_code)
     || ', reason_code =' || hex(reason_code)
     || ', offset =' || currentOffset );

    return(0);
  end;

  end_of_input:
    proc( userToken, addr_xml, length_xml )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl addr_xml       byaddr pointer;
    dcl length_xml     byaddr fixed bin(31);

    return(1);
  end;
end;
```

PLISAXC coding example - event routines (continued)

前のプログラムで生成される出力は次のとおりです。

PLISAXC コーディング例 - プログラム出力

```
  start_of_document
  version_information <1.0>
  standalone_declaration <yes>
  comment <This document is just an example>
  start_of_element <sandwich>
  prefix =  <>
  Uri =  <>
  start_of_element <bread>
  prefix =  <>
  Uri =  <>
  attribute_name <type>
  prefix =  <>
  Uri =  <>
  attribute_characters <baker"s best>
  end_of_element <bread>
  prefix =  <>
  Uri =  <>
  processing_instruction <spread>
```

```
piData =  <please use real mayonnaise >
start_of_element <meat>
prefix =  <>
Uri =  <>
content_characters <Ham & turkey>
end_of_element <meat>
prefix =  <>
Uri =  <>
start_of_element <filling>
prefix =  <>
Uri =  <>
content_characters <Cheese, lettuce, tomato, etc.>
!!flags = 01000000
end_of_element <filling>
prefix =  <>
Uri =  <>
start_of_cdata
content_characters <We should add a <relish> element in future >
end_of_cdata
end_of_element <sandwich>
prefix =  <>
Uri =  <>
end_of_document
```

PLISAXD 組み込みサブルーチンの使用例

この例では、 PLISAXD 組み込みサブルーチンの使い方を説明する。

こ の例では、 「 サンプル XML 文書 」 で示 し たサンプル XML 文書 と、 412 ページの『PLISAXC 組み込みサブルーチンの使用例』 の例で引用 し た XML ス キーマ を使用 し ます。

この例には、同じ stock.osr スキーマに対する 8 個の XML ファイルの妥当性検査が含まれています。 次の出力で、スキーマに対する妥当性検査が失敗した saxdtest プログラム内の XML 文書を確認できます。

PLISAXD 組み込みサブルーチンでは、XML スキーマ・ファイルをバッファに読み込む必要があります。 次の例の OSR ファイルは PDS 内にあります。 OSR バッファーの初期サイズは 4096 に設定されています。 それより大きいサイズの OSR ファイルがある場合は、それに応じて OSR バッファーの初期サイズを大きくすることができます。

インバウンド・スキーマ・ファイルが zFS ファイル内にある場合は、次のコードを使用して、OSR ファイルをバッファーへ読み取ることができます。

```
dcl osrin file input stream environment(u);
dcl fileddint builtin;
dcl fileread builtin;

/* Read the zFS OSR file into buffer*/

open file(osrin);
osr_length = fileddint( osrin, 'filesize');
osr_ptr = allocate(osr_length);
rc = fileread(osrin,osr_ptr,osr_length);
```

PDS 内の OSR を使用してプログラムを実行するには、JCL で次の DD ステートメントを指定します。

```
//OSRIN DD DSN=HLQ.XML.OSR(STOCK),DISP=SHR
```

関連付けられた ddname OSRIN が zFS ファイルである場合は、代わりに次の JCL ステートメントを使用します。

```
//OSRIN DD PATH=&ldquo;/u/HLQ/xml/stock.osr&rdquo;
```

PLISAXD コーディング例 - イベントルーチン

```
saxdtest: package exports(saxdtest);
  /*************************************************************/
  /* saxdtest: Test PL/I XML validation support                */
  /* expected output:                                          */
  /*                                                           */
  /*   SAXDTEST: PL/I XML Validation sample                    */
  /*   SAXDTEST: Document Successfully parsed                  */
  /*   SAXDTEST: Document Successfully parsed                  */
```

```
    /*   Invalid: missing attribute itemNumber.                 */
    /*   exception return_code =00000018, reason_code =8613     */
    /*   Invalid: unexpected attribute warehouse.               */
    /*   exception return_code =00000018, reason_code =8612     */
    /*   Invalid: illegal attribute value 123-Ab.               */
    /*   exception return_code =00000018, reason_code =8809     */
    /*   Invalid: missing element quantityOnHand.               */
    /*   exception return_code =00000018, reason_code =8611     */
    /*   Invalid: unexpected element comment.                   */
    /*   exception return_code =00000018, reason_code =8607     */
    /*   Invalid: out-of-range element value 100                */
    /*   exception return_code =00000018, reason_code =8803     */
    /*                                                          */
    /************************************************************/

    define alias event
        limited entry( pointer, pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_with_flag
        limited entry( pointer, pointer, fixed bin(31),
                       bit(8) aligned )
        returns( byvalue fixed bin(31) )
        options( nodescriptor byvalue linkage(optlink) );
```

```
    define alias event_with_namespace
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_without_data
        limited entry( pointer )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_pi
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_namespace_dcl
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_exception
        limited entry( pointer, fixed bin(31),
                       fixed bin(31),
                       fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_end_of_input
        limited entry( pointer,
                       pointer byaddr,
                       fixed bin(31) byaddr )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );
```

PLISAXD coding example - event routines (continued)

```
  saxdtest: proc options( main );

    dcl
      1 eventHandler static

        ,2 e01 type event_without_data
               init( start_of_document )

        ,2 e02 type event
               init( version_information )

        ,2 e03 type event
               init( encoding_declaration )
```

```
        ,2 e04 type event
               init( standalone_declaration )

        ,2 e05 type event
               init( document_type_declaration )

        ,2 e06 type event_without_data
               init( end_of_document )

        ,2 e07 type event_with_namespace
               init( start_of_element )

        ,2 e08 type event_with_namespace
               init( attribute_name )

        ,2 e09 type event_with_flag
               init( attribute_characters )

        ,2 e10 type event_with_namespace
               init( end_of_element )

        ,2 e11 type event_without_data
               init( start_of_CDATA )

        ,2 e12 type event_without_data
               init( end_of_CDATA )

        ,2 e13 type event_with_flag
               init( content_characters )

        ,2 e14 type event_pi
               init( processing_instruction )

        ,2 e15 type event_with_flag
               init( comment )

        ,2 e16 type event_namespace_dcl
               init( namespace_declare )

        ,2 e17 type event_end_of_input
               init( end_of_input )

        ,2 e18 type event
               init( unresolved_reference )

        ,2 e19 type event_exception
               init( exception )
        ;
```

PLISAXD coding example - event routines (continued)

```
  saxdtest: package exports(saxdtest);
    /*************************************************************/
    /* saxdtest: Test PL/I XML validation support                */
    /* expected output:                                          */
    /*                                                           */
    /*   SAXDTEST: PL/I XML Validation sample                    */
    /*   SAXDTEST: Document Successfully parsed                  */
    /*   SAXDTEST: Document Successfully parsed                  */
    /*   Invalid: missing attribute itemNumber.                  */
    /*   exception return_code =00000018, reason_code =8613      */
    /*   Invalid: unexpected attribute warehouse.                */
    /*   exception return_code =00000018, reason_code =8612      */
    /*   Invalid: illegal attribute value 123-Ab.                */
    /*   exception return_code =00000018, reason_code =8809      */
    /*   Invalid: missing element quantityOnHand.                */
    /*   exception return_code =00000018, reason_code =8611      */
    /*   Invalid: unexpected element comment.                    */
    /*   exception return_code =00000018, reason_code =8607      */
    /*   Invalid: out-of-range element value 100                 */
    /*   exception return_code =00000018, reason_code =8803      */
    /*                                                           */
    /*************************************************************/

    define alias event
        limited entry( pointer, pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_with_flag
        limited entry( pointer, pointer, fixed bin(31),
```

```
                        bit(8) aligned )
        returns( byvalue fixed bin(31) )
        options( nodescriptor byvalue linkage(optlink) );
```

```
    define alias event_with_namespace
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_without_data
        limited entry( pointer )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_pi
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_namespace_dcl
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_exception
        limited entry( pointer, fixed bin(31),
                       fixed bin(31),
                       fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_end_of_input
        limited entry( pointer,
                       pointer byaddr,
                       fixed bin(31) byaddr )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );
```

PLISAXD coding example - event routines (continued)

```
  saxdtest: proc options( main );

    dcl
      1 eventHandler static

        ,2 e01 type event_without_data
               init( start_of_document )

        ,2 e02 type event
               init( version_information )

        ,2 e03 type event
               init( encoding_declaration )

        ,2 e04 type event
               init( standalone_declaration )

        ,2 e05 type event
               init( document_type_declaration )

        ,2 e06 type event_without_data
               init( end_of_document )

        ,2 e07 type event_with_namespace
               init( start_of_element )

        ,2 e08 type event_with_namespace
               init( attribute_name )

        ,2 e09 type event_with_flag
               init( attribute_characters )

        ,2 e10 type event_with_namespace
               init( end_of_element )

        ,2 e11 type event_without_data
```

```
               init( start_of_CDATA )

        ,2 e12 type event_without_data
               init( end_of_CDATA )

        ,2 e13 type event_with_flag
               init( content_characters )

        ,2 e14 type event_pi
               init( processing_instruction )

        ,2 e15 type event_with_flag
               init( comment )

        ,2 e16 type event_namespace_dcl
               init( namespace_declare )

        ,2 e17 type event_end_of_input
               init( end_of_input )

        ,2 e18 type event
               init( unresolved_reference )

        ,2 e19 type event_exception
               init( exception )
        ;
```

PLISAXD coding example - event routines (continued)

```
    dcl token       char(45);

    dcl rc             fixed bin(31);
    dcl i              fixed bin(31);
    dcl xml_document(8) char(300) var;
    dcl xml_valid_msg(8) char(45) var;
    dcl osr_ptr        pointer;
    dcl record         char(80);
    dcl osr_index      fixed bin;
    dcl osr_buf_tail   fixed bin;
    dcl temp_osr       pointer;
    dcl buf_size       fixed bin(31) init(4096);
    dcl rec_size       fixed bin(31);
    dcl osr_length     fixed bin(31);
    dcl buf_length     fixed bin(31);
    dcl eof fixed      bin(31) init(0);
    dcl osrin          file input;

    on endfile(osrin) begin;
      eof = 1;
    end;

    /* read the entire PDS osr file into the buffer */

    put skip list ('SAXDTEST: PL/I XML Validation sample ');

    osr_length = buf_size;
    osr_index = 0;
    osr_buf_tail = 0;
    rec_size = length(record);

    osr_ptr = allocate(osr_length);

    do while (eof = 0 );

      read file(osrin) into(record);
      osr_buf_tail += rec_size;
      if osr_buf_tail &gt; buf_size then
        do;
          buf_length = osr_length;
          osr_length +=buf_size;
          temp_osr = allocate(osr_length);

          call plimove(temp_osr, osr_ptr, buf_length);
          call plifree(osr_ptr);
          osr_ptr = temp_osr;
          osr_buf_tail = rec_size;

          call plimove(osr_ptr+osr_index, addr(record), rec_size);
          osr_index += rec_size;
        end;
      else
```

```
        do;
          call plimove(osr_ptr+osr_index, addr(record), rec_size);
          osr_index  +=rec_size;
        end;

    end;
```

PLISAXD coding example - event routines (continued)

```
    /* Valid XMLFILE */
    xml_document(1) = '&ltstockItem itemNumber="453-SR"&gt;'
    || '&ltitemName&gt;Stainless steel rope thimbles&lt/itemName&gt;'
    || '&ltquantityOnHand&gt;23&lt/quantityOnHand&gt;'
    || '&lt/stockItem&gt;';

    xml_valid_msg(1) = 'Valid XMLFILE ';
    /* Valid: the ITEMNAME element can be omitted */
    xml_document(2) = '&ltstockItem  itemNumber="453-SR"&gt;'
    || '&ltquantityOnHand&gt;23&lt/quantityOnHand&gt;'
    || '&lt/stockItem&gt;';

    xml_valid_msg(2) = 'Valid: the ITEMNAME element can be omitted.';

    /* Invalid: missing attribute itemNumber */
    xml_document(3) = '&ltstockItem&gt;'
    || '&ltitemName&gt;Stainless steel rope thimbles&lt/itemName&gt;'
    || '&ltquantityOnHand&gt;23&lt/quantityOnHand&gt;'
    || '&lt/stockItem&gt;';

    xml_valid_msg(3) = 'Invalid: missing attribute itemNumber.';

    /* Invalid: unexpected attribute warehouse */
    xml_document(4) = '&ltstockItem itemNumber="453-SR" warehouse="NY"&gt;'
    || '&ltitemName&gt;Stainless steel rope thimbles&lt/itemName&gt;'
    || '&ltquantityOnHand&gt;23&lt/quantityOnHand&gt;'
    || '&lt/stockItem&gt;';

    xml_valid_msg4) = 'Invalid: unexpected attribute warehouse.';

    /* Invalid: illegal attribute value 123-Ab */
    xml_document(5) = '&ltstockItem itemNumber="123-Ab"&gt;'
    || '&ltitemName&gt;Stainless steel rope thimbles&lt/itemName&gt;'
    || '&ltquantityOnHand&gt;23lt;/quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(5) = 'Invalid: illegal attribute value 123-Ab.';

    /* Invalid: missing element quantityOnHand */
    xml_document(6) = '<stockItem itemNumber="074-UN">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '</stockItem>';

    xml_valid_msg(6) = 'Invalid: missing element quantityOnHand.';

    /* Invalid: unexpected element comment */
    xml_document(7) = '<stockItem itemNumber="453-SR">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '<quantityOnHand>1</quantityOnHand>'
    || '<commnet>Nylon bristles</comment>'
    || '</stockItem>';

    xml_valid_msg(7) = 'Invalid: unexpected element comment.';

    /* Invalid: out-of-range element value 100 */
    xml_document(8) = '<stockItem itemNumber="123-AB">'
    || '<itemName>Paintbrush</itemName>'
    || '<quantityOnHand>100</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(8) = 'Invalid: out-of-range element value 100';
```

PLISAXD coding example - event routines (continued)

```
    do i = 1 to hbound(xml_document);;
      token = xml_valid_msg(i);
      call plisaxd( eventHandler,
                    addr(token),
                    addrdata(xml_document(i)),
                    length(xml_document(i)),
                    osr_ptr,
```

```
                    37 );
    end;

    close file(osrin);
    call plifree(osr_ptr);

  end;

  start_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    return(0);
  end;

  version_information:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    return(0);
  end;

  encoding_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    return(0);
  end;

  standalone_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  document_type_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    return(0);
  end;

  end_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( 'SAXDTEST: Document Successfully parsed ');

    return(0);
```

```
  end;

  start_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  attribute_name:
    proc( userToken, xmlToken, tokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;

  attribute_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     ||' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  end_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  start_of_CDATA:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;


    return(0);
  end;

  end_of_CDATA:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    return(0);
  end;

  content_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    if flags = ''b then;
    else

    return(0);
  end;

  processing_instruction:
    proc( userToken,
          piTarget, piTargetLength,
          piData, piDataLength,
          flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl piTarget       pointer;
    dcl piTargetLength fixed bin(31);
    dcl piData         pointer;
    dcl piDataLength   fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  comment:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    if flags = ''b then;
    else
```

```
      put skip list( '!!flags = ' || flags );    return(0);
  end;

  namespace_declare:
    proc( userToken, nsPrefix, nsPrefixLength,
                     nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;

  unresolved_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    return(0);
  end;

  exception:
    proc( userToken, currentOffset, return_code, reason_code )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl currentOffset  fixed bin(31);
    dcl return_code    fixed bin(31);
    dcl reason_code    fixed bin(31);
    dcl validmsg       char(45) based;

    put skip list( userToken -> validmsg);
    put skip list( lowercase( procname() )
     || ' return_code =' || hex(return_code)
     || ', reason_code =' || substr(hex(reason_code),5,4));

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  end_of_input:
    proc( userToken, addr_xml, length_xml )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl addr_xml       byaddr pointer;
    dcl length_xml     byaddr fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

428 ページの『#unique_704/unique_704_Connect_42_saxdxmp2』に、サンプル・プログラムの結果が示されています。 有効でないドキュメントに対して、PLISAXD 組み込みサブルーチンは、リターンコードと理由コードを記載した XML 例外イベントを呼び出します。 それぞれの戻りコードおよび理由コードについて詳しくは、「*XML System Services* ユーザーズ・ガイドおよび解説書」を参照してください。

PLISAXD サンプルからの出力

```
SAXDTEST: PL/I XML Validation sample
SAXDTEST: Document Successfully parsed
SAXDTEST: Document Successfully parsed
Invalid: missing attribute itemNumber.
exception return_code =00000018, reason_code =8613
```

```
Invalid: unexpected attribute warehouse.
exception return_code =00000018, reason_code =8612
Invalid: illegal attribute value 123-Ab.
exception return_code =00000018, reason_code =8809
Invalid: missing element quantityOnHand.
exception return_code =00000018, reason_code =8611
Invalid: unexpected element comment.
exception return_code =00000018, reason_code =8607
Invalid: out-of-range element value 100
exception return_code =00000018, reason_code =8803
```

第 22 章 PLIDUMP の用法

このセクションでは、PLIDUMP を呼び出すのに使用されるダンプ・オプションと構文を記載し、ダンプに含まれていてユーザーがユーザー・ルーチンをデバッグする際に役立つ PL/I に 固有の情報について説明します。

注 : PLIDUMP National Language Support の規格に準拠している。

431 ページの『#unique_709/unique_709_Connect_42_pligen』 は、 ダンプを生成するために を呼び出す ルーチンの例を示している。 z/OS 言語環境プログラム PLIDUMP PL/I この例では、メインルーチン PLIDMP が PLIDMPA を呼び出し、次に PLIDMPB を呼び出す。 PLIDUMP への呼び出しは、ルーチン PLIDMPB で行われる。

例 PL/I ルーチンの呼び出し PLIDUMP

```
%PROCESS MAP SOURCE STG LIST OFFSET LC(101);
 PLIDMP: PROC  OPTIONS(MAIN) ;

   Declare   (H,I) Fixed bin(31) Auto;
   Declare   Names Char(17) Static init('Bob Teri Bo Jason');
   H = 5;  I = 9;
   Put skip list('PLIDMP Starting');
   Call PLIDMPA;

     PLIDMPA:  PROC;
       Declare  (a,b)  Fixed bin(31)   Auto;
       a = 1;  b = 3;
       Put skip list('PLIDMPA Starting');
       Call PLIDMPB;

         PLIDMPB:  PROC;
           Declare  1 Name  auto,
                      2  First   Char(12) Varying,
                      2  Last    Char(12) Varying;
           First = 'Teri';
           Last  = 'Gillispy';
           Put skip list('PLIDMPB Starting');
           Call PLIDUMP('TBFC','PLIDUMP called from procedure PLIDMPB');
           Put Data;
         End PLIDMPB;

     End PLIDMPA;

 End PLIDMP;
```

PLIDUMP のシンタックスとオプションは以下の通り：

character-string-expression 1

A dump options character string consisting of one or more of the following options:

A
Requests information relevant to all tasks in a multitasking program.

B
BLOCKS (PL/I hexadecimal dump).

C
Continue. The routine continues after the dump.

E
Exit from current task of a multitasking program. The program continues to run after the requested dump is completed.

F
FILES.

© Copyright IBM Corp. 1999, 2025

H
STORAGE.

This includes all Language Environment storage, and hence all the BASED and CONTROLLED storage acquired through ALLOCATE statements.

注 : A ddname of CEESNAP should be specified with the H option to produce a SNAP dump of a PL/I routine, but if this is omitted, Language Environment will issue a message but still produce a dump with useful information.

K
BLOCKS (when running under CICS). The Transaction Work Area is included.

NB
NOBLOCKS.

NF
NOFILES.

NH
NOSTORAGE.

NK
NOBLOCKS (when running under CICS).

NT
NOTRACEBACK.

O
Only information relevant to the current task in a multitasking program.

S
Stop. The enclave is terminated with a dump.

T
TRACEBACK.

T, F, and C are the default options.

character-string-expression 2
A user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP 使用上の注意

PLIDUMP を使用する場合、以下の注意が必要である：

- ルーチンが PLIDUMP を何度も呼び出す場合は、 PLIDUMP の呼び出しごとに一意のユーザー識別子を使用する。 そうすると、各ダンプの開始を簡単に識別できます。
- ddname PLIDUMP、 PL1DUMP、 CEEDUMP の DD ステートメントを使用して、ダンプ用のデータセットを定義することができる。
- PLIDUMP、 PL1DUMP、 CEEDUMP DD 文で定義されたデータ・セットは、ダンプ・レコードの折り返しを防ぐため、少なくとも 133 の論理レコード長（LRECL）を指定する必要がある。 これらの DD のいずれかで SYSOUT がターゲットとして使用されている場合、行が折り返されないように、MSGFILE(SYSOUT,FBA,133,0) または MSGFILE(SYSOUT,VBA,137,0) を指定しなければならない。
- PLIDUMP の呼び出しで H オプションを指定すると、PL/I ライブラリは OS の SNAP マクロを発行し、仮想ストレージのダンプを取得する。 最初に PLIDUMP を呼び出すと、SNAP 識別子は 0 となる。 その後、呼び出されるたびに、最大 256 まで ID が 1 つずつ増加し、256 に達すると 0 にリセットされます。
- PLIDUMP を使用した SNAP ダンプのサポートは、 z/OS でのみ提供される。 スナップ・ダンプは、CICS 環境では作成されません。
  - SNAP が成功しなかった場合、 CEE3DMP DUMP ファイルにメッセージが表示される：
    ```
    Snap was unsuccessful
    ```
  - SNAP が成功すると、 CEE3DMP にメッセージが表示される：
    ```
    Snap was successful; snap ID = nnn
    ```

ここで、 *nnn* は前述の SNAP 識別子に対応する。 失敗した SNAP 、識別子はインクリメントされない。

- プログラム単位名、プログラム単位アドレス、およびプログラム単位オフセットをダンプ・トレースバック・テーブルにおいて正しくリストしたい場合は、TEST(NONE,NOSYM) 以外のコンパイル時オプションを指定して PL/I プログラム単位をコンパイルします。 例えば、 TEST(NOSYM,NOHOOK,BLOCK) のようにオプションを指定することができる。

システム・プラットフォーム間での移植性を確保したい場合は、 PLIDUMP を使用して、 PL/I ルーチンのダンプを生成する。

PLIDUMP 出力の変数を探す

PLIDUMP の出力から変数を見つけるには、 MAP オプションをつけてプログラムをコンパイルする必要があります。 MAP オプションを指定すると、 AUTOMATIC または STATIC であるすべての level-1 変数の、AUTOMATIC および STATIC ストレージ内のオフセットを示す表がコンパイラによってリストに追加されます。

構造体の要素である変数を見つけるには、 AGGREGATE オプションをつけてプログラムをコンパイルすると便利です。 このオプションを指定すると、プログラム内のすべての構造体のすべてのエレメント のオフセットを示すテーブルが、コンパイラーによってリストに追加されます。

AUTOMATIC 変数を探す

ダンプ内の AUTOMATIC 変数を見つけるには、MAP オプション（および必要に応じて AGGREGATE オプション）の出力を使って、automatic 内のオフセットを見つける必要がある。

PLIDUMP 、B オプションで呼び出された場合、ダンプ出力には各ブロックのダイナミックセーブエリア（DSA）の 16 進ダンプが含まれる。 これが、そのブロックの自動ストレージになります。

例えば、以下の簡単なプログラムがあるとします。

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31);
     5.0        dcl b fixed bin(31);
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBC');
    10.0          end;
    11.0
    12.0        a = 0;
    13.0        b = 29;
    14.0        b = 17 / a;
```

このプログラムに対するコンパイラーの MAP オプションの結果は、実際には右側にもう 1 つ列 があること、列の間のスペースはもっと離れていることを除き、以下のようになります。

```
* * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = automatic,  Location = 160 : 0xA0(r13),
 B          1-0:5      Class = automatic,  Location = 164 : 0xA4(r13),
```

したがって、A はレジスター 13 の 16 進 A0 のオフセット位置にあり、B はレジスター 13 の 16 進 A4 のオフセット位置にあること になります (ここで、レジスター 13 は DSA を指しています)。

このプログラムでは、PLIDUMP が B オプションで呼び出されるため、現在の呼び出しチェーンの各ブロックの自動ストレージの 16 進ダンプが含まれる。 これは以下のようになります (これも、右側の列が省略されています)。

```
Dynamic save area (TEST): 0AD963C8
      +000000 0AD963C8  10000000 0AD96188 00000000 00000000
      +000020 0AD963E8  00000000 00000000 00000000 00000000
```

```
      +000040 0AD96408  00000000 00000000 00000000 0AD96518
      +000060 0AD96428  00000000 00000000 00000000 00000000
      +000080 0AD96448 - +00009F 0AD96467   same as above
      +0000A0 0AD96468  00000000 0000001D 00100000 00000000
      +0000C0 0AD96488  0B300000 0A700930 0AD963C8 00000000
      +0000E0 0AD964A8  00000000 00000000 00000000 00000000
      +000100 0AD964C8  0AA47810 0A70E6D0 0AD96540 0AD960F0
      +000120 0AD964E8  00000001 0A70F4F8 0AD96318 00000000
      +000140 0AD96508  00000000 00000000 00000000 00000000
```

A は A0 の 16 進オフセットにあり、B は AUTOMATIC の 16 進オフセット A4 にあるので、ダンプは A と B がそれぞれ 00000000 と 0000001D の（予想される）16 進値を持っていることを示す。

OPT(2)および OPT(3)というコンパイラー・オプションの下では、いくつかの変数、特に FIXED BIN および POINTER スカラ変数が、ストレージに割り当てられることがなく、ダンプ出力で見つからない可能性があることに注意してください。

STATIC 変数を探す

RENT オプションでコードをコンパイルした場合、静的変数は現在のロード・モジュールの Writeable Static Area (WSA) に配置されます。

WSA 内の変数のオフセットは MAP オプションの出力から求めることができ、 WSA は CAA と呼ばれる Language Environment 制御ブロックに保持される。 WSA の値は、Language Environment ダンプにも記載されている。

しかし、NORENT オプションを使ってコードをコンパイルした場合、 EXTERNAL STATIC は通常通り（リンカのリストとコンパイラの MAP オプションの出力を使って）見つかる。 INTERNAL STATIC は、 ダンプの一部としてダンプされる（ が B オプションで呼び出された場合)。 Language Environment PLIDUMP

旧 PL/I コンパイラーとは異なり、STATIC のアドレスがいずれか 1 つのレジスターを占有することはありません。

例えば、上記のプログラムで、変数を STATIC に変更したとします。

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31) static;
     5.0        dcl b fixed bin(31) static;
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBC');
    10.0          end;
    11.0
    12.0        a = 0;
    13.0        b = 29;
    14.0        b = 17 / a;
```

プログラムを NORENT オプションでコンパイルすると、このプログラムのコンパイラ MAP オプションの結果はこのようになる。ただし、実際には右側に列が 1 つ増え、列の間隔もかなり離れている：

```
* * * * *  S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = static,  Location = 0 : 0x0 + CSECT ***TEST2
 B          1-0:5      Class = static,  Location = 4 : 0x4 + CSECT ***TEST2
```

したがって、A はコンパイル単位 TEST の静的 CSECT の中の 16 進オフセット 00 の位置にあり、B は 16 進オフセット 04 の位置にあることになります。

このプログラムでは、PLIDUMP が B オプションで呼び出されるため、現在の呼び出しチェーンの各コンパイルについて、スタティック・ストレージの 16 進ダンプが含まれる。 これは、以下のようになります (これも、右側の列が省略されています)。

```
Static for procedure TEST    Timestamp: 2004.08.12
      +000000 0FC00AA0  00000000 0000001D 0FC00DC8 0FC00AC0
      +000020 0FC00AC0  0FC00AA8 00444042 00A3AE01 0FC009C8
```

```
      +000040 0FC00AE0  6E3BFFE0 00000000 00000000 00000000
      +000060 0FC00B00  00000000 00000000 00000000 00000000
      +000000 0AD963C8  10000000 0AD96188 00000000 00000000
```

したがって、16 進オフセット 00 にある A は (予想どおりの) 16 進値 00000000 を持ち、16 進オフセット 04 にある B は (これも予想どおりの) 16 進値 0000001D、つまり 10 進値 29 を持つことになります。

CONTROLLED 変数を探す

CONTROLLED 変数は基本的に スタックであり、各 変数には、そのスタックの先頭を指す「アンカー」がある。 LIFO CONTROLLED CONTROLLED 変数の位置を特定する鍵は、このアンカーの位置を特定することであり、その位置はコンパイラのオプションに依存する。

この後の CONTROLLED 変数の説明では、プログラムソースは 431 ページの『#unique_709/unique_709_Connect_42_pligen』 と同じプログラムですが、ストレージクラスは CONTROLLED に変更されています：

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31) controlled;
     5.0        dcl b fixed bin(31) controlled;
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBHC');
    10.0          end;
    11.0
    12.0        allocate a, b;
    13.0        a = 0;
    14.0        b = 29;
    15.0        b = 17 / a;
```

NORENT WRITABLE の下で

コンパイラー MAP オプションの結果は、この場合も、実際には右側にもう 1 つ列 があること、列の間のスペースはもっと離れていることを除き、以下のようになります。

```
* * * * *  S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = static,  Location = 8 : 0x8 + CSECT ***TEST2
 B          1-0:5      Class = static,  Location = 12 : 0xC + CSECT ***TEST2
```

これらの行は、(A や B 自体の場所ではなく) A および B のアンカーの場所を記述していることに注意してください。 つまり、A のアンカーはコンパイル単位 TEST のスタティック CSECT のヘクス 08 に位置し、B のアンカーはヘクス 0C に位置する。

PLIDUMP を B オプションで呼び出すと、現在の呼び出しチェーンの各コンパイルについて、スタティック・ストレージの 16 進ダンプが含まれる。 これは、以下のようになります (これも、右側の列が省略されています)。

```
Static for procedure TEST    Timestamp: . . .

  +000000 0FC00A88  0FC00DB0 0FC00AA8 102B8A30 102B8A50
  +000020 0FC00AA8  0FC00A88 00444042 00A3AE01 0FC009B0
  +000040 0FC00AC8  6E3BFFE0 00000000 00000000 00000000
```

そのため、A のアンカーは 102B8A30 にあり、B のアンカーは 102B8A50 にあります。 しかし、これらは CONTROLLED 変数であるため、そのストレージは ALLOCATE ステートメントによって取得され、したがってこれらのアドレスはヒープ・ストレージを指している。 しかし、 PLIDUMP を H オプションで呼び出すと、ヒープストレージの 16 進ダンプが含まれる。 これは、以下のようになります (これも、右側の列が省略されています)。

```
Enclave Storage:
  Initial (User) Heap
```

```
    +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
      . . .
    +001A00 102B8A18  102B7018 00000020 0FC00A90 00000014
                      00000000 00000000 00000000 00000000
    +001A20 102B8A38  102B7018 00000020 0FC00A94 00000014
                      00000000 00000000 0000001D 00000000
```

A のアンカーが 102B8A30 にあったため、A は 16 進値 00000000 を持ちます。 B のアンカーが 102B8A50 にあったため、B は (予想どおりの) 16 進値 0000001D を持ちます。

NORENT NOWRITABLE(FWS) の下で

これらのオプションの下で、コンパイラの MAP オプションの結果は次のようになる。ただし、この場合も、実際には右側に列が 1 つ増え、列の間隔もかなり離れている：

```
 * * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = automatic,  Location = 236 : 0xEC(r13)
 B          1-0:5      Class = automatic,  Location = 240 : 0xF0(r13)
```

注 : これらのオプションの下では、 CONTROLLED 変数の位置を特定する際に、さらに一段高いレベルの間接性がある。したがって、上の行は A と B のアンカーのアドレスの位置を記述している。 したがって、A のアンカーのアドレスは 16 進数の EC でブロック TEST の自動に配置され、B のアンカーは 16 進数の F0 に配置されます。

PLIDUMP 、B オプションで呼び出されるため、現在の呼び出しチェーンの各ブロックの自動ストレージの 16 進ダンプが含まれる。 これは、以下のようになります (これも、右側の列が省略されています)。

```
 Dynamic save area (TEST): 102973C8
  +000000 102973C8  10000000 10297188 00000000 8FC007DA
          ....
  +0000E0 102974A8  0FC00998 00000000 00000000 102B8A40
                    102B8A28 10297030 102977D0 8FDF3D7E
```

したがって、A のアンカーのアドレスは 102B8A40 であり、B のアンカーのアドレスは 102B8A28 です。

PLIDUMP も H オプションで呼び出されたため、ヒープストレージの 16 進ダンプが含まれる。 これは、以下のようになります (これも、右側の列が省略されています)。

```
 Enclave Storage:
    Initial (User) Heap
      +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
        . . .
      +001A00 102B8A18  102B7018 00000018 00000000 0FC00A78
                        102B8A80 00000000 102B7018 00000018
      +001A20 102B8A38  102B8A20 0FC00A74 102B8A60 00000000
                        102B7018 00000020 102B8A40 00000014
      +001A40 102B8A58  00000000 00000000 00000000 00000000
                        102B7018 00000020 102B8A28 00000014
      +001A60 102B8A78  00000000 00000000 0000001D 00000000
                        00000000 00000000 00000000 00000000
```

B のアンカーのアドレスが以前は 102B8A28 にあって、B のアンカーは現在は 102B8A80 にあるため、B は予想どおり 16 進値 0000001D (つまり 10 進値 29) を持ちます。

NORENT NOWRITABLE(PRV) の下で

これらのオプションを使ってプログラムをコンパイルすると、 MAP のリストは次のようになる：

```
* * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 ***TEST3     1-0:4  Class = ext def,  Location = CSECT ***TEST3
 ***TEST4     1-0:5  Class = ext def,  Location = CSECT ***TEST4
_PRV_OFFSETS  1-0:1  Class = static,   Location = 8 : 0x8 + CSECT ***TEST2
```

ここで重要なのは、この出力の最後の行である。 _PRV_OFFSETS は静的テーブルで、 CONTROLLED の各変数の PRV テーブルへのオフセットを保持する。 この静的テーブルは、 MAP オプションが指定された場合のみ生成される。

このテーブルを解釈するために、コンパイラーは、ブロック名テーブルのすぐ後に、通常は小さな別のリストも生成します。このリストは、このサンプルでは、以下のようになります。

```
           PRV Offsets

    Number   Offset Name
         1        8 A
         1        C B
```

この表は、名前付き CONTROLLED 変数のそれぞれについて、ランタイム _PRV_OFFSETS テーブル内の 16 進オフセットの一覧である。 ブロック番号 (最初の列) を使用して、同じ名前であるが、異なるブロックに宣言されている変数 を区別することができます。

_PRV_OFFSETS テーブルはスタティック・ストレージにあり（16 進オフセット 8 にある）、 PLIDUMP が B オプションで呼び出されたため、ダンプ出力には次のように表示される：

```
Static for procedure TEST    Timestamp: . . .
  +000000 10908EC8  02020240 00000005 6DD7D9E5 6DD6C6C6
                    00000000 00000004 D00000A0 00100000
  +000020 10908EE8  6E3BFFE0 00000000 00000000 00000000
                    00000000 90010000 00000000 00000000
```

したがって、PRV テーブル内の A のオフセットは 0 で、PRV テーブル内の B のオフセットは 4 です。 また、 _PRV_OFFSETS テーブルの最初の 8 バイトを占めるアイキャッチ"_PRV_OFF "にも注目してほしい。

PRV テーブルは常に CAA 内のオフセット 4 に位置し、 PLIDUMP が H オプションで呼び出されたため、ダンプ出力に含まれる。 CAA は次のようになります。

```
Control Blocks Associated with the Thread:
  CAA: 0A7107D0
   +000000 0A7107D0  00000800 0ADB7DE0 0AD97018 0ADB7018
                     00000000 00000000 00000000 00000000
```

そのため、PRV テーブルのアドレスは 0ADB7DE0、ダンプ出力の HEAP ：

```
 Enclave Storage:
    Initial (User) Heap
      +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
        . . .
      +000DC0 0ADB7DD8  00000000 00000000 0ADB8A38 0ADB8A58
                        0ADB7018 00000488 00000000 00000000
```

したがって、PRV テーブルには、 0ADB8A38 0ADB8A58、などが含まれている。 _PRV_OFFSETS テーブルから導かれるように、PRV テーブルへの A のオフセットは 0、B のオフセットは 4 であるため、これらはそれぞれ A と B のアドレスでもある。

これらのアドレスは、ダンプ内のヒープ・ストレージにも表示されます。

```
  Enclave Storage:
    Initial (User) Heap
      +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
        . . .
      +001A00 0ADB8A18  00000000 00000000 0ADB7018 00000020
                        00000000 00000014 0A7107D4 00000000
      +001A20 0ADB8A38  00000000 00000000 0ADB7018 00000020
                        00000004 00000014 0A7107D4 00000000
      +001A40 0ADB8A58  0000001D 00000000 00000000 00000000
                        00000000 00000000 00000000 00000000
```

したがって、A のアドレスは 0ADB8A38 であるため A の 16 進値は予想どおり 00000000 となり、B のアドレスは 0ADB8A58 であるため B の 16 進値も予想どおり 0000001D となります。

保存されたコンパイル・データ

コンパイルの間、コンパイラーはロード・モジュールにコンパイルに関するさまざまな情報を保存します。この情報は、デバッグや将来のマイグレーション作業のときに大きな助けになることがあります。 このセクションでは、この保存された情報について述べます。

Copyright

Copyright コンパイラー・オプションを指定すると、コンパイラーは Copyright 文字列を、オブジェクト内のタイムスタンプ・データの直前に置かれた CHARACTER VARYING 文字列として保存する。

このストリングの後にはブランクが必要なだけ続いて、ストリングにこれらのブランクを加えたものが 4 バイトの倍数を占めるようになります。

タイム・スタンプ

コンパイラーは各ロード・モジュールにタイム・スタンプ を保存します。 タイム・スタンプは *YYYYMMDDHHMISSVNRNML* 形式の 20 バイト文字ストリングです。 これにより、コンパイル日時のほか、このストリングを生成したコンパイラーのバージョンが記録されます。

ストリングのエレメントの意味は次のとおりです。

YYYY
コンパイルされた年

MM
コンパイルされた月

DD
コンパイルされた日

HH
コンパイルされた時刻 (時)

MI
コンパイルされた時刻 (分)

SS
コンパイルされた時刻 (秒)

VN
コンパイラーのバージョン番号

RN
コンパイラーのリリース番号

ML
コンパイラーの保守レベル

タイム・スタンプは、PPA2 で見付けることができます。PPA2 の中のオフセット 12 は 4 バイト整数で、PPA2 のアドレスからタイム・スタンプにオフセット (負の数のことがあります) を与えます。

次に、PPA2 は、PPA1 で見付けることができます。PPA1 の中のオフセット 4 は、4 バイト整数で、PPA1 に対応するエントリー・ポイント・アドレスから PPA2 にオフセット (負の数のことがあります) を与えます。

PPA1 は次のように位置指定できます。

- コードが LP(32) でコンパイルされた場合、PPA1 はブロックのエントリー・ポイント・アドレスから位置指定できます。 エントリー・ポイント・アドレスからのオフセット 12 の位置には、そのエントリー・ポイント・アドレスから PPA1 までのオフセットを示す 4 バイト整数 (負の数の場合もある) があります。
- コードが LP(64) でコンパイルされた場合、PPA1 までのオフセットは、各エントリー・ポイント・アドレスの前にフルワードの 8 バイトで表されます。

保存されたオプション・ストリング

コンパイラーは、ロード・モジュールのビルドに使用されたコンパイラー・オプションが記録された 32 バイト・ストリングをロード・モジュールに格納します。

保存されたオプション文字列の宣言は、サンプル・データ・セット SIBMZSAM のインクルード・ファイル `ibmvsos` で提供されています。

構造体の中のほとんどのフィールドにおいて、そのフィールドの意味は名前によって明白ですが、いくつかのフィールドについてはいくらかの説明が必要です。

- sos_words は、構造体のバイト数を 4 で割った数を保持する。
- sos_version はこの構造体のバージョン番号である。 コンパイラーのバージョン番号ではありません。
- 構造体のサイズ、および設定されているフィールドは、バージョン番号によって異なります。

保存されたオプション・ストリングは、次の 2 つのどちらかの方法でタイム・スタンプの後に位置指定されます。

1. サービス・オプションが指定されている場合、サービス・オプションの中に指定されたストリングは、文字がストリングを変更すると即時にタイム・スタンプに続きます。 次に、保存されたオプション・ストリングが、2 番目の文字がストリングを変更するとサービス・ストリングに続きます。
2. サービス・オプションが指定されていない場合、保存されたオプション・ストリングは、文字がストリングを変更すると即時にタイム・スタンプに続きます。

保存されたオプション・ストリングを保持する可変ストリングの長さは、保存されたオプション・ストリングそれ自体のサイズより長くなります。

サービス・ストリングの存在 (または不在) は、PPA2 内に、PPA2 の 10 進数のオフセット 20 にフラグ・バイトで示されます。このバイトと '20'BX の AND 演算結果がゼロでなければ、サービス・ストリングが存在します。

以前にリリースされた PL/I コンパイラーには、保存されたオプション・ストリングを、コンパイラーがロード・モジュールに置かないものがありました。 保存されたオプション・ストリングの存在 (または不在) は、PPA2 内に、10 進数のオフセット 20 にフラグ・バイトで示されます。このバイトと '02'BX の AND 演算結果がゼロでなければ、保存されたオプション・ストリングが存在します。

PLIDUMP_XCALL 環境変数

環境変数 PLIDUMP_XCALL を使って、PLIDUMP の処理の一部として呼び出される外部プログラムの名前を指定する。

例えば、プログラム `runtest` 、その中に CALL PLIDUMP：

- USS 傘下

  次のコマンドを使用します。

```
export PLIDUMP_XCALL=PRTMORE
./runtest
```

- バッチ環境の場合

  プログラム内で putenv 組み込み関数を使用する：

```
...
xx = putenv( "PLIDUMP_XCALL=PRTMORE" );
```

プログラム `runtest` が実行されると、プログラム `prtmore` がフェッチされ、 CALL PLIDUMP の処理の一部として呼び出される。 なお、 FETCH の制限はすべて適用される。

この良い用途は、より多くのデバッグ情報を提供するために、フォールト・アナライザーなどの外部インターフェイスを呼び出すことである。

第 23 章 割り込みとアテンションの処理

PL/I プログラムにアテンション割り込みを認識させるには、 次の 2 つの操作が可能でなければなりません。

- 割り込みを作成できなければなりません。 それを行う方法は、使用する端末とオペレーティング・システムによって異なります。
- ユーザーのプログラムで、割り込みに対応する準備が整っていなければなりません。 プログラム中に ON ATTENTION ステートメントを記述し、 ATTENTION 条件が発生したときにプログラムが制御を受けるようにすることができる。

  注 : プログラムに ATTENTION ON-unit があり、それを呼び出したい場合は、以下のいずれかのオプションをつけてコンパイルする必要がある：

  – INTERRUPT オプション（TSO のみ対応）
  – NOTEST または TEST(NONE,NOSYM)

    この方法でコンパイルすると、 PLIXOPT で INTERRUPT(OFF) を明示的に指定しない限り、 INTERRUPT(ON) が有効になる。

割り込みを作成する手順は、使用しているオペレーティング・システムと端末に関する IBM マニュアルに記載されています。

割り込み（オペレーティング・システムがあなたの要求を認識）と ATTENTION 。

割り込み とは、オペレーティング・システムが 実行中のプログラムに通知するユーザーの要求のことです。 PL/I プログラムが INTERRUPT コンパイル・タイム・オプションでコンパイルされた場合、プログラム中の個別のポイントで内部割り込みスイッチをテストする命令が含まれます。 ロード・モジュール内のプログラムが INTERRUPT コンパイル時オプションでコンパイルされている場合、内部割り込みスイッチを設定することができます。

内部スイッチは、割り込み要求が出されたことを オペレーティング・システムが認識した時に設定されます。 特別検査命令（ポーリング）の実行により、 ATTENTION 。 ポーリングが発生する前にデバッグツールのフック（または CALL PLITEST）に遭遇した場合、 ATTENTION 条件処理が始まる前にデバッグツールに制御を委ねることができる。

ポーリングは、 ATTENTION 条件が、ステートメント内ではなく、 PL/I ステートメント間で発生することを保証する。

441 ページの『#unique_124/unique_124_Connect_42_ell2300』 は、スケルトン・プログラム、 -ユニット、ポーリング命令が生成されるいくつかの状況を示している。 ATTENTION ON プログラム内では、次の位置でポーリングが行われます。

- LABEL1
- の各反復。 DO
- ELSE PUT SKIP ステートメント
- ブロック END ステートメント

ATTENTION ON-ユニットの使用

```
%PROCESS INTERRUPT;
     .
     .
     .
 ON ATTENTION
   BEGIN;
     DCL X FIXED BINARY(15);
     PUT SKIP LIST ('Enter 1 to terminate, 0 to continue.');
     GET SKIP LIST (X);
     IF X = 1 THEN
       STOP;
     ELSE
       PUT SKIP LIST ('Attention was ignored');
```

© Copyright IBM Corp. 1999, 2025

```
   END;
     .
     .
     .
  LABEL1:
   IF EMPNO ...
     .
     .
     .
   DO I = 1 TO 10;
     .
     .
     .
   END;
     .
     .
     .
```

ATTENTION ON-ユニットの使用

ATTENTION ON-unit 内の処理を使用することで、プログラム内の無限ループを終了させることができる。

ポーリング命令が割り込みの発生を認識すると、ATTENTION ON-ユニットに制御が与えられる。 ON-ユニットからの通常のリターンは、ポーリングコードに続くステートメントになります。

デバッグ・ツールとの対話

プログラムに TEST(ALL) または TEST(ERROR) ランタイム・オプションが有効になっている場合、割り込みによって、次にフックに遭遇したときにデバッグ・ツールが制御を受けるようになります。 これは、割り込みが発生したことをプログラムのポーリング・コードが認識する前で ある可能性があります。

その後、 ATTENTION 条件が発生すると、デバッグツールは条件処理のために再び制御を受ける。

第 24 章 チェックポイント/再始動機能の使用

この章では、バッチ環境で長時間稼働するプログラムの実行途中でチェックポイントをとるために役立つ手段である PL/I チェックポイント/再始動機能について説明します。

プログラム内の指定されたポイントで、プログラムの現状についての情報が、 データ・セットにレコードとして書き込まれます。 システム障害が原因でプログラムが終了したときに、この情報を使えば、 プログラムを全部実行し直す必要がなく、障害の発生地点の近くから プログラムを再始動することができます。

この再始動は、自動再始動または据え置き再始動のどちらもあり得ます。 自動再始動は、即時に実行される再始動です (ただし、システム・メッセージで要求されたときに、オペレーターがそれを許可することを前提とします)。 据え置き再始動は、新しいジョブとして、あとから実行される再始動です。

システム障害が起きていなくても、プログラム内から自動再始動を要求することができます。

PL/I チェックポイント/再始動は、オペレーティング・システムの 拡張チェックポイント/再始動機能を使用します。 この機能については、477 ページの『参考文献』に掲載したマニュアルを参照してください。

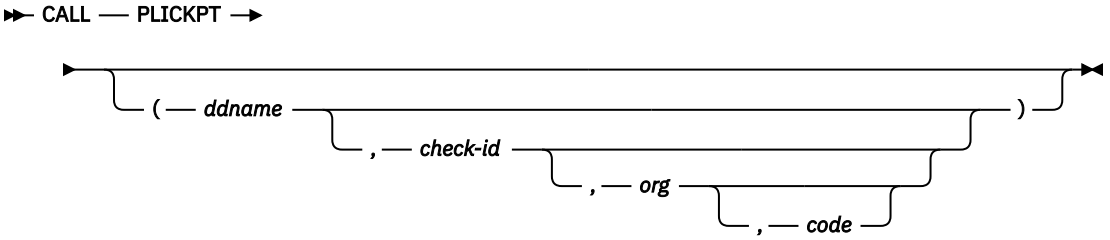
チェックポイント/再始動を使用するには、以下の操作を行う必要があります。

- ユーザーのプログラム内の適切なポイントで、 チェックポイント・レコードを書き込むように要求します。 これは、組み込みサブルーチン PLICKPT で実行される。
- チェックポイント・レコードを書き込めるデータ・セットを提供します。
- 希望する再起動アクティビティを確保するには、EXEC または JOB ステートメントで RD パラメータを指定する必要がある場合があります（ *z/OS JCL Reference* を参照)。

注 : プログラムで使用するデータ・セットに関連した制約事項に注意する必要があります。 この制約事項の詳細については、477 ページの『参考文献』を参照してください。

チェックポイント・レコードの要求

チェックポイント記録を書き込むたびに、PL/I プログラムから組み込みサブルーチン PLICKPT を呼び出す必要があります。



4 つの引数はすべてオプショナルです。 引数を使わない場合は、所定の順序でその引数に続く別の引数を指定するとき以外は、 引数を指定する必要はありません。 引数を指定する場合は、未使用引数をヌル・ストリング ('') として指定します。

DD 名 (ddname)
文字ストリングの定数または変数で、チェックポイント・レコードに使用するデータ・セットを定義する DD ステートメントの名前を指定します。 この引数を省略すると、システムはデフォルトの ddname SYSCHK を使用する。

check-id
あとからチェックポイント・レコードを識別できるようにするためにチェックポイント・レコードに割り当てる名前を指定する文字ストリングの定数または変数です。 この引数を省略すると、システムは固有の ID を提供し、 それをオペレーターのコンソールに印刷します。

org
値がオペレーティング・システム用語でチェックポイント・データ・セットの編成を示す属性 CHARACTER(2) を持つ文字ストリング定数または変数です。 指定できる値は以下のとおりです。

© Copyright IBM Corp. 1999, 2025

PS
順次 (すなわち、CONSECUTIVE) 編成を指示します。

PO
区分編成を指示します。

この引数を省略すると、PS がとられます。

コード
FIXED BINARY (31)の属性を持つ変数で、 PLICKPT からリターンコードを受け取ることができる。 戻りコードには、次の値があります。

0
チェックポイントは正常にとられた。

4
再始動は正常に行われた。

8
チェックポイントはとられていない。 PLICKPT 。

12
チェックポイントはとられていない。 欠落した DD ステートメント、ハードウェア・エラーがないか、 またはデータ・セットのスペースが不十分でなかったかを調べてください。 REPLY オプション付きの DISPLAY ステートメントがまだ不完全な状態でチェックポイントを行うと、チェックポイントは失敗します。

16
チェックポイントは取られたが、 ENQ マクロ呼び出しが未解決で、再起動しても復元されない。 このような事態は、通常、PL/I プログラムの場合は生じません。

チェックポイント・データ・セットの定義

チェックポイント・レコードを入れる先の データ・セットを定義するには、ジョブ制御プロシージャー内に DD ステートメントを入れなければなりません。

このデータセットは、 CONSECUTIVE 、またはパーティション化された組織のいずれかを持つことができる。 任意の有効 DD 名を使用することができます。 ddname SYSCHK を使用する場合、 PLICKPT を起動する際に ddname を指定する必要はない。

データ・セット名を指定しなければならないのは、 据え置き再始動用にデータ・セットをとっておきたい場合だけです。 入出力装置は、直接アクセス装置であれば使用できます。

最後のチェックポイント・レコードだけを取得するには、状況を NEW (または、データ・セットが既に存在する場合は OLD) と指定します。 これにより、各チェックポイント・レコードが直前のレコードを上書きします。

複数のチェックポイント記録を保持するには、ステータスを MOD に指定する。 これで、各チェックポイント・レコードが直前のレコードのあとに続けて追加されます。

チェックポイント・データ・セットがライブラリーの場合は、 "チェック ID" がメンバー名として使用されます。 したがって、チェックポイントは、それ以前にとられた同一名を 持つすべてのチェックポイントを削除します。

直接アクセス・ストレージの場合、保持しようとする最大数の チェックポイント・レコードを保管するのに十分な 1 次スペースを 割り振らなければなりません。 増分スペース割り振りを指定することはできますが、ここでは使用されません。 チェックポイント・レコードはこのステップで 割り振られる主記憶域の大きさより約 5000 バイト長くなっています。

DCB 情報は必要ありませんが、 適切であれば次のどれでも組み込むことができます。

```
OPTCD=W, OPTCD=C, RECFM=UT
```

これらのサブパラメーターについては、「*z/OS MVS JCL* ユーザーズ・ガイド」を参照してください。

再始動の要求

再始動には、自動再始動または据え置き再始動があります。

自動再始動にできるのは、システム障害のあとか、またはプログラム内から行われる場合です。 システム・オペレーターは、システムから要求があれば、 すべての自動再始動を許可しなければなりません。

システム障害後の自動再始動

チェックポイントの取得後にシステム障害が発生した場合、 EXEC または JOB ステートメントで RD=R を指定（または RD パラメータを省略）していれば、最後のチェックポイントで自動再起動が行われます。

チェックポイントが取られる前にシステム障害が発生した場合でも、 EXEC または JOB ステートメントで RD=R を指定していれば、ジョブ・ステップの最初から自動的に再起動することができます。

システム障害が発生した後でも、 EXEC または JOB ステートメントで RD=RNC を指定することにより、ジョブ・ステップの最初から自動再起動を強制することができます。 別のシステム障害が生じても、RD=RNC を指定すれば、 チェックポイントを処理しないで自動ステップ再始動を要求することになります。

プログラム内の自動再始動

再始動は、プログラム内の任意の点で要求できます。

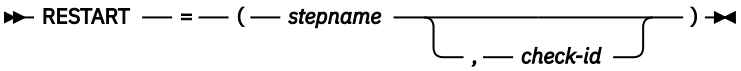
この再始動に関する規則は、システム障害後の再始動の場合と同じです。 再始動を要求するには、次のステートメントを実行する必要があります。

```
CALL PLIREST;
```

再始動を行うために、コンパイラーは、 システム完了コード 4092 でプログラムを異常終了させます。 したがって、この機能を使用するには、 システム生成時に適格コードのテーブルからシステム完了コード 4092 を削除してはなりません。

据え置き再始動

自動再起動のアクティビティはキャンセルされるが、延期された再起動のためのチェックポイントはまだ利用可能であることを確認するには、プログラムが最初に実行されるときに、 EXEC または JOB 文で RD=NR を指定します。



その後、再起動の延期が必要になった場合は、JOB 文に RESTART パラメータを指定して、新しいジョブとしてプログラムを投入する必要があります。 RESTART パラメータを使用して、再スタートを行うジョブ・ステップを指定し、チェックポイントで再スタートする場合は、チェックポイント・レコードの名前を指定します。

チェックポイントからの再始動の場合、チェックポイント・レコードが 入ったデータ・セットを定義する DD ステートメントも提供する必要があります。 DD ステートメントには、SYSCHK という名前を付けなければならない。 DD 文は、ジョブ・ステップの EXEC 。

チェックポイント/再始動活動の変更

プログラム内に設定されているチェックポイントからの自動再始動アクティビティーを取り消すことができます。

自動再始動を取り消すには、次のステートメントを実行します。

```
CALL PLICANC;
```

ただし、JOB または EXEC 文で RD=R または RD=RNC を指定した場合は、ジョブ・ステップの最初から自動再起動を行うことができる。

また、以前にとられているチェックポイントがあれば、 それをそのまま据え置き再始動に使用することができます。

自動リスタートとチェックポイントの取得は、たとえプログラムで要求されていたとしても、JOB または EXEC ステートメントで RD=NC を指定することでキャンセルできます。

第 25 章 ユーザー出口の用法

PL/I にはユーザー出口がいくつか備わっています。 そのユーザー出口を使用すれば、自分のニーズに合うように PL/I 製品をカスタマイズできます。

PL/I 製品は、デフォルト出口と、関連するソース・ファイルを提供します。

デフォルト出口で提供される機能とは異なる機能を出口で実行したい場合は、提供されているソース・ファイルを適宜変更することをお勧めします。

場合によっては、ユーザーの組織の要件を満たすようにコンパイラーを 調整できることが役立つこともあります。 例えば、特定のメッセージを抑止したり、 ほかのメッセージの重大度を変更したりする必要が生じることがあります。 コンパイルについての統計情報のログをファイルに書き込むなど、 各コンパイルごとに特定の機能を実行するように指定する必要が生じることがあります。 コンパイラー・ユーザー出口は、この種の機能を処理します。

PL/I を使用すれば、 独自のユーザー出口を作成することもできますし、製品に付属の出口を「そのまま」使用することも、変更して使用することもできます。 製品に付属するユーザー終了ソースコードは、サンプルデータセット SIBMZSAM のメンバー IBMUEXIT で入手できます。

この章で提供される情報は以下のとおりです。

- コンパイラー・ユーザー出口がサポートするプロシージャー
- コンパイラー・ユーザー出口を活動化する方法
- IBMUEXIT この場合、 IBM が提供するコンパイラのユーザー終了が必要です
- 独自のコンパイラー・ユーザー出口作成の要件

コンパイラー・ユーザー出口によって実行されるプロシージャー

コンパイラー・ユーザー出口は、3 つの特定のプロシージャー (初期化、コンパイラー・メッセージの代行受信およびフィルター操作、終了) を実行します。

447 ページの図 9 に示してあるように、 コンパイラーは、初期化プロシージャー、メッセージ・フィルター・プロシージャー、 および終了プロシージャーへ制御を渡します。 これらの 3 つのプロシージャーは、それぞれ、 要求されたプロシージャーが完了したならば、制御をコンパイラーへ戻します。

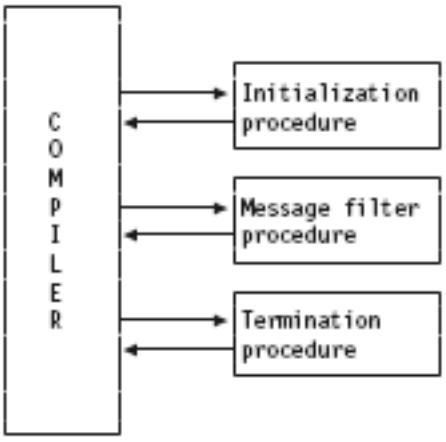


図 *9. PL/I* コンパイラー・ユーザー出口のプロシージャー

これらの各プロシージャーには、次の 2 つの制御ブロックが渡されます。

© Copyright IBM Corp. 1999, 2025

- コンパイルについての情報が含まれているグローバル制御ブロック。 これは、最初のパラメーターとして渡されます。
- 2 番目のパラメーターとして渡される機能専用の制御ブロック。 この制御ブロックの内容は、どのプロシージャーが呼び出されているかによって異なります。 詳細については、451 ページの『初期化プロシージャーの作成』、451 ページの『メッセージ・フィルター操作プロシージャーの作成』、および 453 ページの『終了プロシージャーの作成』を参照してください。

関連情報

448 ページの『グローバル制御ブロックの構造』
グローバル制御ブロックは、3 つのユーザー出口プロシージャー (初期化、 フィルター操作、および終了) が呼び出されるたびに、それぞれに渡されます。

グローバル制御ブロックの構造

グローバル制御ブロックは、3 つのユーザー出口プロシージャー (初期化、 フィルター操作、および終了) が呼び出されるたびに、それぞれに渡されます。

次のコードと説明は、グローバル制御ブロック内の各フィールドの内容を示すものです。

```
Dcl
  1 Uex_UIB           native based,
    2 Uex_UIB_Length         fixed bin(31),

    2 Uex_UIB_Exit_token     pointer(32),        /* for user exit's use */

    2 Uex_UIB_User_char_str  pointer(32),        /* to exit option str  */
    2 Uex_UIB_User_char_len  fixed bin(31),

    2 Uex_UIB_Filename_str   pointer(32),        /* to source filename  */
    2 Uex_UIB_Filename_len   fixed bin(31),

    2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs   */
    2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs   */

    2 Uex_UIB_Exit_Routs,                    /* exit entries set at
                                                initialization      */
      3 ( Uex_UIB_Termination,
          Uex_UIB_Message_Filter,            /* call for each msg   */
          *, *, *, * )
        limited entry (
             *,                               /* to Uex_UIB          */
             *,                               /* to a request area   */
         );
```

データ入力フィールド

Uex_UIB_ Length
Contains the length of the control block in bytes. The value is `storage (Uex_UIB)`.

Uex_UIB_Exit_token
Used by the user exit procedure. For example, the initialization might set it to a data structure that is used by both the message filter and the termination procedures.

Uex_UIB_User_char_str
Points to an optional character string, if you specify it. For example, `pli filename (EXIT ('string'))…fn` can be a character string up to thirty-one characters in length.

Uex_UIB_char_len
Contains the length of the string pointed to by the `User_char_str`. The compiler sets this value.

Uex_UIB_Filename_str
Contains the name of the source file that you are compiling, and includes the drive and subdirectories as well as the filename. The compiler sets this value.

Uex_UIB_Filename_len
Contains the length of the name of the source file pointed to by the `Filename_str`. The compiler sets this value.

Uex_UIB_return_code
Contains the return code from the user exit procedure. The user sets this value.

Uex__UIB_reason_code
Contains the procedure reason code. The user sets this value.

Uex_UIB_Exit_Routs
Contains the exit entries set up by the initialization procedure.

Uex_UIB_Termination
Contains the entry that is to be called by the compiler at termination time. The user sets this value.

Uex_UIB_Message_Filter
Contains the entry that is to be called by the compiler whenever a message needs to be generated. The user sets this value.

IBM -supplied compiler exit、 IBMUEXIT

IBM を使用すると、コンパイラのユーザー終了のサンプル、、メッセージをフィルタリングすることができます。 IBMUEXIT

このコンパイラー出口は、メッセージをモニターし、指定されたメッセージ番号に基づいてメッセージを抑止したりメッセージの重大度を変更したりします。

製品に付属するユーザー終了ソースコードは、サンプルデータセット SIBMZSAM のメンバー IBMUEXIT で入手できます。

コンパイラー・ユーザー出口の活動化

コンパイラのユーザー終了を有効にするには、 EXIT コンパイル時オプションを指定する必要があります。

EXIT コンパイル時オプションでは、ユーザー終了入力ファイルの DDname を指定するユーザー・オプション文字列を指定できる。 文字列を指定しない場合、ユーザー終了入力ファイルの DDname として SYSUEXIT 。

ユーザー・オプション・ストリングは、グローバル制御ブロックにおいてユーザー出口関数に渡されます。追加情報については、 448 ページの『グローバル制御ブロックの構造』の "Uex_UIB_User_char_str" を参照してください。

z/OS UNIX では、ユーザー出口入力ファイルが以下のような場所にあるとします。

```
/a/b/ibmuexit.inf
```

`-qexit` オプションを指定してコンパイルする前に `export DD_SYSUEXIT=/a/b/ibmuexit.inf` を指定するか、// シンボルが必要な `-qexit=//a/b/ibmuexit.inf` を指定してコンパイルすることができます。

関連情報
35 ページの『出口』
EXIT オプションにより、コンパイラー・ユーザー出口を呼び出す ことができます。

コンパイラー・ユーザー出口のカスタマイズ

独自のコンパイラー・ユーザー出口を作成することも、コンパイラーに付属の出口を単に使用することもできます。 いずれの場合も、コンパイラー・ユーザー終了時のフェッチ可能なファイル名は IBMUEXIT でなければならない。

このセクションでは、以下の作業を行う方法について説明します。

- カスタマイズされたメッセージ・フィルター操作用にユーザー出口入力ファイルを変更する
- 独自のコンパイラー・ユーザー出口を作成する

変更 SYSUEXIT

まったく新しいコンパイラー・ユーザー出口を作成するのではなく、 簡単にユーザー出口入力ファイルを変更できます。

ファイルを編集して、抑止するメッセージ番号と、 変更するメッセージ番号重大度レベルを指示します。 ユーザー終了入力ファイルの例にサンプルファイルを示す。

ユーザー出口入力ファイルの例

```
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-------------------------------
  'IBM'     1042       -1          1      String spans multiple lines
  'IBM'     1044       -1          1      FIXED BIN 7 mapped to 1 byte
  'IBM'     1047        8          0      Order inhibits optimization
  'IBM'     1052       -1          1      Nodescriptor with * extent arg
  'IBM'     1059        0          0      Select without OTHERWISE
  'IBM'     1169        0          1      Precision of result determined
```

最初の 2 行はヘッダー行で、 IBMUEXIT では無視される。 残りの行には、可変数のブランクで区切られた入力データが含まれています。

ファイルの各列は、コンパイラー・ユーザー出口に次のように関係しています。

- 出口が適用されるすべてのコンパイラー・メッセージについて、単一引用符で囲まれた 'IBM' という文字が最初の列に含まれていなければなりません。
- 2 番目の列は、4 桁のメッセージ番号です。
- 3 番目の列は、新しいメッセージ重大度です。 重大度 -1 は、重大度がデフォルト値のままにすることを表します。
- 4 番目の列は、メッセージを抑止するかどうかを示します。 以下のいずれかの値を指定します。

  1
  メッセージを抑止します。

  0
  メッセージを出力します。
- 最後の欄にあるコメント欄は、あなたの情報を提供するためのもので、 IBMUEXIT では無視されます。

最初の 2 行はヘッダー行で、 IBMUEXIT では無視される。 残りの行には、可変数のブランクで区切られた入力データが含まれています。

ファイルの各列は、コンパイラー・ユーザー出口に次のように関係しています。

- 出口が適用されるすべてのコンパイラー・メッセージについて、単一引用符で囲まれた 'IBM' という文字が最初の列に含まれていなければなりません。
- 2 番目の列は、4 桁のメッセージ番号です。
- 3 番目の列は、新しいメッセージ重大度です。 重大度 -1 は、重大度がデフォルト値のままにすることを表します。
- 4 番目の列は、メッセージを抑止するかどうかを示します。 以下のいずれかの値を指定します。

  1
  Suppresses the message.

  0
  Prints the message.
- 最後の欄にあるコメント欄は、あなたの情報を提供するためのもので、 IBMUEXIT では無視されます。

独自のコンパイラー出口の作成

独自のユーザーエグジットを書くには、 IBMUEXIT をモデルとして使用できます。 出口を作成するときは、初期化、メッセージ・フィルター操作、および終了の領域を必ずカバーするようにしてください。

製品に付属するユーザー終了ソースコードは、サンプルデータセット SIBMZSAM のメンバー IBMUEXIT で入手できます。

コンパイラのユーザー終了は、LP(64) および NOEXPORTALL オプションを使用してコンパイルし、64 ビット・フェッチ可能な DLL としてリンクする必要があります。 31 ビット・コードを 31 ビット・フェッチ可能な DLL に入れ、 AMODE31 ENTRY オプションを使って 31 ビット・コードを呼び出すことで、31 ビット・コードを呼び出すことができます（『言語リファレンス・マニュアル』や本書の他の箇所で説明されています)。 これを簡単にするため、コンパイラがユーザー終了ルーチンに渡すストレージはすべてバーの下にあり、 POINTER はすべて POINTER(32)となる。

初期化プロシージャーの作成

初期化プロシージャーは、出口が必要とする初期化、 例えば、ファイルのオープンやストレージの割り振りなどを実行する必要があります。

初期化プロシージャー固有の制御ブロックは、以下のようにコーディングします。

```
Dcl 1 Uex_ISA native based,
    2 Uex_ISA_Length_fixed bin(31);   /* storage(Uex_ISA) *  /
```

初期化プロシージャーのグローバル制御ブロック構文については、448 ページの『グローバル制御ブロックの構造』を参照してください。

初期化プロシージャーの完了時には、戻りコード/理由コードを次のように設定する必要があります。

0/0
Continue compilation

4/n
Reserved for future use

8/n
Reserved for future use

12/n
Reserved for future use

16/n
Abort compilation

メッセージ・フィルター操作プロシージャーの作成

メッセージ・フィルター操作プロシージャーを使うと、メッセージを抑止したり、メッセージの重大度を変更したりすることができます。

どのメッセージの重大度も高くすることはできますが、 低くすることができるのは、**ERROR** (重大度コード 8) メッセージ または、**WARNING** (重大度コード 4) メッセージだけです。

プロシージャー特有の制御ブロックには、メッセージについての情報が含まれています。 これは、特定のメッセージの取り扱い方法を示す情報をコンパイラーに渡すために使われます。

プロシージャー固有のメッセージ・フィルター制御ブロックを以下に例示します。

```
Dcl 1 Uex_MFX native based,
      2 Uex_MFX_Length   fixed bin(31),

      2 Uex_MFX_Facility_Id  char(3),        /* of component writing
                                               message             */
      2 *                   char(1),
      2 Uex_MFX_Message_no   fixed bin(31),
      2 Uex_MFX_Severity     fixed bin(15),
      2 Uex_MFX_New_Severity fixed bin(15),  /* set by exit proc    */
      2 Uex_MFX_Inserts                fixed bin(15),
      2 Uex_MFX_Inserts_Data( 6 refer(Uex_MFX_Inserts) ),
        3 Uex_MFX_Ins_Type             fixed bin(7),
        3 Uex_MFX_Ins_Type_Data union unaligned,
          4 *                          char(8),
          4 Uex_MFX_Ins_Bin8           fixed bin(63),
          4 Uex_MFX_Ins_Bin            fixed bin(31),
          4 Uex_MFX_Ins_Str,
            5 Uex_MFX_Ins_Str_Len      fixed bin(15),
            5 Uex_MFX_Ins_Str_Addr     pointer(32),
          4 Uex_MFX_Ins_Series,
```

```
            5 Uex_MFX_Ins_Series_Sep   char(1),
            5 Uex_MFX_Ins_Series_Addr  pointer(32);
```

Uex_MFX_Ins_Series_Addr はこの構造を指している：

```
       dcl
         1 series based,
           2 series_count                              fixed bin(31),
           2 series_string( 1 refer(series_Count )  )  pointer(32);
```

データ入力フィールド

Uex_MFX_Length
Contains the length of the control block in bytes. The value is storage (Uex_MFX).

Uex_MFX_Facility_Id
Contains the ID of the facility; for the compiler, the ID is IBM. The compiler sets this value.

Uex_MFX_Message_no
Contains the message number that the compiler is going to generate. The compiler sets this value.

Uex_MFX_Severity
Contains the severity level of the message; it can be from one to fifteen characters in length. The compiler sets this value.

Uex_MFX_New_Severity
Contains the new severity level of the message; it can be from one to fifteen characters in length. The user sets this value.

Uex_MFX_Inserts
Contains the number of inserts for the message; it can range from zero to six. The compiler sets this value.

Uex_MFX_Inserts_Data
Contains fields to describe each of the inserts. The compiler sets these values.

Uex_MFX_Ins_Type
Contains the type of the insert. These are the possible insert types:

Uex_Ins_Type_Xb31
Is used for a FIXED BIN(31) and has the value 1.

Uex_Ins_Type_Char
Is used for a CHAR string and has the value 2.

Uex_Ins_Type_Series
Is used for a series of CHAR strings and has the value 3.

Uex_Ins_Type_Xb63
Is used for a FIXED BIN(63) and has the value 4.

The compiler sets this value.

Uex_MFX_Ins_Bin
Contains the integer value for an insert that has integer type. The compiler sets this value.

Uex_MFX_Ins_Str_Len
Contains the length (in bytes) for an insert that has character type. The compiler sets this value.

Uex_MFX_Ins_Str_Addr
Contains the address of the character string for an insert that has character type. The compiler sets this value.

Uex_MFX_Ins_Series_Sep
Contains the character that should be inserted between each element for an insert that has series type. Typically, this is a blank, period, or comma. The compiler sets this value.

Uex_MFX_Ins_Series_Addr
Contains the address of the series of varying character strings for an insert that has series type. The address points to a FIXED BIN(31) field holding the number of strings to concatenate followed by the addresses of those strings. The compiler sets this value.

メッセージ・フィルター操作プロシージャーの完了時に、戻りコード/理由コードを次のいずれかに設定してください。

0/0
Continue compilation, output message

0/1
Continue compilation, do not output message

4/n
Reserved for future use

8/n
Reserved for future use

16/n
Abort compilation

終了プロシージャーの作成

ファイルのクローズのような、必要なクリーンアップを実行するには、 終了プロシージャーを使います。また、エラー・メッセージ・フィルター・プロシージャーと 初期化プロシージャーの実行時に収集される情報に基づいて、 最終統計レポートを作成することもできます。

終了プロシージャー固有の制御ブロックは、以下のようにコーディングします。

```
Dcl 1 Uex_ISA native based,
    2 Uex_ISA_Length_fixed bin(31); /* storage(Uex_ISA)       */
```

終了プロシージャーのグローバル制御ブロック構文については、448 ページの『グローバル制御ブロックの構造』を参照してください。 終了プロシージャーの完了時に、戻りコード/理由コードを次のいずれかに設定してください。

0/0
コンパイルを続行する

4/n
将来の利用のために予約済み

8/n
将来の利用のために予約済み

12/n
将来の利用のために予約済み

16/n
コンパイルを打ち切る

SQL メッセージの抑制例

サンプル・データ・セット SIBMZSAM のメンバ IBMUEXSQ には、メッセージの挿入を検査し、2 つの SQL 情報メッセージと 1 つの SQL 警告メッセージを抑制するユーザ exit を修正したソース・コードが含まれています。

このバージョンのユーザー・エグジットは、LP(64) および NOEXPORTALL オプションを付けてコンパイルし、64 ビットのフェッチ可能な DLL としてリンクする必要があります。

第 26 章 PL/I 記述子

この章では、実行時の PL/I ルーチン間での PL/I パラメーターの引き渡しの規則について説明します。

記述子以外の Language Environment・ランタイム環境に関する考慮事項について詳しくは、「*z/OS Language Environment Programming Guide*」を参照してください。 この資料では、ランタイム環境の規則と、その規則をサポートするアセンブラー・マクロについて説明しています。

引数の引き渡し

文字列、配列、構造体が引数として渡される場合、呼び出されるルーチンが OPTIONS(NODESCRIPTOR) で宣言されていない限り、コンパイラーはその引数の記述子を渡します。

このような記述子を渡す方法として次の 2 つがあります。

- 記述子リストを使う
- 記述子ロケーターを使う

これら 2 つの方法それぞれについて、次の主要機能に注意してください。

- **引数が記述子リストを使って渡されるとき**
  - 渡される引数の数は、いずれかの引数に記述子が必要であるときには、 指定された引数の数より 1 大きい数になります。
  - ディスクリプタで渡された引数は、値 (BYVALUE) で渡されたポインタとして受け取ることができる。
  - DEFAULT(DESCLIST) コンパイラー・オプションが有効な場合、コンパイラーはこのメソッドを使用する。
- **引数が記述子ロケーターによって渡されるとき**
  - 渡される引数の数は常に、指定された引数の数と一致します。
  - 記述子とともに渡される引数は、参照渡しポインタとして受け取ることができる (BYADDR)。
  - DEFAULT(DESCLOCATOR) コンパイラー・オプションが有効な場合、コンパイラーはこのメソッドを使用する。

記述子リストによる引数の引き渡し

引数とその記述子が記述子リストを使って渡される 場合は、1 つでも記述子を必要とする引数があれば、1 つの追加の引数が渡されます。

この追加の引数は、ポインターのリストを指すポインターです。 このリストの入力項目の数は、渡される引数の数と一致します。 記述子を必要としない引数の場合、記述子リストの対応するポインタは SYSNULL に設定される。 記述子が必要である引数の場合は、記述子リストの中で、 対応するポインターがその引数の記述子のアドレスに設定されます。

例えば、次のようにルーチン `sample` が宣言されたとします。

```
declare sample entry( fixed bin(31), varying char(*) )
               options( byaddr descriptor );
```

`sample` が次のようなステートメントで呼び出されたとします。

```
call sample( 1, 'test' );
```

この場合、次の 3 つの引数がルーチンに渡されます。

- 値が 1 の固定 bin(31) 一時変数のアドレス
- 値が `test` の可変 char(4) 一時変数のアドレス
- 次の項目で構成される記述子リストのアドレス

© Copyright IBM Corp. 1999, 2025

- SYSNULL()
- 可変 char(4) ストリング用の記述子のアドレス

ロケーター/記述子で引数を渡す

引数とその記述子がロケーター/記述子で渡される場合、引数に記述子が必要となるときは必ず、その引数のロケーター/記述子のアドレスが代わりに渡されます。

ストリングを除いて、ロケーター/記述子は、ポインターの対です。 最初のポインターはデータのアドレスで、2 番目のポインターは記述子のアドレスです。 文字列の場合、CMPAT(LE) の下では、ロケータ/ディスクリプタはまだそのようなポインタのペアである。 しかし、他の CMPAT・オプションでは、ロケーター／ディスクリプタは、文字列のアドレスと、文字列ディスクリプタ自体で構成される。

例えば、ルーチン sample が次のように宣言されるとします。

```
declare sample entry( fixed bin(31), varying char(*) )
               options( byaddr descriptor );
```

sample が次のようなステートメントで呼び出されたとします。

```
call sample( 1, 'test' );
```

この場合、次の 2 つの引数がルーチンに渡されます。

- 値が 1 の固定 bin(31) 一時変数のアドレス
- 以下のアドレスおよび記述子で構成されるロケーター/記述子のアドレス
  - 値が test の可変 char(4) 一時変数のアドレス
  - CMPAT(LE) の下で、char(4) 文字列を変化させるための CMPAT(LE) 記述子のアドレス
  - CMPAT(V*) では、char(4) 文字列を変化させるための CMPAT(V*) 記述子

CMPAT(V*) 記述子

LE 記述子とは異なり、 CMPAT(V*) 記述子は自己記述的ではない。 しかし、文字列記述子はすべての CMPAT(V*) オプションで同じであり、LE 文字列記述子 と同じコードページ符号化方式を共有する。

ストリング記述子

ストリング記述子では、最初の 2 バイトは、ストリングの最大長を指定します。 この最大長は常にネイティブ・フォーマットで保持されます。

3 バイト目にはさまざまなフラグが含まれる（たとえば、 VARYING の文字列の長さがリトルエンディアンかビッグエンディアンのどちらで保持されているか、WIDECHAR の文字列のデータがリトルエンディアンかビッグエンディアンのどちらで保持されているかを示す）。

非可変ビット・ストリングのストリング記述子では、 4 番目のバイトはビット・オフセットを表します。

CHARACTER 文字列の文字列記述子では、4 バイト目がコンパイラの CODEPAGE オプションをエンコードする。

CMPAT(V1) and CMPAT(V2 ）の文字列記述子の宣言は以下の通りである：

```
declare
  1 dso_string based,
    2 dso_string_length          fixed bin(15),
    2 dso_string_flags,
      3 dso_string_is_varying           bit(1),
      3 dso_string_is_varyingz          bit(1),
      3 dso_string_has_nonnative_len    bit(1), /* for varying     */
      3 dso_string_is_ascii             bit(1), /* for char        */
      3 dso_string_has_nonnative_data   bit(1), /* for wchar       */
      3 *                               bit(1), /* reserved, '0'b  */
      3 *                               bit(1), /* reserved, '0'b  */
      3 *                               bit(1), /* reserved, '0'b  */
    2 * union,
```

```
      3 dso_String_Codepage      ordinal ccs_Codepage_Enum,
      3 dso_string_bitofs        fixed bin(8) unsigned,
    2 dso_string_end     char(0);
```

CMPAT(V3) 、文字列記述子の宣言は以下のようになる：

```
declare
  1 dso_longstr based,
    2 dso_longstr_info,
      3 *                   fixed bin(8) unsigned,
      3 dso_datatype        fixed bin(8) unsigned,
      3 * union,
        4 dso_longstr_bitofs    fixed bin(8) unsigned,
        4 dso_longstr_codepage  type ccs_Codepage_Enum,
      3 dso_longstr_info2,
        4 dso_longstr_has_nonnative_len   bit(1), /* for varying    */
        4 dso_longstr_is_ebcdic           bit(1), /* for char       */
        4 dso_longstr_has_nonnative_data  bit(1), /* for wchar      */
        4 *                               bit(1),
        4 dso_longstr_is_varying          bit(1),
        4 dso_longstr_is_varyingz         bit(1),
        4 dso_longstr_is_varying4         bit(1),
        4 *                               bit(1),
    2 dso_longstr_length          fixed bin(31),
    2 dso_longstr_end     char(0);
```

コード・ページ・エンコードに使用できる値は次のように定義されます。

```
define ordinal
  ccs_Codepage_Enum
    ( ccs_Codepage_01047  value(1)
     ,ccs_Codepage_01140
     ,ccs_Codepage_01141
     ,ccs_Codepage_01142
     ,ccs_Codepage_01143
     ,ccs_Codepage_01144
     ,ccs_Codepage_01145
     ,ccs_Codepage_01146
     ,ccs_Codepage_01147
     ,ccs_Codepage_01148
     ,ccs_Codepage_01149
     ,ccs_Codepage_00819
     ,ccs_Codepage_00813
     ,ccs_Codepage_00920
     ,ccs_Codepage_00037
     ,ccs_Codepage_00273
     ,ccs_Codepage_00277
     ,ccs_Codepage_00278
     ,ccs_Codepage_00280
     ,ccs_Codepage_00284
     ,ccs_Codepage_00285
     ,ccs_Codepage_00297
     ,ccs_Codepage_00500
     ,ccs_Codepage_00871
     ,ccs_Codepage_01026
     ,ccs_Codepage_01155
    ) unsigned prec(8);
```

配列記述子

以下の宣言では、配列の上限は 15 として宣言されていますが、実際の上限は常に記述されている配列の次元数に一致しているということを理解する必要があります。

CMPAT(V1) 配列記述子の宣言は以下の通り：

```
declare
  1 dso_v1  based,
    2 dso_v1_rvo      fixed bin(31),    /* relative virtual origin */
    2 dso_v1_data(1:15),
      3 dso_v1_stride fixed bin(31),    /*   multiplier            */
      3 dso_v1_hbound fixed bin(15),    /*   hbound                */
      3 dso_v1_lbound fixed bin(15);    /*   lbound                */
```

CMPAT(V2) 配列記述子の宣言は以下の通り：

```
declare
  1 dso_v2  based,
    2 dso_v2_rvo      fixed bin(31),    /* relative virtual origin */
    2 dso_v2_data(1:15),
      3 dso_v2_stride fixed bin(31),    /*   multiplier            */
      3 dso_v2_hbound fixed bin(31),    /*   hbound                */
      3 dso_v2_lbound fixed bin(31);    /*   lbound                */
```

CMPAT(V3) 配列記述子の宣言は以下の通り：

```
declare
  1 dso_v3  based,
    2 dso_v3_rvo      fixed bin(63),    /* relative virtual origin */
    2 dso_v3_data(1:15),
      3 dso_v3_stride fixed bin(63),    /*   multiplier            */
      3 dso_v3_hbound fixed bin(63),    /*   hbound                */
      3 dso_v3_lbound fixed bin(63);    /*   lbound                */
```

CMPAT(LE) 記述子

すべての LE 記述子は 4 バイトのフィールドで始まる。 最初のバイトは、記述子のタイプ (スカラー、配列、構造体、または共用体) を指定します。 残りの 3 バイトは、特定の記述子タイプで設定されていない限り、ゼロになります。

記述子ヘッダーの宣言は次のとおりです。

```
declare
  1 dsc_Header based( sysnull() ),
    2 dsc_Type          fixed bin(8) unsigned,
    2 dsc_Datatype      fixed bin(8) unsigned,
    2 *                 fixed bin(8) unsigned,
    2 *                 fixed bin(8) unsigned;
```

これらは、 dsc_Type フィールドで取り得る値である：

```
declare
  dsc_Type_Unset               fixed bin(8) value(0),
  dsc_Type_Element             fixed bin(8) value(2),
  dsc_Type_Array               fixed bin(8) value(3),
  dsc_Type_Structure           fixed bin(8) value(4),
  dsc_Type_Union               fixed bin(8) value(4);
```

ストリング記述子

ストリング記述子では、ヘッダーの 2 番目のバイトがストリング・タイプ (文字、ビット、グラフィック、uchar、またはワイド文字 のほかに、nonvarying、varying、または varyingz) を表します。

非可変ビット・ストリングのストリング記述子では、 ヘッダーの 3 番目のバイトがビット・オフセットを表します。

CHARACTER 文字列の文字列記述子では、ヘッダーの 3 バイト目がコンパイラの CODEPAGE オプションをエンコードする。

可変ストリングのストリング記述子では、4 番目のバイトに、ストリング長がネイティブ以外のフォーマットで保持されるかどうかを表すビットが含まれます。

文字列の文字列記述子では、4 バイト目にも文字列データが EBCDIC であるかどうかを示すビットがある。

ストリング記述子の宣言は次のとおりです。

```
declare
  1 dsc_String  based( sysnull() ),
    2 dsc_String_Header,
      3 *                    fixed bin(8) unsigned,
      3 dsc_String_Type      fixed bin(8) unsigned,
      3 * union,
        4 dsc_String_Codepage ordinal ccs_Codepage_Enum,
```

```
        4 dsc_String_BitOfs   fixed bin(8) unsigned,
      3 *,
        4 dsc_String_Has_Nonnative_Len    bit(1),
        4 dsc_String_Is_Ebcdic            bit(1),
        4 dsc_String_Has_Nonnative_Data   bit(1),
        4 *                               bit(5),
    2 dsc_String_Length   fixed bin(31); /* max length of string */
```

これらは、 dsc_String_Type フィールドで取り得る値である：

```
declare
    dsc_datatype_unset                  fixed bin(7) value(0),
    dsc_datatype_char_nonvarying        fixed bin(7) value(2),
    dsc_datatype_char_varyingz          fixed bin(7) value(3),
    dsc_datatype_char_varying2          fixed bin(7) value(4),
    dsc_datatype_char_varying4          fixed bin(7) value(5),
    dsc_datatype_bit_nonvarying         fixed bin(7) value(6),
    dsc_datatype_bit_varying2           fixed bin(7) value(7),
    dsc_datatype_bit_varying4           fixed bin(7) value(8),
    dsc_datatype_graphic_nonvarying     fixed bin(7) value(9),
    dsc_datatype_graphic_varyingz       fixed bin(7) value(10),
    dsc_datatype_graphic_varying2       fixed bin(7) value(11),
    dsc_datatype_graphic_varying4       fixed bin(7) value(12),
    dsc_datatype_widechar_nonvarying    fixed bin(7) value(13),
    dsc_datatype_widechar_varyingz      fixed bin(7) value(14),
    dsc_datatype_widechar_varying2      fixed bin(7) value(15),
    dsc_datatype_widechar_varying4      fixed bin(7) value(16),
    dsc_datatype_uchar_nonvarying       fixed bin(7) value(17),
    dsc_datatype_uchar_varyingz         fixed bin(7) value(18),
    dsc_datatype_uchar_varying2         fixed bin(7) value(19),
    dsc_datatype_uchar_varying4         fixed bin(7) value(20);
```

配列記述子

配列記述子の宣言は次のとおりです。

```
declare
  1 dsc_Array  based( sysnull() ),
    2 dsc_Array_Header   like dsc_Header,
    2 dsc_Array_EltLen   fixed bin(31), /* Length of array element */
    2 dsc_Array_Rank     fixed bin(31), /* Count of dimensions     */
    2 dsc_Array_RVO      fixed bin(31), /* Relative virtual origin */
    2 dsc_Array_Data( 1: 1 refer(dsc_Array_Rank) ),
      3 dsc_Array_LBound fixed bin(31), /*   LBound                */
      3 dsc_Array_Extent fixed bin(31), /*   HBound - LBound + 1   */
      3 dsc_Array_Stride fixed bin(31); /*   Multiplier            */
```

付録 A SYSADATA メッセージ情報

XINFO コンパイル時オプションの MSG サブオプションを指定すると、コンパイラーは SYSADATA ファイルを生成する。

SYSADATA ファイルには以下のレコードが含まれている：

- カウンター・レコード
- リテラル・レコード
- ファイル・レコード
- メッセージ・レコード
- オプション・レコード

すべての ADATA レコードの宣言の完全なセットは、サンプルデータセット SIBMZSAM のメンバー `ibmwxin` と `ibmwxop` で利用可能です。

ファイル内のレコードは、必ずしも上記にリストした順序で生成されるとはかぎらず、例えば、リテラル・レコードとファイル・レコードが交互に混ざり合う場合があることに注意してください。 SYSADATA ファイルを読み込むコードを書く場合、以下の例外を除き、ファイル内のレコードの順番に依存してはならない：

- カウンター・レコードは、ファイル内の最初のレコードになります。
- それぞれのリテラル・レコードは、それが定義するリテラルの参照よりも前にきます。
- それぞれのファイル・レコードは、それが記述するファイルの参照よりも前にきます。

SYSADATA

SYSADATA ファイルはシーケンシャルなバイナリファイルである。

z/OS バッチでは、コンパイラーは SYSADATA DD 文で指定されたファイルに SYSADATA レコードを書き込み、そのファイルは PDS のメンバーであってはならない。 その他のすべてのシステムでは、コンパイラーは拡張子 `adt` を持つファイルに書き込みます。

ファイル内のそれぞれのレコードには、ヘッダーが入っています。 この 12 バイトのヘッダーには、以下のフィールドがあり、これはそのファイル内のすべてのレコードで同じです。

コンパイラー
データを生成したコンパイラーを表す番号。 PL/I の場合、この番号は 40 です。

エディション番号
データを生成したコンパイラーのエディション番号。 この製品の場合、この番号は 2 です。

SYSADATA レベル
このファイル形式が表す SYSADATA のレベルを表す数字。 この製品の場合、この番号は 4 です。

ヘッダーには、レコードごとに変わる、以下のフィールドもあります。

- レコード・タイプ
- レコードが次のレコードに継続するかどうか

可能なレコードタイプは、 レコードタイプを序数値としてエンコードするで示したように、序数値としてエンコードされる。

序数値としてエンコードされたレコード・タイプ

```
Define ordinal xin_Rect
 (Xin_Rect_Msg       value(50),   /*  Message record              */
  Xin_Rect_Fil       value(57),   /*  File record                 */
  Xin_Rect_Opt       value(60),   /*  Options record              */
  Xin_Rect_Sum       value(61),   /*  Summary record              */
  Xin_Rect_Rep       value(62),   /*  Replace record              */
  Xin_Rect_Src       value(63),   /*  Source record               */
```

© Copyright IBM Corp. 1999, 2025

```
   Xin_Rect_Tok       value(64),   /*  Token record                */
   Xin_Rect_Sym       value(66),   /*  Symbol record               */
   Xin_Rect_Lit       value(67),   /*  Literal record              */
   Xin_Rect_Syn       value(69),   /*  Syntax record               */
   Xin_Rect_Ord_Type  value(80),   /*  Ordinal type record         */
   Xin_Rect_Ord_Elem  value(81),   /*  Ordinal element record      */
   Xin_Rect_Ctr       value(82) )  /*  Counter record              */
                      prec(15);
```

レコードのヘッダー部分の宣言を以下に示す。

レコードのヘッダー部分を宣言する：

```
Dcl
  1 Xin_Hdr based,                 /*  Header portion              */
                                   /*                              */
    2 Xin_Hdr_Prod                 /*  Language code               */
           fixed bin(8) unsigned, /*                              */
                                   /*                              */
    2 Xin_Hdr_Rect                 /*  Record type                 */
        unal ordinal xin_Rect,     /*                              */
                                   /*                              */
    2 Xin_Hdr_Level                /*  SYSADATA level              */
           fixed bin(8) unsigned, /*                              */
                                   /*                              */
    2 * union,                     /*                              */
      3 xin_Hdr_Flags     bit(8),  /*  flags                       */
      3 *,                         /*                              */
        4 *               bit(6),  /*  Reserved                    */
        4 Xin_Hdr_Little_Endian    /*  ints are little endian      */
                          bit(1),  /*                              */
        4 Xin_Hdr_Cont    bit(1),  /*  Record continued in next rec */
                                   /*                              */
    2 Xin_Hdr_Edition              /*  compiler "edition"          */
           fixed bin(8) unsigned, /*                              */
                                   /*                              */
    2 Xin_Hdr_Fill        bit(32), /*  reserved                    */
                                   /*                              */
    2 Xin_Hdr_Data_Len             /*  length of data part         */
          fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Hdr_End         char(0); /*                              */
```

サマリー・レコード

タイプ Xin_Rect_Sum のレコードは要約レコードで、ファイルの最初のレコードである。

宣言を 462 ページの『#unique_753/unique_753_Connect_42_sysads』 に示します。

サマリー・レコードの宣言

```
Dcl
  1 Xin_Sum     Based( ),          /* summary record              */
                                   /*                             */
    2 Xin_Sum_Hdr                  /* standard header             */
           like Xin_Hdr,           /*                             */
                                   /*                             */
    2 Xin_Sum_Max_Severity         /* max severity from compiler  */
           fixed bin(32) unsigned, /*                             */
                                   /*                             */
    2 Xin_Sum_Left_Margin          /* left margin                 */
           fixed bin(16) unsigned, /*                             */
                                   /*                             */
    2 Xin_Sum_Right_Margin         /* right margin                */
           fixed bin(16) unsigned, /*                             */
                                   /*                             */
    2 xin_Sum_Rsrvd(15)            /* reserved                    */
           fixed bin(32) unsigned; /*                             */
```

オプション・レコード

タイプ Xin_Rect_Opt のレコードがオプション・レコードである。 これは、コンパイル時に有効となるすべてのコンパイラー・オプションをリストします。

オプション・レコードの宣言は、サンプル・データセット SIBMZSAM のメンバ ibmwxop にあります。

カウンター・レコード

各カウンター・レコードは、後続のレコード・タイプについて、ファイルに入っている そのタイプのレコードの数と、それらのレコードが占有するバイト数を指定します。

カウンター・レコードの宣言

```
Dcl
  1 xin_Ctr     based,             /* counter/size record           */
                                   /*                               */
    2 xin_Ctr_Hdr                  /* standard header               */
          like xin_Hdr,            /*                               */
                                   /*                               */
    2 xin_Ctr_Rect                 /* record type                   */
        unal ordinal xin_Rect,     /*                               */
                                   /*                               */
    2 *                            /*                               */
          fixed bin(16) unsigned, /*                               */
                                   /*                               */
    2 xin_Ctr_Count                /* count of that record type     */
          fixed bin(31) unsigned, /*                               */
                                   /*                               */
    2 xin_Ctr_Size                 /* size used                     */
          fixed bin(31) unsigned; /*                               */
```

リテラル・レコード

それぞれのリテラル・レコードは、リテラル索引と呼ばれる 1 つの番号を割り当て、 この特定のレコードに指定された文字を参照するために、後のレコードによって使用されるようにします。

リテラル・レコードの宣言

```
Dcl
 1 xin_Lit     based,              /* literal record                */
                                   /*                               */
    2 xin_Lit_Hdr                  /* standard header               */
          like xin_Hdr,            /*                               */
                                   /*                               */
    2 xin_Lit_Inx                  /* adata index for literal       */
          fixed bin(31) unsigned, /*                               */
                                   /*                               */
    2 xin_Lit_Len                  /* length of literal             */
          fixed bin(31) unsigned, /*                               */
                                   /*                               */
    2 xin_Lit_Val      char(2000); /* literal value                 */
```

ファイル・レコード

それぞれのファイル・レコードは、ファイル索引と呼ばれる 1 つの番号を割り当て、このレコード に記述されたファイルを参照するために、後のレコードによって使用されるようにします。 記述されたファイルは、プライマリ・ソース・ファイル（ PL/I ）または INCLUDEd 。 それぞれのファイル・レコードは、そのファイルの完全修飾名に対するリテラル索引 を指定します。

INCLUDEd ファイルの場合、各ファイルレコードには、 INCLUDE リクエストの元となったファイルインデックスとソース行番号も含まれる。 (1 次ソース・ファイルの場合、これらのフィールドはゼロになります。)

ファイル・レコードの宣言

```
Dcl
  1 xin_Fil      based,             /* file record                   */
                                   /*                               */
    2 xin_Fil_Hdr                  /* standard header               */
          like xin_Hdr,            /*                               */
                                   /*                               */
    2 xin_Fil_File_Id              /* file id from whence it        */
          fixed bin(31) unsigned, /*  was INCLUDEd                  */
                                   /*                               */
    2 xin_Fil_Line_No              /* line no within that file      */
          fixed bin(31) unsigned, /*                               */
                                   /*                               */
    2 xin_Fil_Id                   /* id assigned to this file      */
```

```
           fixed bin(31) unsigned, /*                                */
                                   /*                                */
    2 xin_Fil_Name                 /* literal index of the           */
           fixed bin(31) unsigned; /*  fully qualified file name      */
```

メッセージ・レコード

それぞれのメッセージ・レコードは、コンパイル中に出されたメッセージを記述します。 抑止されたメッセージに対しては、メッセージ・レコードは生成されません。

各メッセージ・レコードには以下のデータが含まれています。

- そのメッセージの起因となったファイルおよび行のファイル索引およびソース行番号。 メッセージがすべてコンパイルに関係するものである場合、このフィールドはゼロになります。
- メッセージに関連する識別子(例えば、 IBM1502)と重大度。
- そのメッセージのテキスト。

メッセージ・レコードの宣言は、以下のとおりです。

メッセージ・レコードの宣言

```
Dcl
  1 xin_Msg     based,             /* message record                */
                                   /*                               */
    2 xin_Msg_Hdr                  /* standard header               */
           like xin_Hdr,           /*                               */
                                   /*                               */
    2 xin_Msg_File_Id              /* file id                       */
           fixed bin(31) unsigned, /*                               */
                                   /*                               */
    2 xin_Msg_Line_No              /* line no within file           */
           fixed bin(31) unsigned, /*                               */
                                   /*                               */
    2 xin_Msg_Id                   /* identifier (i.e. IBM1502)     */
                         char(16), /*                               */
                                   /*                               */
    2 xin_Msg_Severity             /* severity (0, 4, 8, 12 or 16)  */
           fixed bin(15)   signed, /*                               */
                                   /*                               */
    2 xin_Msg_Length               /* length of message             */
           fixed bin(16) unsigned, /*                               */
                                   /*                               */
    2 Xin_Msg_Text                 /* actual message                */
        char( 100 refer(xin_Msg_Length) );
```

SYSADATA シンボル情報を理解する

XINFO コンパイル時オプションの SYM サブオプションを指定すると、コンパイラーは MSG サブオプションで生成されるレコードに加えて、シンボル情報を含む SYSADATA ファイルを生成します。

以下のレコードにシンボル情報が含まれています。

- 序数タイプ・レコード
- 序数エレメント・レコード
- シンボル・レコード

シンボル・レコードは、組み込み関数、総称変数、または非定数エクステントを持つ変数の場合には生成されません。

序数タイプ・レコード

それぞれの序数タイプ・レコードは、序数タイプ索引と呼ばれる 1 つの番号を割り当て、このレコードに記述された序数タイプを参照するために、後のレコードによって使用されるようにします。 タイプの名前は、リテラル索引によって示されます。 それぞれの序数タイプ・レコードには、その序数タイプが宣言されたファイルおよび行のファイル索引およびソース行番号が入っています。

それぞれの序数タイプ・レコードには、以下のものが入っています。

- そのタイプによって定義された値の数のカウント
- そのタイプに関連する精度
- 符号付きであるか符号なしであるかを示すビット

序数タイプ・レコードの宣言

```
declare                              /*                              */
  1 xin_Ord_Type   based,            /*                              */
                                     /*                              */
    2 xin_Ord_Type_Hdr               /* standard header              */
         like xin_Hdr,               /*                              */
                                     /*                              */
    2 xin_Ord_Type_File_Id           /* file id                      */
         fixed bin(31) unsigned,     /*                              */
                                     /*                              */
    2 xin_Ord_Type_Line_No           /* line no within file          */
         fixed bin(31) unsigned,     /*                              */
                                     /*                              */
    2 xin_Ord_Type_Id                /* identifying number           */
                  fixed bin(31),     /*                              */
                                     /*                              */
    2 xin_Ord_Type_Count             /* count of elements            */
                  fixed bin(31),     /*                              */
                                     /*                              */
    2 xin_Ord_Type_Prec              /* precision for ordinal        */
         fixed bin(08) unsigned,     /*                              */
                                     /*                              */
    2 *,                             /*                              */
      3 xin_Ordinal_Type_Signed      /* signed attribute applies     */
                        bit(1),      /*                              */
      3 xin_Ordinal_Type_Unsigned    /* unsigned attribute applies   */
                        bit(1),      /*                              */
      3 *                            /* unused                       */
                        bit(6),      /*                              */
                                     /*                              */
    2 *                              /* unused                       */
                       char(2),      /*                              */
                                     /*                              */
    2 xin_Ord_Type_Name              /* type name                    */
                  fixed bin(31);     /*                              */
```

序数エレメント・レコード

それぞれの序数タイプ・レコードの直後には、その序数によって指定される値 を記述する、一連のレコードが (序数タイプ・カウントで指定された数だけ) 続きます。 それぞれの序数エレメント・レコードは、序数エレメント索引と呼ばれる 1 つの番号を割り当て、このレコードに記述された序数エレメントを参照するために、後のレコードによって使用されるようにします。

エレメントの名前は、リテラル索引によって示されます。 それぞれの序数エレメント・レコードには、その序数エレメントが宣言されたファイルおよび行のファイル索引およびソース行番号が入っています。

さらに、各序数エレメント・レコードには、以下のデータが含まれています。

- そのエレメントが属している序数タイプの序数タイプ索引
- そのエレメントの値

序列要素レコードの宣言：

```
declare                              /*                              */
  1 xin_Ord_Elem   based,            /*                              */
                                     /*                              */
    2 xin_Ord_Elem_Hdr               /* standard header              */
         like xin_Hdr,               /*                              */
                                     /*                              */
    2 xin_Ord_Elem_File_Id           /* file id                      */
         fixed bin(31) unsigned,     /*                              */
                                     /*                              */
    2 xin_Ord_Elem_Line_No           /* line no within file          */
         fixed bin(31) unsigned,     /*                              */
                                     /*                              */
    2 xin_Ord_Elem_Id                /* identifying number           */
                  fixed bin(31),     /*                              */
                                     /*                              */
    2 xin_Ord_Elem_Type_Id           /* id of ordinal type           */
```

```
                    fixed bin(31),  /*                              */
                                    /*                              */
    2 xin_Ord_Elem_Value            /* ordinal value                */
                    fixed bin(31),  /*                              */
                                    /*                              */
    2 xin_Ord_Elem_Name             /* ordinal name                 */
                    fixed bin(31);  /*                              */
```

シンボル・レコード

シンボルレコードの宣言は、サンプルデータセット SIBMZSAM のメンバー ibmwxin にある。 シンボル・レコードにはそれぞれ、シンボル索引と呼ばれる数値が割り当てられます。 索引は、このレコードによって記述されるシンボルを参照するために、それ以降のレコードによって使用されます。

例えば、ユーザー変数または定数の名前に索引を使用することができます。 ID の名前は、リテラル索引によって示されます。 それぞれのシンボル・レコードには、そのシンボルが宣言されたファイルおよび行のファイル索引およびソース行番号が入っています。

ID が構造体または共用体の一部である場合、シンボル・レコード には、以下のそれぞれに対するシンボル索引が入ります。

- 最初の兄弟 (存在する場合)
- 親 (存在する場合)
- 最初の子 (存在する場合)

以下の構造体を見てください。

```
      dcl
        1 a
          , 3 b     fixed bin
          , 3 c     fixed bin
          , 3 d
            , 5 e   fixed bin
            , 5 f   fixed bin
      ;
```

上記の構造体のエレメントに割り当てられたシンボル索引は、以下のようになります。

構造体のエレメントに割り当てられたシンボル索引

symbol	index	sibling	parent	child
a	1	0	0	2
b	2	3	1	0
c	3	4	1	0
d	4	0	1	5
e	5	6	4	0
f	6	0	4	0

それぞれのシンボル・レコードには、さまざまな属性がこの変数に適用されるかどうか を示す一連の bit(1) フィールドも含まれています。

それぞれのシンボル・レコードには、以下のエレメントも入っています。

ユーザー指定の構造体レベル
これは、ID に対するユーザー指定の構造体レベルです。 上記の構造のエレメント c の場合、値は 3 です。 非構造体メンバーの場合、値は 1 に設定されます。

論理構造レベル
ID の論理構造レベル。上記の構造のエレメント c の場合、値は 2 です。 非構造体メンバーの場合、値は 1 に設定されます。

次元

継承次元をカウントせずに、変数に宣言された次元の数。

すべての継承次元を含む、変数の次元の数。

オフセット
最外部の親構造体の中でのオフセット。

基本サイズ
基本サイズは、変数がビット位置合わせしている場合はビット単位、 それ以外の場合はバイト単位です。 いずれの場合も、これは、どの次元の因数でもありません。

サイズ
その次元の因数となるバイト単位でのサイズ。

位置合わせ
以下によって、示されます。

- 0 (ビット位置合わせ)
- 7 (バイト位置合わせ)
- 15 (ハーフワード位置合わせ)
- 31 (フルワード位置合わせ)
- 63 (4 倍長ワード位置合わせ)

レコード内の共用体は、変数のストレージ・クラスに従属する情報を記述するための専用のものです。

静的変数
変数が、別個の外部名を持つ外部 (dcl x ext('y')) として宣言された場合、 その名前のリテラル索引が指定されます。

基底付き変数
変数が、配列のエレメントではない別のマップ変数を基にするものとして宣言された場合、 その変数のシンボル索引が指定されます。

定義済み変数
変数が、配列のエレメントではない別のマップされた変数で定義されたものとして宣言された場合、その変数のシンボル索引がここに指定されます。 これは、その位置属性が定数である場合にも指定されます。

変数のデータ・タイプは、467 ページの『#unique_762/unique_762_Connect_42_ibma1ap011008655』に示された序数 によって指定されます。

変数のデータ・タイプ

```
define
  ordinal
    xin_Data_Kind
      (  xin_Data_Kind_Unset
        ,xin_Data_Kind_Character
        ,xin_Data_Kind_Bit
        ,xin_Data_Kind_Graphic
        ,xin_Data_Kind_Fixed
        ,xin_Data_Kind_Float
        ,xin_Data_Kind_Picture
        ,xin_Data_Kind_Pointer
        ,xin_Data_Kind_Offset
        ,xin_Data_Kind_Entry
        ,xin_Data_Kind_File
        ,xin_Data_Kind_Label
        ,xin_Data_Kind_Format
        ,xin_Data_Kind_Area
        ,xin_Data_Kind_Task
        ,xin_Data_Kind_Event
        ,xin_Data_Kind_Condition
        ,xin_Data_Kind_Structure
        ,xin_Data_Kind_Union
        ,xin_Data_Kind_Descriptor
        ,xin_Data_Kind_Ordinal
        ,xin_Data_Kind_Handle
        ,xin_Data_Kind_Type
        ,xin_Data_Kind_Builtin
        ,xin_Data_Kind_Generic
        ,xin_Data_Kind_Widechar
                                  )  prec(8) unsigned;
```

レコード内の共用体は、変数のデータ・タイプに従属する情報を記述するための専用のものです。 この情報 (算術型の精度など) の多くは説明不要です。 ただし、以下の変数は例外と考えられます。

ピクチャー変数
ピクチャー指定のリテラル索引が指定されます。

入り口変数
変数が戻り属性を持っている場合、戻り記述のシンボル索引が指定されます。

序数変数
序数タイプ索引が指定されます。

タイプ付き変数およびハンドル
基礎となるタイプのシンボル索引が指定されます。

ストリングおよび区域変数
戻り記述のシンボル索引に加えて、エクステントのタイプと値が指定されます。 エクステントのタイプは、以下の値によってエンコードされます。

```
declare
  (  xin_Extent_Constant     value(01)
    ,xin_Extent_Star         value(02)
    ,xin_Extent_Nonconstant  value(04)
    ,xin_Extent_Refer        value(08)
    ,xin_Extent_In_Error     value(16)
  )
  fixed bin;
```

エレメントが何らかの次元を持つ場合、その下部と上部の境界のタイプと値が、 レコードの最後に指定されます。 これらのフィールドは、エレメントが次元を持たない場合には存在しません。

属性フラグには、コンパイラーがすべてのデフォルトを適用した後の属性が反映されることにも注意してください。 例えば、すべての数値変数（数値 PICTURE 変数を含む）には、 REAL または COMPLEX 属性フラグが設定されています。

SYSADATA 構文情報を理解する

XINFO コンパイル時オプションの SYN サブオプションを指定すると、コンパイラーは、MSG サブオプションと SYM サブオプションで生成されるレコードに加えて、構文情報を含む SYSADATA ファイルを生成する。

以下のレコードに構文情報が含まれています。

- ソース・レコード
- トークン・レコード
- 構文レコード

ソース・レコード

各ソース・レコードでは、ソース ID と呼ばれる 1 つの番号が割り当てられます。後のレコードでこのレコードに記述されたソース行を参照するときは、この番号が使用されます。 その行は、レコード内のソースファイル ID と行番号フィールドによって示されるように、プライマリ PL/I ソースファイルまたは INCLUDE ファイルからのものである可能性がある。 このレコードの残りの部分は、ソース行の実データを保持します。

ソース・レコードの宣言

```
Dcl
  1 Xin_Src     based,             /* source record               */
                                   /*                             */
    2 Xin_Src_Hdr                  /* standard header             */
          like Xin_Hdr,            /*                             */
                                   /*                             */
    2 Xin_Src_File_Id              /* file id                     */
          fixed bin(32) unsigned,  /*                             */
                                   /*                             */
    2 Xin_Src_Line_No              /* line no within file         */
          fixed bin(32) unsigned,  /*                             */
                                   /*                             */
    2 Xin_Src_Id                   /* id for this source record   */
          fixed bin(32) unsigned,  /*                             */
                                   /*                             */
```

```
    2 Xin_Src_Length               /* length of text               */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Src_Text                 /* actual text                  */
        char( 137 refer(xin_Src_Length) );
```

トークン・レコード

各トークン・レコードでは、トークン索引と呼ばれる 1 つの番号が割り当てられます。後のレコードで PL/I コンパイラーに認識されるトークンを参照するときは、この番号が使用されます。 また、このレコードは、トークンのタイプに加えて、トークンの開始と終了それぞれの列と行を示します。

トークン・レコードを宣言する：

```
Dcl
  1 Xin_Tok     based,             /* token record                 */
                                   /*                              */
    2 Xin_Tok_Hdr                  /* standard header              */
           like Xin_Hdr,           /*                              */
                                   /*                              */
    2 Xin_Tok_Inx                  /* adata index for token        */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_Begin_Line           /* starting line no within file */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_End_Line_Offset      /* offset of end line from first */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_Kind_Value           /* token kind                   */
           ordinal xin_Tok_Kind,   /*                              */
                                   /*                              */
    2 Xin_Tok_Rsrvd                /* reserved                     */
           fixed bin(8) unsigned,  /*                              */
                                   /*                              */
    2 Xin_Tok_Begin_Col            /* starting column              */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_End_Col              /* ending column                */
           fixed bin(16) unsigned; /*                              */
```

序数（ordinal） xin_Tok_Kind は、トークン・レコードのタイプを識別する。

トークン・レコードの種類を宣言する：

```
 Define
  ordinal
    xin_Tok_Kind
      (  xin_Tok_Kind_Unset
        ,xin_Tok_Kind_Lexeme
        ,xin_Tok_Kind_Comment
        ,xin_Tok_Kind_Literal
        ,xin_Tok_Kind_Identifier
        ,xin_Tok_Kind_Keyword
                                   )  prec(8) unsigned;
```

構文レコード

各構文レコードでは、ノード ID と呼ばれる 1 つの番号が割り当てられます。後のレコードで他の構文レコードを参照するときは、この番号が使用されます。 最初の構文レコードの種類は xin_Syn_Kind_Package で、コンパイル・ユニットにプロシージャーがあれば、このレコードの子ノードはこれらのプロシージャーの先頭を指す。 次に親ノード、兄弟ノード、および子ノードに基づいて、コンパイル単位に含まれるすべてのプロシージャーと開始ブロックの特有の関係が 1 つのマップに設定されます。

次の単純なプログラムで考えてみます。

```
       a: proc;
         call b;
         call c;
         b: proc;
         end b;
         c: proc;
           call d;
```

```
          d: proc;
          end d;
        end c;
      end a;
```

このプログラムのブロックにノード索引が次のように割り当てられます。

```
 symbol  index     sibling      parent    child
 ------  -----     -------      ------    -----
   -       1         0            0         2
   a       2         0            1         3
   b       3         4            2         0
   c       4         0            2         5
   d       5         0            4         0
```

シンタックスレコードを宣言する：

```
Dcl
  1 Xin_Syn     based,             /* syntax record                */
                                   /*                              */
    2 Xin_Syn_Hdr                  /* standard header              */
          like Xin_Hdr,            /*                              */
                                   /*                              */
    2 Xin_Syn_Node_Id              /* node id                      */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Node_Kind            /* node type                    */
          ordinal xin_syn_kind,    /*                              */
                                   /*                              */
    2 Xin_Syn_Node_Exp_Kind        /* node sub type                */
          ordinal xin_exp_kind,    /*                              */
                                   /*                              */
    2 *                            /* reserved                     */
          fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Parent_Node_Id       /* node id of parent            */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Sibling_Node_Id      /* node id of sibling           */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Child_Node_Id        /* node id of child             */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 xin_Syn_First_Tok            /* id of first spanned token    */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 xin_Syn_Last_Tok             /* id of last spanned token     */
          fixed bin(32) unsigned, /*                              */
```

```
                                   /*                               */
    2 * union,                     /* qualifier for node            */
                                   /*                               */
      3 Xin_Syn_Int_Value          /* used if int                   */
           fixed bin(31),          /*                               */
                                   /*                               */
      3 Xin_Syn_Literal_Id         /* used if name, number, picture */
           fixed bin(31),          /*                               */
                                   /*                               */
      3 Xin_Syn_Node_Lex           /* used if lexeme, assignment,   */
           ordinal xin_Lex_kind,   /* infix_op, prefix_op           */
                                   /*                               */
      3 Xin_Syn_Node_Voc           /* used if keyword, end_for_do   */
           ordinal xin_Voc_kind,   /*                               */
                                   /*                               */
      3 Xin_Syn_Block_Node         /* used if call_begin            */
           fixed bin(31),          /*   to hold node of begin block */
                                   /*                               */
      3 Xin_Syn_Bif_Id             /* used if bif_rfrnc             */
           fixed bin(32) unsigned, /*                               */
                                   /*                               */
      3 Xin_Syn_Sym_Id             /* used if label, unsub_rfrnc,   */
           fixed bin(32) unsigned, /* subscripted_rfrnc             */
                                   /*                               */
      3 Xin_Syn_Proc_Data,         /* used if package, proc or begin*/
                                   /*                               */
        4 Xin_Syn_First_Sym        /*   id of first contained sym   */
           fixed bin(32) unsigned, /*                               */
```

```
                                /*                                */
       4 Xin_Syn_Block_Sym      /*   id of sym for this block     */
          fixed bin(32) unsigned, /*                               */
```

Declare for a syntax record (continued)

```
                                  /*                               */
     3 Xin_Syn_Number_Data,       /* used if number                */
                                  /*                               */
       4 Xin_Syn_Number_Id        /*   id of literal               */
          fixed bin(32) unsigned, /*                               */
                                  /*                               */
       4 Xin_Syn_Number_Type      /*   type                        */
          ordinal xin_Number_Kind,/*                               */
                                  /*                               */
       4 Xin_Syn_Number_Prec      /*   precision                   */
          fixed bin(8) unsigned,  /*                               */
                                  /*                               */
       4 Xin_Syn_Number_Scale     /*   scale factor                */
          fixed bin(7) signed,    /*                               */
                                  /*                               */
       4 Xin_Syn_Number_Bytes     /*   bytes it would occupy       */
          fixed bin(8) unsigned,  /*   in its internal form        */
                                  /*                               */
     3 Xin_Syn_String_Data,       /* used if char_string,          */
                                  /* bit_string, graphic_string    */
                                  /*                               */
       4 Xin_Syn_String_Id        /*   id of literal               */
          fixed bin(32) unsigned, /*                               */
                                  /*                               */
       4 Xin_Syn_String_Len       /*   string length in its units  */
          fixed bin(32) unsigned, /*                               */
                                  /*                               */
     3 Xin_Syn_Stmt_Data,         /* used if stmt                  */
                                  /*                               */
       4 Xin_Syn_File_Id          /*   file id                     */
          fixed bin(32) unsigned, /*                               */
                                  /*                               */
       4 Xin_Syn_Line_No          /*   line no within file         */
          fixed bin(32) unsigned, /*                               */
                                  /*                               */
    2 *               char(0);    /*                               */
```

Declare for a syntax record (continued)

序数 xin_Syn_Kind 、シンタックス・レコードのタイプを識別する：

```
Define
  ordinal
    xin_Syn_Kind
      (  xin_Syn_Kind_Unset
       ,xin_Syn_Kind_Lexeme
       ,xin_Syn_Kind_Asterisk
       ,xin_Syn_Kind_Int
       ,xin_Syn_Kind_Name
       ,xin_Syn_Kind_Expression
       ,xin_Syn_Kind_Parenthesized_Expr
       ,xin_Syn_Kind_Argument_List
       ,xin_Syn_Kind_Keyword
       ,xin_Syn_Kind_Proc_Stmt
       ,xin_Syn_Kind_Begin_Stmt
       ,xin_Syn_Kind_Stmt
       ,xin_Syn_Kind_Substmt
       ,xin_Syn_Kind_Label
       ,xin_Syn_Kind_Invoke_Begin
       ,xin_Syn_Kind_Assignment
       ,xin_Syn_Kind_Assignment_Byname
       ,xin_Syn_Kind_Do_Fragment
       ,xin_Syn_Kind_Keyed_List
       ,xin_Syn_Kind_Iteration_Factor
       ,xin_Syn_Kind_If_Clause
       ,xin_Syn_Kind_Else_Clause
       ,xin_Syn_Kind_Do_Stmt
       ,xin_Syn_Kind_Select_Stmt
       ,xin_Syn_Kind_When_Stmt
       ,xin_Syn_Kind_Otherwise_Stmt
       ,xin_Syn_Kind_Procedure
       ,xin_Syn_Kind_Package
       ,xin_Syn_Kind_Begin_Block
```

```
        ,xin_Syn_Kind_Picture
        ,xin_Syn_Kind_Raw_Rfrnc
        ,xin_Syn_Kind_Generic_Desc
                                  )  prec(8) unsigned;
```

次の単純なプログラムで考えてみます。

```
       a: proc(x);
         dcl x char(8);
         x = substr(datetime(),1,8);
       end;
```

このプログラムのブロックにノード索引が次のように割り当てられます。

```
node_kind      index     sibling      parent    child
-----------    -----     -------      ------    -----
 package         1          0           0         2
 procedure       2          0           1         0
 expression      3          0           0         0
 stmt            4          5           2         6
 stmt            5         10           2        11
 label           6          7           4         0
 keyword         7          8           4         0
 expression      8          9           4         0
 lexeme          9          0           4         0
 stmt           10          0           2        18
 assignment     11         12           5        13
 lexeme         12          0           5         0
 expression     13         14          11         0
 expression     14          0          11        15
 expression     15         16          14         0
 expression     16         17          14         0
 expression     17          0          14         0
 keyword        18         19          10         0
 lexeme         19          0          10         0
```

プロシージャー・レコードには、ENTRY A のシンボル・レコードの ID が含まれます (block_sym フィールド内)。このシンボル・レコードには、そのプロシージャーの最初のステートメントのノード ID が含まれます (first_stmt_id フィールド内)。

ステートメント・レコードの場合、兄弟ノード ID は次のステートメント・レコードを指します (次のステートメント・レコードが存在する場合)。 子ノード ID は、そのステートメント・レコードの最初のエレメントを指します。

PROCEDURE 、4 つのレコードから構成される：

- ラベル・レコード。
- キーワードレコード（ PROCEDURE キーワード用）
- 式レコード (パラメーター X 用)。これには、式の種類 unsub_rfrnc とシンボル X の sym_id が含まれます。
- 字句レコード (セミコロン用)。

割り当てステートメントのレコードは、次の 2 つのレコードから成ります。

- 次の 2 つの子を持つ割り当てレコード。
  - (ターゲット X の)式レコード。式の種類は unsub_rfrnc 、シンボル X の sym_id
  - ソースの）式レコードで、式の種類は builtin_rfrnc 、シンボル SUBSTR に対する sym_id ：
    - 式レコード (最初の引数用)。これには、式の種類 builtin_rfrnc とシンボル DATETIME の sym_id が含まれます。
    - 式レコード (2 番目の引数用)。これには、式の種類 number と値 1 の literal_id が含まれます。
    - 式レコード (3 番目の引数用)。これには、式の種類 number と値 8 の literal_id が含まれます。
- 字句レコード (セミコロン用)。

END 、2 つのレコードから構成される：

- キーワードレコード（ END キーワード用）

- 字句レコード (セミコロン用)。

序数（ordinal） xin_Exp_Kind は、式を記述する構文レコードの式のタイプを識別する。 このレコードの中には、子ノードの数がゼロでないものもあります。例えば、次のような場合です。

- 挿入演算子 (減算を表すマイナスなど) は、その左オペランドが記述された子ノードを持ちます (このオペランドの兄弟ノードに、右演算子が記述されます)。
- 接頭演算子 (否定を表すマイナスなど) は、そのオペランドが記述された子ノードを持ちます。

表現の種類を宣言する：

```
Define
 ordinal
   xin_Exp_Kind
     (  xin_Exp_Kind_Unset
       ,xin_Exp_Kind_Bit_String
       ,xin_Exp_Kind_Char_String
       ,xin_Exp_Kind_Graphic_String
       ,xin_Exp_Kind_Number
       ,xin_Exp_Kind_Infix_Op
       ,xin_Exp_Kind_Prefix_Op
       ,xin_Exp_Kind_Builtin_Rfrnc
       ,xin_Exp_Kind_Entry_Rfrnc
       ,xin_Exp_Kind_Qualified_Rfrnc
       ,xin_Exp_Kind_Unsub_Rfrnc
       ,xin_Exp_Kind_Subscripted_Rfrnc
       ,xin_Exp_Kind_Type_Func
       ,xin_Exp_Kind_Widechar_String
                                  )  prec(8) unsigned;
```

序数（ordinal） xin_Number_Kind は、数値を記述する構文レコードの数値のタイプを識別する。

数字の種類を宣言する：

```
Define
 ordinal
   xin_Number_Kind
     (  xin_Number_Kind_Unset
       ,xin_Number_Kind_Real_Fixed_Bin
       ,xin_Number_Kind_Real_Fixed_Dec
       ,xin_Number_Kind_Real_Float_Bin
       ,xin_Number_Kind_Real_Float_Dec
       ,xin_Number_Kind_Cplx_Fixed_Bin
       ,xin_Number_Kind_Cplx_Fixed_Dec
       ,xin_Number_Kind_Cplx_Float_Bin
       ,xin_Number_Kind_Cplx_Float_Dec
                                  )  prec(8) unsigned;
```

序数 xin_Lex_Kind は、語彙ユニットを記述する構文レコードの語彙のタイプを識別する。

- これらの序数名にある「vrule」は「垂直罫線」(「or」記号などとして使用される) を意味します。
- これらの序数名にある「dbl」は「二重」を意味します。 したがって dbl_Vrule は、「連結」記号などとして使用される二重垂直罫線です。

字句の種類を宣言する：

```
Define
 ordinal
   xin_Lex_Kind
     (  xin_Lex_Undefined
       ,xin_Lex_Period
       ,xin_Lex_Colon
       ,xin_Lex_Semicolon
       ,xin_Lex_Lparen
       ,xin_Lex_Rparen
       ,xin_Lex_Comma
       ,xin_Lex_Equals
       ,xin_Lex_Gt
       ,xin_Lex_Ge
       ,xin_Lex_Lt
       ,xin_Lex_Le
       ,xin_Lex_Ne
       ,xin_Lex_Lctr
       ,xin_Lex_Star
       ,xin_Lex_Dbl_Colon
```

```
        ,xin_Lex_Not
        ,xin_Lex_Vrule
        ,xin_Lex_Dbl_Vrule
        ,xin_Lex_And
        ,xin_Lex_Dbl_Star
        ,xin_Lex_Plus
        ,xin_Lex_Minus
        ,xin_Lex_Slash
        ,xin_Lex_Equals_Gt
        ,xin_Lex_Lparen_Colon
        ,xin_Lex_Colon_Rparen
        ,xin_Lex_Plus_Equals
        ,xin_Lex_Minus_Equals
        ,xin_Lex_Star_Equals
        ,xin_Lex_Slash_Equals
        ,xin_Lex_Vrule_Equals
        ,xin_Lex_And_Equals
        ,xin_Lex_Dbl_Star_Equals
        ,xin_Lex_Dbl_Vrule_Equals
        ,xin_Lex_Dbl_Slash
      ) unsigned prec(16);
```

序数（ordinal） xin_Voc_Kind は、コンパイラの「語彙（vocabulary）」の項目を記述する構文レコードのキーワードを特定する。

voc kind の宣言は、サンプルデータセット SIBMZSAM のメンバ `ibmwxin` にある。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。 IBM は、本書に記載の製品、サービス、または機能を日本においては提供していない場合があります。 日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。 本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。 これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。 ただし、IBM 以外の製品、プログラム、またはサービスの操作の評価および検証は、お客様の責任で行っていただきます。&

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。 本書の提供は、お客様にこれらの特許権について 実施権を許諾することを意味するものではありません。 実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

IBM Corporation
J74/G4
IBM World Trade Asia Corporation
法務・知的財産
U.S.A.

2 バイト (DBCS) 情報に関するライセンス照会については、国内の IBM 知的財産部門に連絡するか、書面でお問い合わせをお送りください。

〒 103-8510
Legal and Intellectual Property Law
IBM 日本国株式会社
106-8711 東京都港区六本木 3-2-12

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

インターナショナル・ビジネス・マシンズ・コーポレーション が提供
本書は、特定物として現存するままの状態で提供され、
明示または黙示を問わず、以下のものを含みますが、これに限定されるものではありません。
を、非侵害の黙示的保証に変更することができます。
「商品性」または「特定目的適合性」

国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。

本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものでは ありません。 それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

商標

IBM、IBM ロゴ、および ibm.com は、世界の多くの国で登録された International Business Machines Corporation の商標です。 世界中の多くの国で登録されています。 他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。 現時点での IBM の商標リストについては、"著作権および商標情報" (www.ibm.com/legal/copytrade) をご覧ください。

Intel および Pentium は、Intel Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

© Copyright IBM Corp. 1999, 2025

Microsoft、Windows、および Windows NT は、Microsoft Corporation の米国およびその他の国における商標です。

Pentium は Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

:NONE.

UNIX は The Open Group の米国およびその他の国における登録商標です。

Other company, product or service names may be the trademarks or service marks of others.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は 表示されない場合があります。

参考文献

PL/I 資料

Enterprise PL/I for z/OS

プログラミング・ガイド, GI13-5620
言語リファレンス, SC31-5716
メッセージとコード, GC31-5717
コンパイラーおよびランタイムマイグレーション・ガイド, GC31-5715

PL/I (MVS とは & VM 版)

「*PL/I MVS&VM VA PL/I MEL* インストールとカストマイズ」、SC88-7221
「言語解説書」、SC88-7219
「コンパイル時メッセージおよびコード」、SC88-7224
「診断の手引き」、SC88-7223
「移行の手引き」、SC88-7220
「プログラミング・ガイド」、SC88-7218
「参照要約」、SX88-7011

Enterprise PL/I for AIX®

「プログラミング・ガイド」、SA88-4427
「言語解説書」、SA88-4429
「メッセージおよびコード」、GA88-4430
「インストール・ガイド」、GA88-4428

関連資料

Db2 for z/OS

「管理ガイド」、SC43-3430
「アプリケーション・プログラミングおよび *SQL* 解説書」、SC43-3444
「コマンド解説書」、SC43-3431
「メッセージ」、GC43-3438
「コード」、GC43-3442
「*SQL* 解説書」、SC43-3449
「*LOBs with Db2 for z/OS: Stronger and Faster*」、SG24-7270
Db2 for z/OS Product Documentation も参照してください

DFSORT

「アプリケーション・プログラミング・ガイド」、SA88-5607
「インストールおよびカスタマイズ」、SA88-7075

IMS/ESA®

「アプリケーション・プログラミング: データベース管理プログラム」、SC88-7552
「*Application Programming: Database Manager Summary*」、SC26-8037
「アプリケーション・プログラミング: 設計の手引き」、SC88-7542

© Copyright IBM Corp. 1999, 2025

「アプリケーション・プログラミング: トランザクション管理プログラム」、SC88-7553
「*Application Programming: Transaction Manager Summary*」、SC26-8038
「アプリケーション・プログラミング: *EXEC DL/I* コマンド *(CICS* および *IMS™)*」、SC88-7554
「*Application Programming: EXEC DL/I Commands for CICS and IMS Summary*」、SC26-8036
「*IMS/ESA V6R1 Bookindex*」、GC27-1557

TXSeries® for Multiplatforms

「*Encina* 管理の手引き 第 *2* 巻：サーバー管理」、SD88-7403
「*Encina SFS Programming Guide*」、SC09-4483
TXSeries for Multiplatforms Knowledge Center を参照してください

z/Architecture

「解説書」、SA88-8773
オンラインの *Principles of Operation* を参照してください

z/OS 言語環境プログラム

「概念」、SC43-2659
「デバッグ・ガイド」、GA88-5524
「ランタイム・メッセージ」、SA88-7094
「カスタマイズ」、SA88-7096
「プログラミング・ガイド」、SA88-7098
「*64* ビット仮想アドレッシング・モード向けプログラミング・ガイド」、SA88-7101
「プログラミング・リファレンス」、SA88-7099
「ランタイム・アプリケーション マイグレーション・ガイド」、GC43-2660
「*Vendor Interfaces*」、SA38-0688
「言語間通信 アプリケーションの作成」、SC43-2661
z/OS Language Environment Knowledge Center も参照してください

z/OS MVS

「*JCL* 解説書」、SA88-7091
「*JCL* ユーザーズ・ガイド」、SA88-5468
「システム・コマンド」、SA88-5490
z/OS MVS Knowledge Center を参照してください

z/OS TSO/E

「コマンド解説書」、SA88-7049
「ユーザーズ・ガイド」、SA88-5454

z/OS UNIX システム・サービス

「*z/OS UNIX* システム・サービス コマンド解説書」、SA88-5452
「*z/OS UNIX* システム・サービス プログラミング: アセンブラー呼び出し可能サービス 解説書」、SA88-7054
「*z/OS UNIX* システム・サービス ユーザーズ・ガイド」、SA88-7053

Unicode および文字表現

「*z/OS Unicode* サポート: 変換サービスの使用法」、SD88-6163
「*z/OS Unicode* サービス ユーザーズ・ガイドおよびリファレンス」、SA88-7093

用語集

この用語集では、PL/I のすべてのプラットフォームおよびリリースについて用語を定義します。 本書では使用しない用語が含まれている場合があります。 該当する用語が見つからない場合は、本書の索引を調べるか、「*IBM Dictionary of Computing*」(SC20-1699) を参照してください。

A

アクセス
データを参照するかまたは取り出すこと。

処置指定 (action specification)
ON ステートメント内にある、ON ユニットまたは単一のキーワード SYSTEM。 該当する条件が発生すれば、2 つのうちいずれかがとるべき処置を指定する。

活動化 (ブロックの) (activate (a block))
ブロックの実行を開始すること。 プロシージャー・ブロックは、呼び出されるときに、活動化される。 開始ブロックが活動化するのは、分岐を含め、通常の制御の流れ内に現れたと きである。 パッケージを活動化することはできない。

活動化 (プリプロセッサー変数またはプリプロセッサー・エントリー・ポイントの) (activate (a preprocessor variable or preprocessor entry point))
マクロ機能 ID を、それに後続するソース・コード内で置換 可能にすること。 %ACTIVATE ステートメントは、プリプロセッサー変数や プリプロセッサー・エントリー・ポイントを活動化する。

アクティブ
活動化から終了にいたるまでのブロックの状態。 ソース・プログラム・テキスト中の対応する ID をプリプロセッサー変数や プリプロセッサー入り口名の値に置き換えることができるときの、 その変数や入り口の状態。 イベント変数が非同期操作に関連付けられている間に置かれている状態。 タスク変数に関連するタスクが付加されるときにタスク変数が 置かれている状態。 タスクが終了する前に置かれている状態。

実際の起点 (actual origin (AO))
配列または構造体内の最初の項目の位置。

追加属性 (additive attribute)
デフォルトがないファイル記述属性。必要に応じて、明示的に指定された別の属性によって明示的または暗黙的に指定する必要があります。 代替属性と対比。

調節可能エクステント (adjustable extent)
関連変数の世代によって異なることのある境界 (配列の)、 長さ (ストリングの)、またはサイズ (区域の)。 調節可能エクステントは、世代ごとに別々に評価される式またはアスタリスク (ただし、 基底付き変数の場合は REFER オプション) で指定される。 静的変数に使用することはできない。

集合 (aggregate)
「データ集合 *(data aggregate)*」を参照。

集合式 (aggregate expression)
配列式、構造式、または共用体式のこと。

集合タイプ (aggregate type)
どのデータ項目の場合も、それが構造体、共用体、または配列の いずれであるかの指定。

割り振られた変数 (allocated variable)
主記憶域が関連付けられ解放されない変数。

割り振り
変数用の主記憶域の予約。 割り振られた変数の世代。 PL/I ファイルを、システム・データ・セット、装置、またはファイルに 関連付けること。

位置合わせ
機械に依存する特定の境界 (例えば、フルワード境界 またはハーフワード境界) に関連付けて、データ項目を保管すること。

© Copyright IBM Corp. 1999, 2025

英字 (alphabetic character)
A から Z までの任意の英字と、#、$、@ (これらのグラフィック表記は国によって異なる 場合がある) の拡張英字。

英数字 (alphameric character)
英字または数字。

代替属性 (alternative attribute)
属性グループから選択するファイル記述属性。 何も指定しないと、デフォルトがとられる。 「追加属性 *(additive attribute)*」と対比。

あいまい参照 (ambiguous reference)
参照時点で認識されている名前をただ 1 つだけ識別するためには 修飾が不十分な参照。

サイト
基底付き変数を割り振ることのできる、ストレージ中の部分。

argument
サブルーチンまたは機能の呼び出しの一部である引数リスト内 にある式。

引数リスト (argument list)
コンマで区切られ、入り口名定数、入り口名変数、総称名、または組み込み関数名に 続く、括弧で囲まれたゼロまたはそれ以上の引数のリスト。 そのリストは、エントリー・ポイントのパラメーター・リストである。

算術比較 (arithmetic comparison)
数値の比較。 「ビット比較 *(bit comparison)*」、「文字比較 *(character comparison)*」も参照。

算術定数 (arithmetic constant)
固定小数点定数または浮動小数点定数。 大部分の算術定数には符号を付けることができるが、符号は定数の 一部ではない。

算術変換 (arithmetic conversion)
ある 1 つの算術表現から別の表現に値を変換すること。

算術データ
基数、スケール、モード、および精度の特性を持つデータ。 コード化算術データとピクチャー数字データも含まれる。

算術演算子
接頭演算子の + と -、あるいは挿入演算子 + - * / ** のうちのいずれか。

アレイ
同じ属性を持ち、1 つ以上の次元別にグループ分けされた 1 つ以上のデータ・エレメントに、名前を付けて順番に並べた集合体。

配列式 (array expression)
評価されると値の配列が生成される式。

構造体の配列 (array of structures)
次元属性を構造体名に与えて指定される、順番に並べられた同一構造体の 集まり。

配列変数 (array variable)
同じ属性を持っていなければならないデータ項目の 集合を表す変数。 「構造変数 *(structure variable)*」と対比。

ASCII
情報交換用米国標準コード (American National Standard Code for Information Interchange)。

割り当て
値を変数に与える処理。

非同期操作 (asynchronous operation)
ステートメントの実行と、入出力操作が並行して行われること。 各種タスクに複数の制御の流れを使った、プロシージャーの 並行実行。

タスクの生成 (attachment of a task)
呼び込まれたプロシージャー (およびこれが呼び出す プロシージャー) を、呼び出しプロシージャーの実行と一緒に、非同期で 実行するために、プロシージャーを呼び出して別に制御の流れを確立 すること。

アテンション (attention)
タスクに割り込みが生じる原因となるような、タスクにとっては外部の事柄の発生。

属性
表明された特性を記述するのに名前と関連付けた記述特性。 式の計算の結果の特性を説明するために用いられる記述特性。

自動ストレージ割り振り (automatic storage allocation)
自動変数用のストレージ割り振り。

自動変数 (automatic variable)
ブロックの起動時に自動的にストレージを割り振られ、そのブロックの 終了時に自動的にそれを解除される変数。

B

基数
算術値を表現するための数体系。

基本エレメント (base element)
それ自身は別の構造体や共用体ではない、構造体や共用体のメンバー。

基本項目 (base item)
定義変数を定義するための、自動、被制御、または静的変数、 またはパラメーター。

基底付き参照 (based reference)
基底付きストレージ・クラスを持った参照。

基底付きストレージ割り振り (based storage allocation)
基底付き変数用のストレージの割り振り。

基底付き変数 (based variable)
ストレージ・アドレスがロケーターによって与えられる変数。 同一変数の複数の世代をアクセスすることができる。 これは、ストレージ内の固定位置を識別しない。

開始ブロック (begin-block)
BEGIN ステートメントと END ステートメントによって区切られ、名前有効範囲を形成するステートメントの集まり。 開始ブロックの活動化は、条件が生じたために行われる (開始ブロックが ON ユニットへの処置指定である場合) か、または GOTO ステートメントの 結果の分岐を含め、通常の制御の流れを介して行われる。

2 進数 (binary)
0 と 1 が唯一の数表示である数体系。

2 進数字 (binary digit)
「ビット *(bit)*」を参照。

2 進固定小数点値 (binary fixed-point value)
2 進数字で構成され、オプションの 2 進小数点とオプションの符号を持った整数。 「*10* 進固定小数点値 *(decimal fixed-point value)*」と対比。

2 進浮動小数点値 (binary floating-point value)
2 進小数部と見なすことができる仮数と、2 の基数に対する整数指数と見なすことができる指数の形式での実数の近似値。 *10* 進浮動小数値と対比。

ビット
0 または 1 です。 コンピューター・ストレージの最小スペース量。

ビット比較 (bit comparison)
2 進数字を、左から右へビットごとに比較すること。 「算術比較 *(arithmetic comparison)*」、「文字比較 *(character comparison)*」も参照。

ビット・ストリング定数 (bit string constant)
一連の 2 進数字で囲まれ、直後に接尾部 B が付きます。 文字定数と対比。 単一引用符に囲まれ、後に接尾部 B4 が付いた 16 進数字の連なり。

ビット・ストリング (bit string)
ゼロ以上のビットで構成されたストリング。

ビット・ストリング演算子 (bit string operators)
論理演算子 NOT と排他 OR (¬)、 AND (&)、および OR (|)。

ビット値 (bit value)
ビット・タイプを表す値。

ブロック
その中で宣言された名前の有効範囲と、その名前用のストレージ割り振りを 指定する、1 つの単位として処理される一連のステートメント。 ブロックとしては、パッケージ、プロシージャー、または開始ブロックの いずれもありえる。

境界 (bounds)
任意の配列次元の上限と下限。

区切り文字 (break character)
下線記号 (_)。 これは、ID の読みやすさを向上させるために使用できます。 例えば、変数を OLDINVENTORYTOTAL とする代わりに OLD_INVENTORY_TOTAL と記述できる。

組み込み関数 (built in function)
SQRT (平方根) のような、言語が提供する定義済み関数。

組み込み関数参照 (built-in function reference)
オプションの引数リストを持つ組み込み関数名。

組み込み名 (built-in name)
組み込みサブルーチンの入り口名。

組み込みサブルーチン (built-in subroutine)
コンパイル時に定義され、CALL ステートメントによって呼び出される 入り口名を持つサブルーチン。

バッファー
レコードが入力時に読み込まれ、レコードが出力時に書き出される、 入出力操作に使用する中間記憶域。

C

呼び出し
CALL ステートメントまたは CALL オプションを使用してサブルーチンを 呼び出すこと。

文字比較 (character comparison)
照合順序に従って、左から右へ文字単位で行われる比較。「算術比較 *(arithmetic comparison)*」、「ビット比較 *(bit comparison)*」も参照。

文字ストリング定数 (character string constant)
単一引用符で囲まれた一連の文字。例えば、'Shakespeare''s 'Hamlet:'' など。

文字セット (character set)
あらかじめ決められた文字の集まり。 *ASCII* および *EBCDIC* も参照してください。

文字ストリング・ピクチャー・データ (character string picture data)
文字値だけを持ったピクチャー・データ。 このタイプのピクチャー・データは、少なくとも 1 つの A または X ピクチャー指定文字を持っていなければならない。 「数値ピクチャー・データ *(numeric picture data)*」と対比。

クローズ (ファイルの) (closing (of a file))
ファイルをデータ・セットまたは装置と切り離すこと。

コード化算術データ
数値を表し、基数 (10 進数または 2 進数)、スケール (固定小数点または浮動小数点)、および精度 (個々に持ちうる桁数) を特徴とするデータ項目。 このデータは、変換しなくても、算術計算用に受け入れることのできる 形式で保管される。

複合ネストの深さ (combined nesting depth)
ネストの最も深いレベル。手順/開始/開始、実行、選択、および IF のレベルをカウントすることにより決定されます。THEN...プログラム内に ELSE ネストがあります。

コメント
文書化のために使用され、/* および */ で区切られる、ゼロ以上 の文字数の文字ストリング。

商用文字 (commercial character)

- CR (貸方) ピクチャー指定文字。
- DB (借方) ピクチャー指定文字。

比較演算子 (comparison operator)
関係内の項目を相互に比較するよう指示するための算術、ストリング・ロケーター、 または論理関係で使用される演算子。 比較演算子は次のとおり。

= (に等しい)
> (より大)
< (より小)
>= (より大か等しい)
<= (より小か等しい)
¬= (等しくない)
¬> (より大ではない)
¬< (より小ではない)

コンパイル時 (間) (compile time)
一般に、ソース・プログラムがオブジェクト・モジュールに 変換されている時間。 PL/I では、変更したい場合に、ソース・プログラムを変更して、 オブジェクト・プログラムに変換し終わるまでに経過する時間。

コンパイラー・オプション
コンパイルの特定の面を制御するために指定されるキーワード。例えば、 生成するオブジェクト・モジュールの特徴や、作成する印刷出力のタイプなどがある。

複素数データ (complex data)
おのおのの項目が実数部と虚数部で構成された算術データ。

合成演算子 (composite operator)
<=、**、および /* などの特殊文字を複数含む演算子。

複合ステートメント (compound statement)
他のステートメントが含まれているステートメント。 PL/I では、IF、ON、OTHERWISE、および WHEN だけが、複合ステートメントである。 「ステートメント本体 *(statement body)*」を参照。

連結
2 つのストリングを指定順に結合し、元の 2 つのストリング の合計長に等しい長さを持った 1 つのストリングを作成する操作。 これは、演算子 || で指定する。

条件
エラー (オーバーフローなど) または予期される状況 (入力ファイルの 終わりなど) のいずれかの例外的な状態。 条件が発生する (検出される) と、その条件に対する規定のアクションが処理される。 「確立された処置 *(established action)*」および「暗黙処置 *(implicit action)*」も参照。

条件名 (condition name)
PL/I 定義またはプログラマー定義の条件の名前。

条件接頭語 (condition prefix)
ステートメントの接頭部として付けられる、括弧で囲まれた 1 つ以上の条件名のリスト。 条件接頭語は、指定した条件を使用可能にするか使用不能にするかを指定する。

連結集合 (connected aggregate)
エレメントが、間にデータ項目の入らない連続したストレージを占有する配列または構造体。 「非連結集合 *(nonconnected aggregate)*」と対比。

連結参照 (connected reference)
連結ストレージに対する参照。 プログラムを実行するには、ストレージが連結されていることが明らかでなければならない。

連結ストレージ (connected storage)
単一名を使って参照することができる諸項目の非中断かつ線形の 連なりの主記憶域。

定数
名前が付いておらず、変更できない値を持った、算術または ストリング・データ項目。 VALUE 属性を指定して宣言された ID。 FILE 属性または ENTRY 属性を指定し、VARIABLE 属性を指定しないで宣言された ID。

定数参照 (constant reference)
対象として定数を持つ値参照。

含まれているブロック、宣言、またはソース・テキスト (contained block, declaration, or source text)
開始、プロシージャー、またはパッケージのブロック内の すべてのブロック、プロシージャー、ステートメント、宣言、または ソース・テキスト。 パッケージ、プロシージャー、および BEGIN ステートメントと それに対応する END ステートメント全体は、ブロック内には含まれて いない。

収容ブロック (containing block)
該当する宣言、ステートメント、プロシージャー、またはその他の ソース・テキストを収容している、パッケージ、プロシージャー、または開始ブロック。

コンテキスト宣言 (contextual declaration)
DECLARE ステートメントで明示的に宣言されていないが、その使用の前後関係から、特定の属性が ID に関連付けられるような ID の存在。

制御文字 (control character)
特定コンテキスト内に存在することによって 制御機能が指定される、文字セット内の文字。 1 つの例としてファイルの終わり (EOF) マーカーがある。

制御フォーマット項目 (control format item)
ストリーム内または印刷ページの内での、あるデータ項目の位置付けを 指定するために、編集指示伝送の中で使用される指定。

制御変数 (control variable)
DO ステートメントの反復実行を制御するのに使用する変数。

被制御パラメーター (controlled parameter)
DECLARE ステートメント内で CONTROLLED 属性を指定される パラメーター。 これは、CONTROLLED 属性を持った引数としか関連付けることはできない。

被制御ストレージ割り振り (controlled storage allocation)
被制御変数用のストレージの割り振り。

被制御変数 (controlled variable)
現行世代にだけアクセスすることができ、ALLOCATE と FREE ステートメントによって割り振りと解放が制御される変数。

制御セクション (control sections)
オブジェクト・モジュール内のグループ化された機械命令。

変換
ある 1 つの表現法から、一組の特定属性に合うよう別の表現法に値を変換すること。 例えば、文字ストリングを FIXED BINARY (15,0) などの算術値に 変換すること。

配列のクロス・セクション (cross section of an array)
配列の少なくとも 1 つの次元のエクステントで表すことの できるエレメント。 配列参照内に添え字の代わりにアスタリスクがあれば、それはその次元 のエクステント全体を表す。

現行世代 (current generation)
変数名を参照して、現在使用できる自動変数または被制御変数の世代。

D

データ
処理に適合した形式の情報または値の表現。

データ集合 (data aggregate)
異なったデータ項目の集まりであるデータ項目。

データ属性 (data attribute)
FIXED BINARY などの、データ項目が表すデータのタイプを指定するキーワード。

データ・ディレクティブ伝送 (data-directed transmission)
データを伝送するための、ストリーム指向伝送のタイプ。 これは代入ステートメントに似ており、`name = constant` という形式です。

データ項目
単一の名前付きデータ単位。

データ・リスト (data list)
ストリーム指向伝送における、GET および PUT ステートメント 内で使用するデータ項目を括弧で囲んだリスト。 「フォーマット・リスト *(format list)*」と対比。

データ・セット
単一のファイル名の参照によってアクセスすることができる、 プログラムの外部にあるデータの集まり。 参照されることが可能な装置。

データ指定 (data specification)
伝送モード (DATA、LIST、または EDIT) を指示し、さらに データ・リストと、編集ディレクティブ・モードの場合はフォーマット・リストを含む、 ストリーム指向伝送ステートメントの一部分。

データ・ストリーム (data stream)
ストリーム指向伝送でデータ・セットから、またはデータ・セット へ、文字形式のデータ・エレメントの連続ストリームとして転送されるデータ。

データ伝送
データ・セットからプログラムへ、およびその逆に、データを 転送すること。

データ・タイプ (data type)
一連のデータ属性。

DBCS
文字セットにおいて、それぞれの文字は 2 つの連続するバイトで表される。

非アクティブ (deactivated)
ある ID の値で、ソース・プログラム・テキスト内のプリプロセッサー ID を置き換えることができない状態。 「アクティブ *(active)*」と対比。

デバッグ (debugging)
プログラムからバグを除去する処理。

10 進
0 から 9 までの数字を使った数体系。

10 進ピクチャー文字 (decimal digit picture character)
ピクチャー指定文字 9 のこと。

10 進固定小数点定数 (decimal fixed-point constant)
1 つ以上の 10 進数 (および任意で小数点を付けたもの) から成る定数。

10 進固定小数点値 (decimal fixed-point value)
小数点の想定位置を持つ 10 進数の連なりで構成される有理数。「*2* 進固定小数点値 *(binary fixed-point value)*」と対比。

10 進浮動小数点定数
10 進固定小数点定数から成る仮数と、3 桁を超えないオプションの 符号付き整定数が後に付いた文字 E から成る指数とで構成される値。

10 進浮動小数点値 (decimal floating-point value)
10 進小数部と考えることができる仮数形式の実数、 および 10 を底とする整数のべき乗と考えることができる指数の近似値。 「*2* 進浮動小数点値 *(binary floating-point value)*」と対比。

10 進ピクチャー・データ (decimal picture data)
「数値ピクチャー・データ *(numeric picture data)*」を参照。

宣言 (declaration)
ID を名前として確立し、その ID 用に一連の属性 を (部分的または全体的に) 指定すること。 特定名の属性のソース。

デフォルト
指定がされていないときに、とられる値、属性、またはオプション。

定義された変数 (defined variable)
指定された基底付き変数用の一部または全部のストレージに関連付けられる変数。

区切る (delimit)
1 つ以上の項目またはステートメントの前後を、文字またはキーワードで囲むこと。

区切り文字
すべてのコメントと、パーセント記号、括弧、コンマ、ピリオド、セミコロン、コロン、 割り当て記号、ブランク、ポインター、アスタリスク、および単一引用符。 これらは ID、定数、ピクチャー指定、iSUB、およびキーワードの 限界を定めるものとなる。

記述子
区域サイズ、配列境界、またはストリング長 などの変数に関する情報を保持する制御ブロック。

数字 (digit)
0 から 9 までの文字の 1 つ。

次元属性 (dimension attribute)
配列の次元数を指定し、各次元の境界を示す属性。

使用不可の (disabled)
割り込みが発生せず、規定の処置も取られないような事態になった状態。

DO グループ (do-group)
DO ステートメントで区切られ、それに対応する END ステートメント で終了する、制御目的に使用される一連のステートメント。 「ブロック *(block)*」と対比。

DO ループ (do-loop)
「反復 *DO* グループ *(iterative do-group)*」を参照。

仮引数 (dummy argument)
参照によって渡すことのできない引数の値を保持するため自動的に 作成される一時記憶域。

ダンプ
エラーの原因のトレースなどの、プログラムが使用するストレージの一部 または全部、または他のプログラム情報の印刷出力。

E

EBCDIC
拡張 2 進化 10 進コード (Extended Binary-Coded Decimal Interchange Code)。 8 ビットのコード化文字からなるコード化文字セット。

編集ディレクティブ伝送 (edit-directed transmission)
データが連続した文字ストリームとして中にあり、関連データ・ リストに対して行いたい編集を指定するにはフォーマット・リストを必要とするような タイプのストリーム指向伝送。

エレメント
配列などのデータ項目の集まりとは対照的な、単一のデータ項目。スカラー項目。

要素式 (element expression)
評価されるとエレメント値を生じる式。

要素変数 (element variable)
エレメントを表す変数。スカラー変数。

エレメント名 (elementary name)
「基本エレメント *(base element)*」を参照。

使用可能 (enabled)
条件により割り込みが生じて、該当する規定 ON ユニットが呼び出される条件の状態。

end-of-step メッセージ (end-of-step message)
ジョブ制御ステートメントとジョブ・スケジューラー・メッセージのリストに続いており、 各ステップの成功または失敗を示す戻りコードを含むメッセージ。

入り口定数 (entry constant)
PROCEDURE ステートメントのラベル接頭部 (入り口名)。 ENTRY 属性を指定し、VARIABLE 属性を指定しないで名前を宣言すること。

入り口データ (entry data)
プロシージャーへのエントリー・ポイントを表すデータ項目。

入り口式 (entry expression)
評価されると入り口名を生じるような式。

入り口名 (entry name)
ENTRY 属性を持つものとして明示的または内容に従って宣言された ID (ただし、VARIABLE 属性が与えられていない場合に限る)。または、ENTRY 属性を暗黙指定された入り口変数の値を持った ID。

エントリー・ポイント (entry point)
プロシージャー内の、そのプロシージャーを呼び出すことができるポイント。 *1* 次エントリー・ポイント および *2* 次エントリー・ポイント。

入り口参照 (entry reference)
入り口値を返す入り口定数、入り口変数参照、または関数参照。

入り口変数 (entry variable)
入り口値を割り当てる対象となりうる変数。 これは、ENTRY 属性と VARIABLE 属性を両方とも持っている必要がある。

入り口値 (entry value)
入り口定数または入り口変数によって表されるエントリー・ポイント。 入り口値には、その入り口定数に関連した活動化環境が含まれる。

環境 (活動化の) (environment (of an activation))
収容ブロック内で宣言されたデータに関して、呼び出されたブロックと関連し、 そのブロック内で使用される情報。

環境 (ラベル定数の) (environment (of a label constant))
ステートメント・ラベル定数への参照が適用されるブロックの個々の 活動化の識別情報。 この情報が決定されるのは、ステートメント・ラベル定数が、引数として 渡されたり、またはステートメント・ラベル変数に割り当てられ、 それが定数と一緒に渡されたり割り当てられたときである。

確立された処置 (established action)
条件が生じたときにとられる処置。 「暗黙の処置 *(implicit action)*」および「*ON* ステートメント処置 *(ON-statement action)*」も参照。

エピローグ (epilogue)
ブロックまたはタスクの終了時に自動的に生じる各種処理。

評価 (evaluation)
単一の値、値の配列、または値の構造化値へ式を換算すること。

イベント
状況および完了を、関連したイベント変数から決定することのできる プログラムの活動。

イベント変数 (event variable)
イベントと関連付けることができる EVENT 属性を持つ変数。 その値は、処置が完了したかどうか、および完了の状況を示す。

明示宣言 (explicit declaration)
ラベル接頭部として DECLARE ステートメント内、または パラメーター・リスト内に ID (名前) を出すこと。 「暗黙宣言 *(implicit declaration)*」と対比。

指数文字 (exponent characters)
以下のピクチャー指定文字のこと。

1. K および E。指数フィールドの先頭を示すため、浮動小数点 ピクチャー指定内で使用される文字。
2. F。10 進小数点をその想定位置から 右方向へ (正定数の場合) かまたは左方向へ (負定数の場合) 移動する ときに、小数部の桁数を示す整定数を使って指定される位取り係数文字。

式
値、値の配列、または一連の構造化値セットを表すのに、プログラム内で使われる表記。 単独で使用される定数または参照、あるいは、定数または 参照あるいはその両方を演算子と組み合わせたもの。

拡張英字 (extended alphabet)
A から Z までの大文字、小文字の 英字、$、@、および #、または NAMES コンパイラー・オプションで指定されたもの。

エクステント
配列の次元の境界、ストリング長、または区域サイズによって示される範囲。 この区域がターゲット区域に割り当てられる場合は、ターゲット区域のサイズ。

外部名 (external name)
有効範囲が必ずしも 1 つのブロックとその収容ブロックだけに 限定されない (EXTERNAL 属性を持つ) 名前。

外部プロシージャー
他のいずれのプロシージャーにも組み込まれないプロシージャー。 パッケージ内に入っていて同様にエクスポートされるレベル 2 の プロシージャー。

外部シンボル (external symbol)
それ自身が定義されている制御セクションを除く制御セクション内で参照できる名前。

外部シンボル辞書 (External Symbol Dictionary (ESD))
オブジェクト・モジュール内で使われるすべての外部シンボルの一覧表。

特別言語文字
英数字にも特殊文字にも分類されない文字 ($、@、および # など)。 このグループには、NAMES コンパイラー・オプションで指定された文字も 含まれる。

F

属性分配 (factoring)
1 つ以上の属性を、DECLARE ステートメント内の括弧で囲まれた名前リストに対して適用して、 複数の名前に共通する属性を反復する必要をなくすこと。

フィールド (データ・ストリーム中の) (field (in the data stream))
単一データまたはスペーシング・フォーマット項目によって、幅 (文字数) を定義 されるデータ・ストリームの部分。

フィールド (ピクチャー指定の) (field (of a picture specification))
任意の文字ストリング・ピクチャー指定、または固定小数点数を 記述した数字ピクチャー指定の部分 (または全部)。

ファイル
プログラムにおいて、単数または複数のデータ・セットを 名前付きで表現したもの。 ファイルは、オープンするごとに、単数または複数の データ・セットに関連付けられる。

ファイル定数 (file constant)
FILE 属性を指定し、VARIABLE 属性を指定しないで宣言された名前。

ファイル記述属性 (file description attribute)
各ファイル定数の個々の特性を記述したキーワード。 「代替属性 *(alternative attribute)*」と「追加属性 *(additive attribute)*」も参照。

ファイル式 (file expression)
評価されるとファイル・タイプを生じる式。

ファイル名 (file name)
ファイル用に宣言された名前。

ファイル変数 (file variable)
ファイル定数を割り当てることのできる変数。 この場合、ファイルは、FILE 属性と VARIABLE 属性を持って いなければならず、ファイル記述属性を持っていることはできない。

固定小数点定数 (fixed-point constant)
「算術定数 *(arithmetic constant)*」を参照。

修正 (fix-up)
コンパイル済みプログラムを実行可能にするために、 コンパイル時にエラーを検出したあとでコンパイラーが実行する解決手段。

浮動小数点定数 (floating-point constant)
「算術定数 *(arithmetic constant)*」を参照。

制御の流れ (flow of control)
実行の連なり。

形式
ストリーム内のデータ項目の表現法を記述したり (データ・フォーマット項目)、またはストリーム内のデータ項目の個々の位置決めを 記述する (制御フォーマット項目) ために編集ディレクティブ・データ伝送内で使用される仕様。

フォーマット定数 (format constant)
FORMAT ステートメントでのラベル接頭部。

フォーマット・データ (format data)
FORMAT 属性を指定された変数。

フォーマット・ラベル (format label)
FORMAT ステートメントでのラベル接頭部。

フォーマット・リスト (format list)
ストリーム指向伝送における外部メディアでのデータ項目のフォーマット を指定したリスト。 「データ・リスト *(data list)*」と対比。

完全修飾名 (fully-qualified name)
名前が参照するメンバーより上の階層順序内のすべての名前と、 そのメンバー自身の名前が組み込まれている名前。

関数 (プロシージャー) (function (procedure))
PROCEDURE ステートメント内に RETURNS オプションのある プロシージャー。 RETURNS 属性を指定して宣言された名前。 これは、関数参照内にその入り口名のうちの 1 つがあると呼び出され、 スカラー値を参照点に返す。 「サブルーチン *(subroutine)*」と対比。

関数参照 (function reference)
入り口定数または入り口変数のことで、このどちらも関数を 表さなければならないが、その後に空と考えられる引数リストが 続く。 「サブルーチン呼び出し *(subroutine call)*」と対比。

G

世代 (変数の) (generation (of a variable))
静的変数の割り振り、被制御変数または自動変数の特定の割り振り、 または基底付き変数の特定のロケーター修飾で、または定義された 変数かパラメーターで指示されるストレージ。

総称記述子 (generic descriptor)
GENERIC 属性内で使用する記述子。

総称キー (generic key)
キー・クラスを識別する文字ストリング。 そのストリングで始まるキーはすべて、そのクラスのメンバーである。 例えば、 「ABCD」、 「ABCE」、および「ABDF」、という記録済みキーは、すべて総称キー「A」および「AB」で識別されるクラスのメンバーであり、最初の 2 つは、「ABC」というクラスのメンバーでもある。そして、これら 3 つの記録済みキーは、 それぞれ「ABCD」、「ABCE」、「ABDF」というクラスの 固有のメンバーであると見なすことができる。

総称名 (generic name)
入り口名ファミリーの名前。 総称名への参照は、呼び出し点にある引数リスト内の引数の属性に一致するパラメーター記述子を持った入り口名によって置き換えられる。

グループ
より大きいプログラム単位に入っているステートメントの 集まり。 グループは、DO グループまたは選択グループのどちらかであるが、 ON ユニットとしての場合を除き、単一ステートメントを使用できるところでは常に使用することができる。

H

16 進
「*16* 進数字 *(hexadecimal digit)*」を参照。

16 進数
16 の基数を持った数体系。有効数は 0 から 9 の数字と、A は 10 を、F は 15 を 表す A から F までの文字。

16 進数字 (hexadecimal digit)
0 から 9 および A から F の数字の 1 つ。 A から F は、それぞれ 10 から 15 までの 10 進値を表します。

I

ID
コメントや定数内に入ることがなく、前後に区切り文字を伴う文字のストリング。 ID の先頭文字は、26 個の英字、または特別言語文字 (ある場合) でなければならない。 その他の文字がある場合には、拡張英字、数字、区切り文字を追加して入れることができる。

IEEE
米国電気電子学会 (Institute of Electrical and Electronics Engineers)。

暗黙的な (implicit)
明示指定のないまま取られる処置。

暗黙処置 (implicit action)
使用可能な条件が生じたときに、その条件用に現在確立されている ON ユニットがない場合にとられる処置。 「*ON* ステートメント処置 *(ON-statement action)*」と対比。

暗黙宣言 (implicit declaration)
DECLARE ステートメント内で明示的に宣言されていないか、 または内容に従って宣言されていない名前。

暗黙オープン
OPEN ステートメント以外の入力ステートメントまたは出力ステートメント が原因で、ファイルがオープンされること。

挿入演算子 (infix operator)
2 つのオペランド間にある演算子。

継承次元 (inherited dimension)
構造体、共用体、またはエレメントでの、収容構造から派生する次元。 名前が配列ではないエレメントであれば、その次元 全体が継承次元で構成される。 名前が配列であるエレメントであれば、その次元は、継承次元および 明示的に宣言された次元で構成される。 1 つ以上の継承次元を持つ構造体を、非結合集合と呼ぶ。 「結合集合 *(connected aggregate)*」と対比。

入出力 (input/output)
補助メディアと主記憶装置との間でデータを転送すること。

挿入点文字 (insertion point character)
関連データを文字ストリングへ割り当てるときに、指示位置に 挿入されるピクチャー指定文字。 入力のときに P フォーマット項目内で使用される挿入文字は、検査の目的で用いられる。

整数
符号を付けるか付けないかは任意の、10 進または 2 進小数点のない一連の数字、または 一連のビット。 通常は、FIXED BINARY (p,0) または FIXED DECIMAL (p,0) と記述される、 符号を付けるか付けないかは任意の整数。

規定境界 (integral boundary)
そこでデータを位置合わせすることができる任意の 8 ビット単位のバイト・マルチアドレス。 通常はハーフワード、フルワード、 またはダブルワード (2、4、または 8 バイトの長さの整倍数) 境界である。

介在配列 (interleaved array)
非連結ストレージを参照する配列。

介在添え字 (interleaved subscripts)
添え字付き修飾参照の最下位レベル以外のレベルに存在する添え字。

内部ブロック (internal block)
ブロックの中に組み込まれている別のブロック。

内部名 (internal name)
名前が宣言されたブロック内のみで認識されている名前、また そのブロック内に入っているブロックの中でも認識されている可能性もある名前。

内部プロシージャー (internal procedure)
ブロックの中に組み込まれている別のプロシージャー。 「外部プロシージャー *(external procedure)*」と対比。

割り込み (interrupt)
条件やアテンションの発生の結果として、プログラムの制御の流れを変更すること。

起動 (invocation)
プロシージャーの活動化。

起動する (invoke)
プロシージャーを活動化すること。

呼び出されたプロシージャー (invoked procedure)
活動化されているプロシージャー。

呼び出し側ブロック (invoking block)
プロシージャーを活動化するブロック。

反復因数 (iteration factor)
INITIAL 属性指定において、特定の値を使って初期化されることになっている 配列の連続エレメント数を指定するための式。 フォーマット・リストにおける、特定のフォーマット項目またはフォーマット項目のリストを 連続して使用する回数を指定するための式。

反復 DO グループ (iterative do-group)
制御変数または WHILE や UNTIL オプション、またはこの両方 を指定した DO ステートメントを持つ DO グループ。

K

キー
直接アクセス・データ・セット内のレコードを 識別するデータ。 「ソース・キー *(source key)*」および「記録済みキー *(recorded key)*」を参照。

キーワード
PL/I において定義されたコンテキスト内で使用されると特定の意味を持つ ID。

キーワード・ステートメント (keyword statement)
ステートメントの機能を示すキーワードで始まる単純ステートメント。

認識された (名前に関する用語) (known (applied to a name))
宣言された意味で認識されること。 名前は、その有効範囲内で認識される。

l

ラベル
ステートメントの接頭部として付く名前。 PROCEDURE ステートメント上の名前を、入り口定数と呼び、FORMAT ステートメント上の名前を、 フォーマット定数と呼ぶ。その他の種類のステートメント上の名前を、ラベル定数と呼ぶ。 LABEL 属性を持つデータ項目。

ラベル定数 (label constant)
ステートメント (PROCEDURE、ENTRY、FORMAT、または PACKAGE を除く) のラベル接頭語として書き込まれる名前。実行時に、そのラベル接頭語が 参照されれば、そのステートメントにプログラムの制御を渡す ことができる。

ラベル・データ
ラベル定数または、ラベル変数の値。

ラベル接頭部 (label prefix)
ステートメントの接頭部として付けられたラベル。

ラベル変数 (label variable)
LABEL 属性を指定して宣言された変数。 その値は、プログラム内でのラベル定数である。

先行ゼロ (leading zeroes)
算術値としては意味のないゼロ。 ある数値内で最初の非ゼロより左側にあるすべてのゼロ。

レベル番号 (level number)
DECLARE ステートメント中の名前の前に付く番号で、 構造体名の階層内のその相対位置を指定するもの。

レベル 1 変数 (level-one variable)
大構造体または共用体の名前。 構造体または共用体の中に含まれていない、添え字なし変数。

字句単位の (lexically)
単位を左から右への順序に扱うことに関連した用語。

ライブラリー
メンバーと呼ばれるその他のデータ・セットを保管するのに使用できる MVS 区分データ・セット 、または CMS MACLIB のこと。

リスト指示 (list-directed)
ストリーム内のデータがブランクやコンマで区切られた定数になり、 フォーマット設定が自動的に行われるタイプのストリーム指向伝送。

ロケーター
変数のアドレスまたはその記述子を保持する制御ブロック。

ロケーター/記述子 (locator/descriptor)
その後に記述子の付いたロケーター。 ロケーターは、記述子のアドレスではなく、変数のアドレスを保持する。

ロケーター修飾
基底付き変数への参照において、その参照が参照している基底付き変数 の世代を指定するために、基底付き変数の左側に矢印で接続されているロケーター変数 または関数参照。 これは、暗黙参照であることもある。

ロケーター値 (locator value)
ストレージ・アドレスを識別する値か、またはストレージ・アドレスを識別するのに使用できる値。

ロケーター変数 (locator variable)
変数またはバッファーの主記憶域内の位置を識別する値を 持った変数。 これは、POINTER 属性または OFFSET 属性を 持つ。

ロック・レコード (locked record)
EXCLUSIVE DIRECT UPDATE ファイル内のレコードであって、1 つのタスクにだけ しか使用することはできず、そのレコードを使用しているタスクによって 解放されるまで、他のタスクからアクセスできないレコード。

論理レベル (構造体または共用体メンバーの) (logical level (of a structure or union member))
全レベル番号が直接順序になっているとき (あるレベル番号から次の レベル番号までの増分が 1 のとき) に、レベル番号で示される深さ。

論理演算子 (logical operators)
ビット・ストリング演算子 NOT と排他 OR (¬)、AND (&)、および OR (|)。

ループ (loop)
繰り返し実行される一連の命令。

下限 (lower bound)
配列次元の下限。

m

主プロシージャー (main procedure)
OPTIONS (MAIN) 属性を持った PROCEDURE ステートメントのある 外部プロシージャー。 このプロシージャーは、プログラム実行の 最初のステップで自動的に呼び出される。

大構造 (major structure)
レベル番号 1 を指定して宣言された名前を持つ構造。

メンバー
構造体または共用体の中の、構造体、共用体、あるいはエレメントの名前。 ライブラリー内のデータ・セット。

小構造 (minor structure)
別の構造体または共用体の中に組み込まれている構造体。 小構造の名前は、1 よりも大きくかつ親構造体または 親共用体よりも大きいレベル番号を指定して宣言される。

モード (算術データの) (mode (of arithmetic data))
算術データの属性。 これは、実数 または複素数 のどちらかである。

多重宣言 (multiple declaration)
同一ブロックに対して内部であり、別の修飾を持たない同一 ID の複数宣言。 同一 ID の複数外部宣言。

マルチプロセッシング (multiprocessing)
複数のプログラムを同時に実行するために、複数の処理装置 を備えた計算機システムを使用すること。

マルチプログラミング (multiprogramming)
単一の処理装置を使って、複数のプログラムを並行して 処理するのに、計算機システムを使用すること。

マルチタスキング
複数の PL/I プロシージャーをプログラムが同時に実行できるようにする機能。

N

名前
変数や定数にユーザーが与える ID。 コンテキスト中に現れる、キーワードではない ID。 場合によっては、ユーザー定義名とも呼ぶ。

ネスト (nesting)
次のものの発生。

- ブロック内にある別のブロック。
- グループ内にある別のグループ。
- THEN 文節または ELSE 文節内の IF ステートメント。
- 関数参照の引数としての関数参照。
- FORMAT ステートメントのフォーマット・リスト内のリモート・フォーマット項目。
- パラメーター記述子リスト内の別のパラメーター記述子リスト。
- 1 つ以上の属性が分配されている括弧で囲まれた名前リスト内の属性の指定。

非結合ストレージ (nonconnected storage)
非結合データ項目が占有するストレージ。 例えば、継承次元を持つ介在配列や構造体は、非結合ストレージ 内にある。

ヌル・ロケーター値 (null locator value)
内部記憶域内のどの位置も識別できない特殊ロケーター値。 これは、現在ロケーター変数がデータの世代を識別できないことを示すのに役立つ。

ヌル・ステートメント (null statement)
セミコロン記号 (;) のみが含まれているステートメント。 これは、アクションが実行されないことを示します。

ヌル・ストリング (null string)
長さゼロの文字ストリング、漢字ストリング、またはビット・ストリング。

数字データ (numeric-character data)
「*10* 進ピクチャー・データ *(decimal picture data)*」を参照。

数値ピクチャー・データ (numeric picture data)
算術値と文字値を持ったピクチャー・データ。 このタイプのピクチャー・データは、'A' または 'X' という文字を含むことはできない。

O

オブジェクト
単一名で参照されるデータの集まり。

オフセット変数 (offset variable)
OFFSET 属性を持ったロケーター変数のことであり、その値は、ストレージ内のある区域の先頭からの相対位置を識別する。

オン条件 (ON-condition)
PL/I プログラムにおける、プログラム割り込みの原因となりうる オカレンス。 予期しないエラーが検出されたり、予期できる出来事ではあるものの、 予期しない時にそれが起きたときに発生する。

ON ステートメント処置 (ON-statement action)
ある条件が生じたときに処置を取れるよう、条件に対して 明示的に設定された処置法。 プログラムの制御の流れ内で ON ステートメントが見つかると、とられる処置で、その条件に対する処置が設定される。 この処置は、ON ユニットが設定されたままであるか、RESIGNAL ステートメントで再設定されて条件が生じたときにとられる。 「暗黙の処置 *(implicit action)*」と対比。

ON ユニット (ON-unit)
該当する条件が起きたときに、とられるよう指定された処置。

オープン (ファイルの) (opening (of a file))
ファイルをデータ・セットに関連付けること。

オペランド
ID、定数、または式。式には演算子が、時には他のオペランドと ともに使用される。

演算式 (operational expression)
1 つ以上の演算子から成る式。

演算子
実行する演算を指定する記号。

オプション
ステートメントの実行や解釈に影響を及ぼすのに使われるステートメント中 の指定。

P

パッケージ定数 (package constant)
PACKAGE ステートメントのラベル接頭語。

パック 10 進 (packed decimal)
固定小数点 10 進データ項目の内部表現。

埋め込み
ストリングの長さを必要な長さまで拡張するために、ストリングの右側に 連結される、1 つ以上の文字、漢字、またはビット。 構造体または共用体の中に挿入される、1 つ以上のバイトまたはビット。 その構造、または共用体内の後続エレメントが正しい規定境界に位置合わせされるようにするためのもの。

パラメーター
PROCEDURE ステートメントの後に続くパラメーター・リスト中の名前。 そのプロシージャーが呼び出されれば、渡される引数を指定する。

パラメーター記述子 (parameter descriptor)
ENTRY 属性指定内でパラメーター用に指定される一連の属性。

パラメーター記述子リスト (parameter descriptor list)
ENTRY 属性指定内のすべてのパラメーター記述子のリスト。

パラメーター・リスト (parameter list)
コンマで区切られ、プロシージャー・ステートメント内の キーワード PROCEDURE の後に続くか、または ENTRY ステートメント内の キーワード ENTRY の後に続く、括弧で囲まれた 1 つ以上のパラメーターのリスト。 このリストは、呼び出し時に渡される引数リストと対応する。

部分修飾名 (partially-qualified name)
不完全な修飾名。 これには名前が参照する構造メンバーまたは共用体メンバーより上の階層順序内にある 名前のうちの全部ではない 1 つ以上の名前、およびそれ自身のメンバー名が 含まれる。

ピクチャー・データ (picture data)
文字形式で表された、数値データ、文字データ、またはそれらの混合。

ピクチャー指定 (picture specification)
PICTURE 属性を指定した宣言内で、または P フォーマット項目内でピクチャー文字を 使用して宣言されたデータ項目。

ピクチャー指定文字 (picture specification character)
ピクチャー指定で使用できるすべての文字。

PL/I 文字セット (PL/I character set)
PL/I のプログラム・エレメントを表現するために定義されている文字セット。

PL/I プロンプター (PL/I prompter)
PL/I コマンドのコマンド・プロセッサー・プログラムで、オペランドを調べ、コンパイラーに必要なデータ・セットを割り振る。

呼び出し点 (point of invocation)
呼び込まれたプロシージャーへの参照が現れる呼び込み側 ブロック内の地点。

ポインタ
ストレージ内の位置を識別するための変数のタイプ。

ポインター値 (pointer value)
ポインター型を識別する値。

ポインター変数 (pointer variable)
ポインター値が入った POINTER 属性を持ったロケーター変数。

精度 (precision)
固定小数点データ項目内にある桁数またはビット数、または、 浮動小数点データ項目での最小確保有効数字 (指数は除く) の数。

接頭部 (prefix)
ステートメントの先頭に付けられるラベル、または 1 つ以上 の条件名の括弧で囲まれたリスト。

接頭演算子 (prefix operator)
オペランドの前に置かれ、そのオペランドにだけ適用される演算子。 接頭演算子には、プラス (+)、マイナス (-)、および not (¬) がある。

プリプロセッサー
コンパイルを実行する前に、ソース・プログラムを調べるためのプログラム。

プリプロセッサー・ステートメント (preprocessor statement)
プリプロセッサーがとる 処置を指定するために、ソース・プログラム内に入れる特殊ステートメント。これは、プリプロセッサーによって検出されると実行 される。

1 次エントリー・ポイント (primary entry point)
PROCEDURE ステートメントのラベル・リスト内の任意の名前によって識別される エントリー・ポイント。

優先
タスクに関連する値で、他のタスクに対するそのタスクの優先順位 (指名順 位) を指定する。

問題データ (problem data)
コード化算術データ、ビット・データ、文字データ、グラフィック・データ、およびピクチャー・データ。

問題状態プログラム (problem-state program)
オペレーティング・システムの問題プログラム状態内で 稼働するプログラム。 これには、入出力指示やその他の特権命令は入らない。

プロシージャー
PROCEDURE ステートメントと END ステートメントで 区切られたステートメントの集まり。 プロシージャーとはプログラムまたはプログラムの一部であり、名前の有効範囲を区切り、 そのプロシージャーまたは入り口名の 1 つへの参照によって活動化される。 「外部プロシージャー *(external procedure)*」および「内部プロシージャー *(internal procedure)*」も参照。

プロシージャー参照 (procedure reference)
入り口定数または入り口変数。 この後に引数リストを続けることができる。 プロシージャー参照は、CALL ステートメントや CALL オプションに入れることも、または 関数参照として使用することもできる。

プログラム
1 つ以上の外部プロシージャーまたはパッケージのセット。 外部プロシージャーのうちの 1 つは、PROCEDURE ステートメント内に OPTIONS(MAIN) 指定を 持っていなければならない。

プログラム制御データ (program control data)
PL/I プログラムの処理を制御するのに使用するための 区域、ロケーター、ラベル、フォーマット、項目、およびファイルのデータ。

プロローグ (prologue)
ブロックの起動時に自動的に生じる処理。

疑似変数 (pseudovariable)
ターゲット変数を指定するのに使用できるすべての組み込み関数の名前。 これは通常、代入ステートメントの左側にある。

Q

修飾名 (qualified name)
構造メンバーまたは共用体メンバーの階層順序。ピリオドで結合されていて、 構造体の中の名前を識別するのに使用される。 どの名前にも添え字を付けることができる。

R

範囲 (デフォルト指定の) (range (of a default specification))
DEFAULT ステートメント内の属性を適用される ID またはパラメーター記述子のどちらか、またはこの両方のセット。

レコード
レコード単位入力または出力の操作における、伝送の 論理単位。 1 つ以上の関連データ項目の集まり。これらの項目には通常、それぞれ異なったデータ属性があり、 また通常は構造体か共用体の宣言で記述される。

記録済みキー (recorded key)
直接アクセス・データ・セット内でレコードを識別する 文字ストリングのことであり、そこでは文字ストリングそのものも データの一部として記録される。

レコード単位データ伝送 (record-oriented data transmission)
別々のレコードの形式でデータを伝送すること。「ストリーム指向のデータ伝送 *(stream data transmission)*」と対比。

再帰的プロシージャー (recursive procedure)
そのプロシージャー自身からでも、または別のアクティブ・プロシージャー からでも呼び出すことのできるプロシージャー。

再入可能プロシージャー (reentrant procedure)
複数のタスク、スレッド、またはプロセスから同時に活動化でき、 しかもこれらのタスク、スレッド、およびプロセス間で相互に干渉が 生じないプロシージャー。

REFER 式 (REFER expression)
REFER というキーワードの前に付いた式。この式は、REFER オプションを含む基底付き変数が、ALLOCATE ステートメントまたは LOCATE ステートメントのいずれかによって割り振られるときの境界、長さ、または サイズとして使用される。

REFER オブジェクト (REFER object)
REFER オプション中の変数。メンバーの現行境界、長さ、またはサイズを保持しているか、 あるいは保持する予定のもの。 REFER オブジェクトは、同一構造体または共用体のメンバーでなければならない。 これは、ロケーター修飾したり添え字を付けてはならず、また REFER オプションを持ったメンバーの前になければならない。

reference
明示宣言を生じることになる 1 つのコンテキスト内以外の名前の出現。

相対仮想起点 (relative virtual origin (RVO))
配列の実際の原点から配列の仮想原点を引いたもの。

リモート・フォーマット項目 (remote format item)
R という文字の後に FORMAT ステートメントのラベル (括弧に囲まれている) のあるもの。 フォーマット指定ステートメントは、転送するデータのフォーマットを制御するために、 編集ディレクティブ・データ伝送ステートメントにより使用する。

反復因数 (repetition factor)
以下のものを指定する、括弧に入れられた符号なし整定数。

1. 後続するストリング定数を繰り返す回数。
2. 後続するピクチャー文字を繰り返す回数。

反復指定 (repetitive specification)
1 つ以上のデータ項目伝送の被制御反復を指定するための データ・リストの 1 エレメントであり、通常は配列と同時に使用する。

制限付き式 (restricted expression)
コンパイル時にコンパイラーによって評価されて定数を生じる式。 このような式のオペランドは、定数、指定した定数、および 制限付きの式になる。

戻り値 (returned value)
関数プロシージャーから返される値。

RETURNS 記述子 (RETURNS descriptor)
RETURNS 属性内と、PROCEDURE および ENTRY ステートメントの RETURNS オプション内 で使用する記述子。

S

スカラー変数 (scalar variable)
構造、共用体、配列ではない変数。

スケール (scale)
1 つの数値表記体系であり、その算術値は固定小数点 または浮動小数点で表現される。

位取り係数 (scale factor)
固定小数点数内の小数桁数の指定。

位取り係数 (scaling factor)
「位取り係数 *(scale factor)*」を参照。

有効範囲 (条件接頭語の) (scope (of a condition prefix))
全体にわたって特定の条件接頭語が適用される、プログラムの部分。

有効範囲 (宣言の) (scope (of a declaration or name))
全体にわたって特定名が認識されているプログラムの部分。

2 次エントリー・ポイント (secondary entry point)
入り口ステートメントのラベル・リスト内の任意の名前によって識別されるエントリー・ポイント。

選択グループ (select-group)
SELECT ステートメントと END ステートメントで区切られたステートメントの連なり。

選択文節 (selection clause)
選択グループの WHEN 文節または OTHERWISE 文節。

自己定義データ (self-defining data)
プログラム実行時に決定され、集合のメンバー内に保管される 境界、長さ、およびサイズを持つデータ項目を含む集合。

分離文字 (separator)
「区切り文字 *(delimiter)*」を参照。

シフト (shift)
ストレージ内のデータを元の位置の左または右へ変更すること。

シフトイン (shift-in)
2 バイト・ストリングの終わりをコンパイラーに知らせるために 使用される記号。

シフトアウト (shift-out)
2 バイト・ストリングの先頭でコンパイラーにシグナルを送るために 使用される記号。

符号および通貨記号 (sign and currency symbol characters)
ピクチャー指定文字。 S、+、-、および $ (または < と > で囲まれたその他の通貨記号)。

単純パラメーター (simple parameter)
ストレージ・クラス属性が指定されていないパラメーター。 単純パラメーターはどのストレージ・クラスの引数も表すことができるが、 被制御引数の現行世代だけを表すことができる。

単純ステートメント (simple statement)
IF、ON、 WHEN、および OTHERWISE 以外のステートメント。

ソース
問題データに変換されるデータ項目。

ソース・キー (source key)
直接アクセス・データ・セット内で個々のレコードを 識別するため、レコード単位伝送ステートメント内で参照されるキー。

ソース・プログラム
ソース・プログラム・プロセッサー、およびコンパイラーへの入力となる プログラム。

ソース変数 (source variable)
他の演算に使用されるが、その演算で変更されることのない変数。「ターゲット変数 *(target variable)*」と対比。

予備ファイル (spill file)
一時作業ファイルとして使われる SYSUT1 という名前のデータ・セット。

標準デフォルト (値) (standard default)
属性またはオプションの指定がなく、適用できる DEFAULT ステートメント がない場合の、代替属性または代替オプション。

標準ファイル (standard file)
GET ステートメントや PUT ステートメントで FILE オプションまたは STRING オプションが ない場合に、PL/I が想定するファイル。 SYSIN が標準入力ファイルであり、SYSPRINT が標準出力ファイルである。

標準システム処置 (standard system action)
使用可能な条件のための ON ユニットがないときにその条件が発生した場合にとられる、言語で指定された処置。

ステートメント
キーワード、区切り文字、ID、演算子、および定数で構成され、セミコロン (;) で終了する PL/I ステートメント。 オプションで、条件接頭語リストとラベルのリストを持つことができます。「キーワード・ステートメント *(keyword statement)*」と「ヌル・ステートメント *(null statement)*」も参照。

ステートメント本体 (statement body)
ステートメント本体は、単純ステートメントまたは 複合ステートメントのどちらでもかまわない。

ステートメント・ラベル (statement label)
「ラベル定数 *(label constant)*」を参照。

静的ストレージ割り振り (static storage allocation)
静的変数用のストレージの割り振り。

静的変数 (static variable)
プログラム実行の開始前に割り振られ、その実行の継続時間中は その割り振りの変わらない変数。

ストリーム指向データ伝送
文字形式になった個々のデータ値の連続ストリームであるものとしてデータを 扱って、データを伝送すること。 「レコード単位データ伝送 *(record-oriented data transmission)*」と対比。

ストリング
単一のデータ項目として処理される、連続した文字、グラフィックス、 またはビットの列。

ストリング変数 (string variable)
BIT、 CHARACTER、または GRAPHIC 属性を指定して宣言される変数。 この変数の値は、ビット・ストリング、文字ストリング、または漢字ストリング のいずれでもかまわない。

構造化
必ずしも同じ属性を持たなくても差し支えないデータ項目の集まり。 「配列 *(array)*」と対比。

構造式 (structure expression)
評価されると構造体の値セットを生成する式。

配列の構造体 (structure of arrays)
次元属性を持つ構造体。

構造体メンバー (structure member)
「メンバー *(member)*」を参照。

構造化 (structuring)
メンバー数、配置順、属性、および論理レベルによって表現される構造階層。

サブルーチン (subroutine)
PROCEDURE ステートメント内に RETURNS オプションのないプロシージャー。 「関数 *(function)*」と対比。

サブルーチン呼び出し (subroutine call)
後に CALL ステートメント内にあるオプションの引数リストが付く、サブルーチンを表さなければならないエントリー参照。 「関数参照 *(function reference)*」と対比。

添え字
配列の次元内の位置を指定するための要素式。 添え字がアスタリスクであれば、次元のすべてのエレメントを 指定する。

添え字リスト (subscript list)
括弧に入れられた、1 つ以上の添え字のリスト。配列の各々の 次元に対して 1 つの添え字が対応する。これらによって配列の単一エレメントまたはクロスセクションを一意的に識別する。

サブタスク (subtask)
特定のタスクによって生成されるタスク、または特定のタスクから最後に 生成されたタスクへの直接ライン内の任意のタスク。

同期 (synchronous)
プログラムの順次実行での単一の制御の流れ。

T

ターゲット
データ項目 (ソース) が変換される属性。

ターゲット参照 (target reference)
受取側変数 (または受取側変数の一部) を指定する参照。

ターゲット変数 (target variable)
値が割り当てられる変数。

タスク
単一の制御の流れによる 1 つ以上のプロシージャーの実行。

タスク名 (task name)
タスク変数を参照するのに使用される ID。

タスク変数 (task variable)
TASK 属性を持ち、その値がタスクの相対優先順位を示す変数。

終了 (ブロックの) (termination (of a block))
ブロックの実行が終了して、RETURN ステートメントまたは END ステートメントによって、 制御がその起動側ブロックに戻るか、または GO TO ステートメントによって起動側ブロックまたは 他のアクティブ・ブロックに制御が渡ること。

終了 (タスクの) (termination (of a task))
タスクへの制御の流れを停止すること。

切り捨て (truncation)
ターゲット変数のストリング長や精度が限度を超えたときに、 データ項目の片方の端から 1 つ以上の数字文字、グラフィックス、 またはビットを除去すること。

タイプ
データの世代、値、または項目に対して適用される一連の データ属性とストレージ属性。

U

未定義 (undefined)
ユーザーが行ってはならないことを示す。 未定義機能が使用されると、PL/I 製品の個々のインプリメンテーションによって違った 結果が出る可能性がある。 このような場合、アプリケーション・プログラムはエラーとなる。

union
同一のストレージを占有し、相互にオーバーレイしたデータ・エレメントの集まり。 メンバーは構造体、共用体、基本変数、または配列のいずれであっても構わない。 それらは、同一の属性を持っていなくても構わない。

配列の共用体 (union of arrays)
次元属性を持った共用体。

上限 (upper bound)
配列次元の上限。

V

値参照 (value reference)
データ項目の値を得るのに使用する参照。

変数
データを参照するのに使用され、値を割り当てる対象となりうる 名前付きのエンティティー。 その属性は一定のままであるが、場合に応じてそれぞれ 異なる値を参照することができる。

変数参照 (variable reference)
変数全体またはその一部を指定する参照。

仮想起点 (virtual origin (VO))
すべてゼロの添え字を持った配列のエレメントを保持するための位置。 このようなエレメントが配列内になければ、仮想起点は本来それが保持されるべき 場所になる。

Z

ゼロ抑止文字 (zero-suppression characters)
ピクチャー指定文字の Z と *。これは、対応する桁位置の ゼロを抑止し、それぞれをブランクまたはアスタリスクで置き換えるのに使用する。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

アクセス
  相対レコード・データ・セット 301
  ESDS 286
  REGIONAL(1) データ・セット 269
アクセス方式サービス
  領域データ・セット 270
  REGIONAL(1) データ・セット
    順次アクセス 268
    直接アクセス 269
アセンブラー・ルーチン
  FETCH 174, 179
アテンション処理
  アテンション割り込み、効果 45
  主要な説明 441
  デバッグ・ツール 442
  ATTENTION ON ユニット 442
アプリケーション・パフォーマンスの向上 305
一時作業ファイル
  SYSUT1 162
インクルード・プリプロセッサー
  構文 117
印刷
  PRINT ファイル
    行長 241
    形式 249
    ストリーム入出力 241
印刷制御文字 53, 241
エラー
  エラー・コンパイルの重大度 16
エラー装置
  リダイレクト 225
オブジェクト
  モジュール
    作成と保管 60
    レコード・サイズ 161
オプション
  コメントの指定 102
  コンパイル用 106
  コンパイル用に指定する 163
  ストリングの指定 102
  領域データ・セットの作成用の 263
  PLIDUMP に保存されたオプション・ストリング 439
オプション、z/OS UNIX での
  使用
    タイトル 200
    DD 情報 200
  DD_DDNAME 環境変数
    タイプ 206
    APPEND 202
    DELAY 204
    DELIMIT 204
    LRECL 204
    LRMSKIP 204
オプション、z/OS UNIX での (続き)
  DD_DDNAME 環境変数 (続き)
    PROMPT 204
    PUTPAGE 204
    RECCOUNT 205
    RECSIZE 205
    SAMELINE 205
    SKIP0 206
  PL/I ENVIRONMENT 属性
    BUFSIZE 203
オプション・レコード、SYSADATA 462
オフセット
  タブ・カウントの 243
  表 109
オペレーティング・システム
  z/OS UNIX でのデータ定義 (DD) 情報 199

[カ行]

改行 (LF)
  定義 206
外部参照
  連結名 195
カウンター・レコード、SYSADATA 463
拡張 2 進化 10 進コード (EBCDIC) 208
カスタマイズ
  ユーザー出口
    グローバル制御ブロックの構造 448
    独自のコンパイラー出口の作成 450
    SYSUEXIT の変更 450
カタログ式プロシージャー
  コンパイル、バインド、および実行 150, 153
  コンパイルおよびバインド 149, 152
  コンパイルのみ 148
  コンパイルのみ - 64 ビット 151
  コンパイル用入力データ 150, 153
  複数呼び出し 154
  変更
    DD ステートメント 156
    EXEC ステートメント 155
  呼び出し 154
  リスト 154
  OS/390 の下での
    変更する 155
    呼び出す 154
    IBM 提供の 147
可変長レコード
  形式 209
  ソート・プログラム 338
紙送り制御文字 53, 241
環境変数
  z/OS UNIX での設定 224
漢字ストリング定数のコンパイル 39
関数
  ILC で C 関数を使用する 343
キー
  代替索引

キー (続き)
代替索引 (続き)
固有 292
非固有 292
REGIONAL(1) データ・セット
ダミー・レコード 267
VSAM
索引付きデータ・セット 276
相対バイト・アドレス 276
相対レコード 277
キー索引付き VSAM データ・セット 276
キー順データ・セット
ステートメントとオプション 286
ロード 288
DIRECT ファイルを使用したアクセス 290
SEQUENTIAL ファイルを使ったアクセス 289
記述子 455
記述子域、SQL 130
記述子リスト、引数の引き渡し 455
記述子ロケーター、引数の引き渡し 456
行
長さ 241
メッセージ内の数 39
記録済みキー (recorded key)
領域データ・セット 266
句読点
自動プロンプト
指定変更 167
使用 167
OS/390
継続文字 168
端末での ENDFILE の入力 169
GET DATA ステートメント 168
GET EDIT での自動埋め込み 168
GET LIST ステートメント 168
SKIP オプション 169
PRINT ファイルからの出力 166
組み込み
SQL ステートメント 131
グラフィック・データ 233
グローバル制御ブロック
終了プロシージャーの作成 453
初期化プロシージャーの作成 451
メッセージ・フィルター操作プロシージャーの作成 451
グローバル制御ブロックの構造
終了プロシージャーの作成 453
初期化プロシージャーの作成 451
メッセージ・フィルター操作プロシージャーの作成 451
言語間通信
リンケージの考慮事項 347
ATTACH ステートメントの使用 349
C との
構造化データ・タイプ 342
出力の共用 349
ストリング・パラメーター・タイプの一致 345
データ・タイプ 341
入力の共有 349
パラメーターの一致 344
ファイル・タイプ 343
C 関数を使用する 343
ENTRY を戻す関数 346
enum データ・タイプ 342
C 標準ストリームのリダイレクト 350
交換コード 208
更新
相対レコード・データ・セット 301
ESDS 286
REGIONAL(1) データ・セット 269
構文図の読み方 xxi
構文レコード、SYSADATA 469
コーディング
パフォーマンスの向上 311
CICS ステートメント 144
SQL ステートメント 129
固定長レコード 208
コメント
オプション内 102
混合文字ストリング定数のコンパイル 39
コントロール
インターバル 274
サイト 274
CONTROL オプション
EXEC ステートメント 163
コンパイラー
一時作業ファイル (SYSUT1) 162
エラー条件の重大度 16
オプションの説明 3
概要 157
漢字ストリング定数 39
混合ストリング定数 39
削減、ストレージ要件の 61
呼び出し 157
リスト
印刷オプション 162
コンパイラーへの入力 106
集合長さテーブル 108
使用 106
使用されるスタック・ストレージ 88
ステートメント・オフセット・アドレス 109
相互参照テーブル 108
ソース・プログラム 87
ソース・プログラムのインクルード 44
属性テーブル 108
ファイル参照テーブル 111
プリプロセッサーへの入力 107
ブロック・レベル 107
見出し情報 106
メッセージ 112
DO レベル 107
SOURCE オプション・プログラム 107
SYSPRINT。 162
DBCS ID 39
JCL ステートメントの使用 159
OS/390 バッチの下での 159
PROCESS ステートメント 103
% ステートメント 104
コンパイラー・オプション
エックスレフ 102
オプション 62
サービス 87
システム 90
省略語 9
ストレージ 88
説明 3
ソース 87
属性 10
端末 91
出口 35

コンパイラー・オプション (続き)
テスト 91
デフォルト 3, 23, 308
なし 59
名前 58
表示 34
ブランク 12
または 62
マックス・スタティック 56
マックスルーノニフ 56
利用方法 94
AGGREGATE 9
ARCH 9, 305
ASSERT 10
BACKREG 11
BIFPREC 11
BLKOFF 12
CASERULES 13
CHECK 14
CMPAT 14
CODEPAGE 15
COMPILE 16
COPYRIGHT 17
CSECT 17
CSECTCUT 18
CURRENCY 18
DBCS 18
DBRMLIB 19
DD 19
DDSQL 20
DEPRECATE 32
DEPRECATENEXT 33
DLL 34
DLLINIT 34
EXTRN 35
FILEREF 35
FLAG (フラグ) 36
FLOAT 36
FLOATINMATH 38
GOFF 38
GONUMBER 39, 305
GRAPHIC 39
IGNORE 40
INCAFTER 41
INCDIR 41
INCLUDE 42
INSOURCE 44
INTERRUPT 45
LANGLVL 47
LIMITS 48
LINEDIR 49
LIST 50
LISTVIEW 50
MACRO 51
MAP 52
MARGINI 52
MARGINS 52
MAXBRANCH 53
MAXGEN 54
MAXINIT 54
MAXMEM 54
MAXMSG 55
MAXNEST 55
MAXSTMT 56
コンパイラー・オプション (続き)
MAXTEMP 57
MDECK 57
MSGSUMMARY 57
NAMES 58
NATLANG 59
NEST 59
NOMARGINS 52
OBJECT 60
OFFSET 60
OFFSETSIZE 60
ONSNAP 60
OPTIMIZE 61, 305
PP 62
PPCICS 63
PPINCLUDE 64
PPLIST 64
PPMACRO 64
PPSQL 65
PPTRACE 65
PRECTYPE 65
PREFIX 66, 307
PROCEED 66
QUOTE 67
REDUCE 67, 305
RENT 68
RESEXP 69
RESPECT 69
RTCHECK 69
RULES 70, 306
SEMANTIC 86
SPILL 87
STATIC 87
STDSYS 88
STMT 88
STRINGOFGRAPHIC 89
SYNTAX 89
SYSPARM 89
UNROLL 94
WIDECHAR 96
WINDOW 97
WRITABLE 97
XINFO 98
XML 45, 101
コンパイラー・ユーザー出口の初期化プロシージャー 451
コンパイル
ユーザー出口
カスタマイズ 450
活動化 449
プロシージャー 447
例 453
IBMUEXIT 449
z/OS UNIX の下での 157
コンパイルおよびバインド用の入力データ 149, 152
コンパイル時オプション
z/OS UNIX の下での 158
コンパイル時オプションの指定
フラグの使用 159

[サ行]

再始動
要求 445
要求する

再始動 (続き)
要求する (続き)
システム障害後の自動 445
据え置き再始動 445
取り消す 445
プログラム内の自動 445
変更する 445
RESTART パラメーター 445
最適コーディング
コーディング・スタイル 311
コンパイラー・オプション 305
作業データ・セット、ソート用の 330
索引付き ESDS (入力順データ・セット)
ロード 288
DIRECT ファイル 290
SEQUENTIAL ファイル 289
索引データ・セット
索引順次データ・セット 210
削減、ストレージ要件の 61
サマリー・レコード、SYSADATA 462
サンプル・プログラム、実行 356, 360, 363, 368
システム
障害 445
障害後の再始動 445
SYSTEM コンパイラー・オプション
パラメーター・リストのタイプ 90
SYSTEM(CICS) 90
SYSTEM(IMS) 90
SYSTEM(MVS) 90
SYSTEM(OS) 90
SYSTEM(TSO) 90
自動
埋め込み 168
再始動
システム障害後 445
プログラム内の 445
プロンプト
指定変更 167
使用 167
シフト・コード・コンパイル 39
集合 (aggregate)
長さテーブル 108
終了
コンパイル 16
終了プロシージャー
構文
グローバル 448
特有の 453
コンパイラー・ユーザー出口 453
プロシージャー特有の制御ブロックの例 453
主記憶域、ソート用の 326
出力
ストリーム・ファイル用のデータ・セットの定義 233
ソート・データ・セット 331
ソート・プログラム用のルーチン 331
ソート用データ 331
ソート用の骨組みコード 335
プリプロセッサー出力の制限 57
リダイレクト 225
PLISRTA 用のデータ 335
SEQUENTIAL 257
SYSLIN 161
SYSPUNCH 161
順次アクセス
順次アクセス (続き)
REGIONAL(1) データ・セット 268
順次データ・セット 210
使用
ホスト変数としての配列、 SQL プリプロセッサー 132
条件付きコンパイル 16
条件付きサブパラメーター 211
情報交換コード 208
初期ボリューム・ラベル 211
序数エレメント・レコード、SYSADATA 465
序数タイプ・レコード、SYSADATA 464
処理ルーチン
ソート用データ
可変長レコード 338
出力 (ソート出口 E35) 334
成功かどうかの判別 330
入力 (ソート出口 E15) 332
PLISRTB 336
PLISRTC 336
PLISRTD 337
シンボル・テーブル 91
シンボル・レコード、SYSADATA 466
据え置き再始動 445
ステートメント
オフセット・アドレス 109
ネスト・レベル 107
ステップの異常終了 211
ストリーム入出力
データ・セット
アクセス 239
作成 236
レコード・フォーマット 240
データ伝送のレコード・フォーマット 218
ファイル
定義 233
PRINT ファイル 241
SYSIN および SYSPRINT ファイル 244
連続データ・セット 233
DD ステートメント 238, 240
ENVIRONMENT オプション 234
ストリーム・ファイルおよびレコード・ファイル 247, 249
ストリング
漢字ストリング定数のコンパイル 39
ストリング記述子
ストリング記述子 456, 458
ストリングの割り当て 312
ストレージ
印刷ファイルのブロック化 241
ソート・プログラム
主記憶域 326
補助記憶域 327
標準データ・セット 160
要件の削減 61
ライブラリー・データ・セット 228
リスト内のレポート 88
スラッシュ (/) 200
制御ブロック
機能専用 447
グローバル制御 450
制御文字
印刷 53, 241
紙送り機構 53, 241
ゼロ値 219
宣言 (declaration)

宣言 (declaration) (続き)
OS/390 におけるファイルの 193, 194
宣言
ホスト変数、SQL プリプロセッサー 132
相互参照テーブル
コンパイラー・リスト 108
XREF オプションの使用 107
相対バイト・アドレス (RBA) 276
相対レコード 277
相対レコード・データ・セット
ステートメントとオプション 297
ロード 299
DIRECT ファイルを使用したアクセス 301
SEQUENTIAL ファイルを使ったアクセス 300
ソース
キー
REGIONAL(1) データ・セット内の 266
プログラム
外部テキストのシフト 52
コンパイラー・リスト 107
コンパイラー・リストに組み込まれた 44
データ・セット 161
プリプロセッサー 51
リスト 87
ID 10
リスト
ロケーション 52
ソース・ステートメント・ライブラリー 162
ソース・レコード、SYSADATA 468
ソート
最大レコード長 326
準備 321
ストレージ
補助 327
メイン 326
説明 321
ソート・タイプの選択 322
ソート・フィールド 324
ソート・プログラムの呼び出し 327
データ 321
データの入出力 331
入出力ルーチンの作成 331
評価結果 330
CHKPT オプション 325
CKPT オプション 325
DYNALLOC オプション 325
E15 入力処理ルーチン 332
EQUALS オプション 325
FILSZ オプション 325
PLISRT 321
PLISRTA(x) コマンド 335, 338
RECORD ステートメント 332
RETURN ステートメント 332
SKIPREC オプション 325
SORTCKPT 331
SORTCNTL 331
SORTIN 331
SORTLIB 330
SORTOUT 331
SORTWK 327, 330
ソート用のフィールド 324
ソート用のフローチャート 332
属性テーブル 108

[タ行]

大量順次挿入 291
対話式プログラム
アテンション割り込み 45
タブ制御テーブル 243
ダミー・レコード
REGIONAL(1) データ・セット 267
VSAM 277
ダンプ
先頭の識別 432
定義、データ・セット
論理レコード長 432
DD ステートメント 432
PLIDUMP 組み込みサブルーチン 431
PLIDUMP の呼び出し 431
SNAP 432
z/OS 言語環境プログラム ダンプの生成 431
ダンプ・トレースバック・テーブル内のプログラム単位名 433
端末
出力
ストリーム・ファイルおよびレコード・ファイル 249
PRINT ファイルのフォーマット 249
PUT EDIT コマンドからの出力 249
入力
大文字と小文字 247
ストリーム・ファイルおよびレコード・ファイル 247
データのフォーマット 246
ファイル終わり 248
GET ステートメントの COPY オプション 248
QSAM ファイルの定義 247
地域
サイズ、EXEC ステートメント 160
REGION パラメーター 155
チェックポイント/再始動
据え置き再始動 445
PLICANC ステートメント 445
チェックポイント/再始動機能
活動の変更 445
再始動の要求 445
説明 443
チェックポイント・データ・セット 444
チェックポイント・レコードの要求 443
戻りコード 443
CALL PLIREST ステートメント 445
PLICKPT 組み込みサブルーチン 443
RESTART パラメーター 445
チェックポイント・データ
ソート用 331
チェックポイント・データの定義、PLICKPT 組み込みサブオプション 444
直接データ・セット 210
定義、OS/390 におけるファイル
複数のデータ・セットと 1 つのファイルの関連付け 196
定義、ファイル
特性の指定 214
ファイルのオープン
システム決定ブロック・サイズ 213
ファイルのクローズ 214
複数のデータ・セットの連結 197
複数のファイルと 1 つのデータ・セットの関連付け 196

定義、ファイル (続き)
領域データ・セット
キー 266
ENV オプション 266
ENVIRONMENT 属性 214
VSAM データ・セット 279
データ
ソート
説明 321
ソート・プログラム
PLISRT(x) コマンド 335
タイプ
Java と PL/I の同等な 369
SQL と PL/I の同等な 135
ファイル
z/OS UNIX での作成 201
z/OS UNIX での変換 199
データ指示入出力
パフォーマンスのためのコーディング 311
データ・セット
一時的な 162
区分 227
クローズ 214
索引付き
順次 210
順次 210
使用 193
条件付きサブパラメーターの特性 211
情報交換コード 208
ストリーム・ファイル 233
相対レコードの定義 299
ソース・ステートメント・ライブラリー 162
ソート
SORTWK 327
ソート・プログラム
出力データ・セット 331
ソート作業データ・セット 330
チェックポイント・データ・セット 331
入力データ・セット 331
タイプ
比較 223
PL/I レコード入出力によって使用される 223
ダンプ用の定義
論理レコード長 432
DD ステートメント 432
チェックポイント 444
直接 210
データ・セットとファイルとの関連付け 194
特性の設定 207
ファイルとの関連付け解除
システム決定ブロック・サイズ 213
ファイルの割り振り 193
複数のデータ・セットと 1 つのファイルの関連付け 196
ブロックおよびレコード 207
編成
条件付きサブパラメーター 211
タイプ 210
データ定義 (DD) ステートメント 211
ライブラリー
使用 227
情報の取り出し 231
タイプ 227
SPACE パラメーター 228
ラベル 211, 227
データ・セット (続き)
ラベルなし 211
ラベルの変更 213
領域の 263
レコード 207
レコード・フォーマット
可変長 209
固定長 208
不定長 210
レコード・フォーマットのデフォルト値 217
連続ストリーム指向データ 233
DD 名 160
OS/390 でのデータ・セットの定義 195
PL/I ファイルとの関連付け解除 197
PL/I ファイルの関連付け
ファイルのオープン 212
ファイルのクローズ 214
ENVIRONMENT 属性での特性の指定 214
REGIONAL(1)
アクセスと更新 268
作成 267
SPACE パラメーター 160
VSAM
キー 276
索引付きデータ・セット 286
大量順次挿入 291
ダミー・データ・セット 277
定義 283
データ・セット・タイプ 277
ファイルの定義 279
プログラムの実行 273
ブロック化 274
編成 274
ENVIRONMENT オプションの指定 280
VSAM オプション 282
VSAM。
パフォーマンス・オプション 282
データセット z/OS UNIX
特性の設定
DD_DDNAME 環境変数 202
データ・セット、OS/390 での
連結 197
1 つのデータ・セットと複数のファイルの関連付け 196
zFS 197
データ・セット、z/OS UNIX での
出力での拡張 202
出力の再作成 202
デフォルトの識別 198
パスの設定 201
領域数 205
領域の最大値 205
DD_DDNAME 環境変数 198
PL/I ファイルとデータ・セットの関連付け
環境変数の使用 199
関連付けされていないファイルの使用 201
OPEN ステートメントの TITLE オプションの使用 200
PL/I によるデータ・セットの検索方法 201
データ・セットの定義
特性の指定 214
ファイルのオープン
システム決定ブロック・サイズ 213
ファイルのクローズ 214
複数のデータ・セットと 1 つのファイルの関連付け 196

データ・セットの定義 (続き)
複数のデータ・セットの連結 197
複数のファイルと 1 つのデータ・セットの関連付け 196
ENVIRONMENT 属性 214
ESDS 285
出口 (E15) 入力処理ルーチン 332
出口 (E35) 出力処理ルーチン 334
トークン・レコード、SYSADATA 469
通し番号ボリューム・ラベル 211
トレースバック・テーブル
内のプログラム単位名 433
トレーラー・ラベル 211

[ナ行]

長さ、レコードの
z/OS UNIX での指定 205
名前付き定数
対静的変数 315
定義 315
入出力
カタログ式プロシージャーでの 148, 151
コンパイラー
データ・セット 161
コンパイルおよびバインド用のデータ 149, 152
ソート・データ・セット 331
ソート用の骨組みコード 333
OS/390、長い行の句読法 168
入力
ストリーム・ファイル用のデータ・セットの定義 233
ソート・データ・セット 331
ソート・プログラム用のルーチン 331
ソート用データ 331
ソート用の骨組みコード 335
リダイレクト 225
PLISRTA 用のデータ 335
SEQUENTIAL 257
入力順データ・セット
定義 285
VSAM
ステートメントとオプション 283
ESDS のロード 284
SEQUENTIAL ファイル 284

[ハ行]

配列記述子
配列記述子 457, 459
バッチ・コンパイル
OS/390 157, 159
パフォーマンス
VSAM オプション 282
パフォーマンスの向上
コンパイラー・オプションの選択
デフォルト 308
ARCH 305
GONUMBER 305
OPTIMIZE 305
PREFIX 307
REDUCE 305
RULES 306
パフォーマンスのためのコーディング
ストリングの割り当て 312
パフォーマンスの向上 (続き)
パフォーマンスのためのコーディング (続き)
名前付き定数対静的変数 315
入力専用パラメーター 311
ライブラリー・ルーチンの呼び出しの回避 316
ライブラリー・ルーチンの呼び出しのプリロード 317
ループ制御変数 312
DATA ディレクティブ入出力 311
DEFINED 対 UNION 314
GOTO ステートメント 312
PACKAGE 対ネストされた PROCEDURE 313
REDUCIBLE 関数 314
パラメーター引き渡し
引数の引き渡し 455
CMPAT(LE) 記述子 458
CMPAT(V*) 記述子 456
引数の引き渡し
記述子リストによる 455
記述子ロケーターによる 456
日付/時刻組み込み関数
パターン 314
標識変数、SQL 139
標準データ・セット 160
標準ファイル (SYSPRINT と SYSIN) 224
ファイル
クローズ 214
データ・セットとファイルとの関連付け 194
特性の設定 207
ファイルの割り振り 193
OS/390 でのデータ・セットの定義 195
ファイル・レコード、SYSADATA 463
フォーマット表記規則 xxi
複数
呼び出し
カタログ式プロシージャー 154
復帰 - 改行 (CR - LF) 206
フック
位置サブオプション 91
不定長レコード 210
負の値
ブロック・サイズ 220
レコード長 219
フラグ、コンパイル時オプションの指定 159
プリプロセス
ソース・プログラム 51
入力 107
80 バイトまでに出力を制限 57
MACRO を使った 51
%INCLUDE ステートメント 105
プリプロセッサー
包含 117
マクロ・プリプロセッサー 118
CICS オプション 144
PL/I とともに提供される 117
SQL オプション 125
SQL プリプロセッサー 123
プリプロセッサー・オプション
CCSID0 126
CODEPAGE 127
DEPRECATE 127
EMPTYDBRM 127
HOSTCOPY 128
INCONLY 128

プリプロセッサー・オプション (続き)
LINEFILE 128
LINEONLY 128
WARNDECP 129
プリント
制御ステートメント 104
プロシージャー
カタログ式、OS/390 の下での用法 147
コンパイル、バインド、および実行 - 64 ビット (IBMQCBG) 153
コンパイル、バインド、および実行 (IBMZCBG) 150
コンパイルおよびバインド - 64 ビット (IBMQCB) 152
コンパイルおよびバインド (IBMZCB) 149
コンパイルのみ - 64 ビット (IBMQC) 151
コンパイルのみ (IBMZC) 148
ブロック
およびレコード 207
サイズ
オブジェクト・モジュール 161
最大 220
指定 207
領域データ・セット 272
レコード長 220
PRINT ファイル 242
ブロック間ギャップ (IBG) 207
プロンプト
自動、指定変更 167
自動、使用 167
ページ
制御ステートメント 104
ヘッダー 40
ヘッダー・ラベル 211
変更の要約 xxiii
補助記憶域、ソート用の 327
ホスト
構造体 138
変数、SQL ステートメント内での使用 132
ホスト変数の使用、SQL プリプロセッサー 132
ボリューム通し番号
領域データ・セット 270

[マ行]

マクロ・プリプロセッサー
マクロ定義 118
マクロ・プリプロセッサーのオプション
CASE オプション 119
DBCS オプション 119
DEPRECATE オプション 119
DEPRECATENEXT オプション 120
EOLCOMM オプション 120
FIXED オプション 120
ID オプション 120
IGNORE オプション 121
INCONLY オプション 121
NAMEPREFIX オプション 121
RESCAN オプション 122
マックスルーノニフコンパイラー・オプション 56
未参照 ID 10
メッセージ
印刷フォーマット 244
コンパイラー・ユーザー出口での変更 450
コンパイラー・リスト 112
フィルター機能 451
メッセージ (続き)
ランタイム・メッセージ行番号 39
メッセージのフィルター操作 449
メッセージ・レコード、SYSADATA 464
文字
印刷制御 53, 241
紙送り制御 53, 241
文字ストリング属性テーブル 108
モジュール
オブジェクト・モジュールの作成と保管 60
戻りコード
チェックポイント/再始動ルーチン 443
PLIRETC 330

[ヤ行]

ユーザー出口
カスタマイズ
グローバル制御ブロックの構造 448
独自のコンパイラー出口の作成 450
SYSUEXIT の変更 450
機能 447
コンパイラー 447
ソート 323
容量レコード
REGIONAL(1) 267
呼び出し
カタログ式プロシージャー
複数呼び出し 154
マルチタスキング・プログラムのリンク・エディット 155
予備ファイル (spill file) 162

[ラ行]

ライブラリー
概要 210
構造化 231
コンパイルされたオブジェクト・モジュール 229
作成に必要な情報 228
システム・プロシージャー (SYS1.PROCLIB) 227
使用 227
ソース・ステートメント 162
ソース・ステートメント・ライブラリー 157
タイプ 227
ディレクトリー 228
データ・セット・ライブラリーの作成 228
用法 227
ライブラリー・ディレクトリーからの情報の取り出し 231
ライブラリー・メンバーの作成と更新 228
ロード・モジュールの配置 229
SPACE パラメーター 228
ライブラリー・ルーチンのプリロード 317
ライブラリー・ルーチンの呼び出しの回避 316
ラベル
データ・セットの 211
ランタイム
メッセージ行番号 39
OS/390 の考慮事項
自動プロンプト 167
長い入力行の句読法 168
フォーマット設定規則 166

ランタイム (続き)
OS/390 の考慮事項 (続き)
GET EDIT ステートメント 168
GET LIST ステートメントと GET DATA ステートメント 168
SKIP オプション 169
PLIXOPT の使用 165, 177
リスト
カタログ式プロシージャー 154
コンパイラー
オプション 106
集合長さテーブル 108
ステートメント・オフセット・アドレス 109
ステートメントのネスト・レベル 107
ストレージ・オフセット・リスト 110
ファイル参照テーブル 111
プリプロセッサー入力 107
見出し情報 106
メッセージ 112
ATTRIBUTE と相互参照テーブル 107
DD 名リスト 3
SOURCE オプション・プログラム 107
ステートメント・オフセット・アドレス 109
ストレージ・オフセット・リスト 110
ソース・プログラム 87
OS/390 バッチ・コンパイル 157, 162
SYSPRINT。 162
リテラル・レコード、SYSADATA 463
領域、z/OS UNIX での 205
領域データ・セット
オペレーティング・システム要件 270
ファイルの定義
キーの用法 266
領域データ・セット 265
ENVIRONMENT オプションの指定 266
DD ステートメント
アクセス 272
作成 271
REGIONAL(1) データ・セット
アクセスと更新 268
作成 267
使用 266
リンク・エディット
説明 165, 177
ループ
制御変数 312
例
PLIDUMP の呼び出し 431
レコード
コンパイラー入力の最大サイズ 161
ソート・プログラム 325
チェックポイント
データ・セット 444
ブロック化解除 208
z/OS UNIX での長さ 205
レコード長
値 219
指定 207
領域データ・セット 263
レコード入出力
形式 217
データ・セット
アクセス 258
作成 257
レコード入出力 (続き)
データ・セット (続き)
タイプ 223
連続データ・セット 259
データ伝送 251
レコード・フォーマット 252
ENVIRONMENT オプション 252
レコードのブロック化解除 208
レコード・フォーマット
オプション 235
可変長レコード 209
固定長レコード 208
指定する 252
ストリーム入出力 240
タイプ 208
不定長レコード 210
連結
外部参照 195
データ・セット 197
連続データ・セット
ストリーム指向データ伝送
データ・セットの作成 236
データ・セットへのアクセス 239
ファイルの定義 233
ENVIRONMENT オプションの指定 234
PRINT ファイルの用法 241
SYSIN ファイルおよび SYSPRINT ファイルの使用方法 244
端末からの入力 245
端末からの入力の制御
大文字と小文字 247
条件のフォーマット 245
ストリーム・ファイルおよびレコード・ファイル 247
データのフォーマット 246
ファイルの終わり 248
GET ステートメントの COPY オプション 248
QSAM ファイルの定義 247
端末への出力 249
端末への出力の制御
条件 249
ストリーム・ファイルおよびレコード・ファイル 249
PRINT ファイルのフォーマット 249
PUT EDIT コマンドの出力 249
定義と用法 233
レコード単位データ伝送 (record-oriented data transmission)
使用できるステートメントとオプション 251
データ・セットのアクセスと更新 258
データ・セットの作成 257
ファイルの定義 252
ENVIRONMENT オプションの指定 252
レコード単位入出力 251
連絡域、SQL 129
論理否定 59
論理和 62

[ワ行]

割り込み
主要な説明 441
対話式システムでのアテンション割り込み 45
デバッグ・ツール 442

割り込み (続き)
ATTENTION ON ユニット 442

[数字]

31 および 64 の相互運用性 373

A

ACCT EXEC ステートメント・パラメーター 155
AGGREGATE コンパイラー・オプション 9
ALIGNED コンパイラー・サブオプション 24
ALL オプション
フック位置サブオプション 91
ALLOCATE ステートメント 108
AMP パラメーター 273
ANS
コンパイラー・サブオプション 25
APPEND オプション、z/OS UNIX での 202
ARCH コンパイラー・オプション 9, 305
ASCII
コンパイラー・サブオプション
説明 25
ASSERT コンパイラー・オプション 10
ASSIGNABLE コンパイラー・サブオプション 25
ATTENTION ON ユニット 442
ATTRIBUTES オプション 10
automatic
restart
checkpoint/restart facility 443

B

BACKREG コンパイラー・オプション 11
BIFPREC コンパイラー・オプション 11
BIN1ARG コンパイラー・サブオプション 25
BKWD オプション 214, 280
BLANK コンパイラー・オプション 12
BLKOFF コンパイラー・オプション 12
BLKSIZE
サブパラメーター 212
ENVIRONMENT
レコード入出力用の 219
DCB サブパラメーターとの比較 217
ENVIRONMENT オプションの
ストリーム入出力用の 234
BRACKETS 12
BRACKETS コンパイラー・オプション 12
BUFFERS オプション
ストリーム入出力用の 234
BUFND オプション 280
BUFNI オプション 281
BUFSIZE オプション、z/OS UNIX での 203
BUFSP オプション 281
BYADDR
説明 308
パフォーマンスへの影響 308
DEFAULT オプションでの用法 26
BYVALUE
説明 308
DEFAULT オプションでの用法 26
effect on performance 309

C

C ルーチン
FETCH 174
CASE 119
CASERULES コンパイラー・オプション 13
cataloged procedure
description of 147
CHECK コンパイラー・オプション 14
checkpoint/restart facility
PLICKPT built-in subroutine 443
CHKPT ソート・オプション 325
CICS
サポート 143
プリプロセッサー・オプション 144
CKPT ソート・オプション 325
CMPAT コンパイラー・オプション 14
COBOL
マップ構造 108
CODE サブパラメーター 212
CODEPAGE コンパイラー・オプション 15
COMPILE コンパイラー・オプション 16
compiler
listing
return codes 113
COND EXEC ステートメント・パラメーター 155
CONNECTED コンパイラー・サブオプション
説明 26
パフォーマンスへの影響 309
CONSECUTIVE
ENVIRONMENT オプションの 234, 253
COPY オプション 248
COPYRIGHT オプション 438
COPYRIGHT コンパイラー・オプション 17
CSECT コンパイラー・オプション 17
CSECTCUT コンパイラー・オプション 18
CTLASA オプションと CTL360 オプション
ENVIRONMENT オプション
連続データ・セット用の 254
SCALARVARYING 222
CURRENCY コンパイラー・オプション 18
CYLOFL サブパラメーター
DCB パラメーター 212

D

data set
data set control block (DSCB) 211
DBCS 119
DBCS ID コンパイル 39
DBCS コンパイラー・オプション 18
DBRMLIB コンパイラー・オプション 19
DCB サブパラメーター
主な説明 211
カタログ式プロシージャー内の一時変更 156
同等の ENVIRONMENT オプション 217
領域データ・セット 272
DD (データ定義) 情報、z/OS UNIX での 199
DD コンパイラー・オプション 19
DD 情報、z/OS UNIX での
TITLE ステートメント 200
DD ステートメント
カタログ式プロシージャーの変更 155, 156
カタログ式プロシージャーへの追加 156

DD ステートメント (続き)
チェックポイント/再始動 443
標準データ・セット
出力 (SYSLIN、SYSPUNCH) 161
入力 (SYSIN) 161
ライブラリーの作成 228
領域データ・セット 271
OS/390 バッチ・コンパイル 160
%INCLUDE 105
DD 名 (ddname)
標準データ・セット 160
%INCLUDE 105
DD_DDNAME 環境変数
タイプ 206
の下の代替 ddname z/OS UNIX 200
APPEND 202
DELAY 204
DELIMIT 204
LRECL 204
LRMSKIP 204
PROMPT 204
PUTPAGE 204
RECCOUNT 205
RECSIZE 205
SAMELINE 205
SKIP0 206
z/OS UNIX での特性の指定 202
DDSQL コンパイラー・オプション 20
DECOMP 23
DECOMP コンパイラー・オプション 23
DEFAULT コンパイラー・オプション
サブオプション
ALIGNED 24
ASCII または EBCDIC 25
ASSIGNABLE または NONASSIGNABLE 25
BIN1ARG または NOBIN1ARG 25
BYADDR または BYVALUE 26
CONNECTED または NONCONNECTED 26
DESCLIST または DESCLOCATOR 26
DESCRIPTOR または NODESCRIPTOR 26
DUMMY 27
E 27
EVENDEC または NOEVENDEC 27
HEXADEC 27
IBM または ANS 25
INITFILL または NOINITFILL 28
INLINE または NOINLINE 28
LAXQUAL または NOLAXQUAL 28
LINKAGE 28
LOWERINC | UPPERINC 29
NATIVE または NONNATIVE 29
NATIVEADDR または NONNATIVEADDR 29
NULLINIT 29
NULLSTRADDR または NONULLSTRADDR 30
NULLSTRPTR 30
NULLSYS または NULL370 30
ORDER または REORDER 30
ORDINAL(MIN | MAX) 30
OVERLAP | NOOVERLAP 30
PADDING または NOPADDING 30
PSEUDODUMMY または NOPSEUDODUMMY 31
RECURSIVE または NONRECURSIVE 31
RETCODE 31
RETURNS 31
DEFAULT コンパイラー・オプション (続き)
サブオプション (続き)
SHORT 31
説明および構文 23
DEFINED
対 UNION 314
DELAY オプション、z/OS UNIX での
説明 204
DELIMIT オプション、z/OS UNIX での
説明 204
DEPRECATE 119
DEPRECATE コンパイラー・オプション 32
DEPRECATENEXT 120
DEPRECATENEXT コンパイラー・オプション 33
DESCLIST コンパイラー・サブオプション 26
DESCLOCATOR コンパイラー・サブオプション 26
DESCRIPTOR コンパイラー・オプション
パフォーマンスへの影響 309
DESCRIPTOR コンパイラー・サブオプション
説明 26
DFSORT 321
DIRECT ファイル
RRDS
データ・セットへのアクセス 301
VSAM での索引付き ESDS
データ・セットの更新 291
データ・セットへのアクセス 290
DISP パラメーター
データ・セットを削除する 227
連続データ・セット 259
DISPLAY コンパイラー・オプション 34
DLL
リンクの考慮事項およびサイド・デック 165, 177
RENT コンパイラー・オプションおよびフェッチ 171
DLL コンパイラー・オプション 34
DLLINIT コンパイラー・オプション 34
DSA
各ブロックごとの PLIDUMP 組み込みサブルーチンに保存された 433
DSCB (data set control block) 211
DSCB (データ・セット制御ブロック) 228
DSNAME パラメーター
連続データ・セット用の 259
DSORG サブパラメーター 212
DUMMY コンパイラー・サブオプション 27
DYNALLOC ソート・オプション 325

E

E コンパイラー・サブオプション 27
E コンパイラー・メッセージ 112
E15 入力処理ルーチン 332
E35 出力処理ルーチン 334
EBCDIC
コンパイラー・サブオプション 25
EBCDIC (拡張 2 進化 10 進コード) 208
embedded
CICS statements 144
ENDFILE
OS/390 の下での 169
Enterprise PL/I ライブラリー
Enterprise PL/I for z/OS ライブラリー xix
Language Environment ライブラリー xx
ENVIRONMENT オプション

ENVIRONMENT オプション (続き)
同等の DCB サブパラメーター 217
編成オプション 217
領域データ・セット 266
レコード・フォーマット・オプション 235
BLKSIZE オプション
DCB サブパラメーターとの比較 217
CONSECUTIVE 234, 253
CTLASA および CTL360
DCB サブパラメーターとの比較 217
GRAPHIC オプション 236
KEYLENGTH オプション
DCB サブパラメーターとの比較 217
LEAVE および REREAD 256
RECSIZE オプション
使用量 235
レコード・フォーマット 236
DCB サブパラメーターとの比較 217
VSAM
BKWD オプション 280
BUFND オプション 280
BUFNI オプション 281
BUFSP オプション 281
GENKEY オプション 281
PASSWORD オプション 281
REUSE オプション 281
SKIP オプション 282
VSAM オプション 282
ENVIRONMENT 属性
リスト 214
z/OS UNIX での特性の指定
BUFSIZE 203
ENVIRONMENT の F オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の FB オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の FBS オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の FS オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の REGIONAL オプション 266
ENVIRONMENT の U オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の V オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の VB オプション
ストリーム入出力用の 234
レコード入出力用の 218
ENVIRONMENT の VBS オプション
ストリーム入出力用の 234
ENVIRONMENT の VS オプション
ストリーム入出力用の 234
EOLCOMM (E) 120
EQUALS ソート・オプション 325
ESDS (入力順データ・セット)
固有キー代替索引パス 292
定義 285
非固有キー代替索引パス 293
ESDS (入力順データ・セット) (続き)
VSAM
ステートメントとオプション 283
ロード 284
EVENDEC コンパイラー・サブオプション 27
EXEC SQL ステートメント 123
EXEC ステートメント
オプションの指定 163
オプション・リストの最大長 163
概要 160
カタログ式プロシージャーの変更 155
コンパイラー 160
最小領域サイズ 160
OS/390 バッチ・コンパイル 157, 160
PARM パラメーター 163
EXIT コンパイラー・オプション 35
export コマンド 202
EXPORTALL 35
EXPORTALL コンパイラー・オプション 35
EXTERNAL 属性 108
EXTRN コンパイラー・オプション 35

F

F フォーマット・レコード 208
FB フォーマット・レコード 208
FETCH
アセンブラー・ルーチン 174, 179
Enterprise PL/I ルーチン 170, 178
OS/390 C ルーチン 174
PL/I MAIN ルーチン 173, 178
FILE 属性 108
FILEREF コンパイラー・オプション 35
filespec 202
FILLERS、タブ制御テーブルの 243
FILSZ ソート・オプション 325
FIXED
TYPE オプション z/OS UNIX 207
FLAG コンパイラー・オプション 36
FLOAT オプション 36
FLOATINMATH コンパイラー・オプション 38
FUNC サブパラメーター
使用量 212

G

GENKEY オプション
キーの分類 221
使用量 214
VSAM 280
GET DATA ステートメント 168
GET EDIT ステートメント 168
GET LIST ステートメント 168
GOFF コンパイラー・オプション 38
GONUMBER コンパイラー・オプション
定義 39
GOTO ステートメント 312
GRAPHIC オプション
コンパイラー 39
ストリーム入出力 234
ENVIRONMENT の 214, 236

H

HEADER コンパイラー・オプション 40
HEXADEC コンパイラー・サブオプション 27

I

I コンパイラー・メッセージ 112
IBM コンパイラー・サブオプション 25
IBMQC カタログ式プロシージャー 151
IBMQCB カタログ式プロシージャー 152
IBMQCBG カタログ式プロシージャー 153
IBMUEXIT コンパイラー出口 449
IBMZC カタログ式プロシージャー 148
IBMZCB カタログ式プロシージャー 149
IBMZCBG カタログ式プロシージャー 150
ID
    参照されない 10
    ソース・プログラム 10
IEC225I 212, 271
IGNORE 121
IGNORE オプション 40
ILC
    リンケージの考慮事項 347
    C との
        構造化データ・タイプ 342
        出力の共用 349
        ストリング・パラメーター・タイプの一致 345
        データ・タイプ 341
        入力の共有 349
        パラメーターの一致 344
        ファイル・タイプ 343
        ATTACH ステートメントの使用 349
        C 標準ストリームのリダイレクト 350
        ENTRY を戻す関数 346
        enum データ・タイプ 342
ILC でのリンケージ考慮事項 347
INCAFTER コンパイラー・オプション 41
INCDIR コンパイラー・オプション 41
INCLUDE コンパイラー・オプション 42
INCLUDE ステートメント
    コンパイラー 105
INCONLY 121
INDEXAREA オプション 214
INITFILL コンパイラー・サブオプション 28
INLINE コンパイラー・サブオプション 28
INSOURCE オプション 44
INTERNAL 属性 108
INTERRUPT コンパイラー・オプション 45

J

Java 352, 353, 355–364, 367, 368
JAVA 351
Java コード、コンパイル 353, 357, 361, 364
Java コード、作成 352, 356, 360, 364
JCL (job control language)
    improving efficiency 147
JCL (ジョブ制御言語)
    コンパイル中の使用 159
jni
    JNI サンプル・プログラム 352, 356, 360, 364

K

KEYLEN サブパラメーター 212
KEYLENGTH オプション 214, 222
KEYLOC オプション
    使用量 214
KEYTO オプション
    VSAM の下での 284
KSDS (キー順データ・セット)
    更新 290
    固有キー代替索引パス 293
    定義とロード 288
    VSAM
        ロード 288
        DIRECT ファイル 290
        SEQUENTIAL ファイル 289

L

LANGLVL コンパイラー・オプション 47
Language Environment ライブラリー xx
LAXQUAL コンパイラー・サブオプション 28
LEAVE および REREAD オプション
    ENVIRONMENT オプション
        連続データ・セット用の 256
LIMCT サブパラメーター 212, 272
LIMITS コンパイラー・オプション 48
LINE オプション 235, 241
LINEDIR コンパイラー・オプション 49
LINESIZE オプション
    タブ制御テーブルの 243
    OPEN ステートメント 235
LINKAGE コンパイラー・サブオプション
    構文 28
    パフォーマンスへの影響 310
LIST コンパイラー・オプション 50
listing
    compiler
        return codes 113
LISTVIEW コンパイラー・オプション 50
LOWERINC コンパイラー・サブオプション 29
LP 51
LP コンパイラー・オプション 51
LRECL オプション、z/OS UNIX での 204
LRECL サブパラメーター 207, 212
LRMSKIP オプション、z/OS UNIX での 204

M

MACRO オプション 51
MAP コンパイラー・オプション 52
MARGINI コンパイラー・オプション 52
MARGINS コンパイラー・オプション 52
MAXBRANCH コンパイラー・オプション 53
MAXGEN コンパイラー・オプション 54
MAXINIT コンパイラー・オプション 54
MAXMEM コンパイラー・オプション 54
MAXMSG コンパイラー・オプション 55
MAXSTATIC コンパイラー・オプション 56
MAXSTMT コンパイラー・オプション 56
MAXTEMP コンパイラー・オプション 57
MDECK コンパイラー・オプション
    説明 57

MODE サブパラメーター
使用量 212
MSGSUMMARY コンパイラー・オプション 57

N

NAME コンパイラー・オプション 58
NAMEPREFIX 121
NAMES コンパイラー・オプション 58
NATIVE コンパイラー・サブオプション
説明 29
NATIVEADDR コンパイラー・サブオプション 29
NATLANG コンパイラー・オプション 59
NEST オプション 59
NOBIN1ARG コンパイラー・サブオプション 25
NODESCRIPTOR コンパイラー・サブオプション 26
NOEQUALS ソート・オプション 325
NOEVENDEC コンパイラー・サブオプション 27
NOIGNORE 121
NOINITFILL コンパイラー・サブオプション 28
NOINLINE コンパイラー・サブオプション 28
NOINTERRUPT コンパイラー・オプション 45
NOLAXQUAL コンパイラー・サブオプション 28
NOMAP オプション 108
NOMARGINS コンパイラー・オプション 52
NONASSIGNABLE コンパイラー・サブオプション 25
NONCONNECTED コンパイラー・サブオプション 26
NONE、フック位置サブオプション 91
NONNATIVE コンパイラー・サブオプション 29
NONNATIVEADDR コンパイラー・サブオプション 29
NONRECURSIVE コンパイラー・サブオプション 31
NONULLSTRADDR コンパイラー・サブオプション 30
NOOVERLAP コンパイラー・サブオプション
パフォーマンスへの影響 310
NOPADDING コンパイラー・サブオプション 30
NOPSEUDODUMMY コンパイラー・サブオプション 31
NOSYNTAX コンパイラー・オプション 89
NOT コンパイラー・オプション 59
note ステートメント 112
NTM サブパラメーター
使用量 212
NULL370 コンパイラー・サブオプション 30
NULLDATE 59
NULLDATE コンパイラー・オプション 59
NULLINIT コンパイラー・サブオプション 29
NULLSTRADDR コンパイラー・サブオプション 30
NULLSTRPTR コンパイラー・サブオプション 30
NULLSYS コンパイラー・サブオプション 30

O

OBJECT コンパイラー・オプション
定義 60
OFFSET コンパイラー・オプション 60
OFFSETSIZE コンパイラー・オプション 60
ONSNAP コンパイラー・オプション 60
OPEN ステートメント
PL/I ライブラリーのサブルーチン 212
TITLE オプション 212, 225
OPTCD サブパラメーター 211, 212
OPTIMIZE オプション 61
OPTIMIZE コンパイラー・オプション 305
OPTIONS (option) 62
OR コンパイラー・オプション 62
ORDER コンパイラー・サブオプション
説明 30
パフォーマンスへの影響 310
ORDINAL コンパイラー・サブオプション 30
ORGANIZATION オプション
使用量 214
OS/390
一般コンパイル 157
長い入力行 168
バッチ・コンパイル
一時作業ファイル (SYSUT1) 162
オプションの指定 163
ソース・ステートメント・ライブラリー (SYSLIB) 162
リスト (SYSPRINT) 162
DD ステートメント 160
EXEC ステートメント 160, 163
OVERLAP コンパイラー・サブオプション 30

P

PACKAGE 対ネストされた PROCEDURE 313
PADDING コンパイラー・サブオプション 30
PAGE オプション 235
PAGELENGTH、タブ制御テーブルの 243
PAGESIZE、タブ制御テーブルの 243
PARM パラメーター
オプションの指定 163
カタログ式プロシージャー 155
PASSWORD オプション 281
PL/I
コンパイラー
ユーザー出口のプロシージャー 447
ファイル
z/OS UNIX でデータ・セットと関連付ける 198
PL/I MAIN ルーチン
FETCH 173, 178
PL/I コード、コンパイル 355, 359, 363, 367
PL/I コード、作成 353, 358, 362, 364
PL/I コード、リンク 355, 359, 363, 367
PL/I 動的割り振り
データ・セットの割り振り 193
ファイルの定義
ストリーム・ファイル 233
QSAM ファイル 247
REGIONAL(1) データ・セット 265
VSAM ファイル 273
z/OS UNIX でのデータ・セットとファイルの関連付け 198
z/OS での zFS ファイルへのアクセス 197
PLICANC ステートメント、およびチェックポイント/要求 445
PLICKPT built-in subroutine 443
PLIDUMP built-in subroutine
syntax of 431
PLIDUMP 組み込みサブルーチン
各ブロックごとの DSA に保存された 433
出力
変数の検出 433
ダンプ・トレースバック・テーブル内のプログラム単位名 433
変数
AUTOMATIC 変数の検出 433

PLIDUMP 組み込みサブルーチン (続き)
変数 (続き)
CONTROLLED 変数の検出 435
PLIDUMP 出力内での検出 433
STATIC 変数の検出 434
保存されたオプション・ストリング 439
保存されたロード・モジュール のタイム・スタンプ 438
ユーザー ID 432
H オプション 432
z/OS 言語環境プログラム ダンプを生成するための呼び出し 431
PLIDUMP に保存されたオプション・ストリング 439
PLIREST ステートメント 445
PLIRETC 組み込みサブルーチン
ソートでの戻りコード 330
PLISAXA 377, 378
PLISAXB 377, 378
PLISAXC 401, 402
PLISAXD 401, 402
PLISRTA インターフェース 335
PLISRTB インターフェース 336
PLISRTC インターフェース 336
PLISRTD インターフェース 337
PLITABS 外部構造
制御セクション 244
宣言 (declaration) 166
PLIXOPT 変数 165, 177
PP コンパイラー・オプション 62
PPCICS コンパイラー・オプション 63
PPINCLUDE コンパイラー・オプション 64
PPLIST コンパイラー・オプション 64
PPMACRO コンパイラー・オプション 64
PPSQL コンパイラー・オプション 65
PPTRACE コンパイラー・オプション 65
PRECTYPE コンパイラー・オプション 65
PREFIX コンパイラー・オプション
デフォルトのサブオプションの使用 307
PRINT ファイル
出力に句読点を付ける
レコード入出力 259
フォーマット設定規則 166
PROCEED コンパイラー・オプション 66
PROCESS ステートメント
オプション・デフォルトの指定変更 163
説明 103
PROMPT オプション、z/OS UNIX での 204
PRTSP サブパラメーター
使用量 212
PSEUDODUMMY コンパイラー・サブオプション 31
PUT EDIT コマンド 249
PUTPAGE オプション、z/OS UNIX での 204

Q

QUOTE コンパイラー・オプション 67

R

REAL 属性 108
RECCOUNT オプション、z/OS UNIX での 205
RECFM サブパラメーター
使用量 212
データ・セットの編成における 212
RECORD ステートメント 325
RECSIZE オプション
ストリーム入出力用の 234, 235
定義 218
デフォルト値 236
連続データ・セット 235
z/OS UNIX での記述 205
RECURSIVE コンパイラー・サブオプション 31
REDUCE コンパイラー・オプション
パフォーマンスへの影響 305
REDUCIBLE 関数 314
REGIONAL(1) データ・セット
ファイルの定義
領域データ・セット 265
RENT コンパイラー・オプション 68
REORDER コンパイラー・サブオプション
説明 30
パフォーマンスへの影響 310
RESCAN 122
RESEXP コンパイラー・オプション 69
RESPECT コンパイラー・オプション 69
RETCODE コンパイラー・サブオプション 31
return codes
in compiler listing 113
RETURNS コンパイラー・サブオプション 31, 310
REUSE オプション 214, 281
RRDS (相対レコード・データ・セット)
更新 301
定義 299
VSAM
ロード 299
DIRECT ファイル 301
SEQUENTIAL ファイル 300
VSAM でのロード 299
RTCHECK コンパイラー・オプション 69
RULES コンパイラー・オプション
パフォーマンスへの影響 306

S

S コンパイラー・メッセージ 112
SAMELINE オプション、z/OS UNIX での 205
SAX パーサー 377, 401
SCALARVARYING オプション 222
SEMANTIC コンパイラー・オプション 86
SEQUENTIAL ファイル
RRDS、アクセス・データ・セット 300
VSAM での ESDS
更新 286
定義とロード 285
VSAM での索引付き ESDS
データ・セットへのアクセス 289
SERVICE コンパイラー・オプション 87
SHORT コンパイラー・サブオプション 31
SKIP オプション
ストリーム入出力における 235
OS/390 の下での 169
SKIP0 オプション、z/OS UNIX での 206
SKIPREC ソート・オプション 325
SOURCE コンパイラー・オプション 87
SPACE パラメーター
標準データ・セット 160
ライブラリー 228
SPILL コンパイラー・オプション 87

SQL プリプロセッサー
オプション 125
記述子域 130
標識変数の使用 139
ホスト構造体の使用 138
ホスト変数の使用 132
連絡域 129
EXEC SQL ステートメント 123
IBMUEXIT の使用 143
SQL プリプロセッサー・オプション 126–129
SQLCA 129
SQLDA 130
STACK サブパラメーター
使用量 212
STATIC コンパイラー・オプション 87
STDSYS コンパイラー・オプション 88
STMT コンパイラー・オプション 88
STMT サブオプション、テストの 91
STORAGE コンパイラー・オプション 88
STREAM 属性 233
STRINGOFGRAPHIC コンパイラー・オプション 89
SUB 制御文字 208
SYNTAX オプション 89
SYS1.PROCLIB (システム・プロシージャー・ライブラリー) 227
SYSADATA 情報、オプション・レコード 462
SYSADATA 情報、カウンター・レコード 463
SYSADATA 情報、構文情報 468
SYSADATA 情報、構文レコード 469
SYSADATA 情報、サマリー・レコード 462
SYSADATA 情報、序数エレメント・レコード 465
SYSADATA 情報、序数タイプ・レコード 464
SYSADATA 情報、シンボル情報 464
SYSADATA 情報、シンボル・レコード 466
SYSADATA 情報、ソース・レコード 468
SYSADATA 情報、トークン・レコード 469
SYSADATA 情報、ファイル・レコード 463
SYSADATA 情報、メッセージ・レコード 464
SYSADATA 情報、リテラル・レコード 463
SYSADATA 情報の概要 461
SYSCHK デフォルト 443, 444
SYSIN
C との共用 349
SYSIN および SYSPRINT ファイル 244
syslib
プリプロセス 162
%INCLUDE 105
SYSLIN 161
SYSOUT 330
SYSPARM コンパイラー・オプション 89
SYSPRINT。
エンクレーブ間での共用 169
および z/OS UNIX 224
コンパイラー・リストの書き込み先 162
年上と共有 PL/I 170
必須 DD ステートメント 160
C との共用 349, 350
DD オプションでの指定 19
MSGFILE(SYSPRINT) の使用 170
PUT ステートメントでの使用 244
STDSYS オプションの作用 88
SYSPUNCH 161
SYSUT1 コンパイラー・データ・セット 162

T

TERMINAL コンパイラー・オプション 91
TEST コンパイラー・オプション
定義 91
TIME パラメーター 155
TIMESTAMP
PLIDUMP 内の保存されたロード・モジュール のタイム・スタンプ 438
TITLE オプション
使用 212, 225
標準 SYSPRINT ファイルの関係付け 170
z/OS UNIX での記述 200
TITLE オプション、OS/390 での
文字ストリング値 195
TITLE オプション、z/OS UNIX での
データ・セットに関連付けられていないファイルの使用 201

U

U コンパイラー・メッセージ 112
U フォーマット 210
UNDEFINEDFILE 条件
BLKSIZE エラー 220
OPEN での行サイズの矛盾 242
z/OS UNIX でのファイル・オープン時に発生する 207
UNDEFINEDFILE 条件、OS/390 での
DD ステートメント・エラー 195
UNDEFINEDFILE 条件、z/OS UNIX での
データ・セットに関連付けられていないファイルの使用 207
UNIT パラメーター
連続データ・セット 259
UNROLL コンパイラー・オプション 94
UPPERINC コンパイラー・サブオプション 29
USAGE コンパイラー・オプション 94

V

VB フォーマット・レコード 209
volume serial number
direct access volumes 211
VOLUME パラメーター
連続データ・セット 259
VSAM (仮想記憶アクセス方式)
索引付きデータ・セット
ロード・ステートメントとオプション 286
相対レコード・データ・セット 299
大量順次挿入 291
データ・セット
キー 276
キー順および索引付き入力順 286
使用 273
相対レコード 297
代替索引 292
代替索引パス 282
タイプの選択 277
ダミー・データ・セット 277
定義 273, 283
入力順 283
パフォーマンス・オプション 282
ファイルの定義 279

VSAM (仮想記憶アクセス方式) (続き)
データ・セット (続き)
プログラムの実行 273
ブロック化 274
編成 274
ENVIRONMENT オプションの指定 280
ファイルの定義
パフォーマンス・オプション 282
ENV オプション 280
VSAM オプション 282
VSAM での KSDS
VSAM での索引付き ESDS
データ・セットの更新 291
データ・セットへのアクセス 289, 290
VTOC 211

W

W コンパイラー・メッセージ 112
WIDECHAR コンパイラー・オプション 96
WINDOW コンパイラー・オプション 97
WRITABLE コンパイラー・オプション 97

X

XINFO コンパイラー・オプション 98
XML
SAX パーサーでのサポート 377, 401
XML コンパイラー・オプション 45, 101
XREF コンパイラー・オプション 102

Z

z/OS UNIX
環境変数の設定 224
コンパイル 157
コンパイル時オプション
指定 158
コンパイル時オプションの指定
コマンド行 158
フラグの使用 159
DD_DDNAME 環境変数 202
export コマンド 202
z/OS UNIX での TYPE オプション 206
z/OS UNIX でのデータ定義 (DD) 情報 199
z/OS UNIX での代替 dd 名、TITLE オプションの 200

[特殊文字]

*PROCESS、内のオプションの指定 103
/ (スラッシュ) 200
% ステートメント 104
%INCLUDE 120
%INCLUDE ステートメント
制御ステートメント 104
ソース・ステートメント・ライブラリー 162
%NOPRINT
制御ステートメント 104
%NOPRINT ステートメント 104, 121
%PAGE ステートメント 104
%POP ステートメント 104
%PRINT ステートメント 104
%PROCESS、内のオプションの指定 103
%PUSH ステートメント 105
%SKIP
制御ステートメント 104
%SKIP ステートメント 104
++INCLUDE 120

プログラム番号: 5655-PL6

GI13-5620-00