IBM COBOL for Linux on x86 1.1

Programming Guide

SC28-3118-00

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 587.

First edition

This edition applies to Version 1.1 of IBM® COBOL for Linux® on x86 (program number 5737-L11) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

You can view or download softcopy publications free of charge in the COBOL for Linux on x86 library.

© Copyright International Business Machines Corporation 2021.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables ..... **xv**

Preface ..... **xix**
About this information xix
How this information will help you xix
Abbreviated terms xix
How to read syntax diagrams xx
How examples are shown xxi
Related information xxi
How to send your comments xxi
Accessibility xxi

Part 1. Coding your program **1**

Chapter 1. Structuring your program 3
Identifying a program 3
Identifying a program as recursive 4
Marking a program as callable by containing programs 4
Setting a program to an initial state 4
Changing the header of a source listing 4
Describing the computing environment 5
Example: FILE-CONTROL paragraph 5
Specifying the collating sequence 6
Defining symbolic characters 7
Defining a user-defined class 8
Identifying files to the operating system (ASSIGN) 8
Describing the data 9
Using data in input and output operations 9
Comparison of WORKING-STORAGE and LOCAL-STORAGE 11
Using data from another program 12
Processing the data 13
How logic is divided in the PROCEDURE DIVISION 14
Declaratives 17

Chapter 2. Using data 19
Using variables, structures, literals, and constants 19
Using variables 19
Using data items and group items 20
Using literals 21
Using constants 22
Using figurative constants 22
Assigning values to data items 23
Examples: initializing data items 24
Initializing a structure (INITIALIZE) 27
Assigning values to elementary data items (MOVE) 28
Assigning values to group data items (MOVE) 29
Assigning arithmetic results (MOVE or COMPUTE) 30
Assigning input from a screen or file (ACCEPT) 30
Displaying values on a screen or in a file (DISPLAY) 31
Using intrinsic functions (built-in functions) 32

Using tables (arrays) and pointers....33

Chapter 3. Working with numbers and arithmetic....35
Defining numeric data....35
Displaying numeric data....37
Controlling how numeric data is stored....38
Formats for numeric data....39
Examples: numeric data and internal representation....42
Data format conversions....46
Conversions and precision....47
Sign representation of zoned and packed-decimal data....47
Checking for incompatible data (numeric class test)....48
Performing arithmetic....48
Using COMPUTE and other arithmetic statements....49
Using arithmetic expressions....50
Using numeric intrinsic functions....50
Examples: numeric intrinsic functions....51
Fixed-point contrasted with floating-point arithmetic....53
Examples: fixed-point and floating-point evaluations....55
Using currency signs....56
Example: multiple currency signs....56

Chapter 4. Handling tables....59
Defining a table (OCCURS)....59
Nesting tables....61
Example: subscripting....62
Example: indexing....62
Referring to an item in a table....62
Subscripting....63
Indexing....64
Putting values into a table....65
Loading a table dynamically....65
Initializing a table (INITIALIZE)....65
Assigning values when you define a table (VALUE)....66
Example: PERFORM and subscripting....68
Example: PERFORM and indexing....69
Creating variable-length tables (DEPENDING ON)....70
Loading a variable-length table....72
Assigning values to a variable-length table....72
Complex OCCURS DEPENDING ON....73
Example: complex ODO....73
Effects of change in ODO object value....74
Searching a table....76
Doing a serial search (SEARCH)....77
Doing a binary search (SEARCH ALL)....78
Sorting a table....79
Processing table items using intrinsic functions....79
Example: processing tables using intrinsic functions....80

Chapter 5. Selecting and repeating program actions....81
Selecting program actions....81
Coding a choice of actions....81
Coding conditional expressions....85
Repeating program actions....88
Choosing inline or out-of-line PERFORM....89
Coding a loop....90
Looping through a table....90
Executing multiple paragraphs or sections....91

Chapter 6. Handling strings 93
Joining data items (STRING) 93
Example: STRING statement 94
Splitting data items (UNSTRING) 95
Example: UNSTRING statement 96
Manipulating null-terminated strings 98
Example: null-terminated strings 98
Referring to substrings of data items 99
Reference modifiers 100
Example: arithmetic expressions as reference modifiers 101
Example: intrinsic functions as reference modifiers 102
Tallying and replacing data items (INSPECT) 102
Examples: INSPECT statement 103
Converting data items (intrinsic functions) 104
Changing case (UPPER-CASE, LOWER-CASE) 104
Transforming to reverse order (REVERSE) 105
Converting to numbers (NUMVAL, NUMVAL-C) 105
Converting from one code page to another 106
Evaluating data items (intrinsic functions) 106
Evaluating single characters for collating sequence 107
Finding the largest or smallest data item 107
Finding the length of data items 109
Finding the date of compilation 110

Chapter 7. Processing files 111
File concepts and terminology 111
Identifying files 112
Identifying Db2 files 114
Identifying SFS files 115
Precedence of file-system determination 116
File systems 116
Db2 file system 117
QSAM file system 118
RSD file system 119
SdU file system 119
SFS file system 119
STL file system 120
Specifying a file organization and access mode 120
File organization and access mode 121
Generation data groups 123
Creating generation data groups 124
Using generation data groups 126
Name format of generation files 127
Insertion and wrapping of generation files 128
Limit processing of generation data groups 130
Concatenating files 131
Opening optional files 132
Setting up a field for file status 132
Describing the structure of a file in detail 132
Coding input and output statements for files 133
Example: COBOL coding for files 133
File position indicator 135
Opening a file 135
Reading records from a file 137
Statements used when writing records to a file 138
Adding records to a file 139
Replacing records in a file 140

Deleting records from a file....140
PROCEDURE DIVISION statements used to update files....140
Using Db2 files....142
Using Db2 files and SQL statements in the same program....143
Using QSAM files....144
Using SFS files....145
Example: accessing SFS files....146
Improving SFS performance....147

Chapter 8. Sorting and merging files....151
Sort and merge process....151
Describing the sort or merge file....152
Describing the input to sorting or merging....152
Example: describing sort and input files for SORT....153
Coding the input procedure....153
Describing the output from sorting or merging....154
Coding the output procedure....155
Restrictions on input and output procedures....155
Requesting the sort or merge....156
Setting sort or merge criteria....156
Choosing alternate collating sequences....157
Example: sorting with input and output procedures....157
Determining whether the sort or merge was successful....158
Sort and merge error numbers....159
Stopping a sort or merge operation prematurely....162

Chapter 9. Handling errors....163
Handling errors in joining and splitting strings....163
Handling errors in arithmetic operations....164
Example: checking for division by zero....164
Handling errors in input and output operations....164
Using the end-of-file condition (AT END)....166
Coding ERROR declaratives....166
Using file status keys....166
Using file system status codes....168
Coding INVALID KEY phrases....170
Handling errors when calling programs....170

Part 2. Enabling programs for international environments....173

Chapter 10. Processing data in an international environment....175
Unicode and the encoding of language characters....176
Using national data (Unicode) in COBOL....177
Defining national data items....177
Using national literals....178
COBOL statements and national data....179
Intrinsic functions and national data....181
Using national-character figurative constants....182
Defining national numeric data items....183
National groups....183
Converting to or from national (Unicode) representation....184
Using national groups....187
Storage of character data....190
Comparing national (UTF-16) data....190
Processing UTF-8 data using UTF-16 (national) data types....193
Processing Chinese GB 18030 data....193
Coding for use of DBCS support....194

Defining DBCS data....195
Using DBCS literals....195
Testing for valid DBCS characters....196
Processing alphanumeric data items that contain DBCS data....196

Chapter 11. Setting the locale....199
The active locale....199
Specifying the code page for character data....200
Using environment variables to specify a locale....201
Determination of the locale from system settings....202
Types of messages for which translations are available....202
Locales and code pages that are supported....202
Controlling the collating sequence with a locale....205
Controlling the alphanumeric collating sequence with a locale....206
Controlling the DBCS collating sequence with a locale....207
Controlling the national collating sequence with a locale....207
Intrinsic functions that depend on collating sequence....208
Accessing the active locale and code-page values....208
Example: get and convert a code-page ID....209

Part 3. Compiling, linking, running, and debugging your program....211

Chapter 12. Compiling, linking, and running programs....213
Setting environment variables....213
Compiler and runtime environment variables....214
Compiler environment variables....216
Runtime environment variables....218
Example: setting and accessing environment variables....221
Compiling programs....222
Compiling from the command line....223
Compiling using shell scripts....224
Specifying compiler options in the PROCESS (CBL) statement....224
Modifying the default compiler configuration....225
Correcting errors in your source program....227
Severity codes for compiler diagnostic messages....228
Generating a list of compiler messages....228
cob2 options....230
Linking programs....232
Passing options to the linker....232
Linker input and output files....233
Correcting errors in linking....235
Running programs....235

Chapter 13. Specifying compiler options on the command line....237
Flag options....237
-# (pound sign)....238
-?, ?....238
-q32, -q64....238
-c....239
-comprc_ok....240
-dll | -shared....240
-F....240
-g....241
-host....241
-I....242
-main....243
-o....243

-v ... 244
-q options ... 245
Compiler options ... 245
Option settings for 85 COBOL Standard conformance ... 248
Conflicting compiler options ... 248
ADATA ... 249
ADDR ... 249
ARITH ... 251
BINARY ... 252
CALLINT ... 252
CHAR ... 253
CICS ... 255
COLLSEQ ... 256
COMPILE ... 257
CURRENCY ... 258
DATEPROC ... 259
DATETIME ... 259
DEFINE ... 260
DIAGTRUNC ... 261
DYNAM ... 262
EXIT ... 263
FLAG ... 265
FLAGSTD ... 266
FLOAT ... 267
LINECOUNT ... 268
LIST ... 268
LSTFILE ... 269
MAP ... 269
MAXMEM ... 270
MDECK ... 270
NCOLLSEQ ... 271
NSYMBOL ... 272
NUMBER ... 272
OPTIMIZE ... 273
PGMNAME ... 274
APOST/QUOTE ... 275
SEPOBJ ... 275
SEQUENCE ... 276
SIZE ... 277
SOSI ... 277
SOURCE ... 278
SPACE ... 279
SPILL ... 279
SQL ... 279
SRCFORMAT ... 280
SSRANGE ... 281
TERMINAL ... 282
TEST ... 283
THREAD ... 283
TRUNC ... 283
UTF16 ... 285
VBREF ... 286
WSCLEAR ... 286
XREF ... 287
YEARWINDOW ... 288
ZWB ... 289

Chapter 14. Compiler-directing statements ... 291

Chapter 15. Runtime options 297
CHECK 297
DEBUG 298
ERRCOUNT 298
FILESYS 298
TRAP 300
UPSI 300

Chapter 16. Debugging 301
Debugging with source language 301
Tracing program logic 301
Finding and handling input-output errors 302
Validating data 302
Moving, initializing or setting uninitialized data 303
Generating information about procedures 303
Debugging using compiler options 304
Finding coding errors 305
Finding line sequence problems 305
Checking for valid ranges 306
Selecting the level of error to be diagnosed 306
Finding program entity definitions and references 308
Listing data items 309
Getting listings 309
Example: short listing 311
Example: SOURCE and NUMBER output 313
Example: MAP output 314
Example: XREF output: data-name cross-references 317
Example: VBREF compiler output 321
Debugging with messages that have offset information 321

Part 4. Targeting COBOL programs for certain environments 323

Chapter 17. Programming for a Db2 environment 325
Db2 coprocessor 325
Coding SQL statements 326
Using SQL INCLUDE with the Db2 coprocessor 326
Using binary items in SQL statements 327
Determining the success of SQL statements 327
Starting Db2 before compiling 327
Compiling with the SQL option 327
Separating Db2 suboptions 328
Using package and bind file-names 328

Chapter 18. Developing COBOL programs for CICS 329
Coding COBOL programs to run under CICS 330
Getting the system date under CICS 331
Making dynamic calls under CICS 332
Accessing SFS data 334
Calling between COBOL and C/C++ under CICS 334
Compiling and running CICS programs 334
Integrated CICS translator 335

Part 5. Using XML and COBOL together 337

Chapter 19. Processing XML input 339
XML parser in COBOL 339

Accessing XML documents ... 340
Parsing XML documents ... 341
Writing procedures to process XML ... 342
XML events ... 343
Transforming XML text to COBOL data items ... 345
The encoding of XML documents ... 346
XML input document encoding ... 346
Parsing XML documents encoded in UTF-8 ... 349
Handling XML PARSE exceptions ... 349
How the XML parser handles errors ... 350
Handling encoding conflicts ... 351
Terminating XML parsing ... 352
XML PARSE examples ... 353
Example: parsing a simple document ... 353
Example: program for processing XML ... 354

Chapter 20. Producing XML output ... 359
Generating XML output ... 359
Controlling the encoding of generated XML output ... 364
Handling XML GENERATE exceptions ... 364
Example: generating XML ... 365
Enhancing XML output ... 369
Example: enhancing XML output ... 369

Part 6. Working with more complex applications ... 373

Chapter 21. Porting applications between platforms ... 375
Getting IBM Enterprise COBOL for z/OS applications to compile ... 375
Getting IBM Enterprise COBOL for z/OS applications to run: overview ... 375
Fixing differences caused by data representations ... 376
Fixing environment differences that affect portability ... 378
Fixing differences caused by language elements ... 378
Writing code to run with IBM Enterprise COBOL for z/OS ... 379

Chapter 22. Using subprograms ... 381
Main programs, subprograms, and calls ... 381
Ending and reentering main programs or subprograms ... 381
Calling nested COBOL programs ... 382
Nested programs ... 383
Example: structure of nested programs ... 384
Scope of names ... 384
Calling nonnested COBOL programs ... 385
CALL identifier and CALL literal ... 385
Example: dynamic call using CALL identifier ... 386
Calling between COBOL and C/C++ programs ... 387
Initializing environments ... 387
Passing data between COBOL and C/C++ ... 388
Collapsing stack frames and terminating run units or processes ... 388
COBOL and C/C++ data types ... 389
Example: COBOL program calling C functions ... 390
Example: C programs that are called by and call COBOL ... 391
Example: COBOL program called by a C program ... 392
Example: results of compiling and running examples ... 392
Example: COBOL program calling C++ function ... 392
Making recursive calls ... 393
Passing return codes ... 393

Chapter 23. Sharing data....395
Passing data....395
Describing arguments in the calling program....397
Describing parameters in the called program....397
Testing for OMITTED arguments....397
Coding the LINKAGE SECTION....398
Coding the PROCEDURE DIVISION for passing arguments....399
Grouping data to be passed....399
Handling null-terminated strings....399
Using pointers to process a chained list....400
Using procedure and function pointers....402
Passing return-code information....403
Using the RETURN-CODE special register....403
Using PROCEDURE DIVISION RETURNING 403
Specifying CALL . . . RETURNING....403
Sharing data by using the EXTERNAL clause....404
Sharing files between programs (external files)....404
Example: using external files....405
Using command-line arguments....407
Example: command-line arguments without -host option....408
Example: command-line arguments with -host option....409

Chapter 24. Using shared libraries....411
Static linking versus using shared libraries....411
How the linker resolves references to shared libraries....412
Example: creating a sample shared library....412
Example: creating a makefile for the sample shared library....414

Chapter 25. Preinitializing the COBOL runtime environment....415
Initializing persistent COBOL environment....415
Terminating preinitialized COBOL environment....416
Example: preinitializing the COBOL environment....417

Chapter 26. Processing two-digit-year dates....421
Millennium language extensions (MLE)....422
Principles and objectives of these extensions....422
Resolving date-related logic problems....423
Using a century window....424
Using internal bridging....425
Moving to full field expansion....426
Using year-first, year-only, and year-last date fields....428
Compatible dates....428
Example: comparing year-first date fields....429
Using other date formats....429
Example: isolating the year....430
Manipulating literals as dates....430
Assumed century window....431
Treatment of nondates....432
Using sign conditions....433
Performing arithmetic on date fields....434
Allowing for overflow from windowed date fields....434
Specifying the order of evaluation....435
Controlling date processing explicitly....435
Using DATEVAL....436
Using UNDATE....436
Example: DATEVAL....437
Example: UNDATE....437

Analyzing and avoiding date-related diagnostic messages....437
Avoiding problems in processing dates....439
Avoiding problems with packed-decimal fields....439
Moving from expanded to windowed date fields....439

Part 7. Improving performance and productivity....441

Chapter 27. Tuning your program....443
Using an optimal programming style....443
Using structured programming....444
Factoring expressions....444
Using symbolic constants....444
Grouping constant computations....444
Grouping duplicate computations....445
Choosing efficient data types....445
Choosing efficient computational data items....446
Using consistent data types....446
Making arithmetic expressions efficient....446
Making exponentiations efficient....447
Handling tables efficiently....447
Optimization of table references....448
Optimizing your code....450
Optimization....450
Choosing compiler features to enhance performance....450
Performance-related compiler options....451
Evaluating performance....452

Chapter 28. Simplifying coding....455
Eliminating repetitive coding....455
Example: using the COPY statement....456
Manipulating dates and times....457
Getting feedback from date and time callable services....457
Handling conditions from date and time callable services....458
Example: manipulating dates....458
Example: formatting dates for output....458
Feedback token....459
Picture character terms and strings....460
Example: date-and-time picture strings....462
Century window....463
Using the format 2 SORT statement to sort a table....464

Appendix A. Summary of differences from IBM Enterprise COBOL for z/OS....467
Compiler options....467
Data representation....467
Binary data....467
Zoned decimal data....467
Packed-decimal data....468
Display floating-point data....468
National data....468
EBCDIC and ASCII data....468
Code-page determination for data conversion....468
DBCS character strings....468
Runtime environment variables....469
File specification....469
Interlanguage communication (ILC)....470
Input and output....470
Runtime options....471

Source code line size.... 471
Language elements....471

Appendix B. IBM Z host data format considerations....475
CICS access.... 475
Date and time callable services.... 475
Floating-point overflow exceptions.... 475
Db2....475
Distributed Computing Environment applications....476
File data.... 476
SORT....476

Appendix C. Intermediate results and arithmetic precision....477
Terminology used for intermediate results....478
Example: calculation of intermediate results....479
Fixed-point data and intermediate results.... 479
Addition, subtraction, multiplication, and division.... 479
Exponentiation.... 480
Example: exponentiation in fixed-point arithmetic....481
Truncated intermediate results.... 482
Binary data and intermediate results.... 482
Intrinsic functions evaluated in fixed-point arithmetic.... 482
Integer functions....482
Mixed functions.... 483
Floating-point data and intermediate results....484
Exponentiations evaluated in floating-point arithmetic.... 485
Intrinsic functions evaluated in floating-point arithmetic.... 485
Arithmetic expressions in nonarithmetic statements.... 485

Appendix D. Date and time callable services....487
CEECBLDY: convert date to COBOL integer format.... 488
CEEDATE: convert Lilian date to character format.... 492
CEEDATM: convert seconds to character time stamp.... 495
CEEDAYS: convert date to Lilian format....499
CEEDYWK: calculate day of week from Lilian date....502
CEEGMT: get current Greenwich Mean Time....504
CEEGMTO: get offset from Greenwich Mean Time to local time....506
CEEISEC: convert integers to seconds....508
CEELOCT: get current local date or time....510
CEEQCEN: query the century window....512
CEESCEN: set the century window.... 513
CEESECI: convert seconds to integers....514
CEESECS: convert time stamp to seconds....517
CEEUTC: get coordinated universal time.... 521
IGZEDT4: get current date.... 521

Appendix E. XML reference material....523
XML PARSE exceptions....523
XML PARSE exceptions that allow continuation....523
XML PARSE exceptions that do not allow continuation.... 528
XML conformance.... 531
XML GENERATE exceptions....533

Appendix F. EXIT compiler option....535
User-exit work area and work area extension.... 535
Parameter list for exit modules....536
Processing of INEXIT....537

Processing of LIBEXIT 538
Processing of PRTEXIT 539
Processing of MSGEXIT 539
Customizing compiler-message severities 540
Example: MSGEXIT user exit 542
Error handling for exit modules 546

Appendix G. Runtime messages 549

Notices 587
Trademarks 589
Glossary 591
List of resources 631
COBOL for Linux publications 631
Related publications 631

Index 633

Tables

1. FILE SECTION entries 10

2. Assignment to data items in a program 23

3. Ranges in value of COMP-5 data items 40

4. Internal representation of binary numeric items 42

5. Internal representation of native numeric items 43

6. Internal representation of numeric items when CHAR(EBCDIC) and FLOAT(BE) are in effect 45

7. Order of evaluation of arithmetic operators 50

8. Numeric intrinsic functions 51

9. File organization and access mode 121

10. Valid COBOL statements for sequential files 136

11. Valid COBOL statements for line-sequential files 136

12. Valid COBOL statements for indexed and relative files 137

13. Statements used when writing records to a file 138

14. PROCEDURE DIVISION statements used to update files 141

15. Sort and merge error numbers 159

16. COBOL statements and national data 179

17. Intrinsic functions and national character data 181

18. National group items that are processed with group semantics 189

19. Encoding and size of alphanumeric, DBCS, and national data 190

20. Supported locales and code pages 203

21. Intrinsic functions that depend on collating sequence 208

22. TZ environment parameter variables 216

23. Output from the cob2 command 224

24. Examples of compiler-option syntax in a shell script..... 224

25. Stanza attributes..... 227

26. Severity codes for compiler diagnostic messages..... 228

27. Common linker options..... 233

28. Default file-names assumed by the linker..... 235

29. Compiler options..... 246

30. Mutually exclusive compiler options..... 248

31. Effect of comparand data type and collating sequence on comparisons..... 256

32. Runtime options..... 297

33. Severity levels of compiler messages..... 306

34. Using compiler options to get listings..... 309

35. Terms and symbols used in MAP output..... 316

36. Special registers used by the XML parser..... 342

37. Results of processing-procedure changes to XML-CODE..... 344

38. Hexadecimal values of white-space characters..... 347

39. Hexadecimal values of special characters for various EBCDIC CCSIDs..... 348

40. XML events and special registers..... 353

41. Encoding of generated XML if the ENCODING phrase is omitted..... 364

42. ASCII characters contrasted with EBCDIC..... 376

43. ASCII comparisons contrasted with EBCDIC..... 376

44. IEEE contrasted with hexadecimal..... 377

45. COBOL and C/C++ data types..... 389

46. Methods for passing data in the CALL statement..... 395

47. Advantages and disadvantages of Year 2000 solutions..... 424

48. Performance-related compiler options..... 451

49. Performance-tuning worksheet....452
50. Picture character terms and strings....460
51. Japanese Eras....462
52. Examples of date-and-time picture strings....462
53. Comparison of format 1 and format 2 SORT statements....464
54. Language differences between Enterprise COBOL for z/OS and COBOL for Linux on x86....471
55. Maximum floating-point values....475
56. Date and time callable services....487
57. Date and time intrinsic functions....488
58. CEECBLDY symbolic conditions....490
59. CEEDATE symbolic conditions....492
60. CEEDATM symbolic conditions....496
61. CEEDAYS symbolic conditions....500
62. CEEDYWK symbolic conditions....502
63. CEEGMT symbolic conditions....505
64. CEEGMTO symbolic conditions....506
65. CEEISEC symbolic conditions....509
66. CEELOCT symbolic conditions....511
67. CEEQCEN symbolic conditions....512
68. CEESCEN symbolic conditions....513
69. CEESECI symbolic conditions....515
70. CEESECS symbolic conditions....518
71. XML PARSE exceptions that allow continuation....524
72. XML PARSE exceptions that do not allow continuation....528
73. XML GENERATE exceptions....533

74. Parameter list for exit modules..... 536

75. MSGEXIT processing.....539

76. FIPS (FLAGSTD) message categories.....541

77. Runtime messages.....549

Preface

About this information

Welcome to IBM COBOL for Linux on x86, IBM's COBOL compiler and runtime for Linux on x86.

This information describes use of the IBM COBOL compiler and runtime environment for Linux on x86, referred to in this information as COBOL for Linux.

There are some differences between host and workstation COBOL. For details about language and system differences between COBOL for Linux and Enterprise COBOL for z/OS®, see Appendix A, “Summary of differences from IBM Enterprise COBOL for z/OS,” on page 467.

How this information will help you

This information will help you write, compile, link-edit, and run IBM COBOL for Linux on x86 programs.

This information assumes experience in developing application programs and some knowledge of COBOL. It focuses on using COBOL to meet your programming objectives and not on the definition of the COBOL language. For complete information about COBOL syntax, see the *COBOL for Linux on x86 Language Reference*.

This information also assumes familiarity with Linux. For information about Linux, see your operating system documentation.

Abbreviated terms

Certain terms are used in a shortened form in this information. Abbreviations for the product names used most frequently are listed alphabetically in the table below.

Term used	Long form
CICS® TXSeries® or TXSeries	IBM TXSeries for Multiplatforms
CICS TX	IBM CICS TX on Cloud
CICS	Either IBM TXSeries for Multiplatforms or IBM CICS TX on Cloud
COBOL for Linux	IBM COBOL for Linux on x86
Db2®	IBM Db2 for Linux, UNIX and Windows

In addition to these abbreviated terms, the term "85 COBOL Standard" is used in this information to refer to the combination of the following standards:

- ISO 1989:1985, Programming languages - COBOL
- ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module
- ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL
- ANSI INCITS 23-1985, Programming Languages - COBOL
- ANSI INCITS 23a-1989, Programming Languages - Intrinsic Function Module for COBOL
- ANSI INCITS 23b-1993, Programming Language - Correction Amendment for COBOL

Other terms, if not commonly understood, are shown in *italics* the first time they appear and are listed in the glossary.

© Copyright IBM Corp. 2021

How to read syntax diagrams

Use the following description to read the syntax diagrams in this information:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The **>>---** symbol indicates the beginning of a syntax diagram.

  The **--->** symbol indicates that the syntax diagram is continued on the next line.

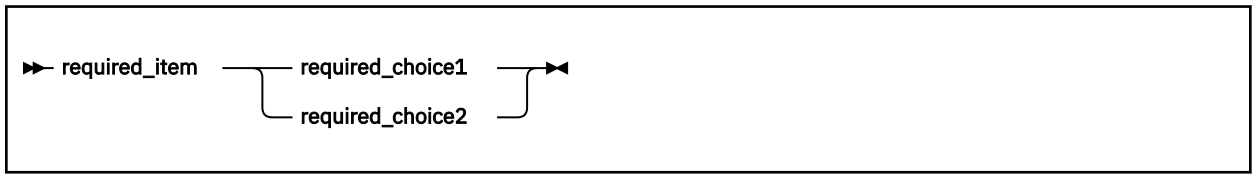
  The **>---** symbol indicates that the syntax diagram is continued from the previous line.

  The **---><** symbol indicates the end of a syntax diagram.

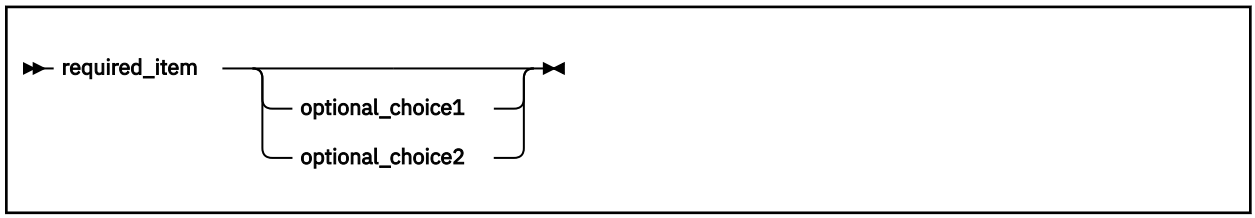
  Diagrams of syntactical units other than complete statements start with the **>---** symbol and end with the **--->** symbol.
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

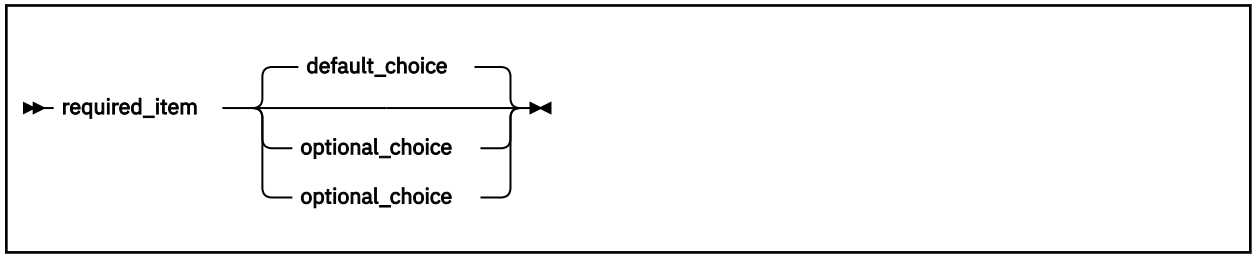
- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.



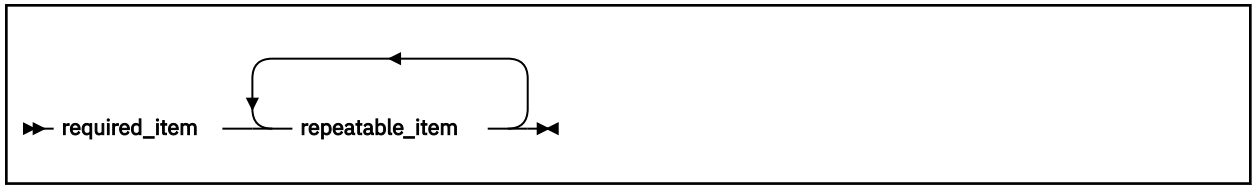
  If choosing one of the items is optional, the entire stack appears below the main path.



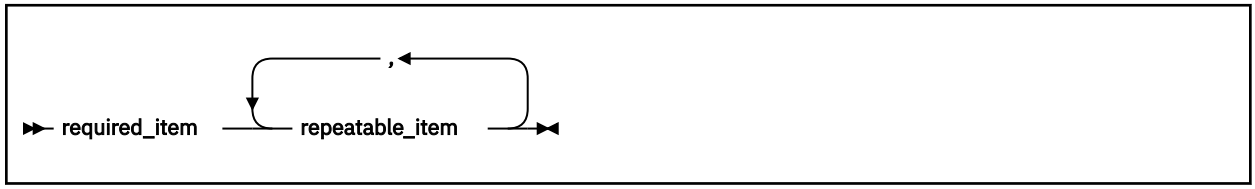
  If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.



If the repeat arrow contains a comma, you must separate repeated items with a comma.



- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

How examples are shown

This information shows numerous examples of sample COBOL statements, program fragments, and small programs to illustrate the coding techniques being discussed. The examples of program code are written in lowercase, uppercase, or mixed case to demonstrate that you can write your programs in any of these ways.

To more clearly separate some examples from the explanatory text, they are presented in a `monospace font`.

COBOL keywords and compiler options that appear in text are generally shown in SMALL UPPERCASE. Other terms such as program variable names are sometimes shown in *an italic font* for clarity.

Related information

The information in this Programming Guide is available online in the IBM COBOL for Linux documentation at http://www.ibm.com/support/knowledgecenter/SS7FZ2_1.1.0. The IBM Documentation website also has the *COBOL for Linux on x86 Language Reference* and information about using the debugger.

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this information or any other COBOL for Linux documentation, send your comments to: compinfo@cn.ibm.com.

Be sure to include the name of the document, the publication number, the version of COBOL for Linux, and, if applicable, the specific location (for example, the page number or section heading) of the text that you are commenting on.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way that IBM believes appropriate without incurring any obligation to you.

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

IBM COBOL for Linux on x86 uses the latest W3C Standard, WAI-ARIA 1.0, to ensure compliance to US Section 508 and Web Content Accessibility Guidelines (WCAG) 2.0. To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

Keyboard navigation

This product uses standard navigation keys.

Interface information

You can use speech recognition software like a Text-to-speech (TTS) tool to view the output generated by the product.

The online product documentation is available in IBM Documentation, which is viewable from a standard web browser.

PDF files have limited accessibility support. With PDF documentation, you can use optional font enlargement, high-contrast display settings, and can navigate by keyboard alone.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains the period or comma PICTURE symbols, you must set the screen reader to speak all punctuation.

Related accessibility information

In addition to standard IBM help desk and support websites, IBM has established a TTY telephone service for use by deaf or hard of hearing customers to access sales and support services:

TTY service 800-IBM-3383 (800-426-3383) (within North America)

IBM and accessibility

For more information about the commitment that IBM has to accessibility, see IBM Accessibility.

Part 1. Coding your program

© Copyright IBM Corp. 2021

Chapter 1. Structuring your program

COBOL programs consist of four divisions: `IDENTIFICATION DIVISION`, `ENVIRONMENT DIVISION`, `DATA DIVISION`, and `PROCEDURE DIVISION`. Each division has a specific logical function.

To define a program, only the `IDENTIFICATION DIVISION` is required.

Related tasks
"Identifying a program" on page 3
"Describing the computing environment" on page 5
"Describing the data" on page 9
"Processing the data" on page 13

Identifying a program

Use the `IDENTIFICATION DIVISION` to name a program and optionally provide other identifying information.

You can use the optional `AUTHOR`, `INSTALLATION`, `DATE-WRITTEN`, and `DATE-COMPILED` paragraphs for descriptive information about a program. The data you enter in the `DATE-COMPILED` paragraph is replaced with the latest compilation date.

```
IDENTIFICATION DIVISION.
Program-ID.    Helloprog.
Author.        A. Programmer.
Installation.  Computing Laboratories.
Date-Written.  06/30/2020.
Date-Compiled. 07/05/2020.
```

Use the `PROGRAM-ID` paragraph to name your program. The program-name that you assign is used in these ways:

- Other programs use that name to call your program.
- The name appears in the header on each page, except the first, of the program listing that is generated when you compile the program.

Tip: If a program-name is case sensitive, avoid mismatches with the name that the compiler is looking for. Verify that the appropriate setting of the PGMNAME compiler option is in effect.

Related tasks
"Changing the header of a source listing" on page 4
"Identifying a program as recursive" on page 4
"Marking a program as callable by containing programs" on page 4
"Setting a program to an initial state" on page 4

Related references
Compiler limits (*COBOL for Linux on x86 Language Reference*)
Conventions for program-names (*COBOL for Linux on x86 Language Reference*)

© Copyright IBM Corp. 2021

Identifying a program as recursive

Code the RECURSIVE attribute on the PROGRAM-ID clause to specify that a program can be recursively reentered while a previous invocation is still active.

You can code RECURSIVE only on the outermost program of a compilation unit. Neither nested subprograms nor programs that contain nested subprograms can be recursive.

Related tasks
“Sharing data in recursive programs” on page 13
“Making recursive calls” on page 393

Marking a program as callable by containing programs

Use the COMMON attribute in the PROGRAM-ID paragraph to specify that a program can be called by the containing program or by any program in the containing program. The COMMON program cannot be called by any program contained in itself.

Only contained programs can have the COMMON attribute.

Related concepts
“Nested programs” on page 383

Setting a program to an initial state

Use the INITIAL clause in the PROGRAM-ID paragraph to specify that whenever a program is called, that program and any nested programs that it contains are to be placed in their initial state.

When a program is set to its initial state:

- Data items that have VALUE clauses are set to the specified values.
- Changed GO TO statements and PERFORM statements are in their initial states.
- Non-EXTERNAL files are closed.

Related tasks
“Ending and reentering
main programs or subprograms” on page 381

Related references
“WSCLEAR” on page 286

Changing the header of a source listing

The header on the first page of a source listing contains the identification of the compiler and the current release level, the date and time of compilation, and the page number.

The following example shows these five elements:

```
PP  5737-L11 IBM COBOL for Linux 1.1.0     Date 05/29/2020   Time 17:38:17   Page     1
```

The header indicates the compilation platform. You can customize the header on succeeding pages of the listing by using the compiler-directing TITLE statement.

Related references
TITLE statement (*COBOL for Linux on x86 Language Reference*)

Describing the computing environment

In the ENVIRONMENT DIVISION of a program, you describe the aspects of the program that depend on the computing environment.

Use the CONFIGURATION SECTION to specify the following items:

- Computer for compiling the program (in the SOURCE-COMPUTER paragraph)
- Computer for running the program (in the OBJECT-COMPUTER paragraph)
- Special items such as the currency sign and symbolic characters (in the SPECIAL-NAMES paragraph)
- User-defined classes (in the REPOSITORY paragraph)

Use the FILE-CONTROL and I-O-CONTROL paragraphs of the INPUT-OUTPUT SECTION to:

- Identify and describe the characteristics of the files in the program.
- Associate your files with the corresponding system file-name, directly or indirectly.
- Optionally identify the file system (for example, SFS or STL) that is associated with a file. You can also do so at run time.
- Provide information about how the files are accessed.

"Example: FILE-CONTROL paragraph" on page 5

Related tasks

"Specifying the collating sequence" on page 6
"Defining symbolic characters" on page 7
"Defining a user-defined class" on page 8
"Identifying files to the operating system (ASSIGN)" on page 8

Related references

Sections and paragraphs (*COBOL for Linux on x86 Language Reference*)

Example: FILE-CONTROL paragraph

The following example shows how the FILE-CONTROL paragraph associates each file in the COBOL program with a physical file known to the file system. This example shows a FILE-CONTROL paragraph for an indexed file.

```
SELECT COMMUTER-FILE (1)
  ASSIGN TO COMMUTER (2)
  ORGANIZATION IS INDEXED (3)
  ACCESS IS RANDOM (4)
  RECORD KEY IS COMMUTER-KEY (5)
  FILE STATUS IS (5)
    COMMUTER-FILE-STATUS
    COMMUTER-STL-STATUS.
```

(1)
The SELECT clause associates a file in the COBOL program with a corresponding system file.

(2)
The ASSIGN clause associates the name of the file in the program with the name of the file as known to the system. COMMUTER might be the system file-name or the name of the environment variable whose runtime value is used as the system file-name with optional directory and path names.

(3)
The ORGANIZATION clause describes the organization of the file. If you omit this clause, ORGANIZATION IS SEQUENTIAL is assumed.

(4)
The ACCESS MODE clause defines the manner in which the records in the file are made available for processing: sequential, random, or dynamic. If you omit this clause, ACCESS IS SEQUENTIAL is assumed.

(5)
You might have additional statements in the FILE-CONTROL paragraph depending on the type of file and file system you use.

Related tasks
"Describing the computing environment" on page 5

Specifying the collating sequence

You can use the PROGRAM COLLATING SEQUENCE clause and the ALPHABET clause of the SPECIAL-NAMES paragraph to establish the collating sequence that is used in several operations on alphanumeric items.

These clauses specify the collating sequence for the following operations on alphanumeric items:

- Comparisons explicitly specified in relation conditions and condition-name conditions
- HIGH-VALUE and LOW-VALUE settings
- SEARCH ALL
- SORT and MERGE unless overridden by a COLLATING SEQUENCE phrase in the SORT or MERGE statement

"Example: specifying the collating sequence" on page 7

The sequence that you use can be based on one of these alphabets:

- EBCDIC: references the collating sequence associated with the EBCDIC character set
- NATIVE: references the collating sequence specified by the locale setting. The locale setting refers to the national language locale name in effect at compile time. It is usually set at installation.
- STANDARD-1: references the collating sequence associated with the ASCII character set defined by *ANSI INCITS X3.4, Coded Character Sets - 7-bit American National Standard Code for Information Interchange (7-bit ASCII)*
- STANDARD-2: references the collating sequence associated with the character set defined by *ISO/IEC 646 -- Information technology -- ISO 7-bit coded character set for information interchange, International Reference Version*
- An alteration of the ASCII sequence that you define in the SPECIAL-NAMES paragraph

You can also specify a collating sequence that you define.

Restriction: If the code page is DBCS, Extended UNIX Code (EUC), or UTF-8, you cannot use the ALPHABET clause.

The PROGRAM COLLATING SEQUENCE clause does not affect comparisons that involve national or DBCS operands.

Related tasks
"Choosing alternate collating sequences" on page 157
"Comparing national (UTF-16) data" on page 190
Chapter 11, "Setting the locale," on page 199
"Controlling the collating sequence with a locale" on page 205

Example: specifying the collating sequence

The following example shows the ENVIRONMENT DIVISION coding that you can use to specify a collating sequence in which uppercase and lowercase letters are similarly handled in comparisons and in sorting and merging.

When you change the ASCII sequence in the SPECIAL-NAMES paragraph, the overall collating sequence is affected, not just the collating sequence of the characters that are included in the SPECIAL-NAMES paragraph.

```
IDENTIFICATION DIVISION.
. . .
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
   Object-Computer.
      Program Collating Sequence Special-Sequence.
   Special-Names.
      Alphabet Special-Sequence Is
          "A" Also "a"
          "B" Also "b"
          "C" Also "c"
          "D" Also "d"
          "E" Also "e"
          "F" Also "f"
          "G" Also "g"
          "H" Also "h"
          "I" Also "i"
          "J" Also "j"
          "K" Also "k"
          "L" Also "l"
          "M" Also "m"
          "N" Also "n"
          "O" Also "o"
          "P" Also "p"
          "Q" Also "q"
          "R" Also "r"
          "S" Also "s"
          "T" Also "t"
          "U" Also "u"
          "V" Also "v"
          "W" Also "w"
          "X" Also "x"
          "Y" Also "y"
          "Z" Also "z".
```

Related tasks
“Specifying the collating sequence” on page 6

Defining symbolic characters

Use the SYMBOLIC CHARACTERS clause to give symbolic names to any character of the specified alphabet. Use ordinal position to identify the character, where position 1 corresponds to character X'00'.

For example, to give a name to the plus character (X'2B' in the ASCII alphabet), code:

```
SYMBOLIC CHARACTERS PLUS IS 44
```

You cannot use the SYMBOLIC CHARACTERS clause when the code page indicated by the locale is a multibyte-character code page.

Related tasks
Chapter 11, “Setting the locale,” on page 199

Defining a user-defined class

Use the CLASS clause to give a name to a set of characters that you list in the clause.

For example, name the set of digits by coding the following clause:

```
CLASS DIGIT IS "0" THROUGH "9"
```

You can reference the class-name only in a class condition. (This user-defined class is not the same as an object-oriented class.)

You cannot use the CLASS clause when the code page indicated by the locale is a multibyte-character code page.

Identifying files to the operating system (ASSIGN)

The ASSIGN clause associates the name of a file as it is known within a program to the associated file that will be used by the operating system.

You can use an environment variable, a system file-name, a literal, or a data-name in the ASSIGN clause. If you specify an environment variable as the assignment-name, the environment variable is evaluated at run time and the value (including optional directory and path names) is used as the system file-name.

If you use a file system other than the default, you need to indicate the file system explicitly, for example, by specifying the file-system identifier before the system file-name. For example, if MYFILE is an STL file, and you use F1 as the name of the file in your program, you can code the ASSIGN clause as follows:

```
SELECT F1 ASSIGN TO STL-MYFILE
```

If MYFILE is not an environment variable, or is an environment variable that is set to the empty string, the code shown above treats MYFILE as a system file-name. If MYFILE is an environment variable that has a value at run time other than the empty string, the value of the environment variable is used.

For example, if MYFILE is set by the command `export MYFILE=RSD-YOURFILE`, the system file-name is YOURFILE, and the file is treated as an RSD file, overriding the file-system ID (STL) coded in the ASSIGN clause.

If you enclose an assignment-name in quotation marks or single quotation marks (for example, "STL-MYFILE"), the value of any environment variable is ignored. The literal assignment-name is used.

Related tasks
“Varying the input or output file at run time” on page 8
“Identifying files” on page 112

Related references
“Precedence of file-system determination” on page 116
“FILESYS” on page 298
ASSIGN clause (*COBOL for Linux on x86 Language Reference*)

Varying the input or output file at run time

The *file-name* that you code in a SELECT clause is used as a constant throughout your COBOL program, but you can associate the name of that file with a different system file at run time.

Changing a file-name within a COBOL program would require changing the input statements and output statements and recompiling the program. Alternatively, you can change the *assignment-name* in the `export` command to use a different file at run time.

Environment variable values that are in effect at the time of the OPEN statement are used for associating COBOL file-names to the system file-names (including any path specifications).

Example: using different input files

This example shows that you can use the same COBOL program to access different files by setting an environment variable before the programs runs.

Consider a COBOL program that contains the following SELECT clause:

```
SELECT MAINFILE ASSIGN TO MAINA
```

Suppose you want the program to access either the `checking` or `savings` file using the file called `MAINFILE` within the program. To do so, set the MAINA environment variable before the program runs by using one of the following two statements as appropriate, assuming that the `checking` and `savings` files are in the `/accounts` directory:

```
export MAINA=/accounts/checking
export MAINA=/accounts/savings
```

You can thus use the same program to access either the `checking` or `savings` file as the file called `MAINFILE` within the program without having to change or recompile the source.

Describing the data

Define the characteristics of your data, and group your data definitions into one or more of the sections in the `DATA DIVISION`.

You can use these sections for defining the following types of data:

- Data used in input-output operations: `FILE SECTION`
- Data developed for internal processing:
  - To have storage be statically allocated and exist for the life of the *run unit*: `WORKING-STORAGE SECTION`
  - To have storage be allocated each time a program is entered, and deallocated on return from the program: `LOCAL-STORAGE SECTION`
- Data from another program: `LINKAGE SECTION`

The COBOL for Linux compiler limits the maximum size of `DATA DIVISION` elements. For details, see the related reference about compiler limits below.

Related concepts
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 11

Related tasks
"Using data in input and output operations" on page 9
"Using data from another program" on page 12

Related references
Compiler limits (*COBOL for Linux on x86 Language Reference*)

Using data in input and output operations

Define the data that you use in input and output operations in the `FILE SECTION`.

Provide the following information about the data:

- Name the input and output files that the program will use. Use the FD entry to give names to the files that the input-output statements in the `PROCEDURE DIVISION` can refer to.

Data items defined in the FILE SECTION are not available to PROCEDURE DIVISION statements until the file has been successfully opened.

- In the record description that follows the FD entry, describe the records in the file and their fields. The record-name is the object of WRITE and REWRITE statements.

Programs in the same run unit can refer to the same COBOL file-names.

You can use the EXTERNAL clause for separately compiled programs. A file that is defined as EXTERNAL can be referenced by any program in the run unit that describes the file.

You can use the GLOBAL clause for programs in a nested, or contained, structure. If a program contains another program (directly or indirectly), both programs can access a common file by referencing a GLOBAL file-name.

You can share physical files without using external or global file definitions in COBOL source programs. For example, you can specify that an application has two COBOL file-names, but these COBOL files are associated with one system file:

```
SELECT F1 ASSIGN TO MYFILE.
SELECT F2 ASSIGN TO MYFILE.
```

Related concepts
"Nested programs" on page 383

Related tasks
"Sharing files between programs (external files)" on page 404

Related references
"FILE SECTION entries" on page 10

FILE SECTION entries

The entries that you can use in the FILE SECTION are summarized in the table below.

Table 1. ***FILE SECTION entries***

Clause	To define
FD	The *file-name* to be referred to in PROCEDURE DIVISION input-output statements (OPEN, CLOSE, READ, START, and DELETE). Must match *file-name* in the SELECT clause. *file-name* is associated with the system file through the *assignment-name*.
RECORD CONTAINS *n*	Size of logical records (fixed length). Integer size indicates the number of bytes in a record regardless of the USAGE of the data items in the record.
RECORD IS VARYING	Size of logical records (variable length). If integer size or sizes are specified, they indicate the number of bytes in a record regardless of the USAGE of the data items in the record.
RECORD CONTAINS *n* TO *m*	Size of logical records (variable length). The integer sizes indicate the number of bytes in a record regardless of the USAGE of the data items in the record.
VALUE OF	An item in the label records associated with file. Comments only.
DATA RECORDS	Names of records associated with file. Comments only.
RECORDING MODE	Record type for sequential files

Related references
FILE SECTION (*COBOL for Linux on x86 Language Reference*)

Comparison of WORKING-STORAGE and LOCAL-STORAGE

How data items are allocated and initialized varies depending on whether the items are in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION.

When a program is invoked, the WORKING-STORAGE associated with the program is allocated.

Any data items that have VALUE clauses are initialized to the appropriate value at that time. For the duration of the run unit, WORKING-STORAGE items persist in their last-used state. Exceptions are:

- A program with INITIAL specified in the PROGRAM-ID paragraph

  In this case, WORKING-STORAGE data items are reinitialized each time that the program is entered.
- A subprogram that is dynamically called and then canceled

  In this case, WORKING-STORAGE data items are reinitialized on the first reentry into the program following the CANCEL.

WORKING-STORAGE is deallocated at the termination of the run unit.

See the Related tasks for information about WORKING-STORAGE in COBOL class definitions.

A separate copy of LOCAL-STORAGE data is allocated for each call of a program, and is freed on return from the program. If you specify a VALUE clause for a LOCAL-STORAGE item, the item is initialized to that value on each call. If a VALUE clause is not specified, the initial value of the item is undefined.

"Example: storage sections" on page 11

Related tasks
"Ending and reentering
main programs or subprograms" on page 381

Related references
WORKING-STORAGE SECTION (*COBOL for Linux on x86 Language Reference*)
LOCAL-STORAGE SECTION (*COBOL for Linux on x86 Language Reference*)

Example: storage sections

The following example is a recursive program that uses both WORKING-STORAGE and LOCAL-STORAGE.

```
CBL apost,pgmn(lu)
*********************************
* Recursive Program - Factorials
*********************************
 IDENTIFICATION DIVISION.
 Program-Id. factorial recursive.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  numb  pic 9(4)  value 5.
 01  fact  pic 9(8)  value 0.
 LOCAL-STORAGE SECTION.
 01  num  pic 9(4).
 PROCEDURE DIVISION.
     move numb to num.

     if numb = 0
        move 1 to fact
     else
        subtract 1 from numb
        call 'factorial'
        multiply num by fact
     end-if.
```

```
     display num '! = ' fact.
     goback.
 End Program factorial.
```

The program produces the following output:

```
0000! = 00000001
0001! = 00000001
0002! = 00000002
0003! = 00000006
0004! = 00000024
0005! = 00000120
```

The following tables show the changing values of the data items in LOCAL-STORAGE and WORKING-STORAGE in the successive recursive calls of the program, and in the ensuing gobacks. During the gobacks, `fact` progressively accumulates the value of 5! (five factorial).

Recursive calls	Value for num in LOCAL-STORAGE	Value for numb in WORKING-STORAGE	Value for fact in WORKING-STORAGE
Main	5	5	0
1	4	4	0
2	3	3	0
3	2	2	0
4	1	1	0
5	0	0	0

Gobacks	Value for num in LOCAL-STORAGE	Value for numb in WORKING-STORAGE	Value for fact in WORKING-STORAGE
5	0	0	1
4	1	0	1
3	2	0	2
2	3	0	6
1	4	0	24
Main	5	0	120

Related concepts
“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 11

Using data from another program

How you share data depends on the type of program. You share data differently in programs that are separately compiled than you do for programs that are nested or for programs that are recursive or multithreaded.

Related tasks
“Sharing data in separately compiled programs” on page 13
“Sharing data in nested programs” on page 13

“Sharing data in recursive programs” on page 13
“Passing data” on page 395

Sharing data in separately compiled programs

Many applications consist of separately compiled programs that call and pass data to one another. Use the LINKAGE SECTION in the called program to describe the data passed from another program.

In the calling program, code a CALL . . . USING statement to pass the data.

Related tasks
“Passing data” on page 395
“Coding the LINKAGE SECTION” on page 398

Sharing data in nested programs

Some applications consist of nested programs, that is, programs that are contained in other programs. Level-01 data items can include the GLOBAL attribute. This attribute allows any nested program that includes the declarations to access these data items.

A nested program can also access data items in a sibling program (one at the same nesting level in the same containing program) that is declared with the COMMON attribute.

Related concepts
“Nested programs” on page 383

Sharing data in recursive programs

If your program has the RECURSIVE attribute, data that is defined in the LINKAGE SECTION is not accessible on subsequent invocations of the program.

To address a record in the LINKAGE SECTION, use either of these techniques:

- Pass an argument to the program and specify the record in an appropriate position in the USING phrase in the program.
- Use the format-5 SET statement.

If your program has the RECURSIVE attribute, the address of the record is valid for a particular instance of the program invocation. The address of the record in another execution instance of the same program must be reestablished for that execution instance. Unpredictable results will occur if you refer to a data item for which the address has not been established.

Related tasks
“Making recursive calls” on page 393

Related references

SET statement (*COBOL for Linux on x86 Language Reference*)

Processing the data

In the PROCEDURE DIVISION of a program, you code the executable statements that process the data that you defined in the other divisions. The PROCEDURE DIVISION contains one or two headers and the logic of your program.

The PROCEDURE DIVISION begins with the division header and a procedure-name header. The division header for a program can simply be:

```
PROCEDURE DIVISION.
```

You can code the division header to receive parameters by using the USING phrase, or to return a value by using the RETURNING phrase.

To receive an argument that was passed by reference (the default) or by content, code the division header for a program in either of these ways:

```
PROCEDURE DIVISION USING dataname
PROCEDURE DIVISION USING BY REFERENCE dataname
```

Be sure to define *dataname* in the LINKAGE SECTION of the DATA DIVISION.

To receive a parameter that was passed by value, code the division header for a program as follows:

```
PROCEDURE DIVISION USING BY VALUE dataname
```

To return a value as a result, code the division header as follows:

```
PROCEDURE DIVISION RETURNING dataname2
```

You can also combine USING and RETURNING in a PROCEDURE DIVISION header:

```
PROCEDURE DIVISION USING dataname RETURNING dataname2
```

Be sure to define *dataname* and *dataname2* in the LINKAGE SECTION.

Related concepts
"How logic is divided in the PROCEDURE DIVISION" on page 14

Related tasks
"Coding the LINKAGE SECTION" on page 398
"Coding the PROCEDURE DIVISION for passing arguments" on page 399
"Using PROCEDURE DIVISION RETURNING . . ." on page 403
"Eliminating repetitive coding" on page 455

Related references
The procedure division header (*COBOL for Linux on x86 Language Reference*)
The USING phrase (*COBOL for Linux on x86 Language Reference*)
CALL statement (*COBOL for Linux on x86 Language Reference*)

How logic is divided in the PROCEDURE DIVISION

The PROCEDURE DIVISION of a program is divided into sections and paragraphs, which contain sentences, statements, and phrases.

Section
Logical subdivision of your processing logic.

A section has a section header and is optionally followed by one or more paragraphs.

A section can be the subject of a PERFORM statement. One type of section is for declaratives.

Paragraph
Subdivision of a section, procedure, or program.

A paragraph has a name followed by a period and zero or more sentences.

A paragraph can be the subject of a statement.

Sentence
Series of one or more COBOL statements that ends with a period.

Statement
Performs a defined step of COBOL processing, such as adding two numbers.

A statement is a valid combination of words, and begins with a COBOL statement. Statements are imperative (indicating unconditional action), conditional, or compiler-directing. Using explicit scope terminators instead of periods to show the logical end of a statement is preferred.

Phrase
A subdivision of a statement.

Related concepts
“Compiler-directing statements” on page 16
“Scope terminators” on page 16
“Imperative statements” on page 15
“Conditional statements” on page 15
“Declaratives” on page 17

Related references
PROCEDURE DIVISION structure (*COBOL for Linux on x86 Language Reference*)

Imperative statements

An imperative statement (such as ADD, MOVE, CALL, or CLOSE) indicates an unconditional action to be taken.

You can end an imperative statement with an implicit or explicit scope terminator.

A conditional statement that ends with an explicit scope terminator becomes an imperative statement called a *delimited scope statement*. Only imperative statements (or delimited scope statements) can be nested.

Related concepts
“Conditional statements” on page 15
“Scope terminators” on page 16

Conditional statements

A conditional statement is either a simple conditional statement (IF, EVALUATE, SEARCH) or a conditional statement made up of an imperative statement that includes a conditional phrase or option.

You can end a conditional statement with an implicit or explicit scope terminator. If you end a conditional statement explicitly, it becomes a delimited scope statement (which is an imperative statement).

You can use a delimited scope statement in these ways:

- To delimit the range of operation for a COBOL conditional statement and to explicitly show the levels of nesting

  For example, use an END-IF phrase instead of a period to end the scope of an IF statement within a nested IF.
- To code a conditional statement where the COBOL syntax calls for an imperative statement

  For example, code a conditional statement as the object of an inline PERFORM:

```
PERFORM UNTIL TRANSACTION-EOF
    PERFORM 200-EDIT-UPDATE-TRANSACTION
    IF NO-ERRORS
        PERFORM 300-UPDATE-COMMUTER-RECORD
    ELSE
        PERFORM 400-PRINT-TRANSACTION-ERRORS
    END-IF
    READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
        AT END
            SET TRANSACTION-EOF TO TRUE
    END-READ
END-PERFORM
```

An explicit scope terminator is required for the inline PERFORM statement, but it is not valid for the out-of-line PERFORM statement.

For additional program control, you can use the NOT phrase with conditional statements. For example, you can provide instructions to be performed when a particular exception does not occur, such as NOT ON SIZE ERROR. The NOT phrase cannot be used with the ON OVERFLOW phrase of the CALL statement, but it can be used with the ON EXCEPTION phrase.

Do not nest conditional statements. Nested statements must be imperative statements (or delimited scope statements) and must follow the rules for imperative statements.

The following statements are examples of conditional statements if they are coded without scope terminators:

- Arithmetic statement with ON SIZE ERROR
- Data-manipulation statements with ON OVERFLOW
- CALL statements with ON OVERFLOW
- I/O statements with INVALID KEY, AT END, or AT END-OF-PAGE
- RETURN with AT END

Related concepts
“Imperative statements” on page 15
“Scope terminators” on page 16

Related tasks
“Selecting program actions” on page 81

Related references
Conditional statements (*COBOL for Linux on x86 Language Reference*)

Compiler-directing statements

A compiler-directing statement causes the compiler to take specific action about the program structure, COPY processing, listing control, control flow, or CALL interface convention.

A compiler-directing statement is not part of the program logic.

Related references
Chapter 14, “Compiler-directing statements,” on page 291
Compiler-directing statements (*COBOL for Linux on x86 Language Reference*)

Scope terminators

A scope terminator ends a statement. Scope terminators can be explicit or implicit.

Explicit scope terminators end a statement without ending a sentence. They consist of END followed by a hyphen and the name of the statement being terminated, such as END-IF. An implicit scope terminator is a period (.) that ends the scope of all previous statements not yet ended.

Each of the two periods in the following program fragment ends an IF statement, making the code equivalent to the code after it that instead uses explicit scope terminators:

```
IF ITEM = "A"
    DISPLAY "THE VALUE OF ITEM IS " ITEM
    ADD 1 TO TOTAL
    MOVE "C" TO ITEM
    DISPLAY "THE VALUE OF ITEM IS NOW " ITEM.
IF ITEM = "B"
    ADD 2 TO TOTAL.
```

```
IF ITEM = "A"
```

```
    DISPLAY "THE VALUE OF ITEM IS " ITEM
    ADD 1 TO TOTAL
    MOVE "C" TO ITEM
    DISPLAY "THE VALUE OF ITEM IS NOW " ITEM
END-IF
IF ITEM = "B"
    ADD 2 TO TOTAL
END-IF
```

If you use implicit terminators, the end of statements can be unclear. As a result, you might end statements unintentionally, changing your program's logic. Explicit scope terminators make a program easier to understand and prevent unintentional ending of statements. For example, in the program fragment below, changing the location of the first period in the first implicit scope example changes the meaning of the code:

```
IF ITEM = "A"
    DISPLAY "VALUE OF ITEM IS " ITEM
    ADD 1 TO TOTAL.
    MOVE "C" TO ITEM
    DISPLAY " VALUE OF ITEM IS NOW " ITEM
IF ITEM = "B"
    ADD 2 TO TOTAL.
```

The MOVE statement and the DISPLAY statement after it are performed regardless of the value of ITEM, despite what the indentation indicates, because the first period terminates the IF statement.

For improved program clarity and to avoid unintentional ending of statements, use explicit scope terminators, especially within paragraphs. Use implicit scope terminators only at the end of a paragraph or the end of a program.

Be careful when coding an explicit scope terminator for an imperative statement that is nested within a conditional statement. Ensure that the scope terminator is paired with the statement for which it was intended. In the following example, the scope terminator will be paired with the second READ statement, though the programmer intended it to be paired with the first.

```
READ FILE1
  AT END
    MOVE A TO B
    READ FILE2
END-READ
```

To ensure that the explicit scope terminator is paired with the intended statement, the preceding example can be recoded in this way:

```
READ FILE1
  AT END
    MOVE A TO B
    READ FILE2
    END-READ
END-READ
```

Related concepts
“Conditional statements” on page 15
“Imperative statements” on page 15

Declaratives

Declaratives provide one or more special-purpose sections that are executed when an exception condition occurs.

Start each declarative section with a USE statement that identifies the function of the section. In the procedures, specify the actions to be taken when the condition occurs.

Related tasks
"Finding and handling input-output errors" on page 302

Related references
Declaratives (*COBOL for Linux on x86 Language Reference*)

Chapter 2. Using data

This information is intended to help non-COBOL programmers relate terms for data used in other programming languages to COBOL terms. It introduces COBOL fundamentals for variables, structures, literals, and constants; assigning and displaying values; intrinsic (built-in) functions, and tables (arrays) and pointers.

Related tasks
"Using variables, structures, literals, and constants" on page 19
"Assigning values to data items" on page 23
"Displaying values on a screen or in a file (DISPLAY)" on page 31
"Using intrinsic functions (built-in functions)" on page 32
"Using tables (arrays) and pointers" on page 33
Chapter 10, "Processing data in an international environment," on page 175

Using variables, structures, literals, and constants

Most high-level programming languages share the concept of data being represented as variables, structures (group items), literals, or constants.

The data in a COBOL program can be alphabetic, alphanumeric, double-byte character set (DBCS), national, or numeric. You can also define index-names and data items described as `USAGE POINTER`, `USAGE FUNCTION-POINTER`, or `USAGE PROCEDURE-POINTER`. You place all data definitions in the `DATA DIVISION` of your program.

Related tasks
"Using variables" on page 19
"Using data items and group items" on page 20
"Using literals" on page 21
"Using constants" on page 22
"Using figurative constants" on page 22

Related references
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)

Using variables

A *variable* is a data item whose value can change during a program. The value is restricted, however, to the data type that you define when you specify a name and a length for the data item.

For example, if a customer name is an alphanumeric data item in your program, you could define and use the customer name as shown below:

```
Data Division.
01  Customer-Name           Pic X(20).
01  Original-Customer-Name  Pic X(20).
. . .
Procedure Division.
    Move Customer-Name to Original-Customer-Name
    . . .
```

© Copyright IBM Corp. 2021

You could instead define the customer names above as national data items by specifying their PICTURE clauses as `Pic N(20)` and specifying the `USAGE NATIONAL` clause for the items. National data items are represented in Unicode UTF-16, in which most characters are represented in 2 bytes of storage.

Related concepts
"Unicode and the encoding of language characters" on page 176

Related tasks
"Using national data (Unicode) in COBOL" on page 177

Related references
"NSYMBOL" on page 272
"Storage of character data" on page 190
PICTURE clause (*COBOL for Linux on x86 Language Reference*)

Using data items and group items

Related data items can be parts of a hierarchical data structure. A data item that does not have subordinate data items is called an *elementary item*. A data item that is composed of one or more subordinate data items is called a *group item*.

A record can be either an elementary item or a group item. A group item can be either an *alphanumeric group item* or a *national group item*.

For example, `Customer-Record` below is an alphanumeric group item that is composed of two subordinate alphanumeric group items (`Customer-Name` and `Part-Order`), each of which contains elementary data items. These groups items implicitly have `USAGE DISPLAY`. You can refer to an entire group item or to parts of a group item in `MOVE` statements in the `PROCEDURE DIVISION` as shown below:

```
Data Division.
File Section.
FD  Customer-File
    Record Contains 45 Characters.
01  Customer-Record.
    05  Customer-Name.
        10 Last-Name        Pic x(17).
        10 Filler           Pic x.
        10 Initials         Pic xx.
    05  Part-Order.
        10 Part-Name        Pic x(15).
        10 Part-Color       Pic x(10).
Working-Storage Section.
01  Orig-Customer-Name.
    05  Surname            Pic x(17).
    05  Initials           Pic x(3).
01  Inventory-Part-Name    Pic x(15).
. . .
Procedure Division.
    Move Customer-Name to Orig-Customer-Name
    Move Part-Name to Inventory-Part-Name
    . . .
```

You could instead define `Customer-Record` as a national group item that is composed of two subordinate national group items by changing the declarations in the `DATA DIVISION` as shown below. National group items behave in the same way as elementary category national data items in most operations. The `GROUP-USAGE NATIONAL` clause indicates that a group item and any group items subordinate to it are national groups. Subordinate elementary items in a national group must be explicitly or implicitly described as `USAGE NATIONAL`.

```
Data Division.
File Section.
FD  Customer-File
    Record Contains 90 Characters.
```

```
01  Customer-Record         Group-Usage National.
    05  Customer-Name.
        10 Last-Name        Pic n(17).
        10 Filler           Pic n.
        10 Initials         Pic nn.
    05  Part-Order.
        10 Part-Name        Pic n(15).
        10 Part-Color       Pic n(10).
Working-Storage Section.
01  Orig-Customer-Name      Group-Usage National.
    05  Surname            Pic n(17).
    05  Initials           Pic n(3).
01  Inventory-Part-Name     Pic n(15)  Usage National.
. . .
Procedure Division.
    Move Customer-Name to Orig-Customer-Name
    Move Part-Name to Inventory-Part-Name
    . . .
```

In the example above, the group items could instead specify the USAGE NATIONAL clause at the group level. A USAGE clause at the group level applies to each elementary data item in a group (and thus serves as a convenient shorthand notation). However, a group that specifies the USAGE NATIONAL clause is *not* a national group despite the representation of the elementary items within the group. Groups that specify the USAGE clause are alphanumeric groups and behave in many operations, such as moves and compares, like elementary data items of USAGE DISPLAY (except that no editing or conversion of data occurs).

Related concepts
“Unicode and the encoding of language characters” on page 176
“National groups” on page 183

Related tasks
“Using national data (Unicode) in COBOL” on page 177
“Using national groups” on page 187

Related references
“FILE SECTION entries” on page 10
“Storage of character data” on page 190
Classes and categories of group items (*COBOL for Linux on x86 Language Reference*)
PICTURE clause (*COBOL for Linux on x86 Language Reference*)
MOVE statement (*COBOL for Linux on x86 Language Reference*)
USAGE clause (*COBOL for Linux on x86 Language Reference*)

Using literals

A *literal* is a character string whose value is given by the characters themselves. If you know the value you want a data item to have, you can use a literal representation of the data value in the PROCEDURE DIVISION.

You do not need to define a data item for the value nor refer to it by using a data-name. For example, you can prepare an error message for an output file by moving an alphanumeric literal:

```
Move "Name is not valid" To Customer-Name
```

You can compare a data item to a specific integer value by using a numeric literal. In the example below, "Name is not valid" is an alphanumeric literal, and 03519 is a numeric literal:

```
01  Part-number     Pic 9(5).
. . .
    If Part-number = 03519 then display "Part number was found"
```

You can use hexadecimal-notation format (X') to express control characters X'00' through X'1F' within an alphanumeric literal. Results are unpredictable if you specify these control characters in the basic format of alphanumeric literals.

You can use the opening delimiter N" or N' to designate a national literal if the NSYMBOL(NATIONAL) compiler option is in effect, or to designate a DBCS literal if the NSYMBOL(DBCS) compiler option is in effect.

You can use the opening delimiter NX" or NX' to designate national literals in hexadecimal notation (regardless of the setting of the NSYMBOL compiler option). Each group of four hexadecimal digits designates a single national character.

Related concepts
"Unicode and the encoding of language characters" on page 176

Related tasks
"Using national literals" on page 178
"Using DBCS literals" on page 195

Related references
"NSYMBOL" on page 272
Literals (*COBOL for Linux on x86 Language Reference*)

Using constants

A *constant* is a data item that has only one value. COBOL does not define a construct for constants. However, you can define a data item with an initial value by coding a VALUE clause in the data description (instead of coding an INITIALIZE statement).

```
Data Division.
01  Report-Header   pic x(50)  value "Company Sales Report".
. . .
01  Interest        pic 9v9999 value 1.0265.
```

The example above initializes an alphanumeric and a numeric data item. You can likewise use a VALUE clause in defining a national or DBCS constant.

Related tasks
"Using national data (Unicode) in COBOL" on page 177
"Coding for use of DBCS support" on page 194

Using figurative constants

Certain commonly used constants and literals are available as reserved words called *figurative constants*: ZERO, SPACE, HIGH-VALUE, LOW-VALUE, QUOTE, NULL, and ALL *literal*. Because they represent fixed values, figurative constants do not require a data definition.

For example:

```
Move Spaces To Report-Header
```

Related tasks
"Using national-character figurative constants" on page 182
"Coding for use of DBCS support" on page 194

Related references
Figurative constants (*COBOL for Linux on x86 Language Reference*)

Assigning values to data items

After you have defined a data item, you can assign a value to it at any time. Assignment takes many forms in COBOL, depending on what you want to do.

Table 2. ***Assignment to data items in a program***

What you want to do	How to do it
Assign values to a data item or large data area.	Use one of these ways: • `INITIALIZE` statement • `MOVE` statement • `STRING` or `UNSTRING` statement • `VALUE` clause (to set data items to the values you want them to have when the program is in initial state)
Assign the results of arithmetic.	Use `COMPUTE`, `ADD`, `SUBTRACT`, `MULTIPLY`, or `DIVIDE` statements.
Examine or replace characters or groups of characters in a data item.	Use the `INSPECT` statement.
Receive values from a file.	Use the `READ` (or `READ INTO`) statement.
Receive values from a system input device or a file.	Use the `ACCEPT` statement.
Establish a constant.	Use the `VALUE` clause in the definition of the data item, and do not use the data item as a receiver. Such an item is in effect a constant even though the compiler does not enforce read-only constants.
One of these actions: • Place a value associated with a table element in an index. • Set the status of an external switch to `ON` or `OFF`. • Move data to a condition-name to make the condition true. • Set a `POINTER`, `PROCEDURE-POINTER`, or `FUNCTION-POINTER` data item to an address.	Use the `SET` statement.

"Examples: initializing data items" on page 24

Related tasks
"Initializing a structure (INITIALIZE)" on page 27
"Assigning values to elementary data items (MOVE)" on page 28
"Assigning values to group data items (MOVE)" on page 29
"Assigning input from a screen or file (ACCEPT)" on page 30
"Joining data items (STRING)" on page 93
"Splitting data items (UNSTRING)" on page 95
"Assigning arithmetic results

(MOVE or COMPUTE)" on page 30
"Tallying and replacing
data items (INSPECT)" on page 102
Chapter 10, "Processing data in an international
environment," on page 175

Examples: initializing data items

The following examples show how you can initialize many kinds of data items, including alphanumeric, national-edited, and numeric-edited data items, by using INITIALIZE statements.

An INITIALIZE statement is functionally equivalent to one or more MOVE statements. The related tasks about initializing show how you can use an INITIALIZE statement on a group item to conveniently initialize all the subordinate data items that are in a given data category.

Initializing a data item to blanks or zeros:

```
INITIALIZE identifier-1
```

***identifier-1* PICTURE**	***identifier-1* before**	***identifier-1* after**
9(5)	12345	00000
X(5)	AB123	*bbbbb*[1]
N(3)	410042003100[2]	200020002000[3]
99XX9	12AB3	*bbbbb*[1]
XXBX/XX	ABbC/DE	*bbbb/bb*[1]
**99.9CR	1234.5CR	**00.0*bb*[1]
A(5)	ABCDE	*bbbbb*[1]
+99.99E+99	+12.34E+02	+00.00E+00

1. The symbol *b* represents a blank space.
2. Hexadecimal representation of the national (UTF-16) characters 'AB1'. The example assumes that *identifier-1* has Usage National.
3. Hexadecimal representation of the national (UTF-16) characters ' ' (three blank spaces). Note that if *identifier-1* were not defined as Usage National, and if NSYMBOL(DBCS) were in effect, INITIALIZE would instead store DBCS spaces ('2020') into *identifier-1*.

Initializing an alphanumeric data item:

```
01  ALPHANUMERIC-1    PIC X    VALUE "y".
01  ALPHANUMERIC-3    PIC X(1) VALUE "A".
. . .
    INITIALIZE ALPHANUMERIC-1
        REPLACING ALPHANUMERIC DATA BY ALPHANUMERIC-3
```

ALPHANUMERIC-3	**ALPHANUMERIC-1 before**	**ALPHANUMERIC-1 after**
A	y	A

Initializing an alphanumeric right-justified data item:

```
01  ANJUST           PIC X(8)  VALUE SPACES  JUSTIFIED RIGHT.
01  ALPHABETIC-1     PIC A(4)  VALUE "ABCD".
```

```
. . .
    INITIALIZE ANJUST
        REPLACING ALPHANUMERIC DATA BY ALPHABETIC-1
```

ALPHABETIC-1	ANJUST before	ANJUST after
ABCD	*bbbbbbbb*[1]	*bbbb*ABCD[1]
1. The symbol *b* represents a blank space.		

Initializing an alphanumeric-edited data item:

```
01  ALPHANUM-EDIT-1   PIC XXBX/XXX  VALUE "AB*b*C/DEF".
01  ALPHANUM-EDIT-3   PIC X/BB      VALUE "M/*bb*".
. . .
    INITIALIZE ALPHANUM-EDIT-1
        REPLACING ALPHANUMERIC-EDITED DATA BY ALPHANUM-EDIT-3
```

ALPHANUM-EDIT-3	ALPHANUM-EDIT-1 before	ALPHANUM-EDIT-1 after
M/*bb*[1]	AB*b*C/DEF[1]	M/*bb*/*bbb*[1]
1. The symbol *b* represents a blank space.		

Initializing a national data item:

```
01  NATIONAL-1        PIC NN  USAGE NATIONAL  VALUE N"AB".
01  NATIONAL-3        PIC NN  USAGE NATIONAL  VALUE N"CD".
. . .
    INITIALIZE NATIONAL-1
        REPLACING NATIONAL DATA BY NATIONAL-3
    INITIALIZE NATIONAL-1 NATIONAL TO VALUE
```

NATIONAL-3	NATIONAL-1 before first INITIALIZE	NATIONAL-1 after first INITIALIZE	NATIONAL-1 after second INITIALIZE
43004400[1]	41004200[2]	43004400[1]	41004200
1. Hexadecimal representation of the national characters 'CD' 2. Hexadecimal representation of the national characters 'AB'			

Initializing a national-edited data item:

```
01  NATL-EDIT-1       PIC 0NN  USAGE NATIONAL  VALUE N"123".
01  NATL-3            PIC NNN  USAGE NATIONAL  VALUE N"456".
. . .
    INITIALIZE NATL-EDIT-1
        REPLACING NATIONAL-EDITED DATA BY NATL-3
```

NATL-3	NATL-EDIT-1 before	NATL-EDIT-1 after
340035003600[1]	310032003300[2]	300034003500[3]
1. Hexadecimal representation of the national characters '456' 2. Hexadecimal representation of the national characters '123' 3. Hexadecimal representation of the national characters '045'		

Initializing a numeric (zoned decimal) data item:

```
01  NUMERIC-1         PIC 9(8)        VALUE 98765432.
01  NUM-INT-CMPT-3    PIC 9(7)  COMP  VALUE 1234567.
. . .
    INITIALIZE NUMERIC-1
        REPLACING NUMERIC DATA BY NUM-INT-CMPT-3
```

NUM-INT-CMPT-3	NUMERIC-1 before	NUMERIC-1 after
1234567	98765432	01234567

Initializing a numeric (national decimal) data item:

```
01  NAT-DEC-1         PIC 9(3)  USAGE  NATIONAL VALUE 987.
01  NUM-INT-BIN-3     PIC 9(2)  BINARY VALUE 12.
. . .
    INITIALIZE NAT-DEC-1
        REPLACING NUMERIC DATA BY NUM-INT-BIN-3
```

NUM-INT-BIN-3	NAT-DEC-1 before	NAT-DEC-1 after
12	390038003700[1]	300031003200[2]

1. Hexadecimal representation of the national characters '987'
2. Hexadecimal representation of the national characters '012'

Initializing a numeric-edited (USAGE DISPLAY) data item:

```
01  NUM-EDIT-DISP-1   PIC $ZZ9V  VALUE "$127".
01  NUM-DISP-3        PIC 999V   VALUE 12.
. . .
    INITIALIZE NUM-EDIT-DISP-1
        REPLACING NUMERIC-EDITED DATA BY NUM-DISP-3
```

NUM-DISP-3	NUM-EDIT-DISP-1 before	NUM-EDIT-DISP-1 after
012	$127	$ 12

Initializing a numeric-edited (USAGE NATIONAL) data item:

```
01  NUM-EDIT-NATL-1   PIC $ZZ9V  NATIONAL VALUE N"$127".
01  NUM-NATL-3        PIC 999V   NATIONAL VALUE 12.
. . .
    INITIALIZE NUM-EDIT-NATL-1
        REPLACING NUMERIC-EDITED DATA BY NUM-NATL-3
```

NUM-NATL-3	NUM-EDIT-NATL-1 before	NUM-EDIT-NATL-1 after
300031003200[1]	2400310032003700[2]	2400200031003200[3]

1. Hexadecimal representation of the national characters '012'
2. Hexadecimal representation of the national characters '$127'
3. Hexadecimal representation of the national characters '$ 12'

Related tasks
“Initializing a structure

(INITIALIZE)" on page 27
"Initializing a table (INITIALIZE)" on page 65
"Defining numeric data" on page 35

Related references
"NSYMBOL" on page 272

Initializing a structure (INITIALIZE)

You can reset the values of all subordinate data items in a group item by applying the INITIALIZE statement to that group item. However, it is inefficient to initialize an entire group unless you really need all the items in the group to be initialized.

The following example shows how you can reset fields to spaces and zeros in transaction records that a program produces. The values of the fields are not identical in each record that is produced. (The transaction record is defined as an alphanumeric group item, TRANSACTION-OUT.)

```
01  TRANSACTION-OUT.
    05  TRANSACTION-CODE        PIC X.
    05  PART-NUMBER             PIC 9(6).
    05  TRANSACTION-QUANTITY    PIC 9(5).
    05  PRICE-FIELDS.
        10  UNIT-PRICE          PIC 9(5)V9(2).
        10  DISCOUNT            PIC V9(2).
        10  SALES-PRICE         PIC 9(5)V9(2).
. . .
    INITIALIZE TRANSACTION-OUT
```

Record	TRANSACTION-OUT before	TRANSACTION-OUT after
1	R00138300024000000000000000	*b*00000000000000000000000000[1]
2	R00139000048000000000000000	*b*00000000000000000000000000[1]
3	S00141000012000000000000000	*b*00000000000000000000000000[1]
4	C00138300000000425000000000	*b*00000000000000000000000000[1]
5	C00201000000000000100000000	*b*00000000000000000000000000[1]
1. The symbol *b* represents a blank space.		

You can likewise reset the values of all the subordinate data items in a national group item by applying the INITIALIZE statement to that group item. The following structure is similar to the preceding structure, but instead uses Unicode UTF-16 data:

```
01  TRANSACTION-OUT GROUP-USAGE NATIONAL.
    05  TRANSACTION-CODE        PIC N.
    05  PART-NUMBER             PIC 9(6).
    05  TRANSACTION-QUANTITY    PIC 9(5).
    05  PRICE-FIELDS.
        10  UNIT-PRICE          PIC 9(5)V9(2).
        10  DISCOUNT            PIC V9(2).
        10  SALES-PRICE         PIC 9(5)V9(2).
. . .
    INITIALIZE TRANSACTION-OUT
```

Regardless of the previous contents of the transaction record, after the INITIALIZE statement above is executed:

- TRANSACTION-CODE contains NX"2000" (a national space).
- Each of the remaining 27 national character positions of TRANSACTION-OUT contains NX"3000" (a national-decimal zero).

When you use an INITIALIZE statement to initialize an alphanumeric or national group data item, the data item is processed as a group item, that is, with group semantics. The elementary data items within the group are recognized and processed, as shown in the examples above. If you do not code the REPLACING phrase of the INITIALIZE statement:

- SPACE is the implied sending item for alphabetic, alphanumeric, alphanumeric-edited, DBCS, category national, and national-edited receiving items.
- ZERO is the implied sending item for numeric and numeric-edited receiving items.

Related concepts
"National groups" on page 183

Related tasks
"Initializing a table (INITIALIZE)" on page 65
"Using national groups" on page 187

Related references
INITIALIZE statement (*COBOL for Linux on x86 Language Reference*)

Assigning values to elementary data items (MOVE)

Use a MOVE statement to assign a value to an elementary data item.

The following statement assigns the contents of an elementary data item, Customer-Name, to the elementary data item Orig-Customer-Name:

```
Move Customer-Name to Orig-Customer-Name
```

If Customer-Name is longer than Orig-Customer-Name, truncation occurs on the right. If Customer-Name is shorter, the extra character positions on the right in Orig-Customer-Name are filled with spaces.

For data items that contain numbers, moves can be more complicated than with character data items because there are several ways in which numbers can be represented. In general, the algebraic values of numbers are moved if possible, as opposed to the digit-by-digit moves that are performed with character data. For example, after the MOVE statement below, Item-x contains the value 3.0, represented as 0030:

```
01  Item-x      Pic 999v9.
. . .
    Move 3.06 to Item-x
```

You can move an alphabetic, alphanumeric, alphanumeric-edited, DBCS, integer, or numeric-edited data item to a category national or national-edited data item; the sending item is converted. You can move a national data item to a category national or national-edited data item. If the content of a category national data item has a numeric value, you can move that item to a numeric, numeric-edited, external floating-point, or internal floating-point data item. You can move a national-edited data item only to a category national data item or another national-edited data item. Padding or truncation might occur.

For complete details about elementary moves, see the related reference below about the MOVE statement.

The following example shows an alphanumeric data item in the Greek language that is moved to a national data item:

```
. . .
01 Data-in-Unicode   Pic N(100) usage national.
01 Data-in-Greek     Pic X(100).
. . .
    Read Greek-file into Data-in-Greek
    Move Data-in-Greek to Data-in-Unicode
```

Related concepts
“Unicode and the encoding
of language characters” on page 176

Related tasks
“Assigning values to group data items (MOVE)” on page 29
“Converting to or from national (Unicode) representation” on page 184

Related references
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
MOVE statement (*COBOL for Linux on x86 Language Reference*)

Assigning values to group data items (MOVE)

Use the MOVE statement to assign values to group data items.

You can move a national group item (a data item that is described with the GROUP-USAGE NATIONAL clause) to another national group item. The compiler processes the move as though each national group item were an elementary item of category national, that is, as if each item were described as PIC N(*m*), where *m* is the length of that item in national character positions.

You can move an alphanumeric group item to an alphanumeric group item or to a national group item. You can also move a national group item to an alphanumeric group item. The compiler performs such moves as group moves, that is, without consideration of the individual elementary items in the sending or receiving group, and without conversion of the sending data item. Be sure that the subordinate data descriptions in the sending and receiving group items are compatible. The moves occur even if a destructive overlap could occur at run time.

You can code the CORRESPONDING phrase in a MOVE statement to move subordinate elementary items from one group item to the identically named corresponding subordinate elementary items in another group item:

```
01  Group-X.
    02 T-Code    Pic X    Value "A".
    02 Month     Pic 99   Value 04.
    02 State     Pic XX   Value "CA".
    02 Filler    PIC X.
01  Group-N      Group-Usage National.
    02 State     Pic NN.
    02 Month     Pic 99.
    02 Filler    Pic N.
    02 Total     Pic 999.
. . .
    MOVE CORR Group-X TO Group-N
```

In the example above, State and Month within Group-N receive the values in national representation of State and Month, respectively, from Group-X. The other data items in Group-N are unchanged. (Filler items in a receiving group item are unchanged by a MOVE CORRESPONDING statement.)

In a MOVE CORRESPONDING statement, sending and receiving group items are treated as group items, not as elementary data items; group semantics apply. That is, the elementary data items within each group are recognized, and the results are the same as if each pair of corresponding data items were referenced in a separate MOVE statement. Data conversions are performed according to the rules for the MOVE statement as specified in the related reference below. For details about which types of elementary data items correspond, see the related reference about the CORRESPONDING phrase.

Related concepts
“Unicode and the encoding
of language characters” on page 176
“National groups” on page 183

Related tasks
“Assigning values to elementary

data items (MOVE)” on page 28
“Using national groups” on page 187
“Converting to or from national (Unicode) representation” on page 184

Related references
Classes and categories of group items (*COBOL for Linux on x86 Language Reference*)
MOVE statement (*COBOL for Linux on x86 Language Reference*)
CORRESPONDING phrase (*COBOL for Linux on x86 Language Reference*)

Assigning arithmetic results (MOVE or COMPUTE)

When assigning a number to a data item, consider using the COMPUTE statement instead of the MOVE statement.

```
Move w to z
Compute z = w
```

In the example above, the two statements in most cases have the same effect. The MOVE statement however carries out the assignment with truncation. You can use the DIAGTRUNC compiler option to request that the compiler issue a warning for MOVE statements that might truncate numeric receivers.

When significant left-order digits would be lost in execution, the COMPUTE statement can detect the condition and allow you to handle it. If you use the ON SIZE ERROR phrase of the COMPUTE statement, the compiler generates code to detect a size-overflow condition. If the condition occurs, the code in the ON SIZE ERROR phrase is performed, and the content of z remains unchanged. If you do not specify the ON SIZE ERROR phrase, the assignment is carried out with truncation. There is no ON SIZE ERROR support for the MOVE statement.

You can also use the COMPUTE statement to assign the result of an arithmetic expression or intrinsic function to a data item. For example:

```
Compute z = y + (x ** 3)
Compute x = Function Max(x y z)
```

Related references
“DIAGTRUNC” on page 261
Intrinsic functions (*COBOL for Linux on x86 Language Reference*)

Assigning input from a screen or file (ACCEPT)

One way to assign a value to a data item is to read the value from a screen or a file.

To enter data from the screen, first associate the monitor with a mnemonic-name in the SPECIAL-NAMES paragraph. Then use ACCEPT to assign the line of input entered at the screen to a data item. For example:

```
Environment Division.
Configuration Section.
Special-Names.
    Console is Names-Input.
. . .
    Accept Customer-Name From Names-Input
```

To read from a file instead of the screen, make either of the following changes:

- Change Console to *device*, where *device* is any valid system device (for example, SYSIN). For example:

```
SYSIN is Names-Input
```

- Set the environment variable CONSOLE to a valid file specification by using the `export` command. For example:

```
export CONSOLE=/myfiles/myinput.rpt
```

  The environment-variable name must be the same as the system device name used. In the example above, the system device is `Console`, but the method of assigning an environment variable to the system device name is supported for all valid system devices. For example, if the system device is `SYSIN`, the environment variable that must be assigned a file specification is also SYSIN.

The ACCEPT statement assigns the input line to the data item. If the input line is shorter than the data item, the data item is padded with spaces of the appropriate representation. When you read from a screen and the input line is longer than the data item, the remaining characters are discarded. When you read from a file and the input line is longer than the data item, the remaining characters are retained as the next input line for the file.

When you use the ACCEPT statement, you can assign a value to an alphanumeric or national group item, or to an elementary data item that has `USAGE DISPLAY`, `USAGE DISPLAY-1`, or `USAGE NATIONAL`.

When you assign a value to a `USAGE NATIONAL` data item, the input data is converted from the code page associated with the current runtime locale to national (Unicode UTF-16) representation only if the input is from the terminal.

To have conversion done when the input data is from any other device, use the `NATIONAL-OF` intrinsic function.

Related concepts
“Unicode and the encoding of language characters” on page 176

Related tasks
“Setting environment variables” on page 213
“Converting alphanumeric or DBCS to national (NATIONAL-OF)” on page 185
“Getting the system date under CICS” on page 331

Related references
ACCEPT statement (*COBOL for Linux on x86 Language Reference*)
SPECIAL-NAMES paragraph (*COBOL for Linux on x86 Language Reference*)

Displaying values on a screen or in a file (DISPLAY)

You can display the value of a data item on a screen or write it to a file by using the `DISPLAY` statement.

```
Display "No entry for surname '" Customer-Name "' found in the file.".
```

In the example above, if the content of data item *Customer-Name* is JOHNSON, then the statement displays the following message on the screen:

```
No entry for surname 'JOHNSON' found in the file.
```

To write data to a destination other than the screen, use the UPON phrase. For example, the following statement writes to the file that you specify as the value of the SYSOUT environment variable:

```
Display "Hello" upon sysout.
```

When you display the value of a `USAGE NATIONAL` data item, the output data is converted to the code page that is associated with the current locale.

Related concepts
"Unicode and the encoding of language characters" on page 176

Related tasks
"Converting national to alphanumeric (DISPLAY-OF)" on page 186
"Coding COBOL programs to run under CICS" on page 330

Related references
"Runtime environment variables" on page 218
DISPLAY statement (*COBOL for Linux on x86 Language Reference*)

Using intrinsic functions (built-in functions)

Some high-level programming languages have built-in functions that you can reference in your program as if they were variables that have defined attributes and a predetermined value. In COBOL, these functions are called *intrinsic functions*. They provide capabilities for manipulating strings and numbers.

Because the value of an intrinsic function is derived automatically at the time of reference, you do not need to define functions in the DATA DIVISION. Define only the nonliteral data items that you use as arguments. Figurative constants are not allowed as arguments.

A *function-identifier* is the combination of the COBOL reserved word FUNCTION followed by a function name (such as Max), followed by any arguments to be used in the evaluation of the function (such as x, y, z). (Optionally, the reserved word FUNCTION may be omitted if the function name is referenced in the REPOSITORY paragraph.) For example, the groups of highlighted words below are function-identifiers:

```
Unstring Function Upper-case(Name) Delimited By Space
    Into Fname Lname
Compute A = 1 + Function Log10(x)
Compute M = Function Max(x y z)
```

A function-identifier represents both the invocation of the function and the data value returned by the function. Because it actually represents a data item, you can use a function-identifier in most places in the PROCEDURE DIVISION where a data item that has the attributes of the returned value can be used.

The COBOL word function is a reserved word, but the function-names are not reserved. You can use them in other contexts, such as for the name of a data item. For example, you could use Sqrt to invoke an intrinsic function and to name a data item in your program:

```
Working-Storage Section.
01  x                 Pic 99  value 2.
01  y                 Pic 99  value 4.
01  z                 Pic 99  value 0.
01  Sqrt              Pic 99  value 0.
. . .
    Compute Sqrt = 16 ** .5
    Compute z = x + Function Sqrt(y)
    . . .
```

A function-identifier represents a value that is of one of these types: alphanumeric, national, numeric, or integer. You can include a substring specification (reference modifier) in a function-identifier for alphanumeric or national functions. Numeric intrinsic functions are further classified according to the type of numbers they return.

The functions MAX, MIN, DATEVAL, and UNDATE can return either type of value depending on the type of arguments you supply.

The functions DATEVAL, UNDATE, and YEARWINDOW are provided with the millennium language extensions to assist with manipulating and converting windowed date fields.

Functions can reference other functions as arguments provided that the results of the nested functions meet the requirements for the arguments of the outer function. For example, `Function Sqrt(5)` returns a numeric value. Thus, the three arguments to the MAX function below are all numeric, which is an allowable argument type for this function:

```
Compute x = Function Max((Function Sqrt(5)) 2.5 3.5)
```

Related tasks
"Processing table items using intrinsic functions" on page 79
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 106

Using tables (arrays) and pointers

In COBOL, arrays are called *tables*. A table is a set of logically consecutive data items that you define in the `DATA DIVISION` by using the `OCCURS` clause.

Pointers are data items that contain virtual storage addresses. You define them either explicitly with the `USAGE IS POINTER` clause in the `DATA DIVISION` or implicitly as `ADDRESS OF` special registers.

You can perform the following operations with pointer data items:

- Pass them between programs by using the `CALL . . . BY REFERENCE` statement.
- Set a pointer to allocated storage or free storage by using the `ALLOCATE` and `FREE` statements.
- Move them to other pointers by using the `SET` statement.
- Compare them to other pointers for equality by using a relation condition.
- Initialize them to contain an invalid address by using `VALUE IS NULL`.

Use pointer data items to:

- Accomplish limited base addressing, particularly if you want to pass and receive addresses of a record area that is defined with `OCCURS DEPENDING ON` and is therefore variably located.
- Handle a chained list.

Related tasks
"Defining a table (OCCURS)" on page 59
"Using procedure and function pointers" on page 402

Chapter 3. Working with numbers and arithmetic

In general, you can view COBOL numeric data as a series of decimal digit positions. However, numeric items can also have special properties such as an arithmetic sign or a currency sign.

To define, display, and store numeric data so that you can perform arithmetic operations efficiently:

- Use the PICTURE clause and the characters 9, +, -, P, S, and V to define numeric data.
- Use the PICTURE clause and editing characters (such as Z, comma, and period) along with MOVE and DISPLAY statements to display numeric data.
- Use the USAGE clause with various formats to control how numeric data is stored.
- Use the numeric class test to validate that data values are appropriate.
- Use ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements to perform arithmetic.
- Use the CURRENCY SIGN clause and appropriate PICTURE characters to designate the currency you want.

Related tasks
"Defining numeric data" on page 35
"Displaying numeric data" on page 37
"Controlling how numeric data is stored" on page 38
"Checking for incompatible data (numeric class test)" on page 48
"Performing arithmetic" on page 48
"Using currency signs" on page 56

Defining numeric data

Define numeric items by using the PICTURE clause with the character 9 in the data description to represent the decimal digits of the number. Do not use an X, which is for alphanumeric data items.

For example, Count-y below is a numeric data item, an external decimal item that has USAGE DISPLAY (a *zoned decimal item*):

```
05  Count-y        Pic 9(4)  Value 25.
05  Customer-name  Pic X(20) Value "Johnson".
```

You can similarly define numeric data items to hold national characters (UTF-16). For example, Count-n below is an external decimal data item that has USAGE NATIONAL (a *national decimal item*):

```
05  Count-n        Pic 9(4)  Value 25  Usage National.
```

You can code up to 18 digits in the PICTURE clause when you compile using the default compiler option ARITH(COMPAT) (referred to as *compatibility mode*). When you compile using ARITH(EXTEND) (referred to as *extended mode*), you can code up to 31 digits in the PICTURE clause.

Other characters of special significance that you can code are:

P
Indicates leading or trailing zeros

S
Indicates a sign, positive or negative

V
Implies a decimal point

© Copyright IBM Corp. 2021

The s in the following example means that the value is signed:

```
05  Price  Pic s99v99.
```

The field can therefore hold a positive or a negative value. The v indicates the position of an implied decimal point, but does not contribute to the size of the item because it does not require a storage position. An s usually does not contribute to the size of a numeric item, because by default s does not require a storage position.

However, if you plan to port your program or data to a different machine, you might want to code the sign for a zoned decimal data item as a separate position in storage. In the following case, the sign takes 1 byte:

```
05  Price  Pic s99V99  Sign Is Leading, Separate.
```

This coding ensures that the convention your machine uses for storing a nonseparate sign will not cause unexpected results on a machine that uses a different convention.

Separate signs are also preferable for zoned decimal data items that will be printed or displayed.

Separate signs are required for national decimal data items that are signed. The sign takes 2 bytes of storage, as in the following example:

```
05 Price Pic s99V99 Usage National Sign Is Leading, Separate.
```

You cannot use the PICTURE clause with internal floating-point data (COMP-1 or COMP-2). However, you can use the VALUE clause to provide an initial value for an internal floating-point literal:

```
05  Compute-result  Usage Comp-2  Value 06.23E-24.
```

For information about external floating-point data, see the examples referenced below and the related concept about formats for numeric data.

“Examples: numeric data and internal representation” on page 42

Related concepts
“Formats for numeric data” on page 39
Appendix C, “Intermediate results and arithmetic precision,” on page 477

Related tasks
“Displaying numeric data” on page 37
“Controlling how numeric data is stored” on page 38
“Performing arithmetic” on page 48
“Defining national numeric data items” on page 183

Related references
“Sign representation of zoned and packed-decimal data” on page 47
“Storage of character data” on page 190
“ARITH” on page 251
SIGN clause (*COBOL for Linux on x86 Language Reference*)

Displaying numeric data

You can define numeric items with certain editing symbols (such as decimal points, commas, dollar signs, and debit or credit signs) to make the items easier to read and understand when you display or print them.

For example, in the code below, `Edited-price` is a numeric-edited item that has `USAGE DISPLAY`. (You can specify the clause `USAGE IS DISPLAY` for numeric-edited items; however, it is implied. It means that the items are stored in character format.)

```
05  Price          Pic     9(5)v99.
05  Edited-price   Pic  $zz,zz9.99.
. . .
Move Price To Edited-price
Display Edited-price
```

If the contents of `Price` are 0150099 (representing the value 1,500.99), `$ 1,500.99` is displayed when you run the code. The `z` in the `PICTURE` clause of `Edited-price` indicates the suppression of leading zeros.

You can define numeric-edited data items to hold national (UTF-16) characters instead of alphanumeric characters. To do so, define the numeric-edited items as `USAGE NATIONAL`. The effect of the editing symbols is the same for numeric-edited items that have `USAGE NATIONAL` as it is for numeric-edited items that have `USAGE DISPLAY`, except that the editing is done with national characters. For example, if `Edited-price` is declared as `USAGE NATIONAL` in the code above, the item is edited and displayed using national characters.

You can cause an elementary numeric or numeric-edited item to be filled with spaces when a value of zero is stored into it by coding the `BLANK WHEN ZERO` clause for the item. For example, each of the `DISPLAY` statements below causes blanks to be displayed instead of zeros:

```
05 Price           Pic     9(5)v99.
05 Edited-price-D  Pic  $99,999.99
       Blank When Zero.
05 Edited-price-N  Pic  $99,999.99 Usage National
       Blank When Zero.
. . .
Move 0 to Price
Move Price to Edited-price-D
Move Price to Edited-price-N
Display Edited-price-D
Display Edited-price-N
```

You cannot use numeric-edited items as sending operands in arithmetic expressions or in `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, or `COMPUTE` statements. (Numeric editing takes place when a numeric-edited item is the receiving field for one of these statements, or when a `MOVE` statement has a numeric-edited receiving field and a numeric-edited or numeric sending field.) You use numeric-edited items primarily for displaying or printing numeric data.

You can move numeric-edited items to numeric or numeric-edited items. In the following example, the value of the numeric-edited item (whether it has `USAGE DISPLAY` or `USAGE NATIONAL`) is moved to the numeric item:

```
Move Edited-price to Price
Display Price
```

If these two statements immediately followed the statements in the first example above, then `Price` would be displayed as 0150099, representing the value 1,500.99. `Price` would also be displayed as 0150099 if `Edited-price` had `USAGE NATIONAL`.

You can also move numeric-edited items to alphanumeric, alphanumeric-edited, floating-point, and national data items. For a complete list of the valid receiving items for numeric-edited data, see the related reference about the MOVE statement.

"Examples: numeric data and internal representation" on page 42

Related tasks
"Displaying values on a screen or in a file (DISPLAY)" on page 31
"Controlling how numeric data is stored" on page 38
"Defining numeric data" on page 35
"Performing arithmetic" on page 48
"Defining national numeric data items" on page 183
"Converting to or from national (Unicode) representation" on page 184

Related references
MOVE statement (*COBOL for Linux on x86 Language Reference*)
BLANK WHEN ZERO clause (*COBOL for Linux on x86 Language Reference*)

Controlling how numeric data is stored

You can control how the computer stores numeric data items by coding the USAGE clause in your data description entries.

You might want to control the format for any of several reasons such as these:

- Arithmetic performed with computational data types is more efficient than with USAGE DISPLAY or USAGE NATIONAL data types.
- Packed-decimal format requires less storage per digit than USAGE DISPLAY or USAGE NATIONAL data types.
- Packed-decimal format converts to and from DISPLAY or NATIONAL format more efficiently than binary format does.
- Floating-point format is well suited for arithmetic operands and results with widely varying scale, while maintaining the maximal number of significant digits.
- You might need to preserve data formats when you move data from one machine to another.

The numeric data you use in your program will have one of the following formats available with COBOL:

- External decimal (USAGE DISPLAY or USAGE NATIONAL)
- External floating point (USAGE DISPLAY or USAGE NATIONAL)
- Internal decimal (USAGE PACKED-DECIMAL)
- Binary (USAGE BINARY)
- Native binary (USAGE COMP-5)
- Internal floating point (USAGE COMP-1 or USAGE COMP-2)

COMP and COMP-4 are synonymous with BINARY, and COMP-3 is synonymous with PACKED-DECIMAL.

The compiler converts displayable numbers to the internal representation of their numeric values before using them in arithmetic operations. Therefore it is often more efficient if you define data items as BINARY or PACKED-DECIMAL than as DISPLAY or NATIONAL. For example:

```
05  Initial-count   Pic S9(4)  Usage Binary   Value 1000.
```

Regardless of which USAGE clause you use to control the internal representation of a value, you use the same PICTURE clause conventions and decimal value in the VALUE clause (except for internal floating-point data, for which you cannot use a PICTURE clause).

“Examples: numeric data and internal representation” on page 42

Related concepts
“Formats for numeric data” on page 39
“Data format conversions” on page 46
Appendix C, “Intermediate results and arithmetic precision,” on page 477

Related tasks
“Defining numeric data” on page 35
“Displaying numeric data” on page 37
“Performing arithmetic” on page 48

Related references
“Conversions and precision” on page 47
“Sign representation of zoned and packed-decimal data” on page 47

Formats for numeric data

Several formats are available for numeric data.

External decimal (DISPLAY and NATIONAL) items

When USAGE DISPLAY is in effect for a category numeric data item (either because you have coded it, or by default), each position (byte) of storage contains one decimal digit. The items are stored in displayable form. External decimal items that have USAGE DISPLAY are referred to as *zoned decimal* data items.

When USAGE NATIONAL is in effect for a category numeric data item, 2 bytes of storage are required for each decimal digit. The items are stored in UTF-16 format. External decimal items that have USAGE NATIONAL are referred to as *national decimal* data items.

National decimal data items, if signed, must have the SIGN SEPARATE clause in effect. All other rules for zoned decimal items apply to national decimal items. You can use national decimal items anywhere that other category numeric data items can be used.

External decimal (both zoned decimal and national decimal) data items are primarily intended for receiving and sending numbers between your program and files, terminals, or printers. You can also use external decimal items as operands and receivers in arithmetic processing. However, if your program performs a lot of intensive arithmetic, and efficiency is a high priority, COBOL's computational numeric types might be a better choice for the data items used in the arithmetic.

External floating-point (DISPLAY and NATIONAL) items

When USAGE DISPLAY is in effect for a floating-point data item (either because you have coded it, or by default), each PICTURE character position (except for v, an implied decimal point, if used) takes 1 byte of storage. The items are stored in displayable form. External floating-point items that have USAGE DISPLAY are referred to as *display floating-point* data items in this information when necessary to distinguish them from external floating-point items that have USAGE NATIONAL.

In the following example, Compute-Result is implicitly defined as a display floating-point item:

```
05  Compute-Result  Pic -9v9(9)E-99.
```

The minus signs (-) do not mean that the mantissa and exponent must necessarily be negative numbers. Instead, they mean that when the number is displayed, the sign appears as a blank for positive numbers or a minus sign for negative numbers. If you instead code a plus sign (+), the sign appears as a plus sign for positive numbers or a minus sign for negative numbers.

When USAGE NATIONAL is in effect for a floating-point data item, each PICTURE character position (except for v, if used) takes 2 bytes of storage. The items are stored as national characters (UTF-16). External floating-point items that have USAGE NATIONAL are referred to as *national floating-point* data items.

The existing rules for display floating-point items apply to national floating-point items.

In the following example, Compute-Result-N is a national floating-point item:

```
05  Compute-Result-N  Pic -9v9(9)E-99  Usage National.
```

If Compute-Result-N is displayed, the signs appear as described above for Compute-Result, but in national characters.

You cannot use the VALUE clause for external floating-point items.

As with external decimal numbers, external floating-point numbers have to be converted (by the compiler) to an internal representation of their numeric value before they can be used in arithmetic operations. If you compile with the default option ARITH (COMPAT), external floating-point numbers are converted to long (64-bit) floating-point format. If you compile with ARITH (EXTEND), they are instead converted to extended-precision (128-bit) floating-point format.

Binary (COMP) items

BINARY, COMP, and COMP-4 are synonyms. Binary-format numbers occupy 2, 4, or 8 bytes of storage. If the PICTURE clause specifies that an item is signed, the leftmost bit is used as the operational sign.

A binary number with a PICTURE description of four or fewer decimal digits occupies 2 bytes; five to nine decimal digits, 4 bytes; and 10 to 18 decimal digits, 8 bytes. Binary items with nine or more digits require more handling by the compiler.

You can use binary items, for example, for indexes, subscripts, switches, and arithmetic operands or results.

Use the TRUNC(STD|OPT|BIN) compiler option to indicate how binary data (BINARY, COMP, or COMP-4) is to be truncated.

Native binary (COMP-5) items

Data items that you define as USAGE COMP-5 are represented in storage as binary data. However, unlike USAGE COMP items, they can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) rather than being limited to the value implied by the number of 9s in the PICTURE clause.

When you move or store numeric data into a COMP-5 item, truncation occurs at the binary field size rather than at the COBOL PICTURE size limit. When you reference a COMP-5 item, the full binary field size is used in the operation.

COMP-5 is thus particularly useful for binary data items that originate in non-COBOL programs where the data might not conform to a COBOL PICTURE clause.

The table below shows the ranges of possible values for COMP-5 data items.

Table 3. ***Ranges in value of COMP-5 data items***

PICTURE	**Storage representation**	**Numeric values**
S9(1) through S9(4)	Binary halfword (2 bytes)	-32768 through +32767
S9(5) through S9(9)	Binary fullword (4 bytes)	-2,147,483,648 through +2,147,483,647
S9(10) through S9(18)	Binary doubleword (8 bytes)	-9,223,372,036,854,775,808 through +9,223,372,036,854,775,807

*Table 3. **Ranges in value of COMP-5 data items** (continued)*		
PICTURE	**Storage representation**	**Numeric values**
9(1) through 9(4)	Binary halfword (2 bytes)	0 through 65535
9(5) through 9(9)	Binary fullword (4 bytes)	0 through 4,294,967,295
9(10) through 9(18)	Binary doubleword (8 bytes)	0 through 18,446,744,073,709,551,615

You can specify scaling (that is, decimal positions or implied integer positions) in the PICTURE clause of COMP-5 items. If you do so, you must appropriately scale the maximal capacities listed above. For example, a data item you describe as PICTURE S99V99 COMP-5 is represented in storage as a binary halfword, and supports a range of values from -327.68 through +327.67.

Large literals in VALUE clauses: Literals specified in VALUE clauses for COMP-5 items can, with a few exceptions, contain values of magnitude up to the capacity of the native binary representation. See *COBOL for Linux on x86 Language Reference* for the exceptions.

Regardless of the setting of the TRUNC compiler option, COMP-5 data items behave like binary data does in programs compiled with TRUNC(BIN).

Packed-decimal (COMP-3) items

PACKED-DECIMAL and COMP-3 are synonyms. Packed-decimal items occupy 1 byte of storage for every two decimal digits you code in the PICTURE description, except that the rightmost byte contains only one digit and the sign. This format is most efficient when you code an odd number of digits in the PICTURE description, so that the leftmost byte is fully used. Packed-decimal items are handled as fixed-point numbers for arithmetic purposes.

Internal floating-point (COMP-1 and COMP-2) items

COMP-1 refers to short floating-point format and COMP-2 refers to long floating-point format, which occupy 4 and 8 bytes of storage, respectively.

COMP-1 and COMP-2 data items are represented in IEEE format if the FLOAT(NATIVE) compiler option (the default) is in effect. If FLOAT(BE) is in effect, COMP-1 and COMP-2 data items are represented consistently with System z®, that is, in hexadecimal floating-point format. For details, see the related reference about the FLOAT option.

Related concepts
“Unicode and the encoding of language characters” on page 176
Appendix C, “Intermediate results and arithmetic precision,” on page 477

Related tasks
“Defining numeric data” on page 35
“Defining national numeric data items” on page 183

Related references
“Storage of character data” on page 190
“TRUNC” on page 283
“FLOAT” on page 267
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
SIGN clause (*COBOL for Linux on x86 Language Reference*)
VALUE clause (*COBOL for Linux on x86 Language Reference*)

Examples: numeric data and internal representation

The following tables show the internal representation of numeric items.

The following table shows the internal representation of numeric items for binary data types.

Table 4. ***Internal representation of binary numeric items***

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
Binary	PIC S9999 BINARY PIC S9999 COMP PIC S9999 COMP-4	+ 1234	D2 04
		- 1234	2E FB
	PIC S9999 COMP-5	+ 12345[1]	39 30
		- 12345[1]	C7 CF
	PIC 9999 BINARY PIC 9999 COMP PIC 9999 COMP-4	1234	D2 04
	PIC 9999 COMP-5	60000[1]	60 EA

1. The example demonstrates that COMP-5 data items can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes), rather than being limited to the value implied by the number of 9s in the PICTURE clause.

The following table shows the internal representation of numeric items in native data format. Assume that the CHAR(NATIVE) and FLOAT(NATIVE) compiler options are in effect.

Table 5. ***Internal representation of native numeric items***

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234	31 32 33 34
		- 1234	31 32 33 74
		1234	31 32 33 34
	PIC 9999 DISPLAY	1234	31 32 33 34
	PIC 9999 NATIONAL	1234	31 00 32 00 33 00 34 00
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	31 32 33 34
		- 1234	71 32 33 34
	PIC S9999 DISPLAY SIGN LEADING SEPARATE	+ 1234	2B 31 32 33 34
		- 1234	2D 31 32 33 34
	PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	31 32 33 34 2B
		- 1234	31 32 33 34 2D
	PIC S9999 NATIONAL SIGN LEADING SEPARATE	+ 1234	2B 00 31 00 32 00 33 00 34 00
		- 1234	2D 00 31 00 32 00 33 00 34 00
	PIC S9999 NATIONAL SIGN TRAILING SEPARATE	+ 1234	31 00 32 00 33 00 34 00 2B 00
		- 1234	31 00 32 00 33 00 34 00 2D 00
Internal decimal	PIC S9999 PACKED-DECIMAL PIC S9999 COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL PIC 9999 COMP-3	1234	01 23 4C
Internal floating point	COMP-1	+ 1234	00 40 9A 44
		- 1234	00 40 9A C4
	COMP-2	+ 1234	00 00 00 00 00 48 93 40
		- 1234	00 00 00 00 00 48 93 C0

*Table 5. **Internal representation of native numeric items** (continued)*			
Numeric type	**PICTURE and USAGE and optional SIGN clause**	**Value**	**Internal representation**
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 12.34E+02	2B 31 32 2E 33 34 45 2B 30 32
		- 12.34E+02	2D 31 32 2E 33 34 45 2B 30 32
	PIC +9(2).9(2)E+99 NATIONAL	+ 12.34E+02	2B 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32 00
		- 12.34E+02	2D 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32 00

The following table shows the internal representation of numeric items in IBM Z host data format. Assume that the CHAR(EBCDIC) and FLOAT(BE) compiler options are in effect.

Table 6. ***Internal representation of numeric items when CHAR(EBCDIC) and FLOAT(BE) are in effect***

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234	F1 F2 F3 C4
		- 1234	F1 F2 F3 D4
		1234	F1 F2 F3 C4
	PIC 9999 DISPLAY	1234	F1 F2 F3 F4
	PIC 9999 NATIONAL	1234	00 31 00 32 00 33 00 34
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	C1 F2 F3 F4
		- 1234	D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE	+ 1234	4E F1 F2 F3 F4
		- 1234	60 F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	F1 F2 F3 F4 4E
		- 1234	F1 F2 F3 F4 60
	PIC S9999 NATIONAL SIGN LEADING SEPARATE	+ 1234	00 2B 00 31 00 32 00 33 00 34
		- 1234	00 2D 00 31 00 32 00 33 00 34
	PIC S9999 NATIONAL SIGN TRAILING SEPARATE	+ 1234	00 31 00 32 00 33 00 34 00 2B
		- 1234	00 31 00 32 00 33 00 34 00 2D
Internal decimal	PIC S9999 PACKED-DECIMAL PIC S9999 COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL PIC 9999 COMP-3	1234	01 23 4C
Internal floating point	COMP-1	+ 1234	43 4D 20 00
		- 1234	C3 4D 20 00
	COMP-2	+ 1234	43 4D 20 00 00 00 00 00
		- 1234	C3 4D 20 00 00 00 00 00

Table 6. ***Internal representation of numeric items when CHAR(EBCDIC) and FLOAT(BE) are in effect*** *(continued)*

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 12.34E+02	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 12.34E+02	60 F1 F2 4B F3 F4 C5 4E F0 F2
	PIC +9(2).9(2)E+99 NATIONAL	+ 12.34E+02	00 2B 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32
		- 12.34E+02	00 2D 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32

Data format conversions

When the code in your program involves the interaction of items that have different data formats, the compiler converts those items either temporarily, for comparisons and arithmetic operations, or permanently, for assignment to the receiver in a MOVE, COMPUTE, or other arithmetic statement.

A conversion is actually a move of a value from one data item to another. The compiler performs any conversions that are required during the execution of arithmetic or comparisons by using the same rules that are used for MOVE and COMPUTE statements.

When possible, the compiler performs a move to preserve numeric value instead of a direct digit-for-digit move.

Conversion generally requires additional storage and processing time because data is moved to an internal work area and converted before the operation is performed. The results might also have to be moved back into a work area and converted again.

Conversions between fixed-point data formats (external decimal, packed decimal, or binary) are without loss of precision provided that the target field can contain all the digits of the source operand.

A loss of precision is possible in conversions between fixed-point data formats and floating-point data formats (short floating point, long floating point, or external floating point). These conversions happen during arithmetic evaluations that have a mixture of both fixed-point and floating-point operands.

Related references
“Conversions and precision” on page 47
“Sign representation of zoned and packed-decimal data” on page 47

Conversions and precision

In some numeric conversions, a loss of precision is possible; other conversions preserve precision or result in rounding.

Because both fixed-point and external floating-point items have decimal characteristics, references to fixed-point items in the following examples include external floating-point items unless stated otherwise.

When the compiler converts from fixed-point to internal floating-point format, fixed-point numbers in base 10 are converted to the numbering system used internally.

When the compiler converts short form to long form for comparisons, zeros are used for padding the shorter number.

Conversions that lose precision

When a USAGE COMP-2 data item is moved to a fixed-point data item that has more than 18 digits, the fixed-point data item will receive only 18 significant digits, and the remaining digits will be zero.

When a USAGE COMP-1 data item is moved to a fixed-point data item that has more than six digits, the fixed-point data item will receive only six significant digits, and the remaining digits will be zero.

Conversions that preserve precision

If a fixed-point data item that has six or fewer digits is moved to a USAGE COMP-1 data item and then returned to the fixed-point data item, the original value is recovered.

If a USAGE COMP-1 data item is moved to a fixed-point data item of six or more digits and then returned to the USAGE COMP-1 data item, the original value is recovered.

If a fixed-point data item that has 15 or fewer digits is moved to a USAGE COMP-2 data item and then returned to the fixed-point data item, the original value is recovered.

If a USAGE COMP-2 data item is moved to a fixed-point (not external floating-point) data item of 18 or more digits and then returned to the USAGE COMP-2 data item, the original value is recovered.

Conversions that result in rounding

If a USAGE COMP-1 data item, a USAGE COMP-2 data item, an external floating-point data item, or a floating-point literal is moved to a fixed-point data item, rounding occurs in the low-order position of the target data item. Fractional values .5 and greater are rounded up; fractional values less than .5 are rounded down.

If a USAGE COMP-2 data item is moved to a USAGE COMP-1 data item, rounding occurs in the low-order position of the target data item.

If a fixed-point data item is moved to an external floating-point data item and the PICTURE of the fixed-point data item contains more digit positions than the PICTURE of the external floating-point data item, rounding occurs in the low-order position of the target data item.

Related concepts
Appendix C, “Intermediate results and arithmetic precision,” on page 477

Sign representation of zoned and packed-decimal data

Sign representation affects the processing and interaction of zoned decimal and internal decimal data.

Given X'*sd*', where *s* is the sign representation and *d* represents the digit, the valid sign representations for zoned decimal (USAGE DISPLAY) data without the SIGN IS SEPARATE clause are:

Positive:
3, C, and F

Negative:
7 and D

When the CHAR(NATIVE) compiler option is in effect, signs generated internally are 3 for positive and unsigned, and 7 for negative.

When the CHAR(EBCDIC) compiler option is in effect, signs generated internally are C for positive, F for unsigned, and D for negative.

Given X'*ds*', where *d* represents the digit and *s* is the sign representation, the valid sign representations for internal decimal (USAGE PACKED-DECIMAL) data are:

Positive:
A, C, E, and F

Negative:
B and D

Signs generated internally are C for positive and unsigned, and D for negative.

The sign representation of unsigned internal decimal numbers is different between COBOL for Linux and host COBOL. Host COBOL generates F internally as the sign of unsigned internal decimal numbers.

Related references
"ZWB" on page 289
"Data representation" on page 467

Checking for incompatible data (numeric class test)

The compiler assumes that values you supply for a data item are valid for the PICTURE and USAGE clauses, and does not check their validity. Ensure that the contents of a data item conform to the PICTURE and USAGE clauses before using the item in additional processing.

It can happen that values are passed into your program and assigned to items that have incompatible data descriptions for those values. For example, nonnumeric data might be moved or passed into a field that is defined as numeric, or a signed number might be passed into a field that is defined as unsigned. In either case, the receiving fields contain invalid data. When you give an item a value that is incompatible with its data description, references to that item in the PROCEDURE DIVISION are undefined and your results are unpredictable.

You can use the numeric class test to perform data validation. For example:

```
Linkage Section.
01  Count-x   Pic 999.
. . .
Procedure Division Using Count-x.
    If Count-x is numeric then display "Data is good"
```

The numeric class test checks the contents of a data item against a set of values that are valid for the PICTURE and USAGE of the data item.

Performing arithmetic

You can use any of several COBOL language features (including COMPUTE, arithmetic expressions, numeric intrinsic functions, and date/time callable services) to perform arithmetic. Your choice depends on whether a feature meets your particular needs.

For most common arithmetic evaluations, the COMPUTE statement is appropriate. If you need to use numeric literals, numeric data, or arithmetic operators, you might want to use arithmetic expressions. In places where numeric expressions are allowed, you can save time by using numeric intrinsic functions.

Related tasks
"Using COMPUTE and other arithmetic statements" on page 49
"Using arithmetic expressions" on page 50
"Using numeric intrinsic functions" on page 50

Using COMPUTE and other arithmetic statements

Use the COMPUTE statement for most arithmetic evaluations rather than ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. Often you can code only one COMPUTE statement instead of several individual arithmetic statements.

The COMPUTE statement assigns the result of an arithmetic expression to one or more data items:

```
Compute z     = a + b / c ** d - e
Compute x y z = a + b / c ** d - e
```

Some arithmetic calculations might be more intuitive using arithmetic statements other than COMPUTE. For example:

COMPUTE	Equivalent arithmetic statements
`Compute Increment = Increment + 1`	`Add 1 to Increment`
`Compute Balance =` `  Balance - Overdraft`	`Subtract Overdraft from Balance`

`Compute IncrementOne =`
`  IncrementOne + 1`
`Compute IncrementTwo =`
`  IncrementTwo + 1`
`Compute IncrementThree =`
`  IncrementThree + 1`
	`Add 1 to IncrementOne,`
	`  IncrementTwo,`
	`  IncrementThree`

You might also prefer to use the DIVIDE statement (with its REMAINDER phrase) for division in which you want to process a remainder. The REM intrinsic function also provides the ability to process a remainder.

When you perform arithmetic calculations, you can use national decimal data items as operands just as you use zoned decimal data items. You can also use national floating-point data items as operands just as you use display floating-point operands.

Related concepts
"Fixed-point contrasted with floating-point arithmetic" on page 53
Appendix C, "Intermediate results and arithmetic precision," on page 477

Related tasks
"Defining numeric data" on page 35

Using arithmetic expressions

You can use arithmetic expressions in many (but not all) places in statements where numeric data items are allowed.

For example, you can use arithmetic expressions as comparands in relation conditions:

```
If (a + b) > (c - d + 5) Then. . .
```

Arithmetic expressions can consist of a single numeric literal, a single numeric data item, or a single intrinsic function reference. They can also consist of several of these items connected by arithmetic operators.

Arithmetic operators are evaluated in the following order of precedence:

Table 7. ***Order of evaluation of arithmetic operators***

Operator	**Meaning**	**Order of evaluation**
Unary + or -	Algebraic sign	First
**	Exponentiation	Second
/ or *	Division or multiplication	Third
Binary + or -	Addition or subtraction	Last

Operators at the same level of precedence are evaluated from left to right; however, you can use parentheses to change the order of evaluation. Expressions in parentheses are evaluated before the individual operators are evaluated. Parentheses, whether necessary or not, make your program easier to read.

Related concepts
“Fixed-point contrasted with floating-point arithmetic” on page 53
Appendix C, “Intermediate results and arithmetic precision,” on page 477

Using numeric intrinsic functions

You can use numeric intrinsic functions only in places where numeric expressions are allowed. These functions can save you time because you don't have to code the many common types of calculations that they provide.

Numeric intrinsic functions return a signed numeric value, and are treated as temporary numeric data items.

Numeric functions are classified into the following categories:

Integer
Those that return an integer

Floating point
Those that return a long (64-bit) or extended-precision (128-bit) floating-point value (depending on whether you compile using the default option `ARITH(COMPAT)` or using `ARITH(EXTEND)`)

Mixed
Those that return an integer, a floating-point value, or a fixed-point number with decimal places, depending on the arguments

You can use intrinsic functions to perform several different arithmetic operations, as outlined in the following table.

Table 8. ***Numeric intrinsic functions***

Number handling	Date and time	Finance	Mathematics	Statistics
LENGTH MAX MIN NUMVAL NUMVAL-C ORD-MAX ORD-MIN	ADD-DURATION CONVERT-DATE-TIME CURRENT-DATE DATE-OF-INTEGER DATE-TO-YYYYMMDD DATEVAL DAY-OF-INTEGER DAY-TO-YYYYDDD EXTRACT-DATE-TIME FIND-DURATION INTEGER-OF-DATE INTEGER-OF-DAY UNDATE SUBTRACT-DURATION TEST-DATE-TIME WHEN-COMPILED YEAR-TO-YYYY YEARWINDOW	ANNUITY PRESENT-VALUE	ACOS ASIN ATAN COS FACTORIAL INTEGER INTEGER-PART LOG LOG10 MOD REM SIN SQRT SUM TAN	MEAN MEDIAN MIDRANGE RANDOM RANGE STANDARD-DEVIATION VARIANCE

"Examples: numeric intrinsic functions" on page 51

You can reference one function as the argument of another. A nested function is evaluated independently of the outer function (except when the compiler determines whether a mixed function should be evaluated using fixed-point or floating-point instructions).

You can also nest an arithmetic expression as an argument to a numeric function. For example, in the statement below, there are three function arguments (a, b, and the arithmetic expression `(c / d)`):

```
Compute x = Function Sum(a b (c / d))
```

You can reference all the elements of a table (or array) as function arguments by using the `ALL` subscript.

You can also use the integer special registers as arguments wherever integer arguments are allowed.

Related concepts
"Fixed-point contrasted with floating-point arithmetic" on page 53
Appendix C, "Intermediate results and arithmetic precision," on page 477

Related references
"ARITH" on page 251

Examples: numeric intrinsic functions

The following examples and accompanying explanations show intrinsic functions in each of several categories.

Where the examples below show zoned decimal data items, national decimal items could instead be used. (Signed national decimal items, however, require that the `SIGN SEPARATE` clause be in effect.)

General number handling

Suppose you want to find the maximum value of two prices (represented below as alphanumeric items with dollar signs), put this value into a numeric field in an output record, and determine the length of the output record. You can use NUMVAL-C (a function that returns the numeric value of an alphanumeric or national literal, or an alphanumeric or national data item) and the MAX and LENGTH functions to do so:

```
01  X                   Pic 9(2).
01  Price1              Pic x(8)   Value "$8000".
01  Price2              Pic x(8)   Value "$2000".
01  Output-Record.
    05  Product-Name    Pic x(20).
    05  Product-Number  Pic 9(9).
    05  Product-Price   Pic 9(6).
. . .
Procedure Division.
    Compute Product-Price =
      Function Max (Function Numval-C(Price1) Function Numval-C(Price2))
    Compute X = Function Length(Output-Record)
```

Additionally, to ensure that the contents in Product-Name are in uppercase letters, you can use the following statement:

```
Move Function Upper-case (Product-Name) to Product-Name
```

Date and time

The following example shows how to calculate a due date that is 90 days from today. The first eight characters returned by the CURRENT-DATE function represent the date in a four-digit year, two-digit month, and two-digit day format (YYYYMMDD). The date is converted to its integer value; then 90 is added to this value and the integer is converted back to the YYYYMMDD format.

```
01  YYYYMMDD         Pic 9(8).
01  Integer-Form     Pic S9(9).
. . .
    Move Function Current-Date(1:8) to YYYYMMDD
    Compute Integer-Form = Function Integer-of-Date(YYYYMMDD)
    Add 90 to Integer-Form
    Compute YYYYMMDD = Function Date-of-Integer(Integer-Form)
    Display 'Due Date: ' YYYYMMDD
```

Finance

Business investment decisions frequently require computing the present value of expected future cash inflows to evaluate the profitability of a planned investment. The present value of an amount that you expect to receive at a given time in the future is that amount, which, if invested today at a given interest rate, would accumulate to that future amount.

For example, assume that a proposed investment of $1,000 produces a payment stream of $100, $200, and $300 over the next three years, one payment per year respectively. The following COBOL statements calculate the present value of those cash inflows at a 10% interest rate:

```
01  Series-Amt1      Pic 9(9)V99       Value 100.
01  Series-Amt2      Pic 9(9)V99       Value 200.
01  Series-Amt3      Pic 9(9)V99       Value 300.
01  Discount-Rate    Pic S9(2)V9(6)    Value .10.
01  Todays-Value     Pic 9(9)V99.
. . .
    Compute Todays-Value =
      Function
        Present-Value(Discount-Rate Series-Amt1 Series-Amt2 Series-Amt3)
```

You can use the ANNUITY function in business problems that require you to determine the amount of an installment payment (annuity) necessary to repay the principal and interest of a loan. The series of

payments is characterized by an equal amount each period, periods of equal length, and an equal interest rate each period. The following example shows how you can calculate the monthly payment required to repay a $15,000 loan in three years at a 12% annual interest rate (36 monthly payments, interest per month = .12/12):

```
01  Loan             Pic 9(9)V99.
01  Payment          Pic 9(9)V99.
01  Interest         Pic 9(9)V99.
01  Number-Periods   Pic 99.
. . .
    Compute Loan = 15000
    Compute Interest = .12
    Compute Number-Periods = 36
    Compute Payment =
      Loan * Function Annuity((Interest / 12) Number-Periods)
```

Mathematics

The following COBOL statement demonstrates that you can nest intrinsic functions, use arithmetic expressions as arguments, and perform previously complex calculations simply:

```
Compute Z = Function Log(Function Sqrt (2 * X + 1)) + Function Rem(X 2)
```

Here in the addend the intrinsic function REM (instead of a DIVIDE statement with a REMAINDER clause) returns the remainder of dividing X by 2.

Statistics

Intrinsic functions make calculating statistical information easier. Assume you are analyzing various city taxes and want to calculate the mean, median, and range (the difference between the maximum and minimum taxes):

```
01  Tax-S            Pic 99v999 value .045.
01  Tax-T            Pic 99v999 value .02.
01  Tax-W            Pic 99v999 value .035.
01  Tax-B            Pic 99v999 value .03.
01  Ave-Tax          Pic 99v999.
01  Median-Tax       Pic 99v999.
01  Tax-Range        Pic 99v999.
. . .
    Compute Ave-Tax    = Function Mean   (Tax-S Tax-T Tax-W Tax-B)
    Compute Median-Tax = Function Median (Tax-S Tax-T Tax-W Tax-B)
    Compute Tax-Range  = Function Range  (Tax-S Tax-T Tax-W Tax-B)
```

Related tasks

“Converting to numbers (NUMVAL, NUMVAL-C)” on page 105

Fixed-point contrasted with floating-point arithmetic

How you code arithmetic in a program (whether an arithmetic statement, an intrinsic function, an expression, or some combination of these nested within each other) determines whether the evaluation is done with floating-point or fixed-point arithmetic.

Many statements in a program could involve arithmetic. For example, each of the following types of COBOL statements requires some arithmetic evaluation:

- General arithmetic

```
compute report-matrix-col = (emp-count ** .5) + 1
add report-matrix-min to report-matrix-max giving report-matrix-tot
```

- Expressions and functions

```
compute report-matrix-col = function sqrt(emp-count) + 1
compute whole-hours       = function integer-part((average-hours) + 1)
```

- Arithmetic comparisons

```
if report-matrix-col <     function sqrt(emp-count) + 1
if whole-hours       not = function integer-part((average-hours) + 1)
```

Floating-point evaluations

In general, if your arithmetic coding has either of the characteristics listed below, it is evaluated in floating-point arithmetic:

- An operand or result field is floating point.

  An operand is floating point if you code it as a floating-point literal or if you code it as a data item that is defined as `USAGE COMP-1`, `USAGE COMP-2`, or external floating point (`USAGE DISPLAY` or `USAGE NATIONAL` with a floating-point `PICTURE`).

  An operand that is a nested arithmetic expression or a reference to a numeric intrinsic function results in floating-point arithmetic when any of the following conditions is true:

  – An argument in an arithmetic expression results in floating point.
  – The function is a floating-point function.
  – The function is a mixed function with one or more floating-point arguments.
- An exponent contains decimal places.

  An exponent contains decimal places if you use a literal that contains decimal places, give the item a `PICTURE` that contains decimal places, or use an arithmetic expression or function whose result has decimal places.

An arithmetic expression or numeric function yields a result that has decimal places if any operand or argument (excluding divisors and exponents) has decimal places.

Fixed-point evaluations

In general, if an arithmetic operation contains neither of the characteristics listed above for floating point, the compiler causes it to be evaluated in fixed-point arithmetic. In other words, arithmetic evaluations are handled as fixed point only if all the operands are fixed point, the result field is defined to be fixed point, and none of the exponents represent values with decimal places. Nested arithmetic expressions and function references must also represent fixed-point values.

Arithmetic comparisons (relation conditions)

When you compare numeric expressions using a relational operator, the numeric expressions (whether they are data items, arithmetic expressions, function references, or some combination of these) are comparands in the context of the entire evaluation. That is, the attributes of each can influence the evaluation of the other: both expressions are evaluated in fixed point, or both are evaluated in floating point. This is also true of abbreviated comparisons even though one comparand does not explicitly appear in the comparison. For example:

```
if (a + d) = (b + e) and c
```

This statement has two comparisons: `(a + d) = (b + e)`, and `(a + d) = c`. Although `(a + d)` does not explicitly appear in the second comparison, it is a comparand in that comparison. Therefore, the attributes of c can influence the evaluation of `(a + d)`.

The compiler handles comparisons (and the evaluation of any arithmetic expressions nested in comparisons) in floating-point arithmetic if either comparand is a floating-point value or resolves to a floating-point value.

The compiler handles comparisons (and the evaluation of any arithmetic expressions nested in comparisons) in fixed-point arithmetic if both comparands are fixed-point values or resolve to fixed-point values.

Implicit comparisons (no relational operator used) are not handled as a unit, however; the two comparands are treated separately as to their evaluation in floating-point or fixed-point arithmetic. In the following example, five arithmetic expressions are evaluated independently of one another's attributes, and then are compared to each other.

```
evaluate (a + d)
    when (b + e) thru c
    when (f / g) thru (h * i)
    . . .
end-evaluate
```

"Examples: fixed-point and floating-point evaluations" on page 55

Related references
"Arithmetic expressions
in nonarithmetic statements" on page 485

Examples: fixed-point and floating-point evaluations

The following example shows statements that are evaluated using fixed-point arithmetic and using floating-point arithmetic.

Assume that you define the data items for an employee table in the following manner:

```
01  employee-table.
    05  emp-count         pic  9(4).
    05  employee-record occurs 1 to 1000 times
            depending on emp-count.
        10 hours          pic +9(5)ve+99.
. . .
01  report-matrix-col     pic  9(3).
01  report-matrix-min     pic  9(3).
01  report-matrix-max     pic  9(3).
01  report-matrix-tot     pic  9(3).
01  average-hours         pic  9(3)v9.
01  whole-hours           pic  9(4).
```

These statements are evaluated using floating-point arithmetic:

```
compute report-matrix-col = (emp-count ** .5) + 1
compute report-matrix-col = function sqrt(emp-count) + 1
if report-matrix-tot < function sqrt(emp-count) + 1
```

These statements are evaluated using fixed-point arithmetic:

```
add report-matrix-min to report-matrix-max giving report-matrix-tot
compute report-matrix-max =
    function max(report-matrix-max report-matrix-tot)
if whole-hours not = function integer-part((average-hours) + 1)
```

Using currency signs

Many programs need to process financial information and present that information using the appropriate currency signs. With COBOL currency support (and the appropriate code page for your printer or display unit), you can use several currency signs in a program.

You can use one or more of the following signs:

- Symbols such as the dollar sign ($)
- Currency signs of more than one character (such as USD or EUR)
- Euro sign, established by the Economic and Monetary Union (EMU)

To specify the symbols for displaying financial information, use the CURRENCY SIGN clause (in the SPECIAL-NAMES paragraph in the CONFIGURATION SECTION) with the PICTURE characters that relate to those symbols. In the following example, the PICTURE character $ indicates that the currency sign $US is to be used:

```
    Currency Sign is "$US" with Picture Symbol "$".
    . . .
77  Invoice-Amount      Pic $$,$$9.99.
. . .
    Display "Invoice amount is " Invoice-Amount.
```

In this example, if Invoice-Amount contained 1500.00, the display output would be:

```
Invoice amount is  $US1,500.00
```

By using more than one CURRENCY SIGN clause in your program, you can allow for multiple currency signs to be displayed.

You can use a hexadecimal literal to indicate the currency sign value. Using a hexadecimal literal could be useful if the data-entry method for the source program does not allow the entry of the intended characters easily. The following example shows the hexadecimal value X'80' used as the currency sign:

```
    Currency Sign X'80' with Picture Symbol 'U'.
    . . .
01  Deposit-Amount       Pic UUUUU9.99.
```

If there is no corresponding character for the euro sign on your keyboard, you need to specify it as a hexadecimal value in the CURRENCY SIGN clause.

The hexadecimal value for the euro sign is X'80' with code page 1252 (Latin 1).

Related references
“CURRENCY” on page 258
CURRENCY SIGN clause (*COBOL for Linux on x86 Language Reference*)

Example: multiple currency signs

The following example shows how you can display values in both euro currency (as EUR) and Swiss francs (as CHF).

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EuroSamp.
Environment Division.
Configuration Section.
Special-Names.
    Currency Sign is "CHF "  with Picture Symbol "F"
    Currency Sign is "EUR "  with Picture Symbol "U".
```

```
Data Division.
WORKING-STORAGE SECTION.
01  Deposit-in-Euro        Pic S9999V99 Value 8000.00.
01  Deposit-in-CHF         Pic S99999V99.
01  Deposit-Report.
    02  Report-in-Franc    Pic -FFFFF9.99.
    02  Report-in-Euro     Pic -UUUUU9.99.
01  EUR-to-CHF-Conv-Rate   Pic 9V99999  Value 1.53893.
. . .
PROCEDURE DIVISION.
Report-Deposit-in-CHF-and-EUR.
    Move Deposit-in-Euro to Report-in-Euro
    Compute Deposit-in-CHF Rounded
          = Deposit-in-Euro * EUR-to-CHF-Conv-Rate
      On Size Error
        Perform Conversion-Error
      Not On Size Error
        Move Deposit-in-CHF to Report-in-Franc
        Display "Deposit in euro  = " Report-in-Euro
        Display "Deposit in franc = " Report-in-Franc
    End-Compute
    Goback.
Conversion-Error.
           Display "Conversion error from EUR to CHF"
           Display "Euro value: " Report-in-Euro.
```

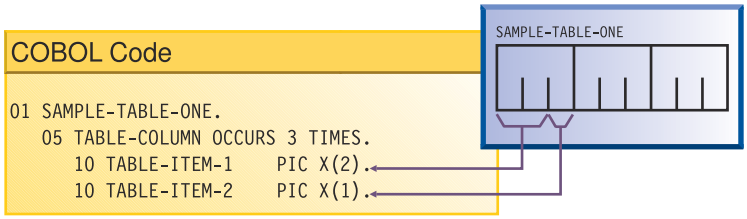
The above example produces the following display output:

```
Deposit in euro  =  EUR 8000.00
Deposit in franc = CHF 12311.44
```

The exchange rate used in this example is for illustrative purposes only.

Chapter 4. Handling tables

A *table* is a collection of data items that have the same description, such as account totals or monthly averages. A table consists of a table name and subordinate items called *table elements*. A table is the COBOL equivalent of an array.



In the example above, `SAMPLE-TABLE-ONE` is the group item that contains the table. `TABLE-COLUMN` names the table element of a one-dimensional table that occurs three times.

Rather than defining repetitious items as separate, consecutive entries in the `DATA DIVISION`, you use the `OCCURS` clause in the `DATA DIVISION` entry to define a table. This practice has these advantages:

- The code clearly shows the unity of the items (the table elements).
- You can use subscripts and indexes to refer to the table elements.
- You can easily repeat data items.

Tables are important for increasing the speed of a program, especially a program that looks up records.

Related concepts
"Complex OCCURS DEPENDING ON" on page 73

Related tasks
"Defining a table (OCCURS)" on page 59
"Nesting tables" on page 61
"Referring to an item in a table" on page 62
"Putting values into a table" on page 65
"Creating variable-length tables (DEPENDING ON)" on page 70
"Searching a table" on page 76
"Processing table items using intrinsic functions" on page 79

"Handling tables efficiently" on page 447

Defining a table (OCCURS)

To code a table, give the table a group name and define a subordinate item (the table element) to be repeated *n* times.

```
01  table-name.
    05 element-name OCCURS n TIMES.
    . . . (subordinate items of the table element)
```

In the example above, `table-name` is the name of an alphanumeric group item. The table element definition (which includes the `OCCURS` clause) is subordinate to the group item that contains the table. The `OCCURS` clause cannot be used in a level-01 description.

© Copyright IBM Corp. 2021

If a table is to contain only Unicode (UTF-16) data, and you want the group item that contains the table to behave like an elementary category national item in most operations, code the GROUP-USAGE NATIONAL clause for the group item:

```
01  table-nameN Group-Usage National.
    05  element-nameN OCCURS m TIMES.
        10  elementN1  Pic nn.
        10  elementN2  Pic S99 Sign Is Leading, Separate.
        . . .
```

Any elementary item that is subordinate to a national group must be explicitly or implicitly described as USAGE NATIONAL, and any subordinate numeric data item that is signed must be implicitly or explicitly described with the SIGN IS SEPARATE clause.

To create tables of two to seven dimensions, use nested OCCURS clauses.

To create a variable-length table, code the DEPENDING ON phrase of the OCCURS clause.

To specify that table elements will be arranged in ascending or descending order based on the values in one or more key fields of the table, code the ASCENDING or DESCENDING KEY phrases of the OCCURS clause, or both. Specify the names of the keys in decreasing order of significance. Keys can be of class alphabetic, alphanumeric, DBCS, national, or numeric. (If it has USAGE NATIONAL, a key can be of category national, or can be a national-edited, numeric-edited, national decimal, or national floating-point item.)

You must code the ASCENDING or DESCENDING KEY phrase of the OCCURS clause to do a binary search (SEARCH ALL) of a table. You can use a format 2 SORT statement to order the table according to its defined keys, thereby making the table searchable by the SEARCH ALL statement. Note that SEARCH ALL will return unpredictable results if the table has not been ordered according to the keys.

"Example: binary search" on page 78

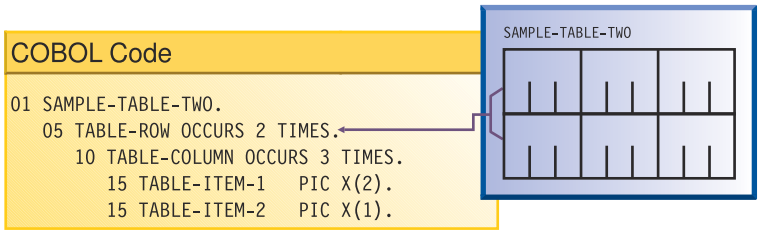
Related concepts
"National groups" on page 183

Related tasks
"Nesting tables" on page 61
"Referring to an item in a table" on page 62
"Putting values into a table" on page 65
"Creating variable-length tables (DEPENDING ON)" on page 70
"Using national groups" on page 187
"Doing a binary search (SEARCH ALL)" on page 78
"Defining numeric data" on page 35

Related references
OCCURS clause (*COBOL for Linux on x86 Language Reference*)
SIGN clause (*COBOL for Linux on x86 Language Reference*)
ASCENDING KEY and DESCENDING KEY phrases (*COBOL for Linux on x86 Language Reference*)
SORT statement (*COBOL for Linux on x86 Language Reference*)

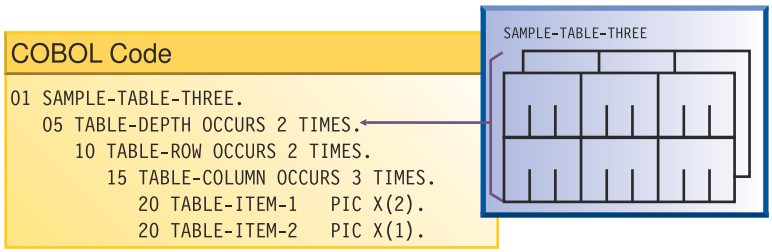
Nesting tables

To create a two-dimensional table, define a one-dimensional table in each occurrence of another one-dimensional table.



For example, in SAMPLE-TABLE-TWO above, TABLE-ROW is an element of a one-dimensional table that occurs two times. TABLE-COLUMN is an element of a two-dimensional table that occurs three times in each occurrence of TABLE-ROW.

To create a three-dimensional table, define a one-dimensional table in each occurrence of another one-dimensional table, which is itself contained in each occurrence of another one-dimensional table. For example:



In SAMPLE-TABLE-THREE, TABLE-DEPTH is an element of a one-dimensional table that occurs two times. TABLE-ROW is an element of a two-dimensional table that occurs two times within each occurrence of TABLE-DEPTH. TABLE-COLUMN is an element of a three-dimensional table that occurs three times within each occurrence of TABLE-ROW.

In a two-dimensional table, the two subscripts correspond to the row and column numbers. In a three-dimensional table, the three subscripts correspond to the depth, row, and column numbers.

“Example: subscripting” on page 62
“Example: indexing” on page 62

Related tasks

“Defining a table (OCCURS)” on page 59
“Referring to an item in a table” on page 62
“Putting values into a table” on page 65
“Creating variable-length tables (DEPENDING ON)” on page 70
“Searching a table” on page 76
“Processing table items using intrinsic functions” on page 79
“Handling tables efficiently” on page 447

Related references

OCCURS clause (*COBOL for Linux on x86 Language Reference*)

Example: subscripting

The following example shows valid references to SAMPLE-TABLE-THREE that use literal subscripts. The spaces are required in the second example.

```
TABLE-COLUMN (2, 2, 1)
TABLE-COLUMN (2 2 1)
```

In either table reference, the first value (2) refers to the second occurrence within TABLE-DEPTH, the second value (2) refers to the second occurrence within TABLE-ROW, and the third value (1) refers to the first occurrence within TABLE-COLUMN.

The following reference to SAMPLE-TABLE-TWO uses variable subscripts. The reference is valid if SUB1 and SUB2 are data-names that contain positive integer values within the range of the table.

```
TABLE-COLUMN (SUB1 SUB2)
```

Related tasks
"Subscripting" on page 63

Example: indexing

The following example shows how displacements to elements that are referenced with indexes are calculated.

Consider the following three-dimensional table, SAMPLE-TABLE-FOUR:

```
01  SAMPLE-TABLE-FOUR
    05 TABLE-DEPTH OCCURS 3 TIMES INDEXED BY INX-A.
       10 TABLE-ROW OCCURS 4 TIMES INDEXED BY INX-B.
          15 TABLE-COLUMN OCCURS 8 TIMES INDEXED BY INX-C  PIC X(8).
```

Suppose you code the following relative indexing reference to SAMPLE-TABLE-FOUR:

```
TABLE-COLUMN (INX-A + 1, INX-B + 2, INX-C - 1)
```

This reference causes the following computation of the displacement to the TABLE-COLUMN element:

```
  (contents of INX-A) + (256 * 1)
+ (contents of INX-B) + (64 * 2)
+ (contents of INX-C) - (8 * 1)
```

This calculation is based on the following element lengths:

- Each occurrence of TABLE-DEPTH is 256 bytes in length (4 * 8 * 8).
- Each occurrence of TABLE-ROW is 64 bytes in length (8 * 8).
- Each occurrence of TABLE-COLUMN is 8 bytes in length.

Related tasks
"Indexing" on page 64

Referring to an item in a table

A table element has a collective name, but the individual items within it do not have unique data-names.

To refer to an item, you have a choice of three techniques:

- Use the data-name of the table element, along with its occurrence number (called a *subscript*) in parentheses. This technique is called *subscripting*.
- Use the data-name of the table element, along with a value (called an *index*) that is added to the address of the table to locate an item (as a displacement from the beginning of the table). This technique is called *indexing*, or subscripting using index-names.
- Use both subscripts and indexes together.

Related tasks
“Subscripting” on page 63
“Indexing” on page 64

Subscripting

The lowest possible subscript value is 1, which references the first occurrence of a table element. In a one-dimensional table, the subscript corresponds to the row number.

You can use a literal or a data-name as a subscript. If a data item that has a literal subscript is of fixed length, the compiler resolves the location of the data item.

When you use a data-name as a variable subscript, you must describe the data-name as an elementary numeric integer. The most efficient format is COMPUTATIONAL (COMP) with a PICTURE size that is smaller than five digits. You cannot use a subscript with a data-name that is used as a subscript. The code generated for the application resolves the location of a variable subscript at run time.

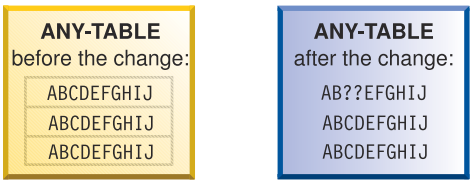
You can increment or decrement a literal or variable subscript by a specified integer amount. For example:

```
TABLE-COLUMN (SUB1 - 1, SUB2 + 3)
```

You can change part of a table element rather than the whole element. To do so, refer to the character position and length of the substring to be changed. For example:

```
01  ANY-TABLE.
    05 TABLE-ELEMENT     PIC X(10)
        OCCURS 3 TIMES   VALUE "ABCDEFGHIJ".
. . .
    MOVE "??" TO TABLE-ELEMENT (1) (3 : 2).
```

The MOVE statement in the example above moves the string '??' into table element number 1, beginning at character position 3, for a length of 2 characters.



“Example: subscripting” on page 62

Related tasks
“Indexing” on page 64
“Putting values into a table” on page 65
“Searching a table” on page 76
“Handling tables efficiently” on page 447

Indexing

You create an index by using the INDEXED BY phrase of the OCCURS clause to identify an index-name.

For example, INX-A in the following code is an index-name:

```
05 TABLE-ITEM PIC X(8)
     OCCURS 10 INDEXED BY INX-A.
```

The compiler calculates the value contained in the index as the occurrence number (subscript) minus 1, multiplied by the length of the table element. Therefore, for the fifth occurrence of TABLE-ITEM, the binary value contained in INX-A is (5 - 1) * 8, or 32.

You can use an index-name to reference another table only if both table descriptions have the same number of table elements, and the table elements are of the same length.

You can use the USAGE IS INDEX clause to create an index data item, and can use an index data item with any table. For example, INX-B in the following code is an index data item:

```
77  INX-B  USAGE IS INDEX.
. . .
    SET INX-A TO 10
    SET INX-B TO INX-A.
    PERFORM VARYING INX-A FROM 1 BY 1 UNTIL INX-A > INX-B
        DISPLAY TABLE-ITEM (INX-A)
        . . .
    END-PERFORM.
```

The index-name INX-A is used to traverse table TABLE-ITEM above. The index data item INX-B is used to hold the index of the last element of the table. The advantage of this type of coding is that calculation of offsets of table elements is minimized, and no conversion is necessary for the UNTIL condition.

You can use the SET statement to assign to an index data item the value that you stored in an index-name, as in the statement SET INX-B TO INX-A above. For example, when you load records into a variable-length table, you can store the index value of the last record into a data item defined as USAGE IS INDEX. Then you can test for the end of the table by comparing the current index value with the index value of the last record. This technique is useful when you look through or process a table.

You can increment or decrement an index-name by an elementary integer data item or a nonzero integer literal, for example:

```
SET INX-A DOWN BY 3
```

The integer represents a number of occurrences. It is converted to an index value before being added to or subtracted from the index.

Initialize the index-name by using a SET, PERFORM VARYING, or SEARCH ALL statement. You can then use the index-name in SEARCH or relational condition statements. To change the value, use a PERFORM, SEARCH, or SET statement.

Because you are comparing a physical displacement, you can directly use index data items only in SEARCH and SET statements or in comparisons with indexes or other index data items. You cannot use index data items as subscripts or indexes.

“Example: indexing” on page 62

Related tasks

“Subscripting” on page 63
“Putting values into a table” on page 65
“Searching a table” on page 76
“Processing table items
using intrinsic functions” on page 79
“Handling tables efficiently” on page 447

Related references
INDEXED BY phrase (*COBOL for Linux on x86 Language Reference*)
INDEX phrase (*COBOL for Linux on x86 Language Reference*)
SET statement (*COBOL for Linux on x86 Language Reference*)

Putting values into a table

You can put values into a table by loading the table dynamically, initializing the table with the INITIALIZE statement, or assigning values with the VALUE clause when you define the table.

Related tasks
"Loading a table dynamically" on page 65
"Loading a variable-length table" on page 72
"Initializing a table (INITIALIZE)" on page 65
"Assigning values when you define a table (VALUE)" on page 66
"Assigning values to a variable-length table" on page 72

Loading a table dynamically

If the initial values of a table are different with each execution of your program, you can define the table without initial values. You can instead read the changed values into the table dynamically before the program refers to the table.

To load a table, use the PERFORM statement and either subscripting or indexing.

When reading data to load your table, test to make sure that the data does not exceed the space allocated for the table. Use a named value (rather than a literal) for the maximum item count. Then, if you make the table bigger, you need to change only one value instead of all references to a literal.

"Example: PERFORM and subscripting" on page 68
"Example: PERFORM and indexing" on page 69

Related references
PERFORM statement (*COBOL for Linux on x86 Language Reference*)

Initializing a table (INITIALIZE)

You can load a table by coding one or more INITIALIZE statements.

For example, to move the value 3 into each of the elementary numeric data items in a table called TABLE-ONE, shown below, you can code the following statement:

```
INITIALIZE TABLE-ONE REPLACING NUMERIC DATA BY 3.
```

To move the character 'X' into each of the elementary alphanumeric data items in TABLE-ONE, you can code the following statement:

```
INITIALIZE TABLE-ONE REPLACING ALPHANUMERIC DATA BY "X".
```

When you use the INITIALIZE statement to initialize a table, the table is processed as a group item (that is, with group semantics); elementary data items within the group are recognized and processed. For example, suppose that TABLE-ONE is an alphanumeric group that is defined like this:

```
01  TABLE-ONE.
  02  Trans-out  Occurs 20.
      05  Trans-code       Pic X    Value "R".
      05  Part-number      Pic XX   Value "13".
      05  Trans-quan       Pic 99   Value 10.
```

```
      05  Price-fields.
          10  Unit-price   Pic 99V  Value 50.
          10  Discount     Pic 99V  Value 25.
          10  Sales-Price  Pic 999  Value 375.
      . . .
      Initialize TABLE-ONE Replacing Numeric Data By 3
                                     Alphanumeric Data By "X"
```

The table below shows the content that each of the twenty 12-byte elements `Trans-out(n)` has before execution and after execution of the `INITIALIZE` statement shown above:

Trans-out(*n*) before	**Trans-out(*n*) after**
R13105025375	XX*b*030303003[1]
1. The symbol *b* represents a blank space.	

You can similarly use an `INITIALIZE` statement to load a table that is defined as a national group. For example, if `TABLE-ONE` shown above specified the `GROUP-USAGE NATIONAL` clause, and `Trans-code` and `Part-number` had `N` instead of `X` in their `PICTURE` clauses, the following statement would have the same effect as the `INITIALIZE` statement above, except that the data in `TABLE-ONE` would instead be encoded in UTF-16:

```
Initialize TABLE-ONE Replacing Numeric  Data By 3
                               National Data By N"X"
```

The `REPLACING NUMERIC` phrase initializes floating-point data items also.

You can use the `REPLACING` phrase of the `INITIALIZE` statement similarly to initialize all of the elementary `ALPHABETIC`, `DBCS`, `ALPHANUMERIC-EDITED`, `NATIONAL-EDITED`, and `NUMERIC-EDITED` data items in a table.

The `INITIALIZE` statement cannot assign values to a variable-length table (that is, a table that was defined using the `OCCURS DEPENDING ON` clause).

"Examples: initializing data items" on page 24

Related tasks
"Initializing a structure (INITIALIZE)" on page 27
"Assigning values when you define a table (VALUE)" on page 66
"Assigning values to a variable-length table" on page 72
"Looping through a table" on page 90
"Using data items and group items" on page 20
"Using national groups" on page 187

Related references
INITIALIZE statement (*COBOL for Linux on x86 Language Reference*)

Assigning values when you define a table (VALUE)

If a table is to contain stable values (such as days and months), you can set the specific values when you define the table.

Set static values in tables in one of these ways:

- Initialize each table item individually.
- Initialize an entire table at the group level.
- Initialize all occurrences of a given table element to the same value.

Related tasks
"Initializing each table item individually" on page 67
"Initializing a table at the group level" on page 68
"Initializing all occurrences of a given table element" on page 68
"Initializing a structure (INITIALIZE)" on page 27

Initializing each table item individually

If a table is small, you can set the value of each item individually by using a VALUE clause.

Use the following technique, which is shown in the example code below:

1. Define a record (such as `Error-Flag-Table` below) that contains the items that are to be in the table.
2. Set the initial value of each item in a VALUE clause.
3. Code a REDEFINES entry to make the record into a table.

```
*************************************************************
***           E R R O R   F L A G   T A B L E             ***
*************************************************************
 01  Error-Flag-Table                  Value Spaces.
   88 No-Errors                        Value Spaces.
     05 Type-Error                     Pic X.
     05 Shift-Error                    Pic X.
     05 Home-Code-Error                Pic X.
     05 Work-Code-Error                Pic X.
     05 Name-Error                     Pic X.
     05 Initials-Error                 Pic X.
     05 Duplicate-Error                Pic X.
     05 Not-Found-Error                Pic X.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 8 Times
          Indexed By Flag-Index         Pic X.
```

In the example above, the VALUE clause at the 01 level initializes each of the table items to the same value. Each table item could instead be described with its own VALUE clause to initialize that item to a distinct value.

To initialize larger tables, use MOVE, PERFORM, or INITIALIZE statements.

Related tasks
"Initializing a structure (INITIALIZE)" on page 27
"Assigning values to a variable-length table" on page 72

Related references
REDEFINES clause (*COBOL for Linux on x86 Language Reference*)
OCCURS clause (*COBOL for Linux on x86 Language Reference*)

Initializing a table at the group level

Code an alphanumeric or national group data item and assign to it, through the VALUE clause, the contents of the whole table. Then, in a subordinate data item, use an OCCURS clause to define the individual table items.

In the following example, the alphanumeric group data item TABLE-ONE uses a VALUE clause that initializes each of the four elements of TABLE-TWO:

```
01  TABLE-ONE                    VALUE "1234".
    05 TABLE-TWO OCCURS 4 TIMES   PIC X.
```

In the following example, the national group data item Table-OneN uses a VALUE clause that initializes each of the three elements of the subordinate data item Table-TwoN (each of which is implicitly USAGE NATIONAL). Note that you can initialize a national group data item with a VALUE clause that uses an alphanumeric literal, as shown below, or a national literal.

```
01  Table-OneN  Group-Usage National  Value "AB12CD34EF56".
    05  Table-TwoN   Occurs 3 Times   Indexed By MyI.
        10  ElementOneN  Pic nn.
        10  ElementTwoN  Pic 99.
```

After Table-OneN is initialized, ElementOneN(1) contains NX"41004200" (the UTF-16 representation of 'AB'), the national decimal item ElementTwoN(1) contains NX"31003200" (the UTF-16 representation of '12'), and so forth.

Related references
OCCURS clause (*COBOL for Linux on x86 Language Reference*)
GROUP-USAGE clause (*COBOL for Linux on x86 Language Reference*)

Initializing all occurrences of a given table element

You can use the VALUE clause in the data description of a table element to initialize all instances of that element to the specified value.

```
01  T2.
    05 T-OBJ                   PIC 9   VALUE 3.
    05 T OCCURS 5 TIMES
           DEPENDING ON T-OBJ.
       10 X                    PIC XX  VALUE "AA".
       10 Y                    PIC 99  VALUE 19.
       10 Z                    PIC XX  VALUE "BB".
```

For example, the code above causes all the X elements (1 through 5) to be initialized to AA, all the Y elements (1 through 5) to be initialized to 19, and all the Z elements (1 through 5) to be initialized to BB. T-OBJ is then set to 3.

Related tasks
"Assigning values to a variable-length table" on page 72

Related references
OCCURS clause (*COBOL for Linux on x86 Language Reference*)

Example: PERFORM and subscripting

This example traverses an error-flag table using subscripting until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```
*********************************************************
***          E R R O R   F L A G   T A B L E          ***
```

```
*************************************************************
 01  Error-Flag-Table                    Value Spaces.
   88 No-Errors                           Value Spaces.
     05 Type-Error                        Pic X.
     05 Shift-Error                       Pic X.
     05 Home-Code-Error                   Pic X.
     05 Work-Code-Error                   Pic X.
     05 Name-Error                        Pic X.
     05 Initials-Error                    Pic X.
     05 Duplicate-Error                   Pic X.
     05 Not-Found-Error                   Pic X.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 8 Times
          Indexed By Flag-Index           Pic X.
 77  Error-on                            Pic X  Value "E".
*************************************************************
***         E R R O R   M E S S A G E   T A B L E       ***
*************************************************************
 01  Error-Message-Table.
     05  Filler                           Pic X(25) Value
          "Transaction Type Invalid".
     05  Filler                           Pic X(25) Value
          "Shift Code Invalid".
     05  Filler                           Pic X(25) Value
          "Home Location Code Inval.".
     05  Filler                           Pic X(25) Value
          "Work Location Code Inval.".
     05  Filler                           Pic X(25) Value
          "Last Name - Blanks".
     05  Filler                           Pic X(25) Value
          "Initials - Blanks".
     05  Filler                           Pic X(25) Value
          "Duplicate Record Found".
     05  Filler                           Pic X(25) Value
          "Commuter Record Not Found".
 01  Filler Redefines Error-Message-Table.
     05  Error-Message Occurs 8 Times
           Indexed By Message-Index        Pic X(25).
 . . .
 PROCEDURE DIVISION.
     . . .
     Perform
          Varying Sub From 1 By 1
          Until No-Errors
        If Error-Flag (Sub) = Error-On
          Move Space To Error-Flag (Sub)
          Move Error-Message (Sub) To Print-Message
          Perform 260-Print-Report
        End-If
     End-Perform
     . . .
```

Example: PERFORM and indexing

This example traverses an error-flag table using indexing until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```
*************************************************************
***          E R R O R   F L A G   T A B L E          ***
*************************************************************
 01  Error-Flag-Table                    Value Spaces.
   88 No-Errors                           Value Spaces.
     05 Type-Error                        Pic X.
     05 Shift-Error                       Pic X.
     05 Home-Code-Error                   Pic X.
     05 Work-Code-Error                   Pic X.
     05 Name-Error                        Pic X.
     05 Initials-Error                    Pic X.
     05 Duplicate-Error                   Pic X.
     05 Not-Found-Error                   Pic X.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 8 Times
          Indexed By Flag-Index           Pic X.
 77  Error-on                            Pic X  Value "E".
*************************************************************
***         E R R O R   M E S S A G E   T A B L E       ***
*************************************************************
 01  Error-Message-Table.
```

```
     05  Filler                          Pic X(25) Value
          "Transaction Type Invalid".
     05  Filler                          Pic X(25) Value
          "Shift Code Invalid".
     05  Filler                          Pic X(25) Value
          "Home Location Code Inval.".
     05  Filler                          Pic X(25) Value
          "Work Location Code Inval.".
     05  Filler                          Pic X(25) Value
          "Last Name - Blanks".
     05  Filler                          Pic X(25) Value
          "Initials - Blanks".
     05  Filler                          Pic X(25) Value
          "Duplicate Record Found".
     05  Filler                          Pic X(25) Value
          "Commuter Record Not Found".
 01  Filler Redefines Error-Message-Table.
     05  Error-Message Occurs 8 Times
            Indexed By Message-Index      Pic X(25).
 . . .
 PROCEDURE DIVISION.
     . . .
     Set Flag-Index To 1
     Perform Until No-Errors
       Search Error-Flag
         When Error-Flag (Flag-Index) = Error-On
           Move Space To Error-Flag (Flag-Index)
           Set Message-Index To Flag-Index
           Move Error-Message (Message-Index) To
             Print-Message
           Perform 260-Print-Report
       End-Search
     End-Perform
     . . .
```

Creating variable-length tables (DEPENDING ON)

If you do not know before run time how many times a table element occurs, define a variable-length table. To do so, use the OCCURS DEPENDING ON (ODO) clause.

```
X OCCURS 1 TO 10 TIMES DEPENDING ON Y
```

In the example above, X is called the *ODO subject*, and Y is called the *ODO object*.

Two factors affect the successful manipulation of variable-length records:

- Correct calculation of record lengths

  The length of the variable portions of a group item is the product of the object of the DEPENDING ON phrase and the length of the subject of the OCCURS clause.
- Conformance of the data in the object of the OCCURS DEPENDING ON clause to its PICTURE clause

  If the content of the ODO object does not match its PICTURE clause, the program could terminate abnormally. You must ensure that the ODO object correctly specifies the current number of occurrences of table elements.

The following example shows a group item (REC-1) that contains both the subject and object of the OCCURS DEPENDING ON clause. The way the length of the group item is determined depends on whether it is sending or receiving data.

```
WORKING-STORAGE SECTION.
01  MAIN-AREA.
    03 REC-1.
        05 FIELD-1                      PIC 9.
        05 FIELD-2 OCCURS 1 TO 5 TIMES
           DEPENDING ON FIELD-1         PIC X(05).
01  REC-2.
    03 REC-2-DATA                      PIC X(50).
```

If you want to move REC-1 (the sending item in this case) to REC-2, the length of REC-1 is determined immediately before the move, using the current value in FIELD-1. If the content of FIELD-1 conforms to its PICTURE clause (that is, if FIELD-1 contains a zoned decimal item), the move can proceed based on the actual length of REC-1. Otherwise, the result is unpredictable. You must ensure that the ODO object has the correct value before you initiate the move.

When you do a move to REC-1 (the receiving item in this case), the length of REC-1 is determined using the maximum number of occurrences. In this example, five occurrences of FIELD-2, plus FIELD-1, yields a length of 26 bytes. In this case, you do not need to set the ODO object (FIELD-1) before referencing REC-1 as a receiving item. However, the sending field's ODO object (not shown) must be set to a valid numeric value between 1 and 5 for the ODO object of the receiving field to be validly set by the move.

However, if you do a move to REC-1 (again the receiving item) where REC-1 is followed by a variably located group (a type of *complex ODO*), the actual length of REC-1 is calculated immediately before the move, using the current value of the ODO object (FIELD-1). In the following example, REC-1 and REC-2 are in the same record, but REC-2 is not subordinate to REC-1 and is therefore variably located:

```
01  MAIN-AREA
    03 REC-1.
       05 FIELD-1                      PIC 9.
       05 FIELD-3                      PIC 9.
       05 FIELD-2 OCCURS 1 TO 5 TIMES
            DEPENDING ON FIELD-1       PIC X(05).
    03 REC-2.
       05 FIELD-4 OCCURS 1 TO 5 TIMES
            DEPENDING ON FIELD-3       PIC X(05).
```

The compiler issues a message that lets you know that the actual length was used. This case requires that you set the value of the ODO object before using the group item as a receiving field.

The following example shows how to define a variable-length table when the ODO object (LOCATION-TABLE-LENGTH below) is outside the group:

```
 DATA DIVISION.
 FILE SECTION.
 FD  LOCATION-FILE.
 01  LOCATION-RECORD.
     05  LOC-CODE                 PIC XX.
     05  LOC-DESCRIPTION          PIC X(20).
     05  FILLER                   PIC X(58).
 WORKING-STORAGE SECTION.
 01  FLAGS.
     05 LOCATION-EOF-FLAG         PIC X(5) VALUE SPACE.
        88 LOCATION-EOF              VALUE "FALSE".
 01  MISC-VALUES.
     05 LOCATION-TABLE-LENGTH     PIC 9(3) VALUE ZERO.
     05 LOCATION-TABLE-MAX        PIC 9(3) VALUE 100.
*****************************************************************
***               L O C A T I O N   T A B L E                 ***
***               FILE CONTAINS LOCATION CODES.               ***
*****************************************************************
 01  LOCATION-TABLE.
     05 LOCATION-CODE OCCURS 1 TO 100 TIMES
          DEPENDING ON LOCATION-TABLE-LENGTH   PIC X(80).
```

Related concepts
“Complex OCCURS DEPENDING ON” on page 73

Related tasks
“Assigning values to a variable-length table” on page 72
“Loading a variable-length table” on page 72
“Preventing overlay when adding elements to a variable table” on page 75

"Finding the length of data items" on page 109

Related references
OCCURS DEPENDING ON clause
(*COBOL for Linux on x86 Language Reference*)
Variable-length tables (*COBOL for Linux on x86 Language Reference*)

Loading a variable-length table

You can use a *do-until* structure (a TEST AFTER loop) to control the loading of a variable-length table. For example, after the following code runs, LOCATION-TABLE-LENGTH contains the subscript of the last item in the table.

```
 DATA DIVISION.
 FILE SECTION.
 FD  LOCATION-FILE.
 01  LOCATION-RECORD.
     05  LOC-CODE              PIC XX.
     05  LOC-DESCRIPTION       PIC X(20).
     05  FILLER                PIC X(58).
 . . .
 WORKING-STORAGE SECTION.
 01  FLAGS.
     05 LOCATION-EOF-FLAG      PIC X(5) VALUE SPACE.
        88 LOCATION-EOF                 VALUE "YES".
 01  MISC-VALUES.
     05 LOCATION-TABLE-LENGTH  PIC 9(3) VALUE ZERO.
     05 LOCATION-TABLE-MAX     PIC 9(3) VALUE 100.
*****************************************************************
***               L O C A T I O N   T A B L E                 ***
***               FILE CONTAINS LOCATION CODES.               ***
*****************************************************************
 01  LOCATION-TABLE.
     05 LOCATION-CODE OCCURS 1 TO 100 TIMES
          DEPENDING ON LOCATION-TABLE-LENGTH   PIC X(80).
 . . .
 PROCEDURE DIVISION.
     . . .
     Perform Test After
         Varying Location-Table-Length From 1 By 1
           Until Location-EOF
           Or Location-Table-Length = Location-Table-Max
       Move Location-Record To
           Location-Code (Location-Table-Length)
       Read Location-File
           At End Set Location-EOF To True
       End-Read
     End-Perform
```

Assigning values to a variable-length table

You can code a VALUE clause for an alphanumeric or national group item that has a subordinate data item that contains the OCCURS clause with the DEPENDING ON phrase. Each subordinate structure that contains the DEPENDING ON phrase is initialized using the maximum number of occurrences.

If you define the entire table by using the DEPENDING ON phrase, all the elements are initialized using the maximum defined value of the ODO (OCCURS DEPENDING ON) object.

If the ODO object is initialized by a VALUE clause, it is logically initialized after the ODO subject has been initialized.

```
01  TABLE-THREE           VALUE "3ABCDE".
    05 X                  PIC 9.
    05 Y OCCURS 5 TIMES
          DEPENDING ON X  PIC X.
```

For example, in the code above, the ODO subject Y(1) is initialized to 'A', Y(2) to 'B', . . ., Y(5) to 'E', and finally the ODO object X is initialized to 3. Any subsequent reference to TABLE-THREE (such as in a DISPLAY statement) refers to X and the first three elements, Y(1) through Y(3), of the table.

Related tasks
"Assigning values when you define a table (VALUE)" on page 66

Related references
OCCURS DEPENDING ON clause
(*COBOL for Linux on x86 Language Reference*)

Complex OCCURS DEPENDING ON

Several types of complex OCCURS DEPENDING ON (*complex ODO*) are possible. Complex ODO is supported as an extension to the 85 COBOL Standard.

The basic forms of complex ODO permitted by the compiler are as follows:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate elementary or group data item.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate data item described by an OCCURS clause.
- Table that has variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON phrase.
- Index name for a table that has variable-length elements.
- Element of a table that has variable-length elements.

"Example: complex ODO" on page 73

Related tasks
"Preventing index errors when changing ODO object value" on page 75
"Preventing overlay when adding elements to a variable table" on page 75

Related references
"Effects of change in ODO object value" on page 74
OCCURS DEPENDING ON clause
(*COBOL for Linux on x86 Language Reference*)

Example: complex ODO

The following example illustrates the possible types of occurrence of complex ODO.

```
01  FIELD-A.
    02 COUNTER-1                          PIC S99.
    02 COUNTER-2                          PIC S99.
    02 TABLE-1.
       03 RECORD-1 OCCURS 1 TO 5 TIMES
                   DEPENDING ON COUNTER-1    PIC X(3).
    02 EMPLOYEE-NUMBER                    PIC X(5). (1)
    02 TABLE-2 OCCURS 5 TIMES                       (2)(3)
              INDEXED BY INDX.                     (4)
       03 TABLE-ITEM                      PIC 99.   (5)
       03 RECORD-2 OCCURS 1 TO 3 TIMES
                   DEPENDING ON COUNTER-2.
          04 DATA-NUM                     PIC S99.
```

Definition: In the example, COUNTER-1 is an *ODO object*, that is, it is the object of the DEPENDING ON clause of RECORD-1. RECORD-1 is said to be an *ODO subject*. Similarly, COUNTER-2 is the ODO object of the corresponding ODO subject, RECORD-2.

The types of complex ODO occurrences shown in the example above are as follows:

(1)
A variably located item: EMPLOYEE-NUMBER is a data item that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(2)
A variably located table: TABLE-2 is a table that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(3)
A table with variable-length elements: TABLE-2 is a table that contains a subordinate data item, RECORD-2, whose number of occurrences varies depending on the content of its ODO object.

(4)
An index-name, INDX, for a table that has variable-length elements.

(5)
An element, TABLE-ITEM, of a table that has variable-length elements.

How length is calculated

The length of the variable portion of each record is the product of its ODO object and the length of its ODO subject. For example, whenever a reference is made to one of the complex ODO items shown above, the actual length, if used, is computed as follows:

- The length of TABLE-1 is calculated by multiplying the contents of COUNTER-1 (the number of occurrences of RECORD-1) by 3 (the length of RECORD-1).
- The length of TABLE-2 is calculated by multiplying the contents of COUNTER-2 (the number of occurrences of RECORD-2) by 2 (the length of RECORD-2), and adding the length of TABLE-ITEM.
- The length of FIELD-A is calculated by adding the lengths of COUNTER-1, COUNTER-2, TABLE-1, EMPLOYEE-NUMBER, and TABLE-2 times 5.

Setting values of ODO objects

You must set *every* ODO object in a group item before you reference any complex ODO item in the group. For example, before you refer to EMPLOYEE-NUMBER in the code above, you must set COUNTER-1 and COUNTER-2 even though EMPLOYEE-NUMBER does not directly depend on either ODO object for its value.

Restriction: An ODO object cannot be variably located.

Effects of change in ODO object value

If a data item that is described by an OCCURS clause with the DEPENDING ON phrase is followed in the same group by one or more nonsubordinate data items (a form of complex ODO), any change in value of the ODO object affects subsequent references to complex ODO items in the record.

For example:

- The size of any group that contains the relevant ODO clause reflects the new value of the ODO object.
- A MOVE to a group that contains the ODO subject is made based on the new value of the ODO object.
- The location of any nonsubordinate items that follow the item described with the ODO clause is affected by the new value of the ODO object. (To preserve the contents of the nonsubordinate items, move them to a work area before the value of the ODO object changes, then move them back.)

The value of an ODO object can change when you move data to the ODO object or to the group in which it is contained. The value can also change if the ODO object is contained in a record that is the target of a READ statement.

Related tasks
“Preventing index errors
when changing ODO object value” on page 75
“Preventing overlay when adding elements to a variable table” on page 75

Preventing index errors when changing ODO object value

Be careful if you reference a complex-ODO index-name, that is, an index-name for a table that has variable-length elements, after having changed the value of the ODO object for a subordinate data item in the table.

When you change the value of an ODO object, the byte offset in an associated complex-ODO index is no longer valid because the table length has changed. Unless you take precautions, you will have unexpected results if you then code a reference to the index-name such as:

- A reference to an element of the table
- A SET statement of the form SET *integer-data-item* TO *index-name* (format 1)
- A SET statement of the form SET *index-name* UP | DOWN BY *integer* (format 2)

To avoid this type of error, do these steps:

1. Save the index in an integer data item. (Doing so causes an implicit conversion: the integer item receives the table element occurrence number that corresponds to the offset in the index.)
2. Change the value of the ODO object.
3. Immediately restore the index from the integer data item. (Doing so causes an implicit conversion: the index-name receives the offset that corresponds to the table element occurrence number in the integer item. The offset is computed according to the table length then in effect.)

The following code shows how to save and restore the index-name (shown in "Example: complex ODO" on page 73) when the ODO object COUNTER-2 changes.

```
 77  INTEGER-DATA-ITEM-1      PIC 99.
 . . .
     SET INDX TO 5.
*         INDX is valid at this point.
     SET INTEGER-DATA-ITEM-1 TO INDX.
*         INTEGER-DATA-ITEM-1 now has the
*         occurrence number that corresponds to INDX.
     MOVE NEW-VALUE TO COUNTER-2.
*         INDX is not valid at this point.
     SET INDX TO INTEGER-DATA-ITEM-1.
*         INDX is now valid, containing the offset
*         that corresponds to INTEGER-DATA-ITEM-1, and
*         can be used with the expected results.
```

Related references
SET statement (*COBOL for Linux on x86 Language Reference*)

Preventing overlay when adding elements to a variable table

Be careful if you increase the number of elements in a variable-occurrence table that is followed by one or more nonsubordinate data items in the same group. When you increment the value of the ODO object and add an element to a table, you can inadvertently overlay the variably located data items that follow the table.

To avoid this type of error, do these steps:

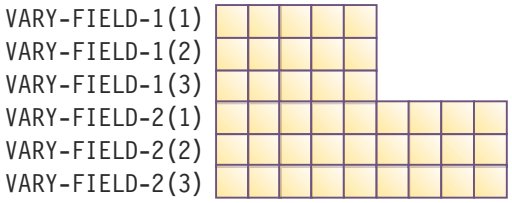
1. Save the variably located data items that follow the table in another data area.
2. Increment the value of the ODO object.
3. Move data into the new table element (if needed).
4. Restore the variably located data items from the data area where you saved them.

In the following example, suppose you want to add an element to the table VARY-FIELD-1, whose number of elements depends on the ODO object CONTROL-1. VARY-FIELD-1 is followed by the nonsubordinate variably located data item GROUP-ITEM-1, whose elements could potentially be overlaid.

```
WORKING-STORAGE SECTION.
```

```
01  VARIABLE-REC.
    05  FIELD-1                              PIC X(10).
    05  CONTROL-1                            PIC S99.
    05  CONTROL-2                            PIC S99.
    05  VARY-FIELD-1 OCCURS 1 TO 10 TIMES
          DEPENDING ON CONTROL-1              PIC X(5).
    05  GROUP-ITEM-1.
        10  VARY-FIELD-2
              OCCURS 1 TO 10 TIMES
              DEPENDING ON CONTROL-2          PIC X(9).
01  STORE-VARY-FIELD-2.
    05  GROUP-ITEM-2.
        10  VARY-FLD-2
              OCCURS 1 TO 10 TIMES
              DEPENDING ON CONTROL-2          PIC X(9).
```

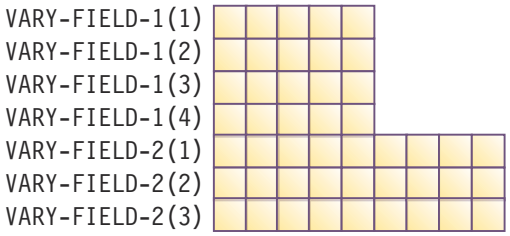
Each element of VARY-FIELD-1 has 5 bytes, and each element of VARY-FIELD-2 has 9 bytes. If CONTROL-1 and CONTROL-2 both contain the value 3, you can picture storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



To add a fourth element to VARY-FIELD-1, code as follows to prevent overlaying the first 5 bytes of VARY-FIELD-2. (GROUP-ITEM-2 serves as temporary storage for the variably located GROUP-ITEM-1.)

```
MOVE GROUP-ITEM-1 TO GROUP-ITEM-2.
ADD 1 TO CONTROL-1.
MOVE five-byte-field TO
  VARY-FIELD-1 (CONTROL-1).
MOVE GROUP-ITEM-2 TO GROUP-ITEM-1.
```

You can picture the updated storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



Note that the fourth element of VARY-FIELD-1 did not overlay the first element of VARY-FIELD-2.

Searching a table

COBOL provides two search techniques for tables: *serial* and *binary*.

To do serial searches, use SEARCH and indexing. For variable-length tables, you can use PERFORM with subscripting or indexing.

To do binary searches, use SEARCH ALL and indexing.

A binary search can be considerably more efficient than a serial search. For a serial search, the number of comparisons is of the order of *n*, the number of entries in the table. For a binary search, the number of comparisons is of the order of only the logarithm (base 2) of *n*. A binary search, however, requires that the table items already be sorted.

Related tasks
"Doing a serial search (SEARCH)" on page 77

"Doing a binary search (SEARCH ALL)" on page 78

Doing a serial search (SEARCH)

Use the SEARCH statement to do a serial (sequential) search beginning at the current index setting. To modify the index setting, use the SET statement.

The conditions in the WHEN phrase are evaluated in the order in which they appear:

- If none of the conditions is satisfied, the index is increased to correspond to the next table element, and the WHEN conditions are evaluated again.
- If one of the WHEN conditions is satisfied, the search ends. The index remains pointing to the table element that satisfied the condition.
- If the entire table has been searched and no conditions were met, the AT END imperative statement is executed if there is one. If you did not code AT END, control passes to the next statement in the program.

You can reference only one level of a table (a table element) with each SEARCH statement. To search multiple levels of a table, use nested SEARCH statements. Delimit each nested SEARCH statement with END-SEARCH.

Performance: If the found condition comes after some intermediate point in the table, you can speed up the search by using the SET statement to set the index to begin the search after that point. Arranging the table so that the data used most often is at the beginning of the table also enables more efficient serial searching. If the table is large and is presorted, a binary search is more efficient.

"Example: serial search" on page 77

Related references
SEARCH statement (*COBOL for Linux on x86 Language Reference*)

Example: serial search

The following example shows how you might find a particular string in the innermost table of a three-dimensional table.

Each dimension of the table has its own index (set to 1, 4, and 1, respectively). The innermost table (TABLE-ENTRY3) has an ascending key.

```
01  TABLE-ONE.
    05 TABLE-ENTRY1 OCCURS 10 TIMES
          INDEXED BY TE1-INDEX.
       10 TABLE-ENTRY2 OCCURS 10 TIMES
             INDEXED BY TE2-INDEX.
          15 TABLE-ENTRY3 OCCURS 5 TIMES
                ASCENDING KEY IS KEY1
                INDEXED BY TE3-INDEX.
             20 KEY1                PIC X(5).
             20 KEY2                PIC X(10).
. . .
PROCEDURE DIVISION.
    . . .
    SET TE1-INDEX TO 1
    SET TE2-INDEX TO 4
    SET TE3-INDEX TO 1
    MOVE "A1234" TO KEY1 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
    MOVE "AAAAAAAA00" TO KEY2 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
    . . .
    SEARCH TABLE-ENTRY3
      AT END
        MOVE 4 TO RETURN-CODE
      WHEN TABLE-ENTRY3(TE1-INDEX, TE2-INDEX, TE3-INDEX)
          = "A1234AAAAAAAA00"
        MOVE 0 TO RETURN-CODE
    END-SEARCH
```

Values after execution:

```
TE1-INDEX = 1
TE2-INDEX = 4
TE3-INDEX points to the TABLE-ENTRY3 item
          that equals "A1234AAAAAAAA00"
RETURN-CODE = 0
```

Doing a binary search (SEARCH ALL)

If you use SEARCH ALL to do a binary search, you do not need to set the index before you begin. The index is always the one that is associated with the first index-name in the OCCURS clause. The index varies during execution to maximize the search efficiency.

To use the SEARCH ALL statement to search a table, the table must specify the ASCENDING or DESCENDING KEY phrases of the OCCURS clause, or both, and must already be ordered on the key or keys that are specified in the ASCENDING and DESCENDING KEY phrases. You can use a format 2 SORT statement to order the table according to its defined keys, thereby making the table searchable by the SEARCH ALL statement. Note that SEARCH ALL will return unpredictable results if the table has not been ordered according to the keys.

In the WHEN phrase of the SEARCH ALL statement, you can test any key that is named in the ASCENDING or DESCENDING KEY phrases for the table, but you must test all preceding keys, if any. The test must be an equal-to condition, and the WHEN phrase must specify either a key (subscripted by the first index-name associated with the table) or a condition-name that is associated with the key. The WHEN condition can be a compound condition that is formed from simple conditions that use AND as the only logical connective.

Each key and its object of comparison must be compatible according to the rules for comparison of data items. Note though that if a key is compared to a national literal or identifier, the key must be a national data item.

"Example: binary search" on page 78

Related tasks
"Defining a table (OCCURS)" on page 59

Related references
SEARCH statement (*COBOL for Linux on x86 Language Reference*)
General relation conditions (*COBOL for Linux on x86 Language Reference*)

Example: binary search

The following example shows how you can code a binary search of a table.

Suppose you define a table that contains 90 elements of 40 bytes each, and three keys. The primary and secondary keys (KEY-1 and KEY-2) are in ascending order, but the least significant key (KEY-3) is in descending order:

```
01  TABLE-A.
    05 TABLE-ENTRY OCCURS 90 TIMES
           ASCENDING KEY-1, KEY-2
           DESCENDING KEY-3
           INDEXED BY INDX-1.
       10 PART-1       PIC 99.
       10 KEY-1        PIC 9(5).
       10 PART-2       PIC 9(6).
       10 KEY-2        PIC 9(4).
       10 PART-3       PIC 9(18).
       10 KEY-3        PIC 9(5).
```

You can search this table by using the following statements:

```
SEARCH ALL TABLE-ENTRY
```

```
  AT END
    PERFORM NOENTRY
  WHEN KEY-1 (INDX-1) = VALUE-1 AND
       KEY-2 (INDX-1) = VALUE-2 AND
       KEY-3 (INDX-1) = VALUE-3
    MOVE PART-1 (INDX-1) TO OUTPUT-AREA
END-SEARCH
```

If an entry is found in which each of the three keys is equal to the value to which it is compared (VALUE-1, VALUE-2, and VALUE-3, respectively), PART-1 of that entry is moved to OUTPUT-AREA. If no matching key is found in the entries in TABLE-A, the NOENTRY routine is performed.

Sorting a table

You can sort a table by using the format 2 SORT statement. It is part of the 2002 COBOL Standard.

The format 2 SORT statement sorts table elements according to the specified table keys, and it is especially useful for tables used with SEARCH ALL. You can specify the keys for sorting as part of the table definition, which can also be used in the SEARCH ALL statement. Alternatively, you can also specify the keys for sorting as part of the SORT statement, either if you want to sort the table using different keys than those specified in the table definition, or if the table has no keys specified.

With the format 2 SORT statement, you don't need to use the input and output procedures as you do with the format 1 SORT statement.

See the following example in which the table is sorted based on specified keys:

```
WORKING-STORAGE SECTION.
01 GROUP-ITEM.
    05 TABL OCCURS 10 TIMES
       10 ELEM-ITEM1 PIC X.
       10 ELEM-ITEM2 PIC X.
       10 ELEM-ITEM3 PIC X.
...
PROCEDURE DIVISION.
    ...
    SORT TABL DESCENDING ELEM-ITEM2 ELEM-ITEM3.
    IF TABL (1)...
```

Related references
SORT statement (*COBOL for Linux on x86 Language Reference*)
“Using the format 2 SORT statement to sort a table” on page 464

Processing table items using intrinsic functions

You can use intrinsic functions to process alphabetic, alphanumeric, national, or numeric table items. (You can process DBCS data items only with the NATIONAL-OF intrinsic function.) The data descriptions of the table items must be compatible with the requirements for the function arguments.

Use a subscript or index to reference an individual data item as a function argument. For example, assuming that Table-One is a 3 x 3 array of numeric items, you can find the square root of the middle element by using this statement:

```
Compute X = Function Sqrt(Table-One(2,2))
```

You might often need to iteratively process the data in tables. For intrinsic functions that accept multiple arguments, you can use the subscript ALL to reference all the items in the table or in a single dimension of the table. The iteration is handled automatically, which can make your code shorter and simpler.

You can mix scalars and array arguments for functions that accept multiple arguments:

```
Compute Table-Median = Function Median(Arg1 Table-One(ALL))
```

“Example: processing tables using intrinsic functions” on page 80

Related tasks
“Using intrinsic functions (built-in functions)” on page 32
“Converting data items (intrinsic functions)” on page 104
“Evaluating data items (intrinsic functions)” on page 106

Related references
Intrinsic functions (*COBOL for Linux on x86 Language Reference*)

Example: processing tables using intrinsic functions

These examples show how you can apply an intrinsic function to some or all of the elements in a table by using the ALL subscript.

Assuming that `Table-Two` is a 2 x 3 x 2 array, the following statement adds the values in elements `Table-Two(1,3,1)`, `Table-Two(1,3,2)`, `Table-Two(2,3,1)`, and `Table-Two(2,3,2)`:

```
Compute Table-Sum = FUNCTION SUM (Table-Two(ALL, 3, ALL))
```

The following example computes various salary values for all the employees whose salaries are encoded in `Employee-Table`:

```
01  Employee-Table.
    05 Emp-Count      Pic s9(4) usage binary.
    05 Emp-Record     Occurs 1 to 500 times
                        depending on Emp-Count.
       10 Emp-Name    Pic x(20).
       10 Emp-Idme    Pic 9(9).
       10 Emp-Salary  Pic 9(7)v99.
. . .
Procedure Division.
    Compute Max-Salary    = Function Max(Emp-Salary(ALL))
    Compute I             = Function Ord-Max(Emp-Salary(ALL))
    Compute Avg-Salary    = Function Mean(Emp-Salary(ALL))
    Compute Salary-Range  = Function Range(Emp-Salary(ALL))
    Compute Total-Payroll = Function Sum(Emp-Salary(ALL))
```

Chapter 5. Selecting and repeating program actions

Use COBOL control language to choose program actions based on the outcome of logical tests, to iterate over selected parts of your program and data, and to identify statements to be performed as a group.

These controls include the IF, EVALUATE, and PERFORM statements, and the use of switches and flags.

Related tasks
“Selecting program actions” on page 81
“Repeating program actions” on page 88

Selecting program actions

You can provide for different program actions depending on the tested value of one or more data items.

The IF and EVALUATE statements in COBOL test one or more data items by means of a conditional expression.

Related tasks
“Coding a choice of actions” on page 81
“Coding conditional expressions” on page 85

Related references
IF statement (*COBOL for Linux on x86 Language Reference*)
EVALUATE statement (*COBOL for Linux on x86 Language Reference*)

Coding a choice of actions

Use IF . . . ELSE to code a choice between two processing actions. (The word THEN is optional.) Use the EVALUATE statement to code a choice among three or more possible actions.

```
IF condition-p
  statement-1
ELSE
  statement-2
END-IF
```

When one of two processing choices is no action, code the IF statement with or without ELSE. Because the ELSE clause is optional, you can code the IF statement as follows:

```
IF condition-q
  statement-1
END-IF
```

Such coding is suitable for simple cases. For complex logic, you probably need to use the ELSE clause. For example, suppose you have nested IF statements in which there is an action for only one of the processing choices. You could use the ELSE clause and code the null branch of the IF statement with the CONTINUE statement:

```
IF condition-q
  statement-1
ELSE
  CONTINUE
END-IF
```

The EVALUATE statement is an expanded form of the IF statement that allows you to avoid nesting IF statements, a common source of logic errors and debugging problems.

© Copyright IBM Corp. 2021

Related tasks
“Using nested IF statements” on page 82
“Using the EVALUATE statement” on page 83
“Coding conditional expressions” on page 85

Using nested IF statements

If an IF statement contains an IF statement as one of its possible branches, the IF statements are said to be *nested*. Theoretically, there is no limit to the depth of nested IF statements.

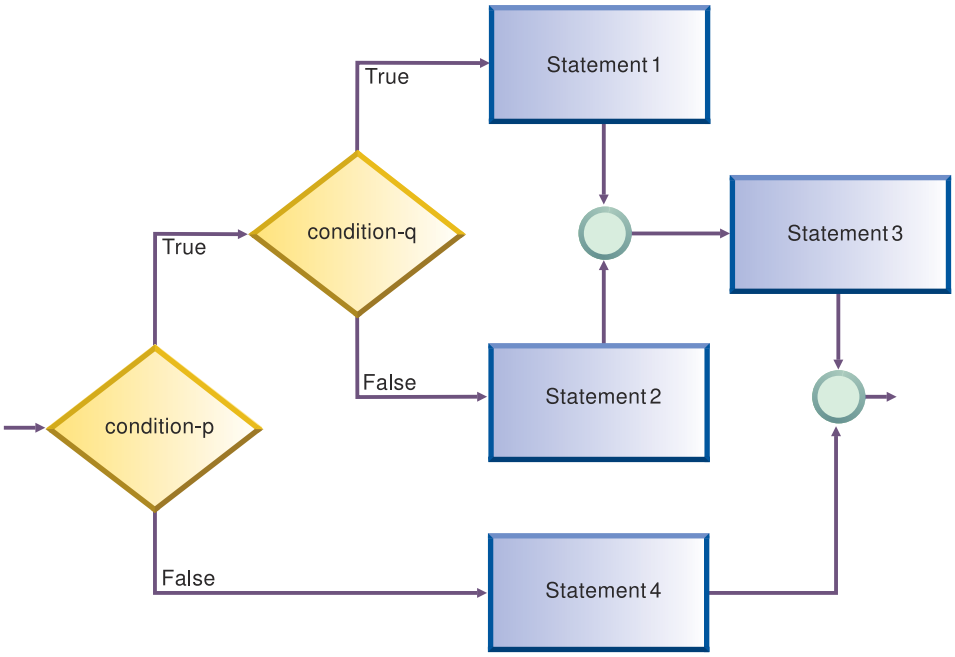
However, use nested IF statements sparingly. The logic can be difficult to follow, although explicit scope terminators and indentation can help. If a program has to test a variable for more than two values, EVALUATE is probably a better choice.

The following pseudocode depicts a nested IF statement:

```
IF condition-p
  IF condition-q
    statement-1
  ELSE
    statement-2
  END-IF
  statement-3
ELSE
  statement-4
END-IF
```

In the pseudocode above, an IF statement and a sequential structure are nested in one branch of the outer IF. In this structure, the END-IF that closes the nested IF is very important. Use END-IF instead of a period, because a period would end the outer IF structure also.

The following figure shows the logic structure of the pseudocode above.



Related tasks
“Coding a choice of actions” on page 81

Related references
Explicit scope terminators (*COBOL for Linux on x86 Language Reference*)

Using the EVALUATE statement

You can use the EVALUATE statement instead of a series of nested IF statements to test several conditions and specify a different action for each. Thus you can use the EVALUATE statement to implement a *case structure* or decision table.

You can also use the EVALUATE statement to cause multiple conditions to lead to the same processing, as shown in these examples:

"Example: EVALUATE using THRU phrase" on page 83
"Example: EVALUATE using multiple WHEN phrases" on page 84

In an EVALUATE statement, the operands before the WHEN phrase are referred to as *selection subjects*, and the operands in the WHEN phrase are called the *selection objects*. Selection subjects can be identifiers, literals, conditional expressions, or the word TRUE or FALSE. Selection objects can be identifiers, literals, conditional or arithmetic expressions, or the word TRUE, FALSE, or ANY.

You can separate multiple selection subjects with the ALSO phrase. You can separate multiple selection objects with the ALSO phrase. The number of selection objects within each set of selection objects must be equal to the number of selection subjects, as shown in this example:

"Example: EVALUATE testing several conditions" on page 84

Identifiers, literals, or arithmetic expressions that appear within a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects. Conditions or the word TRUE or FALSE that appear in a selection object must correspond to a conditional expression or the word TRUE or FALSE in the set of selection subjects. (You can use the word ANY as a selection object to correspond to any type of selection subject.)

The execution of the EVALUATE statement ends when one of the following conditions occurs:

- The statements associated with the selected WHEN phrase are performed.
- The statements associated with the WHEN OTHER phrase are performed.
- No WHEN conditions are satisfied.

WHEN phrases are tested in the order that they appear in the source program. Therefore, you should order these phrases for the best performance. First code the WHEN phrase that contains selection objects that are most likely to be satisfied, then the next most likely, and so on. An exception is the WHEN OTHER phrase, which must come last.

Related tasks
"Coding a choice of actions" on page 81

Related references
EVALUATE statement (*COBOL for Linux on x86 Language Reference*)
General relation conditions (*COBOL for Linux on x86 Language Reference*)

Example: EVALUATE using THRU phrase

This example shows how you can code several conditions in a range of values to lead to the same processing action by coding the THRU phrase. Operands in a THRU phrase must be of the same class.

In this example, CARPOOL-SIZE is the *selection subject*; 1, 2, and 3 THRU 6 are the *selection objects*:

```
EVALUATE CARPOOL-SIZE
  WHEN 1
    MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
  WHEN 2
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  WHEN 3 THRU 6
    MOVE "SMALL GROUP" TO PRINT-CARPOOL STATUS
  WHEN OTHER
    MOVE "BIG GROUP" TO PRINT-CARPOOL STATUS
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF CARPOOL-SIZE = 1 THEN
  MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
ELSE
  IF CARPOOL-SIZE = 2 THEN
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  ELSE
    IF CARPOOL-SIZE >= 3 and CARPOOL-SIZE <= 6 THEN
      MOVE "SMALL GROUP" TO PRINT-CARPOOL-STATUS
    ELSE
      MOVE "BIG GROUP" TO PRINT-CARPOOL-STATUS
    END-IF
  END-IF
END-IF
```

Example: EVALUATE using multiple WHEN phrases

The following example shows that you can code multiple WHEN phrases if several conditions should lead to the same action. Doing so gives you more flexibility than using only the THRU phrase, because the conditions do not have to evaluate to values in a range nor have the same class.

```
EVALUATE MARITAL-CODE
  WHEN "M"
    ADD 2 TO PEOPLE-COUNT
  WHEN "S"
  WHEN "D"
  WHEN "W"
    ADD 1 TO PEOPLE-COUNT
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF MARITAL-CODE = "M" THEN
  ADD 2 TO PEOPLE-COUNT
ELSE
  IF MARITAL-CODE = "S" OR
     MARITAL-CODE = "D" OR
     MARITAL-CODE = "W" THEN
       ADD 1 TO PEOPLE-COUNT
  END-IF
END-IF
```

Example: EVALUATE testing several conditions

This example shows the use of the ALSO phrase to separate two selection subjects (`True ALSO True`) and to separate the two corresponding selection objects within each set of selection objects (for example, `When A + B < 10 Also C = 10`).

Both selection objects in a WHEN phrase must satisfy the TRUE, TRUE condition before the associated action is performed. If both objects do not evaluate to TRUE, the next WHEN phrase is processed.

```
Identification Division.
  Program-ID. MiniEval.
Environment Division.
  Configuration Section.
Data Division.
  Working-Storage Section.
  01   Age             Pic  999.
  01   Sex             Pic  X.
  01   Description     Pic  X(15).
  01   A               Pic  999.
  01   B               Pic  9999.
  01   C               Pic  9999.
  01   D               Pic  9999.
  01   E               Pic  99999.
  01   F               Pic  999999.
Procedure Division.
  PN01.
    Evaluate True Also True
```

```
      When Age < 13 Also Sex = "M"
        Move "Young Boy" To Description
      When Age < 13 Also Sex = "F"
        Move "Young Girl" To Description
      When Age > 12 And Age < 20 Also Sex = "M"
        Move "Teenage Boy" To Description
      When Age > 12 And Age < 20 Also Sex = "F"
        Move "Teenage Girl" To Description
      When Age > 19 Also Sex = "M"
        Move "Adult Man" To Description
      When Age > 19 Also Sex = "F"
        Move "Adult Woman" To Description
      When Other
        Move "Invalid Data" To Description
    End-Evaluate
    Evaluate True Also True
      When A + B < 10 Also C = 10
        Move "Case 1" To Description
      When A + B > 50 Also C = ( D + E ) / F
        Move "Case 2" To Description
      When Other
        Move "Case Other" To Description
    End-Evaluate
    Stop Run.
```

Coding conditional expressions

Using the IF and EVALUATE statements, you can code program actions that will be performed depending on the truth value of a conditional expression.

You can specify the following conditions:

- Relation conditions, such as:
  - Numeric comparisons
  - Alphanumeric comparisons
  - DBCS comparisons
  - National comparisons
- Class conditions; for example, to test whether a data item:
  - IS NUMERIC
  - IS ALPHABETIC
  - IS ALPHABETIC-LOWER
  - IS ALPHABETIC-UPPER
  - IS DBCS
  - IS KANJI
- Condition-name conditions, to test the value of a conditional variable that you define
- Sign conditions, to test whether a numeric operand IS POSITIVE, NEGATIVE, or ZERO
- Switch-status conditions, to test the status of UPSI switches that you name in the SPECIAL-NAMES paragraph
- Complex conditions, such as:
  - Negated conditions; for example, NOT (A IS EQUAL TO B)
  - Combined conditions (conditions combined with logical operators AND or OR)

Related concepts
“Switches and flags” on page 86

Related tasks
“Defining switches and flags” on page 86
“Resetting switches and flags” on page 87
“Checking for incompatible data (numeric class test)” on page 48

“Comparing national (UTF-16) data” on page 190
“Testing for valid DBCS characters” on page 196

Related references
“UPSI” on page 300
General relation conditions (*COBOL for Linux on x86 Language Reference*)
Class condition (*COBOL for Linux on x86 Language Reference*)
Rules for condition-name entries (*COBOL for Linux on x86 Language Reference*)
Sign condition (*COBOL for Linux on x86 Language Reference*)
Combined conditions (*COBOL for Linux on x86 Language Reference*)

Switches and flags

Some program decisions are based on whether the value of a data item is true or false, on or off, yes or no. Control these two-way decisions by using level-88 items with meaningful names (*condition-names*) to act as switches.

Other program decisions depend on the particular value or range of values of a data item. When you use condition-names to give more than just on or off values to a field, the field is generally referred to as a *flag*.

Flags and switches make your code easier to change. If you need to change the values for a condition, you have to change only the value of that level-88 condition-name.

For example, suppose a program uses a condition-name to test a field for a given salary range. If the program must be changed to check for a different salary range, you need to change only the value of the condition-name in the `DATA DIVISION`. You do not need to make changes in the `PROCEDURE DIVISION`.

Related tasks
“Defining switches and flags” on page 86
“Resetting switches and flags” on page 87

Defining switches and flags

In the `DATA DIVISION`, define level-88 items that will act as switches or flags, and give them meaningful names.

To test for more than two values with flags, assign more than one condition-name to a field by using multiple level-88 items.

The reader can easily follow your code if you choose meaningful condition-names and if the values assigned to them have some association with logical values.

“Example: switches” on page 86
“Example: flags” on page 87

Example: switches

The following examples show how you can use level-88 items to test for various binary-valued (on-off) conditions in your program.

For example, to test for the end-of-file condition for an input file named Transaction-File, you can use the following data definitions:

```
WORKING-STORAGE SECTION.
01  Switches.
    05  Transaction-EOF-Switch  Pic X  value space.
        88  Transaction-EOF     value "y".
```

The level-88 description says that a condition named Transaction-EOF is in effect when Transaction-EOF-Switch has value 'y'. Referencing Transaction-EOF in the PROCEDURE DIVISION expresses the same condition as testing Transaction-EOF-Switch = "y". For example, the following statement causes a report to be printed only if Transaction-EOF-Switch has been set to 'y':

```
If Transaction-EOF Then
    Perform Print-Report-Summary-Lines
End-if
```

Example: flags

The following examples show how you can use several level-88 items together with an EVALUATE statement to determine which of several conditions in a program is true.

Consider for example a program that updates a main file. The updates are read from a transaction file. The records in the file contain a field that indicates which of the three functions is to be performed: add, change, or delete. In the record description of the input file, code a field for the function code using level-88 items:

```
01  Transaction-Input Record
    05  Transaction-Type         Pic X.
        88  Add-Transaction      Value "A".
        88  Change-Transaction   Value "C".
        88  Delete-Transaction   Value "D".
```

The code in the PROCEDURE DIVISION for testing these condition-names to determine which function is to be performed might look like this:

```
Evaluate True
  When Add-Transaction
    Perform Add-Main-Record-Paragraph
  When Change-Transaction
    Perform Update-Existing-Record-Paragraph
  When Delete-Transaction
    Perform Delete-Main-Record-Paragraph
End-Evaluate
```

Resetting switches and flags

Throughout your program, you might need to reset switches or flags to the original values they had in their data descriptions. To do so, either use a SET statement or define a data item to move to the switch or flag.

When you use the SET *condition-name* TO TRUE statement, the switch or flag is set to the original value that it was assigned in its data description. For a level-88 item that has multiple values, SET *condition-name* TO TRUE assigns the first value (A in the example below):

```
88 Record-is-Active Value "A" "O" "S"
```

Using the SET statement and meaningful condition-names makes it easier for readers to follow your code.

“Example: set switch on” on page 88
“Example: set switch off” on page 88

Example: set switch on

The following examples show how you can set a switch on by coding a SET statement that moves the condition name value to the conditional variable.

For example, the SET statement in the following example has the same effect as coding the statement `Move "y" to Transaction-EOF-Switch`:

```
01  Switches
    05  Transaction-EOF-Switch   Pic X  Value space.
        88  Transaction-EOF             Value "y".
. . .
Procedure Division.
000-Do-Main-Logic.
    Perform 100-Initialize-Paragraph
    Read Update-Transaction-File
      At End Set Transaction-EOF to True
    End-Read
```

The following example shows how to assign a value to a field in an output record based on the transaction code of an input record:

```
01  Input-Record.
    05  Transaction-Type         Pic X(9).
01  Data-Record-Out.
    05  Data-Record-Type         Pic X.
        88 Record-Is-Active          Value "A".
        88 Record-Is-Suspended       Value "S".
        88 Record-Is-Deleted         Value "D".
    05  Key-Field                Pic X(5).
. . .
Procedure Division.
    Evaluate Transaction-Type of Input-Record
      When "ACTIVE"
        Set Record-Is-Active to TRUE
      When "SUSPENDED"
        Set Record-Is-Suspended to TRUE
      When "DELETED"
        Set Record-Is-Deleted to TRUE
    End-Evaluate
```

Example: set switch off

The following example shows how you can set a switch off by coding a MOVE statement that moves the condition name value to the conditional variable.

For example, you can use a data item called SWITCH-OFF to set an on-off switch to off, as in the following code, which resets a switch to indicate that end-of-file has not been reached:

```
01  Switches
    05  Transaction-EOF-Switch       Pic X  Value space.
        88  Transaction-EOF                Value "y".
01  SWITCH-OFF                      Pic X  Value "n".
. . .
Procedure Division.
    . . .
    Move SWITCH-OFF to Transaction-EOF-Switch
```

Repeating program actions

Use a PERFORM statement to repeat the same code (that is, loop) either a specified number of times or based on the outcome of a decision.

You can also use a PERFORM statement to execute a paragraph and then implicitly return control to the next executable statement. In effect, this PERFORM statement is a way of coding a closed subroutine that you can enter from many different parts of the program.

PERFORM statements can be inline or out-of-line.

Related tasks
“Choosing inline or out-of-line PERFORM” on page 89
“Coding a loop” on page 90
“Looping through a table” on page 90
“Executing multiple paragraphs
or sections” on page 91

Related references
PERFORM statement (*COBOL for Linux on x86 Language Reference*)

Choosing inline or out-of-line PERFORM

An inline PERFORM is an imperative statement that is executed in the normal flow of a program; an out-of-line PERFORM entails a branch to a named paragraph and an implicit return from that paragraph.

To determine whether to code an inline or out-of-line PERFORM statement, answer the following questions:

- Is the PERFORM statement used in several places?

  Use an out-of-line PERFORM when you want to use the same portion of code in several places in your program.
- Which placement of the statement will be easier to read?

  If the code to be performed is short, an inline PERFORM can be easier to read. But if the code extends over several screens, the logical flow of the program might be clearer if you use an out-of-line PERFORM. (Each paragraph in structured programming should perform one logical function, however.)
- What are the efficiency tradeoffs?

  An inline PERFORM avoids the overhead of branching that occurs with an out-of-line PERFORM. But even out-of-line PERFORM coding can improve code optimization, so efficiency gains should not be overemphasized.

In the 1974 COBOL standard, the PERFORM statement is out-of-line and thus requires a branch to a separate procedure and an implicit return. If the performed procedure is in the subsequent sequential flow of your program, it is also executed in that logic flow. To avoid this additional execution, place the procedure outside the normal sequential flow (for example, after the GOBACK) or code a branch around it.

The subject of an inline PERFORM is an imperative statement. Therefore, you must code statements (other than imperative statements) within an inline PERFORM with explicit scope terminators.

“Example: inline PERFORM statement” on page 89

Example: inline PERFORM statement

This example shows the structure of an inline PERFORM statement that has the required scope terminators and the required END-PERFORM phrase.

```
   Perform 100-Initialize-Paragraph
* The following statement is an inline PERFORM:
   Perform Until Transaction-EOF
      Read Update-Transaction-File Into WS-Transaction-Record
         At End
            Set Transaction-EOF To True
         Not At End
            Perform 200-Edit-Update-Transaction
            If No-Errors
               Perform 300-Update-Commuter-Record
            Else
               Perform 400-Print-Transaction-Errors
* End-If is a required scope terminator
            End-If
            Perform 410-Re-Initialize-Fields
* End-Read is a required scope terminator
```

```
      End-Read
   End-Perform
```

Coding a loop

Use the PERFORM . . . TIMES statement to execute a procedure a specified number of times.

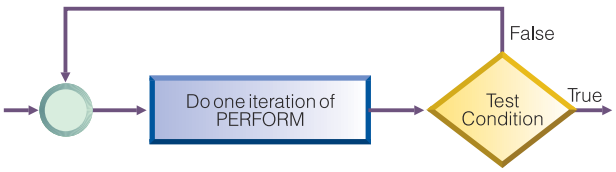
```
PERFORM 010-PROCESS-ONE-MONTH 12 TIMES
INSPECT . . .
```

In the example above, when control reaches the PERFORM statement, the code for the procedure 010-PROCESS-ONE-MONTH is executed 12 times before control is transferred to the INSPECT statement.

Use the PERFORM . . . UNTIL statement to execute a procedure until a condition you choose is satisfied. You can use either of the following forms:

```
PERFORM . . . WITH TEST AFTER  . . . . UNTIL . . .
PERFORM . . . [WITH TEST BEFORE] . . . UNTIL . . .
```

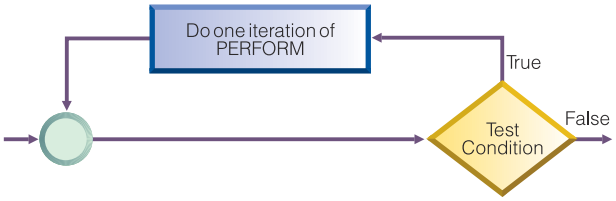
Use the PERFORM . . . WITH TEST AFTER . . . UNTIL statement if you want to execute the procedure at least once, and test before any subsequent execution. This statement is equivalent to a do-until structure:



In the following example, the implicit WITH TEST BEFORE phrase provides a do-while structure:

```
PERFORM  010-PROCESS-ONE-MONTH
  UNTIL MONTH GREATER THAN 12
INSPECT . . .
```

When control reaches the PERFORM statement, the condition MONTH GREATER THAN 12 is tested. If the condition is satisfied, control is transferred to the INSPECT statement. If the condition is not satisfied, 010-PROCESS-ONE-MONTH is executed, and the condition is tested again. This cycle continues until the condition tests as true. (To make your program easier to read, you might want to code the WITH TEST BEFORE clause.)



Looping through a table

You can use the PERFORM . . . VARYING statement to initialize a table. In this form of the PERFORM statement, a variable is increased or decreased and tested until a condition is satisfied.

Thus you use the PERFORM statement to control looping through a table. You can use either of these forms:

```
PERFORM . . . WITH TEST AFTER  . . . . VARYING . . . UNTIL . . .
PERFORM . . . [WITH TEST BEFORE] . . . VARYING . . . UNTIL . . .
```

The following section of code shows an example of looping through a table to check for invalid data:

```
PERFORM TEST AFTER VARYING WS-DATA-IX
      FROM 1 BY 1 UNTIL WS-DATA-IX = 12
   IF WS-DATA (WS-DATA-IX) EQUALS SPACES
      SET SERIOUS-ERROR TO TRUE
      DISPLAY ELEMENT-NUM-MSG5
   END-IF
END-PERFORM
INSPECT . . .
```

When control reaches the PERFORM statement above, WS-DATA-IX is set equal to 1 and the PERFORM statement is executed. Then the condition WS-DATA-IX = 12 is tested. If the condition is true, control drops through to the INSPECT statement. If the condition is false, WS-DATA-IX is increased by 1, the PERFORM statement is executed, and the condition is tested again. This cycle of execution and testing continues until WS-DATA-IX is equal to 12.

The loop above controls input-checking for the 12 fields of item WS-DATA. Empty fields are not allowed in the application, so the section of code loops and issues error messages as appropriate.

Executing multiple paragraphs or sections

In structured programming, you usually execute a single paragraph. However, you can execute a group of paragraphs, or a single section or group of sections, by coding the PERFORM . . . THRU statement.

When you use the PERFORM . . . THRU statement, code a paragraph-EXIT statement to clearly indicate the end point of a series of paragraphs.

Related tasks
“Processing table items
using intrinsic functions” on page 79

Related references
EXIT PERFORM or EXIT PERFORM CYCLE statement
(*COBOL for Linux on x86 Language Reference*)
EXIT PARAGRAPH or EXIT SECTION statement
(*COBOL for Linux on x86 Language Reference*)

Chapter 6. Handling strings

COBOL provides language constructs for performing many different operations on string data items.

For example, you can:

- Join or split data items.
- Manipulate null-terminated strings, such as count or move characters.
- Refer to substrings by their ordinal position and, if needed, length.
- Tally and replace data items, such as count the number of times a specific character occurs in a data item.
- Convert data items, such as change to uppercase or lowercase.
- Evaluate data items, such as determine the length of a data item.

Related tasks
"Joining data items (STRING)" on page 93
"Splitting data items (UNSTRING)" on page 95
"Manipulating null-terminated strings" on page 98
"Referring to substrings of data items" on page 99
"Tallying and replacing data items (INSPECT)" on page 102
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 106
Chapter 10, "Processing data in an international environment," on page 175

Joining data items (STRING)

Use the STRING statement to join all or parts of several data items or literals into one data item. One STRING statement can take the place of several MOVE statements.

The STRING statement transfers data into a receiving data item in the order that you indicate. In the STRING statement you also specify:

- A delimiter for each set of sending fields that, if encountered, causes those sending fields to stop being transferred (DELIMITED BY phrase)
- (Optional) Action to be taken if the receiving field is filled before all of the sending data has been processed (ON OVERFLOW phrase)
- (Optional) An integer data item that indicates the leftmost character position within the receiving field into which data should be transferred (WITH POINTER phrase)

The receiving data item must not be an edited item, or a display or national floating-point item. If the receiving data item has:

- USAGE DISPLAY, each identifier in the statement except the POINTER identifier must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, each identifier in the statement except the POINTER identifier must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, each identifier in the statement except the POINTER identifier must have USAGE DISPLAY-1, and each literal in the statement must be DBCS

© Copyright IBM Corp. 2021

Only that portion of the receiving field into which data is written by the STRING statement is changed.

"Example: STRING statement" on page 94

Related tasks
"Handling errors in joining and splitting strings" on page 163

Related references
STRING statement (*COBOL for Linux on x86 Language Reference*)

Example: STRING statement

The following example shows the STRING statement selecting and formatting information from a record into an output line.

The FILE SECTION defines the following record:

```
01  RCD-01.
    05  CUST-INFO.
        10  CUST-NAME    PIC X(15).
        10  CUST-ADDR    PIC X(35).
    05  BILL-INFO.
        10  INV-NO       PIC X(6).
        10  INV-AMT      PIC $$,$$$.99.
        10  AMT-PAID     PIC $$,$$$.99.
        10  DATE-PAID    PIC X(8).
        10  BAL-DUE      PIC $$,$$$.99.
        10  DATE-DUE     PIC X(8).
```

The WORKING-STORAGE SECTION defines the following fields:

```
77  RPT-LINE            PIC X(120).
77  LINE-POS            PIC S9(3).
77  LINE-NO             PIC 9(5) VALUE 1.
77  DEC-POINT           PIC X VALUE ".".
```

The record RCD-01 contains the following information (the symbol *b* indicates a blank space):

```
J.B.bSMITHbbbbb
444bSPRINGbST.,bCHICAGO,bILL.bbbbbb
A14275
$4,736.85
$2,400.00
09/22/76
$2,336.85
10/22/76
```

In the PROCEDURE DIVISION, these settings occur before the STRING statement:

- RPT-LINE is set to SPACES.
- LINE-POS, the data item to be used as the POINTER field, is set to 4.

Here is the STRING statement:

```
STRING
   LINE-NO SPACE CUST-INFO INV-NO SPACE DATE-DUE SPACE
      DELIMITED BY SIZE
   BAL-DUE
      DELIMITED BY DEC-POINT
   INTO RPT-LINE
   WITH POINTER LINE-POS.
```

Because the POINTER field LINE-POS has value 4 before the STRING statement is performed, data is moved into the receiving field RPT-LINE beginning at character position 4. Characters in positions 1 through 3 are unchanged.

The sending items that specify DELIMITED BY SIZE are moved in their entirety to the receiving field. Because BAL-DUE is delimited by DEC-POINT, the moving of BAL-DUE to the receiving field stops when a decimal point (the value of DEC-POINT) is encountered.

STRING results

When the STRING statement is performed, items are moved into RPT-LINE as shown in the table below.

Item	Positions
LINE-NO	4 - 8
Space	9
CUST-INFO	10 - 59
INV-NO	60 - 65
Space	66
DATE-DUE	67 - 74
Space	75
Portion of BAL-DUE that precedes the decimal point	76 - 81

After the STRING statement is performed, the value of LINE-POS is 82, and RPT-LINE has the values shown below.

Splitting data items (UNSTRING)

Use the UNSTRING statement to split a sending field into several receiving fields. One UNSTRING statement can take the place of several MOVE statements.

In the UNSTRING statement you can specify:

- Delimiters that, when one of them is encountered in the sending field, cause the current receiving field to stop receiving and the next, if any, to begin receiving (DELIMITED BY phrase)
- A field for the delimiter that, when encountered in the sending field, causes the current receiving field to stop receiving (DELIMITER IN phrase)
- An integer data item that stores the number of characters placed in the current receiving field (COUNT IN phrase)
- An integer data item that indicates the leftmost character position within the sending field at which UNSTRING processing should begin (WITH POINTER phrase)
- An integer data item that stores a tally of the number of receiving fields that are acted on (TALLYING IN phrase)
- Action to be taken if all of the receiving fields are filled before the end of the sending data item is reached (ON OVERFLOW phrase)

The sending data item and the delimiters in the DELIMITED BY phrase must be of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

Receiving data items can be of category alphabetic, alphanumeric, numeric, DBCS, or national. If numeric, a receiving data item must be zoned decimal or national decimal. If a receiving data item has:

- USAGE DISPLAY, the sending item and each delimiter item in the statement must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, the sending item and each delimiter item in the statement must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, the sending item and each delimiter item in the statement must have USAGE DISPLAY-1, and each literal in the statement must be DBCS

"Example: UNSTRING statement" on page 96

Related concepts
"Unicode and the encoding of language characters" on page 176

Related tasks
"Handling errors in joining and splitting strings" on page 163

Related references
UNSTRING statement (*COBOL for Linux on x86 Language Reference*)
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)

Example: UNSTRING statement

The following example shows the UNSTRING statement transferring selected information from an input record. Some information is organized for printing and some for further processing.

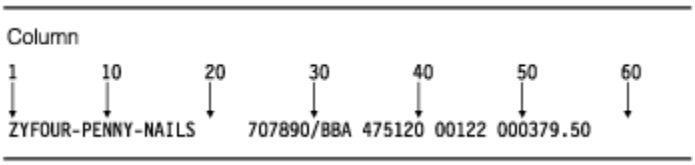
The FILE SECTION defines the following records:

```
*  Record to be acted on by the UNSTRING statement:
 01  INV-RCD.
     05  CONTROL-CHARS              PIC XX.
     05  ITEM-INDENT                PIC X(20).
     05  FILLER                     PIC X.
     05  INV-CODE                   PIC X(10).
     05  FILLER                     PIC X.
     05  NO-UNITS                   PIC 9(6).
     05  FILLER                     PIC X.
     05  PRICE-PER-M                PIC 99999.
     05  FILLER                     PIC X.
     05  RTL-AMT                    PIC 9(6).99.
*
*  UNSTRING receiving field for printed output:
 01  DISPLAY-REC.
     05  INV-NO                     PIC X(6).
     05  FILLER                     PIC X VALUE SPACE.
     05  ITEM-NAME                  PIC X(20).
     05  FILLER                     PIC X VALUE SPACE.
     05  DISPLAY-DOLS               PIC 9(6).
*
*  UNSTRING receiving field for further processing:
 01  WORK-REC.
     05  M-UNITS                    PIC 9(6).
     05  FIELD-A                    PIC 9(6).
     05  WK-PRICE REDEFINES FIELD-A  PIC 9999V99.
     05  INV-CLASS                  PIC X(3).
*
*  UNSTRING statement control fields:
 77  DBY-1                          PIC X.
 77  CTR-1                          PIC S9(3).
 77  CTR-2                          PIC S9(3).
 77  CTR-3                          PIC S9(3).
 77  CTR-4                          PIC S9(3).
 77  DLTR-1                         PIC X.
 77  DLTR-2                         PIC X.
 77  CHAR-CT                        PIC S9(3).
 77  FLDS-FILLED                    PIC S9(3).
```

In the PROCEDURE DIVISION, these settings occur before the UNSTRING statement:

- A period (.) is placed in DBY-1 for use as a delimiter.
- CHAR-CT (the POINTER field) is set to 3.

- The value zero (0) is placed in FLDS-FILLED (the TALLYING field).
- Data is read into record INV-RCD, whose format is as shown below.



Here is the UNSTRING statement:

```
* Move subfields of INV-RCD to the subfields of DISPLAY-REC
* and WORK-REC:
     UNSTRING INV-RCD
       DELIMITED BY ALL SPACES  OR "/"  OR DBY-1
       INTO ITEM-NAME    COUNT IN CTR-1
            INV-NO       DELIMITER IN DLTR-1  COUNT IN CTR-2
            INV-CLASS
            M-UNITS      COUNT IN CTR-3
            FIELD-A
            DISPLAY-DOLS DELIMITER IN DLTR-2  COUNT IN CTR-4
       WITH POINTER CHAR-CT
       TALLYING IN  FLDS-FILLED
       ON OVERFLOW  GO TO UNSTRING-COMPLETE.
```

Because the POINTER field CHAR-CT has value 3 before the UNSTRING statement is performed, the two character positions of the CONTROL-CHARS field in INV-RCD are ignored.

UNSTRING results

When the UNSTRING statement is performed, the following steps take place:

1. Positions 3 through 18 (FOUR-PENNY-NAILS) of INV-RCD are placed in ITEM-NAME, left justified in the area, and the four unused character positions are padded with spaces. The value 16 is placed in CTR-1.
2. Because ALL SPACES is coded as a delimiter, the five contiguous space characters in positions 19 through 23 are considered to be one occurrence of the delimiter.
3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character slash (/) is placed in DLTR-1, and the value 6 is placed in CTR-2.
4. Positions 31 through 33 (BBA) are placed in INV-CLASS. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 34 is bypassed.
5. Positions 35 through 40 (475120) are placed in M-UNITS. The value 6 is placed in CTR-3. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 41 is bypassed.
6. Positions 42 through 46 (00122) are placed in FIELD-A and right justified in the area. The high-order digit position is filled with a zero (0). The delimiter is SPACE, but because no field was defined as a receiving area for delimiters, the space in position 47 is bypassed.
7. Positions 48 through 53 (000379) are placed in DISPLAY-DOLS. The period (.) delimiter in DBY-1 is placed in DLTR-2, and the value 6 is placed in CTR-4.
8. Because all receiving fields have been acted on and two characters in INV-RCD have not been examined, the ON OVERFLOW statement is executed. Execution of the UNSTRING statement is completed.

After the UNSTRING statement is performed, the fields contain the values shown below.

Field	Value
DISPLAY-REC	707890 FOUR-PENNY-NAILS 000379
WORK-REC	475120000122BBA

Field	Value
CHAR-CT (the POINTER field)	55
FLDS-FILLED (the TALLYING field)	6

Manipulating null-terminated strings

You can construct and manipulate null-terminated strings (for example, strings that are passed to or from a C program) by various mechanisms.

For example, you can:

- Use null-terminated literal constants (Z". . . ").
- Use an INSPECT statement to count the number of characters in a null-terminated string:

```
MOVE 0 TO char-count
INSPECT source-field TALLYING char-count
                     FOR CHARACTERS
                     BEFORE X"00"
```

- Use an UNSTRING statement to move characters in a null-terminated string to a target field, and get the character count:

```
WORKING-STORAGE SECTION.
01  source-field          PIC X(1001).
01  char-count     COMP-5 PIC 9(4).
01  target-area.
    02 individual-char OCCURS 1 TO 1000 TIMES DEPENDING ON char-count
                         PIC X.
. . .
PROCEDURE DIVISION.
    UNSTRING source-field DELIMITED BY X"00"
                         INTO target-area
                         COUNT IN char-count
      ON OVERFLOW
        DISPLAY "source not null terminated or target too short"
    END-UNSTRING
```

- Use a SEARCH statement to locate trailing null or space characters. Define the string being examined as a table of single characters.
- Check each character in a field in a loop (PERFORM). You can examine each character in a field by using a reference modifier such as `source-field (I:1)`.

"Example: null-terminated strings" on page 98

Related tasks
"Handling null-terminated strings" on page 399

Related references
Alphanumeric literals (*COBOL for Linux on x86 Language Reference*)

Example: null-terminated strings

The following example shows several ways in which you can process null-terminated strings.

```
 01  L pic X(20) value z'ab'.
 01  M pic X(20) value z'cd'.
 01  N pic X(20).
 01  N-Length pic 99 value zero.
 01  Y pic X(13) value 'Hello, World!'.
 . . .
* Display null-terminated string:
     Inspect N tallying N-length
       for characters before initial x'00'
```

```
     Display 'N: ' N(1:N-Length) ' Length: ' N-Length
     . . .
* Move null-terminated string to alphanumeric, strip null:
     Unstring N delimited by X'00' into X
     . . .
* Create null-terminated string:
     String Y     delimited by size
            X'00' delimited by size
            into N.
     . . .
* Concatenate two null-terminated strings to produce another:
     String L     delimited by x'00'
            M     delimited by x'00'
            X'00' delimited by size
            into N.
```

Referring to substrings of data items

Refer to a substring of a data item that has USAGE DISPLAY, DISPLAY-1, or NATIONAL by using a reference modifier. You can also refer to a substring of an alphanumeric or national character string that is returned by an intrinsic function by using a reference modifier.

The following example shows how to use a reference modifier to refer to a twenty-character substring of a data item called Customer-Record:

```
Move Customer-Record(1:20) to Orig-Customer-Name
```

You code a reference modifier in parentheses immediately after the data item. As the example shows, a reference modifier can contain two values that are separated by a colon, in this order:

1. Ordinal position (from the left) of the character that you want the substring to start with
2. (Optional) Length of the required substring in *character positions*

The reference-modifier position and length for an item that has USAGE DISPLAY are expressed in terms of single-byte characters. The reference-modifier position and length for items that have USAGE DISPLAY-1 or NATIONAL are expressed in terms of DBCS character positions and national character positions, respectively.

If you omit the length in a reference modifier (coding only the ordinal position of the first character, followed by a colon), the substring extends to the end of the item. Omit the length where possible as a simpler and less error-prone coding technique.

You can refer to substrings of USAGE DISPLAY data items, including alphanumeric groups, alphanumeric-edited data items, numeric-edited data items, display floating-point data items, and zoned decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category alphanumeric. When you reference-modify an alphabetic data item, the result is of category alphabetic.

You can refer to substrings of USAGE NATIONAL data items, including national groups, national-edited data items, numeric-edited data items, national floating-point data items, and national decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category national. For example, suppose that you define a national decimal data item as follows:

```
01  NATL-DEC-ITEM  Usage National  Pic 999  Value 123.
```

You can use NATL-DEC-ITEM in an arithmetic expression because NATL-DEC-ITEM is of category numeric. But you cannot use NATL-DEC-ITEM(2:1) (the national character 2, which in hexadecimal notation is NX"3200") in an arithmetic expression, because it is of category national.

You can refer to substrings of table entries, including variable-length entries, by using reference modifiers. To refer to a substring of a table entry, code the subscript expression before the reference

modifier. For example, assume that PRODUCT-TABLE is a properly coded table of character strings. To move D to the fourth character in the second string in the table, you can code this statement:

```
MOVE 'D' to PRODUCT-TABLE (2), (4:1)
```

You can code either or both of the two values in a reference modifier as a variable or as an arithmetic expression.

"Example: arithmetic expressions as reference modifiers" on page 101

Because numeric function identifiers can be used anywhere that arithmetic expressions can be used, you can code a numeric function identifier in a reference modifier as the leftmost character position or as the length, or both.

"Example: intrinsic functions as reference modifiers" on page 102

Each number in the reference modifier must have a value of at least 1. The sum of the two numbers must not exceed the total length of the data item by more than 1 character position so that you do not reference beyond the end of the substring.

If the leftmost character position or the length value is a fixed-point noninteger, truncation occurs to create an integer. If either is a floating-point noninteger, rounding occurs to create an integer.

The following options detect out-of-range reference modifiers, and flag violations with a runtime message:

- SSRANGE compiler option
- CHECK runtime option

Related concepts
"Reference modifiers" on page 100
"Unicode and the encoding of language characters" on page 176

Related tasks
"Referring to an item in a table" on page 62

Related references
"SSRANGE" on page 281
Reference modification (*COBOL for Linux on x86 Language Reference*)
Function definitions (*COBOL for Linux on x86 Language Reference*)

Reference modifiers

Reference modifiers let you easily refer to a substring of a data item.

For example, assume that you want to retrieve the current time from the system and display its value in an expanded format. You can retrieve the current time with the ACCEPT statement, which returns the hours, minutes, seconds, and hundredths of seconds in this format:

```
HHMMSSss
```

However, you might prefer to view the current time in this format:

```
HH:MM:SS
```

Without reference modifiers, you would have to define data items for both formats. You would also have to write code to convert from one format to the other.

With reference modifiers, you do not need to provide names for the subfields that describe the TIME elements. The only data definition you need is for the time as returned by the system. For example:

```
01  REFMOD-TIME-ITEM    PIC X(8).
```

The following code retrieves and expands the time value:

```
     ACCEPT REFMOD-TIME-ITEM FROM TIME.
     DISPLAY "CURRENT TIME IS: "
* Retrieve the portion of the time value that corresponds to
*   the number of hours:
       REFMOD-TIME-ITEM (1:2)
       ":"
* Retrieve the portion of the time value that corresponds to
*   the number of minutes:
       REFMOD-TIME-ITEM (3:2)
       ":"
* Retrieve the portion of the time value that corresponds to
*   the number of seconds:
       REFMOD-TIME-ITEM (5:2)
```

"Example: arithmetic expressions as reference modifiers" on page 101
"Example: intrinsic
functions as reference modifiers" on page 102

Related tasks
"Assigning input from a
screen or file (ACCEPT)" on page 30
"Referring to substrings
of data items" on page 99
"Using national data (Unicode)
in COBOL" on page 177

Related references
Reference modification (*COBOL for Linux on x86 Language Reference*)

Example: arithmetic expressions as reference modifiers

Suppose that a field contains some right-justified characters, and you want to move those characters to another field where they will be left justified. You can do so by using reference modifiers and an INSPECT statement.

Suppose a program has the following data:

```
01  LEFTY     PIC X(30).
01  RIGHTY    PIC X(30)  JUSTIFIED RIGHT.
01  I         PIC 9(9)   USAGE BINARY.
```

The program counts the number of leading spaces and, using arithmetic expressions in a reference modifier, moves the right-justified characters into another field, justified to the left:

```
MOVE SPACES TO LEFTY
MOVE ZERO TO I
INSPECT RIGHTY
   TALLYING I FOR LEADING SPACE.
IF I IS LESS THAN LENGTH OF RIGHTY THEN
   MOVE RIGHTY ( I + 1 : LENGTH OF RIGHTY - I ) TO LEFTY
END-IF
```

The MOVE statement transfers characters from RIGHTY, beginning at the position computed as I + 1 for a length that is computed as LENGTH OF RIGHTY - I, into the field LEFTY.

Example: intrinsic functions as reference modifiers

You can use intrinsic functions in reference modifiers if you do not know the leftmost position or length of a substring at compile time.

For example, the following code fragment causes a substring of `Customer-Record` to be moved into the data item `WS-name`. The substring is determined at run time.

```
05  WS-name        Pic x(20).
05  Left-posn      Pic 99.
05  I              Pic 99.
. . .
Move Customer-Record(Function Min(Left-posn I):Function Length(WS-name)) to WS-name
```

If you want to use a noninteger function in a position that requires an integer function, you can use the `INTEGER` or `INTEGER-PART` function to convert the result to an integer. For example:

```
Move Customer-Record(Function Integer(Function Sqrt(I)): ) to WS-name
```

Related references
INTEGER (*COBOL for Linux on x86 Language Reference*)
INTEGER-PART (*COBOL for Linux on x86 Language Reference*)

Tallying and replacing data items (INSPECT)

Use the `INSPECT` statement to inspect characters or groups of characters in a data item and to optionally replace them.

Use the `INSPECT` statement to do the following tasks:

- Count the number of times a specific character occurs in a data item (`TALLYING` phrase).
- Fill a data item or selected portions of a data item with specified characters such as spaces, asterisks, or zeros (`REPLACING` phrase).
- Convert all occurrences of a specific character or string of characters in a data item to replacement characters that you specify (`CONVERTING` phrase).

You can specify one of the following data items as the item to be inspected:

- An elementary item described explicitly or implicitly as `USAGE DISPLAY`, `USAGE DISPLAY-1`, or `USAGE NATIONAL`
- An alphanumeric group item or national group item

If the inspected item has:

- `USAGE DISPLAY`, each identifier in the statement (except the `TALLYING` count field) must have `USAGE DISPLAY`, and each literal in the statement must be alphanumeric
- `USAGE NATIONAL`, each identifier in the statement (except the `TALLYING` count field) must have `USAGE NATIONAL`, and each literal in the statement must be national
- `USAGE DISPLAY-1`, each identifier in the statement (except the `TALLYING` count field) must have `USAGE DISPLAY-1`, and each literal in the statement must be a DBCS literal

"Examples: INSPECT statement" on page 103

Related concepts
"Unicode and the encoding of language characters" on page 176

Related references
INSPECT statement (*COBOL for Linux on x86 Language Reference*)

Examples: INSPECT statement

The following examples show some uses of the INSPECT statement to examine and replace characters.

In the following example, the INSPECT statement examines and replaces characters in data item DATA-2. The number of times a leading zero (0) occurs in the data item is accumulated in COUNTR. The first instance of the character A that follows the first instance of the character C is replaced by the character 2.

```
77  COUNTR            PIC 9   VALUE ZERO.
01  DATA-2            PIC X(11).
. . .
    INSPECT DATA-2
      TALLYING COUNTR FOR LEADING "0"
      REPLACING FIRST "A" BY "2" AFTER INITIAL "C"
```

DATA-2 before	COUNTR after	DATA-2 after
00ACADEMY00	2	00AC2DEMY00
0000ALABAMA	4	0000ALABAMA
CHATHAM0000	0	CH2THAM0000

In the following example, the INSPECT statement examines and replaces characters in data item DATA-3. Each character that precedes the first instance of a quotation mark (") is replaced by the character 0.

```
77  COUNTR            PIC 9   VALUE ZERO.
01  DATA-3            PIC X(8).
. . .
    INSPECT DATA-3
      REPLACING CHARACTERS BY ZEROS BEFORE INITIAL QUOTE
```

DATA-3 before	COUNTR after	DATA-3 after
456"ABEL	0	000"ABEL
ANDES"12	0	00000"12
"TWAS BR	0	"TWAS BR

The following example shows the use of INSPECT CONVERTING with AFTER and BEFORE phrases to examine and replace characters in data item DATA-4. All characters that follow the first instance of the character / but that precede the first instance of the character ? (if any) are translated from lowercase to uppercase.

```
01  DATA-4            PIC X(11).
. . .
    INSPECT DATA-4
      CONVERTING
         "abcdefghijklmnopqrstuvwxyz" TO
         "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
      AFTER INITIAL "/"
      BEFORE INITIAL"?"
```

DATA-4 before	DATA-4 after
a/five/?six	a/FIVE/?six
r/Rexx/RRRr	r/REXX/RRRR

DATA-4 before	DATA-4 after
zfour?inspe	zfour?inspe

Converting data items (intrinsic functions)

You can use intrinsic functions to convert character-string data items to several other formats, for example, to uppercase or lowercase, to reverse order, to numbers, or to one code page from another.

You can use the NATIONAL-OF and DISPLAY-OF intrinsic functions to convert to and from national (Unicode) strings.

You can also use the INSPECT statement to convert characters.

"Examples: INSPECT statement" on page 103

Related tasks
"Changing case (UPPER-CASE, LOWER-CASE)" on page 104
"Transforming to reverse order (REVERSE)" on page 105
"Converting to numbers (NUMVAL, NUMVAL-C)" on page 105
"Converting from one code page to another" on page 106

Changing case (UPPER-CASE, LOWER-CASE)

You can use the UPPER-CASE and LOWER-CASE intrinsic functions to easily change the case of alphanumeric, alphabetic, or national strings.

```
01  Item-1   Pic x(30)  Value "Hello World!".
01  Item-2   Pic x(30).
. . .
    Display Item-1
    Display Function Upper-case(Item-1)
    Display Function Lower-case(Item-1)
    Move Function Upper-case(Item-1) to Item-2
    Display Item-2
```

The code above displays the following messages on the system logical output device:

```
Hello World!
HELLO WORLD!
hello world!
HELLO WORLD!
```

The DISPLAY statements do not change the actual contents of Item-1, but affect only how the letters are displayed. However, the MOVE statement causes uppercase letters to replace the contents of Item-2.

The conversion uses the case mapping that is defined in the current locale. The length of the function result might differ from the length of the argument.

Related tasks
"Assigning input from a screen or file (ACCEPT)" on page 30
"Displaying values on a screen or in a file (DISPLAY)" on page 31

Transforming to reverse order (REVERSE)

You can reverse the order of the characters in a string by using the REVERSE intrinsic function.

```
Move Function Reverse(Orig-cust-name) To Orig-cust-name
```

For example, the statement above reverses the order of the characters in `Orig-cust-name`. If the starting value is JOHNSON*bbb*, the value after the statement is performed is *bbb*NOSNHOJ, where *b* represents a blank space.

Related concepts
"Unicode and the encoding
of language characters" on page 176

Converting to numbers (NUMVAL, NUMVAL-C)

The NUMVAL, NUMVAL-C and functions convert character strings (alphanumeric or national literals, or class alphanumeric or class national data items) to numbers. Use these functions to convert free-format character-representation numbers to numeric form so that you can process them numerically.

Use NUMVAL-C when the argument includes a currency symbol or comma or both, as shown in the example above. You can also place an algebraic sign before or after the character string, and the sign will be processed. The arguments must not exceed 18 digits when you compile with the default option ARITH(COMPAT) (*compatibility mode*) nor 31 digits when you compile with ARITH(EXTEND) (*extended mode*), not including the editing symbols.

NUMVAL, NUMVAL-C and return long (64-bit) floating-point values in compatibility mode, and return extended-precision (128-bit) floating-point values in extended mode. A reference to either of these functions represents a reference to a numeric data item.

At most 15 decimal digits can be converted accurately to long-precision floating point (as described in the related reference below about conversions and precision). If the argument to NUMVAL, NUMVAL-C, or has more than 15 digits, it is recommended that you specify the ARITH(EXTEND) compiler option so that an extended-precision function result that can accurately represent the value of the argument is returned.

When you use NUMVAL, NUMVAL-C, or , you do not need to statically define numeric data in a fixed format nor input data in a precise manner. For example, suppose you define numbers to be entered as follows:

```
01  X   Pic S999V99  leading sign is separate.
. . .
    Accept X from Console
```

The user of the application must enter the numbers exactly as defined by the PICTURE clause. For example:

```
+001.23
-300.00
```

However, using the NUMVAL function, you could code:

```
01  A   Pic x(10).
01  B   Pic S999V99.
. . .
    Accept A from Console
    Compute B = Function Numval(A)
```

The input could then be:

```
1.23
-300
```

Related concepts
“Formats for numeric data” on page 39
“Data format conversions” on page 46
“Unicode and the encoding of language characters” on page 176

Related tasks
“Converting to or from national (Unicode) representation” on page 184

Related references
“Conversions and precision” on page 47
“ARITH” on page 251

Converting from one code page to another

You can nest the DISPLAY-OF and NATIONAL-OF intrinsic functions to easily convert from any code page to any other code page.

For example, the following code converts an EBCDIC string to an ASCII string:

```
  77  EBCDIC-CCSID PIC 9(4) BINARY VALUE 1140.
  77  ASCII-CCSID  PIC 9(4) BINARY VALUE 819.
  77  Input-EBCDIC PIC X(80).
  77  ASCII-Output PIC X(80).
  . . .
* Convert EBCDIC to ASCII
      Move Function Display-of
          (Function National-of (Input-EBCDIC EBCDIC-CCSID),
              ASCII-CCSID)
      to ASCII-output
```

Related concepts
“Unicode and the encoding of language characters” on page 176

Related tasks
“Converting to or from national (Unicode) representation” on page 184

Evaluating data items (intrinsic functions)

You can use intrinsic functions to determine the ordinal position of a character in the collating sequence, to find the largest or smallest item in a series, to find the length of data item, or to determine when a program was compiled.

Use these intrinsic functions:

- CHAR and ORD to evaluate integers and single alphabetic or alphanumeric characters with respect to the collating sequence used in a program
- MAX, MIN, ORD-MAX, and ORD-MIN to find the largest and smallest items in a series of data items, including USAGE NATIONAL data items
- LENGTH to find the length of data items, including USAGE NATIONAL data items
- WHEN-COMPILED to find the date and time when a program was compiled

Related concepts
"Unicode and the encoding of language characters" on page 176

Related tasks
"Evaluating single characters for collating sequence" on page 107
"Finding the largest or smallest data item" on page 107
"Finding the length of data items" on page 109
"Finding the date of compilation" on page 110

Evaluating single characters for collating sequence

To find out the ordinal position of a given alphabetic or alphanumeric character in the collating sequence, use the ORD function with the character as the argument. ORD returns an integer that represents that ordinal position.

You can use a one-character substring of a data item as the argument to ORD:

```
IF Function Ord(Customer-record(1:1)) IS > 194 THEN . . .
```

If you know the ordinal position in the collating sequence of a character, and want to find the character that it corresponds to, use the CHAR function with the integer ordinal position as the argument. CHAR returns the required character. For example:

```
INITIALIZE Customer-Name REPLACING ALPHABETIC BY Function Char(65)
```

Related references
CHAR (*COBOL for Linux on x86 Language Reference*)
ORD (*COBOL for Linux on x86 Language Reference*)

Finding the largest or smallest data item

To determine which of two or more alphanumeric, alphabetic, or national data items has the largest value, use the MAX or ORD-MAX intrinsic function. To determine which item has the smallest value, use MIN or ORD-MIN. These functions evaluate according to the collating sequence.

To compare numeric items, including those that have USAGE NATIONAL, you can use MAX, ORD-MAX, MIN, or ORD-MIN. With these intrinsic functions, the algebraic values of the arguments are compared.

The MAX and MIN functions return the content of one of the arguments that you supply. For example, suppose that your program has the following data definitions:

```
05  Arg1   Pic x(10)  Value "THOMASSON ".
05  Arg2   Pic x(10)  Value "THOMAS    ".
05  Arg3   Pic x(10)  Value "VALLEJO   ".
```

The following statement assigns VALLEJO*bbb* to the first 10 character positions of Customer-record, where *b* represents a blank space:

```
Move Function Max(Arg1 Arg2 Arg3) To Customer-record(1:10)
```

If you used MIN instead, then THOMAS*bbbb* would be assigned.

The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position (counting from the left) of the argument that has the largest or smallest value in the list of arguments that you supply. If you used the ORD-MAX function in the previous example, the compiler would issue an error

message because the reference to a numeric function is not in a valid place. Using the same arguments as in the previous example, ORD-MAX can be used as follows:

```
Compute x = Function Ord-max(Arg1 Arg2 Arg3)
```

The statement above assigns the integer 3 to x if the same arguments are used as in the previous example. If you used ORD-MIN instead, the integer 2 would be returned. The examples above might be more realistic if Arg1, Arg2, and Arg3 were successive elements of an array (table).

If you specify a national item for any argument, you must specify all arguments as class national.

Related tasks
"Performing arithmetic" on page 48
"Processing table items using intrinsic functions" on page 79
"Returning variable results with alphanumeric or national functions" on page 108

Related references
MAX (*COBOL for Linux on x86 Language Reference*)
MIN (*COBOL for Linux on x86 Language Reference*)
ORD-MAX (*COBOL for Linux on x86 Language Reference*)
ORD-MIN (*COBOL for Linux on x86 Language Reference*)

Returning variable results with alphanumeric or national functions

The results of alphanumeric or national functions could be of varying lengths and values depending on the function arguments.

In the following example, the amount of data moved to R3 and the results of the COMPUTE statement depend on the values and sizes of R1 and R2:

```
01  R1    Pic x(10) value "e".
01  R2    Pic x(05) value "f".
01  R3    Pic x(20) value spaces.
01  L     Pic 99.
. . .
    Move Function Max(R1 R2) to R3
    Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results:

- R2 is evaluated to be larger than R1.
- The string 'f*bbbb*' is moved to R3, where *b* represents a blank space. (The unfilled character positions in R3 are padded with spaces.)
- L evaluates to the value 5.

If R1 contained 'g' instead of 'e', the code would have the following results:

- R1 would evaluate as larger than R2.
- The string 'g*bbbbbbbbb*' would be moved to R3. (The unfilled character positions in R3 would be padded with spaces.)
- The value 10 would be assigned to L.

If a program uses national data for function arguments, the lengths and values of the function results could likewise vary. For example, the following code is identical to the fragment above, but uses national data instead of alphanumeric data.

```
01  R1    Pic n(10) national value "e".
01  R2    Pic n(05) national value "f".
01  R3    Pic n(20) national value spaces.
01  L     Pic 99    national.
```

```
. . .
    Move Function Max(R1 R2) to R3
    Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results, which are similar to the first set of results except that these are for national characters:

- R2 is evaluated to be larger than R1.
- The string NX"6600 2000 2000 2000 2000" (the equivalent in national characters of 'f*bbbb*', where *b* represents a blank space), shown here in hexadecimal notation with added spaces for readability, is moved to R3. The unfilled character positions in R3 are padded with national spaces.
- L evaluates to the value 5, the length in national character positions of R2.

You might be dealing with variable-length output from alphanumeric or national functions. Plan your program accordingly. For example, you might need to think about using variable-length files when the records that you are writing could be of different lengths:

```
File Section.
FD  Output-File Recording Mode V.
01  Short-Customer-Record  Pic X(50).
01  Long-Customer-Record   Pic X(70).
Working-Storage Section.
01  R1    Pic x(50).
01  R2    Pic x(70).
. . .
    If R1 > R2
      Write Short-Customer-Record from R1
    Else
      Write Long-Customer-Record from R2
    End-if
```

Related tasks
“Finding the largest or smallest data item” on page 107
“Performing arithmetic” on page 48

Related references
MAX (*COBOL for Linux on x86 Language Reference*)

Finding the length of data items

You can use the LENGTH function in many contexts (including tables and numeric data) to determine the length of an item. For example, you can use the LENGTH function to determine the length of an alphanumeric or national literal, or a data item of any type except DBCS.

LENGTH intrinsic function

The LENGTH function returns the length of a national item (a literal, or any item that has USAGE NATIONAL, including national group items) as an integer equal to the length of the argument in national character positions. It returns the length of any other data item as an integer equal to the length of the argument in alphanumeric character positions.

The following COBOL statement demonstrates moving a data item into the field in a record that holds customer names:

```
Move Customer-name To Customer-record(1:Function Length(Customer-name))
```

LENGTH OF special register

You can also use the LENGTH OF special register, which returns the length in bytes even for national data. Coding either `Function Length(Customer-name)` or `LENGTH OF Customer-name` returns the same result for alphanumeric items: the length of `Customer-name` in bytes.

You can use the LENGTH function only where arithmetic expressions are allowed. However, you can use the LENGTH OF special register in a greater variety of contexts. For example, you can use the LENGTH OF

special register as an argument to an intrinsic function that accepts integer arguments. (You cannot use an intrinsic function as an operand to the LENGTH OF special register.) You can also use the LENGTH OF special register as a parameter in a CALL statement.

Related tasks
"Performing arithmetic" on page 48
"Creating variable-length tables (DEPENDING ON)" on page 70
"Processing table items using intrinsic functions" on page 79

Related references
"ADDR" on page 249

LENGTH (*COBOL for Linux on x86 Language Reference*)
LENGTH OF (*COBOL for Linux on x86 Language Reference*)

Finding the date of compilation

You can use the WHEN-COMPILED intrinsic function to determine when a program was compiled. The 21-character result indicates the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation, and the difference in hours and minutes from Greenwich mean time.

The first 16 positions are in the following format:

```
YYYYMMDDhhmmsshh
```

You can instead use the WHEN-COMPILED special register to determine the date and time of compilation in the following format:

```
MM/DD/YYhh.mm.ss
```

The WHEN-COMPILED special register supports only a two-digit year, and does not carry fractions of a second. You can use this special register only as the sending field in a MOVE statement.

Related references
WHEN-COMPILED (*COBOL for Linux on x86 Language Reference*)

Chapter 7. Processing files

Reading data from files and writing data to files is an essential part of most COBOL programs. Your program can retrieve information, process it as you request, and then write the results.

Before the processing, however, you must identify the files and describe their physical structure, and indicate whether they are organized as sequential, relative, indexed, or line sequential. Identifying files entails naming the files and their file systems. You might also want to set up a file status field that you can later check to verify that the processing worked properly.

The major tasks you can perform in processing a file are first opening the file and then reading it, and (depending on the type of file organization and access) adding, replacing, or deleting records.

Related concepts
"File concepts and terminology" on page 111
"File systems" on page 116
"Generation data groups" on page 123

Related tasks
"Identifying files" on page 112
"Specifying a file organization and access mode" on page 120
"Concatenating files" on page 131
"Opening optional files" on page 132
"Setting up a field for file status" on page 132
"Describing the structure of a file in detail" on page 132
"Coding input and output statements for files" on page 133
"Using Db2 files" on page 142
"Using QSAM files" on page 144
"Using SFS files" on page 145

File concepts and terminology

The following concepts and terminology are used in COBOL for Linux information about files.

System file-name
The name of a file on a hard drive or other external medium. A system file-name might be qualified by a path or other prefix to ensure uniqueness. A file exists within a specific file system.

File systems usually provide commands to manage files. The following example shows use of the `ls` command to print details about a file `Transaction.log` in the STL file system, and shows the system response:

```
> ls -l Transaction.log
-rw-r--r--    1 cobdev   cobdev     6144 May 27 17:29 Transaction.log
```

Internal file-name
A user-defined word that is specified after the FD keyword in a file description entry in the FILE SECTION, and is used inside a program to refer to a file.

In the following example, LOG-FILE is an internal file-name:

```
Data division.
File section.
FD  LOG-FILE.
01  LOG-FILE-RECORD.
```

Programs operate on internal files by using I/O statements such as OPEN, CLOSE, READ, WRITE, and START. As the term suggests, an internal file-name has no meaning outside a program.

© Copyright IBM Corp. 2021

The ASSIGN clause, described below, is the mechanism that associates an internal file-name with a system file-name.

File-system ID
A three-character string that specifies the file system in which a file is stored and through which it is accessed.

External file-name
A name that acts as an intermediary between an internal file-name and the associated system file-name. The external file-name is visible outside a program and is typically used as the name of an environment variable that is set to the *file-information* (an optional file-system ID followed by the system file-name) before the program is run.

(An external file-name is distinct from the name of an external file, that is, a file that is defined with the EXTERNAL keyword in its FD entry.)

The ASSIGN clause associates an internal file-name to a system file-name, and is specified in the FILE-CONTROL paragraph. The ASSIGN clause has three basic forms:

- ```
  SELECT internal-file-name ASSIGN TO user-defined-word
  ```
- ```
  SELECT internal-file-name ASSIGN TO 'literal'
  ```
- ```
  SELECT internal-file-name ASSIGN USING data-name
  . . .
      MOVE file-information TO data-name
  ```

user-defined word and *literal* each consist of up to three components, separated by hyphens. From left to right:

1. (Optional) Comment
2. (Optional) File-system ID
3. External file-name if a user-defined word was specified; system file-name if a literal was specified

file-information consists of at most two components, separated by a hyphen. From left to right:

1. (Optional) File-system ID
2. System file-name

Related concepts
"File systems" on page 116

Related tasks
"Identifying files" on page 112

Related references
ASSIGN clause (*COBOL for Linux on x86 Language Reference*)

Identifying files

To identify a file, you associate the file-name that is internal to your COBOL program with the corresponding system file-name by using the SELECT and the ASSIGN clauses in the FILE-CONTROL paragraph.

A simple form of this specification is:

```
SELECT internalFilename ASSIGN TO fileSystemID-externalFilename
```

internalFilename specifies the name that you use inside the program to refer to the file. The internal name is typically not the same as the external file-name or the system file-name.
```

In the ASSIGN clause, you designate the external name of the file (*externalFilename*) that you want to access, and optionally specify the file system (*fileSystemID*) in which the file exists or is to be created.

If you code *fileSystemID* to identify the file system, use one of the following values:

Db2
Db2 relational database file system.

LSQ
Line sequential file system.

QSAM
Queued sequential access method file system.

RSD
Record sequential delimited file system.

SdU
SMARTdata Utilities file system.

SFS
CICS Structured File Server file system.

STL
Standard language file system.

VSA
Virtual storage access method, which implies either the SFS or STL file system.

SFS is implied if the initial (leftmost) part of the system file-name begins with `/.:/cics/sfs`. Otherwise VSA implies the STL file system.

For `LINE SEQUENTIAL` files, you can either specify or default to `LSQ`, the line sequential file system.

For `INDEXED`, `RELATIVE`, and `SEQUENTIAL` files, you can specify `Db2`, `SdU`, `SFS`, or `STL`. For `SEQUENTIAL` files, `RSD` or `QSAM` is also a valid choice.

If you do not specify the file system for a given file, its file system is determined according to the precedence described in the related reference about precedence.

You associate an internal file-name to a system file-name using one of these items in the ASSIGN clause, as described below:

- A user-defined word
- A literal
- A data-name

Identifying files using a user-defined word:

To associate an internal file-name to a system file-name using a user-defined word, you can code a SELECT clause and an ASSIGN clause in the following form:

```
SELECT internalFilename ASSIGN TO userDefinedWord
```

The association of the internal file-name to a system file-name is completed at run time. The following example shows how you associate internal file-name `LOG-FILE` with file `Transaction.log` in the STL file system by using environment variable TRANLOG:

```
SELECT LOG-FILE ASSIGN TO TRANLOG
. . .
export TRANLOG=STL-Transaction.log
```

If an environment variable TRANLOG has not been set or is set to the null string when an `OPEN` statement for `LOG-FILE` is executed, `LOG-FILE` is associated with a file named `TRANLOG` in the default file system.

Identifying files using a literal:

To associate an internal file-name to a system file-name using a literal, you can code a SELECT clause and an ASSIGN clause in the following form:

```
SELECT internalFilename ASSIGN TO 'fileSystemID-systemFilename'
```

In the literal, you specify the system file-name and optionally the file system.

The following example shows how you can associate an internal file-name myFile with file extFile on server *sfsServer* in the SFS file system:

```
SELECT myFile ASSIGN TO 'SFS-/.:/cics/sfs/sfsServer/extFile'
```

Because the literal explicitly specifies the file system and the system file-name, the file association can be resolved at compile time.

For further details about coding the ASSIGN clause, see the appropriate related reference.

Identifying files using a data-name:

To associate an internal file-name to a system file-name using a data-name, code a SELECT clause and an ASSIGN clause in the following form:

```
SELECT internalFilename ASSIGN USING dataName
```

Move the file-system and file-name information to the variable *dataName* before the file is processed.

The following example shows how you can associate internal file-name myFile with file FebPayroll in the SdU file system:

```
SELECT myFile ASSIGN USING fileData
. . .
01 fileData PIC X(50).
. . .
    MOVE 'SdU-FebPayroll' TO fileData
    OPEN INPUT myFile
```

The association is resolved unconditionally at run time.

Related concepts
“File systems” on page 116
“Line-sequential file organization” on page 122
“Generation data groups” on page 123

Related tasks
“Identifying files to the operating system (ASSIGN)” on page 8
“Identifying Db2 files” on page 114
“Identifying SFS files” on page 115
“Concatenating files” on page 131

Related references
“Precedence of file-system determination” on page 116
“Runtime environment variables” on page 218
ASSIGN clause (*COBOL for Linux on x86 Language Reference*)

Identifying Db2 files

To identify a file in the Db2 file system, specify or default to file-system ID DB2.

The system file-name must include the schema for the underlying Db2 table. Specify the schema directly as a prefix to the system file-name.

For interoperation with CICS TXSeries or CICS TX, use schema name CICS. For example, to associate a Db2 file named TRANS in schema CICS with environment variable EXTFILENAME, you can use this command:

```
export EXTFILENAME=DB2-CICS.TRANS
```

Alternatively, for more flexibility, specify the file system and system file-name separately:

```
export COBRTOPT=FILESYS=DB2
export EXTFILENAME=CICS.TRANS
```

For more details about using DB2® files, see the appropriate related task below.

Related tasks
“Identifying files” on page 112
“Using Db2 files” on page 142
“Using file system status codes” on page 168
“Setting environment variables” on page 213

Related references
“Db2 file system” on page 117
“FILESYS” on page 298

Identifying SFS files

To identify a file in the SFS file system, specify or default to file-system ID SFS.

The system file-name must start with prefix SFS- followed by the SFS server name and file-name. You can specify the system file name if files are located on multiple SFS servers. The following example shows the system file named INVENTORY is located on the SFS server named sfsServer.

```
export EXTFN=SFS-/.:/cics/sfs/sfsServer/INVENTORY
```

If you set environment variable CICS_TK_SFS_SERVER to the required SFS server, you can use a shorthand specification for the system file-name instead of using the fully qualified name. The system file-name is prefixed with the value of CICS_TK_SFS_SERVER, followed by a forward slash, to create the fully qualified system file-name. For example:

```
export CICS_TK_SFS_SERVER=/.:/cics/sfs/sfsServer
export EXTFN=SFS-INVENTORY
```

The following `export` command shows a more complex example of how you might set an environment variable MYFILE to identify an indexed SFS file that has two alternate indexes:

```
export MYFILE="/.:/cics/sfs/sfsServer/mySFSfil(\
/.:/cics/sfs/sfsServer/mySFSfil;myaltfil1,\
/.:/cics/sfs/sfsServer/mySFSfil;myaltfil2)"
```

The command provides the following information:

- `/.:/cics/sfs/`*`sfsServer`* is the fully qualified name for the CICS server.
- *`mySFSfil`* is the base SFS file.
- `/.:/cics/sfs/`*`sfsServer`*`/`*`mySFSfil`* is the fully qualified base system file-name.
- *`myaltfil1`* and *`myaltfil2`* are the alternate index files.

For each alternate index file, the file name must be in the format of its fully qualified base system file-name followed by a semicolon (`;`) and the alternate index file name: `/.:/cics/sfs/`*`sfsServer`*`/`*`mySFSfil`*`;`*`myaltfil1`*.

A comma is required between specifications of alternate index files in the `export` command.

Related tasks
“Identifying files” on page 112
“Using SFS files” on page 145
“Using file system status codes” on page 168

Related references
“SFS file system” on page 119

Precedence of file-system determination

The file system applicable to a given `SEQUENTIAL`, `INDEXED`, or `RELATIVE` file is determined according to the following precedence, from highest to lowest.

1. The file system specified by the *assignment-name* runtime environment variable or the value of the `USING` data item that is coded in the `ASSIGN` clause
2. The file system specified by the next-to-rightmost component of the literal or user-defined word that is coded in the `ASSIGN` clause if that component is at least three characters long (and meets the other criteria described in the documentation of the `ASSIGN` clause)
3. The default file system designated by the `FILESYS` runtime option (as specified in the COBRTOPT runtime environment variable)

If no file system is determined by the preceding means, the file system defaults to SFS if the leftmost part of the system file-name begins with `/.:/cics/sfs`, otherwise to STL.

Related concepts
“File systems” on page 116

Related tasks
“Identifying files to the operating system (ASSIGN)” on page 8
“Identifying files” on page 112

Related references
“Runtime environment variables” on page 218
“FILESYS” on page 298
ASSIGN clause (*COBOL for Linux on x86 Language Reference*)

File systems

Record-oriented files that have sequential, relative, indexed, or line-sequential organization are accessed through a *file system*.

COBOL for Linux supports the following file systems for sequential, relative, and indexed files:

Db2 (Db2 relational database) file system
Lets batch COBOL programs create and access CICS files that are stored in Db2.

SdU (SMARTdata Utilities) file system
Files in the SdU file system can be shared with PL/I programs.

SFS (CICS Structured File Server) file system
One of the file systems used by CICS. CICS SFS is supplied as part of CICS. SFS files can be shared with PL/I programs.

STL (standard language) file system
Provides the basic facilities for local files.

COBOL for Linux supports the following file systems for sequential files:

QSAM (queued sequential access method) file system
Lets COBOL programs access QSAM files that are transferred from the mainframe to Linux using FTP.

RSD (record sequential delimited) file system
Lets COBOL programs share data with programs written in other languages. RSD files are sequential only, with fixed or variable-length records, and support all COBOL data types in records. Text data in records can be edited by most file editors.

You can specify the file system for a given sequential, relative, or indexed file in any of several ways. For details, see the related reference about precedence of file-system determination.

Record-oriented files that have line-sequential organization can be accessed only through the LSQ (line sequential) file system.

Db2 files are managed by the DB2 command-line utility; SFS files are managed by the `sfsadmin` command-line utility. All other files exist in the line sequential Linux file system, and are managed by standard Linux commands such as `cp`, `ls`, `mv`, and `rm`. (Do not however use the `cp` or `mv` command for SdU files, which consist of multiple component files that refer to one another internally.)

All the file systems let you use COBOL statements to read and write COBOL files. Most programs have the same behaviors in all file systems. However, files written using one file system cannot be read using a different file system.

Related concepts
"Line-sequential file organization" on page 122

Related tasks
"Identifying files to the operating system (ASSIGN)" on page 8
"Identifying files" on page 112

Related references
"Precedence of file-system determination" on page 116
"Db2 file system" on page 117
"QSAM file system" on page 118
"RSD file system" on page 119
"SdU file system" on page 119
"SFS file system" on page 119
"STL file system" on page 120

Appendix B, "IBM Z host data format considerations," on page 475

Db2 file system

The Db2 file system supports sequential, indexed, and relative files. It provides enhanced interoperation with CICS TXSeries or CICS TX, enabling batch COBOL programs to access CICS ESDS, KSDS, and RRDS files that are stored in Db2.

The implementation of the Db2 file system ensures that each COBOL operation is committed to the database so that no transactional or other database semantics show through to the COBOL program.

The Db2 database management system (DBMS) provides backup, compression, encryption, and utility functions, and also provides Db2 users with a familiar maintenance and administration protocol.

The db2 command-line utility provides administrative functions for Db2 files. For example, you might use the db2 `describe` command to print details about a file called `CICS.Transaction.log` in the Db2 file system:

```
> db2 describe table CICS.\"Transaction.log\"
```

```
                        Data type                     Column
Column name             schema    Data type name      Length   Scale Nulls
----------------------- --------- ------------------- -------- ----- ----
RBA                     SYSIBM    CHARACTER                8      0 No
```

```
F1                      SYSIBM    CHARACTER               41      0 No
F2                      SYSIBM    VARCHAR                 29      0 No
```

For more information about the functions that are provided by the db2 utility, enter the command db2.

The Db2 file system is nonhierarchical.

Restrictions:

- A given program can use Db2 files in only one database.
- The Db2 file system is not safe for use with multiple threads.

Interoperation with CICS TXSeries or CICS TX:

For interoperation with CICS TXSeries or CICS TX, there are additional requirements for Db2 files:

- The schema name for the Db2 table must be `CICS`. Specify the fully-qualified name in one of these items:
  - The `ASSIGN TO` literal, for example, `ASSIGN TO 'CICS.MYFILE'`
  - The environment variable value, for example, `export ENVAR=CICS.MYFILE`
  - The `ASSIGN USING` data-name value, for example, `MOVE 'CICS.MYFILE' TO fileData`

  The rest of the file-name after the schema must be uppercase.
- Fixed-length files have the following maximum record lengths:
  - Indexed (KSDS): 4005 bytes
  - Relative (RRDS): 4001 bytes
  - Sequential (ESDS): 4001 bytes
- Files that have record lengths that are larger than the maximum record lengths for fixed-length files must be defined as variable length.
- The maximum record length is 32,767 bytes.
- If a Db2 file is created by a COBOL program, the runtime option `FILEMODE(SMALL)` must be in effect.

Related concepts
“File organization and access mode” on page 121

Related tasks
“Identifying Db2 files” on page 114
“Using Db2 files” on page 142

Related references

Effect of CLOSE statement on file types (*COBOL for Linux on x86 Language Reference*)
Compiler limits (*COBOL for Linux on x86 Language Reference*)
DB2 Database Administration Concepts and Configuration Reference (SQL limits)

QSAM file system

The QSAM (queued sequential access method) file system supports fixed, variable, and spanned records. Using the QSAM file system, you can directly access a QSAM file that you transferred from the mainframe to Linux. QSAM files support all COBOL data types in the record.

You can obtain a QSAM file from the mainframe using FTP with the options `binary` and `quote site rdw`. If the file contains `EBCDIC` character data, compile the Linux COBOL program with `-host` to read or write the character data as `EBCDIC`. If the QSAM file already exists, you can upload the same file to the mainframe. If the file does not exist, you must create it using the correct file attributes.

Note: QSAM file system was referred to as the RAW file system in some earlier PTF documents.

Related concepts
“File organization and access mode” on page 121

Related tasks
"Using QSAM files" on page 144

RSD file system

The RSD (record sequential delimited) file system supports sequential files that have fixed or variable-length records. You can process RSD files by using the standard system file utility functions such as browse, edit, copy, delete, and print.

RSD files provide good performance. They give you the ability to port files easily between Linux and Windows-based systems and to share files between programs and applications written in different languages.

RSD files support all COBOL data types in records of fixed or variable length. Each record that is written is followed by a newline control character.

Related concepts
"File organization and access mode" on page 121

SdU file system

The SdU (SMARTdata Utilities) file system supports sequential, indexed, and relative files.

When you create an SdU file using `OPEN OUTPUT` and `WRITE` statements, multiple files are created:

- The file attributes are stored in files that have suffix `.DDMEA`.
- The primary index is stored in a file whose name begins with the character `@`, followed by the file name.
- The alternate indexes are stored in files whose names end with suffix `.@00`, `.@01`, and so forth.

Because SdU files consist of multiple files that refer to one another internally, do not attempt to copy, move, or rename them. When you delete an SdU file (using the `rm` command), be sure to also delete all its component files, some of which are hidden due to their having names that begin with a period (`.`). To list all the components of an SdU file named `Log123` in the current directory for example, use the following command:

```
ls -l *Log123* .*Log123*
```

The SdU file system conforms to 85 COBOL Standard.

Restrictions:

- SdU files must not be accessed other than from COBOL programs, because the metadata for SdU files is stored separately from the files (as described above).
- The SdU file system is not safe for use with multiple threads.

Related concepts
"File organization and access mode" on page 121

Related references

VSAM File System Reply Messages

SFS file system

The CICS SFS (Structured File Server) file system is a record-oriented file system that supports three types of file organization: sequential (entry-sequenced), relative, and indexed (clustered). The SFS file system provides the basic facilities that you need for accessing files sequentially, randomly, or dynamically.

You can process SFS files by using the standard file operations such as read, write, rewrite, and delete.

Each SFS file has one internal primary index, which defines the physical ordering of the records in the file, and can have any number of secondary indexes, which provide alternate sequences in which the records can be accessed.

All data in SFS files is managed by an SFS server. SFS provides a system tool, `sfsadmin`, for performing administrative functions such as creating files and indexes, determining which volumes are available on the SFS server, and so on, through a command-line interface. For details, see the CICS publication in the related reference.

The SFS file system is nonhierarchical. That is, when identifying SFS files, you can specify only individual file names, not directory names, after the server name.

COBOL access to SFS files is nontransactional: each operation against an SFS file is *atomic*, that is, performed either in its entirety or not at all. In the event of an SFS system failure, the result of a file operation completed by a COBOL application might not be reflected in the SFS file.

The SFS file system conforms to 85 COBOL Standard.

With the SFS file system, you can easily read and write files to be shared with PL/I programs.

Restrictions:

- The SFS file system is not safe for use with multiple threads.
- You cannot process SFS files using 64-bit COBOL for Linux programs.

Related concepts
"File organization and access mode" on page 121

Related tasks
"Identifying files" on page 112
"Identifying SFS files" on page 115
"Using SFS files" on page 145
"Improving SFS performance" on page 147

Related references

CICS TXSeries documentation

STL file system

The STL file system (standard language file system) supports sequential, indexed, and relative files. It provides the basic file facilities for accessing files.

The STL file system conforms to 85 COBOL Standard, and provides good performance and the ability to port easily between Linux and Windows-based systems.

Related concepts
"File organization and access mode" on page 121

Specifying a file organization and access mode

In the `FILE-CONTROL` paragraph, you need to define the physical structure of a file and its access mode, as shown below.

```
FILE-CONTROL.
    SELECT file ASSIGN TO FileSystemID-Filename
    ORGANIZATION IS org ACCESS MODE IS access.
```

For *org*, you can choose `SEQUENTIAL` (the default), `LINE SEQUENTIAL`, `INDEXED`, or `RELATIVE`.

For *access*, you can choose `SEQUENTIAL` (the default), `RANDOM`, or `DYNAMIC`.

Sequential and line-sequential files must be accessed sequentially. For indexed or relative files, all three access modes are possible.

File organization and access mode

You can organize your files as sequential, line-sequential, indexed, or relative. The access mode defines how COBOL reads and writes files, but not how files are organized.

You should decide on the file organization and access modes when you design your program.

The following table summarizes file organization and access modes for COBOL files.

Table 9. ***File organization and access mode***

File organization	Order of records	Records can be deleted or replaced?	Access mode
Sequential	Order in which they were written	A record cannot be deleted, but its space can be reused for a same-length record.	Sequential only
Line-sequential	Order in which they were written	A record can not be deleted, but its space can be reused for a same-length record.	Sequential only
Indexed	Collating sequence by key field	Yes	Sequential, random, or dynamic
Relative	Order of relative record numbers	Yes	Sequential, random, or dynamic

Your file-management system handles the input and output requests and record retrieval from the input-output devices.

Related concepts
“Sequential file organization” on page 121
“Line-sequential file organization” on page 122
“Indexed file organization” on page 122
“Relative file organization” on page 122
“Sequential access” on page 122
“Random access” on page 123
“Dynamic access” on page 123

Related tasks
“Specifying a file organization and access mode” on page 120

Sequential file organization

A sequential file contains records organized by the order in which they were entered. The order of the records is fixed.

Records in sequential files can be read or written only sequentially.

After you place a record into a sequential file, you cannot shorten, lengthen, or delete the record. However, you can update (`REWRITE`) a record if the length does not change. New records are added at the end of the file.

If the order in which you keep records in a file is not important, sequential organization is a good choice whether there are many records or only a few. Sequential output is also useful for printing reports.

Related concepts
“Sequential access” on page 122

Related references
"Valid COBOL statements for sequential files" on page 136

Line-sequential file organization

Line-sequential files are like sequential files, except that the records can contain only characters as data. Line-sequential files are supported by the native byte stream files of the operating system.

Line-sequential files that are created using `WRITE` statements that have the `ADVANCING` phrase can be directed to a printer or to a disk.

Related concepts
"Sequential file organization" on page 121

Related tasks
"Identifying files" on page 112

Related references
"Valid COBOL statements for line-sequential files" on page 136

Indexed file organization

An indexed file contains records ordered by a *record key*. A record key uniquely identifies a record and determines the sequence in which it is accessed with respect to other records.

Each record contains a field that contains the record key. A record key for a record might be, for example, an employee number or an invoice number.

An indexed file can also use *alternate indexes*, that is, record keys that let you access the file using a different logical arrangement of the records. For example, you could access a file through employee department rather than through employee number.

The possible record transmission (access) modes for indexed files are sequential, random, or dynamic. When indexed files are read or written sequentially, the sequence is that of the key values.

EBCDIC consideration: As with any change in the collating sequence, if your indexed file is a local EBCDIC file, the EBCDIC keys will not be recognized as such outside of your COBOL program. For example, an external sort program, unless it also has support for EBCDIC, will not sort records in the order that you might expect.

Related references
"Valid COBOL statements for indexed and relative files" on page 137

Relative file organization

A relative record file contains records ordered by their *relative key*, a record number that represents the location of the record relative to where the file begins.

For example, the first record in a file has a relative record number of 1, the tenth record has a relative record number of 10, and so forth. The records can have fixed length or variable length.

The record transmission modes for relative files are sequential, random, or dynamic. When relative files are read or written sequentially, the sequence is that of the relative record number.

Related references
"Valid COBOL statements for indexed and relative files" on page 137

Sequential access

For sequential access, code `ACCESS IS SEQUENTIAL` in the `FILE-CONTROL` paragraph.

For indexed files, records are accessed in the order of the key field selected (either primary or alternate), beginning at the current position of the file position indicator.

For relative files, records are accessed in the order of the relative record numbers.

Related concepts
“Random access” on page 123
“Dynamic access” on page 123

Related references
“File position indicator” on page 135

Random access

For random access, code `ACCESS IS RANDOM` in the `FILE-CONTROL` paragraph.

For indexed files, records are accessed according to the value you place in a key field (primary, alternate, or relative). There can be one or more alternate indexes.

For relative files, records are accessed according to the value you place in the relative key.

Related concepts
“Sequential access” on page 122
“Dynamic access” on page 123

Dynamic access

For dynamic access, code `ACCESS IS DYNAMIC` in the `FILE-CONTROL` paragraph.

Dynamic access supports a mixture of sequential and random access in the same program. With dynamic access, you can use one COBOL file definition to perform both sequential and random processing, accessing some records in sequential order and others by their keys.

For example, suppose you have an indexed file of employee records, and the employee's hourly wage forms the record key. Also, suppose your program is interested in those employees who earn between $12.00 and $18.00 per hour and those who earn $25.00 per hour and above. To access this information, retrieve the first record randomly (with a random-retrieval READ) based on the key of 1200. Next, begin reading sequentially (using `READ NEXT`) until the salary field exceeds 1800. Then switch back to a random read, this time based on a key of 2500. After this random read, switch back to reading sequentially until you reach the end of the file.

Related concepts
“Sequential access” on page 122
“Random access” on page 123

Generation data groups

A *generation data group (GDG)* is a chronological collection of related files. GDGs simplify the processing of multiple versions of related data.

Each file within a GDG is called a *generation data set (GDS)* or *generation*. (In this information, generation data sets are referred to as *generation files*. The term *file* on the workstation is equivalent to the term *data set* on the host.)

Within a GDG, the generations can have like or unlike attributes including `ORGANIZATION`, record format, and record length. If all generations in a group have consistent attributes and sequential organization, you can retrieve the generations together as a single file.

There are advantages to grouping related files. For example:

- The files in the group can be referred to by a common name.
- The files in the group are kept in generation order.
- The outdated files can be automatically discarded.

The generations within a GDG have sequentially ordered relative and absolute names that represent their age.

The relative name of a generation file is the group name followed by an integer in parentheses. For example, if the name of a group is `hlq.PAY`:

- `hlq.PAY(0)` refers to the most current generation.
- `hlq.PAY(-1)` refers to the previous generation.
- `hlq.PAY(+1)` specifies a new generation to be added.

The absolute name of a generation file contains the generation number and version number. For example, if the name of a group is `hlq.PAY`:

- `hlq.PAY.g0005v00` refers to generation file 5, version 0.
- `hlq.PAY.g0006v00` refers to generation file 6, version 0.

For more information about forming absolute and relative names, see the Related tasks.

Generation order is typically but not necessarily the same as the order in which files were added to a group. Depending on how you add generation files using absolute and relative names, you might insert a generation into an unexpected position in a group. For details, see the related reference about insertion and wrapping of generation files.

GDGs are supported in all of the COBOL for Linux file systems.

Restriction: A GDG cannot contain either an SFS indexed file that has any alternate indexes, or an SdU indexed file that requires a list of alternate indexes in the file name. The restriction is due to the ambiguity between the syntax of a parenthesized alternate index list and the syntax of GDG relative names, which also require a parenthesized expression.

For information about creating and initializing generation data groups, see the appropriate related task.

To delete, rebuild, clean up, modify, or list generation groups, or add or delete generations within a group, use the `gdgmgr` utility. To see a summary of `gdgmgr` functions, issue the following command: `gdgmgr -h`. For further details about the `gdgmgr` utility, see its `man` page.

Related tasks
“Creating generation data groups” on page 124
“Using generation data groups” on page 126

Related references
“Name format of generation files” on page 127
“Insertion and wrapping of generation files” on page 128
“Limit processing of generation data groups” on page 130
“File specification” on page 469

Creating generation data groups

To create a generation data group (GDG), first create its catalog by using the `gdgmgr` command with the `-c` flag. (A *GDG catalog* is a binary file in the Linux native file system; a GDG catalog name is of the form *gdgBaseName*`.catalogue`.)

You then populate the GDG with generation files typically by running COBOL programs that create the files.

In the Linux LSQ file system:

To create a GDG catalog in the LSQ, RSD, SdU, or STL file system, use the `gdgmgr` command with the `-c` flag. For example, to create catalog `./myGroups/transactionGroup.catalogue`, you can issue this command:

```
gdgmgr -c ./myGroups/transactionGroup
```

The catalog is created by default in the current working directory (./). You can optionally precede the catalog name in the `gdgmgr` command with a path name, as shown above. The catalog and the generation files must be created in the same directory.

In the SFS file system:

To create a GDG catalog in the SFS file system, use the gdgmgr command with the -c flag. You can specify the SFS file name in either a fully qualified form or an abbreviated form. For example, the following command creates a GDG catalog using a fully qualified SFS name:

```
gdgmgr -c /.:/cics/sfs/sfsServer/baseName
```

You can instead specify an abbreviated form of the SFS file name by first setting environment variable CICS_TK_SFS_SERVER, and then issuing the gdgmgr command using also the -F flag to specify the SFS file system. For example:

```
export CICS_TK_SFS_SERVER=/.:/cics/sfs/sfsServer
gdgmgr -F SFS -c baseName
```

To override the default GDG home directory (~/gdg) for SFS groups, set environment variable gdg_home. For example, the following commands create a GDG catalog ~/groups/forSFS/sfs/*sfsServer*/myGroup.catalogue:

```
export gdg_home=~/groups/forSFS
gdgmgr -c /.:/cics/sfs/sfsServer/myGroup
```

All SFS generation files in a given group must be on the same SFS server.

In the Db2 file system:

To create a GDG catalog in the Db2 file system, do the following steps:

1. Initialize the Db2 environment by running the profile for the Db2 instance that you want to use.
2. Set environment variable DB2DBDFT to the database for the group.
3. Use the gdgmgr command with the -F flag to specify the Db2 file system, and with the -c flag. Specify the schema directly in the required catalog name.

For example, the following commands create catalog ~/gdg/db2/db2inst1/*database*/cics.dbGroup.catalogue:

```
. /home/db2inst1/sqllib/db2profile
export DB2DBDFT=database
gdgmgr -F DB2 -c cics.dbGroup
```

The home directory for a GDG catalog for Db2 files is taken from environment variable $gdg_home or else defaults to ~/gdg.

All generation files in a generation data group must be in one database under one schema.

Related concepts
“File systems” on page 116
“Generation data groups” on page 123

Related tasks
“Identifying Db2 files” on page 114
“Identifying SFS files” on page 115
“Using generation data groups” on page 126

Related references
“Limit processing of generation data groups” on page 130

Using generation data groups

To use generation data groups, you must understand how to refer to them and how to form absolute and relative names of the generation files within a group.

Refer to an entire group by using the group name optionally followed by an asterisk in parentheses, for example: `abc.SALES(*)`. Either form denotes all the files in the group concatenated in generation order, the most current generation first. To refer to a specific generation and add it to a group, use the group name followed by either an absolute or relative reference.

Absolute names:

To name a specific generation of a group using an absolute reference, use the group name followed by an absolute generation number and version number. For example, if the name of a group is `abc.SALES`:

- `abc.SALES.g0001v00` refers to generation file 1, version 0.
- `abc.SALES.g0002v00` refers to generation file 2, version 0.

Relative names:

To name a specific generation of a group using a relative reference, use the group name followed by an integer in parentheses. For example, if the name of a group is `abc.SALES`:

- `abc.SALES(0)` refers to the most current generation.
- `abc.SALES(-1)` refers to the previous generation.
- `abc.SALES(+1)` specifies a new generation to be added.

For further details about absolute and relative names, see the related reference about name format of generation files.

If an absolute or relative reference is syntactically valid, the run time determines whether an intended generation data reference refers to a generation data group or generation file by checking for the existence of an appropriately named generation data group catalog. If the catalog is not found, the reference is treated as a normal file identifier rather than as the name of a generation data group or generation file.

An unsigned or negative relative reference resolves to the equivalent absolute file identifier; the catalog is consulted to determine the equivalent. If there is no absolute name in the catalog that corresponds with the resolved relative reference, the reference is treated as a normal file identifier.

A signed positive relative reference typically represents a new generation to be added to the group. The increment is added to the generation number of the current (zeroth) generation to form the number of a new generation. A signed positive relative reference is an alternative means of specifying the equivalent absolute name.

Generation wrapping:

If the sum of the increment and the current generation number is greater than 9999, the new generation number is formed by wrapping (subtraction of 9999). For examples, see the related reference about insertion and wrapping of generation files.

If a group does not include a generation that has the new number, and either of the following conditions is true, a new generation is added to the group in the appropriate ordinal position:

- The file is not `OPTIONAL` and the open mode is `OUTPUT`.
- The file is `OPTIONAL` and the open mode is `I-O` or `EXTEND`.

If a group already contains a generation that has the new number, the existing generation is reused and might be overwritten if the open mode is not `INPUT`.

Related concepts
“Generation data groups” on page 123

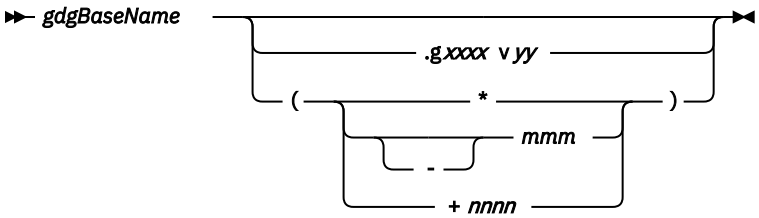
Related tasks
“Creating generation data groups” on page 124

Related references
“Name format of generation files” on page 127
“Limit processing of generation data groups” on page 130
“Insertion and wrapping of generation files” on page 128
“File specification” on page 469

Name format of generation files

The following syntax describes how to form absolute and relative references to generation data groups (GDGs) and generation files.

GDG and GDS syntax



gdgBaseName
The name of the generation data group and the base name of the GDG catalog (a binary file *gdgBaseName*`.catalogue` in the Linux native file system).

gdgBaseName can be any valid file name. However, avoid specifying a base name that looks like an absolute or relative reference, for example, `my(+1)Base`.

.gxxxxvyy
An absolute generation and version number that identify a specific generation, where:

- *xxxx* is an unsigned 4-digit decimal number from 0001 through 9999, inclusive.
- *yy* is 00. 00 is the only version number that is supported. Nonzero version numbers are ignored.

An absolute name is thus of the form *gdgBaseName*`.g`*xxxx*`v`*yy*.

****
A relative suffix that designates an entire group. All generation files in a group are concatenated in generation order, the most current generation first.

A reference to the entire group is of the form *gdgBaseName*`(*)` or *gdgBaseName*.

mmm
A relative generation number from 0 to 999, inclusive, that identifies a specific generation. The number refers to the current generation (0), the previous or less current generation (1) or (-1), and so forth. The negative sign is optional.

It is an error to refer to a nonexistent generation.

A relative name for an existing generation is thus of the form *gdgBaseName*`(`*mmm*`)` or *gdgBaseName*`(-`*mmm*`)`.

+nnnn
A positive relative generation number from 1 to 9999, inclusive, that typically identifies a generation to be created and added to a GDG.

A number greater than 1 can cause generation numbers to be skipped. For example, if only one generation exists and *gdgBaseName*`(+3)` is specified, two generations are skipped.

It is not an error to open for input a reference such as *gdgBaseName*`(+1234)` if the reference resolves to an existing file in the group.

Related concepts
"Generation data groups" on page 123

Related tasks
"Using generation data groups" on page 126

Related references
"Insertion and wrapping of generation files" on page 128
"File specification" on page 469

Insertion and wrapping of generation files

A new generation in a GDG is typically inserted after the current generation and thus becomes the new current generation.

If the sum of the increment and the current generation number is greater than 9999, *wrapping* (subtraction of 9999) occurs, and the new generation number will be less than the current generation number. In this case the new generation might be inserted before the current generation, becoming a previous (less current) generation in the group.

Consider the following initial group:

```
0: base.g0001v00
```

The number before the colon, called an *epoch number*, is used to cause insertion to occur predictably and enforces limits on generation numbers.

Typically a new generation is inserted into the group in the position indicated by the new generation number after any wrapping has occurred, and the epoch numbers within the GDG are not changed.

In each of the following examples, the least current generation is listed first, and the most current generation is listed last. *Italics* indicates a new generation that has been added to the group.

Consider the example initial group shown above. If `base(+1)` is specified, the current generation number (0001) is incremented by 1. The group as a result will contain the new generation 0002 as shown:

```
0: base.g0001v00
0: base.g0002v00
```

If `base(+4)` is then specified, the current generation number (0002) is incremented by 4. The group as a result will contain the new generation (0006) as shown:

```
0: base.g0001v00
0: base.g0002v00
0: base.g0006v00
```

If `base(+9997)` is then specified, the current generation number (0006) is incremented by 9997. The resulting generation number (10003) is greater than 9999 and is therefore wrapped to become 0004. In the resulting group this new generation (0004) will be inserted before the current generation (0006) as shown:

```
0: base.g0001v00
0: base.g0002v00
0: base.g0004v00
0: base.g0006v00
```

After this last insertion, it is not an error to open `base(+9997)` for input, because the reference to `base(+9997)` denotes the existing file `base.g0004v00` and does not cause any change to the structure of the group.

Typically, the epoch number of a new generation is the same as the epoch number of the current generation. There are however two cases in which epoch numbers are adjusted, and the insertion position of the new generation is less obvious:

- If the current generation number is greater than or equal to 9000 and the new generation number is less than 1000, wrapping will occur. But the epoch number of the new generation will increase so that the new generation can be inserted after the current generation despite the fact that the new generation has a lower generation number.

  For example, consider the following initial group:

  ```
  0: base.g1000v00
  0: base.g9000v00
  ```

  If `base(+1499)` is specified, the current generation number (9000) is incremented by 1499. The resulting generation number (10499) is greater than 9999 and is therefore wrapped to become 0500. In the resulting group the new generation (0500) is given a higher epoch number and becomes the new current generation despite having the lowest generation number in the group as shown:

  ```
  0: base.g1000v00
  0: base.g9000v00
  1: base.g0500v00
  ```

- If the current generation number is less than 1000 and the new generation number is greater than or equal to 9000, the epoch number of the new generation is decreased unless the epoch number of the current generation was already zero. In the latter case the epoch number of the existing generations is increased so that the new generation will be inserted before all the generations despite having a higher generation number.

  For example, consider the following initial group:

  ```
  0: base.g0001v00
  0: base.g0999v00
  ```

  If `base(+8501)` is specified, the current generation (0999) is incremented by 8501. The resulting generation number (9500) is less than 9999; therefore no wrapping occurs. The resulting group will contain the new generation (9500) but with epoch number 0. The epoch number of the existing generations is increased to 1. As a result the new generation becomes the least current generation in the group as shown:

  ```
  0: base.g9500v00
  1: base.g0001v00
  1: base.g0999v00
  ```

Related concepts
“Generation data groups” on page 123

Related tasks
“Using generation data groups” on page 126

Related references
“Name format of generation files” on page 127
“File specification” on page 469

Limit processing of generation data groups

Limit processing is done whenever a new generation is added to a generation data group, typically as the result of an OPEN statement. You can also do limit processing manually by using the `-C` (cleanup) option of the `gdgmgr` utility.

If the `empty` option is in effect for a group, limit processing removes all expired files from the group. If the `empty` option is not in effect for a group, expired files are removed from the group, after any addition, in least-current to most-current generation order until the group is at its limit.

A generation file is considered expired if the difference between the current system date and the file creation date exceeds the number of days specified by the `days` property of the group.

Generation files that are removed from a group are also deleted from the file system if the `scratch` option is in effect for the group.

For groups of Db2 and SFS files, the `-D` *`days`* option is not supported, and the `days` property of the group is forced to zero. Thus the age of these files is not a factor during limit processing.

"Example: limit processing" on page 130

Related concepts
"Generation data groups" on page 123

Related tasks
"Creating generation data groups" on page 124
"Using generation data groups" on page 126

Related references
"File specification" on page 469

Example: limit processing

The following example shows how limit processing affects the content of a generation data group.

Consider a generation data group, `audit`, that has a limit of three generations, a `days` option of 5, and the `noempty` option, and that was created with the following four new member generation files:

```
0: audit.g0001v00
0: audit.g0003v00
0: audit.g0005v00
0: audit.g0007v00
```

Because none of the files was beyond its expiration date when the group was created, the group is allowed to be over the limit of three generations. But after seven days, all the existing generations are expired. Therefore if a new generation is then added, the two least-current expired generations are removed to comply with the limit after the addition.

Typically the addition is done by running a program, but the following example shows another way of adding a generation, and shows the resulting group content:

```
gdgmgr -a "audit(+2)" -1
```

```
0: audit.g0005v00
0: audit.g0007v00
0: audit.g0009v00
```

If the original group had the `empty` option instead, the group content after the addition contains only one generation file, as follows:

```
0: audit.g0009v00
```

Concatenating files

In COBOL for Linux, you can concatenate multiple files by separating the individual file identifiers with a colon (:).

For example, the following `export` command sets the MYFILE environment variable to STL-/home/*myUserID*/file1 followed by STL-/home/*myUserID*/file2:

```
export MYFILE='STL-/home/myUserID/file1:STL-/home/myUserID/file2'
```

The `export` command together with a SELECT and ASSIGN clause that associate the environment variable with a COBOL internal file-name causes the two files to be treated as a single file in the COBOL program:

```
SELECT concatfile ASSIGN TO MYFILE
```

Concatenation is supported if the *assignment-name* for the concatenation is an environment variable (as shown above), literal, or USING data-name.

A COBOL internal file assigned to a concatenation of file identifiers must meet the following criteria:

- ORGANIZATION is SEQUENTIAL or LINE SEQUENTIAL.
- ACCESS MODE is SEQUENTIAL.
- OPEN statements for the file have mode INPUT.

You can concatenate files that are in any of the file systems. You can specify the file-system ID for any or all of the file identifiers in a given concatenation. However, all file identifiers in a concatenation must specify or default at run time to the same file system, and all the files must have consistent attributes.

GDGs: You can concatenate entire generation data groups (GDGs) or individual generation files from one or more groups. If you specify a GDG in a concatenation, the most current generation file is read first, then the next most current, and so on until the least current generation file in the group is reached. It is an error to include a newly defined, and thus empty, GDG in a concatenation.

Validation of the individual file identifiers in a concatenation is deferred until the corresponding COBOL file is opened. At that time, the first identifier in the concatenation is resolved, and the open is attempted.

After a successful OPEN, the first file in the concatenation can be read until its last record has been reached. At the next READ statement, the next file identifier in the concatenation is resolved and opened.

If all the files in a concatenation are unavailable when an OPEN statement is executed for an optional COBOL file, the OPEN is successful and the file status key is set to 05. The first READ operation returns end-of-file, and the AT END condition exists.

Related concepts
“Generation data groups” on page 123

Related tasks
“Identifying files” on page 112
“Using the end-of-file condition (AT END)” on page 166

Related references

ASSIGN clause (*COBOL for Linux on x86 Language Reference*)
File concatenation (*COBOL for Linux on x86 Language Reference*)

Opening optional files

If a program tries to open and read a file that does not exist, normally an error occurs.

However, there might be times when opening a nonexistent file makes sense. For such cases, code the keyword OPTIONAL in the SELECT clause:

```
SELECT OPTIONAL file ASSIGN TO filename
```

Whether a file is available or optional affects OPEN processing, file creation, and the resulting file status key. For example, if you open in EXTEND, I-O, or INPUT mode a nonexistent non-OPTIONAL file, the result is an OPEN error, and file status 35 is returned. If the file is OPTIONAL, however, the same OPEN statement returns file status 05, and, for open modes EXTEND and I-O, creates the file.

Related tasks
“Handling errors in input and output operations” on page 164
“Using file status keys” on page 166

Related references
File status key (*COBOL for Linux on x86 Language Reference*)
OPEN statement notes (*COBOL for Linux on x86 Language Reference*)

Setting up a field for file status

Establish a file status key by using the FILE STATUS clause in the FILE-CONTROL paragraph and data definitions in the DATA DIVISION.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    . . .
    FILE STATUS IS file-status
WORKING-STORAGE SECTION.
01  file-status  PIC 99.
```

Specify the file status key *file-status* as a two-character category alphanumeric or category national item, or as a two-digit zoned decimal (USAGE DISPLAY) (as shown above) or national decimal (USAGE NATIONAL) item.

Restriction: The data item referenced in the FILE STATUS clause cannot be variably located; for example, it cannot follow a variable-length table.

Related tasks
“Using file status keys” on page 166

Related references
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)

Describing the structure of a file in detail

In the FILE SECTION of the DATA DIVISION, start a file description by using the keyword FD and the same file-name you used in the corresponding SELECT clause in the FILE-CONTROL paragraph.

```
DATA DIVISION.
FILE SECTION.
FD  filename
01  recordname
    nn . . . fieldlength & type
    nn . . . fieldlength & type
    . . .
```

In the example above, *filename* is also the file-name you use in the OPEN, READ, and CLOSE statements.

recordname is the name of the record used in WRITE and REWRITE statements. You can specify more than one record for a file.

fieldlength is the logical length of a field, and *type* specifies the format of a field. If you break the record description entry beyond level 01 in this manner, map each element accurately to the corresponding field in the record.

Related references
Data relationships (*COBOL for Linux on x86 Language Reference*)
Level-numbers (*COBOL for Linux on x86 Language Reference*)
PICTURE clause (*COBOL for Linux on x86 Language Reference*)
USAGE clause (*COBOL for Linux on x86 Language Reference*)

Coding input and output statements for files

After you identify and describe the files in the ENVIRONMENT DIVISION and the DATA DIVISION, you can process the file records in the PROCEDURE DIVISION of your program.

Code your COBOL program according to the types of files that you use, whether sequential, line sequential, indexed, or relative. The general format for coding input and output (as shown in the example referenced below) involves opening the file, reading it, writing information into it, and then closing it.

"Example: COBOL coding for files" on page 133

Related tasks
"Identifying files" on page 112
"Specifying a file organization and access mode" on page 120
"Opening optional files" on page 132
"Setting up a field for file status" on page 132
"Describing the structure of a file in detail" on page 132
"Opening a file" on page 135
"Reading records from a file" on page 137
"Adding records to a file" on page 139
"Replacing records in a file" on page 140
"Deleting records from a file" on page 140

Related references
"File position indicator" on page 135
"Statements used when writing records to a file" on page 138
"PROCEDURE DIVISION statements used to update files" on page 140

Example: COBOL coding for files

The following example shows the general format of input/output coding. Explanations of user-supplied information (lowercase text in the example) are shown after the code.

```
IDENTIFICATION DIVISION.
. . .
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT filename ASSIGN TO assignment-name  (1) (2)
    ORGANIZATION IS org ACCESS MODE IS access  (3) (4)
    FILE STATUS IS file-status  (5)
    . . .
DATA DIVISION.
FILE SECTION.
FD  filename
01  recordname  (6)
    nn . . . fieldlength & type  (7) (8)
    nn . . . fieldlength & type
    . . .
```

```
WORKING-STORAGE SECTION.
01  file-status    PIC 99.
    . . .
PROCEDURE DIVISION.
    OPEN iomode filename   (9)
    . . .
    READ filename
    . . .
    WRITE recordname
    . . .
    CLOSE filename
  STOP RUN.
```

(1) *filename*

Any valid COBOL name. You must use the same file-name in the SELECT clause and FD entry, and in the OPEN, READ, START, DELETE, and CLOSE statements. This name is not necessarily the system file-name. Each file requires its own SELECT clause, FD entry, and input/output statements. For WRITE and REWRITE, you specify a record defined for the file.

(2) *assignment-name*

You can code ASSIGN TO *assignment-name* to specify the target file-system ID and system file-name directly, or you can set the value indirectly by using an environment variable.

If you want to have the system file-name identified at OPEN time, specify ASSIGN USING *data-name*. The value of *data-name* at the time of the execution of the OPEN statement for that file is used. You can optionally precede the system file-name by the file-system identifier, using a hyphen as the separator.

The following example shows how `inventory-file` is dynamically associated with the file `/user/inventory/parts` by means of a MOVE statement:

```
SELECT inventory-file ASSIGN USING a-file . . .
. . .
FD inventory-file . . .
. . .
77 a-file PIC X(25) VALUE SPACES.
. . .
    MOVE "/user/inventory/parts" TO a-file
    OPEN INPUT inventory-file
```

The following example shows how `inventory-file` is dynamically associated with the indexed CICS SFS file `parts`, and shows how the alternate index files `altpart1` and `altpart2` are associated with the fully qualified name (`/.:/cics/sfs` in this example) of the CICS server.

```
SELECT inventory-file ASSIGN USING a-file . . .
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS FILESYSFILE-KEY
    ALTERNATE RECORD KEY IS ALTKEY1
    ALTERNATE RECORD KEY IS ALTKEY2. . . .
. . .
FILE SECTION.
FD inventory-file . . .
. . .
WORKING-STORAGE SECTION.
01 a-file  PIC X(80). . .
. . .
    MOVE "/.:/cics/sfs/parts(/.:/cics/sfs/parts;altpart1,/.:/
       cics/sfs/parts;altpart2)" TO a-file
    OPEN INPUT inventory-file
```

(3) *org*

Indicates the organization: LINE SEQUENTIAL, SEQUENTIAL, INDEXED, or RELATIVE. If you omit this clause, the default is ORGANIZATION SEQUENTIAL.

(4) *access*

Indicates the access mode, SEQUENTIAL, RANDOM, or DYNAMIC. If you omit this clause, the default is ACCESS SEQUENTIAL.

(5) ***file-status***
The COBOL file status key. You can specify the file status key as a two-character alphanumeric or national data item, or as a two-digit zoned decimal or national decimal item.

(6) ***recordname***
The name of the record used in the WRITE and REWRITE statements. You can specify more than one record for a file.

(7) ***fieldlength***
The logical length of the field.

(8) ***type***
Must match the record format of the file. If you break the record description entry beyond the level-01 description, map each element accurately against the record's fields.

(9) ***iomode***
Specifies the open mode. If you are only reading from a file, code INPUT. If you are only writing to a file, code OUTPUT (to open a new file or write over an existing one) or EXTEND (to add records to the end of the file). If you are doing both, code I-O.

Restriction: For line-sequential files, I-O is not a valid mode of the OPEN statement.

File position indicator

The file position indicator marks the next record to be accessed for sequential COBOL requests.

You do not explicitly set the file position indicator anywhere in your program. It is set by successful OPEN, START, READ, READ NEXT, and READ PREVIOUS statements. Subsequent READ, READ NEXT, or READ PREVIOUS requests use the established file position indicator location and update it.

The file position indicator is not used or affected by the output statements WRITE, REWRITE, or DELETE. The file position indicator has no meaning for random processing.

Opening a file

Before your program can use a WRITE, START, READ, REWRITE, or DELETE statement to process records in a file, the program must first open the file using an OPEN statement.

```
PROCEDURE DIVISION.
    . . .
    OPEN iomode filename
```

In the example above, *iomode* specifies the open mode. If you are only reading from the file, code INPUT for the open mode. If you are only writing to the file, code either OUTPUT (to open a new file or write over an existing one) or EXTEND (to add records to the end of the file) for the open mode.

To open a file that already contains records, use OPEN INPUT, OPEN I-O (not valid for line-sequential files), or OPEN EXTEND.

If you code OPEN OUTPUT for either an SdU or SFS file that contains records, the COBOL run time deletes the file and then creates the file with attributes provided by COBOL. If you do not want an SdU or SFS file to be deleted, open the file by coding OPEN I-O instead.

If you open a sequential, line-sequential, or relative file as EXTEND, the added records are placed after the last existing record in the file. If you open an indexed file as EXTEND, each record that you add must have a record key that is higher than the highest record in the file.

Related concepts
“File organization and access mode” on page 121

Related tasks
“Opening optional files” on page 132

Related references
“Valid COBOL statements for sequential files” on page 136

“Valid COBOL statements for line-sequential files” on page 136
“Valid COBOL statements for indexed and relative files” on page 137

OPEN statement (*COBOL for Linux on x86 Language Reference*)

Valid COBOL statements for sequential files

The following table shows the possible combinations of input-output statements for sequential files. 'X' indicates that the statement can be used with the open mode shown at the top of the column.

Table 10. ***Valid COBOL statements for sequential files***

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START				
	READ	X		X	
	REWRITE			X	
	DELETE				
	CLOSE	X	X	X	X

Related concepts
“Sequential file organization” on page 121
“Sequential access” on page 122

Valid COBOL statements for line-sequential files

The following table shows the possible combinations of input-output statements for line-sequential files. 'X' indicates that the statement can be used with the open mode shown at the top of the column.

Table 11. ***Valid COBOL statements for line-sequential files***

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X		X
	WRITE		X		X
	START				
	READ	X			
	REWRITE				
	DELETE				
	CLOSE	X	X		X

Related concepts
“Line-sequential file organization” on page 122
“Sequential access” on page 122

Valid COBOL statements for indexed and relative files

The following table shows the possible combinations of input-output statements for indexed and relative files. 'X' indicates that the statement can be used with the open mode shown at the top of the column.

Table 12. ***Valid COBOL statements for indexed and relative files***

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	X
Random	OPEN	X	X	X	
	WRITE		X	X	
	START				
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	
Dynamic	OPEN	X	X	X	
	WRITE		X	X	
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

Related concepts

“Indexed file organization” on page 122
“Relative file organization” on page 122
“Sequential access” on page 122
“Random access” on page 123
“Dynamic access” on page 123

Reading records from a file

Use the READ statement to retrieve records from a file. To read a record, you must have opened the file with `OPEN INPUT` or `OPEN I-O` (`OPEN I-O` is not valid for line-sequential files). Check the file status key after each READ.

You can retrieve records in sequential and line-sequential files only in the sequence in which they were written.

You can retrieve records in indexed and relative record files sequentially (according to the ascending order of the key for indexed files, or according to ascending relative record locations for relative files), randomly, or dynamically.

When using dynamic access, you can switch between reading records sequentially and reading a specific record directly. For sequential retrieval, code READ NEXT and READ PREVIOUS; and for random retrieval (by key), use READ.

To read sequentially beginning at a specific record, use a START statement to set the file position indicator to point to a particular record before the READ NEXT or the READ PREVIOUS statement. If you code START followed by READ NEXT, the next record is read, and the file position indicator is reset to the next record. If you code START followed by READ PREVIOUS, the previous record is read, and the file position indicator is reset to the previous record. You can move the file position indicator randomly by using START, but all reading is done sequentially from that point.

You can continue to read records sequentially, or you can use START to move the file position indicator. For example:

```
START file-name KEY IS EQUAL TO ALTERNATE-RECORD-KEY
```

If a direct READ is performed for an indexed file based on an alternate index for which duplicates exist, only the first record in the file (base cluster) with that alternate key value is retrieved. You need a series of READ NEXT statements to retrieve each of the records that have the same alternate key. A file status value of 02 is returned if there are more records with the same alternate key value still to be read. A value of 00 is returned when the last record with that key value has been read.

Related concepts
“Sequential access” on page 122
“Random access” on page 123
“Dynamic access” on page 123
“File organization and access mode” on page 121

Related tasks
“Opening a file” on page 135
“Using file status keys” on page 166

Related references
“File position indicator” on page 135
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)

Statements used when writing records to a file

The following table shows the COBOL statements that you can use when creating or extending a file.

Table 13. ***Statements used when writing records to a file***

Division	Sequential	Line sequential	Indexed	Relative
ENVIRONMENT	SELECT			
	ASSIGN			
	ORGANIZATION IS SEQUENTIAL	ORGANIZATION IS LINE SEQUENTIAL	ORGANIZATION IS INDEXED	ORGANIZATION IS RELATIVE
	n/a		RECORD KEY	RELATIVE KEY
			ALTERNATE RECORD KEY	
	FILE STATUS			
	ACCESS MODE			

Table 13. **Statements used when writing records to a file** *(continued)*				
Division	**Sequential**	**Line sequential**	**Indexed**	**Relative**
DATA	FD entry			
PROCEDURE	OPEN OUTPUT			
	OPEN EXTEND			
	WRITE			
	CLOSE			

Related concepts
“File organization and access mode” on page 121

Related tasks
“Specifying a file organization and access mode” on page 120
“Opening a file” on page 135
“Setting up a field for file status” on page 132
“Adding records to a file” on page 139

Related references
“PROCEDURE DIVISION statements used to update files” on page 140

Adding records to a file

The COBOL WRITE statement adds a record to a file without replacing any existing records. The record to be added must not be larger than the maximum record size set when the file was defined. Check the file status key after each WRITE statement.

Adding records sequentially: To add records sequentially to the end of a file that has been opened with either OUTPUT or EXTEND, use ACCESS IS SEQUENTIAL and code the WRITE statement.

Sequential and line-sequential files are always written sequentially.

For indexed files, you must add new records in ascending key sequence. If a file is opened EXTEND, the record keys of the records to be added must be higher than the highest primary record key that was in the file when it was opened.

For relative files, the records must be in sequence. If you code a RELATIVE KEY data item in the SELECT clause, the relative record number of the record to be written is placed in that data item.

Adding records randomly or dynamically: When you write records to an indexed file for which you coded ACCESS IS RANDOM or ACCESS IS DYNAMIC, you can write the records in any order.

Related concepts
“File organization and access mode” on page 121

Related tasks
“Specifying a file organization and access mode” on page 120
“Using file status keys” on page 166

Related references
“Statements used when writing records to a file” on page 138
“PROCEDURE DIVISION statements used to update files” on page 140

FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)

Replacing records in a file

To replace a record in a file, use REWRITE if you opened the file for I-O. If the file was opened other than for I-O, the record is not replaced, and the status key is set to 49.

Check the file status key after each REWRITE statement.

For sequential files, the length of the replacement record must be the same as the length of the original record. For indexed files or variable-length relative files, you can change the length of the record you replace.

To replace a record randomly or dynamically, you do not have to first READ the record. Instead, locate the record you want to replace as follows:

- For indexed files, move the record key to the RECORD KEY data item, and then use REWRITE.
- For relative files, move the relative record number to the RELATIVE KEY data item, and then use REWRITE.

Related concepts
"File organization and access mode" on page 121

Related tasks
"Opening a file" on page 135
"Using file status keys" on page 166

Related references
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)

Deleting records from a file

To remove an existing record from an indexed or relative file, open the file as I-O and use the DELETE statement. You cannot use DELETE for a sequential or line-sequential file.

If ACCESS IS SEQUENTIAL, the record to be deleted must first be read by the COBOL program. The DELETE statement removes the record that was just read. If the DELETE statement is not preceded by a successful READ, the record is not deleted, and the file status key is set to 92.

If ACCESS IS RANDOM or ACCESS IS DYNAMIC, the record to be deleted need not be read by the COBOL program. To delete a record, move the key of the record to the RECORD KEY data item, and then issue the DELETE.

Check the file status key after each DELETE statement.

Related concepts
"File organization and access mode" on page 121

Related tasks
"Opening a file" on page 135
"Reading records from a file" on page 137
"Using file status keys" on page 166

Related references
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)

PROCEDURE DIVISION statements used to update files

The table below shows the statements that you can use in the PROCEDURE DIVISION for sequential, line-sequential, indexed, and relative files.

Table 14. ***PROCEDURE DIVISION statements used to update files***

Access method	Sequential	Line sequential	Indexed	Relative
ACCESS IS SEQUENTIAL	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE CLOSE	OPEN EXTEND WRITE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE
ACCESS IS RANDOM	Not applicable	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE
ACCESS IS DYNAMIC (sequential processing)	Not applicable	Not applicable	OPEN I-O READ NEXT READ PREVIOUS START CLOSE	OPEN I-O READ NEXT READ PREVIOUS START CLOSE
ACCESS IS DYNAMIC (random processing)	Not applicable	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE

Related concepts
"File organization and access mode" on page 121

Related tasks
"Opening a file" on page 135
"Reading records from a file" on page 137
"Adding records to a file" on page 139
"Replacing records in a file" on page 140
"Deleting records from a file" on page 140

Related references
"Statements used when writing records to a file" on page 138

Using Db2 files

To access Db2 files from a COBOL application that runs under Linux, you must follow guidelines for compiling and linking the application and for identifying the Db2 file system, the Db2 instance and database, and the Db2 files.

1. Compile and link the COBOL programs in your application by using the `cob2` command.
2. Initialize the Db2 environment by executing the profile for the Db2 instance that you want to use.

   For example, you might issue the following command to use instance `db2inst1`:

   ```
   . /home/db2inst1/sqllib/db2profile
   ```

3. Ensure that the Db2 instance is running and that you can connect to the database that you want to access.

   The following example shows the db2 command that you might use to connect to database `db2cob`, and shows the system response:

   ```
   > db2 connect to db2cob
           Database Connection Information
           Database server        = DB2/Linux64 9.7.0
           SQL authorization ID   = MYUID
           Local database alias   = DB2COB
   ```

   Restriction: Note that COBOL applications can access Db2 files in only one database and Db2 instance at a time.
4. Set environment variable DB2DBDFT to the intended database. For example:

   ```
   export DB2DBDFT=db2cob
   ```

5. For the files that use Db2, specify the Db2 file system (either as the value of the `FILESYS` runtime option or directly in the assignment-name value). Use the fully qualified Db2 table name, including the schema name.

   For example, the following command completes the assignment of a transaction file `TRANFILE` to system file-name `TEST.TRANS` in the Db2 file system:

   ```
   export TRANFILE=DB2-TEST.TRANS
   ```

Creating Db2 files:

You can create a Db2 file in any of several ways:

- By using an OPEN statement in your COBOL program
- By using the CICS TXSeries or CICS TX `cicsddt` utility

  For more information about the `cicsddt` command, see the CICS TXSeries or CICS TX documentation referenced below.
- By using the `db2 create` command

For example, you might use the following command sequence to create a relative file called `EXAMPLE` under schema `CICS`:

```
db2 create table cics.example\
"(rba char(4) not null for bit data, f1 varchar(80) not null for bit data)"
db2 create unique index cics.example0 on cics.example\
"(rba) disallow reverse scans"
db2 create unique index cics.example0@ on cics.example\
"(rba desc) disallow reverse scans"
```

You can display the resulting table by issuing the db2 describe command. For example:

```
> db2 describe table cics.example

                        Data type                      Column
Column name             schema    Data type name       Length     Scale Nulls
----------------------- --------- ------------------- ---------- ----- ------
RBA                     SYSIBM    CHARACTER                    4     0 No
F1                      SYSIBM    VARCHAR                     80     0 No

  2 record(s) selected.
```

The file CICS.EXAMPLE has variable-length records that would be compatible with a COBOL FILE SECTION definition of minimum record length between 0 and 79.

For more information about the functions that are provided by the db2 utility, enter the command db2.

For information about the additional requirements that apply to using Db2 files with TXSeries or CICS TX Knowledge Center, see the related reference about the Db2 file system.

Related tasks
"Identifying files" on page 112
"Identifying Db2 files" on page 114
"Using Db2 files and SQL statements in the same program" on page 143
"Compiling from the command line" on page 223
Chapter 17, "Programming for a Db2 environment," on page 325

Related references
"Db2 file system" on page 117
"Compiler and runtime environment variables" on page 214
"FILESYS" on page 298
TXSeries for Multiplatforms documentation
IBM CICS TX on Cloud documentation

Using Db2 files and SQL statements in the same program

To use EXEC SQL statements and Db2 file I/O in the same program, there are some important facts that you must know, as explained below.

- Both facilities (EXEC SQL statements and Db2 file I/O) use the *same* Db2 connection.
- Each COBOL I/O update operation that uses the Db2 file system is committed to the database immediately.
- If an existing Db2 connection is available, the Db2 file system uses that connection.

  If a connection is not available, the Db2 file system establishes a connection to the database that is referenced by the value of the DB2DBDFT environment variable.

EXEC SQL statements and Db2 file I/O can use the same database, or, if you explicitly control the connection, different databases.

Using the same database:

Using the same database for EXEC SQL statements and Db2 file I/O in the same program is simpler than using different databases. But you must handle this configuration carefully nonetheless:

- To avoid anomalies, do not rely on the Db2 file system's use of an existing connection. To ensure consistent results regardless of which type of database access (Db2 file I/O or EXEC SQL) occurs first, set environment variable DB2DBDFT to the same database that the EXEC SQL statements use.
- Db2 file update operations commit all pending work for the database. Therefore roll back or commit any pending EXEC SQL updates before initiating any Db2 file I/O operations.

Although opening a Db2 file for input and reading the file does not cause a database commit, it is recommended that you not rely on this behavior.

Using different databases:

To use different databases for EXEC SQL statements and for Db2 file I/O in the same program, you must explicitly control the database connections as shown in the examples below.

Suppose that you want to use EXEC SQL statements with database db2pli, and do Db2 file I/O using database db2cob by setting environment variable DB2DBDFT:

```
export DB2DBDFT=db2cob
```

In the example below, doing the sequence of steps shown (with angle brackets indicating pseudocode) will not use the intended databases correctly, as the inline comments explain:

```
<DB2 file I/O>                        *> Uses database db2cob (from DB2DBDFT)
EXEC SQL CONNECT TO db2pli END-EXEC   *> Switches to database db2pli
<Other EXEC SQL operations>           *> Use database db2pli
<DB2 file I/O>                        *> Uses the existing connection--and
. . .                                 *>   thus database db2pli--incorrectly!
```

To access the intended databases, first disconnect from the database used by the EXEC SQL statements before doing any Db2 file I/O operations. Then either rely on the value in environment variable DB2DBDFT or explicitly connect to the database that you want to use for Db2 file I/O.

The following sequence of steps illustrates reliance on DB2DBDFT to correctly make the intended connections:

```
<DB2 file I/O>                        *> Uses database db2cob (from DB2DBDFT)
EXEC SQL CONNECT TO db2pli END-EXEC   *> Switches to database db2pli
<Other EXEC SQL operations>           *> Use database db2pli
* Commit or roll back pending operations
* here, because the following statement
* unconditionally commits pending work:
EXEC SQL CONNECT RESET END-EXEC       *> Disconnect from database db2pli
<DB2 file I/O>                        *> Uses database db2cob (from DB2DBDFT)
. . .
```

Related tasks
"Coding SQL statements" on page 326

Related references
"Compiler and runtime environment variables" on page 214

Using QSAM files

QSAM (queued sequential access method) files are unkeyed files in which the records are placed one after another, according to entry order.

Your program can process these files only sequentially, retrieving records using the READ statement in the same order as they are in the file. Each record is placed after the preceding record. To process QSAM files in your program, use COBOL language statements that:

- Identify and describe the QSAM files in the ENVIRONMENT DIVISION and the DATA DIVISION.
- Process the records in these files in the PROCEDURE DIVISION.

After you create a record, you cannot change its length or its position in the file, and you cannot delete it.

Related tasks
"Identifying files" on page 112

Related references
“QSAM file system” on page 118
“FILESYS” on page 298

Using SFS files

To access CICS SFS files from a COBOL application that runs under Linux, you must follow guidelines for compiling and linking, and for identifying the file system, the CICS SFS server, and the SFS files.

1. Compile and link the COBOL programs in the application by using the `cob2` command.
2. Ensure that the CICS SFS server that your application will access is running.
3. (Optional) If your application creates one or more SFS files, and you want to allocate the files on an SFS data volume that has a name other than sfs_SSFS_SERVER, you can specify either or both of the following names:
   - The name of the SFS data volume on which the SFS files are to be created. To do so, assign a value to the runtime environment variable CICS_SFS_INDEX_VOLUME. The data volume must have been defined to the SFS server. If you do not know which data volumes are available to the SFS server, issue the command `sfsadmin list lvols`.

     By default, SFS files are created on the data volume that is named sfs_SSFS_SERVER.
   - The name of the SFS data volume on which alternate index files, if any, are to be created. To do so, assign a value to the runtime environment variable CICS_SFS_INDEX_VOLUME. The data volume must have been defined to the SFS server.

     By default, alternate index files are created on the same volume as the corresponding base files.
4. Identify each SFS file:
   - Either set the default file system to SFS by setting the runtime option `FILESYS` as follows:

     ```
     export COBRTOPT=FILESYS=SFS
     ```

     Or alternatively, in an `export` command for each SFS file, precede the file name and SFS server name with the file-system ID SFS followed by a hyphen (`-`), as shown below.
   - The CICS SFS server name must precede the file name.
   - Any alternate index file names must start with the base file name, followed by a semicolon (`;`) and the alternate index name.

   For example, if /.:/cics/sfs/*sfsServer* is the CICS SFS server, and SFS04A is an SFS file that has alternate index SFS04A1, you could identify SFS04A by issuing the following `export` command:

   ```
   export SFS04AEV="SFS-/.:/cics/sfs/sfsServer/SFS04A(/.:/cics/sfs/sfsServer/SFS04A;SFS04A1)"
   ```

For more information about fully qualified names for SFS servers and files, see the related task about identifying CICS SFS files.

“Example: accessing SFS files” on page 146

Related tasks
“Identifying files” on page 112
“Identifying SFS files” on page 115
“Improving SFS performance” on page 147

Related references
“SFS file system” on page 119
“Runtime environment variables” on page 218
“FILESYS” on page 298

Example: accessing SFS files

The following example shows COBOL file descriptions that you might code and `sfsadmin` and `export` commands that you might issue to create and access two SFS files.

SFS04 is an indexed file that has no alternate index. SFS04A is an indexed file that has one alternate index, SFS04A1.

COBOL file descriptions

```
. . .
Environment division.
Input-output section.
File-control.
    select SFS04-file
        assign to SFS04EV
        access dynamic
        organization is indexed
        record key is SFS04-rec-num
        file status is SFS04-status.

    select SFS04A-file
        assign to SFS04AEV
        access dynamic
        organization is indexed
        record key is SFS04A-rec-num
        alternate record key is SFS04A-date-time
        file status is SFS04A-status.

Data division.
File section.
FD  SFS04-file.
  01 SFS04-record.
     05 SFS04-rec-num              pic x(10).
     05 SFS04-rec-data             pic x(70).
FD  SFS04A-file.
  01 SFS04A-record.
     05 SFS04A-rec-num             pic x(10).
     05 SFS04A-date-time.
        07 SFS04A-date-yyyymmdd    pic 9(8).
        07 SFS04A-time-hhmmsshh    pic 9(8).
        07 SFS04A-date-time-counter pic 9(8).
     05 SFS04A-rec-data            pic x(1000).
```

sfsadmin commands

Create each indexed file by issuing the `sfsadmin create clusteredfile` command, and add an alternate index by issuing the `sfsadmin add index` command:

```
sfsadmin create clusteredfile SFS04 2 \
PrimaryKey byteArray 10 \
DATA byteArray 70 \
primaryIndex -unique 1 PrimaryKey sfsVolume
#
sfsadmin create clusteredfile SFS04A 3 \
PrimaryKey byteArray 10 \
AltKey1 byteArray 24 \
DATA byteArray 1000 \
primaryIndex -unique 1 PrimaryKey sfsVolume
#
sfsadmin add index SFS04A SFS04A1 1 AltKey1
```

As shown in the first `sfsadmin create clusteredfile` command above, you must specify the following items:

- The name of the new indexed file (SFS04 in this example)
- The number of fields per record (2)

- The description of each field (`PrimaryKey` and `DATA`, each a byte array)
- The name of the primary index (`primaryIndex`)
- The `-unique` option
- The number of fields in the primary index (1)
- The names of the fields in the primary index (`PrimaryKey`)
- The name of the data volume on which the file is to be stored (*sfsVolume*)

By default, CICS SFS allows duplicate keys in the primary index of a clustered file. However, you must specify the `-unique` option as shown above because COBOL requires that key values in the primary index be unique within the file.

As shown in the `sfsadmin add index` command above, you must specify the following items:

- The name of the file to which an alternate index is to be added (SFS04A in this example)
- The name of the new index (SFS04A1)
- The number of fields to be used as keys in the new index (1)
- The names of the fields in the new index (`AltKey1`)

For details about the syntax of the commands `sfsadmin create clusteredfile` and `sfsadmin add index`, see the Related references.

export commands

Before you run the program that processes the SFS files, issue these `export` commands to specify the path (`/.:/cics/sfs`) to the CICS SFS server (*sfsServer*) that will access the files, and the data volume (*sfsVolume*) that will store the files:

```
# Set environment variables required by the SFS file system
# for SFS files:

export CICS_SFS_DATA_VOLUME=sfsVolume
export CICS_SFS_INDEX_VOLUME=sfsVolume

# Set SFS as the default file system:
export COBRTOPT=FILESYS=SFS

# Enable use of a short-form SFS specification:
export CICS_TK_SFS_SERVER=/.:/cics/sfs/sfsServer

# Set the environment variable to access SFS file SFS04
# (an example of using a short-form SFS specification):
export SFS04EV=SFS04

# Set the environment variable to access SFS
# file SFS04A and the alternate index SFS04A1:
```

```
export SFS04AEV="/.:/cics/sfs/sfsServer/SFS04A(/.:/cics/sfs/sfsServer/
SFS04A;SFS04A1)"
```

Related references
CICS TXSeries documentation

Improving SFS performance

You can improve the performance of applications that access SFS files in two ways: by using client-side caching on the client machine, and by reducing the frequency of saving changes to SFS files.

Related tasks
“Identifying SFS files” on page 115
“Enabling client-side caching” on page 148

“Reducing the frequency of saving changes” on page 149
Improving performance of the SFS in the CICS TXSeries documentation

Related references
“SFS file system” on page 119

Enabling client-side caching

By default, records in SFS files are written to and read from the SFS server, and a remote procedure call (RPC) is needed whenever a record is accessed. If you enable *client-side caching*, however, you can improve performance because less time is needed to access records.

With client-side caching, records are stored in local memory (a *cache*) on the client machine and sent (*flushed*) to the server in a single RPC. You can specify either or both of two types of caching: *read caching* and *insert caching*:

- If you enable read caching, the first sequential read of a file causes the current record and a number of neighboring records to be read from the server and placed in the read cache. Subsequent sequential reads, updates, and deletes are made in the read cache rather than on the server.
- If you enable insert caching, insert operations (but not reads, updates, or deletes) are done in the insert cache rather than on the server.

To enable client-side caching for all the SFS files in your application, set the CICS_VSAM_CACHE environment variable before you run the application. To see a syntax diagram that describes setting CICS_VSAM_CACHE, see the related reference about runtime environment variables.

You can code a single value for the cache size to indicate that the same number of pages is to be used for the read cache and for the insert cache, or you can code distinct values for the read and insert cache by separating the values by a colon (`:`). Express® size units as numbers of pages. If you code zero as the size of the read cache, insert cache, or both, that type of caching is disabled. For example, the following command sets the size of the read cache to 16 pages and the size of the insert cache to 64 pages for each SFS file in the application:

```
export CICS_VSAM_CACHE=16:64
```

You can also specify one or both of the following flags to make client-side caching more flexible:

- To allow uncommitted data (records that are new or modified but have not yet been sent to the server, known as *dirty* records) to be read, specify ALLOW_DIRTY_READS.

  This flag removes the restriction for read caching that the files being accessed must be locked.
- To allow any inserts to be cached, specify INSERTS_DESPITE_UNIQUE_INDICES.

  This flag removes the restriction for insert caching that all active indices for clustered files and active alternate indices for entry-sequenced and relative files must allow duplicates.

For example, the following command allows maximum flexibility:

```
export CICS_VSAM_CACHE=16:64:ALLOW_DIRTY_READS,INSERTS_DESPITE_UNIQUE_INDICES
```

To set client-side caching differently for certain files, code a putenv() call that sets CICS_VSAM_CACHE before the OPEN statement for each file for which you want to change the caching policy. During a program, the environment-variable settings that you make in a putenv() call take precedence over the environment-variable settings that you make in an `export` command.

“Example: setting and accessing environment variables” on page 221

Related tasks
“Setting environment variables” on page 213

Related references
“SFS file system” on page 119

“Runtime environment variables” on page 218

Reducing the frequency of saving changes

An RPC normally occurs for each write or update operation on an SFS file that does not use client-side caching (that is, the *operational force* feature of SFS is enabled). All file changes that result from input-output operations are committed to disk before control returns to the application.

You can change this behavior so that the result of input-output operations on SFS files might not be committed to disk until the files are closed (that is, a *lazy-write* strategy is used). If each change to an SFS file is not immediately saved, the application can run faster.

To change the default commit behavior for all the SFS files in your application, set the CICS_VSAM_AUTO_FLUSH environment variable to OFF before you run the application:

```
export CICS_VSAM_AUTO_FLUSH=OFF
```

To set the flush value differently for certain files, code a putenv() call that sets CICS_VSAM_AUTO_FLUSH before the OPEN statement for each file for which you want to change the flush value. During a program, the environment-variable settings that you make in a putenv() call take precedence over the environment-variable settings that you make in an `export` command.

“Example: setting and accessing environment variables” on page 221

If client-side caching is in effect for an SFS file (that is, environment variable CICS_VSAM_CACHE is set to a valid nonzero value), the setting of CICS_VSAM_AUTO_FLUSH for the file is ignored. Operational force is disabled for that file.

Related tasks
“Setting environment variables” on page 213

Related references
“SFS file system” on page 119
“Runtime environment variables” on page 218

Chapter 8. Sorting and merging files

You can arrange records in a particular sequence by using a SORT or MERGE statement. You can mix SORT and MERGE statements in the same COBOL program.

SORT statement
Accepts input (from a file or an internal procedure) that is not in sequence, and produces output (to a file or an internal procedure) in a requested sequence. You can add, delete, or change records before or after they are sorted.

MERGE statement
Compares records from two or more sequenced files and combines them in order. You can add, delete, or change records after they are merged.

A program can contain any number of sort and merge operations. They can be the same operation performed many times or different operations. However, one operation must finish before another begins.

The steps you take to sort or merge are generally as follows:

1. Describe the sort or merge file to be used for sorting or merging.
2. Describe the input to be sorted or merged. If you want to process the records before you sort them, code an input procedure.
3. Describe the output from sorting or merging. If you want to process the records after you sort or merge them, code an output procedure.
4. Request the sort or merge.
5. Determine whether the sort or merge operation was successful.

Related concepts
"Sort and merge process" on page 151

Related tasks
"Describing the sort or merge file" on page 152
"Describing the input to sorting or merging" on page 152
"Describing the output from sorting or merging" on page 154
"Requesting the sort or merge" on page 156
"Determining whether the sort or merge was successful" on page 158
"Stopping a sort or merge operation prematurely" on page 162

Related references
SORT statement (*COBOL for Linux on x86 Language Reference*)
MERGE statement (*COBOL for Linux on x86 Language Reference*)

Sort and merge process

During the sorting of a file, all of the records in the file are ordered according to the contents of one or more fields (*keys*) in each record. You can sort the records in either ascending or descending order of each key.

If there are multiple keys, the records are first sorted according to the content of the first (or primary) key, then according to the content of the second key, and so on.

To sort a file, use the COBOL SORT statement.

© Copyright IBM Corp. 2021

During the merging of two or more files (which must already be sorted), the records are combined and ordered according to the contents of one or more keys in each record. You can order the records in either ascending or descending order of each key. As with sorting, the records are first ordered according to the content of the primary key, then according to the content of the second key, and so on.

Use MERGE . . . USING to name the files that you want to combine into one sequenced file. The merge operation compares keys in the records of the input files, and passes the sequenced records one by one to the RETURN statement of an output procedure or to the file that you name in the GIVING phrase.

Related tasks
"Setting sort or merge criteria" on page 156

Related references
SORT statement (*COBOL for Linux on x86 Language Reference*)
MERGE statement (*COBOL for Linux on x86 Language Reference*)

Describing the sort or merge file

Describe the sort file to be used for sorting or merging. You need SELECT clauses and SD entries even if you are sorting or merging data items only from WORKING-STORAGE or LOCAL-STORAGE.

Code as follows:

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name a sort file. For example:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT Sort-Work-1 ASSIGN TO SortFile.
```

   Sort-Work-1 is the name of the file in your program. Use this name to refer to the file.
2. Describe the sort file in an SD entry in the FILE SECTION of the DATA DIVISION. Every SD entry must contain a record description. For example:

```
DATA DIVISION.
FILE SECTION.
SD  Sort-Work-1
    RECORD CONTAINS 100 CHARACTERS.
01  SORT-WORK-1-AREA.
    05  SORT-KEY-1   PIC  X(10).
    05  SORT-KEY-2   PIC  X(10).
    05  FILLER       PIC  X(80).
```

The file described in an SD entry is the working file used for a sort or merge operation. You cannot perform any input or output operations on this file.

Related references
"FILE SECTION entries" on page 10

Describing the input to sorting or merging

Describe the input file or files for sorting or merging by following the procedure below.

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the input files. For example:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT Input-File ASSIGN TO InFile.
```

Input-File is the name of the file in your program. Use this name to refer to the file.

2. Describe the input file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

```
DATA DIVISION.
FILE SECTION.
FD  Input-File
    RECORD CONTAINS 100 CHARACTERS.
01  Input-Record   PIC X(100).
```

Related tasks
"Coding the input procedure" on page 153
"Requesting the sort or merge" on page 156

Related references
"FILE SECTION entries" on page 10

Example: describing sort and input files for SORT

The following example shows the ENVIRONMENT DIVISION and DATA DIVISION entries needed to describe sort work files and an input file.

```
 ID Division.
 Program-ID. SmplSort.
 Environment Division.
 Input-Output Section.
 File-Control.
*
* Assign name for a working file is treated as documentation.
*
     Select Sort-Work-1 Assign To SortFile.
     Select Sort-Work-2 Assign To SortFile.
     Select Input-File  Assign To InFile.
 . . .
 Data Division.
 File Section.
 SD  Sort-Work-1
     Record Contains 100 Characters.
 01  Sort-Work-1-Area.
     05  Sort-Key-1    Pic  X(10).
     05  Sort-Key-2    Pic  X(10).
     05  Filler        Pic  X(80).
 SD  Sort-Work-2
     Record Contains 30 Characters.
 01  Sort-Work-2-Area.
     05  Sort-Key      Pic  X(5).
     05  Filler        Pic  X(25).
 FD  Input-File
     Record Contains 100 Characters.
 01  Input-Record      Pic  X(100).
 . . .
 Working-Storage Section.
 01  EOS-Sw            Pic  X.
 01  Filler.
     05  Table-Entry Occurs 100 Times
             Indexed By X1    Pic X(30).
     . . .
```

Related tasks
"Requesting the sort or merge" on page 156

Coding the input procedure

To process the records in an input file before they are released to the sort program, use the INPUT PROCEDURE phrase of the SORT statement.

You can use an input procedure to:

- Release data items to the sort file from WORKING-STORAGE or LOCAL-STORAGE.

- Release records that have already been read elsewhere in the program.
- Read records from an input file, select or process them, and release them to the sort file.

Each input procedure must be contained in either paragraphs or sections. For example, to release records from a table in WORKING-STORAGE or LOCAL-STORAGE to the sort file SORT-WORK-2, you could code as follows:

```
    SORT SORT-WORK-2
      ON ASCENDING KEY SORT-KEY
      INPUT PROCEDURE 600-SORT3-INPUT-PROC
    . . .
600-SORT3-INPUT-PROC SECTION.
    PERFORM WITH TEST AFTER
      VARYING X1 FROM 1 BY 1 UNTIL X1 = 100
      RELEASE SORT-WORK-2-AREA FROM TABLE-ENTRY (X1)
    END-PERFORM.
```

To transfer records to the sort program, all input procedures must contain at least one RELEASE or RELEASE FROM statement. To release A from X, for example, you can code:

```
MOVE X TO A.
RELEASE A.
```

Alternatively, you can code:

```
RELEASE A FROM X.
```

The following table compares the RELEASE and RELEASE FROM statements.

RELEASE	RELEASE FROM

`MOVE EXT-RECORD`
`  TO SORT-EXT-RECORD`
`PERFORM RELEASE-SORT-RECORD`
`. . .`
`RELEASE-SORT-RECORD.`
`  RELEASE SORT-RECORD`
	`PERFORM RELEASE-SORT-RECORD`
	`. . .`
	`RELEASE-SORT-RECORD.`
	`  RELEASE SORT-RECORD`
	`  FROM SORT-EXT-RECORD`

Related references
"Restrictions on input and output procedures" on page 155
RELEASE statement (*COBOL for Linux on x86 Language Reference*)

Describing the output from sorting or merging

If the output from sorting or merging is a file, describe the file by following the procedure below.

1. Write a SELECT clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the output file. For example:

   ```
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
       SELECT Output-File ASSIGN TO OutFile.
   ```

   Output-File is the name of the file in your program. Use this name to refer to the file.
2. Describe the output file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

```
DATA DIVISION.
FILE SECTION.
FD  Output-File
    RECORD CONTAINS 100 CHARACTERS.
01  Output-Record   PIC X(100).
```

Related tasks
“Coding the output procedure” on page 155
“Requesting the sort or merge” on page 156

Related references
“FILE SECTION entries” on page 10

Coding the output procedure

To select, edit, or otherwise change sorted records before writing them from the sort work file into another file, use the OUTPUT PROCEDURE phrase of the SORT statement.

Each output procedure must be contained in either a section or a paragraph. An output procedure must include both of the following elements:

- At least one RETURN statement or one RETURN statement with the INTO phrase
- Any statements necessary to process the records that are made available, one at a time, by the RETURN statement

The RETURN statement makes each sorted record available to the output procedure. (The RETURN statement for a sort file is similar to a READ statement for an input file.)

You can use the AT END and END-RETURN phrases with the RETURN statement. The imperative statements in the AT END phrase are performed after all the records have been returned from the sort file. The END-RETURN explicit scope terminator delimits the scope of the RETURN statement.

If you use RETURN INTO instead of RETURN, the records will be returned to WORKING-STORAGE, LOCAL-STORAGE, or to an output area.

Related references
“Restrictions on input
and output procedures” on page 155
RETURN statement (*COBOL for Linux on x86 Language Reference*)

Restrictions on input and output procedures

Several restrictions apply to each input or output procedure called by SORT and to each output procedure called by MERGE.

Observe these restrictions:

- The procedure must not contain any SORT or MERGE statements.
- You can use ALTER, GO TO, and PERFORM statements in the procedure to refer to procedure-names outside the input or output procedure. However, control must return to the input or output procedure after a GO TO or PERFORM statement.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the input or output procedure (with the exception of the return of control from a declarative section).
- In an input or output procedure, you can call a program. However, the called program cannot issue a SORT or MERGE statement, and the called program must return to the caller.
- During a SORT or MERGE operation, the SD data item is used. You must not use it in the output procedure before the first RETURN executes. If you move data into this record area before the first RETURN statement, the first record to be returned will be overwritten.

Related tasks
"Coding the input procedure" on page 153
"Coding the output procedure" on page 155

Requesting the sort or merge

To read records from an input file (files for MERGE) without preliminary processing, use SORT . . . USING or MERGE . . . USING and the name of the input file (files) that you declared in a SELECT clause.

To transfer sorted or merged records from the sort or merge program to another file without any further processing, use SORT . . . GIVING or MERGE . . . GIVING and the name of the output file that you declared in a SELECT clause. For example:

```
SORT Sort-Work-1
    ON ASCENDING KEY Sort-Key-1
    USING Input-File
    GIVING Output-File.
```

For SORT . . . USING or MERGE . . . USING, the compiler generates an input procedure to open the file (files), read the records, release the records to the sort or merge program, and close the file (files). The file (files) must not be open when the SORT or MERGE statement begins execution. For SORT . . . GIVING or MERGE . . . GIVING, the compiler generates an output procedure to open the file, return the records, write the records, and close the file. The file must not be open when the SORT or MERGE statement begins execution.

"Example: describing sort and input files for SORT" on page 153

If you want an input procedure to be performed on the sort records before they are sorted, use SORT . . . INPUT PROCEDURE. If you want an output procedure to be performed on the sorted records, use SORT . . . OUTPUT PROCEDURE. For example:

```
SORT Sort-Work-1
    ON ASCENDING KEY Sort-Key-1
    INPUT PROCEDURE EditInputRecords
    OUTPUT PROCEDURE FormatData.
```

"Example: sorting with input and output procedures" on page 157

Restriction: You cannot use an input procedure with the MERGE statement. The source of input to the merge operation must be a collection of already sorted files. However, if you want an output procedure to be performed on the merged records, use MERGE . . . OUTPUT PROCEDURE. For example:

```
MERGE Merge-Work
    ON ASCENDING KEY Merge-Key
    USING Input-File-1 Input-File-2 Input-File-3
    OUTPUT PROCEDURE ProcessOutput.
```

In the FILE SECTION, you must define *Merge-Work* in an SD entry, and the input files in FD entries.

Related references
SORT statement (*COBOL for Linux on x86 Language Reference*)
MERGE statement (*COBOL for Linux on x86 Language Reference*)

Setting sort or merge criteria

To set sort or merge criteria, define the keys on which the operation is to be performed.

Do these steps:

1. In the record description of the files to be sorted or merged, define the key or keys.

   Restriction: A key cannot be variably located.

2. In the SORT or MERGE statement, specify the key fields to be used for sequencing by coding the ASCENDING or DESCENDING KEY phrase, or both. When you code more than one key, some can be ascending, and some descending.

   Specify the names of the keys in decreasing order of significance. The leftmost key is the primary key. The next key is the secondary key, and so on.

SORT and MERGE keys can be of class alphabetic, alphanumeric, national (if the compiler option NCOLLSEQ(BIN) is in effect), or numeric (but not numeric of USAGE NATIONAL). If it has USAGE NATIONAL, a key can be of category national or can be a national-edited or numeric-edited data item. A key cannot be a national decimal data item or a national floating-point data item.

The collation order for national keys is determined by the binary order of the keys. If you specify a national data item as a key, any COLLATING SEQUENCE phrase in the SORT or MERGE statement does not apply to that key.

You can mix SORT and MERGE statements in the same COBOL program. A program can perform any number of sort or merge operations. However, one operation must end before another can begin.

Related tasks
"Controlling the collating sequence with a locale" on page 205

Related references
"NCOLLSEQ" on page 271
SORT statement (*COBOL for Linux on x86 Language Reference*)
MERGE statement (*COBOL for Linux on x86 Language Reference*)

Choosing alternate collating sequences

You can sort or merge records on a collating sequence that you specify for single-byte character keys. The default collating sequence is the collating sequence specified by the locale setting in effect at compile time unless you code the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph.

To override the default sequence, use the COLLATING SEQUENCE phrase of the SORT or MERGE statement. You can use different collating sequences for each SORT or MERGE statement in your program.

The PROGRAM COLLATING SEQUENCE clause and the COLLATING SEQUENCE phrase apply only to keys of class alphabetic or alphanumeric. The COLLATING SEQUENCE phrase is valid only when a single-byte ASCII code page is in effect.

Related tasks
"Specifying the collating sequence" on page 6
"Controlling the collating sequence with a locale" on page 205
"Setting sort or merge criteria" on page 156

Related references
OBJECT-COMPUTER paragraph (*COBOL for Linux on x86 Language Reference*)
SORT statement (*COBOL for Linux on x86 Language Reference*)
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)

Example: sorting with input and output procedures

The following example shows the use of an input and an output procedure in a SORT statement. The example also shows how you can define a primary key (SORT-GRID-LOCATION) and a secondary key (SORT-SHIFT) before using them in the SORT statement.

```
DATA DIVISION.
. . .
```

```
SD  SORT-FILE
    RECORD CONTAINS 115 CHARACTERS
    DATA RECORD SORT-RECORD.
01  SORT-RECORD.
    05  SORT-KEY.
        10  SORT-SHIFT              PIC X(1).
        10  SORT-GRID-LOCATION      PIC X(2).
        10  SORT-REPORT             PIC X(3).
    05  SORT-EXT-RECORD.
        10  SORT-EXT-EMPLOYEE-NUM   PIC X(6).
        10  SORT-EXT-NAME           PIC X(30).
        10  FILLER                  PIC X(73).
. . .
WORKING-STORAGE SECTION.
01  TAB1.
    05 TAB-ENTRY OCCURS 10 TIMES
           INDEXED BY TAB-INDX.
        10  WS-SHIFT                PIC X(1).
        10  WS-GRID-LOCATION        PIC X(2).
        10  WS-REPORT               PIC X(3).
        10  WS-EXT-EMPLOYEE-NUM     PIC X(6).
        10  WS-EXT-NAME             PIC X(30).
        10  FILLER                  PIC X(73).
. . .
PROCEDURE DIVISION.
    . . .
    SORT SORT-FILE
        ON ASCENDING KEY SORT-GRID-LOCATION SORT-SHIFT
        INPUT PROCEDURE 600-SORT3-INPUT
        OUTPUT PROCEDURE 700-SORT3-OUTPUT.
    . . .
600-SORT3-INPUT.
    PERFORM VARYING TAB-INDX FROM 1 BY 1 UNTIL TAB-INDX > 10
        RELEASE SORT-RECORD FROM TAB-ENTRY(TAB-INDX)
    END-PERFORM.
. . .
700-SORT3-OUTPUT.
    PERFORM VARYING TAB-INDX FROM 1 BY 1 UNTIL TAB-INDX > 10
        RETURN SORT-FILE INTO TAB-ENTRY(TAB-INDX)
            AT END DISPLAY 'Out Of Records In SORT File'
        END-RETURN
    END-PERFORM.
```

Related tasks

"Requesting the sort or merge" on page 156

Determining whether the sort or merge was successful

The SORT or MERGE statement returns a completion code of either 0 (successful completion) or 16 (unsuccessful completion) after each sort or merge has finished. The completion code is stored in the SORT-RETURN special register.

You should test for successful completion after each SORT or MERGE statement. For example:

```
    SORT SORT-WORK-2
        ON ASCENDING KEY SORT-KEY
        INPUT PROCEDURE IS 600-SORT3-INPUT-PROC
        OUTPUT PROCEDURE IS 700-SORT3-OUTPUT-PROC.
    IF SORT-RETURN NOT=0
        DISPLAY "SORT ENDED ABNORMALLY. SORT-RETURN = " SORT-RETURN.
    . . .
600-SORT3-INPUT-PROC SECTION.
    . . .
700-SORT3-OUTPUT-PROC SECTION.
    . . .
```

If you do not reference SORT-RETURN anywhere in your program, the COBOL run time tests the completion code. If it is 16, COBOL issues a runtime diagnostic message and terminates the run unit (or the thread, in a multithreaded environment). The diagnostic message contains a sort or merge error number that can help you determine the cause of the problem.

If you test SORT-RETURN for one or more (but not necessarily all) SORT or MERGE statements, the COBOL run time does not check the completion code. However, you can obtain the sort or merge error number after any SORT or MERGE statement by calling the iwzGetSortErrno service; for example:

```
77  sortErrno    PIC 9(9)   COMP-5.
. . .
    CALL 'iwzGetSortErrno'  USING  sortErrno
. . .
```

See the related reference below for a list of the error numbers and their meanings.

Related references
“Sort and merge error numbers” on page 159

Sort and merge error numbers

If you do not reference SORT-RETURN in your program, and the completion code from a sort or merge operation is 16, COBOL for Linux issues a runtime diagnostic message that contains one of the nonzero error numbers shown in the table below.

Table 15. ***Sort and merge error numbers***

Error number	Description
0	No error
1	Record is out of order.
2	Equal-keyed records were detected.
3	Multiple main functions were specified (internal error).
4	Error in the parameter file
5	Parameter file could not be opened.
6	Operand missing from option
7	Operand missing from extended option
8	Invalid operand in option
9	Invalid operand in extended option
10	An invalid option was specified.
11	An invalid extended option was specified.
12	An invalid temporary directory was specified.
13	An invalid file-name was specified.
14	An invalid field was specified.
15	A field was missing in the record.
16	A field was too short in the record.
17	Syntax error in SELECT specification
18	An invalid constant was specified in SELECT.
19	Invalid comparison between constant and data type in SELECT
20	Invalid comparison between two data types in SELECT
21	Syntax error in format specification

*Table 15. **Sort and merge error numbers** (continued)*	
Error number	**Description**
22	Syntax error in reformat specification
23	An invalid constant was specified in the reformat specification.
24	Syntax error in sum specification
25	A flag was specified multiple times.
26	Too many outputs were specified.
27	No input source was specified.
28	No output destination was specified.
29	An invalid modifier was specified.
30	Sum is not allowed.
31	Record is too short.
32	Record is too long.
33	An invalid packed or zoned field was detected.
34	Read error on file
35	Write error on file
36	Cannot open input file.
37	Cannot open message file.
38	SdU or SFS file error
39	Insufficient space in target buffers
40	Not enough temporary disk space
41	Not enough space for output file
42	An unexpected signal was trapped.
43	Error was returned from the input exit.
44	Error was returned from the output exit.
45	Unexpected data was returned from the output user exit.
46	Invalid bytes used value was returned from input exit.
47	Invalid bytes used value was returned from output exit.
48	SMARTsort is not active.
49	Insufficient storage to continue execution
50	Parameter file was too large.
51	Nonmatching single quotation mark
52	Nonmatching quotation mark
53	Conflicting options were specified.
54	Length field in record is invalid.
55	Last field in record is invalid.

*Table 15. **Sort and merge error numbers** (continued)*	
Error number	**Description**
56	Required record format was not specified.
57	Cannot open output file.
58	Cannot open temporary file.
59	Invalid file organization
60	User exit is not supported with the specified file organization.
61	Locale is not known to the system.
62	Record contains an invalid multibyte character.
63	The file was neither SdU nor SFS.
64	No key specified to S0RT is usable for definition of indexed output file.
65	The record length for an SdU or SFS file was not correct.
66	The SMARTsort options file creation failed.
67	A fully qualified, nonrelative path name must be specified as a work directory.
68	A required option must be specified.
69	Path name is not valid.
79	Maximum number of temporary files has been reached.
501	Invalid function
502	Invalid record type
503	Invalid record length
504	Type length error
505	Invalid type
506	Mismatched number of keys
507	Type is too long.
508	Invalid key offset
509	Invalid ascending or descending key
510	Invalid overlapping keys
511	No key was defined.
512	No input file was specified.
513	No output file was specified.
514	Mixed-type input files
515	Mixed-type output files
516	Invalid input work buffer
517	Invalid output work buffer
518	COBOL input I/O error
519	COBOL output I/O error

*Table 15. **Sort and merge error numbers** (continued)*	
Error number	**Description**
520	Unsupported function
521	Invalid key
522	Invalid USING file
523	Invalid GIVING file
524	No work directory was supplied.
525	Work directory does not exist.
526	Sort common was not allocated.
527	No storage for sort common
528	Binary buffer was not allocated.
529	Line-sequential file buffer was not allocated.
530	Work space allocation failed.
531	FCB allocation failed.

Stopping a sort or merge operation prematurely

To stop a sort or merge operation, move the integer 16 into the SORT-RETURN special register.

Move 16 into the register in either of the following ways:

- Use MOVE in an input or output procedure.

  Sort or merge processing will be stopped immediately after the next RELEASE or RETURN statement is performed.
- Reset the register in a declarative section entered during processing of a USING or GIVING file.

  Sort or merge processing will be stopped on exit from the declarative section.

Control then returns to the statement following the SORT or MERGE statement.

Chapter 9. Handling errors

Put code in your programs that anticipates possible system or runtime problems. If you do not include such code, output data or files could be corrupted, and the user might not even be aware that there is a problem.

The error-handling code can take actions such as handling the situation, issuing a message, or halting the program. You might for example create error-detection routines for data-entry errors or for errors as your installation defines them. In any event, coding a warning message is a good idea.

COBOL for Linux contains special elements to help you anticipate and correct error conditions:

- `ON OVERFLOW` in `STRING` and `UNSTRING` operations
- `ON SIZE ERROR` in arithmetic operations
- Elements for handling input or output errors
- `ON EXCEPTION` or `ON OVERFLOW` in `CALL` statements

Related tasks
"Handling errors in joining and splitting strings" on page 163
"Handling errors in arithmetic operations" on page 164
"Handling errors in input and output operations" on page 164
"Handling errors when calling programs" on page 170

Handling errors in joining and splitting strings

During the joining or splitting of strings, the pointer used by `STRING` or `UNSTRING` might fall outside the range of the receiving field. A potential overflow condition exists, but COBOL does not let the overflow happen.

Instead, the `STRING` or `UNSTRING` operation is not completed, the receiving field remains unchanged, and control passes to the next sequential statement. If you do not code the `ON OVERFLOW` phrase of the `STRING` or `UNSTRING` statement, you are not notified of the incomplete operation.

Consider the following statement:

```
String Item-1 space Item-2 delimited by Item-3
    into Item-4
    with pointer String-ptr
    on overflow
        Display "A string overflow occurred"
End-String
```

These are the data values before and after the statement is performed:

Data item	PICTURE	Value before	Value after
Item-1	X(5)	AAAAA	AAAAA
Item-2	X(5)	EEEAA	EEEAA
Item-3	X(2)	EA	EA
Item-4	X(8)	*bbbbbbbb*[1]	*bbbbbbbb*[1]
String-ptr	9(2)	0	0
1. The symbol *b* represents a blank space.			

© Copyright IBM Corp. 2021

Because String-ptr has a value (0) that falls short of the receiving field, an overflow condition occurs and the STRING operation is not completed. (Overflow would also occur if String-ptr were greater than 9.) If ON OVERFLOW had not been specified, you would not be notified that the contents of Item-4 remained unchanged.

Handling errors in arithmetic operations

The results of arithmetic operations might be larger than the fixed-point field that is to hold them, or you might have tried dividing by zero. In either case, the ON SIZE ERROR clause after the ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statement can handle the situation.

For ON SIZE ERROR to work correctly for fixed-point overflow and decimal overflow, you must specify the TRAP(ON) runtime option.

The imperative statement of the ON SIZE ERROR clause will be performed and the result field will not change in these cases:

- Fixed-point overflow
- Division by zero
- Zero raised to the zero power
- Zero raised to a negative number
- Negative number raised to a fractional power

Example: checking for division by zero

The following example shows how you can code an ON SIZE ERROR imperative statement so that the program issues an informative message if division by zero occurs.

```
DIVIDE-TOTAL-COST.
    DIVIDE TOTAL-COST BY NUMBER-PURCHASED
        GIVING ANSWER
        ON SIZE ERROR
          DISPLAY "ERROR IN DIVIDE-TOTAL-COST PARAGRAPH"
          DISPLAY "SPENT " TOTAL-COST, " FOR " NUMBER-PURCHASED
          PERFORM FINISH
    END-DIVIDE
    . . .
    FINISH.
    STOP RUN.
```

If division by zero occurs, the program writes a message and halts program execution.

Handling errors in input and output operations

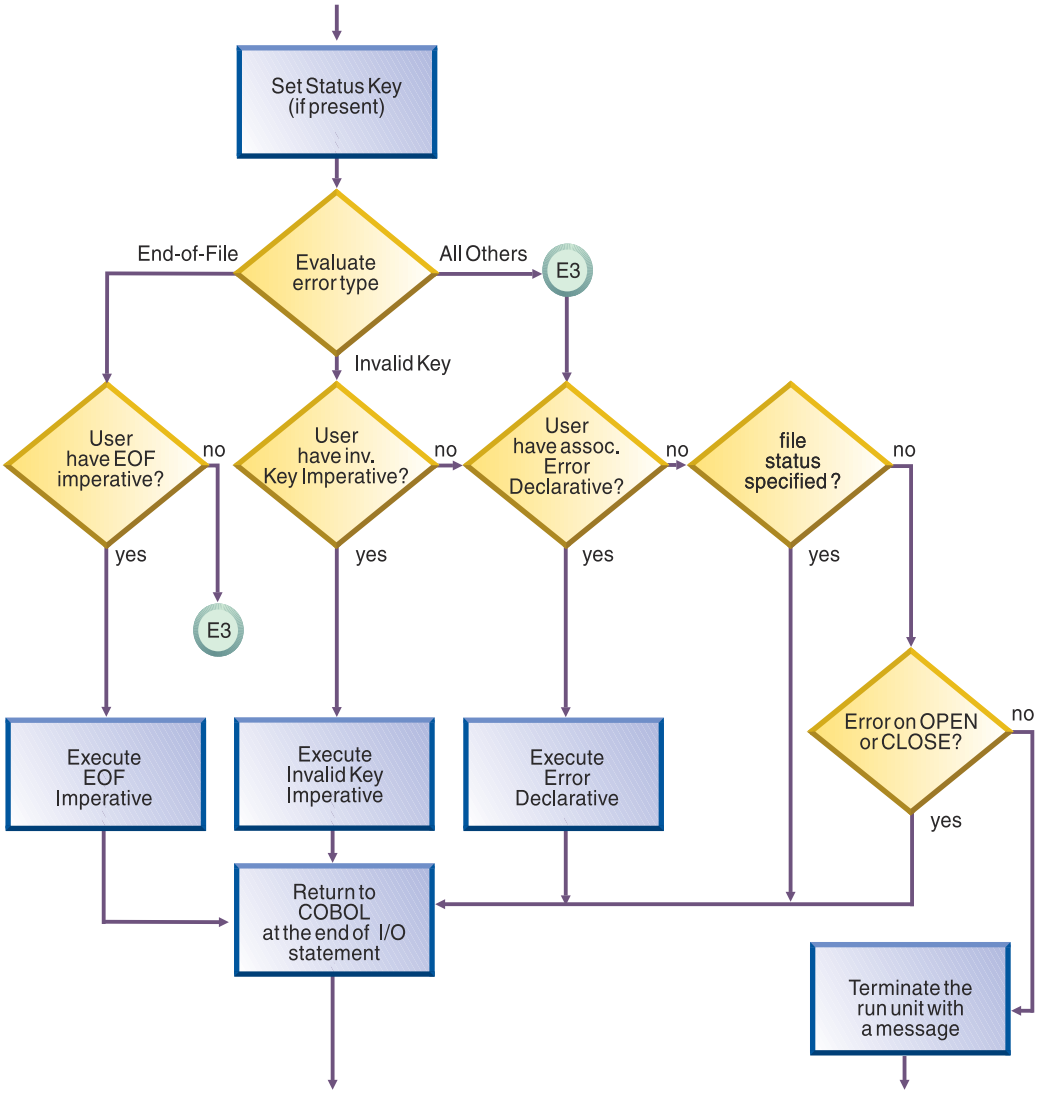
When an input or output operation fails, COBOL does not automatically take corrective action. You choose whether your program will continue running after a less-than-severe input or output error.

You can use any of the following techniques for intercepting and handling certain input or output conditions or errors:

- End-of-file condition (AT END)
- ERROR declaratives
- FILE STATUS clause and file status key
- File system status code
- Imperative-statement phrases in READ or WRITE statements
- INVALID KEY phrase

To have your program continue, you must code the appropriate error-recovery procedure. You might code, for example, a procedure to check the value of the file status key. If you do not handle an input or output error in any of these ways, a COBOL runtime message is written and the run unit ends.

The following figure shows the flow of logic after a file-system input or output error:



Related tasks

"Opening optional files" on page 132
"Using the end-of-file condition (AT END)" on page 166
"Coding ERROR declaratives" on page 166
"Using file status keys" on page 166
"Using file system status codes" on page 168
"Coding INVALID KEY phrases" on page 170

Related references

File status key (*COBOL for Linux on x86 Language Reference*)

Using the end-of-file condition (AT END)

You code the AT END phrase of the READ statement to handle errors or normal conditions, according to your program design. At end-of-file, the AT END phrase is performed. If you do not code an AT END phrase, the associated ERROR declarative is performed.

In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected. For example, suppose you are processing a file that contains transactions in order to update a main file:

```
PERFORM UNTIL TRANSACTION-EOF = "TRUE"
  READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
    AT END
      DISPLAY "END OF TRANSACTION UPDATE FILE REACHED"
      MOVE "TRUE" TO TRANSACTION-EOF
  END READ
  . . .
END-PERFORM
```

Any NOT AT END phrase is performed only if the READ statement completes successfully. If the READ operation fails because of a condition other than end-of-file, neither the AT END nor the NOT AT END phrase is performed. Instead, control passes to the end of the READ statement after any associated declarative procedure is performed.

You might choose not to code either an AT END phrase or an EXCEPTION declarative procedure, but to code a status key clause for the file instead. In that case, control passes to the next sequential instruction after the input or output statement that detected the end-of-file condition. At that place, have some code that takes appropriate action.

Related references
AT END phrases (*COBOL for Linux on x86 Language Reference*)

Coding ERROR declaratives

You can code one or more ERROR declarative procedures that will be given control if an input or output error occurs during the execution of your program. If you do not code such procedures, your job could be canceled or abnormally terminated after an input or output error occurs.

Place each such procedure in the declaratives section of the PROCEDURE DIVISION. You can code:

- A single, common procedure for the entire program
- Procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each file

In an ERROR declarative procedure, you can code corrective action, retry the operation, continue, or end execution. (If you continue processing a blocked file, though, you might lose the remaining records in a block after the record that caused the error.) You can use the ERROR declaratives procedure in combination with the file status key if you want a further analysis of the error.

Related references
EXCEPTION/ERROR declarative (*COBOL for Linux on x86 Language Reference*)

Using file status keys

After each input or output statement is performed on a file, the system updates values in the two digit positions of the file status key. In general, a zero in the first position indicates a successful operation, and a zero in both positions means that nothing abnormal occurred.

Establish a file status key by coding:

- The FILE STATUS clause in the FILE-CONTROL paragraph:

```
FILE STATUS IS data-name-1
```

- Data definitions in the DATA DIVISION (WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION), for example:

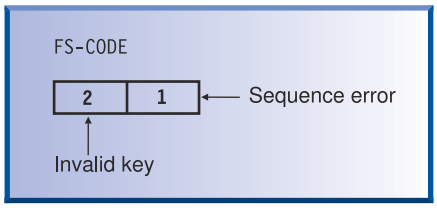
```
WORKING-STORAGE SECTION.
01  data-name-1  PIC 9(2)  USAGE NATIONAL.
```

Specify the file status key *data-name-1* as a two-character category alphanumeric or category national item, or as a two-digit zoned decimal or national decimal item. This *data-name-1* cannot be variably located.

Your program can check the file status key to discover whether an error occurred, and, if so, what type of error occurred. For example, suppose that a FILE STATUS clause is coded like this:

```
FILE STATUS IS FS-CODE
```

FS-CODE is used by COBOL to hold status information like this:



Follow these rules for each file:

- Define a different file status key for each file.

  Doing so means that you can determine the cause of a file input or output exception, such as an application logic error or a disk error.

- Check the file status key after each input or output request.

  If the file status key contains a value other than 0, your program can issue an error message or can take action based on that value.

  You do not have to reset the file status key code, because it is set after each input or output attempt.

In addition to the file status key, you can code a second identifier in the FILE STATUS clause to get more detailed information about file-system input or output requests. For details, see the related task about file system status codes.

You can use the file status key alone or in conjunction with the INVALID KEY phrase, or to supplement the EXCEPTION or ERROR declarative. Using the file status key in this way gives you precise information about the results of each input or output operation.

"Example: file status key" on page 168
"Example: checking file system status codes" on page 169

Related tasks

"Setting up a field for file status" on page 132
"Using file system status codes" on page 168
"Coding INVALID KEY phrases" on page 170
"Finding and handling input-output errors" on page 302

Related references
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)
File status key (*COBOL for Linux on x86 Language Reference*)

Example: file status key

The following example shows how you can perform a simple check of the file status key after opening a file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SIMCHK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MAINFILE ASSIGN TO AS-MAINA
    FILE STATUS IS MAINFILE-CHECK-KEY
    . . .
DATA DIVISION.
. . .
WORKING-STORAGE SECTION.
01  MAINFILE-CHECK-KEY       PIC X(2).
. . .
PROCEDURE DIVISION.
    OPEN INPUT MAINFILE
    IF MAINFILE-CHECK-KEY NOT = "00"
        DISPLAY "Nonzero file status returned from OPEN " MAINFILE-CHECK-KEY
    . . .
```

Using file system status codes

Often the two-digit file status key is too general to pinpoint the result of an input or output request. You can get more detailed information about Db2, LSQ, QSAM, RSD, SdU, SFS, and STL file-system requests by coding a second data item in the `FILE STATUS` clause.

```
FILE STATUS IS data-name-1 data-name-8
```

In the example above, the data item *data-name-1* specifies the two-digit COBOL file status key, which must be a two-character category alphanumeric or category national item, or a two-digit zoned decimal or national decimal item. The data item *data-name-8* specifies a data item that contains the file-system status code if the COBOL file status key is not zero. *data-name-8* is at least 6 bytes long, and must be an alphanumeric item.

LSQ, QSAM, RSD, SFS, STL and SdU files: For LSQ, QSAM, RSD, SFS, STL and SdU file system input and output requests, if *data-name-8* is 6 bytes long, it contains the file status code. If *data-name-8* is longer than 6 bytes, it also contains a message with further information:

```
01  my-file-status-2.
    02 exception-return-value PIC 9(6).
    02 additional-info        PIC X(100).
```

The `exception-return-value` contains a value that can further refine the error noted in `FILE STATUS`. The `additional-info` contains further diagnostic information of the error noted in `exception-return-value` to help you diagnose the problem.

Db2 files: For Db2 file-system input and output requests, define *data-name-8* as a group item. For example:

```
01  FileStatus2.
    02 FS2-SQLCODE  PICTURE S9(9) COMP.
    02 FS2-SQLSTATE PICTURE X(5).
```

The runtime values in `FS2-SQLCODE` and `FS2-SQLSTATE` represent SQL feedback information for the operation previously completed.

“Example: checking file system status codes” on page 169

Related tasks
“Fixing differences caused
by language elements” on page 378

Related references
“Db2 file system” on page 117
“QSAM file system” on page 118
“SdU file system” on page 119
“SFS file system” on page 119
“STL file system” on page 120
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)
File status key (*COBOL for Linux on x86 Language Reference*)

Example: checking file system status codes

The following example reads an indexed file starting at the fifth record and checks the file status key after each input or output request. The file status codes are displayed if the file status key is not zero.

This example also illustrates how output from this program might look if the file being processed contained six records.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILESYSFILE ASSIGN TO FILESYSFILE
    ORGANIZATION IS INDEXED
    ACCESS DYNAMIC
    RECORD KEY IS FILESYSFILE-KEY
    FILE STATUS IS FS-CODE, FILESYS-CODE.
DATA DIVISION.
FILE SECTION.
FD  FILESYSFILE
    RECORD  30.
01  FILESYSFILE-REC.
    10 FILESYSFILE-KEY          PIC X(6).
    10 FILLER                   PIC X(24).
WORKING-STORAGE SECTION.
01  RETURN-STATUS.
    05 FS-CODE                  PIC XX.
    05 FILESYS-CODE             PIC X(6).
PROCEDURE DIVISION.
    OPEN  INPUT FILESYSFILE.
    DISPLAY "OPEN INPUT FILESYSFILE FS-CODE: " FS-CODE.

    IF FS-CODE NOT = "00"
       PERFORM FILESYS-CODE-DISPLAY
       STOP RUN
    END-IF.

    MOVE "000005" TO FILESYSFILE-KEY.
    START FILESYSFILE KEY IS EQUAL TO FILESYSFILE-KEY.
    DISPLAY "START FILESYSFILE KEY="  FILESYSFILE-KEY
            " FS-CODE: "  FS-CODE.

    IF FS-CODE NOT = "00"
       PERFORM FILESYS-CODE-DISPLAY
    END-IF.

    IF FS-CODE = "00"
       PERFORM READ-NEXT UNTIL FS-CODE NOT = "00"
    END-IF.

    CLOSE FILESYSFILE.
    STOP RUN.

READ-NEXT.
    READ FILESYSFILE NEXT.
    DISPLAY "READ NEXT FILESYSFILE FS-CODE: " FS-CODE.
    IF FS-CODE NOT = "00"
```

```
       PERFORM FILESYS-CODE-DISPLAY
    END-IF.
    DISPLAY FILESYSFILE-REC.

FILESYS-CODE-DISPLAY.
    DISPLAY "FILESYS-CODE ==>", FILESYS-CODE.
```

Coding INVALID KEY phrases

You can include an INVALID KEY phrase in READ, START, WRITE, REWRITE, and DELETE statements for indexed and relative files. The INVALID KEY phrase is given control if an input or output error occurs due to a faulty index key.

Use the FILE STATUS clause with the INVALID KEY phrase to evaluate the status key and determine the specific INVALID KEY condition.

INVALID KEY phrases differ from ERROR declaratives in several ways. INVALID KEY phrases:

- Operate for only limited types of errors. ERROR declaratives encompass all forms.
- Are coded directly with the input or output statement. ERROR declaratives are coded separately.
- Are specific for a single input or output operation. ERROR declaratives are more general.

If you code INVALID KEY in a statement that causes an INVALID KEY condition, control is transferred to the INVALID KEY imperative statement. Any ERROR declaratives that you coded are not performed.

If you code a NOT INVALID KEY phrase, it is performed only if the statement completes successfully. If the operation fails because of a condition other than INVALID KEY, neither the INVALID KEY nor the NOT INVALID KEY phrase is performed. Instead, after the program performs any associated ERROR declaratives, control passes to the end of the statement.

“Example: FILE STATUS and INVALID KEY” on page 170

Example: FILE STATUS and INVALID KEY

The following example shows how you can use the file status code and the INVALID KEY phrase to determine more specifically why an input or output statement failed.

Assume that you have a file that contains main customer records and you need to update some of these records with information from a transaction update file. The program reads each transaction record, finds the corresponding record in the main file, and makes the necessary updates. The records in both files contain a field for a customer number, and each record in the main file has a unique customer number.

The FILE-CONTROL entry for the main file of customer records includes statements that define indexed organization, random access, MAIN-CUSTOMER-NUMBER as the prime record key, and CUSTOMER-FILE-STATUS as the file status key.

```
.
.  (read the update transaction record)
.
MOVE "TRUE" TO TRANSACTION-MATCH
MOVE UPDATE-CUSTOMER-NUMBER TO MAIN-CUSTOMER-NUMBER
READ MAIN-CUSTOMER-FILE INTO WS-CUSTOMER-RECORD
  INVALID KEY
    DISPLAY "MAIN CUSTOMER RECORD NOT FOUND"
    DISPLAY "FILE STATUS CODE IS: " CUSTOMER-FILE-STATUS
    MOVE "FALSE" TO TRANSACTION-MATCH
END-READ
```

Handling errors when calling programs

When a program dynamically calls a separately compiled program, the called program might be unavailable. For example, the system might be out of storage or unable to locate the program object. If

the CALL statement does not have an ON EXCEPTION or ON OVERFLOW phrase, your application might abend.

Use the ON EXCEPTION phrase to perform a series of statements and to perform your own error handling. For example, in the code fragment below, if program REPORTA is unavailable, control passes to the ON EXCEPTION phrase.

```
MOVE "REPORTA" TO REPORT-PROG
CALL REPORT-PROG
  ON EXCEPTION
    DISPLAY "Program REPORTA not available, using REPORTB."
    MOVE "REPORTB" TO REPORT-PROG
    CALL REPORT-PROG
    END-CALL
END-CALL
```

The ON EXCEPTION phrase applies only to the availability of the called program on its initial load. If the called program is loaded but fails for any other reason (such as initialization), the ON EXCEPTION phrase is not performed.

Part 2. Enabling programs for international environments

© Copyright IBM Corp. 2021

Chapter 10. Processing data in an international environment

COBOL for Linux supports Unicode UTF-16 as national character data at run time. UTF-16 is a fixed-width Unicode encoding that provides a consistent and efficient way to encode plain text. Using UTF-16, you can develop software that will work with various national languages.

Use these COBOL facilities to code and compile programs that process national data and culturally sensitive collation orders for such data:

- Data types and literals:
  - Character data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines data of category national, national-edited, or numeric-edited
  - Numeric data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines a numeric data item (a *national decimal item*) or an external floating-point data item (a *national floating-point item*)
  - National literals, specified with literal prefix `N` or `NX`
  - Figurative constant `ALL` *national-literal*
  - Figurative constants `QUOTE`, `SPACE`, `HIGH-VALUE`, `LOW-VALUE`, or `ZERO`, which have national character (UTF-16) values when used in national-character contexts
- The COBOL statements shown in the related reference below about COBOL statements and national data
- Intrinsic functions:
  - `NATIONAL-OF` to convert an alphanumeric or double-byte character set (DBCS) character string to `USAGE NATIONAL` (UTF-16)
  - `DISPLAY-OF` to convert a national character string to `USAGE DISPLAY` in a selected code page (EBCDIC, ASCII, EUC, or UTF-8)
  - The other intrinsic functions shown in the related reference below about intrinsic functions and national data
- The `GROUP-USAGE NATIONAL` clause to define groups that contain only `USAGE NATIONAL` data items and that behave like elementary category national items in most operations
- Compiler options:
  - `NSYMBOL` to control whether national or DBCS processing is used for the `N` symbol in literals and `PICTURE` clauses
  - `NCOLLSEQ` to specify the collating sequence for comparison of national operands

You can also take advantage of implicit conversions of alphanumeric or DBCS data items to national representation. The compiler performs such conversions (in most cases) when you move these items to national data items, or compare these items with national data items.

Related concepts
"Unicode and the encoding of language characters" on page 176
"National groups" on page 183

Related tasks
"Using national data (Unicode) in COBOL" on page 177
"Converting to or from national (Unicode) representation" on page 184
"Processing UTF-8 data using UTF-16 (national) data types" on page 193

© Copyright IBM Corp. 2021

"Processing Chinese GB 18030 data" on page 193
"Comparing national (UTF-16) data" on page 190
"Coding for use of DBCS support" on page 194
Chapter 11, "Setting the locale," on page 199

Related references
"COBOL statements and national data" on page 179
"Intrinsic functions and national data" on page 181
"NCOLLSEQ" on page 271
"NSYMBOL" on page 272
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
Data categories and PICTURE rules (*COBOL for Linux on x86 Language Reference*)
MOVE statement (*COBOL for Linux on x86 Language Reference*)
General relation conditions (*COBOL for Linux on x86 Language Reference*)

Unicode and the encoding of language characters

COBOL for Linux provides basic runtime support for Unicode, which can handle tens of thousands of characters that cover all commonly used characters and symbols in the world.

A *character set* is a defined set of characters, but is not associated with a coded representation. A *coded character set* (also referred to in this documentation as a *code page*) is a set of unambiguous rules that relate the characters of the set to their coded representation. Each code page has a name and is like a table that sets up the symbols for representing a character set; each symbol is associated with a unique bit pattern, or *code point*. Each code page also has a *coded character set identifier* (*CCSID*), which is a value from 1 to 65,536.

Unicode has several encoding schemes, called *Unicode Transformation Format (UTF)*, such as UTF-8, UTF-16, and UTF-32. COBOL for Linux uses UTF-16 (CCSID 1200) in little-endian format as the representation for national literals and data items that have `USAGE NATIONAL`.

UTF-8 represents ASCII invariant characters a-z, A-Z, 0-9, and certain special characters such as ' @ , . + - = / * ( ) the same way that they are represented in ASCII. UTF-16 represents these characters as `NX'`*nn*`00'`, where `X'`*nn*`'` is the representation of the character in ASCII.

For example, the string `'ABC'` is represented in UTF-16 as `NX'410042004300'`. In UTF-8, `'ABC'` is represented as `X'414243'`.

One or more *encoding units* are used to represent a character from a coded character set. For UTF-16, an encoding unit takes 2 bytes of storage. Any character defined in any EBCDIC, ASCII, or EUC code page is represented in one UTF-16 encoding unit when the character is converted to the national data representation.

Cross-platform considerations: Enterprise COBOL for z/OS and COBOL for AIX® support UTF-16 in big-endian format in national data. By default, COBOL for Linux supports UTF-16 in little-endian format in national data. If you are porting Unicode data that is encoded in UTF-16BE representation to COBOL for Linux from another platform, you must either convert that data to UTF-16 in little-endian format to process the data as national data, or use the UTF16 compiler option to change the way the compiler treats UTF-16 endianness. With COBOL for Linux, you can perform such conversions by using the NATIONAL-OF intrinsic function.

Related tasks
"Converting to or from national (Unicode) representation" on page 184

Related references
"Storage of character data" on page 190
"Locales and code pages that are supported" on page 202
Character sets and code pages (*COBOL for Linux on x86 Language Reference*)

Using national data (Unicode) in COBOL

In COBOL for Linux, you can specify national (UTF-16) data in any of several ways.

These types of national data are available:

- National data items (categories national, national-edited, and numeric-edited)
- National literals
- Figurative constants as national characters
- Numeric data items (national decimal and national floating-point)

In addition, you can define national groups that contain only data items that explicitly or implicitly have `USAGE NATIONAL`, and that behave in the same way as elementary category national data items in most operations.

These declarations affect the amount of storage that is needed.

Related concepts
"Unicode and the encoding of language characters" on page 176
"National groups" on page 183

Related tasks
"Defining national data items" on page 177
"Using national literals" on page 178
"Using national-character figurative constants" on page 182
"Defining national numeric data items" on page 183
"Using national groups" on page 187
"Converting to or from national (Unicode) representation" on page 184
"Comparing national (UTF-16) data" on page 190

Related references
"Storage of character data" on page 190
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)

Defining national data items

Define national data items with the `USAGE NATIONAL` clause to hold national (UTF-16) character strings.

You can define national data items of the following categories:

- National
- National-edited
- Numeric-edited

To define a category national data item, code a `PICTURE` clause that contains only one or more `PICTURE` symbols `N`.

To define a national-edited data item, code a PICTURE clause that contains at least one of each of the following symbols:

- Symbol N
- Simple insertion editing symbol B, 0, or /

To define a numeric-edited data item of class national, code a PICTURE clause that defines a numeric-edited item (for example, -$999.99) and code a USAGE NATIONAL clause. You can use a numeric-edited data item that has USAGE NATIONAL in the same way that you use a numeric-edited item that has USAGE DISPLAY.

You can also define a data item as numeric-edited by coding the BLANK WHEN ZERO clause for an elementary item that is defined as numeric by its PICTURE clause.

If you code a PICTURE clause but do not code a USAGE clause for data items that contain only one or more PICTURE symbols N, you can use the compiler option NSYMBOL(NATIONAL) to ensure that such items are treated as national data items instead of as DBCS items.

Related tasks
"Displaying numeric data" on page 37

Related references
"NSYMBOL" on page 272
BLANK WHEN ZERO clause (*COBOL for Linux on x86 Language Reference*)

Using national literals

To specify national literals, use the prefix character N and compile with the option NSYMBOL(NATIONAL).

You can use either of these notations:

- N"*character-data*"
- N'*character-data*'

If you compile with the option NSYMBOL(DBCS), the literal prefix character N specifies a DBCS literal, not a national literal.

To specify a national literal as a hexadecimal value, use the prefix NX. You can use either of these notations:

- NX"*hexadecimal-digits*"
- NX'*hexadecimal-digits*'

Each of the following MOVE statements sets the national data item Y to the UTF-16 value of the characters 'AB':

```
01 Y pic NN usage national.
. . .
    Move NX"41004200" to Y
    Move N"AB"        to Y
    Move "AB"         to Y
```

Do not use alphanumeric hexadecimal literals in contexts that call for national literals, because such usage is easily misunderstood. For example, the following statement also results in moving the UTF-16 characters 'AB' (not the hexadecimal bit pattern 4142) to Y, where Y is defined as USAGE NATIONAL:

```
Move X"4142" to Y
```

You cannot use national literals in the SPECIAL-NAMES paragraph or as program-names. You can use a national literal to name an object-oriented method in the METHOD-ID paragraph or to specify a method-name in an INVOKE statement.

Use the SOSI compiler option to control how shift-out and shift-in characters within a national literal are handled.

Related tasks
"Using literals" on page 21

Related references
"NSYMBOL" on page 272
"SOSI" on page 277
National literals (*COBOL for Linux on x86 Language Reference*)

COBOL statements and national data

You can use national data with the PROCEDURE DIVISION and compiler-directing statements shown in the table below.

Table 16. ***COBOL statements and national data***

COBOL statement	Can be national	Comment	For more information
ACCEPT	*identifier-1*, *identifier-2*	*identifier-1* is converted from the code page indicated by the runtime locale only if input is from the terminal.	"Assigning input from a screen or file (ACCEPT)" on page 30
ADD	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 49
CALL	*identifier-2*, *identifier-3*, *identifier-4*, *identifier-5*; *literal-2*, *literal-3*		"Passing data" on page 395
COMPUTE	*identifier-1* can be numeric or numeric-edited with USAGE NATIONAL. *arithmetic-expression* can contain numeric items that have USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 49
COPY . . . REPLACING	*operand-1*, *operand-2* of the REPLACING phrase		Chapter 14, "Compiler-directing statements," on page 291
DISPLAY	*identifier-1*	*identifier-1* is converted to the code page associated with the current locale.	"Displaying values on a screen or in a file (DISPLAY)" on page 31
DIVIDE	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) and *identifier-4* (REMAINDER) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 49

*Table 16. **COBOL statements and national data** (continued)*

COBOL statement	Can be national	Comment	For more information
INITIALIZE	*identifier-1*; *identifier-2* or *literal-1* of the REPLACING phrase	If you specify REPLACING NATIONAL or REPLACING NATIONAL-EDITED, *identifier-2* or *literal-1* must be valid as a sending operand in a move to *identifier-1*.	"Examples: initializing data items" on page 24
INSPECT	All identifiers and literals. (*identifier-2*, the TALLYING integer data item, can have USAGE NATIONAL.)	If any of these (other than *identifier-2*, the TALLYING identifier) have USAGE NATIONAL, all must be national.	"Tallying and replacing data items (INSPECT)" on page 102
MERGE	Merge keys, if you specify NCOLLSEQ(BIN)	The COLLATING SEQUENCE phrase does not apply.	"Setting sort or merge criteria" on page 156
MOVE	Both the sender and receiver, or only the receiver	Implicit conversions are performed for valid MOVE operands.	"Assigning values to elementary data items (MOVE)" on page 28 "Assigning values to group data items (MOVE)" on page 29
MULTIPLY	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 49
SEARCH ALL (binary search)	Both the key data item and its object of comparison	The key data item and its object of comparison must be compatible according to the rules of comparison. If the object of comparison is of class national, the key must be also.	"Doing a binary search (SEARCH ALL)" on page 78
SORT	Sort keys, if you specify NCOLLSEQ(BIN)	The COLLATING SEQUENCE phrase does not apply.	"Setting sort or merge criteria" on page 156
STRING	All identifiers and literals. (*identifier-4*, the POINTER integer data item, can have USAGE NATIONAL.)	If *identifier-3*, the receiving data item, is national, all identifiers and literals (other than *identifier-4*, the POINTER identifier) must be national.	"Joining data items (STRING)" on page 93

Table 16. **COBOL statements and national data** *(continued)*			
COBOL statement	**Can be national**	**Comment**	**For more information**
SUBTRACT	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 49
UNSTRING	All identifiers and literals. (*identifier-6* and *identifier-7*, the COUNT and TALLYING integer data items, respectively, can have USAGE NATIONAL.)	If *identifier-4*, a receiving data item, has USAGE NATIONAL, the sending data item and each delimiter must have USAGE NATIONAL, and each literal must be national.	"Splitting data items (UNSTRING)" on page 95
XML GENERATE	*identifier-1* (the generated XML document); *identifier-2* (the source field or fields); *identifier-4* or *literal-4* (the namespace identifier); *identifier-5* or *literal-5* (the namespace prefix)		Chapter 20, "Producing XML output," on page 359
XML PARSE	*identifier-1* (the XML document)	The XML-NTEXT special register contains national character document fragments during parsing.	Chapter 19, "Processing XML input," on page 339

Related tasks
"Defining numeric data" on page 35
"Displaying numeric data" on page 37
"Using national data (Unicode) in COBOL" on page 177
"Comparing national (UTF-16) data" on page 190

Related references
"NCOLLSEQ" on page 271
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)

Intrinsic functions and national data

You can use arguments of class national with the intrinsic functions shown in the table below.

Table 17. **Intrinsic functions and national character data**		
Intrinsic function	**Function type**	**For more information**
DISPLAY-OF	Alphanumeric	"Converting national to alphanumeric (DISPLAY-OF)" on page 186
LENGTH	Integer	"Finding the length of data items" on page 109

*Table 17. **Intrinsic functions and national character data** (continued)*		
Intrinsic function	**Function type**	**For more information**
LOWER-CASE, UPPER-CASE	National	“Changing case (UPPER-CASE, LOWER-CASE)” on page 104
NUMVAL, NUMVAL-C,	Numeric	“Converting to numbers (NUMVAL, NUMVAL-C)” on page 105
MAX, MIN	National	“Finding the largest or smallest data item” on page 107
ORD-MAX, ORD-MIN	Integer	“Finding the largest or smallest data item” on page 107
REVERSE	Alphanumeric or national	“Transforming to reverse order (REVERSE)” on page 105

You can use national decimal arguments wherever zoned decimal arguments are allowed. You can use national floating-point arguments wherever display floating-point arguments are allowed. (See the related reference below about arguments for a complete list of intrinsic functions that can take integer or numeric arguments.)

Related tasks
“Defining numeric data” on page 35
“Using national data (Unicode) in COBOL” on page 177

Related references
Arguments (*COBOL for Linux on x86 Language Reference*)
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
Intrinsic functions (*COBOL for Linux on x86 Language Reference*)

Using national-character figurative constants

You can use the figurative constant ALL *national-literal* in a context that requires national characters. ALL *national-literal* represents all or part of the string that is generated by successive concatenations of the encoding units that make up the national literal.

You can use the figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO in a context that requires national characters, such as a MOVE statement, an implicit move, or a relation condition that has national operands. In these contexts, the figurative constant represents a national-character (UTF-16) value.

When you use the figurative constant HIGH-VALUE in a context that requires national characters, its value is NX'FFFF'. When you use LOW-VALUE in a context that requires national characters, its value is NX'0000'. You can use HIGH-VALUE or LOW-VALUE in a context that requires national characters only if the NCOLLSEQ(BIN) compiler option is in effect.

Restrictions: You must not use HIGH-VALUE or the value assigned from HIGH-VALUE in a way that results in conversion of the value from one data representation to another (for example, between USAGE DISPLAY and USAGE NATIONAL, or between ASCII and EBCDIC when the CHAR(EBCDIC) compiler option is in effect). X'FF' (the value of HIGH-VALUE in an alphanumeric context when the EBCDIC collating sequence is being used) does not represent a valid EBCDIC or ASCII character, and NX'FFFF' does not represent a valid national character. Conversion of such a value to another representation results in a *substitution character* being used (not X'FF' or NX'FFFF'). Consider the following example:

```
01 natl-data  PIC NN  Usage National.
01 alph-data  PIC XX.
. . .
    MOVE HIGH-VALUE TO natl-data, alph-data
    IF natl-data = alph-data. . .
```

The IF statement above evaluates as false even though each of its operands was set to HIGH-VALUE. Before an elementary alphanumeric operand is compared to a national operand, the alphanumeric operand is treated as though it were moved to a temporary national data item, and the alphanumeric characters are converted to the corresponding national characters. When X'FF' is converted to UTF-16, however, the UTF-16 item gets a substitution character value and so does not compare equally to NX'FFFF'.

Related tasks
"Converting to or from national (Unicode) representation" on page 184
"Comparing national (UTF-16)
data" on page 190

Related references
"CHAR" on page 253
"NCOLLSEQ" on page 271
Figurative constants (*COBOL for Linux on x86 Language Reference*)
DISPLAY-OF (*COBOL for Linux on x86 Language Reference*)

Defining national numeric data items

Define data items with the USAGE NATIONAL clause to hold numeric data that is represented in national characters (UTF-16). You can define national decimal items and national floating-point items.

To define a national decimal item, code a PICTURE clause that contains only the symbols 9, P, S, and V. If the PICTURE clause contains S, the SIGN IS SEPARATE clause must be in effect for that item.

To define a national floating-point item, code a PICTURE clause that defines a floating-point item (for example, +99999.9E-99).

You can use national decimal items in the same way that you use zoned decimal items. You can use national floating-point items in the same way that you use display floating-point items.

Related tasks
"Defining numeric data" on page 35
"Displaying numeric data" on page 37

Related references
SIGN clause (*COBOL for Linux on x86 Language Reference*)

National groups

National groups, which are specified either explicitly or implicitly with the GROUP-USAGE NATIONAL clause, contain only data items that have USAGE NATIONAL. In most cases, a national group item is processed as though it were redefined as an elementary category national item described as PIC N(*m*), where *m* is the number of national (UTF-16) characters in the group.

For some operations on national groups, however (just as for some operations on alphanumeric groups), group semantics apply. Such operations (for example, MOVE CORRESPONDING and INITIALIZE) recognize or process the elementary items within the national group.

Where possible, use national groups instead of alphanumeric groups that contain USAGE NATIONAL items. National groups provide several advantages for the processing of national data compared to the processing of national data within alphanumeric groups:

- When you move a national group to a longer data item that has USAGE NATIONAL, the receiving item is padded with national characters. By contrast, if you move an alphanumeric group that contains national characters to a longer alphanumeric group that contains national characters, alphanumeric spaces are used for padding. As a result, mishandling of data items could occur.

- When you move a national group to a shorter data item that has USAGE NATIONAL, the national group is truncated at national-character boundaries. By contrast, if you move an alphanumeric group that contains national characters to a shorter alphanumeric group that contains national characters, truncation might occur between the 2 bytes of a national character.
- When you move a national group to a national-edited or numeric-edited item, the content of the group is edited. By contrast, if you move an alphanumeric group to an edited item, no editing takes place.
- When you use a national group as an operand in a STRING, UNSTRING, or INSPECT statement:
  - The group content is processed as national characters rather than as single-byte characters.
  - TALLYING and POINTER operands operate at the logical level of national characters.
  - The national group operand is supported with a mixture of other national operand types.

  By contrast, if you use an alphanumeric group that contains national characters in these contexts, the characters are processed byte by byte. As a result, invalid handling or corruption of data could occur.

USAGE NATIONAL groups: A group item can specify the USAGE NATIONAL clause at the group level as a convenient shorthand for the USAGE of each of the elementary data items within the group. Such a group is *not* a national group, however, but an alphanumeric group, and behaves in many operations, such as moves and compares, like an elementary data item of USAGE DISPLAY (except that no editing or conversion of data occurs).

Related tasks
"Assigning values to group data items (MOVE)" on page 29
"Joining data items (STRING)" on page 93
"Splitting data items (UNSTRING)" on page 95
"Tallying and replacing
data items (INSPECT)" on page 102
"Using national groups" on page 187

Related references
GROUP-USAGE clause (*COBOL for Linux on x86 Language Reference*)

Converting to or from national (Unicode) representation

You can implicitly or explicitly convert data items to national (UTF-16) representation.

You can implicitly convert alphabetic, alphanumeric, DBCS, or integer data to national data by using the MOVE statement. Implicit conversions also take place in other COBOL statements, such as IF statements that compare an alphanumeric data item with a data item that has USAGE NATIONAL.

You can explicitly convert to and from national data items by using the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively. By using these intrinsic functions, you can specify a code page for the conversion that is different from the code page that is in effect for a data item.

Related tasks
"Converting alphanumeric, DBCS, and integer to national (MOVE)" on page 184
"Converting alphanumeric or DBCS to national (NATIONAL-OF)" on page 185
"Converting national to
alphanumeric (DISPLAY-OF)" on page 186
"Overriding the default
code page" on page 186
"Comparing national (UTF-16)
data" on page 190
Chapter 11, "Setting the locale," on page 199

Converting alphanumeric, DBCS, and integer to national (MOVE)

You can use a MOVE statement to implicitly convert data to national representation.

You can move the following kinds of data to category national or national-edited data items, and thus convert the data to national representation:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- DBCS
- Integer of USAGE DISPLAY
- Numeric-edited of USAGE DISPLAY

You can likewise move the following kinds of data to numeric-edited data items that have USAGE NATIONAL:

- Alphanumeric
- Display floating-point (floating-point of USAGE DISPLAY)
- Numeric-edited of USAGE DISPLAY
- Integer of USAGE DISPLAY

For complete rules about moves to national data, see the related reference about the MOVE statement.

For example, the MOVE statement below moves the alphanumeric literal "AB" to the national data item UTF16-Data:

```
01  UTF16-Data  Pic N(2) Usage National.
    . . .
    Move "AB" to UTF16-Data
```

After the MOVE statement above, UTF16-Data contains NX'41004200', the national representation of the alphanumeric characters 'AB'.

If padding is required in a receiving data item that has USAGE NATIONAL, the default UTF-16 space character (NX'2000') is used. If truncation is required, it occurs at the boundary of a national-character position.

Related tasks
“Assigning values to elementary data items (MOVE)” on page 28
“Assigning values to group data items (MOVE)” on page 29
“Displaying numeric data” on page 37
“Coding for use of DBCS support” on page 194

Related references
MOVE statement (*COBOL for Linux on x86 Language Reference*)

Converting alphanumeric or DBCS to national (NATIONAL-OF)

Use the NATIONAL-OF intrinsic function to convert alphabetic, alphanumeric, or DBCS data to a national data item. Specify the source code page as the second argument if the source is encoded in a different code page than is in effect for the data item.

“Example: converting to and from national data” on page 186

Related tasks
“Processing UTF-8 data using UTF-16 (national) data types” on page 193
“Processing Chinese GB 18030 data” on page 193
“Processing alphanumeric data items that contain DBCS data” on page 196

Related references
NATIONAL-OF (*COBOL for Linux on x86 Language Reference*)

Converting national to alphanumeric (DISPLAY-OF)

Use the DISPLAY-OF intrinsic function to convert national data to an alphanumeric (USAGE DISPLAY) character string that is represented in a code page that you specify as the second argument.

If you omit the second argument, the output code page is determined from the runtime locale.

If you specify an EBCDIC or ASCII code page that combines single-byte character set (SBCS) and DBCS characters, the returned string might contain a mixture of SBCS and DBCS characters. The DBCS substrings are delimited by shift-in and shift-out characters if the code page in effect for the function is an EBCDIC code page.

"Example: converting to and from national data" on page 186

Related concepts
"The active locale" on page 199

Related tasks
"Processing UTF-8 data using UTF-16 (national) data types" on page 193
"Processing Chinese GB 18030
data" on page 193

Related references
DISPLAY-OF (*COBOL for Linux on x86 Language Reference*)

Overriding the default code page

In some cases, you might need to convert data to or from a code page that differs from the code page that is in effect at run time. To do so, convert the item by using a conversion function in which you explicitly specify the code page.

If you specify a code page as an argument to the DISPLAY-OF intrinsic function, and the code page differs from the code page that is in effect at run time, do not use the function result in any operations that involve implicit conversion (such as an assignment to, or comparison with, a national data item). Such operations assume the runtime code page.

Example: converting to and from national data

The following example shows the NATIONAL-OF and DISPLAY-OF intrinsic functions and the MOVE statement for converting to and from national (UTF-16) data items. It also demonstrates the need for explicit conversions when you operate on strings that are encoded in multiple code pages.

```
* . . .
 01  Data-in-Unicode          pic N(100) usage national.
 01  Data-in-Greek            pic X(100).
 01  other-data-in-US-English pic X(12) value "PRICE in $ =".
* . . .
     Read Greek-file into Data-in-Greek
     Move function National-of(Data-in-Greek, "ISO8859-7")
         to Data-in-Unicode
* . . . process Data-in-Unicode here . . .
     Move function Display-of(Data-in-Unicode, "ISO8859-7")
         to Data-in-Greek
     Write Greek-record from Data-in-Greek
```

The example above works correctly because the input code page is specified. Data-in-Greek is converted as data represented in ISO8859-7 (Ascii Greek). However, the following statement results in an

incorrect conversion unless all the characters in the item happen to be among those that have a common representation in both the Greek and the English code pages:

```
Move Data-in-Greek to Data-in-Unicode
```

Assuming that the locale in effect is en_US.ISO8859-1, the MOVE statement above converts Data-in-Greek to Unicode based on the code page ISO8859-1 to UTF-16LE conversion. This conversion does not produce the expected results because Data-in-Greek is encoded in ISO8859-7.

If you set the locale to el_GR.ISO8859-7 (that is, your program handles ASCII data in Greek), you can code the same example correctly as follows:

```
* . . .
 01  Data-in-Unicode pic N(100) usage national.
 01  Data-in-Greek   pic X(100).
* . . .
     Read Greek-file into Data-in-Greek
* . . . process Data-in-Greek here ...
* . . . or do the following (if need to process data in Unicode):
     Move Data-in-Greek to Data-in-Unicode
* . . . process Data-in-Unicode
     Move function Display-of(Data-in-Unicode) to Data-in-Greek
     Write Greek-record from Data-in-Greek
```

Related tasks
Chapter 11, “Setting the locale,” on page 199

Using national groups

To define a group data item as a national group, code a GROUP-USAGE NATIONAL clause at the group level for the item. The group can contain only data items that explicitly or implicitly have USAGE NATIONAL.

The following data description entry specifies that a level-01 group and its subordinate groups are national group items:

```
01  Nat-Group-1    GROUP-USAGE NATIONAL.
    02  Group-1.
        04  Month    PIC 99.
        04  DayOf    PIC 99.
        04  Year     PIC 9999.
    02  Group-2    GROUP-USAGE NATIONAL.
        04  Amount   PIC 9(4).99  USAGE NATIONAL.
```

In the example above, Nat-Group-1 is a national group, and its subordinate groups Group-1 and Group-2 are also national groups. A GROUP-USAGE NATIONAL clause is implied for Group-1, and USAGE NATIONAL is implied for the subordinate items in Group-1. Month, DayOf, and Year are national decimal items, and Amount is a numeric-edited item that has USAGE NATIONAL.

You can subordinate national groups within alphanumeric groups as in the following example:

```
01  Alpha-Group-1.
    02  Group-1.
        04  Month   PIC 99.
        04  DayOf   PIC 99.
        04  Year    PIC 9999.
    02  Group-2   GROUP-USAGE NATIONAL.
        04  Amount  PIC 9(4).99.
```

In the example above, Alpha-Group-1 and Group-1 are alphanumeric groups; USAGE DISPLAY is implied for the subordinate items in Group-1. (If Alpha-Group-1 specified USAGE NATIONAL at the group level, USAGE NATIONAL would be implied for each of the subordinate items in Group-1. However, Alpha-Group-1 and Group-1 would be alphanumeric groups, not national groups, and would behave

like alphanumeric groups during operations such as moves and compares.) Group-2 is a national group, and USAGE NATIONAL is implied for the numeric-edited item Amount.

You cannot subordinate alphanumeric groups within national groups. All elementary items within a national group must be explicitly or implicitly described as USAGE NATIONAL, and all group items within a national group must be explicitly or implicitly described as GROUP-USAGE NATIONAL.

Related concepts
"National groups" on page 183

Related tasks
"Using national groups as elementary items" on page 188
"Using national groups as group items" on page 188

Related references
GROUP-USAGE clause (*COBOL for Linux on x86 Language Reference*)

Using national groups as elementary items

In most cases, you can use a national group as though it were an elementary data item.

In the following example, a national group item, Group-1, is moved to a national-edited item, Edited-date. Because Group-1 is treated as an elementary data item during the move, editing takes place in the receiving data item. The value in Edited-date after the move is 06/23/2010 in national characters.

```
01  Edited-date  PIC NN/NN/NNNN  USAGE NATIONAL.
01  Group-1    GROUP-USAGE NATIONAL.
    02 Month     PIC 99   VALUE 06.
    02 DayOf     PIC 99   VALUE 23.
    02 Year      PIC 9999 VALUE 2010.
    . . .
    MOVE Group-1 to Edited-date.
```

If Group-1 were instead an alphanumeric group in which each of its subordinate items had USAGE NATIONAL (specified either explicitly with a USAGE NATIONAL clause on each elementary item, or implicitly with a USAGE NATIONAL clause at the group level), a group move, rather than an elementary move, would occur. Neither editing nor conversion would take place during the move. The value in the first eight character positions of Edited-date after the move would be 06232010 in national characters, and the value in the remaining two character positions would be 4 bytes of alphanumeric spaces.

Related tasks
"Assigning values to group data items (MOVE)" on page 29
"Comparing national data and alphanumeric-group operands" on page 192
"Using national groups as group items" on page 188

Related references
MOVE statement (*COBOL for Linux on x86 Language Reference*)

Using national groups as group items

In some cases when you use a national group, it is handled with group semantics; that is, the elementary items in the group are recognized or processed.

In the following example, an INITIALIZE statement that acts upon national group item Group-OneN causes the value 15 in national characters to be moved to only the numeric items in the group:

```
01  Group-OneN    Group-Usage National.
    05  Trans-codeN     Pic N   Value "A".
    05  Part-numberN    Pic NN  Value "XX".
    05  Trans-quanN     Pic 99  Value 10.
    . . .
    Initialize Group-OneN Replacing Numeric Data By 15
```

Because only Trans-quanN in Group-OneN above is numeric, only Trans-quanN receives the value 15. The other subordinate items are unchanged.

The table below summarizes the cases where national groups are processed with group semantics.

Table 18. ***National group items that are processed with group semantics***

Language feature	Uses of national group items	Comment
CORRESPONDING phrase of the ADD, SUBTRACT, or MOVE statement	Specify a national group item for processing as a group in accordance with the rules of the CORRESPONDING phrase.	Elementary items within the national group are processed like elementary items that have USAGE NATIONAL within an alphanumeric group.
INITIALIZE statement	Specify a national group for processing as a group in accordance with the rules of the INITIALIZE statement.	Elementary items within the national group are initialized like elementary items that have USAGE NATIONAL within an alphanumeric group.
Name qualification	Use the name of a national group item to qualify the names of elementary data items and of subordinate group items in the national group.	Follow the same rules for qualification as for an alphanumeric group.
THROUGH phrase of the RENAMES clause	To specify a national group item in the THROUGH phrase, use the same rules as for an alphanumeric group item.	The result is an alphanumeric group item.
FROM phrase of the XML GENERATE statement	Specify a national group item in the FROM phrase for processing as a group in accordance with the rules of the XML GENERATE statement.	Elementary items within the national group are processed like elementary items that have USAGE NATIONAL within an alphanumeric group.

Related tasks
“Initializing a structure (INITIALIZE)” on page 27
“Initializing a table (INITIALIZE)” on page 65
“Assigning values to elementary data items (MOVE)” on page 28
“Assigning values to group data items (MOVE)” on page 29
“Finding the length of data items” on page 109
“Generating XML output” on page 359

Related references
Qualification (*COBOL for Linux on x86 Language Reference*)
RENAMES clause (*COBOL for Linux on x86 Language Reference*)

Storage of character data

Use the table below to compare alphanumeric (DISPLAY), DBCS (DISPLAY-1), and Unicode (NATIONAL) encoding and to plan storage usage.

Table 19. ***Encoding and size of alphanumeric, DBCS, and national data***

Characteristic	DISPLAY	DISPLAY-1	NATIONAL
Character encoding unit	1 byte	2 bytes	2 bytes
Code page	ASCII, EUC, UTF-8, or EBCDIC[3]	ASCII DBCS or EBCDIC DBCS[3]	UTF-16LE[1]
Encoding units per graphic character	1	1	1 or 2[2]
Bytes per graphic character	1 byte	2 bytes	2 or 4 bytes

1. National literals in your source program are converted to UTF-16 for use at run time.
2. Most characters are represented in UTF-16 using one encoding unit. In particular, the following characters are represented using a single UTF-16 encoding unit per character:
   - COBOL characters A-Z, a-z, 0-9, space, + - * / = $, ; . " ( ) > < :'
   - All characters that are converted from an EBCDIC, ASCII, or EUC code page
3. Depending on the locale, the CHAR(NATIVE) or CHAR(EBCDIC) option, and the EBCDIC_CODEPAGE environment variable settings

Related concepts
"Unicode and the encoding of language characters" on page 176

Related tasks
"Specifying the code page for character data" on page 200

Related references
"CHAR" on page 253

Comparing national (UTF-16) data

You can compare national (UTF-16) data, that is, national literals and data items that have USAGE NATIONAL (whether of class national or class numeric), explicitly or implicitly with other kinds of data in relation conditions.

You can code conditional expressions that use national data in the following statements:

- EVALUATE
- IF
- INSPECT
- PERFORM
- SEARCH
- STRING
- UNSTRING

For full details about comparing national data items to other data items, see the Related references.

Related tasks
"Comparing two class national operands" on page 191
"Comparing class national

and class numeric operands” on page 191
“Comparing national numeric
and other numeric operands” on page 192
“Comparing national and other character-string operands” on page 192
“Comparing national data
and alphanumeric-group operands” on page 192

Related references
Relation conditions (*COBOL for Linux on x86 Language Reference*)
General relation conditions (*COBOL for Linux on x86 Language Reference*)
National comparisons (*COBOL for Linux on x86 Language Reference*)
Group comparisons (*COBOL for Linux on x86 Language Reference*)

Comparing two class national operands

You can compare the character values of two operands of class national.

Either operand (or both) can be any of the following types of items:

- A national group
- An elementary category national or national-edited data item
- A numeric-edited data item that has `USAGE NATIONAL`

One of the operands can instead be a national literal or a national intrinsic function.

Use the `NCOLLSEQ` compiler option to determine which type of comparison to perform:

`NCOLLSEQ(BINARY)`
When you compare two class national operands of the same length, they are determined to be equal if all pairs of the corresponding characters are equal. Otherwise, comparison of the binary values of the first pair of unequal characters determines the operand with the larger binary value.

When you compare operands of unequal lengths, the shorter operand is treated as if it were padded on the right with default UTF-16 space characters (`NX'2000'`) to the length of the longer operand.

`NCOLLSEQ(LOCALE)`
When you use a locale-based comparison, the operands are compared by using the algorithm for collation order that is associated with the locale in effect. Trailing spaces are truncated from the operands, except that an operand that consists of all spaces is truncated to a single space.

When you compare operands of unequal lengths, the shorter operand is not extended with spaces because such an extension could alter the expected results for the locale.

The `PROGRAM COLLATING SEQUENCE` clause does not affect the comparison of two class national operands.

Related concepts
“National groups” on page 183

Related tasks
“Using national groups” on page 187

Related references
“NCOLLSEQ” on page 271
National comparisons (*COBOL for Linux on x86 Language Reference*)

Comparing class national and class numeric operands

You can compare national literals or class national data items to integer literals or numeric data items that are defined as integer (that is, national decimal items or zoned decimal items). At most one of the operands can be a literal.

You can also compare national literals or class national data items to floating-point data items (that is, display floating-point or national floating-point items).

Numeric operands are converted to national (UTF-16) representation if they are not already in national representation. A comparison is made of the national character values of the operands.

Related references
General relation conditions (*COBOL for Linux on x86 Language Reference*)

Comparing national numeric and other numeric operands

National numeric operands (national decimal and national floating-point operands) are data items of class numeric that have `USAGE NATIONAL`.

You can compare the algebraic values of numeric operands regardless of their USAGE. Thus you can compare a national decimal item or a national floating-point item with a binary item, an internal-decimal item, a zoned decimal item, a display floating-point item, or any other numeric item.

Related tasks
"Defining national numeric data items" on page 183

Related references
General relation conditions (*COBOL for Linux on x86 Language Reference*)

Comparing national and other character-string operands

You can compare the character value of a national literal or class national data item with the character value of any of the following other character-string operands: alphabetic, alphanumeric, alphanumeric-edited, DBCS, or numeric-edited of `USAGE DISPLAY`.

These operands are treated as if they were moved to an elementary national data item. The characters are converted to national (UTF-16) representation, and the comparison proceeds with two national character operands.

Related tasks
"Using national-character figurative constants" on page 182
"Comparing DBCS literals" on page 196

Related references
National comparisons (*COBOL for Linux on x86 Language Reference*)

Comparing national data and alphanumeric-group operands

You can compare a national literal, a national group item, or any elementary data item that has `USAGE NATIONAL` to an alphanumeric group.

Neither operand is converted. The national operand is treated as if it were moved to an alphanumeric group item of the same size in bytes as the national operand, and the two groups are compared. An alphanumeric comparison is done regardless of the representation of the subordinate items in the alphanumeric group operand.

For example, Group-XN is an alphanumeric group that consists of two subordinate items that have USAGE NATIONAL:

```
01  Group-XN.
    02 TransCode PIC NN   Value "AB"  Usage National.
    02 Quantity  PIC 999  Value 123   Usage National.
    . . .
    If N"AB123" = Group-XN  Then Display "EQUAL"
    Else Display "NOT EQUAL".
```

When the IF statement above is executed, the 10 bytes of the national literal N"AB123" are compared byte by byte to the content of Group-XN. The items compare equally, and "EQUAL" is displayed.

Related references
Group comparisons (*COBOL for Linux on x86 Language Reference*)

Processing UTF-8 data using UTF-16 (national) data types

To process UTF-8 data, first convert the UTF-8 data to UTF-16 in a national data item. After processing the national data, convert it back to UTF-8 for output. For the conversions, use the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively. Use code page 1208 for UTF-8 data.

As an alternative to the recommended method of processing UTF-8 data using

USAGE UTF-8

data items, you can also process UTF-8 data by storing it in alphanumeric data items and then converting it to UTF-16 in a national data item.

Take the following steps to convert ASCII or EBCDIC data to UTF-8 (unless the code page of the locale in effect is UTF-8, in which case the native alphanumeric data is already encoded in UTF-8):

1. Use the function NATIONAL-OF to convert the ASCII or EBCDIC string to a national (UTF-16) string.
2. Use the function DISPLAY-OF to convert the national string to UTF-8.

The following example converts Greek EBCDIC data to UTF-8:

```
01 Greek-EBCDIC pic X(10) value "αβγδεζηθ".
01 UnicodeString pic N(10).
01 UTF-8-String pic X(20).
    Move function National-of(Greek-EBCDIC, 00875) to UnicodeString
    Move function Display-of(UnicodeString, 01208) to UTF-8-String
```

Usage note: Use care if you use reference modification to refer to data encoded in UTF-8. UTF-8 characters are encoded with a varying number of bytes per character. Avoid operations that might split a multibyte character.

Related tasks
“Referring to substrings
of data items” on page 99
“Converting to or from national (Unicode) representation” on page 184
“Parsing XML documents
encoded in UTF-8” on page 349

Processing Chinese GB 18030 data

GB 18030 is a national-character standard specified by the government of the People's Republic of China.

COBOL for Linux supports GB 18030. If the code page specified for the locale in effect is GB18030 (a code page that supports GB 18030), USAGE DISPLAY data items that contain GB 18030 characters encoded in GB18030 can be processed in a program. GB 18030 characters take 1 to 4 bytes each. Therefore the program logic must be sensitive to the multibyte nature of the data.

You can process GB 18030 characters in these ways:

- Use national data items to define and process GB 18030 characters that are represented in UTF-16, CCSID 01200.
- Process data in any code page (including GB18030, which has CCSID 1392) by converting the data to UTF-16, processing the UTF-16 data, and then converting the data back to the original code-page representation.

When you need to process Chinese GB 18030 data that requires conversion, first convert the input data to UTF-16 in a national data item. After you process the national data item, convert it back to Chinese GB 18030 for output. For the conversions, use the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively, and specify GB18030 or 1392 as the second argument of each function.

The following example illustrates these conversions:

```
 01 Chinese-ASCII pic X(16) value "奥林匹克运动会".
 01 Chinese-GB18030-String pic X(16).
 01 UnicodeString pic N(14).
 . . .
     Move function National-of(Chinese-ASCII, 1392) to UnicodeString
*  Process data in Unicode
     Move function Display-of(UnicodeString, 1392) to Chinese-GB18030-String
```

Related tasks
“Converting to or from national (Unicode) representation” on page 184
“Coding for use of DBCS support” on page 194

Related references
“Storage of character data” on page 190

Coding for use of DBCS support

IBM COBOL for Linux on x86 supports using applications in any of many national languages, including languages that use double-byte character sets (DBCS).

The following list summarizes the support for DBCS:

- DBCS characters in user-defined words (multibyte names)
- DBCS characters in comments
- DBCS data items (defined with PICTURE N, G, or G and B)
- DBCS literals
- Collating sequence
- SOSI compiler option
- DBCS_CODEPAGE environment variable

Related tasks
“Defining DBCS data” on page 195
“Using DBCS literals” on page 195
“Testing for valid DBCS characters” on page 196
“Processing alphanumeric data items that contain DBCS data” on page 196
Chapter 11, “Setting the locale,” on page 199
“Controlling the collating sequence with a locale” on page 205

Related references
“SOSI” on page 277

Defining DBCS data

Use the PICTURE and USAGE clauses to define DBCS data items. DBCS data items can use PICTURE symbols G, G and B, or N. Each DBCS character position is 2 bytes in length.

You can specify a DBCS data item by using the USAGE DISPLAY-1 clause. When you use PICTURE symbol G, you must specify USAGE DISPLAY-1. When you use PICTURE symbol N but omit the USAGE clause, USAGE DISPLAY-1 or USAGE NATIONAL is implied depending on the setting of the NSYMBOL compiler option.

If you use a VALUE clause with the USAGE clause in the definition of a DBCS item, you must specify a DBCS literal or the figurative constant SPACE or SPACES.

If a data item has USAGE DISPLAY-1 (either explicitly or implicitly), the selected locale must indicate a code page that includes DBCS characters. If the code page of the locale does not include DBCS characters, such data items are flagged as errors.

For the purpose of handling reference modifications, each character in a DBCS data item is considered to occupy the number of bytes that corresponds to the code-page width (that is, 2).

Related tasks
Chapter 11, “Setting the locale,” on page 199

Related references
“Locales and code pages that are supported” on page 202
“NSYMBOL” on page 272

Using DBCS literals

You can use the prefix N or G to represent a DBCS literal.

That is, you can specify a DBCS literal in either of these ways:

- N'*dbcs characters*' (provided that the compiler option NSYMBOL(DBCS) is in effect)
- G'*dbcs characters*'

You can use quotation marks (") or apostrophes (') as the delimiters of a DBCS literal irrespective of the setting of the APOST or QUOTE compiler option. You must code the same opening and closing delimiter for a DBCS literal.

If the SOSI compiler option is in effect, the shift-out (SO) control character X'1E' must immediately follow the opening delimiter, and the shift-in (SI) control character X'1F' must immediately precede the closing delimiter.

In addition to DBCS literals, you can use alphanumeric literals to specify any character in one of the supported code pages. However, if the SOSI compiler option is in effect, any string of DBCS characters that is within an alphanumeric literal must be delimited by the SO and SI characters.

You cannot continue an alphanumeric literal that contains multibyte characters. The length of a DBCS literal is likewise limited by the available space in Area B on a single source line. The maximum length of a DBCS literal is thus 28 double-byte characters.

An alphanumeric literal that contains multibyte characters is processed byte by byte, that is, with semantics appropriate for single-byte characters, except when it is converted explicitly or implicitly to national data representation, as for example in an assignment to or comparison with a national data item.

Related tasks
“Comparing DBCS literals” on page 196
“Using figurative constants” on page 22

Related references
“NSYMBOL” on page 272
“SOSI” on page 277
DBCS literals (*COBOL for Linux on x86 Language Reference*)

Comparing DBCS literals

Comparisons of DBCS literals are based on the compile-time locale. Therefore, do not use DBCS literals within a statement that expresses an implied relational condition between two DBCS literals (such as `VALUE G'`*literal-1*`' THRU G'`*literal-2*`'`) unless the intended runtime locale is the same as the compile-time locale.

Related tasks
“Comparing national (UTF-16) data” on page 190
Chapter 11, “Setting the locale,” on page 199

Related references
“COLLSEQ” on page 256
DBCS literals (*COBOL for Linux on x86 Language Reference*)
DBCS comparisons (*COBOL for Linux on x86 Language Reference*)

Testing for valid DBCS characters

The Kanji class test tests for valid Japanese graphic characters. This testing includes Katakana, Hiragana, Roman, and Kanji character sets.

Kanji and DBCS class tests are defined to be consistent with their IBM Z definitions. Both class tests are performed internally by converting the double-byte characters to the double-byte characters defined for z/OS. The converted double-byte characters are tested for DBCS and Japanese graphic characters.

The Kanji class test is done by checking the converted characters for the range `X'41'` through `X'7E'` in the first byte and `X'41'` through `X'FE'` in the second byte, plus the space character `X'4040'`.

The DBCS class test tests for valid graphic characters for the code page.

The DBCS class test is done by checking the converted characters for the range `X'41'` through `X'FE'` in both the first and second byte of each character, plus the space character `X'4040'`.

Related tasks
“Coding conditional expressions” on page 85

Related references
Class condition (*COBOL for Linux on x86 Language Reference*)

Processing alphanumeric data items that contain DBCS data

If you use byte-oriented operations (for example, `STRING`, `UNSTRING`, or reference modification) on an alphanumeric data item that contains DBCS characters, results are unpredictable. You should instead convert the item to a national data item before you process it.

That is, do these steps:

1. Convert the item to UTF-16 in a national data item by using a `MOVE` statement or the `NATIONAL-OF` intrinsic function.
2. Process the national data item as needed.
3. Convert the result back to an alphanumeric data item by using the `DISPLAY-OF` intrinsic function.

Related tasks
“Joining data items (STRING)” on page 93
“Splitting data items (UNSTRING)” on page 95

“Referring to substrings
of data items” on page 99
“Converting to or from national (Unicode) representation” on page 184

Chapter 11. Setting the locale

You can write applications to reflect the cultural conventions of the locale that is in effect when the applications are run. Cultural conventions include sort order, character classification, and national language; and formats of dates and times, numbers, monetary units, postal addresses, and telephone numbers.

With COBOL for Linux, you can select the appropriate code pages and collating sequences, and you can use language elements and compiler options to handle Unicode, single-byte character sets, and double-byte character sets (DBCS).

Related concepts
"The active locale" on page 199

Related tasks
"Specifying the code page for character data" on page 200
"Using environment variables to specify a locale" on page 201
"Controlling the collating sequence with a locale" on page 205
"Accessing the active locale and code-page values" on page 208

The active locale

A *locale* is a collection of data that encodes information about a cultural environment. The active locale is the locale that is in effect when you compile or run your program. You can establish a cultural environment for an application by specifying the active locale.

Only one locale can be active at a time.

The active locale affects the behavior of these culturally sensitive interfaces for the entire program:

- Code pages used for character data
- Messages
- Collating sequence
- Date and time formats
- Character classification and case conversion

The active locale does not affect the following items, for which 85 COBOL Standard defines specific language and behavior:

- Decimal point and grouping separators
- Currency sign

The active locale determines the code page for compiling and running programs:

- The code page that is used for compilation is based on the locale setting at compile time.
- The code page that is used for running an application is based on the locale setting at run time.

The evaluation of literal values in the source program is handled with the locale that is active at compile time. For example, the conversion of national literals from the source representation to UTF-16 for running the program uses the compile-time locale.

COBOL for Linux determines the setting of the active locale from a combination of the applicable environment variables and system settings. Environment variables are used first. If an applicable locale category is not defined by environment variables, COBOL uses defaults and system settings.

Related concepts
"Determination of the locale from system settings" on page 202

© Copyright IBM Corp. 2021

Related tasks
"Specifying the code page for character data" on page 200
"Using environment variables to specify a locale" on page 201
"Controlling the collating
sequence with a locale" on page 205

Related references
"Types of messages for which translations are available" on page 202

Specifying the code page for character data

In a source program, you can use the characters that are represented in a supported code page in COBOL names, literals, and comments. At run time, you can use the characters that are represented in a supported code page in data items described with `USAGE DISPLAY`, `USAGE DISPLAY-1`, or `USAGE NATIONAL`.

The code page that is in effect for a particular data item depends on the following aspects:

- Which USAGE clause you used
- Whether you used the `NATIVE` phrase with the USAGE clause
- Whether you used the `CHAR(NATIVE)` or `CHAR(EBCDIC)` compiler option
- The value of the EBCDIC_CODEPAGE environment variable
- The value of the DBCS_CODEPAGE environment variable
- Which locale is active

For `USAGE NATIONAL` data items, the code page defaults to UTF-16 in little-endian format.

For `USAGE DISPLAY` data items, COBOL for Linux chooses between ASCII, UTF-8, EUC, and EBCDIC code pages as follows:

- Data items that are described with the `NATIVE` phrase in the USAGE clause or that are compiled with the `CHAR(NATIVE)` option in effect are encoded in an ASCII, EUC, or UTF-8 code page.
- Data items that are described without the `NATIVE` phrase in the USAGE clause and that are compiled with the `CHAR(EBCDIC)` option in effect are encoded in an EBCDIC code page.

For `USAGE DISPLAY-1` data items, COBOL for Linux chooses between ASCII and EBCDIC code pages as follows:

- Data items that are described with the `NATIVE` phrase in the USAGE clause or that are compiled with the `CHAR(NATIVE)` option in effect are encoded in an ASCII DBCS code page.
- Data items that are described without the `NATIVE` phrase in the USAGE clause and that are compiled with the `CHAR(EBCDIC)` option in effect are encoded in an EBCDIC DBCS code page.

COBOL determines the appropriate code page as follows:

ASCII, UTF-8, EUC
From the active locale at run time

ASCII DBCS
From the DBCS_CODEPAGE environment variable, if set, otherwise the default DBCS code page from the current local setting

EBCDIC
From the EBCDIC_CODEPAGE environment variable, if set, otherwise the default EBCDIC code page from the current locale setting

Related tasks
"Using environment variables to specify a locale" on page 201

Related references
"Locales and code pages that are supported" on page 202
"Runtime environment

variables” on page 218
“CHAR” on page 253
COBOL words with single-byte characters
(*COBOL for Linux on x86 Language Reference*)
User-defined words with multibyte characters
(*COBOL for Linux on x86 Language Reference*)

Using environment variables to specify a locale

Use any of several environment variables to provide the locale information for a COBOL program.

To specify a code page to use for all of the locale categories (messages, collating sequence, date and time formats, character classification, and case conversion), use LC_ALL.

To set the value for a specific locale category, use the appropriate environment variable:

- Use LC_MESSAGES to specify the format for affirmative and negative responses. You can use it also to affect whether messages (for example, error messages and listing headers) are in US English or Japanese. For any locale other than Japanese, US English is used.
- Use LC_COLLATE to specify the collating sequence in effect for greater-than or less-than comparisons, such as in relation conditions or in the SORT and MERGE statements.
- Use LC_TIME to specify the format of the date and time shown in compiler listings. All other date and time values are controlled through COBOL language syntax.
- Use LC_CTYPE to specify character classification, case conversion, and other character attributes.

Any locale category that has not been specified by one of the locale environment variables above is set from the value of the LANG environment variable.

To set the locale environment variables, use a command of the following format (`.`*`codepageID`* is optional):

```
export LC_xxxx=ll_CC.codepageID
```

Here LC_*xxxx* is the name of the locale category, *ll* is a lowercase two-letter language code, *CC* is an uppercase two-letter ISO country code, and *codepageID* is the code page to be used for native DISPLAY and DISPLAY-1 data. COBOL for Linux uses the POSIX-defined locale conventions.

For example, to set the locale to Canadian French encoded in ISO 8859-1, issue this command in the command window from which you compile and run a COBOL application:

```
export LC_ALL=fr_CA.iso88591
```

You must code a valid value for the locale name (*ll_CC*), and the code page (*codepageID*) that you specify must be valid for the locale name. Valid values are shown in the table of supported locales and code pages referenced below.

Related concepts
“Determination of the
locale from system settings” on page 202

Related tasks
“Specifying the code page for character data” on page 200

Related references
“Locales and code pages that are supported” on page 202
“Compiler and runtime
environment variables” on page 214

Determination of the locale from system settings

If COBOL for Linux cannot determine the value of an applicable locale category from the environment variables, it uses default settings.

When the language and country codes are determined from environment variables, but the code page is not, COBOL for Linux uses the default system code page for the language and country-code combination.

Multiple code pages might apply to the language and country-code combination. If you do not want the Linux system to select a default code page, you must specify the code page explicitly.

UTF-8: UTF-8 encoding is supported for any language and country-code combination.

Related tasks
"Specifying the code page for character data" on page 200
"Using environment variables to specify a locale" on page 201
"Accessing the active locale and code-page values" on page 208
"Setting environment variables" on page 213

Related references
"Locales and code pages that are supported" on page 202
"Compiler and runtime environment variables" on page 214

Types of messages for which translations are available

The following messages are enabled for national language support: compiler and runtime messages, and listing headers (including locale-based date and time formats).

Appropriate text and formats, as specified in the active locale, are used for these messages and the listing headers.

See the related reference below for information about the LANG and NLSPATH environment variables, which affect the language and locale of messages.

Related concepts
"The active locale" on page 199

Related tasks
"Using environment variables to specify a locale" on page 201

Related references
"Compiler and runtime environment variables" on page 214

Locales and code pages that are supported

The following table shows the locales that could be available on your system, and the code pages that are supported for each locale. COBOL for Linux supports the locales that are available on the system at compile time and at run time.

To query the available locales on the system, enter the following:

```
locale -a
```

Table 20. ***Supported locales and code pages***

Locale name[1]	Language[2]	Country or region[3]	ASCII-based code pages[4]	EBCDIC code pages[5]	Language group
any			utf-8	EBCDIC code pages that are applicable to the locale are based on the *language* and *COUNTRY* portions of the locale name regardless of the code-page value of the locale.	
ar_AE	Arabic	United Arab Emirates	iso88596	IBM-16804, IBM-420	Arabic
be_BY	Byelorussian	Belarus	iso88595	IBM-1025, IBM-1154	Latin 5
bg_BG	Bulgarian	Bulgaria	iso88595	IBM-1025, IBM-1154	Latin 5
ca_ES	Catalan	Spain	iso88591	IBM-285, IBM-1145	Latin 1
cs_CZ	Czech	Czech Republic	iso88592	IBM-870, IBM-1153	Latin 2
da_DK	Danish	Denmark	iso88591	IBM-277, IBM-1142	Latin 1
de_CH	German	Switzerland	iso88591	IBM-500, IBM-1148	Latin 1
de_DE	German	Germany	iso88591	IBM-273, IBM-1141	Latin 1
el_GR	Greek	Greece	iso88597	IBM-4971, IBM-875	Greek
en_AU	English	Australia	iso88591	IBM-037, IBM-1140	Latin 1
en_GB	English	United Kingdom	iso88591	IBM-037, IBM-1140	Latin 1
en_US	English	United States	iso88591	IBM-037, IBM-1140	Latin 1
en_ZA	English	South Africa	iso88591	IBM-037, IBM-1140	Latin 1
es_ES	Spanish	Spain	iso88591	IBM-284, IBM-1145	Latin 1
fi_FI	Finnish	Finland	iso88591	IBM-278, IBM-1143	Latin 1
fr_BE	French	Belgium	iso88591	IBM-297, IBM-1148	Latin 1
fr_CA	French	Canada	iso88591	IBM-037, IBM-1140	Latin 1
fr_CH	French	Switzerland	iso88591	IBM-500, IBM-1148	Latin 1
fr_FR	French	France	iso88591	IBM-297, IBM-1148	Latin 1
hr_HR	Croatian	Croatia	iso88592	IBM-870, IBM-1153	Latin 2
hu_HU	Hungarian	Hungary	iso88592	IBM-870, IBM-1153	Latin 2
is_IS	Icelandic	Iceland	iso88591	IBM-871, IBM-1149	Latin 1
it_CH	Italian	Switzerland	iso88591	IBM-500, IBM-1148	Latin 1
it_IT	Italian	Italy	iso88591	IBM-280, IBM-1144	Latin 1
iw_IL	Hebrew	Israel	iso88598	IBM-12712, IBM-424	Hebrew
ja_JP	Japanese	Japan	IBMeucjp	IBM-930, IBM-939, IBM-1390, IBM-1399	Ideographic languages

Table 20. ***Supported locales and code pages*** *(continued)*

Locale name[1]	Language[2]	Country or region[3]	ASCII-based code pages[4]	EBCDIC code pages[5]	Language group
ko_KR	Korean	Korea, Republic of	euckr	IBM-933, IBM-1364	Ideographic languages
lt_LT	Lithuanian	Lithuania	IBMiso885913	n/a	Lithuanian
lv_LV	Latvian	Latvia	IBMiso885913	n/a	Latvian
mk_MK	Macedonian	Macedonia	iso88595	IBM-1025, IBM-1154	Latin 5
nl_BE	Dutch	Belgium	iso8859-1	IBM-500, IBM-1148	Latin 1
nl_NL	Dutch	Netherlands	iso88591	IBM-037, IBM-1140	Latin 1
no_NO	Norwegian	Norway	iso88591	IBM-277, IBM-1142	Latin 1
pl_PL	Polish	Poland	iso88592	IBM-870, IBM-1153	Latin 2
pt_BR	Portuguese	Brazil	iso88591	IBM-037, IBM-1140	Latin 1
pt_PT	Portuguese	Portugal	iso88591	IBM-037, IBM-1140	Latin 1
ro_RO	Romanian	Romania	iso88592	IBM-870, IBM-1153	Latin 2
ru_RU	Russian	Russian federation	iso88595	IBM-1025, IBM-1154	Latin 5
sk_SK	Slovak	Slovakia	iso88592	IBM-870, IBM-1153	Latin 2
sl_SI	Slovenian	Slovenia	iso8859-2	IBM-870, IBM-1153	Latin 2
sq_AL	Albanian	Albania	iso88591	IBM-500, IBM-1148	Latin 1
sv_SE	Swedish	Sweden	iso88591	IBM-278, IBM-1143	Latin 1
th_TH	Thai	Thailand	tis620	IBM-9030	Thai
tr_TR	Turkish	Turkey	iso88599	IBM-1026, IBM-1155	Turkish
uk_UA	Ukranian	Ukraine	iso88595	IBM-1123, IBM-1154	Latin 5
zh_CN	Chinese	China	gb18030	IBM-1388	Ideographic languages
zh_TW	Chinese (traditional)	Taiwan	IBMeuctw	IBM-1371, IBM-937	Ideographic languages

*Table 20. **Supported locales and code pages** (continued)*

Locale name[1]	Language[2]	Country or region[3]	ASCII-based code pages[4]	EBCDIC code pages[5]	Language group

1. Shows the valid combinations of ISO language code and ISO country code (*language_COUNTRY*) that are supported. The case of each character in the locale name shown in the table is significant and might not reflect the casing of a locale name with a specific code page selected (or implied) for the locale. See the results of the "locale -a" command for proper casing of each character for the locale name selected.
2. Shows the associated language.
3. Shows the associated country or region.
4. Shows the code pages that are valid as the code-page ID for the locale that has the corresponding *language_COUNTRY* value. These table entries are not definitive. For the current list of valid locales, consult your system documentation for the specific version and configuration of Linux that you are running. The locale that you select must be valid, that is, installed both where you develop and where you run the program.
5. Shows the code pages that are valid as the code-page ID for the locale that has the corresponding *language_COUNTRY* value. These code pages are valid as content for the EBCDIC_CODEPAGE environment variable. If the EBCDIC_CODEPAGE environment variable is not set, the rightmost code-page entry shown in this column is selected as the EBCDIC code page for the corresponding locale.

Related tasks
"Specifying the code page for character data" on page 200
"Using environment variables to specify a locale" on page 201

Controlling the collating sequence with a locale

Various operations such as comparisons, sorting, and merging use the collating sequence that is in effect for the program and data items. How you control the collating sequence depends on the code page in effect for the class of the data: alphabetic, alphanumeric, DBCS, or national.

A locale-based collating sequence for items that are class alphabetic, alphanumeric, or DBCS applies only when the `COLLSEQ(LOCALE)` compiler option is in effect, not when `COLLSEQ(BIN)` or `COLLSEQ(EBCDIC)` is in effect. Similarly, a locale-based collating sequence for class national items applies only when the `NCOLLSEQ(LOCALE)` compiler option is in effect, not when `NCOLLSEQ(BIN)` is in effect.

If the `COLLSEQ(LOCALE)` or `NCOLLSEQ(LOCALE)` compiler option is in effect, the compile-time locale is used for language elements that have syntax or semantic rules that are affected by locale-based collation order, such as:

- `THRU` phrase in a condition-name `VALUE` clause
- *literal-3* `THRU` *literal-4* phrase in the `EVALUATE` statement
- *literal-1* `THRU` *literal-2* phrase in the `ALPHABET` clause
- Ordinal positions of characters specified in the `SYMBOLIC CHARACTERS` clause
- `THRU` phrase in the `CLASS` clause

If the `COLLSEQ(LOCALE)` compiler option is in effect, the collating sequence for alphanumeric keys in `SORT` and `MERGE` statements is always based on the runtime locale.

Related tasks
"Specifying the collating sequence" on page 6
"Setting sort or merge criteria" on page 156
"Specifying the code page for character data" on page 200
"Using environment variables to specify a locale" on page 201
"Controlling the alphanumeric collating sequence with a locale" on page 206

“Controlling the DBCS collating sequence with a locale” on page 207
“Controlling the national collating sequence with a locale” on page 207
“Accessing the active locale and code-page values” on page 208

Related references
“Locales and code pages that are supported” on page 202
“COLLSEQ” on page 256
“NCOLLSEQ” on page 271

Controlling the alphanumeric collating sequence with a locale

The collating sequence for single-byte alphanumeric characters for the program collating sequence is based on either the locale at compile time or the locale at run time.

If you specify PROGRAM COLLATING SEQUENCE in the source program, the collating sequence is set at compile time and is used regardless of the locale at run time. If instead you set the collating sequence by using the COLLSEQ compiler option, the locale at run time takes precedence.

If the code page in effect is a single-byte ASCII code page, you can specify the following clauses in the SPECIAL-NAMES paragraph:

- ALPHABET clause
- SYMBOLIC CHARACTERS clause
- CLASS clause

If you specify these clauses when the source code page in effect includes DBCS characters, the clauses will be diagnosed and treated as comments. The rules of the COBOL user-defined alphabet-name and symbolic characters assume a character-by-character collating sequence, not a collating sequence that depends on a sequence of multiple characters.

If you specify the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph, the collating sequence that is associated with the *alphabet-name* is used to determine the truth value of alphanumeric comparisons. The PROGRAM COLLATING SEQUENCE clause also applies to sort and merge keys of USAGE DISPLAY unless you specify the COLLATING SEQUENCE phrase in the SORT or MERGE statement.

If you do not specify the COLLATING SEQUENCE phrase or the PROGRAM COLLATING SEQUENCE clause, the collating sequence in effect is NATIVE by default, and it is based on the active locale setting. This setting applies to SORT and MERGE statements and to the program collating sequence.

The collating sequence affects the processing of the following items:

- ALPHABET clause (for example, *literal-1* THRU *literal-2*)
- SYMBOLIC CHARACTERS specifications
- VALUE range specifications for level-88 items, relation conditions, and SORT and MERGE statements

Related tasks
“Specifying the collating sequence” on page 6
“Controlling the collating sequence with a locale” on page 205
“Controlling the DBCS collating sequence with a locale” on page 207
“Controlling the national collating sequence with a locale” on page 207
“Setting sort or merge criteria” on page 156

Related references
“COLLSEQ” on page 256
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
Alphanumeric comparisons (*COBOL for Linux on x86 Language Reference*)

Controlling the DBCS collating sequence with a locale

The locale-based collating sequence at run time always applies to DBCS data, except for comparisons of literals.

You can use a data item or literal of class DBCS in a relation condition with any relational operator. The other operand must be of class DBCS or class national, or be an alphanumeric group. No distinction is made between DBCS items and edited DBCS items.

When you compare two DBCS operands, the collating sequence is determined by the active locale if the `COLLSEQ(LOCALE)` compiler option is in effect. Otherwise, the collating sequence is determined by the binary values of the DBCS characters. The `PROGRAM COLLATING SEQUENCE` clause has no effect on comparisons that involve data items or literals of class DBCS.

When you compare a DBCS item to a national item, the DBCS operand is treated as if it were moved to an elementary national item of the same length as the DBCS operand. The DBCS characters are converted to national representation, and the comparison proceeds with two national character operands.

When you compare a DBCS item to an alphanumeric group, no conversion or editing is done. The comparison proceeds as for two alphanumeric character operands. The comparison operates on bytes of data without regard to data representation.

Related tasks
"Specifying the collating sequence" on page 6
"Using DBCS literals" on page 195
"Controlling the collating sequence with a locale" on page 205
"Controlling the alphanumeric collating sequence with a locale" on page 206
"Controlling the national collating sequence with a locale" on page 207

Related references
"COLLSEQ" on page 256
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
Alphanumeric comparisons (*COBOL for Linux on x86 Language Reference*)
DBCS comparisons (*COBOL for Linux on x86 Language Reference*)
Group comparisons (*COBOL for Linux on x86 Language Reference*)

Controlling the national collating sequence with a locale

You can use national literals or data items of `USAGE NATIONAL` in a relation condition with any relational operator. The `PROGRAM COLLATING SEQUENCE` clause has no effect on comparisons that involve national operands.

Use the `NCOLLSEQ(LOCALE)` compiler option to effect comparisons based on the algorithm for collation order that is associated with the active locale at run time. If `NCOLLSEQ(BINARY)` is in effect, the collating sequence is determined by the binary values of the national characters.

Keys used in a `SORT` or `MERGE` statement can be class national only if the `NCOLLSEQ(BIN)` option is in effect.

Related tasks
"Comparing national (UTF-16) data" on page 190
"Controlling the collating sequence with a locale" on page 205
"Controlling the DBCS collating sequence with a locale" on page 207
"Setting sort or merge criteria" on page 156

Related references
"NCOLLSEQ" on page 271
Classes and categories of data (*COBOL for Linux on x86 Language Reference*)
National comparisons (*COBOL for Linux on x86 Language Reference*)

Intrinsic functions that depend on collating sequence

The following intrinsic functions depend on the ordinal positions of characters.

For an ASCII code page, these intrinsic functions are supported based on the collating sequence in effect. For an EUC code page or a code page that includes DBCS characters, the ordinal positions of single-byte characters are assumed to correspond to the hexadecimal representations of the single-byte characters. For example, the ordinal position for 'A' is 66 (X'41' + 1) and the ordinal position for '*' is 43 (X'2A' + 1).

Table 21. ***Intrinsic functions that depend on collating sequence***

Intrinsic function	Returns:	Comments
CHAR	Character that corresponds to the ordinal-position argument	
MAX	Content of the argument that contains the maximum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]
MIN	Content of the argument that contains the minimum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]
ORD	Ordinal position of the character argument	
ORD-MAX	Integer ordinal position in the argument list of the argument that contains the maximum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]
ORD-MIN	Integer ordinal position in the argument list of the argument that contains the minimum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]

1. Code page and collating sequence are not applicable when the function has numeric arguments.

These intrinsic functions are not supported for the DBCS data type.

Related tasks
"Specifying the collating sequence" on page 6
"Comparing national (UTF-16) data" on page 190
"Controlling the collating sequence with a locale" on page 205

Accessing the active locale and code-page values

To verify the locale that is in effect at compile time, check the last few lines of the compiler listing.

For some applications, you might want to verify the locale and the EBCDIC code page that are active at run time, and convert a code-page ID to the corresponding CCSID. You can use callable library routines to perform these queries and conversions.

To access the locale and the EBCDIC code page that are active at run time, call the library function _iwzGetLocaleCP as follows:

```
CALL "_iwzGetLocaleCP" USING output1, output2
```

The variable *output1* is an alphanumeric item of 20 characters that represents the null-terminated locale value in the following format:

- Two-character language code
- An underscore (_)
- Two-character country code
- A period (.)
- The code-page value for the locale

For example, en_US.IBM-1252 is the locale value of language code en, country code US, and code page IBM-1252.

The variable *output2* is an alphanumeric item of 10 characters that represents the null-terminated EBCDIC code-page ID in effect, such as IBM-1140.

To convert a code-page ID to the corresponding CCSID, call the library function _iwzGetCCSID as follows:

```
CALL "_iwzGetCCSID" USING input, output RETURNING returncode
```

input is an alphanumeric item that represents a null-terminated code-page ID.

output is a signed 4-byte binary item, such as one defined as PIC S9(5) COMP-5. Either the CCSID that corresponds to the input code-page ID string or the error code of -1 is returned.

returncode is a signed 4-byte binary data item, which is set as follows:

0
Successful.

1
The code-page ID is valid but does not have an associated CCSID; *output* is set to -1.

-1
The code-page ID is not a valid code page; *output* is set to -1.

To call these services, you must use the PGMNAME(MIXED) and NODYNAM compiler options.

"Example: get and convert a code-page ID" on page 209

Related tasks
Chapter 11, "Setting the locale," on page 199

Related references
"DYNAM" on page 262
"PGMNAME" on page 274
Chapter 14, "Compiler-directing statements," on page 291

Example: get and convert a code-page ID

The following example shows how you can use the callable services _iwzGetLocaleCP and _iwzGetCCSID to retrieve the locale and EBCDIC code page that are in effect, respectively, and convert a code-page ID to the corresponding CCSID.

```
cbl pgmname(lm)
       Identification Division.
       Program-ID.  "Samp1".
       Data Division.
       Working-Storage Section.
       01 locale-in-effect.
          05 ll-cc             pic x(5).
          05 filler-period     pic x.
          05 ASCII-CP          Pic x(14).
       01 EBCDIC-CP            pic x(10).
```

```
       01 CCSID                pic s9(5) comp-5.
       01 RC                   pic s9(5) comp-5.
       01 n                    pic 99.

       Procedure Division.
       Get-locale-and-codepages section.
       Get-locale.
           Display "Start Samp1."
           Call "_iwzGetLocaleCP"
             using locale-in-effect, EBCDIC-CP
           Move 0 to n
           Inspect locale-in-effect
             tallying n for characters before initial x'00'
           Display "locale in effect: " locale-in-effect (1 : n)
           Move 0 to n
          Inspect EBCDIC-CP
             tallying n for characters before initial x'00'
          Display "EBCDIC code page in effect: "
            EBCDIC-CP (1 : n).

      Get-CCSID-for-EBCDIC-CP.
          Call "_iwzGetCCSID" using EBCDIC-CP, CCSID returning RC
          Evaluate RC
            When 0
              Display "CCSID for " EBCDIC-CP (1 : n) " is " CCSID
            When 1
              Display EBCDIC-CP (1 : n)
                       " does not have a CCSID value."
            When other
              Display EBCDIC-CP (1 : n) " is not a valid code page."
          End-Evaluate.

      Done.
          Goback.
```

If you set the locale to ja_JP.IBM-943 (`set LC_ALL=ja_JP.IBM-943`), the output from the sample program is:

```
Start Samp1.
locale in effect: ja_JP.IBM-943
EBCDIC code page in effect: IBM-1399
CCSID for IBM-1399 is 0000001399
```

Related tasks

“Using environment variables to specify a locale” on page 201

Part 3. Compiling, linking, running, and debugging your program

© Copyright IBM Corp. 2021

Chapter 12. Compiling, linking, and running programs

The following sections explain how to set environment variables, compile, link, run, and correct errors.

Related tasks
“Setting environment variables” on page 213
“Compiling programs” on page 222
“Correcting errors in your source program” on page 227
“Linking programs” on page 232
“Correcting errors in linking” on page 235
“Running programs” on page 235

Related references
“cob2 options” on page 230

Setting environment variables

You use environment variables to set values that programs need. Specify the value of an environment variable by using the `export` command or the putenv() POSIX function. If you do not set an environment variable, either a default value is applied or the variable is not defined.

An *environment variable* defines some aspect of a user environment or a program environment that can vary. For example, you use the COBPATH environment variable to define the locations where the COBOL run time can find a program when another program dynamically calls it. Environment variables are used by both the compiler and runtime libraries.

When you installed IBM COBOL for Linux on x86, the installation process set environment variables to access the COBOL for Linux compiler and runtime libraries. To compile and run a simple COBOL program, the only environment variables that needs to be set is LANG, and it only needs to be set if you wish to use messages other than the default en_US messages.

You can change the value of an environment variable in either of two places by using the `export` command:

- At the prompt in a command shell (for example, in an XTERM window). This environment variable definition applies to programs (processes or child processes) that you run from that shell or from any of its descendants (that is, any shells called directly or indirectly from that shell).
- In the .profile file in your home directory. If you define environment variables in the .profile file, the values of these variables are defined automatically whenever you begin a Linux session, and the values apply to all shell processes.

You can also set environment variables from within a COBOL program by using the putenv() POSIX function, and access the environment variables by using the getenv() POSIX function.

Some environment variables (such as COBPATH and NLSPATH) define directories in which to search for files. If multiple directory paths are listed, they are delimited by colons. Paths that are defined by environment variables are evaluated in order, from the first path to the last in the `export` command. If multiple files that have the same name are defined in the paths of an environment variable, the first located copy of the file is used.

For example, the following `export` command sets the COBPATH environment variable (which defines the locations where the COBOL run time can find dynamically accessed programs) to include two directories, the first of which is searched first:

```
export COBPATH=/users/me/bin:/mytools/bin
```

© Copyright IBM Corp. 2021

“Example: setting and accessing environment variables” on page 221

Related tasks
“Tailoring your compilation” on page 226

Related references
“Compiler and runtime environment variables” on page 214
“Compiler environment variables” on page 216
“Runtime environment variables” on page 218

Compiler and runtime environment variables

COBOL for Linux uses the following environment variables that are common to both the compiler and the run time.

DB2DBDFT
Specifies the database to use for programs that contain embedded SQL statements or that use the Db2 file system.

DBCS_CODEPAGE
Specifies a DBCS code page applicable to DBCS data, including DBCS literals and DBCS data items.

To set the DBCS code page, issue the following command, where *codepage* is the name of a DBCS code page supported by the International Components for Unicode (ICU) conversion libraries, for example, IBM-943 or IBM-EUCjp:

```
export DBCS_CODEPAGE=codepage
```

If DBCS_CODEPAGE is not set, the default DBCS code page associated with the current locale is used.

LANG
Specifies the locale (as described in the related task about using environment variables to specify a locale). LANG also influences the value of the NLSPATH environment variable as described below.

For example, the following command sets the language locale name to U.S. English:

```
export LANG=en_US
```

LC_ALL
Specifies the locale. A locale setting that uses LC_ALL overrides any setting that uses LANG or any other LC_*xx* environment variable (as described in the related task about using environment variables to specify a locale).

LC_COLLATE
Specifies the collation behavior for the locale. This setting is overridden if LC_ALL is specified.

LC_CTYPE
Specifies the code page for the locale. This setting is overridden if LC_ALL is specified.

LC_MESSAGES
Specifies the language for messages for the locale. This setting is overridden if LC_ALL is specified.

LC_TIME
Determines the locale for date and time formatting information. This setting is overridden if LC_ALL is specified.

LD_LIBRARY_PATH
Specifies the directory paths to be used for shared libraries and user-defined compiler exit programs that the EXIT compiler option has identified.

NLSPATH
Specifies the full path name of message catalogs and help files and uses the form *directory_name*/%L/%N, where %L is substituted by the value specified by the LANG environment variable. %N is substituted by the message catalog name.

COBOL for Linux installs the compiler message catalog in `/opt/ibm/cobol/1.1.0/usr/share/locale/`*xx*, and the runtime message catalog in `/opt/ibm/cobol/rte/usr/share/locale/`*xx* in where *xx* is any language that COBOL for Linux supports. The default is en_US.

When you set NLSPATH, be sure to add to NLSPATH rather than replace it. Other programs might use this environment variable. For example:

```
DIR=xxxx
NLSPATH=$DIR/%L/%N:$NLSPATH
export NLSPATH
```

xxxx is the directory where COBOL was installed. The directory *xxxx* must contain a directory *xxxx*/en_US (in the case of a U.S. English language setup) that contains the COBOL message catalog.

Messages in the following languages are included with the product:

en_US
English

ja_JP
Japanese

You can specify the languages for the messages and for the locale setting differently. For example, you can set the environment variable LANG to en_US and set the environment variable LC_ALL to `ja_JP.eucjp`. In this example, any COBOL compiler or runtime messages will be in English, whereas native ASCII (`DISPLAY` or `DISPLAY-1`) data in the program is treated as encoded in code page ja_JP.eucjp (Japanese EUC code page).

The compiler uses the combination of the NLSPATH and the LANG environment variable values to access the message catalog. If NLSPATH is validly set but LANG is not set to one of the locale values shown above, a warning message is generated and the compiler defaults to the en_US message catalog. If the NLSPATH value is invalid, a terminating error message is generated.

The runtime also library also uses NLSPATH to access the message catalog. If NLSPATH is not set correctly, runtime messages appear in an abbreviated form. The compiler and runtime both automatically manage NLSPATH, so you do not need to handle NLSPATH yourself.

TMPDIR
Specifies the location of temporary work files used by the compiler and runtime. If this value is not set, it defaults to the current directory.

For example:

```
export TMPDIR=/tmp
```

TZ
Describes the time-zone information to be used by the locale. TZ has the following format:

```
export TZ=SSS[+|-]nDDD[,sm,sw,sd,st,em,ew,ed,et,shift]
```

If TZ is not present, the default is EST5EDT, the default locale value. If only the standard time zone is specified, the default value of *n* (difference in hours from GMT) is 0 instead of 5.

If you supply values for any of *sm*, *sw*, *sd*, *st*, *em*, *ew*, *ed*, *et*, or *shift*, you must supply values for all of them. If any of these values is not valid, the entire statement is considered invalid and the time-zone information is not changed.

For example, the following statement sets the standard time zone to CST, sets the daylight saving time to CDT, and sets a difference of six hours between CST and UTC. It does not set any values for the start and end of daylight saving time.

```
export TZ=CST6CDT
```

Other possible values are PST8PDT for Pacific United States and MST7MDT for Mountain United States.

Related tasks
"Using environment variables to specify a locale" on page 201

Related references
"Locales and code pages that are supported" on page 202
"TZ environment parameter variables" on page 216

TZ environment parameter variables

The values for the TZ variable are defined below.

Table 22. ***TZ environment parameter variables***

Variable	Description	Default value
SSS	Standard time-zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EST
n	Difference (in hours) between the standard time zone and coordinated universal time (UTC), formerly called Greenwich mean time (GMT). A positive number denotes time zones west of the Greenwich meridian. A negative number denotes time zones east of the Greenwich meridian.	5
DDD	Daylight saving time (DST) zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EDT
sm	Starting month (1 to 12) of DST	4
sw	Starting week (-4 to 4) of DST	1
sd	Starting day of DST: 0 to 6 if *sw* is not zero; 1 to 31 if *sw* is zero	0
st	Starting time (in seconds) of DST	3600
em	Ending month (1 to 12) of DST	10
ew	Ending week (-4 to 4) of DST	-1
ed	Ending day of DST: 0 to 6 if *ew* is not zero; 1 to 31 if *ew* is zero	0
et	Ending time (in seconds) of DST	7200
shift	Amount of time change (in seconds)	3600

Compiler environment variables

COBOL for Linux uses several compiler-only environment variables, as shown below.

Because COBOL words are case insensitive, all letters in COBOL words are treated as uppercase, including *library-name* and *text-name*. Thus environment variable names that correspond to such names must be uppercase. For example, the environment variable name that corresponds to `COPY MyCopy` is MYCOPY.

COBCPYEXT
Specifies the file suffixes to use in searches for copybooks when the COPY *name* statement does not indicate a file suffix. Specify one or more file suffixes with or without leading periods. Separate multiple file suffixes with a space or comma.

If COBCPYEXT is not defined, the following suffixes are searched: CPY, CBL, COB, and the lowercase equivalents cpy, cbl, and cob.

COBLSTDIR
Specifies the directory into which the compiler listing file is written. Specify any valid path. To indicate an absolute path, specify a leading slash. Otherwise, the path is relative to the current directory. A trailing slash is optional.

If COBLSTDIR is not defined, the compiler listing is written into the current directory.

COBOPT
Specifies compiler options. To specify multiple compiler options, separate each option by a space or comma. Surround the list of options with quotation marks if the list contains blanks or characters that are significant to the command shell. For example:

```
export COBOPT="TRUNC(OPT) TERMINAL"
```

Default values apply to individual compiler options.

Note: The compiler interprets certain shell scripting characters as follows:

- An equal sign (=) is interpreted to a left parenthesis, (
- A colon (:) is interpreted to a right parenthesis,)
- An underscore (_) is interpreted to a single quotation mark (')

You can add a backslash (\) escape character to prevent the interpretation and thus to pass characters in the strings. If you want the backslash (\) to represent itself (rather than as an escape character), use the double backslash (\\).

library-name
If you specify *library-name* as a user-defined word, the name is used as an environment variable, and the value of the environment variable is used as the path in which to locate the copybook.

If you do not specify a library-name, the compiler searches the library paths in the following order:

1. Current® directory
2. Paths specified by the `-I`*xxx* option, if set
3. Paths specified by the SYSLIB environment variable

The search ends when the file is found.

For more details, see the documentation of the COPY statement in the related reference about compiler-directing statements.

SYSLIB
Specifies paths to be used for COBOL COPY statements that have text-names that are unqualified by library-names. It also specifies paths to be used for SQL INCLUDE statements.

text-name
If you specify *text-name* as a user-defined word, the name is used as an environment variable, and the value of the environment variable is used as the file-name and possibly the path name of the copybook.

To specify multiple path names, delimit them with a colon (`:`).

For more details, see the documentation of the COPY statement in the related reference about compiler-directing statements.

Related concepts
“Db2 coprocessor” on page 325

Related tasks
“Using SQL INCLUDE with
the Db2 coprocessor” on page 326

Related references
“cob2 options” on page 230
“Compiler options” on page 245
Chapter 14, “Compiler-directing
statements,” on page 291

Runtime environment variables

The COBOL runtime library uses the following runtime-only environment variables.

assignment-name
The user-defined word that you specify (in the ASSIGN clause) for the external file-name for a COBOL file; for example, OUTPUTFILE in the following ASSIGN clause:

```
SELECT CARPOOL ASSIGN TO OUTPUTFILE
```

At run time, you set the environment variable to the name of the system file that you want to associate with the COBOL file. For example:

```
export OUTPUTFILE=january.car_results
```

After you issue the command above, input/output statements for COBOL file CARPOOL operate on system file `january.car_results` in the current directory.

If you do not set the environment variable, or set it to the empty string, COBOL uses the literal name of the environment variable as the system file-name (OUTPUTFILE in the current directory for the ASSIGN example above).

The ASSIGN clause can specify a file stored in a file system other than the default, such as the standard language file system (STL) or the record sequential delimited file system (RSD). For example:

```
SELECT CARPOOL ASSIGN TO STL-OUTPUTFILE
```

In this case, you still set environment variable OUTPUTFILE (not STL-OUTPUTFILE).

CICS_TK_SFS_SERVER
Specifies the fully qualified CICS SFS server name. For example:

```
export CICS_TK_SFS_SERVER=/.:/cics/sfs/sfsServer
```

COBPATH
Specifies the directory paths to be used by the COBOL runtime to locate dynamically accessed programs such as shared libraries. COBPATH is searched first (and if not set, defaults to the current directory ("./")), followed by LD_LIBRARY_PATH.

For example:

```
export COBPATH=/pgmpath/pgmshlib
```

COBRTOPT
Specifies the COBOL runtime options.

Separate runtime options by a comma or a colon. Use parentheses or equal signs (=) as the delimiter for suboptions. Options are not case sensitive. For example, the following two commands are equivalent:

```
export COBRTOPT="CHECK(ON):UPSI(00000000)"
export COBRTOPT=check=on,upsi=00000000
```

If you specify more than one setting for a given runtime option, the rightmost such setting prevails.

The defaults for individual runtime options apply. For details, see the related reference about runtime options.

EBCDIC_CODEPAGE
Specifies an EBCDIC code page applicable to the EBCDIC data processed by programs compiled with the `CHAR(EBCDIC)` or `CHAR(S390)` compiler option.

To set the EBCDIC code page, issue the following command, where *codepage* is the name of the code page to be used:

```
export EBCDIC_CODEPAGE=codepage
```

If EBCDIC_CODEPAGE is not set, the default EBCDIC code page is selected based on the current locale, as described in the related reference about the locales and code pages supported. When the `CHAR(EBCDIC)` compiler option is in effect and multiple EBCDIC code pages are applicable to the locale in effect, you must set the EBCDIC_CODEPAGE environment variable unless the default EBCDIC code page for the locale is acceptable.

CICS_SFS_DATA_VOLUME
Specifies the name of the SFS data volume on which SFS files are to be created. For example:

```
export CICS_SFS_DATA_VOLUME=sfs_SFS_SERV
```

This data volume must have been defined to the SFS server that your application accesses.

If this variable is not set, the default name sfs_SSFS_SERVER is used.

CICS_SFS_INDEX_VOLUME
Specifies the name of the SFS data volume on which alternate index files are to be created. For example:

```
export CICS_SFS_INDEX_VOLUME=sfs_SFS_SERV
```

This data volume must have been defined to the SFS server that your application accesses.

If this variable is not set, alternate index files are created on the same data volume as the corresponding base index files.

CICS_VSAM_AUTO_FLUSH
Specifies whether all changes to CICS SFS files for each input-output operation are committed to disk before control is returned to the application (that is, whether the *operational force* feature of SFS is enabled). You can improve the performance of applications that use SFS files by specifying 0FF for this environment variable. For example:

```
export CICS_VSAM_AUTO_FLUSH=OFF
```

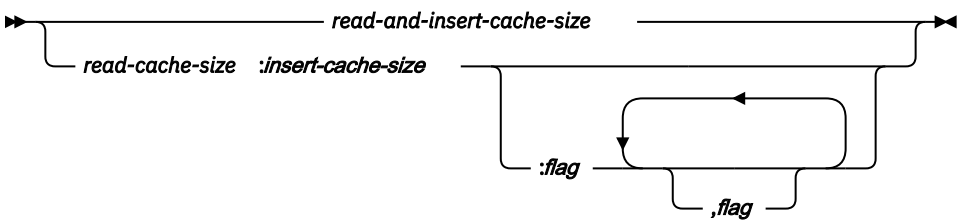
When this environment variable is set to 0FF, SFS uses a *lazy-write* strategy; that is, changes to SFS files might not be committed to disk until the files are closed.

If SFS client-side caching is in effect (that is, environment variable CICS_VSAM_CACHE is set to a valid nonzero value), the setting of CICS_VSAM_AUTO_FLUSH is ignored. Operational force is disabled.

If SFS client-side caching is not in effect, the value of CICS_VSAM_AUTO_FLUSH defaults to ON.

CICS_VSAM_CACHE
Specifies whether client-side caching is enabled for SFS files. You can improve the performance of applications that use SFS files by enabling client-side caching.

CICS_VSAM_CACHE syntax



Size units are in numbers of pages. A size of zero indicates that caching is disabled. The possible flags are:

ALLOW_DIRTY_READS
Removes the restriction for read caching that the files being accessed must be locked.

INSERTS_DESPITE_UNIQUE_INDICES
Removes the restriction for insert caching that inserts are cached only if all active indices for clustered files and all active alternate indices for entry-sequenced and relative files allow duplicates.

For example, the following command sets the read-cache size to 16 pages and the insert-cache size to 64 pages. It also enables dirty reads and enables inserts despite unique indices:

```
export CICS_VSAM_CACHE=16:64:ALLOW_DIRTY_READS, \
    INSERTS_DESPITE_UNIQUE_INDICES
```

By default, client-side caching is not enabled.

CICS_SFS_CACHE_<*filename*>
Specifies whether client-side caching is enabled for a specific SFS file, where you can replace <*filename*> with the SFS file name. If the file name contains (.) characters, you must replace them with (_) (underscore). This setting would help to enable client-side cache based on file operation (read or write) done on any specific file. You can specify CICS_SFS_CACHE_<*filename*> using the same syntax specification as the CICS_SFS_CACHE setting.

For example, to enable file specific client-side caching for a file with the name VSAM.FILE1.TESTFILE, which is always used for SFS write operation, the environment can be specified only to enable WRITE operation cache:

```
export CICS_SFS_CACHE_FILE1_TESTFILE=0:512
```

CICS_SFS_RDM_CACHE
Specifies whether client-side caching is disabled for SFS files used for random read operation. Specify the environment value as 0 to disable client side read operation cache for all files that are opened for random read operation:

```
export CICS_SFS_RDM_CACHE=0
```

The CICS_SFS_CACHE setting is a global setting that is applicable for all SFS files, and it is not recommended to set read operation cache for files that are opened for random read operations. The CICS_VSAM_RDM_CACHE setting can be used to disable the SFS client-side read operation cache.

CICS_SFS_PREALLOC_<*filename*>

Specifies the number of pre-allocated pages for a file, when your program creates the file for the first time on the SFS server. You can replace <*filename*> with the SFS file name. If the file name contains (.) characters, you must replace them with (_) (underscore). For example, to create a SFS file named TESTFILE and pre-allocate 2000 pages, set the following environment variable:

```
export CICS_SFS_PREALLOC_TESTFILE=2000
```

COBCORE

Specifies that the location of core files generated by the runtime will be placed. If not specified, the default is the current working directory. If the name ends with a percent character (%), a file name with the program name and timestamp is created.

COBOUTDIR

Specifies a directory where all files create by COBOL DISPLAY statements (SYSOUT, CONSOLE, SYSPUNCH), and core files (COBCORE) are created, if the variable does not include a path.

PATH

Specifies the directory paths of executable programs.

SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH

These COBOL environment names are used as the environment variable names that correspond to the mnemonic names used in ACCEPT and DISPLAY statements.

If the environment variable is not set, by default SYSIN and SYSIPT are directed to the logical input device (keyboard). SYSOUT, SYSLIST, SYSLST, and CONSOLE are directed to the system logical output device (screen). SYSPUNCH and SYSPCH are not assigned a default value and are not valid unless you explicitly define them. If value of any of these variables ends with a percent character (%), a file name with the program name and timestamp is created.

For example, the following command defines CONSOLE:

```
export CONSOLE=/users/mypath/myfile
```

CONSOLE could then be used in conjunction with the following source code:

```
SPECIAL-NAMES.
    CONSOLE IS terminal
    . . .
    DISPLAY 'Hello World' UPON terminal
```

Related tasks

"Identifying files" on page 112
"Using SFS files" on page 145
"Improving SFS performance" on page 147

Related references

"CHAR" on page 253
Chapter 15, "Runtime options," on page 297

Example: setting and accessing environment variables

The following example shows how you can access and set environment variables from a COBOL program by calling the standard POSIX functions getenv() and putenv().

Because getenv() and putenv() are C functions, you must pass arguments BY VALUE. Pass character strings as BY VALUE pointers that point to null-terminated strings. Compile programs that call these functions with the NODYNAM and PGMNAME(LONGMIXED) options.

```
CBL pgmname(longmixed),nodynam
Identification division.
```

```
 Program-id. "envdemo".
 Data division.
 Working-storage section.
 01 P pointer.
 01 PATH pic x(5) value Z"PATH".
 01 var-ptr pointer.
 01 var-len pic 9(4) binary.
 01 putenv-arg pic x(14) value Z"MYVAR=ABCDEFG".
 01 rc pic 9(9) binary.
 Linkage section.
 01 var pic x(5000).
 Procedure division.
* Retrieve and display the PATH environment variable
     Set P to address of PATH
     Call "getenv" using by value P returning var-ptr
     If var-ptr = null then
         Display "PATH not set"
     Else
         Set address of var to var-ptr
         Move 0 to var-len
         Inspect var tallying var-len
           for characters before initial X"00"
         Display "PATH = " var(1:var-len)
     End-if
* Set environment variable MYVAR to ABCDEFG
     Set P to address of putenv-arg
     Call "putenv" using by value P returning rc
     If rc not = 0 then
         Display "putenv failed"
         Stop run
     End-if
     Goback.
```

Compiling programs

You can compile your COBOL programs from the command line or by using a shell script or makefile.

If you have non-IBM or free-format COBOL source, you might first need to use the source conversion utility, `scu`, to help convert the source so that it can be compiled. To see a summary of the `scu` functions, type the command `scu -h`. For further details, see the `man` page for `scu`, or see the related reference about the source conversion utility.

Specifying compiler options: There are several ways you can specify the options to be used when compiling COBOL programs. For example, you can:

- Set the COBOPT environment variable from the command line.
- Specify compiler options as options to the `cob2` command. You can use this command on the command line, in a shell script, or in a makefile.
- Use PROCESS (CBL) or `*CONTROL` statements. An option that you specify by using PROCESS overrides every other option specification.

For further details about setting compiler options and the relative precedence of the methods of setting them, see the related reference about conflicting compiler options.

Related tasks
“Compiling from the command line” on page 223
“Compiling using shell scripts” on page 224
“Specifying compiler options in the PROCESS (CBL) statement” on page 224
“Modifying the default compiler configuration” on page 225
Chapter 21, “Porting applications between platforms,” on page 375

Related references
“Compiler environment variables” on page 216

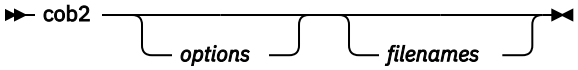
“cob2 options” on page 230
“Compiler options” on page 245
“Conflicting
compiler options” on page 248

Appendix A, “Summary of differences from IBM Enterprise COBOL for z/OS,” on page 467
Reference format (*COBOL for Linux on x86 Language Reference*)
Source conversion utility (scu) (*COBOL for Linux on x86 Language Reference*)

Compiling from the command line

To compile a COBOL program from the command line, issue the cob2 command. This command also invokes the linker.

cob2 command syntax



To compile multiple files, specify the file-names at any position in the command line, using spaces to separate options and file-names. For example, the following two commands are equivalent:

```
cob2 -g filea.cbl fileb.cbl -q"flag(w)"
cob2 filea.cbl -g -q"flag(w)" fileb.cbl
```

The cob2 command accepts compiler and linker options in any order on the command line. Any options that you specify apply to all files on the command line.

Only source files that have suffix .cbl or .cob are passed to the compiler. All other files are passed to the linker.

The default location for compiler input and output is the current directory.

The cob2 command is thread safe. `cob2_r` is provided for compatibility with COBOL for AIX, but uses the same default options as the cob2 command. See “Modifying the default compiler configuration” on page 225 for more information on default options.

Note: The compiler interprets certain shell scripting characters as follows:

- An equal sign (=) is interpreted to a left parenthesis, (
- A colon (:) is interpreted to a right parenthesis,)
- An underscore (_) is interpreted to a single quotation mark (')

You can add a backslash (\) escape character to prevent the interpretation and thus to pass characters in the strings. If you want the backslash (\) to represent itself (rather than as an escape character), use the double backslash (\\).

“Examples: using cob2 for compiling” on page 224

Related tasks
“Modifying the default compiler
configuration” on page 225
“Linking programs” on page 232

Related references
“cob2 options” on page 230
“Compiler options” on page 245

Examples: using cob2 for compiling

The following examples show the output produced for various cob2 invocations.

*Table 23. **Output from the cob2 command***		
To compile:	**Enter:**	**These files are produced:**
alpha.cbl	`cob2 -c alpha.cbl`	alpha.o
alpha.cbl and beta.cbl	`cob2 -c alpha.cbl beta.cbl`	alpha.o, beta.o
alpha.cbl with the LIST and ADATA options	`cob2 -qlist,adata alpha.cbl`	alpha.wlist, alpha.o[1], alpha.lst, alpha.adt, and a.out
1. If the linking is successful, this file is deleted.		

"Examples: using cob2 for linking" on page 233

Related tasks
"Compiling from the command line" on page 223

Related references
"ADATA" on page 249
"LIST" on page 268

Compiling using shell scripts

You can use a shell script to automate the cob2 command.

To prevent invalid syntax from being passed to the command, however, follow these guidelines:

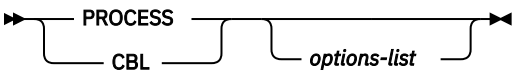
- Use an equal sign and colon rather than parentheses to delimit compiler suboptions.
- Use underscores rather than single quotation marks to delimit compiler suboptions.
- Do not use any blanks in the option string unless you enclose the string in quotation marks (" ").

*Table 24. **Examples of compiler-option syntax in a shell script***	
Use in shell script	**Use on command line**
`-qFLOAT=NATIVE:,CHAR=NATIVE:`	`-qFLOAT(NATIVE),CHAR(NATIVE)`
`-qEXIT=INEXIT=_String_,MYMODULE::`	`-qEXIT(INEXIT('String',MYMODULE))`

Specifying compiler options in the PROCESS (CBL) statement

Within a COBOL program, you can code most compiler options in PROCESS (CBL) statements. Code the statements before the IDENTIFICATION DIVISION header and before any comment lines or compiler-directing statements.

PROCESS(CBL) statement syntax



If you do not use a sequence field, you can start a PROCESS statement in column 1 or after. If you use a sequence field, the sequence number must start in column 1 and must contain six characters; the first character must be numeric. If used with a sequence field, PROCESS can start in column 8 or after.

You can use CBL as a synonym for PROCESS. CBL can likewise start in column 1 or after if you do not use a sequence field. If used with a sequence field, CBL can start in column 8 or after.

You must end PROCESS and CBL statements at or before column 72 if your program uses fixed source format, or column 252 if your program uses extended source format.

Use one or more blanks to separate a PROCESS or CBL statement from the first option in *options-list*. Separate options with a comma or a blank. Do not insert spaces between individual options and their suboptions.

You can code more than one PROCESS or CBL statement. If you do so, the statements must follow one another with no intervening statements. You cannot continue options across multiple PROCESS or CBL statements.

Related references
"SRCFORMAT" on page 280
Reference format (*COBOL for Linux on x86 Language Reference*)
CBL (PROCESS) statement (*COBOL for Linux on x86 Language Reference*)

Modifying the default compiler configuration

The default options used by the `cob2` command are obtained from the configuration file, which is by default `/opt/ibm/cobol/1.1.0/etc/cob2.cfg`. You can display the options used by `cob2` by specifying the `-#` option on the command.

If you are using the default configuration file, the command `cob2 -# abc.cbl` displays output that looks like this:

```
exec: /opt/ibm/cobol/1.1.0/usr/bin/cob3 abc.cbl
exec: /usr/bin/gcc -m32 -shared -fPIC -rdynamic -fasynchronous-unwind-tables -Wl,--hash-
style=gnu
-Wl,--export-dynam -Wl,-Bsymbolic-functions -Wl,--build-id -Wl,--enable-new-dtags -Wl,-zrelro
-Wl,-znow -Wl,-zdefs -Wl,-z,noexecstack -Wl,--allow-shlib-undefined -pie -Wl,--as-needed abc.o
-Wl,--as-needed -L/opt/ibm/cobol/1.1.0/usr/lib/ -L/opt/ibm/cobol/rte/usr/lib/ -lcob2_32s -
lcob2_32r
-ldfp_32r -lm -lpthread -ldl -Wl,-rpath,/opt/ibm/cobol/rte/usr/lib/:
/opt/ibm/cobol/rte/:/opt/ibm/cobol/1.1.0/usr/lib/:/opt/ibm/cics/lib
```

You can modify the `cob2.cfg` configuration file to change the default options.

Instead of modifying the default configuration file, you can tailor a copy of the file for your purposes.

Note: The compiler interprets certain shell scripting characters as follows:

- An equal sign (=) is interpreted to a left parenthesis, (
- A colon (:) is interpreted to a right parenthesis,)
- An underscore (_) is interpreted to a single quotation mark (')

You can add a backslash (\) escape character to prevent the interpretation and thus to pass characters in the strings. If you want the backslash (\) to represent itself (rather than as an escape character), use the double backslash (\\).

Related tasks
"Tailoring your compilation" on page 226

Related references
"cob2 options" on page 230
"Stanzas in the configuration file" on page 227

Tailoring your compilation

To tailor a compilation to your needs, you can change a copy of the default configuration file and use it in various ways.

The configuration file `/opt/ibm/cobol/1.1.0/etc/cob2.cfg` has sections, called *stanzas*, that begin with `cob2:`. To list these stanzas, use the command `ls /opt/ibm/cobol/1.1.0/usr/bin/cob2*`. The resulting list shows these lines:

```
/opt/ibm/cobol/1.1.0/usr/bin/cob2
/opt/ibm/cobol/1.1.0/usr/bin/cob2_r
/opt/ibm/cobol/1.1.0/usr/bin/cob2_cics
/opt/ibm/cobol/1.1.0/usr/bin/cob2_db2
/opt/ibm/cobol/1.1.0/usr/bin/cob2_oracle
```

The `cob2` and `cob2_r` commands execute the same module; however, `cob2` uses the `cob2` stanza and `cob2_r` uses the `cob2_r` stanza.

You can tailor your compilation by doing the following steps:

1. Make a copy of the default configuration file `/opt/ibm/cobol/1.1.0/etc/cob2.cfg`.
2. Change your copy to support specific compilation requirements or other COBOL compilation environments.
3. (Optional) Issue the `cob2` command with the `-#` option to display the effect of your changes.
4. Use your copy instead of `/opt/ibm/cobol/1.1.0/etc/cob2.cfg`.

To use your copy of `cob2.cfg` on every invocation of the compiler, change the symbolic link in the `etc.d` directory with that name to point to your copy. The compiler automatically reads the configuration file pointed to by this symbolic link.

To selectively use a modified copy of the configuration file for a specific compilation, issue the `cob2` command with the `-F` option. For example, to use `/u/myhome/myconfig.cfg` instead of `/opt/ibm/cobol/1.1.0/etc/cob2.cfg` as the configuration file to compile `myfile.cbl`, issue this command:

```
cob2 myfile.cbl -F/u/myhome/myconfig.cfg
```

If you add your own stanza, such as `mycob2`, you can specify it with the `-F` option:

```
cob2 myfile.cbl -F/u/myhome/myconfig.cfg:mycob2
```

Or you can define a `mycob2` command:

```
ln -s /usr/bin/cob2 /u/myhome/mycob2
mycob2 myfile.cbl -F/u/myhome/myconfig
```

Whichever directory you name in the `ln` command (such as `/u/myhome` above) must be in your PATH environment variable.

For a list of the attributes in each stanza, see the related reference about stanzas.

Related tasks
“Setting environment variables” on page 213

Related references
“cob2 options” on page 230
“Stanzas in the configuration file” on page 227

Stanzas in the configuration file

A stanza in the configuration file can contain any of several attributes, as shown in the following table.

Table 25. ***Stanza attributes***

Attribute	Description
compopts	A string of compiler options, separated by commas or spaces. Precede each option by a `-q` flag, or precede the whole string, enclosed in quotation marks, by a `-q` flag. If any option value contains a comma, that option must be enclosed in quotation marks.
coprocessor	Path to the CICS or Db2 coprocessor library.
runlib2, runlib2_64	Additional runtime libraries to link in when working with CICS, Db2, or Oracle.
use	Stanza from which attributes are taken, in addition to the local stanza. For single-valued attributes, values in the `use` stanza apply if no value is provided in the local, or default, stanza. For comma-separated lists, the values from the `use` stanza are added to the values from the local stanza.

Related tasks
"Modifying the default compiler configuration" on page 225
"Tailoring your compilation" on page 226

Related references
"cob2 options" on page 230

Correcting errors in your source program

Messages about source-code errors indicate where the error occurred (`LINEID`). The text of a message tells you what the problem is. With this information, you can correct the source program.

Although you should try to correct errors, it is not always necessary to correct source code for every diagnostic message. You can leave a warning-level or informational-level message in a program without much risk, and you might decide that the recoding and compilation that are needed to remove the message are not worth the effort. Severe-level and error-level errors, however, indicate probable program failure and should be corrected.

In contrast with the four lower levels of severities, an unrecoverable (U-level) error might not result from a mistake in your source program. It could come from a flaw in the compiler itself or in the operating system. In such cases, the problem must be resolved, because the compiler is forced to end early and does not produce complete object code or a complete listing. If the message occurs for a program that has many S-level syntax errors, correct those errors and compile the program again. You can also resolve job set-up problems (such as missing file definitions or insufficient storage for compiler processing) by making changes to the compile job. If your compile job setup is correct and you have corrected the S-level syntax errors, you need to contact IBM to investigate other U-level errors.

After correcting the errors in your source program, recompile the program. If this second compilation is successful, proceed to the link-editing step. If the compiler still finds problems, repeat the above procedure until only informational messages are returned.

Related tasks
"Generating a list of compiler messages" on page 228
"Linking programs" on page 232

Related references
"Messages and listings
for compiler-detected errors" on page 229

Severity codes for compiler diagnostic messages

Conditions that the compiler can detect fall into five levels or categories of severity.

Table 26. ***Severity codes for compiler diagnostic messages***

Level or category of message	Return code	Purpose
Informational (I)	0	To inform you. No action is required, and the program runs correctly.
Warning (W)	4	To indicate a possible error. The program probably runs correctly as written.
Error (E)	8	To indicate a condition that is definitely an error. The compiler attempted to correct the error, but the results of program execution might not be what you expect. You should correct the error.
Severe (S)	12	To indicate a condition that is a serious error. The compiler was unable to correct the error. The program does not run correctly, and execution should not be attempted. Object code might not be created.
Unrecoverable (U)	16	To indicate an error condition of such magnitude that the compilation was terminated.

The final return code at the end of compilation is generally the highest return code that occurred for any message during the compilation.

You can suppress compiler diagnostic messages or change their severities, however, which can have an effect upon the final compilation return code. For details, see the related information.

Related tasks
"Customizing compiler-message severities" on page 540

Related references
"Processing
of MSGEXIT" on page 539

Generating a list of compiler messages

You can generate a complete listing of compiler diagnostic messages with their message numbers, severities, and text by compiling a program that has program-name ERRMSG.

You can code just the PROGRAM-ID paragraph, as shown below, and omit the rest of the program.

```
Identification Division.
Program-ID. ErrMsg.
```

Related tasks
"Customizing compiler-message severities" on page 540

Related references
"Messages and listings
for compiler-detected errors" on page 229

“Format of compiler
diagnostic messages” on page 229

Messages and listings for compiler-detected errors

As the compiler processes your source program, it checks for COBOL language errors, and issues diagnostic messages. These messages are collated in the compiler listing (subject to the FLAG option).

The compiler listing file has the same name as the compiler source file, but with the suffix .lst. For example, the listing file for myfile.cbl is myfile.lst. The listing file is written to the directory from which the cob2 command was issued.

Each message in the listing provides information about the nature of the problem, its severity, and the compiler phase that detected it. Wherever possible, the message provides specific instructions for correcting an error.

The messages for errors found during processing of compiler options, CBL and PROCESS statements, and BASIS, COPY, or REPLACE statements are displayed near the top of the listing.

The messages for compilation errors (ordered by line number) are displayed near the end of the listing for each program.

A summary of all problems found during compilation is displayed near the bottom of the listing.

Related tasks
“Correcting errors in your
source program” on page 227
“Generating a list of compiler
messages” on page 228

Related references
“Format of compiler
diagnostic messages” on page 229
“Severity codes for
compiler diagnostic messages” on page 228
“FLAG” on page 265

Format of compiler diagnostic messages

Each message issued by the compiler has a source line number, a message identifier, and message text.

Each message has the following form:

```
nnnnnn IGYppxxxx-l message-text
```

nnnnnn
The number of the source statement of the last line that the compiler was processing. Source statement numbers are listed on the source printout of your program. If you specified the NUMBER option at compile time, the numbers are the original source program numbers. If you specified NONUMBER, the numbers are those generated by the compiler.

IGY
A prefix that identifies that the message was issued by the COBOL compiler.

pp
Two characters that identify which phase or subphase of the compiler detected the condition that resulted in a message. As an application programmer, you can ignore this information. If you are diagnosing a suspected compiler error, contact IBM for support.

xxxx
A four-digit number that identifies the message.

l
A character that indicates the severity level of the message: I, W, E, S, or U.

message-text
The message text; for an error message, a short explanation of the condition that caused the error.

Tip: If you used the FLAG option to suppress messages, there might be additional errors in your program.

Related references
"Severity codes for compiler diagnostic messages" on page 228
"FLAG" on page 265

cob2 options

The options listed below apply to the cob2 invocation command.

Options that apply to compiling

-c
Compiles programs but does not link them.

-comprc_ok=*n*
Controls the behavior upon return from the compiler. If the return code is less than or equal to *n*, the command continues to the link step, or in the compile-only case, exits with a zero return code. If the return code generated by the compiler is greater than *n*, the command exits with the same return code returned by the compiler.

The default is `-comprc_ok=4`.

-host
Sets these compiler options for host COBOL data representation and language semantics:

- `CHAR(EBCDIC)`
- `COLLSEQ(EBCDIC)`
- `NCOLLSEQ(BIN)`
- `FLOAT(S390)`

The `-host` option changes the format of COBOL program command-line arguments from an array of pointers to an EBCDIC character string that has a halfword prefix that contains the string length. For additional information, see the related task below about using command-line arguments.

If you use the `-host` option and want a main routine in a C object file to be the main entry point of your application, you must use the `-cmain` linker option as described below.

-I*xxx*
Adds path *xxx* to the directories to be searched for copybooks if neither a library-name nor SYSLIB is specified. (This option is the uppercase letter *I*, not the lowercase letter *l*.)

Only a single path is allowed for each `-I` option. To add multiple paths, use multiple `-I` options. Do not insert spaces between `-I` and *xxx*.

-q*xxx*
Passes options to the compiler, where *xxx* is any compiler option or set of compiler options. Do not insert spaces between `-q` and *xxx*.

If a parenthesis is part of the compiler option or suboption, or if a series of options is specified, include them in quotation marks.

To specify multiple options, delimit each option by a blank or comma. For example, the following two option strings are equivalent:

```
-qoptiona,optionb
```

```
-q"optiona optionb"
```

If you plan to use a shell script to automate your cob2 tasks, a special syntax is provided for the -q*xxx* option. For details, see the related task about compiling using shell scripts.

Options that apply to linking

-cmain
(Has an effect only if you also specify -host.) Makes a C object file (that contains a main routine) the main entry point in the executable file. In C, a main routine is identified by the function name main().

If you link a C object file that contains a main routine with one or more COBOL object files, you must use -cmain to designate the C routine as the main entry point. A COBOL program cannot be the main entry point in an executable file that contains a C main routine. Unpredictable behavior occurs if this is attempted, and no diagnostics are issued.

-main:*xxx*
Makes *xxx* the first file in the list of files passed to the linker. The purpose of this option is to make the specified file the main program in the executable file. *xxx* must uniquely identify the object file or the archive library, and the suffix must be either .o or .a, respectively.

If -main is not specified, the first object, archive library, or source file specified in the command is the first file in the list of files passed to the linker.

If the syntax of -main:*xxx* is invalid, or if *xxx* is not the name of an object or source file processed by the command, the command terminates.

-o *xxx*
Names the executable module *xxx*, where *xxx* is any name. If the -o option is not used, the name of the executable module defaults to a.out.

Options that apply to both compiling and linking

-F*xxx*
Uses *xxx* as a configuration file or a stanza rather than the defaults specified in the /opt/ibm/cobol/1.1.0/etc/cob2.cfg configuration file. *xxx* has one of the following forms:

- *configuration_file*:*stanza*
- *configuration_file*
- :*stanza*

-g
Produces symbolic information used by the debugger. Sets the TEST compiler option.

-q32
Specifies that a 32-bit object program is to be generated. Sets the ADDR(32) compiler option. Sets the -m32 linker option, which instructs the linker to create a 32-bit executable module.

-q64
This option is not currently supported. The compiler accepts and ignores this option.

-v
Displays compile and link steps, and executes them.

-#
Displays compile and link steps, but does not execute them.

-?, ?
Displays help for the cob2 command.

Related tasks
"Compiling from the command line" on page 223

“Compiling using shell scripts” on page 224
“Passing options to the linker” on page 232
“Using command-line arguments” on page 407

Related references
“Compiler environment variables” on page 216
“ADDR” on page 249

Linking programs

Use the linker to link specified object files and create an executable file or a shared object.

You have a choice of ways to start the linker. You can use:

- The cob2 command:

  This command calls the linker unless you specify the -c option.

  cob2 links with the COBOL multithreaded libraries, so there is no need for any additional thread options.

  Linking with C: The linker accepts .o files, but does not accept .c files. If you want to link C and COBOL files together, first produce .o files for the C source files by using the gcc command.
- A makefile:

  You can use a makefile to organize the sequence of actions (such as compiling and linking) that are required for building your program. In the makefile, you can use linker statements to specify the kind of output that you need.

You can specify linker options using any of the methods described above.

“Examples: using cob2 for linking” on page 233

Related tasks
“Compiling from the command line” on page 223
“Passing options to the linker” on page 232
Chapter 24, “Using shared libraries,” on page 411

Related references
“cob2 options” on page 230
“Linker input and output files” on page 233
“Linker search rules” on page 234

Passing options to the linker

You can specify linker options in any of these ways: an invocation command (cob2) or makefile statements.

Any options that you specify in an invocation command that are not recognized by the command are passed to the linker.

Several invocation command options influence the linking of your program. For details about the set of options for a given command, consult documentation for that command.

The following table shows the options that you are more likely to need for your COBOL programs. Precede each option with a hyphen (-) as shown.

Table 27. ***Common linker options***	
Option	**Purpose**
`-L`*`dir`*	Specifies the directory to search for libraries specified by the `-l` option (default: /usr/lib)
`-l`*`name`*	Searches the specified library-file, where *name* selects the file lib*name*.a

"Examples: using cob2 for linking" on page 233

Related references
"cob2 options" on page 230
"Linker input and output files" on page 233
"Linker search rules" on page 234

Examples: using cob2 for linking

You can use any of the COBOL invocation commands to both compile and link your programs. The following examples illustrate the use of `cob2`.

- To link two files together after they are compiled, do not use the `-c` option. For example, to compile and link `alpha.cbl` and `beta.cbl` and generate `a.out`, enter:

  ```
  cob2 alpha.cbl beta.cbl
  ```

  This command creates `alpha.o` and `beta.o`, then links `alpha.o`, `beta.o`, and the COBOL libraries. If the link step is successful, it produces an executable program named `a.out` and deletes `alpha.o` and `beta.o`.
- To link a compiled file with a library, enter:

  ```
  cob2 zog.cbl -lmylib
  ```

  This command causes the linker to search for the library `libmylib.so` first, and then the archive library file `libmylib.a` in each directory in the search path consecutively until either is encountered.
- To use options to limit the search for a library, enter:

  ```
  cob2 zog.cbl  -llib1  -llib2
  ```

  In this case, to satisfy the first library specification, the linker searches for the library `liblib1.so` first and then the archive library file `liblib1.a` in each directory (as described in the previous example). However, at the same time the linker searches only for `liblib2.a` in those same libraries.
- To compile and link in separate steps, enter commands such as these:

  ```
  cob2 -c file1.cbl               # Produce one object file
  cob2 -c file2.cbl file3.cbl     # Or multiple object files
  cob2 file1.o file2.o file3.o    # Link with appropriate libraries
  ```

"Examples: using cob2 for compiling" on page 224

Linker input and output files

The linker takes object files, links them with each other and with any library files that you specify, and produces an executable output file. The executable output can be either an executable program file or a shared object.

Linker inputs:

- Options
- Object files (*.o)
- Archive library files (*.a)
- Dynamic library files (*.so)

Linker outputs:

- Executable file (a.out by default)
- Shared object
- Return code

Library files: *Libraries* are files that have suffix .a or .so. To designate a library, you can specify an absolute or relative path name or use the `-l` (lowercase letter L) option in the form `-l`*name*. The last form designates file lib*name*.a, or in dynamic mode, file lib*name*.so, to be searched for in several directories. These search directories include directories that you specify by using `-L` options, and the standard library directories /usr/lib and /lib.

The environment variable LD_LIBRARY_PATH is not used to search for libraries that you specify on the command line either explicitly (for example, `libc.a`) or by using the `-l` option (for example, `-lc`). You must use `-L`*dir* options to indicate the directories to be searched for libraries that you specified with a `-l` option.

You can create library files by combining one or more files into a single archive file by using the Linux `ar` command.

"Example: creating a sample shared library" on page 412

Related tasks
"Passing options to the linker" on page 232

Related references
"Linker search rules" on page 234
"Linker file-name defaults" on page 235

Linker search rules

When searching for an object file (.o) or an archive library file (.a), the linker looks in several locations until the search is satisfied.

The linker searches these locations:

1. The directory that you specify for the file

   If you specify a path with the file, the linker searches only that path and stops linking if the file cannot be found there.
2. The current directory, if you did not specify a path
3. The value of the environment variable LD_LIBRARY_PATH, if defined

If you use libraries other than the default ones in /usr/lib, you can specify one or more `-L` options that point to the locations of the other libraries. You can also set the LD_LIBRARY_PATH environment variable, which lets you specify a search path for libraries at run time.

If the linker cannot locate a file, it generates an error message and stops linking.

Related tasks
"Passing options to the linker" on page 232

Related references
"Linker file-name defaults" on page 235
AIX General Programming Concepts: Writing and Debugging Programs

Linker file-name defaults

If you do not enter a file-name, the linker assumes default names.

*Table 28. **Default file-names assumed by the linker***		
File	**Default file-name**	**Default suffix**
Object files	None. You must enter at least one object file name.	.o
Output file	a.out	None
Library files	The default libraries defined in the object files. Use compiler options to define the default libraries. Any additional libraries that you specify are searched before the default libraries.	.a or .so

Correcting errors in linking

When you use the PGMNAME(UPPER) compiler option, the names of subprograms referenced in CALL statements are translated to uppercase. This change affects the linker, which recognizes case-sensitive names.

For example, the compiler translates `Call "RexxStart"` to `Call "REXXSTART"`. If the real name of the called program is RexxStart, the linker will not find the program, and will produce an error message that indicates that REXXSTART is an unresolved external reference.

This type of error typically occurs when you call API routines that are supplied by another software product. If the API routines have mixed-case names, you must take both of the following actions:

- Use the PGMNAME(MIXED) compiler option.
- Ensure that CALL statements specify the correct mix of uppercase and lowercase in the names of the API routines.

Running programs

To run a COBOL program, you set environment variables, issue the command to run the executable, and then correct any runtime errors.

1. Make sure that any needed environment variables are set.

   For example:

   - If the program uses an environment variable name to assign a value to a system file-name, set that environment variable.
   - If the program uses runtime options, specify their values in the COBRTOPT runtime environment variable.
2. Run the program: At the command line, either enter the name of the executable module or run a command file that invokes that module. Include any needed program arguments on the command line.

   For example, if the command `cob2 alpha.cbl beta.cbl-o gamma` is successful, you can run the program by entering `gamma`.
3. Correct runtime errors.

Tip: If runtime messages are abbreviated or incomplete, the environment variables LANG or NLSPATH or both might be incorrectly set.

Related tasks

"Setting environment variables" on page 213

"Using command-line arguments" on page 407

Related references
Chapter 15, “Runtime options,” on page 297

Appendix G, “Runtime messages,” on page 549

Chapter 13. Specifying compiler options on the command line

Most options specified on the command line override both the default settings of the option and options set in the configuration file.

There are two kinds of command-line options:

- Flag options
- **-q***option_keyword* (compiler-specific)

Related concepts
"Flag options" on page 237
"-q options" on page 245

Related tasks
"Compiling programs" on page 222

Flag options

COBOL for Linux supports a number of common conventional flag options that are used on Linux systems. Flag options are case-sensitive and they apply to the `cob2` invocation command.

COBOL for Linux also supports flags that are directed to other programming tools and utilities (for example, the `scu` command).

Some flag options have arguments that form part of the flag. For example:

```
cob2 stem.cbl -F/home/tools/test3/new.cfg:cob2
```

where `new.cfg` is a custom configuration file.

You can specify flags that do not take arguments in one string. For example:

```
cob2 -ocv file.cbl
```

```
cob2 -o -c -v file.cbl
```

A flag option that takes arguments can be specified as part of a single string, but you can only use one flag that takes arguments, and it must be the last option specified. For example, you can use the `-o` flag (to specify a name for the executable file) together with other flags, only if the `-o` option and its argument are specified last. For example:

```
cob2 -ocv test test.cbl
```

has the same effect as:

```
cob2 -o -c -vtest test.cbl
```

Most flag options are a single letter, but some are two letters. Take care not to specify two or more options in a single string if there is another option that uses that letter combination.

Related concepts
Chapter 13, "Specifying compiler options on the command line," on page 237

© Copyright IBM Corp. 2021

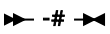
"-q
options" on page 245

-# (pound sign)

Purpose

Previews the compilation steps that are specified on the command line, without actually invoking any compiler components. When this option is enabled, information is written to standard output, showing the names of the programs within the preprocessor, compiler, and linker that would be invoked, and the default options that would be specified for each program. The preprocessor, compiler, and linker are not invoked. This option applies to both compiling and linking.

Syntax



Defaults

The compiler does not display the progress of the compilation.

Usage

You can use this command to determine the commands and files that are involved in a particular compilation. It avoids the overhead of compiling the source code and overwriting any existing files, such as `.lst` files. This option displays the same information as `-v`, but does not invoke the compiler. The `-#` option overrides the `-v` option.

Related references

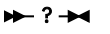
"-v" on page 244

-?, ?

Purpose

Displays help for the `cob2` command. This option applies to both compiling and linking.

Syntax



Defaults

The compiler does not display the command help information.

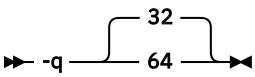
-q32, -q64

Purpose

`-q32`: Specifies that a 32-bit object program is to be generated. Sets the `ADDR(32)` compiler option. Sets the `-m32` linker option, which instructs the linker to create a 32-bit executable module.

`-q64`: This option is not currently supported. The compiler accepts and ignores this option.

Syntax



Defaults

-q32

Related references

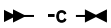
"ADDR" on page 249

-c

Purpose

Prevents the completed object from being sent to the linker. When this option is in effect, the compiler creates an output object file, *file_name*.o. This option applies only to compiling.

Syntax



Defaults

By default, the compiler invokes the linker to link object files into a final executable file.

Examples

- To compile one file that is called alpha.cbl, enter:

```
cob2 -c alpha.cbl
```

  The compiled file is named alpha.o.
- To compile two files that are called alpha.cbl and beta.cbl, enter:

```
cob2 -c alpha.cbl beta.cbl
```

  The compiled files are named alpha.o and beta.o.
- To link two files, compile them without the -c option. For example, to compile and link alpha.cbl and beta.cbl and generate gamma, enter:

```
cob2 alpha.cbl beta.cbl -o gamma
```

  This command creates alpha.o and beta.o, then links alpha.o, beta.o, and the COBOL libraries. If the link step is successful, it produces an executable program named gamma.
- To compile alpha.cbl with the LIST and NODATA options, enter:

```
cob2 -qlist,noadata alpha.cbl
```

-comprc_ok

Purpose

Controls the behavior upon return from the compiler. If the return code is less than or equal to *n*, the command continues to the link step, or in the compile-only case, exits with a zero return code. If the return code generated by the compiler is greater than *n*, the command exits with the same return code returned by the compiler. This option applies only to compiling.

Syntax

Defaults

The default is `-comprc_ok=4`.

Usage

When this option is in effect, the compiler creates an output object file, *`file_name`*`.o`.

-dll | -shared

Purpose

Changes the output of the linkage editor from the default Position Independent Executable (PIE) to a Dynamically Shared Object (DSO).

Syntax

Defaults

The output of the linkage editor is a PIE.

Usage

A PIE can be invoked like any other executable program, but can not be used as the target of a COBOL dynamic call, or a CICS transaction.

A DSO is the opposite. It can not be called from the command line, but can be used as the target of a COBOL dynamic call, or a CICS transaction.

-F

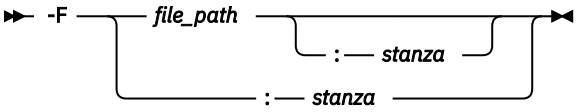
Purpose

Uses *xxx* as a configuration file or a stanza rather than the defaults specified in the `/opt/ibm/cobol/1.1.0/etc/cob2.cfg` configuration file. *xxx* has one of the following forms:

- *configuration_file*:*stanza*
- *configuration_file*
- :*stanza*

This option applies to both compiling and linking.

Syntax



Defaults

By default, the compiler uses the configuration file that is supplied at installation time, and uses the stanza defined in that file for the invocation command currently being used.

Parameters

File_path

The full path name of the alternate compiler configuration file to use.

stanza

The name of the configuration file stanza to use for compilation. This directs the compiler to use the entries under that `stanza` regardless of the invocation command being used.

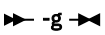
Related references
"Stanzas in the configuration file" on page 227

-g

Purpose

Produces symbolic information used by the debugger. Sets the TEST compiler option. This option applies to both compiling and linking.

Syntax



Defaults

Generates no debugging information. No program state is preserved.

Examples

Use the following command to compile `myprogram.cbl` and generate an executable program called testing for debugging:

```
cob2 myprogram.cbl -o testing -g
```

Related references
"TEST" on page 283

-host

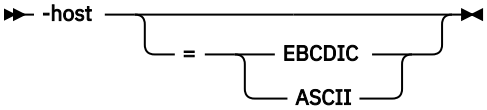
Purpose

Sets these compiler options for host COBOL data representation and language semantics:

- BINARY(BE)
- CHAR(EBCDIC)
- COLLSEQ(EBCDIC)
- FLOAT(BE)
- NCOLLSEQ(BIN)
- UTF16(BE)

The -host option changes the format of COBOL program command-line arguments from an array of pointers to an EBCDIC character string that has a halfword prefix that contains the string length. For additional information, see the related task below about using command-line arguments.

Syntax



Defaults

The compiler does not set compiler options for host COBOL data representation and language semantics.

Parameters

EBCDIC
Same as -host. Passes a z/OS style parameter list in EBCDIC to your program.

ASCII
Passes a z/OS style parameter list in ASCII to your program.

Related references
"BINARY" on page 252
CHAR
COLLSEQ
FLOAT
NCOLLSEQ
"UTF16" on page 285

-I

Purpose

Adds path *xxx* to the directories to be searched for copybooks if neither a library-name nor SYSLIB is specified. (This option is the uppercase letter *I*, not the lowercase letter *l*.)

Only a single path is allowed for each -I option. To add multiple paths, use multiple -I options.

Syntax

Parameters

directory_path

The path for the directory where the compiler should search for the header files.

Defaults

There is no default directory to be search for copybooks.

Usage

If the `-I` directory option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first. The `-I` directory option can be specified more than once on the command line. If you specify more than one `-I` option, directories are searched in the order that they appear on the command line. The `-I` option has no effect on files that are included using an absolute path name

Examples

To compile `myprogram.cbl` and search `/usr/tmp and then /oldstuff/history` for included files, enter:

```
cob2 myprogram.cbl -I/usr/tmp -I/oldstuff/history
```

-main

Purpose

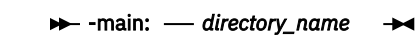
Makes *xxx* the first file in the list of files that are passed to the linker. The purpose of this option is to make the specified file the main program in the executable file. *xxx* must uniquely identify the object file or the archive library, and the suffix must be either `.o` or `.a`, respectively.

If `-main` is not specified, the first object, archive library, or source file specified in the command is the first file in the list of files that are passed to the linker.

If the syntax of `-main:`*xxx* is invalid, or if *xxx* is not the name of an object or source file that is processed by the command, the command terminates.

This option applies only to linking.

Syntax



Defaults

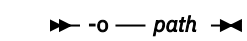
The `-main` option is not specified.

-o

Purpose

Names the executable program or shared library *xxx*, where *xxx* is any name. If the `-o` option is not used, the name of the executable module defaults to `a.out`. This option applies only to linking.

Syntax



Parameters

path

When you are using the option to compile from source files, *path* can be the name of a file or directory. The *path* can be a relative or absolute path name. When you are using the option to link from object files, *path* must be a file name. If *path* is the name of an existing directory, files that are created by the compiler are placed into that directory. If *path* is not an existing directory, *path* is the name of the file that is produced by the compiler. See below for examples

Defaults

If you specify the `-c` option, an output object file, *file_name*`.o`, is produced for each input file. The linker is not invoked, and the object files are placed in your current directory. All processing stops at the completion of the compilation. The compiler gives object files a `.o` suffix, for example, *file_name*`.o`, unless you specify the `-o` option, giving a different suffix or no suffix at all.

Usage

If you use the `-c` option with `-o` together and the path is not an existing directory, you can compile only one source file at a time. In this case, if more than one source file name is listed in the compiler invocation, the compiler issues a warning message and ignores `-o`.

Examples

To compile `myprogram.cbl` so that the resulting executable is called `myaccount`, assuming that no directory with name `myaccount` exists, enter:

```
cob2 myprogram.cbl -o myaccount
```

To compile `test.cbl` to an object file only and name the object file `new.o`, enter:

```
cob2 test.cbl -c -o new.o
```

Related references

-v

Purpose

Displays compile and link steps, and executes them. This option applies to both compiling and linking. When the `-v` option is in effect, information is displayed in a comma-separated list. This option applies to both compiling and linking.

Note: The `-v` option is overridden by the `-#` option.

Syntax

Defaults

The compiler does not display the progress of the compilation.

Examples

To compile `myprogram.cbl` so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

```
cob2 myprogram.cbl -v
```

Related references
"-# (pound sign)" on page 238

-q options

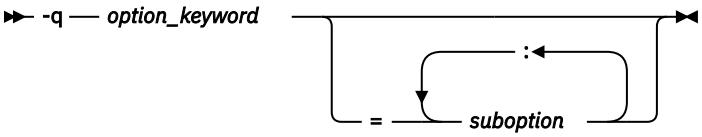
Passes options to the compiler, where *xxx* is any compiler option or set of compiler options. Do not insert spaces between -q and *xxx*. If a parenthesis is part of the compiler option or suboption, or if a series of options is specified, include them in quotation marks. If a parenthesis is part of the compiler option or suboption, or if a series of options is specified, include them in quotation marks.

To specify multiple options, delimit each option by a blank or comma. For example, the following two option strings are equivalent:

```
-qoptiona,optionb
```

```
-q"optiona optionb"
```

If you plan to use a shell script to automate your cob2 tasks, a special syntax is provided for the -q*xxx* option. For details, see the related task about compiling using shell scripts.



Related concepts
Chapter 13, "Specifying compiler options on the command line," on page 237

Related references
"Compiler options" on page 245
"Conflicting compiler options" on page 248

Related tasks
Compiling using shell scripts

Compiler options

You can direct and control your compilation by using compiler options or by using compiler-directing statements (compiler directives).

Compiler options affect the aspects of your program that are listed in the table below. The linked-to information for each option provides the syntax for specifying the option and describes the option, its parameters, and its interaction with other parameters.

Table 29. ***Compiler options***

Aspect of your program	Compiler option	Default	Option abbreviations
Source language	"ARITH" on page 251	ARITH(COMPAT)	AR(C|E)
	"CICS" on page 255	NOCICS	None
	"CURRENCY" on page 258	NOCURRENCY	CURR|NOCURR
	"NSYMBOL" on page 272	NSYMBOL(NATIONAL)	NS(NAT|DBCS)
	"NUMBER" on page 272	NONUMBER	NUM|NONUM
	"APOST/QUOTE" on page 275	QUOTE	Q|APOST
	"SEQUENCE" on page 276	SEQUENCE	SEQ|NOSEQ
	"SOSI" on page 277	NOSOSI	None
	"SQL" on page 279	SQL("")	None
	"SRCFORMAT" on page 280	SRCFORMAT(COMPAT)	SF(C|E)
Date processing	"DATEPROC" on page 259	NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified	DP|NODP
	"DATETIME" on page 259	DATETIME(1900 40)	None
	"YEARWINDOW" on page 288	YEARWINDOW(1900)	YW
Maps and listings	"LINECOUNT" on page 268	LINECOUNT(60)	LC
	"LIST" on page 268	NOLIST	None
	"LSTFILE" on page 269	LSTFILE(LOCALE)	LST
	"MAP" on page 269	NOMAP	None
	"SOURCE" on page 278	SOURCE	S|NOS
	"SPACE" on page 279	SPACE(1)	None
	"TERMINAL" on page 282	TERMINAL	TERM|NOTERM
	"VBREF" on page 286	NOVBREF	None
	"XREF" on page 287	XREF(FULL)	X|NOX
Object module generation	"COMPILE" on page 257	NOCOMPILE(S)	C|NOC
	"PGMNAME" on page 274	PGMNAME(UPPER)	PGMN(LU|LM)
	"SEPOBJ" on page 275	SEPOBJ	None

*Table 29. **Compiler options** (continued)*

Aspect of your program	Compiler option	Default	Option abbreviations
Object code control	"ADDR" on page 249	ADDR(32)	None
	"BINARY" on page 252	BINARY(NATIVE)	None
	"CHAR" on page 253	CHAR(NATIVE)	None
	"COLLSEQ" on page 256	COLLSEQ(BIN)	CS(L|E|BIN|B)
	"DEFINE" on page 260	NODEFINE	DEF | NODEF
	"DIAGTRUNC" on page 261	NODIAGTRUNC	DTR|NODTR
	"FLOAT" on page 267	FLOAT(NATIVE)	None
	"NCOLLSEQ" on page 271	NCOLLSEQ(BINARY)	NCS(L|BIN|B)
	"OPTIMIZE" on page 273	NOOPTIMIZE	OPT|NOOPT
	"TRUNC" on page 283	TRUNC(STD)	None
	"ZWB" on page 289	ZWB	None
CALL statement behavior	"DYNAM" on page 262	NODYNAM	DYN|NODYN
Debugging and diagnostics	"FLAG" on page 265	FLAG(I,I)	F|NOF
	"FLAGSTD" on page 266	NOFLAGSTD	None
	"SSRANGE" on page 281	NOSSRANGE	SSR(MSG|ABD)|NOSSR
	"TEST" on page 283	NOTEST	None
Other	"ADATA" on page 249	NOADATA	None
	"CALLINT" on page 252	CALLINT(SYSTEM,NODESC)	None
	"EXIT" on page 263	NOEXIT	NOEX|EX(INX|NOINX, LIBX|NOLIBX, PRTX|NOPRTX, ADX|NOADX, MSGX|NOMSGX)
	"MAXMEM" on page 270	MAXMEM(2000)	None
	"MDECK" on page 270	NOMDECK	NOMD|MD|MD(C|NOC)
	"SIZE" on page 277	SIZE(8388608)	SZ
	"SPILL" on page 279	SPILL(512)	None
	"THREAD" on page 283	NOTHREAD	None
	"WSCLEAR" on page 286	NOWSCLEAR	None

Installation defaults: The defaults listed for the options above are the defaults shipped with the product.

Option specifications:

- Compiler options and suboptions are not case sensitive.
- For compiler options that are followed by arguments as the suboptions, you must adhere to using the format of `option(argument)`, instead of specifying `option=argument`.

Performance considerations: The ADDR, ARITH, CHAR, DYNAM, FLOAT, OPTIMIZE, SSRANGE, TEST, TRUNC, and WSCLEAR compiler options can affect runtime performance.

Related tasks
"Compiling programs" on page 222
Chapter 27, "Tuning your program," on page 443

Related references
Chapter 14, "Compiler-directing statements," on page 291
"Performance-related compiler options" on page 451

Option settings for 85 COBOL Standard conformance

Compiler options and runtime options are required for conformance with the 85 COBOL Standard.

The following compiler options are required:

- DYNAM
- NOCICS
- NOSOSI
- NOTHREAD
- PGMNAME(COMPAT) or PGMNAME(LONGUPPER)
- QUOTE
- TRUNC(STD)
- ZWB

You can use the FLAGSTD compiler option to flag nonconforming elements such as IBM extensions.

Conflicting compiler options

The COBOL for Linux compiler can encounter conflicting compiler options in either of two ways: both the positive and negative form of an option are specified at the same level in the hierarchy of precedence of options, or mutually exclusive options are specified at the same level.

The compiler recognizes options in the following order of precedence from highest to lowest:

1. Options specified in the PROCESS (or CBL) statement
2. Options specified in the cob2 command invocation
3. Options set in the COBOPT environment variable
4. Options set in the compopts attribute of the configuration (.cfg) file
5. IBM default options

If you specify conflicting options at the same level in the hierarchy, the option specified last takes effect.

If you specify mutually exclusive compiler options at the same level, the compiler forces one of the options to a nonconflicting value, and generates an error message. For example, if you specify both CICS and DYNAM in the PROCESS statement in any order, CICS takes effect and DYNAM is ignored, as shown in the following table.

Table 30. ***Mutually exclusive compiler options***

Specified	Ignored	Forced on
CICS	ADDR(64)	ADDR(32)
	DYNAM	NODYNAM
	THREAD	NOTHREAD

However, options specified at a higher level of precedence override options specified at a lower level of precedence. For example, if you code CICS in the COBOPT environment variable but DYNAM in the PROCESS statement, DYNAM takes effect because the options coded in the PROCESS statement and any options forced on by an option coded in the PROCESS statement have higher precedence.

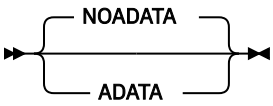
Related tasks
"Compiling programs" on page 222

Related references
"Compiler environment variables" on page 216
"Stanzas in the configuration file" on page 227
"cob2 options" on page 230
Chapter 14, "Compiler-directing statements," on page 291

ADATA

Use ADATA when you want the compiler to create a SYSADATA file that contains records of additional compilation information.

ADATA option syntax



Default is: NOADATA

Abbreviations are: None

SYSADATA information is used by other tools, which will set ADATA ON for their use.

The size of the SYSADATA file generally grows with the size of the associated program.

Option specification: You cannot specify the ADATA option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- As an option in the cob2 command or one of its variants
- In the COBOPT environment variable
- In the compopts attribute of the configuration (.cfg) file

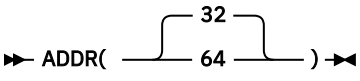
Note: The ADATA option is not currently supported. Use the default, NOADATA for now.

Related references
"Compiler environment variables" on page 216
"Stanzas in the configuration file" on page 227
"cob2 options" on page 230

ADDR

Use the ADDR compiler option to indicate whether a 32-bit or 64-bit object program should be generated.

ADDR option syntax



Default is: ADDR(32)

Abbreviations are: None

Option specification:

Note: The ADDR(64) and -q64 options are not currently supported. The compiler accepts and ignores these options and defaults to ADDR(32).

You can specify the ADDR option in any of the ways that you specify other compiler options, as described in the related task about compiling programs. However, if you specify ADDR in a PROCESS (or CBL) statement:

- In a batch compilation, you can specify ADDR only for the first program. You cannot change the value of the option for subsequent programs in the batch.
- You must use the matching 32-bit or 64-bit option in the link step.

If you specify compiler options using the -q option of the cob2 command, you can abbreviate ADDR(32) as 32 or ADDR(64) as 64. For example:

```
cob2 -q64 prog64.cbl
```

Storage allocation:

The storage allocation for the following COBOL data types depends on the setting of the ADDR compiler option:

- USAGE POINTER (also the ADDRESS OF special register, which implicitly has this usage)
- USAGE PROCEDURE-POINTER
- USAGE FUNCTION-POINTER
- USAGE INDEX

If ADDR(32) is in effect, 4 bytes are allocated for each item in your program that has one of the usages listed above; if ADDR(64) is in effect, 8 bytes are allocated for each of the items.

If the SYNCHRONIZED clause is specified for a data item that has one of the usages shown above, the item is aligned on a fullword boundary if ADDR(32) is in effect, or on a doubleword boundary if ADDR(64) is in effect.

The setting of the ADDR option affects several compiler limits. For details, see the related reference about compiler limits.

LENGTH OF special register:

If ADDR(32) is in effect, the LENGTH OF special register has this implicit definition:

```
PICTURE 9(9) USAGE IS BINARY
```

If ADDR(64) is in effect, the LENGTH OF special register has this implicit definition:

```
PICTURE 9(18) USAGE IS BINARY
```

LENGTH intrinsic function:

If ADDR(32) is in effect, the returned value of the LENGTH intrinsic function is a 9-digit integer. If ADDR(64) is in effect, the returned value is an 18-digit integer.

Programming requirements and restrictions:

- All program components within an application must be compiled using the same setting of the ADDR option. You cannot mix 32-bit programs and 64-bit programs in an application.

- **Interlanguage communication:** In multilanguage applications, 64-bit COBOL programs can be combined with 64-bit C/C++ programs, and 32-bit COBOL programs can be combined with 32-bit C/C++ programs.
- **CICS:** COBOL programs that will run in the CICS TXSeries or CICS TX environment must be 32 bit.

Related concepts

Related tasks
"Finding the length of data items" on page 109
"Compiling programs" on page 222
"Coding COBOL programs to run under CICS" on page 330
"Calling between COBOL and C/C++ programs" on page 387

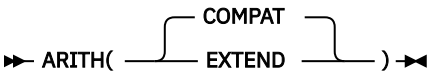
Related references
"cob2 options" on page 230
"Conflicting compiler options" on page 248

Compiler limits (*COBOL for Linux on x86 Language Reference*)

ARITH

ARITH affects the maximum number of digits that you can code for numeric items, and the number of digits used in fixed-point intermediate results.

ARITH option syntax



Default is: `ARITH(COMPAT)`

Abbreviations are: `AR(C | E)`

When you specify `ARITH(EXTEND)`:

- The maximum number of digit positions that you can specify in the `PICTURE` clause for packed-decimal, external-decimal, and numeric-edited data items is raised from 18 to 31.
- The maximum number of digits that you can specify in a fixed-point numeric literal is raised from 18 to 31. You can use numeric literals with large precision anywhere that numeric literals are currently allowed, including:
  - Operands of `PROCEDURE DIVISION` statements
  - `VALUE` clauses (for numeric data items with large-precision `PICTURE`)
  - Condition-name values (on numeric data items with large-precision `PICTURE`)
- The maximum number of digits that you can specify in the arguments to `NUMVAL`, `NUMVAL-C` and is raised from 18 to 31.
- The maximum value of the integer argument to the `FACTORIAL` function is 29.
- Intermediate results in arithmetic statements use *extended mode*.

When you specify `ARITH(COMPAT)`:

- The maximum number of digit positions in the `PICTURE` clause for packed-decimal, external-decimal, and numeric-edited data items is 18.
- The maximum number of digits in a fixed-point numeric literal is 18.

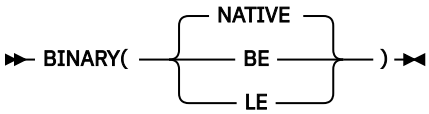
- The maximum number of digits in the arguments to NUMVAL, NUMVAL-C and is 18.
- The maximum value of the integer argument to the FACTORIAL function is 28.
- Intermediate results in arithmetic statements use *compatibility mode*.

Related concepts
Appendix C, “Intermediate results and arithmetic precision,” on page 477

BINARY

BINARY specifies the representation format of binary data items.

BINARY option syntax



Default is: BINARY(NATIVE)

Abbreviations are: None

Specify BINARY(NATIVE) to use the native binary representation format of the platform. For COBOL for Linux, this is *little-endian* format (least significant digit at the lowest address).

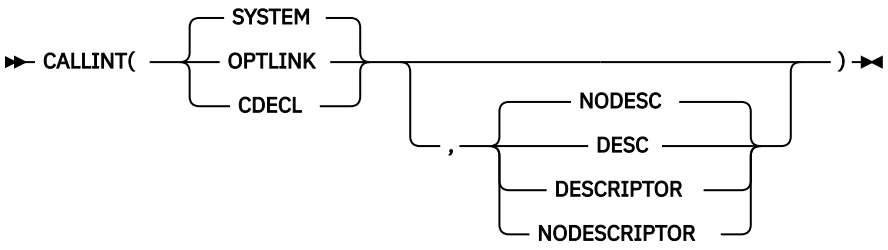
BINARY(BE) indicates that BINARY, COMP, and COMP-4 data items are represented consistently with IBM Z, that is, in *big-endian* format (most significant digit at the lowest address).

BINARY(LE) indicates that BINARY, COMP, and COMP-4 data items are represented in *little-endian* format (least significant digit at the lowest address).

CALLINT

Use CALLINT to indicate the call interface convention applicable to calls made with the CALL statement, and to indicate whether argument descriptors are to be generated.

CALLINT option syntax



Default is: CALLINT(SYSTEM,NODESC)

Abbreviations are: None

You can override this option for specific CALL statements by using the compiler directive >>CALLINT.

CALLINT has two sets of suboptions:

- Selecting a call interface convention:

  SYSTEM
  The SYSTEM suboption specifies that the call convention is that of the standard system linkage convention of the platform.

SYSTEM is the only call interface convention supported on Linux.

OPTLINK
If you code the OPTLINK suboption, the compiler generates an I-level diagnostic message, and the entire directive (not just the first keyword) is ignored.

CDECL
If you code the CDECL suboption, the compiler generates an I-level diagnostic message, and the entire directive (not just the first keyword) is ignored.

- Specifying whether the argument descriptors are to be generated:

DESC
The DESC suboption specifies that an argument descriptor is passed for each argument in a CALL statement. For more information about argument descriptors, see the Related references below.

Attention: Do not specify the DESC suboption in object-oriented programs.

DESCRIPTOR
The DESCRIPTOR suboption is synonymous with the DESC suboption.

NODESC
The NODESC suboption specifies that no argument descriptors are passed for any arguments in a CALL statement.

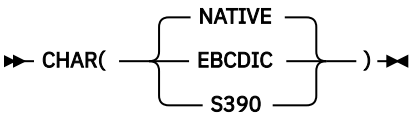
NODESCRIPTOR
The NODESCRIPTOR suboption is synonymous with the NODESC suboption.

Related references
Chapter 14, "Compiler-directing statements," on page 291

CHAR

CHAR affects the representation and runtime treatment of USAGE DISPLAY and USAGE DISPLAY-1 data items.

CHAR option syntax



Default is: CHAR(NATIVE)

Abbreviations are: None

Specify CHAR(NATIVE) to use the native character representation (the native format) of the platform. For COBOL for Linux, the native format is defined by the code page that is indicated by the locale in effect at run time. The code page can be a single-byte ASCII code page or an ASCII-based multibyte code page (UTF-8, EUC, or ASCII DBCS).

CHAR(EBCDIC) and CHAR(S390) are synonymous and indicate that DISPLAY and DISPLAY-1 data items are in the character representation of IBM Z, that is, in EBCDIC.

However, DISPLAY and DISPLAY-1 data items defined with the NATIVE phrase in the USAGE clause are not affected by the CHAR(EBCDIC) option. They are always stored in the native format of the platform.

The CHAR(EBCDIC) compiler option has the following effects on runtime processing:

- **USAGE DISPLAY and USAGE DISPLAY-1 items:** Characters in data items that are described with USAGE DISPLAY are treated as single-byte EBCDIC format. Characters in data items that are described with USAGE DISPLAY-1 are treated as EBCDIC DBCS format. (In the bullets that follow, the term

EBCDIC refers to single-byte EBCDIC format for USAGE DISPLAY and to EBCDIC DBCS format for USAGE DISPLAY-1.)

- Data that is encoded in the native format is converted to EBCDIC format upon ACCEPT from the terminal.
- EBCDIC data is converted to the native format upon DISPLAY to the terminal.
- The content of alphanumeric literals and DBCS literals is converted to EBCDIC format for assignment to data items that are encoded in EBCDIC. For the rules about the comparison of character data when the CHAR(EBCDIC) option is in effect, see the related reference below about the COLLSEQ option.
- Editing is done with EBCDIC characters.
- Padding is done with EBCDIC spaces. Group items that are used in alphanumeric operations (such as assignments and comparisons) are padded with single-byte EBCDIC spaces regardless of the definition of the elementary items within the group.
- Figurative constant SPACE or SPACES used in a VALUE clause for an assignment to, or in a relation condition with, a USAGE DISPLAY item is treated as a single-byte EBCDIC space (that is, X'40').
- Figurative constant SPACE or SPACES used in a VALUE clause for an assignment to, or in a relation condition with, a DISPLAY-1 item is treated as an EBCDIC DBCS space (that is, X'4040').
- Class tests are performed based on EBCDIC value ranges.

- **USAGE DISPLAY items:**
  - The program-name in CALL *identifier*, CANCEL *identifier*, or in a format-6 SET statement is converted to the native format if the data item referenced by *identifier* is encoded in EBCDIC.
  - The file-name in the data item referenced by *data-name* in ASSIGN USING *data-name* is converted to the native format if the data item is encoded in EBCDIC.
  - The file-name in the SORT-CONTROL special register is converted to native format before being passed to a sort or merge function. (SORT-CONTROL has the implicit definition USAGE DISPLAY.)
  - Zoned decimal data (numeric PICTURE clause with USAGE DISPLAY) and display floating-point data are treated as EBCDIC format. For example, the value of PIC S9 value "1" is X'F1' instead of X'31'.
- **Group items:** Alphanumeric group items are treated similarly to USAGE DISPLAY items. (Note that a USAGE clause for an alphanumeric group item applies to the elementary items within the group and not to the group itself.)

Hexadecimal literals are assumed to represent EBCDIC characters if the literals are assigned to, or compared with, character data. For example, X'C1' compares equal to an alphanumeric item that has the value 'A'.

Figurative constants HIGH-VALUE or HIGH-VALUES, LOW-VALUE or LOW-VALUES, SPACE or SPACES, ZERO or ZEROS, and QUOTE or QUOTES are treated logically as their EBCDIC character representations for assignments to or comparisons with data items that are encoded in EBCDIC.

In comparisons between alphanumeric USAGE DISPLAY items, the collating sequence used is the ordinal sequence of the characters based on their binary (hexadecimal) values as modified by an alternate collating sequence for single-byte characters, if specified.

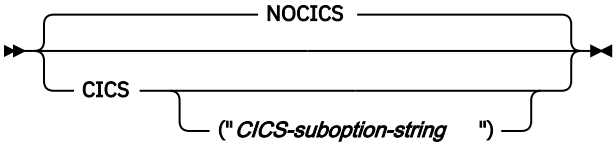
Related tasks
"Specifying the code page for character data" on page 200

Related references
"COLLSEQ" on page 256
"The encoding of XML documents" on page 346
Appendix B, "IBM Z host data format considerations," on page 475

CICS

The CICS compiler option enables the integrated CICS translator and lets you specify CICS suboptions. You must use the CICS option if your COBOL source program contains EXEC CICS statements and the program has not been processed by the separate CICS translator.

CICS option syntax



Default is: NOCICS

Abbreviations are: None

Use the CICS option only to compile CICS programs. Programs compiled with the CICS option will not run in a non-CICS environment.

If you specify the NOCICS option, any CICS statements found in the source program are diagnosed and discarded.

Use either quotation marks or apostrophes to delimit the string of CICS suboptions.

You can use the syntax shown above in either the CBL or PROCESS statement. If you use the CICS option in the cob2 or cob2_r command, only the single quotation mark (') can be used as the string delimiter: -q"CICS('*options*')".

You can partition a long CICS suboption string into multiple suboption strings in multiple CBL or PROCESS statements. The CICS suboptions are concatenated in the order of their appearance. For example, suppose that your source file mypgm.cbl has the following code:

```
cbl . . . CICS("string2") . . .
cbl . . . CICS("string3") . . .
```

When you issue the command cob2_r mypgm.cbl -q"CICS('string1')", the compiler passes the following suboption string to the integrated CICS translator:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same CICS suboption are found, the last specification of that suboption in the concatenated string prevails. The compiler limits the size of the concatenated suboption string to 4 KB.

Related concepts
"Integrated CICS translator" on page 335

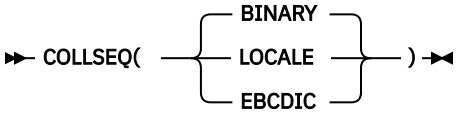
Related tasks
Chapter 18, "Developing COBOL programs for CICS," on page 329

Related references
"Conflicting
compiler options" on page 248

COLLSEQ

COLLSEQ specifies the collating sequence for comparison of alphanumeric and DBCS operands.

COLLSEQ option syntax



Default is: COLLSEQ(BINARY)

Abbreviations are: CS(L|E|BIN|B)

You can specify the following suboptions for COLLSEQ:

- COLLSEQ(EBCDIC): Use the EBCDIC collating sequence rather than the ASCII collating sequence.
- COLLSEQ(LOCALE): Use locale-based collation (consistent with the cultural conventions for collation for the locale).
- COLLSEQ(BIN): Use the hexadecimal values of the characters; the locale setting has no effect. This setting will give better runtime performance.

If you use the PROGRAM COLLATING SEQUENCE clause in your source with an alphabet-name of STANDARD-1, STANDARD-2, or EBCDIC, the COLLSEQ option is ignored for comparison of alphanumeric operands. If you specify PROGRAM COLLATING SEQUENCE is NATIVE, the COLLSEQ option applies. Otherwise, when the alphabet-name specified in the PROGRAM COLLATING SEQUENCE clause is defined with literals, the collating sequence used is that given by the COLLSEQ option, modified by the user-defined sequence given by the alphabet-name. (For details, see the related reference about the ALPHABET clause.)

The PROGRAM COLLATING SEQUENCE clause has no effect on DBCS comparisons.

The former suboption NATIVE is deprecated. If you specify the NATIVE suboption, COLLSEQ(LOCALE) is assumed.

The following table summarizes the conversion and the collating sequence that are applicable, based on the types of data (ASCII or EBCDIC) used in a comparison and the COLLSEQ option in effect when the PROGRAM COLLATING SEQUENCE clause is not specified. If it is specified, the source specification has precedence over the compiler option specification. The CHAR option affects whether data is ASCII or EBCDIC.

Table 31. ***Effect of comparand data type and collating sequence on comparisons***

Comparands	**COLLSEQ(BIN)**	**COLLSEQ(LOCALE)**	**COLLSEQ(EBCDIC)**
Both ASCII	No conversion is performed. The comparison is based on the binary value (ASCII).	No conversion is performed. The comparison is based on the current locale.	Both comparands are converted to EBCDIC. The comparison is based on the binary value (EBCDIC).
Mixed ASCII and EBCDIC	The EBCDIC comparand is converted to ASCII. The comparison is based on the binary value (ASCII).	The EBCDIC comparand is converted to ASCII. The comparison is based on the current locale.	The ASCII comparand is converted to EBCDIC. The comparison is based on the binary value (EBCDIC).

*Table 31. **Effect of comparand data type and collating sequence on comparisons** (continued)*			
Comparands	**COLLSEQ(BIN)**	**COLLSEQ(LOCALE)**	**COLLSEQ(EBCDIC)**
Both EBCDIC	No conversion is performed. The comparison is based on the binary value (EBCDIC).	The comparands are converted to ASCII. The comparison is based on the current locale.	No conversion is performed. The comparison is based on the binary value (EBCDIC).

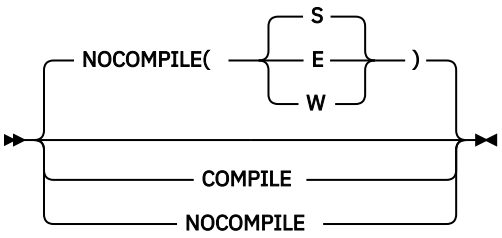
Related tasks
"Specifying the collating sequence" on page 6
"Controlling the collating sequence with a locale" on page 205

Related references
"CHAR" on page 253
ALPHABET clause (*COBOL for Linux on x86 Language Reference*)

COMPILE

Use the COMPILE option only if you want to force full compilation even in the presence of serious errors. All diagnostics and object code will be generated. Do not try to run the object code if the compilation resulted in serious errors: the results could be unpredictable or an abnormal termination could occur.

COMPILE option syntax



Default is: NOCOMPILE(S)

Abbreviations are: C | NOC

Use NOCOMPILE without any suboption to request a syntax check (only diagnostics produced, no object code). If you use NOCOMPILE without any suboption, several compiler options will have no effect because no object code will be produced, for example: LIST, OPTIMIZE, SSRANGE, and TEST.

Use NOCOMPILE with suboption W, E, or S for conditional full compilation. Full compilation (diagnosis and object code) will stop when the compiler finds an error of the level you specify (or higher), and only syntax checking will continue.

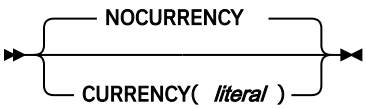
Related tasks
"Finding coding errors" on page 305

Related references
"Messages and listings for compiler-detected errors" on page 229

CURRENCY

You can use the CURRENCY option to provide an alternate default currency symbol to be used for a COBOL program. (The default currency symbol is the dollar sign ($).)

CURRENCY option syntax



Default is: NOCURRENCY

Abbreviations are: CURR | NOCURR

NOCURRENCY specifies that no alternate default currency symbol will be used.

To change the default currency symbol, specify CURRENCY(*literal*), where *literal* is a valid COBOL alphanumeric literal (optionally a hexadecimal literal) that represents a single character. The literal must not be from the following list:

- Digits zero (0) through nine (9)
- Uppercase alphabetic characters A B C D E G N P R S V X Z or their lowercase equivalents
- The space
- Special characters * + - / , . ; ( ) " =
- A figurative constant
- A null-terminated literal
- A DBCS literal
- A national literal

If your program processes only one currency type, you can use the CURRENCY option as an alternative to the CURRENCY SIGN clause for indicating the currency symbol you will use in the PICTURE clause of your program. If your program processes more than one currency type, you should use the CURRENCY SIGN clause with the WITH PICTURE SYMBOL phrase to specify the different currency sign types.

If you use both the CURRENCY option and the CURRENCY SIGN clause in a program, the CURRENCY option is ignored. Currency symbols specified in the CURRENCY SIGN clause or clauses can be used in PICTURE clauses.

When the NOCURRENCY option is in effect and you omit the CURRENCY SIGN clause, the dollar sign ($) is used as the PICTURE symbol for the currency sign.

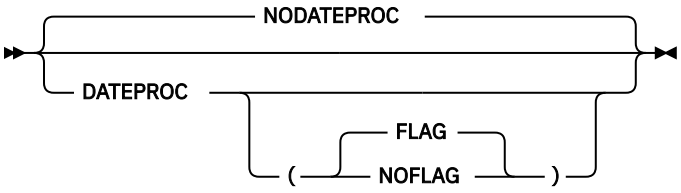
Delimiter: You can delimit the CURRENCY option literal with either quotation marks or apostrophes, regardless of the APOST | QUOTE compiler option setting.

Related tasks
"Using currency signs" on page 56

DATEPROC

Use the DATEPROC option to enable the millennium language extensions of the COBOL compiler.

DATEPROC option syntax



Default is: NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified

Abbreviations are: DP | NODP

DATEPROC(FLAG)
With DATEPROC(FLAG), the millennium language extensions are enabled, and the compiler produces a diagnostic message wherever a language element uses or is affected by the extensions. The message is usually an information-level or warning-level message that identifies statements that involve date-sensitive processing. Additional messages that identify errors or possible inconsistencies in the date constructs might be generated.

Production of diagnostic messages, and their appearance in or after the source listing, is subject to the setting of the FLAG compiler option.

DATEPROC(NOFLAG)
With DATEPROC(NOFLAG), the millennium language extensions are in effect, but the compiler does not produce any related messages unless there are errors or inconsistencies in the COBOL source.

NODATEPROC
NODATEPROC indicates that the extensions are not enabled for this compilation unit. This option affects date-related program constructs as follows:

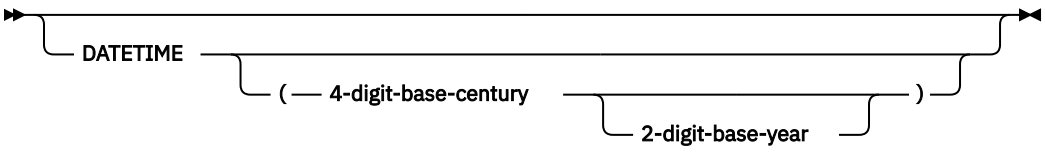
- The DATE FORMAT clause is syntax-checked, but has no effect on the execution of the program.
- The DATEVAL and UNDATE intrinsic functions have no effect. That is, the value returned by the intrinsic function is exactly the same as the value of the argument.
- The YEARWINDOW intrinsic function returns a value of zero.

Related references
"FLAG" on page 265
"YEARWINDOW" on page 288

DATETIME

The DATETIME option specifies the date window that is used for the windowing algorithm.

DATETIME option syntax



Default is: DATETIME(1900, 40)

Abbreviations are: None

4-digit-base-century
This must be the first argument. Defines the base century used for the windowing algorithm. The default value is 1900.

2-digit-base-year
This must be the second argument. Defines the base year used for the windowing algorithm. The default value is 40.

The default option DATETIME(1900, 40) results in a 100-year window of 1940 through 2039. Specifying DATETIME(1900 70) results in a 100-year window of 1970 through 2069.

Option specification:

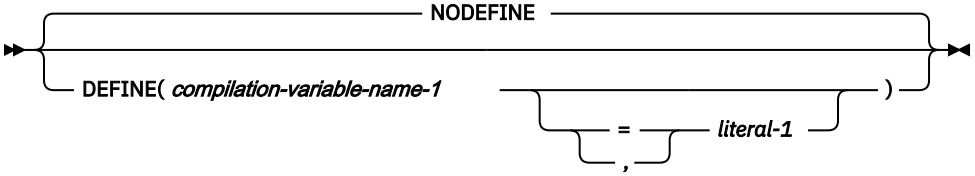
- As an option in the cob2 command, the arguments must be surrounded by single quotes; for example, DATETIME('1900 40').
- As a process statement, the arguments can be written without quotes; for example, DATETIME(1900 40).
- If the DATETIME option is not specified, or no arguments are supplied, both base-century and base-year take their default values. Base-century can be specified without a following base-year argument, in which case base-year will take its default value. If base-year is specified, base-century must also be specified.

DEFINE

Use the DEFINE compiler option to assign a literal value to a compilation variable that is defined in the program by using the DEFINE directive with the PARAMETER phrase. The literal value provided for the compilation variable in the DEFINE option is sometimes referred to as a "parameter value" for the corresponding compilation variable. Compilation variables can be used within any of the conditional compilation directives, including DEFINE, EVALUATE, and IF. When a conditional compilation variable appears in a conditional compilation directive, it is treated as a symbolic reference to the literal value it currently represents.

The DEFINE compiler option provides a way for you to assign values to compilation variables from outside the program source. If that is not needed, it is sufficient to use the DEFINE directive within program source to define compilation variables.

DEFINE option syntax



Default is: NODEFINE

Abbreviations are: DEF | NODEF

compilation-variable-name-1
The name of a compilation variable to be referenced in conditional compilation directives in the program. If no corresponding DEFINE directive with PARAMETER phrase exists for *compilation-variable-name-1* in the program, any instances of the DEFINE compiler option specified for that compilation variable are ignored. *compilation-variable-name-1* is formed according to the rules of a data-name user-defined word, except that DBCS characters are not allowed in the name. For details, see *User-defined words* in the *COBOL for Linux on x86 Language Reference*.

literal-1
The literal value that *compilation-variable-name-1* will represent symbolically in conditional compilation-related directives in the program. *literal-1* must be one of the following items:

- An alphanumeric literal, which can be specified as a regular alphanumeric literal ('abcd') or as a hex literal (x'F1F2F3'). National literals, DBCS literals, and null-terminated alphanumeric literals (Z literals) are not supported.
- An integer literal.
- A boolean literal (only B'0' and B'1' are supported).

If *literal-1* is not specified, a value of B'1' will be assigned to the compilation variable. For example, if you specify:

```
>>define foo
```

`foo` will be assigned the value B'1'.

Note: The compiler interprets certain shell scripting characters as follows:

- An equal sign (=) is interpreted to a left parenthesis, (
- A colon (:) is interpreted to a right parenthesis,)
- An underscore (_) is interpreted to a single quotation mark (')

You can add a backslash (\) escape character to prevent the interpretation and thus to pass characters in the strings. If you want the backslash (\) to represent itself (rather than as an escape character), use the double backslash (\\).

For example, to use the DEFINE option to assign the literal value 1 to a compilation variable VAR1, specify the DEFINE option as follows:

```
DEFINE(VAR1=1)
```

If VAR1 contains an equal sign, a colon, or an underscore that you want to escape from compiler's interpretation, specify the DEFINE option as follows:

```
DEFINE(VAR1\=1)
```

Multiple instances of the `DEFINE` option can be specified to define a value for multiple different compilation variables. If a single conditional compilation variable is defined more than once, the last definition of the variable will be used as the value of the corresponding conditional compilation variable. If `NODEFINE` appears after previous instances of the `DEFINE` option, the definitions for all conditional compilation variables are cancelled.

When `DEFINE` options are specified in CBL statements, they can be used only on the first program of a batch program. Therefore, if a file has multiple COBOL programs in it, there can be CBL statements with `DEFINE` options preceding the first program, but not the other programs. The `DEFINE` options specified for the first program (and DEFINE options specified as cob2 command options) apply to all programs in a file.

Related references
Conditional compilation (*COBOL for Linux on x86 Language Reference*)
DEFINE (*COBOL for Linux on x86 Language Reference*)

DIAGTRUNC

`DIAGTRUNC` causes the compiler to issue a severity-4 (Warning) diagnostic message for `MOVE` statements that have numeric receivers when the receiving data item has fewer integer positions than the sending

data item or literal. In statements that have multiple receivers, the message is issued separately for each receiver that could be truncated.

DIAGTRUNC option syntax

Default is: NODIAGTRUNC

Abbreviations are: DTR | NODTR

The diagnostic message is also issued for implicit moves associated with statements such as these:

- INITIALIZE
- READ . . . INTO
- RELEASE . . . FROM
- RETURN . . . INTO
- REWRITE . . . FROM
- WRITE . . . FROM

The diagnostic message is also issued for moves to numeric receivers from alphanumeric data-names or literal senders, except when the sending field is reference modified.

There is no diagnostic message for COMP-5 receivers, nor for binary receivers when you specify the TRUNC(BIN) option.

Related concepts
"Formats for numeric data" on page 39
"Reference modifiers" on page 100

Related references
"TRUNC" on page 283

DYNAM

Use DYNAM to cause nonnested, separately compiled programs invoked through the CALL *literal* statement to be loaded for CALL, and deleted for CANCEL, dynamically at run time.

CALL *identifier* statements always result in a runtime load of the target program and are not affected by this option.

DYNAM option syntax

Default is: NODYNAM

Abbreviations are: DYN | NODYN

The condition for the ON EXCEPTION phrase can occur for a CALL *literal* statement only if the DYNAM option is in effect.

Restriction: The DYNAM compiler option must not be used for programs that contain EXEC CICS or EXEC SQL statements.

With NODYNAM, the target program-name is resolved through the linker.

With the DYNAM option, the following statement:

```
CALL "myprogram" . . .
```

has the identical behavior to these statements:

```
MOVE "myprogram" to id-1
CALL id-1 ...
```

Related concepts
“CALL identifier and
CALL literal” on page 385
CALLINTERFACE (*COBOL for Linux on x86 Language Reference*)

Related references
“Conflicting
compiler options” on page 248

EXIT

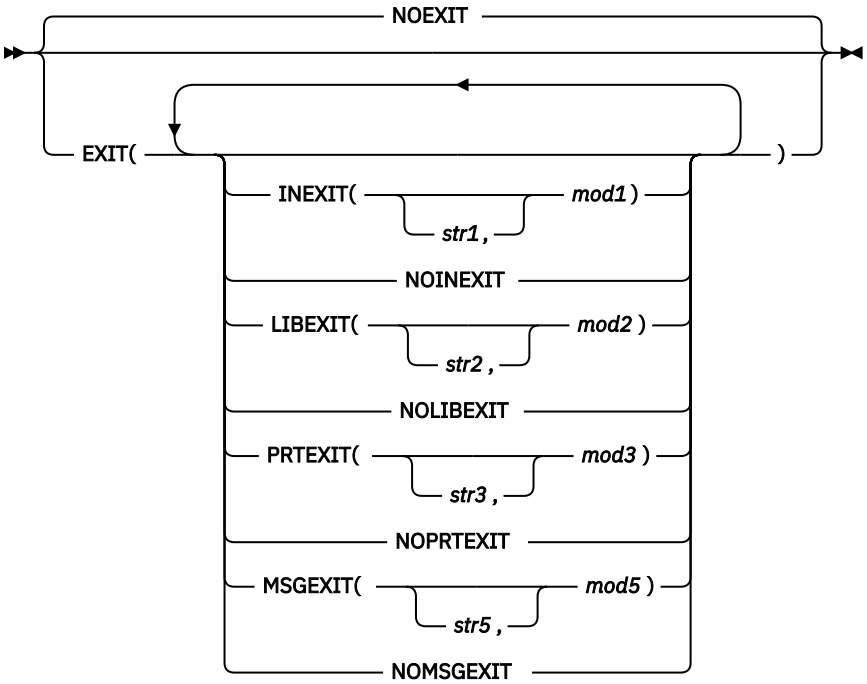
Use the EXIT option to provide user-supplied modules in place of various compiler functions.

For compiler input, use the INEXIT suboption to provide a module in place of SYSIN (primary compiler input), and use the LIBEXIT suboption to provide a module in place of SYSLIB (copy library input). For compiler output, use the PRTEXIT suboption to provide a module in place of SYSPRINT (the compiler listing file).

To customize compiler messages (change their severity or suppress them, including converting FIPS (FLAGSTD) messages to diagnostic messages to which you assign a severity), use the MSGEXIT suboption. The module that you provide to customize the messages will be called each time the compiler issues a diagnostic message or a FIPS message.

When creating your exit module, ensure that the module is linked as a shared library module before you run it with the COBOL compiler. Exit modules are invoked with the system linkage convention of the platform.

EXIT option syntax



Default is: NOEXIT

Abbreviations are: NOEX | EX(INX | NOINX, LIBX | NOLIBX, PRTX | NOPRTX, ADX | NOADX, MSGX | NOMSGX)

Option specification: You cannot specify the EXIT option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- As an option in the cob2 command
- In the COBOPT environment variable

You can specify the suboptions in any order, and can separate them by either commas or spaces. If you specify both the positive and negative form of a suboption, the form specified last takes effect. If you specify the same suboption more than once, the last one specified takes effect.

If you specify the EXIT option without providing at least one suboption (that is, you specify EXIT()), NOEXIT will be in effect.

INEXIT([*'str1'*,]*mod1*)
The compiler reads source code from a user-supplied load module (where *mod1* is the module name) instead of SYSIN.

LIBEXIT([*'str2'*,]*mod2*)
The compiler obtains copybooks from a user-supplied load module (where *mod2* is the module name) instead of *library-name* or SYSLIB. For use with either COPY or BASIS statements.

PRTEXIT([*'str3'*,]*mod3*)
The compiler passes printer-destined output to the user-supplied load module (where *mod3* is the module name) instead of SYSPRINT.

MSGEXIT(['*str5*',]*mod5*)
The compiler passes the message number, and passes the default severity of a compiler diagnostic message, or the category (as a numeric code) of a FIPS compiler message, to the user-supplied load module (where *mod5* is the module name).

The names *mod1*, *mod2*, *mod3*, *mod4*, and *mod5* can refer to the same module.

The suboptions *str1*, *str2*, *str3*, *str4*, and *str5* are character strings that are passed to the load module. These strings are optional. They can be up to 64 characters in length, and you must enclose them in a pair

of apostrophes (' '). You can use any character in the strings, but any included apostrophes must be doubled ('). Lowercase characters are folded to uppercase.

Character string formats: If one of *str1*, *str2*, *str3*, *str4*, or *str5* is specified, that string is passed to the appropriate user-exit module in the following format, where LL is a halfword (on a halfword boundary) that contains the length of the string.

LL	string

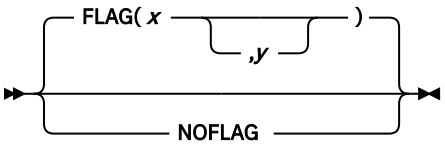
"Example: MSGEXIT user exit" on page 542

Related references
"FLAGSTD" on page 266
Appendix F, "EXIT compiler option," on page 535

FLAG

Use FLAG(*x*) to produce diagnostic messages at the end of the source listing for errors of a severity level *x* or above.

FLAG option syntax



Default is: FLAG(I,I)

Abbreviations are: F | NOF

x and *y* can be either I, W, E, S, or U.

Use FLAG(*x*,*y*) to produce diagnostic messages for errors of severity level *x* or above at the end of the source listing, with error messages of severity *y* and above to be embedded directly in the source listing. The severity coded for *y* must not be lower than the severity coded for *x*. To use FLAG(*x*,*y*), you must also specify the SOURCE compiler option.

Error messages in the source listing are set off by the embedding of the statement number in an arrow that points to the message code. The message code is followed by the message text. For example:

```
  000413     MOVE CORR WS-DATE TO HEADER-DATE
==000413==>    IGYPS2121-S     " WS-DATE " was not defined as a data-name.  . . .
```

When FLAG(*x*,*y*) is in effect, most messages of severity *y* and above are embedded in the listing after the line that caused the message. Messages with the IGYCB prefix will never be embedded in the source. (See the related reference below for information about messages for exceptions.)

Use NOFLAG to suppress error flagging. NOFLAG does not suppress error messages for compiler options.

Embedded messages

- Embedding level-U messages is not recommended. The specification of embedded level-U messages is accepted, but does not produce any messages in the source.
- The FLAG option does not affect diagnostic messages that are produced before the compiler options are processed.

- Diagnostic messages that are produced during processing of compiler options, CBL or PROCESS statements, or BASIS, COPY, or REPLACE statements are not embedded in the source listing. All such messages appear at the beginning of the compiler output.
- Diagnostic messages with the IGYCB prefix are not embedded in the source listing. All such messages appear at the end of the compiler output, regardless of the setting of the FLAG option.
- Messages that are produced during processing of the *CONTROL or *CBL statement are not embedded in the source listing.

Related references
"Messages and listings
for compiler-detected errors" on page 229

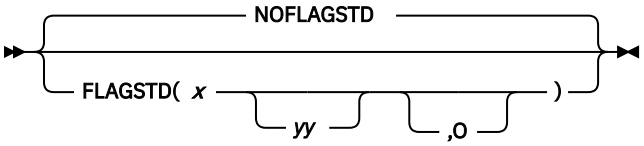
FLAGSTD

Use FLAGSTD to specify the level or subset of the 85 COBOL Standard to be regarded as conforming, and to get informational messages about the 85 COBOL Standard elements that are included in your program.

You can specify any of the following items for flagging:

- A selected Federal Information Processing Standard (FIPS) COBOL subset
- Any of the optional modules
- Obsolete language elements
- Any combination of subset and optional modules
- Any combination of subset and obsolete elements
- IBM extensions (these are flagged any time that FLAGSTD is specified, and identified as "nonconforming nonstandard")

FLAGSTD option syntax



Default is: NOFLAGSTD

Abbreviations are: None

x specifies the subset of the 85 COBOL Standard to be regarded as conforming:

M
Language elements that are not from the minimum subset are to be flagged as "nonconforming standard."

I
Language elements that are not from the minimum or the intermediate subset are to be flagged as "nonconforming standard."

H
The high subset is being used and elements will not be flagged by subset. Elements that are IBM extensions will be flagged as "nonconforming Standard, IBM extension."

yy specifies, by a single character or combination of any two, the optional modules to be included in the subset:

D
Elements from debug module level 1 are not flagged as "nonconforming standard."

N
Elements from segmentation module level 1 are not flagged as "nonconforming standard."

S
Elements from segmentation module level 2 are not flagged as "nonconforming standard."

If S is specified, N is included (N is a subset of S).

0 (the letter) specifies that obsolete language elements are flagged as "obsolete."

The informational messages appear in the source program listing, and identify:

- The element as "obsolete," "nonconforming standard," or "nonconforming nonstandard" (a language element that is both obsolete and nonconforming is flagged as obsolete only)
- The clause, statement, or header that contains the element
- The source program line and beginning location of the clause, statement, or header that contains the element
- The subset or optional module to which the element belongs

FLAGSTD requires the standard set of reserved words.

In the following example, the line number and column where a flagged clause, statement, or header occurred are shown with the associated message code and text. After that is a summary of the total number of flagged items and their type.

```
   LINE.COL CODE       FIPS MESSAGE TEXT

            IGYDS8211  Comment lines before "IDENTIFICATION DIVISION":
                       nonconforming nonstandard, IBM extension to
                       ANS/ISO 1985.

     11.14  IGYDS8111  "GLOBAL clause":  nonconforming standard, ANS/ISO
                       1985 high subset.

     59.12  IGYPS8169  "USE FOR DEBUGGING statement":  obsolete element
                       in ANS/ISO 1985.


 FIPS MESSAGES TOTAL             STANDARD      NONSTANDARD      OBSOLETE

                 3                   1               1              1
```

You can convert FIPS informational messages into diagnostic messages, and can suppress FIPS messages, by using the MSGEXIT suboption of the EXIT compiler option. For details, see the related reference about the processing of MSGEXIT, and see the related task.

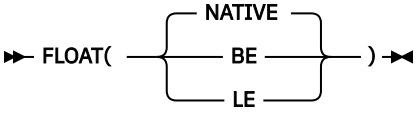
Related tasks
“Customizing compiler-message severities” on page 540

Related references
“Processing
of MSGEXIT” on page 539

FLOAT

Float specifies the representation format of floating-point data items.

FLOAT option syntax



Default is: FLOAT(NATIVE)

Abbreviations are: None

Specify FLOAT(NATIVE) to use the native floating-point representation format of the platform. For COBOL for Linux, this is *little-endian* format (least significant digit at the lowest address).

FLOAT(BE) indicates that COMP-1 and COMP-2 data items are represented consistently with IBM Z, that is, in *big-endian* format (most significant digit at the lowest address).

FLOAT(LE) indicates that COMP-1 and COMP-2 data items are represented in *little-endian* format (least significant digit at the lowest address).

Related references
Appendix B, "IBM Z host data format considerations," on page 475

LINECOUNT

Use LINECOUNT(*nnn*) to specify the number of lines to be printed on each page of the compilation listing, or use LINECOUNT(0) to suppress pagination.

LINECOUNT option syntax

Default is: LINECOUNT(60)

Abbreviations are: LC

nnn must be an integer between 10 and 255, or 0.

If you specify LINECOUNT(0), no page ejects are generated in the compilation listing.

The compiler uses three lines of *nnn* for titles. For example, if you specify LINECOUNT(60), 57 lines of source code are printed on each page of the output listing.

LIST

Use the LIST compiler option to produce a listing of the assembler-language expansion of your source code.

LIST option syntax

Default is: NOLIST

Abbreviations are: None

Any *CONTROL (or *CBL) LIST or NOLIST statements that you code in the PROCEDURE DIVISION have no effect. They are treated as comments.

The assembler listing is written to a file that has the same name as the source program but has the suffix .wlist.

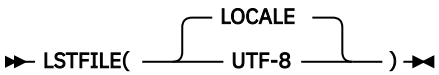
Related tasks
"Getting listings" on page 309

Related references
*CONTROL (*CBL) statement (*COBOL for Linux on x86 Language Reference*)

LSTFILE

Specify LSTFILE(LOCALE) to have your generated compiler listing encoded in the code page specified by the locale in effect. Specify LSTFILE(UTF-8) to have your generated compiler listing encoded in UTF-8.

LSTFILE option syntax



Default is: LSTFILE(LOCALE)

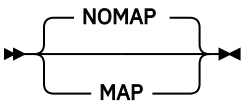
Abbreviations are: LST

Related references
Chapter 11, “Setting the locale,” on page 199

MAP

Use MAP to produce a listing of the items defined in the DATA DIVISION.

MAP option syntax



Default is: NOMAP

Abbreviations are: None

The output includes the following items:

- DATA DIVISION map
- Nested program structure map, and program attributes
- Size of the program's WORKING-STORAGE and LOCAL-STORAGE

If you want to limit the MAP output, use *CONTROL MAP or NOMAP statements in the DATA DIVISION. Source statements that follow *CONTROL NOMAP are not included in the listing until a *CONTROL MAP statement switches the output back to normal MAP format. For example:

```
*CONTROL NOMAP            *CBL NOMAP
    01  A                     01  A
    02  B                     02  B
*CONTROL MAP              *CBL MAP
```

When the MAP option is in effect, you also get an embedded MAP report in the source code listing. The condensed MAP information is shown to the right of data-name definitions in the WORKING-STORAGE SECTION, FILE SECTION, LOCAL-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION. When both XREF data and an embedded MAP summary are on the same line, the embedded MAP summary is listed first.

“Example: MAP output” on page 314

Related concepts
Chapter 16, “Debugging,” on page 301

Related tasks
“Getting listings” on page 309

Related references
*CONTROL (*CBL) statement (*COBOL for Linux on x86 Language Reference*)

MAXMEM

Use MAXMEM in conjunction with OPTIMIZE to limit the amount of memory used by the compiler for local tables of specific, memory-intensive optimizations to *size* KB. If that memory is insufficient for a particular optimization, the scope of the optimization is reduced.

MAXMEM option syntax

Default is: MAXMEM(2048)

Abbreviations are: None

A value of -1 permits each optimization to take as much memory as it needs without checking for limits. Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this amount might exceed available system resources.

Usage notes

- The limit set by MAXMEM is the amount of memory for specific optimizations, not for the compiler as a whole. Tables required during the entire compilation process are not affected or included in this limit.
- Setting a large limit has no negative effect on the compilation of source files where the compiler needs less memory.
- Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that the compiler might finish before finding all opportunities to improve performance.
- Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Depending on the source file being compiled, the size of the subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high might lead to page-space exhaustion. In particular, specifying MAXMEM(-1) lets the compiler try to use an unlimited amount of storage, which in the worst case could exhaust the resources of the machine.

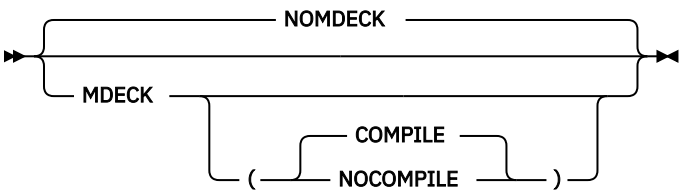
Related references
"OPTIMIZE" on page 273

MDECK

The MDECK compiler option specifies that a copy of the updated input source after library processing (that is, the result of COPY, BASIS, REPLACE, and EXEC SQL INCLUDE statements) is written to a file.

The MDECK output is written in the current directory to a file that has the same name as the COBOL source file and a suffix of .dek.

MDECK option syntax



Default is: NOMDECK

Abbreviations are: NOMD | MD | MD(C | NOC)

Option specification:

You cannot specify the MDECK option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- As an option in the cob2 command
- In the **COBOPT** environment variable
- In the compopts attribute of the configuration (.cfg) file

Suboptions:

- When MDECK(COMPILE) is in effect, compilation continues normally after library processing and generation of the MDECK output file have completed, subject to the setting of the COMPILE | NOCOMPILE option.
- When MDECK(NOCOMPILE) is in effect, compilation is terminated after syntax checking has completed and the expanded source program file has been written. The compiler does no code generation regardless of the setting of the COMPILE option.

If you specify MDECK with no suboption, MDECK(COMPILE) is implied.

Contents of the MDECK output file:

If you use the MDECK option with programs that contain EXEC CICS or EXEC SQL statements, these EXEC statements are included in the MDECK output as is. However, if you compile using the SQL option, the corresponding EXEC SQL INCLUDE statements are expanded in the MDECK output.

CBL, PROCESS, *CONTROL, and *CBL card images are passed to the MDECK output file in the proper locations.

For a batch compilation (multiple COBOL source programs in a single input file), a single MDECK output file that contains the complete expanded source is created.

Any SEQUENCE compiler-option processing is reflected in the MDECK file.

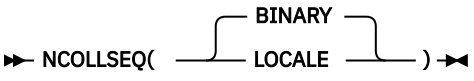
COPY statements are included in the MDECK file as comments.

Related references
“Stanzas in the configuration file” on page 227
“Conflicting
compiler options” on page 248
Chapter 14, “Compiler-directing
statements,” on page 291

NCOLLSEQ

NCOLLSEQ specifies the collating sequence for comparison of class national operands.

NCOLLSEQ option syntax



Default is: NCOLLSEQ(BINARY)

Abbreviations are: NCS(L|BIN|B)

NCOLLSEQ(BIN) uses the hexadecimal values of the character pairs.

NCOLLSEQ(LOCALE) uses the algorithm for collation order that is associated with the locale value that is in effect.

Related tasks
"Comparing two class national operands" on page 191
"Controlling the collating sequence with a locale" on page 205

NSYMBOL

The NSYMBOL option controls the interpretation of the N symbol used in literals and PICTURE clauses, indicating whether national or DBCS processing is assumed.

NSYMBOL option syntax

Default is: NSYMBOL(NATIONAL)

Abbreviations are: NS(NAT | DBCS)

With NSYMBOL(NATIONAL):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE NATIONAL clause is specified.
- Literals of the form N". . ." or N'. . .' are treated as national literals.

With NSYMBOL(DBCS):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE DISPLAY-1 clause is specified.
- Literals of the form N". . ." or N'. . .' are treated as DBCS literals.

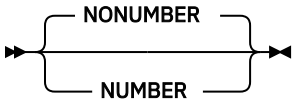
The NSYMBOL(DBCS) option provides compatibility with previous releases of IBM COBOL, and the NSYMBOL(NATIONAL) option makes the handling of the above language elements consistent with the 2002 COBOL Standard in this regard.

NSYMBOL(NATIONAL) is recommended for applications that use Unicode data.

NUMBER

Use the NUMBER compiler option if you have line numbers in your source code and want those numbers to be used in error messages and SOURCE, MAP, LIST, and XREF listings.

NUMBER option syntax



Default is: NONUMBER

Abbreviations are: NUM | NONUM

If you request NUMBER, the compiler checks columns 1 through 6 to make sure that they contain only numbers and that the numbers are in numeric collating sequence. (In contrast, SEQUENCE checks the characters in these columns according to EBCDIC collating sequence.) When a line number is found to be out of sequence, the compiler assigns to it a line number with a value one higher than the line number of the preceding statement. The compiler flags the new value with two asterisks and includes in the listing a message indicating an out-of-sequence error. Sequence-checking continues with the next statement, based on the newly assigned value of the previous line.

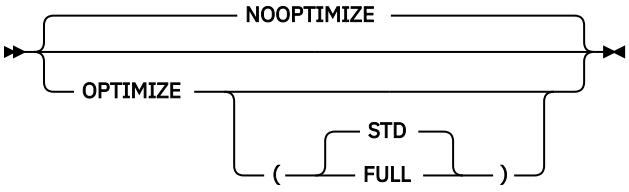
If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the copybook line numbers are coordinated.

Use NONUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With NONUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

OPTIMIZE

Use OPTIMIZE to reduce the run time of your object program. Optimization might also reduce the amount of storage your object program uses. Optimizations performed include the propagation of constants, instruction scheduling, and the elimination of computations whose results are never used.

OPTIMIZE option syntax



Default is: NOOPTIMIZE

Abbreviations are: OPT | NOOPT

If OPTIMIZE is specified without any suboptions, OPTIMIZE(STD) is in effect.

The FULL suboption requests that, in addition to the optimizations performed with OPT(STD), the compiler discard unreferenced data items from the DATA DIVISION and suppress generation of code to initialize these data items to the values in their VALUE clauses. When OPT(FULL) is in effect, all unreferenced level-77 items and elementary level-01 items are discarded. In addition, level-01 group items are discarded if none of their subordinate items are referenced. The deleted items are shown in the listing. If the MAP option is in effect, a BL number of XXXXX in the data map information indicates that the data item was discarded.

Recommendation: Use OPTIMIZE(FULL) for database applications. It can make a huge performance improvement, because unused constants included by the associated COPY statements are eliminated. However, if your database application depends on unused data items, see the recommendations below.

Unused data items: Do not use OPT(FULL) if your programs depend on making use of unused data items. In the past, this was commonly done in two ways:

- A technique sometimes used in old OS/VS COBOL programs was to place an unreferenced table after a referenced table and use out-of-range subscripts on the first table to access the second table. To determine whether your programs use this technique, use the SSRANGE compiler option with the CHECK(ON) runtime option. To work around this problem, use the ability of newer COBOL to code large tables and use just one table.
- Place eye-catcher data items in the WORKING-STORAGE SECTION to identify the beginning and end of the program data or to mark a copy of a program for a library tool that uses the data to identify the version of a program. To solve this problem, initialize these items with PROCEDURE DIVISION statements rather than VALUE clauses. With this method, the compiler will consider these items used and will not delete them.

The OPTIMIZE option is turned off in the case of a severe-level error or higher.

Related concepts
"Optimization" on page 450

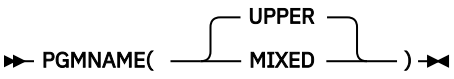
Related references
"Conflicting

compiler options” on page 248
“MAXMEM” on page 270

PGMNAME

The PGMNAME option controls the handling of program-names and entry-point names.

PGMNAME option syntax



Default is: PGMNAME(UPPER)

Abbreviations are: PGMN(LU|LM)

For compatibility with COBOL for OS/390® & VM, LONGMIXED and LONGUPPER are also supported.

LONGUPPER can be abbreviated as UPPER, LU, or U. LONGMIXED can be abbreviated as MIXED, LM, or M.

COMPAT: If you specify PGMNAME(COMPAT), PGMNAME(UPPER) will be set, and you will receive a warning message.

PGMNAME controls the handling of names used in the following contexts:

- Program-names defined in the PROGRAM-ID paragraph
- Program entry-point names in the ENTRY statement
- Program-name references in:
  - CALL statements that reference nested programs, statically linked programs, or shared libraries
  - SET *procedure-pointer* or *function-pointer* statements that reference statically linked programs or shared libraries
  - CANCEL statements that reference nested programs

PGMNAME(UPPER)

With PGMNAME(UPPER), program-names that are specified in the PROGRAM-ID paragraph as COBOL user-defined words must follow the normal COBOL rules for forming a user-defined word:

- The program-name can be up to 30 characters in length.
- All the characters used in the name must be alphabetic, digits, the hyphen, or the underscore.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.
- The underscore cannot be used as the first character.

When a program-name is specified as a literal, in either a definition or a reference, then:

- The program-name can be up to 160 characters in length.
- All the characters used in the name must be alphabetic, digits, the hyphen, or the underscore.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.
- The underscore can be used in any position.

External program-names are processed with alphabetic characters folded to uppercase.

PGMNAME(MIXED)

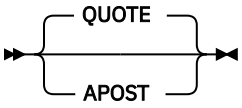
With PGMNAME(MIXED), program-names are processed as is, without truncation, translation, or folding to uppercase.

With PGMNAME(MIXED), all program-name definitions must be specified using the literal format of the program-name in the PROGRAM-ID paragraph or ENTRY statement.

APOST/QUOTE

Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more apostrophe (') characters. Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters.

APOST/QUOTE option syntax



Default is: QUOTE

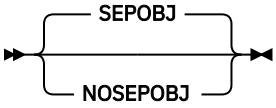
Abbreviations are: Q | APOST

Delimiters: You can use either quotation marks (") or apostrophes (') as literal delimiters regardless of whether the APOST or QUOTE option is in effect. The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal.

SEPOBJ

SEPOBJ specifies whether each of the outermost COBOL programs in a batch compilation is to be generated as a separate object file rather than as a single object file.

SEPOBJ option syntax



Default is: SEPOBJ

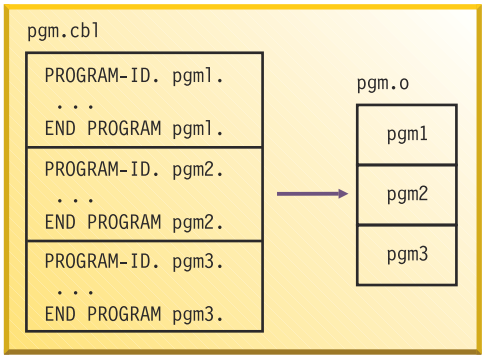
Abbreviations are: None

Batch compilation

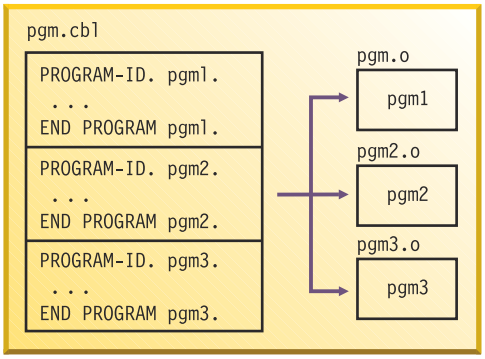
When multiple outermost programs (nonnested programs) are compiled with a single batch invocation of the compiler, the number of files produced for the object program output of the batch compilation depends on the compiler option SEPOBJ.

Assume that the COBOL source file pgm.cbl contains three outermost COBOL programs named pgm1, pgm2, and pgm3. The following figures illustrate whether the object program output is generated as one file (with NOSEPOBJ) or three files (with SEPOBJ).

Batch compilation with NOSEPOBJ



Batch compilation with SEPOBJ



Usage notes

- The SEPOBJ option is required to conform to 85 COBOL Standard where pgm2 or pgm3 in the above example is called using CALL *identifier* from another program.
- If NOSEPOBJ is in effect, the object files are given the name of the source file but with suffix .o. If SEPOBJ is in effect, the names of the object files are based on the PROGRAM-ID name with suffix .o.
- The programs called using CALL *identifier* must be referred to by the names of the object files (rather than the PROGRAM-ID names) where PROGRAM-ID and the object file name do not match.

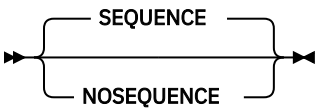
You must give the object file a valid file-name for the platform and the file system.

SEQUENCE

When you use SEQUENCE, the compiler examines columns 1 through 6 to check that the source statements are arranged in ascending order according to their ASCII collating sequence. The compiler issues a diagnostic message if any statements are not in ascending order.

Source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages.

SEQUENCE option syntax



Default is: SEQUENCE

Abbreviations are: SEQ | NOSEQ

If you use COPY statements with the SEQUENCE option in effect, be sure that your source program's sequence fields and the copybook sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than ASCII, collating sequence.

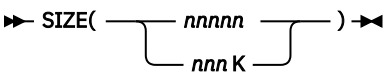
Use NOSEQUENCE to suppress this checking and the diagnostic messages.

Related tasks
"Finding line sequence problems" on page 305

SIZE

Use SIZE to indicate the amount of main storage to be made available to the compiler front end for compilation. The compiler front end is the phase of compilation that occurs before code generation and optimization.

SIZE option syntax



Default is: 8388608 bytes (approximately 8 MB)

Abbreviations are: SZ

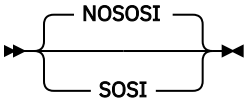
nnnnn specifies a decimal number, which must be at least 800768.

*nnn*K specifies a decimal number in 1 KB increments, where 1 KB = 1024 bytes. The minimum acceptable value is 782K.

SOSI

The SOSI option affects the treatment of values X'1E' and X'1F' in comments; alphanumeric, national, and DBCS literals; and in DBCS user-defined words.

SOSI option syntax



Default is: NOSOSI

Abbreviations are: None

NOSOSI
With NOSOSI, character positions that have values X'1E' and X'1F' are treated as data characters.

NOSOSI conforms to 85 COBOL Standard.

SOSI
With SOSI, shift-out (SO) and shift-in (SI) control characters delimit ASCII DBCS character strings in COBOL source programs. The SO and SI characters have the encoded values of X'1E' and X'1F', respectively.

SO and SI characters have no effect on COBOL for Linux source code, except to act as placeholders for host DBCS SO and SI characters to ensure proper data handling when remote files are converted from EBCDIC to ASCII.

When the SOSI option is in effect, in addition to existing rules for COBOL for Linux, the following rules apply:

- All DBCS character strings (in user-defined words, DBCS literals, alphanumeric literals, national literals, and in comments) must be delimited by the SO and SI characters.

- User-defined words cannot contain both DBCS and SBCS characters.
- The maximum length of a DBCS user-defined word is 14 DBCS characters.
- Double-byte uppercase alphabetic letters are not equivalent to the corresponding double-byte lowercase letters when used in user-defined words.
- A DBCS user-defined word must contain at least one letter that does not have its counterpart in a single-byte representation.
- Double-byte representations of single-byte characters for A-Z, a-z, 0-9, the hyphen (-), and the underscore (_) can be included within a DBCS user-defined word. Rules applicable to these characters in single-byte representation apply to these characters in double-byte representation. For example, in a user-defined word, the hyphen cannot appear as the first or the last character, and the underscore cannot appear as the first character.
- For DBCS and national literals that contain X'1E' or X'1F' values, the following rules apply when the SOSI compiler option is in effect:
  - Character positions with X'1E' and X'1F' are treated as SO and SI characters.
  - Character positions with X'1E' and X'1F' are included in the character string in national hexadecimal notation and removed in basic notation.
- For alphanumeric literals that contain X'1E' or X'1F' values, the following rules apply when the SOSI compiler option is in effect:
  - Character positions with X'1E' and X'1F' are treated as SO and SI characters.
  - Character positions with X'1E' and X'1F' are included in the character string in hexadecimal notation and removed in basic and null-terminated notation.
- To embed DBCS quotation marks within an N-literal delimited by quotation marks, use two consecutive DBCS quotation marks to represent a single DBCS quotation mark. Do not include a single DBCS quotation mark in an N-literal if the literal is delimited by quotation marks. The same rule applies to single quotation marks.
- The SHIFT-OUT and SHIFT-IN special registers are defined with X'0E' and X'0F' regardless of whether the SOSI option is in effect.

In general, host COBOL programs that are sensitive to the encoded values for the SO and SI characters will not have the same behavior on the Linux workstation.

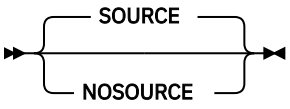
Related tasks
“Handling differences in ASCII multibyte and EBCDIC DBCS strings” on page 378

Related references
Character-strings (*COBOL for Linux on x86 Language Reference*)

SOURCE

Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.

SOURCE option syntax



Default is: SOURCE

Abbreviations are: S | NOS

You must specify SOURCE if you want embedded messages in the source listing.

Use NOSOURCE to suppress the source code from the compiler output listing.

If you want to limit the SOURCE output, use *CONTROL SOURCE or NOSOURCE statements in your PROCEDURE DIVISION. Source statements that follow a *CONTROL NOSOURCE statement are not included in the listing until a subsequent *CONTROL SOURCE statement switches the output back to normal SOURCE format.

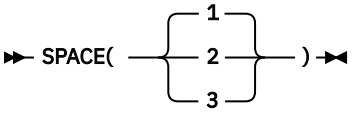
"Example: MAP output" on page 314

Related references
*CONTROL (*CBL) statement (*COBOL for Linux on x86 Language Reference*)

SPACE

Use SPACE to select single-, double-, or triple-spacing in your source code listing.

SPACE option syntax



Default is: SPACE(1)

Abbreviations are: None

SPACE has meaning only when the SOURCE compiler option is in effect.

Related references
"SOURCE" on page 278

SPILL

This option specifies the number of KB set aside for the register spill area. If the program being compiled is very complex or large, this option might be required.

SPILL option syntax

Default is: SPILL(512)

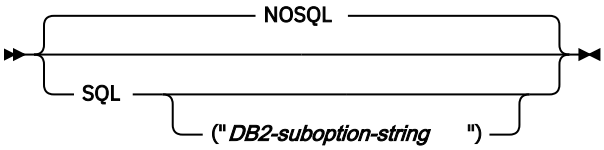
Abbreviations are: None

The spill size, *n*, is any integer between 96 and 32704.

SQL

Use the SQL compiler option to enable the Db2 coprocessor and to specify Db2 suboptions. You must specify the SQL option if your COBOL source program contains SQL statements and the program has not been processed by the Db2 precompiler.

SQL option syntax



Default is: NOSQL

Abbreviations are: None

If NOSQL is in effect, any SQL statements found in the source program are diagnosed and discarded.

Use either quotation marks or single quotation marks to delimit the string of Db2 suboptions.

You can use the syntax shown above in either the CBL or PROCESS statement. If you use the SQL option in the cob2 command, only the single quotation mark (') can be used as the suboption string delimiter: -q"SQL('*suboptions*')".

Note: The compiler interprets certain shell scripting characters as follows:

- An equal sign (=) is interpreted to a left parenthesis, (
- A colon (:) is interpreted to a right parenthesis,)
- An underscore (_) is interpreted to a single quotation mark (')

You can add a backslash (\) escape character to prevent the interpretation and thus to pass characters in the strings. If you want the backslash (\) to represent itself (rather than as an escape character), use the double backslash (\\).

For example, if you want to work with the integrated Db2 coprocessor and use the DEFERRED_PREPARE precompile option, specify the SQL option as follows:

```
SQL('... DEFERRED\_PREPARE ...')
```

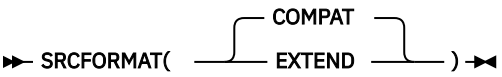
Related tasks
Chapter 17, “Programming for a Db2 environment,” on page 325
“Compiling with the SQL option” on page 327
“Separating Db2 suboptions” on page 328

Related references
“Conflicting
compiler options” on page 248

SRCFORMAT

Use SRCFORMAT to indicate whether your COBOL source conforms to 72-column fixed source format or to 252-column extended source format.

SRCFORMAT option syntax



Default is: SRCFORMAT(COMPAT)

Abbreviations are: SF(C|E)

SRCFORMAT(COMPAT) indicates that each source line of the primary compilation input and of any included COPY text ends at column 72. If a source line is shorter than 72 bytes, space characters are logically added to the source line up to the maximum of 72 bytes. If a source line is longer than 72 bytes, only the first 72 bytes are used as program source. (Bytes 73 through 80, if provided, are assumed to contain serial numbers; they are printed in the compiler listing but otherwise ignored.)

SRCFORMAT(EXTEND) indicates that each source line of the primary compilation input and of any included COPY text ends at column 252. If a source line is shorter than 252 bytes, space characters are logically added to the source line up to the maximum of 252 bytes. If a source line is longer than 252 bytes, only the first 252 bytes are used as program source; the rest are ignored. (In extended source format, there is no provision for serial numbers.)

In either format, columns 1 through 6 are interpreted as sequence numbers.

Option specification: You cannot specify the SRCFORMAT option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- As an option in the cob2 command
- In the COBOPT environment variable
- In the compopts attribute of the configuration (.cfg) file

A source conversion utility, scu, is available to help convert non-IBM or free-format COBOL source so that it can be compiled by COBOL for Linux. To see a summary of the scu functions, type the command scu -h. For further details, see the man page for scu, or see the appropriate related reference.

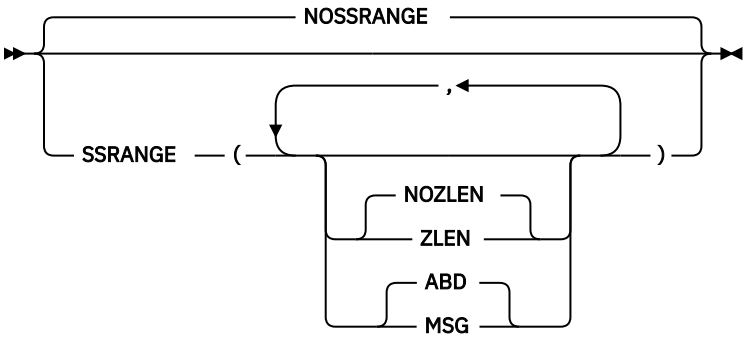
Restriction: Extended source format is not compatible with the stand-alone Db2 precompiler or the separate CICS translator.

Related references
"Stanzas in the configuration file" on page 227
"cob2 options" on page 230
Reference format (*COBOL for Linux on x86 Language Reference*)
Source conversion utility (scu) (*COBOL for Linux on x86 Language Reference*)

SSRANGE

Use SSRANGE to generate code that checks for out-of-range storage references.

SSRANGE option syntax



Default is: NOSSRANGE

Suboption default is: NOZLEN,ABD if only SSRANGE is specified.

Abbreviations are: SSR | NOSSR

SSRANGE generates code that checks whether subscripts, including ALL subscripts, or indexes try to reference areas outside the region of their associated tables. Each subscript or index is not individually checked for validity. Instead, the effective address is checked to ensure that it does not reference outside the table.

If you specify SSRANGE with no suboptions, it will be accepted as a specification of SSRANGE(NOZLEN,ABD).

Note: If the SSRANGE option is in effect, range checks will be generated by the compiler and the checks will always be conducted at run time. You cannot disable the compiled-in range checks at run time by specifying the runtime option CHECK(OFF).

Variable-length items are also checked to ensure that references are within their maximum defined length.

Reference modification expressions are checked to ensure that:

- The starting position is greater than or equal to 1.

- The starting position is not greater than the current length of the subject data item.
- The starting position and length value (if specified) do not reference an area beyond the end of the subject data item.
- The length value (if specified) is greater than or equal to 1.

The ZLEN and NOZLEN suboptions control how the compiler checks reference modification lengths:

- If ZLEN is in effect, the compiler will generate code to ensure that reference modification lengths are greater than or equal to zero. Zero-length reference modification specifications will not get an SSRANGE error at run time.
- If NOZLEN is in effect, the compiler will generate code to ensure that reference modification lengths are greater than or equal to 1. Zero-length reference modification specifications will get an SSRANGE error at run time. This is compatible with how SSRANGE behaved in previous COBOL versions.

The MSG and ABD suboptions control the runtime behavior of the COBOL program when a range check fails.

- If MSG is in effect and a range check fails, a runtime warning message will be issued. This means that the program will continue executing and might potentially identify other out-of-range conditions.
- If ABD is in effect and a range check fails, the first out-of-range condition will result in a runtime error message and the program will ABEND. You can find the next potential out-of-range condition by fixing the first out-of-range condition and then recompiling and running the program again. To identify all other potential out-of-range conditions, you might need to repeat this process several times.

For unbounded groups or their subordinate items, checking is done only for reference modification expressions. Subscripted or indexed references to tables subordinate to an unbounded group are not checked.

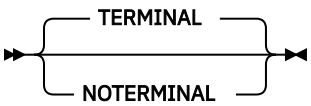
Related concepts
"Reference modifiers" on page 100

Related tasks
"Checking for valid ranges" on page 306

TERMINAL

Use TERMINAL to send progress and diagnostic messages to the display device.

TERMINAL option syntax



Default is: TERMINAL

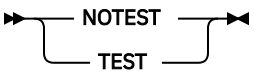
Abbreviations are: TERM | NOTERM

Use NOTERMINAL if you do not want this additional output.

TEST

Use TEST to produce object code that contains symbol and statement information that enables the debugger to perform symbolic source-level debugging.

TEST option syntax



Default is: NOTEST

Abbreviations are: None

Use NOTEST if you do not want to generate object code that has debugging information. Programs compiled with NOTEST run with the debugger, but have limited debugging support.

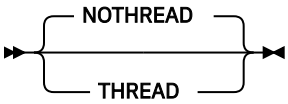
If you use the WITH DEBUGGING MODE clause, the TEST option is turned off. TEST will appear in the list of options, but a diagnostic message is issued to advise you that because of the conflict, TEST was not in effect.

Note: COBOL for Linux does not currently provide a debugger. Use the NOTEST option for now.

THREAD

The THREAD option is accepted and ignored. It is no longer needed to indicate that a COBOL program is to be enabled for execution in a run unit that has multiple threads.

THREAD option syntax



Default is: NOTHREAD

Abbreviations are: None

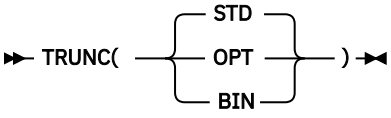
Related tasks

"Compiling from the command line" on page 223

TRUNC

TRUNC affects the way that binary data is truncated during moves and arithmetic operations.

TRUNC option syntax



Default is: TRUNC(STD)

Abbreviations are: None

TRUNC has no effect on COMP-5 data items; COMP-5 items are handled as if TRUNC(BIN) is in effect regardless of the TRUNC suboption specified.

TRUNC(STD)
TRUNC(STD) applies only to USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. When TRUNC(STD) is in effect, the final result of an arithmetic expression, or the sending field in the MOVE statement, is truncated to the number of digits in the PICTURE clause of the BINARY receiving field.

TRUNC(OPT)
TRUNC(OPT) is a performance option. When TRUNC(OPT) is in effect, the compiler assumes that data conforms to PICTURE specifications in USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. The results are manipulated in the most optimal way, either truncating to the number of digits in the PICTURE clause, or to the size of the binary field in storage (halfword, fullword, or doubleword).

Tip: Use the TRUNC(OPT) option only if you are sure that the data being moved into the binary areas will not have a value with larger precision than that defined by the PICTURE clause for the binary item. Otherwise, unpredictable results could occur. This truncation is performed in the most efficient manner possible; therefore, the results are dependent on the particular code sequence generated. It is not possible to predict the truncation without seeing the code sequence generated for a particular statement.

TRUNC(BIN)
The TRUNC(BIN) option applies to all COBOL language that processes USAGE BINARY data. When TRUNC(BIN) is in effect, all binary items (USAGE COMP, COMP-4, or BINARY) are handled as native hardware binary items, that is, as if they were each individually declared USAGE COMP-5:

- BINARY receiving fields are truncated only at halfword, fullword, or doubleword boundaries.
- BINARY sending fields are handled as halfwords, fullwords, or doublewords when the receiver is numeric; TRUNC(BIN) has no effect when the receiver is not numeric.
- The full binary content of fields is significant.
- DISPLAY will convert the entire content of binary fields with no truncation.

Recommendations: TRUNC(BIN) is the recommended option for programs that use binary values set by other products. Other products, such as Db2 and C/C++, might place values in COBOL binary data items that do not conform to the PICTURE clause of the data items. You can use TRUNC(OPT) with CICS programs provided that your data conforms to the PICTURE clause for your BINARY data items.

USAGE COMP-5 has the effect of applying TRUNC(BIN) behavior to individual data items. Therefore, you can avoid the performance overhead of using TRUNC(BIN) for every binary data item by specifying COMP-5 on only some of the binary data items, such as those data items that are passed to non-COBOL programs or other products and subsystems. The use of COMP-5 is not affected by the TRUNC suboption in effect.

Large literals in VALUE clauses: When you use the compiler option TRUNC(BIN), numeric literals specified in VALUE clauses for binary data items (COMP, COMP-4, or BINARY) can generally contain a value of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) rather than being limited to the value implied by the number of 9s in the PICTURE clause.

Note: When TRUNC(BIN) and NUMCHECK(BIN) are both in effect and an error message or an abend is generated, if you intend to switch to TRUNC(STD|OPT) later for better performance, you must correct the data; if not, you can turn off NUMCHECK(BIN) to reduce the execution time of the application and avoid an error message or an abend.

TRUNC example 1

```
01  BIN-VAR     PIC S99 USAGE BINARY.
. . .
    MOVE 123451 to BIN-VAR
```

The following table shows values of the data items after the MOVE statement.

Data item	Decimal	Hex	Display
Sender	123451	3B|E2|01|00	123451
Receiver TRUNC(STD)	51	33|00	51
Receiver TRUNC(OPT)	-7621	3B|E2	2J
Receiver TRUNC(BIN)	-7621	3B|E2	762J

A halfword of storage is allocated for BIN-VAR. The result of this MOVE statement if the program is compiled with the TRUNC(STD) option is 51; the field is truncated to conform to the PICTURE clause.

If you compile the program with TRUNC(BIN), the result of the MOVE statement is -7621. The reason for the unusual result is that nonzero high-order digits are truncated. Here, the generated code sequence would merely move the lower halfword quantity X'E23B' to the receiver. Because the new truncated value overflows into the sign bit of the binary halfword, the value becomes a negative number.

It is better not to compile this MOVE statement with TRUNC(OPT), because 123451 has greater precision than the PICTURE clause for BIN-VAR. With TRUNC(OPT), the results are again -7621. This is because the best performance was obtained by not doing a decimal truncation.

TRUNC example 2

```
01  BIN-VAR     PIC 9(6)  USAGE BINARY
. . .
    MOVE 1234567891 to BIN-VAR
```

The following table shows values of the data items after the MOVE statement.

Data item	Decimal	Hex	Display
Sender	1234567891	D3|02|96|49	1234567891
Receiver TRUNC(STD)	567891	53|AA|08|00	567891
Receiver TRUNC(OPT)	567891	53|AA|08|00	567891
Receiver TRUNC(BIN)	1234567891	D3|02|96|49	1234567891

When you specify TRUNC(STD), the sending data is truncated to six integer digits to conform to the PICTURE clause of the BINARY receiver.

When you specify TRUNC(OPT), the compiler assumes the sending data is not larger than the PICTURE clause precision of the BINARY receiver. The most efficient code sequence in this case is truncation as if TRUNC(STD) were in effect.

When you specify TRUNC(BIN), no truncation occurs because all of the sending data fits into the binary fullword allocated for BIN-VAR.

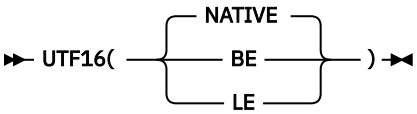
Related concepts
“Formats for numeric data” on page 39

Related references
VALUE clause (*COBOL for Linux on x86 Language Reference*)

UTF16

UTF16 specifies the representation format of UTF-16 data items.

UTF16 option syntax



Default is: UTF16(NATIVE)

Abbreviations are: None

Specify UTF16(NATIVE) to use the native UTF-16 representation format of the platform. For COBOL for Linux, this is *little-endian* format (least significant digit at the lowest address).

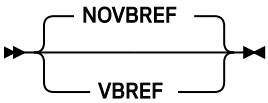
UTF16(BE) indicates that UTF-16 data items are represented consistently with IBM Z, that is, in *big-endian* format (most significant digit at the lowest address).

UTF16(LE) indicates that UTF-16 data items are represented in *little-endian* format (least significant digit at the lowest address).

VBREF

Use VBREF to get a cross-reference between all statements used in the source program and the line numbers in which they are used. VBREF also produces a summary of the number of times each statement was used in the program.

VBREF option syntax



Default is: NOVBREF

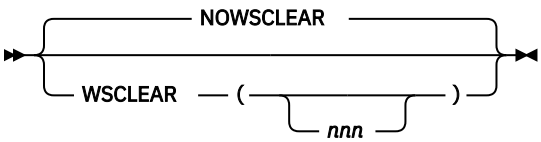
Abbreviations are: None

Use NOVBREF for more efficient compilation.

WSCLEAR

Use WSCLEAR to clear a program's non-EXTERNAL data items in WORKING-STORAGE to binary zeros at initialization. The storage is cleared before any VALUE clauses are applied.

WSCLEAR option syntax



Default is: NOWSCLEAR

Abbreviations are: None

Use NOWSCLEAR to bypass the storage-clearing process.

nnn is any integer from 0 to 255. WSCLEAR without the suboption is the same as WSCLEAR(*0*).

You can specify the WSCLEAR option either in a command line or in a COBOL statement. However, the WSCLEAR option that is specified in the COBOL statement takes precedence over the option specified in the command line.

- To specify WSCLEAR with the *nnn* suboption in the command line, use the format of -qwsclear*(nnn)*. In this case, the command processor scans characters that are in the range of 0 to 255 only, and other excessive characters are ignored. No error message is issued.

  For example, if you specify -qwsclear*(99999)*, the command processor takes WSCLEAR*(99)* only.
- To specify WSCLEAR with the *nnn* suboption in COBOL statements, use the WSCLEAR*(nnn)* format.

If you specify WSCLEAR(*nnn*), the byte value represented by *nnn* is used to initialize each byte of WORKING-STORAGE data to a specific value. This applies only to data items that do not have a VALUE attributed specified.

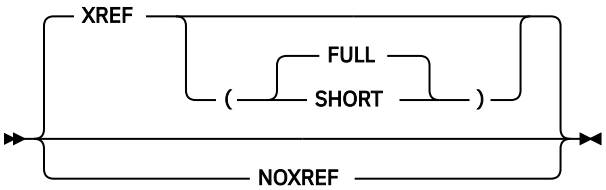
Performance considerations: If you use WSCLEAR and are concerned about the size or performance of an object program, also use OPTIMIZE(FULL). Doing so instructs the compiler to eliminate all unreferenced data items from the DATA DIVISION, which will speed up initialization.

Related references
"OPTIMIZE" on page 273

XREF

Use XREF to produce a sorted cross-reference listing.

XREF option syntax



Default is: XREF(FULL)

Abbreviations are: X | NOX

You can choose XREF, XREF(FULL), or XREF(SHORT). If you specify XREF without any suboptions, XREF(FULL) will be in effect.

A section of the listing shows all the program-names, data-names, and procedure-names that are referenced in your program, and the line numbers where those names are defined. External program-names are identified.

"Example: XREF output:
data-name cross-references" on page 317
"Example: XREF output:
program-name cross-references" on page 319

A section is also included that cross-references COPY or BASIS statements in the program with the files from which associated copybooks were obtained.

"Example: XREF output: COPY/BASIS cross-references" on page 319

Names are listed in the order of the collating sequence that is indicated by the locale setting. This order is used whether the names are in single-byte characters or contain multibyte characters (such as DBCS).

If you use XREF and SOURCE, data-name and procedure-name cross-reference information is printed on the same line as the original source. Line-number references or other information appears on the right-

hand side of the listing page. On the right of source lines that reference an intrinsic function, the letters IFN are printed with the line number of the locations where the function arguments are defined. Information included in the embedded references lets you know if an identifier is undefined (UND) or defined more than once (DUP), if items are implicitly defined (IMP) (such as special registers or figurative constants), or if a program-name is external (EXT).

If you use XREF and NOSOURCE, you get only the sorted cross-reference listing.

XREF(SHORT) prints only the explicitly referenced data items in the cross-reference listing. XREF(SHORT) applies to multibyte data-names and procedure-names as well as to single-byte names.

NOXREF suppresses this listing.

Usage notes

- Group names used in a MOVE CORRESPONDING statement are in the XREF listing. The elementary names in those groups are also listed.
- In the data-name XREF listing, line numbers that are preceded by the letter M indicate that the data item is explicitly modified by a statement on that line.
- XREF listings take additional storage.

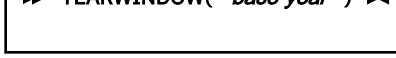
Related concepts
Chapter 16, “Debugging,” on page 301

Related tasks
“Getting listings” on page 309

YEARWINDOW

Use YEARWINDOW to specify the first year of the 100-year window (the *century window*) to be applied to windowed date field processing by the COBOL compiler.

YEARWINDOW option syntax



Default is: YEARWINDOW(1900)

Abbreviations are: YW

base-year represents the first year of the 100-year window. You must specify it with one of the following values:

- An unsigned decimal integer between 1900 and 1999.

  An unsigned integer specifies the starting year of a fixed window. For example, YEARWINDOW(1930) indicates the century window 1930-2029.
- A negative integer from -1 through -99.

  A negative integer indicates a sliding window. The first year of the window is calculated by adding the negative integer to the current year. For example, YEARWINDOW(-80) indicates that the first year of the century window is 80 years before the year in which the program is run.

Usage notes

- The YEARWINDOW option has no effect unless the DATEPROC option is also in effect.
- At run time, two conditions must be true:
  - The century window must have its beginning year in the 1900s.
  - The current year must lie within the century window for the compilation unit.

  For example, if the current year is 2010, the DATEPROC option is in effect, and you use the YEARWINDOW(1900) option, the program will terminate with an error message.

ZWB

If you compile using ZWB, the compiler removes the sign from a signed zoned decimal (DISPLAY) field before comparing this field to an alphanumeric elementary field during execution.

ZWB option syntax

Default is: ZWB

Abbreviations are: None

If the zoned decimal item is a scaled item (that is, it contains the symbol P in its PICTURE string), comparisons that use the decimal item are not affected by ZWB. Such items always have their sign removed before the comparison is made to an alphanumeric field.

ZWB affects how a program runs. The same COBOL program can produce different results depending on the setting of this option.

Use NOZWB if you want to test input numeric fields for SPACES.

Chapter 14. Compiler-directing statements

Several compiler-directing statements help you to direct the compilation of your program.

These are the compiler-directing statements:

BASIS statement
This extended source program library statement provides a complete COBOL program as the source for a compilation. For rules of formation and processing, see the description of *text-name* for the COPY statement.

***CONTROL (*CBL) statement**
This compiler-directing statement selectively suppresses or allows output to be produced. The keywords *CONTROL and *CBL are synonymous.

CALLINTERFACE directive
This compiler directive specifies the interface convention for calls, and indicates whether argument descriptors are to be generated. The convention specified with >>CALLINTERFACE is in effect until another >>CALLINTERFACE specification is made. >>CALLINT is an abbreviation for >>CALLINTERFACE.

>>CALLINTERFACE can be used only in the PROCEDURE DIVISION.

The syntax and usage of the >>CALLINTERFACE directive are similar to that of the CALLINT compiler option. Exceptions are:

- The directive syntax does not include parentheses.
- The directive can be applied to selected calls as described below.
- The directive syntax includes the keyword DESCRIPTOR and its variants.

If you specify >>CALLINT with no suboptions, the call convention used is determined by the CALLINT compiler option.

DESCRIPTOR only: The >>CALLINT directive is treated as a comment except for these forms:

- >>CALLINT SYSTEM DESCRIPTOR, or equivalently >>CALLINT DESCRIPTOR
- >>CALLINT SYSTEM NODESCRIPTOR, or equivalently >>CALLINT NODESCRIPTOR

These directives turn DESCRIPTOR on or off; SYSTEM is ignored.

The >>CALLINT directive can be specified anywhere that a COBOL procedure statement can be specified. For example, this is valid syntax:

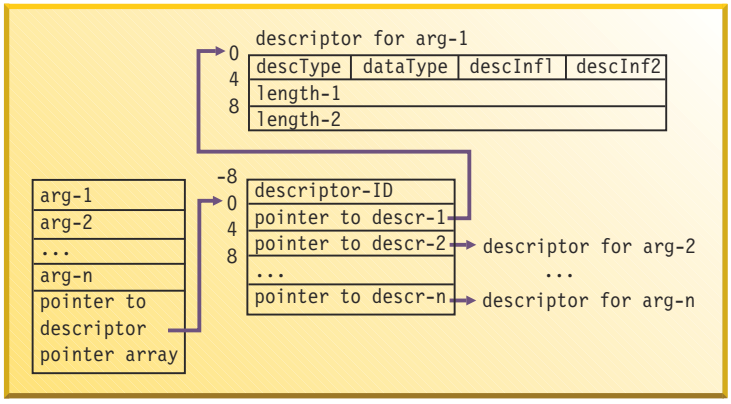
```
MOVE 3 TO
>>CALLINTERFACE SYSTEM
RETURN-CODE.
```

The effect of >>CALLINT is limited to the current program. A nested program or a program compiled in the same batch inherits the calling convention specified in the CALLINT compiler option, but not a convention specified by the >>CALLINT compiler directive.

If you are writing a routine that is to be called with >>CALLINT SYSTEM DESCRIPTOR, this is the argument-passing mechanism:

© Copyright IBM Corp. 2021

CALL "PROGRAM1" USING arg-1, arg-2, ... arg-n



pointer to descr-n
Points to the descriptor for the specific argument; 0 if no descriptor exists for the argument.

descriptor-ID
Set to COBDESC0 to identify this version of the descriptor, allowing for a possible change to the descriptor entry format in the future.

descType
Set to X'02' (descElmt) for an elementary data item of USAGE DISPLAY with PICTURE X(*n*) or USAGE DISPLAY-1 with PICTURE G(*n*) or N(*n*). For all others (numeric fields, structures, tables), set to X'00'.

dataType
Set as follows:

- descType = X'00': dataType = X'00'
- descType = X'02' and the USAGE is DISPLAY: dataType = X'02' (typeChar)
- descType = X'02' and the USAGE is DISPLAY-1: dataType = X'09' (typeGChar)

descInf1
Always set to X'00'.

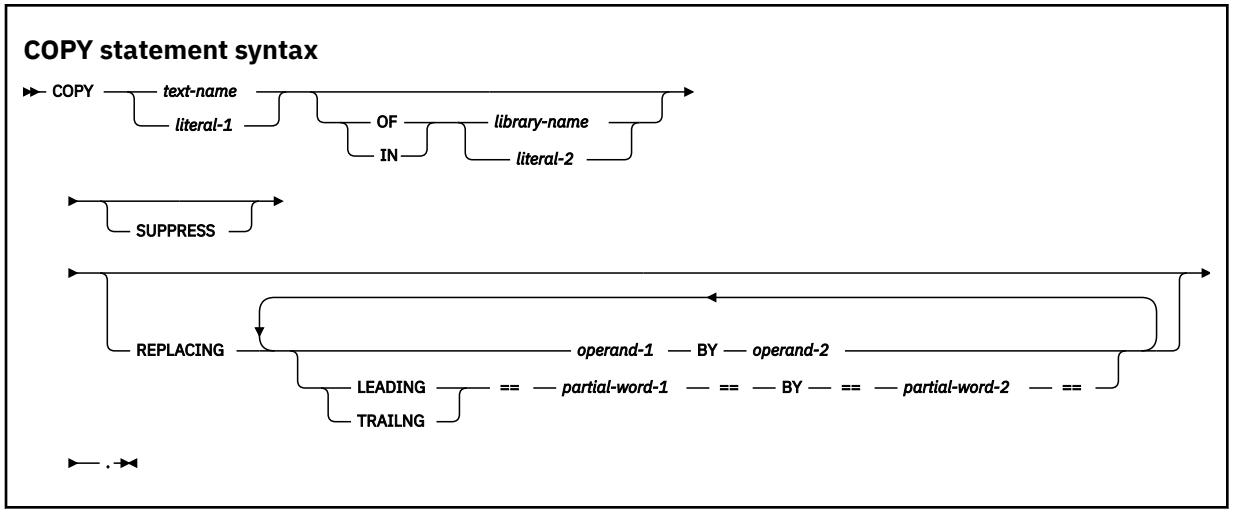
descInf2
Set as follows:

- If descType = X'00'; descInf2 = X'00'
- If descType = X'02':
  - If the CHAR(EBCDIC) option is in effect and the argument is not defined with the NATIVE option in the USAGE clause: descInf2 = X'40'
  - Else: descInf2 = X'00'

length-1
In the argument descriptor is the length of the argument for a fixed-length argument or the current length for a variable-length item.

length-2
The maximum length of the argument if the argument is a variable-length item. For a fixed-length argument, length-2 is equal to length-1.

COPY statement



This compiler-directing statement places prewritten text into a COBOL program.

Neither *text-name* nor *library-name* need to be unique within a program. They can be identical to other user-defined words in the program.

You must specify a *text-name* (the name of a copybook) that contains the prewritten text; for example, `COPY my-text`. You can qualify *text-name* with a *library-name*; for example, `COPY my-text of inventory-lib`. If *text-name* is not qualified, a *library-name* of SYSLIB is assumed.

library-name

If you specify *library-name* as a literal, the content of the literal is treated as the actual path. If you specify *library-name* as a user-defined word, the name is used as an environment variable and the value of the environment variable is used for the path to locate the copybook. To specify multiple path names, delimit them with a colon (:).

If you do not specify *library-name*, the path used is as described under *text-name*.

text-name

If you specify *text-name* as a literal, the content of the literal is treated as the actual path. If you specify *text-name* as a user-defined word, processing depends on whether the environment variable that corresponds to *text-name* is set. If the environment variable is set, the value of the environment variable is used as the file name, and possibly the path name, for the copybook.

A *text-name* is treated as an absolute path if all three of these conditions are met:

- *library-name* is not used.
- *text-name* is a literal or an environment variable.
- The first character is '/'.

For example, this is treated as an absolute path:

```
COPY "/mycpylib/mytext.cpy"
```

If the environment variable that corresponds to *text-name* is not set, the search for the copybook uses the following names:

1. *text-name* with suffix .cpy
2. *text-name* with suffix .cbl
3. *text-name* with suffix .cob
4. *text-name* with no suffix

For example, `COPY MyCopy` searches in the following order:

1. MYCOPY.cpy (in all the specified paths, as described above)
2. MYCOPY.cbl (in all the specified paths, as described above)
3. MYCOPY.cob (in all the specified paths, as described above)
4. MYCOPY (in all the specified paths, as described above)

COBOL defaults *library-name* and *text-name* to uppercase unless the name is contained in a literal ("MyCopy"). In this example, `MyCopy` is not the same as `MYCOPY`. If your file name is mixed case (as in MyCopy.cbl), define *text-name* as a literal in the `COPY` statement.

-I option

For other cases (when neither a *library-name* nor *text-name* indicates the path), the search path is dependent on the `-I` option.

To have `COPY A` be equivalent to `COPY A OF MYLIB`, specify `-I$MYLIB`.

Based on the above rules, `COPY "/X/Y"` will be searched in the root directory, and `COPY "X/Y"` will be searched in the current directory.

`COPY A OF SYSLIB` is equivalent to `COPY A`. The `-I` option does not affect `COPY` statements that have explicit *library-name* qualifications besides those with the library name of `SYSLIB`.

If both *library-name* and *text-name* are specified, the compiler inserts a path separator (/) between the two values if *library-name* does not end in /. For example, `COPY MYCOPY OF MYLIB` with these settings:

```
export MYCOPY=MYPDS(MYMEMBER)
export MYLIB=MYFILE
```

results in `MYFILE/MYPDS(MYMEMBER)`.

If you specify *text-name* as a user-defined word, you can access local files and also access PDS members on z/OS without changing your mainframe source. For example:

```
COPY mycopybook
```

In this example, if the environment variable *mycopybook* is set to `h/mypds(mycopy)`:

- `h` is assigned to the specific host.
- `mypds` is the z/OS PDS name.
- `mycopy` is the PDS member name.

You can access z/OS files from Linux using NFS (Network File System), which let you access z/OS files by using a Linux path name. However, note that NFS converts the path separator to "." to follow z/OS naming conventions. To ensure proper name formation, keep this in mind when assigning values to your environment variables. For example, these settings:

```
export MYCOPY=(MYMEMBER)
export MYLIB=M/MYFILE/MYPDS
```

do not work because the resulting path is:

```
M/MYFILE/MYPDS/(MYMEMBER)
```

which after conversion of the path separator becomes:

```
M.MYFILE.MYPDS.(MYMEMBER)
```

DELETE statement
This extended source library statement removes COBOL statements from the BASIS source program.

EJECT statement
This compiler-directing statement specifies that the next source statement is to be printed at the top of the next page.

ENTER statement
The statement is treated as a comment.

EVALUATE directive
The EVALUATE directive provides a multi-branch method of choosing the source lines to include in a compilation group.

IF directive
The IF directive provides for a one-way or two-way conditional compilation.

INSERT statement
This library statement adds COBOL statements to the BASIS source program.

PROCESS (CBL) statement
This compiler-directing statement, which you can place before the IDENTIFICATION DIVISION header of an outermost program, specifies compiler options that are to be used during compilation of the program.

REPLACE statement
This statement is used to replace source program text.

SKIP1/2/3 statement
These statements indicate lines to be skipped in the source listing.

TITLE statement
This statement specifies that a title (header) should be printed at the top of each page of the source listing.

USE statement
The USE statement provides *declaratives* to specify these elements:

- Error-handling procedures: EXCEPTION/ERROR
- User label-handling procedures: LABEL
- Debugging lines and sections: DEBUGGING

Related tasks
"Changing the header of a source listing" on page 4
"Compiling from the command line" on page 223
"Specifying compiler options in the PROCESS (CBL) statement" on page 224

Related references
"cob2 options" on page 230
CALLINTERFACE (*COBOL for Linux on x86 Language Reference*)
PROCESS (CBL) statement (*COBOL for Linux on x86 Language Reference*)
*CONTROL (*CBL) statement (*COBOL for Linux on x86 Language Reference*)
COPY statement (*COBOL for Linux on x86 Language Reference*)
DEFINE directive (*COBOL for Linux on x86 Language Reference*)
EVALUATE directive (*COBOL for Linux on x86 Language Reference*)
IF directive (*COBOL for Linux on x86 Language Reference*)

Chapter 15. Runtime options

The following table lists the runtime options that are supported.

Table 32. ***Runtime options***

Option	Description	Default	Abbreviation
"CHECK" on page 297	Flags checking errors	CHECK(ON)	CH
"DEBUG" on page 298	Specifies whether the COBOL debugging sections specified by the USE FOR DEBUGGING declarative are active	NODEBUG	None
"ERRCOUNT" on page 298	Specifies how many conditions of severity 1 (W-level) can occur before the run unit terminates abnormally	ERRCOUNT(20)	None
"FILESYS" on page 298	Specifies the file system to use for files for which no explicit file-system selection is made, either through the ASSIGN clause or an environment variable	FILESYS(VSA)	FS(DB2|QSAM|RSD|SdU|SFS |STL|VSA|VSAM)
"TRAP" on page 300	Indicates whether COBOL intercepts exceptions	TRAP(ON)	None
"UPSI" on page 300	Sets the eight UPSI switches on or off for applications that use COBOL routines	UPSI(00000000)	None

Specify runtime options by setting the COBRTOPT runtime environment variable.

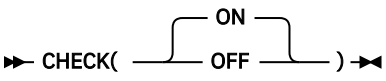
Related tasks
"Setting environment variables" on page 213
"Running programs" on page 235

Related references
"Runtime environment variables" on page 218

CHECK

CHECK causes checking errors to be flagged. In COBOL, index, subscript, and reference-modification ranges can cause checking errors.

CHECK option syntax



Default is: CHECK(ON)

Abbreviation is: CH

ON
Specifies that runtime checking is performed

© Copyright IBM Corp. 2021

OFF
Specifies that runtime checking is not performed

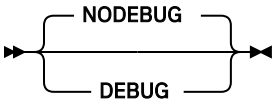
Usage note: CHECK(ON) has no effect if NOSSRANGE was in effect during compilation.

Performance consideration: If you compiled a COBOL program with SSRANGE, and you are not testing or debugging an application, performance improves if you specify CHECK(OFF).

DEBUG

DEBUG specifies whether the COBOL debugging sections specified by the USE FOR DEBUGGING declarative are active.

DEBUG option syntax



Default is: NODEBUG

DEBUG
Activates the debugging sections

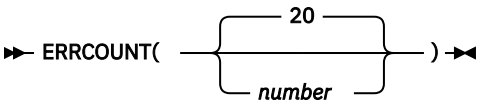
NODEBUG
Suppresses the debugging sections

Performance consideration: To improve performance, use this option only while debugging.

ERRCOUNT

ERRCOUNT indicates how many warning messages can occur before the run unit terminates abnormally.

ERRCOUNT option syntax



Default: ERRCOUNT(20)

number, if positive, is the number of warning messages that can occur while the run unit is running. If the number of warning messages exceeds *number*, the run unit terminates abnormally.

number, if 0, indicates that an unlimited number of warning messages can occur. *number* cannot be negative.

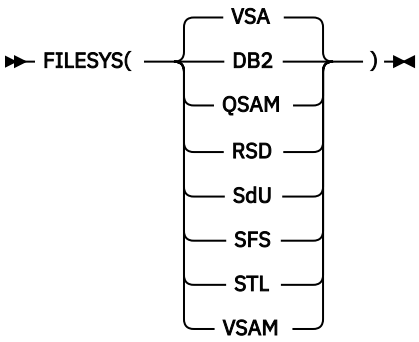
Any message due to a condition that has a severity higher than a warning results in termination of the run unit regardless of the value of the ERRCOUNT option.

FILESYS

FILESYS specifies the file system to be used for files for which no explicit file system was specified by means of an ASSIGN clause or an environment variable. The option applies to sequential, relative, and

indexed files. It does not apply to line-sequential files, for which the file system must be specified as, or default to, LSQ (line sequential).

FILESYS option syntax



Default is: `FILESYS(VSA)`

Abbreviation is: `FS(DB2|QSA|RSD|SdU|SFS|STL|VSA)`

DB2
The file system is Db2 relational database.

QSAM
The file system is compatible with mainframe QSAM files.

RSD
The file system is record sequential delimited.

SdU
The file system is SMARTdata Utilities.

SFS
The file system is CICS Structured File Server.

STL
The file system is the standard language file system.

VSA or VSAM
VSA or VSAM (virtual storage access method) implies either the SFS or STL file system.

If the system file-name begins with the value `/.:/cics/sfs`, this suboption implies the SFS file system. Otherwise, it implies the STL file system.

Related concepts
“File systems” on page 116
“Line-sequential file organization” on page 122

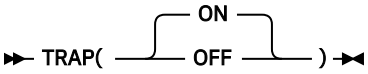
Related tasks
“Identifying files” on page 112

Related references
“Precedence of file-system determination” on page 116
“Runtime environment
variables” on page 218
ASSIGN clause (*COBOL for Linux on x86 Language Reference*)

TRAP

TRAP indicates whether COBOL intercepts exceptions.

TRAP option syntax



Default is: TRAP(ON)

If TRAP(OFF) is in effect, and you do not supply your own trap handler to handle exceptional conditions, the conditions result in a default action by the operating system. For example, if your program attempts to store into an illegal location, the default system action is to issue a message and terminate the process.

ON
Activates COBOL interception of exceptions

OFF
Deactivates COBOL interception of exceptions

Usage notes

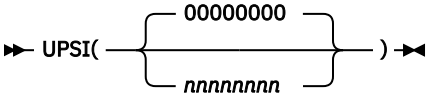
- Use TRAP(OFF) only when you need to analyze a program exception before COBOL handles it.
- When you specify TRAP(OFF) in a non-CICS environment, no exception handlers are established.
- Running with TRAP(OFF) (for exception diagnosis purposes) can cause many side effects, because COBOL requires TRAP(ON). When you run with TRAP(OFF), you can get side effects even if you do not encounter a software-raised condition, program check, or abend. If you do encounter a program check or an abend with TRAP(OFF) in effect, the following side effects can occur:
  - Resources obtained by COBOL are not freed.
  - Files opened by COBOL are not closed, so records might be lost.
  - No messages or dump output are generated.

  The run unit terminates abnormally if such conditions are raised.

UPSI

UPSI sets the eight UPSI switches on or off for applications that use COBOL routines.

UPSI option syntax



Default is: UPSI(00000000)

Each *n* represents one UPSI switch (between 0 and 7); the leftmost *n* represents the first switch. Each *n* can be either 0 (off) or 1 (on).

Chapter 16. Debugging

You can choose between two different approaches to determine the cause of problems in the behavior of your application: source-language debugging or interactive debugging.

For source-language debugging, COBOL provides several language elements, compiler options, and listing outputs that make debugging easier.

Note: IBM COBOL for Linux on x86 does not provide an interactive debugger.

Related tasks
"Debugging with source language" on page 301
"Debugging using compiler options" on page 304

"Getting listings" on page 309
"Debugging with messages that have offset information" on page 321

Debugging with source language

You can use several COBOL language features to pinpoint the cause of a failure in a program.

If a failing program is part of a large application that is already in production (precluding source updates), write a small test case to simulate the failing part of the program. Code debugging features in the test case to help detect these problems:

- Errors in program logic
- Input-output errors
- Mismatches of data types
- Uninitialized data
- Problems with procedures

Related tasks
"Tracing program logic" on page 301
"Finding and handling input-output errors" on page 302
"Validating data" on page 302
"Moving, initializing or setting uninitialized data" on page 303
"Generating information about procedures" on page 303

Related references
Source language debugging (*COBOL for Linux on x86 Language Reference*)

Tracing program logic

Trace the logic of your program by adding DISPLAY statements.

For example, if you determine that the problem is in an EVALUATE statement or in a set of nested IF statements, use DISPLAY statements in each path to see the logic flow. If you determine that the calculation of a numeric value is causing the problem, use DISPLAY statements to check the value of some interim results.

If you use explicit scope terminators to end statements in your program, the logic is more apparent and therefore easier to trace.

© Copyright IBM Corp. 2021

To determine whether a particular routine started and finished, you might insert code like this into your program:

```
DISPLAY "ENTER CHECK PROCEDURE"
    .
    .  (checking procedure routine)
    .
DISPLAY "FINISHED CHECK PROCEDURE"
```

After you are sure that the routine works correctly, disable the DISPLAY statements in one of two ways:

- Put an asterisk in column 7 of each DISPLAY statement line to convert it to a comment line.
- Put a D in column 7 of each DISPLAY statement to convert it to a comment line. When you want to reactivate these statements, include a WITH DEBUGGING MODE clause in the ENVIRONMENT DIVISION; the D in column 7 is ignored and the DISPLAY statements are implemented.

Before you put the program into production, delete or disable the debugging aids you used and recompile the program. The program will run more efficiently and use less storage.

Related concepts
"Scope terminators" on page 16

Related references
DISPLAY statement (*COBOL for Linux on x86 Language Reference*)

Finding and handling input-output errors

File status keys can help you determine whether your program errors are due to input-output errors occurring on the storage media.

To use file status keys in debugging, check for a nonzero value in the status key after each input-output statement. If the value is nonzero (as reported in an error message), look at the coding of the input-output procedures in the program. You can also include procedures to correct the error based on the value of the status key.

If you determine that a problem lies in an input-output procedure, include the USE EXCEPTION/ERROR declarative to help debug the problem. Then, when a file fails to open, the appropriate EXCEPTION/ERROR declarative is performed. The appropriate declarative might be a specific one for the file or one provided for the open attributes INPUT, OUTPUT, I-O, or EXTEND.

Code each USE AFTER STANDARD ERROR statement in a section that follows the DECLARATIVES keyword in the PROCEDURE DIVISION.

Related tasks
"Coding ERROR declaratives" on page 166
"Using file status keys" on page 166

Related references
File status key (*COBOL for Linux on x86 Language Reference*)

Validating data

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is receiving the wrong type of data on an input record, use the class test (the class condition) to validate the type of data.

You can use the class test to check whether the content of a data item is ALPHABETIC, ALPHABETIC-LOWER, ALPHABETIC-UPPER, DBCS, KANJI, or NUMERIC. If the data item is described implicitly or explicitly as USAGE NATIONAL, the class test checks the national character representation of the characters associated with the specified character class.

Related tasks
"Coding conditional expressions" on page 85

"Testing for valid DBCS characters" on page 196

Related references
Class condition (*COBOL for Linux on x86 Language Reference*)

Moving, initializing or setting uninitialized data

Use an INITIALIZE or SET statement to initialize a table or data item when you suspect that a problem might be caused by residual data in those fields.

If the problem happens intermittently and not always with the same data, it could be that a switch was not initialized but is generally set to the right value (0 or 1) by chance. By using a SET statement to ensure that the switch is initialized, you can determine that the uninitialized switch is the cause of the problem or remove it as a possible cause.

Related references
INITIALIZE statement (*COBOL for Linux on x86 Language Reference*)
SET statement (*COBOL for Linux on x86 Language Reference*)

Generating information about procedures

Generate information about your program or test case and how it is running by coding the USE FOR DEBUGGING declarative. This declarative lets you include statements in the program and indicate when they should be performed when you run your program.

For example, to determine how many times a procedure is run, you could include a debugging procedure in the USE FOR DEBUGGING declarative and use a counter to keep track of the number of times that control passes to that procedure. You can use the counter technique to check items such as these:

- How many times a PERFORM statement runs, and thus whether a particular routine is being used and whether the control structure is correct
- How many times a loop runs, and thus whether the loop is executing and whether the number for the loop is accurate

You can use debugging lines or debugging statements or both in your program.

Debugging lines are statements that are identified by a D in column 7. To make debugging lines in your program active, code the WITH DEBUGGING MODE clause on the SOURCE-COMPUTER line in the ENVIRONMENT DIVISION. Otherwise debugging lines are treated as comments.

Debugging statements are the statements that are coded in the DECLARATIVES section of the PROCEDURE DIVISION. Code each USE FOR DEBUGGING declarative in a separate section. Code the debugging statements as follows:

- Only in a DECLARATIVES section.
- Following the header USE FOR DEBUGGING.
- Only in the outermost program; they are not valid in nested programs. Debugging statements are also never triggered by procedures that are contained in nested programs.

To use debugging statements in your program, you must include the WITH DEBUGGING MODE clause and use the DEBUG runtime option.

Options restrictions:

- USE FOR DEBUGGING declaratives, if the WITH DEBUGGING MODE clause has been specified, are mutually exclusive with the TEST compiler option. If USE FOR DEBUGGING declaratives and the WITH DEBUGGING MODE clause are present, the TEST option is cancelled.

"Example: USE FOR DEBUGGING" on page 304

Related references
SOURCE-COMPUTER paragraph (*COBOL for Linux on x86 Language Reference*)

Debugging lines (*COBOL for Linux on x86 Language Reference*)
Debugging sections (*COBOL for Linux on x86 Language Reference*)
DEBUGGING declarative (*COBOL for Linux on x86 Language Reference*)

Example: USE FOR DEBUGGING

This example shows the kind of statements that are needed to use a DISPLAY statement and a USE FOR DEBUGGING declarative to test a program.

The DISPLAY statement writes information to the terminal or to an output file. The USE FOR DEBUGGING declarative is used with a counter to show how many times a routine runs.

```
Environment Division.
. . .
Data Division.
. . .
Working-Storage Section.
. . . (other entries your program needs)
01  Trace-Msg    PIC X(30) Value "  Trace for Procedure-Name : ".
01  Total        PIC 9(9)  Value 1.
. . .
Procedure Division.
Declaratives.
Debug-Declaratives Section.
    Use For Debugging On Some-Routine.
Debug-Declaratives-Paragraph.
    Display Trace-Msg, Debug-Name, Total.
End Declaratives.

Main-Program Section.
    . . . (source program statements)
    Perform Some-Routine.
    . . . (source program statements)
    Stop Run.
Some-Routine.
    . . . (whatever statements you need in this paragraph)
    Add 1 To Total.
Some-Routine-End.
```

The DISPLAY statement in the DECLARATIVES SECTION issues this message every time the procedure Some-Routine runs:

```
  Trace For Procedure-Name : Some-Routine 22
```

The number at the end of the message, 22, is the value accumulated in the data item Total; it indicates the number of times Some-Routine has run. The statements in the debugging declarative are performed before the named procedure runs.

You can also use the DISPLAY statement to trace program execution and show the flow through the program. You do this by dropping Total from the DISPLAY statement and changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to:

```
USE FOR DEBUGGING ON ALL PROCEDURES.
```

As a result, a message is displayed before each nondebugging procedure in the outermost program runs.

Debugging using compiler options

You can use certain compiler options to help you find errors in your program, find various elements in your program, obtain listings, and prepare your program for debugging.

You can find the following errors by using compiler options (the options are shown in parentheses):

- Syntax errors such as duplicate data-names (NOCOMPILE)
- Missing sections (SEQUENCE)

- Invalid subscript values (SSRANGE)

You can find the following elements in your program by using compiler options:

- Error messages and locations of the associated errors (FLAG)
- Program entity definitions and references (XREF)
- Data items in the DATA DIVISION (MAP)
- Statement references (VBREF)

You can get a copy of your source (SOURCE) or a listing of generated code (LIST).

You prepare your program for debugging by using the TEST compiler option.

Related tasks
"Finding coding errors" on page 305
"Finding line sequence problems" on page 305
"Checking for valid ranges" on page 306
"Selecting the level of error to be diagnosed" on page 306
"Finding program entity definitions and references" on page 308
"Listing data items" on page 309
"Getting listings" on page 309

Related references
"Compiler options" on page 245

Finding coding errors

Use the NOCOMPILE option to compile conditionally or to only check syntax. When used with the SOURCE option, NOCOMPILE produces a listing that will help you find coding mistakes such as missing definitions, improperly defined data items, and duplicate data-names.

Checking syntax only: To only check the syntax of your program, and not produce object code, use NOCOMPILE without a suboption. If you also specify the SOURCE option, the compiler produces a listing.

When you specify NOCOMPILE, several compiler options are suppressed. See the related reference below about the COMPILE option for details.

Compiling conditionally: To compile conditionally, use NOCOMPILE(*x*), where *x* is one of the severity levels of errors. Your program is compiled if all the errors are of a lower severity than *x*. The severity levels that you can use, from highest to lowest, are S (severe), E (error), and W (warning).

If an error of level *x* or higher occurs, the compilation stops and your program is only checked for syntax.

Related references
"COMPILE" on page 257

Finding line sequence problems

Use the SEQUENCE compiler option to find statements that are out of sequence. Breaks in sequence indicate that a section of a source program was moved or deleted.

When you use SEQUENCE, the compiler checks the source statement numbers to determine whether they are in ascending sequence. Two asterisks are placed beside statement numbers that are out of sequence. The total number of these statements is printed as the first line in the diagnostics after the source listing.

Related references
"SEQUENCE" on page 276

Checking for valid ranges

Use the SSRANGE compiler option to check whether addresses fall within proper ranges.

SSRANGE causes the following addresses to be checked:

- Subscripted or indexed data references: Is the effective address of the required element within the maximum boundary of the specified table?
- Variable-length data references (a reference to a data item that contains an OCCURS DEPENDING ON clause): Is the actual length positive and within the maximum defined length for the group data item?
- Reference-modified data references: Are the offset and length positive? Is the sum of the offset and length within the maximum length for the data item?

If the SSRANGE option is in effect, checking is performed at run time if both of the following conditions are true:

- The COBOL statement that contains the indexed, subscripted, variable-length, or reference-modified data item is performed.
- The CHECK runtime option is ON.

If an effective address is outside the range of the data item that contains the referenced data, an error message is generated and the program stops. The message identifies the table or identifier that was referenced and the line number where the error occurred. Additional information is provided depending on the type of reference that caused the error.

If all subscripts, indices, and reference modifiers in a given data reference are literals and they result in a reference outside the data item, the error is diagnosed at compile time regardless of the setting of the SSRANGE option.

Performance consideration: SSRANGE can somewhat degrade performance because of the extra overhead to check each subscripted or indexed item.

Related references
"SSRANGE" on page 281
"Performance-related compiler options" on page 451

Selecting the level of error to be diagnosed

Use the FLAG compiler option to specify the level of error to be diagnosed during compilation and to indicate whether error messages are to be embedded in the listing. Use FLAG(I) or FLAG(I,I) to be notified of all errors.

Specify as the first parameter the lowest severity level of the syntax-error messages to be issued. Optionally specify the second parameter as the lowest level of the syntax-error messages to be embedded in the source listing. This severity level must be the same or higher than the level for the first parameter. If you specify both parameters, you must also specify the SOURCE compiler option.

Table 33. ***Severity levels of compiler messages***

Severity level	**Resulting messages**
U (unrecoverable)	U messages only
S (severe)	All S and U messages
E (error)	All E, S, and U messages
W (warning)	All W, E, S, and U messages
I (informational)	All messages

When you specify the second parameter, each syntax-error message (except a U-level message) is embedded in the source listing at the point where the compiler had enough information to detect that error. All embedded messages (except those issued by the library compiler phase) directly follow the statement to which they refer. The number of the statement that had the error is also included with the message. Embedded messages are repeated with the rest of the diagnostic messages at the end of the source listing.

When you specify the NOSOURCE compiler option, the syntax-error messages are included only at the end of the listing. Messages for unrecoverable errors are not embedded in the source listing, because an error of this severity terminates the compilation.

"Example: embedded messages" on page 307

Related tasks
"Generating a list of compiler messages" on page 228

Related references
"Severity codes for compiler diagnostic messages" on page 228
"Messages and listings for compiler-detected errors" on page 229
"FLAG" on page 265

Example: embedded messages

The following example shows the embedded messages generated by specifying a second parameter to the FLAG option. Some messages in the summary apply to more than one COBOL statement.

```
   LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map
and Cross Reference
 ...                                                                                              |
   000977               /                                                                         |
   000978               ************************************************************            |
   000979               ***          I N I T I A L I Z E   P A R A G R A P H           **          |
   000980               ***  Open files. Accept date, time and format header lines.    **          |
   000981         IA4690***   Load location-table.                                     **          |
   000982               ************************************************************            |
   000983                  100-initialize-paragraph.                                              |
   000984                    move spaces to ws-transaction-record                                 | IMP
339
   000985                    move spaces to ws-commuter-record                                    | IMP
315
   000986                    move zeroes to commuter-zipcode                                      | IMP
326
   000987                    move zeroes to commuter-home-phone                                   | IMP
327
   000988                    move zeroes to commuter-work-phone                                   | IMP
328
   000989                    move zeroes to commuter-update-date                                  | IMP
332
   000990                    open input update-transaction-file                                   | 203
 ==000990==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE".  The       |
                          reference to this file was discarded.                                  |
   000991                           location-file                                                 | 192
   000992                           i-o commuter-file                                             | 180
   000993                           output print-file                                             | 216
   000994                    if loccode-file-status not = "00" or                                 | 248
   000995                       update-file-status not = "00" or                                  | 247
   000996                       updprint-file-status not = "00"                                   | 249
   000997      1                 display "Open Error ..."                                         |
   000998      1                 display "  Location File Status = " loccode-file-status          | 248
   000999      1                 display "  Update   File Status = " update-file-status           | 247
   001000      1                 display "  Print    File Status = " updprint-file-status         | 249
   001001      1                perform 900-abnormal-termination                                  | 1433
   001002                    end-if                                                               |
   001003         IA4760     if commuter-file-status not = "00" and not = "97"                    | 240
   001004      1               display "100-OPEN"                                                 |
   001005      1               move 100 to comp-code                                              | 230
   001006      1               perform 500-stl-error                                              | 1387
   001007      1               display "Commuter File Status (OPEN) = "                           |
```

```
   001008      1                       commuter-file-status                                    | 240
   001009      1              perform 900-abnormal-termination                                 | 1433
   001010         IA4790     end-if                                                           |
   001011                    accept ws-date from date                                         | UND
 ==001011==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.  |
   001012         IA4810     move corr ws-date to header-date                                 | UND
463
 ==001012==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.  |
   001013                    accept ws-time from time                                         | UND
 ==001013==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.  |
   001014         IA4830     move corr ws-time to header-time                                 | UND
457
 ==001014==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.  |
   001015         IA4840     read location-file                                               | 192
 ...
 LineID  Message code  Message text
      192  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions.
                         The file definition was discarded.
      899  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".
                         The reference to this file was discarded.
                         Same message on line:    990
     1011  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.
                         Same message on line:   1012
     1013  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.
                         Same message on line:   1014
     1015  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".
                         This input/output statement was discarded.
                         Same message on line:   1027
     1026  IGYPS2121-S   "LOC-CODE" was not defined as a data-name.  The statement was discarded.
     1209  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name.  The statement was discarded.
                         Same message on line:   1230
     1210  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name.  The statement was
discarded.
                         Same message on line:   1231
     1212  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name.  The statement was discarded.
                         Same message on line:   1233
     1213  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was
discarded.
                         Same message on line:   1234
     1223  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.
 Messages    Total    Informational    Warning    Error    Severe    Terminating
 Printed:      19                                   1        18
 * Statistics for COBOL program FLAGOUT:
 *    Source records = 1755
 *    Data Division statements = 279
 *    Procedure Division statements = 479
 Locale = en_US.ISO8859-1                  (1)
 End of compilation 1,  program FLAGOUT,  highest severity: Severe.
 Return code 12
```

(1)
The locale that the compiler used

Finding program entity definitions and references

Use the XREF(FULL) compiler option to find out where a data-name, procedure-name, or program-name is defined and referenced. Use it also to produce a cross-reference of COPY or BASIS statements to the files from which copybooks were obtained.

A sorted cross-reference includes the line number where the data-name, procedure-name, or program-name was defined and the line numbers of all references to it.

To include only the explicitly referenced data items, use the XREF(SHORT) option.

Use both the XREF (either FULL or SHORT) and the SOURCE options to print a modified cross-reference to the right of the source listing. This embedded cross-reference shows the line number where the data-name or procedure-name was defined.

For further details, see the related reference about the XREF compiler option.

“Example: XREF output:
data-name cross-references” on page 317
“Example: XREF output:
program-name cross-references” on page 319

“Example:
XREF output: COPY/BASIS cross-references” on page 319
“Example: XREF output:
embedded cross-reference” on page 320

Related tasks
“Getting listings” on page 309

Related references
“XREF” on page 287

Listing data items

Use the MAP compiler option to create a listing of the DATA DIVISION items and all implicitly declared items.

When you specify the MAP option, an embedded MAP summary that contains condensed MAP information is generated to the right of the COBOL source data definition. When both XREF data and an embedded MAP summary are on the same line, the embedded summary is printed first.

You can select or inhibit parts of the MAP listing and embedded MAP summary by using *CONTROL MAP|NOMAP (or *CBL MAP|NOMAP) statements throughout the source. For example:

```
*CONTROL NOMAP
    01  A
    02  B
*CONTROL MAP
```

“Example: MAP output” on page 314

Related tasks
“Getting listings” on page 309

Related references
“MAP” on page 269

Getting listings

Get the information that you need for debugging by requesting the appropriate compiler listing with the use of compiler options.

Attention: The listings produced by the compiler are not a programming interface and are subject to change.

Table 34. ***Using compiler options to get listings***

Use	Listing	Contents	Compiler option
To check a list of the options in effect for the program, statistics about the content of the program, and diagnostic messages about the compilation To check the locale in effect during compilation	Short listing	• List of options in effect for the program • Statistics about the content of the program • Diagnostic messages about the compilation[1] Locale line that shows the locale in effect	NOSOURCE, NOXREF, NOVBREF, NOMAP, NOLIST

*Table 34. **Using compiler options to get listings** (continued)*

Use	Listing	Contents	Compiler option
To aid in testing and debugging your program; to have a record after the program has been debugged	Source listing	Copy of your source	"SOURCE" on page 278
To find certain data items; to see the final storage allocation after reentrancy or optimization has been accounted for; to see where programs are defined and check their attributes	Map of `DATA DIVISION` items	All `DATA DIVISION` items and all implicitly declared items Embedded map summary (in the right margin of the listing for lines in the `DATA DIVISION` that contain data declarations) Nested program map (if the program contains nested programs)	"MAP" on page 269[2]
To find where a name is defined, referenced, or modified; to determine the context (such as whether a statement was used in a `PERFORM` block) in which a procedure is referenced; to determine the file from which a copybook was obtained	Sorted cross-reference listing of names; sorted cross-reference listing of `COPY/BASIS` statements and copybook files	Data-names, procedure-names, and program-names; references to these names `COPY/BASIS` text-names and library names, and the files from which associated copybooks were obtained Embedded modified cross-reference provides line numbers where data-names and procedure-names were defined	"XREF" on page 287[2,3]
To find the failing statement in a program or the address in storage of a data item that is moved while the program is running	`PROCEDURE DIVISION` code and assembler code produced by the compiler[3]	Generated code	"LIST" on page 268[2,4]
To find an instance of a certain statement	Alphabetic listing of statements	Each statement used, number of times each statement was used, line numbers where each statement was used	"VBREF" on page 286

*Table 34. **Using compiler options to get listings** (continued)*			
Use	**Listing**	**Contents**	**Compiler option**

1. To eliminate messages, turn off the options (such as FLAG) that govern the level of compile diagnostic information.
2. To use your line numbers in the compiled program, use the NUMBER compiler option. The compiler checks the sequence of your source statement line numbers in columns 1 through 6 as the statements are read in. When it finds a line number out of sequence, the compiler assigns to it a number with a value one higher than the line number of the preceding statement. The new value is flagged with two asterisks. A diagnostic message indicating an out-of-sequence error is included in the compilation listing.
3. The context of the procedure reference is indicated by the characters preceding the line number.
4. The assembler listing is written to the listing file (a file that has the same name as the source program but with the suffix .wlist).

"Example: short listing" on page 311
"Example: SOURCE and NUMBER output" on page 313
"Example: MAP output" on page 314
"Example: embedded map summary" on page 315
"Example: nested program map" on page 317
"Example: XREF output: data-name cross-references" on page 317
"Example: XREF output: program-name cross-references" on page 319
"Example: XREF output: COPY/BASIS cross-references" on page 319
"Example: XREF output: embedded cross-reference" on page 320
"Example: VBREF compiler output" on page 321

Related tasks

"Generating a list of compiler messages" on page 228

Related references

"Messages and listings for compiler-detected errors" on page 229

Example: short listing

The parenthetical numbers shown in the listing below correspond to numbered explanations that follow the listing. For illustrative purposes, some errors that cause diagnostic messages were deliberately introduced.

```
PROCESS(CBL) statements:   (1)
 CBL    NOSOURCE,NOXREF,NOVBREF,NOMAP,NOLIST    (2)
Options in effect:    (3)
 NOADATA
   ADDR(32)
   QUOTE
   ARITH(COMPAT)
   CALLINT(NODESCRIPTOR)
   CHAR(NATIVE)
 NOCICS
   COLLSEQ(BINARY)
 NOCOMPILE(S)
 NOCURRENCY
 NODATEPROC
```

```
  NODIAGTRUNC
  NODYNAM
  NOEXIT
    FLAG(I,I)
  NOFLAGSTD
    FLOAT(NATIVE)
    LINECOUNT(60)
  NOLIST
    LSTFILE(LOCALE)
  NOMAP
    MAXMEM(2048K)
  NOMDECK
    NCOLLSEQ(BINARY)
    NSYMBOL(NATIONAL)
  NONUMBER
  NOOPTIMIZE
    PGMNAME(LONGUPPER)
    SEPOBJ
    SEQUENCE
    SIZE(8388608)
  NOSOSI
  NOSOURCE
    SPACE(1)
    SPILL(512)
  NOSQL
    SRCFORMAT(COMPAT)
  NOSSRANGE
    TERM
  NOTEST
  NOTHREAD
    TRUNC(STD)
  NOVBREF
  NOWSCLEAR
  NOXREF
    YEARWINDOW(1900)
    ZWB
```

```
LineID  Message code  Message text    (4)
        IGYDS0139-W   Diagnostic messages were issued during processing of compiler options.  These messages are
                      located at the beginning of the listing.
   193  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions.  The file definition was discarded.
   889  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".  The reference to this file
                      was discarded.
                        Same message on line:    983
   993  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.
                        Same message on line:    994
   995  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.
                        Same message on line:    996
   997  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".  This input/output statement
                      was discarded.
                        Same message on line:   1009
  1008  IGYPS2121-S   "LOC-CODE" was not defined as a data-name.  The statement was discarded.
  1219  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1240
  1220  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1241
  1222  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1243
  1223  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1244
  1233  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.
Messages    Total    Informational    Warning    Error   Severe   Terminating    (5)
Printed:      21                        2         1       18
  * Statistics for COBOL program SLISTING:    (6)
* Source records = 1765
* Data Division statements = 277
* Procedure Division statements = 513
Locale = en_US.ISO8859-1                    (7)
End of compilation 1,  program SLISTING,  highest severity: Severe.   (8)
Return code 12
```

(1)

Message about options specified in a PROCESS (or CBL) statement. This message does not appear if no options were specified.

(2)

Options coded in the PROCESS (or CBL) statement.

(3)

Status of options at the start of this compilation.

(4)

Program diagnostics. The first message refers you to the library phase diagnostics, if there were any. Diagnostics for the library phase are always presented at the beginning of the listing.

(5)
Count of diagnostic messages in this program, grouped by severity level.

(6)
Program statistics for the program SLISTING.

(7)
The locale that the compiler used.

(8)
Program statistics for the compilation unit. When you perform a batch compilation (multiple outermost COBOL programs in a single compilation), the return code is the highest message severity level for the entire compilation.

Example: SOURCE and NUMBER output

In the portion of the listing shown below, the programmer numbered two of the statements out of sequence. The note numbers in the listing correspond to numbered explanations that follow the listing.

```
                    (1)
 LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Cross-
Reference
   (2)    (3)    (4)                                                                             |
                 087000/*****************************************************************        |
                 087100***                D O   M A I N   L O G I C                     **        |
                 087200***                                                              **        |
                 087300*** Initialization. Read and process update transactions until **        |
                 087400*** EOE. Close files and stop run.                               **        |
                 087500*****************************************************************        |
                 087600 procedure division.                                                      |
                 087700   000-do-main-logic.                                                     |
                 087800     display "PROGRAM SRCOUT - Beginning"                                 |
                 087900     perform 050-create-stl-main-file.                                    |
                 088150     display "perform 050-create-stl-main-file finished".                 |
 088151**        088125     perform 100-initialize-paragraph                                     |
                 088200     display "perform 100-initialize-paragraph finished"                  |
                 088300     read update-transaction-file into ws-transaction-record              |
                 088400         at end                                                           |
              1  088500       set transaction-eof to true                                      |
                 088600     end-read                                                             |
                 088700     display "READ completed"                                             |
                 088800     perform until transaction-eof                                        |
              1  088900       display "inside perform until loop"                              |
              1  089000       perform 200-edit-update-transaction                              |
              1  089100       display "After perform 200-edit   "                              |
              1  089200       if no-errors                                                     |
              2  089300         perform 300-update-commuter-record                             |
              2  089400         display "After perform 300-update "                            |
              1  089650       else                                                             |
 089651**     2  089600         perform 400-print-transaction-errors                           |
              2  089700         display "After perform 400-errors "                            |
              1  089800       end-if                                                           |
              1  089900       perform 410-re-initialize-fields                                 |
              1  090000       display "After perform 410-reinitialize"                         |
              1  090100       read update-transaction-file into ws-transaction-record          |
              1  090200           at end                                                       |
              2  090300         set transaction-eof to true                                    |
              1  090400       end-read                                                         |
              1  090500       display "After '2nd READ'   "                                    |
                 090600     end-perform                                                        |
```

(1)
Scale line labels Area A, Area B, and source-code column numbers

(2)
Source-code line number assigned by the compiler

(3)
Program (PL) and statement (SL) nesting level

(4)
Columns 1 through 6 of program (the sequence number area)

Example: MAP output

The following example shows output from the MAP option. The numbers used in the explanation below correspond to the numbers that annotate the output.

```
 Data Division Map
 (1)
 Data Definition Attribute codes (rightmost column) have the following meanings:
     D = Object of OCCURS DEPENDING    G = GLOBAL                            LSEQ= ORGANIZATION LINE
SEQUENTIAL
     E = EXTERNAL                      O = Has OCCURS clause                 SEQ= ORGANIZATION SEQUENTIAL
     VLO=Variably Located Origin       OG= Group has own length definition   INDX= ORGANIZATION INDEXED
     VL= Variably Located              R = REDEFINES                         REL= ORGANIZATION RELATIVE

 (2)      (3) (4)                                          (5)      (6)         (7)           (8)
 Source   Hierarchy and                                                                      Data Def
 LineID   Data Name                                    Length(Displacement) Data Type      Attributes
      4  PROGRAM-ID IGYTCARA----------------------------------------------------------------------------
*
    180   FD COMMUTER-FILE                                                   File            INDX
    182   1  COMMUTER-RECORD                                  80             Group
    183     2  COMMUTER-KEY                                  16(0000000)    Display
    184     2  FILLER                                        64(0000016)    Display
    186   FD COMMUTER-FILE-MST                                               File            INDX
    188   1  COMMUTER-RECORD-MST                              80             Group
    189     2  COMMUTER-KEY-MST                              16(0000000)    Display
    190     2  FILLER                                        64(0000016)    Display
    192   FD LOCATION-FILE                                                   File            SEQ
    203   FD UPDATE-TRANSACTION-FILE                                         File            SEQ
    208   1  UPDATE-TRANSACTION-RECORD                        80             Display
    216   FD PRINT-FILE                                                      File            SEQ
    221   1  PRINT-RECORD                                    121            Display
    228   1  WORKING-STORAGE-FOR-IGYCARA                       1             Display
```

(1)
Explanations of the data definition attribute codes.

(2)
Source line number where the data item was defined.

(3)
Level definition or number. The compiler generates this number in the following way:

- First level of any hierarchy is always 01. Increase 1 for each level (any item you coded as level 02 through 49).
- Level-numbers 66, 77, and 88, and the indicators FD and SD, are not changed.

(4)
Data-name that is used in the source module in source order.

(5)
Length of data item. Base locator value.

(6)
Hexadecimal displacement from the beginning of the containing structure.

(7)
Data type and usage.

(8)
Data definition attribute codes. The definitions are explained at the top of the `DATA DIVISION` map.

“Example: embedded
map summary” on page 315
“Example: nested program map” on page 317

Related references
“Terms and symbols used in MAP output” on page 316

Example: embedded map summary

The following example shows an embedded map summary from specifying the MAP option. The summary appears in the right margin of the listing for lines in the DATA DIVISION that contain data declarations.

```
 000002                Identification Division.                                  |
 000003                                                                          |
 000004                Program-id.    EMBMAP.                                    |
    . . .                                                                        |
 000176                Data division.                                            |
 000177                File section.                                             |
 000178                                                                          |
 000179                                                                          |
 000180                FD  COMMUTER-FILE                                         |
 000181                    record 80 characters.                                 |      (1)   (2)
 000182                  01 commuter-record.                                     |       80
 000183                    05 commuter-key                PIC x(16).             |
16(0000000)
 000184                    05 filler                      PIC x(64).             |
64(0000016)
    . . .                                                                        |
 000221        IA1620    01 print-record                 pic x(121).            |      121
    . . .                                                                        |
 000227                Working-storage section.                                  |
 000228                  01 Working-storage-for-EMBMAP     pic x.                |        1
 000229                                                                          |
 000230                  77 comp-code                     pic S9999 comp.       |        2
 000231                  77 ws-type                       pic x(3)   value spaces.|       3
 000232                                                                          |
 000233                                                                          |
 000234                  01 i-f-status-area.                                     |        2
 000235                    05 i-f-file-status             pic x(2).             |
2(0000000)
 000236                      88 i-o-successful            value zeroes.         |IMP
 000237                                                                          |
 000238                                                                          |
 000239                  01 status-area.                                         |        8
 000240                    05 commuter-file-status       pic x(2).             |(3)
2(0000000)
 000241                     88 i-o-okay                   value zeroes.         |IMP
    . . .                                                                        |
 000246                                                                          |
 000247                  77 update-file-status            pic xx.               |        2
 000248                  77 loccode-file-status           pic xx.               |        2
 000249                  77 updprint-file-status          pic xx.               |        2
    . . .                                                                        |
 000877                procedure division.                                       |
 000878                  000-do-main-logic.                                      |
 000879                    display "PROGRAM EMBMAP - Beginning".                 |
 000880                    perform 050-create-stl-main-file.                   |931
    . . .                                                                        |
```

(1)
Decimal length of data item

(2)
Hexadecimal displacement from the beginning of the base locator value

(3)
Special definition symbols:

UND
The user name is undefined.

DUP
The user name is defined more than once.

IMP
An implicitly defined name, such as special registers or figurative constants.

IFN
An intrinsic function reference.

EXT
An external reference.

Terms and symbols used in MAP output

The following table describes the terms and symbols used in the listings produced by the MAP compiler option.

Table 35. ***Terms and symbols used in MAP output***

Term	Description
ALPHABETIC	Alphabetic (PICTURE A)
ALPHA-EDIT	Alphabetic-edited
AN-EDIT	Alphanumeric-edited
BINARY	Binary (USAGE BINARY, COMPUTATIONAL, or COMPUTATIONAL-5)
COMP-1	Single-precision internal floating point (USAGE COMPUTATIONAL-1)
COMP-2	Double-precision internal floating point (USAGE COMPUTATIONAL-2)
DBCS	DBCS (USAGE DISPLAY-1)
DBCS-EDIT	DBCS edited
DISP-FLOAT	Display floating point (USAGE DISPLAY)
DISPLAY	Alphanumeric (PICTURE X)
DISP-NUM	Zoned decimal (USAGE DISPLAY)
DISP-NUM-EDIT	Numeric-edited (USAGE DISPLAY)
FD	File definition
FUNCTION-PTR	Pointer to an externally callable function (USAGE FUNCTION-POINTER)
GROUP	Alphanumeric fixed-length group
GRP-VARLEN	Alphanumeric variable-length group
INDEX	Index (USAGE INDEX)
INDEX-NAME	Index-name
NATIONAL	Category national (USAGE NATIONAL)
NAT-EDIT	National-edited (USAGE NATIONAL)
NAT-FLOAT	National floating point (USAGE NATIONAL)
NAT-GROUP	National group (GROUP-USAGE NATIONAL)
NAT-GRP-VARLEN	National variable-length group (GROUP-USAGE NATIONAL)
NAT-NUM	National decimal (USAGE NATIONAL)
NAT-NUM-EDIT	National numeric-edited (USAGE NATIONAL)
PACKED-DEC	Internal decimal (USAGE PACKED-DECIMAL or COMPUTATIONAL-3)
POINTER	Pointer (USAGE POINTER)

*Table 35. **Terms and symbols used in MAP output** (continued)*	
Term	**Description**
PROCEDURE-PTR	Pointer to an externally callable program (USAGE PROCEDURE-POINTER)
SD	Sort file definition
01-49, 77	Level-numbers for data descriptions
66	Level-number for RENAMES
88	Level-number for condition-names

Example: nested program map

This example shows a map of nested procedures produced by specifying the MAP compiler option. Numbers in parentheses refer to notes that follow the example.

```
Nested Program Map

    (1)
Program Attribute codes (rightmost column) have the following meanings:
   C = COMMON
   I = INITIAL
   U = PROCEDURE DIVISION USING...

   (2)   (3)      (4)                                        (5)
Source Nesting                                             Program
LineID Level   Program Name from PROGRAM-ID paragraph      Attributes
     2         NESTED. . . . . . . . . . . . . . . . .
    12     1     X1. . . . . . . . . . . . . . . . . .
    20     2       X11 . . . . . . . . . . . . . . . .
    27     2       X12 . . . . . . . . . . . . . . . .
    35     1     X2. . . . . . . . . . . . . . . . . .
```

(1)
Explanations of the program attribute codes

(2)
Source line number where the program was defined

(3)
Depth of program nesting

(4)
Program-name

(5)
Program attribute codes

Example: XREF output: data-name cross-references

The following example shows a sorted cross-reference of data-names that is produced by the XREF compiler option. Numbers in parentheses refer to notes after the example.

```
An "M" preceding a data-name reference indicates that the
data-name is modified by this reference.

     (1)        (2)                            (3)
  Defined   Cross-reference of data-names   References

     265   ABEND-ITEM1
     266   ABEND-ITEM2
     347   ADD-CODE . . . . . . . . . . .  1102 1162
```

```
      381   ADDRESS-ERROR. . . . . . . . .  M1126
      280   AREA-CODE. . . . . . . . . . .  1236 1261 1324 1345
      382   CITY-ERROR . . . . . . . . . .  M1129

        (4)
Context usage is indicated by the letter preceding a procedure-name
reference. These letters and their meanings are:
    A = ALTER (procedure-name)
    D = GO TO (procedure-name) DEPENDING ON
    E = End of range of (PERFORM) through (procedure-name)
    G = GO TO (procedure-name)
    P = PERFORM (procedure-name)
    T = (ALTER) TO PROCEED TO (procedure-name)
    U = USE FOR DEBUGGING (procedure-name)

    (5)       (6)                            (7)
 Defined   Cross-reference of procedures   References

     877   000-DO-MAIN-LOGIC
     930   050-CREATE-STL-MAIN-FILE . . .  P879
     982   100-INITIALIZE-PARAGRAPH . . .  P880
    1441   1100-PRINT-I-F-HEADINGS. . . .  P915
    1481   1200-PRINT-I-F-DATA. . . . . .  P916
    1543   1210-GET-MILES-TIME. . . . . .  P1510
    1636   1220-STORE-MILES-TIME. . . . .  P1511
    1652   1230-PRINT-SUB-I-F-DATA. . . .  P1532
    1676   1240-COMPUTE-SUMMARY . . . . .  P1533
    1050   200-EDIT-UPDATE-TRANSACTION. .  P886
    1124   210-EDIT-THE-REST. . . . . . .  P1116
    1159   300-UPDATE-COMMUTER-RECORD . .  P888
    1207   310-FORMAT-COMMUTER-RECORD . .  P1164 P1179
    1258   320-PRINT-COMMUTER-RECORD. . .  P1165 P1176 P1182 P1192
    1288   330-PRINT-REPORT . . . . . . .  P1178 P1202 P1256 P1280 P1340 P1365 P1369
    1312   400-PRINT-TRANSACTION-ERRORS .  P890
```

Cross-reference of data-names:

(1)
Line number where the name was defined.

(2)
Data-name.

(3)
Line numbers where the name was used. If `M` precedes the line number, the data item was explicitly modified at the location.

Cross-reference of procedure references:

(4)
Explanations of the context usage codes for procedure references.

(5)
Line number where the procedure-name is defined.

(6)
Procedure-name.

(7)
Line numbers where the procedure is referenced, and the context usage code for the procedure.

“Example: XREF output:
program-name cross-references” on page 319
“Example:
XREF output: COPY/BASIS cross-references” on page 319
“Example: XREF output:
embedded cross-reference” on page 320

Example: XREF output: program-name cross-references

The following example shows a sorted cross-reference of program-names produced by the XREF compiler option. Numbers in parentheses refer to notes that follow the example.

```
   (1)       (2)                          (3)
Defined   Cross-reference of programs     References

EXTERNAL   EXTERNAL1. . . . . . . . . . . 25
      2   X. . . . . . . . . . . . . . .  41
     12   X1 . . . . . . . . . . . . . .  33 7
     20   X11. . . . . . . . . . . . . .  25 16
     27   X12. . . . . . . . . . . . . .  32 17
     35   X2 . . . . . . . . . . . . . .  40 8
```

(1)
Line number where the program-name was defined. If the program is external, the word EXTERNAL is displayed instead of a definition line number.

(2)
Program-name.

(3)
Line numbers where the program is referenced.

Example: XREF output: COPY/BASIS cross-references

The following example shows a sorted cross-reference of copybooks to the library-names and file names of the associated copybooks, produced by the XREF compiler option. Numbers in parentheses refer to notes after the example.

```
        COPY/BASIS cross-reference of text-names, library names and file names

   (1)                          (1)                   (2)
Text-name                    Library-name            File
name
                       (3)                       (3)  (4)
"realxrealyrealzlongxlo>    'thisislongdirecto>    <toryname/
realxrealyrealzlongxname.cpy
"realxrealyrealzlongxlo>    SYSLIB (default)    (5)./cbldir1/
realxrealyrealzlongxname.cpy
"copyA.cpy"                  SYSLIB (default)       ./cbldir1/copyA.cpy
'./copydir2/copyM.cbl'       SYSLIB (default)       ./copydir2/copyM.cbl
'/copyB.cpy'                 SYSLIB                 ./cbldir1/copyB.cpy
'/copydir/copyM.cbl'         SYSLIB                 ./cbldir1/copydir/copyM.cbl
'cbldir1/copyC.cpy'          ALTDD2                 ./cbldir1/copyC.cpy
'copydir/copyM.cbl'          SYSLIB                 ./cbldir1/copydir/copyM.cbl
'copydir2/copyM.cbl'         SYSLIB (default)       ./copydir2/copyM.cbl
'copydir3/stuff.cpy'         ALTDD2                 ./copydir3/stuff.cpy
'stuff.cpy'                  ALTDD                  ./copydir3/stuff.cpy
OTHERDD                      ALTDD2                 .//cbldir1/other.cob
REALXLONGXLONGYNAMEX         SYSLIB (default)       ./REALXLONGXLONGYNAMEX
. . .

  (5)
./ = /afs/stllp.sanjose.ibm.com/usr1/cobdev/tmross/stuff/subdir
Note: Some names were truncated.   > = truncated on right   < = truncated on left
```

(1)
From the COPY statement in the source; for example the COPY statement corresponding to the fifth item in the cross-reference above would be:

```
COPY '/copyB.cpy' Of SYSLIB
```

(2)
The fully qualified path of the file from which the COPY member was copied

(3)
Truncation of a long text-name or library-name on the right is marked by a greater-than sign (>).

(4)
Truncation of a long file name on the left is marked by a less-than sign (<).

(5)
The current working directory portion of the file name is indicated by `./` in the cross-reference. The expansion of the current working directory name is shown in full beneath the cross-reference.

Related references
"XREF" on page 287

Example: XREF output: embedded cross-reference

The following example shows a modified cross-reference that is embedded in the source listing. The cross-reference is produced by the XREF compiler option.

```
LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
  . . .                                                                                 |
000878               procedure division.                                                |
000879                 000-do-main-logic.                                               |
000880                   display "PROGRAM IGYTCARA - Beginning".                        |
000881                   perform 050-create-stl-main-file.                              | 932 (1)
000882                   perform 100-initialize-paragraph.                              | 984
000883                   read update-transaction-file into ws-transaction-record        | 204 340
000884                       at end                                                     |
000885     1                    set transaction-eof to true                           | 254
000886                   end-read.                                                      |
  . . .                                                                                 |
000984                 100-initialize-paragraph.                                        |
000985                   move spaces to ws-transaction-record                           | IMP 340 (2)
000986                   move spaces to ws-commuter-record                              | IMP 316
000987                   move zeroes to commuter-zipcode                                | IMP 327
000988                   move zeroes to commuter-home-phone                             | IMP 328
000989                   move zeroes to commuter-work-phone                             | IMP 329
000990                   move zeroes to commuter-update-date                            | IMP 333
000991                   open input update-transaction-file                             | 204
000992                       location-file                                              | 193
000993                       i-o commuter-file                                          | 181
000994                       output print-file                                          | 217
  . . .                                                                                 |
001442               1100-print-i-f-headings.                                           |
001443                                                                                  |
001444                   open output print-file.                                        | 217
001445                                                                                  |
001446                   move function when-compiled to when-comp.                      | IFN 698 (2)
001447                   move when-comp (5:2) to compile-month.                         | 698 640
001448                   move when-comp (7:2) to compile-day.                           | 698 642
001449                   move when-comp (3:2) to compile-year.                          | 698 644
001450                                                                                  |
001451                   move function current-date (5:2) to current-month.             | IFN 649
001452                   move function current-date (7:2) to current-day.               | IFN 651
001453                   move function current-date (3:2) to current-year.              | IFN 653
001454                                                                                  |
001455                   write print-record from i-f-header-line-1                      | 222 635
001456                         after new-page.                                          | 138
  . . .                                                                                 |
```

(1)
Line number of the definition of the data-name or procedure-name in the program

(2)
Special definition symbols:

UND
The user name is undefined.

DUP
The user name is defined more than once.

IMP
Implicitly defined name, such as special registers and figurative constants.

IFN
Intrinsic function reference.

EXT
External reference.

*****
The program-name is unresolved because the NOCOMPILE option is in effect.

Example: VBREF compiler output

The following example shows an alphabetic listing of all the statements in a program, and shows where each is referenced. The listing is produced by the VBREF compiler option.

```
 (1)      (2)                  (3)
2        ACCEPT . . . . . .  1010 1012
2        ADD. . . . . . . .  1290 1306
1        CALL . . . . . . .  1406
5        CLOSE. . . . . . .  898 945 970 1526 1535
20       COMPUTE. . . . . .  1506 1640 1644 1657 1660 1663 1664 1665 1678 1682 1686 1691 1696 1701 1709 1713
                             1718 1723 1728 1733
2        CONTINUE . . . . .  1062 1069
2        DELETE . . . . . .  964 1193
48       DISPLAY. . . . . .  878 906 917 918 919 933 940 942 947 953 960 966 972 996 997 998 999 1003 1006 1037
                             1090 1168 1171 1185 1195 1387 1388 1389 1390 1391 1392 1393 1401 1402 1403 1404
                             1405 1433 1485 1486 1492 1497 1498 1520 1521 1528 1529 1624
2        EVALUATE . . . . .  1161 1557
47       IF . . . . . . . .  887 905 932 939 946 952 959 965 971 993 1002 1036 1051 1054 1071 1074 1077 1089
                             1102 1111 1115 1125 1128 1131 1134 1137 1141 1145 1148 1151 1167 1184 1194 1240
                             1247 1265 1272 1289 1321 1330 1339 1351 1361 1484 1496 1519 1527
183      MOVE . . . . . . .  907 937 957 983 984 985 986 987 988 1004 1011 1013 1025 1038 1052 1055 1060 1067
                             1072 1075 1078 1079 1080 1081 1082 1083 1091 1103 1112 1126 1129 1132 1135 1139
                             1143 1146 1149 1152 1160 1163 1169 1175 1177 1180 1181 1186 1191 1196 1201 1208
                             1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1229 1230
                             1231 1232 1233 1234 1235 1239 1241 1244 1248 1250 1251 1253 1254 1255 1257 1258
                             1259 1260 1264 1266 1269 1273 1275 1276 1278 1279 1291 1294 1299 1301 1303 1307
                             1313 1314 1315 1316 1317 1318 1319 1320 1322 1323 1327 1328 1331 1333 1334 1336
                             1338 1341 1342 1343 1344 1348 1349 1352 1354 1355 1357 1362 1364 1368 1374 1375
                             1376 1377 1378 1379 1380 1381 1414 1417 1422 1425 1445 1446 1447 1448 1450 1451
                             1452 1457 1464 1489 1502 1507 1508 1509 1517 1551 1561 1566 1571 1576 1581 1586
                             1591 1596 1601 1606 1611 1616 1621 1626 1627 1679 1683 1688 1693 1698 1703 1710
                             1715 1720 1725 1730 1735
5        OPEN . . . . . . .  931 951 989 1443 1483
62       PERFORM. . . . . .  879 880 885 886 888 890 892 908 909 915 916 934 935 941 943 948 949 954 955 961
                             962 967 968 973 974 1000 1005 1008 1023 1039 1092 1093 1116 1164 1165 1170 1172
                             1176 1178 1179 1182 1187 1188 1192 1197 1198 1202 1246 1256 1271 1280 1329 1340
                             1350 1359 1365 1369 1504 1510 1511 1532 1533
8        READ . . . . . . .  881 893 958 1014 1026 1085 1490 1514
1        REWRITE. . . . . .  1183
4        SEARCH . . . . . .  1058 1065 1413 1421
46       SET. . . . . . . .  883 895 1016 1028 1041 1057 1064 1084 1087 1363 1412 1420 1493 1499 1516 1522 1548
                             1550 1559 1560 1564 1565 1569 1570 1574 1575 1579 1580 1584 1585 1589 1590 1594
                             1595 1599 1600 1604 1605 1609 1610 1614 1615 1619 1620 1639 1643
2        STOP . . . . . . .  920 1434
4        STRING . . . . . .  1236 1261 1324 1345
33       WRITE. . . . . . .  938 1166 1292 1293 1295 1296 1297 1298 1300 1302 1305 1454 1459 1462 1465 1467 1469
                             1471 1512 1654 1655 1667 1668 1669 1740 1742 1744 1745 1746 1747 1748 1749 1750
```

(1)
Number of times the statement is used in the program

(2)
statement

(3)
Line numbers where the statement is used

Debugging with messages that have offset information

Some IWZ messages include offset information that you can use to identify the particular line of a program that failed.

To use this information:

1. Compile the program with the LIST option. This step produces an assembler listing file, which has a suffix of .wlist.
2. When you get a message that includes a traceback, find the offset information for the COBOL program. In the following example, the program is TEST, and the corresponding hexadecimal offset is 0x678 (highlighted in bold):

```
Traceback:
/opt/ibm/cobol/rte/usr/lib/libcob2_32r.so(+0x66b60)[0xf76f3b60]
```

```
/opt/ibm/cobol/rte/usr/lib/libcob2_32r.so(iwzWriteERRmsg+0x17)[0xf76f41f7]
/opt/ibm/cobol/rte/usr/lib/libcob2_32r.so(_iwzcBCD_CONV_Pckd_To_Int4+0x176)[0xf76be7b6]
test.out(TEST+0x678)[0x5655a844]
/lib/libc.so.6(__libc_start_main+0xf3)[0xf74b12d3]
    --- End of call chain ---
IWZ903S  The system detected a data exception.
IWZ901S  Program exits due to severe or critical error.
```

3. Look in the `.wlist` file for the hexadecimal offset. To the right of the hexadecimal offset, after the instructions bytes, is the COBOL source line number. In the following example, the line number corresponding to 0x678 is 174 (highlighted in bold):

```
000174:            COMPUTE x = FUNCTION FACTORIAL(Packed-Dec-05).
    000669  EC83 6A08               000174        sub   esp, 0x00000008
    00066C  046A                   000174        push  0x00000004
    00066E  0068 0000 E800         000174        push  OFFSET FLAT:LEVEL-01-PACKED-DECIMAL
    000673  00E8 0000 8300         000174        call  OFFSET
FLAT:_iwzcBCD_CONV_Pckd_To_Int4
    000678  C483 8910              000174        add   esp, 0x00000010
```

4. Look in your program listing for the statement.

Related references
“LIST” on page 268

Part 4. Targeting COBOL programs for certain environments

© Copyright IBM Corp. 2021

Chapter 17. Programming for a Db2 environment

In general, the coding for a COBOL program will be the same if you want the program to access a Db2 database. However, to retrieve, update, insert, and delete Db2 data and use other Db2 services, you must use SQL statements.

To communicate with Db2, do these steps:

- Code any SQL statements that you need, delimiting them with EXEC SQL and END-EXEC statements.
- Start Db2 if it is not already started.
- Compile with the SQL compiler option.
- Compile with the NODYNAM compiler option if the application was compiled using the Db2 stand-alone precompiler.

  Note: The NODYNAM compiler option is required for programs that contain EXEC CICS or EXEC SQL statements.

  If EXEC SQL statements are used in COBOL libraries that are loaded by a COBOL dynamic call, then one or more EXEC SQL statements must be in the main program. (Called Db2 APIs cannot be loaded using a COBOL dynamic call.)

Related concepts
"Db2 coprocessor" on page 325

Related tasks
"Using Db2 files" on page 142

"Coding SQL statements" on page 326
"Starting Db2 before compiling" on page 327
"Compiling with the SQL option" on page 327

Related references
"DYNAM" on page 262
SQL reference for Db2 Version 11.1 for Linux, UNIX, and Windows

Db2 coprocessor

When you use the Db2 coprocessor, the compiler handles your source programs that contain embedded SQL statements without your having to use a separate precompiler.

To use the Db2 coprocessor, specify the SQL compiler option.

When the compiler encounters SQL statements in the source program, it interfaces with the Db2 coprocessor. All text between EXEC SQL and END-EXEC statements is passed to the coprocessor. The coprocessor takes appropriate actions for the SQL statements and indicates to the compiler which native COBOL statements to generate for them.

Certain restrictions on the use of COBOL language that apply with the Db2 precompiler do not apply when you use the Db2 coprocessor:

- You can identify host variables used in SQL statements without using EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION statements.
- You can compile in batch a source file that contains multiple nonnested COBOL programs.
- The source program can contain nested programs.
- Extended source format is fully supported.

© Copyright IBM Corp. 2021

Related tasks
"Compiling with the SQL option" on page 327

Related references
"SQL" on page 279
"SRCFORMAT" on page 280

Coding SQL statements

Delimit SQL statements with EXEC SQL and END-EXEC. The EXEC SQL and END-EXEC delimiters must each be complete on one line. You cannot continue them across multiple lines. Do not code COBOL statements within EXEC SQL statements.

You also need to do these special steps:

- Code an EXEC SQL INCLUDE statement to include an SQL communication area (SQLCA) in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION of the outermost program. LOCAL-STORAGE is recommended for recursive programs.
- Define all host variables that you use in SQL statements in the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, or LINKAGE SECTION. However, you do not need to identify them with EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION.

You can use SQL statements even for large objects (such as BLOB and CLOB) and compound SQL.

Related tasks
"Using Db2 files and SQL
statements in the same program" on page 143
"Using SQL INCLUDE with
the Db2 coprocessor" on page 326
"Using binary items in
SQL statements" on page 327
"Determining the success of SQL statements" on page 327

Using SQL INCLUDE with the Db2 coprocessor

An SQL INCLUDE statement is treated identically to a native COBOL COPY statement (including the search path and the file suffixes used) when you use the SQL compiler option.

The following two lines are therefore treated the same way. (The period that ends the EXEC SQL INCLUDE statement is required.)

```
EXEC SQL INCLUDE name END-EXEC.
COPY name.
```

The *name* in an SQL INCLUDE statement follows the same rules as those for COPY *text-name* and is processed identically to a COPY *text-name* statement that does not have a REPLACING phrase.

COBOL does not use the Db2 environment variable DB2INCLUDE for SQL INCLUDE processing. If you use the DB2INCLUDE environment variable for SQL INCLUDE processing, you can concatenate it with the setting of the COBOL SYSLIB environment variable in the .profile file in your home directory or at the prompt in a Linux command shell. For example:

```
export SYSLIB=$DB2INCLUDE:$SYSLIB
```

Related references
Chapter 14, "Compiler-directing
statements," on page 291
COPY statement (*COBOL for Linux on x86 Language Reference*)

Using binary items in SQL statements

For binary data items that you specify in an EXEC SQL statement, you can define the data items as either USAGE COMP-5 or as USAGE BINARY, COMP, or COMP-4.

If you define the binary data items as USAGE BINARY, COMP, or COMP-4, use the TRUNC(BIN) option. (This technique might have a larger effect on performance than using USAGE COMP-5 on individual data items.) If instead TRUNC(OPT) or TRUNC(STD) is in effect, the compiler accepts the items but the data might not be valid because of the decimal truncation rules. You need to ensure that truncation does not affect the validity of the data.

Related concepts
"Formats for numeric data" on page 39

Related references
"TRUNC" on page 283

Determining the success of SQL statements

When Db2 finishes executing an SQL statement, Db2 sends a return code in the SQLCODE and SQLSTATE fields of the SQLCA structure to indicate whether the operation succeeded or failed. In your program, test the return code and take any necessary action.

Related references
SQL communications area (SQLCA) structure
SQLCODE, SQLSTATE, and SQLWARN information

Starting Db2 before compiling

Because the compiler works in conjunction with the Db2 coprocessor, you must start Db2 before you compile your program.

You can connect to the target database for the compilation before you start the compilation, or can have the compiler make the connection. To have the compiler connect, specify the database by either of these means:

- Use the DATABASE suboption in the SQL option.
- Name the database in the DB2DBDFT environment variable.

Compiling with the SQL option

The option string that you provide in the SQL compiler option is made available to the Db2 coprocessor. Only the Db2 coprocessor views the content of the string.

For example, the following cob2 command passes the database name SAMPLE and the Db2 options USER and USING to the coprocessor:

```
cob2 -q"sql('database sample user myname using mypassword')" mysql.cbl. . .
```

The following options, which were meaningful to and used by the Db2 precompiler, are ignored by the coprocessor:

- MESSAGES
- NOLINEMACRO
- OPTLEVEL
- OUTPUT
- SQLCA

- TARGET
- WCHARTYPE

Related tasks
“Separating Db2 suboptions” on page 328
“Using package and bind file-names” on page 328

Related references
“SQL” on page 279
DB2 Command Reference (Precompile)

Separating Db2 suboptions

Because of the concatenation of multiple SQL option specifications, you can separate Db2 suboptions (which might not fit in one CBL statement) into multiple CBL statements.

The options that you include in the suboption string are cumulative. The compiler concatenates these suboptions from multiple sources in the order that they are specified. For example, suppose that your source file mypgm.cbl has the following code:

```
cbl . . . SQL("string2") . . .
cbl . . . SQL("string3") . . .
```

When you issue the command `cob2 mypgm.cbl -q:"SQL('string1')"`, the compiler passes the following suboption string to the Db2 coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. If the compiler finds multiple instances of the same SQL suboption, the last specification of that suboption in the concatenated string takes effect. The compiler limits the length of the concatenated Db2 suboption string to 4 KB.

Using package and bind file-names

Two of the suboptions that you can specify with the SQL option are `package name` and `bind file name`. If you do not specify these names, default names are constructed based on the source file-name for a nonbatch compilation or on the first program for a batch compilation.

For subsequent nonnested programs of a batch compilation, the names are based on the PROGRAM-ID of each program.

For the package name, the base name (the source file-name or the PROGRAM-ID) is modified as follows:

- Names longer than eight characters are truncated to eight characters.
- Lowercase letters are folded to uppercase.
- Any character other than A-Z, 0-9, or _ (underscore) is changed to 0.
- If the first character is not alphabetic, it is changed to A.

Thus if the base name is 9123aB-cd, the package name is A123AB0C.

For the bind file-name, the suffix .bnd is added to the base name. Unless explicitly specified, the file-name is relative to the current directory.

Chapter 18. Developing COBOL programs for CICS

You can write CICS applications in COBOL and run them on the Linux workstation using CICS TXSeries or CICS TX.

To prepare COBOL applications to run under CICS, do these steps:

1. Ensure that your CICS administrator modified the region's environment file to set the environment variables COBPATH and LD_LIBRARY_PATH to include the runtime directory. Also, ensure that the CICS region was granted access to the runtime directory.

   The environment file is `/var/cics_regions/`*xxxxxxxxx*`/environment` (where *xxxxxxxx* is the name of the region).
2. Create the application by using an editor to do the following tasks:
   - Code your program using COBOL statements and CICS commands.
   - Create COBOL copybooks.
   - Create the CICS screen maps that your program uses.
3. Use the command `cicsmap` to process the screen maps.
4. Use the `cicstcl` command to translate the CICS commands with an integrated CICS translator and to compile and link the program. If you don't specify a file extension, `cicstcl` by default uses the COBOL source file extension `.cbl`.

   The following examples show how to use the `cicstcl` command to translate, compile, and link a sample COBOL program APPLCOB that runs under TXSeries or CICS TX. The `cicstcl` command generates the output module with the `.ibmcob` extension.

   - To compile, translate, and link-edit a CICS COBOL application, use the `-l IBMCOB` option to specify the source language as IBM COBOL, and the CICS COBOL program file extension can be `.ccp` or `.cbl`:

     ```
     cicstcl -l IBMCOB APPLCOB.ccp
     ```

     ```
     cicstcl -l IBMCOB APPLCOB.cbl
     ```
   - To compile, translate, and link-edit a CICS COBOL application to be used to debug using CEDF, use the `-e` option. To display CICS statement line numbers, use the `-d` option:

     ```
     cicstcl -e -l IBMCOB APPLCOB.cbl
     ```

     ```
     cicstcl -e -d -l IBMCOB APPLCOB.cbl
     ```
   - To compile, translate, and link-edit a CICS COBOL application to be used to debug, use the `-a` option:

     ```
     cicstcl -a -l IBMCOB APPLCOB.cbl
     ```
   - To compile, translate, and link-edit a CICS COBOL application by specifying the COPYBOOK path, set the SYSLIB environment variable to specify the directory of the COBOL source COPYBOOK path and then use the `cicstcl` command:

     ```
     export SYSLIB="/program_copybook_path"
     ```

     ```
     cicstcl -l IBMCOB APPLCOB.cbl
     ```
   - To compile, translate, and link-edit a CICS COBOL application by statically linking a COBOL module, set the USERLIB environment variable to specify the compiled object module path and then use the `cicstcl` command:

     ```
     export USERLIB="cobol_module.o"
     ```

© Copyright IBM Corp. 2021

```
cicstcl -l IBMCOB APPLCOB.cbl
```

Note: If you want to compile and link a CICS COBOL program without using the `cicstcl` command, use the `cob2` compiler command with the `-qcics` option. The following example shows how to compile and link a CICS COBOL application using the `cob2` command:

```
cob2 -qNOTHREAD -I/opt/ibm/cics/include -qcics -o APPLCOB.ibmcob APPLCOB.cbl -L/opt/ibm/
cics/lib -lcicsprIBMCOB
```

For detailed usage of the `cicstcl` command, see "cicstcl" in TXSeries for Multiplatforms documentation or "cicstcl" in CICS TX on Cloud documentation.

5. Define the resources for your application, such as transactions, application programs, and files, to the CICS region.

   CICS administrator authority is required to perform these actions.
6. Access the CICS region, for example by using the `cicsterm` command.
7. Run the application by entering the four-character transaction ID that is associated with the application.

Related concepts
“Integrated CICS translator” on page 335

Related tasks
“Coding COBOL programs to run under CICS” on page 330
“Compiling and running CICS programs” on page 334

TXSeries for Multiplatforms documentation
IBM CICS TX on Cloud documentation

Coding COBOL programs to run under CICS

To code a program to run under CICS, code CICS commands in the `PROCEDURE DIVISION` by using the `EXEC CICS` command format.

```
EXEC CICS command-name command-options
END-EXEC
```

CICS commands have the basic format shown above. Within EXEC commands, use the space as a word separator; do not use a comma or a semicolon. Do not code COBOL statements within `EXEC CICS` commands.

In general, the COBOL language is supported in a CICS environment. However, there are restrictions and considerations that you should be aware of when you code COBOL programs to run under CICS TXSeries or CICS TX.

Restrictions:

- Db2 files that will interoperate with CICS TXSeries or CICS TX must be created with `FILEMODE(SMALL)` in effect.
- Object-oriented programming and interoperability with Java™ are not supported. COBOL class definitions and methods cannot be run in a CICS environment.
- The source program must not contain any nested programs.
- COBOL programs that will run under CICS TXSeries or CICS TX must be 32 bit.

Do not use `EXEC`, `CICS`, or `END-EXEC` as variable names, and do not use user-specified parameters to the main program. In addition, it is recommended that you not use any of the following COBOL language elements:

- `FILE-CONTROL` entry in the `ENVIRONMENT DIVISION`

- `FILE SECTION` in the `DATA DIVISION`
- USE declaratives, except `USE FOR DEBUGGING`

The following COBOL statements are also not recommended for use in a CICS environment:

- `ACCEPT` format 1
- `CLOSE`
- `DELETE`
- `DISPLAY UPON CONSOLE`, `DISPLAY UPON SYSPUNCH`
- `MERGE`
- `OPEN`
- `READ`
- `REWRITE`
- `SORT`
- `START`
- `STOP` *literal*
- `WRITE`

Apart from some forms of the `ACCEPT` statement, mainframe CICS does not support any of the COBOL language elements in the preceding list. If you use any of those elements, be aware of the following limitations:

- The program is not completely portable to the mainframe CICS environment.
- In the case of a CICS failure, a backout (restoring the resources that are associated with the failed task) for resources that were updated by using the above statements will not be possible.

Restriction: There is no IBM Z host data format support for COBOL programs that are translated by the separate or integrated CICS translator and run on CICS TXSeries or CICS TX.

Related tasks
"Getting the system date under CICS" on page 331
"Making dynamic calls under CICS" on page 332
"Accessing SFS data" on page 334
"Calling between COBOL and C/C++ under CICS" on page 334

Related references
"Db2 file system" on page 117
"ADDR" on page 249
Appendix B, "IBM Z host data format considerations," on page 475

Getting the system date under CICS

To retrieve the system date in a CICS program, use a format-2 `ACCEPT` statement or the `CURRENT-DATE` intrinsic function.

You can use any of these format-2 `ACCEPT` statements in CICS to get the system date:

- `ACCEPT` *`identifier-2`* `FROM DATE` (two-digit year)
- `ACCEPT` *`identifier-2`* `FROM DATE YYYYMMDD`
- `ACCEPT` *`identifier-2`* `FROM DAY` (two-digit year)
- `ACCEPT` *`identifier-2`* `FROM DAY YYYYDDD`
- `ACCEPT` *`identifier-2`* `FROM DAY-OF-WEEK` (one-digit integer, where 1 represents Monday)

You can use this format-2 `ACCEPT` statement in CICS to get the system time:

- ACCEPT *identifier-2* FROM TIME

Alternatively, you can use the CURRENT-DATE intrinsic function, which can also provide the time.

These methods work in both CICS and non-CICS environments.

Do not use a format-1 ACCEPT statement in a CICS program.

Related tasks
"Assigning input from a screen or file (ACCEPT)" on page 30

Related references
CURRENT-DATE (*COBOL for Linux on x86 Language Reference*)

Making dynamic calls under CICS

You can use CALL *identifier* statements to make dynamic calls in the CICS environment. However, you must set the COBPATH environment variable correctly. You also must make sure that the called module has the correct name.

Consider the following example, in which alpha is a COBOL program that contains CICS statements:

```
WORKING-STORAGE SECTION.
01  WS-COMMAREA   PIC 9    VALUE ZERO.
77   SUBPNAME     PIC X(8) VALUE SPACES
. . .
PROCEDURE DIVISION.
    MOVE 'alpha' TO SUBPNAME.
    CALL SUBPNAME USING DFHEIBLK, DFHCOMMAREA, WS-COMMAREA.
```

You must pass the CICS control blocks DFHEIBLK and DFHCOMMAREA (as shown above) to alpha.

The source for alpha is in file alpha.ccp. Use the command cicstcl to translate, compile, and link alpha.ccp. COBOL defaults to uppercase names. Therefore, unless you change this default by using the PGMNAME(MIXED) compiler option, you need to name the source file ALPHA.ccp (not alpha.ccp) to produce ALPHA.ibmcob (not alpha.ibmcob).

Suppose that the CICS region is called green. Then file ALPHA.ibmcob must be copied to /var/cics_regions/green/bin, and the cicsadd command to add new resource definition must be used to define ALPHA as a CICS program. Your installation staff must add the following line to the file /var/cics_regions/green/environment:

```
COBPATH=/var/cics_regions/green/bin
```

Then the staff must shut down the CICS green region and restart it. If you put dynamically called programs in some other directory, make sure that your installation staff adds that directory to COBPATH and that the CICS servers have permission to access that directory.

Related tasks
"Making dynamic calls to shared libraries under CICS" on page 332
"Tuning the performance of dynamic calls under CICS" on page 333

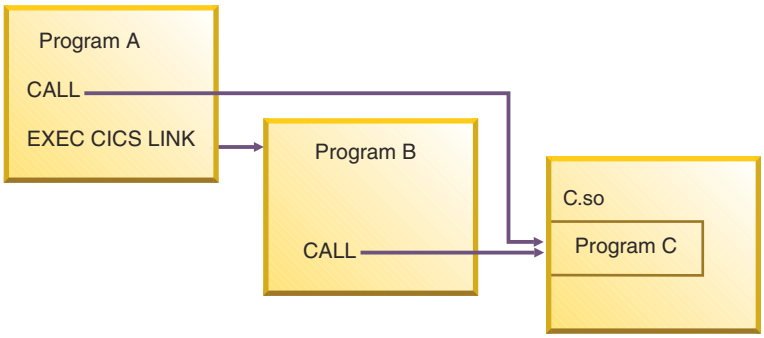
Related references
"Compiler and runtime environment variables" on page 214

Making dynamic calls to shared libraries under CICS

If you have a shared library that contains one or more COBOL programs, do not use it in more than one run unit within the same CICS transaction; otherwise the results are unpredictable.

The following figure shows a CICS transaction in which the same subprogram is called from two different run units:

- Program A calls Program C (in C.so).
- Program A links to Program B using an EXEC CICS LINK command. This combination becomes a new run unit within the same transaction.
- Program B calls Program C (in C.so).



Programs A and B share the same copy of Program C, and any changes to its state affect both programs.

In the CICS environment, programs in a shared library are initialized (whether by the WSCLEAR compiler option or by VALUE clause initialization) only on the first call within a run unit. If a COBOL subprogram is called more than once from either the same or different main programs, the subprogram is initialized on only the first call.

If you need the subprogram to be initialized on the first call from each main program, statically link a separate copy of the subprogram with each calling program. If you need the subprogram to be initialized on every call, use one of the following methods:

- Put data to be reinitialized in the LOCAL-STORAGE SECTION of the subprogram rather than in the WORKING-STORAGE SECTION. This placement affects initialization only by VALUE clauses, not by WSCLEAR.
- Use CANCEL to cancel the subprogram after each use so that the next call will be to the program in its initial state.
- Add the INITIAL attribute to the subprogram.

Related tasks
Chapter 24, “Using shared libraries,” on page 411

Related references
“WSCLEAR” on page 286

Tuning the performance of dynamic calls under CICS

The performance of persistent CICS transactions is improved by default in COBOL for Linux applications by means of module caching.

To enable or disable module caching, set the following environment variables:

```
export COBOL_CPM_CACHE=0    ## To disable caching
export COBOL_CPM_CACHE=1    ## To enable caching
```

If the COBOL_CPM_CACHE environment variable is not specified, the defaults are as follows:

- For a CICS transaction, caching is enabled by default.
- For a non-CICS program, caching is automatically controlled; the COBOL runtime decides if and when to enable caching.

When a module is cached, its execution semantics might change, because its WORKING-STORAGE data items that do not have VALUE clauses and for which there are no explicit initialization statements remain in last-used state. If a program in memory is not cached but is instead reloaded from disk, the last-used state of these uninitialized variables is lost.

Do not rely on the value of uninitialized data items. Either initialize such data items explicitly, or temporarily use the WSCLEAR compiler option to clear WORKING-STORAGE to binary zeros each time the program is entered.

WSCLEAR can impact performance because the option increases the time and possibly space required for program initialization. To obtain the best performance when module caching is in effect, review your application code and decide whether to explicitly initialize variables.

Related references
"WSCLEAR" on page 286

Accessing SFS data

If your program is not running under CICS, it can access SFS files through the SFS file system (the default file system used by CICS TXSeries or CICS TX).

Related tasks
"Identifying files" on page 112
"Identifying SFS files" on page 115
"Using SFS files" on page 145

Related references
"SFS file system" on page 119

Calling between COBOL and C/C++ under CICS

You can make a call under CICS from a COBOL to a C/C++ program or from a C/C++ program to a COBOL program only if the called program does not contain any CICS commands. (The calling program can contain CICS commands.)

COBOL programs can issue an EXEC CICS LINK or EXEC CICS XCTL command to a C/C++ program regardless of whether the C/C++ program contains CICS commands. Therefore, if your COBOL program calls a C/C++ program that contains CICS commands, use EXEC CICS LINK or EXEC CICS XCTL rather than the COBOL CALL statement.

Related tasks
"Calling between COBOL and C/C++ programs" on page 387

Compiling and running CICS programs

To compile COBOL for Linux CICS TXSeries or CICS TX programs, use the cob2 command.

TRUNC(BIN) is a recommended compiler option for a COBOL program that will run under CICS. However, if you are certain that the nontruncated values of BINARY, COMP, and COMP-4 data items conform to their PICTURE specifications, you might be able to improve program performance by using TRUNC(OPT).

You can use a COMP-5 data item instead of a BINARY, COMP, or COMP-4 data item as an EXEC CICS command argument. COMP-5 data items are treated like BINARY, COMP, or COMP-4 data items if TRUNC(BIN) is in effect.

You must use the PGMNAME(MIXED) compiler option for programs that use CICS Client.

Do not use the DYNAM or ADDR(64) compiler option when you translate a COBOL program (using either the separate or integrated CICS translator). All other COBOL compiler options are supported.

Runtime options: Use the FILESYS runtime option to specify the default file system if no file system has been specified for a file by means of an ASSIGN clause.

Related concepts
"Integrated CICS translator" on page 335

Related tasks
"Compiling from the command

line” on page 223
TXSeries for Multiplatforms documentation

Related references
“Compiler options” on page 245
“Conflicting
compiler options” on page 248
“FILESYS” on page 298

Integrated CICS translator

When you compile a COBOL program using the CICS compiler option, the COBOL compiler works with the integrated CICS translator to handle both native COBOL and embedded CICS statements in the source program.

When the compiler encounters CICS statements, and at other significant points in the source program, the compiler interfaces with the integrated CICS translator. All text between EXEC CICS and END-EXEC statements is passed to the translator. The translator takes appropriate actions and then returns to the compiler, typically indicating which native language statements to generate.

If you compile the COBOL program by using the `cicstcl` command, it uses the integrated CICS translator. The `cicstcl` command invokes the compiler with the appropriate suboptions of the CICS compiler option.

Although you can still translate embedded CICS statements separately, it is recommended that you use the integrated CICS translator instead. Certain restrictions that apply when you use the separate translator do not apply when you use the integrated translator, and using the integrated translator provides several advantages:

- You do not need to separately translate the EXEC CICS statements that are in copybooks.
- There is no intermediate file for a translated but not compiled version of the source program.
- Only one output listing instead of two is produced.
- REPLACE statements can affect EXEC CICS statements.
- You can compile programs that contain CICS statements in a batch compilation (compilation of a sequence of programs).
- Extended source format is fully supported.

Related tasks
“Coding COBOL programs to
run under CICS” on page 330
“Compiling and running CICS programs” on page 334

Related references
“CICS” on page 255
“SRCFORMAT” on page 280

Part 5. Using XML and COBOL together

© Copyright IBM Corp. 2021

Chapter 19. Processing XML input

You can process XML input in a COBOL program by using the XML PARSE statement.

The XML PARSE statement is the COBOL language interface to the high-speed XML parser that is part of the COBOL run time.

Processing XML input involves passing control between the XML parser and a processing procedure in which you handle parser events.

Use the following COBOL facilities to process XML input:

- The XML PARSE statement to begin XML parsing and to identify the source XML document and the processing procedure
- The processing procedure to control the parsing, that is, receive and process XML events and associated document fragments, and return to the parser for continued processing
- Special registers to exchange information between the parser and the processing procedure:
  - XML-CODE to receive the status of XML parsing and, in some cases, to return information to the parser
  - XML-EVENT to receive the name of each XML event from the parser
  - XML-NTEXT to receive XML document fragments that are returned as national character data
  - XML-TEXT to receive document fragments that are returned as alphanumeric data

Related concepts
"XML parser in COBOL" on page 339

Related tasks
"Accessing XML documents" on page 340
"Parsing XML documents" on page 341
"Handling XML PARSE exceptions" on page 349
"Terminating XML parsing" on page 352

Related references
"The encoding of XML documents" on page 346
Appendix E, "XML reference material," on page 523
Extensible Markup Language (XML)

XML parser in COBOL

COBOL for Linux provides an event-based interface that lets you parse XML documents and transform them to COBOL data structures.

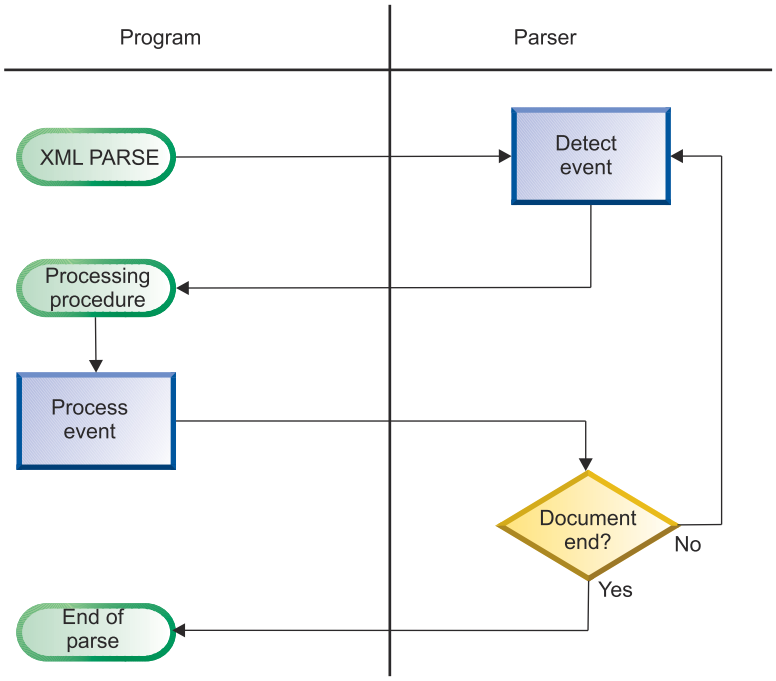
The XML parser finds fragments within the source XML document, and your processing procedure acts on those fragments. The fragments are associated with XML events; you code the processing procedure to handle each XML event.

Execution of the XML PARSE statement begins the parsing and establishes the processing procedure with the parser. The parser transfers control to the processing procedure for each XML event that it detects while processing the document. After processing the event, the processing procedure automatically returns control to the parser. Each normal return from the processing procedure causes the parser to continue analyzing the XML document to report the next event. Throughout this operation, control passes back and forth between the parser and the processing procedure.

In the XML PARSE statement, you can also specify two imperative statements to which you want control to be passed at the end of the parsing: one if a normal end occurs, and the other if an exception condition exists.

© Copyright IBM Corp. 2021

The following figure shows a high-level overview of the basic exchange of control between the parser and your COBOL program:



Normally, parsing continues until the entire XML document has been parsed.

The XML parser checks XML documents for most aspects of well formedness. A document is *well formed* if it adheres to the XML syntax in the *XML specification* and follows some additional rules such as proper use of end tags and uniqueness of attribute names.

Related concepts
"XML input document encoding" on page 346

Related tasks
"Parsing XML documents" on page 341
"Handling XML PARSE exceptions" on page 349
"Terminating XML parsing" on page 352

Related references
"The encoding of XML documents" on page 346
"XML conformance" on page 531
XML specification

Accessing XML documents

Before you can parse an XML document using an XML PARSE statement, you must make the document available to your program. Common methods of acquiring an XML document are from a parameter to your program or by reading the document from a file.

If the XML document that you want to parse is held in a file, use ordinary COBOL facilities to place the document into a data item in your program:

- A `FILE-CONTROL` entry to define the file to your program.
- An `OPEN` statement to open the file.
- `READ` statements to read all the records from the file into a data item (either an elementary item of category alphanumeric or national, or an alphanumeric or national group). You can define the data item in the `WORKING-STORAGE SECTION` or the `LOCAL-STORAGE SECTION`.

- Optionally, the STRING statement to string all of the separate records together into one continuous stream, to remove extraneous blanks, and to handle variable-length records.

Parsing XML documents

To parse XML documents, use the XML PARSE statement, specifying the XML document that is to be parsed and the processing procedure for handling XML events that occur during parsing, as shown in the following code fragment.

```
XML PARSE xml-document
    PROCESSING PROCEDURE xml-event-handler
  ON EXCEPTION
     DISPLAY 'XML document error ' XML-CODE
     STOP RUN
  NOT ON EXCEPTION
     DISPLAY 'XML document was successfully parsed.'
END-XML
```

In the XML PARSE statement, you first identify the *parse data item* (`xml-document` in the example above) that contains the XML document character stream. In the DATA DIVISION, define the parse data item as an elementary data item of category national or as a national group item if the encoding of the document is Unicode UTF-16; otherwise, define the parse data item as an elementary alphanumeric data item or an alphanumeric group item:

- If the parse data item is national, the XML document must be encoded in UTF-16 in little-endian format.
- If the parse data item is alphanumeric, its content must be encoded in one of the supported code pages described in the related reference about the encoding of XML documents.

Next, specify the name of the processing procedure (`xml-event-handler` in the example above) that is to handle the XML events that occur during parsing of the document.

In addition, you can specify either or both of the following optional phrases (as shown in the fragment above) to indicate the action to be taken after parsing finishes:

- ON EXCEPTION, to receive control if an unhandled exception occurs during parsing
- NOT ON EXCEPTION, to receive control otherwise

You can end the XML PARSE statement with the explicit scope terminator END-XML. Use END-XML to nest an XML PARSE statement that uses the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement.

The parser passes control to the processing procedure for each XML event. Control returns to the parser at the end of the processing procedure. This exchange of control between the XML parser and the processing procedure continues until one of the following events occurs:

- The entire XML document was parsed, as indicated by the END-OF-DOCUMENT event.
- The parser detects an error in the document and signals an EXCEPTION event, and the processing procedure does not reset the special register XML-CODE to zero before returning to the parser.
- The parsing process is terminated deliberately by the your code in the processing procedure that sets the XML-CODE special register to -1 before it returns to the parser.

Related concepts
“XML events” on page 343
“XML-CODE” on page 343

Related tasks
“Specifying the code page for character data” on page 200
“Writing procedures to process XML” on page 342
“Parsing XML documents encoded in UTF-8” on page 349

Related references
"The encoding of XML documents" on page 346
"XML PARSE exceptions" on page 523
XML PARSE statement (*COBOL for Linux on x86 Language Reference*)

Writing procedures to process XML

In your processing procedure, code statements to handle XML events.

For each event that the parser encounters, the parser passes information to the processing procedure in several special registers. Use the content of those special registers to populate COBOL data structures and to control the processing.

Examine the XML-EVENT special register to determine which event the parser passed to the processing procedure. XML-EVENT contains an event name, such as 'START-OF-ELEMENT'. Obtain the text associated with the event from the XML-TEXT or XML-NTEXT special register.

When used in nested programs, the XML special registers are implicitly defined as GLOBAL in the outermost program.

For additional details about the XML special registers, see the following table.

Table 36. ***Special registers used by the XML parser***

Special register	Implicit definition and usage	Content
XML-EVENT[1]	PICTURE X(30) USAGE DISPLAY VALUE SPACE	The name of the XML event
XML-CODE[2]	PICTURE S9(9) USAGE BINARY VALUE ZERO	An exception code or zero for each XML event
XML-TEXT[1, 3]	Variable-length elementary category alphanumeric item	Text (corresponding to the event that the parser encountered) from the XML document if you specify an alphanumeric item for the XML PARSE identifier
XML-NTEXT[1]	Variable-length elementary category national item	Text (corresponding to the event that the parser encountered) from the XML document if you specify a national item for the XML PARSE identifier

1. You cannot use this special register as a receiving data item.
2. The XML GENERATE statement also uses XML-CODE. Therefore, if you have an XML GENERATE statement in the processing procedure, save the value of XML-CODE before the XML GENERATE statement, and restore the saved value after the XML GENERATE statement.
3. The content of XML-TEXT has the encoding of the source XML document: ASCII or UTF-8 if the CHAR(NATIVE) compiler option is in effect; EBCDIC if CHAR(EBCDIC) is in effect.

Restrictions:

- A processing procedure must not directly execute an XML PARSE statement. However, if a processing procedure passes control to an outermost program by using a CALL statement, the target program can execute the same or a different XML PARSE statement. You can also execute the same XML statement or different XML statements simultaneously from a program that is running on multiple threads.
- The range of the processing procedure must not cause the execution of any GOBACK or EXIT PROGRAM statement, except to return control from a program to which control was passed by a CALL statement, respectively, that is executed in the range of the processing procedure.

  You can code a STOP RUN statement in a processing procedure to end the run unit.

The compiler inserts a return mechanism after the last statement in each processing procedure.

"Example: program for processing XML" on page 354

Related concepts
“XML events” on page 343
“XML-CODE” on page 343
“XML-TEXT and XML-NTEXT” on page 345

Related tasks
“Terminating XML parsing” on page 352

Related references
“CHAR” on page 253
XML-EVENT (*COBOL for Linux on x86 Language Reference*)

XML events

An *XML event* results when the XML parser detects various conditions (such as `END-OF-INPUT` or `EXCEPTION`) or encounters document fragments (such as `CONTENT-CHARACTERS` or `START-OF-CDATA-SECTION`) while processing an XML document.

For each event that occurs during XML parsing, the parser sets the associated event name in the `XML-EVENT` special register, and passes the `XML-EVENT` special register to the processing procedure. Depending on the event, the parser sets other special registers to contain additional information about the event.

In most cases, the parser sets the `XML-TEXT` or `XML-NTEXT` special register to the XML fragment that caused the event: `XML-NTEXT` if the XML document is in a national data item, or if the parser finds a character reference; otherwise, `XML-TEXT`.

When the parser detects an encoding conflict or a well-formedness error in the document, it sets `XML-EVENT` to `'EXCEPTION'` and provides additional information about the exception in the `XML-CODE` special register.

For a detailed description of the set of XML events, see the related reference about `XML-EVENT`.

Related concepts
“XML parser in COBOL” on page 339
“XML-CODE” on page 343

“XML-TEXT and XML-NTEXT” on page 345

Related tasks
“Writing procedures to process
XML” on page 342

Related references
“XML PARSE exceptions” on page 523
XML-EVENT (*COBOL for Linux on x86 Language Reference*)

XML-CODE

For each XML event except an `EXCEPTION` event, the parser sets the value of the `XML-CODE` special register to zero. For an `EXCEPTION` event, the parser sets `XML-CODE` to a value that identifies the specific exception.

For information about the possible exception codes, see the related references.

When the parser returns control to the `XML PARSE` statement from your processing procedure, `XML-CODE` generally contains the most recent value that was set by the parser. However, for any event other than `EXCEPTION`, if you set `XML-CODE` to `-1` in your processing procedure, parsing terminates with a user-initiated exception condition when control returns to the parser, and `XML-CODE` retains the value `-1`.

For an `EXCEPTION` XML event, your processing procedure can, in some cases, set `XML-CODE` to a meaningful value before control returns to the parser. (For details, see the related tasks about handling

XML PARSE exceptions and handling encoding conflicts.) If you set XML-CODE to any other nonzero value or set it for any other exception, the parser resets XML-CODE to the original exception code.

The following table shows the results of setting XML-CODE to various values. The leftmost column shows the type of XML event passed to the processing procedure; the other column headings show the XML-CODE value set by the processing procedure. The cell at the intersection of each row and column shows the action that the parser takes upon return from the processing procedure for a given combination of XML event and XML-CODE value.

Table 37. ***Results of processing-procedure changes to XML-CODE***

XML event type	-1	0	XML-CODE-100,000 (EBCDIC) XML-CODE-200,000 (ASCII)	Other nonzero value
Encoding-conflict exception (exception codes 50 - 99)	Ignores setting; keeps original XML-CODE value	Chooses encoding depending on the specific exception code[1]	Ignores setting; keeps original XML-CODE value	Ignores setting; keeps original XML-CODE value
Encoding-choice exception (exception codes > 100,000)	Ignores setting; keeps original XML-CODE value	Parses using the external code page[2]	Parses using the difference (shown above) as the encoding value[2]	Ignores setting; keeps original XML-CODE value
Other exception	Ignores setting; keeps original XML-CODE value	Limited continuation only for exception codes 1 - 49[3]	Ignores setting; keeps original XML-CODE value	Ignores setting; keeps original XML-CODE value
Normal event	Ends immediately; XML-CODE = -1[4]	[No apparent change to XML-CODE]	Ends immediately; XML-CODE = -1	Ends immediately; XML-CODE = -1

1. See the exception codes in the related reference about XML PARSE exceptions.
2. See the related task about handling encoding conflicts.
3. See the related task about handling XML PARSE exceptions.
4. See the related task about terminating XML parsing.

XML generation also uses the XML-CODE special register. For details, see the related task about handling XML GENERATE exceptions.

Related concepts
"How the XML parser handles errors" on page 350

Related tasks
"Writing procedures to process XML" on page 342
"Handling XML PARSE exceptions" on page 349
"Handling encoding conflicts" on page 351
"Terminating XML parsing" on page 352
"Handling XML GENERATE exceptions" on page 364

Related references
"XML PARSE exceptions" on page 523
"XML GENERATE exceptions" on page 533
XML-CODE (*COBOL for Linux on x86 Language Reference*)
XML-EVENT (*COBOL for Linux on x86 Language Reference*)

XML-TEXT and XML-NTEXT

For most XML events, the parser sets XML-TEXT or XML-NTEXT to an associated document fragment.

Typically, the parser sets XML-TEXT if the XML document is in an alphanumeric data item. The parser sets XML-NTEXT if:

- The XML document is in a national data item.
- The XML document is in an alphanumeric data item and the ATTRIBUTE-NATIONAL-CHARACTER or CONTENT-NATIONAL-CHARACTER event occurs.

The special registers XML-TEXT and XML-NTEXT are mutually exclusive. When the parser sets XML-TEXT, XML-NTEXT is empty with length zero. When the parser sets XML-NTEXT, XML-TEXT is empty with length zero.

To determine the number of character encoding units in XML-NTEXT, use the LENGTH intrinsic function; for example FUNCTION LENGTH(XML-NTEXT). To determine the number of bytes in XML-NTEXT, use special register LENGTH OF XML-NTEXT. The number of character encoding units differs from the number of bytes.

To determine the number of bytes in XML-TEXT, use either special register LENGTH OF XML-TEXT or the LENGTH intrinsic function; each returns the number of bytes.

The XML-TEXT and XML-NTEXT special registers are undefined outside the processing procedure.

Related concepts
“XML events” on page 343
“XML-CODE” on page 343

Related tasks
“Writing procedures to process XML” on page 342

Related references
XML-TEXT (*COBOL for Linux on x86 Language Reference*)
XML-NTEXT (*COBOL for Linux on x86 Language Reference*)

Transforming XML text to COBOL data items

Because XML data is neither fixed length nor fixed format, you need to use special techniques when you move XML data to a COBOL data item.

For alphanumeric items, decide whether the XML data should go at the left (default) end, or at the right end, of the COBOL data item. If the data should go at the right end, specify the JUSTIFIED RIGHT clause in the definition of the item.

Give special consideration to numeric XML values, particularly "decorated" monetary values such as '$1,234.00' or '$1234'. These two strings might mean the same thing in XML, but need quite different definitions if used as COBOL sending fields.

Use one of the following techniques when you move XML data to COBOL data items:

- If the format is reasonably regular, code a MOVE to an alphanumeric item that you redefine appropriately as a numeric-edited item. Then do the final move to a numeric (operational) item by moving from, and thus de-editing, the numeric-edited item. (A regular format would have the same number of digits after the decimal point, a comma separator for values greater than 999, and so on.)
- For simplicity and vastly increased flexibility, use the following intrinsic functions for alphanumeric XML data:
  - NUMVAL to extract and decode simple numeric values from XML data that represents plain numbers
  - NUMVAL-C to extract and decode numeric values from XML data that represents monetary quantities

  However, using these functions is at the expense of performance.

Related tasks
"Converting to numbers (NUMVAL, NUMVAL-C)" on page 105
"Using national data (Unicode) in COBOL" on page 177
"Writing procedures to process XML" on page 342

The encoding of XML documents

XML documents must be encoded in a supported code page.

XML documents that you parse using XML PARSE statements must be encoded, and XML documents that you create using XML GENERATE statements are encoded, as follows:

- Documents in national data items: in Unicode UTF-16 in little-endian format
- Documents in native alphanumeric data items: in Unicode UTF-8 or a single-byte ASCII code page that is supported by International Components for Unicode (ICU) conversion libraries

  A *native alphanumeric data item* is a category alphanumeric data item that is compiled with the CHAR(NATIVE) compiler option in effect or whose data description entry contains the NATIVE phrase.
- Documents in host alphanumeric data items: in a single-byte EBCDIC code page that is supported by ICU conversion libraries

  A *host alphanumeric data item* is a category alphanumeric data item that is compiled with the CHAR(EBCDIC) compiler option in effect and whose data description entry does not contain the NATIVE phrase.

The encodings supported by the ICU conversion libraries are documented in the related reference about the ICU converter explorer.

Related concepts
"XML input document encoding" on page 346

Related tasks
"Specifying the encoding" on page 348
"Parsing XML documents encoded in UTF-8" on page 349
Chapter 20, "Producing XML output," on page 359

Related references
"CHAR" on page 253
International Components for Unicode: Converter Explorer

XML input document encoding

To parse an XML document using the XML PARSE statement, the document must be encoded in a supported encoding.

The supported encodings for a given parse operation depend on the type of the data item that contains the XML document. The parser supports the following types of data items and encodings:

- Category national data items with content that is encoded in Unicode UTF-16 in little-endian format
- Native alphanumeric data items with content that is encoded in Unicode UTF-8 or one of the supported single-byte ASCII code pages
- Host alphanumeric data items with content that is encoded in one of the supported single-byte EBCDIC code pages

The supported code pages are described in the related reference about the encoding of XML documents.

The parser determines the *actual document encoding* by examining the first few bytes of the XML document. If the actual document encoding is ASCII or EBCDIC, the parser needs specific code-page

information to be able to parse correctly. This additional code-page information is acquired from the document encoding declaration or from the external code-page information.

The document encoding declaration is an optional part of the XML declaration at the beginning of the document. For details, see the related task about specifying the encoding.

The *external code page* for ASCII XML documents (the *external ASCII code page*) is the code page indicated by the current runtime locale. The external code page for EBCDIC XML documents (the *external EBCDIC code page*) is one of these:

- The code page that you specified in the EBCDIC_CODEPAGE environment variable
- The default EBCDIC code page selected for the current runtime locale if you did not set the EBCDIC_CODEPAGE environment variable

If the specified encoding is not one of the supported coded character sets, the parser signals an XML exception event before beginning the parse operation. If the actual document encoding does not match the specified encoding, the parser signals an appropriate XML exception after beginning the parse operation.

To parse an XML document that is encoded in an unsupported code page, first convert the document to national character data (UTF-16) by using the `NATIONAL-OF` intrinsic function. You can convert the individual pieces of document text that are passed to the processing procedure in special register `XML-NTEXT` back to the original code page by using the `DISPLAY-OF` intrinsic function.

XML declaration and white space:

XML documents can begin with *white space* only if they do not have an XML declaration:

- If an XML document begins with an XML declaration, the first angle bracket (<) in the document must be the first character in the document.
- If an XML document does not begin with an XML declaration, the first angle bracket in the document can be preceded only by white space.

White-space characters have the hexadecimal values shown in the following table.

Table 38. ***Hexadecimal values of white-space characters***

White-space character	EBCDIC	Unicode / ASCII
Space	X'40'	X'20'
Horizontal tabulation	X'05'	X'09'
Carriage return	X'0D'	X'0D'
Line feed	X'25'	X'0A'
New line / next line	X'15'	X'85'

Related tasks

"Converting to or from national (Unicode) representation" on page 184
"Specifying the encoding" on page 348
"Parsing XML documents encoded in UTF-8" on page 349
"Handling XML PARSE exceptions" on page 349

Related references

"Locales and code pages that are supported" on page 202
"The encoding of XML documents" on page 346
"EBCDIC code-page-sensitive characters in XML markup" on page 348
"XML PARSE exceptions" on page 523

Specifying the encoding

You can choose how to specify the encoding for parsing an XML document that is in an alphanumeric data item.

The preferred way is to omit the encoding declaration from the document and to rely instead on the external code-page specification.

Omitting the encoding declaration makes it possible to more easily transmit an XML document between heterogeneous systems. (If you included an encoding declaration, you would need to update it to reflect any code-page translation imposed by the transmission process.)

The code page used for parsing an alphanumeric XML document that does not have an encoding declaration is the runtime code page.

You can instead specify an encoding declaration in the XML declaration with which most XML documents begin. For example:

```
<?xml version="1.0" encoding="ibm-1140"?>
```

Note that the XML parser generates an exception if it encounters an XML declaration that does not begin in the first byte of an XML document.

If you specify an encoding declaration, use one of the primary or alias code-page names that are supported by the ICU conversion libraries. The code-page names are documented in the related reference about the ICU converter explorer.

For more information about the CCSIDs that are supported for XML parsing, see the related reference about the encoding of XML documents.

Related concepts
"XML input document encoding" on page 346

Related tasks
"Parsing XML documents
encoded in UTF-8" on page 349
"Handling encoding conflicts" on page 351

Related references
"Locales and code pages that are supported" on page 202
"The encoding of XML
documents" on page 346
International Components for Unicode: Converter Explorer

EBCDIC code-page-sensitive characters in XML markup

Several special characters that are used in XML markup have different hexadecimal representations in different EBCDIC code pages.

The following table shows those special characters and their hexadecimal values for various EBCDIC CCSIDs.

Table 39. ***Hexadecimal values of special characters for various EBCDIC CCSIDs***

Character	1047	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149
[	X'AD'	X'BA'	X'63'	X'9E'	X'B5'	X'90'	X'4A'	X'B1'	X'90'	X'4A'	X'AE'
]	X'BD'	X'BB'	X'FC'	X'9F'	X'9F'	X'51'	X'5A'	X'BB'	X'B5'	X'5A'	X'9E'
!	X'5A'	X'5A'	X'4F'	X'4F'	X'4F'	X'4F'	X'BB'	X'5A'	X'4F'	X'4F'	X'4F'
|	X'4F'	X'4F'	X'BB'	X'BB'	X'BB'	X'BB'	X'4F'	X'4F'	X'BB'	X'BB'	X'BB'
#	X'7B'	X'7B'	X'7B'	X'4A'	X'63'	X'B1'	X'69'	X'7B'	X'B1'	X'7B'	X'7B'

Parsing XML documents encoded in UTF-8

You can parse XML documents that are encoded in Unicode UTF-8 in a manner similar to parsing other XML documents. However, some additional requirements apply.

To parse a UTF-8 XML document, code the XML PARSE statement as you would normally for parsing XML documents:

```
XML PARSE xml-document
    PROCESSING PROCEDURE xml-event-handler
    . . .
END-XML
```

Observe the following additional requirements though:

- The parse data item (`xml-document` in the example above) must be category alphanumeric, and the `CHAR(EBCDIC)` compiler option must not be in effect.
- So that the XML document will be parsed as UTF-8 rather than ASCII, ensure that at least one of the following conditions applies:
  - The runtime locale is a UTF-8 locale.
  - The document contains an XML encoding declaration that specifies UTF-8 (`encoding="UTF-8"`).
  - The document starts with a UTF-8 byte order mark.
- The document must not contain any characters that have a Unicode scalar value that is greater than x'FFFF'. Use a character reference ("&#x*hhhhh*;") for such characters.

The parser returns the XML document fragments in the alphanumeric special register `XML-TEXT`.

UTF-8 characters are encoded using a variable number of bytes per character. Most COBOL operations on alphanumeric data assume a single-byte encoding, in which each character is encoded in 1 byte. When you operate on UTF-8 characters as alphanumeric data, you must ensure that the data is processed correctly. Avoid operations (such as reference modification and moves that involve truncation) that can split a multibyte character between bytes. You cannot reliably use statements such as `INSPECT` to process multibyte characters in alphanumeric data.

Related concepts
“XML-TEXT and XML-NTEXT” on page 345

Related tasks
“Processing UTF-8 data using UTF-16 (national) data types” on page 193
“Parsing XML documents” on page 341
“Specifying the encoding” on page 348

Related references
“CHAR” on page 253
“The encoding of XML documents” on page 346
XML PARSE statement (*COBOL for Linux on x86 Language Reference*)

Handling XML PARSE exceptions

If the XML parser encounters an anomaly or error during parsing, it sets an exception code in the `XML-CODE` special register and signals an XML exception event.

If the exception code is within a certain range, you might be able to handle the exception event within your processing procedure, and resume parsing.

To handle an exception in the processing procedure, follow these steps:

1. Check the contents of `XML-CODE`.
2. Handle the exception appropriately.

3. Set XML-CODE to zero to indicate that you handled the exception.
4. Return control to the parser.

The exception condition no longer exists.

You can handle exceptions in this way only if the exception code that is passed in XML-CODE is within one of the following ranges, which indicates that an encoding conflict was detected:

- 50 - 99
- 100,001 - 165,535
- 200,001 - 265,535

Exception codes 1 - 49: In the processing procedure, you can do limited handling of exceptions for which the exception code is within the range 1 - 49. After an exception in this range occurs, the parser does not signal any further normal events, except the END-OF-DOCUMENT event, even if you set XML-CODE to zero before returning. If you set XML-CODE to zero, the parser continues parsing the document and signals any exceptions that it finds. (Doing so can provide a useful way to discover multiple errors in the document.)

Restriction: The COBOL XML parser might not signal all additional exception events. The number of exceptions is limited to the remaining space in the XML PARSE event token array, probably 8192 events.

At the end of parsing after an exception that has an exception code in the range 1 - 49, control is passed to the statement specified in the ON EXCEPTION phrase. If you did not code an ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. XML-CODE contains the code set by the parser for the most recent exception.

For all exceptions other than those having an exception code within one of the ranges described above, the parser does not signal any further events, but passes control to the statement specified in the ON EXCEPTION phrase. XML-CODE contains the original exception code even if you set XML-CODE in the processing procedure before returning control to the parser.

If you do not want to handle an exception, return control to the parser without changing the value of XML-CODE. The parser transfers control to the statement specified in the ON EXCEPTION phrase. If you did not code an ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement.

If no unhandled exceptions occur before the end of parsing, control is passed to the statement specified in the NOT ON EXCEPTION phrase. If you did not code a NOT ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement. XML-CODE contains zero.

Related concepts
“XML-CODE” on page 343
“XML input document encoding” on page 346
“How the XML parser handles errors” on page 350

Related tasks
“Writing procedures to process XML” on page 342
“Handling encoding conflicts” on page 351

Related references
“The encoding of XML documents” on page 346
“XML PARSE exceptions” on page 523

How the XML parser handles errors

When the XML parser detects an error in an XML document, it generates an XML exception event and passes control to your processing procedure.

The parser passes the following information in special registers to the processing procedure:

- XML-EVENT contains 'EXCEPTION'.

- XML-CODE contains a numeric exception code.

  The exception codes are described in the related reference about XML PARSE exceptions.
- For fatal exceptions, XML-TEXT or XML-NTEXT contains the document text up to and including the point where the exception was detected.
- For the warning exceptions issued for using an undeclared prefix, XML-TEXT or XML-NTEXT contains the fully qualified attribute name or element name. That is, the name includes the undeclared prefix and the separator colon (:).
- XML-TEXT or XML-NTEXT contains the document text up to and including the point where the exception was detected.

All other XML special registers are empty with length zero.

The processing procedure might be able to handle an exception so that parsing continues if the exception code is within one of the following ranges:

- 1 - 99
- 100,001 - 165,535
- 200,001 - 265,535

If the exception code has any other nonzero value, parsing cannot continue.

Encoding conflicts: The exceptions for encoding conflicts (50 - 99 and 300 - 399) are signaled before the parsing of the document begins. For these exceptions, XML-TEXT or XML-NTEXT is either length zero or contains only the encoding declaration value from the document.

Exception codes 1 - 49: An exception for which the exception code is in the range 1 - 49 is a fatal error according to the *XML specification*. Therefore, the parser does not continue normal parsing even if the processing procedure handles the exception. However, the parser does continue scanning for further errors until it reaches the end of the document, or until the existing XML EVENT token array is exhausted. For these exceptions, the parser does not signal any further normal events except the END-OF-DOCUMENT event.

Related concepts
"XML events" on page 343
"XML-CODE" on page 343
"XML input document encoding" on page 346

Related tasks
"Handling XML PARSE exceptions" on page 349
"Handling encoding conflicts" on page 351

"Terminating XML parsing" on page 352

Related references
"The encoding of XML documents" on page 346
"XML PARSE exceptions" on page 523
XML specification

Handling encoding conflicts

Your processing procedure might be able to handle exceptions for document encoding conflicts. Exception events in which the parse data item is alphanumeric and the exception code in XML-CODE is within the range 100,001 - 165,535 or 200,001 - 265,535 indicate that the code page of the document (as specified by its encoding declaration) conflicts with the external code-page information.

In this special case, you can choose to parse using the code page of the document by subtracting 100,000 or 200,000 from the value in XML-CODE (depending on whether the code page is EBCDIC or ASCII, respectively). For instance, if XML-CODE contains 101,140, the code page of the document is

1140. Alternatively, you can choose to parse using the *external code page* by setting XML-CODE to zero before returning to the parser.

The parser takes one of three actions after returning from a processing procedure for an encoding-conflict exception event:

- If you set XML-CODE to zero, the parser uses the external ASCII code page or external EBCDIC code page, depending on whether the parse data item is a native alphanumeric or host alphanumeric item, respectively.
- If you set XML-CODE to the code page of the document (that is, the original XML-CODE value minus 100,000 or 200,000, as appropriate), the parser uses the code page of the document.

  This is the only case in which the parser continues when XML-CODE has a nonzero value upon returning from a processing procedure.
- Otherwise, the parser stops processing the document and returns control to the XML PARSE statement with an exception condition. XML-CODE contains the exception code that was originally passed with the exception event.

Related concepts
"XML-CODE" on page 343
"XML input document encoding" on page 346
"How the XML parser handles errors" on page 350

Related tasks
"Handling XML PARSE exceptions" on page 349

Related references
"The encoding of XML documents" on page 346
"XML PARSE exceptions" on page 523

Terminating XML parsing

You can terminate parsing immediately, without processing any remaining XML text, by setting XML-CODE to -1 in your processing procedure before the procedure returns to the parser from any normal XML event (that is, any event other than EXCEPTION).

You can use this technique when the processing procedure has examined enough of the document or has detected some irregularity in the document that precludes further meaningful processing.

If you terminate parsing in this way, the parser does not signal any further XML events, including the exception event. Control transfers to the ON EXCEPTION phrase of the XML PARSE statement, if that phrase was specified.

In the imperative statement of the ON EXCEPTION phrase, you can determine whether parsing was deliberately terminated by testing whether XML-CODE contains -1. If you do not specify the ON EXCEPTION phrase, control transfers to the end of the XML PARSE statement.

You can also terminate parsing after any XML EXCEPTION event by returning to the parser from the processing procedure without changing the value in XML-CODE. The result is similar to the result of deliberate termination, except that the parser returns to the XML PARSE statement with XML-CODE containing the original exception code.

Related concepts
"XML-CODE" on page 343
"How the XML parser handles errors" on page 350

Related tasks
"Writing procedures to process

XML” on page 342
“Handling XML PARSE exceptions” on page 349

XML PARSE examples

The examples that are referenced below illustrate various uses of the XML PARSE statement.

“Example:
parsing a simple document” on page 353
“Example: program
for processing XML” on page 354

Example: parsing a simple document

This example shows the flow of events and the contents of special register XML-TEXT that result from the parsing of a simple XML document.

Assume that the COBOL program contains the following XML document in data item Doc:

```
<?xml version="1.0"?><msg type="short">Hello, World!</msg>
```

The following code fragment shows an XML PARSE statement for parsing Doc, and a processing procedure, P, for handling the XML events:

```
XML Parse Doc
   Processing procedure P
   . . .
P. Display XML-Event XML-Text.
```

The processing procedure displays the content of XML-EVENT and XML-TEXT for each event that the parser signals during parsing. The following table shows the events and the text.

Table 40. ***XML events and special registers***

XML-EVENT	**XML-TEXT**
START-OF-DOCUMENT	
VERSION-INFORMATION	1.0
START-OF-ELEMENT	msg
ATTRIBUTE-NAME	type
ATTRIBUTE-CHARACTERS	short
CONTENT-CHARACTERS	Hello, World!
END-OF-ELEMENT	msg
END-OF-DOCUMENT	

Related concepts
“XML events” on page 343
“XML-TEXT and XML-NTEXT” on page 345

Example: program for processing XML

This example shows the parsing of an XML document, and a processing procedure that reports the various XML events and their associated text fragments.

The XML document is shown in the program source to make it easier to follow the flow of the parsing. The output of the program is shown after the example.

To understand the interaction of the parser and the processing procedure, and to match events to document fragments, compare the XML document to the output of the program.

```
Process codepage(1047)
       Identification division.
         Program-id. XMLSAMPL.
       Data division.
        Working-storage section.
      ****************************************************************
      * XML document data, encoded as initial values of data items.  *
      ****************************************************************
         1 xml-document-data.
          2 pic x(39) value '<?xml version="1.0" encoding="UTF-8"'.
          2 pic x(19) value ' standalone="yes"?>'.
          2 pic x(39) value '<!--This document is just an example-->'.
          2 pic x(10) value '<sandwich>'.
          2 pic x(33) value '<bread type="baker&apos;s best"/>'.
          2 pic x(36) value '<?spread We'll use real mayonnaise?>'.
          2 pic x(29) value '<meat>Ham &amp; turkey</meat>'.
          2 pic x(34) value '<filling>Cheese, lettuce, tomato, '.
          2 pic x(32) value 'and that's all, Folks!</filling>'.
          2 pic x(25) value '<![CDATA[We should add a '.
          2 pic x(20) value '<relish> element!]]>'.
          2 pic x(28) value '<listprice>$4.99</listprice>'.
          2 pic x(25) value '<discount>0.10</discount>'.
          2 pic x(31) value '</sandwich>'.
      ****************************************************************
      * XML document, represented as fixed-length records.           *
      ****************************************************************
         1 xml-document redefines xml-document-data.
          2 xml-segment pic x(40) occurs 10 times.
         1 xml-segment-no comp pic s9(4).
         1 content-buffer pic x(100).
         1 current-element-stack.
          2 current-element pic x(30) occurs 10 times.
      ****************************************************************
      * Sample data definitions for processing numeric XML content.  *
      ****************************************************************
         1 element-depth comp pic s9(4).
         1 discount computational pic 9v99 value 0.
         1 display-price pic $$9.99.
         1 filling pic x(4095).
         1 list-price computational pic 9v99 value 0.
         1 ofr-ed pic x(9) justified.
         1 ofr-ed-1 redefines ofr-ed pic 999999.99.
        Procedure division.
         Mainline section.
            Move 1 to xml-segment-no
            Display 'Initial segment {' xml-segment(xml-segment-no) '}'
            Display ' '
            XML parse xml-segment(xml-segment-no)
                processing procedure XML-handler
              On exception
                Display 'XML processing error, XML-Code=' XML-Code '.'
                Move 16 to return-code
                Goback
              Not on exception
                Display 'XML document successfully parsed.'
            End-XML
      ****************************************************************
      * Process the transformed content and calculate promo price.   *
      ****************************************************************
            Display ' '
            Display '-----++++++++++ Using information from XML '
                '++++++++++-----'
            Display ' '
            Move list-price to Display-price
            Display ' Sandwich list price: ' Display-price
            Compute Display-price = list-price * (1 - discount)
            Display ' Promotional price: ' Display-price
            Display ' Get one today!'
```

```
           Goback.
         XML-handler section.
            Evaluate XML-Event
       * ==> Order XML events most frequent first
              When 'START-OF-ELEMENT'
                Display 'Start element tag: {' XML-Text '}'
                Add 1 to element-depth
                Move XML-Text to current-element(element-depth)
              When 'CONTENT-CHARACTERS'
                Display 'Content characters: {' XML-Text '}'
       * ==> In general, a split can occur for any element or attribute
       * ==> data, but in this sample, it only occurs for "filling"...
                If xml-information = 2 and
                    current-element(element-depth) not = 'filling'
                  Display 'Unexpected split in content for element '
                     current-element(element-depth)
                  Move -1 to xml-code
                End-if
       * ==> Transform XML content to operational COBOL data item...
                Evaluate current-element(element-depth)
                  When 'filling'
       * ==> After reassembling separate pieces of character content...
                    String xml-text delimited by size into
                        content-buffer with pointer tally
                      On overflow
                        Display 'content buffer ('
                            length of content-buffer
                            ' bytes) is too small'
                        Move -1 to xml-code
                    End-string
                    Evaluate xml-information
                      When 2
                        Display '  Character data for element "filling" '
                            'is incomplete.'
                        Display '  The partial data was buffered for '
                            'content assembly.'
                      When 1
                        subtract 1 from tally
                        move content-buffer(1:tally) to filling
                        Display '  Element "filling" data (' tally
                            ' bytes) is now complete:'
                        Display '  {' filling(1:tally) '}'
                    End-evaluate
                  When 'listprice'
       * ==> Using function NUMVAL-C...
                    Move XML-Text to content-buffer
                    Compute list-price =
                        function numval-c(content-buffer)
                  When 'discount'
       * ==> Using de-editing of a numeric edited item...
                    Move XML-Text to ofr-ed
                    Move ofr-ed-1 to discount
                End-evaluate
              When 'END-OF-ELEMENT'
                Display 'End element tag: {' XML-Text '}'
                Subtract 1 from element-depth
              When 'END-OF-INPUT'
                Display 'End of input'
                Add 1 to xml-segment-no
                Display '  Next segment: {' xml-segment(xml-segment-no)
                    '}'
                Display ' '
                Move 1 to xml-code
              When 'START-OF-DOCUMENT'
                Display 'Start of document'
                Move 0 to element-depth
                Move 1 to tally
              When 'END-OF-DOCUMENT'
                Display 'End of document.'
              When 'VERSION-INFORMATION'
                Display 'Version: {' XML-Text '}'
              When 'ENCODING-DECLARATION'
                Display 'Encoding: {' XML-Text '}'
              When 'STANDALONE-DECLARATION'
                Display 'Standalone: {' XML-Text '}'
              When 'ATTRIBUTE-NAME'
                Display 'Attribute name: {' XML-Text '}'
              When 'ATTRIBUTE-CHARACTERS'
                Display 'Attribute value characters: {' XML-Text '}'
              When 'ATTRIBUTE-CHARACTER'
                Display 'Attribute value character: {' XML-Text '}'
              When 'START-OF-CDATA-SECTION'
```

```
              Display 'Start of CData section'
            When 'END-OF-CDATA-SECTION'
              Display 'End of CData section'
            When 'CONTENT-CHARACTER'
              Display 'Content character: {' XML-Text '}'
            When 'PROCESSING-INSTRUCTION-TARGET'
              Display 'PI target: {' XML-Text '}'
            When 'PROCESSING-INSTRUCTION-DATA'
              Display 'PI data: {' XML-Text '}'
            When 'COMMENT'
              Display 'Comment: {' XML-Text '}'
            When 'EXCEPTION'
              Compute tally = function length (XML-Text)
              Display 'Exception ' XML-Code ' at offset ' tally '.'
            When other
              Display 'Unexpected XML event: ' XML-Event '.'
          End-evaluate
          .
      End program XMLSAMPL.
```

Output from parsing

From the following output you can see which fragments of the document were associated with the events that occurred during parsing:

```
Start of document
Version: {1.0}
Encoding: {UTF-8}
Standalone: {yes}
Comment: {This document is just an example}
Start element tag: {sandwich}
Content characters: {  }
Start element tag: {bread}
Attribute name: {type}
Attribute value characters: {baker}
Attribute value character: {'}
Attribute value characters: {s best}
End element tag: {bread}
Content characters: {  }
PI target: {spread}
PI data: {please use real mayonnaise  }
Content characters: {  }
Start element tag: {meat}
Content characters: {Ham }
Content character: {&}
Content characters: { turkey}
End element tag: {meat}
Content characters: {  }
Start element tag: {filling}
Content characters: {Cheese, lettuce, tomato, etc.}
End element tag: {filling}
Content characters: {  }
Start of CData: {<![CDATA[}
Content characters: {We should add a <relish> element in future!}
End of CData: {]]>}
Content characters: {  }
Start element tag: {listprice}
Content characters: {$4.99 }
End element tag: {listprice}
Content characters: {  }
Start element tag: {discount}
Content characters: {0.10}
End element tag: {discount}
End element tag: {sandwich}
End of document.
XML document successfully parsed

-----++++++++++ Using information from XML **********-----

  Sandwich list price:  $4.99
  Promotional price:    $4.49
  Get one today!
```

Related concepts
“XML events” on page 343

Related references
XML-EVENT (*COBOL for Linux on x86 Language Reference*)

Chapter 20. Producing XML output

You can produce XML output from a COBOL program by using the XML GENERATE statement.

In the XML GENERATE statement, you identify the source and the output data items. You can optionally also identify:

- A field to receive a count of the XML characters generated
- The encoding for the generated XML document
- A *namespace* for the generated document
- A namespace prefix to qualify the start and end tag of each element, if you specify a namespace
- A user-defined element or attribute name in the generated XML document
- Attributes or elements to be suppressed according to some specified conditions
- Particular items to be specified as attributes, elements or content in the generated XML output.
- A statement to receive control if an exception occurs

Optionally, you can generate an XML declaration for the document, and can cause eligible source data items to be expressed as attributes in the output rather than as elements.

You can use the XML-CODE special register to determine the status of XML generation.

After you transform COBOL data items to XML, you can use the resulting XML output in various ways, such as deploying it in a web service, writing it to a file, or passing it as a parameter to another program.

Related tasks
"Generating XML output" on page 359
"Controlling the encoding
of generated XML output" on page 364
"Handling XML GENERATE exceptions" on page 364
"Enhancing XML output" on page 369

Related references
Extensible Markup Language (XML)
XML GENERATE statement (*COBOL for Linux on x86 Language Reference*)

Generating XML output

To transform COBOL data to XML, use the XML GENERATE statement as in the example below.

```
XML GENERATE XML-OUTPUT FROM SOURCE-REC
       COUNT IN XML-CHAR-COUNT
   ON EXCEPTION
       DISPLAY 'XML generation error ' XML-CODE
       STOP RUN
   NOT ON EXCEPTION
       DISPLAY 'XML document was successfully generated.'
END-XML
```

In the XML GENERATE statement, you first identify the data item (XML-OUTPUT in the example above) that is to receive the XML output. Define the data item to be large enough to contain the generated XML output, typically five to 10 times the size of the COBOL source data depending on the length of its data-name or data-names.

In the DATA DIVISION, you can define the receiving identifier as alphanumeric (either an alphanumeric group item or an elementary item of category alphanumeric) or as national (either a national group item or an elementary item of category national).

© Copyright IBM Corp. 2021

Next you identify the source data item that is to be transformed to XML format (SOURCE-REC in the example). The source data item can be an alphanumeric group item, national group item, or elementary data item of class alphanumeric or national.

Some COBOL data items are not transformed to XML, but are ignored. Subordinate data items of an alphanumeric group item or national group item that you transform to XML are ignored if they:

- Specify the REDEFINES clause, or are subordinate to such a redefining item
- Specify the RENAMES clause

These items in the source data item are also ignored when you generate XML:

- Elementary FILLER (or unnamed) data items
- Slack bytes inserted for SYNCHRONIZED data items

No extra white space (for example, new lines or indentation) is inserted to make the generated XML more readable.

Optionally, you can code the COUNT IN phrase to obtain the number of XML character encoding units that are filled during generation of the XML output. If the receiving identifier has category national, the count is in UTF-16 character encoding units. For all other encodings (including UTF-8), the count is in bytes.

You can use the count field as a reference modification length to obtain only that portion of the receiving data item that contains the generated XML output. For example, XML-OUTPUT(1:XML-CHAR-COUNT) references the first XML-CHAR-COUNT character positions of XML-OUTPUT.

Consider the following program excerpt:

```
01  doc pic x(512).
01  docSize pic 9(9) binary.
01  G.
    05  A pic x(3) value "aaa".
    05  B.
        10  C pic x(3) value "ccc".
        10  D pic x(3) value "ddd".
    05  E pic x(3) value "eee".
    . . .
    XML Generate Doc from G
```

The code above generates the following XML document, in which A, B, and E are expressed as child elements of element G, and C and D become child elements of element B:

```
<G><A>aaa</A><B><C>ccc</C><D>ddd</D></B><E>eee</E></G>
```

Alternatively, you can specify the ATTRIBUTES phrase of the XML GENERATE statement. The ATTRIBUTES phrase causes every eligible data item included in the generated XML document to be expressed as an attribute of the containing XML element, rather than as a child element of the containing XML element. To be eligible, the data item must be elementary, must have a name other than FILLER, and must not have an OCCURS clause in its data description entry. The containing XML element corresponds to the group data item that is immediately superordinate to the elementary data item. Optionally, you can specify more precise control of which data items should be expressed as attributes or elements by using the TYPE OF phrase.

For example, suppose that the XML GENERATE statement in the program excerpt above had instead been coded as follows:

```
XML Generate Doc from G with attributes
```

The code would then generate the following XML document, in which A and E are expressed as attributes of element G, and C and D become attributes of element B:

```
<G A="aaa" E="eee"><B C="ccc" D="ddd"></B></G>
```

Optionally, you can code the ENCODING phrase of the XML GENERATE statement to specify the encoding of the generated XML document. If you do not use the ENCODING phrase, the document encoding is

determined by the category of the receiving data item. For further details, see the related task below about controlling the encoding of generated XML output.

Optionally, you can code the XML-DECLARATION phrase to cause the generated XML document to have an XML declaration that includes version information and an encoding declaration. If the receiving data item is of category:

- National: The encoding declaration has the value UTF-16 (encoding="UTF-16").
- Alphanumeric: The encoding declaration is derived from the ENCODING phrase, if specified, or from the runtime locale or EBCDIC_CODEPAGE environment variable if the ENCODING phrase is not specified.

For example, the program excerpt below specifies the XML-DECLARATION phrase of XML GENERATE, and specifies encoding in UTF-8:

```
01  Greeting.
    05 msg  pic x(80)  value 'Hello, world!'.
    . . .
    XML Generate Doc from Greeting
        with Encoding "UTF-8"
        with XML-declaration
    End-XML
```

The code above generates the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?><Greeting><msg>Hello, world!</msg></Greeting>
```

If you do not code the XML-DECLARATION phrase, an XML declaration is not generated.

Optionally, you can code the NAMESPACE phrase to specify a *namespace* for the generated XML document. The namespace value must be a valid *Uniform Resource Identifier (URI)*, for example, a URL (Uniform Resource Locator); for further details, see the related concept about URI syntax below.

Specify the namespace in an identifier or literal of either category national or alphanumeric.

If you specify a namespace, but do not specify a namespace prefix (described below), the namespace becomes the *default namespace* for the document. That is, the namespace define on the root element applies by default to each element name in the document, including the root element.

For example, consider the following data definitions and XML GENERATE statement:

```
01  Greeting.
    05  msg  pic x(80)  value 'Hello, world!'.
01  NS  pic x(20)   value 'http://example'.
    . . .
    XML Generate Doc from Greeting
        namespace is NS
```

The resulting XML document has a default namespace (http://example), as follows:

```
<Greeting xmlns="http://example"><msg>Hello, world!</msg></Greeting>
```

If you do not specify a namespace, the element names in the generated XML document are not in any namespace.

Optionally, you can code the NAMESPACE-PREFIX phrase to specify a prefix to be applied to the start and end tag of each element in the generated document. You can specify a prefix only if you have specified a namespace as described above.

When the XML GENERATE statement is executed, the prefix value must be a valid XML name, but without the colon (:); see the related reference below about namespaces for details. The value can have trailing spaces, which are removed before the prefix is used.

Specify the namespace prefix in an identifier or literal of either category national or alphanumeric.

It is recommended that the prefix be short, because it qualifies the start and end tag of each element.

For example, consider the following data definitions and XML GENERATE statement:

```
01  Greeting.
    05  msg  pic x(80)  value 'Hello, world!'.
01  NS  pic x(20)   value 'http://example'.
01  NP  pic x(5)    value 'pre'.
    . . .
    XML Generate Doc from Greeting
        namespace is NS
        namespace-prefix is NP
```

The resulting XML document has an explicit namespace (`http://example`), and the prefix `pre` is applied to the start and end tag of the elements `Greeting` and `msg`, as follows:

```
<pre:Greeting xmlns:pre="http://example"><pre:msg>Hello, world!</pre:msg></
pre:Greeting>
```

Optionally, you can code the NAME phrase to specify attribute and element names in the generated XML document. The attribute and element names must be alphanumeric or national literals and must be legal names according to the XML 1.0 standard.

For example, consider the following data structure and XML GENERATE statement:

```
01 Msg.
    02 Msg-Severity pic 9 value 1.
    02 Msg-Date pic 9999/99/99 value "2012/04/12".
    02 Msg-Text pic X(50) value "Sell everything!".
01 Doc pic X(500).
```

```
    XML Generate Doc from Msg
        With attributes
        Name of Msg          is  "Message"
                Msg-Severity is  "Severity"
                Msg-Date     is  "Date"
                Msg-Text     is  "Text"
    End-XML
```

The resulting XML document is as follows:

```
<Message Severity="1" Date="2012/04/12" Text="Sell everything!"></Message>
```

Optionally, you can code the SUPPRESS phrase to specify whether individual data items are generated based on whether or not they meet certain criteria.

For example, consider the following data structure and XML GENERATE statement to suppress spaces and zeros:

```
01 G.
    02 SensitiveInfo.
        03 SSN pic x(11) value '123-45-6789'.
        03 HomeAddress pic x(50) value '123 Main St, Anytown, USA'.
    02 Aarray value spaces.
        03 A pic AAA occurs 5.
    02 Barray value spaces.
        03 B pic XXX occurs 5.
    02 Carray value zeros.
        03 C pic 999 occurs 5.
    Move 'abc' to A(1)
    Move 123 to C(3)
    XML Generate Doc from G
        Suppress SensitiveInfo
                 every nonnumeric element when space
                 every numeric element when zero
    End-XML
```

The resulting XML document is as follows:

```
<G>
   <Aarray><A>abc</A></Aarray>
```

```
   <Carray><C>123</C></Carray>
</G>
```

Optionally, you can use the TYPE OF phrase to specify whether individual data items are expressed as attributes, elements or content.

For example, consider the following data structure and XML GENERATE statement:

```
01 Msg.
   02 Msg-Severity pic 9 value 1.
   02 Msg-Date pic 9999/99/99 value "2012/04/12".
   02 Msg-Text pic X(50) value "Sell everything!".
01 Doc pic X(500).
    XML Generate Doc from Msg
        With attributes
        Type of Msg-Severity is attribute
                Msg-Date     is attribute
                Msg-Text     is element
    End-XML
```

The resulting XML document is as follows:

```
<Msg Msg-Severity="1" Msg-Date="2012/04/12">
       <Msg-Text>Sell everything!</Msg-Text></Msg>
```

In addition, you can specify either or both of the following phrases to receive control after generation of the XML document:

- ON EXCEPTION, to receive control if an error occurs during XML generation
- NOT ON EXCEPTION, to receive control if no error occurs

You can end the XML GENERATE statement with the explicit scope terminator END-XML. Code END-XML to nest an XML GENERATE statement that has the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement.

XML generation continues until either the COBOL source record has been transformed to XML or an error occurs. If an error occurs, the results are as follows:

- The XML-CODE special register contains a nonzero exception code.
- Control is passed to the ON EXCEPTION phrase, if specified, otherwise to the end of the XML GENERATE statement.

If no error occurs during XML generation, the XML-CODE special register contains zero, and control is passed to the NOT ON EXCEPTION phrase if specified or to the end of the XML GENERATE statement otherwise.

“Example: generating XML” on page 365

Related concepts
Uniform Resource Identifier (URI): Generic Syntax

Related tasks
“Controlling the encoding
of generated XML output” on page 364
“Handling XML GENERATE exceptions” on page 364
“Processing UTF-8 data using UTF-16 (national) data types” on page 193

Related references
XML GENERATE statement (*COBOL for Linux on x86 Language Reference*)
Extensible Markup Language (XML)
Namespaces in XML 1.0

Controlling the encoding of generated XML output

When you generate XML output by using the XML GENERATE statement, you can control the encoding of the output by the category of the data item that receives the output, and by identifying the document encoding using the WITH ENCODING phrase of the XML GENERATE statement.

If you specify the WITH ENCODING *codepage* phrase, *codepage* must identify one of the code pages supported for COBOL XML processing as described in the related reference below about the encoding of XML documents. If *codepage* is an integer, it must be a valid CCSID number. If *codepage* is of class alphanumeric or national, it must identify a code-page name that is supported by the International Components for Unicode (ICU) conversion libraries as shown in the converter explorer table referenced below.

If you do not code the WITH ENCODING phrase, the generated XML output is encoded as shown in the table below.

Table 41. ***Encoding of generated XML if the ENCODING phrase is omitted***

If you define the receiving XML identifier as:	**The generated XML output is encoded in:**
Native alphanumeric (CHAR(EBCDIC) is not in effect, or the data description contains the NATIVE phrase)	The ASCII or UTF-8 code page indicated by the runtime locale in effect
Host alphanumeric (CHAR(EBCDIC) is in effect, and the data description does not contain the NATIVE phrase)	The EBCDIC code page in effect[1]
National	UTF-16 in little-endian format

1. You can set the EBCDIC code page by using the EBCDIC_CODEPAGE environment variable. If the environment variable is not set, the encoding is in the default EBCDIC code page associated with the current runtime locale.

A byte order mark is not generated.

For details about how data items are converted to XML and how the XML element names and attributes names are formed from the COBOL data-names, see the related reference below about the operation of the XML GENERATE statement.

Related tasks
Chapter 11, “Setting the locale,” on page 199
“Setting environment variables” on page 213

Related references
“CHAR” on page 253
“The encoding of XML documents” on page 346
XML GENERATE statement (*COBOL for Linux on x86 Language Reference*)
Operation of XML GENERATE (*COBOL for Linux on x86 Language Reference*)
International Components for Unicode: Converter Explorer

Handling XML GENERATE exceptions

When an error is detected during generation of XML output, an exception condition exists. You can write code to check the XML-CODE special register, which contains a numeric exception code that indicates the error type.

To handle errors, use either or both of the following phrases of the XML GENERATE statement:

- ON EXCEPTION

- COUNT IN

If you code the ON EXCEPTION phrase in the XML GENERATE statement, control is transferred to the imperative statement that you specify. You might code an imperative statement, for example, to display the XML-CODE value. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML GENERATE statement.

When an error occurs, one problem might be that the data item that receives the XML output is not large enough. In that case, the XML output is not complete, and the XML-CODE special register contains error code 400.

You can examine the generated XML output by doing these steps:

1. Code the COUNT IN phrase in the XML GENERATE statement.

   The count field that you specify holds a count of the XML character encoding units that are filled during XML generation. If you define the XML output as national, the count is in UTF-16 character encoding units; for all other encodings (including for UTF-8), the count is in bytes.
2. Use the count field as a reference modification length to refer to the substring of the receiving data item that contains the XML characters that were generated until the point when the error occurred.

   For example, if XML-OUTPUT is the data item that receives the XML output, and XML-CHAR-COUNT is the count field, then XML-OUTPUT(1:XML-CHAR-COUNT) references the XML output.

Use the contents of XML-CODE to determine what corrective action to take. For a list of the exceptions that can occur during XML generation, see the related reference below.

Related tasks
"Referring to substrings of data items" on page 99

Related references

"XML GENERATE exceptions" on page 533
XML-CODE (*COBOL for Linux on x86 Language Reference*)

Example: generating XML

The following example simulates the building of a purchase order in a group data item, and generates an XML version of that purchase order.

Program XGFX uses XML GENERATE to produce XML output in elementary data item xmlPO from the source record, group data item purchaseOrder. Elementary data items in the source record are converted to character format as necessary, and the characters are inserted as the values of XML attributes whose names are derived from the data-names in the source record.

XGFX calls program Pretty, which uses the XML PARSE statement with processing procedure p to format the XML output with new lines and indentation so that the XML content can more easily be verified.

Program XGFX

```
Identification division.
  Program-id. XGFX.
Data division.
 Working-storage section.
   01 numItems pic 99 global.
   01 purchaseOrder global.
     05 orderDate pic x(10).
     05 shipTo.
       10 country pic xx value 'US'.
       10 name pic x(30).
       10 street pic x(30).
       10 city pic x(30).
       10 state pic xx.
       10 zip pic x(10).
     05 billTo.
```

```
       10 country pic xx value 'US'.
       10 name pic x(30).
       10 street pic x(30).
       10 city pic x(30).
       10 state pic xx.
       10 zip pic x(10).
     05 orderComment pic x(80).
     05 items occurs 0 to 20 times depending on numItems.
       10 item.
         15 partNum pic x(6).
         15 productName pic x(50).
         15 quantity pic 99.
         15 USPrice pic 999v99.
         15 shipDate pic x(10).
         15 itemComment pic x(40).
   01 numChars comp pic 999.
   01 xmlPO pic x(999).
Procedure division.
  m.
    Move 20 to numItems
    Move spaces to purchaseOrder

    Move '1999-10-20' to orderDate

    Move 'US' to country of shipTo
    Move 'Alice Smith' to name of shipTo
    Move '123 Maple Street' to street of shipTo
    Move 'Mill Valley' to city of shipTo
    Move 'CA' to state of shipTo
    Move '90952' to zip of shipTo

    Move 'US' to country of billTo
    Move 'Robert Smith' to name of billTo
    Move '8 Oak Avenue' to street of billTo
    Move 'Old Town' to city of billTo
    Move 'PA' to state of billTo
    Move '95819' to zip of billTo
    Move 'Hurry, my lawn is going wild!' to orderComment

    Move 0 to numItems
    Call 'addFirstItem'
    Call 'addSecondItem'
    Move space to xmlPO
    Xml generate xmlPO from purchaseOrder  count in numChars
        with xml-declaration  with attributes
        namespace 'http://www.example.com'  namespace-prefix 'po'
    Call 'pretty' using xmlPO value numChars
    Goback
    .

Identification division.
  Program-id. 'addFirstItem'.
Procedure division.
    Add 1 to numItems
    Move '872-AA' to partNum(numItems)
    Move 'Lawnmower' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 148.95 to USPrice(numItems)
    Move 'Confirm this is electric' to itemComment(numItems)
    Goback.
End program 'addFirstItem'.

Identification division.
  Program-id. 'addSecondItem'.
Procedure division.
    Add 1 to numItems
    Move '926-AA' to partNum(numItems)
    Move 'Baby Monitor' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 39.98 to USPrice(numItems)
    Move '1999-05-21' to shipDate(numItems)
    Goback.
End program 'addSecondItem'.

End program XGFX.
```

Program Pretty

```
Identification division.
  Program-id. Pretty.
Data division.
 Working-storage section.
   01 prettyPrint.
     05 pose pic 999.
     05 posd pic 999.
     05 depth pic 99.
     05 inx pic 999.
     05 elementName pic x(30).
     05 indent pic x(40).
     05 buffer pic x(998).
     05 lastitem pic 9.
       88 unknown value 0.
       88 xml-declaration value 1.
       88 element value 2.
       88 attribute value 3.
       88 charcontent value 4.
 Linkage section.
  1 doc.
   2 pic x occurs 16384 times depending on len.
  1 len comp-5 pic 9(9).
Procedure division using doc value len.
  m.
    Move space to prettyPrint
    Move 0 to depth
    Move 1 to posd pose
    Xml parse doc processing procedure p
    Goback
    .
  p.
    Evaluate xml-event
      When 'VERSION-INFORMATION'
        String '<?xml version="' xml-text '"' delimited by size
            into buffer with pointer posd
        Set xml-declaration to true
      When 'ENCODING-DECLARATION'
        String ' encoding="' xml-text '"' delimited by size
            into buffer with pointer posd
      When 'STANDALONE-DECLARATION'
        String ' standalone="' xml-text '"' delimited by size
            into buffer with pointer posd
      When 'START-OF-ELEMENT'
        Evaluate true
          When xml-declaration
            String '?>' delimited by size into buffer
                with pointer posd
            Set unknown to true
            Perform printline
            Move 1 to posd
          When element
            String '>' delimited by size into buffer
                with pointer posd
          When attribute
            String '">' delimited by size into buffer
                with pointer posd
        End-evaluate
        If elementName not = space
          Perform printline
        End-if
        Move xml-text to elementName
        Add 1 to depth
        Move 1 to pose
        Set element to true
        String '<' xml-text delimited by size
            into buffer with pointer pose
        Move pose to posd
      When 'ATTRIBUTE-NAME'
        If element
          String ' ' delimited by size into buffer
              with pointer posd
        Else
          String '" ' delimited by size into buffer
              with pointer posd
        End-if
        String xml-text '="' delimited by size into buffer
            with pointer posd
```

```
        Set attribute to true
      When 'ATTRIBUTE-CHARACTERS'
        String xml-text delimited by size into buffer
            with pointer posd
      When 'ATTRIBUTE-CHARACTER'
        String xml-text delimited by size into buffer
            with pointer posd
      When 'CONTENT-CHARACTERS'
        Evaluate true
          When element
            String '>' delimited by size into buffer
                with pointer posd
          When attribute
            String '">' delimited by size into buffer
                with pointer posd
        End-evaluate
        String xml-text delimited by size into buffer
            with pointer posd
        Set charcontent to true
      When 'CONTENT-CHARACTER'
        Evaluate true
          When element
            String '>' delimited by size into buffer
                with pointer posd
          When attribute
            String '">' delimited by size into buffer
                with pointer posd
        End-evaluate
        String xml-text delimited by size into buffer
            with pointer posd
        Set charcontent to true
      When 'END-OF-ELEMENT'
        Move space to elementName
        Evaluate true
          When element
            String '/>' delimited by size into buffer
                with pointer posd
          When attribute
            String '"/>' delimited by size into buffer
                with pointer posd
          When other
            String '</' xml-text '>' delimited by size
                into buffer with pointer posd
        End-evaluate
        Set unknown to true
        Perform printline
        Subtract 1 from depth
        Move 1 to posd
      When other
        Continue
    End-evaluate
    .
  printline.
    Compute inx = function max(0 2 * depth - 2) + posd - 1
    If inx > 120
      compute inx = 117 - function max(0 2 * depth - 2)
      If depth > 1
        Display indent(1:2 * depth - 2) buffer(1:inx) '...'
      Else
        Display buffer(1:inx) '...'
      End-if
    Else
      If depth > 1
        Display indent(1:2 * depth - 2) buffer(1:posd - 1)
      Else
        Display buffer(1:posd - 1)
      End-if
    End-if
    .
End program Pretty.
```

Output from program XGFX

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<po:purchaseOrder xmlns:po="http://www.example.com" orderDate="1999-10-20" orderComment="Hurry, my lawn
is going wild!">
  <po:shipTo country="US" name="Alice Smith" street="123 Maple Street" city="Mill Valley" state="CA"
zip="90952"/>
```

```
  <po:billTo country="US" name="Robert Smith" street="8 Oak Avenue" city="Old Town" state="PA"
zip="95819"/>
  <po:items>
    <po:item partNum="872-AA" productName="Lawnmower" quantity="1" USPrice="148.95" shipDate=" "
itemComment="Confirm...
  </po:items>
  <po:items>
    <po:item partNum="926-AA" productName="Baby Monitor" quantity="1" USPrice="39.98"
shipDate="1999-05-21" itemComme...
  </po:items>
</po:purchaseOrder>
```

Related tasks
Chapter 19, “Processing XML input,” on page 339

Related references
Operation of XML GENERATE (*COBOL for Linux on x86 Language Reference*)

Enhancing XML output

It might happen that the information that you want to express in XML format already exists in a group item in the DATA DIVISION, but you are unable to use that item directly to generate an XML document because of one or more factors.

For example:

- In addition to the required data, the item has subordinate data items that contain values that are irrelevant to the XML output document.
- The names of the required data items are unsuitable for external presentation, and are possibly meaningful only to programmers.
- The required data items are broken up into too many components, and should be output as the content of the containing group.

There are various ways that you can deal with such situations. One possible technique is to define a new data item that has the appropriate characteristics, and move the required data to the appropriate fields of this new data item. However, this approach is somewhat laborious and requires careful maintenance to keep the original and new data items synchronized.

A superior approach that addresses most such problems is to use the new optional phrases of the XML GENERATE statement in order to:

- Provide more meaningful and appropriate names for the selected elementary items and for the group items that contain them.
- Exclude irrelevant data items from the generated XML by suppressing them based on their values.

The example that is referenced below shows a way to do so.

“Example: enhancing XML output” on page 369

Related references
Operation of XML GENERATE (*COBOL for Linux on x86 Language Reference*)

Example: enhancing XML output

The following example shows how you can modify XML output.

Consider the following data structure. The XML that is generated from the structure suffers from several problems that can be corrected.

```
01  CDR-LIFE-BASE-VALUES-BOX.
    15  CDR-LIFE-BASE-VAL-DATE    PIC X(08).
    15  CDR-LIFE-BASE-VALUE-LINE  OCCURS  2 TIMES.
        20  CDR-LIFE-BASE-DESC.
            25 CDR-LIFE-BASE-DESC1 PIC X(15).
            25  FILLER            PIC X(01).
            25  CDR-LIFE-BASE-LIT  PIC X(08).
```

```
            25  CDR-LIFE-BASE-DTE  PIC X(08).
        20  CDR-LIFE-BASE-PRICE.
            25  CDR-LIFE-BP-SPACE  PIC 9(08).
            25  CDR-LIFE-BP-DASH   PIC X.
            25  CDR-LIFE-BP-SPACE1 PIC X(02).
        20  CDR-LIFE-BASE-PRICE-ED  REDEFINES
             CDR-LIFE-BASE-PRICE  PIC $$$.$$.
        20  CDR-LIFE-BASE-QTY.
            25  CDR-LIFE-QTY-SPACE   PIC X(08).
            25  CDR-LIFE-QTY-DASH    PIC X.
            25  CDR-LIFE-QTY-SPACE1  PIC X(03).
            25  FILLER               PIC X(02).
        20  CDR-LIFE-BASE-VALUE   PIC $$$9.99
            BLANK WHEN ZERO.
    15  CDR-LIFE-BASE-TOT-VALUE   PIC X(15)
```

When this data structure is populated with some sample values, and XML is generated directly from it and then formatted using program `Pretty` (shown in "Example: generating XML" on page 365), the result is as follows:

```
<CDR-LIFE-BASE-VALUES-BOX>
  <CDR-LIFE-BASE-VAL-DATE>01/02/03</CDR-LIFE-BASE-VAL-DATE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>First</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>01/01/01</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>23</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>123</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>.</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>000</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>$765.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>Second</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>02/02/02</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>34</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>234</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>.</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>000</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>$654.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-TOT-VALUE>Very high!</CDR-LIFE-BASE-TOT-VALUE>
</CDR-LIFE-BASE-VALUES-BOX>
```

This generated XML suffers from several problems:

- The element names are long and not very meaningful.
- Some fields that are elements should be attributes such as, `CDR-LIFE-BASE-VAL-DATE` and `CDR-LIFE-BASE-DESC1`.
- There is unwanted data, for example, `CDR-LIFE-BASE-LIT` and `CDR-LIFE-BASE-DTE`.
- Required data has an unnecessary parent. For example, `CDR-LIFE-BASE-DESC1` has parent `CDR-LIFE-BASE-DESC`.
- Other required fields are split into too many subcomponents. For example, `CDR-LIFE-BASE-PRICE` has three subcomponents for one amount.

These and other characteristics of the XML output can be remedied by using additional phrases of the XML GENERATE statement as follows:

- Use the NAME OF phrase to provide appropriate tag or attribute names.
- Use the TYPE OF … IS ATTRIBUTE phrase to select the fields which should be XML attributes rather than elements.
- Use the TYPE OF … IS CONTENT phrase to suppress tags for excessive subcomponents.
- Use the SUPPRESS … WHEN phrase to exclude fields that contain uninteresting values.

Here is an example of the XML GENERATE statement to address those problems:

```
XML generate Doc from CDR-LIFE-BASE-VALUES-BOX
  Count in tally
  Name of
          CDR-LIFE-BASE-VALUES-BOX
       is 'Base_Values'
          CDR-LIFE-BASE-VAL-DATE
       is 'Date'
          CDR-LIFE-BASE-DTE
       is 'Date'
          CDR-LIFE-BASE-VALUE-LINE
       is 'BaseValueLine'
          CDR-LIFE-BASE-DESC1
       is 'Description'
          CDR-LIFE-BASE-PRICE
       is 'BasePrice'
          CDR-LIFE-BASE-QTY
       is 'BaseQuantity'
          CDR-LIFE-BASE-VALUE
       is 'BaseValue'
          CDR-LIFE-BASE-TOT-VALUE
       is 'TotalValue'
  Type of
          CDR-LIFE-BASE-VAL-DATE is attribute
          CDR-LIFE-BASE-DESC1 is attribute
          CDR-LIFE-BP-SPACE   is content
          CDR-LIFE-BP-DASH    is content
          CDR-LIFE-BP-SPACE1  is content
          CDR-LIFE-QTY-SPACE  is content
          CDR-LIFE-QTY-DASH   is content
          CDR-LIFE-QTY-SPACE1 is content
  Suppress every nonnumeric when space
           every numeric when zero
```

The result of generating and formatting XML from the statement shown above is more usable:

```
<Base_Values Date="01/02/03">
  <BaseValueLine Description="First">
    <Date>01/01/01</Date>
    <BasePrice>23.00</BasePrice>
    <BaseQuantity>123.000</BaseQuantity>
    <BaseValue>$765.00</BaseValue>
  </BaseValueLine>
  <BaseValueLine Description="Second">
    <Date>02/02/02</Date>
    <BasePrice>34.00</BasePrice>
    <BaseQuantity>234.000</BaseQuantity>
    <BaseValue>$654.00</BaseValue>
  </BaseValueLine>
  <TotalValue>Very high!</TotalValue>
</Base_Values>
```

Note that the COBOL reserved word DATE can now be used as an XML tag name in the output and other characters that are illegal to use in COBOL data names, such and underscore _ can also be used.

Note that the COBOL reserved word DATE can now be used as an XML tag name in the output. Characters such as accented letters and period . that are illegal in single-byte data names can also be used.

Related references
Operation of XML GENERATE (*COBOL for Linux on x86 Language Reference*)
REPLACE statement (*COBOL for Linux on x86 Language Reference*)

Part 6. Working with more complex applications

© Copyright IBM Corp. 2021

Chapter 21. Porting applications between platforms

Your Linux x86 system has a different hardware and operating-system architecture than an IBM Z or IBM Power system. Because of these differences, some problems can arise as you move COBOL programs between these environments.

Your IBM Power system has a different hardware and operating-system architecture than an IBM Z or Linux x86 system.

The Related tasks and reference below describe some of the differences between development platforms, and provide instructions to help you minimize portability problems.

Related tasks
“Getting IBM Enterprise COBOL for z/OS applications to compile” on page 375
“Getting IBM Enterprise COBOL for z/OS applications to run: overview” on page 375
“Writing code to run with IBM Enterprise COBOL for z/OS” on page 379

Related references
Appendix A, “Summary of differences from IBM Enterprise COBOL for z/OS,” on page 467

Getting IBM Enterprise COBOL for z/OS applications to compile

If you move Enterprise COBOL programs from an IBM Z system to a Linux x86 system and compile them using IBM COBOL for Linux on x86, you need to choose the right compiler options and be aware of language features that differ from IBM Enterprise COBOL for z/OS. You can also use the COPY statement to help port programs.

Choosing the right compiler options: For additional information about Enterprise COBOL compiler options that affect portability, see the related reference about compiler options.

Allowing for language features of Enterprise COBOL: Several language features that are valid in Enterprise COBOL programs can create errors or unpredictable results when compiled with COBOL for Linux. For details, see the related reference about language elements.

Using the COPY statement to help port programs: In many cases, you can avoid potential portability problems by using the COPY statement to isolate platform-specific code. For example, you can include platform-specific code in a compilation for a given platform and exclude it from compilation for a different platform. You can also use the COPY REPLACING phrase to globally change nonportable source code elements, such as file-names.

Related tasks
“Setting environment variables” on page 213

Related references
“Compiler options” on page 467
“Language elements” on page 471
COPY statement (*COBOL for Linux on x86 Language Reference*)

Getting IBM Enterprise COBOL for z/OS applications to run: overview

After you download an Enterprise COBOL program and successfully compile it using IBM COBOL for Linux on x86, the next step is to run the program. In many cases, you can get the same results as on IBM z/OS without greatly modifying the source.

To assess whether to modify the source, you need to know how to fix the elements and behavior of the COBOL language that vary due to the underlying hardware or software architecture.

© Copyright IBM Corp. 2021

Related tasks
“Fixing differences caused by data representations” on page 376
“Fixing environment differences that affect portability” on page 378
“Fixing differences caused by language elements” on page 378

Fixing differences caused by data representations

To ensure the same behavior for your programs, you should understand the differences in certain ways of representing data, and take appropriate action.

Character data might be represented differently, depending on the USAGE clause that describes data items and the locale that is in effect at run time. COBOL stores signed packed-decimal in the same manner on both Linux on x86 and IBM z/OS. However, binary, external-decimal, floating-point, and unsigned packed-decimal data are by default represented differently.

Most programs behave the same on IBM z/OS and Linux on x86 regardless of the data representation.

Related tasks
“Handling differences in ASCII SBCS and EBCDIC SBCS characters” on page 376
“Handling differences in IEEE and hexadecimal data” on page 377
“Handling differences in ASCII multibyte and EBCDIC DBCS strings” on page 378

Related references
“Data representation” on page 467

Handling differences in ASCII SBCS and EBCDIC SBCS characters

To avoid problems with the different data representation between ASCII and EBCDIC characters, use the `CHAR(EBCDIC)` compiler option.

COBOL for Linux on x86 uses the ASCII character set, and Enterprise COBOL for z/OS uses the EBCDIC character set. Therefore, most characters have a different hexadecimal value, as shown in the following table.

Table 42. ***ASCII characters contrasted with EBCDIC***

Character	Hexadecimal value if ASCII	Hexadecimal value if EBCDIC
'0' through '9'	X'30' through X'39'	X'F0' through X'F9'
'a'	X'61'	X'81'
'A'	X'41'	X'C1'
blank	X'20'	X'40'

Also, code that depends on the EBCDIC hexadecimal values of character data probably fails when the character data has ASCII values, as shown in the following table.

Table 43. ***ASCII comparisons contrasted with EBCDIC***

Comparison	Evaluation if ASCII	Evaluation if EBCDIC
'a' < 'A'	False	True
'A' < '1'	False	True

*Table 43. **ASCII comparisons contrasted with EBCDIC** (continued)*		
Comparison	**Evaluation if ASCII**	**Evaluation if EBCDIC**
x >= '0'	If true, does not indicate whether *x* is a digit	If true, *x* is probably a digit
x = X'40'	Does not test whether *x* is a blank	Tests whether *x* is a blank

Because of these differences, the results of sorting character strings are different between EBCDIC and ASCII. For many programs, these differences have no effect, but you should be aware of potential logic errors if your program depends on the exact sequence in which some character strings are sorted. If your program depends on the EBCDIC collating sequence and you are porting it to the workstation, you can obtain the EBCDIC collating sequence by using `PROGRAM COLLATING SEQUENCE IS EBCDIC` or the `COLLSEQ(EBCDIC)` compiler option.

Related references
“CHAR” on page 253
“COLLSEQ” on page 256

Handling differences in IEEE and hexadecimal data

To avoid most problems with the different representation between IEEE and hexadecimal floating-point data, use the `FLOAT(BE)` compiler option.

COBOL for Linux on x86 represents floating-point data using the IEEE format. Enterprise COBOL for z/OS uses the IBM Z hexadecimal format. The following table summarizes the differences between normalized floating-point IEEE and normalized hexadecimal for `USAGE COMP-1` data and `USAGE COMP-2` data.

*Table 44. **IEEE contrasted with hexadecimal***				
Specification	**IEEE for `COMP-1` data**	**Hexadecimal for `COMP-1` data**	**IEEE for `COMP-2` data**	**Hexadecimal for `COMP-2` data**
Range	1.17E-38* to 3.37E+38*	5.4E-79* to 7.2E +75*	2.23E-308* to 1.67E+308*	5.4E-79* to 7.2E +75*
Exponent representation	8 bits	7 bits	11 bits	7 bits
Mantissa representation	23 bits	24 bits	53 bits	56 bits
Digits of accuracy	6 digits	6 digits	15 digits	16 digits
* Indicates that the value can be positive or negative.				

For most programs, these differences should create no problems. However, use caution when porting if your program depends on hexadecimal representation of data.

Performance consideration: In general, IBM Z floating-point representation makes a program run more slowly because the software must simulate the semantics of IBM Z hardware instructions. This is a consideration especially if the `FLOAT(BE)` compiler option is in effect and a program has a large number of floating-point calculations.

“Examples: numeric data and internal representation” on page 42

Related references
“FLOAT” on page 267

Handling differences in ASCII multibyte and EBCDIC DBCS strings

To obtain Enterprise COBOL behavior for alphanumeric data items that contain DBCS characters, use the `CHAR(EBCDIC)` and `SOSI` compiler options. To avoid problems with the different data representation between ASCII DBCS and EBCDIC DBCS characters, use the `CHAR(EBCDIC)` compiler option.

In alphanumeric data items, Enterprise COBOL double-byte character strings (containing EBCDIC DBCS characters) are enclosed in shift codes, and COBOL for Linux on x86 multibyte character strings (containing ASCII DBCS, UTF-8, or EUC characters) are not enclosed in shift codes. The hexadecimal values used to represent the same characters are also different.

In DBCS data items, Enterprise COBOL double-byte character strings are not enclosed in shift codes, but the hexadecimal values used to represent characters are different from the hexadecimal values used to represent the same characters in COBOL for Linux on x86 multibyte strings.

For most programs, these differences should not make porting difficult. However, if your program depends on the hexadecimal value of a multibyte string, or expects that an alphanumeric character string contains a mixture of single-byte characters and multibyte characters, use caution in your coding practices.

Related references
“CHAR” on page 253
“SOSI” on page 277

Fixing environment differences that affect portability

Differences in file-names and control codes between Linux x86 and IBM z/OS platforms can affect the portability of your programs.

File naming conventions on Linux x86 are very different from those on IBM z/OS. This difference can affect portability if you use file-names in your COBOL source programs. The following file-name, for example, is valid on Linux x86 but not on IBM z/OS (except in the z/OS UNIX file system):

```
/users/joesmith/programs/cobol/myfile.cbl
```

Case sensitivity: Unlike z/OS, Linux is case sensitive. Names used in source programs (such as uppercase file-names) should be named appropriately in Linux file directories.

Some characters that have no particular meaning on z/OS are interpreted as control characters by Linux. This difference can lead to incorrect processing of ASCII text files. Files should not contain any of the following characters:

- X'0A' (LF: line feed)
- X'0D' (CR: carriage return)
- X'1A' (EOF: end-of-file)

If you use device-dependent (platform-specific) control codes in your programs or files, these control codes can cause problems when you try to port the programs or files to platforms that do not support the control codes. As with all other platform-specific code, it is best to isolate such code as much as possible so that you can replace it easily when you move the application to another platform.

Fixing differences caused by language elements

In general, you can expect portable COBOL programs to behave the same way on Linux as they do on z/OS. However, be aware of the differences in file-status values used in I/O processing.

If your program responds to file-status data items, be concerned with two issues, depending on whether the program is written to respond to the first or the second file-status data item:

- If your program responds to the first file-status data item (*data-name-1*), be aware that values returned in the 9*n* range depend on the platform. If your program relies on the interpretation of a particular 9*n*

value (for example, 97), do not expect the value to have the same meaning on Linux that it has on z/OS. Instead, revise your program so that it responds to any 9*n* value as a generic I/O failure.

- If your program responds to the second file-status data item (*data-name-8*), be aware that the values returned depend on both the platform and file system. For example, the STL file system returns values with a different record structure on Linux than the VSAM file system does on z/OS. If your program relies on the interpretation of the second file-status data item, the program is probably not portable.

Related tasks
"Using file status keys" on page 166
"Using file system status codes" on page 168

Related references
FILE STATUS clause (*COBOL for Linux on x86 Language Reference*)
File status key (*COBOL for Linux on x86 Language Reference*)

Writing code to run with IBM Enterprise COBOL for z/OS

You can use IBM COBOL for Linux on x86 to develop new applications, and take advantage of the productivity gains and increased flexibility of using your Linux x86 system. However, when you develop COBOL programs, you need to avoid using features that are not supported by IBM Enterprise COBOL for z/OS.

Language features: COBOL for Linux supports several language features that are not supported by Enterprise COBOL. As you write code on Linux x86 that is intended to run on z/OS, avoid using these features:

- Code-page names as arguments to the `DISPLAY-OF` and `NATIONAL-OF` intrinsic functions
- `READ` statement using the `PREVIOUS` phrase
- `START` statement using <, <=, or `NOT >` in the `KEY` phrase
- `>>CALLINTERFACE` compiler directive

Compiler options: Several compiler options are not available on Enterprise COBOL. Do not use any of the following compiler options in your source code if you intend to port the code to z/OS:

- `BINARY(NATIVE)`
- `CALLINT` (treated as a comment)
- `CHAR(NATIVE)`
- `FLOAT(NATIVE)`

File names: Be aware of the difference in file-naming conventions between Linux and host file systems. Avoid hard-coding the names of files in your source programs. Instead, use mnemonic names that you define on each platform, and map them in turn to mainframe ddnames or environment variables. You can then compile your program to accommodate the changes in file-names without having to change the source code.

Specifically, consider how you refer to files in the following language elements:

- `ACCEPT` or `DISPLAY` target names
- `ASSIGN` clause
- `COPY` statement (*text-name* or *library-name*)

File suffixes: In COBOL for Linux, when you compile using one of the `cob2` commands, COBOL source files that have suffix .cbl or .cob are passed to the compiler. In mainframe COBOL, when you compile in the z/OS UNIX file system, however, only files that have suffix .cbl are passed to the compiler.

Nested programs: Multithreaded programs on the mainframe must be recursive. Therefore, avoid coding nested programs if you intend to port your programs to the mainframe and enable them to run in a multithreaded environment.

Chapter 22. Using subprograms

Many applications consist of several separately compiled programs that are linked together. If the programs call each other, they must be able to communicate. They need to transfer control and usually need access to common data.

COBOL programs that are nested within each other can also communicate. All the required subprograms for an application can be in one source file and thus require only one compilation.

Related concepts
"Main programs, subprograms, and calls" on page 381

Related tasks
"Ending and reentering main programs or subprograms" on page 381
"Calling nested COBOL programs" on page 382
"Calling nonnested COBOL programs" on page 385
"Calling between COBOL and C/C++ programs" on page 387
"Making recursive calls" on page 393

Main programs, subprograms, and calls

If a COBOL program is the first program in a run unit, that COBOL program is the *main program*. Otherwise, it and all other COBOL programs in the run unit are *subprograms*. No specific source-code statements or options identify a COBOL program as a main program or subprogram.

Whether a COBOL program is a main program or subprogram can be significant for either of two reasons:

- Effect of program termination statements
- State of the program when it is reentered after returning

In the PROCEDURE DIVISION, a program can call another program (generally called a *subprogram*), and this called program can itself call other programs. The program that calls another program is referred to as the *calling* program, and the program it calls is referred to as the *called* program. When the processing of the called program is completed, the called program can either transfer control back to the calling program or end the run unit.

The called COBOL program starts running at the top of the PROCEDURE DIVISION.

Related tasks
"Ending and reentering main programs or subprograms" on page 381
"Calling nested COBOL programs" on page 382
"Calling nonnested COBOL programs" on page 385
"Calling between COBOL and C/C++ programs" on page 387
"Making recursive calls" on page 393

Ending and reentering main programs or subprograms

Whether a program is left in its last-used state or its initial state, and to which caller it returns, can depend on the termination statements that you use.

To end execution in the main program, you must code a STOP RUN or GOBACK statement in the main program. STOP RUN terminates the run unit and closes all files opened by the main program and its called subprograms. Control is returned to the caller of the main program, which is often the operating system.

© Copyright IBM Corp. 2021

GOBACK has the same effect in the main program. An EXIT PROGRAM performed in a main program has no effect.

You can end a subprogram by using an EXIT PROGRAM, a GOBACK, or a STOP RUN statement. If you use an EXIT PROGRAM or a GOBACK statement, control returns to the immediate caller of the subprogram without the run unit ending. An implicit EXIT PROGRAM statement is generated if there is no next executable statement in a called program. If you end the subprogram with a STOP RUN statement, the effect is the same as it is in a main program: all COBOL programs in the run unit are terminated, and control returns to the caller of the main program.

A subprogram is usually left in its *last-used state* when it terminates with EXIT PROGRAM or GOBACK. The next time the subprogram is called in the run unit, its internal values are as they were left, except that return values for PERFORM statements are reset to their initial values. (In contrast, a main program is initialized each time it is called.)

There are some cases in which programs will be in their initial state:

- A subprogram that is dynamically called and then canceled will be in the initial state the next time it is called.
- A program that has the INITIAL clause in the PROGRAM-ID paragraph will be in the initial state each time it is called.
- Data items defined in the LOCAL-STORAGE SECTION will be reset to the initial state specified by their VALUE clauses each time the program is called.

Related concepts
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 11

Related tasks
"Calling nested COBOL programs" on page 382
"Making recursive calls" on page 393

Calling nested COBOL programs

By calling nested programs, you can create applications that use structured programming techniques. You can also call nested programs instead of PERFORM procedures to prevent unintentional modification of data items.

Use either CALL *literal* or CALL *identifier* statements to make calls to nested programs.

You can call a nested program only from its directly containing program unless you identify the nested program as COMMON in its PROGRAM-ID paragraph. In that case, you can call the *common program* from any program that is nested (directly or indirectly) in the same program as the common program. Only nested programs can be identified as COMMON. Recursive calls are not allowed.

Follow these guidelines when using nested program structures:

- Code an IDENTIFICATION DIVISION in each program. All other divisions are optional.
- Optionally make the name of each nested program unique. Although the names of nested programs are not required to be unique (as described in the related reference about scope of names), making the names unique could help make your application more maintainable. You can use any valid user-defined word or an alphanumeric literal as the name of a nested program.
- In the outermost program, code any CONFIGURATION SECTION entries that might be required. Nested programs cannot have a CONFIGURATION SECTION.
- Include each nested program in the containing program immediately before the END PROGRAM marker of the containing program.
- Use an END PROGRAM marker to terminate nested and containing programs.

Related concepts
"Nested programs" on page 383

Related references
“Scope of names” on page 384

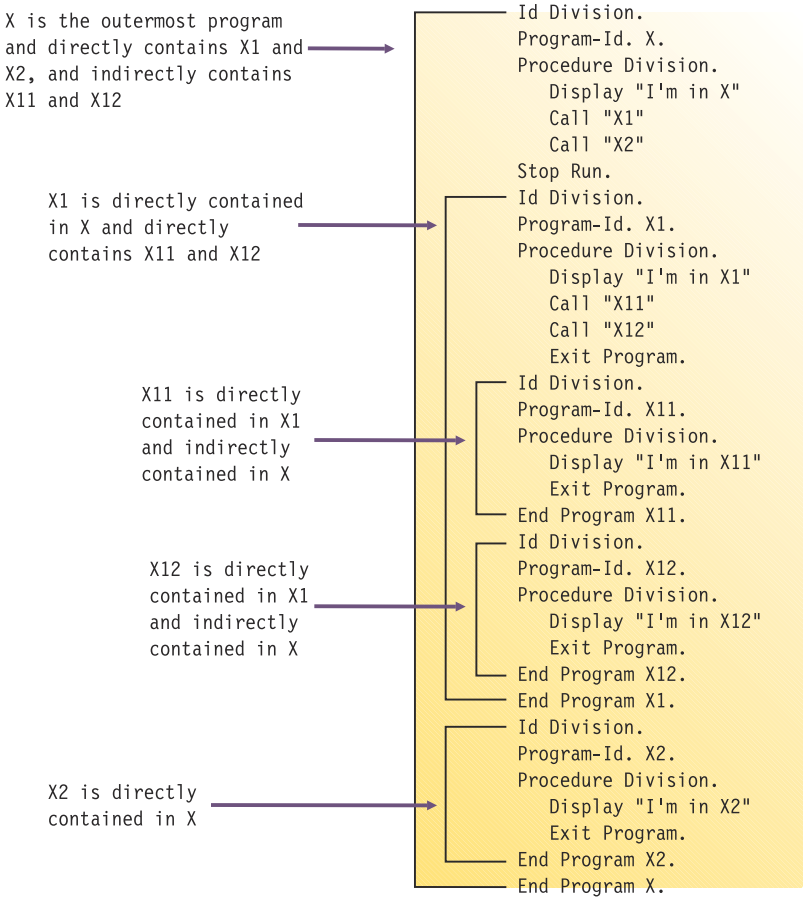
Nested programs

A COBOL program can *nest*, or contain, other COBOL programs. The nested programs can themselves contain other programs. A nested program can be directly or indirectly contained in a program.

There are four main advantages to nesting called programs:

- Nested programs provide a method for creating modular functions and maintaining structured programming techniques. They can be used analogously to perform procedures (using the PERFORM statement), but with more structured control flow and with the ability to protect local data items.
- Nested programs let you debug a program before including it in an application.
- Nested programs enable you to compile an application with a single invocation of the compiler.
- Calls to nested programs have the best performance of all the forms of COBOL CALL statements.

The following example describes a nested structure that has directly and indirectly contained programs:



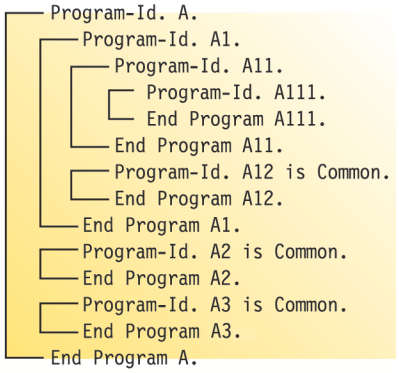
“Example: structure of nested programs” on page 384

Related tasks
“Calling nested COBOL programs” on page 382

Related references
“Scope of names” on page 384

Example: structure of nested programs

The following example shows a nested structure with some nested programs that are identified as COMMON.



The following table describes the calling hierarchy for the structure that is shown in the example above. Programs A12, A2, and A3 are identified as COMMON, and the calls associated with them differ.

This program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

In this example, note that:

- A2 cannot call A1 because A1 is not common and is not contained in A2.
- A1 can call A2 because A2 is common.

Scope of names

Names in nested structures are divided into two classes: local and global. The class determines whether a name is known beyond the scope of the program that declares it. A specific search sequence locates the declaration of a name after it is referenced in a program.

Local names

Names (except the program-name) are local unless declared to be otherwise. Local names are visible or accessible only within the program in which they are declared. They are not visible or accessible to contained and containing programs.

Global names

A name that is global (indicated by using the GLOBAL clause) is visible and accessible to the program in which it is declared and to all the programs that are directly and indirectly contained in that program. Therefore, the contained programs can share common data and files from the containing program simply by referencing the names of the items.

Any item that is subordinate to a global item (including condition-names and indexes) is automatically global.

You can declare the same name with the GLOBAL clause more than one time, provided that each declaration occurs in a different program. Be aware that you can mask, or hide, a name in a nested structure by having the same name occur in different programs in the same containing structure. However, such masking could cause problems during a search for a name declaration.

Searches for name declarations

When a name is referenced in a program, a search is made to locate the declaration for that name. The search begins in the program that contains the reference and continues outward to the containing programs until a match is found. The search follows this process:

1. Declarations in the program are searched.
2. If no match is found, only global declarations are searched in successive outer containing programs.
3. The search ends when the first matching name is found. If no match is found, an error exists.

The search is for a global name, not for a particular type of object associated with the name such as a data item or file connector. The search stops when any match is found, regardless of the type of object. If the object declared is of a different type than that expected, an error condition exists.

Calling nonnested COBOL programs

A COBOL program can call a subprogram that is linked into the same executable module as the caller (*static linking*) or that is provided in a shared library (*dynamic linking*). COBOL for Linux also provides for runtime resolution of a target subprogram from a shared library.

If you link a target program statically, it is part of the executable module of the caller and is loaded with the caller. If you link dynamically or resolve a call at run time, the target program is provided in a library and is loaded either when the caller is loaded or when the target program is called.

Either dynamic or static linking of subprograms is done for COBOL CALL *literal*. Runtime resolution is always done for COBOL CALL *identifier* and is done for CALL *literal* if the DYNAM option is in effect.

Restriction: You cannot mix 32-bit and 64-bit COBOL programs in an application. All program components within an application must be compiled using the same setting of the ADDR compiler option.

Related concepts
“CALL identifier and CALL literal” on page 385
“Static linking versus using shared libraries” on page 411

Related references
“ADDR” on page 249
“DYNAM” on page 262
CALL statement (*COBOL for Linux on x86 Language Reference*)

CALL identifier and CALL literal

CALL *identifier*, where *identifier* is a data item that contains the name of a nonnested subprogram at run time, always results in the target subprogram being loaded when it is called. CALL *literal*, where *literal* is the explicit name of a nonnested target subprogram, can be resolved either statically or dynamically.

With CALL *identifier*, the name of the executable or shared library must match the name of the target entry point.

With CALL *literal*, if the NODYNAM compiler option is in effect, either static or dynamic linking can be done. If DYNAM is in effect, CALL *literal* is resolved in the same way as CALL *identifier*: the target subprogram is loaded when it is called, and the name of the executable must match the name of the target entry point.

These call definitions apply only in the case of a COBOL program calling a nonnested program. If a COBOL program calls a nested program, the call is resolved by the compiler without any system intervention.

Limitation: Two or more separately linked executables in an application must not statically call the same nonnested subprogram.

Related concepts
"Static linking versus using shared libraries" on page 411

Related references
"DYNAM" on page 262
CALL statement (*COBOL for Linux on x86 Language Reference*)

Example: dynamic call using CALL identifier

The following example shows how you might make dynamic calls that use CALL *identifier*.

The first program, `dl1.cbl`, uses CALL *identifier* to call the second program, `dl1a.cbl`.

dl1.cbl

```
* Simple dynamic call to dl1a

 Identification Division.
 Program-id.    dl1.
*
 Environment Division.
 Configuration Section.
 Input-Output Section.
 File-control.
*
 Data Division.
 File Section.
 Working-storage Section.
 01 var pic x(10).
 Linkage Section.
*
 Procedure Division.
     move "Dl1A" to var.
     display "Calling " var.
     call var.
     move "dl1a     " to var.
     display "Calling " var.
     call var.
     stop run.
 End program dl1.
```

dlla.cbl

```
* Called by dl1.cbl using CALL identifier.

 IDENTIFICATION DIVISION.
 PROGRAM-ID. dl1a.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 OBJECT-COMPUTER. ANY-THING.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 num pic 9(4) binary value is zero.
*
 PROCEDURE DIVISION.
 LA-START.
     display "COBOL DL1A function." upon console.
     add 11 to num.
     display "num = " num
     goback.
```

Procedure

To create and run the example above, do these steps:

1. Enter `cob2 dl1.cbl -o dl1` to generate executable module `dl1`.
2. Enter `cob2 dl1a.cbl -o DL1A` to generate executable module `DL1A`.

   Unless you compile using the `PGMNAME(MIXED)` option, executable program-names are changed to uppercase.
3. Enter the command `export COBPATH=.` to cause the current directory to be searched for the targets of dynamic calls.
4. Enter `dl1` to run the program.

Because the `CALL` *identifier* target must match the name of the called program, the executable module in the above example is generated as `DL1A`, not as `dl1a`.

Related references
"PGMNAME" on page 274

Calling between COBOL and C/C++ programs

You can call functions written in C/C++ from COBOL programs and can call COBOL programs from C/C++ functions.

In an interlanguage application, you can combine 64-bit COBOL programs with 64-bit C/C++ functions, or 32-bit COBOL programs with 32-bit C/C++ functions.

Restriction:

- You cannot mix 32-bit components and 64-bit components in an application.
- The `ADDR(64)` and `-q64` options are not currently supported. Only 32-bit COBOL programs can be created at this time.

Interlanguage communication between COBOL and C++: In an interlanguage application that mixes COBOL and C++, follow these guidelines:

- Specify `extern "C"` in function prototypes for COBOL programs that are called from C++, and in C++ functions that are called from COBOL.
- In COBOL, use `BY VALUE` parameters to match the normal C++ parameter convention.
- In C++, use reference parameters to match the COBOL `BY REFERENCE` convention.

The rules and guidelines referenced below provide further information about how to perform these interlanguage calls.

Unqualified references to "C/C++" in the referenced sections are to GNU GCC compiler.

Related tasks
"Calling between COBOL and C/C++ under CICS" on page 334
"Initializing environments" on page 387
"Passing data between COBOL and C/C++" on page 388
"Collapsing stack frames and terminating run units or processes" on page 388
Chapter 23, "Sharing data," on page 395

Related references
"ADDR" on page 249
"COBOL and C/C++ data types" on page 389

Initializing environments

To make a call between C/C++ and COBOL, you must properly initialize the target environment.

If your main program is written in C/C++ and makes multiple calls to a COBOL program, use one of the following approaches:

- Preinitialize the COBOL environment in the C/C++ program before it calls any COBOL program. This approach is recommended because it provides the best performance.
- Put the COBOL program in an executable that is not part of the C/C++ routine that calls COBOL. Then every time that you want to call the main COBOL program, do the following steps in the C/C++ program:
  1. Load the program.
  2. Call the program.
  3. Unload the program.

Related concepts
Chapter 25, “Preinitializing the COBOL runtime environment,” on page 415

Passing data between COBOL and C/C++

Some COBOL data types have C/C++ equivalents, but others do not. When you pass data between COBOL programs and C/C++ functions, be sure to limit data exchange to appropriate data types.

By default, COBOL passes arguments BY REFERENCE. If you pass an argument BY REFERENCE, C/C++ gets a pointer to the argument. If you pass an argument BY VALUE, COBOL passes the actual argument. You can use BY VALUE only for the following data types:

- An alphanumeric character
- A USAGE NATIONAL character
- BINARY
- COMP
- COMP-1
- COMP-2
- COMP-4
- COMP-5
- FUNCTION-POINTER
- POINTER
- PROCEDURE-POINTER

“Example: COBOL program calling C functions” on page 390
“Example: C programs that are called by and call COBOL” on page 391
“Example: COBOL program calling C++ function” on page 392

Related tasks
Chapter 23, “Sharing data,” on page 395

Related references
“COBOL and C/C++ data types” on page 389

Collapsing stack frames and terminating run units or processes

Do not invoke functions in one language that collapse program stack frames of another language.

This guideline includes these situations:

- Collapsing some active stack frames from one language when there are active stack frames written in another language in the to-be-collapsed stack frames (C/C++ longjmp()).

- Terminating a run unit or process from one language while stack frames written in another language are active, such as issuing a COBOL STOP RUN or a C/C++ exit() or _exit(). Instead, structure the application in such a way that an invoked program terminates by returning to its invoker.

You can use C/C++ longjmp() or COBOL STOP RUN and C/C++ exit() or _exit() calls if doing so does not collapse active stack frames of a language other than the language that initiates that action. For the languages that do not initiate the collapsing and the termination, these adverse effects might otherwise occur:

- Normal cleanup or exit functions of the language might not be performed, such as the closing of files by COBOL during run-unit termination, or the cleanup of dynamically acquired resources by the involuntarily terminated language.
- User-specified exits or functions might not be invoked for the exit or termination, such as destructors and the C/C++ atexit() function.

In general, exceptions incurred during the execution of a stack frame are handled according to the rules of the language that incurs the exception. Because the COBOL implementation does not depend on the interception of exceptions through system services for the support of COBOL language semantics, you can specify the TRAP(OFF) runtime option to enable the exception-handling semantics of the non-COBOL language.

COBOL for Linux saves the exception environment at initialization of the COBOL runtime environment and restores it on termination of the COBOL environment. COBOL expects interfacing languages and tools to follow the same convention.

Related references
“TRAP” on page 300

COBOL and C/C++ data types

The following table shows the correspondence between the data types that are available in COBOL and C/C++.

Table 45. ***COBOL and C/C++ data types***

C/C++ data types	**COBOL data types**
wchar_t	USAGE NATIONAL (PICTURE N)
char	PIC X
signed char	No appropriate COBOL equivalent
unsigned char	No appropriate COBOL equivalent
short signed int	PIC S9-S9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
short unsigned int	PIC 9-9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long int	PIC 9(5)-9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long long int	PIC 9(10)-9(18) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
float	COMP-1
double	COMP-2
enumeration	Analogous to level 88, but not identical
char(*n*)	PICTURE X(*n*)

Table 45. **COBOL and C/C++ data types** *(continued)*	
C/C++ data types	**COBOL data types**
array pointer (*) to type	No appropriate COBOL equivalent
pointer(*) to function	PROCEDURE-POINTER or FUNCTION-POINTER

Related tasks
"Passing data between COBOL and C/C++" on page 388

Related references
"TRUNC" on page 283

Example: COBOL program calling C functions

The following example shows a COBOL program that calls C functions by using the CALL statement.

The example illustrates the following concepts:

- The CALL statement does not indicate whether the called program is written in COBOL or C.
- COBOL supports calling programs that have mixed-case names.
- You can pass arguments to C programs in various ways (for example, BY REFERENCE or BY VALUE).
- You must declare a function return value on a CALL statement that calls a non-void C function.
- You must map COBOL data types to appropriate C data types.

```
 CBL PGMNAME(MIXED)
* This compiler option allows for case-sensitive names for called programs.
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "COBCALLC".
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 N4              PIC 9(4)  COMP-5.
 01 NS4             PIC S9(4) COMP-5.
 01 N9              PIC 9(9)  COMP-5.
 01 NS9             PIC S9(9) COMP-5.
 01 NS18            USAGE COMP-2.
 01 D1              USAGE COMP-2.
 01 D2              USAGE COMP-2.
 01 R1.
     02 NR1         PIC 9(8) COMP-5.
     02 NR2         PIC 9(8) COMP-5.
     02 NR3         PIC 9(8) COMP-5.
 PROCEDURE DIVISION.
     MOVE 123 TO N4
     MOVE -567 TO NS4
     MOVE 98765432 TO N9
     MOVE -13579456 TO NS9
     MOVE 222.22 TO NS18
     DISPLAY "Call MyFun with n4=" N4 " ns4=" NS4 " N9=" n9
     DISPLAY "                 ns9="  NS9" ns18="  NS18
* The following CALL illustrates several ways to pass arguments.
*
     CALL "MyFun" USING N4 BY VALUE NS4 BY REFERENCE N9 NS9 NS18
     MOVE 1024 TO N4
* The following CALL returns the C function return value.
*
     CALL "MyFunR" USING BY VALUE N4 RETURNING NS9
     DISPLAY "n4=" N4 " and ns9= n4 times n4= " NS9
     MOVE -357925680.25 TO D1
     CALL "MyFunD" USING BY VALUE D1 RETURNING D2
     DISPLAY "d1=" D1 " and d2= 2.0 times d2= " D2
     MOVE 11111 TO NR1
     MOVE 22222 TO NR2
     MOVE 33333 TO NR3
     CALL "MyFunV" USING R1
     STOP RUN.
```

Related tasks
“Passing data between COBOL and C/C++” on page 388

Related references
CALL statement (*COBOL for Linux on x86 Language Reference*)

Example: C programs that are called by and call COBOL

The following example illustrates that a called C function receives arguments in the order in which they were passed in a COBOL CALL statement.

The file MyFun.c contains the following source code, which calls the COBOL program tprog1:

```
#include <stdio.h>
extern void TPROG1(double *);
void
MyFun(short *ps1, short s2, long *k1, long *k2, double *m)
{
    double x;
    x = 2.0*(*m);
    printf("MyFun got s1=%d s2=%d k1=%d k2=%d x=%f\n",
             *ps1, s2, *k1,*k2, x);
}

long
MyFunR(short s1)
{
    return(s1 * s1);
}

double
MyFunD(double d1)
{
    double z;
    /* calling COBOL */
    z = 1122.3344;
    (void) TPROG1(&z);
    /* returning a value to COBOL */
    return(2.0 * d1);
}

void
MyFunV(long *pn)
{
    printf("MyFunV got %d %d %d\n", *pn, *(pn+1), *(pn+2));
}
```

MyFun.c consists of the following functions:

MyFun
Illustrates passing a variety of arguments.

MyFunR
Illustrates how to pass and return a long variable.

MyFunD
Illustrates C calling a COBOL program and illustrates how to pass and return a double variable.

MyFunV
Illustrates passing a pointer to a record and accessing the items of the record in a C program.

Example: COBOL program called by a C program

The following example shows how to write COBOL programs that are called by C programs.

The COBOL program tprog1 is called by the C function MyFunD in program MyFun.c (see “Example: C programs that are called by and call COBOL” on page 391). The called COBOL program contains the following source code:

```
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TPROG1.
*
 DATA DIVISION.
 LINKAGE SECTION.
*
 01 X                USAGE COMP-2.
*
 PROCEDURE DIVISION USING X.
     DISPLAY "TPROG1 got x= " X
     GOBACK.
```

Related tasks
“Calling between COBOL and C/C++ programs” on page 387

Example: results of compiling and running examples

This example shows how you can compile, link, and run COBOL programs cobcallc.cbl and tprog.cbl and the C program MyFun.c, and shows the results of running the programs.

Compile and link cobcallc.cbl, tprog.cbl, and MyFun.c by issuing the following commands:

1. `gcc -m32 -c MyFun.c`
2. `cob2 cobcallc.cbl MyFun.o tprog1.cbl -o cobcallc`

Run the program by issuing the command `cobcallc`. The results are as follows:

```
call MyFun with n4=00123 ns4=-00567 n9=0098765432
            ns9=-0013579456 ns18=.222220000000000000E 03
MyFun got s1=123 s2=-567 k1=98765432 k2=-13579456 x=444.440000
n4=01024 and ns9= n4 times n4= 0001048576
TPROG1 got x=  .112233440000000000E 04
d1=-.35792568025000000E 09 and d2= 2.0 times d2= -.71585136050000000E 09
MyFunV got 11111 22222 33333
```

“Example: COBOL program calling C functions” on page 390
“Example: C programs that are called by and call COBOL” on page 391
“Example: COBOL program called by a C program” on page 392

Example: COBOL program calling C++ function

The following example shows a COBOL program that calls a C++ function by using a `CALL` statement that passes arguments `BY REFERENCE`.

The example illustrates the following concepts:

- The `CALL` statement does not indicate whether the called program is written in COBOL or C++.
- You must declare a function return value on a `CALL` statement that calls a non-void C++ function.
- The COBOL data types must be mapped to appropriate C++ data types.
- The C++ function must be declared `extern "C"`.

- The COBOL arguments are passed BY REFERENCE. The C++ function receives them by using reference parameters.

COBOL program `driver`:

```
cbl pgmname(mixed)
Identification Division.
Program-Id. "driver".
Data division.
Working-storage section.
01 A pic 9(8) binary value 11111.
01 B pic 9(8) binary value 22222.
01 R pic 9(8) binary.
Procedure Division.
    Display "Hello World, from COBOL!"
    Call "sub" using by reference A B
      returning R
    Display R
    Stop Run.
```

C++ function sub:

```
#include <iostream.h>
extern "C" long sub(long& A, long& B) {
  cout << "Hello from C++" << endl;
  return A + B;
}
```

Output:

```
Hello World, from COBOL!
Hello from C++
00033333
```

Related tasks
"Passing data between COBOL and C/C++" on page 388

Related references
CALL statement (*COBOL for Linux on x86 Language Reference*)

Making recursive calls

A called program can directly or indirectly execute its caller. For example, program X calls program Y, program Y calls program Z, and program Z then calls program X. This type of call is *recursive*.

To make a recursive call, you must code the RECURSIVE clause in the PROGRAM-ID paragraph of the recursively called program. If you try to recursively call a COBOL program that does not have the RECURSIVE clause in the PROGRAM-ID paragraph, the run unit will end abnormally.

Related tasks
"Identifying a program as recursive" on page 4

Related references
PROGRAM-ID paragraph (*COBOL for Linux on x86 Language Reference*)

Passing return codes

You can use the RETURN-CODE special register to pass information between separately compiled programs.

You can set RETURN-CODE in a called program before returning to the caller, and then test this returned value in the calling program. This technique is typically used to indicate the level of success of the called

program. For example, a RETURN-CODE of zero can be used to indicate that the called program executed successfully.

Normal termination: When a main program ends normally, the value of RETURN-CODE is passed to the operating system as a user return code. However, Linux restricts user return code values to 0 through 255. Therefore, if for example RETURN-CODE contains 258 when the program ends, Linux wraps the value within the supported range, resulting in a user return code of 2.

Unrecoverable exception: When a program encounters an unrecoverable exception, the user return code is set to 128 plus the signal number. For a nonthreaded program, the run unit is terminated; for a threaded program, the thread in which the program is executing, not the run unit, is terminated.

Related tasks
"Passing return-code information" on page 403

Related references
RETURN-CODE (*COBOL for Linux on x86 Language Reference*)

Chapter 23. Sharing data

If a run unit consists of several separately compiled programs that call each other, the programs must be able to communicate with each other. They also usually need access to common data.

This information describes how you can write programs that share data with other programs. In this information, a *subprogram* is any program that is called by another program.

Related tasks
"Using data from another program" on page 12

"Passing data" on page 395
"Coding the LINKAGE SECTION" on page 398
"Coding the PROCEDURE DIVISION for passing arguments" on page 399
"Using procedure and function pointers" on page 402
"Passing return-code information" on page 403
"Sharing data by using the EXTERNAL clause" on page 404
"Sharing files between programs (external files)" on page 404
"Using command-line arguments" on page 407

Passing data

You can choose among three ways of passing data between programs: `BY REFERENCE`, `BY CONTENT`, or `BY VALUE`.

`BY REFERENCE`
The subprogram refers to and processes the data items in the storage of the calling program rather than working on a copy of the data. `BY REFERENCE` is the assumed passing mechanism for a parameter if none of the three ways is specified or implied for the parameter.

`BY CONTENT`
The calling program passes only the contents of the *literal* or *identifier*. The called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the data item in which it received the *literal* or *identifier*.

`BY VALUE`
The calling program or method passes the value of the *literal* or *identifier*, not a reference to the sending data item. The called program or invoked method can change the parameter. However, because the subprogram or method has access only to a temporary copy of the sending data item, any change does not affect the argument in the calling program.

Determine which of these data-passing methods to use based on what you want your program to do with the data.

Table 46. ***Methods for passing data in the CALL statement***

Code	Purpose	Comments
`CALL . . . BY REFERENCE` *identifier*	To have the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called program share the same memory	Any changes made by the subprogram to the parameter affect the argument in the calling program.

© Copyright IBM Corp. 2021

Table 46. ***Methods for passing data in the CALL statement*** *(continued)*

Code	Purpose	Comments
`CALL . . . BY REFERENCE ADDRESS OF` *identifier*	To pass the address of *identifier* to a called program, where *identifier* is an item in the `LINKAGE SECTION`	Any changes made by the subprogram to the address affect the address in the calling program.
`CALL . . . BY CONTENT ADDRESS OF` *identifier*	To pass a copy of the address of *identifier* to a called program	Any changes to the copy of the address will not affect the address of *identifier*, but changes to *identifier* using the copy of the address will cause changes to *identifier*.
`CALL . . . BY CONTENT` *identifier*	To pass a copy of the identifier to the subprogram	Changes to the parameter by the subprogram will not affect the caller's identifier.
`CALL . . . BY CONTENT` *literal*	To pass a copy of a literal value to a called program	
`CALL . . . BY CONTENT LENGTH OF` *identifier*	To pass a copy of the length of a data item	The calling program passes the length of the *identifier* from its `LENGTH` special register.

A combination of `BY REFERENCE` and `BY CONTENT` such as:
`CALL 'ERRPROC'`
`  USING BY REFERENCE A`
`  BY CONTENT LENGTH OF A.`
	To pass both a data item and a copy of its length to a subprogram	

`CALL . . . BY VALUE` *identifier*	To pass data to a program, such as a C/C++ program, that uses `BY VALUE` parameter linkage conventions	A copy of the identifier is passed directly in the parameter list.
`CALL . . . BY VALUE` *literal*	To pass data to a program, such as a C/C++ program, that uses `BY VALUE` parameter linkage conventions	A copy of the literal is passed directly in the parameter list.
`CALL . . . BY VALUE ADDRESS OF` *identifier*	To pass the address of *identifier* to a called program. This is the recommended way to pass data to a C/C++ program that expects a pointer to the data.	Any changes to the copy of the address will not affect the address of *identifier*, but changes to *identifier* using the copy of the address will cause changes to *identifier*.
`CALL . . . RETURNING`	To call a C/C++ function with a function return value	

Related tasks

“Describing arguments in the calling program” on page 397
“Describing parameters in the called program” on page 397
“Testing for OMITTED arguments” on page 397
“Specifying CALL . . . RETURNING” on page 403
“Sharing data by using the EXTERNAL clause” on page 404
“Sharing files between programs

(external files)" on page 404

Related references
CALL statement (*COBOL for Linux on x86 Language Reference*)
The USING phrase (*COBOL for Linux on x86 Language Reference*)

Describing arguments in the calling program

In the calling program, describe arguments in the DATA DIVISION in the same manner as other data items in the DATA DIVISION.

Storage for arguments is allocated only in the outermost program. For example, program A calls program B, which calls program C. Data items are allocated in program A. They are described in the LINKAGE SECTION of programs B and C, making the one set of data available to all three programs.

If you reference data in a file, the file must be open when the data is referenced.

Code the USING phrase of the CALL statement to pass the arguments. If you pass a data item BY VALUE, it must be an elementary item.

Related tasks
"Coding the LINKAGE SECTION" on page 398
"Coding the PROCEDURE DIVISION
for passing arguments" on page 399

Related references
The USING phrase (*COBOL for Linux on x86 Language Reference*)

Describing parameters in the called program

You must know what data is being passed from the calling program and describe it in the LINKAGE SECTION of each program that is called directly or indirectly by the calling program.

Code the USING phrase after the PROCEDURE DIVISION header to name the parameters that receive the data that is passed from the calling program.

When arguments are passed to the subprogram BY REFERENCE, it is invalid for the subprogram to specify any relationship between its parameters and any fields other than those that are passed and defined in the main program. The subprogram must not:

- Define a parameter to be larger in total number of bytes than the corresponding argument.
- Use subscript references to refer to elements beyond the limits of tables that are passed as arguments by the calling program.
- Use reference modification to access data beyond the length of defined parameters.
- Manipulate the address of a parameter in order to access other data items that are defined in the calling program.

If any of the rules above are violated, unexpected results might occur.

Related tasks
"Coding the LINKAGE SECTION" on page 398

Related references
The USING phrase (*COBOL for Linux on x86 Language Reference*)

Testing for OMITTED arguments

You can specify that one or more BY REFERENCE arguments are not to be passed to a called program by coding the OMITTED keyword in place of those arguments in the CALL statement.

For example, to omit the second argument when calling program sub1, code this statement:

```
Call 'sub1' Using PARM1, OMITTED, PARM3
```

The arguments in the USING phrase of the CALL statement must match the parameters of the called program in number and position.

In a called program, you can test whether an argument was passed as OMITTED by comparing the address of the corresponding parameter to NULL. For example:

```
Program-ID. sub1.
. . .
Procedure Division Using RPARM1, RPARM2, RPARM3.
    If Address Of RPARM2 = Null Then
        Display 'No 2nd argument was passed this time'
    Else
        Perform Process-Parm-2
    End-If
```

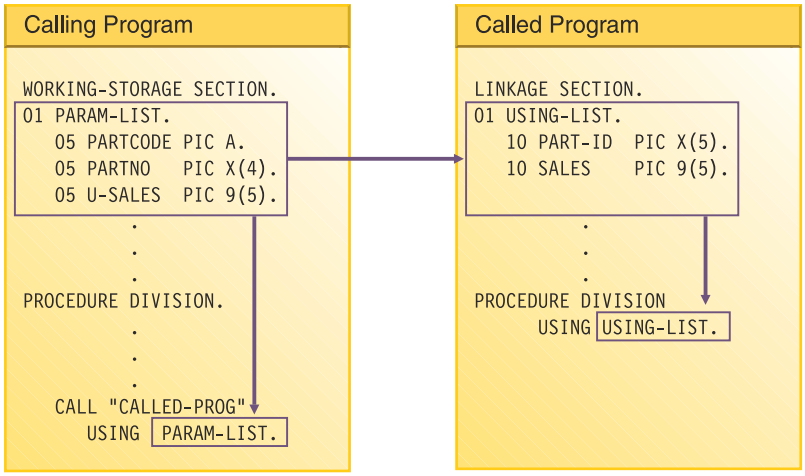
Related references
CALL statement (*COBOL for Linux on x86 Language Reference*)
The USING phrase (*COBOL for Linux on x86 Language Reference*)

Coding the LINKAGE SECTION

Code the same number of data-names in the identifier list of the called program as the number of arguments in the calling program. Synchronize by position, because the compiler passes the first argument from the calling program to the first identifier of the called program, and so on.

You will introduce errors if the number of data-names in the identifier list of a called program is greater than the number of arguments passed from the calling program. The compiler does not try to match arguments and parameters.

The following figure shows a data item being passed from one program to another (implicitly BY REFERENCE):



In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are distinct data items. In the called program, by contrast, the code for parts and the part number are combined into one data item (PART-ID). In the called program, a reference to PART-ID is the only valid reference to these items.

Coding the PROCEDURE DIVISION for passing arguments

If you pass an argument BY VALUE, code the USING BY VALUE clause in the PROCEDURE DIVISION header of the subprogram. If you pass an argument BY REFERENCE or BY CONTENT, you do not need to indicate in the header how the argument was passed.

```
PROCEDURE DIVISION USING BY VALUE. . .
PROCEDURE DIVISION USING. . .
PROCEDURE DIVISION USING BY REFERENCE. . .
```

The first header above indicates that the data items are passed BY VALUE; the second or third headers indicate that the items are passed BY REFERENCE or BY CONTENT.

Related references
The procedure division header (*COBOL for Linux on x86 Language Reference*)
The USING phrase (*COBOL for Linux on x86 Language Reference*)
CALL statement (*COBOL for Linux on x86 Language Reference*)

Grouping data to be passed

Consider grouping all the data items that you need to pass between programs and putting them under one level-01 item. If you do so, you can pass a single level-01 record.

Note that if you pass a data item BY VALUE, it must be an elementary item.

To lessen the possibility of mismatched records, put the level-01 record into a copy library and copy it into both programs. That is, copy it in the WORKING-STORAGE SECTION of the calling program and in the LINKAGE SECTION of the called program.

Related tasks
"Coding the LINKAGE SECTION" on page 398

Related references
CALL statement (*COBOL for Linux on x86 Language Reference*)

Handling null-terminated strings

COBOL supports null-terminated strings when you use string-handling statements together with null-terminated literals and the hexadecimal literal X'00'.

You can manipulate null-terminated strings (passed from a C program, for example) by using string-handling mechanisms such as those in the following code:

```
01 L        pic X(20) value z'ab'.
01 M        pic X(20) value z'cd'.
01 N        pic X(20).
01 N-Length pic 99    value zero.
01 Y        pic X(13) value 'Hello, World!'.
```

To determine the length of a null-terminated string, and display the value of the string and its length, code:

```
Inspect N tallying N-length for characters before initial X'00'
Display 'N: ' N(1:N-length) ' Length: ' N-length
```

To move a null-terminated string to an alphanumeric string, but delete the null, code:

```
Unstring N  delimited by X'00' into X
```

To create a null-terminated string, code:

```
String Y      delimited by size
       X'00'  delimited by size
       into N.
```

To concatenate two null-terminated strings, code:

```
String L      delimited by x'00'
       M      delimited by x'00'
       X'00'  delimited by size
       into N.
```

Related tasks
“Manipulating null-terminated strings” on page 98

Related references
Null-terminated alphanumeric literals
(*COBOL for Linux on x86 Language Reference*)

Using pointers to process a chained list

When you need to pass and receive addresses of record areas, you can use pointer data items, which are either data items that are defined with the USAGE IS POINTER clause or are ADDRESS OF special registers.

A typical application for using pointer data items is in processing a *chained list*, a series of records in which each record points to the next.

When you pass addresses between programs in a chained list, you can use NULL to assign the value of an address that is not valid (nonnumeric 0) to a pointer item in either of two ways:

- Use a VALUE IS NULL clause in its data definition.
- Use NULL as the sending field in a SET statement.

In the case of a chained list in which the pointer data item in the last record contains a null value, you can use this code to check for the end of the list:

```
IF PTR-NEXT-REC = NULL
. . .
   (logic for end of chain)
```

If the program has not reached the end of the list, the program can process the record and move on to the next record.

The data passed from a calling program might contain header information that you want to ignore. Because pointer data items are not numeric, you cannot directly perform arithmetic on them. However, to bypass header information, you can use the SET statement to increment the passed address.

“Example: using pointers to process a chained list” on page 401

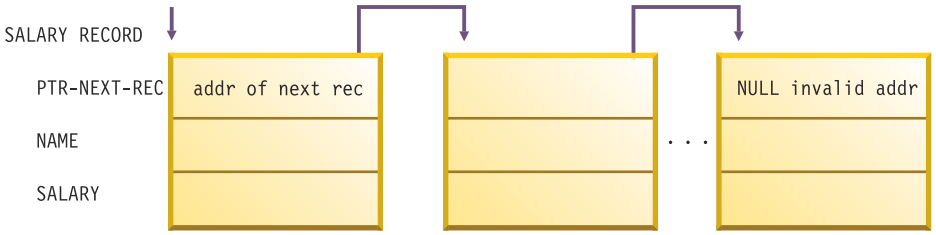
Related tasks
“Coding the LINKAGE SECTION” on page 398
“Coding the PROCEDURE DIVISION for passing arguments” on page 399

Related references
“ADDR” on page 249
SET statement (*COBOL for Linux on x86 Language Reference*)

Example: using pointers to process a chained list

The following example shows how you might process a linked list, that is, a chained list of data items.

For this example, picture a chained list of data that consists of individual salary records. The following figure shows one way to visualize how the records are linked in storage. The first item in each record except the last points to the next record. The first item in the last record contains a null value (instead of a valid address) to indicate that it is the last record.



The high-level pseudocode for an application that processes these records might be:

```
Obtain address of first record in chained list from routine
Check for end of the list
Do until end of the list
   Process record
   Traverse to the next record
End
```

The following code contains an outline of the calling program, LISTS, used in this example of processing a chained list.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. LISTS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
******
 WORKING-STORAGE SECTION.
 77  PTR-FIRST         POINTER  VALUE IS NULL.                  (1)
 77  DEPT-TOTAL        PIC 9(4) VALUE IS 0.
******
 LINKAGE SECTION.
 01  SALARY-REC.
     02  PTR-NEXT-REC    POINTER.                               (2)
     02  NAME            PIC X(20).
     02  DEPT            PIC 9(4).
     02  SALARY          PIC 9(6).
 01  DEPT-X            PIC 9(4).
******
 PROCEDURE DIVISION USING DEPT-X.
******
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND ACCUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
******
     CALL "CHAIN-ANCH" USING PTR-FIRST                          (3)
     SET ADDRESS OF SALARY-REC TO PTR-FIRST                     (4)
******
     PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL (5)

      IF DEPT = DEPT-X
        THEN ADD SALARY TO DEPT-TOTAL
        ELSE CONTINUE
      END-IF
      SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC                 (6)

     END-PERFORM
******
     DISPLAY DEPT-TOTAL
     GOBACK.
```

(1)
PTR-FIRST is defined as a pointer data item with an initial value of NULL. On a successful return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list. If something goes wrong with the call, and PTR-FIRST never receives the value of the address of the first record in the chain, a null value remains in PTR-FIRST and, according to the logic of the program, the records will not be processed.

(2)
The LINKAGE SECTION of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed in the USING clause of the CALL statement.

(3)
To obtain the address of the first SALARY-REC record area, the LISTS program calls the program CHAIN-ANCH.

(4)
The SET statement bases the record description SALARY-REC on the address contained in PTR-FIRST.

(5)
The chained list in this example is set up so that the last record contains an address that is not valid. This check for the end of the chained list is accomplished with a do-while structure where the value NULL is assigned to the pointer data item in the last record.

(6)
The address of the record in the LINKAGE-SECTION is set equal to the address of the next record by means of the pointer data item sent as the first field in SALARY-REC. The record-processing routine repeats, processing the next record in the chained list.

To increment addresses received from another program, you could set up the LINKAGE SECTION and PROCEDURE DIVISION like this:

```
LINKAGE SECTION.
01  RECORD-A.
    02  HEADER          PIC X(12).
    02  REAL-SALARY-REC PIC X(30).
. . .
01  SALARY-REC.
    02  PTR-NEXT-REC    POINTER.
    02  NAME            PIC X(20).
    02  DEPT            PIC 9(4).
    02  SALARY          PIC 9(6).
. . .
PROCEDURE DIVISION USING DEPT-X.
. . .
    SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

The address of SALARY-REC is now based on the address of REAL-SALARY-REC, or RECORD-A + 12.

Related tasks
“Using pointers to process a chained list” on page 400

Using procedure and function pointers

Procedure pointers are data items defined with the USAGE IS PROCEDURE-POINTER clause. *Function pointers* are data items defined with the USAGE IS FUNCTION-POINTER clause.

In this information, "pointer" refers to either a procedure-pointer data item or a function-pointer data item. You can set either of these data items to contain entry addresses of, or pointers to, the following entry points:

- Another COBOL program that is not nested.

- A program written in another language. For example, to receive the entry address of a C function, call the function with the CALL RETURNING format of the CALL statement. It returns a pointer that you can convert to a procedure pointer by using a form of the SET statement.
- An alternate entry point in another COBOL program (as defined in an ENTRY statement).

You can set a pointer data item only by using the SET statement. For example:

```
CALL 'MyCFunc' RETURNING ptr.
SET  proc-ptr  TO ptr.
CALL proc-ptr  USING dataname.
```

Suppose that you set a pointer item to an entry address in a load module that is called by a CALL *identifier* statement, and your program later cancels that called module. The pointer item becomes undefined, and reference to it thereafter is not reliable.

Related references
"ADDR" on page 249

PROCEDURE-POINTER phrase (*COBOL for Linux on x86 Language Reference*)
SET statement (*COBOL for Linux on x86 Language Reference*)

Passing return-code information

Use the RETURN-CODE special register to pass return codes between programs.

Related tasks
"Passing return codes" on page 393

Using the RETURN-CODE special register

When a COBOL program returns to its caller, the contents of the RETURN-CODE special register are set according to the value of the RETURN-CODE special register in the called program.

Setting of the RETURN-CODE by the called program is limited to calls between COBOL programs. Therefore, if your COBOL program calls a C program, you cannot expect the RETURN-CODE special register of the COBOL program to be set.

For equivalent function between COBOL and C programs, have your COBOL program call the C program with the RETURNING phrase. If the C program (function) correctly declares a function value, the RETURNING value of the calling COBOL program will be set.

Using PROCEDURE DIVISION RETURNING . . .

Use the RETURNING phrase in the PROCEDURE DIVISION header of a program to return information to the calling program.

```
PROCEDURE DIVISION RETURNING dataname2
```

When the called program in the example above successfully returns to its caller, the value in *dataname2* is stored into the identifier that was specified in the RETURNING phrase of the CALL statement:

```
CALL . . . RETURNING dataname2
```

Specifying CALL . . . RETURNING

You can specify the RETURNING phrase of the CALL statement for calls to C/C++ functions or to COBOL subroutines.

The RETURNING phrase has the following format.

```
CALL . . . RETURNING dataname2
```

The return value of the called program is stored into *dataname2*. You must define *dataname2* in the DATA DIVISION of the calling program. The data type of the return value that is declared in the target function must be identical to the data type of *dataname2*.

Sharing data by using the EXTERNAL clause

Use the EXTERNAL clause to enable separately compiled programs (including programs in a batch sequence) to share data items. Code EXTERNAL in the level-01 data description in the WORKING-STORAGE SECTION.

The following rules apply:

- Items that are subordinate to an EXTERNAL group item are themselves EXTERNAL.
- You cannot use the name of an EXTERNAL data item as the name for another EXTERNAL item in the same program.
- You cannot code the VALUE clause for any elementary item, group item or subordinate item that is EXTERNAL.

In the run unit, any COBOL program that has the same data description for the item as the program that contains the item can access and process that item. For example, suppose program A has the following data description:

```
01 EXT-ITEM1     EXTERNAL     PIC 99.
```

Program B can access that data item if B has the identical data description in its WORKING-STORAGE SECTION.

Any program that has access to an EXTERNAL data item can change the value of that item. Therefore do not use this clause for data items that you need to protect.

Related references

Sharing files between programs (external files)

To enable separately compiled programs in a run unit to access a file as a common file, use the EXTERNAL clause for the file.

It is recommended that you follow these guidelines:

- Use the same data-name in the FILE STATUS clause of all the programs that check the file status code.
- For each program that checks the same file status field, code the EXTERNAL clause in the level-01 data definition for the file status field.

Using an external file has these benefits:

- Even if the main program does not contain any input or output statements, it can reference the record area of the file.
- Each subprogram can control a single input or output function, such as OPEN or READ.
- Each program has access to the file.

“Example: using external files” on page 405

Related tasks
"Using data in input and output operations" on page 9

Related references
EXTERNAL clause (*COBOL for Linux on x86 Language Reference*)

Example: using external files

The following example shows the use of an external file in several programs. COPY statements ensure that each subprogram contains an identical description of the file.

The following table describes the main program and subprograms.

Name	Function
ef1	The main program, which calls all the subprograms and then verifies the contents of a record area
ef1openo	Opens the external file for output and checks the file status code
ef1write	Writes a record to the external file and checks the file status code
ef1openi	Opens the external file for input and checks the file status code
ef1read	Reads a record from the external file and checks the file status code
ef1close	Closes the external file and checks the file status code

Each program uses three copybooks:

- `efselect` is placed in the `FILE-CONTROL` paragraph:

```
Select ef1
Assign To ef1
File Status Is efs1
Organization Is Sequential.
```

- `effile` is placed in the `FILE SECTION`:

```
Fd ef1 Is External
          Record Contains 80 Characters
          Recording Mode F.
01  ef-record-1.
    02 ef-item-1  Pic X(80).
```

- `efwrkstg` is placed in the `WORKING-STORAGE SECTION`:

```
01  efs1          Pic 99 External.
```

Input/output using external files

```
 IDENTIFICATION DIVISION.
 Program-Id.
     ef1.
*
* This main program controls external file processing.
*
 ENVIRONMENT DIVISION.
 Input-Output Section.
 File-Control.
     Copy efselect.
 DATA DIVISION.
 FILE SECTION.
     Copy effile.
 WORKING-STORAGE SECTION.
```

```
     Copy efwrkstg.
 PROCEDURE DIVISION.
     Call "ef1openo"
     Call "ef1write"
     Call "ef1close"
     Call "ef1openi"
     Call "ef1read"
     If ef-record-1 = "First record" Then
       Display "First record correct"
     Else
       Display "First record incorrect"
       Display "Expected: " "First record"
       Display "Found   : " ef-record-1
     End-If
     Call "ef1close"
     Goback.
 End Program ef1.
 IDENTIFICATION DIVISION.
 Program-Id.
     ef1openo.
*
* This program opens the external file for output.
*
 ENVIRONMENT DIVISION.
 Input-Output Section.
 File-Control.
     Copy efselect.
 DATA DIVISION.
 FILE SECTION.
     Copy effile.
 WORKING-STORAGE SECTION.
     Copy efwrkstg.
 PROCEDURE DIVISION.
     Open Output ef1
     If efs1 Not = 0
       Display "file status " efs1 " on open output"
       Stop Run
     End-If
     Goback.
 End Program ef1openo.
 IDENTIFICATION DIVISION.
 Program-Id.
     ef1write.
*
* This program writes a record to the external file.
*
 ENVIRONMENT DIVISION.
 Input-Output Section.
 File-Control.
     Copy efselect.
 DATA DIVISION.
 FILE SECTION.
     Copy effile.
 WORKING-STORAGE SECTION.
     Copy efwrkstg.
 PROCEDURE DIVISION.
     Move "First record" to ef-record-1
     Write ef-record-1
     If efs1 Not = 0
       Display "file status " efs1 " on write"
       Stop Run
     End-If
     Goback.
 End Program ef1write.
 Identification Division.
 Program-Id.
     ef1openi.
*
* This program opens the external file for input.
*
 ENVIRONMENT DIVISION.
 Input-Output Section.
 File-Control.
     Copy efselect.
 DATA DIVISION.
 FILE SECTION.
     Copy effile.
 WORKING-STORAGE SECTION.
     Copy efwrkstg.
 PROCEDURE DIVISION.
     Open Input ef1
     If efs1 Not = 0
```

```
       Display "file status " efs1 " on open input"
       Stop Run
     End-If
     Goback.
 End Program ef1openi.
 Identification Division.
 Program-Id.
     ef1read.
*
* This program reads a record from the external file.
*
 ENVIRONMENT DIVISION.
 Input-Output Section.
 File-Control.
     Copy efselect.
 DATA DIVISION.
 FILE SECTION.
     Copy effile.
 WORKING-STORAGE SECTION.
     Copy efwrkstg.
 PROCEDURE DIVISION.
     Read ef1
     If efs1 Not = 0
       Display "file status " efs1 " on read"
       Stop Run
     End-If
     Goback.
 End Program ef1read.
 Identification Division.
 Program-Id.
     ef1close.
*
* This program closes the external file.
*
 ENVIRONMENT DIVISION.
 Input-Output Section.
 File-Control.
     Copy efselect.
 DATA DIVISION.
 FILE SECTION.
     Copy effile.
 WORKING-STORAGE SECTION.
     Copy efwrkstg.
 PROCEDURE DIVISION.
     Close ef1
     If efs1 Not = 0
       Display "file status " efs1 " on close"
       Stop Run
     End-If
     Goback.
 End Program ef1close.
```

Using command-line arguments

You can pass arguments to a main program on the command line. The operating system calls main programs with null-terminated strings that contain the arguments.

If the arguments that are entered are shorter than the COBOL data-names that receive them, use the technique to isolate them that is shown in the related task below about manipulating null-terminated strings.

How the arguments are treated depends on whether you use the `-host` option of the `cob2` command.

If you do not specify the `-host` option, command-line arguments are passed in native data format, and Linux calls all main programs with the following arguments:

- Number of command-line arguments plus one
- Pointer to the name of the program
- Pointer to the first argument
- Pointer to the second argument
- . . .
- Pointer to the *n*th argument

If you specify the `-host` option, Linux calls all main programs with an EBCDIC character string that has a halfword prefix that contains the string length. You must enter the command-line arguments as a single string enclosed in quotation marks ("). To pass a quotation-mark character in the string, precede the quotation mark with the backslash (\) escape character.

"Example: command-line arguments without -host option" on page 408
"Example: command-line arguments with -host option" on page 409

Related tasks
"Manipulating null-terminated strings" on page 98
"Handling null-terminated strings" on page 399

Related references
"cob2 options" on page 230

Example: command-line arguments without -host option

This example shows how to read command-line arguments if you are not using the `-host` option.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "targlinux".
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 LINKAGE SECTION.
 01  PARM-LEN PIC S9(9) COMP.
 01  OS-PARM.
     02 PARMPTR-TABLE OCCURS 1 TO 100 TIMES DEPENDING ON PARM-LEN.
        03 PARMPTR  POINTER.
 01 PARM-STRING PIC XX.
*
 PROCEDURE DIVISION USING BY VALUE PARM-LEN BY REFERENCE OS-PARM.
     display "parm-len=" parm-len
     SET ADDRESS OF PARM-STRING TO PARMPTR(2).
     display "parm-string= '" PARM-STRING "'";
     EVALUATE PARM-STRING
         when "01"  display "case one"
         when "02"  display "case two"
         when "95"  display "case ninety-five"
         when other display "case unknown"
     END-EVALUATE
     GOBACK.
```

Suppose you compile and run the following program:

```
cob2 targlinux.cbl
a.out 95
```

The result is:

```
parm-len=000000002
parm-string= '95'
case ninety-five
```

Example: command-line arguments with -host option

This example shows how to read the command-line arguments if you are using the `-host` option.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "testarg".
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 linkage section.
 01  os-parm.
     05 parm-len         pic s999 comp.
     05 parm-string.
         10 parm-char    pic x occurs 0 to 100 times
                         depending on parm-len.
*
 PROCEDURE DIVISION using os-parm.
     display "parm-len=" parm-len
     display "parm-string='" parm-string "'"
     evaluate parm-string
       when "01"  display "case one"
       when "02"  display "case two"
       when "95"  display "case ninety-five"
       when other display "case unknown"
     end-evaluate
     GOBACK.
```

Suppose you compile and run the program as follows:

```
cob2 -host testarg.cbl
a.out "95"
```

Then the resulting output is:

```
parm-len=002
parm-string='95'
case ninety-five
```

Chapter 24. Using shared libraries

Shared libraries provide a convenient and efficient means of packaging applications, and are widely used in Linux.

In COBOL, a *shared library* is a collection of one or more outermost programs. Just as you can compile and link several COBOL programs into a single executable file, you can link one or more compiled outermost COBOL programs to create a shared library. You typically use a shared library as a collection of frequently called functions.

Although the outermost programs in a shared library can contain nested programs, programs external to a shared library can call only the outermost programs (known as *entry points*) in the shared library. Each program in a shared library is referred to as a *subprogram*.

You can call COBOL shared libraries or a mixture of COBOL and C/C++ shared libraries.

"Example: creating a sample shared library" on page 412

Related concepts
"Main programs, subprograms, and calls" on page 381
"Static linking versus using shared libraries" on page 411
"How the linker resolves references to shared libraries" on page 412

Static linking versus using shared libraries

Static linking is the linking of a calling program and one or more called programs into a single executable module. When the program is loaded, the operating system places into memory a single file that contains the executable code and data.

The primary advantage of static linking is that you can use it to create self-contained, independent executable modules.

Static linking has these disadvantages, however:

- Because external programs are built into the executable file, the executable file increases in size.
- You cannot change the behavior of the executable file without recompiling and relinking it.
- If more than one calling program needs to access the called programs, duplicate copies of the called programs must be loaded in memory.

To overcome these disadvantages, you can use shared libraries:

- You can build one or more subprograms into a shared library; and several programs can call the subprograms that are in the shared library. Because the shared library code is separate from that of the calling programs, the calling programs can be smaller.
- You can change the subprograms that are in the shared library without having to recompile or relink the calling programs.
- Only a single copy of the shared library needs to be in memory.

Shared libraries typically provide common functions that can be used by a number of programs. For example, you can use shared libraries to implement subprogram packages, subsystems, and interfaces in other programs or to create object-oriented class libraries.

"Example: creating a sample shared library" on page 412

Related tasks
"Calling nonnested COBOL programs" on page 385

© Copyright IBM Corp. 2021

How the linker resolves references to shared libraries

When you compile a program, the compiler generates an object module for the code in the program. If you call any subprograms (*functions* in C/C++, *subroutines* in other languages) that are in an external object module, the compiler adds an external program reference to the target object module.

To resolve an external reference to a shared library, the linker adds information to the executable file that tells the loader where to find the shared library code when the executable file is loaded.

The linker does not resolve all references to shared libraries that are made by COBOL CALL statements. If the DYNAM compiler option is in effect, COBOL resolves CALL *literal* statements when these calls are executed. CALL *identifier* calls are also dynamically resolved.

"Example: creating a sample shared library" on page 412

Related concepts
"Static linking versus using shared libraries" on page 411

Related tasks
"Making dynamic calls to shared libraries under CICS" on page 332

Related references
"Linker input and output files" on page 233
"DYNAM" on page 262

Example: creating a sample shared library

The following example shows three COBOL programs, one of which (alpha) calls the other two (beta and gamma). The procedure after the programs shows how to create a shared library that contains the two called programs, create an archive library that contains that shared library, and compile and link the calling program into a module that accesses the called programs in the archive library.

Example 1: alpha.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. alpha.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 hello1   pic x(30)  value is "message from alpha to beta".
 01 hello2   pic x(30)  value is "message from alpha to gamma".
*
 PROCEDURE DIVISION.
     display "alpha begins"
     call "beta" using hello1
     display "alpha after beta"
     call "gamma" using hello2
     display "alpha after gamma"
     goback.
```

Example 2: beta.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. beta.
*
 ENVIRONMENT DIVISION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 Linkage section.
 01 msg   pic x(30).
*
```

```
 PROCEDURE DIVISION using msg.
     DISPLAY "beta gets msg=" msg.
     goback.
```

Example 3: gamma.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. gamma.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 Linkage section.
 01 msg            pic x(30).
*
PROCEDURE DIVISION using msg.
    DISPLAY "gamma gets msg=" msg.
    goback.
```

Procedure

The simplest way to combine the three programs is to compile and link them into a single executable module by using the following command:

```
cob2 -o m_abg alpha.cbl beta.cbl gamma.cbl
```

You can then run the programs by issuing the command `m_abg`, which results in the following output:

```
alpha begins
beta gets msg=message from alpha to beta
alpha after beta
gamma gets msg=message from alpha to gamma
alpha after gamma
```

Instead of linking the programs into a single executable module, you can instead put `beta` and `gamma` in a shared library (called `sh_bg` in the following procedure), and compile and link `alpha` into an executable module that accesses `beta` and `gamma` in the shared library. To do so, do these steps:

1. Create an version script that specifies the symbols that the shared library must export:

   ```
   {
   global:
        GAMMA;
        BETA;
   local:
        *;
   };
   ```

   The symbol names in export file `bg.version` shown above are uppercase because the COBOL default is to use uppercase names for external symbols. If you need mixed-case names, use the `PGMNAME(MIXED)` compiler option.

   If you name the export file `bg.version`, you must use option `-Wl,--version-script,bg.version` when you create the shared library (as shown in the next step).
2. Use the following command to combine `beta` and `gamma` into a shared library object called `sh_bg`:

   ```
   cob2 -o sh_bg.so beta.cbl gamma.cbl -Wl,--version-script,bg.version
   ```

   This command provides the following information to the compiler and linker:

   - `-o sh_bg beta.cbl gamma.cbl` compiles and links `beta.cbl` and `gamma.cbl`, and names the resulting output module `sh_bg.so`.

- `-Wl,--version-script,bg.version` tells the linker to export the symbols that are named in export file `bg.version`.

3. Issue the following commands to recompile `alpha.cbl` and produce executable `m_alpha` that has external references resolved to `sh_bg`:

```
cob2 -o m_alpha alpha.cbl sh_bg.so
```

You can then run the program by issuing the command `m_alpha`, which produces the same output as that shown before the steps above.

Note that `sh_bg.so` must be in your LD_LIBRARY_PATH environment variable when executing `m_alpha`. For example you can add the current directory to your LD_LIBRARY_PATH by issuing the command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

As an alternative to issuing the commands in the last two steps, you can create a makefile that includes the commands.

"Example: creating a makefile for the sample shared library" on page 414

Related tasks
"Passing options to the linker" on page 232

Related references
"cob2 options" on page 230
"Linker input and output files" on page 233
"PGMNAME" on page 274
AIX General Programming Concepts: Writing and Debugging Programs
AIX Commands Reference, Volume 1 (ar command)

Example: creating a makefile for the sample shared library

The following example shows how you can create a makefile for the shared object, `sh_bg.so`, that contains two called programs, `beta` and `gamma`.

The makefile assumes that the version script `bg.version` defines the symbols that the shared library exports.

(The creation of the shared library is shown in "Example: creating a sample shared library" on page 412.)

```
#
#
all:     m_abg libbg.a m_alpha

#  Create m_abg containing alpha, beta, and gamma

m_abg:   alpha.cbl beta.cbl gamma.cbl
    cob2 -o m_abg alpha.cbl beta.cbl gamma.cbl

#  Create sh_bg.so containing beta and gamma
#  sh_bg.so is a shared object that exports the symbols defined in bg.version
#    contains both beta and gamma

sh_bg.so: beta.cbl gamma.cbl bg.version
    rm -f sh_bg.so
    cob2 -o sh_bg.so beta.cbl gamma.cbl -Wl,--version-script,bg.version

#  Create m_alpha containing alpha and using shared object sh_bg.so
m_alpha: alpha.cbl
    cob2 -o m_alpha alpha.cbl sh_bg.so

clean:
    rm -f m_abg m_alpha sh_bg.so *.lst
```

Executing either the command `m_abg` or the command `m_alpha` provides the same output.

Chapter 25. Preinitializing the COBOL runtime environment

Using *preinitialization*, an application can initialize the COBOL runtime environment once, perform multiple executions using that environment, and then explicitly terminate the environment.

You can use preinitialization to invoke COBOL programs multiple times from a non-COBOL environment such as C/C++.

Preinitialization has two primary benefits:

- The COBOL environment stays ready for program calls.

  Because a COBOL run unit is not terminated on return from the first COBOL program in the run unit, the COBOL programs that are invoked from a non-COBOL environment can be invoked in their last-used state.
- Performance is better.

  Repeatedly creating and taking down the COBOL runtime environment involves overhead and can slow down an application.

Use preinitialization services for multilanguage applications in which non-COBOL programs need to use a COBOL program in its last-used state. For example, a file might be opened on the first call to a COBOL program, and the invoking program expects subsequent calls to the program to find the file open.

Restriction: Preinitialization is not supported under CICS.

Use the interfaces described in the Related tasks to initialize and terminate a persistent COBOL runtime environment. Any shared library that contains a COBOL program used in a preinitialized environment cannot be deleted until the preinitialized environment is terminated.

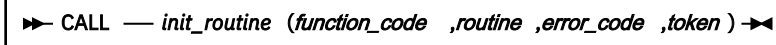
"Example: preinitializing the COBOL environment" on page 417

Related tasks
"Initializing persistent COBOL environment" on page 415
"Terminating preinitialized COBOL environment" on page 416

Initializing persistent COBOL environment

Use the following interface to initialize a persistent COBOL environment.

CALL init_routine syntax



CALL
Invocation of *init_routine*, using language elements appropriate to the language from which the call is made

init_routine
The name of the initialization routine: `_iwzCOBOLInit` or `IWZCOBOLINIT`

***function_code* (input)**
A 4-byte binary number, passed by value. *function_code* can be:

1
The first COBOL program invoked after this function invocation is treated as a subprogram.

© Copyright IBM Corp. 2021

***routine* (input)**
Address of the routine to be invoked if the run unit terminates. The token argument passed to this function is passed to the run-unit termination exit routine. This routine, when invoked upon run-unit termination, must not return to the invoker of the routine but instead use longjmp() or exit().

If you do not provide an exit routine address, an *error_code* is generated that indicates that preinitialization failed.

***error_code* (output)**
A 4-byte binary number. *error_code* can be:

0
Preinitialization was successful.

1
Preinitialization failed.

***token* (input)**
A 4-byte token to be passed to the exit *routine* specified above when that routine is invoked upon run-unit termination.

Related tasks
"Terminating preinitialized COBOL environment" on page 416

Terminating preinitialized COBOL environment

Use the following interface to terminate the preinitialized persistent COBOL environment.

CALL term_routine syntax

CALL
Invocation of *term_routine*, using language elements appropriate to the language from which the call is made

term_routine
The name of the termination routine: `_iwzCOBOLTerm` or `IWZCOBOLTERM`

***function_code* (input)**
A 4-byte binary number, passed by value. *function_code* can be:

1
Clean up the preinitialized COBOL runtime environment as if a COBOL `STOP RUN` statement were performed; for example, all COBOL files are closed. However, the control returns to the caller of this service.

***error_code* (output)**
A 4-byte binary number. *error_code* can be:

0
Termination was successful.

1
Termination failed.

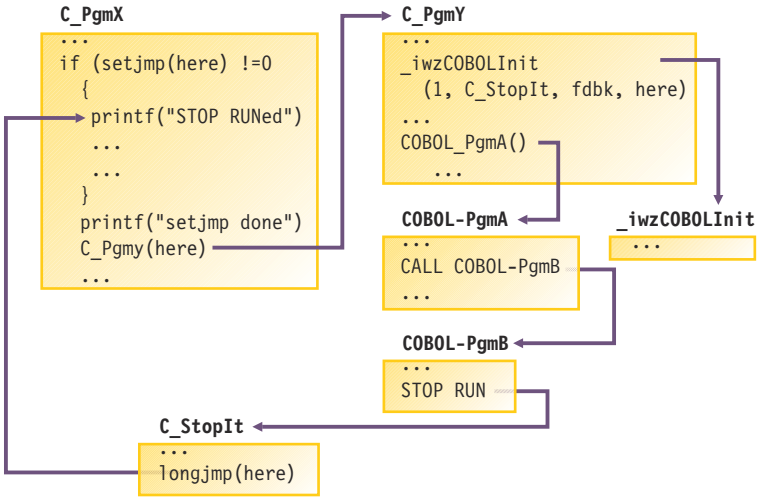
The first COBOL program called after the invocation of the preinitialization routine is treated as a subprogram. Thus a `GOBACK` from this (initial) program does not trigger run-unit termination semantics such as the closing of files. Run-unit termination (such as with `STOP RUN`) does free the preinitialized COBOL environment before the invocation of the run-unit exit routine.

COBOL environment not active: If your program invokes the termination routine and the COBOL environment is not already active, the invocation has no effect on execution, and control is returned to the invoker with an error code of 0.

“Example: preinitializing the COBOL environment” on page 417

Example: preinitializing the COBOL environment

The following figure illustrates how the preinitialized COBOL environment works. The example shows a C program initializing the COBOL environment, calling COBOL programs, then terminating the COBOL environment.



The following example shows the use of COBOL preinitialization. A C main program calls the COBOL program XIO several times. The first call to XIO opens the file, the second call writes one record, and so on. The final call closes the file. The C program then uses C-stream I/O to open and read the file.

To test and run the program, enter the following commands from a command shell:

```
xlc -c testinit.c
cob2 testinit.o xio.cbl
a.out
```

The result is:

```
_iwzCOBOLinit got 0
xio entered with x=0000000000
xio entered with x=0000000001
xio entered with x=0000000002
xio entered with x=0000000003
xio entered with x=0000000004
xio entered with x=0000000099
StopArg=0
_iwzCOBOLTerm expects rc=0 and got rc=0
FILE1 contains ----
11111
22222
33333
---- end of FILE1
```

Note that in this example, the run unit was not terminated by a COBOL STOP RUN; it was terminated when the main program called `_iwzCOBOLTerm`.

The following C program is in the file `testinit.c`:

```
#ifdef _Linux
typedef int (*PFN)();
#define LINKAGE
#else
#include <windows.h>
#define LINKAGE _System
#endif
```

```
#include   <stdio.h>
#include   <setjmp.h>

extern void _iwzCOBOLInit(int fcode, PFN StopFun, int *err_code, void *StopArg);
extern void _iwzCOBOLTerm(int fcode, int *err_code);
extern void LINKAGE XIO(long *k);

jmp_buf Jmpbuf;
long StopArg = 0;

int LINKAGE
StopFun(long *stoparg)
{
        printf("inside StopFun\n");
        *stoparg = 123;
        longjmp(Jmpbuf,1);
}

main()
{
        int rc;
        long k;
        FILE *s;
        int c;

        if (setjmp(Jmpbuf) ==0) {
                _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
                printf( "_iwzCOBOLinit got %d\n",rc);
                for (k=0; k <= 4; k++) XIO(&k);
                k = 99; XIO(&k);
        }
        else printf("return after STOP RUN\n");
        printf("StopArg=%d\n", StopArg);
        _iwzCOBOLTerm(1, &rc);
        printf("_iwzCOBOLTerm expects rc=0 and got rc=%d\n",rc);
        printf("FILE1 contains ---- \n");
        s = fopen("FILE1", "r");
        if (s) {
                while (  (c = fgetc(s) ) != EOF ) putchar(c);
        }
        printf("---- end of FILE1\n");
}
```

The following COBOL program is in the file `xio.cbl`:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.     xio.
*****************************************************************
 ENVIRONMENT    DIVISION.
 CONFIGURATION   SECTION.
 INPUT-OUTPUT    SECTION.
 FILE-CONTROL.
     SELECT file1 ASSIGN TO FILE1
       ORGANIZATION IS LINE SEQUENTIAL
       FILE STATUS IS file1-status.
 . . .
 DATA           DIVISION.
 FILE SECTION.
 FD FILE1.
 01 file1-id pic x(5).
 . . .
 WORKING-STORAGE SECTION.
 01 file1-status  pic xx    value is zero.
 . . .
 LINKAGE SECTION.
*
 01 x               PIC S9(8) COMP-5.
 . . .
 PROCEDURE DIVISION using x.
 . . .
     display "xio entered with x=" x
     if x = 0 then
       OPEN output FILE1
     end-if
     if x = 1  then
        MOVE ALL "1" to file1-id
        WRITE file1-id
     end-if
```

```
      if x = 2 then
         MOVE ALL "2" to file1-id
         WRITE file1-id
      end-if
      if x = 3 then
         MOVE ALL "3" to file1-id
         WRITE file1-id
      end-if
      if x = 99 then
        CLOSE file1
      end-if
      GOBACK.
```

Chapter 26. Processing two-digit-year dates

With the millennium language extensions (MLE), you can make simple changes in your COBOL programs to define date fields. The compiler recognizes and acts on these dates by using a century window to ensure consistency.

Use the following steps to implement automatic date recognition in a COBOL program:

1. Add the `DATE FORMAT` clause to the data description entries of the data items in the program that contain dates. You must identify all dates with `DATE FORMAT` clauses, even those that are not used in comparisons.
2. To expand dates, use `MOVE` or `COMPUTE` statements to copy the contents of windowed date fields to expanded date fields.
3. If necessary, use the `DATEVAL` and `UNDATE` intrinsic functions to convert between date fields and nondates.
4. Use the `YEARWINDOW` compiler option to set the century window as either a fixed window or a sliding window.
5. Compile the program with the `DATEPROC(FLAG)` compiler option, and review the diagnostic messages to see if date processing has produced any unexpected side effects.
6. When the compilation has only information-level diagnostic messages, you can recompile the program with the `DATEPROC(NOFLAG)` compiler option to produce a clean listing.

You can use certain programming techniques to take advantage of date processing and control the effects of using date fields such as when comparing dates, sorting and merging by date, and performing arithmetic operations involving dates. The millennium language extensions support year-first, year-only, and year-last date fields for the most common operations on date fields: comparisons, moving and storing, and incrementing and decrementing.

Related concepts
“Millennium language extensions (MLE)” on page 422

Related tasks
“Resolving date-related logic problems” on page 423
“Using year-first, year-only, and year-last date fields” on page 428
“Manipulating literals as dates” on page 430
“Performing arithmetic on date fields” on page 434
“Controlling date processing explicitly” on page 435
“Analyzing and avoiding date-related diagnostic messages” on page 437
“Avoiding problems in processing dates” on page 439

Related references
“DATEPROC” on page 259
“YEARWINDOW” on page 288
DATE FORMAT clause (*COBOL for Linux on x86 Language Reference*)

© Copyright IBM Corp. 2021

Millennium language extensions (MLE)

The term *millennium language extensions* (MLE) refers to the features of COBOL for Linux that the DATEPROC compiler option activates to help with logic problems that involve dates in the year 2000 and beyond.

When enabled, the extensions include:

- The DATE FORMAT clause. To identify date fields and to specify the location of the year component within the date, add this clause to items in the DATA DIVISION.

  There are several restrictions on use of the DATE FORMAT clause; for example, you cannot specify it for items that have USAGE NATIONAL. For details, see the related references below.
- The reinterpretation as a date field of the function return value for the following intrinsic functions:
  - DATE-OF-INTEGER
  - DATE-TO-YYYYMMDD
  - DAY-OF-INTEGER
  - DAY-TO-YYYYDDD
  - YEAR-TO-YYYY
- The reinterpretation as a date field of the conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD in the following forms of the ACCEPT statement:
  - ACCEPT *identifier* FROM DATE
  - ACCEPT *identifier* FROM DATE YYYYMMDD
  - ACCEPT *identifier* FROM DAY
  - ACCEPT *identifier* FROM DAY YYYYDDD
- The intrinsic functions UNDATE and DATEVAL, used for selective reinterpretation of date fields and nondates.
- The intrinsic function YEARWINDOW, which retrieves the starting year of the century window set by the YEARWINDOW compiler option.

The DATEPROC compiler option enables special date-oriented processing of identified date fields. The YEARWINDOW compiler option specifies the 100-year window (the century window) to use for interpreting two-digit windowed years.

Related concepts
"Principles and objectives of these extensions" on page 422

Related references
"DATEPROC" on page 259
"YEARWINDOW" on page 288
Restrictions on using date fields (*COBOL for Linux on x86 Language Reference*)

Principles and objectives of these extensions

To gain the most benefit from the millennium language extensions, you need to understand the reasons for their introduction into the COBOL language.

The millennium language extensions focus on a few key principles:

- Programs to be recompiled with date semantics are fully tested and valuable assets of the enterprise. Their only relevant limitation is that two-digit years in the programs are restricted to the range 1900-1999.
- No special processing is done for the nonyear part of dates. That is why the nonyear part of the supported date formats is denoted by Xs. To do otherwise might change the meaning of existing programs. The only date-sensitive semantics that are provided involve automatically expanding (and contracting) the two-digit year part of dates with respect to the century window for the program.

- Dates with four-digit year parts are generally of interest only when used in combination with windowed dates. Otherwise there is little difference between four-digit year dates and nondates.

Based on these principles, the millennium language extensions are designed to meet several objectives. You should evaluate the objectives that you need to meet in order to resolve your date-processing problems, and compare them with the objectives of the millennium language extensions, to determine how your application can benefit from them. You should not consider using the extensions in new applications or in enhancements to existing applications, unless the applications are using old data that cannot be expanded until later.

The objectives of the millennium language extensions are as follows:

- Extend the useful life of your application programs as they are currently specified.
- Keep source changes to a minimum, preferably limited to augmenting the declarations of date fields in the `DATA DIVISION`. To implement the century window solution, you should not need to change the program logic in the `PROCEDURE DIVISION`.
- Preserve the existing semantics of the programs when adding date fields. For example, when a date is expressed as a literal, as in the following statement, the literal is considered to be compatible (windowed or expanded) with the date field to which it is compared:

  ```
  If Expiry-Date Greater Than 980101 . . .
  ```

  Because the existing program assumes that two-digit-year dates expressed as literals are in the range 1900-1999, the extensions do not change this assumption.
- The windowing feature is not intended for long-term use. It can extend the useful life of applications as a start toward a long-term solution that can be implemented later.
- The expanded date field feature is intended for long-term use, as an aid for expanding date fields in files and databases.

The extensions do not provide fully specified or complete date-oriented data types, with semantics that recognize, for example, the month and day parts of Gregorian dates. They do, however, provide special semantics for the year part of dates.

Resolving date-related logic problems

You can adopt any of three approaches to assist with date-processing problems: use a century window, internal bridging, or full field expansion.

Century window
: You define a century window and specify the fields that contain windowed dates. The compiler then interprets the two-digit years in these data fields according to the century window.

Internal bridging
: If your files and databases have not yet been converted to four-digit-year dates, but you prefer to use four-digit expanded-year logic in your programs, you can use an internal bridging technique to process the dates as four-digit-year dates.

Full field expansion
: This solution involves explicitly expanding two-digit-year date fields to contain full four-digit years in your files and databases and then using these fields in expanded form in your programs. This is the only method that assures reliable date processing for all applications.

You can use the millennium language extensions with each approach to achieve a solution, but each has advantages and disadvantages, as shown below.

Table 47. ***Advantages and disadvantages of Year 2000 solutions***			
Aspect	**Century window**	**Internal bridging**	**Full field expansion**
Implementation	Fast and easy but might not suit all applications	Some risk of corrupting data	Must ensure that changes to databases, copybooks, and programs are synchronized
Testing	Less testing is required because no changes to program logic	Testing is easy because changes to program logic are straightforward	
Duration of fix	Programs can function beyond 2000, but not a long-term solution	Programs can function beyond 2000, but not a permanent solution	Permanent solution
Performance	Might degrade performance	Good performance	Best performance
Maintenance			Maintenance is easier.

“Example: century window” on page 425
“Example: internal bridging” on page 426
“Example: converting files to expanded date form” on page 427

Related tasks
“Using a century window” on page 424
“Using internal bridging” on page 425
“Moving to full field expansion” on page 426

Using a century window

A *century window* is a 100-year interval, such as 1950-2049, within which any two-digit year is unique. For windowed date fields, you specify the century window start date by using the YEARWINDOW compiler option.

When the DATEPROC option is in effect, the compiler applies this window to two-digit date fields in the program. For example, given the century window 1930-2029, COBOL interprets two-digit years as follows:

- Year values from 00 through 29 are interpreted as years 2000-2029.
- Year values from 30 through 99 are interpreted as years 1930-1999.

To implement this century window, you use the DATE FORMAT clause to identify the date fields in your program and use the YEARWINDOW compiler option to define the century window as either a fixed window or a sliding window:

- For a fixed window, specify a four-digit year between 1900 and 1999 as the YEARWINDOW option value.

  For example, `YEARWINDOW(1950)` defines a fixed window of 1950-2049.
- For a sliding window, specify a negative integer from -1 through -99 as the YEARWINDOW option value.

  For example, `YEARWINDOW(-50)` defines a sliding window that starts 50 years before the year in which the program is running. So if the program is running in 2010, the century window is 1960-2059; in 2011 the century window automatically becomes 1961-2060, and so on.

The compiler automatically applies the century window to operations on the date fields that you have identified. You do not need any extra program logic to implement the windowing.

“Example: century window” on page 425

Related references
“DATEPROC” on page 259

“YEARWINDOW” on page 288
DATE FORMAT clause (*COBOL for Linux on x86 Language Reference*)
Restrictions on using date fields (*COBOL for Linux on x86 Language Reference*)

Example: century window

The following example shows (in bold) how to modify a program by using the DATE FORMAT clause to take advantage of automatic date windowing.

```
CBLQUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-60)
. . .
01  Loan-Record.
    05  Member-Number   Pic X(8).
    05  DVD-ID          Pic X(8).
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
    05  Date-Returned   Pic X(6) Date Format yyxxxx.
. . .
    If Date-Returned > Date-Due-Back Then
       Perform Fine-Member.
```

There are no changes to the PROCEDURE DIVISION. The addition of the DATE FORMAT clause to the two date fields means that the compiler recognizes them as windowed date fields and therefore applies the century window when processing the IF statement. For example, if Date-Due-Back contains 100102 (January 2, 2010) and Date-Returned contains 091231 (December 31, 2009), Date-Returned is less than (earlier than) Date-Due-Back, and thus the program does not perform the Fine-Member paragraph. (The program checks whether a DVD was returned on time.)

Using internal bridging

For internal bridging, you need to structure your program appropriately.

Do the following steps:

1. Read the input files with two-digit-year dates.
2. Declare these two-digit dates as windowed date fields and move them to expanded date fields, so that the compiler automatically expands them to four-digit-year dates.
3. In the main body of the program, use the four-digit-year dates for all date processing.
4. Window the dates back to two-digit years.
5. Write the two-digit-year dates to the output files.

This process provides a convenient migration path to a full expanded-date solution, and can have performance advantages over using windowed dates.

When you use this technique, your changes to the program logic are minimal. You simply add statements to expand and contract the dates, and change the statements that refer to dates to use the four-digit-year date fields in WORKING-STORAGE instead of the two-digit-year fields in the records.

Because you are converting the dates back to two-digit years for output, you should allow for the possibility that the year is outside the century window. For example, if a date field contains the year 2020, but the century window is 1920-2019, then the date is outside the window. Simply moving the year to a two-digit-year field will be incorrect. To protect against this problem, you can use a COMPUTE statement to store the date, with the ON SIZE ERROR phrase to detect whether the date is outside the century window.

“Example: internal bridging” on page 426

Related tasks
“Using a century window” on page 424
“Performing arithmetic on date fields” on page 434
“Moving to full field expansion” on page 426

Example: internal bridging

The following example shows (in bold) how a program can be changed to implement internal bridging.

```
CBL  DATEPROC(FLAG),YEARWINDOW(-60)
    . . .
    File Section.
    FD  Customer-File.
    01  Cust-Record.
        05  Cust-Number     Pic 9(9) Binary.
        . . .
        05  Cust-Date       Pic 9(6) Date Format yyxxxx.
    Working-Storage Section.
    77  Exp-Cust-Date       Pic 9(8) Date Format yyyyxxxx.
    . . .
    Procedure Division.
        Open I-O Customer-File.
        Read Customer-File.
        Move Cust-Date to Exp-Cust-Date.
        . . .
   *=====================================================*
   * Use expanded date in the rest of the program logic  *
   *=====================================================*
        . . .
        Compute Cust-Date = Exp-Cust-Date
             On Size Error
                Display "Exp-Cust-Date outside century window"
        End-Compute
        Rewrite Cust-Record.
```

Moving to full field expansion

Using the millennium language extensions, you can move gradually toward a solution that fully expands the date field.

Do the following steps:

1. Apply the century window solution, and use this solution until you have the resources to implement a more permanent solution.
2. Apply the internal bridging solution. This way you can use expanded dates in your programs while your files continue to hold dates in two-digit-year form. You can progress more easily to a full-field-expansion solution because there will be no further changes to the logic in the main body of the programs.
3. Change the file layouts and database definitions to use four-digit-year dates.
4. Change your COBOL copybooks to reflect these four-digit-year date fields.
5. Run a utility program (or special-purpose COBOL program) to copy files from the old format to the new format.
6. Recompile your programs and do regression testing and date testing.

After you have completed the first two steps, you can repeat the remaining steps any number of times. You do not need to change every date field in every file at the same time. Using this method, you can select files for progressive conversion based on criteria such as business needs or interfaces with other applications.

When you use this method, you need to write special-purpose programs to convert your files to expanded-date form.

“Example: converting files to expanded date form” on page 427

Example: converting files to expanded date form

The following example shows a simple program that copies from one file to another while expanding the date fields. The record length of the output file is larger than that of the input file because the dates are expanded.

```
CBL  QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-80)
      ******************************************
      ** CONVERT - Read a file, convert the date    **
      **           fields to expanded form, write   **
      **           the expanded records to a new    **
      **           file.                            **
      ******************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CONVERT.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INPUT-FILE
                  ASSIGN TO INFILE
                  FILE STATUS IS INPUT-FILE-STATUS.

           SELECT OUTPUT-FILE
                  ASSIGN TO OUTFILE
                  FILE STATUS IS OUTPUT-FILE-STATUS.

       DATA DIVISION.
       FILE SECTION.
       FD  INPUT-FILE
           RECORDING MODE IS F.
       01  INPUT-RECORD.
           03  CUST-NAME.
               05  FIRST-NAME  PIC X(10).
               05  LAST-NAME   PIC X(15).
           03  ACCOUNT-NUM     PIC 9(8).
           03  DUE-DATE        PIC X(6) DATE FORMAT YYXXXX.    (1)
           03  REMINDER-DATE   PIC X(6) DATE FORMAT YYXXXX.
           03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

       FD  OUTPUT-FILE
           RECORDING MODE IS F.
       01  OUTPUT-RECORD.
           03  CUST-NAME.
               05  FIRST-NAME  PIC X(10).
               05  LAST-NAME   PIC X(15).
           03  ACCOUNT-NUM     PIC 9(8).
           03  DUE-DATE        PIC X(8) DATE FORMAT YYYYXXXX.  (2)
           03  REMINDER-DATE   PIC X(8) DATE FORMAT YYYYXXXX.
           03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

       WORKING-STORAGE SECTION.

       01  INPUT-FILE-STATUS   PIC 99.
       01  OUTPUT-FILE-STATUS  PIC 99.

       PROCEDURE DIVISION.

           OPEN INPUT INPUT-FILE.
           OPEN OUTPUT OUTPUT-FILE.

       READ-RECORD.
           READ INPUT-FILE
                AT END GO TO CLOSE-FILES.
           MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.   (3)
           WRITE OUTPUT-RECORD.

           GO TO READ-RECORD.

       CLOSE-FILES.
           CLOSE INPUT-FILE.
           CLOSE OUTPUT-FILE.

           EXIT PROGRAM.

       END PROGRAM CONVERT.
```

Notes:

(1)
The fields DUE-DATE and REMINDER-DATE in the input record are Gregorian dates with two-digit year components. They are defined with a DATE FORMAT clause so that the compiler recognizes them as windowed date fields.

(2)
The output record contains the same two fields in expanded date format. They are defined with a DATE FORMAT clause so that the compiler treats them as four-digit-year date fields.

(3)
The MOVE CORRESPONDING statement moves each item in INPUT-RECORD to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded date fields, the compiler expands the year values using the current century window.

Using year-first, year-only, and year-last date fields

When you compare two date fields of either year-first or year-only types, the two dates must be compatible; that is, they must have the same number of nonyear characters. The number of digits for the year component need not be the same.

A *year-first* date field is a date field whose DATE FORMAT specification consists of YY or YYYY, followed by one or more Xs. The date format of a *year-only* date field has just the YY or YYYY. A *year-last* date field is a date field whose DATE FORMAT clause specifies one or more Xs preceding YY or YYYY.

Year-last date formats are commonly used to display dates, but are less useful computationally because the year, which is the most significant part of the date, is in the least significant position of the date representation.

Functional support for year-last date fields is limited to equal or unequal comparisons and certain kinds of assignment. The operands must be either dates with identical (year-last) date formats, or a date and a nondate. The compiler does not provide automatic windowing for operations on year-last dates. When an unsupported usage (such as arithmetic on year-last dates) occurs, the compiler provides an error-level message.

If you need more general date-processing capability for year-last dates, you should isolate and operate on the year part of the date.

"Example: comparing year-first date fields" on page 429

Related concepts
"Compatible dates" on page 428

Related tasks
"Using other date formats" on page 429

Compatible dates

The meaning of the term *compatible dates* depends on whether the usage occurs in the DATA DIVISION or the PROCEDURE DIVISION.

The DATA DIVISION usage deals with the declaration of date fields, and the rules that govern COBOL language elements such as subordinate data items and the REDEFINES clause. In the following example, Review-Date and Review-Year are compatible because Review-Year can be declared as a subordinate data item to Review-Date:

```
01  Review-Record.
    03  Review-Date              Date Format yyxxxx.
        05  Review-Year Pic XX    Date Format yy.
        05  Review-M-D  Pic XXXX.
```

The PROCEDURE DIVISION usage deals with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. For year-first and year-only date fields to be considered

compatible, date fields must have the same number of nonyear characters. For example, a field with DATE FORMAT YYXXXX is compatible with another field that has the same date format and with a YYYYXXXX field, but not with a YYXXX field.

Year-last date fields must have identical DATE FORMAT clauses. In particular, operations between windowed date fields and expanded year-last date fields are not allowed. For example, you can move a date field that has a date format of XXXXYY to another XXXXYY date field, but not to a date field that has a format of XXXXYYYY.

You can perform operations on date fields, or on a combination of date fields and nondates, provided that the date fields in the operation are compatible. For example, assume the following definitions:

```
01  Date-Gregorian-Win  Pic 9(6) Packed-Decimal Date Format yyxxxx.
01  Date-Julian-Win     Pic 9(5) Packed-Decimal Date Format yyxxx.
01  Date-Gregorian-Exp  Pic 9(8) Packed-Decimal Date Format yyyyxxxx.
```

The following statement is inconsistent because the number of nonyear digits is different between the two fields:

```
If Date-Gregorian-Win Less than Date-Julian-Win . . .
```

The following statement is accepted because the number of nonyear digits is the same for both fields:

```
If Date-Gregorian-Win Less than Date-Gregorian-Exp . . .
```

In this case the century window is applied to the windowed date field (`Date-Gregorian-Win`) to ensure that the comparison is meaningful.

When a nondate is used in conjunction with a date field, the nondate is either assumed to be compatible with the date field or is treated as a simple numeric value.

Example: comparing year-first date fields

The following example shows a windowed date field that is compared with an expanded date field.

```
77  Todays-Date         Pic X(8) Date Format yyyyxxxx.
01  Loan-Record.
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
. . .
    If Date-Due-Back > Todays-Date Then . . .
```

The century window is applied to `Date-Due-Back`. `Todays-Date` must have a DATE FORMAT clause to define it as an expanded date field. If it did not, it would be treated as a nondate field and would therefore be considered to have the same number of year digits as `Date-Due-Back`. The compiler would apply the assumed century window of 1900-1999, which would create an inconsistent comparison.

Using other date formats

To be eligible for automatic windowing, a date field should contain a two-digit year as the first or only part of the field. The remainder of the field, if present, must contain between one and four characters, but its content is not important.

If there are date fields in your application that do not fit these criteria, you might have to make some code changes to define just the year part of the date as a date field with the DATE FORMAT clause. Some examples of these types of date formats are:

- A seven-character field that consists of a two-digit year, three characters that contain an abbreviation of the month, and two digits for the day of the month. This format is not supported because date fields can have only one through four nonyear characters.

- A Gregorian date of the form DDMMYY. Automatic windowing is not provided because the year component is not the first part of the date. Year-last dates such as these are fully supported as windowed keys in SORT or MERGE statements, and are also supported in a limited number of other COBOL operations.

If you need to use date windowing in cases like these, you will need to add some code to isolate the year portion of the date.

Example: isolating the year

The following example shows how you can isolate the year portion of a data field that is in the form DDMMYY.

```
03  Last-Review-Date Pic 9(6).
03  Next-Review-Date Pic 9(6).
. . .
Add 1 to Last-Review-Date Giving Next-Review-Date.
```

In the code above, if Last-Review-Date contains 230110 (January 23, 2010), then Next-Review-Date will contain 230111 (January 23, 2011) after the ADD statement is executed. This is a simple method for setting the next date for an annual review. However, if Last-Review-Date contains 230199, then adding 1 yields 230200, which is not the required result.

Because the year is not the first part of these date fields, the DATE FORMAT clause cannot be applied without some code to isolate the year component. In the next example, the year component of both date fields has been isolated so that COBOL can apply the century window and maintain consistent results:

```
03  Last-Review-Date Date Format xxxxyy.
    05  Last-R-DDMM  Pic 9(4).
    05  Last-R-YY    Pic 99 Date Format yy.
03  Next-Review-Date Date Format xxxxyy.
    05  Next-R-DDMM  Pic 9(4).
    05  Next-R-YY    Pic 99 Date Format yy.
. . .
Move Last-R-DDMM to Next-R-DDMM.
Add 1 to Last-R-YY Giving Next-R-YY.
```

Manipulating literals as dates

If a windowed date field has an associated level-88 condition-name, then the literal in the VALUE clause is windowed against the century window of the compile unit rather than against the assumed century window of 1900-1999.

For example, suppose you have these data definitions:

```
05  Date-Due        Pic 9(6)  Date Format yyxxxx.
    88  Date-Target           Value 101220.
```

If the century window is 1950-2049, and the contents of Date-Due are 101220 (representing December 20, 2010), then the first condition below evaluates to true, but the second condition evaluates to false:

```
If Date-Target. . .
If Date-Due = 101220
```

The literal 101220 is treated as a nondate; therefore it is windowed against the assumed century window of 1900-1999, and represents December 20, 1909. But where the literal is specified in the VALUE clause of a level-88 condition-name, the literal becomes part of the data item to which it is attached. Because this data item is a windowed date field, the century window is applied whenever that data item is referenced.

You can also use the DATEVAL intrinsic function in a comparison expression to convert a literal to a date field. The resulting date field is treated as either a windowed date field or an expanded date field to ensure a consistent comparison. For example, using the above definitions, both of the following conditions evaluate to true:

```
If Date-Due = Function DATEVAL (101220 "YYXXXX")
If Date-Due = Function DATEVAL (20101220 "YYYYXXXX")
```

With a level-88 condition-name, you can specify the THRU option on the VALUE clause, but you must specify a fixed century window in the YEARWINDOW compiler option rather than a sliding window. For example:

```
05  Year-Field  Pic 99  Date Format yy.
    88 In-Range         Value 98 Thru 06.
```

With this form, the windowed value of the second item in the range must be greater than the windowed value of the first item. However, the compiler can verify this difference only if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-70)).

The windowed order requirement does not apply to year-last date fields. If you specify a condition-name VALUE clause with the THROUGH phrase for a year-last date field, the two literals must follow normal COBOL rules. That is, the first literal must be less than the second literal.

Related concepts
“Assumed century window” on page 431
“Treatment of nondates” on page 432

Related tasks
“Controlling date processing explicitly” on page 435

Assumed century window

When a program uses windowed date fields, the compiler applies the century window that is defined by the YEARWINDOW compiler option to the compilation unit. When a windowed date field is used in conjunction with a nondate, and the context demands that the nondate be treated as a windowed date, the compiler uses an assumed century window to resolve the nondate field.

The assumed century window is 1900-1999, which typically is not the same as the century window for the compilation unit.

In many cases, particularly for literal nondates, this assumed century window is the correct choice. In the following construct, the literal should retain its original meaning of January 1, 1972, and not change to 2072 if the century window is, for example, 1975-2074:

```
01  Manufacturing-Record.
    03  Makers-Date Pic X(6) Date Format yyxxxx.
. . .
    If Makers-Date Greater than "720101" . . .
```

Even if the assumption is correct, it is better to make the year explicit and eliminate the warning-level diagnostic message (which results from applying the assumed century window) by using the DATEVAL intrinsic function:

```
If Makers-Date Greater than
    Function Dateval("19720101" "YYYYXXXX") . . .
```

In some cases, the assumption might not be correct. For the following example, assume that Project-Controls is in a copy member that is used by other applications that have not yet been upgraded for year 2000 processing, and therefore Date-Target cannot have a DATE FORMAT clause:

```
01  Project-Controls.
    03  Date-Target     Pic 9(6).
. . .
01  Progress-Record.
    03  Date-Complete   Pic 9(6) Date Format yyxxxx.
. . .
    If Date-Complete Less than Date-Target . . .
```

In the example above, the following three conditions need to be true to make Date-Complete earlier than (less than) Date-Target:

- The century window is 1910-2009.
- Date-Complete is 991202 (Gregorian date: December 2, 1999).
- Date-Target is 000115 (Gregorian date: January 15, 2000).

However, because Date-Target does not have a DATE FORMAT clause, it is a nondate. Therefore, the century window applied to it is the assumed century window of 1900-1999, and it is processed as January 15, 1900. So Date-Complete will be greater than Date-Target, which is not the required result.

In this case, you should use the DATEVAL intrinsic function to convert Date-Target to a date field for this comparison. For example:

```
If Date-Complete Less than
    Function Dateval (Date-Target "YYXXXX") . . .
```

Related tasks
"Controlling date processing explicitly" on page 435

Treatment of nondates

How the compiler treats a nondate depends upon its context.

The following items are nondates:

- A literal value.
- A data item whose data description does not include a DATE FORMAT clause.
- The results (intermediate or final) of some arithmetic expressions. For example, the difference of two date fields is a nondate, whereas the sum of a date field and a nondate is a date field.
- The output from the UNDATE intrinsic function.

When you use a nondate in conjunction with a date field, the compiler interprets the nondate either as a date whose format is compatible with the date field or as a simple numeric value. This interpretation depends on the context in which the date field and nondate are used, as follows:

- Comparison

  When a date field is compared with a nondate, the nondate is considered to be compatible with the date field in the number of year and nonyear characters. In the following example, the nondate literal 971231 is compared with a windowed date field:

```
01  Date-1    Pic 9(6) Date Format yyxxxx.
. . .
    If Date-1 Greater than 971231 . . .
```

The nondate literal 971231 is treated as if it had the same DATE FORMAT as Date-1, but with a base year of 1900.

- Arithmetic operations

  In all supported arithmetic operations, nondate fields are treated as simple numeric values. In the following example, the numeric value 10000 is added to the Gregorian date in Date-2, effectively adding one year to the date:

```
01  Date-2    Pic 9(6) Date Format yyxxxx.
. . .
    Add 10000 to Date-2.
```

- MOVE statement

  Moving a date field to a nondate is not supported. However, you can use the UNDATE intrinsic function to do this.

  When you move a nondate to a date field, the sending field is assumed to be compatible with the receiving field in the number of year and nonyear characters. For example, when you move a nondate to a windowed date field, the nondate field is assumed to contain a compatible date with a two-digit year.

Using sign conditions

Some applications use special values such as zeros in date fields to act as a trigger, that is, to signify that some special processing is required.

For example, in an Orders file, a value of zero in Order-Date might signify that the record is a customer totals record rather than an order record. The program compares the date to zero, as follows:

```
01  Order-Record.
    05  Order-Date      Pic S9(5) Comp-3 Date Format yyxxx.
. . .
    If Order-Date Equal Zero Then . . .
```

However, this comparison is not valid because the literal value Zero is a nondate, and is therefore windowed against the assumed century window to give a value of 1900000.

Alternatively, you can use a sign condition instead of a literal comparison as follows. With a sign condition, Order-Date is treated as a nondate, and the century window is not considered.

```
If Order-Date Is Zero Then . . .
```

This approach applies only if the operand in the sign condition is a simple identifier rather than an arithmetic expression. If an expression is specified, the expression is evaluated first, with the century window being applied where appropriate. The sign condition is then compared with the results of the expression.

You could use the UNDATE intrinsic function instead to achieve the same result.

Related concepts
“Treatment of nondates” on page 432

Related tasks
“Controlling date processing explicitly” on page 435

Related references
“DATEPROC” on page 259

Performing arithmetic on date fields

You can perform arithmetic operations on numeric date fields in the same manner as on any numeric data item. Where appropriate, the century window will be used in the calculation.

However, there are some restrictions on where date fields can be used in arithmetic expressions and statements. Arithmetic operations that include date fields are restricted to:

- Adding a nondate to a date field
- Subtracting a nondate from a date field
- Subtracting a date field from a compatible date field to give a nondate result

The following arithmetic operations are not allowed:

- Any operation between incompatible date fields
- Adding two date fields
- Subtracting a date field from a nondate
- Unary minus applied to a date field
- Multiplication, division, or exponentiation of or by a date field
- Arithmetic expressions that specify a year-last date field
- Arithmetic expressions that specify a year-last date field, except as a receiving data item when the sending field is a nondate

Date semantics are provided for the year parts of date fields but not for the nonyear parts. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

Related tasks
"Allowing for overflow
from windowed date fields" on page 434
"Specifying the order of evaluation" on page 435

Allowing for overflow from windowed date fields

A (nonyear-last) windowed date field that participates in an arithmetic operation is processed as if the value of the year component of the field were first incremented by 1900 or 2000, depending on the century window.

```
01 Review-Record.
    03 Last-Review-Year Pic 99 Date Format yy.
    03 Next-Review-Year Pic 99 Date Format yy.
. . .
    Add 10 to Last-Review-Year Giving Next-Review-Year.
```

In the example above, if the century window is 1910-2009, and the value of `Last-Review-Year` is 98, then the computation proceeds as if `Last-Review-Year` is first incremented by 1900 to give 1998. Then the ADD operation is performed, giving a result of 2008. This result is stored in `Next-Review-Year` as 08.

However, the following statement would give a result of 2018:

```
Add 20 to Last-Review-Year Giving Next-Review-Year.
```

This result falls outside the range of the century window. If the result is stored in `Next-Review-Year`, it will be incorrect because later references to `Next-Review-Year` will interpret it as 1918. In this case, the result of the operation depends on whether the `ON SIZE ERROR` phrase is specified on the ADD statement:

- If SIZE ERROR is specified, the receiving field is not changed, and the SIZE ERROR imperative statement is executed.
- If SIZE ERROR is not specified, the result is stored in the receiving field with the left-hand digits truncated.

This consideration is important when you use internal bridging. When you contract a four-digit-year date field back to two digits to write it to the output file, you need to ensure that the date falls within the century window. Then the two-digit-year date will be represented correctly in the field.

To ensure appropriate calculations, use a COMPUTE statement to do the contraction, with a SIZE ERROR phrase to handle the out-of-window condition. For example:

```
Compute Output-Date-YY = Work-Date-YYYY
On Size Error Perform CenturyWindowOverflow.
```

SIZE ERROR processing for windowed date receivers recognizes any year value that falls outside the century window. That is, a year value less than the starting year of the century window raises the SIZE ERROR condition, as does a year value greater than the ending year of the century window.

Related tasks
"Using internal bridging" on page 425

Specifying the order of evaluation

Because of the restrictions on date fields in arithmetic expressions, you might find that programs that previously compiled successfully now produce diagnostic messages when some of the data items are changed to date fields.

```
01 Dates-Record.
    03 Start-Year-1 Pic 99 Date Format yy.
    03 End-Year-1   Pic 99 Date Format yy.
    03 Start-Year-2 Pic 99 Date Format yy.
    03 End-Year-2   Pic 99 Date Format yy.
. . .
    Compute End-Year-2 = Start-Year-2 + End-Year-1 - Start-Year-1.
```

In the example above, the first arithmetic expression evaluated is:

```
Start-Year-2 + End-Year-1
```

However, the addition of two date fields is not permitted. To resolve these date fields, you should use parentheses to isolate the parts of the arithmetic expression that are allowed. For example:

```
Compute End-Year-2 = Start-Year-2 + (End-Year-1 - Start-Year-1).
```

In this case, the first arithmetic expression evaluated is:

```
End-Year-1 - Start-Year-1
```

The subtraction of one date field from another is permitted and gives a nondate result. This nondate result is then added to the date field End-Year-1, giving a date field result that is stored in End-Year-2.

Controlling date processing explicitly

There might be times when you want COBOL data items to be treated as date fields only under certain conditions or only in specific parts of the program. Or your application might contain two-digit-year date

fields that cannot be declared as windowed date fields because of some interaction with another software product.

For example, if a date field is used in a context where it is recognized only by its true binary contents without further interpretation, the date in that field cannot be windowed. Such date fields include:

- A key in an SdU file
- A search field in a database system such as Db2
- A key field in a CICS command

Conversely, there might be times when you want a date field to be treated as a nondate in specific parts of the program.

COBOL provides two intrinsic functions to deal with these conditions:

DATEVAL
Converts a nondate to a date field

UNDATE
Converts a date field to a nondate

Related tasks
“Using DATEVAL” on page 436
“Using UNDATE” on page 436

Using DATEVAL

You can use the DATEVAL intrinsic function to convert a nondate to a date field, so that COBOL will apply the relevant date processing to the field.

The first argument in the function is the nondate to be converted, and the second argument specifies the date format. The second argument is a literal string with a specification similar to that of the date pattern in the DATE FORMAT clause.

In most cases, the compiler makes the correct assumption about the interpretation of a nondate but accompanies this assumption with a warning-level diagnostic message. This message typically happens when a windowed date is compared with a literal:

```
03  When-Made        Pic x(6) Date Format yyxxxx.
. . .
If When-Made = "850701" Perform Warranty-Check.
```

The literal is assumed to be a compatible windowed date but with a century window of 1900-1999, thus representing July 15, 1985. You can use the DATEVAL intrinsic function to make the year of the literal date explicit and eliminate the warning message:

```
If When-Made = Function Dateval("19850701" "YYYYXXXX")
    Perform Warranty-Check.
```

“Example: DATEVAL” on page 437

Using UNDATE

You can use the UNDATE intrinsic function to convert a date field to a nondate so that it can be referenced without any date processing.

Attention: Avoid using UNDATE except as a last resort, because the compiler will lose the flow of date fields in your program. This problem could result in date comparisons not being windowed properly.

Use more DATE FORMAT clauses instead of function UNDATE for MOVE and COMPUTE.

“Example: UNDATE” on page 437

Example: DATEVAL

This example shows a case where it is better to leave a field as a nondate, and use the DATEVAL intrinsic function in a comparison statement.

Assume that a field `Date-Copied` is referenced many times in a program, but that most of the references just move the value between records or reformat it for printing. Only one reference relies on it to contain a date (for comparison with another date). In this case, it is better to leave the field as a nondate, and use the DATEVAL intrinsic function in the comparison statement. For example:

```
03  Date-Distributed Pic 9(6) Date Format yyxxxx.
03  Date-Copied      Pic 9(6).
. . .
If Function DATEVAL(Date-Copied "YYXXXX") Less than Date-Distributed . . .
```

In this example, DATEVAL converts `Date-Copied` to a date field so that the comparison will be meaningful.

Related references
DATEVAL (*COBOL for Linux on x86 Language Reference*)

Example: UNDATE

The following example shows a case where you might want to convert a date field to a nondate.

The field `Invoice-Date` is a windowed Julian date. In some records, it contains the value `00999` to indicate that the record is not a true invoice record, but instead contains file-control information.

`Invoice-Date` has a `DATE FORMAT` clause because most of its references in the program are date-specific. However, when it is checked for the existence of a control record, the value 00 in the year component will lead to some confusion. A year value of 00 in `Invoice-Date` could represent either 1900 or 2000, depending on the century window. This is compared with a nondate (the literal `00999` in the example), which will always be windowed against the assumed century window and therefore always represents the year 1900.

To ensure a consistent comparison, you should use the UNDATE intrinsic function to convert `Invoice-Date` to a nondate. Therefore, if the IF statement is not comparing date fields, it does not need to apply windowing. For example:

```
01  Invoice-Record.
    03  Invoice-Date    Pic x(5) Date Format yyxxx.
. . .
    If FUNCTION UNDATE(Invoice-Date) Equal "00999" . . .
```

Related references
UNDATE (*COBOL for Linux on x86 Language Reference*)

Analyzing and avoiding date-related diagnostic messages

When the `DATEPROC(FLAG)` compiler option is in effect, the compiler produces diagnostic messages for every statement that defines or references a date field.

As with all compiler-generated messages, each date-related message has one of the following severity levels:

- Information-level, to draw your attention to the definition or use of a date field.
- Warning-level, to indicate that the compiler has had to make an assumption about a date field or nondate because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

- Error-level, to indicate that the usage of the date field is incorrect. Compilation continues, but runtime results are unpredictable.
- Severe-level, to indicate that the usage of the date field is incorrect. The statement that caused this error is discarded from the compilation.

The easiest way to use the MLE messages is to compile with a FLAG option setting that embeds the messages in the source listing after the line to which the messages refer. You can choose to see all MLE messages or just certain severities.

To see all MLE messages, specify the FLAG(I,I) and DATEPROC(FLAG) compiler options. Initially, you might want to see all of the messages to understand how MLE is processing the date fields in your program. For example, if you want to do a static analysis of the date usage in a program by using the compile listing, use FLAG (I,I).

However, it is recommended that you specify FLAG(W,W) for MLE-specific compiles. You must resolve all severe-level (S-level) error messages, and you should resolve all error-level (E-level) messages as well. For the warning-level (W-level) messages, you need to examine each message and use the following guidelines to either eliminate the message or, for unavoidable messages, ensure that the compiler makes correct assumptions:

- The diagnostic messages might indicate some date data items that should have had a DATE FORMAT clause. Either add DATE FORMAT clauses to these items or use the DATEVAL intrinsic function in references to them.
- Pay particular attention to literals in relation conditions that involve date fields or in arithmetic expressions that include date fields. You can use the DATEVAL function on literals (as well as nondate data items) to specify a DATE FORMAT pattern to be used. As a last resort, you can use the UNDATE function to enable a date field to be used in a context where you do not want date-oriented behavior.
- With the REDEFINES and RENAMES clauses, the compiler might produce a warning-level diagnostic message if a date field and a nondate occupy the same storage location. You should check these cases carefully to confirm that all uses of the aliased data items are correct, and that none of the perceived nondate redefinitions actually is a date or can adversely affect the date logic in the program.

In some cases, a the W-level message might be acceptable, but you might want to change the code to get a compile with a return code of zero.

To avoid warning-level diagnostic messages, follow these guidelines:

- Add DATE FORMAT clauses to any data items that will contain dates, even if the items are not used in comparisons. But see the Related references below about restrictions on using date fields. For example, you cannot use the DATE FORMAT clause on a data item that is described implicitly or explicitly as USAGE NATIONAL.
- Do not specify a date field in a context where a date field does not make sense, such as a FILE STATUS, PASSWORD, ASSIGN USING, LABEL RECORD, or LINAGE item. If you do, you will get a warning-level message and the date field will be treated as a nondate.
- Ensure that implicit or explicit aliases for date fields are compatible, such as in a group item that consists solely of a date field.
- Ensure that if a date field is defined with a VALUE clause, the value is compatible with the date field definition.
- Use the DATEVAL intrinsic function if you want a nondate treated as a date field, such as when moving a nondate to a date field or when comparing a windowed date with a nondate and you want a windowed date comparison. If you do not use DATEVAL, the compiler will make an assumption about the use of the nondate and produce a warning-level diagnostic message. Even if the assumption is correct, you might want to use DATEVAL to eliminate the message.
- Use the UNDATE intrinsic function if you want a date field treated as a nondate, such as moving a date field to a nondate, or comparing a nondate and a windowed date field when you do not want a windowed comparison.

Related tasks
“Controlling date processing

explicitly" on page 435
COBOL Millennium Language Extensions Guide (Analyzing date-related diagnostic messages)

Related references
Restrictions on using date fields (*COBOL for Linux on x86 Language Reference*)

Avoiding problems in processing dates

When you change a COBOL program to use the millennium language extensions, you might find that some parts of the program need special attention to resolve unforeseen changes in behavior. For example, you might need to avoid problems with packed-decimal fields and problems that occur if you move from expanded to windowed date fields.

Related tasks
"Avoiding problems with packed-decimal fields" on page 439
"Moving from expanded to windowed date fields" on page 439

Avoiding problems with packed-decimal fields

`COMPUTATIONAL-3` fields (packed-decimal format) are often defined as having an odd number of digits even if the field will not be used to hold a number of that magnitude. The internal representation of packed-decimal numbers always allows for an odd number of digits.

A field that holds a six-digit Gregorian date, for example, can be declared as `PIC S9(6) COMP-3`. This declaration will reserve 4 bytes of storage. But a programmer might have declared the field as `PIC S9(7)`, knowing that this would reserve 4 bytes with the high-order digit always containing a zero.

If you add a `DATE FORMAT YYXXXX` clause to this field, the compiler will issue a diagnostic message because the number of digits in the `PICTURE` clause does not match the size of the date format specification. In this case, you need to carefully check each use of the field. If the high-order digit is never used, you can simply change the field definition to `PIC S9(6)`. If it is used (for example, if the same field can hold a value other than a date), you need to take some other action, such as:

- Using a `REDEFINES` clause to define the field as both a date and a nondate (this usage will also produce a warning-level diagnostic message)
- Defining another `WORKING-STORAGE` field to hold the date, and moving the numeric field to the new field
- Not adding a `DATE FORMAT` clause to the data item, and using the `DATEVAL` intrinsic function when referring to it as a date field

Moving from expanded to windowed date fields

When you move an expanded alphanumeric date field to a windowed date field, the move does not follow the normal COBOL conventions for alphanumeric moves. When both the sending and receiving fields are date fields, the move is right justified, not left justified as normal. For an expanded-to-windowed (contracting) move, the leading two digits of the year are truncated.

Depending on the contents of the sending field, the results of such a move might be incorrect. For example:

```
77  Year-Of-Birth-Exp  Pic x(4) Date Format yyyy.
77  Year-Of-Birth-Win  Pic xx   Date Format yy.
. . .
    Move Year-Of-Birth-Exp to Year-Of-Birth-Win.
```

If `Year-Of-Birth-Exp` contains '1925', `Year-Of-Birth-Win` will contain '25'. However, if the century window is 1930-2029, subsequent references to `Year-Of-Birth-Win` will treat it as 2025, which is incorrect.

Part 7. Improving performance and productivity

© Copyright IBM Corp. 2021

Chapter 27. Tuning your program

When a program is comprehensible, you can assess its performance. A tangled control flow makes a program difficult to understand and maintain, and inhibits the optimization of its code.

To improve the performance of your program, examine at least these aspects:

- Underlying algorithms: For best performance, using sound algorithms is essential. For example:
  - A sophisticated algorithm for sorting a million items might be hundreds of thousands of times faster than a simple algorithm.
  - If the program frequently accesses data, reduce the number of steps to access the data.
- Data structures: Using data structures that are appropriate for the algorithms is essential.

You can write programs that result in better generated code sequences and use system services more efficiently. These additional aspects can affect performance:

- Coding techniques: Use a programming style that enables the optimizer to choose efficient data types and handle tables efficiently.
- Optimization: You can optimize code by using the `OPTIMIZE` compiler option.
- Compiler options and `USE FOR DEBUGGING ON ALL PROCEDURES`: Some compiler options and language affect program efficiency.
- Runtime environment: Consider your choice of runtime options.
- **CICS**: To improve transaction response time, convert instances of `EXEC CICS LINK` statements to `CALL` statements.

  For information about improving performance of dynamic calls under CICS, see the Related tasks.

Related concepts
"Optimization" on page 450

Related tasks
"Tuning the performance of dynamic calls under CICS" on page 333
"Using an optimal programming style" on page 443
"Choosing efficient data types" on page 445
"Handling tables efficiently" on page 447
"Optimizing your code" on page 450
"Choosing compiler features to enhance performance" on page 450
"Improving SFS performance" on page 147

Related references
Chapter 15, "Runtime options," on page 297
"Performance-related compiler options" on page 451

Using an optimal programming style

The coding style you use can affect how the optimizer handles your code. You can improve optimization by using structured programming techniques, factoring expressions, using symbolic constants, and grouping constant and duplicate computations.

Related tasks
"Using structured programming" on page 444
"Factoring expressions" on page 444

© Copyright IBM Corp. 2021

"Using symbolic constants" on page 444
"Grouping constant computations" on page 444
"Grouping duplicate computations" on page 445

Using structured programming

Using structured programming statements, such as EVALUATE and inline PERFORM, makes your program more comprehensible and generates a more linear control flow. As a result, the optimizer can operate over larger regions of the program, which gives you more efficient code.

Use top-down programming constructs. Out-of-line PERFORM statements are a natural means of doing top-down programming. Out-of-line PERFORM statements can often be as efficient as inline PERFORM statements, because the optimizer can simplify or remove the linkage code.

Avoid using the following constructs:

- ALTER statements
- Backward branches (except as needed for loops for which PERFORM is unsuitable).
- PERFORM procedures that involve irregular control flow (such as preventing control from passing to the end of the procedure and returning to the PERFORM statement)

Factoring expressions

By factoring expressions in your programs, you can potentially eliminate a lot of unnecessary computation.

For example, the first block of code below is more efficient than the second block of code:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT
```

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM
```

The optimizer does not factor expressions.

Using symbolic constants

To have the optimizer recognize a data item as a constant throughout the program, initialize it with a VALUE clause and do not change it anywhere in the program.

If you pass a data item to a subprogram BY REFERENCE, the optimizer treats it as an external data item and assumes that it is changed at every subprogram call.

If you move a literal to a data item, the optimizer recognizes the data item as a constant only in a limited area of the program after the MOVE statement.

Grouping constant computations

When several items in an expression are constant, ensure that the optimizer is able to optimize them. The compiler is bound by the left-to-right evaluation rules of COBOL. Therefore, either move all the constants to the left side of the expression or group them inside parentheses.

For example, if V1, V2, and V3 are variables and C1, C2, and C3 are constants, the expressions on the left below are preferable to the corresponding expressions on the right:

More efficient	Less efficient
`V1 * V2 * V3 * (C1 * C2 * C3)`	`V1 * V2 * V3 * C1 * C2 * C3`
`C1 + C2 + C3 + V1 + V2 + V3`	`V1 + C1 + V2 + C2 + V3 + C3`

In production programming, there is often a tendency to place constant factors on the right-hand side of expressions. However, such placement can result in less efficient code because optimization is lost.

Grouping duplicate computations

When components of different expressions are duplicates, ensure that the compiler is able to optimize them. For arithmetic expressions, the compiler is bound by the left-to-right evaluation rules of COBOL. Therefore, either move all the duplicates to the left side of the expressions or group them inside parentheses.

If V1 through V5 are variables, the computation `V2 * V3 * V4` is a duplicate (known as a common subexpression) in the following two statements:

```
COMPUTE A = V1 * (V2 * V3 * V4)
COMPUTE B = V2 * V3 * V4 * V5
```

In the following example, `V2 + V3` is a common subexpression:

```
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V2 + V3 + V4
```

In the following example, there is no common subexpression:

```
COMPUTE A = V1 * V2 * V3 * V4
COMPUTE B = V2 * V3 * V4 * V5
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V4 + V2 + V3
```

The optimizer can eliminate duplicate computations. You do not need to introduce artificial temporary computations; a program is often more comprehensible and faster without them.

Choosing efficient data types

Choosing the appropriate data type and `PICTURE` clause can produce more efficient code, as can avoiding `USAGE DISPLAY` and `USAGE NATIONAL` data items in areas that are heavily used for computations.

Making a data item too large can reduce performance, but a data item whose length is a small power of 2 bytes (1, 2, 4, 8 or 16), and which is aligned on a power of 2-byte boundary matching its size can usually be initialized and moved more quickly and with fewer instructions than one with an odd length or alignment.

Arithmetic is faster with binary than with packed-decimal, which is faster than zoned-decimal or DISPLAY, which is faster than national-decimal.

Options that affect types can also affect performance. For example, `FLOAT(BE)` is more expensive than `FLOAT(NATIVE)`.

Consistent data types can reduce the need for conversions during operations on data items. You can also improve program performance by carefully determining when to use fixed-point and floating-point data types.

Related concepts
“Formats for numeric data” on page 39

Related tasks
“Choosing efficient computational data items” on page 446
“Using consistent data types” on page 446
“Making arithmetic expressions efficient” on page 446
“Making exponentiations efficient” on page 447

Choosing efficient computational data items

When you use a data item mainly for arithmetic or as a subscript, code USAGE BINARY on the data description entry for the item. The operations for manipulating binary data are faster than those for manipulating decimal data.

However, if a fixed-point arithmetic statement has intermediate results with a large precision (number of significant digits), the compiler uses decimal arithmetic anyway, after converting the operands to packed-decimal form. For fixed-point arithmetic statements, the compiler normally uses binary arithmetic for simple computations with binary operands if the precision is eight or fewer digits. Above 18 digits, the compiler always uses decimal arithmetic. With a precision of nine to 18 digits, the compiler uses either form.

To produce the most efficient code for a BINARY data item, ensure that it has:

- A sign (an S in its PICTURE clause)
- Eight or fewer digits

For a data item that is larger than eight digits or is used with DISPLAY or NATIONAL data items, use PACKED-DECIMAL.

To produce the most efficient code for a PACKED-DECIMAL data item, ensure that it has:

- A sign (an S in its PICTURE clause)
- An odd number of digits (9s in the PICTURE clause), so that it occupies an exact number of bytes without a half byte left over

Using consistent data types

In operations on operands of different types, one of the operands must be converted to the same type as the other. Each conversion requires several instructions. For example, one of the operands might need to be scaled to give it the appropriate number of decimal places.

You can largely avoid conversions by using consistent data types and by giving both operands the same usage and also appropriate PICTURE specifications. That is, you should ensure that two numbers to be compared, added, or subtracted not only have the same usage but also the same number of decimal places (9s after the V in the PICTURE clause).

Making arithmetic expressions efficient

Computation of arithmetic expressions that are evaluated in floating point is most efficient when the operands need little or no conversion. Use operands that are COMP-1 or COMP-2 to produce the most efficient code.

Define integer items as BINARY or PACKED-DECIMAL with nine or fewer digits to afford quick conversion to floating-point data. Also, conversion from a COMP-1 or COMP-2 item to a fixed-point integer with nine or fewer digits, without SIZE ERROR in effect, is efficient when the value of the COMP-1 or COMP-2 item is less than 1,000,000,000.

Making exponentiations efficient

Use floating point for exponentiations for large exponents to achieve faster evaluation and more accurate results.

For example, the first statement below is computed more quickly and accurately than the second statement:

```
COMPUTE fixed-point1 = fixed-point2 ** 100000.E+00
COMPUTE fixed-point1 = fixed-point2 ** 100000
```

A floating-point exponent causes floating-point arithmetic to be used to compute the exponentiation.

Handling tables efficiently

You can use several techniques to improve the efficiency of table-handling operations, and to influence the optimizer. The return for your efforts can be significant, particularly when table-handling operations are a major part of an application.

The following two guidelines affect your choice of how to refer to table elements:

- Use indexing rather than subscripting.

  Although the compiler can eliminate duplicate indexes and subscripts, the original reference to a table element is more efficient with indexes (even if the subscripts were BINARY). The value of an index has the element size factored into it, whereas the value of a subscript must be multiplied by the element size when the subscript is used. The index already contains the displacement from the start of the table, and this value does not have to be calculated at run time. However, subscripting might be easier to understand and maintain.

- Use relative indexing.

  Relative index references (that is, references in which an unsigned numeric literal is added to or subtracted from the index-name) are executed at least as fast as direct index references, and sometimes faster. There is no merit in keeping alternative indexes with the offset factored in.

Whether you use indexes or subscripts, the following coding guidelines can help you get better performance:

- Put constant and duplicate indexes or subscripts on the left.

  You can reduce or eliminate runtime computations this way. Even when all the indexes or subscripts are variable, try to use your tables so that the rightmost subscript varies most often for references that occur close to each other in the program. This practice also improves the pattern of storage references and also paging. If all the indexes or subscripts are duplicates, then the entire index or subscript computation is a common subexpression.

- Specify the element length so that it matches that of related tables.

  When you index or subscript tables, it is most efficient if all the tables have the same element length. That way, the stride for the last dimension of the tables is the same, and the optimizer can reuse the rightmost index or subscript computed for one table. If both the element lengths and the number of occurrences in each dimension are equal, then the strides for dimensions other than the last are also equal, resulting in greater commonality between their subscript computations. The optimizer can then reuse indexes or subscripts other than the rightmost.

- Avoid errors in references by coding index and subscript checks into your program.

  If you need to validate indexes and subscripts, it might be faster to code your own checks than to use the SSRANGE compiler option.

You can also improve the efficiency of tables by using these guidelines:

- Use binary data items for all subscripts.

When you use subscripts to address a table, use a BINARY signed data item with eight or fewer digits. In some cases, using four or fewer digits for the data item might also improve processing time.

- Use binary data items for variable-length table items.

  For tables with variable-length items, you can improve the code for OCCURS DEPENDING ON (ODO). To avoid unnecessary conversions each time the variable-length items are referenced, specify BINARY for OCCURS . . . DEPENDING ON objects.

- Use fixed-length data items whenever possible.

  Copying variable-length data items into a fixed-length data item before a period of high-frequency use can reduce some of the overhead associated with using variable-length data items.

- Organize tables according to the type of search method used.

  If the table is searched sequentially, put the data values most likely to satisfy the search criteria at the beginning of the table. If the table is searched using a binary search algorithm, put the data values in the table sorted alphabetically on the search key field.

Related concepts
"Optimization of table references" on page 448

Related tasks
"Referring to an item in a table" on page 62
"Choosing efficient data types" on page 445

Related references
"SSRANGE" on page 281

Optimization of table references

The COBOL compiler optimizes table references in several ways.

For the table element reference ELEMENT(S1 S2 S3), where S1, S2, and S3 are subscripts, the compiler evaluates the following expression:

```
comp_s1 * d1 + comp_s2 * d2 + comp_s3 * d3 + base_address
```

Here comp_s1 is the value of S1 after conversion to binary, comp-s2 is the value of S2 after conversion to binary, and so on. The strides for each dimension are d1, d2, and d3. The *stride* of a given dimension is the distance in bytes between table elements whose occurrence numbers in that dimension differ by 1 and whose other occurrence numbers are equal. For example, the stride d2 of the second dimension in the above example is the distance in bytes between ELEMENT(S1 1 S3) and ELEMENT(S1 2 S3).

Index computations are similar to subscript computations, except that no multiplication needs to be done. Index values have the stride factored into them. They involve loading the indexes into registers, and these data transfers can be optimized, much as the individual subscript computation terms are optimized.

Because the compiler evaluates expressions from left to right, the optimizer finds the most opportunities to eliminate computations when the constant or duplicate subscripts are the leftmost.

Optimization of constant and variable items

Assume that C1, C2, . . . are constant data items and that V1, V2, . . . are variable data items. Then, for the table element reference ELEMENT(V1 C1 C2) the compiler can eliminate only the individual terms comp_c1 * d2 and comp_c2 * d3 as constant from the expression:

```
comp_v1 * d1 + comp_c1 * d2 + comp_c2 * d3 + base_address
```

However, for the table element reference `ELEMENT(C1 C2 V1)` the compiler can eliminate the entire subexpression `comp_c1 * d1 + comp_c2 * d2` as constant from the expression:

```
comp_c1 * d1 + comp_c2 * d2 + comp_v1 * d3 + base_address
```

In the table element reference `ELEMENT(C1 C2 C3)`, all the subscripts are constant, and so no subscript computation is done at run time. The expression is:

```
comp_c1 * d1 + comp_c2 * d2 + comp_c3 * d3 + base_address
```

With the optimizer, this reference can be as efficient as a reference to a scalar (nontable) item.

Optimization of duplicate items

In the table element references `ELEMENT(V1 V3 V4)` and `ELEMENT(V2 V3 V4)` only the individual terms `comp_v3 * d2` and `comp_v4 * d3` are common subexpressions in the expressions needed to reference the table elements:

```
comp_v1 * d1 + comp_v3 * d2 + comp_v4 * d3 + base_address
comp_v2 * d1 + comp_v3 * d2 + comp_v4 * d3 + base_address
```

However, for the two table element references `ELEMENT(V1 V2 V3)` and `ELEMENT(V1 V2 V4)` the entire subexpression `comp_v1 * d1 + comp_v2 * d2` is common between the two expressions needed to reference the table elements:

```
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
comp_v1 * d1 + comp_v2 * d2 + comp_v4 * d3 + base_address
```

In the two references `ELEMENT(V1 V2 V3)` and `ELEMENT(V1 V2 V3)`, the expressions are the same:

```
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
```

With the optimizer, the second (and any subsequent) reference to the same element can be as efficient as a reference to a scalar (nontable) item.

Optimization of variable-length items

A group item that contains a subordinate `OCCURS DEPENDING ON` data item has a variable length. The program must perform special code every time a variable-length data item is referenced.

Because this code is out-of-line, it might interrupt optimization. Furthermore, the code to manipulate variable-length data items is much less efficient than that for fixed-size data items and can significantly increase processing time. For instance, the code to compare or move a variable-length data item might involve calling a library routine and is much slower than the same code for fixed-length data items.

Comparison of direct and relative indexing

Relative index references are as fast as or faster than direct index references.

The direct indexing in `ELEMENT (I5, J3, K2)` requires this preprocessing:

```
SET I5 TO I
SET I5 UP BY 5
SET J3 TO J
SET J3 DOWN BY 3
SET K2 TO K
SET K2 UP BY 2
```

This processing makes the direct indexing less efficient than the relative indexing in `ELEMENT (I + 5, J - 3, K + 2)`.

Related concepts
"Optimization" on page 450

Related tasks
"Handling tables efficiently" on page 447

Optimizing your code

When your program is ready for final testing, specify the `OPTIMIZE` compiler option so that the tested code and the production code are identical. Note that IBM recommends that all users use `OPT(FULL)` for the best performance.

If you frequently run a program without recompiling it during development, you might also want to use `OPTIMIZE`. However, if you recompile frequently, the overhead for `OPTIMIZE` might outweigh its benefits unless you are using the assembler language expansion (`LIST` compiler option) to fine-tune the program.

For unit-testing a program, you will probably find it easier to debug code that has not been optimized.

To see how the optimizer works on a program, compile it with and without using `OPTIMIZE` and compare the generated code. (Use the `LIST` compiler option to request the assembler listing of the generated code.)

Related concepts
"Optimization" on page 450

Related references
"LIST" on page 268
"OPTIMIZE" on page 273

Optimization

To improve the efficiency of the generated code, you can use the `OPTIMIZE` compiler option.

`OPTIMIZE` causes the COBOL optimizer to do the following optimizations:

- Eliminate unnecessary transfers of control and inefficient branches, including those generated by the compiler that are not evident from looking at the source program.
- Where possible, the optimizer places the statements inline, eliminating the need for linkage code. This optimization is known as *procedure integration*.
- Eliminate duplicate computations (such as subscript computations and repeated statements) that have no effect on the results of the program.
- Eliminate constant computations by performing them when the program is compiled.
- Eliminate constant conditional expressions.
- Aggregate moves of contiguous items (such as those that often occur with the use of `MOVE CORRESPONDING`) into a single move. Both the source and target must be contiguous for the moves to be aggregated.
- Discard unreferenced data items from the `DATA DIVISION`, and suppress generation of code to initialize these data items to their `VALUE` clauses. (The optimizer takes this action only when you use the `FULL` suboption.)

Choosing compiler features to enhance performance

Your choice of performance-related compiler options and your use of the `USE FOR DEBUGGING ON ALL PROCEDURES` statement can affect how well your program is optimized.

You might have a customized system that requires certain options for optimum performance. Do these steps:

1. To see what your system defaults are, get a short listing for any program and review the listed option settings.
2. Select performance-related options for compiling your programs.

   Important: Confer with your system programmer about how to tune COBOL programs. Doing so will ensure that the options you choose are appropriate for programs at your site.

Another compiler feature to consider is the `USE FOR DEBUGGING ON ALL PROCEDURES` statement. It can greatly affect the compiler optimizer. The `ON ALL PROCEDURES` option generates extra code at each transfer to a procedure name. Although very useful for debugging, it can make the program significantly larger and inhibit optimization substantially.

Related concepts
"Optimization" on page 450

Related tasks
"Optimizing your code" on page 450
"Getting listings" on page 309

Related references
"Performance-related compiler options" on page 451

Performance-related compiler options

In the table below you can see a description of the purpose of each option, its performance advantages and disadvantages, and usage notes where applicable.

Table 48. ***Performance-related compiler options***

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
`ARITH(EXTEND)` (see "ARITH" on page 251)	To increase the maximum number of digits allowed for decimal numbers	None	`ARITH(EXTEND)` causes some degradation in performance for all decimal data types because of larger intermediate results.	The amount of degradation that you experience depends directly on the amount of decimal data that you use.
`DYNAM` (see "DYNAM" on page 262)	To have subprograms (called through the `CALL` statement) dynamically loaded at run time	Subprograms are easier to maintain, because the application does not have to be link-edited again if a subprogram is changed.	There is a slight performance penalty, because the call must go through a library routine.	To free virtual storage that is no longer needed, issue the `CANCEL` statement.
`OPTIMIZE(STD)` (see "OPTIMIZE" on page 273)	To optimize generated code for better performance	Generally results in more efficient runtime code	Longer compile time: `OPTIMIZE` requires more processing time for compiles than `NOOPTIMIZE`.	`NOOPTIMIZE` is generally used during program development when frequent compiles are needed; it also allows for symbolic debugging. For production runs, `OPTIMIZE` is recommended.
`OPTIMIZE(FULL)`	To optimize generated code for better performance and also optimize the `DATA DIVISION`	Generally results in more efficient runtime code and less storage usage	Longer compile time: `OPTIMIZE` requires more processing time for compiles than `NOOPTIMIZE`.	`OPT(FULL)` deletes unused data items, which might be undesirable in the case of time stamps or data items that are used only as markers for dump reading.
`SSRANGE` (see "SSRANGE" on page 281)	To verify that all table references and reference modification expressions are in proper bounds	`SSRANGE` generates additional code for verifying table references. Using `NOSSRANGE` causes that code not to be generated.	With `SSRANGE` specified, checks for valid ranges do affect compiler performance.	In general, if you need to verify the table references only a few times instead of at every reference, coding your own checks might be faster than using `SSRANGE`. You can turn off `SSRANGE` at run time by using the `CHECK(OFF)` runtime option. For performance-sensitive applications, `NOSSRANGE` is recommended.

Table 48. **Performance-related compiler options** *(continued)*				
Compiler option	**Purpose**	**Performance advantages**	**Performance disadvantages**	**Usage notes**
NOTEST (see "TEST" on page 283)	To avoid the additional object code that is needed to take full advantage of the debugger. **Note:** COBOL for Linux does not currently provide a debugger. Use the NOTEST option for now.	None	TEST significantly enlarges the object file because it adds debugging information. When linking the program, you can direct the linker to exclude the debugging information, resulting in approximately the same size executable as would be created if the modules were compiled with NOTEST. If the debugging information is included, a slight performance degradation might occur because a larger executable takes longer to load and could increase paging.	For production runs, using NOTEST is recommended.
TRUNC(OPT) (see "TRUNC" on page 283)	To avoid having code generated to truncate the receiving fields of arithmetic operations	Does not generate extra code and generally improves performance	Both TRUNC(BIN) and TRUNC(STD) generate extra code whenever a BINARY data item is changed. TRUNC(BIN) is usually the slowest of these options.	TRUNC(STD) conforms to the 85 COBOL Standard, but TRUNC(BIN) and TRUNC(OPT) do not. With TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. TRUNC(OPT) is recommended where possible.

Related concepts
"Optimization" on page 450

Related tasks
"Generating a list of compiler messages" on page 228
"Choosing compiler features to enhance performance" on page 450
"Handling tables efficiently" on page 447

Related references
"Sign representation of zoned and packed-decimal data" on page 47
"Compiler options" on page 245

Evaluating performance

Fill in the following worksheet to help you evaluate the performance of your program. If you answer yes to each question, you are probably improving the performance.

In thinking about the performance tradeoff, be sure you understand the function of each option as well as the performance advantages and disadvantages. You might prefer function over increased performance in many instances.

Table 49. **Performance-tuning worksheet**		
Compiler option	**Consideration**	**Yes?**
DYNAM	Can you use NODYNAM? Consider the performance tradeoffs.	

*Table 49. **Performance-tuning worksheet** (continued)*		
Compiler option	**Consideration**	**Yes?**
OPTIMIZE	Do you use OPTIMIZE for production runs? Can you use OPTIMIZE(FULL)?	
SSRANGE	Do you use NOSSRANGE for production runs?	
TEST	Do you use NOTEST for production runs?	
TRUNC	Do you use TRUNC(OPT) when possible?	

Related tasks
"Choosing compiler features to enhance performance" on page 450

Related references
"Performance-related compiler options" on page 451

Chapter 28. Simplifying coding

You can use coding techniques to improve your productivity. By using the COPY statement, COBOL intrinsic functions, and callable services, you can avoid repetitive coding and having to code many arithmetic calculations or other complex tasks.

If your program contains frequently used code sequences (such as blocks of common data items, input-output routines, error routines, or even entire COBOL programs), write the code sequences once and put them in a COBOL copy library. You can use the COPY statement to retrieve these code sequences and have them included in your program at compile time. Using copybooks in this manner eliminates repetitive coding.

COBOL provides various capabilities for manipulating strings and numbers. These capabilities can help you simplify your coding.

The date and time callable services store dates as fullword binary integers and store time stamps as long (64-bit) floating-point values. These formats let you do arithmetic calculations on date and time values simply and efficiently. You do not need to write special subroutines that use services outside the language library to perform such calculations.

Related tasks
"Using numeric intrinsic functions" on page 50
"Eliminating repetitive coding" on page 455
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 106
"Manipulating dates and times" on page 457

Eliminating repetitive coding

To include stored source statements in a program, use the COPY statement in any program division and at any code sequence level. You can nest COPY statements to any depth.

To specify more than one copy library, either set the environment variable SYSLIB to multiple path names separated by a colon (:), or define your own environment variables and include the following phrase in the COPY statement:

```
IN/OF library-name
```

For example:

```
COPY MEMBER1 OF COPYLIB
```

If you omit this qualifying phrase, the default is SYSLIB.

COPY and debugging line: In order for the text copied to be treated as debug lines, for example, as if there were a D inserted in column 7, put the D on the first line of the COPY statement. A COPY statement cannot itself be a debugging line; if it contains a D, and `WITH DEBUGGING` mode is not specified, the COPY statement is nevertheless processed.

"Example: using the COPY statement" on page 456

© Copyright IBM Corp. 2021

Related references
Chapter 14, “Compiler-directing statements,” on page 291

Example: using the COPY statement

These examples show how you can use the COPY statement to include library text in a program.

Suppose the library entry CFILEA consists of the following FD entries:

```
    BLOCK CONTAINS 20 RECORDS
    RECORD CONTAINS 120 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS FILE-OUT.
01  FILE-OUT       PIC X(120).
```

You can retrieve the text-name CFILEA by using the COPY statement in a source program as follows:

```
FD FILEA
         COPY CFILEA.
```

The library entry is copied into your program, and the resulting program listing looks like this:

```
FD FILEA
         COPY CFILEA.
C    BLOCK CONTAINS 20 RECORDS
C    RECORD CONTAINS 120 CHARACTERS
C    LABEL RECORDS ARE STANDARD
C    DATA RECORD IS FILE-OUT.
C    01  FILE-OUT   PIC X(120).
```

In the compiler source listing, the COPY statement prints on a separate line. C precedes copied lines.

Assume that a copybook with the text-name DOWORK is stored by using the following statements:

```
COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND
MOVE QTY-ON-HAND to PRINT-AREA
```

To retrieve the copybook identified as DOWORK, code:

```
paragraph-name.
    COPY DOWORK.
```

The statements that are in the DOWORK procedure will follow *paragraph-name*.

If you use the EXIT compiler option to provide a LIBEXIT module, your results might differ from those shown here.

Related tasks
“Eliminating repetitive coding” on page 455

Related references
Chapter 14, “Compiler-directing statements,” on page 291

Manipulating dates and times

To invoke a date or time callable service, use a CALL statement with the correct parameters for that service. You define the data items for the CALL statement in the DATA DIVISION with the data definitions required by that service.

```
77  argument          pic  s9(9)  comp.
01  format.
    05 format-length  pic  s9(4)  comp.
    05 format-string  pic  x(80).
77  result            pic  x(80).
77  feedback-code     pic  x(12)  display.
. . .
    CALL "CEEDATE" using argument, format, result, feedback-code.
```

In the example above, the callable service CEEDATE converts a number that represents a Lilian date in the data item `argument` to a date in character format, which is written to the data item `result`. The picture string contained in the data item `format` controls the format of the conversion. Information about the success or failure of the call is returned in the data item `feedback-code`.

In the CALL statements that you use to invoke the date and time callable services, you must use a literal for the program-name rather than an identifier.

A program calls the date and time callable services by using the standard system linkage convention.

"Example: manipulating dates" on page 458

Related concepts
Appendix D, "Date and time callable services," on page 487

Related tasks
"Getting feedback from date and time callable services" on page 457
"Handling conditions from date and time callable services" on page 458

Related references
"Feedback token" on page 459
"Picture character terms and strings" on page 460
CALL statement (*COBOL for Linux on x86 Language Reference*)

Getting feedback from date and time callable services

You can specify a feedback code parameter (which is optional) in any date and time callable service. Specify OMITTED for this parameter if you do not want the service to return information about the success or failure of the call.

However, if you do not specify this parameter and the callable service does not complete successfully, the program will abend.

When you call a date and time callable service and specify OMITTED for the feedback code, the RETURN-CODE special register is set to 0 if the service is successful, but it is not altered if the service is unsuccessful. If the feedback code is not OMITTED, the RETURN-CODE special register is always set to 0 regardless of whether the service completed successfully.

"Example: formatting dates for output" on page 458

Related references
"Feedback token" on page 459

Handling conditions from date and time callable services

Condition handling by COBOL for Linux is significantly different from that provided by IBM Language Environment on the host. COBOL for Linux adheres to the native COBOL condition handling scheme and does not provide the level of support that is in Language Environment.

If you pass a feedback token an argument, it will simply be returned after the appropriate information has been filled in. You can code logic in the calling routine to examine the contents and perform any actions if necessary. The condition will not be signaled.

Related references
"Feedback token" on page 459

Example: manipulating dates

The following example shows how to use date and time callable services to convert a date to a different format and do a simple calculation with the formatted date.

```
CALL CEEDAYS USING dateof_hire, 'YYMMDD', doh_lilian, fc.
CALL CEELOCT USING todayLilian, today_seconds, today_Gregorian, fc.
COMPUTE servicedays  = today_Lilian - doh_Lilian.
COMPUTE serviceyears = service_days / 365.25.
```

The example above uses the original date of hire in the format YYMMDD to calculate the number of years of service for an employee. The calculation is as follows:

1. Call CEEDAYS (Convert Date to Lilian Format) to convert the date to Lilian format.
2. Call CEELOCT (Get Current Local Time) to get the current local time.
3. Subtract doh_Lilian from today_Lilian (the number of days from the beginning of the Gregorian calendar to the current local time) to calculate the employee's number of days of employment.
4. Divide the number of days by 365.25 to get the number of service years.

Example: formatting dates for output

The following example uses date and time callable services to format and display a date obtained from an ACCEPT statement.

Many callable services offer capabilities that would otherwise require extensive coding. Two such services are CEEDAYS and CEEDATE, which you can use effectively when you want to format dates.

```
CBL QUOTE
       ID DIVISION.
       PROGRAM-ID. HOHOHO.
      *************************************************************
      * FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
      *            WWWWWWWWW, MMMMMMMM DD, YYYY                  *
      *                                                          *
      *            For example:  MONDAY, OCTOBER 18, 2010        *
      *                                                          *
      *************************************************************
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

       01   CHRDATE.
            05 CHRDATE-LENGTH     PIC S9(4) COMP VALUE 10.
            05 CHRDATE-STRING     PIC X(10).
       01   PICSTR.
            05 PICSTR-LENGTH      PIC  S9(4) COMP.
            05 PICSTR-STRING      PIC  X(80).

       77   LILIAN PIC           S9(9) COMP.
       77   FORMATTED-DATE       PIC X(80).

       PROCEDURE DIVISION.
      ************************************************************
      *    USE  DATE/TIME CALLABLE SERVICES TO PRINT OUT          *
```

```
      *    TODAY'S DATE FROM COBOL ACCEPT STATEMENT.               *
      ************************************************************
           ACCEPT CHRDATE-STRING FROM DATE.

           MOVE "YYMMDD" TO PICSTR-STRING.
           MOVE 6 TO PICSTR-LENGTH.
           CALL "CEEDAYS" USING CHRDATE , PICSTR , LILIAN , OMITTED.

           MOVE " WWWWWWWWWZ, MMMMMMMMMZ DD, YYYY " TO PICSTR-STRING.
           MOVE 50 TO PICSTR-LENGTH.
           CALL "CEEDATE" USING LILIAN , PICSTR , FORMATTED-DATE ,
                 OMITTED.

           DISPLAY "******************************".
           DISPLAY FORMATTED-DATE.
           DISPLAY "******************************".

           STOP RUN.
```

Feedback token

A feedback token contains feedback information in the form of a condition token. The condition token set by the callable service is returned to the calling routine, indicating whether the service completed successfully.

COBOL for Linux uses the same feedback token as Language Environment, which is defined as follows:

```
01  FC.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) COMP.
            04  Msg-No      PIC S9(4) COMP.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) COMP.
            04  Cause-Code  PIC S9(4) COMP.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) COMP.
```

The contents of each field and the differences from IBM Language Environment on the host are as follows:

Severity
This is the severity number with the following possible values:

0
Information only (or, if the entire token is zero, no information)

1
Warning: service completed, probably correctly

2
Error detected: correction was attempted; service completed, perhaps incorrectly

3
Severe error: service did not complete

4
Critical error: service did not complete

Msg-No
This is the associated message number.

Case-Sev-Ctl
This field always contains the value 1.

Facility-ID
This field always contains the characters CEE.

I-S-Info
This field always contains the value 0.

The sample copybook CEEIGZCT.CPY defines the condition tokens. The condition tokens in the file are equivalent to those provided by Language Environment, except that character representations are in ASCII instead of EBCDIC. You must take these differences into account if you compare the condition tokens with those provided by Language Environment.

The descriptions of the individual callable services include a listing of the symbolic feedback codes that might be returned in the feedback code output field specified on invocation of the service. In addition to these, the symbolic feedback code CEE0PD might be returned for any callable service. See message IWZ0813S for details.

All date and time callable services are based on the Gregorian calendar. Date variables associated with this calendar have architectural limits. These limits are:

Starting Lilian date
The beginning of the Lilian date range is Friday 15 October 1582, the date of adoption of the Gregorian calendar. Lilian dates before this date are undefined. Therefore:

- Day zero is 00:00:00 14 October 1582.
- Day one is 00:00:00 15 October 1582.

All valid input dates must be after 00:00:00 15 October 1582.

End Lilian date
The end Lilian date is set to 31 December 9999. Lilian dates after this date are undefined because 9999 is the highest possible four-digit year.

Related references
Appendix G, “Runtime messages,” on page 549

Picture character terms and strings

You use *picture strings* (templates that indicate the format of the input data or the required format of the output data) for several of the date and time callable services.

Table 50. ***Picture character terms and strings***

Picture terms	Explanations	Valid values	Notes®
Y	One-digit year	0-9	Y valid for output only. YY assumes range set by CEESCEN. YYY/ZYY used with <JJJJ>, <CCCC>, and <CCCCCCCC>.
YY	Two-digit year	00-99	
YYY	Three-digit year	000-999	
ZYY	Three-digit year within era	1-999	
YYYY	Four-digit year	1582-9999	
<JJJJ>	Japanese Era name in Kanji characters with UTF-16 hexadecimal encoding	Reiwa (NX'E44E8C54')	Affects YY field: if <JJJJ> is specified, YY means the year within Japanese Era. For example, 1988 equals Showa 63.
		Heisei (NX'735E1062')	
		Showa (NX'2D668C54')	
		Taisho (NX'2759636B')	
		Meiji (NX'0E66BB6C')	
MM	Two-digit month	01-12	For output, leading zero suppressed. For input, ZM treated as MM.
ZM	One- or two-digit month	1-12	

Table 50. ***Picture character terms and strings*** *(continued)*

Picture terms	Explanations	Valid values	Notes®
RRRR	Roman numeral month	I*bbb*-XII*b* (left justified)	For input, source string is folded to uppercase. For output, uppercase only. I=Jan, II=Feb, ..., XII=Dec.
RRRZ			
MMM	Three-character month, uppercase	JAN-DEC	For input, source string always folded to uppercase. For output, M generates uppercase and m generates lowercase. Output is padded with blanks (*b*) (unless Z specified) or truncated to match the number of Ms, up to 20.
Mmm	Three-character month, mixed case	Jan-Dec	
MMMM...M	3–20-character month, uppercase	JANUARY*bb*-DECEMBER*b*	
Mmmm...m	3–20-character month, mixed case	January*bb*-December*b*	
MMMMMMMMMMZ	Trailing blanks suppressed	JANUARY-DECEMBER	
Mmmmmmmmmmz	Trailing blanks suppressed	January-December	
DD	Two-digit day of month	01-31	For output, leading zero is always suppressed. For input, ZD treated as DD.
ZD	One- or two-digit day of month	1-31	
DDD	Day of year (Julian day)	001-366	
HH	Two-digit hour	00-23	For output, leading zero suppressed. For input, ZH treated as HH. If AP specified, valid values are 01-12.
ZH	One- or two-digit hour	0-23	
MI	Minute	00-59	
SS	Second		
9	Tenths of a second	0-9	No rounding
99	Hundredths of a second	00-99	
999	Thousandths of a second	000-999	
AP	AM/PM indicator	AM or PM	AP affects HH/ZH field. For input, source string always folded to uppercase. For output, AP generates uppercase and ap generates lowercase.
ap		am or pm	
A.P.		A.M. or P.M.	
a.p.		a.m. or p.m.	
W	One-character day-of-week	S, M, T, W, T, F, S	For input, Ws are ignored. For output, W generates uppercase and w generates lowercase. Output padded with blanks (unless Z specified) or truncated to match the number of Ws, up to 20.
WWW	Three-character day, uppercase	SUN-SAT	
Www	Three-character day, mixed case	Sun-Sat	
WWW...W	3–20-character day, uppercase	SUNDAY*bbb*-SATURDAY*b*	
Www...w	3–20-character day, mixed case	Sunday*bbb*-Saturday*b*	
WWWWWWWWWZ	Trailing blanks suppressed	SUNDAY-SATURDAY	
Wwwwwwwwwz	Trailing blanks suppressed	Sunday-Saturday	

Table 50. ***Picture character terms and strings*** *(continued)*			
Picture terms	**Explanations**	**Valid values**	**Notes®**
All others	Delimiters	X'01'-X'FF' (X'00' is reserved for internal use by the date and time callable services.)	For input, treated as delimiters between the month, day, year, hour, minute, second, and fraction of a second. For output, copied exactly as is to the target string.
Note: Blank characters are indicated by the symbol *b*.			

The following table defines Japanese Eras used by date and time services when <JJJJ> is specified.

Table 51. ***Japanese Eras***			
First date of Japanese Era	**Era name**	**Era name in Kanji with UTF-16 hexadecimal encoding**	**Valid year values**
1868-09-08	Meiji	NX'0E66BB6C'	01-45
1912-07-30	Taisho	NX'2759636B'	01-15
1926-12-25	Showa	NX'2D668C54'	01-64
1989-01-08	Heisei	NX'735E1062'	01-31
2019-05-01	Reiwa	NX'E44E8C54'	01-999 (01 = 2019)

"Example: date-and-time picture strings" on page 462

Example: date-and-time picture strings

These are examples of picture strings that are recognized by the date and time services.

Table 52. ***Examples of date-and-time picture strings***		
Picture strings	**Examples**	**Comments**
YYMMDD	880516	
YYYYMMDD	19880516	
YYYY-MM-DD	1988-05-16	1988-5-16 would also be valid input.
<JJJJ> YY.MM.DD	Showa 63.05.16	*Showa* is a Japanese Era name. Showa 63 equals 1988.
MMDDYY	050688	One-digit year format (Y) is valid for output only.
MM/DD/YY	05/06/88	
ZM/ZD/YY	5/6/88	
MM/DD/YYYY	05/06/1988	
MM/DD/Y	05/06/8	

Table 52. **Examples of date-and-time picture strings** *(continued)*		
Picture strings	**Examples**	**Comments**
DD.MM.YY	09.06.88	Z suppresses zeros and blanks.
DD-RRRR-YY	09-VI -88	
DD MMM YY	09 JUN 88	
DD Mmmmmmmmm YY	09 June 88	
ZD Mmmmmmmmmz YY	9 June 88	
Mmmmmmmmmz ZD, YYYY	June 9, 1988	
ZDMMMMMMMMzYY	9JUNE88	
YY.DDD	88.137	Julian date
YYDDD	88137	
YYYY/DDD	1988/137	
YYMMDDHHMISS	880516204229	Time stamp: valid only for CEESECS and CEEDATM. If used with CEEDATE, time positions are filled with zeros. If used with CEEDAYS, HH, MI, SS, and 999 fields are ignored.
YYYYMMDDHHMISS	19880516204229	
YYYY-MM-DD HH:MI:SS.999	1988-05-16 20:42:29.046	
WWW, ZM/ZD/YY HH:MI AP	MON, 5/16/88 08:42 PM	
Wwwwwwwwwz, DD Mmm YYYY, ZH:MI AP	Monday, 16 May 1988, 8:42 PM	
Note: Lowercase characters can be used only for alphabetic picture terms.		

Century window

To process two-digit years in the year 2000 and beyond, the date and time callable services use a sliding scheme in which all two-digit years are assumed to lie within a 100-year interval (the *century window*) that starts 80 years before the current system date.

In the year 2010 for example, the 100 years that span from 1930 to 2029 are the default century window for the date and time callable services. Thus in 2010, years 30 through 99 are recognized as 1930-1999, and years 00 through 29 are recognized as 2000-2029.

By year 2080, all two-digit years will be recognized as 20*nn*. In 2081, 00 will be recognized as year 2100.

Some applications might need to set up a different 100-year interval. For example, banks often deal with 30-year bonds, which could be due 01/31/30. The two-digit year 30 would be recognized as the year 1930 if the century window described above were in effect.

The CEESCEN callable service lets you change the century window. A companion service, CEEQCEN, queries the current century window.

You can use CEEQCEN and CEESCEN, for example, to cause a subroutine to use a different interval for date processing than that used by its parent routine. Before returning, the subroutine should reset the interval to its previous value.

"Example: querying and changing the century window" on page 464

Example: querying and changing the century window

The following example shows how to query, set, and restore the starting point of the century window using the CEEQCEN and CEESCEN services.

The example calls CEEQCEN to obtain an integer (OLDCEN) that indicates how many years earlier the current century window began. It then temporarily changes the starting point of the current century window to a new value (TEMPCEN) by calling CEESCEN with that value. Because the century window is set to 30, any two-digit years that follow the CEESCEN call are assumed to lie within the 100-year interval starting 30 years before the current system date.

Finally, after it processes dates (not shown) using the temporary century window, the example again calls CEESCEN to reset the starting point of the century window to its original value.

```
 WORKING-STORAGE SECTION.
 77 OLDCEN  PIC S9(9) COMP.
 77 TEMPCEN PIC S9(9) COMP.
 77 QCENFC  PIC X(12).
 . . .
 77 SCENFC1 PIC X(12).
 77 SCENFC2 PIC X(12).
 . . .
 PROCEDURE DIVISION.
     . . .
** Call CEEQCEN to retrieve and save current century window
     CALL "CEEQCEN" USING OLDCEN, QCENFC.
** Call CEESCEN to temporarily change century window to 30
     MOVE 30 TO TEMPCEN.
     CALL "CEESCEN" USING TEMPCEN, SCENFC1.
** Perform date processing with two-digit years
     . . .
** Call CEESCEN again to reset century window
     CALL "CEESCEN" USING OLDCEN, SCENFC2.
     . . .
     GOBACK.
```

Related references

Appendix D, “Date and time callable services,” on page 487

Using the format 2 SORT statement to sort a table

It is recommended to use the format 2 SORT statement to sort a table. It provides the following benefits when compared to the format 1 SORT statement.

Table 53. Comparison of format 1 and format 2 SORT statements

Characteristics	Format 1 SORT statements	Format 2 SORT statements
Can be used to sort a file or a table	Yes	No, it is for tables only
Requires DFSORT or equivalent sorting program	Yes	No
Supported in CICS	Limited	Yes
Supported in UNIX System Services	No	Yes
Table can be sorted by using a single SORT statement, which simplifies coding	No, it requires the SELECT clauses, SD entries with record descriptions, and input and output procedures	Yes

Table 53. Comparison of format 1 and format 2 SORT statements (continued)		
Characteristics	**Format 1 SORT statements**	**Format 2 SORT statements**
Keys for sorting can be specified as part of the table definition, which can also be used in the SEARCH ALL statement	No, keys must be specified in the SORT statement. If the table is to be searched by using SEARCH ALL as well, the keys must also be redundantly specified as part of the table definition.	Yes, and it also supports specifying keys in the SORT statement if needed
Can filter or preprocess table elements during the sorting process	Yes, using input and output procedures	No, all of the table elements are passed to SORT as-is
Uses special registers that include SORT-CONTROL, SORT-CORE-SIZE, SORT-FILE-SIZE, SORT-MESSAGE, SORT-MODE-SIZE, and SORT-RETURN	Yes	No
Can be executed within the range of an input or output procedure	No	Yes

Note: Do not use the format 2 SORT with large tables in an environment where storage is constrained, because the format 2 SORT uses heap storage to do the sort.

Related references
SORT statement (*COBOL for Linux on x86 Language Reference*)

Appendix A. Summary of differences from IBM Enterprise COBOL for z/OS

COBOL for Linux on x86 implements certain items differently from the way that Enterprise COBOL for z/OS implements them. See the Related references below for details.

Related tasks
Chapter 21, “Porting applications between platforms,” on page 375

Related references
“Compiler options” on page 467
“Data representation” on page 467
“Runtime environment variables” on page 469
“File specification” on page 469
“Interlanguage communication (ILC)” on page 470
“Input and output” on page 470
“Runtime options” on page 471
“Source code line size” on page 471
“Language elements” on page 471

Compiler options

COBOL for Linux on x86 does not support some Enterprise COBOL compiler options.

These unsupported options are as follows:

ADATA, ADV, AFP, ARCH, AWO, BLOCK0, BUFSIZE, CODEPAGE, COPYLOC, COPYRIGHT, DATA, DBCS, DECK, DISPSIGN, DLL, DUMP, EXPORTALL, FASTSRT, HGPR, INITCHECK, INITIAL, INLINE, INTDATE, LANGUAGE, LP, MAXPCF, NAME, NUMCHECK, NUMPROC, OBJECT, OFFSET, OPTFILE, OUTDD, PARMCHECK, QUALIFY, RENT, RMODE, RULES, SERVICE, SQLCCSID, SQLIMS, STGOPT, SUPPRESS, THREAD, VLR, VSAMOPENFS, WORD, XMLPARSE, ZONECHECK, and ZONEDATA

Related tasks
“Getting IBM Enterprise COBOL for z/OS applications to compile” on page 375

Related references
“cob2 options” on page 230
“CHAR” on page 253

Data representation

The representation of data can differ between IBM Enterprise COBOL and COBOL for Linux on x86.

Binary data

By default, COBOL for Linux on x86 uses little-endian format and Enterprise COBOL uses big-endian format for binary data.

Zoned decimal data

Sign representation for zoned decimal data is based on ASCII or EBCDIC depending on the setting of the CHAR compiler option (NATIVE or EBCDIC) and whether the USAGE clause is specified with the NATIVE

© Copyright IBM Corp. 2021

phrase. COBOL for Linux on x86 processes the sign representation of zoned decimal data consistently with the processing that occurs on z/OS when the compiler option NUMPROC(NOPFD) is in effect.

Packed-decimal data

Sign representation for unsigned packed-decimal numbers is different between COBOL for Linux on x86 and Enterprise COBOL. COBOL for Linux on x86 always uses a sign nibble of x'C' for unsigned packed-decimal numbers. Enterprise COBOL uses a sign nibble of x'F' for unsigned packed-decimal numbers. If you are going to share data files that contain packed-decimal numbers between Linux and z/OS, it is recommended that you use signed packed-decimal numbers instead of unsigned packed-decimal numbers.

Display floating-point data

You can use the FLOAT(BE) compiler option to indicate that display floating-point data items are in the IBM Z data representation (hexadecimal) as opposed to the native (IEEE) format.

Do not specify the IBM Z formats of display floating-point items as arguments on INVOKE statements or as method parameters. You must specify these arguments and parameters in the native formats in order for them to be interoperable with the Java data types.

National data

By default, COBOL for Linux on x86 uses UTF-16 little-endian format, and Enterprise COBOL uses UTF-16 big-endian format for national data.

EBCDIC and ASCII data

You can specify the EBCDIC collating sequence for alphanumeric data items using the following language elements:

- ALPHABET clause
- PROGRAM COLLATING SEQUENCE clause
- COLLATING SEQUENCE phrase of the SORT or MERGE statement

You can specify the CHAR(EBCDIC) compiler option to indicate that DISPLAY data items are in the IBM Z data representation (EBCDIC).

Code-page determination for data conversion

For alphabetic, alphanumeric, DBCS, and national data items, the source code page used for implicit conversion of native characters is determined from the locale in effect at run time.

For alphanumeric, DBCS, and national literals, the source code page used for implicit conversion of characters is determined from the locale in effect at compile time.

DBCS character strings

Under COBOL for Linux on x86, ASCII DBCS character strings are not delimited with the shift-in and shift-out characters except possibly with the dummy shift-in and shift-out characters as discussed below.

Use the SOSI compiler option to indicate that Linux workstation shift-out (X'1E') and shift-in (X'1F') control characters delimit DBCS character strings in the source program, including user-defined words, DBCS literals, alphanumeric literals, national literals, and comments. Host shift-out and shift-in control characters (X'0E' and X'0F', respectively) are usually converted to workstation shift-out and shift-in control characters when COBOL for Linux on x86 source code is downloaded, depending on the download method that you use.

Using control characters X'00' through X'1F' within an alphanumeric literal can yield unpredictable results.

Related tasks
Chapter 11, “Setting the locale,” on page 199
“Fixing differences caused
by data representations” on page 376

Related references
“CHAR” on page 253
“SOSI” on page 277

Runtime environment variables

COBOL for Linux on x86 recognizes several runtime environment variables that are not used in Enterprise COBOL, as listed below.

- CICS_TK_SFS_SERVER
- COBPATH
- COBRTOPT
- EBCDIC_CODEPAGE
- CICS_SFS_DATA_VOLUME
- CICS_SFS_INDEX_VOLUME
- CICS_VSAM_AUTO_FLUSH
- CICS_VSAM_CACHE
- CICS_SFS_CACHE_<*filename*>
- CICS_SFS_RDM_CACHE
- CICS_SFS_PREALLOC_<*filename*>
- COBCORE
- COBOUTDIR
- PATH
- SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH

File specification

There are some differences between the way COBOL for Linux on x86 handles files and the way Enterprise COBOL handles files.

The differences between COBOL for Linux on x86 and Enterprise COBOL in file handling are in the following areas:

- Single-volume files
- Source-file suffixes
- Generation data groups (GDGs)
- File concatenation

Single-volume files: COBOL for Linux on x86 treats all files as single-volume files. All other file specifications are treated as comments. This difference affects the following items: `REEL`, `UNIT`, `MULTIPLE FILE TAPE` clause, and `CLOSE. . .UNIT/REEL`.

Source-file suffixes: In COBOL for Linux on x86, when you compile using one of the `cob2` commands, COBOL source files that either have suffix .cbl or .cob are passed to the compiler. In Enterprise COBOL, when you compile in the z/OS UNIX file system, only files that have suffix .cbl are passed to the compiler.

Generation data groups (GDGs): GDG support is almost identical to GDG support in Enterprise COBOL. However, there are differences in COBOL for Linux on x86:

- Generation data sets (GDSs), or *generation files* as they are referred to in this information, are supported for all file organizations and access modes in all the supported file systems.
- GDG support is not integrated into the file systems. A stand-alone utility, `gdgmgr`, is provided for creating and deleting GDGs, managing and querying GDG entries, performing limit processing, and reconciling the GDG catalog against the existing files.
- The resolution of generation file names occurs when the files are opened rather than at job initialization.
- Limit processing is done when a new generation is added to a group, rather than at job termination.
- The generational range within any given epoch is from 1 to 9999, inclusive, instead of being limited to 1000. Therefore generations 0001 and 9999 can exist in the same epoch.
- A group can contain 1000 generations instead of 255.
- Versioning is not supported. The automatically generated version is always `v00`.

File concatenation: In COBOL for Linux, you concatenate multiple files by separating the file identifiers with a colon (:). A COBOL file that is concatenated must have sequential or line-sequential organization, must be accessed sequentially, and can be opened only for input.

Related concepts
"Generation data groups" on page 123

Related tasks
"Concatenating files" on page 131
"Compiling from the command line" on page 223

Related references
"Limit processing of generation data groups" on page 130

Interlanguage communication (ILC)

ILC is available with C/C++ programs.

These are the differences in ILC behavior on Linux x86 compared to using ILC on z/OS with Language Environment:

- There are differences in termination behavior when a COBOL `STOP RUN` or a C `exit()` is used.
- There is no coordinated condition handling with COBOL for Linux. Avoid using a C `longjmp()` that crosses COBOL programs.
- With Enterprise COBOL, the first program that is invoked within the process and that is enabled for Language Environment is considered to be the "main" program. With COBOL for Linux, the first COBOL program invoked within the process is considered to be the main program by COBOL. This difference affects language semantics that are sensitive to the definition of the run unit (the execution unit that starts with a main program). For example, a `STOP RUN` results in the return of control to the invoker of the main program, which in a mixed-language environment might be different as stated above.

Related concepts
Chapter 25, "Preinitializing the COBOL runtime environment," on page 415

Input and output

COBOL for Linux on x86 supports input and output for sequential, relative, and indexed files with the Db2, SdU, SFS, and STL file systems, and also supports input and output for sequential files with the QSAM file system and RSD file system.

Line-sequential input and output is supported by the native byte stream file support of the operating system.

Sizes and values of the returned file status information can vary depending on which file system is used.

COBOL for Linux on x86 does not provide direct support for tape drives or diskette drives.

Related concepts
"File systems" on page 116
"Line-sequential file organization" on page 122

Related tasks
"Using file status keys" on page 166
"Using file system status codes" on page 168

Runtime options

COBOL for Linux on x86 does not recognize the following Enterprise COBOL runtime options, and treats them as not valid: `AIXBLD`, `ALL31`, `CBLPSHPOP`, `CBLQDA`, `COUNTRY`, `HEAP`, `MSGFILE`, `NATLANG`, `SIMVRD`, and `STACK`.

With Enterprise COBOL, you can use the `STORAGE` runtime option to initialize COBOL `WORKING-STORAGE`. With COBOL for Linux on x86, use the `WSCLEAR` compiler option.

Related references
"WSCLEAR" on page 286

Source code line size

In COBOL for Linux on x86, COBOL source lines can have varying lengths. A source line ends when a newline control character is encountered or when the maximum line length has been reached.

In Enterprise COBOL, each source line has the same length.

Related references
"SRCFORMAT" on page 280

Language elements

The following table lists language elements that are different between Enterprise COBOL and COBOL for Linux on x86 compilers, and where possible offers advice about how to handle such differences in COBOL for Linux on x86 programs.

Many COBOL clauses and phrases that are valid in Enterprise COBOL are syntax checked but have no effect on the execution of COBOL for Linux on x86 programs. These clauses and phrases should have minimal effect on existing applications that you download. COBOL for Linux on x86 recognizes most Enterprise COBOL language syntax even if that syntax has no functional effect.

Table 54. ***Language differences between Enterprise COBOL for z/OS and COBOL for Linux on x86***

Language element	**COBOL for Linux on x86 implementation or restriction**
`ACCEPT` statement	If your Enterprise COBOL program expects ddnames as the targets of `ACCEPT` statements, define these targets by using equivalent environment variables with values set to appropriate file-names. In COBOL for Linux on x86, *environment-name* and the associated environment-variable value, if set, determine file identification.
`APPLY WRITE-ONLY` clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
`ASSIGN` clause	COBOL for Linux on x86 uses a different syntax and mapping to the system file-name based on *assignment-name*. `ASSIGN. . .USING` *data-name* is not supported in Enterprise COBOL.
`BLOCK CONTAINS` clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86

Table 54. ***Language differences between Enterprise COBOL for z/OS and COBOL for Linux on x86*** *(continued)*	
Language element	**COBOL for Linux on x86 implementation or restriction**
CALL statement	A file-name as a CALL argument is not supported in COBOL for Linux on x86.
CLOSE statement	The following phrases are syntax checked, but have no effect on the execution of the program in COBOL for Linux on x86: FOR REMOVAL, WITH NO REWIND, and UNIT/REEL. Avoid use of these phrases in programs that are intended to be portable.
CODE-SET clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
DATA RECORDS clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
DISPLAY statement	If your Enterprise COBOL program expects ddnames as the targets of DISPLAY statements, define these targets by using equivalent environment variables with values set to appropriate file-names. In COBOL for Linux on x86, *environment-name* and the associated environment-variable value, if set, determine file identification.
DYNAMIC LENGTH clause	Dynamic-length elementary items are not currently supported in COBOL for Linux on x86.
File status *data-name-1*	Some values and meanings for file status 9*x* are different in Enterprise COBOL than in COBOL for Linux on x86.
File status *data-name-8*	The format and values are different depending on the platform and the file system.
INDEX data items	In Enterprise COBOL, INDEX data items are implicitly defined as 4 bytes. In COBOL for Linux on x86 programs compiled with ADDR(32), their size is 4 bytes; with ADDR(64), their size is 8 bytes.
JSON GENERATE and JSON PARSE statements	JSON is not currently supported in COBOL for Linux on x86.
LABEL RECORDS clause	The phrases LABEL RECORD IS *data-name*, USE. . .AFTER. . .LABEL PROCEDURE, and GO TO MORE-LABELS are syntax checked, but have no effect on the execution of the program in COBOL for Linux on x86. A warning is issued if you use any of these phrases. The user-label declaratives are not called at run time. You cannot port programs that depend on the user-label processing that z/OS QSAM supports.
MULTIPLE FILE TAPE	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86. On the Linux workstation, all files are treated as single-volume files.
OBJECT REFERENCE data items	OBJECT REFERENCE data items are not supported in COBOL for Linux on x86.
OPEN statement	The following phrases are syntax checked, but have no effect on the execution of the program in COBOL for Linux on x86: REVERSED and WITH NO REWIND.
PASSWORD clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
POINTER, PROCEDURE-POINTER, and FUNCTION-POINTER data items	In Enterprise COBOL, POINTER and FUNCTION-POINTER data items are implicitly defined as 4 bytes as is the special register ADDRESS OF; PROCEDURE-POINTER data items are implicitly defined as 8 bytes. In COBOL for Linux on x86 programs compiled with ADDR(32), the size of each of these items is 4 bytes; with ADDR(64), their size is 8 bytes.

Table 54. ***Language differences between Enterprise COBOL for z/OS and COBOL for Linux on x86*** *(continued)*	
Language element	**COBOL for Linux on x86 implementation or restriction**
READ. . .PREVIOUS	In COBOL for Linux on x86 only, allows you to read the previous record for relative or indexed files with DYNAMIC access mode
RECORD CONTAINS clause	The RECORD CONTAINS *n* CHARACTERS clause is accepted with one exception: RECORD CONTAINS 0 CHARACTERS is syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86.
RECORDING MODE clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86 for relative, indexed, and line-sequential files. RECORDING MODE U is syntax checked, but has no effect on the execution of the program for sequential files.
RERUN clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
RESERVE clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
SAME AREA clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
SAME SORT clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
SHIFT-IN, SHIFT-OUT special registers	The COBOL for Linux on x86 compiler puts out an E-level message if it encounters these registers unless the CHAR(EBCDIC) compiler option is in effect.
SORT-CONTROL special register	The implicit definition and contents of this special register differ between host and workstation COBOL.
SORT-CORE-SIZE special register	The contents of this special register differ between host and workstation COBOL.
SORT-FILE-SIZE special register	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86. Values in this special register are not used.
SORT-MESSAGE special register	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
SORT-MODE-SIZE special register	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86. Values in this special register are not used.
SORT MERGE AREA clause	Syntax checked, but has no effect on the execution of the program in COBOL for Linux on x86
START...	In COBOL for Linux on x86, the following relational operators are allowed: IS LESS THAN, IS <, IS NOT GREATER THAN, IS NOT >, IS LESS THAN OR EQUAL TO, IS <=.
STOP RUN	Not supported in COBOL for Linux on x86 multithreaded programs; can be replaced in a multithreaded program with a call to the C exit() function
UTF-8 phrase of the USAGE clause and the 'U' PICTURE symbol	The UTF-8 data class and UTF-8 data category are not currently supported in COBOL for Linux on x86.
WRITE statement	In COBOL for Linux on x86, if you specify WRITE. . .ADVANCING with environment names C01 through C12 or S01 through S05, one line is advanced.

Table 54. ***Language differences between Enterprise COBOL for z/OS and COBOL for Linux on x86*** *(continued)*	
Language element	**COBOL for Linux on x86 implementation or restriction**
XML PARSE statement	In Enterprise COBOL programs compiled using the host-only option XMLPARSE(XMLSS), additional syntax (the ENCODING phrase and RETURNING NATIONAL phrase) and special registers for namespace processing are available that are not available with COBOL for Linux on x86.
Names known to the platform environment	The following names are identified differently: *program-name*, *text-name*, *library-name*, *assignment-name*, file-name in the SORT-CONTROL special register, *basis-name*, DISPLAY or ACCEPT target identification, and system-dependent names.

Appendix B. IBM Z host data format considerations

The following information is about considerations, restrictions, and limitations that apply to the use of IBM Z host data internal representation.

The CHAR and FLOAT compiler options determine whether IBM Z host data format or native data format is used (other than for COMP-5 items or items defined with the NATIVE phrase in the USAGE clause). (The terms *host data format* and *native data format* in this information refer to the internal representation of data items.)

CICS access

There is no IBM Z host data format support for COBOL programs that are translated by the separate or integrated CICS translator and run on CICS TXSeries or CICS TX.

Date and time callable services

You can use the date and time callable services with the IBM Z host data format internal representations. All of the parameters passed to the callable services must be in IBM Z host data format. You cannot mix native and host data internal representations in the same call to a date and time service.

Floating-point overflow exceptions

Due to differences in the limits of floating-point data representations on the Linux workstation and the IBM Z host, it is possible if FLOAT(BE) is in effect that a floating-point overflow exception could occur during conversion between the two formats. For example, you might receive the following message on the workstation when you run a program that runs successfully on the host:

```
IWZ053S An overflow occurred on conversion to floating point
```

To avoid this problem, you must be aware of the maximum floating-point values supported on each platform for the respective data types. The limits are shown in the following table.

Table 55. ***Maximum floating-point values***

Data type	Maximum workstation value	Maximum IBM Z host value
COMP-1	*(2**128 - 2**4) (approx.*3.4028E+38)	*(16**63 - 16**57) (approx.*7.2370E+75)
COMP-2	*(2**1024 - 2**971) (approx.*1.7977E+308)	*(16**63 - 16**49) (approx.*7.2370E+75)

* Indicates that the value can be positive or negative.

As shown above, the host can carry a larger COMP-1 value than the workstation and the workstation can carry a larger COMP-2 value than the host.

Db2

The IBM Z host data format compiler options can be used with Db2 programs.

© Copyright IBM Corp. 2021

Distributed Computing Environment applications

The IBM Z host data format compiler options should not be used with Distributed Computing Environment programs.

File data

- EBCDIC data and hexadecimal binary data can be read from and written to any sequential, relative, or indexed files. No automatic conversion takes place.
- If you are accessing files that contain host data, use the compiler options `BINARY(BE)`, `COLLSEQ(EBCDIC)`, `CHAR(EBCDIC)`, and `FLOAT(BE)` to process binary data, EBCDIC character data and hexadecimal floating-point data that is acquired from these files.

SORT

All of the IBM Z host data formats except DBCS (`USAGE DISPLAY-1`) can be used as sort keys.

Related concepts
"Formats for numeric data" on page 39

Related references
"Compiler options" on page 245

Appendix C. Intermediate results and arithmetic precision

The compiler handles arithmetic statements as a succession of operations performed according to operator precedence, and sets up intermediate fields to contain the results of those operations. The compiler uses algorithms to determine the number of integer and decimal places to reserve.

Intermediate results are possible in the following cases:

- In an ADD or SUBTRACT statement that contains more than one operand immediately after the verb
- In a COMPUTE statement that specifies a series of arithmetic operations or multiple result fields
- In an arithmetic expression contained in a conditional statement or in a reference-modification specification
- In an ADD, SUBTRACT, MULTIPLY, or DIVIDE statement that uses the GIVING option and multiple result fields
- In a statement that uses an intrinsic function as an operand

"Example: calculation of intermediate results" on page 479

The precision of intermediate results depends on whether you compile using the default option ARITH(COMPAT) (referred to as *compatibility mode*) or using ARITH(EXTEND) (referred to as *extended mode*).

In compatibility mode, evaluation of arithmetic operations is unchanged from that in IBM COBOL Set for Linux:

- A maximum of 30 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return long-precision (64-bit) floating-point results.
- Expressions that contain floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that are not in floating point are converted to long-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to long-precision floating point for processing.

In extended mode, evaluation of arithmetic operations has the following characteristics:

- A maximum of 31 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return extended-precision (128-bit) floating-point results.
- Expressions that contain floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that are not in floating point are converted to extended-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to extended-precision floating point for processing.

Related concepts
"Formats for numeric data" on page 39
"Fixed-point contrasted with floating-point arithmetic" on page 53

Related references
"Fixed-point data and intermediate results" on page 479
"Floating-point data and intermediate results" on page 484
"Arithmetic expressions

© Copyright IBM Corp. 2021

in nonarithmetic statements” on page 485
“ARITH” on page 251

Terminology used for intermediate results

To understand this information about intermediate results, you need to understand the following terminology.

i
The number of integer places carried for an intermediate result. (If you use the ROUNDED phrase, one more integer place might be carried for accuracy if necessary.)

d
The number of decimal places carried for an intermediate result. (If you use the ROUNDED phrase, one more decimal place might be carried for accuracy if necessary.)

dmax
In a particular statement, the largest of the following items:

- The number of decimal places needed for the final result field or fields
- The maximum number of decimal places defined for any operand, except divisors or exponents
- The *outer-dmax* for any function operand

inner-dmax
In reference to a function, the largest of the following items:

- The number of decimal places defined for any of its elementary arguments
- The *dmax* for any of its arithmetic expression arguments
- The *outer-dmax* for any of its embedded functions

outer-dmax
The number of decimal places that a function result contributes to operations outside of its own evaluation (for example, if the function is an operand in an arithmetic expression, or an argument to another function).

op1
The first operand in a generated arithmetic statement (in division, the divisor).

op2
The second operand in a generated arithmetic statement (in division, the dividend).

i1 , i2
The number of integer places in *op1* and *op2*, respectively.

d1 , d2
The number of decimal places in *op1* and *op2*, respectively.

ir
The intermediate result when a generated arithmetic statement or operation is performed. (Intermediate results are generated either in registers or storage locations.)

ir1 , ir2
Successive intermediate results. (Successive intermediate results might have the same storage location.)

Related references
ROUNDED phrase (*COBOL for Linux on x86 Language Reference*)

Example: calculation of intermediate results

The following example shows how the compiler performs an arithmetic statement as a succession of operations, storing intermediate results as needed.

```
COMPUTE Y = A + B * C - D / E + F ** G
```

The result is calculated in the following order:

1. Exponentiate F by G yielding *ir1*.
2. Multiply B by C yielding *ir2*.
3. Divide E into D yielding *ir3*.
4. Add A to ir2 yielding *ir4*.
5. Subtract *ir3* from *ir4* yielding *ir5*.
6. Add *ir5* to *ir1* yielding Y.

Related tasks
"Using arithmetic expressions" on page 50

Related references
"Terminology used for intermediate results" on page 478

Fixed-point data and intermediate results

The compiler determines the number of integer and decimal places in an intermediate result.

Addition, subtraction, multiplication, and division

The following table shows the precision theoretically possible as the result of addition, subtraction, multiplication, or division.

Operation	Integer places	Decimal places
+ or -	(*i1* or *i2*) + 1, whichever is greater	*d1* or *d2*, whichever is greater
*	*i1* + *i2*	*d1* + *d2*
/	*i2* + *d1*	(*d2* - *d1*) or *dmax*, whichever is greater

You must define the operands of any arithmetic statements with enough decimal places to obtain the accuracy you want in the final result.

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *compatibility mode* (that is, when the default compiler option ARITH(COMPAT) is in effect):

Value of *i* + *d*	Value of *d*	Value of *i* + *dmax*	Number of places carried for *ir*
<30 or =30	Any value	Any value	*i* integer and *d* decimal places
>30	<*dmax* or =*dmax*	Any value	30-*d* integer and *d* decimal places
	>*dmax*	<30 or =30	*i* integer and 30-*i* decimal places
		>30	30-*dmax* integer and *dmax* decimal places

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *extended mode* (that is, when the compiler option `ARITH(EXTEND)` is in effect):

Value of *i* + *d*	**Value of *d***	**Value of *i* + *dmax***	**Number of places carried for *ir***
<31 or =31	Any value	Any value	*i* integer and *d* decimal places
>31	<*dmax* or =*dmax*	Any value	31-*d* integer and *d* decimal places
	>*dmax*	<31 or =31	*i* integer and 31-*i* decimal places
		>31	31-*dmax* integer and *dmax* decimal places

Exponentiation

Exponentiation is represented by the expression *op1* ** *op2*. Based on the characteristics of *op2*, the compiler handles exponentiation of fixed-point numbers in one of three ways:

- When *op2* is expressed with decimals, floating-point instructions are used.
- When *op2* is an integral literal or constant, the value *d* is computed as

```
d = d1 * |op2|
```

  and the value *i* is computed based on the characteristics of *op1*:

  – When *op1* is a data-name or variable,

```
i = i1 * |op2|
```

  – When *op1* is a literal or constant, *i* is set equal to the number of integers in the value of *op1* ** |*op2*|.

  In compatibility mode (compilation using `ARITH(COMPAT)`), the compiler having calculated *i* and *d* takes the action indicated in the table below to handle the intermediate results *ir* of the exponentiation.

Value of *i* + *d*	**Other conditions**	**Action taken**
<30	Any	*i* integer and *d* decimal places are carried for *ir*.
=30	*op1* has an odd number of digits.	*i* integer and *d* decimal places are carried for *ir*.
	op1 has an even number of digits.	Same action as when *op2* is an integral data-name or variable (shown below). Exception: for a 30-digit integer raised to the power of literal 1, *i* integer and *d* decimal places are carried for *ir*.
>30	Any	Same action as when *op2* is an integral data-name or variable (shown below)

  In extended mode (compilation using `ARITH(EXTEND)`), the compiler having calculated *i* and *d* takes the action indicated in the table below to handle the intermediate results *ir* of the exponentiation.

Value of *i* + *d*	**Other conditions**	**Action taken**
<31	Any	*i* integer and *d* decimal places are carried for *ir*.

Value of *i* + *d*	Other conditions	Action taken
=31 or >31	Any	Same action as when *op2* is an integral data-name or variable (shown below). Exception: for a 31-digit integer raised to the power of literal 1, *i* integer and *d* decimal places are carried for *ir*.

If *op2* is negative, the value of 1 is then divided by the result produced by the preliminary computation. The values of *i* and *d* that are used are calculated following the division rules for fixed-point data already shown above.

- When *op2* is an integral data-name or variable, *dmax* decimal places and 30-*dmax* (compatibility mode) or 31-*dmax* (extended mode) integer places are used. *op1* is multiplied by itself (|*op2*| - 1) times for nonzero *op2*.

  If *op2* is equal to 0, the result is 1. Division-by-0 and exponentiation SIZE ERROR conditions apply.

Fixed-point exponents with more than nine significant digits are always truncated to nine digits. If the exponent is a literal or constant, an E-level compiler diagnostic message is issued; otherwise, an informational message is issued at run time.

"Example: exponentiation in fixed-point arithmetic" on page 481

Related references

"Terminology used for intermediate results" on page 478
"Truncated intermediate results" on page 482
"Binary data and intermediate results" on page 482
"Floating-point data and intermediate results" on page 484
"Intrinsic functions evaluated in fixed-point arithmetic" on page 482
"ARITH" on page 251
SIZE ERROR phrases (*COBOL for Linux on x86 Language Reference*)

Example: exponentiation in fixed-point arithmetic

The following example shows how the compiler performs an exponentiation to a nonzero integer power as a succession of multiplications, storing intermediate results as needed.

```
COMPUTE Y = A ** B
```

If B is equal to 4, the result is computed as shown below. The values of *i* and *d* that are used are calculated according to the multiplication rules for fixed-point data and intermediate results (referred to below).

1. Multiply A by A yielding *ir1*.
2. Multiply *ir1* by A yielding *ir2*.
3. Multiply *ir2* by A yielding *ir3*.
4. Move *ir3* to *ir4*.

   ir4 has *dmax* decimal places. Because B is positive, *ir4* is moved to Y. If B were equal to -4, however, an additional fifth step would be performed:
5. Divide *ir4* into 1 yielding *ir5*.

ir5 has *dmax* decimal places, and would then be moved to Y.

Related references
"Terminology used for intermediate results" on page 478
"Fixed-point data and intermediate results" on page 479

Truncated intermediate results

Whenever the number of digits in an intermediate result exceeds 30 in compatibility mode or 31 in extended mode, the compiler truncates to 30 (compatibility mode) or 31 (extended mode) digits and issues a warning. If truncation occurs at run time, a message is issued and the program continues running.

If you want to avoid the truncation of intermediate results that can occur in fixed-point calculations, use floating-point operands (`COMP-1` or `COMP-2`) instead.

Related concepts
"Formats for numeric data" on page 39

Related references
"Fixed-point data and intermediate results" on page 479
"ARITH" on page 251

Binary data and intermediate results

If an operation that involves binary operands requires intermediate results longer than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the compiler converts the result from internal decimal to binary.

Binary operands are most efficient when intermediate results will not exceed nine digits.

Related references
"Fixed-point data and intermediate results" on page 479
"ARITH" on page 251

Intrinsic functions evaluated in fixed-point arithmetic

The compiler determines the *inner-dmax* and *outer-dmax* values for an intrinsic function from the characteristics of the function.

Integer functions

Integer intrinsic functions return an integer; thus their *outer-dmax* is always zero. For those integer functions whose arguments must all be integers, the *inner-dmax* is thus also always zero.

The following table summarizes the *inner-dmax* and the precision of the function result.

Function	*Inner-dmax*	Digit precision of function result
`DATE-OF-INTEGER`	0	8
`DATE-TO-YYYYMMDD`	0	8
`DAY-OF-INTEGER`	0	7
`DAY-TO-YYYYDDD`	0	7

Function	*Inner-dmax*	Digit precision of function result
FACTORIAL	0	30 in compatibility mode, 31 in extended mode
INTEGER-OF-DATE	0	7
INTEGER-OF-DAY	0	7
LENGTH	n/a	9
MOD	0	min(*i1 i2*)
ORD	n/a	3
ORD-MAX		9
ORD-MIN		9
YEAR-TO-YYYY	0	4
INTEGER		For a fixed-point argument: one more digit than in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.
INTEGER-PART		For a fixed-point argument: same number of digits as in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.

Mixed functions

A *mixed* intrinsic function is a function whose result type depends on the type of its arguments. A mixed function is fixed point if all of its arguments are numeric and none of its arguments is floating point. (If any argument of a mixed function is floating point, the function is evaluated with floating-point instructions and returns a floating-point result.) When a mixed function is evaluated with fixed-point arithmetic, the result is integer if all of the arguments are integer; otherwise, the result is fixed point.

For the mixed functions MAX, MIN, RANGE, REM, and SUM, the *outer-dmax* is always equal to the *inner-dmax* (and both are thus zero if all the arguments are integer). To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic and intermediate results (as referred to below) to each step in the algorithm.

MAX

1. Assign the first argument to the function result.
2. For each remaining argument, do the following steps:
   a. Compare the algebraic value of the function result with the argument.
   b. Assign the greater of the two to the function result.

MIN

1. Assign the first argument to the function result.
2. For each remaining argument, do the following steps:
   a. Compare the algebraic value of the function result with the argument.
   b. Assign the lesser of the two to the function result.

RANGE

1. Use the steps for MAX to select the maximum argument.
2. Use the steps for MIN to select the minimum argument.
3. Subtract the minimum argument from the maximum.
4. Assign the difference to the function result.

REM

1. Divide argument one by argument two.
2. Remove all noninteger digits from the result of step 1.
3. Multiply the result of step 2 by argument two.
4. Subtract the result of step 3 from argument one.
5. Assign the difference to the function result.

SUM

1. Assign the value 0 to the function result.
2. For each argument, do the following steps:
   a. Add the argument to the function result.
   b. Assign the sum to the function result.

Related references
"Terminology used for intermediate results" on page 478
"Fixed-point data and intermediate results" on page 479
"Floating-point data and intermediate results" on page 484
"ARITH" on page 251

Floating-point data and intermediate results

If any operation in an arithmetic expression is computed in floating-point arithmetic, the entire expression is computed as if all operands were converted to floating point and the operations were performed using floating-point instructions.

Floating-point instructions are used to compute an arithmetic expression if any of the following conditions is true of the expression:

- A receiver or operand is COMP-1, COMP-2, external floating point, or a floating-point literal.
- An exponent contains decimal places.
- An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.
- An intrinsic function is a floating-point function.

In compatibility mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are COMP-1 data items and the expression contains no multiplication or exponentiation operations.
- In all other cases, long precision is used.

Whenever long-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if long floating-point instructions were used.

In extended mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are COMP-1 data items and the expression contains no multiplication or exponentiation operations.
- Long precision is used if all receivers and operands are COMP-1 or COMP-2 data items, at least one receiver or operand is a COMP-2 data item, and the expression contains no multiplication or exponentiation operations.
- In all other cases, extended precision is used.

Whenever extended-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if extended-precision floating-point instructions were used.

Alert: If a floating-point operation has an intermediate result field in which exponent overflow occurs, the job is abnormally terminated.

Exponentiations evaluated in floating-point arithmetic

In compatibility mode, floating-point exponentiations are always evaluated using long floating-point arithmetic. In extended mode, floating-point exponentiations are always evaluated using extended-precision floating-point arithmetic.

The value of a negative number raised to a fractional power is undefined in COBOL. For example, (-2) ** 3 is equal to -8, but (-2) ** (3.000001) is undefined. When an exponentiation is evaluated in floating point and there is a possibility that the result is undefined, the exponent is evaluated at run time to determine if it has an integral value. If not, a diagnostic message is issued.

Intrinsic functions evaluated in floating-point arithmetic

In compatibility mode, floating-point intrinsic functions always return a long (64-bit) floating-point value. In extended mode, floating-point intrinsic functions always return an extended-precision (128-bit) floating-point value.

Mixed functions that have at least one floating-point argument are evaluated using floating-point arithmetic.

Related references
“Terminology used for intermediate results” on page 478
“ARITH” on page 251

Arithmetic expressions in nonarithmetic statements

Arithmetic expressions can appear in contexts other than arithmetic statements. For example, you can use an arithmetic expression with the IF or EVALUATE statement.

In such statements, the rules for intermediate results with fixed-point data and for intermediate results with floating-point data apply, with the following changes:

- Abbreviated IF statements are handled as though the statements were not abbreviated.
- In an explicit relation condition where at least one of the comparands is an arithmetic expression, *dmax* is the maximum number of decimal places for any operand of either comparand, excluding divisors and exponents. The rules for floating-point arithmetic apply if any of the following conditions is true:
  - Any operand in either comparand is COMP-1, COMP-2, external floating point, or a floating-point literal.
  - An exponent contains decimal places.
  - An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.

  For example:

```
IF operand-1 = expression-1 THEN . . .
```

  If *operand-1* is a data-name defined to be COMP-2, the rules for floating-point arithmetic apply to *expression-1* even if it contains only fixed-point operands, because it is being compared to a floating-point operand.
- When the comparison between an arithmetic expression and another data item or arithmetic expression does not use a relational operator (that is, there is no explicit relation condition), the arithmetic expression is evaluated without regard to the attributes of its comparand. For example:

```
EVALUATE expression-1
  WHEN expression-2 THRU expression-3
  WHEN expression-4
  . . .
END-EVALUATE
```

In the statement above, each arithmetic expression is evaluated in fixed-point or floating-point arithmetic based on its own characteristics.

Related concepts
“Fixed-point contrasted with floating-point arithmetic” on page 53

Related references
“Terminology used for intermediate results” on page 478
“Fixed-point data and intermediate results” on page 479
“Floating-point data and intermediate results” on page 484
IF statement (*COBOL for Linux on x86 Language Reference*)
EVALUATE statement (*COBOL for Linux on x86 Language Reference*)
Conditional expressions (*COBOL for Linux on x86 Language Reference*)

Appendix D. Date and time callable services

By using the date and time callable services, you can get the current local time and date in several formats and can convert dates and times.

The available date and time callable services are shown below. Two of the services, CEEQCEN and CEESCEN, provide a predictable way to handle two-digit years, such as 91 for 1991 or 10 for 2010.

Table 56. ***Date and time callable services***

Callable service	**Description**
"CEECBLDY: convert date to COBOL integer format" on page 488	Converts character date value to COBOL integer date format. Day one is 01 January 1601, and the value is incremented by one for each subsequent day.
"CEEDATE: convert Lilian date to character format" on page 492	Converts dates in the Lilian format back to character values.
"CEEDATM: convert seconds to character time stamp" on page 495	Converts number of seconds to character time stamp.
"CEEDAYS: convert date to Lilian format" on page 499	Converts character date values to the Lilian format. Day one is 15 October 1582 and the value is incremented by one for each subsequent day.
"CEEDYWK: calculate day of week from Lilian date" on page 502	Provides day of week calculation.
"CEEGMT: get current Greenwich Mean Time" on page 504	Gets current Greenwich Mean Time (date and time).
"CEEGMTO: get offset from Greenwich Mean Time to local time" on page 506	Gets difference between Greenwich Mean Time and local time.
"CEEISEC: convert integers to seconds" on page 508	Converts binary year, month, day, hour, second, and millisecond to a number that represents the number of seconds since 00:00:00 15 October 1582.
"CEELOCT: get current local date or time" on page 510	Gets current date and time.
"CEEQCEN: query the century window" on page 512	Queries the callable services century window.
"CEESCEN: set the century window" on page 513	Sets the callable services century window.
"CEESECI: convert seconds to integers" on page 514	Converts a number that represents the number of seconds since 00:00:00 15 October 1582 to seven separate binary integers that represent year, month, day, hour, minute, second, and millisecond.
"CEESECS: convert time stamp to seconds" on page 517	Converts character time stamps (a date and time) to the number of seconds since 00:00:00 15 October 1582.
"CEEUTC: get coordinated universal time" on page 521	Same as CEEGMT.
"IGZEDT4: get current date" on page 521	Returns the current date with a four-digit year in the form YYYMMDD.

© Copyright IBM Corp. 2021

All of these date and time callable services allow source code compatibility with Enterprise COBOL for z/OS. There are, however, significant differences in the way conditions are handled.

The date and time callable services are in addition to the date/time intrinsic functions shown below.

Table 57. ***Date and time intrinsic functions***

Intrinsic function	Description
CURRENT-DATE	Current date and time and difference from Greenwich mean time
DATE-OF-INTEGER[1]	Standard date equivalent (YYYYMMDD) of integer date
DATE-TO-YYYYMMDD[1]	Standard date equivalent (YYYYMMDD) of integer date with a windowed year, according to the specified 100-year interval
DATEVAL[1]	Date field equivalent of integer or alphanumeric date
DAY-OF-INTEGER[1]	Julian date equivalent (YYYYDDD) of integer date
DAY-TO-YYYYDDD[1]	Julian date equivalent (YYYYMMDD) of integer date with a windowed year, according to the specified 100-year interval
INTEGER-OF-DATE	Integer date equivalent of standard date (YYYYMMDD)
INTEGER-OF-DAY	Integer date equivalent of Julian date (YYYYDDD)
UNDATE[1]	Nondate equivalent of integer or alphanumeric date field
YEAR-TO-YYYY[1]	Expanded year equivalent (YYYY) of windowed year, according to the specified 100-year interval
YEARWINDOW[1]	Starting year of the century window specified by the YEARWINDOW compiler option

1. Behavior depends on the setting of the DATEPROC compiler option.

"Example: formatting dates for output" on page 458

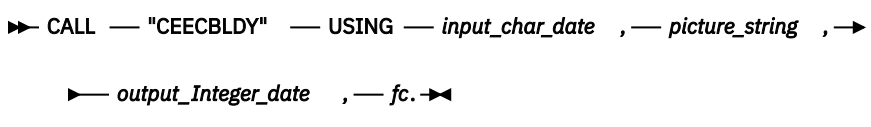
Related references
"Feedback token" on page 459
CALL statement (*COBOL for Linux on x86 Language Reference*)
Function definitions (*COBOL for Linux on x86 Language Reference*)

CEECBLDY: convert date to COBOL integer format

CEECBLDY converts a string that represents a date into the number of days since 31 December 1600. Use CEECBLDY to access the century window of the date and time callable services and to perform date calculations with COBOL intrinsic functions.

This service is similar to CEEDAYS except that it provides a string in COBOL integer format, which is compatible with COBOL intrinsic functions.

CALL CEECBLDY syntax



***input_char_date* (input)**
A halfword length-prefixed character string that represents a date or time stamp in a format conforming to that specified by *picture_string*.

The character string must contain between 5 and 255 characters, inclusive. *input_char_date* can contain leading or trailing blanks. Parsing for a date begins with the first nonblank character (unless the picture string itself contains leading blanks, in which case CEECBLDY skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture_string*, CEECBLDY ignores all remaining characters. Valid dates range between and include 01 January 1601 to 31 December 9999.

***picture_string* (input)**
A halfword length-prefixed character string indicating the format of the date specified in *input_char_date*.

Each character in the *picture_string* corresponds to a character in *input_char_date*. For example, if you specify MMDDYY as the *picture_string*, CEECBLDY reads an *input_char_date* of 060288 as 02 June 1988.

If delimiters such as the slash (/) appear in the picture string, you can omit leading zeros. For example, the following calls to CEECBLDY each assign the same value, 141502 (02 June 1988), to `COBINTDTE`:

```
MOVE '6/2/88' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

```
MOVE '06/02/88' TO DATEVAL-STRING.
MOVE 8 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

```
MOVE '060288' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MMDDYY' TO PICSTR-STRING.
MOVE 6 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

```
MOVE '88154' TO DATEVAL-STRING.
MOVE 5 TO DATEVAL-LENGTH.
MOVE 'YYDDD' TO PICSTR-STRING.
MOVE 5 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

Whenever characters such as colons or slashes are included in the *picture_string* (such as `HH:MI:SS YY/MM/DD`), they count as placeholders but are otherwise ignored.

If *picture_string* includes a Japanese Era symbol <`JJJJ`>, the YY position in *input_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

***output_Integer_date* (output)**
A 32-bit binary integer that represents the COBOL integer date, the number of days since 31 December 1600. For example, 16 May 1988 is day number 141485.

If *input_char_date* does not contain a valid date, *output_Integer_date* is set to 0, and CEECBLDY terminates with a non-CEE000 symbolic feedback code.

Date calculations are performed easily on the *output_Integer_date*, because *output_Integer_date* is an integer. Leap year and end-of-year anomalies do not affect the calculations.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 58. ***CEECBLDY symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The era passed to CEEDAYS or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EO	3	2520	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
CEE2EP	3	2521	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.

Usage notes

- Call CEECBLDY only from COBOL programs that use the returned value as input to COBOL intrinsic functions. Unlike CEEDAYS, there is no inverse function of CEECBLDY, because it is only for COBOL users who want to use the date and time century window service together with COBOL intrinsic functions for date calculations. The inverse of CEECBLDY is provided by the `DATE-OF-INTEGER` and `DAY-OF-INTEGER` intrinsic functions.
- To perform calculations on dates earlier than 1 January 1601, add 4000 to the year in each date, convert the dates to COBOL integer format, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.
- By default, two-digit years lie within the 100-year range that starts 80 years before the system date. Thus in 2010, all two-digit years represent dates between 1930 and 2029, inclusive. You can change this default range by using the CEESCEN callable service.

Example

```
      ************************************************
      **                                            **
      ** Function: Invoke CEECBLDY callable service  **
      ** to convert date to COBOL integer format.    **
      ** This service is used when using the         **
      ** Century Window feature of the date and time **
      ** callable services mixed with COBOL          **
      ** intrinsic functions.                        **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDY.
      *
       DATA DIVISION.
```

```
       WORKING-STORAGE SECTION.
       01  CHRDATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of CHRDATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of  PICSTR.
       01  INTEGER                 PIC S9(9) BINARY.
       01  NEWDATE                 PIC 9(8).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                       REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
      *
       PROCEDURE DIVISION.
       PARA-CBLDAYS.
      ************************************************
      ** Specify input date and length               **
      ************************************************
           MOVE 25 TO Vstring-length of CHRDATE.
           MOVE '1 January 00'
               to Vstring-text of CHRDATE.
      ************************************************
      ** Specify a picture string that describes     **
      ** input date, and set the string's length.    **
      ************************************************
           MOVE 23 TO Vstring-length of PICSTR.
           MOVE 'ZD Mmmmmmmmmmmmmmz YY'
                     TO Vstring-text of PICSTR.
      ************************************************
      ** Call CEECBLDY to convert input date to a    **
      ** COBOL integer date                          **
      ************************************************
           CALL 'CEECBLDY' USING CHRDATE, PICSTR,
                                 INTEGER, FC.
      ************************************************
      ** If CEECBLDY runs successfully, then compute **
      **     the date of the 90th day after the      **
      **     input date using Intrinsic Functions    **
      ************************************************
           IF CEE000 of FC  THEN
               COMPUTE INTEGER = INTEGER + 90
               COMPUTE NEWDATE = FUNCTION
                   DATE-OF-INTEGER (INTEGER)
               DISPLAY NEWDATE
                   ' is Lilian day: ' INTEGER
           ELSE
               DISPLAY 'CEEBLDY failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.
      *
           GOBACK.
```

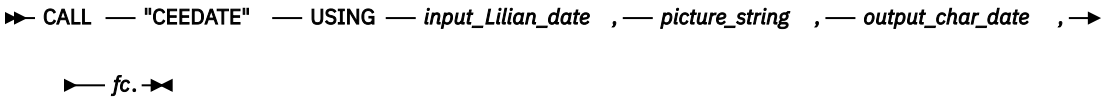
Related references

“Picture character terms and strings” on page 460

CEEDATE: convert Lilian date to character format

CEEDATE converts a number that represents a Lilian date to a date written in character format. The output is a character string, such as 2010/04/23.

CALL CEEDATE syntax



***input_Lilian_date* (input)**
A 32-bit integer that represents the Lilian date. The Lilian date is the number of days since 14 October 1582. For example, 16 May 1988 is Lilian day number 148138. The valid range of Lilian dates is 1 to 3,074,324 (15 October 1582 to 31 December 9999).

***picture_string* (input)**
A halfword length-prefixed character string that represents the required format of *output_char_date*, for example MM/DD/YY. Each character in *picture_string* represents a character in *output_char_date*. If delimiters such as the slash (/) appear in the picture string, they are copied as is to *output_char_date*.

If *picture_string* includes a Japanese Era symbol <JJJJ>, the YY position in *output_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

***output_char_date* (output)**
A fixed-length 80-character string that is the result of converting *input_Lilian_date* to the format specified by *picture_string*. If *input_Lilian_date* is invalid, *output_char_date* is set to all blanks and CEEDATE terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 59. ***CEEDATE symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EG	3	2512	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EQ	3	2522	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.
CEE2EU	2	2526	The date string returned by CEEDATE was truncated.
CEE2F6	1	2534	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Usage note: The inverse of CEEDATE is CEEDAYS, which converts character dates to the Lilian format.

Example

```
*************************************************
**                                             **
** Function: CEEDATE - convert Lilian date to **
**                     character format        **
**                                             **
** In this example, a call is made to CEEDATE **
** to convert a Lilian date (the number of     **
** days since 14 October 1582) to a character **
** format (such as 6/22/98). The result is     **
** displayed.  The Lilian date is obtained     **
** via a call to CEEDAYS.                      **
**                                             **
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLDATE.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LILIAN                  PIC S9(9) BINARY.
 01  CHRDATE                 PIC X(80).
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
*
 PROCEDURE DIVISION.
 PARA-CBLDAYS.
************************************************
** Call CEEDAYS to convert date of 6/2/98 to   **
**     Lilian representation                    **
************************************************
     MOVE 6 TO Vstring-length of IN-DATE.
     MOVE '6/2/98' TO Vstring-text of IN-DATE(1:6).
     MOVE 8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR(1:8).
     CALL 'CEEDAYS' USING IN-DATE, PICSTR,
                          LILIAN, FC.

************************************************
** If CEEDAYS runs successfully, display result**
************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

************************************************
** Specify picture string that describes the   **
**  required format of the output from CEEDATE, **
**  and the picture string's length.           **
************************************************
     MOVE 23 TO Vstring-length OF PICSTR.
```

```
           MOVE 'ZD Mmmmmmmmmmmmmmz YYYY' TO
                        Vstring-text OF PICSTR(1:23).

      ************************************************
      ** Call CEEDATE to convert the Lilian date     **
      **     to  a picture string.                    **
      ************************************************
           CALL 'CEEDATE' USING LILIAN, PICSTR,
                                CHRDATE, FC.

      ************************************************
      ** If CEEDATE runs successfully, display result**
      ************************************************
           IF CEE000 of FC  THEN
               DISPLAY 'Input Lilian date of ' LILIAN
                   ' corresponds to:  ' CHRDATE
           ELSE
               DISPLAY 'CEEDATE failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

           GOBACK.
```

The following table shows the sample output from CEEDATE.

input_Lilian_date	picture_string	output_char_date
148138	YY	98
	YYMM	9805
	YY-MM	98-05
	YYMMDD	980516
	YYYYMMDD	19980516
	YYYY-MM-DD	1998-05-16
	YYYY-ZM-ZD	1998-5-16
	<JJJJ> YY.MM.DD	Showa 63.05.16 (in a DBCS string)
148139	MM	05
	MMDD	0517
	MM/DD	05/17
	MMDDYY	051798
	MM/DD/YYYY	05/17/1998
	ZM/DD/YYYY	5/17/1998
148140	DD	18
	DDMM	1805
	DDMMYY	180598
	DD.MM.YY	18.05.98
	DD.MM.YYYY	18.05.1998
	DD Mmm YYYY	18 May 1998

input_Lilian_date	picture_string	output_char_date
148141	DDD	140
	YYDDD	98140
	YY.DDD	98.140
	YYYY.DDD	1998.140
148142	YY/MM/DD HH:MI:SS.99	98/05/20 00:00:00.00
	YYYY/ZM/ZD ZH:MI AP	1998/5/20 0:00 AM
148143	WWW., MMM DD, YYYY	SAT., MAY 21, 1998
	Www., Mmm DD, YYYY	Sat., May 21, 1998
	Wwwwwwwwww, Mmmmmmmmmm DD, YYYY	Saturday, May 21, 1998
	Wwwwwwwwwz, Mmmmmmmmmz DD, YYYY	Saturday, May 21, 1998

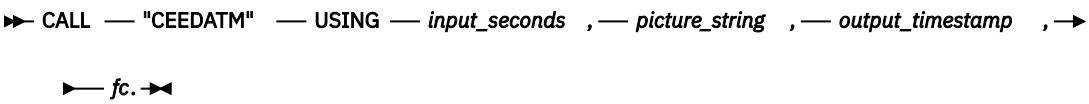
"Example: date-and-time picture strings" on page 462

Related references
"Picture character terms and strings" on page 460

CEEDATM: convert seconds to character time stamp

CEEDATM converts a number that represents the number of seconds since 00:00:00 14 October 1582 to a character string. The output is a character string time stamp such as 1988/07/26 20:37:00.

CALL CEEDATM syntax



***input_seconds* (input)**
A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The valid range of *input_seconds* is 86,400® to 265,621,679,999.999 (23:59:59.999 31 December 9999).

***picture_string* (input)**
A halfword length-prefixed character string that represents the required format of *output_timestamp*, for example, MM/DD/YY HH:MI AP.

Each character in the *picture_string* represents a character in *output_timestamp*. If delimiters such as a slash (/) are used in the picture string, they are copied as is to *output_timestamp*.

If *picture_string* includes the Japanese Era symbol <JJJJ>, the YY position in *output_timestamp* represents the year within Japanese Era.

***output_timestamp* (output)**
A fixed-length 80-character string that is the result of converting *input_seconds* to the format specified by *picture_string*.

If necessary, the output is truncated to the length of *output_timestamp*.

If *input_seconds* is invalid, *output_timestamp* is set to all blanks and CEEDATM terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 60. ***CEEDATM symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
CEE2EA	3	2506	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date or time service.
CEE2EV	2	2527	The time-stamp string returned by CEEDATM was truncated.
CEE2F6	1	2534	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Usage note: The inverse of CEEDATM is CEESECS, which converts a time stamp to number of seconds.

Example

```
************************************************
**                                            **
** Function: CEEDATM - convert seconds to     **
**                     character time stamp   **
**                                            **
** In this example, a call is made to CEEDATM  **
** to convert a date represented in Lilian    **
** seconds (the number of seconds since       **
** 00:00:00 14 October 1582) to a character   **
** format (such as 06/02/88 10:23:45). The    **
** result is displayed.                       **
**                                            **
************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLDATM.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  DEST          PIC S9(9) BINARY VALUE 2.
 01  SECONDS                 COMP-2.
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  TIMESTP                 PIC X(80).
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
```

```
           03  Case-1-Condition-ID.
               04  Severity    PIC S9(4) COMP.
               04  Msg-No      PIC S9(4) COMP.
           03  Case-2-Condition-ID
                     REDEFINES Case-1-Condition-ID.
               04  Class-Code  PIC S9(4) COMP.
               04  Cause-Code  PIC S9(4) COMP.
           03  Case-Sev-Ctl    PIC X.
           03  Facility-ID     PIC XXX.
       02  I-S-Info            PIC S9(9) COMP.
*
 PROCEDURE DIVISION.
 PARA-CBLDATM.
************************************************
** Call CEESECS to convert time stamp of 6/2/88**
**     at 10:23:45 AM to Lilian representation **
************************************************
     MOVE 20 TO Vstring-length of IN-DATE.
     MOVE '06/02/88 10:23:45 AM'
         TO Vstring-text of IN-DATE.
     MOVE 20 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI:SS AP'
         TO Vstring-text of PICSTR.
     CALL 'CEESECS' USING IN-DATE, PICSTR,
                          SECONDS, FC.

************************************************
** If CEESECS runs successfully, display result**
************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian second:  ' SECONDS
     ELSE
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

************************************************
** Specify required format of the output.       **
************************************************
     MOVE 35 TO Vstring-length OF PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY at HH:MI:SS'
            TO Vstring-text OF PICSTR.

************************************************
** Call CEEDATM to convert Lilian seconds to   **
**     a character time stamp                   **
************************************************
     CALL 'CEEDATM' USING SECONDS, PICSTR,
                          TIMESTP, FC.

************************************************
** If CEEDATM runs successfully, display result**
************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input seconds of ' SECONDS
             ' corresponds to: ' TIMESTP
     ELSE
         DISPLAY 'CEEDATM failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

The following tables show sample output of CEEDATM.

input_seconds	picture_string	output_timestamp
12,799,191,601.000	YYMMDD	880516
	HH:MI:SS	19:00:01
	YY-MM-DD	88-05-16
	YYMMDDHHMISS	880516190001
	YY-MM-DD HH:MI:SS	88-05-16 19:00:01
	YYYY-MM-DD HH:MI:SS AP	1988-05-16 07:00:01 PM
12,799,191,661.986	DD Mmm YY	16 May 88
	DD MMM YY HH:MM	16 MAY 88 19:01
	WWW, MMM DD, YYYY	MON, MAY 16, 1988
	ZH:MI AP	7:01 PM
	Wwwwwwwwwz, ZM/ZD/YY	Monday, 5/16/88
	HH:MI:SS.99	19:01:01.98
12,799,191,662.009	YYYY	1988
	YY	88
	Y	8
	MM	05
	ZM	5
	RRRR	V

input_seconds	picture_string	output_timestamp
12,799,191,662.009	MMM	MAY
	Mmm	May
	Mmmmmmmmmm	May
	Mmmmmmmmmz	May
	DD	16
	ZD	16
	DDD	137
	HH	19
	ZH	19
	MI	01
	SS	02
	99	00
	999	009
	AP	PM
	WWW	MON
	Www	Mon
	Wwwwwwwwww	Monday
	Wwwwwwwwwz	Monday

"Example: date-and-time picture strings" on page 462

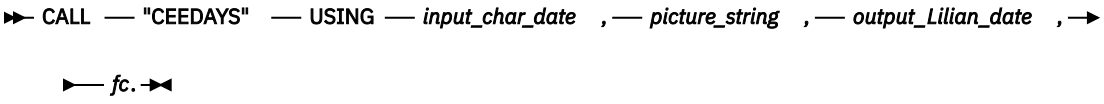
Related references
"Picture character terms and strings" on page 460

CEEDAYS: convert date to Lilian format

CEEDAYS converts a string that represents a date into a Lilian format, which represents a date as the number of days from the beginning of the Gregorian calendar (Friday, 14 October, 1582).

Do not use CEEDAYS in combination with COBOL intrinsic functions. Use CEECBLDY for programs that use intrinsic functions.

CALL CEEDAYS syntax



***input_char_date* (input)**
A halfword length-prefixed character string that represents a date or a time stamp in a format conforming to that specified by *picture_string*.

The character string must contain between 5 and 255 characters, inclusive. *input_char_date* can contain leading or trailing blanks. Parsing for a date begins with the first nonblank character (unless the picture string itself contains leading blanks, in which case CEEDAYS skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture_string*, CEEDAYS ignores all remaining characters. Valid dates range between and include 15 October 1582 to 31 December 9999.

***picture_string* (input)**
A halfword length-prefixed character string, indicating the format of the date specified in *input_char_date*.

Each character in the *picture_string* corresponds to a character in *input_char_date*. For example, if you specify MMDDYY as the *picture_string*, CEEDAYS reads an *input_char_date* of 060288 as 02 June 1988.

If delimiters such as a slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEEDAYS each assign the same value, 148155 (02 June 1988), to *lildate*:

```
CALL CEEDAYS USING '6/2/88'   , 'MM/DD/YY', lildate, fc.
CALL CEEDAYS USING '06/02/88', 'MM/DD/YY', lildate, fc.
CALL CEEDAYS USING '060288'   , 'MMDDYY'  , lildate, fc.
CALL CEEDAYS USING '88154'    , 'YYDDD'   , lildate, fc.
```

Whenever characters such as colons or slashes are included in the *picture_string* (such as `HH:MI:SS YY/MM/DD`), they count as placeholders but are otherwise ignored.

If *picture_string* includes a Japanese Era symbol `<JJJJ>`, the YY position in *input_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

***output_Lilian_date* (output)**
A 32-bit binary integer that represents the Lilian date, the number of days since 14 October 1582. For example, 16 May 1988 is day number 148138.

If *input_char_date* does not contain a valid date, *output_Lilian_date* is set to 0 and CEEDAYS terminates with a non-CEE000 symbolic feedback code.

Date calculations are performed easily on the *output_Lilian_date*, because it is an integer. Leap year and end-of-year anomalies do not affect the calculations.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 61. ***CEEDAYS symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The era passed to CEEDAYS or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EO	3	2520	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.

*Table 61. **CEEDAYS symbolic conditions** (continued)*			
Symbolic feedback code	**Severity**	**Message number**	**Message text**
CEE2EP	3	2521	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.

Usage notes

- The inverse of CEEDAYS is CEEDATE, which converts *output_Lilian_date* from Lilian format to character format.
- To perform calculations on dates earlier than 15 October 1582, add 4000 to the year in each date, convert the dates to Lilian, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.
- By default, two-digit years lie within the 100-year range that starts 80 years before the system date. Thus in 2010, all two-digit years represent dates between 1930 and 2029, inclusive. You can change the default range by using the callable service CEESCEN.
- You can easily perform date calculations on the *output_Lilian_date*, because it is an integer. Leap-year and end-of-year anomalies are avoided.

Example

```
******************************************
**                                      **
** Function: CEEDAYS - convert date to  **
**                     Lilian format    **
**                                      **
******************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLDAYS.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CHRDATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of CHRDATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of PICSTR.
 01  LILIAN                  PIC S9(9) BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
*
 PROCEDURE DIVISION.
 PARA-CBLDAYS.
************************************************
** Specify input date and length              **
************************************************
     MOVE 16 TO Vstring-length of CHRDATE.
     MOVE '1 January 2005'
         TO Vstring-text of CHRDATE.
```

```
      ************************************************
      ** Specify a picture string that describes     **
      ** input date, and the picture string's length.**
      ************************************************
           MOVE 25 TO Vstring-length of PICSTR.
           MOVE 'ZD Mmmmmmmmmmmmmmz YYYY'
                   TO Vstring-text of PICSTR.

      ************************************************
      ** Call CEEDAYS to convert input date to a     **
      ** Lilian date                                 **
      ************************************************
           CALL 'CEEDAYS' USING CHRDATE, PICSTR,
                                LILIAN, FC.

      ************************************************
      ** If CEEDAYS runs successfully, display result**
      ************************************************
           IF  CEE000 of FC  THEN
               DISPLAY Vstring-text of CHRDATE
                   ' is Lilian day: ' LILIAN
           ELSE
               DISPLAY 'CEEDAYS failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

           GOBACK.
```

"Example: date-and-time picture strings" on page 462

Related references
"Picture character terms and strings" on page 460

CEEDYWK: calculate day of week from Lilian date

CEEDYWK calculates the day of the week on which a Lilian date falls as a number between 1 and 7.

The number returned by CEEDYWK is useful for end-of-week calculations.

CALL CEEDYWK syntax

***input_Lilian_date* (input)**
A 32-bit binary integer that represents the Lilian date, the number of days since 14 October 1582.

For example, 16 May 1988 is day number 148138. The valid range of *input_Lilian_date* is between 1 and 3,074,324 (15 October 1582 and 31 December 9999).

***output_day_no* (output)**
A 32-bit binary integer that represents *input_Lilian_date*'s day-of-week: 1 equals Sunday, 2 equals Monday, . . ., 7 equals Saturday.

If *input_Lilian_date* is invalid, *output_day_no* is set to 0 and CEEDYWK terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 62. ***CEEDYWK symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.

*Table 62. **CEEDYWK symbolic conditions** (continued)*

Symbolic feedback code	Severity	Message number	Message text
CEE2EG	3	2512	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Example

```
*************************************************
**                                             **
** Function: Call CEEDYWK to calculate the    **
**           day of the week from Lilian date **
**                                             **
** In this example, a call is made to CEEDYWK **
** to return the day of the week on which a   **
** Lilian date falls. (A Lilian date is the   **
** number of days since 14 October 1582)      **
**                                             **
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLDYWK.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LILIAN                  PIC S9(9) BINARY.
 01  DAYNUM                  PIC S9(9) BINARY.
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.

 PROCEDURE DIVISION.
 PARA-CBLDAYS.
** Call CEEDAYS to convert date of 6/2/88 to
**     Lilian representation
     MOVE 6 TO Vstring-length of IN-DATE.
     MOVE '6/2/88' TO Vstring-text of IN-DATE(1:6).
     MOVE 8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR(1:8).
     CALL 'CEEDAYS' USING IN-DATE, PICSTR,
         LILIAN, FC.

** If CEEDAYS runs successfully, display result.
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.
```

```
 PARA-CBLDYWK.

** Call CEEDYWK to return the day of the week on
** which the Lilian date falls
     CALL 'CEEDYWK' USING LILIAN , DAYNUM , FC.

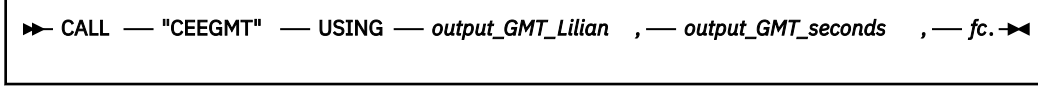
** If CEEDYWK runs successfully, print results
     IF CEE000 of FC  THEN
         DISPLAY 'Lilian day ' LILIAN
             ' falls on day ' DAYNUM
             ' of the week, which is a:'
** Select DAYNUM to display the name of the day
**     of the week.
           EVALUATE DAYNUM
             WHEN 1
               DISPLAY 'Sunday.'
             WHEN 2
               DISPLAY 'Monday.'
             WHEN 3
               DISPLAY 'Tuesday'
             WHEN 4
               DISPLAY 'Wednesday.'
             WHEN 5
               DISPLAY 'Thursday.'
             WHEN 6
               DISPLAY 'Friday.'
             WHEN 7
               DISPLAY 'Saturday.'
           END-EVALUATE
     ELSE
         DISPLAY 'CEEDYWK failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEEGMT: get current Greenwich Mean Time

CEEGMT returns the current Greenwich Mean Time (GMT) as both a Lilian date and as the number of seconds since 00:00:00 14 October 1582. The returned values are compatible with those generated and used by the other date and time callable services.

CALL CEEGMT syntax



***output_GMT_Lilian* (output)**

A 32-bit binary integer that represents the current date in Greenwich, England, in the Lilian format (the number of days since 14 October 1582).

For example, 16 May 1988 is day number 148138. If GMT is not available from the system, *output_GMT_Lilian* is set to 0 and CEEGMT terminates with a non-CEE000 symbolic feedback code.

***output_GMT_seconds* (output)**

A 64-bit long floating-point number that represents the current date and time in Greenwich, England, as the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). 19:00:01.078 on 16 May 1988 is second number 12,799,191,601.078. If GMT is not available from the system, *output_GMT_seconds* is set to 0 and CEEGMT terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**

A 12-byte feedback code (optional) that indicates the result of this service.

*Table 63. **CEEGMT symbolic conditions***			
Symbolic feedback code	**Severity**	**Message number**	**Message text**
CEE000	0	--	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.

Usage notes

- CEEDATE converts *output_GMT_Lilian* to a character date, and CEEDATM converts *output_GMT_seconds* to a character time stamp.
- In order for the results of this service to be meaningful, your system's clock must be set to the local time and the environment variable TZ must be set correctly.
- The values returned by CEEGMT are handy for elapsed time calculations. For example, you can calculate the time elapsed between two calls to CEEGMT by calculating the differences between the returned values.
- CEEUTC is identical to this service.

Example

```
      ************************************************
      **                                            **
      ** Function: Call CEEGMT to get current       **
      **           Greenwich Mean Time              **
      **                                            **
      ** In this example, a call is made to CEEGMT   **
      ** to return the current GMT as a Lilian date  **
      ** and as Lilian seconds. The results are      **
      ** displayed.                                 **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTGMT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
       01  SECS                    COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                        REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLGMT.
           CALL 'CEEGMT' USING LILIAN , SECS , FC.

           IF CEE000 of FC  THEN
               DISPLAY 'The current GMT is also '
                   'known as Lilian day: ' LILIAN
               DISPLAY 'The current GMT in Lilian '
                   'seconds is: ' SECS
           ELSE
               DISPLAY 'CEEGMT failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

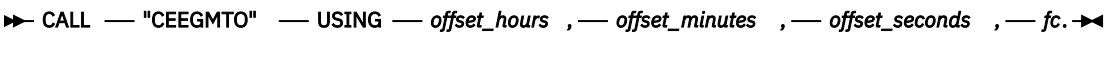
           GOBACK.
```

Related tasks
"Setting environment variables" on page 213

CEEGMTO: get offset from Greenwich Mean Time to local time

CEEGMTO returns values to the calling routine that represent the difference between the local system time and Greenwich Mean Time (GMT).

CALL CEEGMTO syntax



***offset_hours* (output)**
A 32-bit binary integer that represents the offset from GMT to local time, in hours.

For example, for Pacific Standard Time, *offset_hours* equals -8.

The range of *offset_hours* is -12 to +13 (+13 = daylight saving time in the +12 time zone).

If local time offset is not available, *offset_hours* equals 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

***offset_minutes* (output)**
A 32-bit binary integer that represents the number of additional minutes that local time is ahead of or behind GMT.

The range of *offset_minutes* is 0 to 59.

If the local time offset is not available, *offset_minutes* equals 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

***offset_seconds* (output)**
A 64-bit long floating-point number that represents the offset from GMT to local time, in seconds.

For example, Pacific Standard Time is eight hours behind GMT. If local time is in the Pacific time zone during standard time, CEEGMTO would return -28,800 (-8 * 60 * 60). The range of *offset_seconds* is -43,200 to +46,800. *offset_seconds* can be used with CEEGMT to calculate local date and time.

If the local time offset is not available from the system, *offset_seconds* is set to 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 64. ***CEEGMTO symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E7	3	2503	The offset from UTC/GMT to local time was not available from the system.

Usage notes

- CEEDATM converts *offset_seconds* to a character time stamp.
- In order for the results of this service to be meaningful, your system clock must be set to the local time, and the environment variable TZ must be set correctly.

Example

```
      ************************************************
      **                                            **
      ** Function:  Call CEEGMTO to get offset from  **
      **            Greenwich Mean Time to local     **
      **            time                             **
      **                                            **
      ** In this example, a call is made to CEEGMTO  **
      ** to return the offset from GMT to local time **
      ** as separate binary integers representing    **
      ** offset hours, minutes, and seconds. The     **
      ** results are displayed.                      **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTGMTO.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  HOURS                   PIC S9(9) BINARY.
       01  MINUTES                 PIC S9(9) BINARY.
       01  SECONDS COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLGMTO.
           CALL 'CEEGMTO' USING HOURS , MINUTES ,
               SECONDS , FC.

           IF CEE000 of FC  THEN
               DISPLAY 'Local time differs from GMT '
                   'by: ' HOURS ' hours, '
                   MINUTES ' minutes, OR  '
                   SECONDS ' seconds. '
           ELSE
               DISPLAY 'CEEGMTO failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

           GOBACK.
```

Related tasks

“Setting environment variables” on page 213

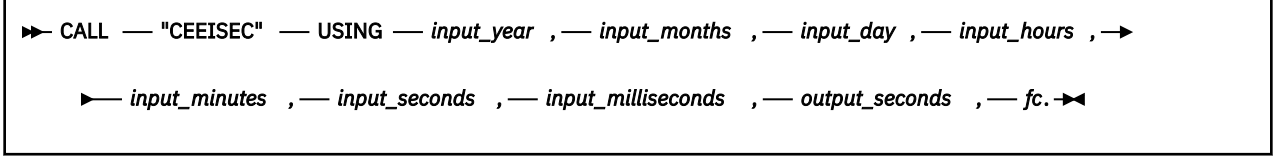
Related references

“Compiler and runtime environment variables” on page 214
“CEEGMT: get current Greenwich Mean Time” on page 504

CEEISEC: convert integers to seconds

CEEISEC converts binary integers that represent year, month, day, hour, minute, second, and millisecond to a number that represents the number of seconds since 00:00:00 14 October 1582.

CALL CEEISEC syntax



***input_year* (input)**
A 32-bit binary integer that represents the year.

The range of valid values for *input_year* is 1582 to 9999, inclusive.

***input_month* (input)**
A 32-bit binary integer that represents the month.

The range of valid values for *input_month* is 1 to 12.

***input_day* (input)**
A 32-bit binary integer that represents the day.

The range of valid values for *input_day* is 1 to 31.

***input_hours* (input)**
A 32-bit binary integer that represents the hours.

The range of valid values for *input_hours* is 0 to 23.

***input_minutes* (input)**
A 32-bit binary integer that represents the minutes.

The range of valid values for *input_minutes* is 0 to 59.

***input_seconds* (input)**
A 32-bit binary integer that represents the seconds.

The range of valid values for *input_seconds* is 0 to 59.

***input_milliseconds* (input)**
A 32-bit binary integer that represents milliseconds.

The range of valid values for *input_milliseconds* is 0 to 999.

***output_seconds* (output)**
A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The valid range of *output_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

If any input values are invalid, *output_seconds* is set to zero.

To convert *output_seconds* to a Lilian day number, divide *output_seconds* by 86,400 (the number of seconds in a day).

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

*Table 65. **CEEISEC symbolic conditions***			
Symbolic feedback code	**Severity**	**Message number**	**Message text**
CEE000	0	--	The service completed successfully.
CEE2EE	3	2510	The hours value in a call to CEEISEC or CEESECS was not recognized.
CEE2EF	3	2511	The day parameter passed in a CEEISEC call was invalid for year and month specified.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EI	3	2514	The year value passed in a CEEISEC call was not within the supported range.
CEE2EJ	3	2515	The milliseconds value in a CEEISEC call was not recognized.
CEE2EK	3	2516	The minutes value in a CEEISEC call was not recognized.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EN	3	2519	The seconds value in a CEEISEC call was not recognized.

Usage note: The inverse of CEEISEC is CEESECI, which converts number of seconds to integer year, month, day, hour, minute, second, and millisecond.

Example

```
      ************************************************
      **                                            **
      ** Function: Call CEEISEC to convert integers  **
      **           to seconds                       **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLISEC.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  YEAR                    PIC S9(9) BINARY.
       01  MONTH                   PIC S9(9) BINARY.
       01  DAYS                    PIC S9(9) BINARY.
       01  HOURS                   PIC S9(9) BINARY.
       01  MINUTES                 PIC S9(9) BINARY.
       01  SECONDS                 PIC S9(9) BINARY.
       01  MILLSEC                 PIC S9(9) BINARY.
       01  OUTSECS                 COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLISEC.
      ************************************************
      ** Specify seven binary integers representing  **
      ** the date and time as input to be converted  **
      ** to Lilian seconds                          **
      ************************************************
           MOVE 2000 TO YEAR.
           MOVE 1 TO MONTH.
```

```
           MOVE 1 TO DAYS.
           MOVE 0 TO HOURS.
           MOVE 0 TO MINUTES.
           MOVE 0 TO SECONDS.
           MOVE 0 TO MILLSEC.
      ************************************************
      ** Call CEEISEC to convert the integers        **
      ** to seconds                                  **
      ************************************************
           CALL 'CEEISEC' USING YEAR, MONTH, DAYS,
                                HOURS, MINUTES, SECONDS,
                                MILLSEC, OUTSECS , FC.
      ************************************************
      ** If CEEISEC runs successfully, display result**
      ************************************************
           IF CEE000 of FC  THEN
               DISPLAY MONTH '/' DAYS '/' YEAR
                 ' AT ' HOURS ':' MINUTES ':' SECONDS
                 ' is equivalent to ' OUTSECS ' seconds'
           ELSE
               DISPLAY 'CEEISEC failed with msg '
                 Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

           GOBACK.
```

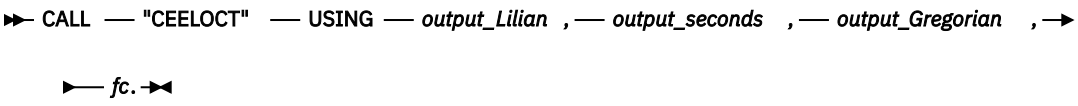
CEELOCT: get current local date or time

CEELOCT returns the current local date or time as a Lilian date (the number of days since 14 October 1582), as Lilian seconds (the number of seconds since 00:00:00 14 October 1582), and as a Gregorian character string (YYYYMMDDHHMISS999).

These values are compatible with other date and time callable services and with existing intrinsic functions.

CEELOCT performs the same function as calling the CEEGMT, CEEGMTO, and CEEDATM services separately. Calling CEELOCT, however, is much faster.

CALL CEELOCT syntax



***output_Lilian* (output)**

A 32-bit binary integer that represents the current local date in the Lilian format, that is, day 1 equals 15 October 1582, day 148,887 equals 4 June 1990.

If the local time is not available from the system, *output_Lilian* is set to 0 and CEELOCT terminates with a non-CEE000 symbolic feedback code.

***output_seconds* (output)**

A 64-bit long floating-point number that represents the current local date and time as the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds. For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). 19:00:01.078 on 4 June 1990 is second number 12,863,905,201.078.

If the local time is not available from the system, *output_seconds* is set to 0 and CEELOCT terminates with a non-CEE000 symbolic feedback code.

***output_Gregorian* (output)**

A 17-byte fixed-length character string in the form YYYYMMDDHHMISS999 that represents local year, month, day, hour, minute, second, and millisecond.

If the format of *output_Gregorian* does not meet your needs, you can use the CEEDATM callable service to convert *output_seconds* to another format.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 66. ***CEELOCT symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2F3	3	2531	The local time was not available from the system.

Usage notes

- You can use the CEEGMT callable service to determine Greenwich Mean Time (GMT).
- You can use the CEEGMTO callable service to obtain the offset from GMT to local time.
- The character value returned by CEELOCT is designed to match that produced by existing intrinsic functions. The numeric values returned can be used to simplify date calculations.

Example

```
      ********************************************
      **                                        **
      ** Function: Call CEELOCT to get current  **
      **           local time                   **
      **                                        **
      ** In this example, a call is made to CEELOCT **
      ** to return the current local time in Lilian **
      ** days (the number of days since 14 October  **
      ** 1582), Lilian seconds (the number of       **
      ** seconds since 00:00:00 14 October 1582),   **
      ** and a Gregorian string (in the form        **
      ** YYYMMDDMISS999). The Gregorian character   **
      ** string is then displayed.                  **
      **                                        **
      ********************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLLOCT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
       01  SECONDS                 COMP-2.
       01  GREGORN                 PIC X(17).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLLOCT.
           CALL 'CEELOCT' USING LILIAN, SECONDS,
                                GREGORN, FC.
      ********************************************
      ** If CEELOCT runs successfully, display      **
      **     Gregorian character string             **
      ********************************************
           IF CEE000 of FC  THEN
               DISPLAY 'Local Time is ' GREGORN
           ELSE
               DISPLAY 'CEELOCT failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.
```

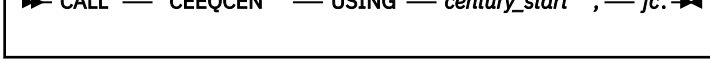
```
           GOBACK.
```

CEEQCEN: query the century window

CEEQCEN queries the century window, which is a two-digit year value.

When you want to change the century window, use CEEQCEN to get the setting and then use CEESCEN to save and restore the current setting.

CALL CEEQCEN syntax



***century_start* (output)**

An integer between 0 and 100 that indicates the year on which the century window is based.

For example, if the date and time callable services default is in effect, all two-digit years lie within the 100-year window that starts 80 years before the system date. CEEQCEN then returns the value 80. For example, in the year 2010, 80 indicates that all two-digit years lie within the 100-year window between 1930 and 2029, inclusive.

***fc* (output)**

A 12-byte feedback code (optional) that indicates the result of this service.

*Table 67. **CEEQCEN symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.

Example

```
*************************************************
**                                             **
** Function: Call CEEQCEN to query the         **
**           date and time callable services   **
**           century window                    **
**                                             **
** In this example, CEEQCEN is called to query **
** the date at which the century window starts **
** The century window is the 100-year window   **
** within which the date and time callable     **
** services assume all two-digit years lie.    **
**                                             **
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLQCEN.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  STARTCW                PIC S9(9) BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.
```

```
 PARA-CBLQCEN.
*************************************************
** Call CEEQCEN to return the start of the     **
**     century window                            **
*************************************************

     CALL 'CEEQCEN' USING STARTCW, FC.
*************************************************
** CEEQCEN has no nonzero feedback codes to    **
**     check, so just display result.           **
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'The start of the century '
              'window is: ' STARTCW
     ELSE
         DISPLAY 'CEEQCEN failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEESCEN: set the century window

CEESCEN sets the century window to a two-digit year value for use by other date and time callable services.

Use CEESCEN in conjunction with CEEDAYS or CEESECS when:

- You process date values that contain two-digit years (for example, in the YYMMDD format).
- The default century interval does not meet the requirements of a particular application.

To query the century window, use CEEQCEN.

CALL CEESCEN syntax

century_start
An integer between 0 and 100, which sets the century window.

A value of 80, for example, places all two-digit years within the 100-year window that starts 80 years before the system date. In 2010, therefore, all two-digit years are assumed to represent dates between 1930 and 2029, inclusive.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

*Table 68. **CEESCEN symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.
CEE2F5	3	2533	The value passed to CEESCEN was not between 0 and 100.

Example

```
*************************************************
**                                             **
** Function: Call CEESCEN to set the           **
**           date and time callable services   **
**           century window                    **
```

```
      **                                               **
      ** In this example, CEESCEN is called to change **
      ** the start of the century window to 30 years  **
      ** before the system date. CEEQCEN is then      **
      ** called to query that the change made.  A     **
      ** message that this has been done is then      **
      ** displayed.                                   **
      **                                               **
      ***************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSCEN.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  STARTCW                PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLSCEN.
      ***************************************************
      ** Specify 30 as century start, and two-digit
      **     years will be assumed to lie in the
      **     100-year window starting 30 years before
      **     the system date.
      ***************************************************
           MOVE 30 TO STARTCW.

      ***************************************************
      ** Call CEESCEN to change the start of the century
      **     window.
      ***************************************************
           CALL 'CEESCEN' USING STARTCW, FC.
           IF NOT CEE000 of FC  THEN
               DISPLAY 'CEESCEN failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

       PARA-CBLQCEN.
      ***************************************************
      ** Call CEEQCEN to return the start of the century
      **     window
      ***************************************************
           CALL 'CEEQCEN' USING STARTCW, FC.

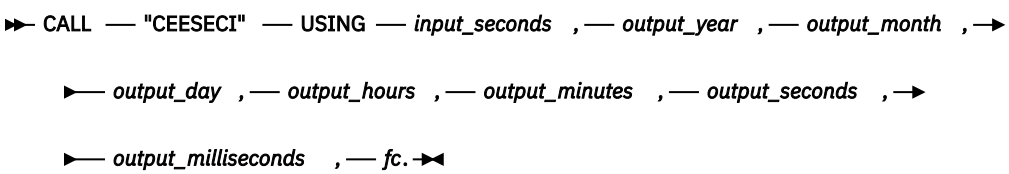
      ***************************************************
      ** CEEQCEN has no nonzero feedback codes to
      **     check, so just display result.
      ***************************************************
               DISPLAY 'The start of the century '
                   'window is: ' STARTCW
           GOBACK.
```

CEESECI: convert seconds to integers

CEESECI converts a number that represents the number of seconds since 00:00:00 14 October 1582 to binary integers that represent year, month, day, hour, minute, second, and millisecond.

Use CEESECI instead of CEEDATM when the output is needed in numeric format rather than in character format.

CALL CEESECI syntax



input_seconds
A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The range of valid values for *input_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

If *input_seconds* is invalid, all output parameters except the feedback code are set to 0.

***output_year* (output)**
A 32-bit binary integer that represents the year.

The range of valid values for *output_year* is 1582 to 9999, inclusive.

***output_month* (output)**
A 32-bit binary integer that represents the month.

The range of valid values for *output_month* is 1 to 12.

***output_day* (output)**
A 32-bit binary integer that represents the day.

The range of valid values for *output_day* is 1 to 31.

***output_hours* (output)**
A 32-bit binary integer that represents the hour.

The range of valid values for *output_hours* is 0 to 23.

***output_minutes* (output)**
A 32-bit binary integer that represents the minutes.

The range of valid values for *output_minutes* is 0 to 59.

***output_seconds* (output)**
A 32-bit binary integer that represents the seconds.

The range of valid values for *output_seconds* is 0 to 59.

***output_milliseconds* (output)**
A 32-bit binary integer that represents milliseconds.

The range of valid values for *output_milliseconds* is 0 to 999.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 69. ***CEESECI symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.

Usage notes

- The inverse of CEESECI is CEEISEC, which converts separate binary integers that represent year, month, day, hour, second, and millisecond to a number of seconds.
- If the input value is a Lilian date instead of seconds, multiply the Lilian date by 86,400 (number of seconds in a day), and pass the new value to CEESECI.

Example

```
      ************************************************
      **                                            **
      ** Function: Call CEESECI to convert seconds   **
      **           to integers                      **
      **                                            **
      ** In this example a call is made to CEESECI   **
      ** to convert a number representing the number **
      ** of seconds since 00:00:00 14 October 1582   **
      ** to seven binary integers representing year, **
      ** month, day, hour, minute, second, and       **
      ** millisecond.  The results are displayed in  **
      ** this example.                              **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSECI.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  INSECS                  COMP-2.
       01  YEAR                    PIC S9(9) BINARY.
       01  MONTH                   PIC S9(9) BINARY.
       01  DAYS                    PIC S9(9) BINARY.
       01  HOURS                   PIC S9(9) BINARY.
       01  MINUTES                 PIC S9(9) BINARY.
       01  SECONDS                 PIC S9(9) BINARY.
       01  MILLSEC                 PIC S9(9) BINARY.
       01  IN-DATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                       OCCURS 0 TO 256 TIMES
                       DEPENDING ON Vstring-length
                           of IN-DATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                       OCCURS 0 TO 256 TIMES
                       DEPENDING ON Vstring-length
                          of PICSTR.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                        REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLSECS.
      ************************************************
      ** Call CEESECS to convert time stamp of 6/2/88
      **     at 10:23:45 AM to Lilian representation
      ************************************************
           MOVE 20 TO Vstring-length of IN-DATE.
           MOVE '06/02/88 10:23:45 AM'
                   TO Vstring-text of IN-DATE.
           MOVE 20 TO Vstring-length of PICSTR.
           MOVE 'MM/DD/YY HH:MI:SS AP'
                   TO Vstring-text of PICSTR.
           CALL 'CEESECS' USING IN-DATE, PICSTR,
                                INSECS, FC.
           IF NOT CEE000 of FC  THEN
```

```
            DISPLAY 'CEESECS failed with msg '
                Msg-No of FC UPON CONSOLE
            STOP RUN
        END-IF.

     PARA-CBLSECI.
    ************************************************
    ** Call CEESECI to convert seconds to integers
    ************************************************
        CALL 'CEESECI' USING INSECS, YEAR, MONTH,
                             DAYS, HOURS,  MINUTES,
                             SECONDS, MILLSEC, FC.
    ************************************************
    ** If CEESECI runs successfully, display results
    ************************************************
        IF CEE000 of FC  THEN
            DISPLAY 'Input seconds of ' INSECS
                ' represents:'
            DISPLAY '   Year......... ' YEAR
            DISPLAY '   Month........ ' MONTH
            DISPLAY '   Day.......... ' DAYS
            DISPLAY '   Hour......... ' HOURS
            DISPLAY '   Minute....... ' MINUTES
            DISPLAY '   Second....... ' SECONDS
            DISPLAY '   Millisecond.. ' MILLSEC
        ELSE
            DISPLAY 'CEESECI failed with msg '
                Msg-No of FC UPON CONSOLE
            STOP RUN
        END-IF.

        GOBACK.
```

CEESECS: convert time stamp to seconds

CEESECS converts a string that represents a time stamp into Lilian seconds (the number of seconds since 00:00:00 14 October 1582). This service makes it easier to perform time arithmetic, such as calculating the elapsed time between two time stamps.

CALL CEESECS syntax

***input_timestamp* (input)**

A halfword length-prefixed character string that represents a date or time stamp in a format matching that specified by *picture_string*.

The character string must contain between 5 and 80 picture characters, inclusive. *input_timestamp* can contain leading or trailing blanks. Parsing begins with the first nonblank character (unless the picture string itself contains leading blanks; in this case, CEESECS skips exactly that many positions before parsing begins).

After a valid date is parsed, as determined by the format of the date you specify in *picture_string*, all remaining characters are ignored by CEESECS. Valid dates range between and including the dates 15 October 1582 to 31 December 9999. A full date must be specified. Valid times range from 00:00:00.000 to 23:59:59.999.

If any part or all of the time value is omitted, zeros are substituted for the remaining values. For example:

```
1992-05-17-19:02 is equivalent to 1992-05-17-19:02:00
1992-05-17       is equivalent to 1992-05-17-00:00:00
```

***picture_string* (input)**

A halfword length-prefixed character string, indicating the format of the date or time-stamp value specified in *input_timestamp*.

Each character in the *picture_string* represents a character in *input_timestamp*. For example, if you specify MMDDYY HH.MI.SS as the *picture_string*, CEESECS reads an *input_char_date* of 060288 15.35.02 as 3:35:02 PM on 02 June 1988. If delimiters such as the slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEESECS all assign the same value to data item *secs*:

```
CALL CEESECS USING '92/06/03 15.35.03',
                   'YY/MM/DD HH.MI.SS', secs, fc.
CALL CEESECS USING '92/6/3 15.35.03',
                   'YY/MM/DD HH.MI.SS', secs, fc.
CALL CEESECS USING '92/6/3 3.35.03 PM',
                   'YY/MM/DD HH.MI.SS AP', secs, fc.
CALL CEESECS USING '92.155 3.35.03 pm',
                   'YY.DDD   HH.MI.SS AP', secs, fc.
```

If *picture_string* includes a Japanese Era symbol <JJJJ>, the YY position in *input_timestamp* represents the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

***output_seconds* (output)**

A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds. For example, 00:00:01 on 15 October 1582 is second 86,401 (24*60*60 + 01) in the Lilian format. 19:00:01.12 on 16 May 1988 is second 12,799,191,601.12.

The largest value represented is 23:59:59.999 on 31 December 9999, which is second 265,621,679,999.999 in the Lilian format.

A 64-bit long floating-point value can accurately represent approximately 16 significant decimal digits without loss of precision. Therefore, accuracy is available to the nearest millisecond (15 decimal digits).

If *input_timestamp* does not contain a valid date or time stamp, *output_seconds* is set to 0 and CEESECS terminates with a non-CEE000 symbolic feedback code.

Elapsed time calculations are performed easily on the *output_seconds*, because it represents elapsed time. Leap year and end-of-year anomalies do not affect the calculations.

***fc* (output)**

A 12-byte feedback code (optional) that indicates the result of this service.

Table 70. ***CEESECS symbolic conditions***

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The era passed to CEEDAYS or CEESECS was not recognized.
CEE2EE	3	2510	The hours value in a call to CEEISEC or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.

Table 70. **CEESECS symbolic conditions** *(continued)*			
Symbolic feedback code	**Severity**	**Message number**	**Message text**
CEE2EK	3	2516	The minutes value in a CEEISEC call was not recognized.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EN	3	2519	The seconds value in a CEEISEC call was not recognized.
CEE2EP	3	2521	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.
CEE2ET	3	2525	CEESECS detected nonnumeric data in a numeric field, or the time-stamp string did not match the picture string.

Usage notes

- The inverse of CEESECS is CEEDATM, which converts *output_seconds* to character format.
- By default, two-digit years lie within the 100-year range that starts 80 years before the system date. Thus in 2010, all two-digit years represent dates between 1930 and 2029, inclusive. You can change this range by using the callable service CEESCEN.

Example

```
      ************************************************
      **                                            **
      ** Function: Call CEESECS to convert          **
      **           time stamp to number of seconds  **
      **                                            **
      ** In this example, calls are made to CEESECS **
      ** to convert two time stamps to the number   **
      ** of seconds since 00:00:00 14 October 1582. **
      ** The Lilian seconds for the earlier         **
      ** time stamp are then subtracted from the    **
      ** Lilian seconds for the later time stamp    **
      ** to determine the number of between the     **
      ** two.  This result is displayed.            **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSECS.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  SECOND1                 COMP-2.
       01  SECOND2                 COMP-2.
       01  TIMESTP.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of TIMESTP.
       01  TIMESTP2.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of TIMESTP2.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
```

```
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.

       PARA-SECS1.
      ***********************************************
      ** Specify first time stamp and a picture string
      **     describing the format of the time stamp
      **     as input to CEESECS
      ***********************************************
           MOVE 25 TO Vstring-length of TIMESTP.
           MOVE '1969-05-07 12:01:00.000'
                   TO Vstring-text of TIMESTP.
           MOVE 25 TO Vstring-length of PICSTR.
           MOVE 'YYYY-MM-DD HH:MI:SS.999'
                   TO Vstring-text of PICSTR.

      ***********************************************
      ** Call CEESECS to convert the first time stamp
      ** to Lilian seconds
      ***********************************************
           CALL 'CEESECS' USING TIMESTP, PICSTR,
                                SECOND1, FC.
           IF NOT CEE000 of FC  THEN
               DISPLAY 'CEESECS failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

       PARA-SECS2.
      ***********************************************
      ** Specify second time stamp and a picture string
      **     describing the format of the time stamp as
      **     input to CEESECS.
      ***********************************************
           MOVE 25 TO Vstring-length of TIMESTP2.
           MOVE '2004-01-01 00:00:01.000'
                   TO Vstring-text of TIMESTP2.
           MOVE 25 TO Vstring-length of PICSTR.
           MOVE 'YYYY-MM-DD HH:MI:SS.999'
                   TO Vstring-text of PICSTR.

      ***********************************************
      ** Call CEESECS to convert the second time stamp
      **     to Lilian seconds
      ***********************************************
           CALL 'CEESECS' USING TIMESTP2, PICSTR,
                                SECOND2, FC.
           IF NOT CEE000 of FC  THEN
               DISPLAY 'CEESECS failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

       PARA-SECS2.
      ***********************************************
      ** Subtract SECOND2 from SECOND1 to determine the
      **     number of seconds between the two time stamps
      ***********************************************
           SUBTRACT SECOND1 FROM SECOND2.
           DISPLAY 'The number of seconds between '
               Vstring-text OF TIMESTP ' and '
               Vstring-text OF TIMESTP2 ' is: ' SECOND2.

           GOBACK.
```

"Example: date-and-time picture strings" on page 462

Related references
“Picture character terms and strings” on page 460

CEEUTC: get coordinated universal time

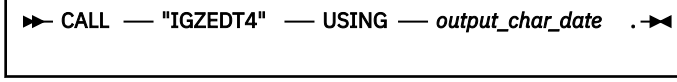
CEEUTC is identical to CEEGMT.

Related references
“CEEGMT: get current Greenwich Mean Time” on page 504

IGZEDT4: get current date

IGZEDT4 returns the current date with a four-digit year in the form YYYYMMDD.

CALL IGZEDT4 syntax



***output_char_date* (output)**
An 8-byte fixed-length character string in the form YYYYMMDD, which represents current year, month, and day.

Usage note: IGZEDT4 is not supported under CICS.

Example

```
************************************************
** Function: IGZEDT4 - get current date in the  **
**                     format YYYYMMDD.         **
************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLEDT4.
 . . .
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CHRDATE                PIC S9(8) USAGE DISPLAY.
 . . .
 PROCEDURE DIVISION.
 PARA-CBLEDT4.
************************************************
** Call IGZEDT4.
************************************************
     CALL 'IGZEDT4' USING BY REFERENCE CHRDATE.
************************************************
** IGZEDT4 has no nonzero return code to
**     check, so just display result.
************************************************
     DISPLAY 'The current date is: '
         CHRDATE
     GOBACK.
```

Appendix E. XML reference material

The following information describes the XML exception codes that might be returned during XML parsing or XML generation. The information also documents the well-formedness constraints from the *XML specification* that the parser checks.

Related references
"XML PARSE exceptions" on page 523
"XML conformance" on page 531
"XML GENERATE exceptions" on page 533
XML specification

XML PARSE exceptions

When an exception event occurs, the XML parser sets special register XML-CODE to a value that identifies the exception. Depending on the value in XML-CODE, the parser might or might not be able to continue processing after the exception, as detailed in the information referenced below.

Related references
"XML PARSE exceptions
that allow continuation" on page 523
"XML PARSE exceptions
that do not allow continuation" on page 528

XML PARSE exceptions that allow continuation

Whether the XML parser can continue processing after an exception event depends upon the value of the exception code.

The parser can continue processing if the exception code, which is in special register XML-CODE, is within one of the following ranges:

- 1 - 99
- 100,001 - 165,535
- 200,001 - 265,535

The following table describes each exception, and identifies the actions that the parser takes if you request that it continue after the exception. Some of the descriptions use the following terms:

- *Actual document encoding*
- *Document encoding declaration*
- *External ASCII code page*
- *External EBCDIC code page*

For definitions of the terms, see the related concept about XML input document encoding.

© Copyright IBM Corp. 2021

Table 71. ***XML PARSE exceptions that allow continuation***		
Exception code (decimal)	**Description**	**Parser action on continuation**
1	The parser found an invalid character while scanning white space outside element content. For further information about white space, see the related concept about XML input document encoding.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
3	The parser found a duplicate attribute name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
4	The parser found the markup character '<' in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
5	The start and end tag names of an element did not match.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
6	The parser found an invalid character in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
8	The parser found in element content the CDATA closing character sequence ']]>' without the matching opening character sequence '<![CDATA['.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

*Table 71. **XML PARSE exceptions that allow continuation** (continued)*

Exception code (decimal)	Description	Parser action on continuation
9	The parser found an invalid character in a comment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
10	The parser found in a comment the character sequence '- -' (two hyphens) not followed by '>'.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
11	The parser found an invalid character in a processing instruction data segment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
12	The XML declaration was not at the beginning of the document.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
13	The parser found an invalid digit in a hexadecimal character reference (of the form &#x*dddd*;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
14	The parser found an invalid digit in a decimal character reference (of the form &#*dddd*;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
15	The encoding declaration value in the XML declaration did not begin with lowercase or uppercase A through Z.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
16	A character reference did not refer to a legal XML character.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
17	The parser found an invalid character in an entity reference name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

*Table 71. **XML PARSE exceptions that allow continuation** (continued)*		
Exception code (decimal)	**Description**	**Parser action on continuation**
18	The parser found an invalid character in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
70	The actual document encoding was EBCDIC, and the external EBCDIC code page is supported, but the document encoding declaration did not specify a supported EBCDIC code page.	The parser uses the encoding specified by theexternal EBCDIC code page.
71	The actual document encoding was EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the external EBCDIC code page is not supported.	The parser uses the encoding specified by the document encoding declaration.
72	The actual document encoding was EBCDIC, the external EBCDIC code page is not supported, and the document did not contain an encoding declaration.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
73	The actual document encoding was EBCDIC, but neither the external EBCDIC code page nor the document encoding declaration specified a supported EBCDIC code page.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
80	The actual document encoding was ASCII, and the external ASCII code page is supported, but the document encoding declaration did not specify a supported ASCII code page.	The parser uses the encoding specified by the external ASCII code page.
81	The actual document encoding was ASCII, and the document encoding declaration specified a supported ASCII encoding, but the external ASCII code page is not supported.	The parser uses the encoding specified by the document encoding declaration.
82	The actual document encoding was ASCII, but the external ASCII code page is not supported, and the document did not contain an encoding declaration.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).

*Table 71. **XML PARSE exceptions that allow continuation** (continued)*

Exception code (decimal)	Description	Parser action on continuation
83	The actual document encoding was ASCII, but neither the external ASCII code page nor the document encoding declaration specified a supported ASCII code page.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
84	The actual document encoding was ASCII but not valid UTF-8, the external code page specified UTF-8, and the document did not contain an encoding declaration.	The parser uses UTF-8.
85	The actual document encoding was ASCII but not valid UTF-8, the external code page specified UTF-8, and the document encoding declaration specified neither a supported ASCII code page nor UTF-8.	The parser uses UTF-8.
86	The actual document encoding was ASCII but not valid UTF-8, the external code page specified a supported ASCII code page, and the document encoding declaration specified UTF-8.	The parser uses UTF-8.
87	The actual document encoding was ASCII but not valid UTF-8, and the external code page and the document encoding declaration both specified UTF-8.	The parser uses UTF-8.
88	The actual document encoding was ASCII but not valid UTF-8, the external code page specified neither a supported ASCII code page nor UTF-8, and the document encoding declaration specified UTF-8.	The parser uses UTF-8.
89	The actual document encoding was ASCII but not valid UTF-8, the external code page specified UTF-8, and the document encoding declaration specified a supported ASCII code page.	The parser uses UTF-8.
92	The document data item was alphanumeric, but the actual document encoding was Unicode UTF-16.	The parser uses code page 1200 (Unicode UTF-16).

*Table 71. **XML PARSE exceptions that allow continuation** (continued)*		
Exception code (decimal)	**Description**	**Parser action on continuation**
100,001 - 165,535	The external EBCDIC code page and the document encoding declaration specified different supported EBCDIC code pages. XML-CODE contains the code page CCSID for the encoding declaration plus 100,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the external EBCDIC code page. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 100,000), the parser uses this encoding.
200,001 - 265,535	The external ASCII code page and the document encoding declaration specified different supported ASCII code pages. XML-CODE contains the CCSID for the encoding declaration plus 200,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the external ASCII code page. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 200,000), the parser uses this encoding.

Related concepts
"XML-CODE" on page 343
"XML input document encoding" on page 346

Related tasks
"Handling XML PARSE exceptions" on page 349

XML PARSE exceptions that do not allow continuation

The XML parser cannot continue processing if any of the exceptions described below occurs.

No further events are returned from the parser for any of these exceptions even if the processing procedure sets XML-CODE to zero before passing control back to the parser. The parser transfers control to the statement in the ON EXCEPTION phrase, if specified, otherwise to the end of the XML PARSE statement.

*Table 72. **XML PARSE exceptions that do not allow continuation***	
Exception code (decimal)	**Description**
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.

*Table 72. **XML PARSE exceptions that do not allow continuation** (continued)*	
Exception code (decimal)	**Description**
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, '_', or ':'.
125	The first character of the first attribute name of an element was not a letter, '_', or ':'.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than '=' following an attribute name.
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, '_', or ':'.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a '>' following the '/'.
133	The first character of an element end tag name was not a letter, '_', or ':'.
134	An element end tag name was not terminated by a '>'.
135	The first character of an element name was not a letter, '_', or ':'.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, '_', or ':'.

*Table 72. **XML PARSE exceptions that do not allow continuation** (continued)*	
Exception code (decimal)	**Description**
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence '?>'.
141	The parser found an invalid character following '&' in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by '='.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.
148	'encoding' in the XML declaration was not followed by '='.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional `standalone` declaration in the XML declaration.
153	`standalone` in the XML declaration was not followed by =.
154	The `standalone` declaration value in the XML declaration is either missing or improperly delimited.
155	The `standalone` declaration value was neither 'yes' nor 'no' only.
156	The `standalone` declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
157	The parser found an invalid character following the `standalone` declaration value closing delimiter in the XML declaration.
158	The XML declaration was not terminated by the proper character sequence '?>', or contained an invalid attribute.
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
161	The parser found an invalid UTF-8 byte sequence.

Table 72. ***XML PARSE exceptions that do not allow continuation*** *(continued)*	
Exception code (decimal)	**Description**
162	The parser found a UTF-8 character that has a Unicode scalar value greater than x'FFFF'.
315	The *actual document encoding* was UTF-16 little-endian, which the parser does not support on this platform.
316	The actual document encoding was UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document might be damaged.
318	The actual document encoding was UTF-8, which the parser does not support.
320	The document data item was national, but the actual document encoding was EBCDIC.
321	The document data item was national, but the actual document encoding was ASCII.
322	The document data item was a native alphanumeric data item, but the actual document encoding was EBCDIC.
323	The document data item was a host alphanumeric data item, but the actual document encoding was ASCII.
324	The document data item was national, but the actual document encoding was UTF-8.
325	The document data item was a host alphanumeric data item, but the actual document encoding was UTF-8.
500 - 599	Internal error. Report the error to your service representative.

Related concepts
"XML-CODE" on page 343

Related tasks
"Handling XML PARSE exceptions" on page 349

XML conformance

The built-in COBOL XML parser that is included in COBOL for Linux is not a conforming XML processor according to the definition in the *XML specification*. The parser does not validate the XML documents that you parse. Although it does check for many well-formedness errors, it does not perform all of the actions required of a nonvalidating XML processor.

In particular, the parser does not process the internal document type definition (DTD internal subset). Thus it does not supply default attribute values, does not normalize attribute values, and does not include the replacement text of internal entities except for the predefined entities. Instead, it passes the entire document type declaration as the contents of `XML-TEXT` or `XML-NTEXT` for the `DOCUMENT-TYPE-DECLARATION` XML event, which allows the application to perform these actions if required.

The parser optionally lets programs continue processing an XML document after some errors. The purpose of allowing processing to continue is to facilitate the debugging of XML documents and processing procedures.

Recapitulating the definition in the *XML specification*, a textual object is a *well-formed* XML document if:

- Taken as a whole, it conforms to the grammar for XML documents.
- It meets all the explicit well-formedness constraints listed in the *XML specification*.

- Each parsed entity (piece of text) that is referenced directly or indirectly within the document is well formed.

The COBOL XML parser does check that documents conform to the XML grammar, except for any document type declaration. The declaration is supplied in its entirety, unchecked, to your application.

The following information is an annotation from the *XML specification*. The W3C is not responsible for any content not found at the original URL (www.w3.org/TR/REC-xml). All the annotations are non-normative and are shown in *italic*.

Copyright © 1994-2001 W3C® (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), All Rights Reserved. W3C liability, trademark, document use, and software licensing rules apply. (www.w3.org/Consortium/Legal/ipr-notice-20000612)

The *XML specification* also contains twelve explicit well-formedness constraints. The constraints that the COBOL XML parser checks partly or completely are shown in **bold** type:

1. Parameter Entities (PEs) in Internal Subset: "In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)"

   The parser does not process the internal DTD subset, so it does not enforce this constraint.

2. External Subset: "The external subset, if any, must match the production for extSubset."

   The parser does not process the external subset, so it does not enforce this constraint.

3. Parameter Entity Between Declarations: "The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl."

   The parser does not process the internal DTD subset, so it does not enforce this constraint.

4. **Element Type Match:** "The Name in an element's end-tag must match the element type in the start-tag."

   The parser enforces this constraint.

5. **Unique Attribute Specification:** "No attribute name may appear more than once in the same start-tag or empty-element tag."

   The parser partly supports this constraint by checking up to 10 attribute names in a given element for uniqueness. The application can check any attribute names beyond this limit.

6. No External Entity References: "Attribute values cannot contain direct or indirect entity references to external entities."

   The parser does not enforce this constraint.

7. No '<' in Attribute Values: "The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a '<'."

   The parser does not enforce this constraint.

8. **Legal Character:** "Characters referred to using character references must match the production for Char."

   The parser enforces this constraint.

9. Entity Declared: "In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with standalone='yes', for an entity reference that does not occur within the external subset or a parameter entity, the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration."

"Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'."

The parser does not enforce this constraint.

10. Parsed Entity: "An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type ENTITY or ENTITIES."

    The parser does not enforce this constraint.

11. No Recursion: "A parsed entity must not contain a recursive reference to itself, either directly or indirectly."

    The parser does not enforce this constraint.

12. In DTD: "Parameter-entity references may only appear in the DTD."

    The parser does not enforce this constraint, because the error cannot occur.

The preceding material is an annotation from the *XML specification*. The W3C is not responsible for any content not found at the original URL (www.w3.org/TR/REC-xml); all these annotations are non-normative. This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. The normative version of the specification is the English version found at the W3C site; any translated document may contain errors from the translation.

Related concepts
"XML parser in COBOL" on page 339

Related references
Extensible Markup Language (XML)
XML specification (Prolog and document type declaration)

XML GENERATE exceptions

One of several exception codes might be returned in the XML-CODE special register during XML generation. If one of these exceptions occurs, control is passed to the statement in the ON EXCEPTION phrase, or to the end of the XML GENERATE statement if you did not code an ON EXCEPTION phrase.

Table 73. ***XML GENERATE exceptions***

Exception code (decimal)	Description
400	The receiver was too small to contain the generated XML document. The COUNT IN data item, if specified, contains the count of character positions that were actually generated.
401	A multibyte data-name contained a character that, when converted to Unicode, was not valid in an XML element or attribute name.
402	The first character of a multibyte data-name, when converted to Unicode, was not valid as the first character of an XML element or attribute name.
403	The value of an OCCURS DEPENDING ON variable exceeded 16,777,215.
410	The CCSID page specified by the EBCDIC_CODEPAGE environment variable is not supported for conversion to Unicode.
411	The CCSID specified by the EBCDIC_CODEPAGE environment variable is not a supported single-byte EBCDIC code page.

*Table 73. **XML GENERATE exceptions** (continued)*	
Exception code (decimal)	**Description**
412	The receiver was native alphanumeric, but the encoding specified for the document was not UTF-8 or a supported single-byte ASCII code page.
413	The receiver was alphanumeric, but the runtime locale was not consistent with the compile-time locale.
414	The encoding specified for the XML document was invalid or was not a supported code page.
415	The receiver was national, but the encoding specified for the document was not UTF-16.
416	The XML namespace identifier contained invalid XML characters.
417	Element character content or an attribute value contained characters that are illegal in XML content. XML generation has continued, with the element tag name or the attribute name prefixed with 'hex.' and the original data value represented in the document in hexadecimal.
418	Substitution characters were generated by encoding conversion.
419	The XML namespace prefix was invalid.
420	The source data item included a multibyte name or multibyte content, and the receiver was native alphanumeric, but the encoding specified for the document was not UTF-8.
600-699	Internal error. Report the error to your service representative.

Related tasks

“Handling XML GENERATE exceptions” on page 364

Related references

Appendix F. EXIT compiler option

You can use the EXIT compiler option to provide user-supplied modules in place of various compiler functions. For details about processing of each exit module, error handling for exit modules, or using the EXIT option with CICS and SQL statements, see the following topics.

Related references
“User-exit work area and work area extension” on page 535
“Parameter list for exit modules” on page 536
“Processing of INEXIT” on page 537
“Processing of LIBEXIT” on page 538
“Processing of PRTEXIT” on page 539

“Processing
of MSGEXIT” on page 539
“Error handling
for exit modules” on page 546
“EXIT” on page 263

User-exit work area and work area extension

When you use one of the user exits, the compiler provides a work area and work area extension in which you can save the address of storage obtained by the exit module. Having such work areas lets the module be reentrant.

The user-exit work area (for use by INEXIT, LIBEXIT, and PRTEXIT) consists of 4 fullwords that reside on a fullword boundary. The user-exit work area extension (for use by MSGEXIT) consists of 8 fullwords also on a fullword boundary. These fullwords are initialized to binary zeros before the first exit routine is invoked, and are passed to the exit module in a parameter list. After initialization, the compiler makes no further reference to the work areas.

Related references
“Processing of INEXIT” on page 537
“Processing of LIBEXIT” on page 538
“Processing of PRTEXIT” on page 539

“Processing
of MSGEXIT” on page 539

© Copyright IBM Corp. 2021

Parameter list for exit modules

The compiler uses a structure, passed by reference, to communicate with the exit module.

Table 74. ***Parameter list for exit modules***

Parameter offset	Contains	Description of item
0	User-exit type	Halfword that identifies which user exit is to perform the operation: • 1=INEXIT • 2=LIBEXIT • 3=PRTEXIT • 5=Reserved • 6=MSGEXIT
2	Operation code	Halfword that indicates the type of operation: • 0=OPEN • 1=CLOSE • 2=GET • 3=PUT • 4=FIND • 5=MSGSEV: customize message severity
4	Return code	Fullword, set by the exit module, that indicates the success of the requested operation. For op codes 0 through 4: • 0=Successful • 4=End-of-data • 12=Failed For op code 5: • 0=Message not customized • 4=Message found and customized • 12=Operation failed
8	Record length	Fullword, set by the exit module, that indicates the length of the record being returned by the GET operation, or supplied by the PUT operation.
12	Address of record or *str2*	Fullword, either set by the exit module to the address of the record in a user-owned buffer for the GET operation, or set by the compiler to the address of the record of the PUT operation. *str2* applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.

*Table 74. **Parameter list for exit modules** (continued)*

Parameter offset	Contains	Description of item
16	User-exit work area	Four-fullword work area provided by the compiler for use by the user-exit module: • First word: for use by INEXIT • Second word: for use by LIBEXIT • Third word: for use by PRTEXIT
32	*Text-name*	Fullword that contains the address of a null-terminated string that contains the fully qualified *text-name*. Applies only to FIND. (Used only by LIBEXIT)
36	User exit parameter string	Fullword that contains the address of a six-element array, each element of which is a structure that contains a 2-byte length field followed by a 64-character string that contains the exit parameter string. The sixth element is the MSGEXIT string.
40	Type of source code line	Halfword (used only by INEXIT)
42	Statement indicator	Halfword (used only by INEXIT)
44	Statement column number	Halfword (used only by INEXIT)
46	Reserved	Halfword
48	User-exit work area extension	Eight-fullword work area provided by the compiler for use by the user-exit module: • First word: reserved • Second word: MSGEXIT
80	Message exit data	Three-halfword area provided by the compiler: • First halfword: the message number of the message to be customized • Second halfword: for a diagnostic message, the default severity; for a FIPS message, the FIPS category as a numeric code • Third halfword: the user-requested severity for the messages (-1 to indicate suppression)

Related references
"User-exit work area and work area extension" on page 535

Processing of INEXIT

If INEXIT is specified, the compiler loads the exit module (*mod1*) during initialization, and invokes the module using the OPEN operation code (op code). The module can then prepare its source for processing and pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler requires a source statement, the exit module is invoked with the GET op code.

The exit module then returns either the address and length of the next statement, or the end-of-data indication if no more source statements exist. When end-of-data occurs, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources that are related to its input.

The compiler uses a structure, passed by reference, to communicate with the exit module.

Related references
"Parameter list for exit modules" on page 536

Processing of LIBEXIT

If LIBEXIT is specified, the compiler loads the exit module (*mod2*) during initialization. The compiler makes calls to the module to obtain copybooks whenever COPY or BASIS statements are encountered.

The first call invokes the module with an OPEN op code. The module can then prepare the specified *library-name* for processing. The OPEN op code is also issued the first time a new *library-name* is specified. The exit module returns the status of the OPEN request to the compiler by passing a return code.

When the exit invoked with the OPEN op code returns, the compiler invokes the exit module with a GET op code, and the exit module passes the compiler the length and address of the record to be copied from the active copybook. The GET operation is repeated until the end-of-data indicator is passed to the compiler.

When end-of-data occurs, the compiler issues a CLOSE request so that the exit module can release any resources related to its input.

Nested COPY statements: Any record from the active copybook can contain a COPY statement. (However, nested COPY statements cannot contain the REPLACING phrase, and a COPY statement with the REPLACING phrase cannot contain nested COPY statements.) When a valid nested COPY statement is encountered, the compiler issues an OPEN and then a series of GET until the compiler receives an EOD from LIBEXIT.

Recursive calls cannot be made to *text-name*. That is, a copybook can be named only once in a set of nested COPY statements until the end-of-data for that copybook is reached.

When the exit module receives the OPEN request, it should push its control information about the active copybook onto a stack and then complete the requested action by OPEN. The newly requested *text-name* (or *basis-name*) becomes the active copybook.

Processing continues in the normal manner with a series of GET requests until the end-of-data indicator is passed to the compiler.

Note: The user exit should continue to pass an end-of-data indicator when the compiler issues a GET until the compiler issues a CLOSE command.

At end-of-data for the nested active copybook and when the compiler issues a CLOSE operation, the exit module should pop its control information from the stack. The next request from the compiler will be a GET. The user exit should continue where it has left off.

The compiler then invokes the exit module with a GET request, and the exit module must pass the same record that was passed previously from this copybook. The compiler verifies that the same record was passed, and then the processing continues with GET requests until the end-of-data indicator is passed.

The compiler uses a structure, passed by reference, to communicate with the exit module.

Related references
"Parameter list for exit modules" on page 536

Processing of PRTEXIT

If PRTEXIT is specified, the compiler loads the exit module (*mod3*) during initialization. The exit module is used in place of the SYSPRINT file.

The compiler invokes the module using the OPEN operation code (op code). The module can then prepare its output destination for processing and pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has a line to print, the exit module is invoked with the PUT op code. The compiler supplies the address and length of the record that is to be printed, and the exit module returns the status of the PUT request to the compiler by a return code. The first byte of the record to be printed contains an ANSI printer control character.

Before the compilation completes, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources that are related to its output destination.

The compiler uses a structure, passed by reference, to communicate with the exit module.

Related references
"Parameter list for exit modules" on page 536

Processing of MSGEXIT

The MSGEXIT module is used to customize compiler diagnostic messages and FIPS messages. The module can customize a message either by changing its severity or suppressing it.

If the MSGEXIT module assigns a severity to a FIPS message, the message is converted into a diagnostic message. (The message is shown in the summary of diagnostic messages in the listing.)

A MSGEXIT summary at the end of the compiler listing shows how many messages were changed in severity and how many messages were suppressed.

Table 75. ***MSGEXIT processing***

Action by compiler	**Action by exit module**
Loads the exit module (*mod5*) during initialization	
Calls the exit module with an OPEN operation code (op code)	Optionally processes *str5* and passes the status of the OPEN request to the compiler
Calls the exit module with a MSGSEV operation code (op code) when the compiler is about to issue a diagnostic message or FIPS message	One of the following actions: • Indicates no customization of the message (by setting return code to 0) • Specifies a new severity for (or suppression of) the message, and sets return code to 4 • Indicates that the operation failed (by setting return code to 12)
Calls the exit module with a CLOSE op code	Optionally frees storage and passes the status of the CLOSE request to the compiler
Deletes the exit module (*mod5*) during compiler termination	

"Example: MSGEXIT user exit" on page 542

Related tasks
"Customizing compiler-message severities" on page 540

Related references
"Parameter list for exit modules" on page 536

Customizing compiler-message severities

To change the severities of compiler messages or suppress compiler messages (including FIPS messages), do the steps described below.

1. Code and compile a COBOL program named ERRMSG. The program needs only a PROGRAM-ID paragraph, as described in the related task.
2. Review the ERRMSG listing, which contains a complete list of compiler messages with their message numbers, severities, and message text.
3. Decide which messages you want to customize.

   To understand the customizations that are possible, see the related reference about customizable compiler-message severities.
4. Code a MSGEXIT module to implement the customizations.

   a. Verify that the operation-code parameter indicates message-severity customization.

   b. Check the two input values in the message-exit-data parameter: the message number; and the default severity for a diagnostic message or the FIPS category for a FIPS message.

      The FIPS category is expressed as numeric code. For details, see the related reference about customizable compiler-message severities.

   c. For a message that you want to customize, set the user-requested severity in the message-exit-data parameter to indicate either:

      - A new message severity, by coding severity 0, 4, 8, or 12
      - Message suppression, by coding severity -1

   d. Set the return code to one of the following values:

      - 0, to indicate that the message was not customized
      - 4, to indicate that the message was found and customized
      - 12, to indicate that the operation failed and that compilation should be terminated
5. Compile and link your MSGEXIT module.

   Ensure that the module is linked as a shared library. For example:

   ```
   cob2 -o IGYMGXT -q32 IGYMSGXT.cbl -e IGYMSGXT
   ```
6. Set LD_LIBRARY_PATH to make the MSGEXIT module available to the compiler.

   For example, if the shared object is in `/u1/cobdev/exits`, use this command:

   ```
   export LD_LIBRARY_PATH=/u1/cobdev/exits:$LD_LIBRARY_PATH
   ```
7. Recompile program ERRMSG, but use compiler option EXIT(MSGEXIT(*msgmod*)), where *msgmod* is the name of your MSGEXIT module.
8. Review the listing and check for:

   - Updated message severities
   - Suppressed messages (indicated by XX in place of the severity)
   - Unsupported severity changes or unsupported message suppression (indicated by a severity-U diagnostic message, and compiler termination with return code 16)

Related tasks
"Generating a list of compiler messages" on page 228

Related references
"Runtime environment variables" on page 218
"Severity codes for

compiler diagnostic messages” on page 228
“Customizable
compiler-message severities” on page 541
“Effect
of message customization on compilation return code” on page 542
“Error handling
for exit modules” on page 546

Customizable compiler-message severities

To customize compiler-message severities, you need to understand the possible severities of compiler diagnostic messages, the levels or categories of FIPS messages, and the permitted customizations of message severities.

The possible severity codes for compiler diagnostic messages are described in the related reference about severity codes.

The eight categories of FIPS (FLAGSTD) messages are shown in the following table. The category of any given FIPS message is passed as a numeric code to the MSGEXIT module. Those numeric codes are shown in the second column.

Table 76. ***FIPS (FLAGSTD) message categories***

FIPS level or category	Numeric code	Description
D	81	Debug module level 1
E	82	Extension (IBM)
H	83	High level
I	84	Intermediate level
N	85	Segmentation module level 1
O	86	Obsolete elements
Q	87	High-level and obsolete elements
S	88	Segmentation module level 2

FIPS messages have an implied severity of zero (severity I).

Permitted message-severity customizations:

You can change the severity of a compiler message in the following ways:

- Severity-I and severity-W compiler diagnostic messages, and FIPS messages, can be changed to have any severity from I through S.

  Assigning a severity to a FIPS message converts the FIPS message to a diagnostic message of the assigned severity.

  As examples, you can:

  – Lower an optimizer warning to severity I.
  – Disallow `REDEFINING` a smaller item with a larger item by raising the severity of message 1154.
  – Disallow complex `OCCURS DEPENDING ON` by changing FIPS message 8235 from a category-E FIPS message to a severity-S compiler diagnostic message.
- Severity-E messages can be raised to severity S, but not lowered to severity I or W, because an error condition has occurred in the program.
- Severity-S and severity-U messages cannot be changed to have a different severity.

You can request suppression of compiler messages as follows:

- I, W, and FIPS messages can be suppressed.
- E and S messages cannot be suppressed.

Related references
“Severity codes for
compiler diagnostic messages” on page 228
“FLAGSTD” on page 266
“Effect
of message customization on compilation return code” on page 542

Effect of message customization on compilation return code

If you use a MSGEXIT module, the final return code from the compilation of a program could be affected as described below.

If you change the severity of a message, the return code from the compilation might also be changed. For example, if a compilation produces one diagnostic message, and it is a severity-E message, the compilation return code would normally be 8. But if the MSGEXIT module changes the severity of that message to severity S, then the return code from compilation would be 12.

If you suppress a message, the return code from the compilation is no longer affected by the severity of that message. For example, if a compilation produces one diagnostic message, and it is a severity-W message, the compilation return code would normally be 4. But if the MSGEXIT module suppresses that message, then the return code from compilation would be 0.

Related tasks
“Customizing compiler-message severities” on page 540

Related references
“Severity codes for
compiler diagnostic messages” on page 228

Example: MSGEXIT user exit

The following example shows a MSGEXIT user-exit module that changes message severities and suppresses messages.

You can also find the complete source code for the example in the `samples` subdirectory of the COBOL install directory (typically in `/opt/ibm/cobol/1.1.0/samples/msgexit`).

For helpful tips about using a message-exit module, see the comments within the code.

```
*****************************************************************
*  IGYMSGXT - Sample COBOL program for MSGEXIT                  *
*****************************************************************
*  Function:  This is a SAMPLE user exit for the MSGEXIT        *
*             suboption of the EXIT compiler option.  This exit *
*             can be used to customize the severity of or       *
*             suppress compiler diagnostic messages and FIPS    *
*             messages.  This example program includes several  *
*             sample customizations to show how customizations  *
*             are done.  Feel free to change the customizations *
*             as appropriate to meet your requirements.         *
*                                                               *
*---------------------------------------------------------------*
*  COMPILE NOTE: To prepare a compiler user exit in COBOL,      *
*                it should be a shared library module:          *
*   cob2 -o IGYMSGXT -q32 IGYMSGXT.cbl -e IGYMSGXT              *
*                                                               *
*  USAGE NOTE: The compiler needs to have access to IGYMSGXT at *
*              compile time, so set LD_LIBRARY_PATH accordingly:*
*                                                               *
*   EX:       export LD_LIBRARY_PATH=/u1/cobdev/exits:          *
*             $LD_LIBRARY_PATH                                  *
*                                                               *
*               (This assumes the shared object is in           *
*                 /u1/cobdev/exits )                            *
*                                                               *
```

```
*****************************************************************
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  IGYMSGXT.
 DATA DIVISION.

   WORKING-STORAGE SECTION.

*****************************************************************
*                                                               *
*   Local variables.                                            *
*                                                               *
*****************************************************************

    77 EXIT-TYPEN            PIC 9(4).
    77 EXIT-DEFAULT-SEV-FIPS PIC X.

*****************************************************************
*                                                               *
*   Definition of the User-Exit Parameter List, which is        *
*   passed from the COBOL compiler to the user-exit module      *
*                                                               *
*****************************************************************

   LINKAGE SECTION.
    01 UXPARM.
       02 EXIT-TYPE        PIC 9(4)   COMP.
       02 EXIT-OPERATION   PIC 9(4)   COMP.
       02 EXIT-RETURNCODE  PIC 9(9)   COMP.
       02 EXIT-DATALENGTH  PIC 9(9)   COMP.
       02 EXIT-DATA        POINTER.
       02 EXIT-WORK-AREA.
          03 EXIT-WORK-AREA-PTR  OCCURS 4  POINTER.
       02 EXIT-TEXT-NAME   POINTER.
       02 EXIT-PARMS       POINTER.
       02 EXIT-LINFO       PIC X(8).
       02 EXIT-X-WORK-AREA PIC X(4) OCCURS 8.
       02 EXIT-MESSAGE-PARMS.
          03 EXIT-MESSAGE-NUM PIC 9(4)   COMP.
          03 EXIT-DEFAULT-SEV PIC 9(4)   COMP.
          03 EXIT-USER-SEV    PIC S9(4)  COMP.

    01 EXIT-STRINGS.
       02 EXIT-STRING OCCURS 6.
          03 EXIT-STR-LEN PIC 9(4)   COMP.
          03 EXIT-STR-TXT PIC X(64).

*****************************************************************
*                                                               *
*  Begin PROCEDURE DIVISION                                     *
*                                                               *
*  Invoke the section to handle the exit.                       *
*                                                               *
*****************************************************************

 Procedure Division Using UXPARM.

     Set Address of EXIT-STRINGS to EXIT-PARMS

     COMPUTE EXIT-RETURNCODE = 0

     Evaluate
TRUE

*****************************************************************
* Handle a bad invocation of this exit by the compiler.         *
* This could happen if this routine was used for one of the     *
* other EXITs, such as INEXIT, PRTEXIT or LIBEXIT.              *
*****************************************************************
       When EXIT-TYPE Not = 6
       Move EXIT-TYPE   to  EXIT-TYPEN
         Display '**** Invalid exit routine identifier'
         Display '**** EXIT TYPE =  '  EXIT-TYPE
         Compute EXIT-RETURNCODE = 16

*****************************************************************
* Handle the OPEN call to this exit by the compiler             *
*        Display the exit string (labeled 'str5' in the syntax  *
*        diagram in the COBOL for Linux Programming Guide) from   *
*        the EXIT(MSGEXIT('str5',mod5)) option specification.   *
*        (Note that str5 is placed in element 6 of the array of *
*        user exit parameter strings.)                          *
```

```
*****************************************************************
       When EXIT-OPERATION = 0
*        Display 'Opening MSGEXIT'
*        If EXIT-STR-LEN(6) Not Zero Then
*          Display ' str5 len = ' EXIT-STR-LEN(6)
*          Display ' str5 = ' EXIT-STR-TXT(6)(1:EXIT-STR-LEN(6))
*        End-If
         Continue

*****************************************************************
* Handle the CLOSE call to this exit by the compiler            *
*        NOTE: Unlike the z/OS MSGEXIT, you must not use        *
*             STOP RUN here.  On Linux, use GOBACK.               *
*****************************************************************
       When EXIT-OPERATION = 1
*        Display 'Closing MSGEXIT'
         Goback

*****************************************************************
* Handle the customize message severity call to this exit       *
*        Display information about every customized severity.   *
*****************************************************************
       When EXIT-OPERATION = 5
*        Display 'MSGEXIT called with MSGSEV'
         If EXIT-MESSAGE-NUM < 8000 Then
           Perform Error-Messages-Severity
         Else
           Perform FIPS-Messages-Severity
         End-If

*        If EXIT-RETURNCODE = 4 Then
*          Display '>>>> Customizing message ' EXIT-MESSAGE-NUM
*                  ' with new severity ' EXIT-USER-SEV '  <<<<'
*          If EXIT-MESSAGE-NUM > 8000 Then
*            Display 'FIPS sev =' EXIT-DEFAULT-SEV-FIPS '<<<<'
*          End-If
*        End-If

*****************************************************************
* Handle a bad invocation of this exit by the compiler          *
* The compiler should not invoke this exit with EXIT-TYPE = 6   *
* and an opcode other than 0, 1, or 5.  This should not happen  *
* and IBM service should be contacted if it does.               *
*****************************************************************
       When Other
         Display '**** Invalid MSGEXIT routine operation '
         Display '**** EXIT OPCODE =  '  EXIT-OPERATION
         Compute EXIT-RETURNCODE = 16

     End-Evaluate


Goback.


*****************************************************************
*    ERROR MESSAGE   PROCESSOR                                  *
*****************************************************************
 Error-Messages-Severity.

*    Assume message severity will be customized...
     COMPUTE EXIT-RETURNCODE = 4

     Evaluate EXIT-MESSAGE-NUM

*****************************************************************
*      Change severity of message 1154(W) to 12 ("S")
*      This is the case of redefining a large item
*      with a smaller item, IBM Req # MR0904063236
*****************************************************************
       When(1154)
         COMPUTE EXIT-USER-SEV = 12

*****************************************************************
*      Message severity Not customized
*****************************************************************
       When Other
         COMPUTE EXIT-RETURNCODE = 0

     End-Evaluate
     .
*****************************************************************
```

```
*    FIPS MESSAGE   PROCESSOR                                   *
*****************************************************************
 Fips-Messages-Severity.

*    Assume message severity will be customized...
     COMPUTE EXIT-RETURNCODE = 4

*    Convert numeric 'category' to character
     EVALUATE EXIT-DEFAULT-SEV
       When 81
         MOVE 'D' To EXIT-DEFAULT-SEV-FIPS
       When 82
         MOVE 'E' To EXIT-DEFAULT-SEV-FIPS
       When 83
         MOVE 'H' To EXIT-DEFAULT-SEV-FIPS
       When 84
         MOVE 'I' To EXIT-DEFAULT-SEV-FIPS
       When 85
         MOVE 'N' To EXIT-DEFAULT-SEV-FIPS
       When 86
         MOVE 'O' To EXIT-DEFAULT-SEV-FIPS
       When 87
         MOVE 'Q' To EXIT-DEFAULT-SEV-FIPS
       When 88
         MOVE 'S' To EXIT-DEFAULT-SEV-FIPS
       When Other
         Continue
     End-Evaluate

*****************************************************************
*  Examples of using FIPS category to force coding
*  restrictions.  These are not recommendations!
*****************************************************************
*      Change severity of all OBSOLETE item FIPS
*       messages to 'S'
*****************************************************************
*    If EXIT-DEFAULT-SEV-FIPS = 'O' Then
*      DISPLAY ">>>> Default customizing FIPS category "
*        EXIT-DEFAULT-SEV-FIPS " msg " EXIT-MESSAGE-NUM "<<<<"
*      COMPUTE EXIT-USER-SEV = 12
*    End-If

     Evaluate EXIT-MESSAGE-NUM
*****************************************************************
*      Change severity of message 8062(O) to 8 ("E")
*        8062 = GO TO without proc name
*****************************************************************
       When(8062)
*        DISPLAY ">>>> Customizing message 8062 with 8 <<<<"
*        DISPLAY 'FIPS sev =' EXIT-DEFAULT-SEV-FIPS '='
         COMPUTE EXIT-USER-SEV = 8

*****************************************************************
*      Change severity of message 8193(E) to 0("I")
*        8193 = GOBACK
*****************************************************************
       When(8193)
*        DISPLAY ">>>> Customizing message 8193 with 0 <<<<"
*        DISPLAY 'FIPS sev =' EXIT-DEFAULT-SEV-FIPS '='
         COMPUTE EXIT-USER-SEV = 0

*****************************************************************
*      Change severity of message 8235(E) to 8 (Error)
*      to disalllow Complex Occurs Depending On
*        8235 = Complex Occurs Depending On
*****************************************************************
       When(8235)
*        DISPLAY ">>>> Customizing message 8235 with 8 <<<<"
*        DISPLAY 'FIPS sev =' EXIT-DEFAULT-SEV-FIPS '='
         COMPUTE EXIT-USER-SEV = 08

*****************************************************************
*      Change severity of message 8270(O) to -1 (Suppress)
*        8270 = SERVICE LABEL
*****************************************************************
       When(8270)
*        DISPLAY ">>>> Customizing message 8270 with -1 <<<<"
*        DISPLAY 'FIPS sev =' EXIT-DEFAULT-SEV-FIPS '='
         COMPUTE EXIT-USER-SEV = -1

*****************************************************************
*      Message severity Not customized
```

```
*****************************************************************
       When Other
*        For the default set '0' to 'S' case...
*        If EXIT-USER-SEV = 12 Then
*          COMPUTE EXIT-RETURNCODE = 4
*        Else
           COMPUTE EXIT-RETURNCODE = 0
*        End-If

     End-Evaluate
     .
 END PROGRAM IGYMSGXT.
```

Error handling for exit modules

The conditions described below can occur during processing of the user exits.

Exit load failure:

Message IGYSI5207-U is written to the operator if a LOAD request for any of the user exits fails:

```
An error occurred while attempting to load user exit exit-name.
```

Exit open failure:

Message IGYSI5208-U is written to the operator if an OPEN request for any of the user exits fails:

```
An error occurred while attempting to open user exit exit-name.
```

PRTEXIT PUT failure:

- Message IGYSI5203-U is written to the listing:

  ```
  A PUT request to the PRTEXIT user exit failed with return code nn.
  ```

- Message IGYSI5217-U is written to the operator:

  ```
  An error occurred in PRTEXIT user exit exit-name. Compiler terminated.
  ```

SYSIN GET failures:

The following messages might be written to the listing:

- IGYSI5204-U:

  ```
  The record address was not set by the exit-name user exit.
  ```

- IGYSI5205-U:

  ```
  A GET request from the INEXIT user exit failed with return code nn.
  ```

- IGYSI5206-U:

  ```
  The record length was not set by the exit-name user exit.
  ```

MSGEXIT failures:

Customization failure: Message IGY*PP*5293-U is written to the listing if an unsupported severity change or unsupported message suppression is attempted:

```
MSGEXIT user exit exit-name specified a message severity customization that is
not supported. The message number, default severity, and user-specified severity
were: mm, ds, us. Change MSGEXIT user exit exit-name to correct this error.
```

General failure: Message IGY*PP*5064-U is written to the listing if the MSGEXIT module sets the return code to a nonzero value other than 4:

```
A call to the MSGEXIT user exit routine exit-name failed with return code nn.
```

In the MSGEXIT messages, the two characters *PP* indicate the phase of the compiler that issued the message that resulted in a call to the MSGEXIT module.

Related tasks

“Customizing compiler-message severities” on page 540

Appendix G. Runtime messages

Messages for COBOL for Linux contain a message prefix, message number, severity code, and descriptive text.

The message prefix is always IWZ. The severity code is either I (information), W (warning), S (severe), or C (critical). The message text provides a brief explanation of the condition.

```
IWZ2519S  The seconds value in a CEEISEC call was not recognized.
```

In the example message above:

- The message prefix is IWZ.
- The message number is 2519.
- The severity code is S.
- The message text is "The seconds value in a CEEISEC call was not recognized."

The date and time callable services messages also contain a symbolic feedback code, which represents the first 8 bytes of a 12-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. The callable services messages contain a four-digit message number.

When running your application from the command line, you can capture any runtime messages by redirecting stdout and stderr to a file. For example:

program-name program-arguments >combined-output-file 2>&1

The following example shows how to write the output to separate files:

program-name program-arguments >output-file 2>error-file

Table 77. ***Runtime messages***

Message number	Message text
“IWZ006S” on page 556	The reference to table *table-name* by verb number *verb-number* on line *line-number* addressed an area outside the region of the table.
“IWZ007S” on page 556	The reference to variable-length group *group-name* by verb number *verb-number* on line *line-number* addressed an area outside the maximum defined length of the group.
“IWZ012I” on page 557	Invalid run unit termination occurred while sort or merge is running.
“IWZ013S” on page 557	Sort or merge requested while sort or merge is running in a different thread.
“IWZ026W” on page 557	The SORT-RETURN special register was never referenced, but the current content indicated the sort or merge operation in program *program-name* on line number *line-number* was unsuccessful. The sort or merge return code was *return code*.
“IWZ029S” on page 557	Argument-1 for function *function-name* in program *program-name* at line *line-number* was less than zero.
“IWZ030S” on page 558	Argument-2 for function *function-name* in program *program-name* at line *line-number* was not a positive integer.
“IWZ036W” on page 558	Truncation of high order digit positions occurred in program *program-name* on line number *line-number*.

© Copyright IBM Corp. 2021

*Table 77. **Runtime messages** (continued)*	
“IWZ037I” on page 558	The flow of control in program *program-name* proceeded beyond the last line of the program. Control returned to the caller of the program *program-name*.
“IWZ038S” on page 558	A reference modification length value of *reference-modification-value* on line *line-number* which was not equal to 1 was found in a reference to data item *data-item*.
“IWZ039S” on page 559	An invalid overpunched sign was detected.
“IWZ040S” on page 559	An invalid separate sign was detected.
“IWZ048W” on page 559	A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.
“IWZ049W” on page 560	A zero base was raised to a zero power in an exponentiation expression. The result was set to one.
“IWZ050S” on page 560	A zero base was raised to a negative power in an exponentiation expression.
“IWZ051S” on page 560	No significant digits remain in a fixed-point exponentiation operation in program *program-name* due to excessive decimal positions specified in the operands or receivers.
“IWZ053S” on page 560	An overflow occurred on conversion to floating point.
“IWZ054S” on page 560	A floating-point exception occurred.
“IWZ055W” on page 561	An underflow occurred on conversion to floating point. The result was set to zero.
“IWZ058S” on page 561	Exponent overflow occurred.
“IWZ059W” on page 561	An exponent with more than nine digits was truncated.
“IWZ060W” on page 561	Truncation of high order digit positions occurred.
“IWZ061S” on page 561	Division by zero occurred.
“IWZ063S” on page 562	An invalid sign was detected in a numeric edited sending field in *program-name* on line number *line-number*.
“IWZ064S” on page 562	A recursive call to active program *program-name* in compilation unit *compilation-unit* was attempted.
“IWZ065I” on page 562	A CANCEL of active program *program-name* in compilation unit *compilation-unit* was attempted.
“IWZ066S” on page 562	The length of external data record *data-record* did not match the existing length of the record.
“IWZ071S” on page 563	ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* had an ALL subscript specified for an OCCURS DEPENDING ON dimension, and the object was less than or equal to 0.

*Table 77. **Runtime messages** (continued)*	
“IWZ072S” on page 563	A reference modification start position value of *reference-modification-value* on line *line-number* referenced an area outside the region of data item *data-item*.
“IWZ073S” on page 563	A nonpositive reference modification length value of *reference-modification-value* on line *line-number* was found in a reference to data item *data-item*.
“IWZ074S” on page 563	A reference modification start position value of *reference-modification-value* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of data item *data-item*.
“IWZ075S” on page 564	Inconsistencies were found in EXTERNAL file *file-name* in program *program-name*. The following file attributes did not match those of the established external file: *attribute-1 attribute-2 attribute-3 attribute-4 attribute-5 attribute-6 attribute-7*.
“IWZ076W” on page 564	The number of characters in the INSPECT REPLACING CHARACTERS BY data-name was not equal to one. The first character was used.
“IWZ077W” on page 564	The lengths of the INSPECT data items were not equal. The shorter length was used.
“IWZ078S” on page 564	ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* will exceed the upper bound of the table.
“IWZ096C” on page 565	Message variants include: • Dynamic call of program *program-name* failed. A load of module *module-name* failed with an error code of *error-code*. • Dynamic call of program *program-name* failed. A load of module *module-name* failed with a return code of *return-code*. • Dynamic call of program *program-name* failed. Insufficient resources. • Dynamic call of program *program-name* failed. COBPATH not found in environment. • Dynamic call of program *program-name* failed. Entry *entry-name* not found. • Dynamic call failed. The name of the target program does not contain any valid characters. • Dynamic call of program *program-name* failed. The load module *load-module* could not be found in the directories identified in the COBPATH environment variable.
“IWZ097S” on page 565	Argument-1 for function *function-name* contained no digits.
“IWZ100S” on page 565	Argument-1 for function *function-name* was less than or equal to -1.
“IWZ103S” on page 566	Argument-1 for function *function-name* was less than zero or greater than 99.
“IWZ104S” on page 566	Argument-1 for function *function-name* was less than zero or greater than 99999.
“IWZ105S” on page 566	Argument-1 for function *function-name* was less than zero or greater than 999999.
“IWZ151S” on page 566	Argument-1 for function *function-name* contained more than 18 digits.

*Table 77. **Runtime messages** (continued)*	
“IWZ152S” on page 566	Invalid character *character* was found in column *column-number* in argument-1 for function *function-name*.
“IWZ155S” on page 566	Invalid character *character* was found in column *column-number* in argument-2 for function *function-name*.
“IWZ156S” on page 567	Argument-1 for function *function-name* was less than zero or greater than 28.
“IWZ157S” on page 567	The length of Argument-1 for function *function-name* was not equal to 1.
“IWZ158S” on page 567	Argument-1 for function *function-name* was less than zero or greater than 29.
“IWZ159S” on page 567	Argument-1 for function *function-name* was less than 1 or greater than 3067671.
“IWZ160S” on page 567	Argument-1 for function *function-name* was less than 16010101 or greater than 99991231.
“IWZ161S” on page 567	Argument-1 for function *function-name* was less than 1601001 or greater than 9999365.
“IWZ162S” on page 568	Argument-1 for function *function-name* was less than 1 or greater than the number of positions in the program collating sequence.
“IWZ163S” on page 568	Argument-1 for function *function-name* was less than zero.
“IWZ165S” on page 568	A reference modification start position value of *start-position-value* on line *line number* referenced an area outside the region of the function result of *function-result*.
“IWZ166S” on page 568	A nonpositive reference modification length value of *length* on line *line-number* was found in a reference to the function result of *function-result*.
“IWZ167S” on page 569	A reference modification start position value of *start-position* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of the function result of *function-result*.
“IWZ168W” on page 569	SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.
“IWZ169S” on page 569	Unknown device type for DISPLAY statement.
“IWZ170S” on page 569	Illegal data type for DISPLAY operand.
“IWZ171I” on page 570	*string-name* is not a valid runtime option.
“IWZ172I” on page 570	The string *string-name* is not a valid suboption of the runtime option *option-name*.
“IWZ173I” on page 570	The suboption string *string-name* of the runtime option *option-name* must be *number* characters long. The default will be used.
“IWZ174I” on page 570	The suboption string *string-name* of the runtime option *option-name* contains one or more invalid characters. The default will be used.
“IWZ175S” on page 570	There is no support for routine *routine-name* on this system.

*Table 77. **Runtime messages** (continued)*	
“IWZ176S” on page 570	Argument-1 for function *function-name* was greater than *decimal-value*.
“IWZ177S” on page 571	Argument-2 for function *function-name* was equal to *decimal-value*.
“IWZ178S” on page 571	Argument-1 for function *function-name* was less than or equal to *decimal-value*.
“IWZ179S” on page 571	Argument-1 for function *function-name* was less than *decimal-value*.
“IWZ180S” on page 571	Argument-1 for function *function-name* was not an integer.
“IWZ181I” on page 571	An invalid character was found in the numeric string *string* of the runtime option *option-name*. The default will be used.
“IWZ182I” on page 571	The number *number* of the runtime option *option-name* exceeded the range of *min-range* to *max-range*. The default will be used.
“IWZ183S” on page 572	The function name in _IWZCOBOLInit did a return.
“IWZ200S” on page 572	Error detected during *I/O operation* for file *file-name*. File status is: *file-status*.
“IWZ200S” on page 572	STOP or ACCEPT failed with an I/O error, *error-code*. The run unit is terminated.
“IWZ201C” on page 573	
“IWZ203S” on page 573	The code page in effect is not a DBCS code page.
“IWZ204S” on page 573	An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.
“IWZ221S” on page 574	ICU converter for the code page, *codepage value,* can not be opened. The error code is *error code value*.
“IWZ222S” on page 574	Data conversion via ICU failed with error code *error code value*.
“IWZ223W” on page 574	Close of ICU converter failed with error code *error code value*.
“IWZ224S” on page 574	ICU collator for the locale value, *locale value*, can not be opened. The error code is *error code value*.
“IWZ225S” on page 574	Unicode case mapping function using ICU failed with error code *error code value*. The locale in effect is *locale value*.
“IWZ230W” on page 575	The conversion table for the current code page, *ASCII codeset-id*, to the EBCDIC code page, *EBCDIC codeset-id*, is not available. The default ASCII to EBCDIC conversion table will be used.
“IWZ230W” on page 575	The EBCDIC code page specified, *EBCDIC codepage*, is not consistent with the locale *locale*, but will be used as requested.
“IWZ230W” on page 575	The EBCDIC code page specified, *EBCDIC codepage*, is not supported. The default EBCDIC code page, *EBCDIC codepage*, will be used.

*Table 77. **Runtime messages** (continued)*	
"IWZ230S" on page 575	The EBCDIC conversion table cannot be opened.
"IWZ230S" on page 575	The EBCDIC conversion table cannot be built.
"IWZ230S" on page 576	The main program was compiled with both the `-host` flag and the `CHAR(NATIVE)` option, which are not compatible.
"IWZ231S" on page 576	Query of current locale setting failed.
"IWZ232W" on page 576	Message variants include: • An error occurred during the conversion of data item *data-name* to EBCDIC in program *program-name* on line number *decimal-value*. • An error occurred during the conversion of data item *data-name* to ASCII in program *program-name* on line number *decimal-value*. • An error occurred during the conversion to EBCDIC for data item *data-name* in program *program-name* on line number *decimal-value*. • An error occurred during the conversion to ASCII for data item *data-name* in program *program-name* on line number *decimal-value*. • An error occurred during the conversion from ASCII to EBCDIC in program *program-name* on line number *decimal-value*. • An error occurred during the conversion from EBCDIC to ASCII in program *program-name* on line number *decimal-value*.
"IWZ240S" on page 577	The base year for program *program-name* was outside the valid range of 1900 through 1999. The sliding window value *window-value* resulted in a base year of *base-year*.
"IWZ241S" on page 577	The current year was outside the 100-year window, *year-start* through *year-end*, for program *program-name*.
"IWZ242S" on page 577	There was an invalid attempt to start an XML PARSE statement.
"IWZ243S" on page 577	There was an invalid attempt to end an XML PARSE statement.
"IWZ813S" on page 578	Insufficient storage was available to satisfy a get storage request.
"IWZ901S" on page 578	Message variants include: • Program exits due to severe or critical error. • Program exits: more than ERRCOUNT errors occurred.
"IWZ902S" on page 578	The system detected a Decimal-divide exception.
"IWZ903S" on page 578	The system detected a data exception.
"IWZ907S" on page 579	Message variants include: • Insufficient storage. • Insufficient storage. Cannot get *number-bytes* bytes of space for *storage*.

*Table 77. **Runtime messages** (continued)*	
“IWZ993W” on page 579	Insufficient storage. Cannot find space for message *message-number*.
“IWZ994W” on page 579	Cannot find message *message-number* in *message-catalog*.
“IWZ995C” on page 579	Message variants include: • *System exception* signal received while executing routine *routine-name* at offset 0x*offset-value*. • *System exception* signal received while executing code at location 0x*offset-value*. • *System exception* signal received. The location could not be determined.
“IWZ2502S” on page 580	The UTC/GMT was not available from the system.
“IWZ2503S” on page 580	The offset from UTC/GMT to local time was not available from the system.
“IWZ2505S” on page 580	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
“IWZ2506S” on page 580	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.
“IWZ2507S” on page 581	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
“IWZ2508S” on page 581	The date value passed to CEEDAYS or CEESECS was invalid.
“IWZ2509S” on page 581	The era passed to CEEDAYS or CEESECS was not recognized.
“IWZ2510S” on page 581	The hours value in a call to CEEISEC or CEESECS was not recognized.
“IWZ2511S” on page 582	The day parameter passed in a CEEISEC call was invalid for year and month specified.
“IWZ2512S” on page 582	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.
“IWZ2513S” on page 582	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
“IWZ2514S” on page 582	The year value passed in a CEEISEC call was not within the supported range.
“IWZ2515S” on page 583	The milliseconds value in a CEEISEC call was not recognized.
“IWZ2516S” on page 583	The minutes value in a CEEISEC call was not recognized.
“IWZ2517S” on page 583	The month value in a CEEISEC call was not recognized.
“IWZ2518S” on page 583	An invalid picture string was specified in a call to a date/time service.

*Table 77. **Runtime messages** (continued)*	
"IWZ2519S" on page 584	The seconds value in a CEEISEC call was not recognized.
"IWZ2520S" on page 584	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
"IWZ2521S" on page 584	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.
"IWZ2522S" on page 584	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.
"IWZ2525S" on page 585	CEESECS detected nonnumeric data in a numeric field, or the time-stamp string did not match the picture string.
"IWZ2526S" on page 585	The date string returned by CEEDATE was truncated.
"IWZ2527S" on page 585	The time-stamp string returned by CEEDATM was truncated.
"IWZ2531S" on page 585	The local time was not available from the system.
"IWZ2533S" on page 586	The value passed to CEESCEN was not between 0 and 100.
"IWZ2534W" on page 586	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

IWZ006S

The reference to table *table-name* by verb number *verb-number* on line *line-number* addressed an area outside the region of the table.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a fixed-length table has been subscripted in a way that exceeds the defined size of the table, or, for variable-length tables, the maximum size of the table.

The range check was performed on the composite of the subscripts and resulted in an address outside the region of the table. For variable-length tables, the address is outside the region of the table defined when all OCCURS DEPENDING ON objects are at their maximum values; the ODO object's current value is not considered. The check was not performed on individual subscripts.

Programmer response: Ensure that the value of literal subscripts and the value of variable subscripts as evaluated at run time do not exceed the subscripted dimensions for subscripted data in the failing statement.

System action: The application was terminated.

IWZ007S

The reference to variable-length group *group-name* by verb number *verb-number* on line *line-number* addressed an area outside the maximum defined length of the group.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a variable-length group generated by OCCURS DEPENDING ON has a length that is less than zero, or is greater than the limits defined in the OCCURS DEPENDING ON clauses.

The range check was performed on the composite length of the group, and not on the individual OCCURS DEPENDING ON objects.

Programmer response: Ensure that OCCURS DEPENDING ON objects as evaluated at run time do not exceed the maximum number of occurrences of the dimension for tables within the referenced group item.

System action: The application was terminated.

IWZ012I

Invalid run unit termination occurred while sort or merge is running.

Explanation: A sort or merge initiated by a COBOL program was in progress and one of the following was attempted:

1. A STOP RUN was issued.
2. A GOBACK or an EXIT PROGRAM was issued within the input procedure or the output procedure of the COBOL program that initiated the sort or merge. Note that the GOBACK and EXIT PROGRAM statements are allowed in a program called by an input procedure or an output procedure.

Programmer response: Change the application so that it does not use one of the above methods to end the sort or merge.

System action: The application was terminated.

IWZ013S

Sort or merge requested while sort or merge is running in a different thread.

Explanation: Running sort or merge in two or more threads at the same time is not supported.

Programmer response: Always run sort or merge in the same thread. Alternatively, include code before each call to the sort or merge that determines if sort or merge is running in another thread. If sort or merge is running in another thread, then wait for that thread to finish. If it isn't, then set a flag to indicate sort or merge is running and call sort or merge.

System action: The thread is terminated.

IWZ026W

The SORT-RETURN special register was never referenced, but the current content indicated the sort or merge operation in program *program-name* on line number *line-number* was unsuccessful. The sort or merge return code was *return code*.

Explanation: The COBOL source does not contain any references to the SORT-RETURN register. The compiler generates a test after each sort or merge verb. A nonzero return code has been passed back to the program by Sort or Merge.

Programmer response: Determine why the Sort or Merge was unsuccessful and fix the problem. See “Sort and merge error numbers” on page 159 for the list of possible return codes.

System action: No system action was taken.

IWZ029S

Argument-1 for function *function-name* in program *program-name* at line *line-number* was less than zero.

Explanation: An illegal value for argument-1 was used.

Programmer response: Ensure that argument-1 is greater than or equal to zero.

System action: The application was terminated.

IWZ030S

Argument-2 for function *function-name* in program *program-name* at line *line-number* was not a positive integer.

Explanation: An illegal value for argument-1 was used.

Programmer response: Ensure that argument-2 is a positive integer.

System action: The application was terminated.

IWZ036W

Truncation of high order digit positions occurred in program *program-name* on line number *line-number*.

Explanation: The generated code has truncated an intermediate result (that is, temporary storage used during an arithmetic calculation) to 30 digits; some of the truncated digits were not 0.

Programmer response: See the Related concepts at the end of this section for a description of intermediate results.

System action: No system action was taken.

IWZ037I

The flow of control in program *program-name* proceeded beyond the last line of the program. Control returned to the caller of the program *program-name*.

Explanation: The program did not have a terminator (STOP, GOBACK, or EXIT), and control fell through the last instruction.

Programmer response: Check the logic of the program. Sometimes this error occurs because of one of the following logic errors:

- The last paragraph in the program was only supposed to receive control as the result of a PERFORM statement, but due to a logic error it was branched to by a GO TO statement.
- The last paragraph in the program was executed as the result of a "fall-through" path, and there was no statement at the end of the paragraph to end the program.

System action: No system action was taken.

IWZ038S

A reference modification length value of *reference-modification-value* on line *line-number* which was not equal to 1 was found in a reference to data item *data-item*.

Explanation: The length value in a reference modification specification was not equal to 1. The length value must be equal to 1.

Programmer response: Check the indicated line number in the program to ensure that any reference modified length values are (or will resolve to) 1.

System action: The application was terminated.

IWZ039S

An invalid overpunched sign was detected.

Explanation: The value in the sign position was not valid.

Given X'*sd*', where *s* is the sign representation and *d* represents the digit, the valid sign representations for external decimal (`USAGE DISPLAY` without the `SIGN IS SEPARATE` clause) are:

Positive:	0, 1, 2, 3, 8, 9, A, and B
Negative:	4, 5, 6, 7, C, D, E, and F

Signs generated internally are 3 for positive and unsigned, and 7 for negative.

Given X'*ds*', where *d* represents the digit and *s* is the sign representation, the valid sign representations for internal decimal (`USAGE PACKED-DECIMAL`) COBOL data are:

Positive:	A, C, E, and F
Negative:	B and D

Signs generated internally are C for positive and unsigned, and D for negative.

Programmer response: This error might have occurred because of a REDEFINES clause involving the sign position or a group move involving the sign position, or the position was never initialized. Check for the above cases.

System action: The application was terminated.

IWZ040S

An invalid separate sign was detected.

Explanation: An operation was attempted on data defined with a separate sign. The value in the sign position was not a plus (+) or a minus (-).

Programmer response: This error might have occurred because of a REDEFINES clause involving the sign position or a group move involving the sign position, or the position was never initialized. Check for the above cases.

System action: The application was terminated.

IWZ048W

A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.

Explanation: A negative number raised to a fractional power occurred in a library routine.

The value of a negative number raised to a fractional power is undefined in COBOL. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present, so the absolute value of the base was used in the exponentiation.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: No system action was taken.

IWZ049W

A zero base was raised to a zero power in an exponentiation expression. The result was set to one.

Explanation: The value of zero raised to the power zero occurred in a library routine.

The value of zero raised to the power zero is undefined in COBOL. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present, so the value returned was one.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: No system action was taken.

IWZ050S

A zero base was raised to a negative power in an exponentiation expression.

Explanation: The value of zero raised to a negative power occurred in a library routine.

The value of zero raised to a negative number is not defined. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ051S

No significant digits remain in a fixed-point exponentiation operation in program *program-name* due to excessive decimal positions specified in the operands or receivers.

Explanation: A fixed-point calculation produced a result that had no significant digits because the operands or receiver had too many decimal positions.

Programmer response: Modify the PICTURE clauses of the operands or the receiving numeric item as needed to have additional integer positions and fewer decimal positions.

System action: The application was terminated.

IWZ053S

An overflow occurred on conversion to floating point.

Explanation: A number was generated in the program that is too large to be represented in floating point.

Programmer response: You need to modify the program appropriately to avoid an overflow.

System action: The application was terminated.

IWZ054S

A floating point exception occurred.

Explanation: A floating-point calculation has produced an illegal result. Floating-point calculations are done using IEEE floating-point arithmetic, which can produce results called NaN (Not a Number). For example, the result of 0 divided by 0 is NaN.

Programmer response: Modify the program to test the arguments to this operation so that NaN is not produced.

System action: The application was terminated.

IWZ055W

An underflow occurred on conversion to floating point. The result was set to zero.

Explanation: On conversion to floating point, the negative exponent exceeded the limit of the hardware. The floating-point value was set to zero.

Programmer response: No action is necessary, although you may want to modify the program to avoid an underflow.

System action: No system action was taken.

IWZ058S

Exponent overflow occurred.

Explanation: Floating-point exponent overflow occurred in a library routine.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ059W

An exponent with more than nine digits was truncated.

Explanation: Exponents in fixed point exponentiations may not contain more than nine digits. The exponent was truncated back to nine digits; some of the truncated digits were not 0.

Programmer response: No action is necessary, although you may want to adjust the exponent in the failing statement.

System action: No system action was taken.

IWZ060W

Truncation of high-order digit positions occurred.

Explanation: Code in a library routine has truncated an intermediate result (that is, temporary storage used during an arithmetic calculation) back to 30 digits; some of the truncated digits were not 0.

Programmer response: See the Related concepts at the end of this section for a description of intermediate results.

System action: No system action was taken.

IWZ061S

Division by zero occurred.

Explanation: Division by zero occurred in a library routine. Division by zero is not defined. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ063S

An invalid sign was detected in a numeric edited sending field in *program-name* on line number *line-number*.

Explanation: An attempt has been made to move a signed numeric edited field to a signed numeric or numeric edited receiving field in a MOVE statement. However, the sign position in the sending field contained a character that was not a valid sign character for the corresponding PICTURE.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ064S

A recursive call to active program *program-name* in compilation unit *compilation-unit* was attempted.

Explanation: COBOL does not allow reinvocation of an internal program which has begun execution, but has not yet terminated. For example, if internal programs A and B are siblings of a containing program, and A calls B and B calls A, this message will be issued.

Programmer response: Examine your program to eliminate calls to active internal programs.

System action: The application was terminated.

IWZ065I

A CANCEL of active program *program-name* in compilation unit *compilation-unit* was attempted.

Explanation: An attempt was made to cancel an active internal program. For example, if internal programs A and B are siblings in a containing program and A calls B and B cancels A, this message will be issued.

Programmer response: Examine your program to eliminate cancellation of active internal programs.

System action: The application was terminated.

IWZ066S

The length of external data record *data-record* in program *program-name* did not match the existing length of the record.

Explanation: While processing External data records during program initialization, it was determined that an External data record was previously defined in another program in the run-unit, and the length of the record as specified in the current program was not the same as the previously defined length.

Programmer response: Examine the current file and ensure the External data records are specified correctly.

System action: The application was terminated.

IWZ071S

ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* had an ALL subscript specified for an OCCURS DEPENDING ON dimension, and the object was less than or equal to 0.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that there are 0 occurrences of dimension subscripted by ALL.

The check is performed against the current value of the `OCCURS DEPENDING ON` object.

Programmer response: Ensure that ODO objects of ALL-subscripted dimensions of any subscripted items in the indicated statement are positive.

System action: The application was terminated.

IWZ072S

A reference modification start position value of *reference-modification-value* on line *line-number* referenced an area outside the region of data item *data-item*.

Explanation: The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the data item that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified data item.

Programmer response: Check the value of the starting position in the reference modification specification.

System action: The application was terminated.

IWZ073S

A nonpositive reference modification length value of *reference-modification-value* on line *line-number* was found in a reference to data item *data-item*.

Explanation: The length value in a reference modification specification was less than or equal to 0. The length value must be a positive integer.

Programmer response: Check the indicated line number in the program to ensure that any reference modified length values are (or will resolve to) positive integers.

System action: The application was terminated.

IWZ074S

A reference modification start position value of *reference-modification-value* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of data item *data-item*.

Explanation: The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified data item. The sum of the starting position and

length value minus one must be less than or equal to the number of characters in the reference modified data item.

Programmer response: Check the indicated line number in the program to ensure that any reference modified start and length values are set such that a reference is not made beyond the rightmost character of the data item.

System action: The application was terminated.

IWZ075S

Inconsistencies were found in EXTERNAL file *file-name* in program *program-name*. The following file attributes did not match those of the established external file: *attribute-1 attribute-2 attribute-3 attribute-4 attribute-5 attribute-6 attribute-7*.

Explanation: One or more attributes of an external file did not match between two programs that defined it.

Programmer response: Correct the external file. For a summary of file attributes that must match between definitions of the same external file, see the *COBOL for Linux on x86 Language Reference*.

System Action: The application was terminated.

IWZ076W

The number of characters in the INSPECT REPLACING CHARACTERS BY data-name was not equal to one. The first character was used.

Explanation: A data item which appears in a CHARACTERS phrase within a REPLACING phrase in an INSPECT statement must be defined as being one character in length. Because of a reference modification specification for this data item, the resultant length value was not equal to one. The length value is assumed to be one.

Programmer response: You may correct the reference modification specifications in the failing INSPECT statement to ensure that the reference modification length is (or will resolve to) 1; programmer action is not required.

System action: No system action was taken.

IWZ077W

The lengths of the INSPECT data items were not equal. The shorter length was used.

Explanation: The two data items which appear in a REPLACING or CONVERTING phrase in an INSPECT statement must have equal lengths, except when the second such item is a figurative constant. Because of the reference modification for one or both of these data items, the resultant length values were not equal. The shorter length value is applied to both items, and execution proceeds.

Programmer response: You may adjust the operands of unequal length in the failing INSPECT statement; programmer action is not required.

System action: No system action was taken.

IWZ078S

ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* will exceed the upper bound of the table.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a multidimensional table with ALL specified as one or more of the subscripts will result in a reference beyond the upper limit of the table.

The range check was performed on the composite of the subscripts and the maximum occurrences for the ALL subscripted dimensions. For variable-length tables the address is outside the region of the table defined when all OCCURS DEPENDING ON objects are at their maximum values; the ODO object's current value is not considered. The check was not performed on individual subscripts.

Programmer response: Ensure that OCCURS DEPENDING ON objects as evaluated at run time do not exceed the maximum number of occurrences of the dimension for table items referenced in the failing statement.

System action: The application was terminated.

IWZ096C

Message variants include:

- **Dynamic call of program *program-name* failed. A load of module *module-name* failed with an error code of *error-code*.**
- **Dynamic call of program *program-name* failed. A load of module *module-name* failed with a return code of *return-code*.**
- **Dynamic call of program *program-name* failed. Insufficient resources.**
- **Dynamic call of program *program-name* failed. COBPATH not found in environment.**
- **Dynamic call of program *program-name* failed. Entry *entry-name* not found.**
- **Dynamic call failed. The name of the target program does not contain any valid characters.**
- **Dynamic call of program *program-name* failed. The load module *load-module* could not be found in the directories identified in the COBPATH environment variable.**

Explanation: A dynamic call failed due to one of the reasons listed in the message variants above. In the above, the value of *error-code* is the errno set by `load`.

Programmer response: Check that COBPATH is defined. Check that the module exists. Check that the name of the module to be loaded matches the name of the entry called. Check that the module to be loaded is built correctly using the appropriate `cob2` options.

System action: The application was terminated.

IWZ097S

Argument-1 for function *function-name* contained no digits.

Explanation: Argument-1 for the indicated function must contain at least 1 digit.

Programmer response: Adjust the number of digits in Argument-1 in the failing statement.

System action: The application was terminated.

IWZ100S

Argument-1 for function *function* was less than or equal to -1.

Explanation: An illegal value was used for Argument-1.

Programmer response: Ensure that argument-1 is greater than -1.

System action: The application was terminated.

IWZ103S

Argument-1 for function *function-name* was less than zero or greater than 99.

Explanation: An illegal value was used for Argument-1.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ104S

Argument-1 for function *function-name* was less than zero or greater than 99999.

Explanation: An illegal value was used for Argument-1.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ105S

Argument-1 for function *function-name* was less than zero or greater than 999999.

Explanation: An illegal value was used for Argument-1.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ151S

Argument-1 for function *function-name* contained more than 18 digits.

Explanation: The total number of digits in Argument-1 of the indicated function exceeded 18 digits.

Programmer response: Adjust the number of digits in Argument-1 in the failing statement.

System action: The application was terminated.

IWZ152S

Invalid character *character* was found in column *column-number* in argument-1 for function *function-name*.

Explanation: A nondigit character other than a decimal point, comma, space or sign (+,-,CR,DB) was found in argument-1 for NUMVAL/NUMVAL-C function.

Programmer response: Correct argument-1 for NUMVAL or NUMVAL-C in the indicated statement.

System action: The application was terminated.

IWZ155S

Invalid character *character* was found in column *column-number* in argument-2 for function *function-name*.

Explanation: Illegal character was found in argument-2 for NUMVAL-C function.

Programmer response: Check that the function argument does follow the syntax rules.

System action: The application was terminated.

IWZ156S

Argument-1 for function *function-name* was less than zero or greater than 28.

Explanation: Input argument to function FACTORIAL is greater than 28 or less than 0.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ157S

The length of Argument-1 for function *function-name* was not equal to 1.

Explanation: The length of input argument to ORD function is not 1.

Programmer response: Check that the function argument is only 1 byte long.

System action: The application was terminated.

IWZ158S

Argument-1 for function *function-name* was less than zero or greater than 29.

Explanation: Input argument to function FACTORIAL is greater than 29 or less than 0.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ159S

Argument-1 for function *function-name* was less than 1 or greater than 3067671.

Explanation: The input argument to DATE-OF-INTEGER or DAY-OF-INTEGER function is less than 1 or greater than 3067671.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ160S

Argument-1 for function *function-name* was less than 16010101 or greater than 99991231.

Explanation: The input argument to function INTEGER-OF-DATE is less than 16010101 or greater than 99991231.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ161S

Argument-1 for function *function-name* was less than 1601001 or greater than 9999365.

Explanation: The input argument to function INTEGER-OF-DAY is less than 1601001 or greater than 9999365.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ162S

Argument-1 for function *function-name* was less than 1 or greater than the number of positions in the program collating sequence.

Explanation: The input argument to function CHAR is less than 1 or greater than the highest ordinal position in the program collating sequence.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ163S

Argument-1 for function *function-name* was less than zero.

Explanation: The input argument to function RANDOM is less than 0.

Programmer response: Correct the argument for function RANDOM in the failing statement.

System action: The application was terminated.

IWZ165S

A reference modification start position value of *start-position-value* on line *line number* referenced an area outside the region of the function result of *function-result*.

Explanation: The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the function result that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified function result.

Programmer response: Check the value of the starting position in the reference modification specification and the length of the actual function result.

System action: The application was terminated.

IWZ166S

A nonpositive reference modification length value of *length* on line *line-number* was found in a reference to the function result of *function-result*.

Explanation: The length value in a reference modification specification for a function result was less than or equal to 0. The length value must be a positive integer.

Programmer response: Check the length value and make appropriate correction.

System action: The application was terminated.

IWZ167S

A reference modification start position value of *start-position* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of the function result of *function-result*.

Explanation: The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified function result. The sum of the starting position and length value minus one must be less than or equal to the number of characters in the reference modified function result.

Programmer response: Check the length of the reference modification specification against the actual length of the function result and make appropriate corrections.

System action: The application was terminated.

IWZ168W

SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.

Explanation: COBOL environment names (such as SYSPUNCH/SYSPCH) are used as the environment variable names corresponding to the mnemonic names used on ACCEPT and DISPLAY statements. Set them equal to files, not existing directory names. To set environment variables, use the `export` command.

You can set environment variables either persistently or temporarily.

Programmer response: If you do not want SYSPUNCH/SYSPCH to default to the screen, set the corresponding environment variable.

System action: No system action was taken.

IWZ169S

Unknown device type for DISPLAY statement.

Explanation: An unknown device type was specified in *environment-name-1* or the environment name associated with *mnemonic-name-1* of the DISPLAY statement.

Programmer response: Specify a valid device type. For valid types, see the SPECIAL-NAMES paragraph.

System action: The application was terminated.

IWZ170S

Illegal data type for DISPLAY operand.

Explanation: An invalid data type was specified as the target of the DISPLAY statement.

Programmer response: Specify a valid data type. The following data types are **not** valid:

- Data items defined with `USAGE IS FUNCTION-POINTER`
- Data items defined with `USAGE IS PROCEDURE-POINTER`
- Data items or index names defined with `USAGE IS INDEX`

System action: The application was terminated.

IWZ171I

string-name is not a valid runtime option.

Explanation: *string-name* is not a valid option.

Programmer response: CHECK, DEBUG, ERRCOUNT, FILESYS, TRAP, and UPSI are valid runtime options.

System action: *string-name* is ignored.

IWZ172I

The string _string-name_ is not a valid suboption of the runtime option _option-name_.

Explanation: *string-name* was not in the set of recognized values.

Programmer response: Remove the invalid suboption *string* from the runtime option *option-name*.

System action: The invalid suboption is ignored.

IWZ173I

The suboption string _string-name_ of the runtime option _option-name_ must be _number of_ characters long. The default will be used.

Explanation: The number of characters for the suboption string *string-name* of runtime option *option-name* is invalid.

Programmer response: If you do not want to accept the default, specify a valid character length.

System action: The default value will be used.

IWZ174I

The suboption string _string-name_ of the runtime option _option-name_ contains one or more invalid characters. The default will be used.

Explanation: At least one invalid character was detected in the specified suboption.

Programmer response: If you do not want to accept the default, specify valid characters.

System action: The default value will be used.

IWZ175S

There is no support for routine _routine-name_ on this system.

Explanation: *routine-name* is not supported.

Programmer response:

System action: The application was terminated.

IWZ176S

Argument-1 for function _function-name_ was greater than _decimal-value_.

Explanation: An illegal value for argument-1 was used.

Programmer response: Ensure argument-1 is less than or equal to *decimal-value*.

System action: The application was terminated.

IWZ177S

Argument-2 for function *function-name* was equal to *decimal-value*.

Explanation: An illegal value for argument-2 was used.

Programmer response: Ensure argument-1 is not equal to *decimal-value*.

System action: The application was terminated.

IWZ178S

Argument-1 for function *function-name* was less than or equal to *decimal-value*.

Explanation: An illegal value for Argument-1 was used.

Programmer response: Ensure that Argument-1 is greater than *decimal-value*.

System action: The application was terminated.

IWZ179S

Argument-1 for function *function-name* was less than *decimal-value*.

Explanation: An illegal value for Argument-1 was used.

Programmer response: Ensure that Argument-1 is equal to or greater than *decimal-value*.

System action: The application was terminated.

IWZ180S

Argument-1 for function *function-name* was not an integer.

Explanation: An illegal value for Argument-1 was used.

Programmer response: Ensure that Argument-1 is an integer.

System action: The application was terminated.

IWZ181I

An invalid character was found in the numeric string *string* of the runtime option *option-name*. The default will be used.

Explanation: *string* did not contain all decimal numeric characters.

Programmer response: If you do not want the default value, correct the runtime option's string to contain all numeric characters.

System action: The default will be used.

IWZ182I

The number *number* of the runtime option *option-name* exceeded the range of *min-range* to *max-range*. The default will be used.

Explanation: *number* exceeded the range of *min-range* to *max-range*.

Programmer response: Correct the runtime option's string to be within the valid range.

System action: The default will be used.

IWZ183S

The function name in _iwzCOBOLInit did a return.

Explanation: The run unit termination exit routine returned to the function that invoked the routine (the function specified in `function_code`).

Programmer response: Rewrite the function so that the run unit termination exit routine does a longjump or exit instead of return to the function.

System action: The application was terminated.

IWZ200S

Error detected during *I/O operation* for file *file-name*. File status is: *file-status*.

Explanation: An error was detected during a file I/O operation. No file status was specified for the file and no applicable error declarative is in effect for the file.

Programmer response: Correct the condition described in this message. You can specify the FILE STATUS clause for the file if you want to detect the error and take appropriate actions within your source program.

System action: The application was terminated.

IWZ200S

STOP or ACCEPT failed with an I/O error, *error-code*. The run unit is terminated.

Explanation: A STOP or ACCEPT statement failed.

Programmer response: Check that the STOP or ACCEPT refers to a legitimate file or terminal device.

System action: The application was terminated.

IWZ201C

```
Message variants include:
Access Intent List Error.
Concurrent Opens Exceeds Maximum.
Cursor Not Selecting a Record Position.
Data Stream Syntax Error.
Duplicate Key Different Index.
Duplicate Key Same Index.
Duplicate Record Number.
File Temporarily Not Available.
File system cannot be found.
File Space Not Available.
File Closed with Damage.
Invalid Key Definition.
Invalid Base File Name.
Key Update Not Allowed by Different Index.
Key Update Not Allowed by Same Index.
No Update Intent on Record.
Not Authorized to Use Access Method.
Not Authorized to Directory.
Not Authorized to Function.
Not authorized to File.
Parameter Value Not Supported.
Parameter Not Supported.
Record Number Out of Bounds.
Record Length Mismatch.
Resource Limits Reached in Target System.
Resource Limits Reached in Source System.
```

```
Address Error.
Command Check.
Duplicate File Name.
End of File Condition.
Existing Condition.
File Handle Not Found.
Field Length Error.
File Not Found.
File Damaged.
File is Full.
File In Use.
Function Not Supported.
Invalid Access Method.
Invalid Data Record.
Invalid Key Length.
Invalid File Name.
Invalid Request.
Invalid Flag.
Object Not Supported.
Record Not Available.
Record Not Found.
Record Inactive.
Record Damaged.
Record In Use.
Update Cursor Error.
```

Explanation: An error was detected during an I/O operation for an STL file. No file status was specified for the file and no applicable error declarative is in effect for the file.

Programmer response: Correct the condition described in this message.

System action: The application was terminated.

IWZ203S

The code page in effect is not a DBCS code page.

Explanation: References to DBCS data were made with a non-DBCS code page in effect.

Programmer response: For DBCS data, specify a valid DBCS code page. Valid DBCS code pages are:

Country or region	Code page
Japan	IBM-932
Korea	IBM-1363
People's Republic of China (Simplified)	IBM-1386
Taiwan (Traditional)	

Note: The code pages listed above might not be supported for a specific version or release of that platform.

System Action: The application was terminated.

IWZ204S

An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.

Explanation: A Kanji or DBCS class test failed due to an error detected during the ASCII character string EBCDIC string conversion.

Programmer response: Verify that the locale in effect is consistent with the ASCII character string being tested. No action is likely to be required if the locale setting is correct. The class test is likely to indicate the string to be non-Kanji or non-DBCS correctly.

System action: The application was terminated.

IWZ221S

The ICU converter for the code page, *codepage value*, can not be opened. The error code is *error code value*.

Explanation: The ICU converter to convert between the code page and UTF-16 cannot be opened.

Programmer response: Verify that the code-page value identifies a primary or alias code-page name that is supported by ICU conversion libraries (see *International Components for Unicode: Converter Explorer*). If the code-page value is valid, contact your IBM representative.

System action: The application was terminated.

IWZ222S

Data conversion via ICU failed with error code *error code value*.

Explanation: The data conversion through ICU failed.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ223W

Close of ICU converter failed with error code *error code value*.

Explanation: The close of an ICU converter failed.

Programmer response: Contact your IBM representative.

System action: No system action was taken.

IWZ224S

ICU collator for the locale value, *locale value*, can not be opened. The error code is *error code value*.

Explanation: The ICU collator for the locale cannot be opened.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ225S

Unicode case mapping function using ICU failed with error code *error code value*. The locale in effect is *locale value*.

Explanation: The ICU case mapping function failed.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ230W

The conversion table for the current code page, *ASCII codeset-id*, to the EBCDIC code page, *EBCDIC codeset-id*, is not available. The default ASCII to EBCDIC conversion table will be used.

Explanation: The application has a module that was compiled with the CHAR(EBCDIC) compiler option. At run time a translation table will be built to handle the conversion from the current ASCII code page to an EBCDIC code page specified by the EBCDIC_CODEPAGE environment variable. This error occurred because either a conversion table is not available for the specified code pages, or the specification of the EBCDIC_CODE page is invalid. Execution will continue with a default conversion table based on ASCII code page IBM-1252 or equivalent and EBCDIC code page IBM-037 or equivalent.

Programmer response: Verify that the EBCDIC_CODEPAGE environment variable has a valid value.

If EBCDIC_CODEPAGE is not set, the default value, IBM-037, will be used. This is the default code page used by Enterprise COBOL for z/OS.

System action: No system action was taken.

IWZ230W

The EBCDIC code page specified, *EBCDIC codepage*, is not consistent with the locale *locale*, but will be used as requested.

Explanation: The application has a module that was compiled with the CHAR(EBCDIC) compiler option. This error occurred because the code page specified is not the same language as the current locale.

Programmer response: Verify that the EBCDIC_CODEPAGE environment variable is valid for this locale.

System action: No system action was taken.

IWZ230W

The EBCDIC code page specified, *EBCDIC codepage*, is not supported. The default EBCDIC code page, *EBCDIC codepage*, will be used.

Explanation: The application has a module that was compiled with the CHAR(EBCDIC) compiler option. This error occurred because the specification of the EBCDIC_CODEPAGE environment variable is invalid. Execution will continue with the default host code page that corresponds to the current locale.

Programmer response: Verify that the EBCDIC_CODEPAGE environment variable has a valid value.

System action: No system action was taken.

IWZ230S

The EBCDIC conversion table cannot be opened.

Explanation: The current system installation does not include the translation table for the default ASCII and EBCDIC code pages.

Programmer response: Reinstall the compiler and run time. If the problem still persists, call your IBM representative.

System action: The application was terminated.

IWZ230S

The EBCDIC conversion table cannot be built.

Explanation: The ASCII to EBCDIC conversion table has been opened, but the conversion failed.

Programmer response: Retry the execution from a new window.

System action: The application was terminated.

IWZ230S

The main program was compiled with both the `-host` flag and the `CHAR(NATIVE)` option, which are not compatible.

Explanation: Compilation with both the `-host` flag and the `CHAR(NATIVE)` option is not supported.

Programmer response: Either remove the `-host` flag, or remove the `CHAR(NATIVE)` option. The `-host` flag sets `CHAR(EBCDIC)`.

System action: The application was terminated.

IWZ231S

Query of current locale setting failed.

Explanation: A query of the execution environment failed to identify a valid locale setting. The current locale needs to be established to access appropriate message files and set the collating order. It is also used by the date/time services and for EBCDIC character support.

Programmer response: Check the settings for the following environment variable:

LANG
This should be set to a locale that has been installed on your machine. Enter `locale -a` to get a list of the valid values. The default value is en_US.

System action: The application was terminated.

IWZ232W

Message variants include:

- **An error occurred during the conversion of data item *data-name* to EBCDIC in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion of data item *data-name* to ASCII in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion to EBCDIC for data item *data-name* in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion to ASCII for data item *data-name* in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion from ASCII to EBCDIC in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion from EBCDIC to ASCII in program *program-name* on line number *decimal-value*.**

Explanation: The data in an identifier could not be converted between ASCII and EBCDIC formats as requested by the `CHAR(EBCDIC)` compiler option.

Programmer response: Check that the appropriate ASCII and EBCDIC locales are installed and selected. Check that the data in the identifier is valid and can be represented in both ASCII and EBCDIC format.

System action: No system action was taken. The data remains in its unconverted form.

IWZ240S

The base year for program *program-name* was outside the valid range of 1900 through 1999. The sliding window value *window-value* resulted in a base year of *base-year*.

Explanation: When the 100-year window was computed using the current year and the sliding window value specified with the YEARWINDOW compiler option, the base year of the 100-year window was outside the valid range of 1900 through 1999.

Programmer response: Examine the application design to determine if it will support a change to the YEARWINDOW option value. If the application can run with a change to the YEARWINDOW option value, then compile the program with an appropriate YEARWINDOW option value. If the application cannot run with a change to the YEARWINDOW option value, then convert all date fields to expanded dates and compile the program with NODATEPROC.

System action: The application was terminated.

IWZ241S

The current year was outside the 100-year window, *year-start* through *year-end*, for program *program-name*.

Explanation: The current year was outside the 100-year fixed window specified by the YEARWINDOW compiler option value.

For example, if a COBOL program is compiled with YEARWINDOW(1920), the 100-year window for the program is 1920 through 2019. When the program is run in the year 2020, this error message would occur since the current year is not within the 100-year window.

Programmer response: Examine the application design to determine if it will support a change to the YEARWINDOW option value. If the application can run with a change to the YEARWINDOW option value, then compile the program with an appropriate YEARWINDOW option value. If the application cannot run with a change to the YEARWINDOW option value, then convert all date fields to expanded dates and compile the program with NODATEPROC.

System action: The application was terminated.

IWZ242S

There was an invalid attempt to start an XML PARSE statement.

Explanation: An XML PARSE statement initiated by a COBOL program was already in progress when another XML PARSE statement was attempted by the same COBOL program. Only one XML PARSE statement can be active in a given invocation of a COBOL program.

Programmer response: Change the application so that it does not initiate another XML PARSE statement from within the same COBOL program.

System action: The application is terminated.

IWZ243S

There was an invalid attempt to end an XML PARSE statement.

Explanation: An XML PARSE statement initiated by a COBOL program was in progress and one of the following actions was attempted:

- A GOBACK or an EXIT PROGRAM statement was issued within the COBOL program that initiated the XML PARSE statement.
- A user handler associated with the program that initiated the XML PARSE statement moved the condition handler resume cursor and resumed the application.

Programmer response: Change the application so that it does not use one of the above methods to end the XML PARSE statement.

System action: The application is terminated.

IWZ813S

Insufficient storage was available to satisfy a get storage request.

Explanation: There was not enough free storage available to satisfy a get storage or reallocate request. This message indicates that storage management could not obtain sufficient storage from the operating system.

Programmer response: Ensure that you have sufficient storage available to run your application.

System action: No storage is allocated.

Symbolic feedback code: CEE0PD

IWZ901S

Message variants include:

- **Program exits due to severe or critical error.**
- **Program exits: more than ERRCOUNT errors occurred.**

Explanation: Every severe or critical message is followed by an IWZ901 message. An IWZ901 message is also issued if you used the ERRCOUNT runtime option and the number of warning messages exceeds ERRCOUNT.

Programmer response: See the severe or critical message, or increase ERRCOUNT.

System action: The application was terminated.

IWZ902S

The system detected a Decimal-divide exception.

Explanation: An attempt to divide a number by 0 was detected.

Programmer response: Modify the program. For example, add ON SIZE ERROR to the flagged statement.

System action: The application was terminated.

IWZ903S

The system detected a data exception.

Explanation: An operation on packed-decimal or zoned decimal data failed because the data contained an invalid value.

Programmer response: Verify the data is valid packed-decimal or zoned decimal data.

System action: The application was terminated.

IWZ907S

Message variants include:

- **Insufficient storage.**
- **Insufficient storage. Cannot get *number-bytes* bytes of space for *storage*.**

Explanation: The runtime library requested virtual memory space and the operating system denied the request.

Programmer response: Your program uses a large amount of virtual memory and it ran out of space. The problem is usually not due to a particular statement, but is associated with the program as a whole. Look at your use of OCCURS clauses and reduce the size of your tables.

System action: The application was terminated.

IWZ993W

Insufficient storage. Cannot find space for message *message-number*.

Explanation: The runtime library requested virtual memory space and the operating system denied the request.

Programmer response: Your program uses a large amount of virtual memory and it ran out of space. The problem is usually not due to a particular statement, but is associated with the program as a whole. Look at your use of OCCURS clauses and reduce the size of your tables.

System action: No system action was taken.

IWZ994W

Cannot find message *message-number* in *message-catalog*.

Explanation: The runtime library cannot find either the message catalog or a particular message in the message catalog.

Programmer response: Check that the COBOL library and messages were correctly installed and that LANG and NLSPATH are specified correctly.

System action: No system action was taken.

IWZ995C

Message variants include:

- ***system exception* signal received while executing routine *routine-name* at offset 0x*offset-value*.**
- ***system exception* signal received while executing code at location 0x*offset-value*.**
- ***system exception* signal received. The location could not be determined.**

Explanation: The operating system has detected an illegal action, such as an attempt to store into a protected area of memory or the operating system has detected that you pressed the interrupt key (typically the Control-C key, but it can be reconfigured).

Programmer response: If the signal was due to an illegal action, run the program under the debugger and it will give you more precise information as to where the error occurred. An example of this type of error is a pointer with an illegal value.

System action: The application was terminated.

IWZ2502S

The UTC/GMT was not available from the system.

Explanation: A call to CEEUTC or CEEGMT failed because the system clock was in an invalid state. The current time could not be determined.

Programmer response: Notify systems support personnel that the system clock is in an invalid state.

System action: All output values are set to 0.

Symbolic feedback code: CEE2E6

IWZ2503S

The offset from UTC/GMT to local time was not available from the system.

Explanation: A call to CEEGMTO failed because either (1) the current operating system could not be determined, or (2) the time zone field in the operating system control block appears to contain invalid data.

Programmer response: Notify systems support personnel that the local time offset stored in the operating system appears to contain invalid data.

System action: All output values are set to 0.

Symbolic feedback code: CEE2E7

IWZ2505S

The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.

Explanation: The input_seconds value passed in a call to CEEDATM or CEESECI was not a floating-point number between 86,400.0 and 265,621,679,999.999 The input parameter should represent the number of seconds elapsed since 00:00:00 on 14 October 1582, with 00:00:00.000 15 October 1582 being the first supported date/time, and 23:59:59.999 31 December 9999 being the last supported date/time.

Programmer response: Verify that input parameter contains a floating-point value between 86,400.0 and 265,621,679,999.999.

System action: For CEEDATM, the output value is set to blanks. For CEESECI, all output parameters are set to 0.

Symbolic feedback code: CEE2E9

IWZ2506S

An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.

Explanation: In a CEEDATM call, the picture string indicates that the input value is to be converted to an era; however the input value that was specified lies outside the range of supported eras.

Programmer response: Verify that the input value contains a valid number-of-seconds value within the range of supported eras.

System action: The output value is set to blanks.

IWZ2507S

Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.

Explanation: The picture string passed in a CEEDAYS or CEESECS call did not contain enough information. For example, it is an error to use the picture string 'MM/DD' (month and day only) in a call to CEEDAYS or CEESECS, because the year value is missing. The minimum information required to calculate a Lilian value is either (1) month, day and year, or (2) year and Julian day.

Programmer response: Verify that the picture string specified in a call to CEEDAYS or CEESECS specifies, as a minimum, the location in the input string of either (1) the year, month, and day, or (2) the year and Julian day.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EB

IWZ2508S

The date value passed to CEEDAYS or CEESECS was invalid.

Explanation: In a CEEDAYS or CEESECS call, the value in the DD or DDD field is not valid for the given year and/or month. For example, 'MM/DD/YY' with '02/29/90', or 'YYYY.DDD' with '1990.366' are invalid because 1990 is not a leap year. This code may also be returned for any nonexistent date value such as June 31st, January 0.

Programmer response: Verify that the format of the input data matches the picture string specification and that input data contains a valid date.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EC

IWZ2509S

The era passed to CEEDAYS or CEESECS was not recognized.

Explanation: The value in the <JJJJ>, <CCCC>, or <CCCCCCCC> field passed in a call to CEEDAYS or CEESECS does not contain a supported era name.

Programmer response: Verify that the format of the input data matches the picture string specification and that the spelling of the era name is correct. Note that the era name must be a proper DBCS string where the '<' position must contain the first byte of the era name.

System action: The output value is set to 0.

IWZ2510S

The hours value in a call to CEEISEC or CEESECS was not recognized.

Explanation: (1) In a CEEISEC call, the hours parameter did not contain a number between 0 and 23, or (2) in a CEESECS call, the value in the HH (hours) field does not contain a number between 0 and 23, or the "AP" (a.m./p.m.) field is present and the HH field does not contain a number between 1 and 12.

Programmer response: For CEEISEC, verify that the hours parameter contains an integer between 0 and 23. For CEESECS, verify that the format of the input data matches the picture string specification, and that the hours field contains a value between 0 and 23, (or 1 and 12 if the "AP" field is used).

System action: The output value is set to 0.

Symbolic feedback code: CEE2EE

IWZ2511S

The day parameter passed in a CEEISEC call was invalid for year and month specified.

Explanation: The day parameter passed in a CEEISEC call did not contain a valid day number. The combination of year, month, and day formed an invalid date value. Examples: year=1990, month=2, day=29; or month=6, day=31; or day=0.

Programmer response: Verify that the day parameter contains an integer between 1 and 31, and that the combination of year, month, and day represents a valid date.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EF

IWZ2512S

The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Explanation: The Lilian day number passed in a call to CEEDATE or CEEDYWK was not a number between 1 and 3,074,324.

Programmer response: Verify that the input parameter contains an integer between 1 and 3,074,324.

System action: The output value is set to blanks.

Symbolic feedback code: CEE2EG

IWZ2513S

The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.

Explanation: The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was earlier than 15 October 1582, or later than 31 December 9999.

Programmer response: For CEEISEC, verify that the year, month, and day parameters form a date greater than or equal to 15 October 1582. For CEEDAYS and CEESECS, verify that the format of the input date matches the picture string specification, and that the input date is within the supported range.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EH

IWZ2514S

The year value passed in a CEEISEC call was not within the supported range.

Explanation: The year parameter passed in a CEEISEC call did not contain a number between 1582 and 9999.

Programmer response: Verify that the year parameter contains valid data, and that the year parameter includes the century, for example, specify year 1990, not year 90.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EI

IWZ2515S

The milliseconds value in a CEEISEC call was not recognized.

Explanation: In a CEEISEC call, the milliseconds parameter (*input_milliseconds*) did not contain a number between 0 and 999.

Programmer response: Verify that the milliseconds parameter contains an integer between 0 and 999.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EJ

IWZ2516S

The minutes value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the minutes parameter (*input_minutes*) did not contain a number between 0 and 59, or (2) in a CEESECS call, the value in the MI (minutes) field did not contain a number between 0 and 59.

Programmer response: For CEEISEC, verify that the minutes parameter contains an integer between 0 and 59. For CEESECS, verify that the format of the input data matches the picture string specification, and that the minutes field contains a number between 0 and 59.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EK

IWZ2517S

The month value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the month parameter (*input_month*) did not contain a number between 1 and 12, or (2) in a CEEDAYS or CEESECS call, the value in the MM field did not contain a number between 1 and 12, or the value in the MMM, MMMM, etc. field did not contain a correctly spelled month name or month abbreviation in the currently active National Language.

Programmer response: For CEEISEC, verify that the month parameter contains an integer between 1 and 12. For CEEDAYS and CEESECS, verify that the format of the input data matches the picture string specification. For the MM field, verify that the input value is between 1 and 12. For spelled-out month names (MMM, MMMM, etc.), verify that the spelling or abbreviation of the month name is correct in the currently active National Language.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EL

IWZ2518S

An invalid picture string was specified in a call to a date/time service.

Explanation: The picture string supplied in a call to one of the date/time services was invalid. Only one era character string can be specified.

Programmer response: Verify that the picture string contains valid data. If the picture string contains more than one era descriptor, such as both <JJJJ> and <CCCC>, then change the picture string to use only one era.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EM

IWZ2519S

The seconds value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the seconds parameter (*input_seconds*) did not contain a number between 0 and 59, or (2) in a CEESECS call, the value in the SS (seconds) field did not contain a number between 0 and 59.

Programmer response: For CEEISEC, verify that the seconds parameter contains an integer between 0 and 59. For CEESECS, verify that the format of the input data matches the picture string specification, and that the seconds field contains a number between 0 and 59.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EN

IWZ2520S

CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.

Explanation: The input value passed in a CEEDAYS call did not appear to be in the format described by the picture specification, for example, nonnumeric characters appear where only numeric characters are expected.

Programmer response: Verify that the format of the input data matches the picture string specification and that numeric fields contain only numeric data.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EO

IWZ2521S

The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.

Explanation: In a CEEDAYS or CEESECS call, if the YY or ZYY picture token is specified, and if the picture string contains one of the era tokens such as <CCCC> or <JJJJ>, then the year value must be greater than or equal to 1 and must be a valid year value for the era. In this context, the YY or ZYY field means year within era.

Programmer response: Verify that the format of the input data matches the picture string specification and that the input data is valid.

System action: The output value is set to 0.

IWZ2522S

An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.

Explanation: In a CEEDATE call, the picture string indicates that the Lilian date is to be converted to an era, but the Lilian date lies outside the range of supported eras.

Programmer response: Verify that the input value contains a valid Lilian day number within the range of supported eras.

System action: The output value is set to blanks.

IWZ2525S

CEESECS detected nonnumeric data in a numeric field, or the time-stamp string did not match the picture string.

Explanation: The input value passed in a CEESECS call did not appear to be in the format described by the picture specification. For example, nonnumeric characters appear where only numeric characters are expected, or the a.m./p.m. field (AP, A.P., etc.) did not contain the strings 'AM' or 'PM'.

Programmer response: Verify that the format of the input data matches the picture string specification and that numeric fields contain only numeric data.

System action: The output value is set to 0.

Symbolic feedback code: CEE2ET

IWZ2526S

The date string returned by CEEDATE was truncated.

Explanation: In a CEEDATE call, the output string was not large enough to contain the formatted date value.

Programmer response: Verify that the output string data item is large enough to contain the entire formatted date. Ensure that the output parameter is at least as long as the picture string parameter.

System action: The output value is truncated to the length of the output parameter.

Symbolic feedback code: CEE2EU

IWZ2527S

The time-stamp string returned by CEEDATM was truncated.

Explanation: In a CEEDATM call, the output string was not large enough to contain the formatted time-stamp value.

Programmer response: Verify that the output string data item is large enough to contain the entire formatted time stamp. Ensure that the output parameter is at least as long as the picture string parameter.

System action: The output value is truncated to the length of the output parameter.

Symbolic feedback code: CEE2EV

IWZ2531S

The local time was not available from the system.

Explanation: A call to CEELOCT failed because the system clock was in an invalid state. The current time cannot be determined.

Programmer response: Notify systems support personnel that the system clock is in an invalid state.

System action: All output values are set to 0.

Symbolic feedback code: CEE2F3

IWZ2533S

The value passed to CEESCEN was not between 0 and 100.

Explanation: The *century_start* value passed in a CEESCEN call was not between 0 and 100, inclusive.

Programmer response: Ensure that the input parameter is within range.

System action: No system action is taken; the 100-year window assumed for all two-digit years is unchanged.

Symbolic feedback code: CEE2F5

IWZ2534W

Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Explanation: The CEEDATE or CEEDATM callable services issues this message whenever the picture string contained MMM, MMMMMZ, WWW, Wwww, etc., requesting a spelled out month name or weekday name, and the month name currently being formatted contained more characters than can fit in the indicated field.

Programmer response: Increase the field width by specifying enough Ms or Ws to contain the longest month or weekday name being formatted.

System action: The month name and weekday name fields that are of insufficient width are set to blanks. The rest of the output string is unaffected. Processing continues.

Symbolic feedback code: CEE2F6

Related concepts
Appendix C, “Intermediate results and arithmetic precision,” on page 477

Related tasks
“Setting environment variables” on page 213
“Generating a list of compiler messages” on page 228

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

© Copyright IBM Corp. 2021

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1995, 2019.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies,"

and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms might or might not have the same meaning in other languages.

glossary.html

This glossary includes terms and definitions from the following publications:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*, and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module*
- *ANSI X3.172-2002, American National Standard Dictionary for Information Systems*
- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*
- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

American National Standard definitions are preceded by an asterisk (*).

A

*** abbreviated combined relation condition**
The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend
Abnormal termination of a program.

*** access mode**
The manner in which records are to be operated upon within a file.

*** actual decimal point**
The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

actual document encoding
For an XML document, one of the following encoding categories that the XML parser determines by examining the first few bytes of the document:

- ASCII
- EBCDIC
- UTF-8
- UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

Linux native file system
Any of the local or network file systems that directly support encoded or binary stream files.

The Linux native file systems support line-sequential files directly, and are used as the file store for all the other COBOL file types.

*** alphabet-name**
A user-defined word, in the `SPECIAL-NAMES` paragraph of the `ENVIRONMENT DIVISION`, that assigns a name to a specific character set or collating sequence or both.

*** alphabetic character**
A letter or a space character.

© Copyright IBM Corp. 2021

alphabetic data item
A data item that is described with a PICTURE character string that contains only the symbol A. An alphabetic data item has USAGE DISPLAY.

*** alphanumeric character**
Any character in the single-byte character set of the computer.

alphanumeric character position
See *character position*.

alphanumeric data item
A general reference to a data item that is described implicitly or explicitly as USAGE DISPLAY, and that has category alphanumeric, alphanumeric-edited, or numeric-edited.

alphanumeric-edited data item
A data item that is described by a PICTURE character string that contains at least one instance of the symbol A or X and at least one of the simple insertion symbols B, 0, or /. An alphanumeric-edited data item has USAGE DISPLAY.

*** alphanumeric function**
A function whose value is composed of a string of one or more characters from the alphanumeric character set of the computer.

alphanumeric group item
A group item that is defined without a GROUP-USAGE NATIONAL clause. For operations such as INSPECT, STRING, and UNSTRING, an alphanumeric group item is processed as though all its content were described as USAGE DISPLAY regardless of the actual content of the group. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, or INITIALIZE, an alphanumeric group item is processed using group semantics.

alphanumeric literal
A literal that has an opening delimiter from the following set: ', ", X', X", Z', or Z". The string of characters can include any character in the character set of the computer.

*** alternate record key**
A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute)
An organization that consists of producers, consumers, and general-interest groups and establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

argument
(1) An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function. (2) An operand of the USING phrase of a CALL statement, used for passing values to a called program.

*** arithmetic operation**
The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**
A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

*** arithmetic statement**
A statement that causes an arithmetic operation to be executed. The arithmetic statements are ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

array
An aggregate that consists of data objects, each of which can be uniquely referenced by subscripting. An array is roughly analogous to a COBOL table.

*** ascending key**
A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII
American National Standard Code for Information Interchange. The standard code uses a coded character set that is based on 7-bit coded characters (8 bits including parity check). The standard is used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (characters 128-255).

ASCII-based multibyte code page
A UTF-8, EUC, or ASCII DBCS code page. Each ASCII-based multibyte code page includes both single-byte and multibyte characters. The encoding of the single-byte characters is the ASCII encoding.

ASCII DBCS
See *double-byte ASCII*.

assignment-name
A name that identifies the organization of a COBOL file and the name by which it is known to the system.

*** assumed decimal point**
A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

AT END condition
A condition that is caused during the execution of a READ, RETURN, or SEARCH statement under certain conditions:

- A READ statement runs on a sequentially accessed file when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not available.
- A RETURN statement runs when no next logical record exists for the associated sort or merge file.
- A SEARCH statement runs when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

basic character set
The basic set of characters used in writing words, character-strings, and separators of the language. The basic character set is implemented in single-byte characters. The extended character set includes DBCS, UTF-8, or EUC characters, which can be used in comments, literals, and user-defined words.

Synonymous with *COBOL character set* in the 85 COBOL Standard.

big-endian
The default format that the mainframe and the Linux workstation use to store binary data and UTF-16 characters. In this format, the least significant byte of a binary data item is at the highest address and the least significant byte of a UTF-16 character is at the highest address. Compare with *little-endian*.

binary item
A numeric data item that is represented in binary notation (on the base 2 numbering system). The decimal equivalent consists of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search
A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

*** block**
A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical records that are either contained within the block or that overlap the block. Synonymous with *physical record*.

boolean condition
A boolean condition determines whether a boolean literal is true or false. A boolean condition can only be used in a constant conditional expression.

boolean literal
Can be either B'1', indicating a true value, or B'0', indicating a false value. Boolean literals can only be used in constant conditional expressions.

breakpoint
A place in a computer program, usually specified by an instruction, where external intervention or a monitor program can interrupt the program as it runs.

buffer
A portion of storage that is used to hold input or output data temporarily.

built-in function
See *intrinsic function*.

byte
A string that consists of a certain number of bits, usually eight, treated as a unit, and representing a character or a control function.

byte order mark (BOM)
A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big-endian or little-endian.

bytecode
Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. (Oracle)

C

called program
A program that is the object of a CALL statement. At run time the called program and calling program are combined to produce a *run unit*.

*** calling program**
A program that executes a CALL to another program.

case structure
A program-processing logic in which a series of conditions is tested in order to choose between a number of resulting actions.

CCSID
See *coded character set identifier*.

century window
A 100-year interval within which any two-digit year is unique. Several types of century window are available to COBOL programmers:

- For windowed date fields, you use the YEARWINDOW compiler option.
- For the windowing intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY, you specify the century window with *argument-2*.

*** character**
The basic indivisible unit of the language.

character encoding unit
A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For USAGE NATIONAL, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For USAGE DISPLAY, a character encoding unit corresponds to a byte.

For USAGE DISPLAY-1, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

character position
The amount of physical storage or presentation space required to hold or present one character. The term applies to any class of character. For specific classes of characters, the following terms apply:

- *Alphanumeric character position*, for characters represented in USAGE DISPLAY
- *DBCS character position*, for DBCS characters represented in USAGE DISPLAY-1
- *National character position*, for characters represented in USAGE NATIONAL; synonymous with *character encoding unit* for UTF-16

character set
A collection of elements that are used to represent textual information, but for which no coded representation is assumed. See also *coded character set*.

character string
A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character string, or a comment-entry. A character string must be delimited by separators.

checkpoint
A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.

*** class**
The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class. Classes can be defined hierarchically, allowing one class to inherit from another.

*** class condition**
The proposition (for which a truth value can be determined) that the content of an item is wholly alphabetic, is wholly numeric, is wholly DBCS, is wholly Kanji, or consists exclusively of the characters that are listed in the definition of a class-name.

*** class-name (of data)**
A user-defined word that is defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this word assigns a name to the proposition (for which a truth value can be defined) that the content of a data item consists exclusively of the characters that are listed in the definition of the class-name.

*** clause**
An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

COBOL character set
The set of characters used in writing COBOL syntax. The complete COBOL character set consists of these characters:

Character	Meaning
0,1, . . . ,9	Digit
A,B, . . . ,Z	Uppercase letter
a,b, . . . ,z	Lowercase letter
	Space

Character	Meaning
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
$	Currency sign
,	Comma
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
'	Apostrophe
(	Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

*** COBOL word**
See *word*.

code page
An assignment of graphic characters and control function meanings to all code points. For example, one code page could assign characters and meanings to 256 code points for 8-bit code, and another code page could assign characters and meanings to 128 code points for 7-bit code. For example, one of the IBM code pages for English on the workstation is IBM-1252 and on the host is IBM-1047.

code point
A unique bit pattern that is defined in a coded character set (code page). Graphic symbols and control characters are assigned to code points.

coded character set
A set of unambiguous rules that establish a character set and the relationship between the characters of the set and their coded representation. Examples of coded character sets are the character sets as represented by ASCII or EBCDIC code pages or by the UTF-16 encoding scheme for Unicode.

coded character set identifier (CCSID)
An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

*** collating sequence**
The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

*** column**
A byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

*** combined condition**
A condition that is the result of connecting two or more conditions with the AND or the OR logical operator. See also *condition* and *negated combined condition*.

*** comment-entry**
An entry in the IDENTIFICATION DIVISION that is used for documentation and has no effect on execution.

comment line
A source program line represented by an asterisk (*) in the indicator area of the line or by an asterisk followed by greater-than sign (*>) as the first character string in the program text area (Area A plus Area B), and any characters from the character set of the computer that follow in Area A and Area B of that line. A comment line serves only for documentation. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the character set of the computer in Area A and Area B of that line causes page ejection before the comment is printed.

*** common program**
A program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

compatible date field
The meaning of the term *compatible*, when applied to date fields, depends on the COBOL division in which the usage occurs:

- DATA DIVISION: Two date fields are compatible if they have identical USAGE and meet at least one of the following conditions:
  - They have the same date format.
  - Both are windowed date fields, where one consists only of a windowed year, DATE FORMAT YY.
  - Both are expanded date fields, where one consists only of an expanded year, DATE FORMAT YYYY.
  - One has DATE FORMAT YYXXXX, and the other has YYXX.
  - One has DATE FORMAT YYYYXXXX, and the other has YYYYXX.

  A windowed date field can be subordinate to a data item that is an expanded date group. The two date fields are compatible if the subordinate date field has USAGE DISPLAY, starts two bytes after the start of the group expanded date field, and the two fields meet at least one of the following conditions:
  - The subordinate date field has a DATE FORMAT pattern with the same number of Xs as the DATE FORMAT pattern of the group date field.
  - The subordinate date field has DATE FORMAT YY.
  - The group date field has DATE FORMAT YYYYXXXX and the subordinate date field has DATE FORMAT YYXX.
- PROCEDURE DIVISION: Two date fields are compatible if they have the same date format except for the year part, which can be windowed or expanded. For example, a windowed date field with DATE FORMAT YYXXX is compatible with:
  - Another windowed date field with DATE FORMAT YYXXX
  - An expanded date field with DATE FORMAT YYYYXXX

*** compile**
(1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine-language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

compilation variable
A symbolic name for a particular literal value or the value of a compile-time arithmetic expression as specified by the DEFINE directive or by the DEFINE compiler option.

*** compile time**
The time at which COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

compile-time arithmetic expression
A subset of arithmetic expressions that are specified in the DEFINE and EVALUATE directives or in a constant conditional expression. The difference between compile-time arithmetic expressions and regular arithmetic expressions is that in a compile-time arithmetic expression:

- The exponentiation operator shall not be specified.
- All operands shall be integer numeric literals or arithmetic expressions in which all operands are integer numeric literals.
- The expression shall be specified in such a way that a division by zero does not occur.

compiler
A program that translates source code written in a higher-level language into machine-language object code.

compiler-directing statement
A statement that causes the compiler to take a specific action during compilation. The standard compiler-directing statements are COPY, REPLACE, and USE.

compiler directive
A directive that causes the compiler to take a specific action during compilation. COBOL for Linux supports the CALLINTERFACE compiler directive, as well as Conditional compilation compiler directives (DEFINE, EVALUATE, and IF).

*** complex condition**
A condition in which one or more logical operators act upon one or more conditions. See also *condition*, *negated simple condition*, and *negated combined condition*.

complex ODO
Certain forms of the OCCURS DEPENDING ON clause:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item described by an OCCURS clause.
- Table with variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON option.
- Index name for a table with variable-length elements.
- Element of a table with variable-length elements.

component
(1) A functional grouping of related files. (2) In object-oriented programming, a reusable object or program that performs a specific function and is designed to work with other components and applications. JavaBeans is Oracle's architecture for creating components.

*** computer-name**
A system-name that identifies the computer where the program is to be compiled or run.

condition (exception)
Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware or the operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition (expression)
A status of data at run time for which a truth value can be determined. Where used in this information in or in reference to "condition" (*condition-1*, *condition-2*,. . .) of a general format, the term refers to a conditional expression that consists of either a simple condition optionally parenthesized or a combined condition (consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses) for which a truth value can be determined. See also *simple condition*, *complex condition*, *negated simple condition*, *combined condition*, and *negated combined condition*.

*** conditional expression**
A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. See also *simple condition* and *complex condition*.

*** conditional phrase**
A phrase that specifies the action to be taken upon determination of the truth value of a condition that results from the execution of a conditional statement.

*** conditional statement**
A statement that specifies that the truth value of a condition is to be determined and that the subsequent action of the object program depends on this truth value.

*** conditional variable**
A data item one or more values of which has a condition-name assigned to it.

*** condition-name**
A user-defined word that assigns a name to a subset of values that a conditional variable can assume; or a user-defined word assigned to a status of an implementor-defined switch or device.

*** condition-name condition**
The proposition (for which a truth value can be determined) that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

*** CONFIGURATION SECTION**
A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs.

CONSOLE
A COBOL environment-name associated with the operator console.

constant conditional expression
A subset of conditional expressions that may be used in IF directives or WHEN phrases of the EVALUATE directives.

A constant conditional expression shall be one of the following items:

- A relation condition in which both operands are literals or arithmetic expressions that contain only literal terms. The condition shall follow the rules for relation conditions, with the following additions:
  - The operands shall be of the same category. An arithmetic expression is of the category numeric.
  - If literals are specified and they are not numeric literals, the relational operator shall be "IS EQUAL TO", "IS NOT EQUAL TO", "IS =", "IS NOT =", or "IS <>".

  See also *relation condition*.
- A defined condition. See also *defined condition*.
- A boolean condition. See also *boolean condition*.
- A complex condition formed by combining the above forms of simple conditions into complex conditions by using AND, OR, and NOT. Abbreviated combined relation conditions shall not be specified. See also *complex condition*.

contained program
A COBOL program that is nested within another COBOL program.

*** contiguous items**
Items that are described by consecutive entries in the DATA DIVISION, and that bear a definite hierarchic relationship to each other.

copybook
A file or library member that contains a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product. Synonymous with *copy file*.

*** counter**
A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cross-reference listing
The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

currency-sign value
A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are $, USD, and EUR. A currency-sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency-sign value. See also *currency symbol*.

currency symbol
A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

*** current record**
In file processing, the record that is available in the record area associated with a file.

*** current volume pointer**
A conceptual entity that points to the current volume of a sequential file.

D

*** data clause**
A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

*** data description entry**
An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

DATA DIVISION
The division of a COBOL program that describes the data to be processed by the program: the files to be used and the records contained within them; internal WORKING-STORAGE records that will be needed; data to be made available in more than one program in the COBOL run unit.

*** data item**
A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

*** data-name**
A user-defined word that names a data item described in a data description entry. When used in the general formats, data-name represents a word that must not be reference-modified, subscripted, or qualified unless specifically permitted by the rules for the format.

date field
Any of the following items:

- A data item whose data description entry includes a DATE FORMAT clause.
- A value returned by one of the following intrinsic functions:

```
DATE-OF-INTEGER
DATE-TO-YYYYMMDD
DATEVAL
DAY-OF-INTEGER
DAY-TO-YYYYDDD
YEAR-TO-YYYY
YEARWINDOW
```

- The conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD of the ACCEPT statement.

- The result of certain arithmetic operations. For details, see Arithmetic with date fields (*COBOL for Linux on x86 Language Reference*).

The term *date field* refers to both *expanded date field* and *windowed date field*. See also *nondate*.

date format
The date pattern of a date field, specified in either of the following ways:

- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function argument-2
- Implicitly, by statements and intrinsic functions that return date fields. For details, see Date field (*COBOL for Linux on x86 Language Reference*).

Db2 file system
The Db2 file system supports sequential, indexed, and relative files. It provides enhanced interoperation with CICS, enabling batch COBOL programs to access CICS ESDS, KSDS, and RRDS files that are stored in Db2.

DBCS
See *double-byte character set (DBCS)*.

DBCS character
Any character defined in IBM's double-byte character set.

DBCS character position
See *character position*.

DBCS data item
A data item that is described by a PICTURE character string that contains at least one symbol G, or, when the NSYMBOL(DBCS) compiler option is in effect, at least one symbol N. A DBCS data item has USAGE DISPLAY-1.

*** debugging line**
Any line with a D in the indicator area of the line.

*** debugging section**
A section that contains a USE FOR DEBUGGING statement.

*** declarative sentence**
A compiler-directing sentence that consists of a single USE statement terminated by the separator period.

*** declaratives**
A set of one or more special-purpose sections, written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the key word DECLARATIVE and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

*** de-edit**
The logical removal of all editing characters from a numeric-edited data item in order to determine the unedited numeric value of the item.

defined condition
A compile-time condition that tests whether a compilation variable is defined. Defined conditions are specified in IF directives or WHEN phrases of the EVALUATE directives.

*** delimited scope statement**
Any statement that includes its explicit scope terminator.

*** delimiter**
A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

*** descending key**
A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit
Any of the numerals from 0 through 9. In COBOL, the term is not used to refer to any other symbol.

*** digit position**
The amount of physical storage required to store a single digit. This amount can vary depending on the usage specified in the data description entry that defines the data item.

*** direct access**
The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

display floating-point data item
A data item that is described implicitly or explicitly as USAGE DISPLAY and that has a PICTURE character string that describes an external floating-point data item.

*** division**
A collection of zero, one, or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

*** division header**
A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

do construct
In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an inline PERFORM statement functions in the same way.

do-until
In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

do-while
In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

document type declaration
An XML element that contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.

document type definition (DTD)
The grammar for a class of XML documents. See *document type declaration*.

double-byte ASCII
An IBM character set that includes DBCS and single-byte ASCII characters. (Also known as ASCII DBCS.)

double-byte EBCDIC
An IBM character set that includes DBCS and single-byte EBCDIC characters. (Also known as EBCDIC DBCS.)

double-byte character set (DBCS)
A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

DWARF
DWARF was developed by the UNIX International Programming Languages Special Interest Group (SIG). It is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by supplying language-independent debugging information. A DWARF file contains debugging data organized into different elements. For more information, see *DWARF program information* in the *DWARF/ELF Extensions Library Reference*.

*** dynamic access**
An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

dynamic CALL
A CALL *literal* statement in a program that has been compiled with the DYNAM option, or a CALL *identifier* statement in a program.

E

*** EBCDIC (Extended Binary-Coded Decimal Interchange Code)**
A coded character set based on 8-bit coded characters.

EBCDIC character
Any one of the symbols included in the EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

EBCDIC DBCS
See *double-byte EBCDIC*.

edited data item
A data item that has been modified by suppressing zeros or inserting editing characters or both.

*** editing character**
A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
	Space
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
Z	Zero suppress
*	Check protect
$	Currency sign
,	Comma (decimal point)
.	Period (decimal point)
/	Slant (forward slash)

element (text element)
One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

*** elementary item**
A data item that is described as not being further logically subdivided.

CICS SFS file system
See *SFS file system*.

encoding unit
See *character encoding unit*.

*** end of PROCEDURE DIVISION**
The physical position of a COBOL source program after which no further procedures appear.

*** end program marker**
A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

```
END PROGRAM program-name.
```

*** entry**
Any descriptive set of consecutive clauses terminated by a separator period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL program.

*** environment clause**
A clause that appears as part of an ENVIRONMENT DIVISION entry.

ENVIRONMENT DIVISION
One of the four main component parts of a COBOL program. The ENVIRONMENT DIVISION describes the computers where the source program is compiled and those where the object program is run. It provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

environment-name
A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, program switches or all of these. When an environment-name is associated with a mnemonic-name in the ENVIRONMENT DIVISION, the mnemonic-name can be substituted in any format in which such substitution is valid.

environment variable
Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

execution time
See *run time*.

execution-time environment
See *runtime environment*.

expanded date field
A date field containing an expanded (four-digit) year. See also *date field* and *expanded year*.

expanded year
A date field that consists only of a four-digit year. Its value includes the century: for example, 1998. Compare with *windowed year*.

*** explicit scope terminator**
A reserved word that terminates the scope of a particular PROCEDURE DIVISION statement.

exponent
A number that indicates the power to which another number (the base) is to be raised. Positive exponents denote multiplication; negative exponents denote division; and fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol ** followed by the exponent.

*** expression**
An arithmetic or conditional expression.

*** extend mode**
The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

Extensible Markup Language
See *XML*.

extensions
COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

external code page
For ASCII or UTF-8 XML documents, the code page indicated by the current runtime locale. For EBCDIC XML documents, either:

- The code page specified in the EBCDIC_CODEPAGE environment variable
- The default EBCDIC code page selected for the current runtime locale if the EBCDIC_CODEPAGE environment variable is not set

*** external data**
The data that is described in a program as external data items and external file connectors.

*** external data item**
A data item that is described as part of an external record in one or more programs of a run unit and that can be referenced from any program in which it is described.

*** external data record**
A logical record that is described in one or more programs of a run unit and whose constituent data items can be referenced from any program in which they are described.

external decimal data item
See *zoned decimal data item* and *national decimal data item*.

*** external file connector**
A file connector that is accessible to one or more object programs in the run unit.

external floating-point data item
See *display floating-point data item* and *national floating-point data item*.

external program
The outermost program. A program that is not nested.

*** external switch**
A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

*** figurative constant**
A compiler-generated value referenced through the use of certain reserved words.

*** file**
A collection of logical records.

*** file attribute conflict condition**
An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file clause**
A clause that appears as part of any of the following `DATA DIVISION` entries: file description entry (`FD` entry) and sort-merge file description entry (`SD` entry).

*** file connector**
A storage area that contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

*** file control entry**
A `SELECT` clause and all its subordinate clauses that declare the relevant physical attributes of a file.

`FILE-CONTROL` paragraph
A paragraph in the `ENVIRONMENT DIVISION` in which the data files for a given source unit are declared.

*** file description entry**
An entry in the `FILE SECTION` of the `DATA DIVISION` that is composed of the level indicator `FD`, followed by a file-name, and then followed by a set of file clauses as required.

*** file-name**
A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the FILE SECTION of the DATA DIVISION.

*** file organization**
The permanent logical file structure established at the time that a file is created.

file position indicator
A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not available, or that the AT END condition already exists, or that no valid next record has been established.

*** FILE SECTION**
The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

file system
The collection of files that conform to a specific set of data-record and file-description protocols, and a set of programs that manage these files.

*** fixed file attributes**
Information about a file that is established when a file is created and that cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

*** fixed-length record**
A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of bytes.

fixed-point item
A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format can be either binary, packed decimal, or external decimal.

floating comment indicators (*>)
A floating comment indicator indicates a comment line if it is the first character string in the program-text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program-text area.

floating point
A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral) and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral). For example, a floating-point representation of the number 0.0001234 is 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

floating-point data item
A numeric data item that contains a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power that the exponent specifies.

*** format**
A specific arrangement of a set of data.

*** function**
A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function-identifier**
A syntactically correct combination of character strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier can include a reference-modifier. A function-identifier that references an alphanumeric function can be specified anywhere in the general formats that an identifier can be

specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

function-name
A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

function-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the `USAGE IS FUNCTION-POINTER` clause contains the address of a function entry point. Typically used to communicate with C and Java programs.

G

garbage collection
The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

GDG
See *generation data group (GDG)*.

GDS
See *generation data set (GDS)*.

generation data group (GDG)
A collection of chronologically related files; each such file is called a *generation data set (GDS)* or generation.

generation data set (GDS)
One of the files in a *generation data group (GDG)*; each such file is chronologically related to the other files in the group.

*** global name**
A name that is declared in only one program but that can be referenced from the program and from any program contained within the program. Condition-names, data-names, file-names, record-names, report-names, and some special registers can be global names.

group item
(1) A data item that is composed of subordinate data items. See *alphanumeric group item* and *national group item*. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

grouping separator
A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

H

header label
(1) A label that precedes the data records in a unit of recording media. (2) Synonym for *beginning-of-file label*.

*** high-order end**
The leftmost character of a string of characters.

host alphanumeric data item
(Of XML documents) A category alphanumeric data item whose data description entry does not contain the `NATIVE` phrase, and that was compiled with the `CHAR(EBCDIC)` option in effect. The encoding for the data item is the EBCDIC code page in effect. This code page is determined from the EBCDIC_CODEPAGE environment variable, if set, otherwise from the default code page associated with the runtime locale.

I

IBM COBOL extension
COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

ICU
See *International Components for Unicode (ICU)*.

IDENTIFICATION DIVISION
One of the four main component parts of a COBOL program. The IDENTIFICATION DIVISION identifies the program, class. The IDENTIFICATION DIVISION can include the following documentation: author name, installation, or date.

*** identifier**
A syntactically correct combination of character strings and separators that names a data item. When referencing a data item that is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference-modifier, as required for uniqueness of reference. When referencing a data item that is a function, a function-identifier is used.

*** imperative statement**
A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement can consist of a sequence of imperative statements.

*** implicit scope terminator**
A separator period that terminates the scope of any preceding unterminated statement, or a phrase of a statement that by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

*** index**
A computer storage area or register, the content of which represents the identification of a particular element in a table.

*** index data item**
A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name
An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

*** indexed file**
A file with indexed organization.

*** indexed organization**
The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

indexing
Synonymous with *subscripting* using index-names.

*** index-name**
A user-defined word that names an index associated with a specific table.

*** initial program**
A program that is placed into an initial state every time the program is called in a run unit.

*** initial state**
The state of a program when it is first called in a run unit.

inline
In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

*** input file**
A file that is opened in the input mode.

*** input mode**
The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

*** input-output file**
A file that is opened in the I-O mode.

*** INPUT-OUTPUT SECTION**
The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program and that provides information required for transmission and handling of data at run time.

*** input-output statement**
A statement that causes files to be processed by performing operations on individual records or on the file as a unit. The input-output statements are ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

*** input procedure**
A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

*** integer**
(1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point. (3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

integer function
A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

interlanguage communication (ILC)
The ability of routines written in different programming languages to communicate. ILC support lets you readily build applications from component routines written in a variety of languages.

intermediate result
An intermediate field that contains the results of a succession of arithmetic operations.

*** internal data**
The data that is described in a program and excludes all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

*** internal data item**
A data item that is described in one program in a run unit. An internal data item can have a global name.

internal decimal data item
A data item that is described as USAGE PACKED-DECIMAL or USAGE COMP-3, and that has a PICTURE character string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with *packed-decimal data item*.

*** internal file connector**
A file connector that is accessible to only one object program in the run unit.

internal floating-point data item
A data item that is described as USAGE COMP-1 or USAGE COMP-2. COMP-1 defines a single-precision floating-point data item. COMP-2 defines a double-precision floating-point data item. There is no PICTURE clause associated with an internal floating-point data item.

International Components for Unicode (ICU)
An open-source development project sponsored, supported, and used by IBM. ICU libraries provide robust and full-featured Unicode services on a wide variety of platforms, including AIX and Linux.

*** intrarecord data structure**
The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

intrinsic function
A predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

*** invalid key condition**
A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

*** I-O-CONTROL**
The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

*** I-O-CONTROL entry**
An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

*** I-O mode**
The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

*** I-O status**
A conceptual entity that contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

iteration structure
A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

J

J2EE
See *Java 2 Platform, Enterprise Edition (J2EE)*.

Java 2 Platform, Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications, defined by Oracle. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Oracle)

Java Native Interface (JNI)
A programming interface that lets Java code that runs inside a Java virtual machine (JVM) interoperate with applications and libraries written in other programming languages.

Java virtual machine (JVM)
A software implementation of a central processing unit that runs compiled Java programs.

JSON
JSON (JavaScript Object Notation) is a lightweight data-interchange format.

JVM
See *Java virtual machine (JVM)*.

K

K
When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key**
A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

*** key of reference**
The key, either prime or alternate, currently being used to access records within an indexed file.

*** keyword**
A context-sensitive word or a reserved word whose presence is required when the format in which the word appears is used in a source unit.

kilobyte (KB)
One kilobyte equals 1024 bytes.

L

*** language-name**
A system-name that specifies a particular programming language.

last-used state
A state that a program is in if its internal values remain the same as when the program was exited (the values are not reset to their initial values).

*** letter**
A character belonging to one of the following two sets:

1. Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

*** level indicator**
Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the `DATA DIVISION` are: `CD`, `FD`, and `SD`.

*** level-number**
A user-defined word (expressed as a two-digit number) that indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

*** library-name**
A user-defined word that names a COBOL library that the compiler is to use for compiling a given source program.

*** library text**
A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

Lilian date
The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

*** linage-counter**
A special register whose value points to the current position within the page body.

link
(1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs; for example, linking object programs by a linkage-editor to produce a shared library.

`LINKAGE SECTION`
The section in the `DATA DIVISION` of the called program that describes data items available from the calling program. Both the calling program and the called program can refer to these data items.

literal
A character string whose value is specified either by the ordered set of characters comprising the string or by the use of a figurative constant.

little-endian
The default format that Intel processors use to store binary data and UTF-16 characters. In this format, the most significant byte of a binary data item is at the highest address and the most significant byte of a UTF-16 character is at the highest address. Compare with *big-endian*.

locale
A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as character code page, collating sequence, date and time format, monetary value representation, numeric value representation, or language.

*** LOCAL-STORAGE SECTION**
The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in the VALUE clauses.

*** logical operator**
One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

*** logical record**
The most inclusive data item. The level-number for a record is 01. A record can be either an elementary item or a group of items. Synonymous with *record*.

*** low-order end**
The rightmost character of a string of characters.

LSQ file system
The LSQ file system supports only LINE SEQUENTIAL files.

M

main program
In a hierarchy of programs and subroutines, the first program that receives control when the programs are run within a process.

makefile
A text file that contains a list of the files for your application. The make utility uses this file to update the target files with the latest changes.

*** mass storage**
A storage medium in which data can be organized and maintained in both a sequential manner and a nonsequential manner.

*** mass storage device**
A device that has a large storage capacity, such as a magnetic disk.

*** mass storage file**
A collection of records that is stored in a mass storage medium.

MBCS
See *multibyte character set (MBCS)*.

*** megabyte (MB)**
One megabyte equals 1,048,576 bytes.

*** merge file**
A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

*** mnemonic-name**
A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

module definition file
A file that describes the code segments within a load module.

multibyte character
Any character that is represented in 2 or more bytes in a multibyte character set. For example, a DBCS character or any UTF-8 character that is represented in two or more bytes. UTF-16 characters are not multibyte characters because UTF-16 is not a multibyte character set.

multibyte character set (MBCS)
A coded character set that is composed of characters represented in a varying number of bytes. Examples are: EUC (Extended Unix Code), UTF-8, and character sets composed of a mixture of single-byte and double-byte EBCDIC or ASCII characters.

multitasking
A mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks.

multithreading
Concurrent operation of more than one path of execution within a computer. Synonymous with *multiprocessing*.

N

name
A word (composed of not more than 30 characters) that defines a COBOL operand.

namespace
See *XML namespace*.

national character
(1) A UTF-16 character in a USAGE NATIONAL data item or national literal. (2) Any character represented in UTF-16.

national character data
A general reference to data represented in UTF-16.

national character position
See *character position*.

national data
See *national character data*.

national data item
A data item of category national, national-edited, or numeric-edited of USAGE NATIONAL.

national decimal data item
An external decimal data item that is described implicitly or explicitly as USAGE NATIONAL and that contains a valid combination of PICTURE symbols 9, S, P, and V.

national-edited data item
A data item that is described by a PICTURE character string that contains at least one instance of the symbol N and at least one of the simple insertion symbols B, 0, or /. A national-edited data item has USAGE NATIONAL.

national floating-point data item
An external floating-point data item that is described implicitly or explicitly as USAGE NATIONAL and that has a PICTURE character string that describes a floating-point data item.

national group item
A group item that is explicitly or implicitly described with a GROUP-USAGE NATIONAL clause. A national group item is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This processing ensures correct padding and truncation of national characters, as contrasted with defining USAGE NATIONAL data items within an alphanumeric group item. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE, a national group is processed using group semantics.

native alphanumeric data item
(Of XML documents) A category alphanumeric data item that is described with the NATIVE phrase, or that was compiled with the CHAR(NATIVE) option in effect. The encoding for the data item is the ASCII or UTF-8 code page of the runtime locale in effect.

*** native character set**
The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** native collating sequence**
The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** negated combined condition**
The NOT logical operator immediately followed by a parenthesized combined condition. See also *condition* and *combined condition*.

*** negated simple condition**
The NOT logical operator immediately followed by a simple condition. See also *condition* and *simple condition*.

nested program
A program that is directly contained within another program.

*** next executable sentence**
The next sentence to which control will be transferred after execution of the current statement is complete.

*** next executable statement**
The next statement to which control will be transferred after execution of the current statement is complete.

*** next record**
The record that logically follows the current record of a file.

*** noncontiguous items**
Elementary data items in the WORKING-STORAGE SECTION and LINKAGE SECTION that bear no hierarchic relationship to other data items.

nondate
Any of the following items:

- A data item whose date description entry does not include the DATE FORMAT clause
- A literal
- A date field that has been converted using the UNDATE function
- A reference-modified date field
- The result of certain arithmetic operations that can include date field operands; for example, the difference between two compatible date fields

null
A figurative constant that is used to assign, to pointer data items, the value of an address that is not valid. NULLS can be used wherever NULL can be used.

*** numeric character**
A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric data item
(1) A data item whose description restricts its content to a value represented by characters chosen from the digits 0 through 9. If signed, the item can also contain a +, -, or other representation of an operational sign. (2) A data item of category numeric, internal floating-point, or external floating-point. A numeric data item can have USAGE DISPLAY, NATIONAL, PACKED-DECIMAL, BINARY, COMP, COMP-1, COMP-2, COMP-3, COMP-4, or COMP-5.

numeric-edited data item
A data item that contains numeric data in a form suitable for use in printed output. The data item can consist of external decimal digits from 0 through 9, the decimal separator, commas, the currency sign, sign control characters, and other editing characters. A numeric-edited item can be represented in either USAGE DISPLAY or USAGE NATIONAL.

*** numeric function**
A function whose class and category are numeric but that for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric literal**
A literal composed of one or more numeric characters that can contain a decimal point or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

object code
Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

*** OBJECT-COMPUTER**
The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the object program is run, is described.

*** object computer entry**
An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the object program is to be executed.

*** object of entry**
A set of operands and reserved words, within a DATA DIVISION entry of a COBOL program, that immediately follows the subject of the entry.

object program
A set or group of executable machine-language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program definition. Where there is no danger of ambiguity, the word *program* can be used in place of *object program*.

*** object time**
The time at which an object program is executed. Synonymous with *run time*.

*** obsolete element**
A COBOL language element in the 85 COBOL Standard that was deleted from the 2002 COBOL Standard.

ODBC
See *Open Database Connectivity (ODBC)*.

ODO object
In the example below, X is the object of the OCCURS DEPENDING ON clause (ODO object).

```
WORKING-STORAGE SECTION.
01  TABLE-1.
    05  X                   PIC S9.
    05  Y OCCURS 3 TIMES
          DEPENDING ON X    PIC X.
```

The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject
In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

Open Database Connectivity (ODBC)
A specification for an application programming interface (API) that provides access to data in a variety of databases and file systems.

*** open mode**
The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

*** operand**
(1) The general definition of operand is "the component that is operated upon." (2) For the purposes of this document, any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

operation
A service that can be requested of an object.

*** operational sign**
An algebraic sign that is associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

optional file
A file that is declared as being not necessarily available each time the object program is run.

*** optional word**
A reserved word that is included in a specific format only to improve the readability of the language. Its presence is optional to the user when the format in which the word appears is used in a source unit.

*** output file**
A file that is opened in either output mode or extend mode.

*** output mode**
The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**
A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition
A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

P

packed-decimal data item
See *internal decimal data item*.

padding character
An alphanumeric or national character that is used to fill the unused character positions in a physical record.

page
A vertical division of output data that represents a physical separation of the data. The separation is based on internal logical requirements or external characteristics of the output medium or both.

*** page body**
That part of the logical page in which lines can be written or spaced or both.

*** paragraph**
In the PROCEDURE DIVISION, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, a paragraph header followed by zero, one, or more entries.

*** paragraph header**
A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

```
PROGRAM-ID. (Program IDENTIFICATION
   DIVISION)
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

The permissible paragraph headers in the ENVIRONMENT DIVISION are:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
REPOSITORY. (Program
   CONFIGURATION SECTION)
FILE-CONTROL.
I-O-CONTROL.
```

*** paragraph-name**
A user-defined word that identifies and begins a paragraph in the PROCEDURE DIVISION.

parameter
Data passed between a calling program and a called program.

*** phrase**
An ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**
See *block*.

pointer data item
A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

port
(1) To modify a computer program to enable it to run on a different platform. (2) In the Internet suite of protocols, a specific logical connector between the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and a higher-level protocol or application. A port is identified by a port number.

portability
The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

*** prime record key**
A key whose contents uniquely identify a record within an indexed file.

*** priority-number**
A user-defined word that classifies sections in the PROCEDURE DIVISION for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as either one or two digits.

*** procedure**
A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the PROCEDURE DIVISION.

*** procedure branching statement**
A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source code. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase), XML PARSE.

PROCEDURE DIVISION
The COBOL division that contains instructions for solving a problem.

procedure integration
One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. Contained program procedure integration is the process where a call to a contained program is replaced by the program code.

*** procedure-name**
A user-defined word that is used to name a paragraph or section in the PROCEDURE DIVISION. It consists of a paragraph-name (which can be qualified) or a section-name.

procedure pointer
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

procedure-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point. Typically used to communicate with COBOL programs.

process
The course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources.

program
(1) A sequence of instructions suitable for processing by a computer. Processing may include the use of a compiler to prepare the program for execution, as well as a runtime environment to execute it. (2) A logical assembly of one or more interrelated modules. Multiple copies of the same program can be run in different processes.

*** program identification entry**
In the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION, an entry that contains clauses that specify the program-name and assign selected program attributes to the program.

program-name
In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or an alphanumeric literal that identifies a COBOL source program.

project
The complete set of data and actions that are required to build a target, such as a dynamic link library (DLL) or other executable (EXE).

*** pseudo-text**
A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**
Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**
A character that belongs to the following set:

Character	Meaning
,	Comma
;	Semicolon
:	Colon
.	Period (full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
	Space
=	Equal sign

Q

QSAM (Queued Sequential Access Method)
An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

QSAM file system
The QSAM (Queued Sequential Access Method) file system supports fixed, variable, and spanned records, and it enables you to directly access a QSAM file that you transferred (using z/OS FTP) from z/OS to AIX or Linux with the options `binary and quote site rdw`. A QSAM file supports all COBOL data types in the record.

*** qualified data-name**
An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

*** qualifier**
(1) A data-name or a name associated with a level indicator that is used in a reference either together with another data-name (which is the name of an item that is subordinate to the qualifier) or together with a condition-name. (2) A section-name that is used in a reference together with a paragraph-name specified in that section. (3) A library-name that is used in a reference together with a text-name associated with that library.

R

*** random access**
An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

*** record**
See *logical record*.

*** record area**
A storage area allocated for the purpose of processing the record described in a record description entry in the `FILE SECTION` of the `DATA DIVISION`. In the `FILE SECTION`, the current number of character positions in the record area is determined by the explicit or implicit `RECORD` clause.

*** record description**
See *record description entry*.

*** record description entry**
The total set of data description entries associated with a particular record. Synonymous with *record description*.

record key
A key whose contents identify a record within an indexed file.

record-key-name
A user-defined word that names a key associated with an indexed file.

*** record-name**
A user-defined word that names a record described in a record description entry in the `DATA DIVISION` of a COBOL program.

*** record number**
The ordinal number of a record in the file whose organization is sequential.

recording mode
The format of the logical records in a file. Recording mode can be F (fixed length), V (variable length), S (spanned), or U (undefined).

recursion
A program calling itself or being directly or indirectly called by one of its called programs.

recursively capable
A program is recursively capable (can be called recursively) if the `RECURSIVE` attribute is on the `PROGRAM-ID` statement.

reel
A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. Synonymous with *unit* and *volume*.

reentrant
The attribute of a program or routine that lets more than one user share a single copy of a load module.

*** reference format**
A format that provides a standard method for describing COBOL source programs.

reference modification
A method of defining a new category alphanumeric, category DBCS, or category national data item by specifying the leftmost character and length relative to the leftmost character position of a `USAGE DISPLAY`, `DISPLAY-1`, or `NATIONAL` data item.

*** reference-modifier**
A syntactically correct combination of character strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**
See *relational operator* or *relation condition*.

*** relation character**
A character that belongs to the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

*** relation condition**
The proposition (for which a truth value can be determined) that the value of an arithmetic expression, data item, alphanumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, alphanumeric literal, or index name. See also *relational operator*.

*** relational operator**
A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Character	Meaning
`IS GREATER THAN`	Greater than
`IS >`	Greater than
`IS NOT GREATER THAN`	Not greater than
`IS NOT >`	Not greater than
`IS LESS THAN`	Less than
`IS <`	Less than
`IS NOT LESS THAN`	Not less than
`IS NOT <`	Not less than
`IS EQUAL TO`	Equal to
`IS =`	Equal to
`IS NOT EQUAL TO`	Not equal to
`IS NOT =`	Not equal to
`IS GREATER THAN OR EQUAL TO`	Greater than or equal to
`IS >=`	Greater than or equal to
`IS LESS THAN OR EQUAL TO`	Less than or equal to
`IS <=`	Less than or equal to

*** relative file**
A file with relative organization.

*** relative key**
A key whose contents identify a logical record in a relative file.

*** relative organization**
The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the logical ordinal position of the record in the file.

*** relative record number**
The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal that is an integer.

*** reserved word**
A COBOL word that is specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as a user-defined word or system-name.

*** resource**
A facility or service, controlled by the operating system, that an executing program can use.

*** resultant identifier**
A user-defined data item that is to contain the result of an arithmetic operation.

routine
A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations.

*** routine-name**
A user-defined word that identifies a procedure written in a language other than COBOL.

RSD file system
The record sequential delimited file system is a workstation file system that supports sequential files. An RSD file supports all COBOL data types in fixed or variable-length records, can be edited by most file editors, and can be read by programs written in other languages. This system only supports sequential files.

*** run time**
The time at which an object program is executed. Synonymous with *object time*.

runtime environment
The environment in which a COBOL program executes.

*** run unit**
A stand-alone object program, or several object programs, that interact by means of COBOL `CALL` statements and function at run time as an entity.

S

SBCS
See *single-byte character set (SBCS)*.

scope terminator
A COBOL reserved word that marks the end of certain `PROCEDURE DIVISION` statements.It can be either explicit (`END-ADD`, for example) or implicit (separator period).

*** section**
A set of zero, one, or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**
A combination of words followed by a separator period that indicates the beginning of a section in any of these divisions: `ENVIRONMENT`, `DATA`, or `PROCEDURE`. In the `ENVIRONMENT DIVISION` and `DATA DIVISION`, a section header is composed of reserved words followed by a separator period. The permissible section headers in the `ENVIRONMENT DIVISION` are:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

The permissible section headers in the DATA DIVISION are:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LOCAL-STORAGE SECTION.
LINKAGE SECTION.
```

In the PROCEDURE DIVISION, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a separator period.

*** section-name**
A user-defined word that names a section in the PROCEDURE DIVISION.

segmentation
A feature of COBOL for Linux that is based on the 85 COBOL Standard segmentation module. The segmentation feature uses priority-numbers in section headers to assign sections to fixed segments or independent segments. Segment classification affects whether procedures contained in a segment receive control in initial state or last-used state.

selection structure
A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

*** sentence**
A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately compiled program**
A program that, together with its contained programs, is compiled separately from all other programs.

*** separator**
A character or two contiguous characters used to delimit character strings.

*** separator comma**
A comma (,) followed by a space used to delimit character strings.

*** separator period**
A period (.) followed by a space used to delimit character strings.

*** separator semicolon**
A semicolon (;) followed by a space used to delimit character strings.

sequence structure
A program processing logic in which a series of statements is executed in sequential order.

*** sequential access**
An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

*** sequential file**
A file with sequential organization.

*** sequential organization**
The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

serial search
A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

SFS file system
The CICS Structured File Server file system is a record-oriented file system that supports sequential, relative, and key-indexed file access.

shared library
A library created by the linker that contains at least one subroutine that can be used by multiple processes. Programs and subroutines are linked as usual, but the code common to different subroutines is combined in one library file that can be loaded at run time and shared by many programs. A key to identify the shared library file is in the header of each subroutine.

*** sign condition**
The proposition (for which a truth value can be determined) that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

signature
The name of an operation and its parameters.

*** simple condition**
Any single condition chosen from this set:

- Relation condition
- Class condition
- Condition-name condition
- Switch-status condition
- Sign condition

See also *condition* and *negated simple condition*.

single-byte character set (SBCS)
A set of characters in which each character is represented by a single byte. See also *ASCII* and *EBCDIC (Extended Binary-Coded Decimal Interchange Code)*.

slack bytes (within records)
Bytes inserted by the compiler between data items to ensure correct alignment of some elementary data items. Slack bytes contain no meaningful data. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment.

slack bytes (between records)
Bytes inserted by the programmer between blocked logical records of a file, to ensure correct alignment of some elementary data items. In some cases, slack bytes between records improve performance for records processed in a buffer.

*** sort file**
A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

*** sort-merge file description entry**
An entry in the `FILE SECTION` of the `DATA DIVISION` that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

*** `SOURCE-COMPUTER`**
The name of an `ENVIRONMENT DIVISION` paragraph in which the computer environment, where the source program is compiled, is described.

*** source computer entry**
An entry in the `SOURCE-COMPUTER` paragraph of the `ENVIRONMENT DIVISION`; this entry contains clauses that describe the computer environment in which the source program is to be compiled.

*** source item**
An identifier designated by a SOURCE clause that provides the value of a printable item.

source program
Although a source program can be represented by other forms and symbols, in this document the term always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the `IDENTIFICATION DIVISION` or a `COPY` statement and terminates with the end program marker, if specified, or with the absence of additional source program lines.

source unit
A unit of COBOL source code that can be separately compiled: a program. Also known as a *compilation unit*.

special character
A character that belongs to the following set:

Character	Meaning
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
$	Currency sign
,	Comma
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
'	Apostrophe
(	Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

SPECIAL-NAMES
The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

*** special names entry**
An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this entry provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

*** special registers**
Certain compiler-generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

*** statement**
A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

STL file system
The standard language file system is the native workstation file system for COBOL. This system supports sequential, relative, and indexed files.

structured programming
A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

*** subject of entry**
An operand or reserved word that appears immediately following the level indicator or the level-number in a DATA DIVISION entry.

*** subprogram**
See *called program*.

*** subscript**
An occurrence number that is represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript can be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**
An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

substitution character
A character that is used in a conversion from a source code page to a target code page to represent a character that is not defined in the target code page.

surrogate pair
In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a *high surrogate* and the second a *low surrogate*. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

switch-status condition
The proposition (for which a truth value can be determined) that an UPSI switch, capable of being set to an on or off status, has been set to a specific status.

*** symbolic-character**
A user-defined word that specifies a user-defined figurative constant.

syntax
(1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

SYSADATA
A file of additional compilation information that is produced if the ADATA compiler option is in effect.

SYSIN
The primary compiler input file or files.

SYSLIB
The secondary compiler input file or files, which are processed if the LIB compiler option is in effect.

SYSPRINT
The compiler listing file.

*** system-name**
A COBOL word that is used to communicate with the operating environment.

T

*** table**
A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

*** table element**
A data item that belongs to the set of repeated items comprising a table.

*** text-name**
A user-defined word that identifies library text.

*** text word**
A character or a sequence of contiguous characters between margin A and margin R in a COBOL library, source program, or pseudo-text that is any of the following characters:

- A separator, except for space; a pseudo-text delimiter; and the opening and closing delimiters for alphanumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.
- A literal including, in the case of alphanumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word COPY bounded by separators that are neither a separator nor a literal.

thread
A stream of computer instructions (initiated by an application within a process) that is in control of a process.

token
In the COBOL editor, a unit of meaning in a program. A token can contain data, a language keyword, an identifier, or other part of the language syntax.

top-down design
The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

top-down development
See *structured programming*.

trailer-label
(1) A label that follows the data records on a unit of recording medium. (2) Synonym for *end-of-file label*.

troubleshoot
To detect, locate, and eliminate problems in using computer software.

*** truth value**
The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

U

*** unary operator**
A plus (+) or a minus (-) sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

Unicode
A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. There are multiple encoding schemes to represent Unicode, including UTF-8, UTF-16, and UTF-32. COBOL for Linux supports Unicode using UTF-16 in little-endian format as the representation for the national data type.

Uniform Resource Identifier (URI)
A sequence of characters that uniquely names a resource; in COBOL for Linux, the identifier of a namespace. URI syntax is defined by the document *Uniform Resource Identifier (URI): Generic Syntax*.

unit
A module of direct access, the dimensions of which are determined by IBM.

*** unsuccessful execution**
The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but can affect status indicators.

UPSI switch
A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

URI
See *Uniform Resource Identifier (URI)*.

*** user-defined word**
A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

*** variable**
A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

variable-length item
A group item that contains a table described with the DEPENDING phrase of the OCCURS clause.

*** variable-length record**
A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable-occurrence data item**
A variable-occurrence data item is a table element that is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry or be subordinate to such an item.

*** variably located group**
A group item following, and not subordinate to, a variable-length table in the same record. The group item can be an alphanumeric group or a national group.

*** variably located item**
A data item following, and not subordinate to, a variable-length table in the same record.

*** verb**
A word that expresses an action to be taken by a COBOL compiler or object program.

volume
A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

VSAM file system
A file system that supports COBOL sequential, relative, and indexed organizations.

VSAM
A generic term for the *STL file system* or *SFS file system*.

W

web service
A modular application that performs specific tasks and is accessible through open protocols like HTTP and SOAP.

white space
Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line

as named in the Unicode Standard.

windowed date field
A date field containing a windowed (two-digit) year. See also *date field* and *windowed year*.

windowed year
A date field that consists only of a two-digit year. This two-digit year can be interpreted using a century window. For example, 10 could be interpreted as 2010. See also *century window*. Compare with *expanded year*.

*** word**
A character string of not more than 30 characters that forms a user-defined word, a system-name, a reserved word, or a function-name.

*** WORKING-STORAGE SECTION**
The section of the DATA DIVISION that describes WORKING-STORAGE data items, composed either of noncontiguous items or WORKING-STORAGE records or of both.

workstation
A generic term for computers, including personal computers, 3270 terminals, intelligent workstations, and UNIX terminals. Often a workstation is connected to a mainframe or to a network.

wrapper
An object that provides an interface between object-oriented code and procedure-oriented code. Using wrappers lets programs be reused and accessed by other systems.

X

x
The symbol in a PICTURE clause that can hold any character in the character set of the computer.

XML
Extensible Markup Language. A standard metalanguage for defining markup languages that was derived from and is a subset of SGML. XML omits the more complex and less-used parts of SGML and makes it much easier to write applications to handle document types, author and manage structured information, and transmit and share structured information across diverse computing systems. The use of XML does not require the robust applications and processing that is necessary for SGML. XML is developed under the auspices of the World Wide Web Consortium (W3C).

XML data
Data that is organized into a hierarchical structure with XML elements. The data definitions are defined in XML element type declarations.

XML declaration
XML text that specifies characteristics of the XML document such as the version of XML being used and the encoding of the document.

XML document
A data object that is well formed as defined by the W3C XML specification.

XML namespace
A mechanism, defined by the W3C XML Namespace specifications, that limits the scope of a collection of element names and attribute names. A uniquely chosen XML namespace ensures the unique identity of an element name or attribute name across multiple XML documents or multiple contexts within an XML document.

Y

year field expansion
Explicit expansion of date fields that contain two-digit years to contain four-digit years in files and databases, and then use of these fields in expanded form in programs. This is the only method for assuring reliable date processing for applications that have used two-digit years.

Z

zoned decimal data item
An external decimal data item that is described implicitly or explicitly as USAGE DISPLAY and that contains a valid combination of PICTURE symbols 9, S, P, and V. The content of a zoned decimal data item is represented in characters 0 through 9, optionally with a sign. If the PICTURE string specifies a sign and the SIGN IS SEPARATE clause is specified, the sign is represented as characters + or -. If SIGN IS SEPARATE is not specified, the sign is one hexadecimal digit that overlays the first 4 bits of the sign position (leading or trailing).

#

77-level-description-entry
A data description entry that describes a noncontiguous data item that has level-number 77.

85 COBOL Standard
The COBOL language defined by the following standards:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module*

2002 COBOL Standard
The COBOL language defined by the following standard:

- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*

2014 COBOL Standard
The COBOL language defined by the following standard:

- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

List of resources

COBOL for Linux publications

Installation Guide, GC28-3116-00

Language Reference, SC28-3117-00

Programming Guide, SC28-3118-00

Support

If you have a problem using COBOL for Linux, visit the IBM Support website, which provides up-to-date support information.

Related publications

DB2 for Linux, UNIX, and Windows

You can find the following publications in the IBM Documentation:

- *Command Reference*
- *Database Administration Concepts and Configuration Reference*
- *SQL reference for Db2 Version 11.1 for Linux, UNIX, and Windows*

TXSeries for Multiplatforms

- *IBM TXSeries for Multiplatforms documentation*

IBM CICS TX on Cloud

- *IBM CICS TX on Cloud documentation*

Unicode and character representation

- *Unicode*, www.unicode.org/
- *International Components for Unicode: Converter Explorer*, http://demo.icu-project.org/icu-bin/convexp/
- *Character Data Representation Architecture: Reference and Registry*, http://www-01.ibm.com/software/globalization/cdra/

XML

- *Extensible Markup Language (XML)*, www.w3.org/XML/
- *Namespaces in XML 1.0*, www.w3.org/TR/xml-names/
- *Namespaces in XML 1.1*, www.w3.org/TR/xml-names11/
- *XML specification*, www.w3.org/TR/xml/

© Copyright IBM Corp. 2021

Index

Special Characters

_iwzGetCCSID: convert code-page ID to CCSID
  example 209
  syntax 209
_iwzGetLocaleCP: get locale and EBCDIC code-page values
  example 209
  syntax 208
-? cob2 option 231, 238
-# cob2 option 225, 231, 238
-c cob2 option 230, 239
-cmain cob2 option 231
-comprc_ok cob2 option 230, 240
-dll cob2 option 240
-Fxxx cob2 option 231, 240
-g cob2 option
  for debugging 231, 241
-host cob2 option
  effect on command-line arguments 407
  effect on compiler options 230
  for host data format 230
-I cob2 option
  searching copybooks 230, 242
-main cob2 option
  specifying main program 231, 243
-o cob2 option
  specifying main program 231, 243
-q cob2 option 230
-q32 cob2 option
  description 231, 238
-q64 cob2 option
  description 238
-shared cob2 option 240
-v cob2 option 231, 244
! character, hexadecimal values 348
? cob2 option 231, 238
.adt file 249
.cbl file suffix 223
.cob file suffix 223
.lst file suffix 229
.profile file, setting environment variables in 213
.wlist file 268
[character, hexadecimal values 348
] character, hexadecimal values 348
*CBL statement 291
*CONTROL statement 291
character, hexadecimal values 348
| character, hexadecimal values 348

Numerics

64-bit mode
  -q64 compiler option
    description 238
  ADDR compiler option 249
  interlanguage communication 387
  programming requirements 250
64-bit mode *(continued)*
  restrictions
    can't mix 64-bit and 32-bit COBOL programs 385
    CICS 330
    overview 250
    SFS files 120
85 COBOL Standard
  definition xix
  options 248

A

abends, using ERRCOUNT runtime option to induce 298
ACCEPT statement
  assigning input 30
  environment variables used in 221
  under CICS 331
active locale 199
ADATA compiler option 249
adding alternate indexes 147
adding records to files
  overview 139
  randomly or dynamically 139
  sequentially 139
ADDR compiler option 249
ADDRESS OF special register
  size depends on ADDR 250
  use in CALL statement 396
addresses
  incrementing 400
  NULL value 400
  passing between programs 400
  passing entry-point addresses 402
alignment depending on ADDR 250
ALL subscript
  examples 80
  processing table elements iteratively 79
  table elements as function arguments 51
ALPHABET clause, establishing collating sequence with 6
alphabetic data
  comparing to national 192
  MOVE statement with 28
alphanumeric comparison 85
alphanumeric data
  comparing
    effect of collating sequence 206
    effect of ZWB 289
    to national 192
  converting
    to national with MOVE 184
    to national with NATIONAL-OF 185
  MOVE statement with 28
alphanumeric date fields, contracting 439
alphanumeric group item
  a group without GROUP-USAGE NATIONAL 21
  definition 20
alphanumeric literals

alphanumeric literals *(continued)*
control characters within 22
description 21
with multibyte content 195
alphanumeric-edited data
initializing
example 25
using INITIALIZE 66
MOVE statement with 28
alternate collating sequence
choosing 157
example 7
alternate index
adding 147
reading duplicates 138
alternate index file
specifying data volume for 145
specifying file name for 115
alternate index, definition 122
ANNUITY intrinsic function 52
APOST compiler option 275
applications, porting
differences between platforms 375
language differences 375
mainframe to workstation
running mainframe applications on the workstation 375
using COPY to isolate platform-specific code 375
workstation to mainframe
file names 379
file suffixes 379
language features 379
nested programs 379
arguments
describing in calling program 397
passing between COBOL and C 392
passing between COBOL and C, example 391
passing BY VALUE 397
passing from COBOL to C, example 390
passing from COBOL to C++, example 392
specifying OMITTED 397
testing for OMITTED arguments 398
to main program 407
ARITH compiler option
description 251
performance considerations 451
arithmetic
calculation on dates
convert date to COBOL integer format (CEECBLDY) 488
convert date to Lilian format (CEEDAYS) 499
convert time stamp to number of seconds (CEESECS) 517
get current Greenwich Mean Time (CEEGMT) 504
COMPUTE statement simpler to code 49
error handling 164
with intrinsic functions 50
arithmetic comparisons 54
arithmetic evaluation
conversions and precision 47
data format conversion 46
examples 53, 55
fixed-point contrasted with floating-point 53
intermediate results 477
arithmetic evaluation *(continued)*
performance tips 445
precedence 50, 479
precision 477
arithmetic expression
as reference modifier 101
description of 50
in nonarithmetic statement 485
in parentheses 50
with MLE 434
arithmetic operation
with MLE 433, 434
arrays
COBOL 33
ASCII
code pages supported in XML documents 346
converting to EBCDIC 106
multibyte portability 378
SBCS portability 376
assembler
programs
listing of 268, 450
ASSIGN clause
assignment-name environment variable 218
identifying files to the operating system 8
precedence for determining file system 116
assigning values 23
assignment-name environment variable 218
assumed century window for nondates 431
AT END (end-of-file) phrase 166

B

base file, CICS SFS 115
base locator 315
BASIS statement 291
batch compilation 275
batch debugging, activating 298
Bibliography 631
big-endian
format for data representation 252, 268, 286
big-endian, converting to little-endian 176
BINARY compiler option 252
binary data item
general description 40
intermediate results 482
synonyms 38
using efficiently 40, 446
binary data, data representation 252
binary search
description 78
example 78
BLANK WHEN ZERO clause
coded for numeric data 178
example with numeric-edited data 37
branch, implicit 89
BY CONTENT 395
BY REFERENCE 395
BY VALUE
description 395
restrictions 397
valid data types 397
byte order mark not generated 364
BYTE-LENGTH intrinsic function

BYTE-LENGTH intrinsic function *(continued)*
using 106

C

C
functions called from COBOL, example 390
functions calling COBOL, example 391, 392
C/C++
and COBOL 387
communicating with COBOL
overview 387
restriction 387
data types, correspondence with COBOL 389
multiple calls to a COBOL program 387
C++
function called from COBOL, example 392
caching
client-side
environment variable for 220
insert caching 148
read caching 148
modules under CICS 333
CALL identifier
example dynamic call 386
call interface conventions
indicating with CALLINT 252
CALL statement
BY CONTENT 395
BY REFERENCE 395
BY VALUE
description 395
restrictions 397
CALL identifier 385
CALL literal 385
effect of CALLINT option 252
exception condition 170
for error handling 170
handling of program-name in 274
overflow condition 170
RETURNING 403
to invoke date and time services 457
USING 397
with DYNAM 262
with ON EXCEPTION 170
with ON OVERFLOW 16, 170
callable services
_iwzGetCCSID: convert code-page ID to CCSID 209
_iwzGetLocaleCP: get locale and EBCDIC code-page values 208
CEECBLDY: convert date to COBOL integer format 488
CEEDATE: convert Lilian date to character format 492
CEEDATM: convert seconds to character time stamp 495
CEEDAYS: convert date to Lilian format 499
CEEDYWK: calculate day of week from Lilian date 502
CEEGMT: get current Greenwich Mean Time 504
CEEGMTO: get offset from Greenwich Mean Time 506
CEEISEC: convert integers to seconds 508
CEELOCT: get current local time 510
CEEQCEN: query the century window 512
CEESCEN: set the century window 513
CEESECI: convert seconds to integers 514
CEESECS: convert time stamp to number of seconds 517
callable services *(continued)*
CEEUTC: get Coordinated Universal Time 521
IGZEDT4: get current date with four-digit year 521
CALLINT compiler option
description 252
CALLINT statement
description 291
calls
between COBOL and C/C++ under CICS 334
dynamic 385
exception condition 170
LINKAGE SECTION 398
OMITTED arguments 397
overflow condition 170
passing arguments 397
passing data 395
receiving parameters 397
recursive 393
static 385
to date and time services 457
CANCEL statement
handling of program-name in 274
case structure, EVALUATE statement for 83
CBL statement
description 291
specifying compiler options 224
CCSID
conflict in XML documents 351
definition 176
in PARSE statement 341
of XML documents 346
of XML documents to be parsed 341
CEECBLDY: convert date to COBOL integer format
example 488
syntax 488
CEEDATE: convert Lilian date to character format
example 493
syntax 492
table of sample output 494
CEEDATM: convert seconds to character time stamp
CEESECI 514
example 496
syntax 495
table of sample output 497
CEEDAYS: convert date to Lilian format
example 501
syntax 499
CEEDYWK: calculate day of week from Lilian date
example 503
syntax 502
CEEGMT: get current Greenwich Mean Time
example 505
syntax 504
CEEGMTO: get offset from Greenwich Mean Time
example 507
syntax 506
CEEISEC: convert integers to seconds
example 509
syntax 508
CEELOCT: get current local time
example 511
syntax 510
CEEQCEN: query the century window
example 512

CEEQCEN: query the century window *(continued)*
syntax 512
CEESCEN: set the century window
example 513
syntax 513
CEESECI: convert seconds to integers
example 516
syntax 514
CEESECS: convert time stamp to number of seconds
example 519
syntax 517
century window
assumed for nondates 431
CEECBLDY 490
CEEDAYS 501
CEEQCEN 512
CEESCEN 513
CEESECS 519
example of querying and changing 464
fixed 424
overview 463
sliding 424
chained-list processing
example 401
overview 400
changing
characters to numbers 105
file-name 8
title on source listing 4
CHAR compiler option
description 253
effect on XML document encodings 346
multibyte portability 378
SBCS portability 376
CHAR intrinsic function, example 107
character set, definition 176
character time stamp
converting Lilian seconds to (CEEDATM)
example 496
converting to COBOL integer format (CEECBLDY)
example 490
converting to Lilian seconds (CEESECS)
example 517
CHECK runtime option
performance considerations 451
reference modification 100
checking errors, flagging at run time 297
checking for valid data
conditional expressions 85
Chinese GB 18030 data
processing 193
CICS
accessing files from non-CICS applications 334
calls between COBOL and C/C++ 334
coding programs to run under
overview 330
commands and the PROCEDURE DIVISION 330
commands relevant to COBOL 329
compiler options 334
Db2 interoperation 118
developing COBOL programs for 329
DFHCOMMAREA parameter for dynamic calls 332
DFHEIBLK parameter for dynamic calls 332
dynamic calls
CICS *(continued)*
dynamic calls *(continued)*
overview 332
performance 333
shared libraries 332
host data format not supported 331
integrated translator
advantages 335
overview 335
module caching 333
performance
module caching 333
overview 443
portability considerations 331
restrictions
Db2 files 330
DYNAM compiler option 262
overview 330
preinitialization 415
separate translator 335
runtime options 334
separate translator
restrictions 335
shared libraries 332
system date, getting 331
TRAP runtime option, effect of 300
CICS compiler option
description 255
enables integrated translator 335
multioption interaction 248
specifying suboptions 255
CICS SFS file system
accessing SFS files
example 146
overview 145
description 119
fully qualified file names 115
nonhierarchical 120
restrictions 120
system administration of 120
CICS SFS files
accessing
example 146
non-CICS 334
overview 145
adding alternate indexes 147
alternate index file name 115
base file name 115
COBOL coding example 146
creating alternate index files 145
creating SFS files
environment variables for 145
sfsadmin command for 146
determining available data volumes 145
error processing 164
file names 115
identifying
server 115
nontransactional access 120
organization 119
primary and secondary indexes 120
processing 116
restriction with GDGs 124
specifying data volume for 145

CICS SFS server
fully qualified name 115
specifying server name 145
CICS_CDS_ROOT environment variable
matching system file-name 115
CICS_SFS_DATA_VOLUME environment variable 219
CICS_SFS_INDEX_VOLUME environment variable 219
CICS_TK_SFS_SERVER environment variable
description 218
identifying SFS server 115
CICS_VSAM_AUTO_FLUSH environment variable 219
CICS_VSAM_CACHE environment variable 220
cicsddt utility 142
cicsmap command 329
cicstcl command 329, 335
cicsterm command 329
class
user-defined 8
class condition
testing
for DBCS 196
for Kanji 196
for numeric 48
overview 85
validating data 302
clustered files 119
cob2 command
command-line argument format 407
description 223
examples
compiling 224
linking 233
modifying configuration file 225
options
-# 225
-host 407
-q abbreviation for ADDR 250
description 230
modifying defaults 225
stanza used 226
cob2 stanza 226
cob2_j command
command-line argument format 407
options
-q abbreviation for ADDR 250
stanza used 225
cob2_j stanza 225
cob2_r command
stanza used 226
cob2_r stanza 226
cob2.cfg configuration file 225
COBCPYEXT environment variable 217
COBLSTDIR environment variable 217
COBOL
and C/C++ 387
called by C functions, example 391, 392
calling C functions, example 390
calling C++ function, example 392
data types, correspondence with C/C++ 389
COBOL for Linux
runtime messages 549
COBOL terms 19
COBOPT environment variable 217
COBPATH environment variable
COBPATH environment variable *(continued)*
CICS dynamic calls 332
description 218
COBRTOPT environment variable 218
code
copy 455
optimized 450
code page
accessing 209
ASCII 200
conflict in XML documents 351
definition 176
EBCDIC 200
EUC 200
euro currency support 56
for alphabetic data item 200
for alphanumeric data item 200
for DBCS data item 200
for national data item 200
hexadecimal values of special characters 348
overriding 186
querying 209
specifying for alphanumeric XML document 348
system default 202
using characters from 200
UTF-8 200
valid 202
code point, definition 176
coded character set
definition 176
in XML documents 346
coding
condition tests 86
DATA DIVISION 9
decisions 81
efficiently 443
ENVIRONMENT DIVISION 5
errors, avoiding 443
EVALUATE statement 83
file input/output 121
for files
example 133
overview 133
for SFS files, example 146
IDENTIFICATION DIVISION 3
IF statement 81
input/output
example 133
overview 133
loops 88
PROCEDURE DIVISION 13
programs to run under CICS
overview 330
steps to follow 329
system date, getting 331
programs to run under Db2
overview 325
restrictions under CICS 330
simplifying 455
SQL statements
overview 326
tables 59
techniques 9, 443
test conditions 86

collating sequence
alphanumeric 206
alternate
choosing 157
example 7
ASCII 6
binary for national keys 157
binary for national sort or merge keys 207
COLLSEQ effect on alphanumeric and DBCS operands 256
controlling 205
DBCS 207
EBCDIC 6
HIGH-VALUE 6
intrinsic functions and 208
ISO 7-bit code 6
LOW-VALUE 6
MERGE 6, 157
national 207
NATIVE 6
NCOLLSEQ effect on national operands 271
nonnumeric comparisons 6
ordinal position of a character 107
portability considerations 377
SEARCH ALL 6
SORT 6, 157
specifying 6
STANDARD-1 6
STANDARD-2 6
symbolic characters in the 7
COLLATING SEQUENCE phrase
does not apply to national keys 157
effect on sort and merge keys 206
overrides PROGRAM COLLATING SEQUENCE clause 6, 157
portability considerations 377
use in SORT or MERGE 157
COLLSEQ compiler option
description 256
effect on alphanumeric collating sequence 205
effect on DBCS collating sequence 207
portability considerations 377
columns in tables 59
command prompt, defining environment variables 213
command-line arguments
example with -host option 409
example without -host option 408
using 407
comment lines 597
comments
sending xxi
COMMON attribute 4, 382
COMP (COMPUTATIONAL) 40
COMP-1 (COMPUTATIONAL-1)
format 41
performance tips 446
COMP-2 (COMPUTATIONAL-2)
format 41
performance tips 446
COMP-3 (COMPUTATIONAL-3) 41
COMP-4 (COMPUTATIONAL-4) 40
COMP-5 (COMPUTATIONAL-5) 40
comparing data items
alphanumeric
comparing data items *(continued)*
alphanumeric *(continued)*
effect of collating sequence 206
effect of COLLSEQ 256
date fields 428
DBCS
effect of collating sequence 207
effect of COLLSEQ 256
literals 196
to alphanumeric groups 207
to national 207
national
effect of collating sequence 207
effect of NCOLLSEQ 191
overview 190
to alphabetic, alphanumeric, or DBCS 192
to alphanumeric groups 192
to numeric 191
two operands 191
zoned decimal and alphanumeric, effect of ZWB 289
compatibility
dates
in comparisons 428
in DATA DIVISION 428
in PROCEDURE DIVISION 428
compatibility between workstation and host 379
compatibility mode 35, 477
compilation
statistics 313
tailoring 226
COMPILE compiler option
description 257
use NOCOMPILE to find syntax errors 305
compile-time considerations
compiler-directed errors 229
compiling programs 230
compiling without linking 230, 239
display cob2 help 231, 238
display compile and link steps 231, 238
error messages
determining what severity level to produce 265
severity levels 228
executing compile and link steps after display 231, 244
using a nondefault configuration file 231, 240
compiler
calculation of intermediate results 478
date-related messages, analyzing 437
generating list of error messages 228
invoking 223
limits
DATA DIVISION 9
messages
choosing severity to be flagged 306
customizing 540
determining what severity level to produce 265
embedding in source listing 306
from exit modules 546
severity levels 228, 541
return code
depends on highest severity 228
effect of message customization 542
overview 228
compiler listings
getting 309

compiler listings *(continued)*
specifying output directory 217
compiler options
abbreviations 245
ADATA 249
ADDR 249
APOST 275
ARITH
description 251
performance considerations 451
BINARY 252
CALLINT 252
CHAR 253
CICS 255
COLLSEQ 256
COMPILE 257
conflicting 248
CURRENCY 258
DATEPROC 259
DATETIME 259
DEFINE 260
DIAGTRUNC 261
DYNAM 262, 451
EXIT 263
FLAG 265, 306
FLAGSTD 266
FLOAT 267
for CICS 334
for debugging
overview 304
TEST restriction 303
THREAD restriction 303
LINECOUNT 268
LIST 268, 309
LSTFILE 269
MAP 269, 309
MAXMEM 270
MDECK 270
NCOLLSEQ 271
NOCOMPILE 305
NSYMBOL 272
NUMBER 272, 311
on compiler invocation 312
OPTIMIZE
description 273
performance considerations 450, 451
performance considerations 451
PGMNAME 274
precedence 248
QUOTE 275
SEPOBJ 275
SEQUENCE 276
SIZE 277
SOSI 277
SOURCE 278, 309
SPACE 279
specifying
cob2 command 223
environment variable 217
in shell script 224
using COBOPT 217
using PROCESS (CBL) 224
specifying compiler options
command line 237
compiler options *(continued)*
SPILL 279
SQL
coding suboptions 327
description 279
SRCFORMAT 280
SSRANGE
performance considerations 451
status 312
table of 245
TERMINAL 282
TEST
description 283
performance considerations 452
THREAD
debugging restriction 303
description 283
TRUNC
description 283
performance considerations 452
UTF16 285
VBREF 286, 309
WSCLEAR
overview 286
performance considerations 287
XREF 287, 308
YEARWINDOW 288
ZWB 289
compiler output 309
compiler-directing statements
description 291
overview 16
compiling
programs 222
setting options 222
tailoring the configuration file 226
completion code
merge 158
sort 158
complex OCCURS DEPENDING ON
basic forms of 73
complex ODO item 73
variably located data item 74
variably located group 74
computation
arithmetic data items 446
constant data items 444
duplicate 445
of indexes 64
of subscripts 448
COMPUTATIONAL (COMP) 40
COMPUTATIONAL-1 (COMP-1)
format 41
performance tips 446
COMPUTATIONAL-2 (COMP-2)
format 41
performance tips 446
COMPUTATIONAL-3 (COMP-3)
date fields, potential problems 439
description 41
COMPUTATIONAL-4 (COMP-4) 40
COMPUTATIONAL-5 (COMP-5) 40
COMPUTE statement
assigning arithmetic results 30

COMPUTE statement *(continued)*
simpler to code 49
computer, describing 5
concatenating data items (STRING) 93
concatenating files
differences from Enterprise COBOL 470
GDGs 131
overview 131
condition handling
date and time services and 458
effect of ERRCOUNT 298
condition testing 86
condition-name 430
conditional expression
EVALUATE statement 81
IF statement 81
PERFORM statement 90
conditional statement
overview 15
with NOT phrase 16
configuration file
default 225
modifying 225
stanzas 227
tailoring 226
CONFIGURATION SECTION 5
conflicting compiler options 248
constants
computations 444
data items 444
definition 22
figurative, definition 22
continuation
of program 164
syntax checking 257
CONTINUE statement 81
contracting alphanumeric dates 439
control
in nested programs 382
program flow 81
transfer 381
CONTROL statement 291
convert character format to Lilian date (CEEDAYS) 499
convert Lilian date to character format (CEEDATE) 492
converting data items
between code pages 106
between data formats 46
precision 47
reversing order of characters 105
to alphanumeric
with DISPLAY 31
with DISPLAY-OF 186
to Chinese GB 18030 from national 193
to integers with INTEGER, INTEGER-PART 102
to national
from Chinese GB 18030 193
from UTF-8 193
with ACCEPT 31
with MOVE 184
with NATIONAL-OF 185
to numbers with NUMVAL, NUMVAL-C 105
to uppercase or lowercase
with INSPECT 103
with intrinsic functions 104
converting data items *(continued)*
to UTF-8 from national 193
with INSPECT 102
with intrinsic functions 104
converting files to expanded date form, example 427
CONVERTING phrase (INSPECT), example 103
coprocessor, Db2
overview 325
using SQL INCLUDE with 326
copy code, obtaining from user-supplied module 264
copy libraries
example 456
COPY name
file suffixes searched 217
COPY statement
description 293
example 456
nested 455, 538
search rules 293
using for portability 375
copybook cross-reference, description 308
copybooks
cross-reference 319
library-name environment variable 217
search rules 293
searching for 230, 242
specifying search paths with SYSLIB 217
using 455
COUNT IN phrase
UNSTRING 95
XML GENERATE 365
counting
characters (INSPECT) 102
generated XML characters 360
creating
alternate index files 145
SFS files
environment variables for 145
sfsadmin command for 146
variable-length tables 70
cross-reference
COPY/BASIS 319
COPY/BASIS statements 309
copybooks 309
data and procedure-names 308
embedded 309
list 287
program-name 319
special definition symbols 320
statement list 286
statements 309
text-names and file names 308
cultural conventions, definition 199
CURRENCY compiler option 258
currency signs
euro 56
hexadecimal literals 56
multiple-character 56
using 56
CURRENT-DATE intrinsic function
example 52
under CICS 332
customer support 631
customizing

customizing *(continued)*
setting environment variables 213

D

data
concatenating (STRING) 93
efficient execution 443
format conversion of 46
format, numeric types 38
grouping 399
incompatible 48
naming 10
numeric 35
passing 395
record size 10
splitting (UNSTRING) 95
validating 48
data and procedure-name cross-reference, description 308
data areas, dynamic 262
data definition 314
data definition attribute codes 314
data description entry 9
DATA DIVISION
coding 9
description 9
FD entry 9
FILE SECTION 9
GROUP-USAGE NATIONAL clause 60
limits 9
LINKAGE SECTION 9, 13
listing 309
LOCAL-STORAGE SECTION 9
mapping of items 269, 309
OCCURS clause 59
OCCURS DEPENDING ON (ODO) clause 70
REDEFINES clause 67
restrictions 9
USAGE clause at the group level 21
USAGE IS INDEX clause 64
USAGE NATIONAL clause at the group level 184
WORKING-STORAGE SECTION 9
data item
alignment depends on ADDR 250
common, in subprogram linkage 397
concatenating (STRING) 93
converting characters (INSPECT) 102
converting characters to numbers 105
converting to uppercase or lowercase 104
converting with intrinsic functions 104
counting characters (INSPECT) 102
elementary, definition 20
evaluating with intrinsic functions 106
finding the smallest or largest item 107
group, definition 20
index, referring to table elements with 62
initializing, examples of 24
numeric 35
reference modification 99
referring to a substring 99
replacing characters (INSPECT) 102
reversing characters 105
splitting (UNSTRING) 95
unused 273
data item *(continued)*
variably located 74
data manipulation
character data 93
DATA RECORDS clause 10
data representation
compiler option affecting 252, 285
portability 376
data types, correspondence between COBOL and C/C++ 389
data-name
cross-reference 317
in MAP listing 314
date and time
format
converting from character format to COBOL integer format (CEECBLDY) 488
converting from character format to Lilian format (CEEDAYS) 499
converting from integers to seconds (CEEISEC) 508
converting from Lilian format to character format (CEEDATE) 492
converting from seconds to character time stamp (CEEDATM) 495
converting from seconds to integers (CEESECI) 514
converting from time stamp to number of seconds (CEESECS) 517
getting date and time (CEELOCT) 510
intrinsic functions 488
picture strings
examples 462
overview 460
services
CEECBLDY: convert date to COBOL integer format 488
CEEDATE: convert Lilian date to character format 492
CEEDATM: convert seconds to character time stamp 495
CEEDAYS: convert date to Lilian format 499
CEEDYWK: calculate day of week from Lilian date 502
CEEGMT: get current Greenwich Mean Time 504
CEEGMTO: get offset from Greenwich Mean Time 506
CEEISEC: convert integers to seconds 508
CEELOCT: get current local time 510
CEEQCEN: query the century window 512
CEESCEN: set the century window 513
CEESECI: convert seconds to integers 514
CEESECS: convert time stamp to number of seconds 517
CEEUTC: get Coordinated Universal Time 521
condition feedback 459
condition handling 458
examples of using 458
feedback code 457
invoking with a CALL statement 457
list of 487
overview 487
performing calculations with 458
return code 457
RETURN-CODE special register 457
syntax 517
date arithmetic 434

date comparisons 428
date field expansion
advantages 423
description 426
date fields, potential problems with 439
DATE FORMAT clause
cannot use with national data 422
use for automatic date recognition 421
date information, formatting 214
date operations
finding date of compilation 110
intrinsic functions for 32
date processing with internal bridges, advantages 423
date windowing
advantages 423
example 425, 430
how to control 435
MLE approach 424
when not supported 429
DATE-COMPILED paragraph 3
DATE-OF-INTEGER intrinsic function 52
DATEPROC compiler option
analyzing warning-level messages 437
description 259
DATETIME compiler option 259
DATEVAL intrinsic function
example 437
using 436
day of week, calculating with CEEDYWK 502
Db2
bind file name 328
coding considerations 325
coprocessor
overview 325
using SQL INCLUDE with 326
ignored options 327
options 327
package name 328
precompiler requires NODYNAM 325
precompiler restrictions 325
SQL statements
coding 326
overview 325
return codes 327
SQL INCLUDE 326
using binary data in 327
Db2 file system
accessing Db2 files 142
CICS interoperation
requirements 118
creating Db2 files 142
description 117
nonhierarchical 118
restrictions 118
using
overview 142
with SQL statements 143
DB2 file system
system administration of 117
Db2 files
accessing 142
CICS interoperation
requirements 118
setting up 142
Db2 files *(continued)*
creating 142
error processing 164
identifying
overview 113
schema 114
processing 116
schema
default 114
specifying 114
using
overview 142
with SQL statements 143
db2 utility
db2 connect, example 142
db2 create, example 142
db2 describe, example 117
overview 117
DB2DBDFT environment variable 214, 327
DB2INCLUDE environment variable 326
DBCS comparison 85
DBCS data
comparing
effect of collating sequence 207
literals 196
to alphanumeric groups 207
to national 192, 207
converting
to national, overview 196
declaring 195
encoding and storage 190
literals
comparing 196
description 22
maximum length 195
using 195
MOVE statement with 28
testing for 196
DEBUG runtime option 298
debugging
activating batch features 298
compiler options for
overview 304
TEST restriction 303
THREAD restriction 303
overview 301
producing symbolic information 231, 241
runtime options for 303
using COBOL language features 301
with message offset information 321
debugging, language features
class test 302
debugging lines 303
debugging statements 303
declaratives 303
DISPLAY statements 301
file status keys 302
INITIALIZE statements 303
scope terminators 301
SET statements 303
WITH DEBUGGING MODE clause 303
declarative procedures
EXCEPTION/ERROR 166
USE FOR DEBUGGING 303

DEFINE compiler option 260
defining
  files
    example 133
    overview 133
  SFS files, example 146
DELETE statement
  compiler-directing 295
deleting records from files 140
delimited scope statement
  description of 15
  nested 17
depth in tables 61
DESC suboption of CALLINT compiler option 253
DESCRIPTOR suboption of CALLINT compiler option 253
DFHCOMMAREA parameter
  use in CICS dynamic calls 332
DFHEIBLK parameter
  use in CICS dynamic calls 332
diagnostics, program 312
DIAGTRUNC compiler option 261
differences from host COBOL 467
direct-access
  direct indexing 64
directories
  adding a path to 230, 242
  for listing file 229
dirty read 148
DISPLAY (USAGE IS)
  encoding and storage 190
  external decimal 39
  floating point 40
display device, sending messages to 282
display floating-point data (USAGE DISPLAY) 39
DISPLAY statement
  displaying data values 31
  environment variables used in 221
  using in debugging 301
DISPLAY-1 (USAGE IS)
  encoding and storage 190
DISPLAY-OF intrinsic function
  example with Chinese data 194
  example with Greek data 186
  example with UTF-8 data 193
  using 186
  with XML documents 347
do loop 90
do-until 90
do-while 90
document encoding declaration 347
documentation of program 5
dumps, TRAP(OFF) side effect 300
duplicate computations, grouping 445
DYNAM compiler option
  description 262
  effect on CALL literal 385
  performance considerations 451
dynamic calls
  cannot use for Db2 APIs 325
  CICS
    overview 332
    performance 333
    shared libraries 332
  example of CALL identifier 386
dynamic linking
  definition 385
  resolution of shared library references 412
dynamic loading, requirements for 218

E

E-level error message 228, 306
EBCDIC
  code pages supported in XML documents 346
  converting to ASCII 106
  multibyte portability 378
  SBCS portability 376
EBCDIC_CODEPAGE environment variable
  setting 219
efficiency of coding 443
EJECT statement 295
embedded cross-reference
  description 309
  example 320
embedded error messages 307
embedded MAP summary 309, 315
Encina SFS file system
  performance 147
Encina SFS files
  performance 147
encoding
  conflicts in XML documents 351
  controlling in generated XML output 364
  description 190
  language characters 176
  of XML documents 346
  of XML documents to be parsed 341
  specifying for alphanumeric XML document 348
encoding declaration
  preferable to omit 348
  specifying 348
end-of-file (AT END phrase) 166
enhancing XML output
  example of modifying data definitions 369
  rationale and techniques 369
ENTER statement 295
entry point
  alternate in ENTRY statement 403
  definition 411
  function-pointer data item 402
  passing addresses of 402
  procedure-pointer data item 402
ENTRY statement
  for alternate entry points 403
  handling of program-name in 274
environment differences, IBM Z and the workstation 378
ENVIRONMENT DIVISION
  collating sequence coding 6
  CONFIGURATION SECTION 5
  description 5
  INPUT-OUTPUT SECTION 5
environment variables
  accessing 213
  accessing files with 218
  assignment-name 218
  CICS_CDS_ROOT
    matching system file-name 115
  CICS_SFS_DATA_VOLUME 219

environment variables *(continued)*
CICS_SFS_INDEX_VOLUME 219
CICS_TK_SFS_SERVER
description 218
identifying SFS server 115
CICS_VSAM_AUTO_FLUSH 219
CICS_VSAM_CACHE 220
COBCPYEXT 217
COBLSTDIR 217
COBOPT 217
COBPATH
CICS dynamic calls 332
description 218
COBRTOPT 218
compiler 216
compiler and runtime 214
DB2DBDFT 214
definition 213
EBCDIC_CODEPAGE 219
example of setting and accessing 221
LANG 201, 214
LC_ALL 201, 214
LC_COLLATE 201, 214
LC_CTYPE 201, 214
LC_MESSAGES 201, 214
LC_TIME 201, 214
library-name 217, 293
NLSPATH 215
PATH
description 221
precedence of paths 213
runtime 218
setting
in .profile 213
in command shell 213
in program 213
locale 201
overview 213
SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 221
SYSLIB 217
text-name 217, 293
TMP 215
TZ 215
environment-name 5
environment, preinitializing
example 417
for C/C++ program 387
overview 415
ERRCOUNT runtime option 298
ERRMSG, for generating list of error messages 228
error
arithmetic 164
compiler options, conflicting 248
flagging at run time 297
handling 163
message table
example using indexing 69
example using subscripting 68
processing
XML GENERATE 364
XML PARSE 350
error messages
compiler
error messages *(continued)*
compiler *(continued)*
choosing severity to be flagged 306
correcting source 227
customizing 540
determining what severity level to produce 265
embedding in source listing 306
format 229
from exit modules 546
generating a list of 228
location in listing 229
severity levels 228, 541
compiler-directed 229
runtime
format 549
incomplete or abbreviated 235
list of 549
setting national language 214
EUC code page 200
euro currency sign 56
EVALUATE statement
case structure 83
coding 83
contrasted with nested IFs 84
example that tests several conditions 84
example with multiple WHEN phrases 84
example with THRU phrase 83
performance 83
structured programming 444
testing multiple values, example 87, 88
use to test multiple conditions 81
evaluating data item contents
class test
for numeric 48
overview 85
INSPECT statement 102
intrinsic functions 106
example
_iwzGetCCSID: convert code-page ID to CCSID 209
_iwzGetLocaleCP: get locale and EBCDIC code-page values 209
examples
CEECBLDY: convert date to COBOL integer format 490
CEEDATE: convert Lilian date to character format 493
CEEDATM: convert seconds to character format 496
CEEDAYS: convert date to Lilian format 501
CEEDYWK: calculate day of week from Lilian date 503
CEEGMT: get current GMT 505
CEEGMTO: get offset from Greenwich Mean Time 507
CEEISEC: convert integers to seconds 509
CEELOCT: get current local time 511
CEEQCEN: query century window 512
CEESCEN: set century window 513
CEESECI: convert seconds to integers 516
CEESECS: convert time stamp to number of seconds 519
IGZEDT4: get current date with four-digit year 521
exception condition
CALL 170
XML GENERATE 365
XML PARSE 350
exception handling
with XML GENERATE 364
with XML PARSE 349

EXCEPTION XML event 350
EXCEPTION/ERROR declarative
  description 166
  file status key 167
exceptions, intercepting 300
EXIT compiler option
  character string formats 265
  description 263
  INEXIT suboption 537
  LIBEXIT suboption 538
  MSGEXIT suboption 539
  parameter list 536
  PRTEXIT suboption 539
  user-exit work area 535
  user-exit work area extension 535
  using 263
exit modules
  error messages generated 546
  loading and invoking 537
  message severity customization 539
  parameter list 536
  when used in place of library-name 538
  when used in place of SYSLIB 538
  when used in place of SYSPRINT 539
EXIT PROGRAM statement
  in main program 381
  in subprogram 382
explicit scope terminator 16
exponentiation
  evaluated in fixed-point arithmetic 480
  evaluated in floating-point arithmetic 485
  performance tips 447
export command
  defining environment variables 213
  precedence of paths 213
extended mode 35, 477
EXTERNAL clause
  example for files 405
  for data items 404
  for sharing files 10, 404
external code page, definition 347
external data
  sharing 404
external decimal data
  national 39
  zoned 39
external file 404
external floating-point data
  display 39
  national 40

F

factoring expressions 444
FD (file description) entry 10
feedback
  sending xxi
feedback token
  date and time services and 459
figurative constants
  definition 22
  HIGH-VALUE restriction 182
  national-character 182
file access mode
file access mode *(continued)*
  dynamic 123
  for indexed files 122
  for line-sequential files 122
  for sequential files 121
  random 123
  relative files 122
  sequential 122
  summary table of 121
file conversion
  with millennium language extensions 426
file description (FD) entry 10
file organization
  indexed 122
  line-sequential 122
  overview 121
  QSAM 144
  relative 122
  sequential 121
file position indicator
  overview 135
  setting with START 138
FILE SECTION
  DATA RECORDS clause 10
  description 9
  EXTERNAL clause 10
  FD entry 10
  GLOBAL clause 10
  RECORD CONTAINS clause 10
  record description 9
  RECORD IS VARYING 10
  RECORDING MODE clause 10
  VALUE OF 10
FILE STATUS clause
  example 170
  file loading 138
  using 166
  with status code
    differences from host 378
    example 169
    overview 168
file status code
  differences from host 378
  example 169
  using 168
file status key
  00 138
  02 138
  05 131, 132
  35 132
  49 140
  92 140
  checking for I/O errors 166
  checking for successful OPEN 166, 168
  error handling 302
  setting up 132
  used with status code
    differences from host 378
    example 169
    overview 168
file suffixes
  for messages listing 229
  passed to the compiler 223
FILE-CONTROL paragraph, example 5

file-system support
Db2 299
Encina SFS 299
FILESYS runtime option 298
precedence for determining file system 116
QSAM 299
RSD 299
SdU 299
SFS (Encina) 299
STL 299
VSAM implies SFS or SdU 299
files
accessing using environment variables 218
adding records to 139
associating program files to external files 5
available 132
changing name 8
CICS SFS
accessing 334
identifying 113
using 145
clustered 119
COBOL coding
example 133
overview 133
comparison of file organizations 121
concatenating
differences from Enterprise COBOL 470
GDGs 131
overview 131
concepts and terminology 111
Db2
identifying 113
using 142
deleting records from 140
describing 9
external 404
file position indicator
overview 135
setting with START 138
FILESYS runtime option, effect of 298
generation data groups (GDGs) 123
identifying
to the operating system 8
within your program 112
line-sequential 122
linker
library 234
LSQ 113
multiple, compiling 223
nonexistent 132
opening
optionally 132
overview 135
protecting against errors 132
optional 132
precedence for determining file system 116
processing
CICS SFS files 116
Db2 files 116
QSAM files 116
RSD files 117
SdU files 116
SFS (CICS) files 116
files *(continued)*
processing *(continued)*
STL files 116
protecting against errors when opening 132
QSAM
using 144
reading records from 137
replacing records in 140
RSD 113
SdU
identifying 113
SFS (CICS)
accessing 334
identifying 113
using 145
STL
identifying 113
TRAP runtime option, effect of 300
updating records in 140
usage explanation 8
VSA implies SFS or SdU 113
FILESYS runtime option
description 298
FIPS messages
categories 541
FLAGSTD compiler option 266
fixed century window 424
fixed-point arithmetic
comparisons 54
evaluation 53
example evaluations 55
exponentiation 480
fixed-point data
binary 40
conversions and precision 47
conversions between fixed- and floating-point 46
external decimal 39
intermediate results 479
packed-decimal 41
planning use of 445
FLAG compiler option
compiler output 307
description 265
using 306
Flag option 237
flags and switches 86
FLAGSTD compiler option 266
FLOAT compiler option 267
floating comment indicators (*>) 606
floating-point arithmetic
comparisons 54
evaluation 53
example evaluations 55
exponentiation 485
floating-point data
conversions and precision 47
conversions between fixed- and floating-point 46
external 39
intermediate results 484
internal
format 41
performance tips 446
performance considerations 377
planning use of 445

floating-point data *(continued)*
portability 377
full date field expansion, advantages 423
function-pointer data item
definition 402
passing parameters to callable services 402
size depends on ADDR 250

G

GB 18030 data
converting to or from national 193
processing 193
gdgmgr utility 124
GDGs (generation data groups)
catalog 126
concatenation 131
creating 124
differences from Enterprise COBOL 469
gdgmgr utility 124
limit processing
example 130
overview 130
overview 123
restrictions 124
using 126
GDSs (generation data sets)
absolute names 127
insertion and wrapping 128
relative names 127
generating XML output
example 365
overview 359
generation data groups (GDGs)
catalog 126
concatenation 131
creating 124
differences from Enterprise COBOL 469
gdgmgr utility 124
limit processing
example 130
overview 130
overview 123
restrictions 124
using 126
generation data sets (GDSs)
absolute names 127
insertion and wrapping 128
relative names 127
getenv() to access environment variables 213
GLOBAL clause for files 10, 13
global names 384
Glossary 591
GOBACK statement
in main program 381
in subprogram 382
Greenwich Mean Time (GMT)
getting offset to local time (CEEGMTO) 506
return Lilian date and Lilian seconds (CEEGMT) 504
Gregorian character string
returning local time as a (CEELOCT)
example 511
group item
group item *(continued)*
cannot subordinate alphanumeric group within national group 188
comparing to national data 192
definition 20
for defining tables 59
group move contrasted with elementary move 29, 188
initializing
using a VALUE clause 68
using INITIALIZE 27, 65
MOVE statement with 29
passing as an argument 399
treated as a group item
example with INITIALIZE 65
in INITIALIZE 28
variably located 74
group move contrasted with elementary move 29, 188
GROUP-USAGE NATIONAL clause
defining a national group 187
defining tables 60
example of declaring a national group 20
initializing a national group 27
grouping data to pass as an argument 399

H

header on listing 4
help files
setting national language 214
specifying path name 215
hexadecimal
portability 377
hexadecimal literals
as currency sign 56
national
description 22
using 178

I

I-level message 228, 306
IBM Z host data format
considerations 475
IDENTIFICATION DIVISION
coding 3
DATE-COMPILED paragraph 3
listing header example 4
PROGRAM-ID paragraph 3
required paragraphs 3
TITLE statement 4
IEEE
portability 377
IF statement
coding 81
nested 82
use EVALUATE instead for multiple conditions 81
with null branch 81
IGZEDT4: get current date with four-digit year 521
imperative statement, list 15
implicit scope terminator 16
incrementing addresses 400
index
assigning a value to 64

index *(continued)*
CICS SFS files 120
computation of element displacement, example 62
creating with OCCURS INDEXED BY clause 64
definition 62
incrementing or decrementing 64
initializing 64
key, detecting faulty 170
range checking 306
referencing other tables with 64
index data item
cannot use as subscript or index 64
creating with USAGE IS INDEX clause 64
size depends on ADDR 250
indexed file organization 122
indexed files
CICS SFS files 120
file access mode 122
indexing
computation of element displacement, example 62
definition 62
example 69
preferred to subscripting 447
tables 64
INITIAL clause
effect on main program 382
effect on nested programs 4
setting programs to initial state 4
INITIALIZE statement
examples 24
loading group values 27
loading national group values 27
loading table values 65
REPLACING phrase 65
using for debugging 303
initializing
a group item
using a VALUE clause 68
using INITIALIZE 27, 65
a national group item
using a VALUE clause 68
using INITIALIZE 27, 66
a structure using INITIALIZE 27
a table
all occurrences of an element 68
at the group level 68
each item individually 67
using INITIALIZE 65
using PERFORM VARYING 90
examples 24
runtime environment
overview 415
the runtime environment
example 417
variable-length group 72
inline PERFORM
example 89
overview 89
input
from files 111
overview 121
input procedure
coding 153
example 157
input procedure *(continued)*
requires RELEASE or RELEASE FROM 154
restrictions 155
INPUT-OUTPUT SECTION 5
input/output
checking for errors 166
coding
example 133
overview 133
introduction 111
logic flow after error 164
processing errors
CICS SFS files 164
Db2 files 164
SdU files 164
SFS (CICS) files 164
STL files 164
input/output coding
AT END (end-of-file) phrase 166
checking for successful operation 166
checking status code
differences from host 378
example 169
overview 168
detecting faulty index key 170
error handling techniques 164
EXCEPTION/ERROR declaratives 166
insert cache 148
INSERT statement 295
INSPECT statement
avoid with UTF-8 data 349
examples 103
using 102
inspecting data (INSPECT) 102
INTEGER intrinsic function, example 102
INTEGER-OF-DATE intrinsic function 52
INTEGER-PART intrinsic function 102
integers
converting Lilian seconds to (CEESECI) 514
integrated CICS translator
advantages 335
overview 335
integrated translator 335
interlanguage communication
between COBOL and C/C++
overview 387
restriction 387
intermediate results 477
internal bridges
advantages 423
example 426
for date processing 425
internal floating-point data (COMP-1, COMP-2) 41
intrinsic functions
as reference modifiers 102
collating sequence, effect of 208
compatibility with CEELOCT 510
converting alphanumeric data items with 104
converting national data items with 104
date and time 488
DATEVAL
example 437
using 436
evaluating data items 106

intrinsic functions *(continued)*
example of
ANNUITY 52
CHAR 107
CURRENT-DATE 52
DISPLAY-OF 186
INTEGER 102
INTEGER-OF-DATE 52
LENGTH 52, 108, 109
LOG 53
LOWER-CASE 104
MAX 52, 80, 107, 108
MEAN 53
MEDIAN 53, 80
MIN 102
NATIONAL-OF 186
NUMVAL 105
NUMVAL-C 52, 105
ORD 107
ORD-MAX 80, 107
PRESENT-VALUE 52
RANGE 53, 80
REM 53
REVERSE 105
SQRT 53
SUM 80
UPPER-CASE 104
WHEN-COMPILED 110
finding date of compilation 110
finding largest or smallest item 107
finding length of data items 109
intermediate results 482, 485
introduction to 32
nesting 33
numeric functions
examples of 50
integer, floating-point, mixed 50
nested 51
special registers as arguments 51
table elements as arguments 51
uses for 50
processing table elements 79
UNDATE
example 437
using 436
INVALID KEY phrase
description 170
example 170
INVOKE statement
with PROCEDURE DIVISION RETURNING 403
invoking
compiler and linker 223
date and time services 457
iwzGetSortErrno, obtaining sort or merge error number with 159

J

Java
libraries
specified in cob2.cfg 225
JNI
libraries
specified in cob2.cfg 225
JNIEnvPtr special register
size depends on ADDR 250

K

Kanji comparison 85
Kanji data, testing for 196
keys
for binary search 78
for merging
default 206
defining 156
overview 152
for sorting
default 206
defining 156
overview 151
permissible data types
in MERGE statement 157
in OCCURS clause 60
in SORT statement 157
to specify order of table elements 60
keyword 610

L

LABEL declarative
description 295
LANG environment variable 214
largest or smallest item, finding 107
last-used state
subprograms with EXIT PROGRAM or GOBACK 382
lazy write
enabling 149
environment variable for 219
LC_ALL environment variable 214
LC_COLLATE environment variable 214
LC_CTYPE environment variable 214
LC_MESSAGES environment variable 214
LC_TIME environment variable 214
LENGTH intrinsic function
compared with LENGTH OF special register 109
example 52, 109
result size depends on ADDR 250
using 106
variable-length results 108
with national data 109
length of data items, finding 109
LENGTH OF special register
passing 396
size depends on ADDR 250
using 109
level-88 item
conditional expressions 85
for windowed date fields 430
restriction 431
setting switches off, example 88
setting switches on, example 88
switches and flags 86
testing multiple values, example 87
testing single values, example 86
level-number 314
library file 234

library text
specifying path for 217
library-name
alternative if not specified 230, 242
specifying path for 293
specifying path for library text 217
when not used 538
Lilian date
calculate day of week from (CEEDYWK) 502
convert date to (CEEDAYS) 499
convert date to COBOL integer format (CEECBLDY) 488
convert output_seconds to (CEEISEC) 508
convert to character format (CEEDATE) 492
get current local date or time as a (CEELOCT) 510
get GMT as a (CEEGMT) 504
using as input to CEESECI 516
limits of the compiler
DATA DIVISION 9
user data 9
line number 313
line-sequential files
file access mode 122
organization 122
LINECOUNT compiler option 268
linkage conventions
compiler directive CALLINT for 291
compiler option CALLINT for 252
LINKAGE SECTION
coding 398
for describing parameters 397
with recursive calls 13
with the THREAD option 13
linkages, data 389
linked-list processing, example 401
linker
errors 235
file defaults 235
files
library 234
invoking 223, 232
resolution of references to shared libraries 412
search rules 234
specifying options 232
linking
examples 233
programs 232
static 411
LIST compiler option
description 268
getting output 309
use in debugging 321
List of resources 631
listing output 309
listings
data and procedure-name cross-reference 308
embedded error messages 306
generating a short listing 309
line numbers, user-supplied 311
sorted cross-reference of program-names 319
sorted cross-reference of text-names 319
terms used in MAP output 316
text-name cross-reference 308
literals
alphanumeric
literals *(continued)*
alphanumeric *(continued)*
control characters within 22
description 21
with multibyte content 195
DBCS
description 22
maximum length 195
using 195
definition 21
hexadecimal
using 178
national
description 22
using 178
numeric 21
using 21
little-endian
format for data representation 252, 268, 286
little-endian, converting to big-endian 176
loading a table dynamically 65
local names 384
local time
getting (CEELOCT) 510
LOCAL-STORAGE SECTION
comparison with WORKING-STORAGE
example 11
overview 11
locale
accessing 208
and messages 202
cultural conventions, definition 199
default 202
definition 199
effect of COLLSEQ compiler option 206
effect of PROGRAM COLLATING SEQUENCE 206
locale-based collating 205
querying 208
shown in listing 308, 313
specifying 214
supported values 202
value syntax 201
LOG intrinsic function 53
loops
coding 88
conditional 90
do 90
in a table 90
performed an explicit number of times 90
LOWER-CASE intrinsic function 104
lowercase, converting to 104
LSQ files
identifying 113
lst file suffix 229
LSTFILE compiler option 269

M

main entry point
specifying with cob2 231
main program
and subprograms 381
arguments to 407
specifying with cob2 231, 243

MAP compiler option
description 269
embedded MAP summary 309
example 314, 317
nested program map
example 317
symbols used in output 316
terms used in output 316
using 309
mapping of DATA DIVISION items 309
mathematics
intrinsic functions 50, 53
MAX intrinsic function
example table calculation 80
example with functions 52
using 107
MAXMEM compiler option 270
MDECK compiler option
description 270
multioption interaction 248
MEAN intrinsic function
example statistics calculation 53
example table calculation 80
MEDIAN intrinsic function
example statistics calculation 53
example table calculation 80
merge
alternate collating sequence 157
completion code 158
criteria 156
description 151
determining success 158
diagnostic message 158
error number
list of possible values 159
obtaining with iwzGetSortErrno 159
files, describing 152
keys
default 206
defining 156
overview 152
process 151
terminating 162
work files
describing 152
TMP environment variable 215
MERGE statement
ASCENDING|DESCENDING KEY phrase 156
COLLATING SEQUENCE phrase 6, 157
description 156
GIVING phrase 156
overview 151
USING phrase 156
message catalogs
specifying path name 215
messages
compiler
choosing severity to be flagged 306
customizing 540
date-related 437
determining what severity level to produce 265
embedding in source listing 306
generating a list of 228
millennium language extensions 437
messages *(continued)*
compiler *(continued)*
severity levels 228, 541
compiler-directed 229
from exit modules 546
national language support 202
offset information 321
runtime
effect of ERRCOUNT 298
format 549
incomplete or abbreviated 235
list of 549
sending to display device 282
setting national language 214
TRAP(OFF) side effect 300
millennium language extensions
assumed century window 431
compatible dates 428
concepts 422
date windowing 421
DATEPROC compiler option 259
nondates 432
objectives 423
principles 422
YEARWINDOW compiler option 288
MIN intrinsic function
example 102
using 107
MLE 422
mnemonic-name
SPECIAL-NAMES paragraph 5
module caching under CICS 333
MOVE statement
assigning arithmetic results 30
converting to national data 184
CORRESPONDING 29
effect of ODO on lengths of sending and receiving items 71
group move contrasted with elementary move 29, 188
with elementary receiving items 28
with group receiving items 29
with national items 28
MSGEXIT suboption of EXIT option
effect on compilation return code 542
example user exit 542
message severity levels 541
processing of 539
syntax 264
multiple currency signs
example 56
using 56
multiple thread environment, running in 283
multithreading
effect on return code 394

N

N delimiter for national or DBCS literals 22
name declaration
searching for 385
naming
programs 3
NATIONAL (USAGE IS)
external decimal 39

NATIONAL (USAGE IS) *(continued)*
floating point 40
national comparison 85
national data
cannot use with DATE FORMAT clause 422
comparing
effect of collating sequence 207
effect of NCOLLSEQ 191
overview 190
to alphabetic, alphanumeric, or DBCS 192
to alphanumeric groups 192
to numeric 191
two operands 191
concatenating (STRING) 93
converting
from alphanumeric or DBCS with NATIONAL-OF 185
from alphanumeric, DBCS, or integer with MOVE 184
overview 184
to alphanumeric with DISPLAY-OF 186
to numbers with NUMVAL, NUMVAL-C 105
to or from Chinese GB 18030 193
to or from Greek alphanumeric, example 186
to or from UTF-8 193
to uppercase or lowercase 104
with INSPECT 102
defining 177
encoding in XML documents 346
evaluating with intrinsic functions 106
external decimal 39
external floating-point 40
figurative constants 182
finding the smallest or largest item 107
in conditional expressions 190, 191
in generated XML documents 359
in keys
in MERGE statement 157
in OCCURS clause 60
in SORT statement 157
initializing, example of 25
input with ACCEPT 31
inspecting (INSPECT) 102
LENGTH intrinsic function and 109
LENGTH OF special register 109
literals
using 178
MOVE statement with 28, 184
NSYMBOL compiler option if no USAGE clause 178
output with DISPLAY 31
reference modification of 99
reversing characters 105
specifying 177
splitting (UNSTRING) 96
VALUE clause with alphanumeric literal, example 108
national decimal data (USAGE NATIONAL)
defining 183
example 35
format 39
initializing, example of 26
national floating-point data (USAGE NATIONAL)
defining 183
definition 40
national group item
national group item *(continued)*
advantages over alphanumeric groups 183
can contain only national data 20, 188
contrasted with USAGE NATIONAL group 21
defining 187
example 20
for defining tables 60
in generated XML documents 359
initializing
using a VALUE clause 68
using INITIALIZE 27, 66
LENGTH intrinsic function and 109
MOVE statement with 29
overview 183
passing as an argument 399
treated as a group item
example with INITIALIZE 188
in INITIALIZE 28
in MOVE CORRESPONDING 29
summary 189
treated as an elementary item
example with MOVE 29
in most cases 20, 183
using
as an elementary item 188
overview 187
VALUE clause with alphanumeric literal, example 68
national language support
messages 202
national language support (NLS)
accessing locale and code-page values 208
collating sequence 205
DBCS 194
locale 199
locale-based collating 205
processing data 175
setting the locale 199
specifying locale and code page 214
national literals
description 22
using 178
national-edited data
defining 178
editing symbols 178
initializing
example 25
using INITIALIZE 66
MOVE statement with 28
PICTURE clause 178
NATIONAL-OF intrinsic function
example with Chinese data 194
example with Greek data 186
example with UTF-8 data 193
using 185
with XML documents 347
native files 122
native format
-host option effect on command-line arguments 407
BINARY option 252
CHAR option 253
FLOAT option 267, 268
UTF16 option 286
NCOLLSEQ compiler option
description 271

NCOLLSEQ compiler option *(continued)*
effect on national collating sequence 205, 207
effect on national comparisons 191
effect on sort and merge keys 157
nested COPY statement 455, 538
nested delimited scope statements 17
nested IF statement
coding 82
CONTINUE statement 81
EVALUATE statement preferred 82
with null branches 81
nested intrinsic functions 51
nested program map
description 309
example 317
nested programs
calling 382
description 383
effect of INITIAL clause 4
guidelines 382
map 309, 317
scope of names 384
transfer of control 382
nesting level
program 313, 317
statement 313
NLSPATH environment variable 215
NOCOMPILE compiler option
use to find syntax errors 305
NODESC suboption of CALLINT compiler option 253
NODESCRIPTOR suboption of CALLINT compiler option 253
nondates with MLE 432
NOSSRANGE compiler option
effect on checking errors 297
Notices 587
NSYMBOL compiler option
description 272
effect on N literals 22
for DBCS literals 178
for national data items 178
for national literals 178
null branch 81
null-terminated strings
example 98
handling 399
manipulating 98
NUMBER compiler option
description 272
for debugging 311
numeric class test
checking for valid data 48
numeric comparison 85
numeric data
binary
USAGE BINARY 40
USAGE COMPUTATIONAL (COMP) 40
USAGE COMPUTATIONAL-4 (COMP-4) 40
USAGE COMPUTATIONAL-5 (COMP-5) 40
can compare algebraic values regardless of USAGE 192
comparing to national 191
converting
between fixed- and floating-point 46
precision 47
to national with MOVE 184
numeric data *(continued)*
defining 35
display floating-point (USAGE DISPLAY) 39
editing symbols 37
external decimal
USAGE DISPLAY 39
USAGE NATIONAL 39
external floating-point
USAGE DISPLAY 39
USAGE NATIONAL 40
internal floating-point
USAGE COMPUTATIONAL-1 (COMP-1) 41
USAGE COMPUTATIONAL-2 (COMP-2) 41
national decimal (USAGE NATIONAL) 39
national floating-point (USAGE NATIONAL) 40
packed-decimal
sign representation 47
USAGE COMPUTATIONAL-3 (COMP-3) 41
USAGE PACKED-DECIMAL 41
PICTURE clause 35, 37
storage formats 38
USAGE DISPLAY 35
USAGE NATIONAL 35
zoned decimal (USAGE DISPLAY)
format 39
sign representation 47
numeric intrinsic functions
example of
ANNUITY 52
CURRENT-DATE 52
INTEGER 102
INTEGER-OF-DATE 52
LENGTH 52, 108
LOG 53
MAX 52, 80, 107, 108
MEAN 53
MEDIAN 53, 80
MIN 102
NUMVAL 105
NUMVAL-C 52, 105
ORD 107
ORD-MAX 80
PRESENT-VALUE 52
RANGE 53, 80
REM 53
SQRT 53
SUM 80
integer, floating-point, mixed 50
nested 51
special registers as arguments 51
table elements as arguments 51
uses for 50
numeric literals, description 21
numeric-edited data
BLANK WHEN ZERO clause
coding with numeric data 178
example 37
defining 178
editing symbols 37
initializing
examples 26
using INITIALIZE 66
PICTURE clause 37
USAGE DISPLAY

numeric-edited data *(continued)*
USAGE DISPLAY *(continued)*
displaying 37
initializing, example of 26
USAGE NATIONAL
displaying 37
initializing, example of 26
NUMVAL intrinsic function
description 105
NUMVAL-C intrinsic function
description 105
example 52
NX delimiter for national literals 22

O

object code
generating 257
object references
size depends on ADDR 250
OBJECT-COMPUTER paragraph 5
objectives of millennium language extensions 423
OCCURS clause
ASCENDING|DESCENDING KEY phrase
example 78
needed for binary search 78
specify order of table elements 60
cannot use in a level-01 item 59
defining tables 59
for defining table elements 59
INDEXED BY phrase for creating indexes 64
nested for creating multidimensional tables 60
OCCURS DEPENDING ON (ODO) clause
complex 73
for creating variable-length tables 70
initializing ODO elements 72
ODO object 70
ODO subject 70
optimization 447
simple 70
OCCURS INDEXED BY clause, creating indexes with 64
ODO object 70
ODO subject 70
OMITTED parameters 457
OMITTED phrase for omitting arguments 397
ON SIZE ERROR
with windowed date fields 434
OPEN statement
file availability 132
file status key 166
opening files
optionally 132
overview 135
protecting against errors 132
using environment variables 218
operational force
description 149
environment variable for 219
suppressing 149
optimization
avoid ALTER statement 444
BINARY data items 446
consistent data 446
constant computations 444
optimization *(continued)*
constant data items 444
duplicate computations 445
effect of compiler options on 450
effect on parameter passing 397
effect on performance 443
factor expressions 444
index computations 448
indexing 447
MAXMEM 270
OCCURS DEPENDING ON 447
out-of-line PERFORM 444
packed-decimal data items 446
performance implications 447
structured programming 443
subscript computations 448
subscripting 447
table elements 447
top-down programming 444
unused data items 273
OPTIMIZE compiler option
description 273
effect on parameter passing 397
performance considerations 450, 451
optimizer
overview 450
options
85 COBOL Standard 248
specifying for linker 232
ORD intrinsic function, example 107
ORD-MAX intrinsic function
example table calculation 80
using 107
ORD-MIN intrinsic function 107
order of evaluation
arithmetic operators 50, 479
compiler options 248
out-of-line PERFORM 89
output
overview 121
to files 111
output procedure
coding 155
example 157
requires RETURN or RETURN INTO 155
restrictions 155
overflow condition
CALL 170
joining and splitting strings 163
UNSTRING 95

P

packed-decimal data item
date fields, potential problems 439
description 41
sign representation 47
synonym 38
using efficiently 41, 446
paragraph
definition 14
grouping 91
parameters
describing in called program 397

parameters *(continued)*
in main program 407
parse data item, definition 341
parsing XML documents
description 341
overview 339
UTF-8 349
white space 347
XML declaration 347
passing data between programs
addresses 400
arguments in calling program 397
BY CONTENT 395
BY REFERENCE 395
BY VALUE
overview 395
restrictions 397
EXTERNAL data 404
OMITTED arguments 397
parameters in called program 397
RETURN-CODE special register 403
PATH environment variable
description 221
path name
for copybook search 230, 242, 293
library text 217
multiple, specifying 217, 293
precedence 213
specifying for catalogs and help files 215
specifying for executable programs 221
PERFORM statement
coding loops 88
for a table
example using indexing 69
example using subscripting 68
for changing an index 64
inline 89
out-of-line 89
performed an explicit number of times 90
TEST AFTER 90
TEST BEFORE 90
THRU 91
TIMES 90
UNTIL 90
VARYING 90
VARYING WITH TEST AFTER 90
WITH TEST AFTER . . . UNTIL 90
WITH TEST BEFORE . . . UNTIL 90
performance
arithmetic evaluations 445
arithmetic expressions 446
CICS
dynamic calls 333
module caching 333
overview 443
coding for 443
coding tables 447
compiler option
ARITH 451
DYNAM 451
FLOAT 377
OPTIMIZE 450, 451
SSRANGE 451
TEST 452
performance *(continued)*
compiler option *(continued)*
TRUNC 284, 452
WSCLEAR 287
consistent data types 446
data usage 446
effect of compiler options on 450
exponentiations 447
module caching under CICS 333
OCCURS DEPENDING ON 447
optimizer
overview 450
order of WHEN phrases in EVALUATE 83
out-of-line PERFORM compared with inline 89
programming style 443
SFS (CICS) files
environment variable for 219
reducing frequency of saves 149
SFS files
client-side caching 148
table handling 448
table searching
binary compared with serial 76
improving serial search 77
tuning 443
variable subscript data format 63
worksheet 452
performing calculations
date and time services 458
period as scope terminator 16
PGMNAME compiler option 274
phrase, definition of 15
PICTURE clause
cannot use for internal floating point 36
determining symbol used 258
incompatible data 48
N for national data 177
national-edited data 178
numeric data 35
numeric-edited data 178
Z for zero suppression 37
picture strings
examples 462
overview 460
platform differences 378
pointer data item
description 33
incrementing addresses with 400
NULL value 400
passing addresses 400
processing chained lists 400
size depends on ADDR 250
used to process chained list 401
porting applications
CICS 331
differences between platforms 375
effect of separate sign 36
environment differences 378
file-status keys 378
language differences 375
mainframe to workstation
running mainframe applications on the workstation 375
multibyte 378

porting applications *(continued)*
multitasking 379
overview 375
SBCS 376
using COPY to isolate platform-specific code 375
workstation to mainframe
compiler options 379
file names 379
file suffixes 379
language features 379
nested programs 379
precedence
arithmetic operators 50, 479
compiler options 248
file-system determination 116
paths within environment variables 213
preinitializing the COBOL environment
example 417
for C/C++ program 387
initialization 415
overview 415
restriction under CICS 415
termination 416
PRESENT-VALUE intrinsic function 52
procedure and data-name cross-reference, description 308
PROCEDURE DIVISION
description 13
in subprograms 399
RETURNING
to return a value 13
using 403
statements
compiler-directing 16
conditional 15
delimited scope 15
imperative 15
terminology 13
USING
BY VALUE 399
to receive parameters 13, 397
procedure-pointer data item
definition 402
passing parameters to callable services 402
SET statement and 403
size depends on ADDR 250
using 403
process
terminating 388
PROCESS (CBL) statement
conflicting options in 248
description 291
specifying compiler options 224
processing
chained lists
example 401
overview 400
tables
example using indexing 69
example using subscripting 68
producing XML output 359
product support 631
profile file, setting environment variables in 213
program
attribute codes 317
program *(continued)*
decisions
EVALUATE statement 81
IF statement 81
loops 90
PERFORM statement 90
switches and flags 86
diagnostics 312
limitations 443
main 381
nesting level 313
statistics 313
structure 3
subprogram 381
PROGRAM COLLATING SEQUENCE clause
COLLSEQ interaction 256
does not affect national or DBCS operands 6
effect on alphanumeric comparisons 206
establishing collating sequence 6
no effect on DBCS comparisons 207
no effect on national comparisons 207
overridden by COLLATING SEQUENCE phrase 6
overrides default collating sequence 157
PROGRAM-ID paragraph
coding 3
COMMON attribute 4
INITIAL clause 4
program-names
cross-reference 319
handling of case 274
specifying 3
programs, running 235
putenv() to set environment variables 213

Q

QSAM file system 118
QSAM files
identifying 113
processing 116
using
overview 144
QUOTE compiler option 275

R

railroad track diagrams, how to read xx
RANGE intrinsic function
example statistics calculation 53
example table calculation 80
RAW file system, *see* QSAM file system 118
RCFs
sending xxi
read cache 148
READ NEXT statement 138
READ PREVIOUS statement 138
READ statement
AT END phrase 166
overview 137
reader comments
sending xxi
reading records from files 137
record

record *(continued)*
description 9
format 121
TRAP runtime option, effect of 300
RECORD CONTAINS clause
FILE SECTION entry 10
RECORDING MODE clause 10
recursive calls
and the LINKAGE SECTION 13
coding 393
identifying 4
REDEFINES clause, making a record into a table using 67
reentrant code 467
reference modification
example 100
expression checking with SSRANGE 281
generated XML documents 360
intrinsic functions 99
national data 99
out-of-range values 100
tables 63, 99
UTF-8 documents 193
reference modifier
arithmetic expression as 101
intrinsic function as, example 102
variables as 100
relate items to system-names 5
relation condition 85
relative files
file access mode 122
organization 122
RELEASE FROM statement
compared to RELEASE 154
example 154
RELEASE statement
compared to RELEASE FROM 154
with SORT 153, 154
REM intrinsic function 53
REPLACE statement
description 295
replacing
data items (INSPECT) 102
records in files 140
REPLACING phrase (INSPECT), example 103
REPOSITORY paragraph
coding 5
representation
data 48
sign 47
restrictions
CICS
overview 330
separate translator 335
Db2 files 118
Db2 precompiler 325
generation data groups (GDGs) 124
input/output procedures 155
SdU files 119, 124
SFS files 120, 124
subscripting 63
return code
compiler
depends on highest severity 228
effect of message customization 542
return code *(continued)*
compiler *(continued)*
overview 228
feedback code from date and time services 457
files
differences from host 378
example 169
overview 168
from Db2 SQL statements 327
normal termination 394
RETURN-CODE special register 393, 403, 457
unrecoverable exception 394
RETURN statement
required in output procedure 155
with INTO phrase 155
RETURN-CODE special register
normal termination 394
passing data between programs 403
passing return codes between programs 393
sharing return codes between programs 403
unrecoverable exception 394
value after call to date and time service 457
RETURNING phrase
CALL statement 403
PROCEDURE DIVISION header 403
REVERSE intrinsic function 105
reversing characters 105
ROUNDED phrase 478
rows in tables 61
RSD file system 119
RSD files
identifying 113
processing 117
run time
arguments 407
changing file-name 8
differences between platforms 376
messages 549
performance 443
run unit
terminating 388
running programs 235
runtime environment, preinitializing
example 417
overview 415
runtime messages
format 549
incomplete or abbreviated 235
list of 549
setting national language 214
runtime options
CHECK 297
CHECK(OFF)
performance considerations 451
DEBUG 298, 303
ERRCOUNT 298
FILESYS 298
for CICS 334
overview 297
specifying 218
TRAP
description 300
ON SIZE ERROR 164
UPSI 300

S

S-level error message 228, 306
scope of names
  global 384
  local 384
scope terminator
  aids in debugging 301
  explicit 15, 16
  implicit 16
scu (source conversion utility) 281
SD (sort description) entry, example 153
SdU file system
  description 119
  restrictions 119
SdU files
  error processing 164
  identifying 113
  processing 116
  restriction with GDGs 124
SEARCH ALL statement
  binary search 78
  example 78
  for changing an index 64
  table must be ordered 78
search rules for linker 234
SEARCH statement
  example 77
  for changing an index 64
  nesting to search more than one level of a table 77
  serial search 77
searching
  for name declarations 385
  tables
    binary search 78
    overview 76
    performance 76
    serial search 77
section
  declarative 17
  definition 14
  grouping 91
SELECT clause
  vary input-output file 8
SELECT OPTIONAL clause 132
sentence, definition of 14
separate CICS translator
  restrictions 335
separate sign
  portability 36
  printing 36
  required for signed national decimal 36
SEPOBJ compiler option 275
SEQUENCE compiler option 276
sequence numbers 280
sequential files
  file access mode 121
  organization 121
sequential search
  description 77
  example 77
serial numbers 280
serial search
  description 77
serial search *(continued)*
  example 77
SET condition-name TO TRUE statement
  example 89, 91
  switches and flags 87
SET statement
  for changing an index 64
  for changing index data items 64
  for procedure-pointer data items 403
  for setting a condition, example 88
  handling of program-name in 274
  using for debugging 303
setting
  index data items 64
  indexes 64
  linker options 232
  switches and flags 87
SFS (CICS) file system
  accessing SFS files
    example 146
    overview 145
  description 119
  fully qualified file names 115
  nonhierarchical 120
  restrictions 120
  system administration of 120
SFS (CICS) files
  accessing
    example 146
    non-CICS 334
    overview 145
  adding alternate indexes 147
  alternate index file name 115
  base file name 115
  COBOL coding example 146
  creating alternate index files 145
  creating SFS files
    environment variables for 145
    sfsadmin command for 146
  determining available data volumes 145
  error processing 164
  file names 115
  identifying
    server 115
  nontransactional access 120
  organization 119
  primary and secondary indexes 120
  processing 116
  restriction with GDGs 124
  specifying data volume for 145
SFS (CICS) server
  fully qualified name 115
  specifying server name 145
SFS (Encina) file system
  performance 147
SFS (Encina) files
  performance 147
sfsadmin command
  adding alternate indexes 147
  creating indexed files 146
  description 120
  determining available data volumes 145
shared libraries
  advantages and disadvantages 411

shared libraries *(continued)*
building
example 412
CICS considerations 332
definition 411
overview 411
purpose 411
resolution of references 412
setting directory path 218
subprograms and outermost programs 411
using 411
sharing
data
between separately compiled programs 404
coding the LINKAGE SECTION 398
from another program 12
in recursive or multithreaded programs 13
in separately compiled programs 13
overview 395
parameter-passing mechanisms 395
PROCEDURE DIVISION header 399
RETURN-CODE special register 403
scope of names 384
files
scope of names 384
using EXTERNAL clause 10, 404
using GLOBAL clause 10
shell script, compiling using 224
short listing, example 311
sign condition
testing sign of numeric operand 85
using in date processing 433
SIGN IS SEPARATE clause
portability 36
printing 36
required for signed national decimal data 36
sign representation 47
SIZE compiler option 277
sliding century window 424
sort
alternate collating sequence 157
completion code 158
criteria 156
description 151
determining success 158
diagnostic message 158
error number
list of possible values 159
obtaining with iwzGetSortErrno 159
files, describing 152
input procedures
coding 153
example 157
keys
default 206
defining 156
overview 151
output procedures
coding 155
example 157
process 151
restrictions on input/output procedures 155
terminating 162
work files
sort *(continued)*
work files *(continued)*
describing 152
TMP environment variable 215
SORT statement
ASCENDING|DESCENDING KEY phrase 156
COLLATING SEQUENCE phrase 6, 157
description 156
GIVING phrase 156
overview 151
USING phrase 156
SORT-RETURN special register
determining sort or merge success 158
terminating sort or merge 162
sorting
tables
overview 79
SOSI compiler option
description 277
multibyte portability 378
SOURCE and NUMBER output, example 313
source code
line number 313, 314, 317
listing, description 309
SOURCE compiler option
description 278
getting output 309
source conversion utility (scu) 281
SOURCE-COMPUTER paragraph 5
SPACE compiler option 279
special feature specification 5
special register
ADDRESS OF
size depends on ADDR 250
use in CALL statement 396
arguments in intrinsic functions 51
JNIEnvPtr
size depends on ADDR 250
LENGTH OF 109, 396
RETURN-CODE 393, 403
SORT-RETURN
determining sort or merge success 158
terminating sort or merge 162
using in XML parsing 342, 343
WHEN-COMPILED 110
XML-CODE 342, 343
XML-EVENT 342, 343
XML-NTEXT 342, 345
XML-TEXT 342, 345
SPECIAL-NAMES paragraph
coding 5
SPILL compiler option 279
splitting data items (UNSTRING) 95
SQL compiler option
coding 327
description 279
multioption interaction 248
SQL statements
coding
overview 326
overview 325
return codes 327
SQL INCLUDE 326
use for Db2 services 325

SQL statements *(continued)*
using binary data in 327
SQLCA
declare for programs that use SQL statements 326
return codes from Db2 327
SQRT intrinsic function 53
SRCFORMAT compiler option 280
SSRANGE compiler option
description 281
performance considerations 451
reference modification 100
turn off by using CHECK(OFF) runtime option 451
using 306
stack frames, collapsing 388
stanza
adding 226
attributes in configuration file 227
cob2 226
cob2_j 225
cob2_r 226
description 226
START statement 138
statement
compiler-directing 16
conditional 15
definition 14
delimited scope 15
explicit scope terminator 16
imperative 15
implicit scope terminator 16
statement cross-reference listing
description 309
statement nesting level 313
statements used in program 309
static linking
advantages 411
definition 385, 411
disadvantages 411
statistics intrinsic functions 53
status code, files
differences from host 378
example 169
overview 168
STDCALL interface convention
specified with CALLINT 252
STL file system
description 120
STL files
error processing 164
identifying 113
processing 116
STOP RUN statement
in main program 381
in subprogram 382
storage
allocation depends on ADDR 250
character data 190
for arguments 397
mapping 309
stride, table 448
STRING statement
example 94
overflow condition 163
using 93
strings
handling 93
null-terminated 399
structure, initializing using INITIALIZE 27
structured programming 444
subprogram
and main program 381
definition 395
description 381
linkage
common data items 397
PROCEDURE DIVISION in 399
subprograms
in a shared library 411
using 381
subscript
computations 448
definition 62
literal, example 62
range checking 306
variable, example 62
subscripting
definition 62
example 68
literal, example 62
reference modification 63
relative 63
restrictions 63
use data-name or literal 63
variable, example 62
substitution character 182
substrings
of table elements 99
reference modification of 99
SUM intrinsic function, example table calculation 80
support 631
switch-status condition 85
switches and flags
defining 86
description 86
resetting 87
setting switches off, example 88
setting switches on, example 88
testing multiple values, example 87
testing single values, example 86
SYMBOLIC CHARACTERS clause 7
symbolic constant 444
symbols used in MAP output 316
SYNCHRONIZED clause
alignment depends on ADDR 250
syntax diagrams, how to read xx
syntax errors
finding with NOCOMPILE compiler option 305
SYSADATA
output 249
SYSIN
supplying alternative modules 263
SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH environment variables 221
SYSLIB
supplying alternative modules 263
when not used 538
SYSLIB environment variable 217
SYSPRINT

SYSPRINT *(continued)*
supplying alternative modules 263
when not used 539
system date
under CICS 331
SYSTEM interface convention
specified with CALLINT 252
SYSTEM suboption of CALLINT compiler option 252
system-name 5

T

table
assigning values to 66
columns 59
compare to array 33
defining with OCCURS clause 59
definition 59
depth 61
description 33
dynamically loading 65
efficient coding 447, 448
elements 59
identical element specifications 447
index, definition 62
initializing
all occurrences of an element 68
at the group level 68
each item individually 67
using INITIALIZE 65
using PERFORM VARYING 90
loading values in 65
looping through 90
multidimensional 60
one-dimensional 59
processing with intrinsic functions 79
redefining a record as 67
reference modification 63
referencing substrings of elements 99
referencing with indexes, example 62
referencing with subscripts, example 62
referring to elements 62
rows 61
searching
binary 78
overview 76
performance 76
sequential 77
serial 77
sorting
overview 79
stride computation 448
subscript, definition 62
three-dimensional 61
two-dimensional 61
variable-length
creating 70
example of loading 72
initializing 72
preventing overlay in 75
TALLYING phrase (INSPECT), example 103
temporary work-file location
specifying with TMP 215
TERMINAL compiler option 282
terminal, sending messages to the 282
terminating XML parsing 352
terms used in MAP output 316
test
conditions 90
data 85
numeric operand 85
UPSI switch 85
TEST AFTER 90
TEST BEFORE 90
TEST compiler option
description 283
multioption interaction 248
performance considerations 452
text-name cross-reference, description 308
THREAD compiler option
and the LINKAGE SECTION 13
description 283
time information, formatting 214
time stamp
converting seconds to character time stamp (CEEDATM) 495
converting time stamp to seconds (CEESECS) 517
time-zone information
specifying with TZ 215
time, getting local (CEELOCT) 510
TITLE statement
controlling header on listing 4
TMP environment variable 215
top-down programming
constructs to avoid 444
Trademarks 589
transferring control
between COBOL programs 382
called program 381
calling program 381
main and subprograms 381
nested programs 383
transforming COBOL data to XML
example 365
overview 359
translating CICS into COBOL 329
TRAP runtime option
description 300
ON SIZE ERROR 164
TRUNC compiler option
description 283
performance considerations 452
tuning considerations, performance 450, 451
two-digit years
querying within 100-year range (CEEQCEN)
example 512
setting within 100-year range (CEESCEN)
example 513
TZ environment variable 215

U

U-level error message 228, 306
UNDATE intrinsic function
example 437
using 436
Unicode
description 176

Unicode *(continued)*
encoding and storage 190
processing data 175
UNSTRING statement
example 96
overflow condition 163
using 95
UPPER-CASE intrinsic function 104
uppercase, converting to 104
UPSI runtime option 300
UPSI switches, setting 300
USAGE clause
at the group level 21
incompatible data 48
INDEX phrase, creating index data items with 64
NATIONAL phrase at the group level 184
USE FOR DEBUGGING declaratives
DEBUG runtime option 298
overview 303
USE statement 295
user-defined condition 85
user-exit work area 535
user-exit work area extension 535
USING phrase
PROCEDURE DIVISION header 399
UTF-16
definition 176
encoding for national data 176
UTF-8
avoid INSPECT 349
avoid moves that truncate 349
avoid reference modification with XML documents 193
converting to or from national 193
definition 176
encoding and storage 190
encoding for ASCII invariant characters 176
example of generating an XML document 361
parsing XML documents 349
processing data items 193
XML document encoding 346
UTF16 compiler option 285
UTF16 data representation 285

V

VALUE clause
alphanumeric literal with national data, example 108
alphanumeric literal with national group, example 68
assigning table values
at the group level 68
to each item individually 67
to each occurrence of an element 68
assigning to a variable-length group 72
cannot use for external floating point 40
initializing internal floating-point literals 36
large literals with COMP-5 41
large, with TRUNC(BIN) 284
VALUE IS NULL 400
VALUE OF clause 10
variable
as reference modifier 100
definition 19
variable-length records
OCCURS DEPENDING ON (ODO) clause 447
variable-length table
assigning values to 72
creating 70
example 71
example of loading 72
preventing overlay in 75
variables, environment
accessing 213
assignment-name 218
CICS_CDS_ROOT
matching system file-name 115
CICS_SFS_DATA_VOLUME 219
CICS_SFS_INDEX_VOLUME 219
CICS_TK_SFS_SERVER 218
CICS_VSAM_AUTO_FLUSH 219
CICS_VSAM_CACHE 220
COBCPYEXT 217
COBLSTDIR 217
COBOPT 217
COBPATH
CICS dynamic calls 332
description 218
COBRTOPT 218
compiler 216
compiler and runtime 214
definition 213
EBCDIC_CODEPAGE 219
example of setting and accessing 221
LANG 214
LC_ALL 214
LC_COLLATE 214
LC_CTYPE 214
LC_MESSAGES 214
LC_TIME 214
library-name 217, 293
NLSPATH 215
PATH 221
precedence of paths 213
runtime 218
setting
in .profile 213
in command shell 213
in program 213
locale 201
overview 213
SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 221
SYSLIB 217
text-name 217, 293
TMP 215
TZ 215
variably located data item 74
variably located group 74
VBREF compiler option
description 286
output example 321
using 309

W

W-level message 228, 306
WHEN phrase
EVALUATE statement 83
SEARCH ALL statement 78

WHEN phrase *(continued)*
SEARCH statement 77
WHEN-COMPILED intrinsic function 110
WHEN-COMPILED special register 110
white space in XML documents 347
windowed date fields
contracting 439
WITH DEBUGGING MODE clause
for debugging lines 303
for debugging statements 303
WITH POINTER phrase
STRING 93
UNSTRING 95
wlist file 268
WORKING-STORAGE SECTION
comparison with LOCAL-STORAGE
example 11
overview 11
initializing 286
workstation and workstation COBOL
differences from host 467
WSCLEAR compiler option
overview 286
performance considerations 287

X

X delimiter for control characters in alphanumeric literals 22
XML declaration
generating 361
specifying encoding declaration 348
white space cannot precede 347
XML document
accessing 340
code pages supported 346
controlling the encoding of 364
document encoding declaration 347
EBCDIC special characters 348
encoding 346
enhancing
example of modifying data definitions 369
rationale and techniques 369
external code page 347
generating
example 365
overview 359
handling parsing exceptions 349
national language 346
parser 339
parsing
description 341
example 354
UTF-8 349
processing 339
specifying encoding if alphanumeric 348
UTF-8 encoding 346
white space 347
XML declaration 347
XML event
encoding conflicts 351
EXCEPTION 350
fatal errors 351
overview 343
processing 339, 342
XML event *(continued)*
processing procedure 341
XML exception codes
for generating 533
for parsing
handleable 523
not handleable 528
XML GENERATE statement
COUNT IN 365
NAME 362
NAMESPACE 361
NAMESPACE-PREFIX 361
NOT ON EXCEPTION 363
ON EXCEPTION 364
SUPPRESS 362
TYPE 363
WITH ATTRIBUTES 360
WITH ENCODING 364
XML-DECLARATION 361
XML generation
controlling type of XML data 363
counting generated characters 360
description 359
effect of CHAR(EBCDIC) 346
enhancing output
example of modifying data definitions 369
rationale and techniques 369
example 365
generating attributes 360
generating elements 360
handling errors 364
ignored data items 360
naming attributes or elements 362
no byte order mark 364
overview 359
suppressing generation of specified attributes or elements 362
using namespace prefixes 361
using namespaces 361
XML output
controlling the encoding of 364
enhancing
example of modifying data definitions 369
rationale and techniques 369
generating
example 365
overview 359
XML PARSE statement
NOT ON EXCEPTION 341
ON EXCEPTION 341
overview 339
using 341
XML parser
conformance 531
error handling 350
overview 339
XML parsing
control flow with processing procedure 343
description 341
effect of CHAR(EBCDIC) 346
fatal errors 351
handling encoding conflicts 351
handling exceptions 349
overview 339

XML parsing *(continued)*
special registers 342, 343
terminating 352
XML processing procedure
control flow with parser 343
error with EXIT PROGRAM or GOBACK 342
example
program for processing XML 354
handling encoding conflicts 352
handling parsing exceptions 349
restriction on XML PARSE 342
setting XML-CODE in 352
specifying 341
using special registers 342, 343
writing 342
XML-CODE special register
content 343
continuation after nonzero value 352
control flow between parser and processing procedure 343
description 342
exception codes for generating 533
exception codes for parsing
handleable 523
not handleable 528
exception codes for parsing with XMLPARSE(COMPAT)
encoding conflicts 350
fatal errors 351
setting to -1 343, 352
subtracting 100,000 from 351
subtracting 200,000 from 351
terminating parsing 352
using in generating 363
using in parsing 339
with code-page conflicts 351
with encoding conflicts 351
with generating exceptions 364
with parsing exceptions 350
XML-EVENT special register
content 343, 353
description 342
using 339, 342
with parsing exceptions 350
XML-NTEXT special register
content 345
description 342
using 339
with parsing exceptions 351
XML-TEXT special register
content 345, 353
description 342
encoding 342
using 339
with parsing exceptions 351
XREF compiler option
description 287
finding copybook files 308
finding data- and procedure-names 308
getting output 309
XREF output
COPY/BASIS cross-references 319
data-name cross-references 317
program-name cross-references 319

Y

year field expansion 426
year windowing
advantages 423
how to control 435
MLE approach 424
when not supported 429
year-first date fields 428
year-last date fields 428
year-only date fields 428
YEARWINDOW compiler option
description 288

Z

zero comparison (See sign condition) 433
zero suppression
example of BLANK WHEN ZERO clause 37
PICTURE symbol Z 37
zoned decimal data (USAGE DISPLAY)
effect of ZWB on comparison to alphanumeric 289
example 35
format 39
sign representation 47
ZWB compiler option 289

Product Number: 5737-L11

SC28-3118-00