

COBOL for Windows



Programming Guide

Version 7.5

COBOL for Windows



Programming Guide

Version 7.5

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 753.

First Edition (October 2008)

This edition applies to COBOL for Windows Version 7.5 in IBM Rational Developer for System z and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1996, 2008.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables xi

Preface xiii

About this document	xiii
Accessibility of this document	xiii
How this document will help you	xiii
Abbreviated terms	xiii
How to read syntax diagrams	xiv
How examples are shown	xv
Summary of changes	xv
Version 7 (December 2006)	xv
Version 6 (May 2005)	xvii
How to send your comments	xvii

Part 1. Coding your program 1

Chapter 1. Structuring your program . . . 5

Identifying a program	5
Identifying a program as recursive	6
Marking a program as callable by containing programs	6
Setting a program to an initial state	6
Changing the header of a source listing	6
Describing the computing environment	7
Example: FILE-CONTROL paragraph	7
Specifying the collating sequence	8
Defining symbolic characters	9
Defining a user-defined class	10
Identifying files to the operating system	10
Describing the data	11
Using data in input and output operations	12
Comparison of WORKING-STORAGE and LOCAL-STORAGE	13
Using data from another program	15
Processing the data	16
How logic is divided in the PROCEDURE DIVISION	17
Declaratives	20

Chapter 2. Using data 23

Using variables, structures, literals, and constants	23
Using variables	23
Using data items and group items	24
Using literals	25
Using constants	26
Using figurative constants	26
Assigning values to data items	27
Examples: initializing data items	28
Initializing a structure (INITIALIZE)	30
Assigning values to elementary data items (MOVE)	32
Assigning values to group data items (MOVE)	32
Assigning arithmetic results (MOVE or COMPUTE)	34

Assigning input from a screen or file (ACCEPT)	34
Displaying values on a screen or in a file (DISPLAY)	35
Using intrinsic functions (built-in functions)	36
Using tables (arrays) and pointers	37

Chapter 3. Working with numbers and arithmetic 39

Defining numeric data	39
Displaying numeric data	41
Controlling how numeric data is stored	42
Formats for numeric data	43
External decimal (DISPLAY and NATIONAL) items	43
External floating-point (DISPLAY and NATIONAL) items	44
Binary (COMP) items	44
Native binary (COMP-5) items	45
Byte reversal of binary data	45
Packed-decimal (COMP-3) items	46
Internal floating-point (COMP-1 and COMP-2) items	46
Examples: numeric data and internal representation	46
Data format conversions	49
Conversions and precision	49
Sign representation of zoned and packed-decimal data	50
Checking for incompatible data (numeric class test)	51
Performing arithmetic	52
Using COMPUTE and other arithmetic statements	52
Using arithmetic expressions	53
Using numeric intrinsic functions	53
Examples: numeric intrinsic functions	54
Fixed-point contrasted with floating-point arithmetic	56
Floating-point evaluations	57
Fixed-point evaluations	57
Arithmetic comparisons (relation conditions)	57
Examples: fixed-point and floating-point evaluations	58
Using currency signs	59
Example: multiple currency signs	60

Chapter 4. Handling tables 61

Defining a table (OCCURS)	61
Nesting tables	63
Example: subscripting	64
Example: indexing	64
Referring to an item in a table	64
Subscripting	65
Indexing	66
Putting values into a table	67
Loading a table dynamically	67
Initializing a table (INITIALIZE)	67

Assigning values when you define a table (VALUE)	69
Example: PERFORM and subscripting	70
Example: PERFORM and indexing.	71
Creating variable-length tables (DEPENDING ON)	72
Loading a variable-length table.	74
Assigning values to a variable-length table	75
Searching a table	75
Doing a serial search (SEARCH)	75
Doing a binary search (SEARCH ALL)	77
Processing table items using intrinsic functions	78
Example: processing tables using intrinsic functions	78

Chapter 5. Selecting and repeating program actions 81

Selecting program actions	81
Coding a choice of actions	81
Coding conditional expressions.	86
Repeating program actions	89
Choosing inline or out-of-line PERFORM	90
Coding a loop	91
Looping through a table	91
Executing multiple paragraphs or sections	92

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	99
Referring to substrings of data items	99
Reference modifiers	101
Example: arithmetic expressions as reference modifiers.	102
Example: intrinsic functions as reference modifiers.	102
Tallying and replacing data items (INSPECT).	103
Examples: INSPECT statement	103
Converting data items (intrinsic functions)	104
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)	105
Transforming to reverse order (REVERSE).	105
Converting to numbers (NUMVAL, NUMVAL-C)	105
Converting from one code page to another	107
Evaluating data items (intrinsic functions).	107
Evaluating single characters for collating sequence	107
Finding the largest or smallest data item	108
Finding the length of data items	110
Finding the date of compilation	111

Chapter 7. Processing files 113

Identifying files.	113
Identifying Btrieve files	114
Identifying STL files	114
Identifying RSD files	114
File system	114

STL file system	115
RSD file system.	118
Protecting against errors when opening files	118
Specifying a file organization and access mode	118
File organization and access mode	119
Setting up a field for file status	122
Describing the structure of a file in detail	123
Coding input and output statements for files.	123
Example: COBOL coding for files.	124
File position indicator	125
Opening a file	125
Reading records from a file.	128
Adding records to a file	129
Replacing records in a file	130
Deleting records from a file.	130
PROCEDURE DIVISION statements used to update files	131

Chapter 8. Sorting and merging files 133

Sort and merge process	133
Describing the sort or merge file	134
Describing the input to sorting or merging	134
Example: describing sort and input files for SORT	135
Coding the input procedure	136
Describing the output from sorting or merging	136
Coding the output procedure	137
Restrictions on input and output procedures	137
Requesting the sort or merge	138
Setting sort or merge criteria	139
Choosing alternate collating sequences	139
Example: sorting with input and output procedures	140
Determining whether the sort or merge was successful	141
Sort and merge error numbers.	141
Stopping a sort or merge operation prematurely	144

Chapter 9. Handling errors. 145

Handling errors in joining and splitting strings	145
Handling errors in arithmetic operations	146
Example: checking for division by zero.	146
Handling errors in input and output operations	146
Using the end-of-file condition (AT END)	147
Coding ERROR declaratives	148
Using file status keys.	148
Using file system status codes.	150
Handling errors when calling programs	152

Part 2. Enabling programs for international environments 153

Chapter 10. Processing data in an international environment 155

COBOL statements and national data	156
Intrinsic functions and national data.	158
Unicode and the encoding of language characters	159
Using national data (Unicode) in COBOL	160
Defining national data items	160

Using national literals	161
Using national-character figurative constants	162
Defining national numeric data items	163
National groups	163
Using national groups	164
Storage of national data	167
Converting to or from national (Unicode) representation	167
Converting alphanumeric, DBCS, and integer data to national data (MOVE)	168
Converting alphanumeric and DBCS data to national data (NATIONAL-OF)	169
Converting national data to alphanumeric data (DISPLAY-OF)	169
Overriding the default code page.	170
Example: converting to and from national data	170
Processing UTF-8 data	171
Processing Chinese GB 18030 data	171
Comparing national (UTF-16) data	172
Comparing two class national operands	172
Comparing class national and class numeric operands	173
Comparing national numeric and other numeric operands	174
Comparing national character-string and other character-string operands	174
Comparing national data and alphanumeric-group operands.	174
Coding for use of DBCS support	175
Declaring DBCS data	175
Using DBCS literals	176
Testing for valid DBCS characters	177
Processing alphanumeric data items that contain DBCS data	177

Chapter 11. Setting the locale 179

The active locale	179
Specifying the code page with a locale	180
Using environment variables to specify a locale	181
Determination of the locale from system settings	182
Types of messages for which translations are available	182
Locales and code pages that are supported	183
Controlling the collating sequence with a locale	185
Controlling the alphanumeric collating sequence with a locale.	185
Controlling the DBCS collating sequence with a locale	186
Controlling the national collating sequence with a locale	187
Intrinsic functions that depend on collating sequence	188
Accessing the active locale and code-page values	188
Example: get and convert a code-page ID	189

Part 3. Compiling, linking, running, and debugging your program . . . 191

Chapter 12. Compiling, linking, and running programs 193

Setting environment variables	193
Setting environment variables for COBOL for Windows.	194
Compiler environment variables	195
Linker environment variables	196
Runtime environment variables	196
Compiling programs	201
Compiling from the command line	201
Compiling using batch files or command files	203
Specifying compiler options with the PROCESS (CBL) statement	203
Correcting errors in your source program	203
Severity codes for compiler error messages	204
Generating a list of compiler error messages	204
cob2 options.	206
Options that apply to compiling	206
Options that apply to linking	207
Options that apply to both compiling and linking	208
Linking programs	208
File names and extensions supported by cob2	209
Specifying linker options	210
Linking through the compiler	210
Linking from the command line	211
Linker input and output files	212
File-name defaults.	213
Correcting errors in linking.	213
Linker return codes	214
Linker errors in program-names	214
Using NMAKE to update projects	215
Running NMAKE on the command line	215
Running NMAKE with a command file	216
Defining description files for NMAKE	216
Running programs	217
Redistributing COBOL for Windows DLLs	217

Chapter 13. Compiling, linking, and running OO applications 219

Compiling OO applications.	219
Preparing OO applications	220
Example: compiling and linking a COBOL class definition.	221
Running OO applications	221
Running OO applications that start with a main method	222
Running OO applications that start with a COBOL program	222

Chapter 14. Compiler options 225

Conflicting compiler options	226
ADATA	227
ARITH	228
BINARY	229
CALLINT	229
CHAR.	230
CICS	232
COLLSEQ	233
COMPILE	235
CURRENCY.	236
DATEPROC.	237

DIAGTRUNC	238
DYNAM	238
ENTRYINT	239
EXIT	240
Character string formats.	241
User-exit work area	241
Linkage conventions	242
Parameter list for exit modules	242
Using INEXIT	243
Using LIBEXIT	243
Using PRTEXT	244
Using ADEXIT	244
FLAG	245
FLAGSTD	246
FLOAT	247
LIB	248
LINECOUNT	249
LIST	249
LSTFILE	250
MAP	250
MDECK	251
NCOLLSEQ	252
NSYMBOL	253
NUMBER	253
OPTIMIZE	254
PGMNAME	255
PGMNAME(UPPER)	256
PGMNAME(MIXED)	256
PROBE	256
QUOTE/APOST	257
SEPOBJ	258
Batch compilation	258
SEQUENCE	259
SIZE	260
SOSI	260
SOURCE	261
SPACE	262
SQL	262
SSRANGE	263
TERMINAL	264
TEST	264
THREAD	265
TRUNC	266
TRUNC example 1	267
TRUNC example 2	268
VBREF	269
WSCLEAR	269
XREF	269
YEARWINDOW	271
ZWB	271

Chapter 15. Compiler-directing statements 273

Chapter 16. Linker options. 277

/?	278
/ALIGNADDR	278
/ALIGNFILE	279
/BASE	279
/CODE	280

/DATA	280
/DBGPACK, /NODBGPACK	281
/DEBUG, /NODEBUG	281
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH	281
/DLL	282
/ENTRY	282
/EXECUTABLE	283
/EXTDICTIONARY, /NOEXTDICTIONARY	283
/FIXED, /NOFIXED	284
/FORCE, /NOFORCE	284
/HEAP	285
/HELP	285
/INCLUDE	285
/INFORMATION, /NOINFORMATION	285
/LINENUMBERS, /NOLINENUMBERS	286
/LOGO, /NOLOGO	286
/MAP, /NOMAP	287
/OUT	287
/PMTYPE	288
/SECTION	288
/SEGMENTS	289
/STACK	290
/STUB	290
/SUBSYSTEM	291
/VERBOSE, /NOVERBOSE	291
/VERSION	292

Chapter 17. Runtime options. 293

CHECK	293
DEBUG	294
ERRCOUNT	294
FILESYS	294
TRAP	295
UPSI	296

Chapter 18. Debugging 297

Debugging with source language	297
Tracing program logic	297
Finding and handling input-output errors	298
Validating data	299
Finding uninitialized data	299
Generating information about procedures	299
Debugging using compiler options	301
Finding coding errors	302
Finding line sequence problems	302
Checking for valid ranges	302
Selecting the level of error to be diagnosed	303
Finding program entity definitions and references	305
Listing data items	306
Using the debugger	306
Getting listings	307
Example: short listing	308
Example: SOURCE and NUMBER output	310
Example: MAP output	310
Example: XREF output - data-name cross-references.	314
Example: VBREF compiler output	316
Debugging user exits	317

Debugging assembler routines.	318
---------------------------------------	-----

Part 4. Accessing databases . . . 319

Chapter 19. Programming for a DB2

environment	321
DB2 coprocessor	321
Coding SQL statements	322
Using SQL INCLUDE with the DB2 coprocessor	322
Using binary items in SQL statements	323
Determining the success of SQL statements .	323
Starting DB2 before compiling.	323
Compiling with the SQL option	323
Separating DB2 suboptions.	324
Using package and bind file-names	324

Chapter 20. Developing COBOL

programs for CICS	327
Coding COBOL programs to run under CICS . .	327
Getting the system date under CICS. . . .	328
Making dynamic calls under CICS	329
Compiling and running CICS programs	330
Integrated CICS translator	331
Debugging CICS programs.	332

Chapter 21. Open Database

Connectivity (ODBC)	333
Comparison of ODBC and embedded SQL . . .	333
Background	334
Installing and configuring software for ODBC	334
Coding ODBC calls from COBOL: overview . .	334
Using data types appropriate for ODBC . .	334
Passing pointers as arguments in ODBC calls	335
Accessing function return values in ODBC calls	337
Testing bits in ODBC calls	337
Using COBOL copybooks for ODBC APIs	338
Example: sample program using ODBC	
copybooks	339
Example: copybook for ODBC procedures. .	340
Example: copybook for ODBC data definitions	343
ODBC names truncated or abbreviated for	
COBOL	343
Compiling and linking programs that make ODBC	
calls	345
Understanding ODBC error messages	345

Part 5. Using XML and COBOL

together 347

Chapter 22. Processing XML input 349

XML parser in COBOL	349
Accessing XML documents	351
Parsing XML documents	351
The content of XML-EVENT	352
Example: processing XML events.	355
Writing procedures to process XML	357
Understanding the encoding of XML documents	364
Coded character sets for XML documents .	365

Specifying the code page	365
Handling exceptions that the XML parser finds .	366
How the XML parser handles errors.	367
Handling conflicts in code pages	370
Terminating XML parsing	371

Chapter 23. Producing XML output 373

Generating XML output	373
Example: generating XML	375
Enhancing XML output	379
Example: enhancing XML output.	380
Example: converting hyphens in element names	
to underscores	382
Controlling the encoding of generated XML output	383
Handling errors in generating XML output . .	384

Part 6. Developing object-oriented

programs 385

Chapter 24. Writing object-oriented

programs	387
Example: accounts.	388
Subclasses	389
Defining a class	390
CLASS-ID paragraph for defining a class . .	392
REPOSITORY paragraph for defining a class	392
WORKING-STORAGE SECTION for defining	
class instance data.	394
Example: defining a class	395
Defining a class instance method.	395
METHOD-ID paragraph for defining a class	
instance method	396
INPUT-OUTPUT SECTION for defining a class	
instance method	397
DATA DIVISION for defining a class instance	
method	397
PROCEDURE DIVISION for defining a class	
instance method	398
Overriding an instance method	399
Overloading an instance method	400
Coding attribute (get and set) methods. . .	401
Example: defining a method	402
Defining a client	403
REPOSITORY paragraph for defining a client	404
DATA DIVISION for defining a client	405
Comparing and setting object references . .	406
Invoking methods (INVOKE)	407
Creating and initializing instances of classes	412
Freeing instances of classes.	413
Example: defining a client	414
Defining a subclass	414
CLASS-ID paragraph for defining a subclass	415
REPOSITORY paragraph for defining a subclass	416
WORKING-STORAGE SECTION for defining	
subclass instance data	416
Defining a subclass instance method	417
Example: defining a subclass (with methods)	417
Defining a factory section	418

WORKING-STORAGE SECTION for defining	
factory data	419
Defining a factory method	420
Example: defining a factory (with methods)	422
Wrapping procedure-oriented COBOL programs	427
Structuring OO applications	428
Examples: COBOL applications that you can run	
using the java command	428

Chapter 25. Communicating with Java methods 431

Accessing JNI services	431
Handling Java exceptions	432
Managing local and global references	434
Java access controls	435
Sharing data with Java	435
Coding interoperable data types in COBOL and	
Java	436
Declaring arrays and strings for Java	437
Manipulating Java arrays	438
Manipulating Java strings	441

Part 7. Working with more complex applications 443

Chapter 26. Porting applications between platforms 445

Getting mainframe applications to compile	445
Getting mainframe applications to run: overview	447
Fixing differences caused by data	
representations	447
Fixing environment differences that affect	
portability	450
Fixing differences caused by language elements	450
Writing code to run on the mainframe	451
Writing applications that are portable between the	
Windows-based and AIX workstations	451

Chapter 27. Using subprograms . . . 453

Main programs, subprograms, and calls	453
Ending and reentering main programs or	
subprograms	454
Calling nested COBOL programs	454
Nested programs	455
Example: structure of nested programs	456
Scope of names.	457
Calling nonnested COBOL programs	458
CALL identifier and CALL literal.	458
Call interface conventions	459
CDECL	459
OPTLINK	460
SYSTEM	461
Calling between COBOL and C/C++ programs	462
Initializing environments	462
Passing data between COBOL and C/C++	462
Setting linkage conventions for COBOL and	
C/C++	463
Collapsing stack frames and terminating run	
units or processes	463

COBOL and C/C++ data types	464
Example: COBOL program calling C/C++ DLL	465
Making recursive calls	466

Chapter 28. Sharing data 467

Passing data.	467
Describing arguments in the calling program	469
Describing parameters in the called program	469
Testing for OMITTED arguments	470
Coding the LINKAGE SECTION	470
Coding the PROCEDURE DIVISION for passing	
arguments	471
Grouping data to be passed	471
Handling null-terminated strings	472
Using pointers to process a chained list	472
Using procedure and function pointers	475
Dealing with a Windows restriction	476
Coding multiple entry points	477
Passing return-code information	477
Understanding the RETURN-CODE special	
register	477
Using PROCEDURE DIVISION RETURNING	478
Specifying CALL . . . RETURNING	478
Sharing data by using the EXTERNAL clause.	478
Sharing files between programs (external files)	479
Example: using external files	479
Using command-line arguments	482
Example: command-line arguments	482

Chapter 29. Building dynamic link libraries 485

Static linking and dynamic linking	485
How the linker resolves references to DLLs	486
Creating DLLs	486
Example: DLL source file and related files.	487
Creating module definition files	489
Reserved words for module statements.	490
Summary of module statements	490
BASE	491
DESCRIPTION	491
EXPORTS	492
HEAPSIZE	493
LIBRARY.	493
NAME	494
STACKSIZE	494
STUB	494
VERSION	495

Chapter 30. Preparing COBOL programs for multithreading 497

Multithreading	497
Working with language elements with	
multithreading	498
Working with elements that have run-unit scope	499
Working with elements that have program	
invocation instance scope	499
Scope of COBOL language elements with	
multithreading	500
Choosing THREAD to support multithreading	500

Transferring control to multithreaded programs	501
Ending multithreaded programs	501
Handling COBOL limitations with multithreading	502
Example: using COBOL in a multithreaded environment.	502
Source code for thr cob.c	502
Source code for subd.cbl.	504
Source code for sube.cbl.	504

Chapter 31. Preinitializing the COBOL runtime environment 507

Initializing persistent COBOL environment	507
Terminating preinitialized COBOL environment	508
Example: preinitializing the COBOL environment	509

Chapter 32. Processing two-digit-year dates 513

Millennium language extensions (MLE)	514
Principles and objectives of these extensions	514
Resolving date-related logic problems	515
Using a century window	516
Using internal bridging	517
Moving to full field expansion.	518
Using year-first, year-only, and year-last date fields	520
Compatible dates	521
Example: comparing year-first date fields	522
Using other date formats	522
Example: isolating the year.	523
Manipulating literals as dates	523
Assumed century window	524
Treatment of nondates	525
Using sign conditions	526
Performing arithmetic on date fields.	526
Allowing for overflow from windowed date fields	527
Specifying the order of evaluation	528
Controlling date processing explicitly	528
Using DATEVAL	529
Using UNDATE	529
Example: DATEVAL	530
Example: UNDATE	530
Analyzing and avoiding date-related diagnostic messages	530
Avoiding problems in processing dates	532
Avoiding problems with packed-decimal fields	532
Moving from expanded to windowed date fields	533

Part 8. Improving performance and productivity 535

Chapter 33. Tuning your program. 537

Using an optimal programming style	537
Using structured programming	538
Factoring expressions.	538
Using symbolic constants	538
Grouping constant computations	539
Grouping duplicate computations	539
Choosing efficient data types	539
Choosing efficient computational data items	540

Using consistent data types.	540
Making arithmetic expressions efficient.	540
Making exponentiations efficient	541
Handling tables efficiently	541
Optimization of table references	542
Optimizing your code	544
Optimization	544
Choosing compiler features to enhance performance.	545
Performance-related compiler options	546
Evaluating performance	548

Chapter 34. Simplifying coding 549

Eliminating repetitive coding	549
Example: using the COPY statement.	550
Manipulating dates and times	551
Getting feedback from date and time callable services	551
Handling conditions from date and time callable services	552
Example: manipulating dates	552
Example: formatting dates for output	552
Feedback token.	553
Picture character terms and strings	555
Example: date-and-time picture strings	556
Century window	557

Part 9. Appendixes 559

Appendix A. Summary of differences with host COBOL. 561

Compiler options	561
Data representation	561
Binary data	561
Zoned decimal data	562
Packed-decimal data	562
Display floating-point data	562
National data	562
EBCDIC and ASCII data.	562
Code-page determination for data conversion	562
DBCS character strings	563
Environment variables	563
File specification	564
Interlanguage communication (ILC)	564
Input and output	564
Runtime options	564
Source code line size	565
Language elements	565

Appendix B. zSeries host data format considerations. 567

CICS access	567
Date and time callable services	567
Floating-point overflow exceptions	567
DB2	568
Object-oriented syntax for Java interoperability	568
MQ applications	568
File data	568
SORT	568

Appendix C. Intermediate results and arithmetic precision 569

Terminology used for intermediate results	570
Example: calculation of intermediate results	571
Fixed-point data and intermediate results	571
Addition, subtraction, multiplication, and division	571
Exponentiation	572
Example: exponentiation in fixed-point arithmetic	573
Truncated intermediate results	573
Binary data and intermediate results	574
Intrinsic functions evaluated in fixed-point arithmetic	574
Integer functions	574
Mixed functions	575
Floating-point data and intermediate results	576
Exponentiations evaluated in floating-point arithmetic	577
Intrinsic functions evaluated in floating-point arithmetic	577
Arithmetic expressions in nonarithmetic statements	577

Appendix D. Complex OCCURS DEPENDING ON 579

Example: complex ODO	579
How length is calculated	580
Setting values of ODO objects	580
Effects of change in ODO object value	580
Preventing index errors when changing ODO object value	581
Preventing overlay when adding elements to a variable table	581

Appendix E. Date and time callable services. 585

CEEBCLDY—convert date to COBOL integer format	587
CEEDATE—convert Lilian date to character format	590
CEEDATM—convert seconds to character timestamp	594
CEEDAYS—convert date to Lilian format	597
CEEDYWK—calculate day of week from Lilian date	601
CEEGMT—get current Greenwich Mean Time	603
CEEGMTO—get offset from Greenwich Mean Time to local time	605
CEEISEC—convert integers to seconds	607
CEELOCT—get current local date or time	609
CEEQCEN—query the century window	611
CEESCEN—set the century window	613
CEESECI—convert seconds to integers	615

CEESECS—convert timestamp to seconds	618
CEEUTC—get coordinated universal time	622
IGZEDT4—get current date.	622

Appendix F. XML reference material 625

XML PARSE exceptions that allow continuation	625
XML PARSE exceptions that do not allow continuation.	629
XML conformance.	632
XML GENERATE exceptions	634

Appendix G. JNI.cpy 635

Appendix H. COBOL SYSADATA file contents 641

Existing compiler options affecting the SYSADATA file	641
SYSADATA record types	642
Example: SYSADATA	643
SYSADATA record descriptions	644
Common header section.	645
Job identification record - X'0000'.	646
ADATA identification record - X'0001'.	647
Compilation unit start/end record - X'0002'.	647
Options record - X'0010'.	647
External symbol record - X'0020'.	657
Parse tree record - X'0024'.	657
Token record - X'0030'.	672
Source error record - X'0032'.	685
Source record - X'0038'.	686
COPY REPLACING record - X'0039'.	686
Symbol record - X'0042'.	687
Symbol cross-reference record - X'0044'.	700
Nested program record - X'0046'.	701
Library record - X'0060'.	702
Statistics record - X'0090'.	702
EVENTS record - X'0120'.	703

Appendix I. Runtime messages. 707

Notices 753
Trademarks 754

Glossary 755

List of resources 779

COBOL for Windows.	779
Related publications	779

Index 781

Tables

1. FILE SECTION entries	13	42. Using compiler options to get listings	307
2. Assignment to data items in a program	27	43. Terms and symbols used in MAP output	312
3. Ranges in value of COMP-5 data items	45	44. ODBC C types and corresponding COBOL declarations	334
4. Internal representation of native numeric items	47	45. ODBC copybooks	338
5. Internal representation of numeric items when BINARY(S390), CHAR(EBCDIC), and FLOAT(HEX) are in effect	48	46. ODBC names truncated or abbreviated for COBOL	343
6. Order of evaluation of arithmetic operators	53	47. Special registers used by the XML parser	357
7. Numeric intrinsic functions	54	48. Aliases for XML encoding declarations	366
8. Hexadecimal value of the euro sign	59	49. Encoding of generated XML output	383
9. STL file system return codes	116	50. Structure of class definitions	390
10. STL file system adapter open routine return codes	117	51. Structure of instance method definitions	395
11. File organization and access mode	119	52. Structure of COBOL clients	404
12. Valid COBOL statements for sequential files	126	53. Conformance of arguments in a COBOL client	409
13. Valid COBOL statements for line-sequential files	127	54. Conformance of the returned data item in a COBOL client	411
14. Valid COBOL statements for indexed and relative files	127	55. Structure of factory definitions	419
15. Statements used when writing records to a file	129	56. Structure of factory method definitions	420
16. PROCEDURE DIVISION statements used to update files	131	57. JNI services for local and global references	435
17. Sort and merge error numbers	142	58. Interoperable data types in COBOL and Java	436
18. COBOL statements and national data	156	59. Interoperable arrays and strings in COBOL and Java	437
19. Intrinsic functions and national character data.	158	60. Noninteroperable array types in COBOL and Java	438
20. National group items that are processed with group semantics	166	61. JNI array services	439
21. Encoding and size of alphanumeric, DBCS, and national data	167	62. Services that convert between jstring references and national data	441
22. Supported locales and code pages	183	63. Services that convert between jstring references and UTF-8 data	441
23. Intrinsic functions that depend on collating sequence	188	64. Language features that differ from mainframe COBOL	445
24. TZ environment parameter variables	200	65. ASCII characters contrasted with EBCDIC	447
25. Output from the cob2 command	202	66. ASCII comparisons contrasted with EBCDIC	448
26. Severity codes for compiler error messages	204	67. IEEE contrasted with hexadecimal	449
27. Default file-names assumed by the linker	213	68. Location of returned non-floating-point items with CDECL	459
28. Linker return codes	214	69. Location of returned non-floating-point items with SYSTEM	461
29. Commands for compiling and linking a class definition	220	70. COBOL and C/C++ data types	464
30. java command options for customizing the JVM	222	71. Methods for passing data in the CALL statement	468
31. Compiler options	225	72. Summary of linker module statements	490
32. Mutually exclusive compiler options	227	73. Scope of COBOL language elements with multithreading	500
33. Effect of comparand data type and collating sequence on comparisons	234	74. Advantages and disadvantages of Year 2000 solutions	516
34. Parameter list for exit modules	242	75. Performance-related compiler options	546
35. Linker options	277	76. Performance-tuning worksheet	548
36. Attributes for code sections	280	77. Picture character terms and strings	555
37. Attributes for data sections.	280	78. Japanese Eras	556
38. Attributes for named sections.	289	79. Examples of date-and-time picture strings	556
39. Default attributes for sections.	289	80. Language differences between Enterprise COBOL for z/OS and COBOL for Windows .	565
40. Runtime options	293	81. Maximum floating-point values	567
41. Severity levels of compiler messages	303	82. Date and time callable services	585

83. Date and time intrinsic functions	586	105. SYSADATA options record	648
84. CEECBLDY symbolic conditions	588	106. SYSADATA external symbol record	657
85. CEEDATE symbolic conditions	591	107. SYSADATA parse tree record	658
86. CEEDATM symbolic conditions	594	108. SYSADATA token record	672
87. CEEDAYS symbolic conditions	599	109. SYSADATA source error record	686
88. CEEDYWK symbolic conditions	601	110. SYSADATA source record	686
89. CEEGMT symbolic conditions	604	111. SYSADATA COPY REPLACING record	687
90. CEEGMTO symbolic conditions	606	112. SYSADATA symbol record	687
91. CEEISEC symbolic conditions	608	113. SYSADATA symbol cross-reference record	700
92. CEELOCT symbolic conditions	610	114. SYSADATA nested program record	701
93. CEEQCEN symbolic conditions	612	115. SYSADATA library record	702
94. CEEECEN symbolic conditions	613	116. SYSADATA statistics record	702
95. CEESECI symbolic conditions	616	117. SYSADATA EVENTS TIMESTAMP record layout	703
96. CEESECS symbolic conditions	619	118. SYSADATA EVENTS PROCESSOR record layout	703
97. XML PARSE exceptions that allow continuation	625	119. SYSADATA EVENTS FILE END record layout	704
98. XML PARSE exceptions that do not allow continuation	629	120. SYSADATA EVENTS PROGRAM record layout	704
99. XML GENERATE exceptions	634	121. SYSADATA EVENTS FILE ID record layout	704
100. SYSADATA record types	642	122. SYSADATA EVENTS ERROR record layout	705
101. SYSADATA common header section	645	123. Runtime messages	707
102. SYSADATA job identification record	646		
103. ADATA identification record	647		
104. SYSADATA compilation unit start/end record	647		

Preface

About this document

Welcome to IBM® COBOL for Windows®, IBM's COBOL compiler for Windows 2000 and Windows XP.

There are some differences between host and workstation COBOL. For details about language and system differences between COBOL for Windows and Enterprise COBOL for z/OS®, see Appendix A, “Summary of differences with host COBOL,” on page 561.

Accessibility of this document

The XHTML format of this document is accessible to visually impaired individuals who use a screen reader.

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.

How this document will help you

This document will help you write, compile, link-edit, and run your IBM COBOL for Windows programs. It will also help you define object-oriented classes and methods, invoke methods, and refer to objects in your programs.

This document 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 on COBOL syntax, see *COBOL for Windows Language Reference*.

This document also assumes familiarity with Windows. For information on Windows, see your operating system documentation.

Abbreviated terms

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

Term used	Long form
CICS®	IBM TXSeries®
COBOL for Windows	IBM COBOL for Windows
DB2®	Database 2™
RSD	Record sequential delimited file system
STL	standard language file system

In addition to these abbreviated terms, the term “Standard COBOL 85” is used in this document 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 - COBOL - 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

The ISO standards are identical to the American National Standards.

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

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.

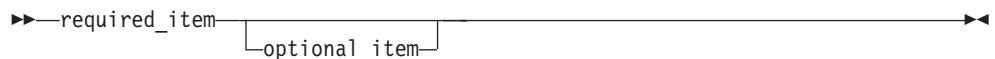
The **—><** symbol indicates the end of a syntax diagram.

Diagrams of syntactical units other than complete statements start with the **>—** symbol and end with the **—>** symbol.

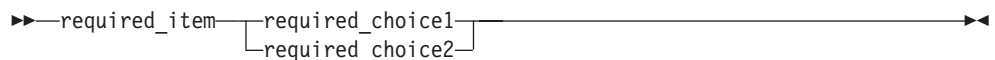
- Required items appear on the horizontal line (the main path).



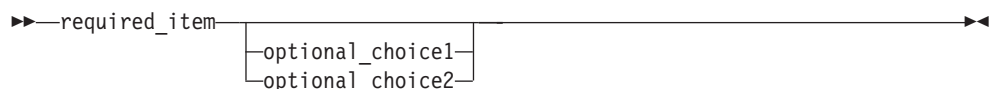
- Optional items appear below the main path.



- 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.

- Numeric-edited items (“Displaying numeric data” on page 41)
- National-edited items (“Using national data (Unicode) in COBOL” on page 160)
- Group (*national group*) items, supported by the GROUP-USAGE NATIONAL clause (“Using national groups” on page 164)
- Many COBOL language elements support the new kinds of UTF-16 data, or newly support the processing of national data:
 - Numeric data with USAGE NATIONAL (national decimal and national floating point) can be used in arithmetic operations and in any language constructs that support numeric operands (“Formats for numeric data” on page 43).
 - Edited data with USAGE NATIONAL is supported in the same language constructs as any existing edited type, including editing and de-editing operations associated with moves (“Displaying numeric data” on page 41 and “Using national data (Unicode) in COBOL” on page 160).
 - Group items that contain all national data can be defined with the GROUP-USAGE NATIONAL clause, which results in the group behaving as an elementary item in most language constructs. This support facilitates use of national groups in statements such as STRING, UNSTRING, and INSPECT (“National groups” on page 163).
 - The XML GENERATE statement supports national groups as receiving data items, and national-edited, numeric-edited of USAGE NATIONAL, national decimal, national floating-point, and national group items as sending data items (“Generating XML output” on page 373).
 - The NUMVAL and NUMVAL-C intrinsic functions can take a national literal or national data item as an argument (“Converting to numbers (NUMVAL, NUMVAL-C)” on page 105).

Using these new national data capabilities, it is now practical to develop COBOL programs that exclusively use Unicode for all application data.

- Support for compilation of programs and copybooks that contain CICS statements, without the need for a separate translation step, has been added (“Integrated CICS translator” on page 331).
- A new compiler option, CICS, enables integrated CICS translation and specification of CICS options (“CICS” on page 232).
- A new callable service, iwzGetSortErrno, makes it possible to obtain the sort or merge error number after each sort or merge operation (“Determining whether the sort or merge was successful” on page 141).
- The REDEFINES clause has been enhanced such that for data items that are not level 01, the subject of the entry can be larger than the data item being redefined.
- The literal in a VALUE clause for a data item of class national can be alphanumeric (“Initializing a table at the group level” on page 70).

These terminology changes were also made in this release:

- The term *alphanumeric group* is introduced to refer specifically to groups other than national groups.
- The term *group* means both alphanumeric groups and national groups except when used in a context that obviously refers to only an alphanumeric group or only a national group.
- The term *external decimal* refers to both zoned decimal items and national decimal items.

- The term *display floating point* is introduced to refer to an external floating-point item that has USAGE DISPLAY.
- The term *external floating point* refers to both display floating-point items and national floating-point items.

Version 6 (May 2005)

- Several limits on COBOL data-item size have been significantly raised (“Describing the data” on page 11). For example:
 - The maximum data-item size has been raised to 2,147,483,646 bytes.
 - The maximum PICTURE symbol replication has been raised to 2,147,483,646.
 - The maximum OCCURS integer has been raised to 2,147,483,646.

This support facilitates programming with large amounts of data, for example:

- DB2/COBOL applications that use DB2 BLOB and CLOB data types
- COBOL XML applications that parse or generate large XML documents

For full details about changed compiler limits, see Compiler limits (*COBOL for Windows Language Reference*).

- A compiler option MDECK, which specifies that output from library processing is written to a file, has been added (“MDECK” on page 251).
- A cob2 option, -s, which affects the amount of stack space that the linker allocates, has been added (“cob2 options” on page 206).
- CICS Transaction Server is no longer supported.

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this document or any other COBOL for Windows documentation, contact us in one of these ways:

- Fill out the Readers’ Comments Form at the back of this document, and return it by mail or give it to an IBM representative. If there is no form at the back of this document, address your comments to:

IBM Corporation
Reader Comments
DTX/E269
555 Bailey Avenue
San Jose, CA 95141-1003
USA

- Use the Online Readers’ Comments Form at www.ibm.com/software/awdtools/rcf/.
- Send your comments to the following e-mail address: comments@us.ibm.com.

Be sure to include the name of the document, the publication number of the document, the version of COBOL for Windows, 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.

Part 1. Coding your program

Chapter 1. Structuring your program	5
Identifying a program	5
Identifying a program as recursive	6
Marking a program as callable by containing programs	6
Setting a program to an initial state.	6
Changing the header of a source listing	6
Describing the computing environment	7
Example: FILE-CONTROL paragraph	7
Specifying the collating sequence	8
Example: specifying the collating sequence	9
Defining symbolic characters	9
Defining a user-defined class	10
Identifying files to the operating system.	10
Varying the input or output file at run time	10
Describing the data.	11
Using data in input and output operations	12
FILE SECTION entries.	12
Comparison of WORKING-STORAGE and LOCAL-STORAGE	13
Example: storage sections.	14
Using data from another program	15
Sharing data in separately compiled programs	15
Sharing data in nested programs	15
Sharing data in recursive or multithreaded programs	16
Processing the data.	16
How logic is divided in the PROCEDURE DIVISION	17
Imperative statements	18
Conditional statements	18
Compiler-directing statements	19
Scope terminators	19
Declaratives	20
Chapter 2. Using data	23
Using variables, structures, literals, and constants	23
Using variables	23
Using data items and group items.	24
Using literals	25
Using constants	26
Using figurative constants	26
Assigning values to data items	27
Examples: initializing data items	28
Initializing a structure (INITIALIZE)	30
Assigning values to elementary data items (MOVE)	32
Assigning values to group data items (MOVE)	32
Assigning arithmetic results (MOVE or COMPUTE)	34
Assigning input from a screen or file (ACCEPT)	34
Displaying values on a screen or in a file (DISPLAY)	35
Using intrinsic functions (built-in functions)	36
Using tables (arrays) and pointers	37
Defining numeric data.	39
Displaying numeric data	41
Controlling how numeric data is stored	42
Formats for numeric data.	43
External decimal (DISPLAY and NATIONAL) items	43
External floating-point (DISPLAY and NATIONAL) items	44
Binary (COMP) items	44
Native binary (COMP-5) items	45
Byte reversal of binary data	45
Packed-decimal (COMP-3) items	46
Internal floating-point (COMP-1 and COMP-2) items	46
Examples: numeric data and internal representation	46
Data format conversions	49
Conversions and precision	49
Conversions that lose precision.	50
Conversions that preserve precision	50
Conversions that result in rounding	50
Sign representation of zoned and packed-decimal data	50
Checking for incompatible data (numeric class test)	51
Performing arithmetic	52
Using COMPUTE and other arithmetic statements.	52
Using arithmetic expressions	53
Using numeric intrinsic functions	53
Examples: numeric intrinsic functions	54
General number handling	55
Date and time	55
Finance.	55
Mathematics	56
Statistics	56
Fixed-point contrasted with floating-point arithmetic	56
Floating-point evaluations	57
Fixed-point evaluations	57
Arithmetic comparisons (relation conditions)	57
Examples: fixed-point and floating-point evaluations	58
Using currency signs	59
Example: multiple currency signs	60
Chapter 4. Handling tables.	61
Defining a table (OCCURS)	61
Nesting tables	63
Example: subscripting	64
Example: indexing	64
Referring to an item in a table	64
Subscripting	65
Indexing	66
Putting values into a table	67
Loading a table dynamically.	67
Initializing a table (INITIALIZE)	67
Chapter 3. Working with numbers and arithmetic	39

Assigning values when you define a table (VALUE)	69
Initializing each table item individually	69
Initializing a table at the group level	70
Initializing all occurrences of a given table element.	70
Example: PERFORM and subscripting	70
Example: PERFORM and indexing.	71
Creating variable-length tables (DEPENDING ON)	72
Loading a variable-length table.	74
Assigning values to a variable-length table	75
Searching a table	75
Doing a serial search (SEARCH)	75
Example: serial search	76
Doing a binary search (SEARCH ALL)	77
Example: binary search	77
Processing table items using intrinsic functions	78
Example: processing tables using intrinsic functions	78
Chapter 5. Selecting and repeating program actions	81
Selecting program actions	81
Coding a choice of actions	81
Using nested IF statements	82
Using the EVALUATE statement	83
Coding conditional expressions.	86
Switches and flags	87
Defining switches and flags	87
Example: switches	87
Example: flags	88
Resetting switches and flags.	88
Example: set switch on	88
Example: set switch off	89
Repeating program actions	89
Choosing inline or out-of-line PERFORM	90
Example: inline PERFORM statement.	90
Coding a loop	91
Looping through a table	91
Executing multiple paragraphs or sections	92
Chapter 6. Handling strings	93
Joining data items (STRING)	93
Example: STRING statement.	94
STRING results	95
Splitting data items (UNSTRING)	95
Example: UNSTRING statement	96
UNSTRING results	97
Manipulating null-terminated strings.	98
Example: null-terminated strings	99
Referring to substrings of data items	99
Reference modifiers	101
Example: arithmetic expressions as reference modifiers.	102
Example: intrinsic functions as reference modifiers.	102
Tallying and replacing data items (INSPECT).	103
Examples: INSPECT statement	103
Converting data items (intrinsic functions)	104
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)	105

Transforming to reverse order (REVERSE)	105
Converting to numbers (NUMVAL, NUMVAL-C)	105
Converting from one code page to another	107
Evaluating data items (intrinsic functions).	107
Evaluating single characters for collating sequence	107
Finding the largest or smallest data item	108
Returning variable-length results with alphanumeric or national functions	109
Finding the length of data items	110
Finding the date of compilation	111
Chapter 7. Processing files	113
Identifying files.	113
Identifying Btrieve files	114
Identifying STL files	114
Identifying RSD files	114
File system	114
STL file system	115
STL file system return codes	116
RSD file system.	118
Protecting against errors when opening files	118
Specifying a file organization and access mode	118
File organization and access mode	119
Sequential file organization.	119
Line-sequential file organization	120
Indexed file organization	120
Relative file organization	121
Sequential access	121
Random access.	121
Dynamic access.	121
File input-output limitations	122
Setting up a field for file status	122
Describing the structure of a file in detail	123
Coding input and output statements for files.	123
Example: COBOL coding for files.	124
File position indicator	125
Opening a file	125
Valid COBOL statements for sequential files	126
Valid COBOL statements for line-sequential files	126
Valid COBOL statements for indexed and relative files	127
Reading records from a file.	128
Statements used when writing records to a file	129
Adding records to a file	129
Replacing records in a file	130
Deleting records from a file.	130
PROCEDURE DIVISION statements used to update files	131
Chapter 8. Sorting and merging files.	133
Sort and merge process	133
Describing the sort or merge file	134
Describing the input to sorting or merging	134
Example: describing sort and input files for SORT	135
Coding the input procedure	136
Describing the output from sorting or merging	136

Coding the output procedure	137
Restrictions on input and output procedures . . .	137
Requesting the sort or merge	138
Setting sort or merge criteria	139
Choosing alternate collating sequences . . .	139
Example: sorting with input and output procedures	140
Determining whether the sort or merge was successful	141
Sort and merge error numbers.	141
Stopping a sort or merge operation prematurely	144
 Chapter 9. Handling errors	 145
Handling errors in joining and splitting strings . .	145
Handling errors in arithmetic operations . . .	146
Example: checking for division by zero. . .	146
Handling errors in input and output operations	146
Using the end-of-file condition (AT END) . . .	147
Coding ERROR declaratives	148
Using file status keys.	148
Example: file status key	149
Using file system status codes.	150
Example: checking file system status codes	150
Example: FILE STATUS and INVALID KEY	151
Handling errors when calling programs	152

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.

To define a COBOL class or method, you need to define some divisions differently than you do for a program.

RELATED TASKS

“Identifying a program”

“Describing the computing environment” on page 7

“Describing the data” on page 11

“Processing the data” on page 16

“Defining a class” on page 390

“Defining a class instance method” on page 395

“Structuring OO applications” on page 428

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. 09/30/2008.

Date-Compiled. 09/30/2008.

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: When a program-name is case sensitive, avoid mismatches with the name 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 6

“Identifying a program as recursive” on page 6

“Marking a program as callable by containing programs” on page 6

“Setting a program to an initial state” on page 6

RELATED REFERENCES

Compiler limits (*COBOL for Windows Language Reference*)

Conventions for program-names (*COBOL for Windows Language Reference*)

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 or multithreaded programs” on page 16

“Making recursive calls” on page 466

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 455

Setting a program to an initial state

Use the **INITIAL** attribute 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 in 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 454

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 5724-T07 IBM COBOL for Windows 7.5.0   Date 09/30/2008 Time 15:05:19   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 Windows 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, STL file system) that is associated with a file. You can also do this at run time.
- Provide information about how the files are accessed.

“Example: FILE-CONTROL paragraph”

RELATED TASKS

“Specifying the collating sequence” on page 8

“Defining symbolic characters” on page 9

“Defining a user-defined class” on page 10

“Identifying files to the operating system” on page 10

RELATED REFERENCES

Sections and paragraphs (*COBOL for Windows 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 may be the system file-name or the name of the environment variable whose value (at run time) is used as the system file-name with optional directory and path names.
- (3) The ORGANIZATION clause describes the file’s organization. If omitted, the default is ORGANIZATION IS SEQUENTIAL.

- (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 7

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:

- Nonnumeric 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 9

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, 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 139

“Controlling the collating sequence with a locale” on page 185

Chapter 11, “Setting the locale,” on page 179

“Comparing national (UTF-16) data” on page 172

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 8

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 179

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

The ASSIGN clause associates the name of a file within a program with the name of the file as it is known to the operating system.

You can use either 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 (with 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 a Btrieve file, and you use F1 as the name of the file in your program, the ASSIGN clause would be:

```
SELECT F1 ASSIGN TO BTR-MYFILE
```

If MYFILE is not an environment variable, then the code above treats MYFILE as a system file-name. If MYFILE is an environment variable, then the value of the environment variable is used. For example, if the environment variable MYFILE is set as MYFILE=STL-YOURFILE, the system file-name becomes YOURFILE at run time, and the file is treated as an STL file, overriding the file-system ID used in the ASSIGN clause.

Note though that if you enclose an assignment-name in quotation marks or single quotation marks (for example, "BTR-MYFILE"), the value of the environment variable, if any, is ignored. The assignment-name is treated as a literal.

RELATED TASKS

"Varying the input or output file at run time"

"Identifying files" on page 113

RELATED REFERENCES

"FILESYS" on page 294

ASSIGN clause (*COBOL for Windows 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 the file with a different actual file at run time.

Changing a file-name in 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 SET 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 drive and path specifications).

“Example: using different input files”

Example: using different input files: This example shows that you 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 MASTER ASSIGN TO MASTERA
```

Suppose you want the program to access either the checking or savings file using the file called MASTER in the program. To do so, set the MASTERA 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 d:\accounts directory:

```
set MASTERA=d:\accounts\checking
set MASTERA=d:\accounts\savings
```

You can use the same program to access either the checking or savings file as the file called MASTER in 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 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 Windows compiler limits the maximum size of DATA DIVISION elements. The compiler limit on the amount of data in the LOCAL-STORAGE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION combined is 2 GB. However, because of compiler-generated temporary data that is required for operations, the practical limit on user data is approximately 1 GB. See also the related reference below for information about the cob2 -s option.

RELATED CONCEPTS

“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 13

RELATED TASKS

“Using data in input and output operations”
“Using data from another program” on page 15

RELATED REFERENCES

“cob2 options” on page 206
Compiler limits (*COBOL for Windows 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 455

RELATED TASKS

“Sharing files between programs (external files)” on page 479

RELATED REFERENCES

“FILE SECTION entries”

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 <i>file-name</i> to be referred to in PROCEDURE DIVISION input-output statements (OPEN, CLOSE, READ, START, and DELETE). Must match <i>file-name</i> in the SELECT clause. <i>file-name</i> is associated with the system file through the <i>assignment-name</i> .
RECORD CONTAINS <i>n</i>	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 <i>n</i> TO <i>m</i>	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 Windows 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 or invocation of a method, and is freed on return from the program or method. If you specify a VALUE clause for a LOCAL-STORAGE item, the item is initialized to that value on each call or invocation. If a VALUE clause is not specified, the initial value of the item is undefined.

“Example: storage sections”

RELATED TASKS

“Ending and reentering main programs or subprograms” on page 454

Chapter 30, “Preparing COBOL programs for multithreading,” on page 497

“WORKING-STORAGE SECTION for defining class instance data” on page 394

RELATED REFERENCES

Working-storage section (*COBOL for Windows Language Reference*)

Local-storage section (*COBOL for Windows Language Reference*)

Example: storage sections

The following is an example of 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

Recursive calls	Value for num in LOCAL-STORAGE	Value for numb in WORKING-STORAGE	Value for fact in WORKING-STORAGE
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 13

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”

“Sharing data in nested programs”

“Sharing data in recursive or multithreaded programs” on page 16

“Passing data” on page 467

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, use a CALL . . . USING or INVOKE . . . USING statement to pass the data.

RELATED TASKS

“Passing data” on page 467

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 455

Sharing data in recursive or multithreaded programs

If your program has the `RECURSIVE` attribute or is compiled with the `THREAD` compiler option, 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 or is compiled with the `THREAD` compiler option, 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 CONCEPTS

“Multithreading” on page 497

RELATED TASKS

“Making recursive calls” on page 466

RELATED REFERENCES

“`THREAD`” on page 265

`SET` statement (*COBOL for Windows 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”

RELATED TASKS

“Eliminating repetitive coding” on page 549

RELATED REFERENCES

The procedure division header (*COBOL for Windows Language Reference*)

The USING phrase (*COBOL for Windows Language Reference*)

CALL statement (*COBOL for Windows 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 verb. 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 19

“Scope terminators” on page 19

“Imperative statements” on page 18

“Conditional statements” on page 18

“Declaratives” on page 20

RELATED REFERENCES

PROCEDURE DIVISION structure (*COBOL for Windows Language Reference*)

Imperative statements

An imperative statement (such as ADD, MOVE, INVOKE, 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”

“Scope terminators” on page 19

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 18

"Scope terminators"

RELATED TASKS

"Selecting program actions" on page 81

RELATED REFERENCES

Conditional statements (*COBOL for Windows 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 15, "Compiler-directing statements," on page 273

Compiler-directing statements (*COBOL for Windows Language Reference*)

Scope terminators

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

Explicit scope terminators end a verb without ending a sentence. They consist of END followed by a hyphen and the name of the verb 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 18

"Imperative statements" on page 18

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 298

RELATED REFERENCES

Declaratives (*COBOL for Windows 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”

“Assigning values to data items” on page 27

“Displaying values on a screen or in a file (DISPLAY)” on page 35

“Using intrinsic functions (built-in functions)” on page 36

“Using tables (arrays) and pointers” on page 37

Chapter 10, “Processing data in an international environment,” on page 155

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, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE. You place all data definitions in the DATA DIVISION of your program.

RELATED TASKS

“Using variables”

“Using data items and group items” on page 24

“Using literals” on page 25

“Using constants” on page 26

“Using figurative constants” on page 26

RELATED REFERENCES

Classes and categories of data (*COBOL for Windows 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  
    .  
    .  
    .
```

You could instead declare 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 159

RELATED TASKS

“Using national data (Unicode) in COBOL” on page 160

RELATED REFERENCES

“NSYMBOL” on page 253

“Storage of national data” on page 167

PICTURE clause (*COBOL for Windows 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 159
 “National groups” on page 163

RELATED TASKS

“Using national data (Unicode) in COBOL” on page 160
 “Using national groups” on page 164

RELATED REFERENCES

“FILE SECTION entries” on page 12
 “Storage of national data” on page 167
 Classes and categories of group items (*COBOL for Windows Language Reference*)
 PICTURE clause (*COBOL for Windows Language Reference*)
 MOVE statement (*COBOL for Windows Language Reference*)
 USAGE clause (*COBOL for Windows 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 declare 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 159

RELATED TASKS

"Using national literals" on page 161

"Using DBCS literals" on page 176

RELATED REFERENCES

"NSYMBOL" on page 253

Literals (*COBOL for Windows 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 160

"Coding for use of DBCS support" on page 175

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 162

“Coding for use of DBCS support” on page 175

RELATED REFERENCES

Figurative constants (*COBOL for Windows 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: <ul style="list-style-type: none">• 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: <ul style="list-style-type: none">• 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.• Associate an OBJECT REFERENCE data item with an object instance.	Use the SET statement.

“Examples: initializing data items” on page 28

RELATED TASKS

“Initializing a structure (INITIALIZE)” on page 30

“Assigning values to elementary data items (MOVE)” on page 32

“Assigning values to group data items (MOVE)” on page 32

“Assigning input from a screen or file (ACCEPT)” on page 34

“Joining data items (STRING)” on page 93

“Splitting data items (UNSTRING)” on page 95

“Assigning arithmetic results (MOVE or COMPUTE)” on page 34

“Tallying and replacing data items (INSPECT)” on page 103

Chapter 10, “Processing data in an international environment,” on page 155

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*

<i>identifier-1</i> PICTURE	<i>identifier-1</i> before	<i>identifier-1</i> after
9(5)	12345	00000
X(5)	AB123	bbbb ¹
N(3)	410042003100 ²	200020002000 ³
99XX9	12AB3	bbbb ¹
XXBX/XX	ABbC/DE	bbbb/bb ¹
**99.9CR	1234.5CR	**00.0bb ¹
A(5)	ABCDE	bbbb 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' in UTF-16LE encoding. The example assumes that *identifier-1* has Usage National.
3. Hexadecimal representation of the national (UTF-16) characters ' ' (three blank spaces) in UTF-16LE encoding. 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	bbbbbb ¹	bbbbABCD ¹

1. The symbol *b* represents a blank space.

Initializing an alphanumeric-edited data item:

```
01 ALPHANUM-EDIT-1 PIC XXBX/XXX VALUE "ABbC/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	ABbC/DEF ¹	M/bb/bbb ¹
1. The symbol <i>b</i> 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
```

NATIONAL-3	NATIONAL-1 before	NATIONAL-1 after
43004400 ¹	41004200 ²	43004400 ¹
1. Hexadecimal representation of the national characters 'CD' in UTF-16LE encoding 2. Hexadecimal representation of the national characters 'AB' in UTF-16LE encoding		

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' in UTF-16LE encoding 2. Hexadecimal representation of the national characters '123' in UTF-16LE encoding 3. Hexadecimal representation of the national characters '045' in UTF-16LE encoding		

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' in UTF-16LE encoding 2. Hexadecimal representation of the national characters '012' in UTF-16LE encoding		

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

```

01 NUM-EDIT-DISP-1 PIC $$$V VALUE "$127".
01 NUM-DISP-3      PIC 999V VALUE 12.
. . .
    INITIALIZE NUM-EDIT-DISP-1
      REPLACING NUMERIC 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 $$$V NATIONAL VALUE N"$127".
01 NUM-NATL-3      PIC 999V NATIONAL VALUE 12.
. . .
    INITIALIZE NUM-EDIT-NATL-1
      REPLACING NUMERIC 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' in UTF-16LE encoding 2. Hexadecimal representation of the national characters '\$127' in UTF-16LE encoding 3. Hexadecimal representation of the national characters '\$ 12' in UTF-16LE encoding		

RELATED TASKS

“Initializing a structure (INITIALIZE)”
 “Initializing a table (INITIALIZE)” on page 67
 “Defining numeric data” on page 39

RELATED REFERENCES

“NSYMBOL” on page 253

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	R0013830002400000000000000000	b000000000000000000000000000 ¹
2	R0013900004800000000000000000	b000000000000000000000000000 ¹
3	S0014100001200000000000000000	b000000000000000000000000000 ¹
4	C001383000000000425000000000	b000000000000000000000000000 ¹
5	C002010000000000000100000000	b000000000000000000000000000 ¹
1. The symbol <i>b</i> 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"0020" (a national space).
- Each of the remaining 27 national character positions of TRANSACTION-OUT contains NX"0030" (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 163

RELATED TASKS

"Initializing a table (INITIALIZE)" on page 67

"Using national groups" on page 164

RELATED REFERENCES

INITIALIZE statement (*COBOL for Windows 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 159

RELATED TASKS

“Assigning values to group data items (MOVE)”

“Converting to or from national (Unicode) representation” on page 167

RELATED REFERENCES

Classes and categories of data (*COBOL for Windows Language Reference*)

MOVE statement (*COBOL for Windows 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 159

“National groups” on page 163

RELATED TASKS

“Assigning values to elementary data items (MOVE)” on page 32

“Using national groups” on page 164

“Converting to or from national (Unicode) representation” on page 167

RELATED REFERENCES

Classes and categories of group items (*COBOL for Windows Language Reference*)

MOVE statement (*COBOL for Windows Language Reference*)

CORRESPONDING phrase (*COBOL for Windows 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 238

Intrinsic functions (*COBOL for Windows 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 SET command. For example:
SET 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 159

RELATED TASKS

“Setting environment variables” on page 193

“Converting alphanumeric and DBCS data to national data (NATIONAL-OF)” on page 169

“Getting the system date under CICS” on page 328

RELATED REFERENCES

ACCEPT statement (*COBOL for Windows Language Reference*)

SPECIAL-NAMES paragraph (*COBOL for Windows 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 159

RELATED TASKS

“Converting national data to alphanumeric data (DISPLAY-OF)” on page 169

“Coding COBOL programs to run under CICS” on page 327

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). 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 as long as 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 78

“Converting data items (intrinsic functions)” on page 104

“Evaluating data items (intrinsic functions)” on page 107

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.
- 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 61

“Using procedure and function pointers” on page 475

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"

"Displaying numeric data" on page 41

"Controlling how numeric data is stored" on page 42

"Checking for incompatible data (numeric class test)" on page 51

"Performing arithmetic" on page 52

"Using currency signs" on page 59

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 zeroes

S Indicates a sign, positive or negative

V Implies a decimal point

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 46

RELATED CONCEPTS

“Formats for numeric data” on page 43

Appendix C, “Intermediate results and arithmetic precision,” on page 569

RELATED TASKS

“Displaying numeric data” on page 41

“Controlling how numeric data is stored” on page 42

“Performing arithmetic” on page 52

“Defining national numeric data items” on page 163

RELATED REFERENCES

“Sign representation of zoned and packed-decimal data” on page 50

“Storage of national data” on page 167

“ARITH” on page 228

SIGN clause (*COBOL for Windows 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, declare 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 zeroes:

```
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 46

RELATED TASKS

“Displaying values on a screen or in a file (DISPLAY)” on page 35

“Controlling how numeric data is stored”

“Defining numeric data” on page 39

“Performing arithmetic” on page 52

“Defining national numeric data items” on page 163

“Converting to or from national (Unicode) representation” on page 167

RELATED REFERENCES

MOVE statement (*COBOL for Windows Language Reference*)

BLANK WHEN ZERO clause (*COBOL for Windows 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 46

RELATED CONCEPTS

“Formats for numeric data”

“Data format conversions” on page 49

Appendix C, “Intermediate results and arithmetic precision,” on page 569

RELATED TASKS

“Defining numeric data” on page 39

“Displaying numeric data” on page 41

“Performing arithmetic” on page 52

RELATED REFERENCES

“Conversions and precision” on page 49

“Sign representation of zoned and packed-decimal data” on page 50

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. This means that 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 (80-bit IEEE) floating-point format.

Binary (COMP) items

BINARY, COMP, and COMP-4 are synonyms. Binary-format numbers occupy 2, 4, or 8 bytes of storage. Except for byte-reversed binary data (where the sign bit is the leftmost bit of the rightmost byte), this format is fixed point with the leftmost bit 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 declare 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
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 Windows 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).

Byte reversal of binary data

On a Windows-based workstation you sometimes need to be concerned with byte reversal. How binary data is stored depends on your hardware and software. For example, Intel® platforms by default store binary data in *little-endian* format (most significant digit at the highest address). zSeries® and AIX® store binary data in *big-endian* format (least significant digit at the highest address).

The `BINARY(NATIVE|S390)` compiler option lets you specify whether the binary data types (`BINARY`, `COMP`, and `COMP-4`) are to be stored in big-endian or little-endian format.

The compiler handles `COMP-5` as the native binary data format regardless of the `BINARY(NATIVE|S390)` setting.

Use `COMP-5` when your application interfaces with other languages (such as C/C++) or other products (such as CICS or DB2) that assume native binary data formats. However, a `SORT` or `MERGE` statement must not contain both big-endian and little-endian binary keys. For example, if the `BINARY(S390)` option is in effect and a `SORT` or `MERGE` key is a `COMP-5` data item, no other `SORT` or `MERGE` key can be a `COMP`, `BINARY`, or `COMP-4` data item.

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(S390)` (or its synonym, `FLOAT(HEX)`) is in effect, `COMP-1` and `COMP-2` data items are represented consistently with zSeries, that is, in hexadecimal floating-point format. For details, see the description of the `FLOAT` option referenced below.

RELATED CONCEPTS

“Unicode and the encoding of language characters” on page 159
Appendix C, “Intermediate results and arithmetic precision,” on page 569

RELATED TASKS

“Defining numeric data” on page 39
“Defining national numeric data items” on page 163

RELATED REFERENCES

“Storage of national data” on page 167
“TRUNC” on page 266
“BINARY” on page 229
“FLOAT” on page 247
Classes and categories of data (*COBOL for Windows Language Reference*)
SIGN clause (*COBOL for Windows Language Reference*)
VALUE clause (*COBOL for Windows 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 in native
| data format. Numeric items that have `USAGE NATIONAL` are represented in UTF16-LE

(little-endian) encoding. Assume that the BINARY(NATIVE), CHAR(NATIVE), and FLOAT(NATIVE) compiler options are in effect.

Table 4. 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
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
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
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

Table 4. Internal representation of native numeric items (continued)

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
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 zSeries data format. Numeric items that have USAGE NATIONAL are represented in UTF16-LE encoding. Assume that the BINARY(S390), CHAR(EBCDIC), and FLOAT(HEX) compiler options are in effect.

Table 5. Internal representation of numeric items when BINARY(S390), CHAR(EBCDIC), and FLOAT(HEX) 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	31 00 32 00 33 00 34 00
	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	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
Binary	PIC S9999 BINARY PIC S9999 COMP PIC S9999 COMP-4	+ 1234	04 D2
		- 1234	FB 2E
	PIC S9999 COMP-5	+ 12345 ¹	39 30
		- 12345 ¹	C7 CF
	PIC 9999 BINARY PIC 9999 COMP PIC 9999 COMP-4	1234	04 D2
	PIC 9999 COMP-5	60000 ¹	60 EA
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

Table 5. Internal representation of numeric items when BINARY(S390), CHAR(EBCDIC), and FLOAT(HEX) are in effect (continued)

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
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
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	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
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.			

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.

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 as long as 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”

“Sign representation of zoned and packed-decimal data” on page 50

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-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.

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 569

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:

0, 1, 2, 3, 8, 9, A, and B

Negative:

4, 5, 6, 7, C, D, E, and F

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 Windows and host COBOL. Host COBOL generates F internally as the sign of unsigned internal decimal numbers.

RELATED REFERENCES

"ZWB" on page 271

"Data representation" on page 561

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 math and date 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”

“Using arithmetic expressions” on page 53

“Using numeric intrinsic functions” on page 53

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 56
Appendix C, “Intermediate results and arithmetic precision,” on page 569

RELATED TASKS

“Defining numeric data” on page 39

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 6. 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 56
Appendix C, “Intermediate results and arithmetic precision,” on page 569

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 (80-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 7. Numeric intrinsic functions

Number handling	Date and time	Finance	Mathematics	Statistics
LENGTH MAX MIN NUMVAL NUMVAL-C ORD-MAX ORD-MIN	CURRENT-DATE DATE-OF-INTEGERS DATE-TO-YYYYMMDD DATEVAL DAY-OF-INTEGERS DAY-TO-YYYYDDD INTEGER-OF-DATE INTEGER-OF-DAY UNDATE 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”

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 56

Appendix C, “Intermediate results and arithmetic precision,” on page 569

RELATED REFERENCES

“ARITH” on page 228

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"

RELATED REFERENCES

"Arithmetic expressions in nonarithmetic statements" on page 577

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)e+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'D5' used as the currency sign:

Currency Sign X'D5' 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 either X'80' or X'88' depending on the code page in use, as shown in the following table.

Table 8. Hexadecimal value of the euro sign

Code page	Euro sign
1250 (Latin 2)	X'80'
1251 (Cyrillic)	X'88'
1252 (Latin 1)	X'80'
1253 (Greek)	X'80'
1254 (Turkish)	X'80'
1255 (Hebrew)	X'80'
1256 (Arabic)	X'80'
1257 (Baltic)	X'80'

Table 8. Hexadecimal value of the euro sign (continued)

Code page	Euro sign
874 (Thai)	X'80'

“Example: multiple currency signs”

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 S9999V99.
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.
```

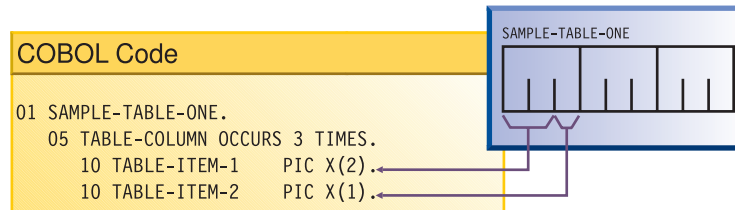
The above example produces the following display output:

```
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; it 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 one that looks up records.

RELATED TASKS

"Nesting tables" on page 63

"Defining a table (OCCURS)"

"Referring to an item in a table" on page 64

"Putting values into a table" on page 67

"Creating variable-length tables (DEPENDING ON)" on page 72

"Searching a table" on page 75

"Processing table items using intrinsic functions" on page 78

"Handling tables efficiently" on page 541

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 appear in a level-01 description.

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.

“Example: binary search” on page 77

RELATED CONCEPTS

“National groups” on page 163

RELATED TASKS

“Nesting tables” on page 63

“Referring to an item in a table” on page 64

“Putting values into a table” on page 67

“Creating variable-length tables (DEPENDING ON)” on page 72

“Using national groups” on page 164

“Doing a binary search (SEARCH ALL)” on page 77

“Defining numeric data” on page 39

RELATED REFERENCES

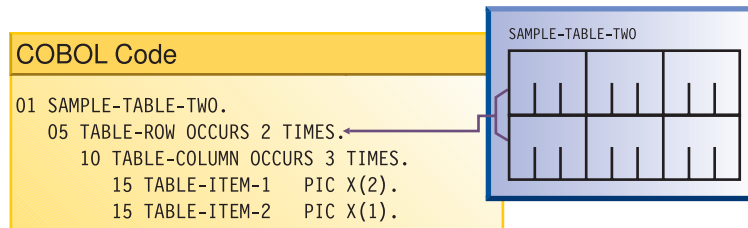
OCCURS clause (*COBOL for Windows Language Reference*)

SIGN clause (*COBOL for Windows Language Reference*)

ASCENDING KEY and DESCENDING KEY phrases (*COBOL for Windows Language Reference*)

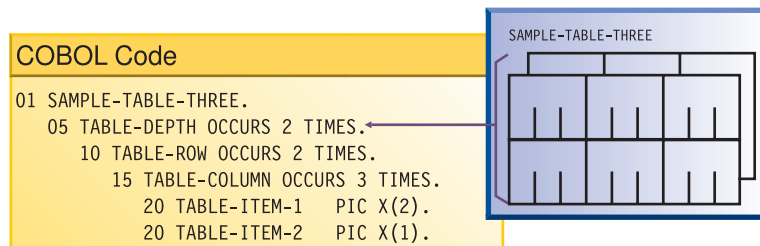
Nesting tables

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 64

"Example: indexing" on page 64

RELATED TASKS

"Defining a table (OCCURS)" on page 61

"Referring to an item in a table" on page 64

"Putting values into a table" on page 67

"Creating variable-length tables (DEPENDENT ON)" on page 72

"Searching a table" on page 75

"Processing table items using intrinsic functions" on page 78

"Handling tables efficiently" on page 541

RELATED REFERENCES

OCCURS clause (*COBOL for Windows 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 65

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 66

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”

“Indexing” on page 66

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.

ANY-TABLE before the change:	
1	ABCDEFGHIJ
2	ABCDEFGHIJ
3	ABCDEFGHIJ

ANY-TABLE after the change:	
1	AB??EFGHIJ
2	ABCDEFGHIJ
3	ABCDEFGHIJ

“Example: subscripting” on page 64

RELATED TASKS

“Indexing” on page 66

“Putting values into a table” on page 67

“Searching a table” on page 75

“Handling tables efficiently” on page 541

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 64

RELATED TASKS

“Subscripting” on page 65

“Putting values into a table”

“Searching a table” on page 75

“Processing table items using intrinsic functions” on page 78

“Handling tables efficiently” on page 541

RELATED REFERENCES

INDEXED BY phrase (*COBOL for Windows Language Reference*)

INDEX phrase (*COBOL for Windows Language Reference*)

SET statement (*COBOL for Windows 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”

“Loading a variable-length table” on page 74

“Initializing a table (INITIALIZE)”

“Assigning values when you define a table (VALUE)” on page 69

“Assigning values to a variable-length table” on page 75

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 70

“Example: PERFORM and indexing” on page 71

RELATED REFERENCES

PERFORM with VARYING phrase (*COBOL for Windows 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-quant 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(<i>n</i>) before	Trans-out(<i>n</i>) after
R13105025375	XXb030303003 ¹
1. The symbol <i>b</i> 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 28

RELATED TASKS

- “Initializing a structure (INITIALIZE)” on page 30
- “Assigning values when you define a table (VALUE)” on page 69
- “Assigning values to a variable-length table” on page 75
- “Looping through a table” on page 91
- “Using data items and group items” on page 24
- “Using national groups” on page 164

RELATED REFERENCES

INITIALIZE statement (*COBOL for Windows 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”

“Initializing a table at the group level” on page 70

“Initializing all occurrences of a given table element” on page 70

“Initializing a structure (INITIALIZE)” on page 30

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. Declare 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 30

“Assigning values to a variable-length table” on page 75

RELATED REFERENCES

REDEFINES clause (*COBOL for Windows Language Reference*)

OCCURS clause (*COBOL for Windows 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"00410042" (the UTF-16 representation of 'AB'), the national decimal item ElementTwoN(1) contains NX"00310032" (the UTF-16 representation of '12'), and so forth.

RELATED REFERENCES

OCCURS clause (*COBOL for Windows Language Reference*)

GROUP-USAGE clause (*COBOL for Windows 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 75

RELATED REFERENCES

OCCURS clause (*COBOL for Windows 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

Appendix D, “Complex OCCURS DEPENDING ON,” on page 579

RELATED TASKS

“Assigning values to a variable-length table” on page 75

“Loading a variable-length table”

“Preventing overlay when adding elements to a variable table” on page 581

“Finding the length of data items” on page 110

RELATED REFERENCES

OCCURS DEPENDING ON clause (*COBOL for Windows 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 69

RELATED REFERENCES

OCCURS DEPENDING ON clause (*COBOL for Windows Language Reference*)

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)"

"Doing a binary search (SEARCH ALL)" on page 77

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”

RELATED REFERENCES

SEARCH statement (*COBOL for Windows 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 "AAAAAAA00" 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)
      = "A1234AAAAAAA00"
      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 "A1234AAAAAAA00"
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.

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”

RELATED TASKS

“Defining a table (OCCURS)” on page 61

RELATED REFERENCES

SEARCH statement (*COBOL for Windows Language Reference*)

General relation conditions (*COBOL for Windows 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.

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”

RELATED TASKS

“Using intrinsic functions (built-in functions)” on page 36

“Converting data items (intrinsic functions)” on page 104

“Evaluating data items (intrinsic functions)” on page 107

RELATED REFERENCES

Intrinsic functions (*COBOL for Windows 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.
. . .
Procédure 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”

“Repeating program actions” on page 89

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”

“Coding conditional expressions” on page 86

RELATED REFERENCES

IF statement (*COBOL for Windows Language Reference*)

EVALUATE statement (*COBOL for Windows 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.

RELATED TASKS

“Using nested IF statements”

“Using the EVALUATE statement” on page 83

“Coding conditional expressions” on page 86

Using nested IF statements

When 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.

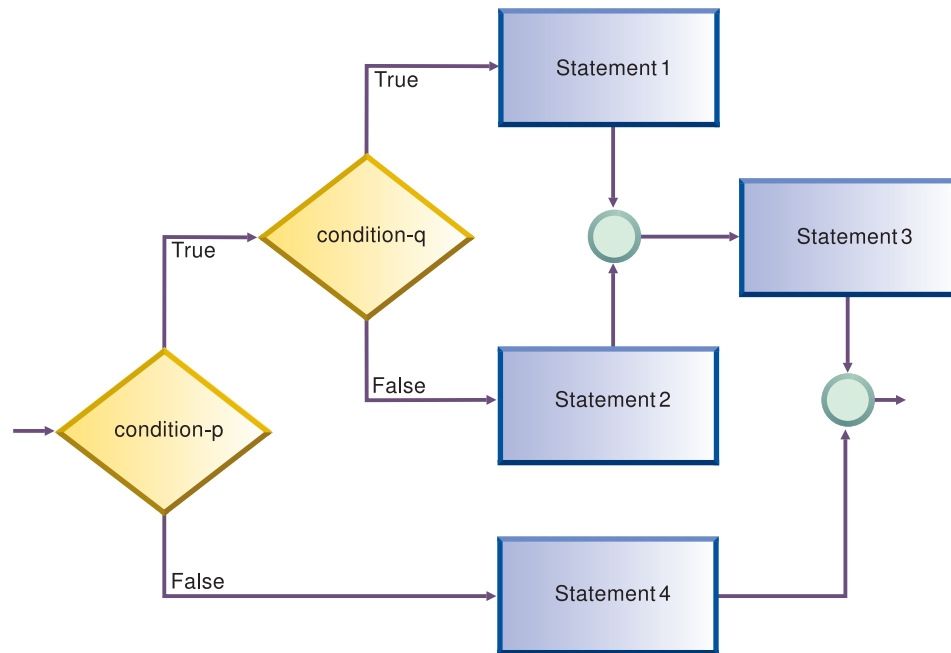
However, use nested IF statements sparingly. The logic can be difficult to follow, although explicit scope terminators and indentation help. When a program has to test a variable for more than two values, EVALUATE is probably a better choice.

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 Windows 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 84

“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 85

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 Windows Language Reference*)

General relation conditions (*COBOL for Windows 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 or 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.

The following are some of the conditions that you can specify:

- 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 DBCS
 - IS KANJI
 - IS NOT 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 87

RELATED TASKS

“Defining switches and flags” on page 87

“Resetting switches and flags” on page 88

“Checking for incompatible data (numeric class test)” on page 51

“Comparing national (UTF-16) data” on page 172

“Testing for valid DBCS characters” on page 177

RELATED REFERENCES

“UPSI” on page 296

General relation conditions (*COBOL for Windows Language Reference*)

Class condition (*COBOL for Windows Language Reference*)

Rules for condition-name entries (*COBOL for Windows Language Reference*)

Sign condition (*COBOL for Windows Language Reference*)
Combined conditions (*COBOL for Windows 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”

“Resetting switches and flags” on page 88

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”

“Example: flags” on page 88

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 turned on 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
```

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 master 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-Master-Record-Paragraph
  When Change-Transaction
    Perform Update-Existing-Record-Paragraph
  When Delete-Transaction
    Perform Delete-Master-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" "0" "S"
```

Using the SET statement and meaningful condition-names makes it easier for readers to follow your code.

“Example: set switch on”

“Example: set switch off” on page 89

Example: set switch on

The following examples show how you can set a switch on by coding a SET statement that moves the value TRUE to a level-88 item.

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 a value to a level-88 item.

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 90

“Coding a loop” on page 91

“Looping through a table” on page 91

“Executing multiple paragraphs or sections” on page 92

RELATED REFERENCES

PERFORM statement (*COBOL for Windows 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 paragraph and an implicit return. If the performed paragraph is in the subsequent sequential flow of your program, it is also executed in that logic flow. To avoid this additional execution, place the paragraph 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”

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 paragraph 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 paragraph `010-PROCESS-ONE-MONTH` is executed 12 times before control is transferred to the `INSPECT` statement.

Use the `PERFORM . . . UNTIL` statement to execute a paragraph 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 . . .
```

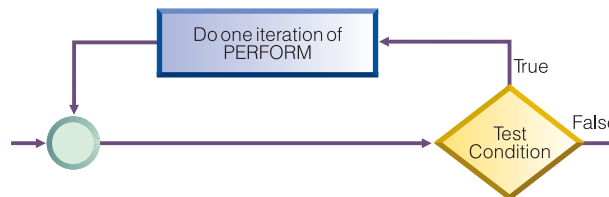
Use the `PERFORM . . . WITH TEST AFTER . . . UNTIL` statement if you want to execute the paragraph at least once, and test before any subsequent execution. This statement is equivalent to a do-until structure:



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 78

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)"

"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 103

"Converting data items (intrinsic functions)" on page 104

"Evaluating data items (intrinsic functions)" on page 107

Chapter 10, "Processing data in an international environment," on page 155

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

Only that portion of the receiving field into which data is written by the STRING statement is changed.

“Example: STRING statement”

RELATED TASKS

“Handling errors in joining and splitting strings” on page 145

RELATED REFERENCES

STRING statement (*COBOL for Windows 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.bSMITHbbbbbb
444bSPRINGbST.,bCHICAGO,bILL.bbbbbbb
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.

Column					
4	10		60	67	76
↓	↓		↓	↓	↓
00001	J.B. SMITH	444 SPRING ST., CHICAGO, ILL.	A14275	10/22/76	\$2,336

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”

RELATED CONCEPTS

“Unicode and the encoding of language characters” on page 159

RELATED TASKS

“Handling errors in joining and splitting strings” on page 145

RELATED REFERENCES

UNSTRING statement (*COBOL for Windows Language Reference*)

Classes and categories of data (*COBOL for Windows 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.

Column	1	10	20	30	40	50	60
	↓	↓	↓	↓	↓	↓	↓
	ZYFOUR-PENNY-NAILS		707890/BBA	475120	00122	000379.50	

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-DOLLS. 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	4751200000122BBA
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”

RELATED TASKS

“Handling null-terminated strings” on page 472

RELATED REFERENCES

Alphanumeric literals (*COBOL for Windows 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:

1. Ordinal position (from the left) of the character that you want the substring to start with
2. (Optional) Length of the desired 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"0032") 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 102

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”

“Unicode and the encoding of language characters” on page 159

RELATED TASKS

“Referring to an item in a table” on page 64

RELATED REFERENCES

“SSRANGE” on page 263

Reference modification (*COBOL for Windows Language Reference*)

Function definitions (*COBOL for Windows 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 102

“Example: intrinsic functions as reference modifiers” on page 102

RELATED TASKS

“Assigning input from a screen or file (ACCEPT)” on page 34

“Referring to substrings of data items” on page 99

“Using national data (Unicode) in COBOL” on page 160

RELATED REFERENCES

Reference modification (*COBOL for Windows 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 Windows Language Reference*)

INTEGER-PART (*COBOL for Windows 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”

RELATED CONCEPTS

“Unicode and the encoding of language characters” on page 159

RELATED REFERENCES

INSPECT statement (*COBOL for Windows 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

DATA-2 before	COUNTR after	DATA-2 after
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
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

“Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)” on page 105

“Transforming to reverse order (REVERSE)”
“Converting to numbers (NUMVAL, NUMVAL-C)”
“Converting from one code page to another” on page 107

Converting to uppercase or lowercase (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 34
“Displaying values on a screen or in a file (DISPLAY)” on page 35

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 JOHNSONbbb, the value after the statement is performed is bbbNOSNH0J, where *b* represents a blank space.

RELATED CONCEPTS

“Unicode and the encoding of language characters” on page 159

Converting to numbers (NUMVAL, NUMVAL-C)

The NUMVAL and NUMVAL-C 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.

```

01 R      Pic x(20) Value "- 1234.5678".
01 S      Pic x(20) Value " $12,345.67CR".
01 Total  Usage is Comp-1.
. . .
    Compute Total = Function Numval(R) + Function Numval-C(S)

```

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 and NUMVAL-C return long (64-bit) floating-point values in compatibility mode, and return extended-precision (80-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 or NUMVAL-C 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 or NUMVAL-C, you do not need to statically declare 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 43

"Data format conversions" on page 49

"Unicode and the encoding of language characters" on page 159

RELATED TASKS

"Converting to or from national (Unicode) representation" on page 167

RELATED REFERENCES

“Conversions and precision” on page 49
“ARITH” on page 228

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 159

RELATED TASKS

“Converting to or from national (Unicode) representation” on page 167

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 159

RELATED TASKS

“Evaluating single characters for collating sequence”
“Finding the largest or smallest data item” on page 108
“Finding the length of data items” on page 110
“Finding the date of compilation” on page 111

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 desired character. For example:

```
INITIALIZE Customer-Name REPLACING ALPHABETIC BY Function Char(65)
```

RELATED REFERENCES

CHAR (*COBOL for Windows Language Reference*)

ORD (*COBOL for Windows 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 VALLEJ0bbb 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 THOMASbbb 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 example above, the compiler would issue an error message because the reference to a numeric function is not in a valid place. The following statement is a valid use of ORD-MAX:

```
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 52

“Processing table items using intrinsic functions” on page 78

“Returning variable-length results with alphanumeric or national functions” on page 109

RELATED REFERENCES

MAX (*COBOL for Windows Language Reference*)

MIN (*COBOL for Windows Language Reference*)

ORD-MAX (*COBOL for Windows Language Reference*)

ORD-MIN (*COBOL for Windows Language Reference*)

Returning variable-length 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 'fbbbb' 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 'gbbbbbbbbb' 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"0066 0020 0020 0020 0020" (the equivalent in national characters of 'fbbbb', 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 108

“Performing arithmetic” on page 52

RELATED REFERENCES

MAX (*COBOL for Windows 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.

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))
```

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 allows 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 52

“Creating variable-length tables (DEPENDING ON)” on page 72

“Processing table items using intrinsic functions” on page 78

RELATED REFERENCES

LENGTH (*COBOL for Windows Language Reference*)

LENGTH OF (*COBOL for Windows 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 carries the time out only to seconds. You can use this special register only as the sending field in a MOVE statement.

RELATED REFERENCES

WHEN-COMPILED (*COBOL for Windows 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 need to 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 them and their file system. You might also want to set up a file status field that you can check later to make sure 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 system" on page 114

RELATED TASKS

"Identifying files"

"Protecting against errors when opening files" on page 118

"Specifying a file organization and access mode" on page 118

"Setting up a field for file status" on page 122

"Describing the structure of a file in detail" on page 123

"Coding input and output statements for files" on page 123

Identifying files

In the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION, you must specify a name for the file that you want to use and optionally specify a name for the file system, as shown below.

```
SELECT file ASSIGN TO FileSystemID-Filename
```

file is the name that you use inside the program to refer to the file. It is not necessarily the actual name of the file as known by the system.

FileSystemID identifies one of the following file systems:

STL Standard language file system

BTR Btrieve (Pervasive.SQL) file system

RSD Record sequential delimited file system

If you do not provide the file-system specification, the runtime option FILESYS is used to select the file system. The default for the FILESYS runtime option is STL.

Filename is the name of the physical file that you want to access. Alternatively, you can use an environment variable to specify the file-name at run time.

RELATED CONCEPTS

"File system" on page 114

RELATED TASKS

"Identifying Btrieve files"

"Identifying STL files"

"Identifying RSD files"

RELATED REFERENCES

"STL file system" on page 115

"RSD file system" on page 118

"Runtime environment variables" on page 196

"FILESYS" on page 294

Identifying Btrieve files

To use the Btrieve (Pervasive.SQL) file system for a file, identify the file with the SELECT clause as shown below.

```
SELECT file1 ASSIGN TO 'BTR-MyFile'
```

If the runtime option FILESYS(BTRIEVE) is in effect, the following assignment is valid:

```
SELECT file1 ASSIGN TO 'MyFile'
```

If you have defined the environment variable *MYFILE* (for example, SET *MYFILE*=BTR-*MYFILE*), the following assignment is valid:

```
SELECT file1 ASSIGN TO MYFILE
```

Identifying STL files

To use the standard language file system for a file, identify the file with the SELECT clause as shown below.

```
SELECT file1 ASSIGN USING 'STL-MyFile'
```

If the runtime option FILESYS(STL) is in effect, identify the file as follows:

```
SELECT file1 ASSIGN TO 'MyFile'
```

If you have defined the environment variable *MYFILE* (for example, SET *MYFILE*=STL-*MYFILE*), identify the file as follows:

```
SELECT file1 ASSIGN TO MYFILE
```

Identifying RSD files

To use the record sequential delimited file system for a file, identify the file with the SELECT clause as shown below.

```
SELECT file1 ASSIGN USING 'RSD-MyFile'
```

If the runtime option FILESYS(RSD) is in effect, identify the file as follows:

```
SELECT file1 ASSIGN TO 'MyFile'
```

If you have defined the environment variable *MYFILE* (for example, SET *MYFILE*=RSD-*MYFILE*), identify the file as follows:

```
SELECT file1 ASSIGN TO MYFILE
```

File system

Record-oriented files that are organized as sequential, relative, or indexed are accessed through a *file system*. You can use file-system functions to create and manipulate the records in any of these types of files.

COBOL for Windows supports the following file systems:

- The STL (standard language) file system, which provides the basic facilities for local files. It is provided with COBOL for Windows, and supports sequential, relative, and indexed files.
- The Btrieve file system, which lets you access Btrieve files. Btrieve is IBM's name for the Pervasive.SQL file system, a separate product available from Pervasive Software.
- The RSD (record sequential delimited) file system, which lets you share data with programs written in other languages. RSD files are sequential only and support all COBOL data types in records. Text data in records can be edited by most file editors.

Most programs will have the same behaviors on all file systems. However, files written using one file system cannot be read using a different file system.

You can select a file system by setting the assignment-name environment variable or by using the FILESYS runtime option. All the file systems let you use COBOL statements to read and write COBOL files.

RELATED TASKS

"Identifying files" on page 113

RELATED REFERENCES

"STL file system"

"RSD file system" on page 118

zSeries host data format considerations ("File data" on page 568)

"FILESYS" on page 294

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 Standard COBOL 85, and gives good performance and the ability to port easily between AIX and Windows-based systems. The LINE SEQUENTIAL organization is the only file organization type not supported by the STL file system.

The file system is safe for use with threads. However, you must ensure that multiple threads do not access the level-01 records for the file at the same time. Multiple threads can perform operations on the same STL file, but you must use an operating system call (for example, `WaitForSingleObject`) to force all but one of the threads to wait for the file access to complete on the active thread.

With the STL file system, you can easily read and write files to be shared with PL/I programs.

RELATED CONCEPTS

"File organization and access mode" on page 119

RELATED TASKS

"Identifying STL files" on page 114

RELATED REFERENCES

"STL file system return codes" on page 116

STL file system return codes

After an input-output operation on an STL file, one of several error codes can occur.

FILE STATUS *data-name-1 data-name-8*

In the FILE-CONTROL paragraph, you can code the clause shown above for an STL file. After an input-output operation on the file, *data-name-1* contains a status code that is independent of the file system used, and *data-name-8* contains one of the STL file system return codes shown in the tables below.

Table 9. STL file system return codes

Code	Meaning	Notes
0	Successful completion	The input-output operation completed successfully.
1	Invalid operation	This return code should not occur; it indicates an error in the file system.
2	I/O error	A call to an operating system I/O routine returned an error code.
3	File not open	Attempt to perform an operation (other than OPEN) on a file that is not open
4	Key value not found	Attempt to read a record using a key that is not in the file
5	Duplicate key value	Attempt to use a key a second time for a key that does not allow duplicates
6	Invalid key number	This return code should not occur; it indicates an error in the file system.
7	Different key number	This return code should not occur; it indicates an error in the file system.
8	Invalid flag for the operation	This return code should not occur; it indicates an error in the file system.
9	End-of-file	End-of-file was detected. This is not an error.
10	I/O operation must be preceded by an I/O GET operation	The operation is looking for the current record, and the current record has not been defined.
11	Error return from get space routine	The operating system indicates that not enough memory is available.
12	Duplicate key accepted	The operation specified a key, and the key is a duplicate.
13	Sequential access and key sequence bad	Sequential access was specified, but the records are not in sequential order.
14	Record length < max key	The record length does not allow enough space for all of the keys.
15	Access to file denied	The operating system reported that it cannot access the file. Either the file does not exist or the user does not have the proper permission of the operating system to access the file.
16	File already exists	You attempted to open a new file, but the operating system reports that the file already exists.
17	(Reserved)	

Table 9. STL file system return codes (continued)

Code	Meaning	Notes
18	File locked	Attempt to open a file that is already open in exclusive mode
19	File table full	The operating system reports that its file table is full.
20	Handle table full	The operating system reports that it cannot allocate any more file handles.
21	Title does not say STL	Files opened for reading by the STL file system must contain a header record that contains "STL" at a certain offset in the file.
22	Bad indexcount argument for create	This return code should not occur; it indicates an error in the file system.
23	Index or relative record > 64 KB	Index and relative records are limited to a length of 64 KB.
24	Error found in file header or data in open of existing file	STL files begin with a header. The header or its associated data has inconsistent values.
25	Indexed open on sequential file	Attempt to open a sequential file as an indexed or relative file

The following table shows return codes for errors detected in the adapter open routines.

Table 10. STL file system adapter open routine return codes

Code	Meaning	Notes
1000	Sequential open on indexed or relative file	Attempt to open an indexed or relative file as a sequential file
1001	Relative open of indexed file	Attempt to open a relative file as an indexed file
1002	Indexed open of sequential file	Attempt to open an indexed file as a sequential file
1003	File does not exist	The operating system reports that the file does not exist.
1004	Number of keys differ	Attempt to open a file with a different number of keys
1005	Record lengths differ	Attempt to open a file with a different record length
1006	Record types differ	Attempt to open a file with a different record type
1007	Key position or length differs	Attempt to open a file with a different key position or length

RELATED TASKS

"Using file status keys" on page 148

RELATED REFERENCES

"STL file system" on page 115

FILE STATUS clause (*COBOL for Windows Language Reference*)

RSD file system

The RSD (record sequential delimited) file system supports sequential files. You can process RSD files with the standard system file utility functions, such as browse, edit, copy, delete, and print.

RSD files give good performance. They give you the ability to port files easily between AIX 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 length. Each record written is followed by a new-line control character. For a read operation, the file record area is filled for the record length. If an end-of-file is encountered before the record area is filled, the record area is padded with space characters.

If you edit an RSD file with a text editor, you must ensure that the length of each record is maintained.

Attempting a WRITE statement with the AFTER ADVANCING or BEFORE ADVANCING phrase to an RSD file causes the statement to fail with the file status key set to 30.

The RSD file system is safe for use with threads. However, you must ensure that multiple threads do not access the level-01 records for the file at the same time. Multiple threads can perform operations on the same RSD file, but you must use an operating system call (for example, `WaitForSingleObject`) to force all but one of the threads to wait for the file access to complete on the active thread.

RELATED CONCEPTS

"File organization and access mode" on page 119

RELATED TASKS

"Identifying RSD files" on page 114

Protecting against errors when opening 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, use the optional keyword `OPTIONAL` with `SELECT`:

```
SELECT OPTIONAL file ASSIGN TO filename
```

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 11. 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	No	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”
“Line-sequential file organization” on page 120
“Indexed file organization” on page 120
“Relative file organization” on page 121
“Sequential access” on page 121
“Random access” on page 121
“Dynamic access” on page 121

RELATED TASKS

“Specifying a file organization and access mode” on page 118

RELATED REFERENCES

“File input-output limitations” on page 122

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 121

RELATED REFERENCES

“Valid COBOL statements for sequential files” on page 126

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 with WRITE statements with the ADVANCING phrase can be directed to a printer or to a disk.

RELATED CONCEPTS

“Sequential file organization” on page 119

RELATED REFERENCES

“Valid COBOL statements for line-sequential files” on page 126

Indexed file organization

An indexed file contains records ordered by a *record key*. The record key uniquely identifies the 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 the file through employee department rather than through employee number.

The record transmission (access) modes allowed 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 127

Relative file organization

A relative record file contains records ordered by their *relative key*, that is, 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 allowed 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 127

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”

“Dynamic access”

RELATED REFERENCES

“File position indicator” on page 125

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”

“Dynamic access”

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 121

“Random access” on page 121

File input-output limitations

The size limits of records and files are as shown below.

- For line-sequential files:
 - Maximum record size: 64 KB
 - Maximum number of bytes allocated for a file: 2 GB
- For STL files:
 - Minimum record size: 1 byte
 - Maximum record size: 65,535 bytes
 - Maximum record key length: 255 bytes
 - Maximum number of alternate keys: 253
 - Maximum relative key value: $2^{32} - 1$
 - Maximum number of bytes allocated for a file: $2^{31} - 1$ (relative and indexed files)
 - Maximum number of bytes allocated for a file: no limit (sequential files)
- For RSD files:
 - Fixed-length records only
 - Minimum record size: 1 byte
 - Maximum record size: 65,535 bytes
 - Maximum number of bytes allocated for a file: 2 GB

Additional or more restrictive limits might be applicable depending on the platform where a target file is located.

RELATED REFERENCES

Compiler limits (*COBOL for Windows 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 148

RELATED REFERENCES

FILE STATUS clause (*COBOL for Windows 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 Windows Language Reference*)

Level-numbers (*COBOL for Windows Language Reference*)

PICTURE clause (*COBOL for Windows Language Reference*)

USAGE clause (*COBOL for Windows 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 124

RELATED TASKS

“Identifying files” on page 113

“Protecting against errors when opening files” on page 118

“Specifying a file organization and access mode” on page 118

“Setting up a field for file status” on page 122

“Describing the structure of a file in detail”

“Opening a file” on page 125
 “Reading records from a file” on page 128
 “Adding records to a file” on page 129
 “Replacing records in a file” on page 130
 “Deleting records from a file” on page 130

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) follow 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 actual name of the file as known to the system. 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 the file-name as it is known to the system directly, or you can set a 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 identification by the file-system type identification.

The following example illustrates how inventory-file is dynamically (by way of a MOVE statement) associated with the file d:\inventory\parts.

```

SELECT inventory-file ASSIGN USING a-file . . .
. . .
FD inventory-file . . .
  
```

```

      . . .
77 a-file PIC X(20) VALUE SPACES.
      . . .
      MOVE "d:\inventory\parts" 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-0.

Restriction: I-0 is not a valid parameter of OPEN for line-sequential files.

File position indicator

The file position indicator marks the next record to be accessed for sequential COBOL requests.

You do not 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 with 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 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 119

RELATED TASKS

“Protecting against errors when opening files” on page 118

RELATED REFERENCES

“Valid COBOL statements for sequential files”

“Valid COBOL statements for line-sequential files”

“Valid COBOL statements for indexed and relative files” on page 127

OPEN statement (*COBOL for Windows 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 12. 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 119

“Sequential access” on page 121

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 13. 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 120

“Sequential access” on page 121

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 14. 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 120

"Relative file organization" on page 121

"Sequential access" on page 121

"Random access" on page 121

"Dynamic access" on page 121

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.

Records in sequential and line-sequential files can be retrieved only in the sequence in which they were written.

Records in indexed and relative record files can be retrieved sequentially (according to the ascending order of the key you are using for indexed files, or according to ascending relative record locations for relative files), randomly, or dynamically.

With dynamic access, you can switch between reading a specific record directly and reading records sequentially by using READ NEXT and READ PREVIOUS for sequential retrieval, and READ for random retrieval (by key).

When you want to read sequentially beginning at a specific record, use START before the READ NEXT or the READ PREVIOUS statement to set the file position indicator to point to a particular record. When you code START followed by READ NEXT, the next record is read and the file position indicator is reset to the next record. When 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 the START statement to move the file position indicator. For example:

```
START file-name KEY IS EQUAL TO ALTERNATE-RECORD-KEY
```

When 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 data set records with the same alternate key. A file status value of 02 is returned if there are more records with the same alternate key value 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 121

"Random access" on page 121

"Dynamic access" on page 121

"File organization and access mode" on page 119

RELATED TASKS

"Opening a file" on page 125

"Using file status keys" on page 148

RELATED REFERENCES

“File position indicator” on page 125

FILE STATUS clause (*COBOL for Windows 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 15. Statements used when writing records to a file

Division	Sequential	Line sequential	Indexed	Relative
ENVIRONMENT	SELECT ASSIGN FILE STATUS ACCESS MODE	SELECT ASSIGN ORGANIZATION IS LINE SEQUENTIAL FILE STATUS ACCESS MODE	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY ALTERNATE RECORD KEY FILE STATUS ACCESS MODE	SELECT ASSIGN ORGANIZATION IS RELATIVE RELATIVE KEY FILE STATUS ACCESS MODE
DATA	FD entry	FD entry	FD entry	FD entry
PROCEDURE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE

RELATED CONCEPTS

“File organization and access mode” on page 119

RELATED TASKS

“Specifying a file organization and access mode” on page 118

“Opening a file” on page 125

“Setting up a field for file status” on page 122

“Adding records to a file”

RELATED REFERENCES

“PROCEDURE DIVISION statements used to update files” on page 131

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, new records must be written in ascending key sequence. If the file is opened EXTEND, the record keys of the records to be added must be higher than the highest primary record key in the file when the file was opened.

For relative files, the records must be in sequence. If you include 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 data set for which you have 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 119

RELATED TASKS

“Specifying a file organization and access mode” on page 118

“Using file status keys” on page 148

RELATED REFERENCES

“Statements used when writing records to a file” on page 129

“PROCEDURE DIVISION statements used to update files” on page 131

FILE STATUS clause (*COBOL for Windows Language Reference*)

Replacing records in a file

To replace a record in a file, use REWRITE on a file that you opened as I-0. If the file was not opened as I-0, 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 issue the REWRITE.
- For relative files, move the relative record number to the RELATIVE KEY data item, and then issue the REWRITE.

You cannot open as I-0 an extended-format data set that you allocate in the compressed format.

RELATED CONCEPTS

“File organization and access mode” on page 119

RELATED TASKS

“Opening a file” on page 125

“Using file status keys” on page 148

RELATED REFERENCES

FILE STATUS clause (*COBOL for Windows Language Reference*)

Deleting records from a file

To remove an existing record from an indexed or relative file, open the file I-0 and use the DELETE statement. You cannot use DELETE for a sequential or line-sequential file.

When 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 value is set to 92.

When 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 119

RELATED TASKS

“Opening a file” on page 125

“Reading records from a file” on page 128

“Using file status keys” on page 148

RELATED REFERENCES

FILE STATUS clause (*COBOL for Windows 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 16. 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 119

RELATED TASKS

"Opening a file" on page 125

"Reading records from a file" on page 128

"Adding records to a file" on page 129

"Replacing records in a file" on page 130

"Deleting records from a file" on page 130

RELATED REFERENCES

"Statements used when writing records to a file" on page 129

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”

RELATED TASKS

“Describing the sort or merge file” on page 134

“Describing the input to sorting or merging” on page 134

“Describing the output from sorting or merging” on page 136

“Requesting the sort or merge” on page 138

“Determining whether the sort or merge was successful” on page 141

“Stopping a sort or merge operation prematurely” on page 144

RELATED REFERENCES

SORT statement (*COBOL for Windows Language Reference*)

MERGE statement (*COBOL for Windows 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.

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 139

RELATED REFERENCES

SORT statement (*COBOL for Windows Language Reference*)

MERGE statement (*COBOL for Windows 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 12

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 136

“Requesting the sort or merge” on page 138

RELATED REFERENCES

“FILE SECTION entries” on page 12

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. SmpISort.
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 138

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 137

RELEASE statement (*COBOL for Windows 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”

“Requesting the sort or merge” on page 138

RELATED REFERENCES

“FILE SECTION entries” on page 12

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”

RETURN statement (*COBOL for Windows Language Reference*)

Restrictions on input and output procedures

The restrictions listed below apply to each input or output procedure called by SORT and to each output procedure called by MERGE.

- 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 136

“Coding the output procedure” on page 137

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 135

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 140

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 Windows Language Reference)

MERGE statement (COBOL for Windows 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 185

RELATED REFERENCES

“NCOLLSEQ” on page 252

SORT statement (COBOL for Windows Language Reference)

MERGE statement (COBOL for Windows 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 8

“Controlling the collating sequence with a locale” on page 185

“Setting sort or merge criteria” on page 139

RELATED REFERENCES

OBJECT-COMPUTER paragraph (*COBOL for Windows Language Reference*)

SORT statement (*COBOL for Windows Language Reference*)

Classes and categories of data (*COBOL for Windows 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 138

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
. . .

```

To call iwzGetSortErrno, use the SYSTEM call interface convention and the PGMNAME(MIXED) and NODYNAM compiler options.

See the related reference below for a list of the error numbers and their meanings.

RELATED REFERENCES

“Sort and merge error numbers”

“SYSTEM” on page 461

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 Windows issues a runtime diagnostic message that contains one of the nonzero error numbers shown in the table below.

Table 17. 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
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	VSAM file error

Table 17. Sort and merge error numbers (continued)

Error number	Description
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.
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	A VSAM organization was specified for the file, but the file is not VSAM.
64	No key specified to SORT is usable for definition of indexed output file.
65	VSAM fixed record length for the file does not agree with the specified record format.
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

Table 17. Sort and merge error numbers (continued)

Error number	Description
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
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 Windows 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"

"Handling errors in arithmetic operations" on page 146

"Handling errors in input and output operations" on page 146

"Handling errors when calling programs" on page 152

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)	EEEEA	EEEEA
Item-3	X(2)	EA	EA
Item-4	X(8)	bbbbbbb ¹	bbbbbbb ¹
String-ptr	9(2)	0	0

Data item	PICTURE	Value before	Value after
1. The symbol <i>b</i> represents a blank space.			

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”

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 on 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.

RELATED TASKS

"Using the end-of-file condition (AT END)"

"Coding ERROR declaratives" on page 148

"Using file status keys" on page 148

"Using file system status codes" on page 150

RELATED REFERENCES

File status key (*COBOL for Windows 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 master 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. 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, you should have some code that takes appropriate action.

RELATED REFERENCES

AT END phrases (*COBOL for Windows 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 Windows 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.
```

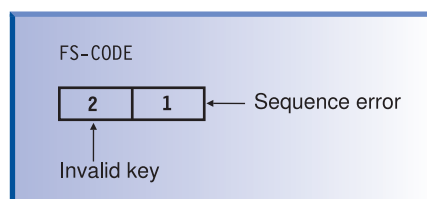
```
01 data-name-1 PIC 9(2) USAGE NATIONAL.
```

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.

Btrieve files: The following values for the file status key are not set for Btrieve files:

- 02
- 21
- 39

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. See the related reference below about file system status codes.

You can use the file status key alone or in conjunction with the INVALID KEY option, 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”

“Example: checking file system status codes” on page 150

RELATED TASKS

“Setting up a field for file status” on page 122

“Using file system status codes” on page 150

RELATED REFERENCES

FILE STATUS clause (*COBOL for Windows Language Reference*)

File status key (*COBOL for Windows 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 MASTERFILE ASSIGN TO AS-MASTERA
    FILE STATUS IS MASTER-CHECK-KEY
    . . .
DATA DIVISION.
    . . .
WORKING-STORAGE SECTION.
01  MASTER-CHECK-KEY          PIC X(2).
    . . .
PROCEDURE DIVISION.
    OPEN INPUT MASTERFILE
    IF MASTER-CHECK-KEY NOT = "00"
        DISPLAY "Nonzero file status returned from OPEN " MASTER-CHECK-KEY
    . . .
```

Using file system status codes

Often the two-digit file status code is too general to pinpoint the disposition of a request. You can get more detailed information about STL and Btrieve file-system input and output requests by coding a second data item in the FILE STATUS clause:

```
FILE STATUS IS data-name-1 data-name-8
```

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 when the COBOL file status key is not zero. *data-name-8* is at least 6 bytes long, and must be an alphanumeric item.

If *data-name-8* is 6 bytes long, it contains the 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).
```

In the example above, if you tried to open a file with a different definition than the one with which it was created, return code 39 would be returned in exception-return-value, and a message that tells which keys you need in order to perform the open would be returned in additional-info.

“Example: checking file system status codes”

RELATED REFERENCES

“STL file system” on page 115

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.

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.

```

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 master 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 master file, and makes the necessary updates. The records in both files contain a field for a customer number, and each record in the master file has a unique customer number.

The FILE-CONTROL entry for the master file of customer records includes statements that define indexed organization, random access, MASTER-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 MASTER-CUSTOMER-NUMBER
READ MASTER-CUSTOMER-FILE INTO WS-CUSTOMER-RECORD
  INVALID KEY
    DISPLAY "MASTER 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 load module. 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. If an error occurs while the called program is running, the ON EXCEPTION phrase is not performed.

Part 2. Enabling programs for international environments

Chapter 10. Processing data in an international environment

COBOL statements and national data	156
Intrinsic functions and national data	158
Unicode and the encoding of language characters	159
Using national data (Unicode) in COBOL	160
Defining national data items	160
Using national literals	161
Using national-character figurative constants	162
Defining national numeric data items	163
National groups	163
Using national groups	164
Using national groups as elementary items	165
Using national groups as group items	166
Storage of national data	167
Converting to or from national (Unicode) representation	167
Converting alphanumeric, DBCS, and integer data to national data (MOVE)	168
Converting alphanumeric and DBCS data to national data (NATIONAL-OF)	169
Converting national data to alphanumeric data (DISPLAY-OF)	169
Overriding the default code page	170
Example: converting to and from national data	170
Processing UTF-8 data	171
Processing Chinese GB 18030 data	171
Comparing national (UTF-16) data	172
Comparing two class national operands	172
Comparing class national and class numeric operands	173
Comparing national numeric and other numeric operands	174
Comparing national character-string and other character-string operands	174
Comparing national data and alphanumeric-group operands	174
Coding for use of DBCS support	175
Declaring DBCS data	175
Using DBCS literals	176
Comparing DBCS literals	176
Testing for valid DBCS characters	177
Processing alphanumeric data items that contain DBCS data	177
Controlling the DBCS collating sequence with a locale	186
Controlling the national collating sequence with a locale	187
Intrinsic functions that depend on collating sequence	188
Accessing the active locale and code-page values	188
Example: get and convert a code-page ID	189

Chapter 11. Setting the locale

The active locale	179
Specifying the code page with a locale	180
Using environment variables to specify a locale	181
Determination of the locale from system settings	182
Types of messages for which translations are available	182
Locales and code pages that are supported	183
Controlling the collating sequence with a locale	185
Controlling the alphanumeric collating sequence with a locale	185

Chapter 10. Processing data in an international environment

COBOL for Windows supports Unicode UTF-16 as national character data at run time. UTF-16 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 159

“National groups” on page 163

RELATED TASKS

“Using national data (Unicode) in COBOL” on page 160

“Converting to or from national (Unicode) representation” on page 167

“Processing UTF-8 data” on page 171
 “Processing Chinese GB 18030 data” on page 171
 “Comparing national (UTF-16) data” on page 172
 “Coding for use of DBCS support” on page 175
 Chapter 11, “Setting the locale,” on page 179

RELATED REFERENCES

“COBOL statements and national data”
 “Intrinsic functions and national data” on page 158
 “NCOLLSEQ” on page 252
 “NSYMBOL” on page 253
 Classes and categories of data (*COBOL for Windows Language Reference*)
 Data categories and PICTURE rules (*COBOL for Windows Language Reference*)
 MOVE statement (*COBOL for Windows Language Reference*)
 General relation conditions (*COBOL for Windows 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 18. COBOL statements and national data

COBOL statement	Can be national	Comment	For more information
ACCEPT	<i>identifier-1, identifier-2</i>	<i>identifier-1</i> 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 34
ADD	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) can be numeric-edited with USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 52
CALL	<i>identifier-2, identifier-3, identifier-4, identifier-5; literal-2, literal-3</i>		“Passing data” on page 467
COMPUTE	<i>identifier-1</i> can be numeric or numeric-edited with USAGE NATIONAL. <i>arithmetic-expression</i> can contain numeric items that have USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 52
COPY . . . REPLACING	<i>operand-1, operand-2</i> of the REPLACING phrase		Chapter 15, “Compiler-directing statements,” on page 273
DISPLAY	<i>identifier-1</i>	<i>identifier-1</i> is converted to the code page associated with the current locale.	“Displaying values on a screen or in a file (DISPLAY)” on page 35

Table 18. COBOL statements and national data (continued)

COBOL statement	Can be national	Comment	For more information
DIVIDE	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) and <i>identifier-4</i> (REMAINDER) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 52
INITIALIZE	<i>identifier-1</i> ; <i>identifier-2</i> or <i>literal-1</i> of the REPLACING phrase	If you specify REPLACING NATIONAL or REPLACING NATIONAL-EDITED, <i>identifier-2</i> or <i>literal-1</i> must be valid as a sending operand in a move to <i>identifier-1</i> .	"Examples: initializing data items" on page 28
INSPECT	All identifiers and literals. (<i>identifier-2</i> , the TALLYING integer data item, can have USAGE NATIONAL.)	If any of these (other than <i>identifier-2</i> , the TALLYING identifier) have USAGE NATIONAL, all must be national.	"Tallying and replacing data items (INSPECT)" on page 103
INVOKE	Method-name as <i>identifier-2</i> or <i>literal-1</i> ; <i>identifier-3</i> or <i>literal-2</i> in the BY VALUE phrase		"Invoking methods (INVOKE)" on page 407
MERGE	Merge keys, if you specify NCOLLSEQ(BIN)	The COLLATING SEQUENCE phrase does not apply.	"Setting sort or merge criteria" on page 139
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 32 "Assigning values to group data items (MOVE)" on page 32
MULTIPLY	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 52
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 77
SORT	Sort keys, if you specify NCOLLSEQ(BIN)	The COLLATING SEQUENCE phrase does not apply.	"Setting sort or merge criteria" on page 139
STRING	All identifiers and literals. (<i>identifier-4</i> , the POINTER integer data item, can have USAGE NATIONAL.)	If <i>identifier-3</i> , the receiving data item, is national, all identifiers and literals (other than <i>identifier-4</i> , the POINTER identifier) must be national.	"Joining data items (STRING)" on page 93

Table 18. 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. <i>identifier-3</i> (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 52
UNSTRING	All identifiers and literals. (<i>identifier-6</i> and <i>identifier-7</i> , the COUNT and TALLYING integer data items, respectively, can have USAGE NATIONAL.)	If <i>identifier-4</i> , 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	<i>identifier-1</i> (the generated XML document); <i>identifier-2</i> (the source field or fields)		Chapter 23, "Producing XML output," on page 373
XML PARSE	<i>identifier-1</i> (the XML document)	The XML-NTEXT special register contains national character document fragments during parsing.	Chapter 22, "Processing XML input," on page 349

RELATED TASKS

"Defining numeric data" on page 39

"Displaying numeric data" on page 41

"Using national data (Unicode) in COBOL" on page 160

"Comparing national (UTF-16) data" on page 172

RELATED REFERENCES

"NCOLLSEQ" on page 252

Classes and categories of data (COBOL for Windows Language Reference)

Intrinsic functions and national data

You can use arguments of class national with the intrinsic functions shown in the table below.

Table 19. Intrinsic functions and national character data

Intrinsic function	Function type	For more information
DISPLAY-OF	Alphanumeric	"Converting national data to alphanumeric data (DISPLAY-OF)" on page 169
LENGTH	Integer	"Finding the length of data items" on page 110
LOWER-CASE, UPPER-CASE	National	"Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)" on page 105
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 108
ORD-MAX, ORD-MIN	Integer	"Finding the largest or smallest data item" on page 108
REVERSE	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 39

“Using national data (Unicode) in COBOL” on page 160

RELATED REFERENCES

Arguments (*COBOL for Windows Language Reference*)

Classes and categories of data (*COBOL for Windows Language Reference*)

Unicode and the encoding of language characters

COBOL for Windows 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 Windows uses UTF-16 (CCSID 1202) 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'00nn', where X'nn' is the representation of the character in ASCII.

For example, the string 'ABC' is represented in UTF-16 as NX'004100420043'. 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: COBOL for Windows supports UTF-16 in little-endian format in national data. Enterprise COBOL for z/OS and COBOL for AIX support UTF-16 in big-endian format (UTF-16BE) in national data. If you are porting Unicode data that is encoded in UTF-16BE representation to COBOL for Windows from another platform, you must convert that data to UTF-16 in little-endian format to process the data as national data. With COBOL for Windows, you can perform such conversions by using the NATIONAL-OF intrinsic function.

RELATED TASKS

“Converting to or from national (Unicode) representation” on page 167

RELATED REFERENCES

“Storage of national data” on page 167

“Locales and code pages that are supported” on page 183

Character sets and code pages (*COBOL for Windows Language Reference*)

Using national data (Unicode) in COBOL

In COBOL for Windows, 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 159

“National groups” on page 163

RELATED TASKS

“Defining national data items”

“Using national literals” on page 161

“Using national-character figurative constants” on page 162

“Defining national numeric data items” on page 163

“Using national groups” on page 164

“Converting to or from national (Unicode) representation” on page 167

“Comparing national (UTF-16) data” on page 172

RELATED REFERENCES

“Storage of national data” on page 167

Classes and categories of data (*COBOL for Windows 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 41

RELATED REFERENCES

"NSYMBOL" on page 253

BLANK WHEN ZERO clause (*COBOL for Windows 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"00410042" 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 25

RELATED REFERENCES

“NSYMBOL” on page 253

“SOSI” on page 260

National literals (*COBOL for Windows 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 167

“Comparing national (UTF-16) data” on page 172

RELATED REFERENCES

“CHAR” on page 230

“NCOLLSEQ” on page 252

Figurative constants (*COBOL for Windows Language Reference*)

DISPLAY-OF (*COBOL for Windows 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 39

“Displaying numeric data” on page 41

RELATED REFERENCES

SIGN clause (*COBOL for Windows 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 32
 “Joining data items (STRING)” on page 93
 “Splitting data items (UNSTRING)” on page 95
 “Tallying and replacing data items (INSPECT)” on page 103
 “Using national groups”

RELATED REFERENCES

GROUP-USAGE clause (*COBOL for Windows Language Reference*)

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 163

RELATED TASKS

“Using national groups as elementary items”

“Using national groups as group items” on page 166

RELATED REFERENCES

GROUP-USAGE clause (*COBOL for Windows 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 06/23/2008 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 2008.
   . . .
   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 06232008 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 32

“Comparing national data and alphanumeric-group operands” on page 174

“Using national groups as group items”

RELATED REFERENCES

MOVE statement (*COBOL for Windows 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-quantN   Pic 99   Value 10.
   . . .
   Initialize Group-OneN Replacing Numeric Data By 15
```

Because only Trans-quantN in Group-OneN above is numeric, only Trans-quantN 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 20. 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.

Table 20. National group items that are processed with group semantics (continued)

Language feature	Uses of national group items	Comment
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 30
 “Initializing a table (INITIALIZE)” on page 67
 “Assigning values to elementary data items (MOVE)” on page 32
 “Assigning values to group data items (MOVE)” on page 32
 “Finding the length of data items” on page 110
 “Generating XML output” on page 373

RELATED REFERENCES

Qualification (*COBOL for Windows Language Reference*)
 RENAMEs clause (*COBOL for Windows Language Reference*)

Storage of national data

Use the table below to compare alphanumeric (DISPLAY), DBCS (DISPLAY-1), and Unicode (NATIONAL) encoding and to plan storage usage.

Table 21. 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 ¹	ASCII	ASCII DBCS	UTF-16LE
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: <ul style="list-style-type: none"> • COBOL characters A-Z, a-z, 0-9, space, + -*/= \$,,:“()><.’ • All characters that are converted from an EBCDIC, ASCII, or EUC code page 			

RELATED CONCEPTS

“Unicode and the encoding of language characters” on page 159

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 data to national data (MOVE)”

“Converting alphanumeric and DBCS data to national data (NATIONAL-OF)” on page 169

“Converting national data to alphanumeric data (DISPLAY-OF)” on page 169

“Overriding the default code page” on page 170

“Comparing national (UTF-16) data” on page 172

Chapter 11, “Setting the locale,” on page 179

Converting alphanumeric, DBCS, and integer data to national data (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'00410042', 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'0020') 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 32

“Assigning values to group data items (MOVE)” on page 32
“Displaying numeric data” on page 41
“Coding for use of DBCS support” on page 175

RELATED REFERENCES

MOVE statement (*COBOL for Windows Language Reference*)

Converting alphanumeric and DBCS data to national data (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 170

RELATED TASKS

“Processing UTF-8 data” on page 171
“Processing Chinese GB 18030 data” on page 171
“Processing alphanumeric data items that contain DBCS data” on page 177

RELATED REFERENCES

NATIONAL-OF (*COBOL for Windows Language Reference*)
Code page names (*COBOL for Windows Language Reference*)

Converting national data to alphanumeric data (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 170

RELATED CONCEPTS

“The active locale” on page 179

RELATED TASKS

“Processing UTF-8 data” on page 171
“Processing Chinese GB 18030 data” on page 171

RELATED REFERENCES

DISPLAY-OF (*COBOL for Windows Language Reference*)
Code page names (*COBOL for Windows 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, "IBM-1253")
      to Data-in-Unicode
* . . . process Data-in-Unicode here . . .
    Move function Display-of(Data-in-Unicode, "IBM-1253")
      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 IBM-1253 (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.IBM-1252`, the `MOVE` statement above converts `Data-in-Greek` to Unicode based on the code page IBM-1252 to UTF-16LE conversion. This conversion does not produce the expected results because `Data-in-Greek` is encoded in IBM-1253.

If you set the locale to `el_GR.IBM-1253` (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
```

Processing UTF-8 data

When you need to process UTF-8 data, first convert the 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.

You need to do two steps to convert ASCII or EBCDIC data to 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
```

Processing Chinese GB 18030 data

GB 18030 is a national-character standard specified by the government of the People’s Republic of China.

GB 18030 characters can be encoded in either UTF-16 or in code page CCSID 1392. Code page 1392 is an ASCII multibyte code page that uses 1, 2, or 4 bytes per character. A subset of the GB 18030 characters can be encoded in the Chinese ASCII code page, CCSID 1386, or in the Chinese EBCDIC code page, CCSID 1388.

COBOL for Windows does not have explicit support for GB 18030, but does support the processing of GB 18030 characters in several ways. You can:

- Use DBCS data items to process GB 18030 characters that are represented in CCSID 1386.
- Use national data items to define and process GB 18030 characters that are represented in UTF-16, CCSID 01202.
- Process data in any code page (including CCSID 1386) 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 code page 1386 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, 1386) to UnicodeString
* Process data in Unicode
    Move function Display-of(UnicodeString, 1386) to Chinese-GB18030-String

```

RELATED TASKS

“Converting to or from national (Unicode) representation” on page 167
“Coding for use of DBCS support” on page 175

RELATED REFERENCES

“Storage of national data” on page 167

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

| The following sections provide an overview about comparing national data to
| other data items. For full details, see the related references.

RELATED TASKS

“Comparing two class national operands”
“Comparing class national and class numeric operands” on page 173
“Comparing national numeric and other numeric operands” on page 174
“Comparing national character-string and other character-string operands” on page 174
“Comparing national data and alphanumeric-group operands” on page 174

RELATED REFERENCES

Relation conditions (*COBOL for Windows Language Reference*)
General relation conditions (*COBOL for Windows Language Reference*)
National comparisons (*COBOL for Windows Language Reference*)
Group comparisons (*COBOL for Windows 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'0020') 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 163

RELATED TASKS

"Using national groups" on page 164

RELATED REFERENCES

"NCOLLSEQ" on page 252

National comparisons (*COBOL for Windows 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 Windows 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 163

RELATED REFERENCES

General relation conditions (*COBOL for Windows Language Reference*)

Comparing national character-string 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 162

“Comparing DBCS literals” on page 176

RELATED REFERENCES

National comparisons (*COBOL for Windows 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.

Coding for use of DBCS support

IBM COBOL for Windows 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

RELATED TASKS

"Declaring DBCS data"

"Using DBCS literals" on page 176

"Testing for valid DBCS characters" on page 177

"Processing alphanumeric data items that contain DBCS data" on page 177

Chapter 11, "Setting the locale," on page 179

"Controlling the collating sequence with a locale" on page 185

RELATED REFERENCES

"SOSI" on page 260

Declaring DBCS data

Use the PICTURE and USAGE clauses to declare 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 declaration 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 179

RELATED REFERENCES

"Locales and code pages that are supported" on page 183

"NSYMBOL" on page 253

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 single quotation marks (') 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"

"Using figurative constants" on page 26

RELATED REFERENCES

"NSYMBOL" on page 253

"QUOTE/APOST" on page 257

"SOSI" on page 260

DBCS literals (*COBOL for Windows 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 172

Chapter 11, "Setting the locale," on page 179

RELATED REFERENCES

"COLLSEQ" on page 233

DBCS literals (*COBOL for Windows Language Reference*)

DBCS comparisons (*COBOL for Windows 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 zSeries 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 86

RELATED REFERENCES

Class condition (*COBOL for Windows 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 167

Chapter 11. Setting the locale

You can write applications to reflect the cultural conventions of the locale 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 Windows, 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"

RELATED TASKS

"Specifying the code page with a locale" on page 180

"Using environment variables to specify a locale" on page 181

"Controlling the collating sequence with a locale" on page 185

"Accessing the active locale and code-page values" on page 188

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 Standard COBOL 85 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 Windows 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 182

RELATED TASKS

“Specifying the code page with a locale”

“Using environment variables to specify a locale” on page 181

“Controlling the collating sequence with a locale” on page 185

RELATED REFERENCES

“Types of messages for which translations are available” on page 182

Specifying the code page with a locale

In a source program, you can use the characters that are represented in a supported code page in COBOL names, literals, and comments. (See the *COBOL for Windows Language Reference* information cited in the related references below for full details about the formation of user-defined words.)

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
- Which locale is active

COBOL for Windows chooses between ASCII and EBCDIC code pages as follows:

- Data items that are described with the NATIVE phrase in the USAGE clause are encoded in an ASCII code page.
- Data items that are described without the NATIVE phrase in the USAGE clause are encoded according to the CHAR compiler option, as follows:
 - CHAR(NATIVE): in ASCII
 - CHAR(EBCDIC): in EBCDIC

COBOL determines the appropriate code page as follows:

ASCII From the active locale at run time.

EBCDIC

From the EBCDIC_CODEPAGE environment variable, if it is set. Otherwise, COBOL uses the default EBCDIC code page from the current locale setting.

The code page for USAGE NATIONAL data items and national literals is UTF-16, CCSID 1202.

RELATED TASKS

“Using environment variables to specify a locale”

RELATED REFERENCES

“Locales and code pages that are supported” on page 183

“Runtime environment variables” on page 196

“CHAR” on page 230

COBOL words with single-byte characters (*COBOL for Windows Language Reference*)

User-defined words with multibyte characters (*COBOL for Windows 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 on the 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.

Use the following format to set the locale environment variables (*.codepageID* is optional):

```
SET 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 Windows uses the POSIX-defined locale conventions.

For example, to set the locale to Canadian French encoded in IBM-863, issue this command in the command window from which you compile and run a COBOL application:

```
SET LC_ALL=fr_CA.IBM-863
```

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”

RELATED TASKS

“Specifying the code page with a locale” on page 180

RELATED REFERENCES

“Locales and code pages that are supported” on page 183

“Runtime environment variables” on page 196

Determination of the locale from system settings

If COBOL for Windows cannot determine the value of an applicable locale category from the environment variables, it uses default settings.

The default locale is set to en_US.ibm-1252.

When the language and country codes are determined from environment variables, but the code page is not, COBOL for Windows uses the Windows OEM code page from the Windows regional settings. The code page and the language and country-code combination must be compatible.

Windows APIs: The COBOL run time does not change the Windows Regional Options settings or the process locale that are used by native Windows APIs.

RELATED TASKS

“Specifying the code page with a locale” on page 180

“Using environment variables to specify a locale” on page 181

“Accessing the active locale and code-page values” on page 188

“Setting environment variables” on page 193

RELATED REFERENCES

“Locales and code pages that are supported” on page 183

“Runtime environment variables” on page 196

Types of messages for which translations are available

The following messages are enabled for national language support: compiler, runtime, and debugger user interface 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 179

RELATED TASKS

“Using environment variables to specify a locale” on page 181

Locales and code pages that are supported

The following table shows the locales that COBOL for Windows supports and the code pages that are valid for each locale.

Table 22. Supported locales and code pages

Locale name ¹	Language ²	Country or region ³	ASCII code pages ⁴	EBCDIC code pages ⁵	Language group
ar_AA	Arabic	Arabic countries	IBM-864, IBM-1256	IBM-16804, IBM-420	Arabic
be_BY	Byelorussian	Belarus	IBM-866, IBM-1251	IBM-1025, IBM-1154	Latin 5
bg_BG	Bulgarian	Bulgaria	IBM-855, IBM-1251	IBM-1025, IBM-1154	Latin 5
ca_ES	Catalan	Spain	IBM-850, IBM-5348, IBM-1252	IBM-285, IBM-1145	Latin 1
cs_CZ	Czech	Czech Republic	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
da_DK	Danish	Denmark	IBM-437, IBM-850, IBM-1252	IBM-277, IBM-1142	Latin 1
de_CH	German	Switzerland	IBM-437, IBM-850, IBM-1252	IBM-500, IBM-1148	Latin 1
de_DE	German	Germany	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-273, IBM-1141	Latin 1
el_GR	Greek	Greece	IBM-1253	IBM-4971, IBM-875	Greek
en_AU	English	Australia	IBM-437, IBM-1252	IBM-037, IBM-1140	Latin 1
en_BE	English	Belgium	IBM-850, IBM-5348, IBM-1252	IBM-500, IBM-1148	Latin 1
en_GB	English	United Kingdom	IBM-437, IBM-850, IBM-1252	IBM-037, IBM-1140	Latin 1
en_JP	English	Japan	IBM-437, IBM-850, IBM-1252	IBM-037, IBM-1140	Latin 1
en_US	English	United States	IBM-437, IBM-850, IBM-1252	IBM-037, IBM-1140	Latin 1
en_ZA	English	South Africa	IBM-437, IBM-1252	IBM-037, IBM-1140	Latin 1
es_ES	Spanish	Spain	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-284, IBM-1145	Latin 1
et_EE	Estonian	Estonia	IBM-1257	IBM-1157	Estonian
fi_FI	Finnish	Finland	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-278, IBM-1143	Latin 1
fr_BE	French	Belgium	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-297, IBM-1148	Latin 1
fr_CA	French	Canada	IBM-863, IBM-850, IBM-1252	IBM-037, IBM-1140	Latin 1
fr_CH	French	Switzerland	IBM-437, IBM-850, IBM-1252	IBM-500, IBM-1148	Latin 1
fr_FR	French	France	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-297, IBM-1148	Latin 1
hr_HR	Croatian	Croatia	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
hu_HU	Hungarian	Hungary	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
is_IS	Icelandic	Iceland	IBM-861, IBM-850, IBM-1252	IBM-871, IBM-1149	Latin 1
it_CH	Italian	Switzerland	IBM-850, IBM-1252	IBM-500, IBM-1148	Latin 1

Table 22. Supported locales and code pages (continued)

Locale name ¹	Language ²	Country or region ³	ASCII code pages ⁴	EBCDIC code pages ⁵	Language group
it_IT	Italian	Italy	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-280, IBM-1144	Latin 1
iw_IL	Hebrew	Israel	IBM-862, IBM-1255	IBM-12712, IBM-424	Hebrew
ja_JP	Japanese	Japan	IBM-943	IBM-930, IBM-939, IBM-1390, IBM-1399	Ideographic languages
ko_KR	Korean	Korea, Republic of	IBM-1363	IBM-933, IBM-1364	Ideographic languages
lt_LT	Lithuanian	Lithuania	IBM-1257	IBM-1112, IBM-1156	Lithuanian
lv_LV	Latvian	Latvia	IBM-1257	IBM-1112, IBM-1156	Latvian
mk_MK	Macedonian	Macedonia,	IBM-855, IBM-1251	IBM-1025, IBM-1154	Latin 5
nl_BE	Dutch	Belgium	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-500, IBM-1148	Latin 1
nl_NL	Dutch	Netherlands	IBM-437, IBM-850, IBM-5348, IBM-1252	IBM-037, IBM-1140	Latin 1
no_NO	Norwegian	Norway	IBM-437, IBM-850, IBM-1252	IBM-277, IBM-1142	Latin 1
pl_PL	Polish	Poland	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
pt_BR	Portuguese	Brazil	IBM-850, IBM-1252	IBM-037, IBM-1140	Latin 1
pt_PT	Portuguese	Portugal	IBM-860, IBM-850, IBM-5348, IBM-1252	IBM-037, IBM-1140	Latin 1
ro_RO	Romanian	Romania	IBM-852, IBM-850, IBM-1250	IBM-870, IBM-1153	Latin 2
ru_RU	Russian	Russian federation	IBM-866, IBM-1251	IBM-1025, IBM-1154	Latin 5
sh_SP	Serbian (Latin)	Serbia	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
sk_SK	Slovak	Slovakia	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
sl_SI	Slovenian	Slovenia	IBM-852, IBM-1250	IBM-870, IBM-1153	Latin 2
sq_AL	Albanian	Albania	IBM-850, IBM-1252	IBM-500, IBM-1148	Latin 1
sv_SE	Swedish	Sweden	IBM-437, IBM-850, IBM-1252	IBM-278, IBM-1143	Latin 1
th_TH	Thai	Thailand	IBM-874	IBM-9030	Thai
tr_TR	Turkish	Turkey	IBM-857, IBM-1254	IBM-1026, IBM-1155	Turkish
uk_UA	Ukrainian	Ukraine	IBM-866, IBM-1251	IBM-1123, IBM-1154	Latin 5
zh_CN	Chinese (simplified)	China	IBM-1386	IBM-1388	Ideographic languages
zh_TW	Chinese (traditional)	Taiwan	IBM-950	IBM-1371, IBM-937	Ideographic languages

1. Shows the valid combinations of ISO language code and ISO country code (*language_COUNTRY*) that are supported.
2. Shows the associated language.
3. Shows the associated country or region.
4. Shows the ASCII code pages that are valid as the code-page ID for the locale that has the corresponding *language_COUNTRY* value.
5. Shows the EBCDIC 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 by default as the EBCDIC code page for the corresponding locale.

RELATED TASKS

“Specifying the code page with a locale” on page 180

“Using environment variables to specify a locale” on page 181

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 with 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.

When 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

When the `COLLSEQ(LOCALE)` compiler option is in effect, the collating sequence for alphanumeric keys in `SORT` or `MERGE` statements is always based on the locale at run time.

RELATED TASKS

“Specifying the collating sequence” on page 8

“Specifying the code page with a locale” on page 180

“Using environment variables to specify a locale” on page 181

“Controlling the alphanumeric collating sequence with a locale”

“Controlling the DBCS collating sequence with a locale” on page 186

“Controlling the national collating sequence with a locale” on page 187

“Accessing the active locale and code-page values” on page 188

“Setting sort or merge criteria” on page 139

RELATED REFERENCES

“Locales and code pages that are supported” on page 183

“`COLLSEQ`” on page 233

“`NCOLLSEQ`” on page 252

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 8
- “Controlling the collating sequence with a locale” on page 185
- “Controlling the DBCS collating sequence with a locale”
- “Controlling the national collating sequence with a locale” on page 187
- “Setting sort or merge criteria” on page 139

RELATED REFERENCES

- “COLLSEQ” on page 233
- Classes and categories of data (*COBOL for Windows Language Reference*)
- Alphanumeric comparisons (*COBOL for Windows 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 8

“Using DBCS literals” on page 176

“Controlling the collating sequence with a locale” on page 185

“Controlling the alphanumeric collating sequence with a locale” on page 185

“Controlling the national collating sequence with a locale”

RELATED REFERENCES

“COLLSEQ” on page 233

Classes and categories of data (*COBOL for Windows Language Reference*)

Alphanumeric comparisons (*COBOL for Windows Language Reference*)

DBCS comparisons (*COBOL for Windows Language Reference*)

Group comparisons (*COBOL for Windows 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 172

“Controlling the collating sequence with a locale” on page 185

“Controlling the DBCS collating sequence with a locale” on page 186

“Setting sort or merge criteria” on page 139

RELATED REFERENCES

“NCOLLSEQ” on page 252

Classes and categories of data (*COBOL for Windows Language Reference*)

National comparisons (*COBOL for Windows 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 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 23. 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 8

"Comparing national (UTF-16) data" on page 172

"Controlling the collating sequence with a locale" on page 185

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; <i>output</i> is set to -1. |
| -1 | The code-page ID is not a valid code page; <i>output</i> is set to -1. |

To call these services, you must use the SYSTEM call interface convention and the PGMNAME(MIXED) and NODYNAM compiler options.

“Example: get and convert a code-page ID”

RELATED TASKS

Chapter 11, “Setting the locale,” on page 179

RELATED REFERENCES

“CALLINT” on page 229

“DYNAM” on page 238

“PGMNAME” on page 255

Chapter 15, “Compiler-directing statements,” on page 273

“SYSTEM” on page 461

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.

```
cb1 pgmname(1m)
    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 181

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

Chapter 12. Compiling, linking, and running

programs	193
Setting environment variables	193
Setting environment variables for COBOL for Windows	194
Compiler environment variables	195
Linker environment variables	196
Runtime environment variables	196
TZ environment parameter variables	200
Compiling programs	201
Compiling from the command line	201
Examples: using cob2 for compiling	202
Compiling using batch files or command files	203
Specifying compiler options with the PROCESS (CBL) statement	203
Correcting errors in your source program	203
Severity codes for compiler error messages	204
Generating a list of compiler error messages	204
Messages and listings for compiler-detected errors	205
Format of compiler error messages	205
cob2 options	206
Options that apply to compiling	206
Options that apply to linking	207
Options that apply to both compiling and linking	208
Linking programs	208
File names and extensions supported by cob2	209
Specifying linker options	210
Linking through the compiler	210
Linking from the command line	211
Examples: using cob2 for linking	211
Example: overriding linker options	212
Linker input and output files	212
Linker search rules	212
Example: linker search rules	213
File-name defaults	213
Correcting errors in linking	213
Linker return codes	214
Linker errors in program-names	214
Using NMAKE to update projects	215
Running NMAKE on the command line	215
Running NMAKE with a command file	216
Defining description files for NMAKE	216
Running programs	217
Redistributing COBOL for Windows DLLs	217

Chapter 13. Compiling, linking, and running OO applications

Compiling OO applications	219
Preparing OO applications	220
Example: compiling and linking a COBOL class definition	221
Running OO applications	221

Running OO applications that start with a main method	222
Running OO applications that start with a COBOL program	222

Chapter 14. Compiler options

Conflicting compiler options	226
ADATA	227
ARITH	228
BINARY	229
CALLINT	229
CHAR	230
CICS	232
COLLSEQ	233
COMPILE	235
CURRENCY	236
DATEPROC	237
DIAGTRUNC	238
DYNAM	238
ENTRYINT	239
EXIT	240
Character string formats	241
User-exit work area	241
Linkage conventions	242
Parameter list for exit modules	242
Using INEXIT	243
Using LIBEXIT	243
Using PRTEXIT	244
Using ADEXIT	244
FLAG	245
FLAGSTD	246
FLOAT	247
LIB	248
LINECOUNT	249
LIST	249
LSTFILE	250
MAP	250
MDECK	251
NCOLLSEQ	252
NSYMBOL	253
NUMBER	253
OPTIMIZE	254
PGMNAME	255
PGMNAME(UPPER)	256
PGMNAME(MIXED)	256
PROBE	256
QUOTE/APOST	257
SEPOBJ	258
Batch compilation	258
SEQUENCE	259
SIZE	260
SOSI	260
SOURCE	261
SPACE	262

SQL	262
SSRANGE	263
TERMINAL	264
TEST	264
THREAD	265
TRUNC	266
TRUNC example 1	267
TRUNC example 2	268
VBREF	269
WSCLEAR	269
XREF	269
YEARWINDOW	271
ZWB	271

Chapter 15. Compiler-directing statements 273

Chapter 16. Linker options 277

/?	278
/ALIGNADDR	278
/ALIGNFILE	279
/BASE	279
/CODE	280
/DATA	280
/DBGPACK, /NODBGPACK	281
/DEBUG, /NODEBUG	281
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH	281
/DLL	282
/ENTRY	282
/EXECUTABLE	283
/EXTDICTIONARY, /NOEXTDICTIONARY	283
/FIXED, /NOFIXED	284
/FORCE, /NOFORCE	284
/HEAP	285
/HELP	285
/INCLUDE	285
/INFORMATION, /NOINFORMATION	285
/LINENUMBERS, /NOLINENUMBERS	286
/LOGO, /NOLOGO	286
/MAP, /NOMAP	287
/OUT	287
/PMTYPE	288
/SECTION	288
/SEGMENTS	289
/STACK	290
/STUB	290
/SUBSYSTEM	291
/VERBOSE, /NOVERBOSE	291
/VERSION	292

Chapter 17. Runtime options 293

CHECK	293
DEBUG	294
ERRCOUNT	294
FILESYS	294
TRAP	295
UPSI	296

Chapter 18. Debugging 297

Debugging with source language	297
Tracing program logic	297

Finding and handling input-output errors	298
Validating data	299
Finding uninitialized data	299
Generating information about procedures	299
Example: USE FOR DEBUGGING	300
Debugging using compiler options	301
Finding coding errors	302
Finding line sequence problems	302
Checking for valid ranges	302
Selecting the level of error to be diagnosed	303
Example: embedded messages	304
Finding program entity definitions and references	305
Listing data items	306
Using the debugger	306
Getting listings	307
Example: short listing	308
Example: SOURCE and NUMBER output	310
Example: MAP output	310
Example: embedded map summary	311
Terms and symbols used in MAP output	312
Example: nested program map	313
Example: XREF output - data-name cross-references	314
Example: XREF output - program-name cross-references	315
Example: embedded cross-reference	315
Example: VBREF compiler output	316
Debugging user exits	317
Debugging assembler routines	318

Chapter 12. Compiling, linking, and running programs

The following sections explain how to set environment variables, compile, link, use NMAKE to update projects, run, correct errors, and redistribute DLLs.

The COBOL for Windows compiler and runtime environment support the high subset of Standard COBOL 85 functions, as do other IBM COBOL products. Although the IBM COBOL language is almost the same across platforms, some minor differences exist between Enterprise COBOL for z/OS and COBOL for Windows.

RELATED TASKS

“Setting environment variables”

“Compiling programs” on page 201

“Correcting errors in your source program” on page 203

“Linking programs” on page 208

“Correcting errors in linking” on page 213

“Using NMAKE to update projects” on page 215

“Running programs” on page 217

“Redistributing COBOL for Windows DLLs” on page 217

RELATED REFERENCES

Appendix A, “Summary of differences with host COBOL,” on page 561

Setting environment variables

You use environment variables to set values that programs need. Specify the value of an environment variable by using the SET command or by using the System Properties window. 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 the 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.

In general, you set environment variables in either of two ways:

- By using the SET command. The values of the variables apply only to programs that you run from the window where you issue the SET command. You can use the SET command at the command prompt or in a command file (.cmd or .bat).
- By using System Properties. If you add or change environment variables in the User variables window, they are in effect for the current Windows user ID. If you add or change environment variables in the System variables window, they are in effect for the system. You must open a new command window to make the values of any changed environment variables available to processes.

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 semicolons. For example, issuing the following command in a window sets the COBPATH environment variable to include two directories when you run programs from that window:

```
SET COBPATH=d:\cobdev\d11;d:\dev\d11
```

Paths that are defined by environment variables are evaluated in order, from the first path to the last. If multiple files with the same name are defined in the path of an environment variable, the *first* located copy of the file is used.

The value that you assign to an environment variable by using the SET command can include other environment variables or the variable itself. For example, assuming that COBPATH has already been set, you can add a directory to the value of COBPATH by issuing the following command:

```
SET COBPATH=%COBPATH%;d:\myown\d11;
```

RELATED TASKS

"Setting environment variables for COBOL for Windows"

RELATED REFERENCES

"Compiler environment variables" on page 195

"Linker environment variables" on page 196

"Runtime environment variables" on page 196

Setting environment variables for COBOL for Windows

Rational® Developer for System z™ provides a command file that you can use to set the environment variables for accessing the COBOL for Windows compiler and runtime libraries.

The name of the command file is `setenvRDZvr.bat`, where *v* is the version number and *r* is the release number of Rational Developer for System z. Thus for example for Rational Developer for System z, Version 7.5, the name of the command file is `setenvRDZ75.bat`.

To set up the COBOL for Windows environment in the current command window, you can issue the following command, which uses the fully qualified path to the command file:

```
"%RDZvrINSTDIR%\bin\setenvRDZvr"
```

Substitute the version number for *v* and the release number for *r* in the command above. Thus for example for Rational Developer for System z, Version 7.5, you would issue the command:

```
"%RDZ75INSTDIR%\bin\setenvRDZ75"
```

Alternatively, you can first set the PATH environment variable to access the command file, and then invoke the command file by using only the file name:

```
SET PATH=%PATH%;%RDZvrINSTDIR%\bin
setenvRDZvr
```

If you want to be able to invoke the `setenvRDZvr` command from any command window that you open, without having to specify the path to the file, append the following string to your PATH system environment variable in the System Properties window:

```
;%RDZvrINSTDIR%\bin
```

Instead of setting the environment variables for COBOL by invoking a command file, you can launch the COBOL environment by clicking **Start -> All Programs -> install_name -> IBM Rational Developer for System z ->**

Command Environment for Local Compilers where *install_name* is a name that you assigned during installation (by default, **IBM Software Delivery Platform**).

Compiler environment variables

The COBOL compiler uses several environment variables.

COBCPYEXT

Specifies the file extensions to use in searches for copybooks when the COPY *name* statement does not indicate a file extension. Specify one or more three-character file extensions with or without leading periods. Separate multiple file extensions with a space or comma.

If COBCPYEXT is not defined, the following extensions are searched: .CPY, .CBL, and .COB (or their lowercase or mixed-case equivalents).

COBLSTDIR

Specifies the directory into which the compiler listing file is written. Specify any valid drive and path. To indicate an absolute path, specify a leading drive letter or backslash. Otherwise, the path is relative to the current directory. A trailing backslash 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. For example:

```
SET COBOPT=TRUNC(OPT) TERMINAL
```

Default values apply to individual compiler options.

COBPATH

Specifies paths to be used for locating user-defined compiler exit programs that the EXIT compiler option has identified.

DB2DBDFT

Specifies the database for compiling programs that have embedded SQL statements.

DB2PATH

Specifies the directory in which DB2 is installed.

LANG

See the related reference about runtime environment variables below for details.

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 to locate the copybook. For example:

```
SET MYLIB=C:\CPYFILES\COBCOPY
```

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 -lxxx 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 below about compiler-directing statements.

NLSPATH

See the related reference about runtime environment variables below for details.

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.

TEMPMEM

Specifies whether compiler work files are stored in memory files or on disk. Using memory files (TEMPMEM=ON) can significantly reduce compilation time.

In some cases with very large source programs, insufficient memory errors can occur. In this event, set TEMPMEM to null.

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 semicolon (;).

For more details, see the documentation of the COPY statement in the related reference below about compiler-directing statements.

RELATED CONCEPTS

"DB2 coprocessor" on page 321

RELATED TASKS

"Using SQL INCLUDE with the DB2 coprocessor" on page 322

RELATED REFERENCES

"Runtime environment variables"

"cob2 options" on page 206

Chapter 14, "Compiler options," on page 225

Chapter 15, "Compiler-directing statements," on page 273

Linker environment variables

The linker uses the LIB environment variable, which specifies directories to be searched for object (.OBJ), library (.LIB), or module definition (.DEF) files.

RELATED TASKS

"Linking programs" on page 208

RELATED REFERENCES

"Linker search rules" on page 212

Runtime environment variables

The COBOL runtime library uses the following environment variables. Case does not matter.

assignment-name

Any COBOL file that you want to specify in an ASSIGN clause. This use of *assignment-name* follows the rules for a COBOL word. For example:

```
SET OUTPUTFILE=d:\january\results.car
```

You must set all assignment-names.

If you make an assignment to a user-defined word that is not set as an environment variable, the assignment is made to a file that has the literal name of the user-defined word (OUTPUTFILE in the example below). If the assignment is valid, this file is written to the current directory.

After you set an assignment-name, you can use that environment variable as a COBOL user-defined word in an ASSIGN clause.

Based on the previous SET statement, a COBOL source program could include the following SELECT and ASSIGN clause:

```
SELECT CARPOOL ASSIGN TO OUTPUTFILE
```

Because OUTPUTFILE is defined in an environment variable, the statement above results in data being written to the file d:\january\results.car.

You can use ASSIGN to specify a file stored in an alternate file system such as the standard language file system (STL), record sequential delimited file system (RSD), or Btrieve file system.

CLASSPATH

Specifies directory paths of Java™ .class files required for an OO application.

COBJVMINITOPTIONS

Specifies Java virtual machine (JVM) options used when COBOL initializes a JVM.

COBMSGSG

Specifies the name of a file to which runtime error messages will be written.

To capture runtime error messages in a file, use the SET command to set COBMSGSG to a file-name. If your program has a runtime error that terminates the application, the file that COBMSGSG is set to will contain an error message that indicates the reason for termination.

If COBMSGSG is not set, error messages are written to the terminal.

COBPAT

Specifies the directory paths to be used by the COBOL run time to locate dynamically accessed programs such as .DLL (dynamic link library) files.

This variable must be set to run programs that require dynamic loading. For example:

```
SET COBPAT=C:\pgmpath\pgmd11
```

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:

```
SET COBRTOPT=TRAP=ON,errcount  
SET COBRTOPT=trap(on):ERRCOUNT
```

The defaults for individual runtime options apply. For details, see the related reference below 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:

```
SET 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 below 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.

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:

```
SET 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.

LOCPATH

Specifies the search path for the locale information database. The path is a semicolon-separated list of directory names.

LOCPATH is used for any operations that reference the locale, including locale-based comparisons of alphanumeric data items.

NLSPATH

Specifies the full path name of message catalogs and help files. NLSPATH must always be set, and is given an initial value during installation.

When you set NLSPATH, be sure to add to NLSPATH rather than replace it. Other programs might use this environment variable. For example:

```
SET NLSPATH=C:\cobolpath\MESSAGES\%L\%N;%NLSPATH%
```

%L and %N must be uppercase. %L is substituted by the value specified by the LANG environment variable. %N is substituted by the message catalog name used by COBOL.

Messages in the following languages are included with the product:

cs_CZ Czech

de_DE	German
en_US	English
es_ES	Spanish
fr_FR	French
hu_HU	Hungarian
ja_JP	Japanese
ko_KR	Korean
pl_PL	Polish
pt_BR	Brazilian Portuguese
ru_RU	Russian
zh_CN	Simplified Chinese (People's Republic of China)
zh_TW	Traditional Chinese (Taiwan)

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.IBM-943`. 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 IBM-943 (ASCII Japanese 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 library also uses `NLSPATH` to access the message catalog. If `NLSPATH` is not set correctly, runtime messages appear in an abbreviated form.

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. Set them to files, not existing directory names.

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.

For example, the following command defines `CONSOLE`:

```
SET CONSOLE=c:\mypath\terminal.txt
```

CONSOLE could then be used in conjunction with the following source code:

```
SPECIAL-NAMES.  
  CONSOLE IS terminal  
  .  
  .  
  .  
  DISPLAY 'Hello World' UPON terminal
```

TMP Specifies the location of temporary work files (if needed) for SORT and MERGE functions. The defaults vary and are set by the sort utility installation program.

For example:

```
SET TMP=c:\shared\temp
```

TZ Describes the time-zone information to be used by the locale. TZ has the following format:

```
SET TZ=SSS[+|-]nDDD[,sm,sw,sd,st,em,ew,ed,et,shift]
```

The defaults depend on the current locale.

When TZ is not present, the default is EST5EDT, the default locale value. When 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:

```
SET TZ=CST6CDT
```

The preceding statement sets the standard time zone to CST, sets the daylight savings 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 savings time.

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 181

“Identifying files” on page 113

RELATED REFERENCES

“Locales and code pages that are supported” on page 183

“TZ environment parameter variables”

“CHAR” on page 230

Chapter 17, “Runtime options,” on page 293

TZ environment parameter variables

The values for the TZ variable are defined below.

Table 24. 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

Table 24. TZ environment parameter variables (continued)

Variable	Description	Default value
<i>n</i>	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
<i>DDD</i>	Daylight saving time (DST) zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EDT
<i>sm</i>	Starting month (1 to 12) of DST	4
<i>sw</i>	Starting week (-4 to 4) of DST	1
<i>sd</i>	Starting day of DST: 0 to 6 if <i>sw</i> is not zero; 1 to 31 if <i>sw</i> is zero	0
<i>st</i>	Starting time (in seconds) of DST	3600
<i>em</i>	Ending month (1 to 12) of DST	10
<i>ew</i>	Ending week (-4 to 4) of DST	-1
<i>ed</i>	Ending day of DST: 0 to 6 if <i>ew</i> is not zero; 1 to 31 if <i>ew</i> is zero	0
<i>et</i>	Ending time (in seconds) of DST	7200
<i>shift</i>	Amount of time change (in seconds)	3600

Compiling programs

There are several ways you can specify the options used to compile your programs.

You can:

- Set the COBOPT environment variable from the command line or in the Windows System Properties window.
- Specify compiler environment variables and cob2 options in a batch file or on the command line.
- Use PROCESS (CBL) or *CONTROL statements. An option that you specify by using PROCESS overrides every other option specification.

RELATED TASKS

“Compiling from the command line”

“Specifying compiler options with the PROCESS (CBL) statement” on page 203
Chapter 26, “Porting applications between platforms,” on page 445

RELATED REFERENCES

“Compiler environment variables” on page 195

“cob2 options” on page 206

Chapter 14, “Compiler options,” on page 225

Appendix A, “Summary of differences with host COBOL,” on page 561

Compiling from the command line

Issue the command cob2 to compile a COBOL program from the command line. (The cob2 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 -v -qflag(w)
cob2 filea.cbl -qflag(w) -g -v fileb.cbl
```

cob2 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. You do not need to capitalize cob2 and its options.

cob2 passes files with certain extensions (such as .cbl) to the compiler and passes all other files to the linker.

The default location for compiler input and output is the current directory.

“Examples: using cob2 for compiling”

RELATED TASKS

“Linking programs” on page 208

RELATED REFERENCES

“File names and extensions supported by cob2” on page 209

“cob2 options” on page 206

Chapter 14, “Compiler options,” on page 225

Examples: using cob2 for compiling

The following examples show the output produced for various cob2 specifications.

Table 25. Output from the cob2 command

To compile:	Enter:	These files are produced:
alpha.cbl	cob2 -c alpha.cbl	alpha.obj and alpha.lst
alpha.cbl and beta.cbl	cob2 -c alpha.cbl c:\mydir\beta.cbl	alpha.obj, beta.obj; alpha.lst, beta.lst in the current directory
alpha.cbl with the LIST and ADATA options	cob2 -qlist,adata alpha.cbl	alpha.asm, alpha.obj, alpha.lst, alpha.adt, and alpha.exe

RELATED TASKS

“Compiling from the command line” on page 201

RELATED REFERENCES

“ADATA” on page 227

“LIST” on page 249

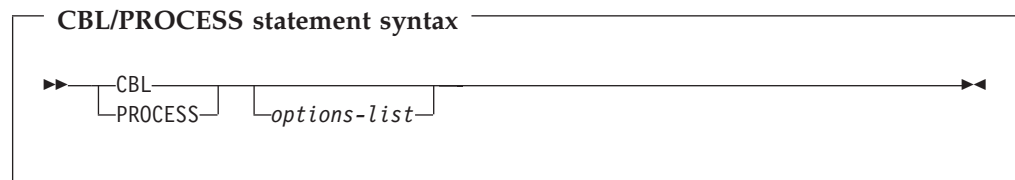
Compiling using batch files or command files

You can use a batch file or command file to automate cob2 tasks.

To prevent invalid syntax from being passed to the cob2 command, however, do not use any blanks in the option string unless you enclose the string in quotation marks ("").

Specifying compiler options with the PROCESS (CBL) statement

You can code compiler options in the PROCESS statement in COBOL programs. Code it before the IDENTIFICATION DIVISION header and before any comment lines or compiler-directing statements.



You can start the PROCESS statement in column 1 through 66 if you don't code a sequence field. A sequence field is allowed in columns 1 through 6; if used, the sequence field must contain six characters, and the first character must be numeric. When used with a sequence field, PROCESS can start in column 8 through 66.

You can use CBL as a synonym for PROCESS. CBL can start in column 1 through 70. When used with a sequence field, CBL can start in column 8 through 70.

Use one or more blanks to separate PROCESS 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 use more than one PROCESS statement. If you do so, the PROCESS statements must follow each another with no intervening statements. You cannot continue options across multiple PROCESS statements.

Your programming organization can inhibit the use of PROCESS statements by using the default options module of the COBOL compiler. When PROCESS statements are found in a COBOL program but are not allowed by the organization, the COBOL compiler generates error diagnostics.

RELATED REFERENCES

CBL (PROCESS) statement (*COBOL for Windows Language Reference*)

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 necessary to fix all of them. 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 error 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 errors, 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 any case, the problem must be resolved, because the compiler is forced to end early and does not produce complete object code or 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 (problems such as missing data-set 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 error messages”

“Linking programs” on page 208

RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 205

Severity codes for compiler error messages

Errors that the compiler can detect fall into five categories of severity.

Table 26. Severity codes for compiler error messages

Level 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.

Generating a list of compiler error messages

You can generate a complete listing of compiler diagnostic messages with their explanations by compiling a program that has the program-name ERRMSG.

You can code just the PROGRAM-ID paragraph, as shown below. Omit the rest of the program.

Identification Division.
Program-ID. ErrMsg.

RELATED REFERENCES

“Messages and listings for compiler-detected errors”

“Format of compiler error messages”

Messages and listings for compiler-detected errors

As the compiler processes your source program, it checks for COBOL language errors. For each error found, the compiler issues a message. 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 extension .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 the following information:

- Nature of the error
- Compiler phase that detected the error
- Severity level of the error

Wherever possible, the message provides specific instructions for correcting the 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 found in your program (ordered by line number) are displayed near the end of the listing for each program.

A summary of all errors found during compilation is displayed near the bottom of the listing.

RELATED TASKS

“Correcting errors in your source program” on page 203

“Generating a list of compiler error messages” on page 204

RELATED REFERENCES

“Format of compiler error messages”

“Severity codes for compiler error messages” on page 204

“FLAG” on page 245

Format of compiler error 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 IGYppxxx-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, these are your original source program numbers. If you specified NONUMBER, the numbers are those generated by the compiler.

IGY	The prefix that identifies this message as coming from the COBOL compiler.
pp	Two characters that identify which phase or subphase of the compiler discovered the error. 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 error message.
l	A character that indicates the severity level of the error: I, W, E, S, or U.
message-text	The message text, which in the case of an error message is 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 error messages" on page 204

"FLAG" on page 245

cob2 options

To specify a cob2 option, precede it by a hyphen (-). Do not use a slash (/) for options unless you want to pass those options to the linker using cob2. Any options that you specify in the cob2 command that are not listed below are passed to the linker.

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*, cob2 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*, cob2 exits with the same return code returned by the compiler.
The default is -comprc_ok=4.
- dll[:*xxx*]**
Causes cob2 to produce linker files (.LIB and .EXP) to create a DLL named *xxx*. If *xxx* is omitted, the name of the first object (.OBJ) or COBOL source (usually .CBL) file specified in the cob2 command is the name of the DLL (and .LIB and .EXP files).
- host** Sets the compiler options for host COBOL data representations and language semantics:
 - BINARY(S390)
 - CHAR(EBCDIC)
 - COLLSEQ(EBCDIC)
 - NCOLLSEQ(BIN)
 - FLOAT(S390)

The -host option also affects the format of command-line arguments, as discussed in the related task referenced below.

- Ixxx** Adds path *xxx* to the directories to be searched for copybooks if a library-name is not specified. (This option is the uppercase letter *I*, not the lowercase letter *i*.)
- 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*.
- If you use the **COPY** statement, you must ensure that the **LIB** compiler option is in effect.
- qxxx** 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"
```

## Options that apply to linking

### **-b"xxx"**

Passes the string *xxx* to the linker as parameters. *xxx* is a list of linker options separated by spaces. The **cob2** default parameters are also passed. Do not use spaces immediately after **-b**.

Alternatively, you can specify linker options directly as individual **cob2** options. For example, to pass the **/DE** option to the linker, use this command:

```
cob2 /DE myprog.cbl
```

### **-cmain**

Makes a C or PL/I object file that contains a main routine the main entry point in the executable file (.EXE). In C, a main routine is identified by the function name **main()**. In PL/I, a main routine is identified by the **PROC OPTIONS(MAIN)** statement.

If you link a C or PL/I object file that contains a main routine with one or more COBOL object files, you must use **-cmain** to designate the C or PL/I routine as the main entry point in the executable file. A COBOL program cannot be the main entry point in an executable file that contains a C or PL/I main program. Unpredictable behavior occurs if this is attempted, and no diagnostics are issued.

For example, the following two commands are equivalent:

```
cob2 -cmain myCmain.obj myCOBOL.obj
cob2 -cmain myCOBOL.obj myCmain.obj -main:myCmain
```

Either command generates the executable file **myCmain.exe**. The main entry is the C **main()** function contained in the **myCmain.obj** object file.

This option presents runtime command-line arguments in host data format, that is, EBCDIC for character data and big-endian for binary data.

### **-imp:xxx.lib**

Indicates that *xxx.lib* is an import library, not a standard library. This value is passed in the generated **ILINK** command, but not in any generated **ILIB** commands.

### **-main:xxx**

Makes object file *xxx* the main program of the executable (.EXE) file. *xxx* must be the name of an object (.OBJ) file or COBOL source file specified to cob2. *xxx* cannot appear in a linker response file. For example, the following command causes abc to be the main program.

```
cob2 -main:abc a1.cbl d:\cats\abc.obj b2.cbl
```

If -main is not specified, the first object or source file specified becomes, in the absence of a linker response file, the COBOL main program.

If the syntax of -main:xxx is invalid, or xxx is not the file-name of an object or source file processed by cob2, cob2 terminates.

### **-s:0xxxxxxxx**

Instructs the linker to allocate *xxxxxxxx* bytes of stack space to the executable program, where *xxxxxxxx* is a hexadecimal number. This space might be needed to run programs that have defined very large data items, especially data items in the LOCAL-STORAGE SECTION.

### **/HEAP:0xxxxxxxx**

Instructs the linker to allocate *xxxxxxxx* bytes of heap space for static user data, where *xxxxxxxx* is a hexadecimal number. (The default value is 0x100000, and the maximum is 0xffffffe.) This space might be needed to run programs that have defined very large data items, particularly data items in the WORKING-STORAGE SECTION.

## **Options that apply to both compiling and linking**

- g** Produces symbolic information used by the debugger. This option is equivalent to compiling with the TEST compiler option and linking with the /DEBUG linker option.
- h** Displays the cob2 help text.
- v** Displays compile and link steps, and executes them.
- #** Displays compile and link steps, but does not execute them.
- ?, ?** Displays the cob2 help text.

#### **RELATED TASKS**

"Using command-line arguments" on page 482

#### **RELATED REFERENCES**

"File names and extensions supported by cob2" on page 209

"Compiler environment variables" on page 195

"Runtime environment variables" on page 196

---

## **Linking programs**

After the compiler has created object modules from your source files, use the linker to link them with the COBOL for Windows runtime libraries to create an .EXE file or a .DLL file.

You can use linker options or statements in the module definition file (.DEF) to specify the kind of output that you want:

- To produce an .EXE file, specify the /EXEC option or include the module statement NAME. The linker produces .EXE files by default.



- To produce a .DLL file, specify the /DLL option or include the module statement LIBRARY.

You can set linker options in any of the following ways:

- On the command line. You can specify options anywhere on the command line, as in `cob2 /M myprog.obj`.
- In the ILINK environment variable from the command line or in a command file. The options will be in effect only for the current session.
- In the ILINK environment variable in the System Properties window, either per user or per system.

Options that you specify on the command line override the options in the ILINK environment variable. Storing frequently used options in the ILINK environment variable saves time if you use the same command-line options whenever you link. You cannot specify file-names in the environment variable, however.

You have a choice of ways to start the linker:

- Through the compiler using `cob2`, which automatically starts the linker
- Through a makefile, which starts both the compiler and the linker
- From the command line

#### RELATED TASKS

“Setting environment variables” on page 193

“Specifying linker options” on page 210

“Linking through the compiler” on page 210

“Linking from the command line” on page 211

“Creating module definition files” on page 489

#### RELATED REFERENCES

“Linker environment variables” on page 196

“Linker input and output files” on page 212

“Linker search rules” on page 212

Chapter 16, “Linker options,” on page 277

## File names and extensions supported by cob2

Files with certain extensions are processed by the compiler. The file extension .CBL is most commonly used for COBOL source.

All other files are passed to the linker. Those with recognized file extensions are processed as follows:

- .DEF** The name of the module definition file.
- .DLL** The name of the generated dynamic link library (DLL). The default DLL is the first source file listed in the `cob2` command that has extension .DLL.
- .EXP** The name of the export file.
- .EXE** The name of the generated executable file. If not specified, the name defaults to the name of the first COBOL source file listed in the `cob2` command that has extension .EXE.
- .IMP** The name of the import library associated with a DLL that contains symbols (usually names of external routines) referenced by your programs. This file is used by the linker to resolve those references.
- .LIB** The name of the import or standard library, which contains symbols

(usually names of external routines) referenced by your programs. This file is used by the linker to resolve those references.

**.MAP** The name of the map file. If /MAP is not specified, no map file is generated.

**.OBJ** The name of the object file to be passed to the linker.

#### RELATED TASKS

“Creating module definition files” on page 489

## Specifying linker options

You can specify linker options in lowercase, uppercase, or mixed case, and in short or long form. You can also substitute a dash (-) for the slash (/) before an option.

For example, /DE, /DEB, or /DEBU are equivalent to /DEBUG, and -DEBUG is equivalent to /DEBUG. The summary of linker options referenced below shows the shortest acceptable form for each option.

Separate options with a space or tab character. For example:

```
cob2 /de -DBGPACK -Map /NOI prog.obj
```

Some linker options and module statements take numeric arguments. Using standard C-language syntax, you can specify numbers in any of the following forms:

#### Decimal

Any number not prefixed with 0 or 0x. For example, 1234 is a decimal number.

**Octal** Any number prefixed with 0 (but not 0x). For example, 01234 is an octal number.

#### Hexadecimal

Any number prefixed with 0x. For example, 0x1234 is a hexadecimal number.

#### RELATED TASKS

“Linking programs” on page 208

#### RELATED REFERENCES

Chapter 16, “Linker options,” on page 277

## Linking through the compiler

When you start the COBOL for Windows compiler, it compiles the object files from your source code and then starts the linker to link the object files into an .EXE or .DLL file.

The compiler passes the object files it creates to the linker along with any object files that you specify on the compiler command line. The compiler does not pass any default parameters to the linker. Use the -b option of the cob2 command to pass options to the linker.

If you do not want the compiler to start the linker, specify the -c option of the cob2 command. You can then start the linker in a separate step.

“Examples: using cob2 for linking” on page 211

## Linking from the command line

Use the IBM Library Manager (ILIB) to create and maintain libraries of object code, create import libraries and export object pairs, and generate module definition (.def) files. You can then invoke the linker either directly by issuing the ILINK command, or indirectly by using the cob2 command.

When you start the linker from the command line, the linker assumes that any inputs that it cannot recognize as other files, options, or directories must be object files. Use a space or tab character to separate files. You must enter at least one object file.

To have the linker search additional directories for input files, specify a drive or directory by itself on the command line. Specify the drive or directory with a slash (/) or backslash (\) character at the end. The paths you specify are searched before the paths in the LIB environment variable.

You can write a makefile to organize the sequence of actions (such as compiling and linking) that are required to build your project. You can use the makefile to invoke all those actions in one step.

“Examples: using cob2 for linking”

“Example: overriding linker options” on page 212

### RELATED TASKS

“Using NMAKE to update projects” on page 215

“Creating module definition files” on page 489

### RELATED REFERENCES

“Linker search rules” on page 212

## Examples: using cob2 for linking

The following examples illustrate the use of cob2 for linking.

To link two files together, compile them without the -c option. For example, to compile and link alpha.cbl and beta.cbl and generate alpha.exe, enter:

```
cob2 alpha.cbl beta.cbl
```

The command above creates alpha.obj and beta.obj, then links alpha.obj, beta.obj, and the COBOL libraries. If the link step is successful, it produces an executable program named alpha.exe.

The following command compiles beta.cbl:

```
cob2 alpha.obj beta.cbl mylib.lib gamma.exe
```

It also passes this string to the linker:

```
alpha.obj beta.obj mylib.lib /out:gamma.exe
```

If linking is successful, the executable gamma.exe is produced.

The following command produces alpha.dll, assuming a valid alpha.def file:

```
cob2 -dll alpha.cbl alpha.def
```

#### RELATED TASKS

“Linking from the command line” on page 211

### Example: overriding linker options

The following example shows that options in the command line override options in an environment variable.

```
SET ILINK=/NOI /AL:256 /DE
cob2 test
cob2 /NODEF /NODEB prog
```

The first command above sets the environment variable ILINK to the options /NOIGNORECASE, /ALIGNMENT:256, and /DEBUG.

The second command links test.obj, using the options specified in ILINK, to produce test.exe.

The last command links prog.obj to produce prog.exe, using the options /NODEFAULTLIBRARYSEARCH and /NODEBUG in addition to the options /NOIGNORECASE and /ALIGNMENT:256. /NODEBUG in the command line overrides the /DEBUG setting in the environment variable.

#### RELATED TASKS

“Linking from the command line” on page 211

## Linker input and output files

The linker takes object files, links them with each other and with any library files you specify, and produces an executable output file, which can be either an executable program file (.EXE) or a dynamic link library (.DLL).

The linker can also produce a map file, which provides information about the contents of the executable output.

### Linker inputs:

- Options
- Object files (\*.OBJ)
- Library files (\*.LIB)
- Import libraries (\*.LIB)
- Module definition file (.DEF)

### Linker outputs:

- Executable file (.EXE or .DLL)
- Map file (.MAP)
- Return code

The linker accepts object files that are produced by IBM COBOL for Windows or by IBM PL/I for Windows.

### Linker search rules

When searching for an object (.OBJ), library (.LIB), or module definition (.DEF) file, the linker looks for the file in several locations.

The linker looks in this order:

1. The directory you specified for the file, or the current directory if you did not specify a path. Default libraries do not include path specifications.

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. Any directories entered by themselves on the command line, which must end with a slash (/) or backslash (\) character.
3. Any directories listed in the LIB environment variable.

If the linker cannot locate a file, it generates an error message and stops linking.

“Example: linker search rules”

#### RELATED REFERENCES

“Linker environment variables” on page 196

“File-name defaults”

### Example: linker search rules

This example links four object files to create an executable file named FUN.EXE.

```
FUN.OBJ TEXT.OBJ TABLE.OBJ CARE.OBJ
NEWLIBV2.LIB
C:\TESTLIB\
```

The linker searches NEWLIBV2.LIB before searching the default libraries to resolve references. To locate NEWLIBV2.LIB and the default libraries, the linker searches the locations in this order:

1. Current directory (because NEWLIBV2.LIB was entered without a path)
2. C:\TESTLIB\ directory
3. Directories listed in the LIB environment variable

#### RELATED REFERENCES

“Linker search rules” on page 212

## File-name defaults

If you do not enter a file-name, the linker assumes default names.

Table 27. Default file-names assumed by the linker

| File                   | Default file-name                                                                                                                                                                        | Default extension |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|
| Object files           | None. You must enter at least one object file name.                                                                                                                                      | .OBJ              |
| Output file            | The base name of the first object file.                                                                                                                                                  | .EXE              |
| Map file               | The base name of the output file.                                                                                                                                                        | .MAP              |
| Library files          | The default libraries defined in the object files. Use compiler options to define the default libraries. Any additional libraries you specify are searched before the default libraries. | .LIB              |
| Module definition file | None. The linker assumes you accept the default for all module statements.                                                                                                               | .DEF              |

---

## 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.

#### RELATED REFERENCES

"Linker errors in program-names"

## Linker return codes

The linker issues one of several return codes.

Table 28. Linker return codes

| Code | Meaning                                                                                                       |
|------|---------------------------------------------------------------------------------------------------------------|
| 0    | The linker completed successfully. The linker detected no errors, and issued no warnings.                     |
| 4    | The linker issued warnings. There might be problems with the output file.                                     |
| 8    | The linker detected errors. The linking might have completed, but the output file cannot be run successfully. |
| 12   | The linker issued warnings and detected errors (see return codes 4 and 8).                                    |
| 16   | The linker detected severe errors. Linking ended abnormally, and the output file cannot be run successfully.  |
| 20   | The linker issued warnings and detected severe errors (see return codes 4 and 16).                            |
| 24   | The linker detected both errors and severe errors (see return codes 8 and 16).                                |
| 28   | The linker issued warnings and detected both errors and severe errors (see return codes 4, 8, and 16).        |

## Linker errors in program-names

The default linkage convention is `SYSTEM(STDCALL)`, which is in effect when you use the compiler option `CALLINT(SYSTEM)`.

With this convention, the name of the called routine is expanded in two ways (known as *name decoration*):

- An underscore character (`_`) is added as a prefix.
- An at symbol (`@`) and a one-digit or two-digit number that signifies the length in bytes of the argument list is added as a suffix.

If you are using this linkage convention, you must ensure that the argument list in the calling program exactly matches the parameter list in the called subroutine. For example, suppose you code this statement:

```
Call SubProg Using Parm-1 Parm-2.
```

The name of the called routine will be `_SubProg@8`. Suppose, however, the `SubProg` routine itself is coded as:

Procedure Division Using Parm-1 Parm-2 Parm-3.

Its system-generated name will be `_SubProg@12`. This different name will cause a linker error because the linker will not be able to resolve the call to `_SubProg@8`.

#### RELATED REFERENCES

"CALLINT" on page 229

"SYSTEM" on page 461

---

## Using NMAKE to update projects

Use NMAKE to automate the process of updating a project after you make a change to a source file.

You set up a special text file, called a *description file* (or *makefile*), that tells NMAKE which files depend on others and which commands, such as compile and link commands, need to be carried out to bring your program up-to-date.

After NMAKE has successfully run, you have an executable file that you can run or a dynamic link library that you can use as a resource.

#### RELATED TASKS

"Running NMAKE on the command line"

"Running NMAKE with a command file" on page 216

"Defining description files for NMAKE" on page 216

## Running NMAKE on the command line

To run NMAKE, issue the command in the format that is shown below.

```
nmake options macrodefinitions targets /F filename
```

### *options*

Specifies options that modify NMAKE's actions. To see a list of supported options, type the following command on the command line:

```
nmake /?
```

### *macrodefinitions*

Lists macro definitions for NMAKE to use. Macro definitions that contain spaces must be enclosed by quotation marks.

### *targets*

Specifies the names of one or more target files to build. If you do not list any targets, NMAKE builds the first target in the description file.

### *filename*

Names the description file where you specify file dependencies and the commands to execute when a file is out-of-date. A common convention is to name the file *program.MAK*, but you can give it any name. If you do not specify a file name, NMAKE will look for and use a file named *makefile*.

```
nmake /s "program = flash" sort.exe search.exe
```

The example above takes the following actions:

- Invokes NMAKE with the `/s` option to suppress the command display
- Defines a macro, assigning the string `flash` to the macro `program`
- Specifies two targets: `sort.exe` and `search.exe`
- Looks for a file named *makefile* and uses it if it is available

## Running NMAKE with a command file

A command file is a response file that you use to extend command-line input to NMAKE. Use a command file for complex and long commands that you type frequently or for strings of command-line arguments, such as macro definitions, that exceed the limit for command-line length.

To provide input to NMAKE with a command file, type:

```
nmake @commandfile
```

For the *commandfile*, type the name of a file that contains the same information that you would normally enter on the command line.

NMAKE treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines if you end each line except the last with a backslash (\). You must enclose macro definitions that contain spaces in quotation marks, just as if you entered them directly on the command line.

The following example is a command file called update:

```
/s "program \
= flash" sort.exe search.exe
```

You can use this command file by typing the following command:

```
nmake @update
```

This command runs NMAKE using these items:

- The /s option to suppress the command display
- The macro definition "program = flash"
- The targets specified as sort.exe and search.exe
- The description file makefile by default

The backslash allows the macro definition to span two lines.

## Defining description files for NMAKE

In its simplest form, a description file tells NMAKE which files depend on others and which commands need to be executed if a file changes.

A description file contains from one to 1048 *description blocks*. A description block indicates the relationship among various parts of the program, and contains commands to bring all components up-to-date. A description file has the following format:

```
targets. . . : dependents. . .
 command
 command
 command
 . . .
```

For example, you might use the following description file to compile and link two files:

```
target.lib: a.cob b.cob
 cob2 a.cob b.cob
 ilink target a.obj b.obj
```



---

## Running programs

To run a COBOL program, set environment variables, run the program, and 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.

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.

For example, if the command `cob2 alpha.cbl beta.cbl` is successful, you can run the program by entering `alpha`.

3. Correct runtime errors.

For example, you can use the debugger to examine the program state at the time of the 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 193

“Setting environment variables for COBOL for Windows” on page 194

“Using the debugger” on page 306

### RELATED REFERENCES

“Runtime environment variables” on page 196

Appendix I, “Runtime messages,” on page 707

---

## Redistributing COBOL for Windows DLLs

If you develop a COBOL application with IBM COBOL for Windows and plan to deploy that application to another Windows system that does not have Rational Developer for System z installed, you must redistribute the IBM COBOL library DLLs along with the application.

These files are located in a zipped file, `cobship.zip`, in the subdirectory `ship\cobol\` of the COBOL install directory. (The location of the COBOL install directory is specified in the Windows environment variable `RDZvrINSTDIR`, where *v* is the version number and *r* is the release number of Rational Developer for System z.) See the readme file in that directory for further details.

If you redistribute these files, you agree to the following conditions:

- Copies of these modules are provided “AS IS.” You are responsible for all technical assistance for your application.
- You will prohibit users from copying (except for backup purposes), reverse assembling, reverse compiling, or otherwise translating the application.
- You will not use the same path name as the original files or modules.
- You will label your application that contains a copy of any of these files or modules as follows:

### CONTAINS

IBM COBOL for Windows V7.5

Runtime Modules

(c) Copyright IBM Corporation 1998,2008

All Rights Reserved

Refer to the license agreement for the full list of terms and conditions for redistributing files.

---

## Chapter 13. Compiling, linking, and running OO applications

You can compile, link, and run object-oriented (OO) applications at the command line by following the instructions below.

### RELATED TASKS

“Compiling OO applications”

“Preparing OO applications” on page 220

“Running OO applications” on page 221

---

### Compiling OO applications

When you compile OO applications, use the `cob2` command to compile COBOL client programs and class definitions, and the `javac` command to compile Java class definitions to produce *bytecode* (extension `.class`).

To compile COBOL source code that contains OO syntax such as `INVOKE` statements or class definitions, or that uses Java services, you must use the `THREAD` compiler option.

A COBOL source file that contains a class definition must not contain any other class or program definitions.

When you compile a COBOL class definition, two output files are generated:

- The object file (`.obj`) for the class definition.
- A Java source program (`.java`) that contains a class definition that corresponds to the COBOL class definition. Do not edit this generated Java class definition in any way. If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

If a COBOL client program or class definition includes the file `JNI.cpy` by using a `COPY` statement, specify the include subdirectory of the COBOL install directory in the search order for copybooks. (The location of the COBOL install directory is specified in the Windows environment variable `RDZvrINSTDIR`, where *v* is the version number and *r* is the release number of Rational Developer for System z.)

You can specify the include subdirectory by using the `-I` option of the `cob2` command or by setting the `SYSLIB` environment variable.

### RELATED TASKS

“Setting environment variables” on page 193

“Compiling programs” on page 201

“Preparing OO applications” on page 220

“Running OO applications” on page 221

“Accessing JNI services” on page 431

### RELATED REFERENCES

“Compiler environment variables” on page 195

“cob2 options” on page 206

“THREAD” on page 265

---

## Preparing OO applications

Use the `cob2` command to link OO COBOL applications.

To prepare an OO COBOL client program for execution, link the object file to create an executable DLL module.

To prepare a COBOL class definition for execution:

1. Link the object file to create an executable DLL module.

You must name the resulting DLL module *Classname.dll*, where *Classname* is the external class-name. If the class is part of a package and thus there are periods in the external class-name, you must change the periods to underscores in the DLL module name. For example, if class `Account` is part of the `com.acme` package, the external class-name (as defined in the `REPOSITORY` paragraph entry for the class) must be `com.acme.Account`, and the DLL module for the class must be `com_acme_Account.dll`.

2. Compile the generated Java source with the Java compiler to create a class file (.class).

For a COBOL source file *Classname.cbl* that contains the class definition for *Classname*, you would use the following commands to compile and link the components of the application:

**Table 29. Commands for compiling and linking a class definition**

| Command                                     | Input                 | Output                                          |
|---------------------------------------------|-----------------------|-------------------------------------------------|
| <code>cob2 -c -qthread Classname.cbl</code> | <i>Classname.cbl</i>  | <i>Classname.obj</i> ,<br><i>Classname.java</i> |
| <code>cob2 -dll Classname.obj</code>        | <i>Classname.obj</i>  | <i>Classname.dll</i>                            |
| <code>javac Classname.java</code>           | <i>Classname.java</i> | <i>Classname.class</i>                          |

After you issue the `cob2` and `javac` commands successfully, you have the executable components for the program: the executable DLL module *Classname.dll* and the class file *Classname.class*. All files from these commands are generated in the current working directory.

“Example: compiling and linking a COBOL class definition” on page 221

### RELATED TASKS

“Compiling programs” on page 201

“`REPOSITORY` paragraph for defining a class” on page 392

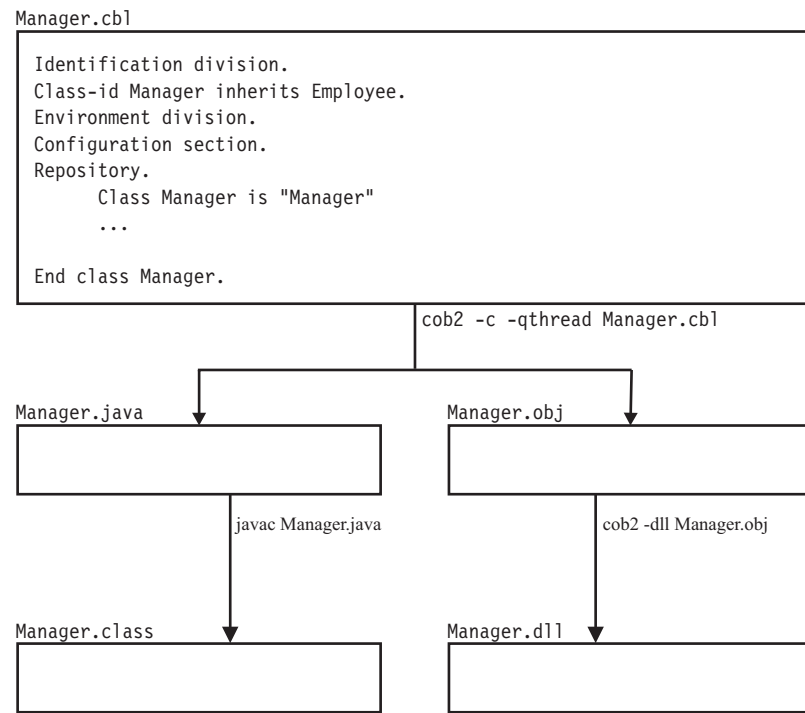
### RELATED REFERENCES

“`cob2` options” on page 206

---

## Example: compiling and linking a COBOL class definition

This example illustrates the commands that you use and the files that are produced when you compile and link a COBOL class definition, Manager.cbl.



The class file `Manager.class` and the DLL module `Manager.dll` are the executable components of the application, and are generated in the current working directory.

---

## Running OO applications

You can run object-oriented COBOL applications from the command line.

Specify the directory that contains the DLLs for the COBOL classes in the `PATH` environment variable. Specify the directory paths for the Java class files that are associated with the COBOL classes in the `CLASSPATH` environment variable as follows:

- For classes that are not part of a package, end the class path with the directory that contains the `.class` files.
- For classes that are part of a package, end the class path with the directory that contains the “root” package (the first package in the full package name).
- For a `.jar` file that contains `.class` files, end the class path with the name of the `.jar` file.

Separate multiple path entries with semicolons.

The following releases of Java are supported for running object-oriented COBOL applications:

- The IBM 32-bit SDK for Windows, Java 2 Technology Edition, Version 1.3.1 or later.

- The Sun Java Developers Kit 1.3.1. If the application starts with a Java program, and is launched with the java command, you must specify the -classic option on the java command.

**Attention:** The Sun Java Developer Kits 1.4.x are not supported for COBOL programs that use object-oriented syntax.

#### RELATED TASKS

“Running OO applications that start with a main method”

“Running OO applications that start with a COBOL program”

“Running programs” on page 217

“Setting environment variables” on page 193

Chapter 24, “Writing object-oriented programs,” on page 387

“Structuring OO applications” on page 428

## Running OO applications that start with a main method

If the first routine of a mixed COBOL and Java application is the main method of a Java class or the main factory method of a COBOL class, run the application by using the java command and by specifying the name of the class that contains the main method.

The java command initializes the Java virtual machine (JVM). To customize the initialization of the JVM, specify options on the java command as in the following examples:

*Table 30. java command options for customizing the JVM*

| Purpose                                                                                                         | Option         |
|-----------------------------------------------------------------------------------------------------------------|----------------|
| To set a system property                                                                                        | -Dname=value   |
| To request that the JVM generate verbose messages about garbage collection                                      | -verbose:gc    |
| To request that the JVM generate verbose messages about class loading                                           | -verbose:class |
| To request that the JVM generate verbose messages about native methods and other Java Native Interface activity | -verbose:jni   |
| To set the initial Java heap size to value bytes                                                                | -Xmsvalue      |
| To set the maximum Java heap size to value bytes                                                                | -Xmxvalue      |
| To use the “classic” JVM that is provided by Sun Java Developers Kit 1.3.1                                      | -classic       |

See the output from the java -h command or the related references for details about the options that the JVM supports.

#### RELATED REFERENCES

JVM options (*Persistent Reusable Java Virtual Machine User’s Guide*)

## Running OO applications that start with a COBOL program

If the first routine of a mixed COBOL and Java application is a COBOL program, run the application by specifying the program name at the command prompt. If a JVM is not already running in the process of the COBOL program, the COBOL run time automatically initializes a JVM.

To customize the initialization of the JVM, specify options by setting the `COBJVMINITOPTIONS` environment variable. Use blanks to separate options. For example:

```
export COBJVMINITOPTIONS="-Xms100000000 -Xmx200000000 -verbose:gc"
```

#### RELATED TASKS

“Running programs” on page 217

“Setting environment variables” on page 193

#### RELATED REFERENCES

JVM options (*Persistent Reusable Java Virtual Machine User’s Guide*)





## Chapter 14. 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 31. Compiler options*

| Aspect of your program   | Compiler option           | Default                                                           | Abbreviations |
|--------------------------|---------------------------|-------------------------------------------------------------------|---------------|
| Source language          | "ARITH" on page 228       | ARITH(COMPAT)                                                     | AR(C E)       |
|                          | "CICS" on page 232        | NOCICS                                                            | None          |
|                          | "CURRENCY" on page 236    | NOCURRENCY                                                        | CURR NOCURR   |
|                          | "LIB" on page 248         | LIB                                                               | None          |
|                          | "NSYMBOL" on page 253     | NSYMBOL(NATIONAL)                                                 | NS(NAT DBCS)  |
|                          | "NUMBER" on page 253      | NONUMBER                                                          | NUM NONUM     |
|                          | "QUOTE/APOST" on page 257 | QUOTE                                                             | Q APOST       |
|                          | "SEQUENCE" on page 259    | SEQUENCE                                                          | SEQ NOSEQ     |
|                          | "SOSI" on page 260        | NOSOSI                                                            | None          |
|                          | "SQL" on page 262         | SQL("")                                                           | None          |
| Date processing          | "DATEPROC" on page 237    | NODATEPROC, or<br>DATEPROC(FLAG) if only<br>DATEPROC is specified | DP NODP       |
|                          | "YEARWINDOW" on page 271  | YEARWINDOW(1900)                                                  | YW            |
| Maps and listings        | "LINECOUNT" on page 249   | LINECOUNT(60)                                                     | LC            |
|                          | "LIST" on page 249        | NOLIST                                                            | None          |
|                          | "LSTFILE" on page 250     | LSTFILE(LOCALE)                                                   | LST           |
|                          | "MAP" on page 250         | NOMAP                                                             | None          |
|                          | "SOURCE" on page 261      | SOURCE                                                            | S NOS         |
|                          | "SPACE" on page 262       | SPACE(1)                                                          | None          |
|                          | "TERMINAL" on page 264    | TERMINAL                                                          | TERM NOTERM   |
|                          | "VBREF" on page 269       | NOVBREF                                                           | None          |
|                          | "XREF" on page 269        | XREF(FULL)                                                        | X NOX         |
| Object module generation | "COMPILE" on page 235     | NOCOMPILE(S)                                                      | C NOC         |
|                          | "OPTIMIZE" on page 254    | NOOPTIMIZE                                                        | OPT NOOPT     |
|                          | "PGMNAME" on page 255     | PGMNAME(UPPER)                                                    | PGMN(LU LM)   |
|                          | "SEPOBJ" on page 258      | SEPOBJ                                                            | None          |

Table 31. Compiler options (continued)

| Aspect of your program    | Compiler option         | Default                      | Abbreviations         |
|---------------------------|-------------------------|------------------------------|-----------------------|
| Object code control       | "BINARY" on page 229    | BINARY (NATIVE)              | None                  |
|                           | "CHAR" on page 230      | CHAR (NATIVE)                | None                  |
|                           | "COLLSEQ" on page 233   | COLLSEQ (BIN)                | None                  |
|                           | "DIAGTRUNC" on page 238 | NODIAGTRUNC                  | DTR NODTR             |
|                           | "FLOAT" on page 247     | FLOAT (NATIVE)               | None                  |
|                           | "NCOLLSEQ" on page 252  | NCOLLSEQ (BINARY)            | NCS(L BIN B)          |
|                           | "TRUNC" on page 266     | TRUNC (STD)                  | None                  |
|                           | "ZWB" on page 271       | ZWB                          | None                  |
| CALL statement behavior   | "DYNAM" on page 238     | NODYNAM                      | DYN NODYN             |
| Debugging and diagnostics | "FLAG" on page 245      | FLAG(I,I)                    | F NOF                 |
|                           | "FLAGSTD" on page 246   | NOFLAGSTD                    | None                  |
|                           | "SSRANGE" on page 263   | NOSSRANGE                    | SSR NOSSR             |
|                           | "TEST" on page 264      | NOTEST                       | None                  |
| Other                     | "ADATA" on page 227     | NOADATA                      | None                  |
|                           | "CALLINT" on page 229   | CALLINT (SYSTEM,NODESC)      | None                  |
|                           | "ENTRYINT" on page 239  | ENTRYINT (SYSTEM)            | None                  |
|                           | "EXIT" on page 240      | NOEXIT                       | EX(INX,LIBX,PRTX,ADX) |
|                           | "MDECK" on page 251     | NOMDECK                      | NOMD MD MD(C) MD(NOC) |
|                           | "PROBE" on page 256     | PROBE                        | None                  |
|                           | "SIZE" on page 260      | 8388608 bytes (approx. 8 MB) | SZ                    |
|                           | "THREAD" on page 265    | NOTHREAD                     | None                  |
|                           | "WSCLEAR" on page 269   | NOWSCLEAR                    | None                  |

**Installation defaults:** The defaults listed with the options above are the defaults shipped with the product.

**Performance considerations:** The BINARY, CHAR, DYNAM, FLOAT, OPTIMIZE, SSRANGE, TEST, and TRUNC compiler options can all affect runtime performance.

#### RELATED TASKS

Chapter 33, "Tuning your program," on page 537

#### RELATED REFERENCES

Chapter 15, "Compiler-directing statements," on page 273

## Conflicting compiler options

The COBOL for Windows 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, or mutually exclusive options are specified at the same level in the hierarchy.

When conflicting options are specified 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 generates an error message and forces one of the options to a nonconflicting value. For example, if you specify both OPTIMIZE and TEST in the PROCESS statement in any order, TEST takes effect and OPTIMIZE is ignored.

However, options specified at a higher level of precedence override any options specified at a lower level of precedence. 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. Default options

For example, if you code TEST in the COBOPT environment variable but OPTIMIZE in the PROCESS statement, OPTIMIZE 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.

Table 32. Mutually exclusive compiler options

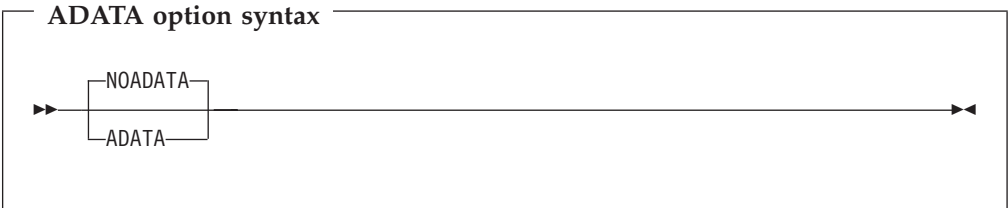
| Specified | Ignored  | Forced on  |
|-----------|----------|------------|
| CICS      | DYNAM    | NODYNAM    |
|           | NOLIB    | LIB        |
|           | NOTHREAD | THREAD     |
| MDECK     | NOLIB    | LIB        |
| SQL       | NOLIB    | LIB        |
| TEST      | OPTIMIZE | NOOPTIMIZE |

RELATED REFERENCES

- “Compiler environment variables” on page 195
- “cob2 options” on page 206
- Chapter 15, “Compiler-directing statements,” on page 273

ADATA

Use ADATA when you want the compiler to create a SYSADATA file that contains records of additional compilation information.



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.

You cannot specify ADATA in a PROCESS (CBL) statement. You can specify it only in one of the following ways:

- As a cob2 command option
- In the COBOPT environment variable

#### RELATED REFERENCES

Appendix H, “COBOL SYSADATA file contents,” on page 641

“Compiler environment variables” on page 195

“cob2 options” on page 206

---

## ARITH

ARITH affects the maximum number of digits that you can code for integers, and the number of digits used in fixed-point intermediate results.

### ARITH option syntax

The diagram shows the syntax for the ARITH option. It starts with a right-pointing arrow followed by the text 'ARITH(' in a monospace font. To the right of the opening parenthesis is a bracketed choice between 'COMPAT' and 'EXTEND', also in monospace. This is followed by a closing parenthesis ')'. A long horizontal line with arrowheads at both ends extends to the right from the closing parenthesis, indicating that the option can be followed by other code.

```
➡ ARITH(COMPAT
 EXTEND)
```

Default is: ARITH(COMPAT)

Abbreviations are: AR(C), AR(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 and NUMVAL-C 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 and NUMVAL-C 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 569

## BINARY

BINARY affects the representation 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 Windows, this is *little-endian* format (most significant digit at the highest address).

BINARY(S390) indicates that BINARY, COMP, and COMP-4 data items are represented consistently with zSeries, that is, in *big-endian* format (least significant digit at the highest address).

However, COMP-5 binary data and data items defined with the NATIVE keyword on the USAGE clause are not affected by the BINARY(S390) option. They are always stored in the native format of the platform.

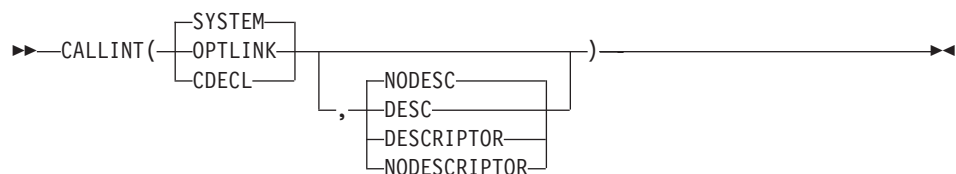
#### RELATED REFERENCES

Appendix B, “zSeries host data format considerations,” on page 567

## 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.

(Use ENTRYINT for the selection of the call interface convention for the program entry point or entry points.)

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. This is STDCALL, the linkage used by the Windows-based system APIs.

**Attention:** This convention cannot be used in all cases when the called program has multiple entry points.

**OPTLINK**

The OPTLINK suboption specifies that the call interface convention is OPTLINK, which provides a faster alternative to the SYSTEM convention. OPTLINK is used by existing IBM C and C++ functions and COBOL and PL/I programs.

**CDECL** The CDECL suboption specifies that the call interface convention is CDECL, which is used to interface with Microsoft® Visual C/C++ for Windows functions. CDECL is the default convention for Microsoft C and C/C++ functions.

- Specifying whether the argument descriptors are to be generated:

**DESC** The DESC suboption specifies that an argument descriptor is passed for each argument on 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 TASKS**

“Setting linkage conventions for COBOL and C/C++” on page 463

“Coding multiple entry points” on page 477

**RELATED REFERENCES**

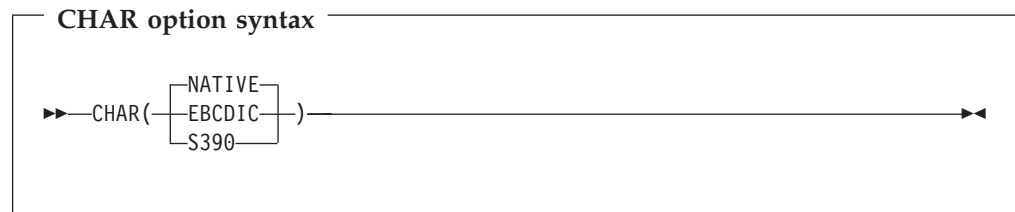
Chapter 15, “Compiler-directing statements,” on page 273

“Call interface conventions” on page 459

---

## CHAR

CHAR affects the representation and runtime treatment of USAGE DISPLAY and USAGE DISPLAY-1 data items.



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 Windows, 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 DBCS code page.

CHAR(EBCDIC) and CHAR(S390) are synonymous and indicate that DISPLAY and DISPLAY-1 data items are in the character representation of zSeries, 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. See the table in the related reference about the COLLSEQ option for the rules about the comparison of character data when the CHAR(EBCDIC) option is in effect.
  - 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 nonnumeric 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 with a locale" on page 180

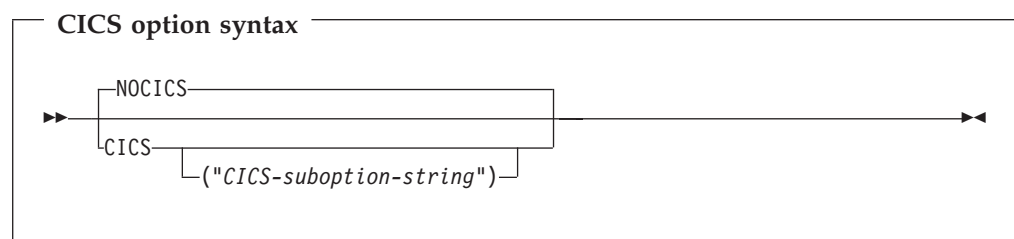
#### RELATED REFERENCES

Appendix B, "zSeries host data format considerations," on page 567

"COLLSEQ" on page 233

## CICS

The CICS compiler option enables the integrated CICS translator and allows specification of 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.



Default is: NOCICS



Abbreviations are: None

Use the CICS option to compile CICS programs only. Programs compiled with the CICS option will not run in a non-CICS environment.

**Attention:** If you specify the CICS option, the compiler needs access to TXSeries V6.1 (or later).

If you specify the NOCICS option, any CICS statements found in the source program are diagnosed and discarded.

Use either quotation marks or single quotation marks 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 command, only the single quotation mark (') can be used as the string delimiter: -q"CICS('options')".

You can partition a long suboption string into multiple suboption strings in multiple CBL 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 mypgm.cbl -q"CICS('string1')"`, the compiler passes the following CICS suboption string to the CICS integrated translator:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. If the compiler finds multiple instances of the same CICS suboption, the last specification of that suboption in the concatenated string takes effect. The compiler limits the length of the concatenated suboption string to 4 KB.

#### RELATED CONCEPTS

"Integrated CICS translator" on page 331

#### RELATED TASKS

"Coding COBOL programs to run under CICS" on page 327

"Compiling and running CICS programs" on page 330

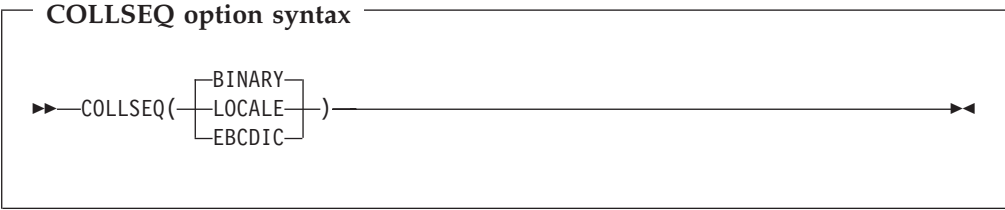
#### RELATED REFERENCES

"Conflicting compiler options" on page 226

---

## COLLSEQ

COLLSEQ specifies the collating sequence for comparison of alphanumeric and DBCS operands.



Default is: COLLSEQ(BIN)

Abbreviations are: CS(L), CS(E), CS(BIN), CS(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 33. 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). |

**Table 33. Effect of comparand data type and collating sequence on comparisons** *(continued)*

| Comparands             | COLLSEQ(BIN)                                                                                     | COLLSEQ(LOCALE)                                                                            | COLLSEQ(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). |
| 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 8

“Controlling the collating sequence with a locale” on page 185

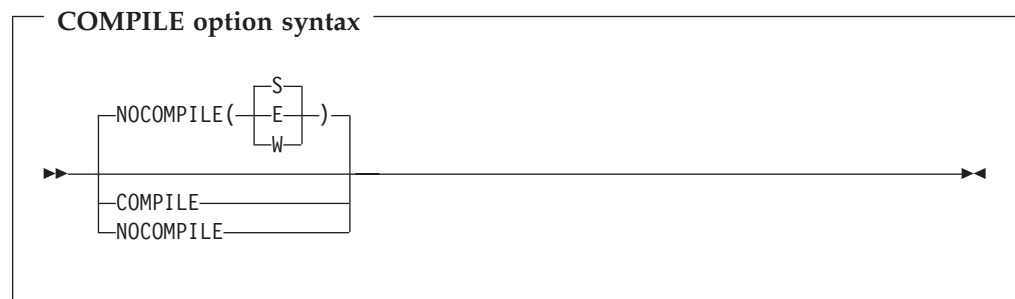
#### RELATED REFERENCES

“CHAR” on page 230

ALPHABET clause

## 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.



Default is: NOCOMPILE(S)

Abbreviations are: C|NOC

Use NOCOMPILE without any suboption to request a syntax check (only diagnostics produced, no object code).

Use NOCOMPILE with 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.

If you request an unconditional NOCOMPILE, the following options have no effect because no object code will be produced:

- LIST
- SSRANGE
- OPTIMIZE
- TEST

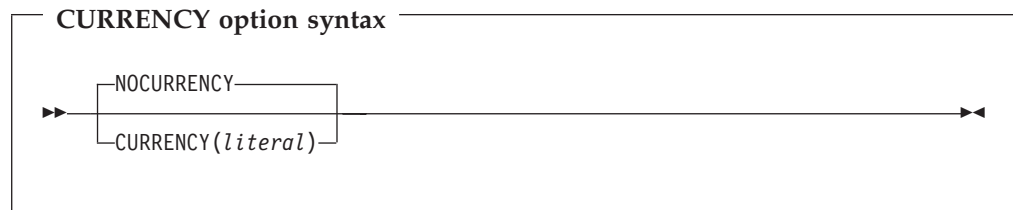
#### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 205

---

## 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 (\$).)



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.

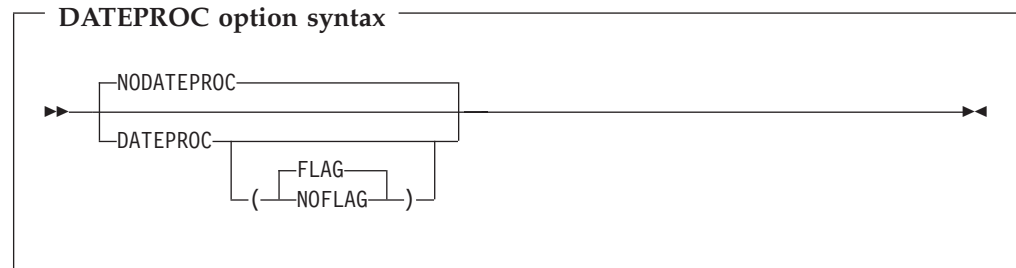
When the NOCURRENCY option is in effect and you omit the CURRENCY SIGN clause, the dollar sign (\$) is used as the PICTURE symbol for the currency sign.

**Delimiter:** You can delimit the CURRENCY option literal with either quotation marks or single quotation marks, regardless of the QUOTE|APOST compiler option setting.

---

## DATEPROC

Use the DATEPROC option to enable the millennium language extensions of the COBOL compiler.



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.

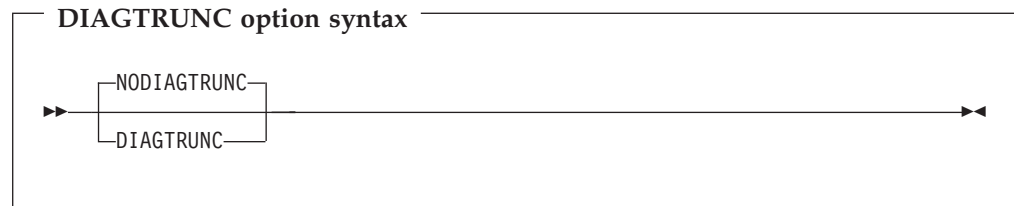
#### RELATED REFERENCES

"FLAG" on page 245

---

## DIAGTRUNC

DIAGTRUNC causes the compiler to issue a severity-4 (Warning) diagnostic message for MOVE statements with numeric receivers when the receiving data item has fewer integer positions than the sending data item or literal. In statements with multiple receivers, the message is issued separately for each receiver that could be truncated.



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 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 for COMP-5 receivers, nor for binary receivers when you specify the TRUNC(BIN) option.

### RELATED CONCEPTS

"Formats for numeric data" on page 43

"Reference modifiers" on page 101

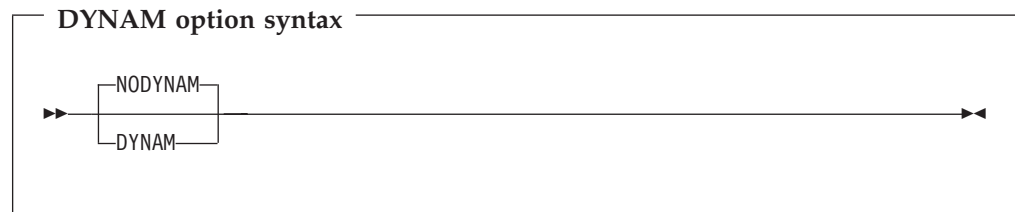
### RELATED REFERENCES

"TRUNC" on page 266

---

## 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.)



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 by programs that are translated by the separate or integrated CICS translator.

With NODYNAM, the target program name is resolved through the linker.

With the DYNAM option, this statement:

```
CALL "myprogram" . . .
```

has the identical behavior to these statements:

```
MOVE "myprogram" to id-1
```

```
CALL id-1 ...
```

#### RELATED CONCEPTS

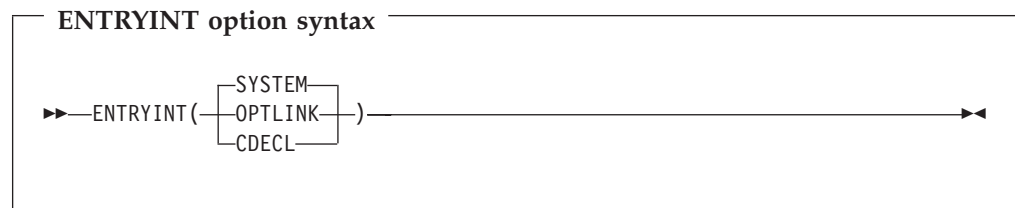
"CALL identifier and CALL literal" on page 458

#### RELATED TASKS

"Creating DLLs" on page 486

## ENTRYINT

Use ENTRYINT to indicate the call interface convention applicable to the program entry points in the USING phrase of either the PROCEDURE DIVISION or ENTRY statement.



Default is: ENTRYINT(SYSTEM)

Abbreviations are: None

(Use CALLINT to select the call interface convention for calls.)

**SYSTEM** The SYSTEM suboption specifies that the call convention is that of the

standard system linkage convention of the platform. This is STDCALL, the linkage used by the system Windows-based APIs.

**Attention:** This convention cannot be used in all cases when the called program has multiple entry points.

**OPTLINK**

The OPTLINK suboption specifies that the call interface convention is OPTLINK, which provides a faster alternative to the SYSTEM convention. OPTLINK is used by existing IBM C and C++ functions and COBOL and PL/I programs. The linkage that is generated for nested programs is always OPTLINK.

**CDECL**

The CDECL suboption specifies that the call interface convention is CDECL, which is used to interface with Microsoft Visual C/C++ for Windows functions. CDECL is the default convention for Microsoft C and C/C++ functions.

**RELATED TASKS**

“Setting linkage conventions for COBOL and C/C++” on page 463  
“Coding multiple entry points” on page 477

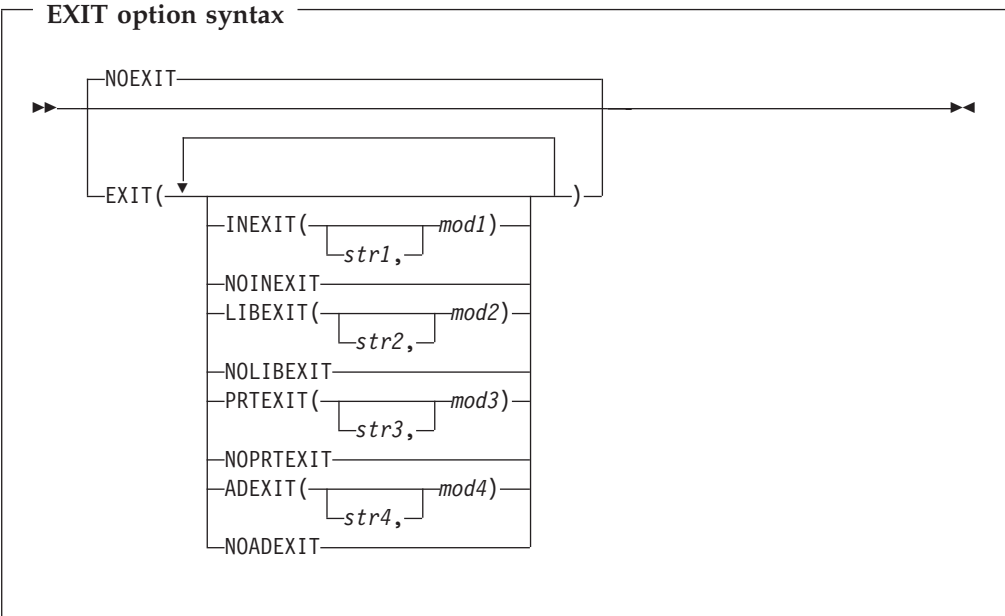
**RELATED REFERENCES**

“Call interface conventions” on page 459

**EXIT**

Use the EXIT option to have the compiler accept user-supplied modules in place of SYSIN, SYSLIB (or copy library), and SYSPRINT.

When creating your EXIT module, ensure that the module is linked as a DLL module before you run it with the COBOL compiler. EXIT modules are invoked with the system linkage convention of the platform.



Default is: NOEXIT



Abbreviations are: EX(INX|NOINX,LIBX|NOLIBX,PRTX|NOPRTX,ADX|NOADX)

If you specify the EXIT option without providing at least one suboption (that is, you specify EXIT()), NOEXIT will be in effect. You can specify the suboptions in any order, separated by either commas or spaces. If you specify both the positive and negative form of a suboption (INEXIT|NOINEXT, LIBEXIT|NOLIBEXIT, PRTEXTIT|NOPRTEXTIT, or ADEXIT|NOADEXIT), the form specified last takes effect. If you specify the same suboption more than once, the suboption specified last takes effect.

For SYSADATA, the ADEXIT suboption provides a module that will be called for each SYSADATA record immediately after the record has been written out to the file.

**No PROCESS:** The EXIT option cannot be specified in a PROCESS(CBL) statement. You can specify it only by using the environment variable COBOPT, or as an option in the cob2 command.

**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.

**PRTEXTIT([*'str3'*],*mod3*)**

The compiler passes printer-destined output to the user-supplied load module (where *mod3* is the module name), instead of SYSPRINT.

**ADEXIT([*'str4'*],*mod4*)**

The compiler passes the SYSADATA output to the user-supplied load module (where *mod4* is the module name).

The module names *mod1*, *mod2*, *mod3*, and *mod4* can refer to the same module.

The suboptions '*str1*', '*str2*', '*str3*', and '*str4*' are character strings that are passed to the load module. These strings are optional; if you use them, they can be up to 64 characters in length and must be enclosed in single quotation marks. Any character is allowed, but included single quotation marks must be doubled, and lowercase characters are folded to uppercase.

## Character string formats

If '*str1*', '*str2*', '*str3*', or '*str4*' is specified, the string is passed to the appropriate user-exit module with the following format, where LL is a halfword (on a halfword boundary) that contains the length of the string. The table shows the location of the character string in the parameter list.

|    |        |
|----|--------|
| LL | string |
|----|--------|

## User-exit work area

When an exit is used, the compiler provides a user-exit work area that can be used to save the address of storage allocated by the exit module. This allows the module to be reentrant.

The user-exit work area is 4 fullwords, residing on a fullword boundary, that is initialized to binary zeroes before the first exit routine is invoked. The address of the work area is passed to the exit module in a parameter list. After initialization, the compiler makes no further reference to the work area. So, you will need to establish your own conventions for using the work area if more than one exit is active during the compilation. For example, the INEXIT module uses the first word in the work area, the LIBEXIT module uses the second word, and the PRTEXT module uses the third word.

## Linkage conventions

Your EXIT modules should use standard linkage conventions between COBOL programs, between library routines, and between COBOL programs and library routines. You need to be aware of these conventions in order to trace the call chain correctly.

## Parameter list for exit modules

The following table shows the format of the parameter list used by the compiler to communicate with the exit module.

*Table 34. Parameter list for exit modules*

| Offset | Contains                   | Description of item                                                                                                                                                                                                                                                          |
|--------|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 00     | User-exit type             | Halfword identifying which user exit is to perform the operation.<br><br>1=INEXIT; 2=LIBEXIT; 3=PRTEXT; 4=ADEXIT                                                                                                                                                             |
| 02     | Operation code             | Halfword indicating the type of operation.<br><br>0=OPEN; 1=CLOSE; 2=GET; 3=PUT; 4=END                                                                                                                                                                                       |
| 04     | Return code                | Fullword, placed by the exit module, indicating status of the requested operation.<br><br>0=Successful; 4=End-of-data; 12=Failed                                                                                                                                             |
| 08     | Data length                | Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation.                                                                                                                                                                |
| 12     | Data or 'str2'             | Data is a fullword, placed by the exit module, that contains the address of the record in a user-owned buffer, for the GET operation.<br><br>'str2' applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string; the string follows. |
| 16     | User-exit work area        | 4-fullword work area provided by the compiler for use by user-exit module.                                                                                                                                                                                                   |
| 32     | Text-name                  | Fullword that contains the address of a null-terminated string that contains the fully qualified text-name. Applies only to FIND.                                                                                                                                            |
| 36     | User exit parameter string | Fullword that contains the address of a four-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.                                                              |

Only the second element of the parameter string array is used for LIBEXIT, to store the length of the LIBEXIT parameter string followed by the parameter string.

## Using INEXIT

When INEXIT is specified, the compiler loads the exit module (*mod1*) during initialization, and invokes the module using the OPEN operation code (op code). This allows the module to prepare its source for processing and then 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 is presented, 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 parameter list to communicate with the exit module. The parameter list consists of 10 fullwords. The return code, data length, and data parameters are placed by the exit module for return to the compiler; and the other items are passed from the compiler to the exit module.

The table shows the contents of the parameter list and a description of each item.

## Using LIBEXIT

When LIBEXIT is specified, the compiler loads the exit module (*mod2*) during initialization. Calls are made to the module by the compiler to obtain copybooks whenever COPY or BASIS statements are encountered.

**Use LIB:** If LIBEXIT is specified, the LIB compiler option must be in effect.

The first call invokes the module with an OPEN op code. This allows the module to 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 exit module is then invoked with a FIND op code. The exit module establishes positioning at the requested text-name (or basis-name) in the specified library-name. This becomes the “active copybook.” When positioning is complete, the exit module passes an appropriate return code to the compiler.

The compiler then 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 is presented, the compiler will issue 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 a request:

- If the requested library-name from the nested COPY statement was not previously opened, the compiler invokes the exit module with an OPEN op code, followed by a FIND for the new text-name.

- If the requested library-name is already open, the compiler issues the FIND op code for the new requested text-name (an OPEN is not issued here).

The compiler does not allow recursive calls 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 or FIND request, it should push its control information concerning the active copybook onto a stack and then complete the requested action (OPEN or FIND). The newly requested text-name (or basis-name) now 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.

At end-of-data for the nested active copybook, the exit module should pop its control information from the stack. The next request from the compiler will be a FIND, so that the exit module can reestablish positioning at the previous active copybook.

The compiler now 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 table shows the contents of the parameter list used for LIBEXIT and a description of each item.

## Using PRTEXT

When PRTEXT is specified, the compiler loads the exit module (*mod3*) during initialization. The exit module is used in place of the SYSPRINT data set.

The compiler invokes the module using the OPEN operation code (op code). This allows the module to prepare its output destination for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has a line to be printed, 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 is ended, 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 table shows the contents of the parameter list used for PRTEXT and a description of each item.

## Using ADEXIT

When ADEXIT is specified, the compiler loads the exit module (*mod4*) during initialization. The exit module is called for each record written to the SYSADATA data set.

The compiler invokes the module using the OPEN operation code (op code). This allows the module to prepare for processing and then pass the status of the OPEN

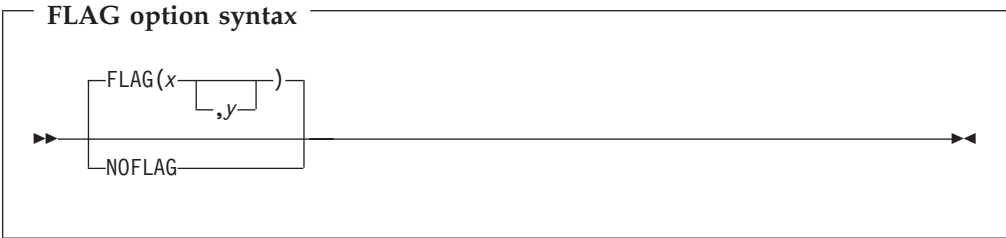
request back to the compiler. Subsequently, each time the compiler has written a SYSADATA record, the exit module is invoked with the PUT op code. The compiler supplies the address and length of the SYSADATA record, and the exit module returns the status of the PUT request to the compiler by a return code.

Before the compilation is ended, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources.

The table shows the contents of the parameter list used for ADEXIT and a description of each item.

# FLAG

Use FLAG(*x*) to produce diagnostic messages at the end of the source listing for errors of a severity level *x* or above.



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==> IGYP2121-S " WS-DATE " was not defined as a data-name. . . .
```

When FLAG(*x*,*y*) is in effect, messages of severity *y* and above are embedded in the listing following the line that caused the message. (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.
- Messages that are produced during processing of the \*CONTROL or \*CBL statement are not embedded in the source listing.

#### RELATED REFERENCES

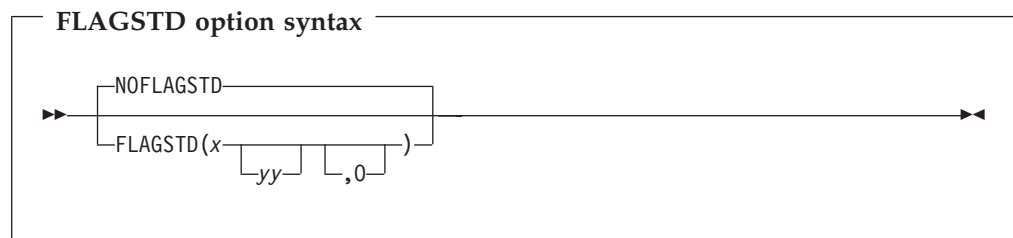
“Messages and listings for compiler-detected errors” on page 205

## FLAGSTD

Use FLAGSTD to specify the level or subset of Standard COBOL 85 to be regarded as conforming, and to get informational messages about Standard COBOL 85 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”)



Default is: NOFLAGSTD

Abbreviations are: None

*x* specifies the subset of Standard COBOL 85 to be regarded as conforming:

- |          |                                                                                                                                                                        |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>M</b> | Language elements that are <i>not</i> from the minimum subset are to be flagged as “nonconforming standard.”                                                           |
| <b>I</b> | Language elements that are <i>not</i> from the minimum or the intermediate subset are to be flagged as “nonconforming standard.”                                       |
| <b>H</b> | 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 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, as well as the message code and text. At the bottom is a summary of the total of the 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                                                                                           |

---

## FLOAT

Specify `FLOAT(NATIVE)` to use the native floating-point representation format of the platform. For COBOL for Windows, the native format is the IEEE format.

#### Float option syntax



Default is: FLOAT(NATIVE)

Abbreviations are: None

FLOAT(HEX) and FLOAT(S390) are synonymous and indicate that COMP-1 and COMP-2 data items are represented consistently with zSeries, that is, in hexadecimal floating-point format:

- Hexadecimal floating-point values are converted to IEEE format before any arithmetic operations (computations or comparisons).
- IEEE floating-point values are converted to hexadecimal format before being stored in floating-point data fields.
- Assignment to a floating-point item is done by converting the source floating-point data (for example, external floating point) to hexadecimal floating point as necessary.

However, COMP-1 and COMP-2 data items defined with the NATIVE keyword on the USAGE clause are not affected by the FLOAT(S390) option. They are always stored in the native format of the platform.

#### RELATED REFERENCES

Appendix B, “zSeries host data format considerations,” on page 567

## LIB

If your program uses COPY, BASIS, or REPLACE statements, the LIB compiler option must be in effect.

#### LIB option syntax



Default is: NOLIB

Abbreviations are: None

For COPY and BASIS statements, you need additionally to define the library or libraries from which the compiler can take the copied code:

- If the library-name is specified with a user-defined word (not a literal), you must set the corresponding environment variable to point to the desired directory and path for the copybook.



- If the library-name is omitted for a COPY statement, you can specify the path to be searched by using the -Ixxx option of the cob2 command.
- If the library-name is specified with a literal, the literal value is treated as the actual path name.

#### RELATED REFERENCES

Chapter 15, “Compiler-directing statements,” on page 273

“Conflicting compiler options” on page 226

---

## 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



```

»»—LINECOUNT(nnn)————««

```

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



```

»»—┌NOLIST┐————««
 │LIST │
 └──────┘

```

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.

**Assembler listing files:** The number of and names of the resulting .asm files depend on the setting of the SEPOBJ option:

**SEP0BJ** The file for the first program in the source file has the name of the source file. The files for all subsequent programs in the source file have the names of the corresponding PROGRAM-IDs.

**NOSEP0BJ**

The one file for all programs in the source file has the name of the source file.

**RELATED TASKS**

“Getting listings” on page 307

**RELATED REFERENCES**

\*CONTROL (\*CBL) statement (*COBOL for Windows 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**

The diagram shows the syntax for the LSTFILE option. It starts with a double arrow pointing right, followed by the text 'LSTFILE(' in a monospace font. To the right of the opening parenthesis is a box containing two options: 'LOCALE' on the top line and 'UTF-8' on the bottom line. This box is enclosed in a bracket on its left side. To the right of the box is a closing parenthesis ')'. This entire sequence is followed by a long horizontal line that ends with a double arrow pointing left.

```
»» LSTFILE([LOCALE
 UTF-8]) _____<<<
```

Default is: LSTFILE(LOCALE)

Abbreviations are: LST

**RELATED REFERENCES**

Chapter 11, “Setting the locale,” on page 179

---

## MAP

Use MAP to produce a listing of the items defined in the DATA DIVISION.

**MAP option syntax**

The diagram shows the syntax for the MAP option. It starts with a double arrow pointing right, followed by a box containing two options: 'NOMAP' on the top line and 'MAP' on the bottom line. This box is enclosed in a bracket on its left side. This entire sequence is followed by a long horizontal line that ends with a double arrow pointing left.

```
»» [NOMAP
 MAP] _____<<<
```

Default is: NOMAP

Abbreviations are: None

The output includes the following items:

- DATA DIVISION map
- Global tables
- Literal pools

- 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
```

By selecting the MAP option, you can also print an embedded MAP report in the source code listing. The condensed MAP information is printed to the right of data-name definitions in the 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 summary is printed first.

“Example: MAP output” on page 310

#### RELATED CONCEPTS

Chapter 18, “Debugging,” on page 297

#### RELATED TASKS

“Getting listings” on page 307

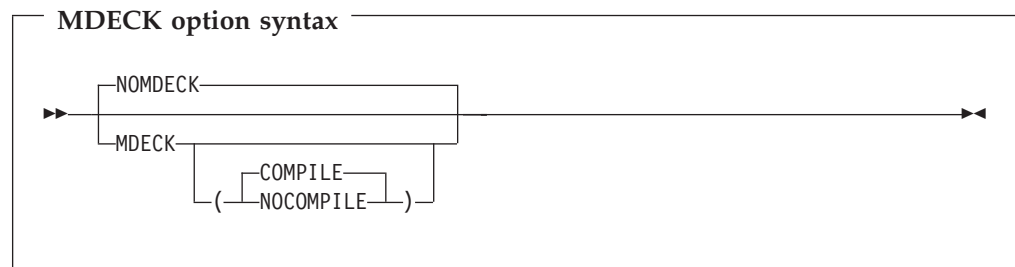
#### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*COBOL for Windows Language Reference*)

## MDECK

The MDECK compiler option specifies that output from library processing (that is, expansion of COPY, BASIS, REPLACE, or 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 an extension of .DEK.



Default is: NOMDECK

Abbreviations are: NOMD, MD, MD(C), MD(NOC)

#### Suboptions:

- When MDECK(COMPILER) 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|NOCOMPILER option.
- When MDECK(NOCOMPILER) is in effect, compilation is terminated after library processing has completed and the expanded source program file has been written. The compiler does no further syntax checking or code generation regardless of the setting of the COMPILE option.

When you specify MDECK with no suboption, MDECK(COMPILER) is implied.

#### Option specification:

You cannot specify MDECK in a PROCESS or CBL statement. You can specify the option only by using:

- A cob2 command option
- The COBOPT environment variable

#### Contents of the MDECK output file:

When you use the MDECK option with the CICS compiler option (integrated CICS translator) or the SQL compiler option (DB2 coprocessor), in general, EXEC CICS or EXEC SQL statements in the COBOL source program are included in the MDECK output as is. However, EXEC SQL INCLUDE statements are expanded in the MDECK output in the same manner as COPY statements.

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

“Conflicting compiler options” on page 226

Chapter 15, “Compiler-directing statements,” on page 273

## NCOLLSEQ

#### NCOLLSEQ option syntax

→ NCOLLSEQ( BINARY  
LOCALE ) →

NCOLLSEQ specifies the collating sequence for comparison of class national operands.

Default is: NCOLLSEQ(BINARY)

Abbreviations are: NCS(L), NCS(BIN), NCS(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 in effect.

#### RELATED TASKS

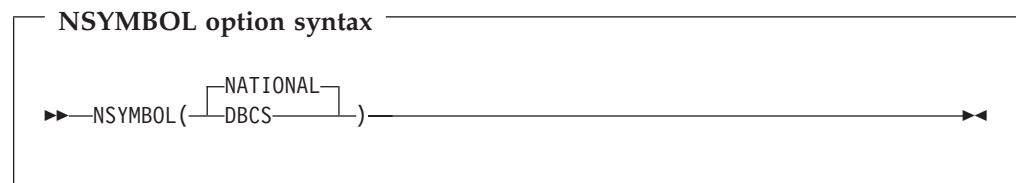
“Comparing two class national operands” on page 172

“Controlling the collating sequence with a locale” on page 185

---

## 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.



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.

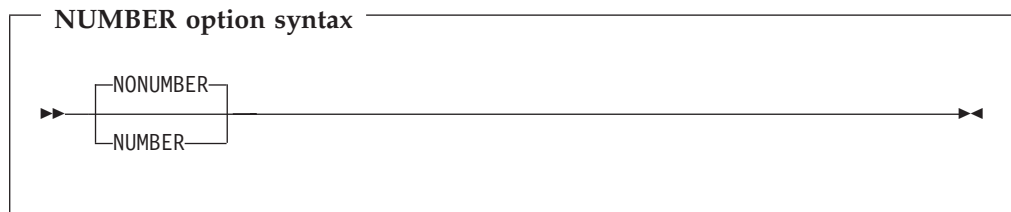
The NSYMBOL(DBCS) option provides compatibility with previous releases of IBM COBOL, and the NSYMBOL(NATIONAL) option makes the handling of the above language elements consistent with Standard COBOL 2002 in this regard.

NSYMBOL(NATIONAL) is recommended for applications that use Unicode data or object-oriented syntax for Java interoperability.

---

## 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.



Default is: NONNUMBER

Abbreviations are: NUM | NONUM

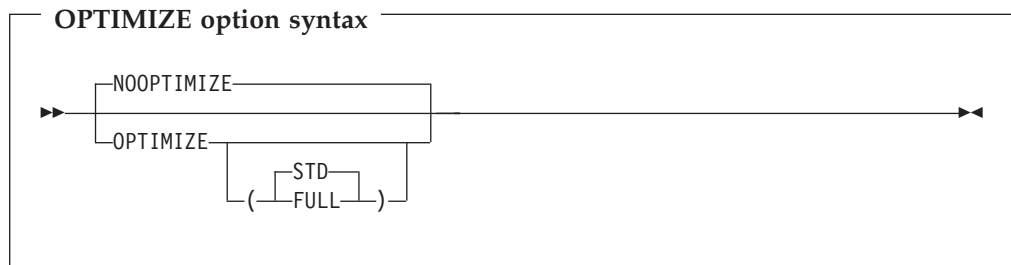
If you request NUMBER, the compiler checks columns 1 through 6 to make sure that they contain only numbers and that the numbers are in numeric collating sequence. (In contrast, SEQUENCE checks the characters in these columns according to EBCDIC collating sequence.) When a line number is found to be out 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 NONNUMBER 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 NONNUMBER 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 and the elimination of computations whose results are never used. Because OPTIMIZE increases compile time and can change the order of statements in your program, you should not use it when debugging.



Default is: NOOPTIMIZE

Abbreviations are: OPT | NOOPT

If OPTIMIZE is specified without any suboptions, OPTIMIZE(STD) will be 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 see if 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.

The OPTIMIZE and TEST options are mutually exclusive; if you use both, OPTIMIZE is ignored.

#### RELATED CONCEPTS

“Optimization” on page 544

#### RELATED REFERENCES

“Conflicting compiler options” on page 226

---

## PGMNAME

The PGMNAME option controls the handling of program-names and entry-point names.

### PGMNAME option syntax

```

 >> PGMNAME (UPPER
MIXED) <<

```

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
  - CANCEL statements
  - SET *procedure-pointer* TO ENTRY statements
  - SET *function-pointer* TO ENTRY statements

## 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, or the hyphen.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last 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, or the hyphen.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

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.

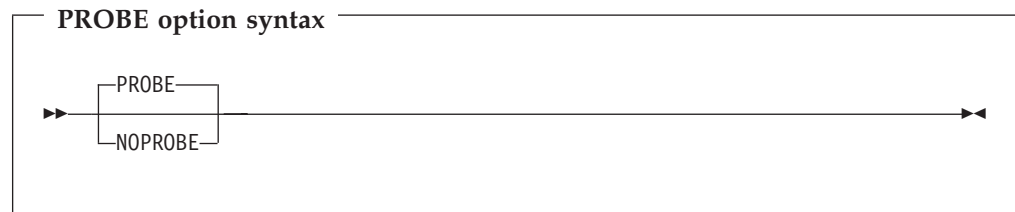
With PGMNAME(MIXED), all program-name definitions must be specified using the literal format of the program-name in the PROGRAM-ID paragraph or ENTRY statement.

---

## PROBE

Use PROBE if the program might be run in a multithreaded environment.





Default is: PROBE

Abbreviations are: None

PROBE requests the generation of stack probes. This extra code causes a protection exception if there is not enough storage available on the stack.

NOPROBE produces more efficient code and is appropriate for nonthreaded environments. However, you should not compile with NOPROBE unless your program meets at least one of the following criteria:

- You use the /STACK linker option to commit more memory than is needed to store all WORKING-STORAGE and LOCAL-STORAGE data items.
- You guarantee that the stack will always be allocated. For example, you write a guard routine to run once at the beginning of each thread and serially access each page, leaving the last page as a guard page.
- Your WORKING-STORAGE and LOCAL-STORAGE data items require less than 1 KB of storage on the stack.

The stack must be large enough for the program to receive exceptions and for the Debug Perspective of Rational Developer for System z to work properly.

#### RELATED TASKS

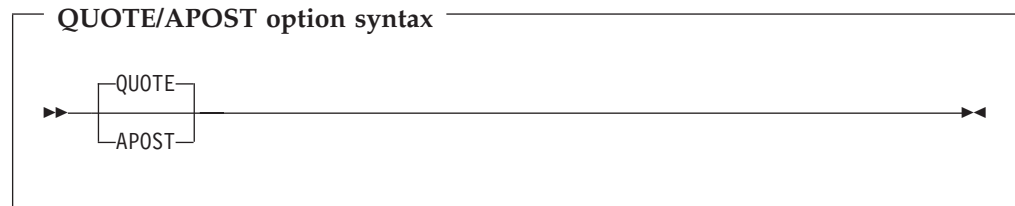
Chapter 30, “Preparing COBOL programs for multithreading,” on page 497

#### RELATED REFERENCES

“/STACK” on page 290

## QUOTE/APOST

Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters. Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more single quotation mark (') characters.



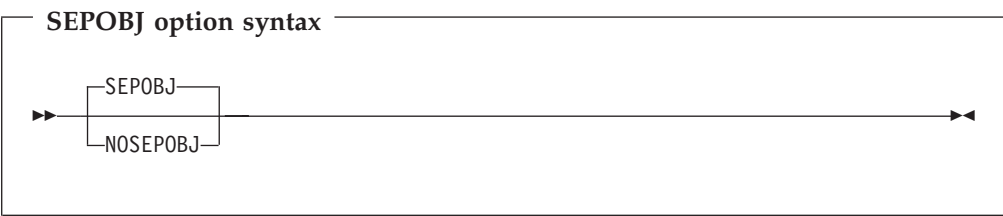
Default is: QUOTE

Abbreviations are: Q|APOST

**Delimiters:** You can use either quotation marks or single quotation marks 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.



Default is: SEPOBJ

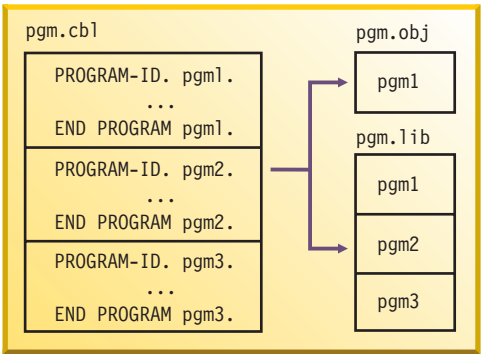
Abbreviations are: None

## Batch compilation

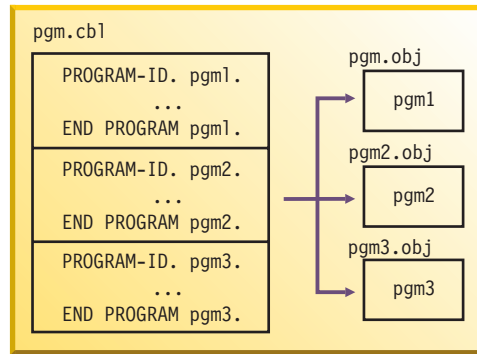
When multiple outermost programs (nonnested programs) are compiled with a single batch invocation of the compiler, the number of files produced for the object program output of the batch compilation depends on the compiler option SEPOBJ.

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 two files (with `NOSEPOBJ`) or three files (with `SEPOBJ`).

### Batch compilation with NOSEPOBJ



### Batch compilation with SEPOBJ



### Usage notes

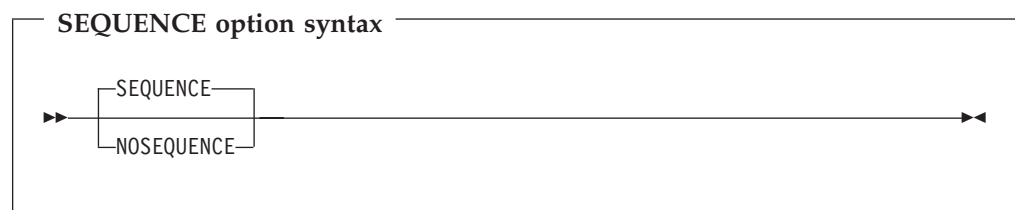
- The SEPOBJ option is required to conform to Standard COBOL 85 where pgm2 or pgm3 in the above example is called using CALL *identifier* from another program.
- If NOSEPOBJ is in effect, the object files are given the name of the source file but with extension .obj or .lib. If SEPOBJ is in effect, the names of the object files (except for the first one) are based on the PROGRAM-ID name with extension .obj.
- 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. For example, if the FAT file system is used, the length in characters of the PROGRAM-ID name must be eight or less *except* when the object file-names are created from the source file-name (as in the case with NOSEPOBJ) as described above.

## 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 EBCDIC 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.



Default is: SEQUENCE

Abbreviations are: SEQ | NOSEQ

If you use COPY statements and SEQUENCE is in effect, be sure that your source program sequence fields and the copybook sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than EBCDIC, collating sequence.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

#### RELATED TASKS

“Finding line sequence problems” on page 302

---

## SIZE

Use SIZE to indicate the amount of main storage to be made available for compilation.

### SIZE option syntax

►►SIZE( nnnnn  
nnnK ) ◄◄

Default is: 8388608 bytes (approximately 8 MB)

Abbreviations are: SZ

*nnnnn* specifies a decimal number, which must be at least 800768.

*nnnK* 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 literals, comments, and DBCS user-defined words.

### SOSI option syntax

►► NOSOSI  
SOSI ◄◄

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 Standard COBOL 85.

**SOSI** With SOSI, COBOL for Windows 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 Windows 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 S0SI option is in effect, in addition to existing rules for COBOL for Windows, 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 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, and the hyphen (-) cannot be included within a DBCS user-defined word. Rules applicable to these characters in single-byte representation apply to those in double-byte representation. For example, the hyphen (-) cannot appear as the first or the last character in a user-defined word.
- For alphanumeric literals that contain X'1E' or X'1F' values, the following rules apply when the S0SI 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 value of the literal unless it is part of a DBCS or alphanumeric literal that is not represented in hexadecimal notation.
  - The SO and SI characters must be paired (with each pair starting with the SO character) with zero or an even number of intervening bytes.
  - The pairs of SO and SI characters cannot be nested.
- 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 S0SI 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 Windows-based workstation.

#### RELATED TASKS

“Handling differences in ASCII DBCS and EBCDIC DBCS strings” on page 449

---

## 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 310

#### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*COBOL for Windows 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.

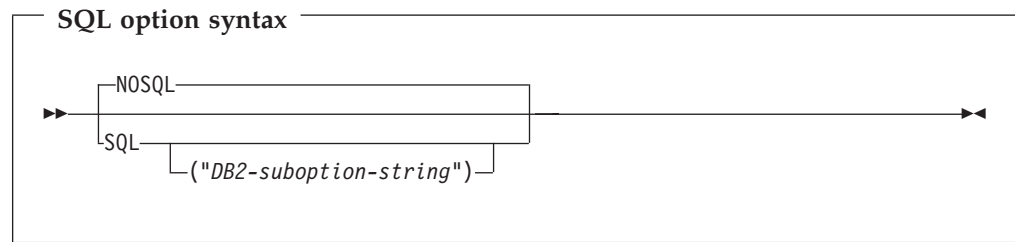
#### RELATED REFERENCES

“SOURCE” on page 261

---

## SQL

Use the SQL compiler option to enable the DB2 coprocessor capability and to specify DB2 suboptions. You must specify the SQL option if your COBOL source program contains SQL statements and it has not been processed by the DB2 precompiler.



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 string delimiter: -q"SQL('options')".

#### RELATED TASKS

Chapter 19, "Programming for a DB2 environment," on page 321

"Compiling with the SQL option" on page 323

"Separating DB2 suboptions" on page 324

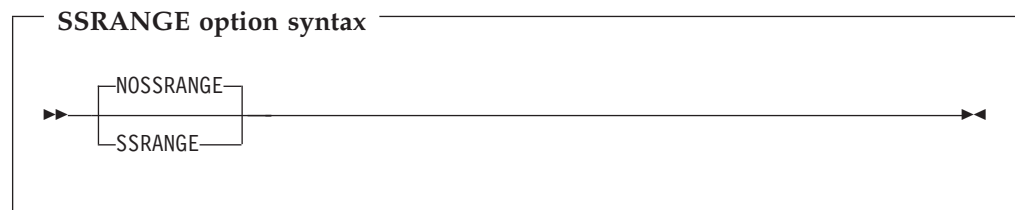
#### RELATED REFERENCES

"Conflicting compiler options" on page 226

## SSRANGE

Use SSRANGE to generate code that checks whether subscripts (including ALL subscripts) or indexes try to reference an area outside the region of the table. Each subscript or index is not individually checked for validity; rather, the effective address is checked to ensure that it does not cause a reference outside the region of the table.

Variable-length items are also checked to ensure that the reference is within their maximum defined length.



Default is: NOSSRANGE

Abbreviations are: SSR | NOSSR

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 length value (if specified) is greater than or equal to 1.
- The starting position and length value (if specified) do not reference an area beyond the end of the subject data item.

If SSRANGE is in effect at compile time, range-checking code is generated. You can inhibit range checking by specifying the CHECK(OFF) runtime option. Doing so leaves range-checking code dormant in the object code. Optionally, the range-checking code can be used to aid in resolving unexpected errors without recompilation.

If an out-of-range condition is detected, an error message is generated and the program is terminated.

**Remember:** Range checking is done only if you compile a program with the SSRANGE option and run it with the CHECK(ON) option.

#### RELATED CONCEPTS

“Reference modifiers” on page 101

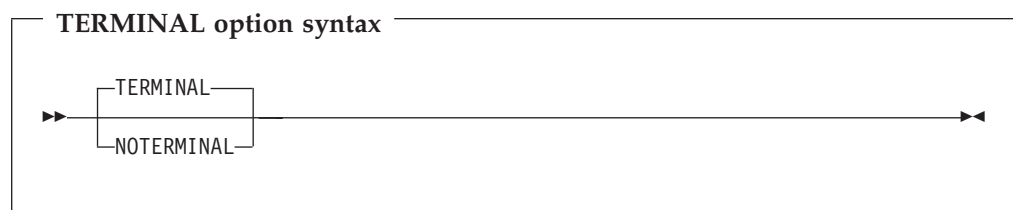
#### RELATED TASKS

“Checking for valid ranges” on page 302

---

## TERMINAL

Use TERMINAL to send progress and diagnostic messages to the display device.



Default is: TERMINAL

Abbreviations are: TERM|NOTERM

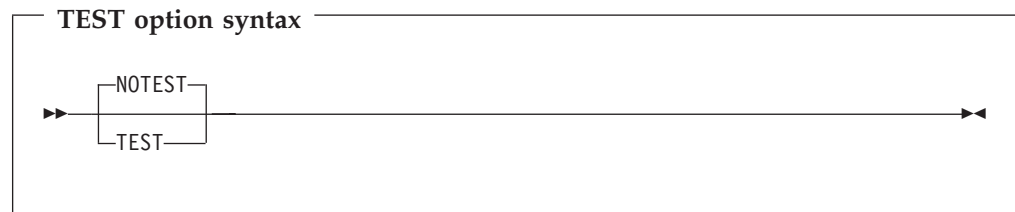
Use NOTERMINAL if you do not want this additional output.

---

## TEST

Use TEST to produce object code that contains symbol and statement information that enables the debugger to perform symbolic source-level debugging.





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 there is limited debugging support.

If you use the WITH DEBUGGING MODE clause, the TEST option will be turned off. TEST will appear in the list of options, but a diagnostic message will be issued to advise you that because of the conflict, TEST was not in effect.

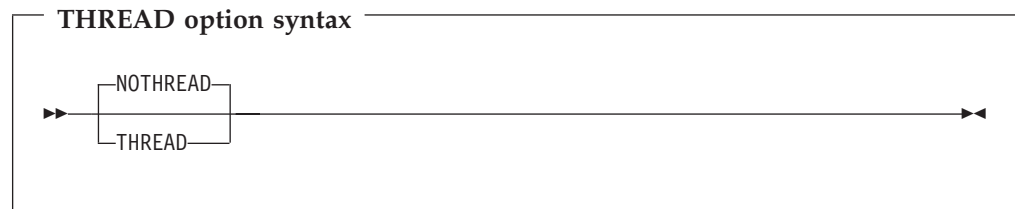
#### RELATED REFERENCES

“Conflicting compiler options” on page 226

---

## THREAD

THREAD indicates that a COBOL application is to be enabled for execution in a run unit that has multiple threads.



Default is: NOTHREAD

Abbreviations are: None

All programs within a run unit must be compiled with the same option: either THREAD or NOTHREAD.

When the THREAD option is in effect, the following items are not supported. If encountered, they are diagnosed as errors:

- ALTER statement
- DEBUG-ITEM special register
- GO TO statement without procedure-name
- INITIAL phrase in PROGRAM-ID clause
- RERUN

- Segmentation module
- STOP *literal* statement
- STOP RUN
- USE FOR DEBUGGING statement

RERUN is flagged as an error with THREAD, but is accepted as a comment with NOTHREAD.

**CICS TXSeries:** The THREAD option is required.

**Performance consideration:** When using the THREAD option, you can expect some runtime performance degradation due to the overhead of serialization logic that is automatically generated.

#### RELATED TASKS

“Compiling and running CICS programs” on page 330

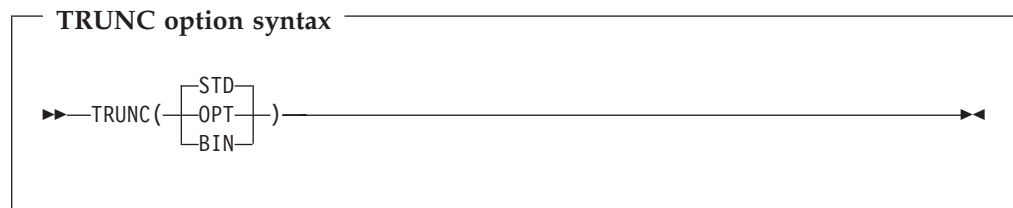
“Compiling from the command line” on page 201

Chapter 30, “Preparing COBOL programs for multithreading,” on page 497

---

## TRUNC

TRUNC affects the way that binary data is truncated during moves and arithmetic operations.



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) were 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, C/C++, and PL/I, 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 as long as 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.

## 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:

| Data item           | Decimal | Hex <sup>1</sup>  | 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    |

1. Values are shown using the default BINARY compiler option.

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 gained by not doing a decimal truncation.

**Assumption:** The preceding example assumes that the BINARY(S390) option is in effect.

## 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:

| Data item                                                     | Decimal    | Hex <sup>1</sup> | Display    |
|---------------------------------------------------------------|------------|------------------|------------|
| Sender                                                        | 1234567891 | D3 02 96 49      | 1234567891 |
| Receiver TRUNC(STD)                                           | 567891     | 53 AA 08 00      | 567891     |
| Receiver TRUNC(OPT)                                           | 567891     | 00 08 AA 53      | 567891     |
| Receiver TRUNC(BIN)                                           | 1234567891 | D3 02 96 49      | 1234567891 |
| 1. Values are shown using the default BINARY compiler option. |            |                  |            |

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.

**Assumption:** The preceding example assumes that BINARY(S390) is in effect.

### RELATED CONCEPTS

“Formats for numeric data” on page 43

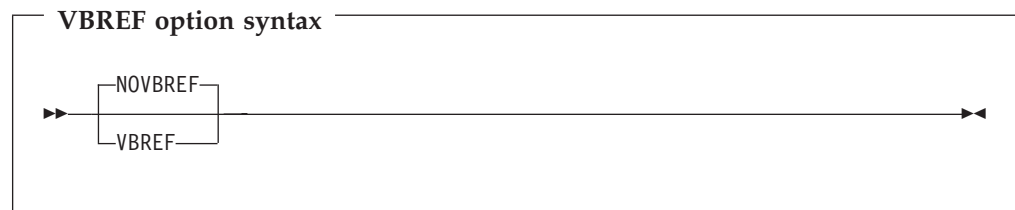
### RELATED TASKS

“Compiling and running CICS programs” on page 330

---

## VBREF

Use VBREF to get a cross-reference among all verb used in the source program and the line numbers in which they are used. VBREF also produces a summary of how many times each verb was used in the program.



Default is: NOVBREF

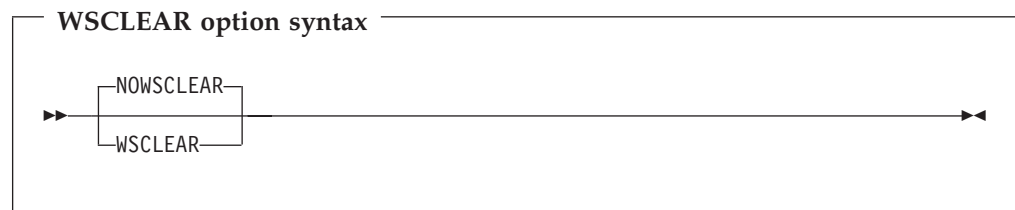
Abbreviations are: None

Use NOVBREF for more efficient compilation.

---

## WSCLEAR

Use WSCLEAR to clear a program's WORKING-STORAGE to binary zeros when the program is initialized. The storage is cleared before any VALUE clauses are applied.



Default is: NOWSCLEAR

Abbreviations are: None

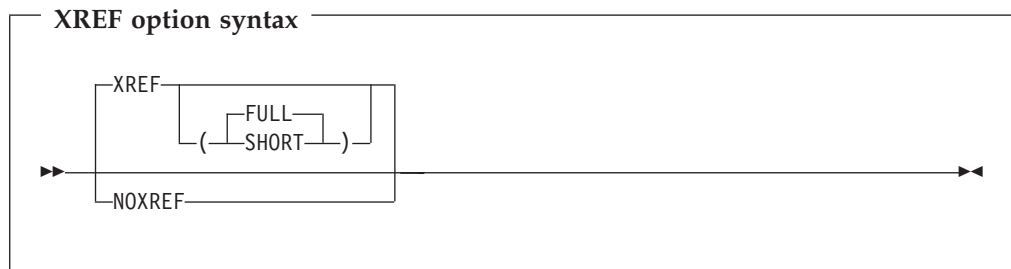
Use NOWSCLEAR to bypass the storage-clearing process.

**Performance considerations:** If you use WSCLEAR and you are concerned about the size or performance of the object program, then you should also use OPTIMIZE(FULL). Doing so instructs the compiler to eliminate all unreferenced data items from the DATA DIVISION, which will speed up initialization.

---

## XREF

Use XREF to get a sorted cross-reference listing.



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) is in effect.

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).

Also included is a section that lists all the program-names that are referenced in your program and the line numbers where they are defined. External program-names are identified.

If you use XREF and SOURCE, 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 18, “Debugging,” on page 297

### RELATED TASKS

“Getting listings” on page 307

---

## 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

►—YEARWINDOW(*base-year*)—◄

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 number between 1900 and 1999.  
This specifies the starting year of a fixed window. For example, YEARWINDOW(1930) indicates a century window of 1930-2029.
- A negative integer from -1 through -99.  
This 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 at the time 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 were 2008, the DATEPROC option were in effect, and you used the YEARWINDOW(1900) option, the program would terminate with an error message.

---

## ZWB

If you compile with 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

►—

|       |
|-------|
| ZWB   |
| NOZWB |

—◄

Default is: ZWB

Abbreviations are: None

I If the zoned decimal item is a scaled item (that is, it contains the symbol P in its PICTURE string), its use in comparisons is 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 source program can give different results, depending on this option setting.

Use NOZWB if you want to test input numeric fields for SPACES.



---

## Chapter 15. Compiler-directing statements

Several compiler-directing statements and one compiler directive help you to direct the compilation of your program.

These are the compiler-directing statements and directive:

### **\*CONTROL (\*CBL) statement**

This compiler-directing statement selectively suppresses output or causes output to be produced. The keywords \*CONTROL and \*CBL are synonymous.

### **>>CALLINTERFACE directive**

This compiler directive specifies the interface convention for calls, including 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.

For example, if PROG1 is a COBOL program compiled with the ENTRYINT(OPTLINK) option, you could use the >>CALLINT OPTLINK directive to change the interface only for the PROG1 call:

```
>>CALLINT OPTLINK DESC
CALL "PROG1" USING PARM1 PARM2.
>>CALLINT
CALL "PROG2" USING PARM1.
```

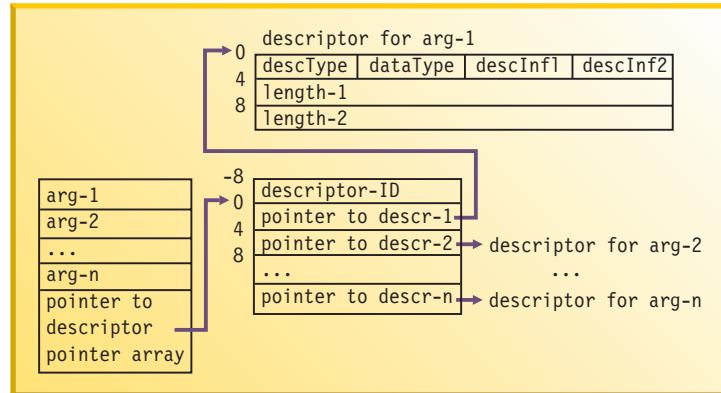
The >>CALLINT directive can be specified anywhere that a COBOL procedure statement can be specified. For example, this is valid syntax:

```
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 with the CALLINT compiler option, but not a convention specified with 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:

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'

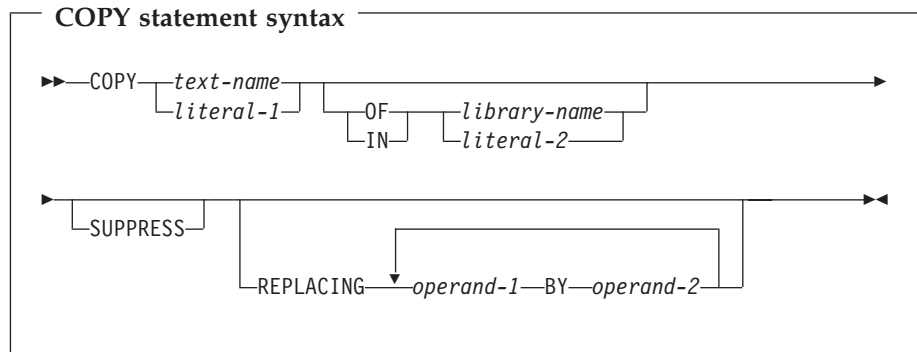
### length-1

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. 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. The following affects *library-name* and *text-name*:

### *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 semicolon (;).

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 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 '\' or the second character is ':'.

For example, these are treated as absolute paths:

COPY "\mycpylib\mytext.cpy" or COPY "d:\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 extension .cpy
2. *text-name* with extension .cbl
3. *text-name* with extension .cob
4. *text-name* with no extension

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)

### **-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 with 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 a \.

For example, COPY MYCOPY OF MYLIB with the settings of:

```
SET MYCOPY=MYPDS(MYMEMBER)
SET MYLIB=MYFILE
```

results in MYFILE\MYPDS(MYMEMBER).

### **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.

#### **RELATED TASKS**

"Changing the header of a source listing" on page 6

"Compiling from the command line" on page 201

"Specifying compiler options with the PROCESS (CBL) statement" on page 203

#### **RELATED REFERENCES**

"cob2 options" on page 206

"Call interface conventions" on page 459

CALLINTERFACE directive (*COBOL for Windows Language Reference*)

CBL (PROCESS) statement (*COBOL for Windows Language Reference*)

\*CONTROL (\*CBL) statement (*COBOL for Windows Language Reference*)

COPY statement (*COBOL for Windows Language Reference*)

## Chapter 16. Linker options

To control the linking process and the files that linking produces, use the linker options. The information for each option shows the syntax and the accepted abbreviations for that option. It also describes the option, its parameters, and its interaction with other parameters.

Table 35. Linker options

| Option                                                       | Description                                               | Default                 | Abbreviation |
|--------------------------------------------------------------|-----------------------------------------------------------|-------------------------|--------------|
| "/?" on page 278                                             | Display a list of valid linker options (same as /HELP)    | None                    | None         |
| "/ALIGNADDR" on page 278                                     | Set address alignment                                     | /A:0x00010000           | /ALIGN       |
| "/ALIGNFILE" on page 279                                     | Set file alignment                                        | /A:512                  | /A           |
| "/BASE" on page 279                                          | Set preferred loading address                             | /BAS:0x00400000         | /BAS         |
| "/CODE" on page 280                                          | Set section attributes for executable                     | /CODE:RX                | None         |
| "/DATA" on page 280                                          | Set section attributes for data                           | /DATA:RW                | None         |
| "/DBGPACK, /NODBGPACK" on page 281                           | Pack debugging information                                | /NODB                   | /DB   /NODB  |
| "/DEBUG, /NODEBUG" on page 281                               | Include debugging information                             | /NODEB                  | /D   /NODEB  |
| "/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH" on page 281 | Search default libraries                                  | /DEF                    | /DEF   /NOD  |
| "/DLL" on page 282                                           | Generate DLL                                              | /EXEC                   | /EXEC        |
| "/ENTRY" on page 282                                         | Specify an entry point in an executable file              | None                    | /EXEC        |
| "/EXECUTABLE" on page 283                                    | Generate executable file                                  | /EXEC                   | /EXEC        |
| "/EXTDICTIONARY, /NOEXTDICTIONARY" on page 283               | Use extended dictionary to search libraries               | /EXT                    | /EXT   /NOE  |
| "/FIXED, /NOFIXED" on page 284                               | Do not relocate the file in memory                        | /NOFI                   | /FI   /NOFI  |
| "/FORCE, /NOFORCE" on page 284                               | Create executable output file even if errors are detected | /NOFO                   | /FO   /NOFO  |
| "/HEAP" on page 285                                          | Set the size of the program heap                          | /HEAP:0x100000,0x1000   | /HEA         |
| "/HELP" on page 285                                          | Display help                                              | None                    | /H           |
| "/INCLUDE" on page 285                                       | Forces a reference to a symbol                            | None                    | /INC         |
| "/INFORMATION, /NOINFORMATION" on page 285                   | Display status of linking process                         | /NOIN                   | /I   /NOIN   |
| "/LINENUMBERS, /NOLINENUMBERS" on page 286                   | Include line numbers in map file                          | /NOLI                   | /L   /NOLI   |
| "/LOGO, /NOLOGO" on page 286                                 | Display logo, echo response file                          | /LO                     | /LO   /NOL   |
| "/MAP, /NOMAP" on page 287                                   | Generate map file                                         | /NOM                    | /M   /NOM    |
| "/OUT" on page 287                                           | Name output file                                          | Name of first .obj file | /O           |
| "/PMTYPE" on page 288                                        | Specify application type                                  | /PMTYPE:VIO             | /PM          |

Table 35. Linker options (continued)

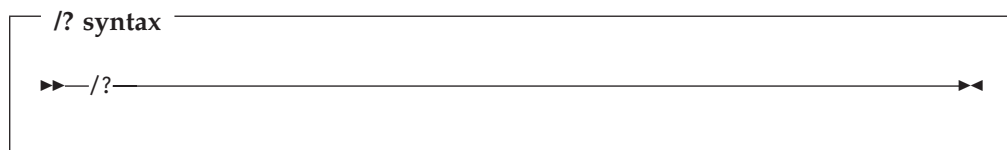
| Option                             | Description                                | Default                    | Abbreviation |
|------------------------------------|--------------------------------------------|----------------------------|--------------|
| "/SECTION" on page 288             | Set attributes for section                 | Set by /CODE and /DATA     | /SEC         |
| "/SEGMENTS" on page 289            | Set maximum number of segments             | /SE:256                    | /SE          |
| "/STACK" on page 290               | Set stack size of application              | /STACK:<br>0x100000,0x1000 | /ST          |
| "/STUB" on page 290                | Specify the name of the DOS stub file      | None                       | /STU         |
| "/SUBSYSTEM" on page 291           | Specify the required subsystem and version | /SUBSYSTEM:<br>WINDOWS,4.0 | /SU          |
| "/VERBOSE, /NOVERBOSE" on page 291 | Display status of linking process          | /NOV                       | /VERB   /NOV |
| "/VERSION" on page 292             | Write a version number in the run file     | /VERSION:0.0               | /VER         |

#### RELATED TASKS

"Linking programs" on page 208

## /?

Use /? to display a list of valid linker options. This option is equivalent to /HELP.

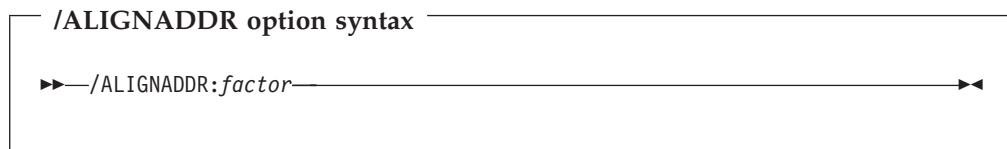


#### RELATED REFERENCES

"/HELP" on page 285

## /ALIGNADDR

Use /ALIGNADDR to set the address alignment for segments.



Default is: /ALIGNADDR:0x00010000

Abbreviation is: /ALIGN

The alignment factor determines the memory addresses for segments in the executable (.EXE) or dynamic link library (.DLL) file. Each segment is assigned to the next unused multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 256 MB. The default factor is 64 KB.

---

## /ALIGNFILE

Use /ALIGNFILE to set the file alignment for segments.

### /ALIGNFILE option syntax

►—/ALIGNFILE:*factor*—◄◄

Default is: /ALIGNFILE:512

Abbreviation is: /A

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 64 KB. The default alignment is 16 bytes.

---

## /BASE

Use /BASE to specify the preferred load address for the first load segment of a run file. Only the last specified address will be used. If no address is specified, the default address will be used.

### /BASE option syntax

►—/BASE:—*address*—◄◄  
    └─@*filename*,*key*—

Default is: /BASE:0x10000

Abbreviations are: /BAS

Specifying @*filename*,*key* in place of *address* bases a set of programs (usually a set of DLLs) so they do not overlap in memory. *filename* is the name of a text file that defines the memory map for a set of files. *key* is a reference to a line in *filename* that begins with the specified key. Each line in the memory-map file has the syntax:

*key address maxsize*, where:

- *key* is a unique name in the file.
- *address* is the location of the memory image in the virtual address space.
- *maxsize* is an amount of memory within which the image must fit.

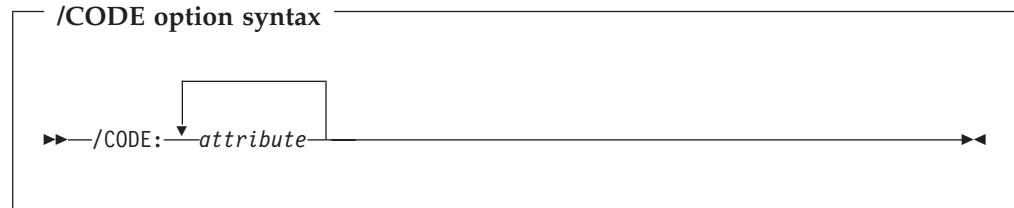
Separate the elements with one or more spaces or tabs. A comment in the memory-map file begins with a semicolon (;) and runs to the end of the line.

The linker issues a warning if the memory image of the program exceeds the specified size.

---

## /CODE

Use /CODE to specify the default attributes for all code sections. You can specify the letters in any order.



Default is: /CODE:RX

Abbreviations are: None

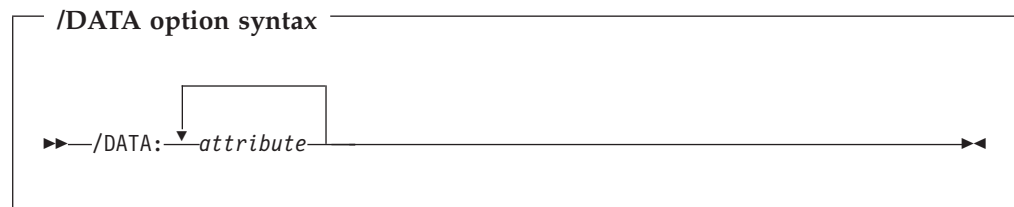
*Table 36. Attributes for code sections*

| Letter                                                  | Attribute          |
|---------------------------------------------------------|--------------------|
| E or X                                                  | EXECUTE            |
| R                                                       | READ               |
| S                                                       | SHARED             |
| W                                                       | WRITE <sup>1</sup> |
| 1. This attribute is not recommended for code segments. |                    |

---

## /DATA

Use /DATA to specify the default attributes for all data sections. You can specify the letters in any order.



Default is: /DATA:RW

Abbreviations are: None

*Table 37. Attributes for data sections*

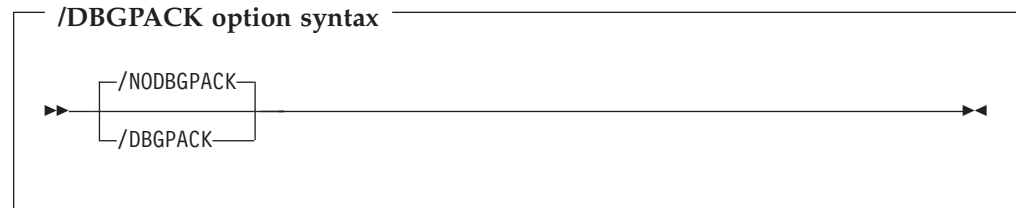
| Letter                                                  | Attribute            |
|---------------------------------------------------------|----------------------|
| E or X                                                  | EXECUTE <sup>1</sup> |
| R                                                       | READ                 |
| S                                                       | SHARED               |
| W                                                       | WRITE                |
| 1. This attribute is not recommended for data segments. |                      |



---

## /DBGPACK, /NODBGPACK

Use /DBGPACK to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.



Default is: /NODBGPACK

Abbreviations are: /DB | /NODB

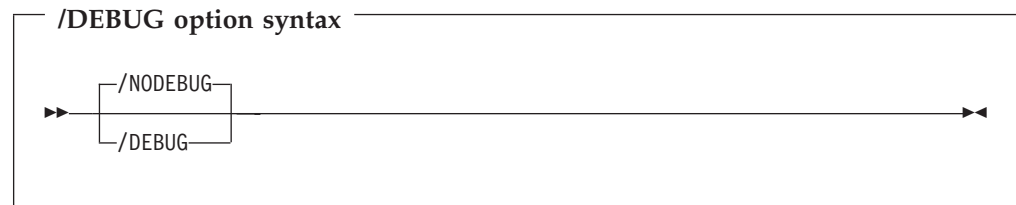
**Performance consideration:** Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can speed up your linking because there is less information to write to the output file.

When you specify /DBGPACK, /DEBUG is turned on by default.

---

## /DEBUG, /NODEBUG

Use /DEBUG to include debug information in the output file, so that you can debug the file with the debugger. The linker embeds symbolic data and line number information in the output file.



Default is: /NODEBUG

Abbreviations are: /D | /NODEB

For debugging, specify the cob2 option -g.

Linking with /DEBUG increases the size of the executable output file.

---

## /DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH

Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references.

**/DEFAULTLIBRARYSEARCH option syntax**

```
graph LR; Start(()) --> Options[/DEFAULTLIBRARYSEARCH
/NODEFAULTLIBRARYSEARCH/]; Options --> Library[:library]; Library --> End(())
```

Abbreviations are: /DEF|/NOD

Use `/NODEFAULTLIBRARYSEARCH` to tell the linker to ignore default libraries when it resolves external references. If you specify a *library* with the option, the linker ignores that default library but searches the rest of the default libraries (and any others that are defined in the object files).

**/DLL**

Diagram illustrating the syntax for the `/DLL` and `/EXECUTABLE` options:

```
graph LR
 Start(()) --> Choice{ }
 Choice --> DLL[/DLL/]
 Choice --> Executable[/EXECUTABLE/]
 Choice --> End(())
```

The diagram shows a horizontal line with a double arrow at the left end and a double arrow at the right end. A bracket is positioned above the line, with the text `/EXECUTABLE` written above the bracket. Another bracket is positioned below the line, with the text `/DLL` written below the bracket.

If you do not specify `/DLL` or `/EXEC`, by default the linker produces an `.EXE` file (`/EXEC`).

## /ENTRY

282 COBOL for Windows Version 7.5 Programming Guide

#### /ENTRY option syntax



Default is: None

Abbreviation is: /EN

---

## /EXECUTABLE

Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default.

#### /DLL and /EXECUTABLE option syntax



Default is: /EXECUTABLE

Abbreviation is: /EXEC

If you specify /EXEC with /DLL, only the last specified of the options takes effect.

If you do not specify /EXEC or /DLL, then by default the linker produces an .EXE file.

---

## /EXTDICTIONARY, /NOEXTDICTIONARY

Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see whether that module requires other modules in the library, and then pulls in the additional modules automatically.

#### /EXTDICTIONARY option syntax



Default is: /EXTDICTIONARY

Abbreviations are: /EXT | /NOE

The linker searches the extended dictionary by default, to speed up the linking process.

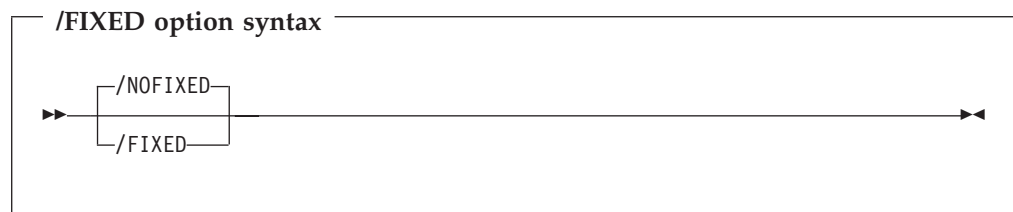
Use `/NOEXTDICTIONARY` to override a definition in a library and replace it with a definition of your own.

If the same symbol is defined in two different places, the linker issues an error. When you link with `/NOEXTDICTIONARY`, the linker searches the dictionary directly instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

---

## **/FIXED, /NOFIXED**

Use `/FIXED` to tell the loader not to relocate a file in memory when the specified base address is not available.



Default is: `/NOFIXED`

Abbreviations are: `/FI` | `/NOFI`

By default, the linker uses `/FIXED` for executable files and `/NOFIXED` for other types of files.

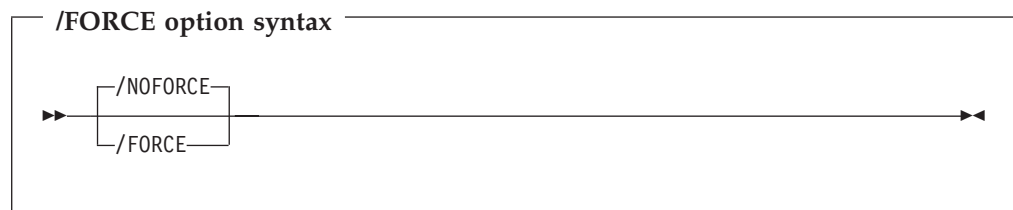
### **RELATED REFERENCES**

`"/BASE"` on page 279

---

## **/FORCE, /NOFORCE**

Use `/FORCE` to produce an executable output file even if there are unresolved externals during the linking process.



Default is: `/NOFORCE`

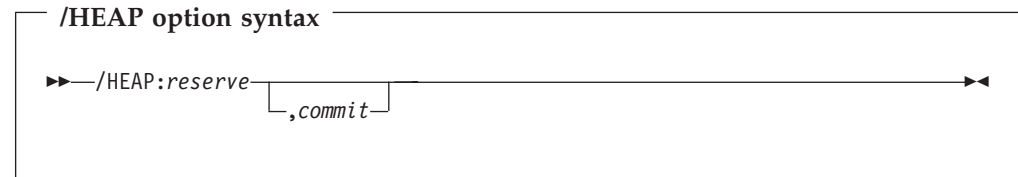
Abbreviations are: `/F0` | `/NOF0`

By default, the linker does not produce an executable output file if it encounters an error.

---

## /HEAP

Use `/HEAP` to set the size of the program heap in bytes. The *reserve* argument sets the total reserved virtual address space. The *commit* argument sets the amount of physical memory to allocate initially.



Default is: `/HEAP:0x100000,0x1000`

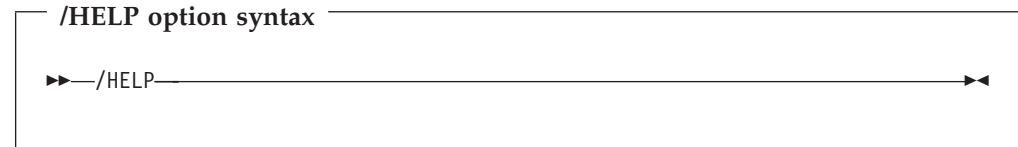
Abbreviation is: `/HEA`

**Performance consideration:** When *commit* is less than *reserve*, memory demands are reduced, but execution time can be slower.

---

## /HELP

Use `/HELP` to display a list of valid linker options. This option is equivalent to `/?`.



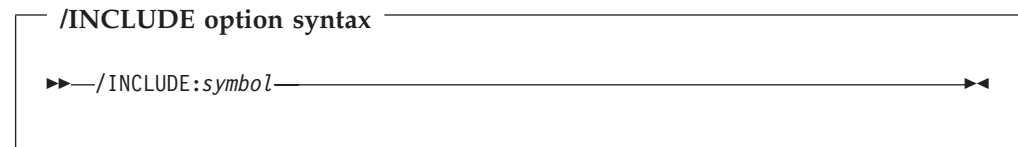
Default is: None

Abbreviation is: `/H`

---

## /INCLUDE

Use `/INCLUDE` to force a reference to a symbol. The linker searches for an object module that defines the symbol.



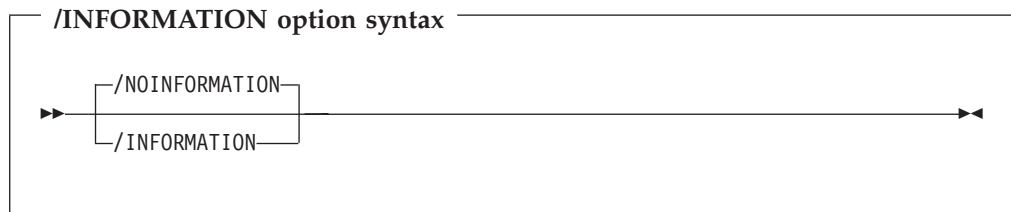
Default is: None

Abbreviation is: `/INC`

---

## /INFORMATION, /NOINFORMATION

`/INFORMATION` is functionally equivalent to `/VERBOSE`. `/VERBOSE`, however, is preferred.



Default is: /NOINFORMATION

Abbreviations are: /I | /NOIN

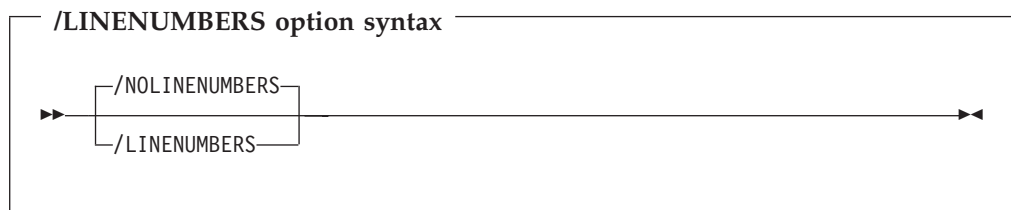
#### RELATED REFERENCES

“/VERBOSE, /NOVERBOSE” on page 291

---

## /LINENUMBERS, /NOLINENUMBERS

Use /LINENUMBERS to include source-file line numbers and associated addresses in the map file. For this option to take effect, there must already be line-number information in the object files that you are linking.



Default is: /NOLINENUMBERS

Abbreviations are: /L | /NOLI

When you compile, use the cob2 option -qNUMBER to include line numbers in the object file (or the cob2 option -g to include all debugging information).

If you provide the linker an object file that does not have line-number information, the /LINENUMBERS option has no effect.

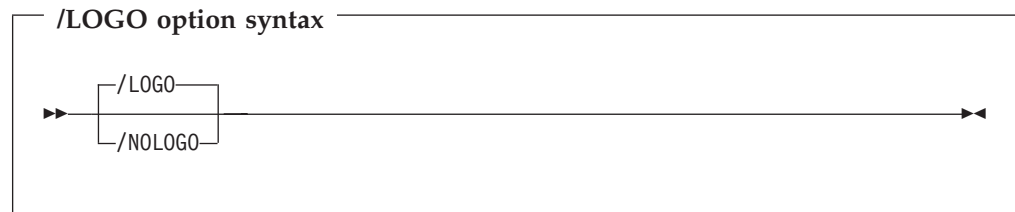
The /LINENUMBERS option forces the linker to create a map file even if you specify /NOMAP.

By default, the map file is given the same name as the output file, with the extension .map. You can override the default name by specifying a map-file name.

---

## /LOGO, /NOLOGO

Use /NOLOGO to suppress the product information that appears when the linker starts.



Default is: /LOGO

Abbreviations are: /LO | /NOL

Specify /NOLOGO before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

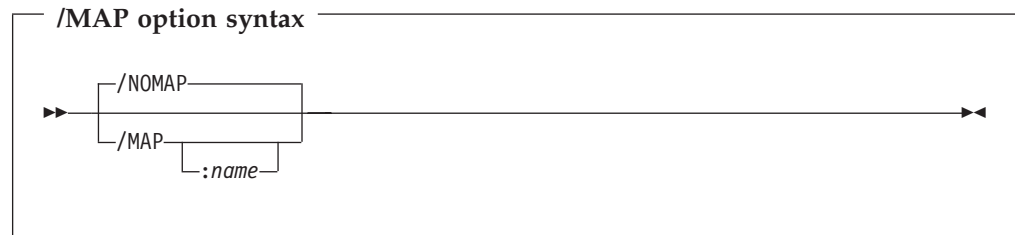
By default, the linker displays product information at the start of the linking process, and displays the contents of the response file as it reads the file.

Unlike most other linker options, /LOGO and /NOLOGO cannot be included in a response file. You must set them on the command line.

---

## /MAP, /NOMAP

Use /MAP to generate a map file called *name*. The map file lists the composition of each segment and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name and in order of address.



Default is: /NOMAP

Abbreviations are: /M | /NOM

If you do not specify a directory, the map file is generated in the current working directory. If you do not specify *name*, the map file has the same name as the executable output file but has the extension .map.

By default, the linker does not produce a map file.

---

## /OUT

Use /OUT to specify a name for the executable output file.

#### **/OUT option syntax**

►► /OUT:*name* ◀◀

Default is: Name of the first .OBJ file with appropriate extension.

Abbreviation is: /O

If you do not provide an extension with *name*, the linker provides an extension based on the type of file that you are producing, as shown in the table below.

| File produced        | Default extension |
|----------------------|-------------------|
| Executable program   | .EXE              |
| Dynamic link library | .DLL              |

If you do not use the /OUT option, the linker uses the file-name of the first object file that you specified, with the appropriate extension.

---

## **/PMTYPE**

Use /PMTYPE to specify the type of executable file that the linker should produce. Do not use this option when generating dynamic link libraries (DLLs).

#### **/PMTYPE option syntax**

►► /PMTYPE:*type* ◀◀

Default is: /PMTYPE:VIO

Abbreviation is: /PM

Specify one of the following types:

**PM**      The executable must be run in a window.

**VIO**     The executable can be run either in a window or in a full screen.

**NOVIO**   The executable must not be run in a window; it must use a full screen.

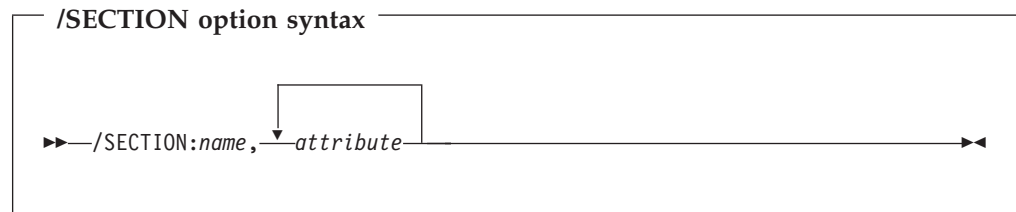
If you set /PMTYPE for any file type other than executable files, the option is ignored.

---

## **/SECTION**

Use /SECTION to specify memory-protection attributes for the *name* section. (*name* is case sensitive).





Default is: Depends on segment type

Abbreviation is: /SEC

You can specify the attributes that are shown in the table below.

**Table 38. Attributes for named sections**

| Letter                                                                                                             | Sets attribute       |
|--------------------------------------------------------------------------------------------------------------------|----------------------|
| E or X                                                                                                             | EXECUTE <sup>2</sup> |
| R                                                                                                                  | READ                 |
| S                                                                                                                  | SHARED               |
| W                                                                                                                  | WRITE <sup>1</sup>   |
| 1. This attribute is not recommended for code segments.<br>2. This attribute is not recommended for data segments. |                      |

For example, the following code sets the READ and SHARED attributes, but not the EXECUTE or WRITE attributes, for the section dseg1 in an .EXE file:

```
/SEC:dseg1,RS
```

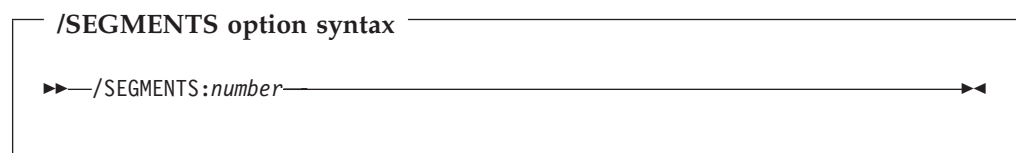
Sections are assigned attributes by default, as shown in the table below.

**Table 39. Default attributes for sections**

| Segment            | Default attributes           |
|--------------------|------------------------------|
| Code sections      | EXECUTE, READ (ER)           |
| Data sections      | READ, WRITE (RW), not shared |
| CONST32_RO section | READ, SHARED (RS)            |

## /SEGMENTS

Use /SEGMENTS to set the number of sections that a program can have. You can set *number* to any value in the range 1 to 16375, in decimal, octal, or hexadecimal format.



Default is: /SEGMENTS:256

Abbreviation is: /SE

For each section, the linker must allocate space to keep track of section information. By using a relatively low limit as a default (256), the linker is able to link faster and allocate less storage space.

When you set the section limit higher than 256, the linker allocates more space for section information. This results in slower linking, but allows you to link programs that have a large number of sections.

For programs with fewer than 256 sections, you can improve link time and reduce linker storage requirements by setting *number* to the actual number of sections in the program.

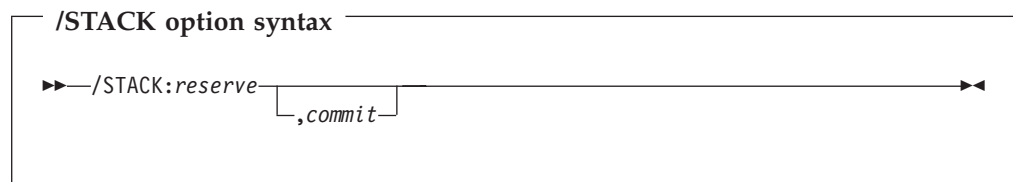
#### RELATED TASKS

"Specifying linker options" on page 210

---

## /STACK

Use /STACK to set the stack size (in bytes) of your program.



Default is: /STACK:0x100000,0x1000

Abbreviation is: /ST

The size must be an even number from 0 to 0xFffffffe. If you specify an odd number, it is rounded up to the next even number.

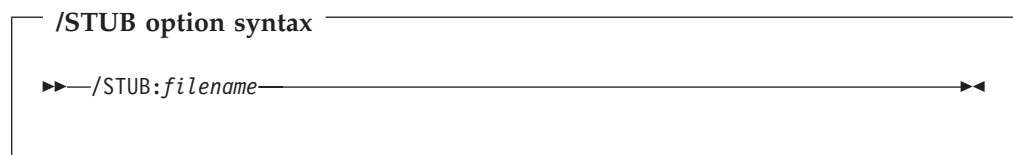
*reserve* indicates the total reserved virtual address space. *commit* sets the amount of physical memory to allocate initially.

**Performance considerations:** When *commit* is less than *reserve*, memory demands are reduced, although execution time may be slower.

---

## /STUB

Use /STUB to specify the name of the executable at the beginning of the output file that is created.



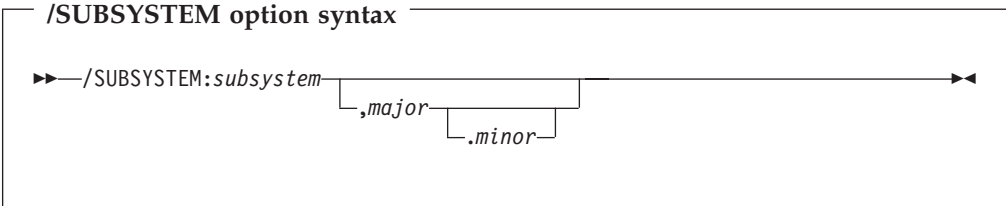
Default is: None

Abbreviation is: /STU

By default, the linker defines its own stub.

## /SUBSYSTEM

Use /SUBSYSTEM to specify the subsystem and version that are required to run the program.



Default is: /SUBSYSTEM:WINDOWS,4.0

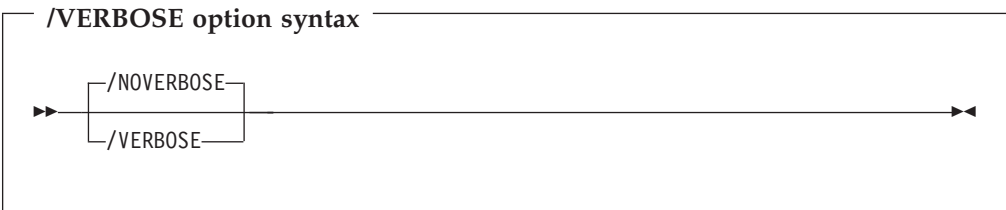
Abbreviation is: /SU

The *major* and *minor* arguments are optional and specify the minimum required version of the subsystem. The *major* and *minor* arguments are integers in the range 0 to 65535.

| Subsystem | Major.minor | Description                                                                 |
|-----------|-------------|-----------------------------------------------------------------------------|
| WINDOWS   | 3.10        | A graphical application that uses the Graphical Device Interface (GDI) API. |
| CONSOLE   | 3.10        | A character-mode application that uses the Console API.                     |

## /VERBOSE, /NOVERBOSE

Use /VERBOSE to have the linker display information during the linking process, including the phase of linking and the names and paths of the object files being linked.



Default is: /NOVERBOSE

Abbreviations are: /VERB | /NOV

If you have trouble linking because the linker is finding the wrong files or finding them in the wrong order, use /VERBOSE to determine the locations of the object files and the order in which they are linked.

The output from this option is sent to stdout. You can redirect the output to a file using Windows redirection symbols.

/VERBOSE is the same as /INFORMATION, but is the preferred form.

#### RELATED REFERENCES

“/INFORMATION, /NOINFORMATION” on page 285

---

## /VERSION

Use /VERSION to write a version number in the header of the run file.

### /VERSION option syntax

The diagram illustrates the syntax for the /VERSION option. It shows the command starting with a double arrow pointing to the text "/VERSION:". This is followed by the word "major" in italics, which is enclosed in a rectangular box. To the right of this box is a period "." followed by the word "minor" in italics, which is also enclosed in a rectangular box. A long horizontal line with double arrows at both ends extends from the right side of the "minor" box across the rest of the diagram.

```
►► /VERSION: major . minor
```

Default is: /VERSION:0.0

Abbreviation is: /VER

The *major* and *minor* arguments are integers in the range 0 to 65535.

## Chapter 17. Runtime options

The runtime options shown in the table below are supported.

Table 40. Runtime options

| Option                 | Description                                                                                                                                        | Default        | Abbreviation |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|----------------|--------------|
| "CHECK"                | Flags checking errors                                                                                                                              | CHECK(ON)      | CH           |
| "DEBUG" on page 294    | Specifies whether the COBOL debugging sections specified by the USE FOR DEBUGGING declarative are active                                           | NODEBUG        | None         |
| "ERRCOUNT" on page 294 | Specifies how many conditions of severity 1 (W-level) can occur before the run unit terminates abnormally                                          | ERRCOUNT(20)   | None         |
| "FILESYS" on page 294  | Specifies the file system to use for files for which no explicit file system selections are made, either through ASSIGN or an environment variable | FILESYS(STL)   | None         |
| "TRAP" on page 295     | Indicates whether COBOL intercepts exceptions                                                                                                      | TRAP(ON)       | None         |
| "UPSI" on page 296     | Sets the eight UPSI switches on or off for applications that use COBOL routines                                                                    | UPSI(00000000) | None         |

### CHECK

CHECK causes checking errors to be flagged. In COBOL, index, subscript, and reference-modification ranges can cause checking errors.

#### CHECK option syntax

►►CHECK()►►

Default is: CHECK(ON).

Abbreviation is: CH

**ON** Specifies that runtime checking is performed.

**OFF** Specifies that runtime checking is not performed.

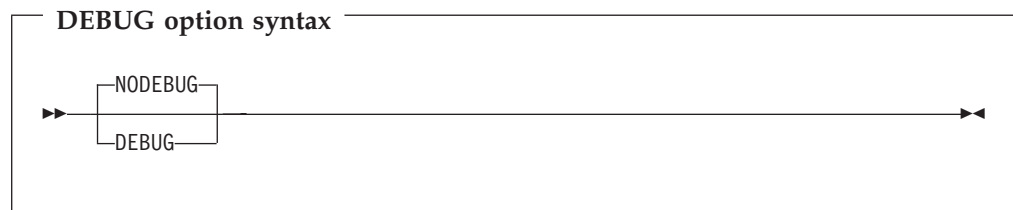
**Usage note:** CHECK(ON) has no effect if NOSSRANGE was in effect during compilation.

**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.



Default is: NODEBUG.

**DEBUG** Activates the debugging sections.

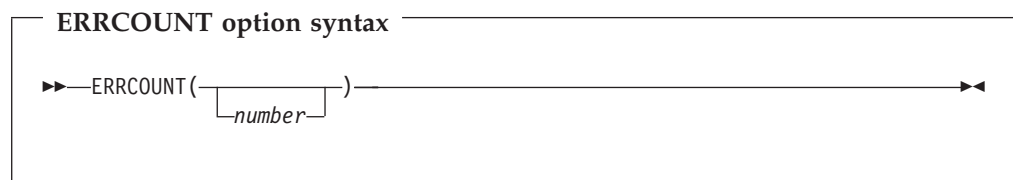
**NODEBUG**  
Suppresses the debugging sections.

**Performance consideration:** To improve performance, use this option only while debugging.

---

## ERRCOUNT

ERRCOUNT specifies how many severity 1 (W-level) conditions can occur before the run unit terminates abnormally.



Default: ERRCOUNT(20).

*number* is the number of severity-1 conditions per individual thread that can occur while this run unit is running. If the number of conditions exceeds *number*, the run unit terminates abnormally.

Any severity-2 (E-level) or higher condition 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 selection is made either through an ASSIGN statement or an environment variable. The option applies to sequential, relative, and indexed files.

#### FILESYS option syntax

```
➡➡ FILESYS([STL
 BTR
 RSD]) ➡➡
```

Default is: FILESYS(STL).

**BTR** The file system is Btrieve (Pervasive.SQL).

**STL** The file system is STL.

**RSD** The file system is RSD.

#### RELATED TASKS

“Identifying files” on page 113

#### RELATED REFERENCES

“Runtime environment variables” on page 196

---

## TRAP

TRAP indicates whether COBOL intercepts exceptions.

#### TRAP option syntax

```
➡➡ TRAP([ON
 OFF]) ➡➡
```

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

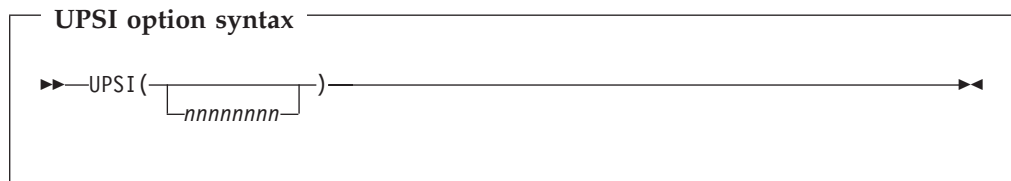
- Use TRAP(OFF) only when you need to analyze a program exception before COBOL handles it.
- When you specify TRAP(OFF) in a non-CICS environment, no exception handlers are established.
- Running with TRAP(OFF) (for exception diagnosis purposes) can cause many side effects because COBOL requires TRAP(ON). 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.



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 18. Debugging

You can choose from two approaches to determine the cause of problems in program 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.

For interactive debugging, you can use the Debug Perspective of Rational Developer for System z, the graphical debugging interface.

### RELATED TASKS

- “Debugging with source language”
- “Debugging using compiler options” on page 301
- “Using the debugger” on page 306
- “Getting listings” on page 307
- “Debugging user exits” on page 317
- “Debugging assembler routines” on page 318

---

## 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”
- “Finding and handling input-output errors” on page 298
- “Validating data” on page 299
- “Finding uninitialized data” on page 299
- “Generating information about procedures” on page 299

### RELATED REFERENCES

Source language debugging (*COBOL for Windows 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.

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 19

#### RELATED REFERENCES

`DISPLAY` statement (*COBOL for Windows 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 148

“Using file status keys” on page 148

#### RELATED REFERENCES

Status key (*COBOL for Windows 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 86

“Testing for valid DBCS characters” on page 177

#### RELATED REFERENCES

Class condition (*COBOL for Windows Language Reference*)

## Finding 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 Windows Language Reference*)

SET statement (*COBOL for Windows 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 routine 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. However, you cannot use the USE FOR DEBUGGING declarative in a program that you compile with the THREAD option.

The WITH DEBUGGING MODE clause and the TEST compiler option are mutually exclusive. If both are present, the WITH DEBUGGING MODE clause takes precedence.

“Example: USE FOR DEBUGGING”

#### RELATED REFERENCES

SOURCE-COMPUTER paragraph (*COBOL for Windows Language Reference*)

Debugging lines (*COBOL for Windows Language Reference*)

Debugging sections (*COBOL for Windows Language Reference*)

DEBUGGING declarative (*COBOL for Windows 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 use find these 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)
- Verb 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 302

"Finding line sequence problems" on page 302

"Checking for valid ranges" on page 302

"Selecting the level of error to be diagnosed" on page 303

"Finding program entity definitions and references" on page 305

"Listing data items" on page 306

"Getting listings" on page 307

"Using the debugger" on page 306

## 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 parameters. If you also specify the SOURCE option, the compiler produces a listing.

The following compiler options are suppressed when you use NOCOMPILE without parameters: LIST, OBJECT, OPTIMIZE, SSRANGE, and TEST.

**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.

## 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.

## 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 desired 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 address is generated 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 263

“Performance-related compiler options” on page 546

## 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 41. 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”

#### RELATED TASKS

“Generating a list of compiler error messages” on page 204

#### RELATED REFERENCES

“FLAG” on page 245

“Messages and listings for compiler-detected errors” on page 205

“Severity codes for compiler error messages” on page 204

### **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==> IGYP52052-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==> IGYP52121-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==> IGYP52121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
001013 accept ws-time from time UND
==001013==> IGYP52121-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==> IGYP52121-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 IGYS1050-E File "LOCATION-FILE" contained no data record descriptions.
The file definition was discarded.
899 IGYP52052-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 IGYP52121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
Same message on line: 1012
1013 IGYP52121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
Same message on line: 1014
1015 IGYP52053-S An error was found in the definition of file "LOCATION-FILE".
This input/output statement was discarded.
Same message on line: 1027
1026 IGYP52121-S "LOC-CODE" was not defined as a data-name. The statement was discarded.
1209 IGYP52121-S "COMMUTER-SHIFT" was not defined as a data-name. The statement was discarded.
Same message on line: 1230
1210 IGYP52121-S "COMMUTER-HOME-CODE" was not defined as a data-name. The statement was discarded.
Same message on line: 1231
1212 IGYP52121-S "COMMUTER-NAME" was not defined as a data-name. The statement was discarded.
Same message on line: 1233
1213 IGYP52121-S "COMMUTER-INITIALS" was not defined as a data-name. The statement was discarded.
Same message on line: 1234
1223 IGYP52121-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.IBM-850 (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. The sorted cross-reference includes the line number where the entity 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.

User-defined words in your program are sorted using the locale that is active. Hence, the collating sequence determines the order for the cross-reference listing, including multibyte words.

Group names in a MOVE CORRESPONDING statement are listed in the XREF listing. The cross-reference listing includes the group names and all the elementary names involved in the move.

“Example: XREF output - data-name cross-references” on page 314

“Example: XREF output - program-name cross-references” on page 315

“Example: embedded cross-reference” on page 315

#### RELATED TASKS

“Getting listings” on page 307

#### RELATED REFERENCES

“XREF” on page 269

## Listing data items

Use the MAP compiler option to produce a listing of the DATA DIVISION items and all implicitly declared items.

When you use the MAP option, an embedded MAP summary that contains condensed MAP information is generated to the right of the COBOL source data declaration. 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
*CBL MAP
```

“Example: MAP output” on page 310

#### RELATED TASKS

“Getting listings” on page 307

#### RELATED REFERENCES

“MAP” on page 250

## Using the debugger

Rational Developer for System z provides a debugger. Use the TEST compiler option to prepare your COBOL program so that you can step through the executable program with the debugger.

Alternatively, you can use the -g option of the cob2 command to prepare your program to use the debugger.

#### RELATED TASKS

“Compiling from the command line” on page 201

#### RELATED REFERENCES

“TEST” on page 264

## 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 42. Using compiler options to get listings*

| Use                                                                                                                                                                                                                 | Listing                                 | Contents                                                                                                                                                                                                                                                             | Compiler option                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------|
| <p>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</p> <p>To check the locale in effect during compilation</p> | Short listing                           | <ul style="list-style-type: none"> <li>List of options in effect for the program</li> <li>Statistics about the content of the program</li> <li>D diagnostic messages about the compilation<sup>1</sup></li> </ul> <p>Locale line that shows the locale in effect</p> | NOSOURCE, NOXREF, NOVBREF, NOMAP, NOLIST |
| 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 261                     |
| 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              | <p>All DATA DIVISION items and all implicitly declared items</p> <p>Embedded map summary (in the right margin of the listing for lines in the DATA DIVISION that contain data declarations)</p> <p>Nested program map (if the program contains nested programs)</p>  | “MAP” on page 250 <sup>2</sup>           |
| To find where a name is defined, referenced, or modified; to determine the context (such as whether a verb was used in a PERFORM block) in which a procedure is referenced                                          | Sorted cross-reference listing of names | <p>Data-names, procedure-names, and program-names; references to these names</p> <p>Embedded modified cross-reference: provides the line number where the data-name or procedure-name was defined</p>                                                                | “XREF” on page 269 <sup>2,3</sup>        |

Table 42. Using compiler options to get listings (continued)

| Use                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | Listing                                                                          | Contents                                                                                  | Compiler option                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-----------------------------------|
| To find the failing verb in a program or the address in storage of a data item that was moved during the program                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | PROCEDURE DIVISION code and assembler code produced by the compiler <sup>3</sup> | Generated code                                                                            | "LIST" on page 249 <sup>2,4</sup> |
| To find an instance of a certain verb                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Alphabetic listing of verbs                                                      | Each verb used, number of times each verb was used, line numbers where each verb was used | "VBREF" on page 269               |
| <ol style="list-style-type: none"> <li>1. To eliminate messages, turn off the options (such as FLAG) that govern the level of compile diagnostic information.</li> <li>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.</li> <li>3. The context of the procedure reference is indicated by the characters preceding the line number.</li> <li>4. The assembler listing is written to a file that has the same name as the source program but with the extension .asm, except for batch compiles that use the SEPOBJ option.</li> </ol> |                                                                                  |                                                                                           |                                   |

"Example: short listing"

"Example: SOURCE and NUMBER output" on page 310

"Example: MAP output" on page 310

"Example: embedded map summary" on page 311

"Example: nested program map" on page 313

"Example: XREF output - data-name cross-references" on page 314

"Example: XREF output - program-name cross-references" on page 315

"Example: embedded cross-reference" on page 315

"Example: VBREF compiler output" on page 316

#### RELATED TASKS

"Generating a list of compiler error messages" on page 204

#### RELATED REFERENCES

"Messages and listings for compiler-detected errors" on page 205

"SEPOBJ" on page 258

## Example: short listing

The note numbers shown in the following listing correspond to the numbered explanations that follow the listing. For illustrative purposes, some errors that cause diagnostic messages were deliberately introduced.

```

Invocation parameters: (1)
NOADATA
PROCESS (CBL) statements:
CBL NOSOURCE,NOXREF,NOVBREF,NOMAP,NOLIST (2) IGY00010
Options in effect: (3)
NOADATA
QUOTE
ARITH (COMPAT)
BINARY (NATIVE)
CALLINT (SYSTEM, NODESCRIPTOR)
CHAR (NATIVE)
NOCICS
COLLSEQ (BINARY)
NOCOMPILE (S)
NOCURRENCY
NODATEPROC
NODIAGTRUNC
NODYNAM
ENTRYINT (SYSTEM)
NOEXIT
FLAG (I)
NOFLAGSTD
FLOAT (NATIVE)
LIB
LINECOUNT (60)
NOLIST
LSTFILE (LOCALE)
NOMAP
NOMDECK
NCOLLSEQ (BINARY)
NSYMBOL (NATIONAL)
NONUMBER
NOOPTIMIZE
PGMNAME (LONGUPPER)
PROBE
NOPROFILE
SEPOBJ
SEQUENCE
NOSOSI
SIZE (2097152)
NOSOURCE
SPACE (1)
SQL
NOSSRANGE
TERM
NOTEST
NOTHREAD
TRUNC (STD)
NOVBREF
NOWORD
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
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.
993 IGYPS2121-S Same message on line: 983
"WS-DATE" was not defined as a data-name. The statement was discarded.
995 IGYPS2121-S Same message on line: 994
"WS-TIME" was not defined as a data-name. The statement was discarded.
997 IGYPS2053-S Same message on line: 996
An error was found in the definition of file "LOCATION-FILE". This input/output statement
was discarded.
1008 IGYPS2121-S Same message on line: 1009
"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 1 18
* Statistics for COBOL program SLISTING: (6)
Source records = 1765
Data Division statements = 277
Procedure Division statements = 513
Locale = en.US.IBM-850 (7)
End of compilation 1, program SLISTING, highest severity: Severe. (8)
Return code 12

```

- (1) Message about options passed to the compiler at compiler invocation. This message does not appear if no options were passed.
- (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.

| LineID<br>(2) | PL<br>(3) | SL<br>(4)    | -----*A-1-B--*-----2-----3-----4-----5-----6-----7- -----8 | Cross-Reference (1) |
|---------------|-----------|--------------|------------------------------------------------------------|---------------------|
|               |           | 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-master-file.                        |                     |
|               |           | 088150       | display "perform 050-create-stl-master 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)<br>Source<br>LineID | (3) (4)<br>Hierarchy and<br>Data Name   | (5)<br>Length(Displacement) | (6)<br>Data Type | (7)<br>Data Def<br>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.

## RELATED REFERENCES

“Terms and symbols used in MAP output” on page 312

## 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 FD COMMUTER-FILE
000180 record 80 characters.
000181 01 commuter-record.
000182 05 commuter-key PIC x(16).
000183 05 filler PIC x(64).
000184
000221 IA1620 01 print-record pic x(121).
000227 Working-storage section.
000228 01 Working-storage-for-EMBMAP pic x.
000229
000230 77 comp-code pic $9999 comp.
000231 77 ws-type pic x(3) value spaces.
000232
000233 01 i-f-status-area.
000234 05 i-f-file-status pic x(2).
000235 88 i-o-successful value zeroes.
000236 IMP
000237
000238 01 status-area.
000239 05 commuter-file-status pic x(2).
000240 88 i-o-okay value zeroes.
000241 IMP
000242 05 commuter-stl-status.
000243 10 stl-r15-return-code pic 9(2) comp.
000244 10 stl-function-code pic 9(1) comp.
000245 10 stl-feedback-code pic 9(3) comp.
000246
000247 77 update-file-status pic xx.
000248 77 loccode-file-status pic xx.
000249 77 updprint-file-status pic xx.
000877 procedure division.
000878 000-do-main-logic.
000879 display "PROGRAM EMBMAP - Beginning".
000880 perform 050-create-stl-master-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.
  - \*** The program-name is unresolved because the NOCOMPILE option is in effect.

## 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 43. 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) |



Table 43. Terms and symbols used in MAP output (continued)

| Term           | Description                                                         |
|----------------|---------------------------------------------------------------------|
| 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)                            |
| OBJECT-REF     | Object reference (USAGE OBJECT REFERENCE)                           |
| PACKED-DEC     | Internal decimal (USAGE PACKED-DECIMAL or COMPUTATIONAL-3)          |
| POINTER        | Pointer (USAGE POINTER)                                             |
| 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 Name from PROGRAM-ID paragraph | Program    |
| LineID | Level   |                                        | 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 that follow 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          |
| 264     | ABEND-ITEM1                   |                     |
| 265     | ABEND-ITEM2                   |                     |
| 347     | ADD-CODE . . . . .            | 1126 1192           |
| 381     | ADDRESS-ERROR. . . . .        | M1156               |
| 280     | AREA-CODE. . . . .            | 1266 1291 1354 1375 |
| 382     | CITY-ERROR . . . . .          | M1159               |

(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              |             |
| 943     | 050-CREATE-STL-MASTER-FILE . . | P879        |
| 995     | 100-INITIALIZE-PARAGRAPH . . . | P881        |
| 1471    | 1100-PRINT-I-F-HEADINGS. . . . | P926        |
| 1511    | 1200-PRINT-I-F-DATA. . . . .   | P928        |
| 1573    | 1210-GET-MILES-TIME. . . . .   | P1540       |
| 1666    | 1220-STORE-MILES-TIME. . . . . | P1541       |
| 1682    | 1230-PRINT-SUB-I-F-DATA. . . . | P1562       |
| 1706    | 1240-COMPUTE-SUMMARY . . . . . | P1563       |
| 1052    | 200-EDIT-UPDATE-TRANSACTION. . | P890        |
| 1154    | 210-EDIT-THE-REST. . . . .     | P1145       |
| 1189    | 300-UPDATE-COMMUTER-RECORD . . | P893        |
| 1237    | 310-FORMAT-COMMUTER-RECORD . . | P1194 P1209 |

```

1258 320-PRINT-COMMUTER-RECORD. . . P1195 P1206 P1212 P1222
1318 330-PRINT-REPORT P1208 P1232 P1286 P1310 P1370
1342 400-PRINT-TRANSACTION-ERRORS . P896

```

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

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)<br>Defined | (2)<br>Cross-reference of programs | (3)<br>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: 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-master-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 verbs in a program, and shows where each is referenced. The listing is produced by the VBREF compiler option.

| (1) | (2)                | (3)                                                                     |
|-----|--------------------|-------------------------------------------------------------------------|
| 2   | ACCEPT . . . . .   | 101 101                                                                 |
| 2   | ADD . . . . .      | 129 130                                                                 |
| 1   | CALL . . . . .     | 140                                                                     |
| 5   | CLOSE . . . . .    | 90 94 97 152 153                                                        |
| 20  | COMPUTE . . . . .  | 150 164 164 165 166 166 166 166 167 168 168 169 169 170 171 171         |
|     |                    | 171 172 172 173                                                         |
| 2   | CONTINUE . . . . . | 106 107                                                                 |
| 2   | DELETE . . . . .   | 96 119                                                                  |
| 47  | DISPLAY . . . . .  | 88 90 91 92 92 93 94 94 94 95 96 96 97 99 99 100 100 100 100            |
|     |                    | 103 109 117 117 118 119 138 139 139 139 139 139 139 139 139 140 140 140 |
|     |                    | 140 143 148 148 149 149 149 152 152 152 153 162                         |
| 2   | EVALUATE . . . . . | 116 155                                                                 |
| 47  | IF . . . . .       | 88 90 93 94 94 95 96 96 97 99 100 103 105 105 107 107 107 109           |
|     |                    | 110 111 111 112 113 113 113 113 114 114 115 115 116 118 119 124         |
|     |                    | 124 126 127 129 132 133 134 135 136 148 149 152 152                     |
| 183 | MOVE . . . . .     | 90 93 95 98 98 98 98 98 99 100 101 101 102 104 105 105 106 106          |
|     |                    | 107 107 108 108 108 108 108 108 109 110 111 112 113 113 113 114         |
|     |                    | 114 114 115 115 116 116 117 117 117 118 118 118 119 119 120 121         |
|     |                    | 121 121 121 121 121 121 121 121 121 122 122 122 122 122 123 123         |
|     |                    | 123 123 123 123 123 124 124 124 125 125 125 125 125 125 126             |
|     |                    | 126 126 126 126 127 127 127 127 128 128 129 129 130 130 130 130         |
|     |                    | 131 131 131 131 131 132 132 132 132 132 133 133 133 133 133             |
|     |                    | 134 134 134 134 134 135 135 135 135 135 135 136 136 137 137             |
|     |                    | 137 137 138 138 138 138 141 141 142 142 144 144 144 144 145 145         |
|     |                    | 145 145 146 149 150 150 150 151 151 155 156 156 157 157 158 158         |
|     |                    | 159 159 160 160 161 161 162 162 162 168 168 168 169 169 170 171         |
|     |                    | 171 172 172 173 173                                                     |
| 5   | OPEN . . . . .     | 93 95 99 144 148                                                        |
| 62  | PERFORM . . . . .  | 88 88 88 88 89 89 89 91 91 91 91 93 93 94 94 95 95 95 95 96             |
|     |                    | 96 96 97 97 97 100 100 100 101 102 104 109 109 111 116 116 117 117      |
|     |                    | 117 118 118 118 118 119 119 119 120 120 124 125 127 128 133 134         |
|     |                    | 135 136 136 137 150 151 151 153 153                                     |
| 8   | READ . . . . .     | 88 89 96 101 102 108 149 151                                            |
| 1   | REWRITE . . . . .  | 118                                                                     |
| 4   | SEARCH . . . . .   | 106 106 141 142                                                         |
| 46  | SET . . . . .      | 88 89 101 103 104 105 106 108 108 136 141 142 149 150 151 152 154       |
|     |                    | 155 156 156 156 156 157 157 157 157 158 158 158 158 159 159 159         |
|     |                    | 159 160 160 160 160 161 161 161 161 162 162 164 164                     |
| 2   | STOP . . . . .     | 92 143                                                                  |
| 4   | STRING . . . . .   | 123 126 132 134                                                         |
| 33  | WRITE . . . . .    | 94 116 129 129 129 129 129 130 130 130 130 145 146 146 146 146 147      |
|     |                    | 147 151 165 165 166 166 167 174 174 174 174 174 174 174 175 175         |

- (1) Number of times the verb is used in the program
- (2) Verb
- (3) Line numbers where the verb is used

## Debugging user exits

To debug a user-exit routine, use the debugger on the main compiler module, not on COB2.EXE. (The main compiler module is a separate process started by cob2, and the debugger can debug only one process.)

Do these steps:

1. Use cob2 with the -# option to see how cob2 calls the main compiler module and what options it passes. For example, the following command compiles *pgmname.cbl* with the IWZRMGUX user exit and links it:

```
cob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

Modify this command as follows:

```
cob2 -# -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

As a result, you will see this (igyccob2 calls your user exit):

```
igyccob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
ilink /free /nol /pm:vio pgmname.obj
```

2. Debug the user exit as follows:

```
idebug igyccob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

The debugger automatically stops at the beginning of the user exit, provided that you built the exit with debug information.

---

## Debugging assembler routines

Use the Disassembly view to debug assembler routines. Because assembler routines have no debug information, the debugger automatically goes to this view.

Set a breakpoint at a disassembled statement in the Disassembly view by double-clicking in the prefix area. By default, the debugger when it starts runs until it hits the first debuggable statement. To cause the debugger to stop at the very first instruction in the application (debuggable or not), you must use the `-i` option. For example:

```
idebug -i progrname
```

---

## Part 4. Accessing databases

### Chapter 19. Programming for a DB2

|                                            |     |
|--------------------------------------------|-----|
| <b>environment</b>                         | 321 |
| DB2 coprocessor                            | 321 |
| Coding SQL statements                      | 322 |
| Using SQL INCLUDE with the DB2 coprocessor | 322 |
| Using binary items in SQL statements       | 323 |
| Determining the success of SQL statements  | 323 |
| Starting DB2 before compiling              | 323 |
| Compiling with the SQL option              | 323 |
| Separating DB2 suboptions                  | 324 |
| Using package and bind file-names          | 324 |

### Chapter 20. Developing COBOL programs for

|                                         |     |
|-----------------------------------------|-----|
| <b>CICS</b>                             | 327 |
| Coding COBOL programs to run under CICS | 327 |
| Getting the system date under CICS      | 328 |
| Making dynamic calls under CICS         | 329 |
| Compiling and running CICS programs     | 330 |
| Integrated CICS translator              | 331 |
| Debugging CICS programs                 | 332 |

### Chapter 21. Open Database Connectivity

|                                                      |     |
|------------------------------------------------------|-----|
| <b>(ODBC)</b>                                        | 333 |
| Comparison of ODBC and embedded SQL                  | 333 |
| Background                                           | 334 |
| Installing and configuring software for ODBC         | 334 |
| Coding ODBC calls from COBOL: overview               | 334 |
| Using data types appropriate for ODBC                | 334 |
| Passing pointers as arguments in ODBC calls          | 335 |
| Example: passing pointers as arguments in ODBC calls | 336 |
| Accessing function return values in ODBC calls       | 337 |
| Testing bits in ODBC calls                           | 337 |
| Using COBOL copybooks for ODBC APIs                  | 338 |
| Example: sample program using ODBC copybooks         | 339 |
| Example: copybook for ODBC procedures                | 340 |
| Example: copybook for ODBC data definitions          | 343 |
| ODBC names truncated or abbreviated for COBOL        | 343 |
| Compiling and linking programs that make ODBC calls  | 345 |
| Understanding ODBC error messages                    | 345 |





---

## Chapter 19. Programming for a DB2 environment

In general, the coding for your 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.

If EXEC SQL statements are used in COBOL DLLs 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"

### RELATED TASKS

"Coding SQL statements" on page 322

"Starting DB2 before compiling" on page 323

"Compiling with the SQL option" on page 323

DB2 UDB Application Development Guide: Programming Client Applications

DB2 UDB Application Development Guide: Programming Server Applications

### RELATED REFERENCES

"DYNAM" on page 238

"SQL" on page 262

DB2 UDB SQL Reference Volume 1

DB2 UDB SQL Reference Volume 2

---

## DB2 coprocessor

When you use the DB2 coprocessor, the compiler handles your source program that contains embedded SQL statements without your having to use a separate precompiler.

When the compiler encounters SQL statements in the source program, it interfaces with the DB2 coprocessor. This 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 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.

- The source program can contain object-oriented COBOL language extensions.

You must specify the SQL compiler option when you compile programs that use the DB2 coprocessor.

#### RELATED TASKS

“Compiling with the SQL option” on page 323

---

## 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.

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 and programs that use the THREAD compiler option.
- Declare 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.

**Restriction:** You cannot use SQL statements in object-oriented classes or methods.

You can use SQL statements even for large objects (such as BLOB and CLOB) and compound SQL. The size of large objects is limited to 2 GB for a group or elementary data item.

#### RELATED TASKS

“Using SQL INCLUDE with the DB2 coprocessor”

“Using binary items in SQL statements” on page 323

“Determining the success of SQL statements” on page 323

DB2 UDB Application Development Guide: Programming Client Applications

## 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 extensions 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. However, if you use the standard DB2 copybooks, there are no other settings that you must make. If the search rules call for using SYSLIB as the library-name, the compiler finds the copybooks by using the DB2 environment variable DB2PATH, which is set during DB2 installation, to extend the setting of

SYSLIB to include the DB2 include directory. The SYSLIB string that is used is essentially %SYSLIB%;%DB2PATH%\INCLUDE\COBOL\_A.

**RELATED REFERENCES**

Chapter 15, “Compiler-directing statements,” on page 273  
COPY statement (*COBOL for Windows Language Reference*)

## Using binary items in SQL statements

For binary data items that you specify in an EXEC SQL statement, you can declare the data items as either USAGE COMP-5 or as USAGE BINARY, COMP, or COMP-4.

If you declare the binary data items as USAGE BINARY, COMP, or COMP-4, use the TRUNC(BIN) option and the BINARY(NATIVE) 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) or both are 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 43

**RELATED REFERENCES**

“TRUNC” on page 266

## 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. Your program should test the return code and take any necessary action.

**RELATED TASKS**

DB2 UDB Application Development Guide: Programming Client Applications

---

## 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.cb1. . .
```

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”

“Using package and bind file-names”

#### RELATED REFERENCES

“SQL” on page 262

Precompile (*DB2 UDB Command Reference*)

## 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 extension .bnd is added to the base name. Unless explicitly specified, the file-name is relative to the current directory.



---

## Chapter 20. Developing COBOL programs for CICS

You can write CICS programs in COBOL and run them under CICS.

After you have installed and configured CICS, you need to prepare a COBOL program to run under CICS. The steps are outlined below:

1. Use an editor to:
  - Code the program using COBOL statements and CICS commands.
  - Create COBOL copybooks.
  - Create the CICS screen maps that the program uses.
2. Use the `cicsmap` command to process the screen maps.
3. Use one of the forms of the `cicstcl` command to translate the CICS commands and to compile and link the program:
  - Use `cicstcl -p` to translate, compile, and link using the integrated CICS translator.
  - Use `cicstcl` without the `-p` flag to translate, compile, and link using the separate CICS translator.
4. Define the resources for the application, such as transactions, application programs, and files, by using the CICS Administration Utility.
5. Start your CICS region by using the CICS Administration Utility.
6. At your CICS terminal, run the application by entering the four-character transaction ID that is associated with the application.

If you want to execute code using CICS ECI (External Call Interface), you need to have CICS Client installed. Otherwise, you will encounter an error due to a missing DLL. You also need to start CICS Client before running the application.

### RELATED CONCEPTS

“Integrated CICS translator” on page 331

### RELATED TASKS

“Compiling and running CICS programs” on page 330

TXSeries for Multiplatforms: CICS Application Programming Guide

TXSeries for Multiplatforms: CICS Administration Guide for Windows Systems

---

## Coding COBOL programs to run under CICS

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.

### Restrictions:

- Object-oriented programming and interoperability with Java are not supported under CICS. COBOL class definitions and methods cannot be run in a CICS environment.
- The source program must not contain any nested programs.

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 these 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 zSeries host data format support for COBOL programs that are translated by the separate or integrated CICS translator and run on TXSeries.

#### RELATED TASKS

“Getting the system date under CICS”

“Making dynamic calls under CICS” on page 329

“Compiling and running CICS programs” on page 330

“Debugging CICS programs” on page 332

## 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 the CICS environment 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 the CICS environment 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 34

#### RELATED REFERENCES

CURRENT-DATE (*COBOL for Windows 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.

Consider the following example, where 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.
```

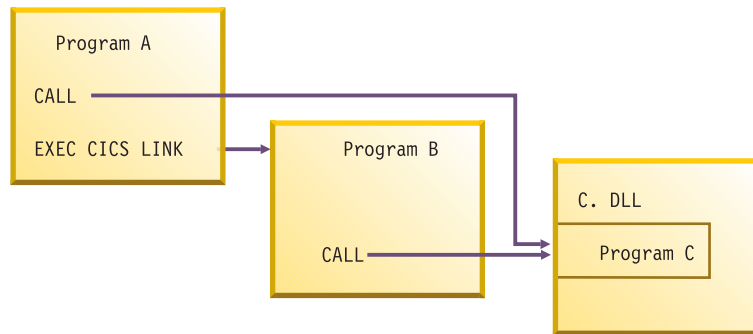
You must pass the CICS control blocks DFHEIBLK and DFHCOMMAREA (as shown above) to alpha. The source for alpha is in file alpha.ccp. Use the command `cicstcl` to translate, compile, and link alpha.ccp.

With TXSeries, the `cicstcl` command creates a DLL called alpha.ibm cob. You must include the directory where alpha.ibm cob resides in the COBPATH environment variable. You must also do the following steps:

- Use the `-LIBMCOB` flag with `cicstcl`.
- Set the COBPATH environment variable in the system environment variables, because the CICS region does not recognize user environment variable settings. Alternatively, you can set COBPATH in the file `\var\cics_regions\xxx\` environment, in which case COBPATH is effective only for the `xxx` region.

**DLL considerations:** If you have a DLL that contains one or more COBOL programs, do not use it in more than one run unit within the same CICS transaction, or the results will be 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.DLL).
- 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.DLL).



Programs A and B share the same copy of Program C, and any changes to its state affect both.

In the CICS environment, programs in a DLL are initialized (both the WSCLEAR compiler option and 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 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 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 by the VALUE clause only, not by the WSCLEAR compiler option.
- 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.

---

## Compiling and running CICS programs

COBOL for Windows TXSeries programs must use the thread-safe version of the COBOL runtime library. Compile such programs using the THREAD compiler option.

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, or COMP-4 data items conform to PICTURE specifications, you might 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 BINARY(NATIVE) and TRUNC(BIN) are in effect.

You must use the PGMNAME(MIXED) compiler option for programs that use CICS Client.

Do not use the DYNAM, NOLIB, or NOTHREAD compiler option when you translate COBOL programs with the separate or integrated CICS translator. All other COBOL compiler options are supported.

**Runtime options:** Use the FILESYS runtime option to specify the file system to use for files when no specific file system has been selected by means of an ASSIGN clause.

#### RELATED CONCEPTS

"Integrated CICS translator"

#### RELATED TASKS

"Compiling from the command line" on page 201

TXSeries for Multiplatforms: CICS Application Programming Guide

#### RELATED REFERENCES

Chapter 14, "Compiler options," on page 225

"FILESYS" on page 294

Appendix B, "zSeries host data format considerations," on page 567

## 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. 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, specify the `-p` flag to use the integrated CICS translator. The `cicstcl -p` command invokes the compiler with the appropriate suboptions of the CICS compiler option.

Although you can still translate embedded CICS statements separately, using the integrated CICS translator is recommended. 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 can use the Debug Perspective of Rational Developer for System z to debug the original source instead of the expanded source that the separate CICS translator provides.
- You do not need to separately translate the EXEC CICS statements that are in copybooks.
- There is no intermediate file for the 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 batch.

**Attention:** To use the integrated CICS translator, TXSeries V6.1 (or later) is required.

#### RELATED TASKS

"Coding COBOL programs to run under CICS" on page 327

#### RELATED REFERENCES

"CICS" on page 232

---

## Debugging CICS programs

| If you compile CICS programs using the integrated translator, you can debug the  
| programs at the original source level instead of debugging the expanded source  
| that the separate CICS translator provides.

| If you use the separate CICS translator, first translate your CICS programs into  
| COBOL. Then debug the resulting COBOL programs the same way you would  
| debug any other COBOL programs.

### RELATED CONCEPTS

Chapter 18, “Debugging,” on page 297

### RELATED TASKS

“Compiling from the command line” on page 201

TXSeries for Multiplatforms: CICS Application Programming Guide

---

## Chapter 21. Open Database Connectivity (ODBC)

Open Database Connectivity (ODBC) is a specification for an application programming interface (API) that enables applications to access multiple database management systems using Structured Query Language (SQL). Using the ODBC interface in your COBOL applications, you can dynamically access databases and file systems that support the ODBC interface.

ODBC permits maximum interoperability: a single application can access many different database management systems. This interoperability enables you to develop, compile, and ship an application without targeting a specific type of data source. Users can then add the database drivers, which link the application to the database management systems of their choice.

When you use the ODBC interface, your application makes calls through a driver manager. The driver manager dynamically loads the necessary driver for the database server to which the application connects. The driver, in turn, accepts the call, sends the SQL to the specified data source (database), and returns any result.

---

### Comparison of ODBC and embedded SQL

Your COBOL applications that use embedded SQL for database access must be processed by a precompiler or coprocessor for a particular database and need to be recompiled if the target database changes. Because ODBC is a call interface, there is no compile-time designation of the target database as there is with embedded SQL. Not only can you avoid having multiple versions of your application for multiple databases, but your application can dynamically determine which database to target.

Some of the advantages of ODBC are:

- ODBC provides a consistent interface regardless of the kind of database server used.
- You can have more than one concurrent connection.
- Applications do not have to be bound to each database on which they will run. Although COBOL for Windows does this bind for you automatically, it binds automatically to only one database. If you want to choose which database to connect to dynamically at run time, you must take extra steps to bind to a different database.

Embedded SQL also has some advantages:

- Static SQL usually provides better performance than dynamic SQL. It does not have to be prepared at run time, thus reducing both processing and network traffic.
- With static SQL, database administrators have to grant users access to a package only rather than access to each table or view that will be used.

---

## Background

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface, referred to as the X/Open Call Level Interface. The goal of this interface is to increase portability of applications by enabling them to become independent of any one database vendor's programming interface.

ODBC was originally developed by Microsoft for Microsoft operating systems based on a preliminary draft of X/Open CLI. Since then, other vendors have provided ODBC drivers that run on other platforms, such as UNIX<sup>®</sup> systems.

---

## Installing and configuring software for ODBC

To enable ODBC for data access in COBOL for Windows, you must install an ODBC driver manager and drivers, add the ODBC database drivers necessary for your installation, and install the RDBMS client (for example, DB2 UDB, Oracle 7 SQL\*NET).

After installing the drivers, you need to configure your data sources by using the ODBC Administrator program. A data source consists of a DBMS and any remote operating system and network necessary to access it. Because Windows can host multiple users, users must configure their own data sources. For detailed configuration information for the specific driver that you need to configure, refer to the appropriate section of the online help for that driver.

---

## Coding ODBC calls from COBOL: overview

To access ODBC from COBOL programs, you need to use data types that are appropriate for ODBC, and understand how to pass arguments, access return values for functions, and test bits. IBM COBOL for Windows provides copybooks that help you make ODBC calls.

### RELATED TASKS

"Using data types appropriate for ODBC"

"Passing pointers as arguments in ODBC calls" on page 335

"Accessing function return values in ODBC calls" on page 337

"Testing bits in ODBC calls" on page 337

"Using COBOL copybooks for ODBC APIs" on page 338

"Compiling and linking programs that make ODBC calls" on page 345

## Using data types appropriate for ODBC

The data types specified in ODBC APIs are defined in terms of ODBC C types in the API definitions. Use the COBOL data declarations that correspond to the indicated ODBC C types of the arguments.

*Table 44. ODBC C types and corresponding COBOL declarations*

| ODBC C type  | COBOL form       | Description                            |
|--------------|------------------|----------------------------------------|
| SQLSMALLINT  | COMP-5 PIC S9(4) | Signed short integer (2-byte binary)   |
| SQLUSMALLINT | COMP-5 PIC 9(4)  | Unsigned short integer (2-byte binary) |
| SQLINTEGER   | COMP-5 PIC S9(9) | Signed long integer (4-byte binary)    |
| SQLUINTEGER  | COMP-5 PIC 9(9)  | Unsigned long integer (4-byte binary)  |
| SQLREAL      | COMP-1           | Floating point (4 bytes)               |
| SQLFLOAT     | COMP-2           | Floating point (8 bytes)               |

Table 44. ODBC C types and corresponding COBOL declarations (continued)

| ODBC C type | COBOL form | Description                   |
|-------------|------------|-------------------------------|
| SQLDOUBLE   | COMP-2     | Floating point (8 bytes)      |
| SQLCHAR     | POINTER    | Pointer to unsigned character |
| SQLHDBC     | POINTER    | Connection handle             |
| SQLHENV     | POINTER    | Environment handle            |
| SQLHSTMT    | POINTER    | Statement handle              |
| SQLHWND     | POINTER    | Window handle                 |

The pointer to an unsigned character in COBOL is a pointer to a null-terminated string. You define the target of the pointer item as PIC X(*n*), where *n* is large enough to represent the null-terminated field. Some ODBC APIs require that you pass null-terminated character strings as arguments.

Do not use zSeries host data formats. ODBC APIs expect parameters to be in native format.

#### RELATED TASKS

“Passing pointers as arguments in ODBC calls”

“Accessing function return values in ODBC calls” on page 337

## Passing pointers as arguments in ODBC calls

If you specify a pointer argument to one of the data types acceptable to ODBC, then you must either pass the target of the pointer BY REFERENCE, define a pointer item that points to the target item and pass that BY VALUE, or pass the ADDRESS OF the target BY VALUE.

To illustrate, assume the function is defined as:

```
RETCODE SQLSomeFunction(PSomeArgument)
```

Here PSomeArgument is defined as an argument that points to SomeArgument. You can pass the argument to SQLSomeFunction in one of the following ways:

- Pass SomeArgument BY REFERENCE:  
CALL "SQLSomeFunction" USING BY REFERENCE SomeArgument  
You can use USING BY CONTENT SomeArgument instead if SomeArgument is an input argument.
- Define a pointer data item PSomeArgument to point to SomeArgument:  
SET PSomeArgument TO ADDRESS OF SomeArgument  
CALL "SQLSomeFunction" USING BY VALUE PSomeArgument
- Pass ADDRESS OF SomeArgument BY VALUE:  
CALL "SQLSomeFunction" USING BY VALUE ADDRESS OF SomeArgument

You can use the last approach only if the target argument, SomeArgument, is a level-01 item in the LINKAGE SECTION. If so, you can set the addressability to SomeArgument in one of the following ways:

- Explicitly by using either a pointer or an identifier, as in the following examples:  
SET ADDRESS OF SomeArgument TO a-pointer-data-item  
SET ADDRESS OF SomeArgument TO ADDRESS OF an-identifier
- Implicitly by having SomeArgument passed in as an argument to the program from which the ODBC function call is being made.

“Example: passing pointers as arguments in ODBC calls”

### Example: passing pointers as arguments in ODBC calls

The following example program fragment shows how to invoke the SQLAllocHandle function.

```
. . .
WORKING-STORAGE SECTION.
 COPY ODBC3.

 . . .
01 SQL-RC COMP-5 PIC S9(4).
01 Henv POINTER.
 . . .
PROCEDURE DIVISION.
 . . .
 CALL "SQLAllocHandle"
 USING
 By VALUE sql=handle-env
 sql=null-handle
 By REFERENCE Henv
 RETURNING SQL-RC
 IF SQL-RC NOT = (SQL-SUCCESS or SQL-SUCCESS-WITH-INFO)
 THEN
 DISPLAY "SQLAllocHandle failed."
 . . .
 ELSE
 . . .
 . . .
```

You can use any one of the following examples for calling the SQLConnect function.

#### Example 1:

```
. . .
CALL "SQLConnect" USING BY VALUE ConnectionHandle
 BY REFERENCE ServerName
 BY VALUE SQL-NTS
 BY REFERENCE UserIdentifier
 BY VALUE SQL-NTS
 BY REFERENCE AuthenticationString
 BY VALUE SQL-NTS
 RETURNING SQL-RC
 . . .
```

#### Example 2:

```
. . .
SET Ptr-to-ServerName TO ADDRESS OF ServerName
SET Ptr-to-UserIdentifier TO ADDRESS OF UserIdentifier
SET Ptr-to-AuthenticationString TO ADDRESS OF AuthenticationString
CALL "SQLConnect" USING BY VALUE ConnectionHandle
 Ptr-to-ServerName
 SQL-NTS
 Ptr-to-UserIdentifier
 SQL-NTS
 Ptr-to-AuthenticationString
 SQL-NTS
 RETURNING SQL-RC
 . . .
```

#### Example 3:

```
. . .
CALL "SQLConnect" USING BY VALUE ConnectionHandle
 ADDRESS OF ServerName
 SQL-NTS
 ADDRESS OF UserIdentifier
```



|           |                                 |
|-----------|---------------------------------|
|           | SQL-NTS                         |
|           | ADDRESS OF AuthenticationString |
|           | SQL-NTS                         |
|           | SQL-RC                          |
| RETURNING |                                 |
| . . .     |                                 |

In Example 3, you must define Servername, UserIdentifier, and AuthenticationString as level-01 items in the LINKAGE SECTION.

The BY REFERENCE or BY VALUE phrase applies to all arguments until another BY REFERENCE, BY VALUE, or BY CONTENT phrase overrides it.

## Accessing function return values in ODBC calls

Specify the function return values for an ODBC call by using the RETURNING phrase of the CALL statement.

```
CALL "SQLAllocEnv" USING BY VALUE Phenv RETURNING SQL-RC
 IF SQL-RC NOT = SQL-SUCCESS
 THEN
 DISPLAY "SQLAllocEnv failed."
 . . .
 ELSE
 . . .
 END-IF
```

## Testing bits in ODBC calls

For some of the ODBC APIs, you must set bit masks and must query bits. You can use the library routine iwzODBCTestBits to query bits. You must link the IWZODBC.LIB import library with any application that calls iwzODBCTestBits.

Call this routine as follows:

```
CALL "iwzODBCTestBits" USING identifier-1, identifier-2 RETURNING identifier-3
```

### *identifier-1*

The field being tested. It must be a 2- or 4-byte binary number field, that is, USAGE COMP-5 PIC 9(4) or PIC 9(9).

### *identifier-2*

The bit mask field to select the bits to be tested. It must be defined with the same USAGE and PICTURE as *identifier-1*.

### *identifier-3*

The return value for the test (define it as USAGE COMP-5 PICS9(4)):

- 0** No bits selected by *identifier-2* are ON in *identifier-1*.
- 1** All bits selected by *identifier-2* are ON in *identifier-1*.
- 1** One or more bits are ON and one or more bits are OFF among the bits selected by *identifier-2* for *identifier-1*.
- 100** Invalid input-argument found (such as an 8-byte binary number field used as *identifier-1*).

You can set multiple bits in a field with COBOL arithmetic expressions that use the bit masks defined in the ODBCCOB copybook. For example, you can set bits for SQL-CVT-CHAR, SQL-CVT-NUMERIC, and SQL-CVT-DECIMAL in the InfoValue field with this statement:

```
COMPUTE InfoValue = SQL-CVT-CHAR + SQL-CVT-NUMERIC + SQL-CVT-DECIMAL
```

After you set InfoValue, you can pass it to the iwzTestBits function as the second argument.

The operands in the arithmetic expression above represent disjoint bits as defined in the ODBCCOB copybook. As a result, addition sets the intended bits on. Be careful to not repeat bits in such an arithmetic expression, however, because the operation is not logical OR. For example, the following code results in InfoValue not having the intended SQL-CVT-CHAR bit on:

```
COMPUTE InfoValue = SQL-CVT-CHAR + SQL-CVT-NUMERIC + SQL-CVT-DECIMAL + SQL-CVT-CHAR
```

The call interface convention in effect at the time of the call must be CALLINT SYSTEM DESCRIPTOR.

---

## Using COBOL copybooks for ODBC APIs

IBM COBOL for Windows provides copybooks that make it easier for you to access databases with ODBC drivers by using ODBC calls from COBOL programs. You can use these copybooks with or without modification.

The copybooks described below are for ODBC Version 3.0. However, Version 2.x copybooks are also supplied, and you can substitute them for the Version 3.0 copybooks if you need to develop applications for ODBC Version 2.x.

*Table 45. ODBC copybooks*

| Copybook for Version 3.0 of ODBC | Copybook for Version 2.x of ODBC | Description                   | Location                                    |
|----------------------------------|----------------------------------|-------------------------------|---------------------------------------------|
| ODBC3.CPY                        | ODBC2.CPY                        | Symbols and constants         | INCLUDE folder for COBOL                    |
| ODBC3D.CPY                       | ODBC2D.CPY                       | DATA DIVISION definitions     | ODBC folder in the SAMPLES folder for COBOL |
| ODBC3P.CPY                       | ODBC2P.CPY                       | PROCEDURE DIVISION statements | ODBC folder in the SAMPLES folder for COBOL |

Include the paths for the INCLUDE and ODBC folders in the SYSLIB environment variable to ensure that the copybooks are available to the compiler.

ODBC3.CPY defines the symbols for constant values described for ODBC APIs. It maps constants used in calls to ODBC APIs to symbols specified in ODBC guides. You can use this copybook to specify and test arguments and function return values.

ODBC3P.CPY lets you use prepared COBOL statements for commonly used functions for initializing ODBC, handling errors, and cleaning up (SQLAllocEnv, SQLAllocConnect, iwzODBCLicInfo, SQLAllocStmt, SQLFreeStmt, SQLDisconnect, SQLFreeConnect, and SQLFreeEnv).

ODBC3D.CPY contains data declarations used by ODBC3.CPY in the WORKING-STORAGE SECTION (or LOCAL-STORAGE SECTION).

Some COBOL-specific adaptations were made in these copybooks:

- Underscores (\_) were replaced with hyphens (-). For example, SQL\_SUCCESS is specified as SQL-SUCCESS.
- Names longer than 30 characters.

To include the copybook ODBC3.CPY, specify a COPY statement in the DATA DIVISION as follows:

- For a program, specify the COPY statement in the WORKING-STORAGE SECTION (in the outermost program if programs are nested).
- For each method that makes ODBC calls, specify the COPY statement in the WORKING-STORAGE SECTION of the method (not the WORKING-STORAGE SECTION of the class definition).

“Example: sample program using ODBC copybooks”

“Example: copybook for ODBC data definitions” on page 343

“Example: copybook for ODBC procedures” on page 340

#### RELATED REFERENCES

“ODBC names truncated or abbreviated for COBOL” on page 343

## Example: sample program using ODBC copybooks

The following example illustrates the use of three ODBC copybooks.

```

cb1 pgmname(mixed)

* ODBC3EG.CBL *

* Sample program using ODBC3, ODBC3D and ODBC3P copybooks *

IDENTIFICATION DIVISION.
PROGRAM-ID. "ODBC3EG".
DATA DIVISION.

WORKING-STORAGE SECTION.
* copy ODBC API constant definitions
COPY "odbc3.cpy" SUPPRESS.
* copy additional definitions used by ODBC3P procedures
COPY "odbc3d.cpy".
* arguments used for SQLConnect
01 ServerName PIC X(10) VALUE Z"Oracle7".
01 ServerNameLength COMP-5 PIC S9(4) VALUE 10.
01 UserId PIC X(10) VALUE Z"TEST123".
01 UserIdLength COMP-5 PIC S9(4) VALUE 10.
01 Authentication PIC X(10) VALUE Z"TEST123".
01 AuthenticationLength COMP-5 PIC S9(4) VALUE 10.
PROCEDURE DIVISION.
Do-ODBC SECTION.
Start-ODBC.
DISPLAY "Sample ODBC 3.0 program starts"
* allocate henv & hdbc
PERFORM ODBC-Initialization
* connect to data source
CALL "SQLConnect" USING BY VALUE Hdbc
BY REFERENCE ServerName
BY VALUE ServerNameLength
BY REFERENCE UserId
BY VALUE UserIdLength
BY REFERENCE Authentication
BY VALUE AuthenticationLength
RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
MOVE "SQLConnect" to SQL-stmt
MOVE SQL-HANDLE-DBC to DiagHandleType

```

```

 SET DiagHandle to Hdbc
 PERFORM SQLDiag-Function
 END-IF
* allocate hstmt
 PERFORM Allocate-Statement-Handle

* add application specific logic here *

* clean-up environment
 PERFORM ODBC-Clean-Up.
* End of sample program execution
 DISPLAY "Sample COBOL ODBC program ended"
 GOBACK.
* copy predefined COBOL ODBC calls which are performed
 COPY "odbc3p.cpy".

* End of ODBC3EG.CBL: Sample program for ODBC 3.0 *

```

## Example: copybook for ODBC procedures

This example shows procedures for initializing ODBC, cleaning up, and handling errors.

```

* ODBC3P.CPY *

* Sample ODBC initialization, clean-up and error handling *
* procedures (ODBC Ver 3.0) *

*** Initialization functions SECTION *****
 ODBC-Initialization SECTION.
*
 Allocate-Environment-Handle.
 CALL "SQLAllocHandle" USING
 BY VALUE SQL-HANDLE-ENV
 BY VALUE SQL-NULL-HANDLE
 BY REFERENCE Henv
 RETURNING SQL-RC

 IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLAllocHandle for Env" TO SQL-stmt
 MOVE SQL-HANDLE-ENV to DiagHandleType
 SET DiagHandle to Henv
 PERFORM SQLDiag-Function
 END-IF.
*
 Set-Env-Attr-to-Ver30-Behavior.
 CALL "SQLSetEnvAttr" USING
 BY VALUE Henv
 BY VALUE SQL-ATTR-ODBC-VERSION
 BY VALUE SQL-OV-ODBC3
 or SQL-OV-ODBC2 *
 for Ver 2.x behavior *
 BY VALUE SQL-IS-INTEGER
 RETURNING SQL-RC

 IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLSetEnvAttr" TO SQL-stmt
 MOVE SQL-HANDLE-ENV to DiagHandleType
 SET DiagHandle to Henv
 PERFORM SQLDiag-Function
 END-IF.
*
 Allocate-Connection-Handle.
 CALL "SQLAllocHandle" USING
 By VALUE SQL-HANDLE-DBC
 BY VALUE Henv
 BY REFERENCE Hdbc

```

```

 RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLAllocHandle for Connection" to SQL-stmt
 MOVE SQL-HANDLE-ENV to DiagHandleType
 SET DiagHandle to Henv
 PERFORM SQLDiag-Function
END-IF.
*** SQLAllocHandle for statement function SECTION *****
Allocate-Statement-Handle SECTION.
Allocate-Stmt-Handle.
 CALL "SQLAllocHandle" USING
 BY VALUE SQL-HANDLE-STMT
 BY VALUE Hdbc
 BY REFERENCE Hstmt
 RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLAllocHandle for Stmt" TO SQL-stmt
 MOVE SQL-HANDLE-DBC to DiagHandleType
 SET DiagHandle to Hdbc
 PERFORM SQLDiag-Function
END-IF.
*** Cleanup Functions SECTION *****
ODBC-Clean-Up SECTION.
*
Free-Statement-Handle.
 CALL "SQLFreeHandle" USING
 BY VALUE SQL-HANDLE-STMT
 BY VALUE Hstmt
 RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLFreeHandle for Stmt" TO SQL-stmt
 MOVE SQL-HANDLE-STMT to DiagHandleType
 SET DiagHandle to Hstmt
 PERFORM SQLDiag-Function
END-IF.
*
SQLDisconnect-Function.
 CALL "SQLDisconnect" USING
 BY VALUE Hdbc
 RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLDisconnect" TO SQL-stmt
 MOVE SQL-HANDLE-DBC to DiagHandleType
 SET DiagHandle to Hdbc
 PERFORM SQLDiag-Function
END-IF.
*
Free-Connection-Handle.
 CALL "SQLFreeHandle" USING
 BY VALUE SQL-HANDLE-DBC
 BY VALUE Hdbc
 RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLFreeHandle for DBC" TO SQL-stmt
 MOVE SQL-HANDLE-DBC to DiagHandleType
 SET DiagHandle to Hdbc
 PERFORM SQLDiag-Function
END-IF.
*
Free-Environment-Handle.
 CALL "SQLFreeHandle" USING
 BY VALUE SQL-HANDLE-ENV
 BY VALUE Henv
 RETURNING SQL-RC
IF SQL-RC NOT = SQL-SUCCESS
 MOVE "SQLFreeHandle for Env" TO SQL-stmt
 MOVE SQL-HANDLE-ENV to DiagHandleType
 SET DiagHandle to Henv

```

```

 PERFORM SQLDiag-Function
 END-IF.
*** SQLDiag function SECTION *****
SQLDiag-Function SECTION.
SQLDiag.
 MOVE SQL-RC TO SAVED-SQL-RC
 DISPLAY "Return Value = " SQL-RC
 IF SQL-RC = SQL-SUCCESS-WITH-INFO
 THEN
 DISPLAY SQL-stmt " successful with information"
 ELSE
 DISPLAY SQL-stmt " failed"
 END-IF
* - get number of diagnostic records - *
 CALL "SQLGetDiagField"
 USING
 BY VALUE DiagHandleType
 DiagHandle
 0
 SQL-DIAG-NUMBER
 BY REFERENCE DiagRecNumber
 BY VALUE SQL-IS-SMALLINT
 BY REFERENCE OMITTED
 RETURNING SQL-RC
 IF SQL-RC = SQL-SUCCESS or SQL-SUCCESS-WITH-INFO
 THEN
* - get each diagnostic record - *
 PERFORM WITH TEST AFTER
 VARYING DiagRecNumber-Index FROM 1 BY 1
 UNTIL DiagRecNumber-Index > DiagRecNumber
 or SQL-RC NOT =
 (SQL-SUCCESS or SQL-SUCCESS-WITH-INFO)
* - get a diagnostic record - *
 CALL "SQLGetDiagRec"
 USING
 BY VALUE DiagHandleType
 DiagHandle
 DiagRecNumber-Index
 BY REFERENCE DiagSQLState
 DiagNativeError
 DiagMessageText
 BY VALUE DiagMessageBufferLength
 BY REFERENCE DiagMessageTextLength
 RETURNING SQL-RC
 IF SQL-RC = SQL-SUCCESS OR SQL-SUCCESS-WITH-INFO
 THEN
 DISPLAY "Information from diagnostic record number"
 " " DiagRecNumber-Index " for "
 SQL-stmt ":"
 DISPLAY " SQL-State = " DiagSQLState-Chars
 DISPLAY " Native error code = " DiagNativeError
 DISPLAY " Diagnostic message = "
 DiagMessageText(1:DiagMessageTextLength)
 ELSE
 DISPLAY "SQLGetDiagRec request for " SQL-stmt
 " failed with return code of: " SQL-RC
 " from SQLError"
 PERFORM Termination
 END-IF
 END-PERFORM
 ELSE
* - indicate SQLGetDiagField failed - *
 DISPLAY "SQLGetDiagField failed with return code of: "
 SQL-RC
 END-IF
 MOVE Saved-SQL-RC to SQL-RC
 IF Saved-SQL-RC NOT = SQL-SUCCESS-WITH-INFO

```

```

 PERFORM Termination
 END-IF.
*** Termination Section*****
Termination Section.
Termination-Function.
 DISPLAY "Application being terminated with rollback"
 CALL "SQLTransact" USING BY VALUE henv
 hdbc
 SQL-ROLLBACK
 SQL-RC
 RETURNING
 IF SQL-RC = SQL-SUCCESS
 THEN
 DISPLAY "Rollback successful"
 ELSE
 DISPLAY "Rollback failed with return code of: "
 SQL-RC
 END-IF
 STOP RUN.

* End of ODBC3P.CPY *

```

## Example: copybook for ODBC data definitions

The following example shows ODBC data definitions for items such as message text and return codes.

```

* ODBC3D.CPY (ODBC Ver 3.0) *

* Data definitions to be used with sample ODBC function calls *
* and included in Working-Storage or Local-Storage Section *

* ODBC Handles
01 Henv POINTER VALUE NULL.
01 Hdbc POINTER VALUE NULL.
01 Hstmt POINTER VALUE NULL.
* Arguments used for GetDiagRec calls
01 DiagHandleType COMP-5 PIC 9(4).
01 DiagHandle POINTER.
01 DiagRecNumber COMP-5 PIC 9(4).
01 DiagRecNumber-Index COMP-5 PIC 9(4).
01 DiagSQLState.
 02 DiagSQLState-Chars PIC X(5).
 02 DiagSQLState-Null PIC X.
01 DiagNativeError COMP-5 PIC S9(9).
01 DiagMessageText PIC X(511) VALUE SPACES.
01 DiagMessageBufferLength COMP-5 PIC S9(4) VALUE 511.
01 DiagMessageTextLength COMP-5 PIC S9(4).
* Misc declarations used in sample function calls
01 SQL-RC COMP-5 PIC S9(4) VALUE 0.
01 Saved-SQL-RC COMP-5 PIC S9(4) VALUE 0.
01 SQL-stmt PIC X(30).

* End of ODBC3D.CPY *

```

## ODBC names truncated or abbreviated for COBOL

The following table shows ODBC names that are longer than 30 characters and shows their corresponding COBOL names.

Table 46. ODBC names truncated or abbreviated for COBOL

| ODBC C #define symbol > 30 characters | Corresponding COBOL name    |
|---------------------------------------|-----------------------------|
| SQL_AD_ADD_CONSTRAINT_DEFERRABLE      | SQL-AD-ADD-CONSTRAINT-DEFER |

Table 46. ODBC names truncated or abbreviated for COBOL (continued)

| ODBC C #define symbol > 30 characters     | Corresponding COBOL name       |
|-------------------------------------------|--------------------------------|
| SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED  | SQL-AD-ADD-CONSTRAINT-INIT-DEF |
| SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE | SQL-AD-ADD-CONSTRAINT-INIT-IMM |
| SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE      | SQL-AD-ADD-CONSTRAINT-NON-DEFE |
| SQL_AD_CONSTRAINT_NAME_DEFINITION         | SQL-AD-CONSTRAINT-NAME-DEFINIT |
| SQL_AT_CONSTRAINT_INITIALLY_DEFERRED      | SQL-AT-CONSTRAINT-INITIALLY-DE |
| SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE     | SQL-AT-CONSTRAINT-INITIALLY-IM |
| SQL_AT_CONSTRAINT_NAME_DEFINITION         | SQL-AT-CONSTRAINT-NAME-DEFINIT |
| SQL_AT_CONSTRAINT_NON_DEFERRABLE          | SQL-AT-CONSTRAINT-NON-DEFERRAB |
| SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE      | SQL-AT-DROP-TABLE-CONSTRAINT-C |
| SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT     | SQL-AT-DROP-TABLE-CONSTRAINT-R |
| SQL_C_INTERVAL_MINUTE_TO_SECOND           | SQL-C-INTERVAL-MINUTE-TO-SECON |
| SQL_CA_CONSTRAINT_INITIALLY_DEFERRED      | SQL-CA-CONSTRAINT-INIT-DEFER   |
| SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE     | SQL-CA-CONSTRAINT-INIT-IMMED   |
| SQL_CA_CONSTRAINT_NON_DEFERRABLE          | SQL-CA-CONSTRAINT-NON-DEFERRAB |
| SQL_CA1_BULK_DELETE_BY_BOOKMARK           | SQL-CA1-BULK-DELETE-BY-BOOKMAR |
| SQL_CA1_BULK_UPDATE_BY_BOOKMARK           | SQL-CA1-BULK-UPDATE-BY-BOOKMAR |
| SQL_CDO_CONSTRAINT_NAME_DEFINITION        | SQL-CDO-CONSTRAINT-NAME-DEFINI |
| SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED     | SQL-CDO-CONSTRAINT-INITIALLY-D |
| SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE    | SQL-CDO-CONSTRAINT-INITIALLY-I |
| SQL_CDO_CONSTRAINT_NON_DEFERRABLE         | SQL-CDO-CONSTRAINT-NON-DEFERRA |
| SQL_CONVERT_INTERVAL_YEAR_MONTH           | SQL-CONVERT-INTERVAL-YEAR-MONT |
| SQL_CT_CONSTRAINT_INITIALLY_DEFERRED      | SQL-CT-CONSTRAINT-INITIALLY-DE |
| SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE     | SQL-CT-CONSTRAINT-INITIALLY-IM |
| SQL_CT_CONSTRAINT_NON_DEFERRABLE          | SQL-CT-CONSTRAINT-NON-DEFERRAB |
| SQL_CT_CONSTRAINT_NAME_DEFINITION         | SQL-CT-CONSTRAINT-NAME-DEFINIT |
| SQL_DESC_DATETIME_INTERVAL_CODE           | SQL-DESC-DATETIME-INTERVAL-COD |
| SQL_DESC_DATETIME_INTERVAL_PRECISION      | SQL-DESC-DATETIME-INTERVAL-PRE |
| SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR         | SQL-DL-SQL92-INTERVAL-DAY-TO-H |
| SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE       | SQL-DL-SQL92-INTERVAL-DAY-TO-M |
| SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND       | SQL-DL-SQL92-INTERVAL-DAY-TO-S |
| SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE      | SQL-DL-SQL92-INTERVAL-HR-TO-M  |
| SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND      | SQL-DL-SQL92-INTERVAL-HR-TO-S  |
| SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND    | SQL-DL-SQL92-INTERVAL-MIN-TO-S |
| SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH       | SQL-DL-SQL92-INTERVAL-YR-TO-MO |
| SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1       | SQL-FORWARD-ONLY-CURSOR-ATTR1  |
| SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2       | SQL-FORWARD-ONLY-CURSOR-ATTR2  |
| SQL_GB_GROUP_BY_CONTAINS_SELECT           | SQL-GB-GROUP-BY-CONTAINS-SELEC |
| SQL_ISV_CONSTRAINT_COLUMN_USAGE           | SQL-ISV-CONSTRAINT-COLUMN-USAG |
| SQL_ISV_REFERENTIAL_CONSTRAINTS           | SQL-ISV-REFERENTIAL-CONSTRAINT |
| SQL_MAXIMUM_CATALOG_NAME_LENGTH           | SQL-MAXIMUM-CATALOG-NAME-LENGT |



Table 46. ODBC names truncated or abbreviated for COBOL (continued)

| ODBC C #define symbol > 30 characters | Corresponding COBOL name       |
|---------------------------------------|--------------------------------|
| SQL_MAXIMUM_COLUMN_IN_GROUP_BY        | SQL-MAXIMUM-COLUMN-IN-GROUP-B  |
| SQL_MAXIMUM_COLUMN_IN_ORDER_BY        | SQL-MAXIMUM-COLUMN-IN-ORDER-B  |
| SQL_MAXIMUM_CONCURRENT_ACTIVITIES     | SQL-MAXIMUM-CONCURRENT-ACTIVIT |
| SQL_MAXIMUM_CONCURRENT_STATEMENTS     | SQL-MAXIMUM-CONCURRENT-STAT    |
| SQL_SQL92_FOREIGN_KEY_DELETE_RULE     | SQL-SQL92-FOREIGN-KEY-DELETE-R |
| SQL_SQL92_FOREIGN_KEY_UPDATE_RULE     | SQL-SQL92-FOREIGN-KEY-UPDATE-R |
| SQL_SQL92_NUMERIC_VALUE_FUNCTIONS     | SQL-SQL92-NUMERIC-VALUE-FUNCTI |
| SQL_SQL92_RELATIONAL_JOIN_OPERATORS   | SQL-SQL92-RELATIONAL-JOIN-OPER |
| SQL_SQL92_ROW_VALUE_CONSTRUCTOR       | SQL-SQL92-ROW-VALUE-CONSTRUCTO |
| SQL_TRANSACTION_ISOLATION_OPTION      | SQL-TRANSACTION-ISOLATION-OPTI |

## Compiling and linking programs that make ODBC calls

You must compile programs that make ODBC calls with the compiler option `CALLINT(SYSTEM)` or the `>>CALLINT SYSTEM` directive, and the compiler option `PGMNAME(MIXED)` (the ODBC entry points are case sensitive).

### RELATED REFERENCES

“CALLINT” on page 229

Chapter 15, “Compiler-directing statements,” on page 273

“PGMNAME” on page 255

## Understanding ODBC error messages

Error messages for ODBC calls can come from the ODBC driver, the database source, or the driver manager.

An error reported on an ODBC driver has the following format, where *ODBC\_component* is the component in which the error occurred:

[*vendor*] [*ODBC\_component*] message

If you get this type of error, check the last ODBC call that your application made for possible problems, or contact your ODBC application vendor.

An error that occurs in the data source includes the data source name in the following format, where *ODBC\_component* is the component that received the error from the data source that is indicated:

[*vendor*] [*ODBC\_component*] [*data\_source*] message

If you get this type of error, you did something incorrectly with the database system. Check your database system documentation for more information or consult your database administrator. In this example, you would check your Oracle documentation.

The driver manager is a DLL that establishes connections with drivers, submits requests to drivers, and returns results to applications. An error that occurs in the driver manager has the following format:

[*vendor*] [ODBC DLL] message

To deal with this type of error, check the documentation for the driver manager.

---

## Part 5. Using XML and COBOL together

|                                                                |         |
|----------------------------------------------------------------|---------|
| <b>Chapter 22. Processing XML input</b>                        | 349     |
| XML parser in COBOL                                            | 349     |
| Accessing XML documents                                        | 351     |
| Parsing XML documents                                          | 351     |
| The content of XML-EVENT                                       | 352     |
| Example: processing XML events                                 | 355     |
| Writing procedures to process XML                              | 357     |
| Example: parsing XML                                           | 359     |
| The content of XML-CODE                                        | 361     |
| The content of XML-TEXT and XML-NTEXT                          | 363     |
| Transforming XML text to COBOL data items                      | 363     |
| Understanding the encoding of XML documents                    | 364     |
| Coded character sets for XML documents                         | 365     |
| Specifying the code page                                       | 365     |
| Handling exceptions that the XML parser finds                  | 366     |
| How the XML parser handles errors                              | 367     |
| Handling conflicts in code pages                               | 370     |
| Terminating XML parsing                                        | 371     |
| <br><b>Chapter 23. Producing XML output</b>                    | <br>373 |
| Generating XML output                                          | 373     |
| Example: generating XML                                        | 375     |
| Program XGFX                                                   | 375     |
| Program Pretty                                                 | 377     |
| Output from program XGFX                                       | 378     |
| Enhancing XML output                                           | 379     |
| Example: enhancing XML output                                  | 380     |
| Example: converting hyphens in element names<br>to underscores | 382     |
| Controlling the encoding of generated XML output               | 383     |
| Handling errors in generating XML output                       | 384     |



---

## Chapter 22. Processing XML input

You can process XML input in your COBOL program by using the XML PARSE statement. The XML PARSE statement is the COBOL language interface to the high-speed XML parser, which is part of the COBOL run time.

Processing XML input involves control being passed to and received from the XML parser. You start this exchange of control with the XML PARSE statement, which specifies a processing procedure that receives control from the XML parser to handle the parser events. You use special registers in your processing procedure to exchange information with the parser.

Use these COBOL facilities to process XML input:

- XML PARSE statement to begin XML parsing and to identify the document and your processing procedure
- Processing procedure to control the parsing: receive and process the XML events and associated document fragments, and optionally handle exceptions
- Special registers to receive and pass information:
  - XML-CODE to determine the status of XML parsing
  - XML-EVENT to receive the name of each XML event
  - XML-TEXT to receive XML document fragments from an alphanumeric document
  - XML-NTEXT to receive XML document fragments from a national document

### RELATED CONCEPTS

“XML parser in COBOL”

### RELATED TASKS

“Accessing XML documents” on page 351

“Parsing XML documents” on page 351

“Understanding the encoding of XML documents” on page 364

“Handling exceptions that the XML parser finds” on page 366

“Terminating XML parsing” on page 371

### RELATED REFERENCES

Appendix F, “XML reference material,” on page 625

Extensible Markup Language (XML)

---

## XML parser in COBOL

COBOL for Windows provides an event-based interface that enables you to parse XML documents and transform them to COBOL data structures.

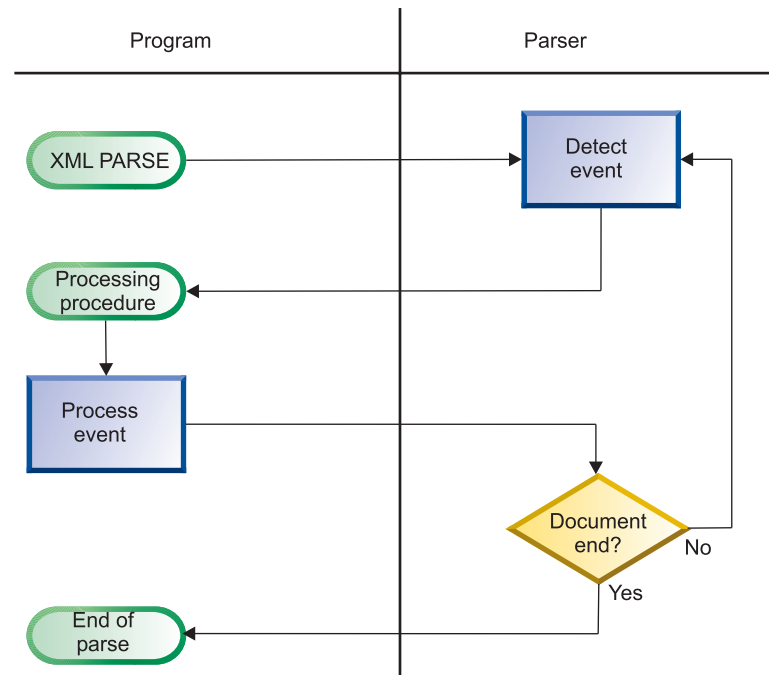
The XML parser finds fragments (associated with XML events) within the document, and your processing procedure acts on these fragments. You code your procedure to handle each XML event. Throughout this operation, control passes back and forth between the parser and your procedure.

You start this exchange with the parser by using the XML PARSE statement, in which you designate your processing procedure. Execution of this XML PARSE statement begins the parsing and establishes your processing procedure with the parser. Each

execution of your procedure causes the XML parser to continue analyzing the XML document and report the next event, passing back to your procedure the fragment that it finds, such as the start of a new element. You can also specify in the XML PARSE statement two imperative statements to which you want control to be passed at the end of the parsing: one when a normal end occurs and one when an exception condition exists.

The following figure shows a high-level overview of the basic exchange of control between the parser and your program.

### XML parsing flow overview



Normally, parsing continues until the entire XML document has been parsed.

When the XML parser parses XML documents, it checks them 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 TASKS

- “Accessing XML documents” on page 351
- “Parsing XML documents” on page 351
- “Writing procedures to process XML” on page 357
- “Understanding the encoding of XML documents” on page 364
- “Handling exceptions that the XML parser finds” on page 366
- “Terminating XML parsing” on page 371

#### RELATED REFERENCES

- “XML conformance” on page 632
- XML specification

---

## Accessing XML documents

Before you can parse an XML document with an XML PARSE statement, you must make the document available to your program. Common methods of acquiring the document are from a parameter to your program or 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) that is defined in the WORKING-STORAGE SECTION or 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 procedure for handling XML events that occur during parsing. You can also optionally specify what action should be taken after parsing finishes.

```
XML PARSE XMLDOCUMENT
 PROCESSING PROCEDURE XMLEVENT-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 data item (XMLDOCUMENT in the example above) that contains the XML document character stream. In the DATA DIVISION, you can declare the identifier as national (either a national group item or an elementary item of category national) or as alphanumeric (either an alphanumeric group item or an elementary item of category alphanumeric). If it is national, its content must be encoded in Unicode UTF-16LE, CCSID 1202. If it is alphanumeric, its content must be encoded with one of the supported single-byte EBCDIC or ASCII character sets. See the related reference below about coded characters sets for more information.

If the CHAR(EBCDIC) compiler option is in effect, do not specify the NATIVE keyword on the data description entry for the identifier if the entry describes an alphanumeric data item. If the CHAR(EBCDIC) option is in effect and the identifier is alphanumeric, the content of the identifier must be encoded in EBCDIC.

ASCII XML documents that do not contain an encoding declaration are parsed with the code page indicated by the current runtime locale. EBCDIC XML documents that do not contain an encoding declaration are parsed with the code page specified in the EBCDIC\_CODEPAGE environment variable. If the EBCDIC\_CODEPAGE environment variable is not set, they are parsed with the default EBCDIC code page selected for the current runtime locale.

**XML declaration:** If the document that you are parsing contains an XML declaration, the declaration must begin in the first byte of the document. If the string `<?xml` starts after the first byte of the document, the parser generates an exception code. The attribute names that are coded in the XML declaration must all be in lowercase characters.

Next you specify the name of the procedure (XMLEVENT-HANDLER in the example) that is to handle the XML events from the document.

In addition, you can specify either or both of the following phrases to receive control after parsing finishes:

- ON EXCEPTION, to receive control when 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 (for example, in another XML PARSE statement or in an XML GENERATE statement).

The parser passes control to the processing procedure for each XML event. Control returns to the parser when the end of the processing procedure is reached. This exchange of control between the XML parser and the processing procedure continues until one of the following events occurs:

- The entire XML document has been 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.
- You terminate the parsing process deliberately by setting the special register XML-CODE to -1 before returning to the parser.

**Special registers:** Use the XML-EVENT special register to determine which event the parser passes to your processing procedure. XML-EVENT contains an event name such as 'START-OF-ELEMENT'. The parser passes the content for the event in special register XML-TEXT or XML-NTEXT, depending on the type of the XML identifier specified in the XML PARSE statement.

#### RELATED TASKS

"Understanding the encoding of XML documents" on page 364

"Writing procedures to process XML" on page 357

"Specifying the code page with a locale" on page 180

#### RELATED REFERENCES

"Locales and code pages that are supported" on page 183

"Coded character sets for XML documents" on page 365

"The content of XML-EVENT"

XML PARSE statement (*COBOL for Windows Language Reference*)

## The content of XML-EVENT

When an event occurs during XML parsing, the XML parser writes the appropriate event name shown below to the XML-EVENT special register. (The event is passed to your processing procedure, and the text that corresponds to the event is written to either the XML-TEXT or XML-NTEXT special register.)



**ATTRIBUTE-CHARACTER**

Occurs in attribute values for the predefined entity references '&amp;','&apos;','&gt;','&lt;',' and '&quot;'. See *XML specification* for details about predefined entities.

**ATTRIBUTE-CHARACTERS**

Occurs for each fragment of an attribute value. XML text contains the fragment. An attribute value normally consists only of a single string, even if it is split across lines. The attribute value might consist of multiple events, however.

**ATTRIBUTE-NAME**

Occurs for each attribute in an element start tag or empty element tag, after a valid name is recognized. XML text contains the attribute name.

**ATTRIBUTE-NATIONAL-CHARACTER**

Occurs in attribute values for numeric character references (Unicode code points or "scalar values") of the form '&#dd.;' or '&#hh.;', where *d* and *h* represent decimal and hexadecimal digits, respectively. If the scalar value of the national character is greater than 65,535 (NX'FFFF'), XML-NTEXT contains two encoding units (a *surrogate pair*) and has a length of 4 bytes. This pair of encoding units represents a single character. Do not create characters that are not valid by splitting this pair.

**COMMENT**

Occurs for any comments in the XML document. XML text contains the data between the opening and closing comment delimiters, '<!--' and '-->', respectively.

**CONTENT-CHARACTER**

Occurs in element content for the predefined entity references '&amp;','&apos;','&gt;','&lt;',' and '&quot;'. See *XML specification* for details about predefined entities.

**CONTENT-CHARACTERS**

This event represents the principal part of an XML document: the character data between element start and end tags. XML text contains this data, which usually consists only of a single string even if it is split across lines. If the content of an element includes any references or other elements, the complete content might consist of several events. The parser also uses the CONTENT-CHARACTERS event to pass the text of CDATA sections to your program.

**CONTENT-NATIONAL-CHARACTER**

Occurs in element content for numeric character references (Unicode code points or "scalar values") of the form '&#dd.;' or '&#hh.;', where *d* and *h* represent decimal and hexadecimal digits, respectively. If the scalar value of the national character is greater than 65,535 (NX'FFFF'), XML-NTEXT contains two encoding units (a *surrogate pair*) and has a length of 4 bytes. This pair of encoding units represents a single character. Do not create characters that are not valid by splitting this pair.

**DOCUMENT-TYPE-DECLARATION**

Occurs when the parser finds a document type declaration. Document type declarations begin with the character sequence '<!DOCTYPE' and end with a right angle bracket ('>') character; some fairly complicated grammar rules describe the content in between. See *XML specification* for details. For this event, XML text contains the entire declaration, including the opening and closing character sequences. This is the only event for which XML text includes the delimiters.

**ENCODING-DECLARATION**

Occurs within the XML declaration for the optional encoding declaration. XML text contains the encoding value.

**END-OF-CDATA-SECTION**

Occurs when the parser recognizes the end of a CDATA section.

**END-OF-DOCUMENT**

Occurs when document parsing has completed.

**END-OF-ELEMENT**

Occurs once for each element end tag or empty element tag when the parser recognizes the closing angle bracket of the tag. XML text contains the element name.

**EXCEPTION**

Occurs when an error in processing the XML document is detected. For encoding conflict exceptions, which are signaled before parsing begins, XML-TEXT (for XML documents in an alphanumeric data item) or XML-NTEXT (for XML documents in a national data item) either is zero length or contains only the encoding declaration value from the document.

**PROCESSING-INSTRUCTION-DATA**

Occurs for the data that follows the PI target, up to but not including the PI closing character sequence, '?>'. XML text contains the PI data, which includes trailing, but not leading, white-space characters.

**PROCESSING-INSTRUCTION-TARGET**

Occurs when the parser recognizes the name that follows the opening character sequence, '<?', of a processing instruction (PI). PIs allow XML documents to contain special instructions for applications.

**STANDALONE-DECLARATION**

Occurs within the XML declaration for the optional standalone declaration. XML text contains the standalone value.

**START-OF-CDATA-SECTION**

Occurs at the start of a CDATA section. CDATA sections begin with the string '<![CDATA[' and end with the string ']]>'. Such sections are used to "escape" blocks of text that contain characters that would otherwise be recognized as XML markup. XML text always contains the opening character sequence '<![CDATA['. The parser passes the content of a CDATA section between these delimiters as a single CONTENT-CHARACTERS event.

**START-OF-DOCUMENT**

Occurs once, at the beginning of the parsing of the document. XML text is the entire document, including any line-control characters such as LF (Line Feed) or NL (New Line).

**START-OF-ELEMENT**

Occurs once for each element start tag or empty element tag. XML text is set to the element name.

**UNKNOWN-REFERENCE-IN-ATTRIBUTE**

Occurs within attribute values for entity references other than the five predefined entity references, as shown for ATTRIBUTE-CHARACTER above.

**UNKNOWN-REFERENCE-IN-CONTENT**

Occurs within element content for entity references other than the predefined entity references, as shown for CONTENT-CHARACTER above.

#### VERSION-INFORMATION

Occurs within the optional XML declaration for the version information. XML text contains the version value. An *XML declaration* is XML text that specifies the version of XML that is used and the encoding of the document.

“Example: processing XML events”

#### RELATED CONCEPTS

“The content of XML-TEXT and XML-NTEXT” on page 363

#### RELATED TASKS

“Parsing XML documents” on page 351

“Writing procedures to process XML” on page 357

#### RELATED REFERENCES

XML-EVENT (*COBOL for Windows Language Reference*)

4.6 Predefined entities (*XML specification*)

## Example: processing XML events

Although short, the following sample XML document contains many XML events. These events are shown below the document in the order in which they would occur during parsing, and the exact text for each XML event is delimited by angle brackets (<<>>).

In general, this text can be the content of either XML-TEXT or XML-NTEXT. The sample XML document below does not contain any text that requires XML-NTEXT, however, and thus uses only XML-TEXT.

This example begins with an XML declaration. If an XML declaration occurs in a document that you are parsing, the declaration must begin in the first byte of the document; otherwise, the parser generates an exception code. The attribute names that are coded in the XML declaration must all be in lowercase characters.

```
<?xml version="1.0" encoding="ibm-1140" standalone="yes" ?>
<!--This document is just an example-->
<sandwich>
 <bread type="baker's best" />
 <?spread please use real mayonnaise ?>
 <meat>Ham & turkey</meat>
 <filling>Cheese, lettuce, tomato, etc.</filling>
 <![CDATA[We should add a <relish> element in future!]]>
</sandwich>junk
```

#### START-OF-DOCUMENT

The text for this sample is 336 characters in length.

#### VERSION-INFORMATION

<<1.0>>

#### ENCODING-DECLARATION

<<ibm-1140>>

#### STANDALONE-DECLARATION

<<yes>>

#### DOCUMENT-TYPE-DECLARATION

The sample does not have a document type declaration.

#### COMMENT

<<This document is just an example>>

**START-OF-ELEMENT**

In the order that they occur as START-OF-ELEMENT events:

1. <<sandwich>>
2. <<bread>>
3. <<meat>>
4. <<filling>>

**ATTRIBUTE-NAME**

<<type>>

**ATTRIBUTE-CHARACTERS**

In the order in which they occur as ATTRIBUTE-CHARACTERS events:

1. <<baker>>
2. <<s best>>

Notice that the value of the type attribute in the sample consists of three fragments: the string 'baker', the single character "'", and the string 's best'. The single-character fragment "'" is passed separately as an ATTRIBUTE-CHARACTER event.

**ATTRIBUTE-CHARACTER**

<<'>>

**ATTRIBUTE-NATIONAL-CHARACTER**

The sample does not contain a numeric character reference.

**END-OF-ELEMENT**

In the order that they occur as END-OF-ELEMENT events:

1. <<bread>>
2. <<meat>>
3. <<filling>>
4. <<sandwich>>

**PROCESSING-INSTRUCTION-TARGET**

<<spread>>

**PROCESSING-INSTRUCTION-DATA**

<<please use real mayonnaise >>

**CONTENT-CHARACTERS**

In the order that they occur as CONTENT-CHARACTERS events:

1. <<Ham >>
2. << turkey>>
3. <<Cheese, lettuce, tomato, etc.>>
4. <<We should add a <relish> element in future!>>

Notice that the content of the meat element in the sample consists of the string 'Ham ', the character '&', and the string ' turkey'. The single-character fragment '&' is passed separately as a CONTENT-CHARACTER event. Also notice the trailing and leading spaces, respectively, in these two string fragments.

**CONTENT-CHARACTER**

<<&>>

**CONTENT-NATIONAL-CHARACTER**

The sample does not contain a numeric character reference.

**START-OF-CDATA-SECTION**

&lt;&lt;&lt;![CDATA[&gt;&gt;

**END-OF-CDATA-SECTION**

&lt;&lt;]]&gt;&gt;&gt;

**UNKNOWN-REFERENCE-IN-ATTRIBUTE**

The sample does not have any unknown entity references.

**UNKNOWN-REFERENCE-IN-CONTENT**

The sample does not have any unknown entity references.

**END-OF-DOCUMENT**

XML text is empty for the END-OF-DOCUMENT event.

**EXCEPTION**

The part of the document that was parsed up to and including the point where the exception (the superfluous 'junk' after the </sandwich> tag) was detected.

**RELATED CONCEPTS**

"The content of XML-TEXT and XML-NTEXT" on page 363

**RELATED TASKS**

"Parsing XML documents" on page 351

"Writing procedures to process XML"

**RELATED REFERENCES**

"The content of XML-EVENT" on page 352

XML-EVENT (*COBOL for Windows Language Reference*)

4.6 Predefined entities (*XML specification*)

2.8 Prolog and document type declaration (*XML specification*)

## Writing procedures to process XML

In your processing procedure, code statements to handle XML events.

For each event that the parser encounters, it passes information to your processing procedure in several special registers, as shown in the following table. Use these registers to populate the data structures and to control the processing.

When used in nested programs, these special registers are implicitly defined as GLOBAL in the outermost program.

Table 47. Special registers used by the XML parser

Special register	Content	Implicit definition and usage
XML-EVENT <sup>1,3</sup>	The name of the XML event	PICTURE X(30) USAGE DISPLAY VALUE SPACE
XML-CODE <sup>2</sup>	An exception code or zero for each XML event	PICTURE S9(9) USAGE BINARY VALUE ZERO
XML-TEXT <sup>1,4</sup>	Text (corresponding to the event that the parser encountered) from the XML document if you specify an alphanumeric item for the XML PARSE identifier	Variable-length elementary category alphanumeric item; size limit of 2,147,483,646 bytes

Table 47. Special registers used by the XML parser (continued)

Special register	Content	Implicit definition and usage
XML-NTEXT <sup>1</sup>	Text (corresponding to the event that the parser encountered) from the XML document if you specify a national item for the XML PARSE identifier	Variable-length elementary category national item; size limit of 2,147,483,646 bytes
<ol style="list-style-type: none"> <li>1. You cannot use this special register as a receiving data item.</li> <li>2. The XML GENERATE statement also uses XML-CODE. Therefore, if you code 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.</li> <li>3. The content of this special register is encoded according to the setting of the CHAR compiler option (EBCDIC, NATIVE, or S390).</li> <li>4. The content of XML-TEXT depends on the encoding of the source XML document: <ul style="list-style-type: none"> <li>• If the source XML document is encoded in EBCDIC, the content of XML-TEXT is encoded in EBCDIC.</li> <li>• If the source XML document is encoded in ASCII in a NATIVE alphanumeric data item, the content of XML-TEXT is encoded in ASCII. You can however convert the ASCII content of XML-TEXT to EBCDIC as follows: Function DISPLAY-OF(Function NATIONAL-OF(XML-TEXT 819))</li> </ul> </li> </ol>		

**Restriction:** A processing procedure must not directly execute an XML PARSE statement. However, if a processing procedure passes control to a method or outermost program by using an INVOKE or CALL statement, the target method or 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 compiler inserts a return mechanism after the last statement in each processing procedure. You can code a STOP RUN statement in a processing procedure to end the run unit. However, an EXIT PROGRAM statement (when a CALL statement is active) or a GOBACK statement does not return control to the parser. Using either of these statements in a processing procedure results in a severe error.

“Example: parsing XML” on page 359

#### RELATED CONCEPTS

“The content of XML-CODE” on page 361

“The content of XML-TEXT and XML-NTEXT” on page 363

#### RELATED TASKS

“Transforming XML text to COBOL data items” on page 363

“Converting to or from national (Unicode) representation” on page 167

#### RELATED REFERENCES

“XML PARSE exceptions that allow continuation” on page 625

“XML PARSE exceptions that do not allow continuation” on page 629

XML-CODE (*COBOL for Windows Language Reference*)

XML-EVENT (*COBOL for Windows Language Reference*)

XML-NTEXT (*COBOL for Windows Language Reference*)

XML-TEXT (*COBOL for Windows Language Reference*)

## Example: parsing XML

This example shows the basic organization of an XML PARSE statement and of a processing procedure.

The XML document is shown in the source so that you can follow the flow of the parsing. The output of the program is also shown below. Compare the document to the output of the program to follow the interaction of the parser and the processing procedure, and to match events to document fragments.

```
Process flag(i,i)
Identification division.
Program-id. xmlsampl.
```

```
Data division.
Working-storage section.
```

```

* XML document, encoded as initial values of data items. *

```

```
1 xml-document.
2 pic x(39) value '<?xml version="1.0" encoding="ibm-1140"'.
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(35) value ' <bread type="baker's best"/>'.
2 pic x(41) value ' <?spread please use real mayonnaise ?>'.
2 pic x(31) value ' <meat>Ham & turkey</meat>'.
2 pic x(40) value ' <filling>Cheese, lettuce, tomato, etc.'.
2 pic x(10) value '</filling>'.
2 pic x(35) value ' <![CDATA[We should add a <relish>'.
2 pic x(22) value ' element in future!]]>'.
2 pic x(31) value ' <listprice>$4.99 </listprice>'.
2 pic x(27) value ' <discount>0.10</discount>'.
2 pic x(11) value '</sandwich>'.
1 xml-document-length computational pic 999.
```

```

* Sample data definitions for processing numeric XML content. *

```

```
1 current-element pic x(30).
1 xfr-ed pic x(9) justified.
1 xfr-ed-1 redefines xfr-ed pic 999999.99.
1 list-price computational pic 9v99 value 0.
1 discount computational pic 9v99 value 0.
1 display-price pic $$9.99.
```

```
Procedure division.
mainline section.
```

```
XML PARSE xml-document PROCESSING PROCEDURE xml-handler
ON EXCEPTION
display 'XML document error ' XML-CODE
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 '*****+-----'
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 '>'
 move XML-TEXT to current-element
 when 'CONTENT-CHARACTERS'
 display 'Content characters: <' XML-TEXT '>'
* ==> Transform XML content to operational COBOL data item...
 evaluate current-element
 when 'listprice'
* ==> Using function NUMVAL-C...
 compute list-price = function numval-c(XML-TEXT)
 when 'discount'
* ==> Using de-editing of a numeric edited item...
 move XML-TEXT to xfr-ed
 move xfr-ed-1 to discount
 end-evaluate
 when 'END-OF-ELEMENT'
 display 'End element tag: <' XML-TEXT '>'
 move spaces to current-element
 when 'START-OF-DOCUMENT'
 compute xml-document-length = function length(XML-TEXT)
 display 'Start of document: length=' xml-document-length
 ' characters.'
 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: <' XML-TEXT '>'
 when 'END-OF-CDATA-SECTION'
 display 'End of CData: <' XML-TEXT '>'
 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 xml-document-length = function length (XML-TEXT)
 display 'Exception ' XML-CODE ' at offset '
 xml-document-length '.'
 when other
 display 'Unexpected XML event: ' XML-EVENT '.'
 end-evaluate
.
End program xmlsaml.

```

**Output from parse example:** From the following output you can see which parsing events came from which fragments of the document:



```

Start of document: length=390 characters.
Version: <1.0>
Encoding: <ibm-1140>
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!


```



## The content of XML-CODE

When the parser returns control to your XML PARSE statement, special register XML-CODE contains the most recent value that was set by the parser or by your processing procedure.

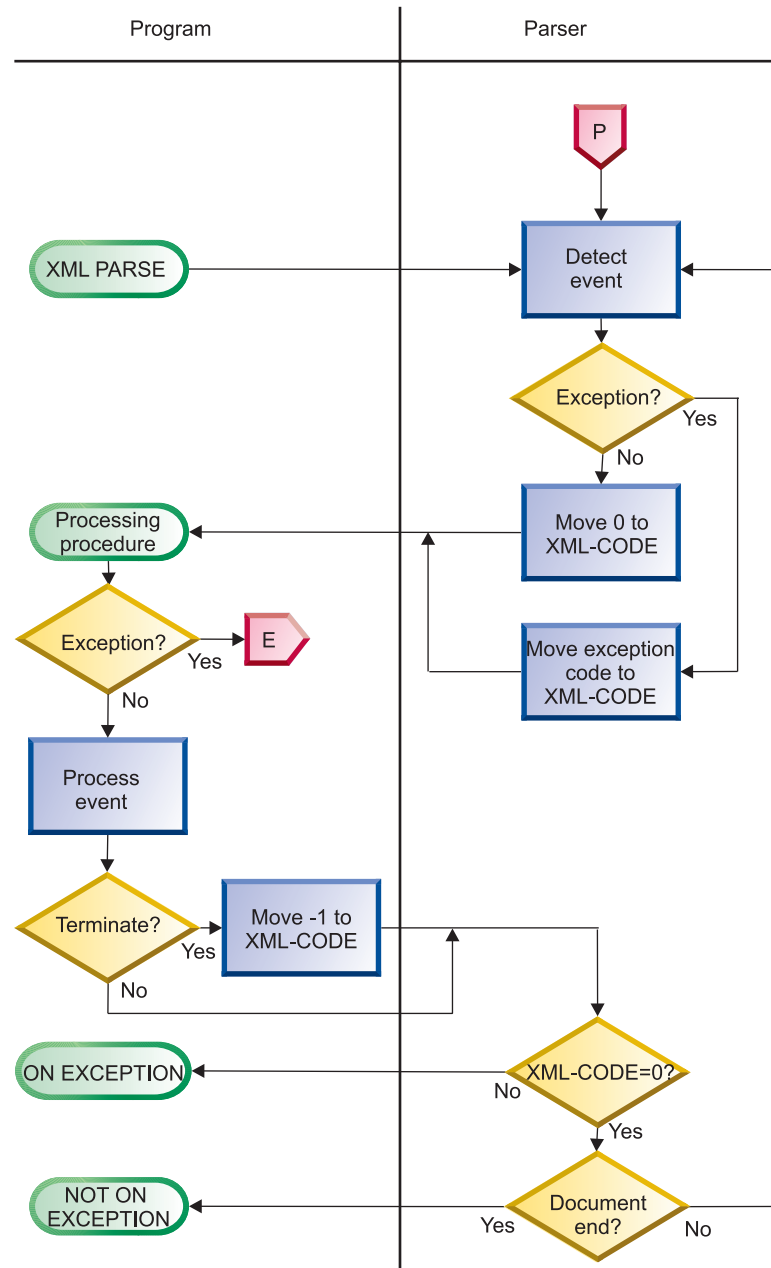
For each event except the EXCEPTION event, the value of XML-CODE is zero. If you set XML-CODE to -1 before you return control to the XML parser for an event other than EXCEPTION, processing stops with a user-initiated exception (as indicated by the returned XML-CODE value of -1). The result of changing XML-CODE to any other nonzero value before returning from any event is undefined.

For the EXCEPTION event, special register XML-CODE contains the exception code.

The following figure shows the flow of control between the parser and your processing procedure and shows how XML-CODE is used to pass information between them. The off-page connectors (for example, ) connect the multiple

charts in this information. In particular,  in the following figure connects to the chart “Control flow for XML exceptions” on page 368, and  connects from “XML CCSID exception flow control” on page 370.

### Control flow between XML parser and program, showing XML-CODE usage



#### RELATED CONCEPTS

“The content of XML-TEXT and XML-NTEXT” on page 363

#### RELATED TASKS

“Writing procedures to process XML” on page 357

“Handling exceptions that the XML parser finds” on page 366

#### RELATED REFERENCES

Appendix F, “XML reference material,” on page 625  
XML-CODE (*COBOL for Windows Language Reference*)

### The content of XML-TEXT and XML-NTEXT

The special registers XML-TEXT and XML-NTEXT are mutually exclusive. When the parser sets XML-TEXT, XML-NTEXT is undefined (length 0). When the parser sets XML-NTEXT, XML-TEXT is undefined (length 0).

The type of the XML PARSE identifier determines which of these two special registers is set, except for the ATTRIBUTE-NATIONAL-CHARACTER and CONTENT-NATIONAL-CHARACTER events. For these two events, XML-NTEXT is set regardless of the data item that you specify as the XML PARSE identifier.

To determine how many national characters XML-NTEXT contains, use the LENGTH function. LENGTH OF XML-NTEXT contains the number of bytes, rather than characters, used by XML-NTEXT.

#### RELATED CONCEPTS

“The content of XML-CODE” on page 361

#### RELATED TASKS

“Writing procedures to process XML” on page 357

#### RELATED REFERENCES

XML-NTEXT (*COBOL for Windows Language Reference*)  
XML-TEXT (*COBOL for Windows 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 COBOL data items.

For alphanumeric items, decide whether the XML data should go at the left (default) end of a COBOL item or at the right end. If it should go at the right end, specify the JUSTIFIED RIGHT clause in the declaration of the COBOL item.

Give special consideration to numeric XML values, particularly “decorated” monetary values such as ‘\$1,234.00’ or ‘\$1234’. These two strings mean the same thing in XML, but would need completely different declarations as COBOL sending fields. Use one of these techniques when you move XML data to COBOL data items:

- If the format is reasonably regular, code a MOVE to an alphanumeric item that is redefined 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 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, use of these functions is at the expense of performance.

#### RELATED TASKS

“Using national data (Unicode) in COBOL” on page 160

“Writing procedures to process XML” on page 357

---

## Understanding the encoding of XML documents

The parser decides how to process your document by using certain sources of information about the document encoding. These sources of encoding information must be consistent with one another. The parser signals an XML exception event if it finds a conflict.

The sources of encoding information are:

- The *basic document encoding*
- The *type of the data item* that contains the document
- Any *document encoding declaration*
- The *external code page*, if applicable
- The set of *supported code pages*

The *basic document encoding* is one of the following encoding categories that the parser determines by examining the first few bytes of the XML document:

- ASCII
- EBCDIC
- Unicode UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

The *type of the data item* that contains the document is also relevant. The parser supports the following combinations:

- Category national data items whose contents are encoded using Unicode UTF-16
- Native alphanumeric data items whose contents are encoded using one of the supported single-byte ASCII code pages
- Host alphanumeric data items whose contents are encoded using one of the supported single-byte EBCDIC code pages

The discovery that the encoding is UTF-16 also provides the code-page information, CCSID 1202 UTF-16LE (little-endian), since Unicode is effectively a single large code page. Thus, if the parser finds a UTF-16 document in a national data item, it ignores external code-page information.

But if the basic 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 pages.

The *document encoding declaration* is an optional part of the XML declaration at the beginning of the document. (See the related task about specifying the code page for details.)

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 either:

- The code page that you specify in the EBCDIC\_CODEPAGE environment variable
- The default EBCDIC code page selected for the current runtime locale if you do not set the EBCDIC\_CODEPAGE environment variable

Finally, the encoding must be one of the *supported code pages*. (See the related reference below about coded character sets for XML documents for details.)

#### RELATED TASKS

“Specifying the code page”

#### RELATED REFERENCES

“Locales and code pages that are supported” on page 183

“Coded character sets for XML documents”

“XML PARSE exceptions that allow continuation” on page 625

“XML PARSE exceptions that do not allow continuation” on page 629

## Coded character sets for XML documents

Documents in national data items must be encoded with Unicode UTF-16, CCSID 1202.

XML documents in alphanumeric data items must be encoded using one of the single-byte ASCII or EBCDIC code pages shown in the table of locales and code pages that are supported, referenced below. Documents in native alphanumeric data items must be encoded with an ASCII code page. Documents in host alphanumeric data items must be encoded with an EBCDIC code page. The single-byte code pages are those for which the “Language group” column (the rightmost column of the table) does not specify ideographic languages. XML documents cannot be encoded in code page 1208, Unicode UTF-8.

To parse XML documents that are encoded in other code pages, first convert the documents to national characters 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.

#### RELATED TASKS

“Converting to or from national (Unicode) representation” on page 167

“Specifying the code page”

#### RELATED REFERENCES

“Locales and code pages that are supported” on page 183

## Specifying the code page

The preferred way to specify the code page for parsing an XML document is to omit the encoding declaration from the document and to rely on external code-page information.

Omitting the XML declaration makes it possible to transmit an XML document between heterogeneous systems without requiring that you update the encoding declaration to reflect any translation imposed by the transmission process.

Unicode XML documents do not require additional code-page information because Unicode effectively consists of a single large code page. The code page used for

parsing an ASCII or EBCDIC XML document that does not have an encoding declaration is the runtime ASCII or EBCDIC code page.

You can also specify the encoding information for an XML document in the XML declaration with which most XML documents begin. (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.) This is an example of an XML declaration that includes an encoding declaration:

```
<?xml version="1.0" encoding="ibm-1140" ?>
```

Specify the encoding declaration in either of the following ways:

- Specify the CCSID number (with or without any number of leading zeros) optionally prefixed by any of the following strings (in any mixture of uppercase and lowercase):
  - IBM-
  - IBM\_
  - CCSID-
  - CCSID\_
- Use one of the following supported aliases (in any mixture of uppercase and lowercase):

*Table 48. Aliases for XML encoding declarations*

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
1202	UTF-16

#### RELATED TASKS

“Understanding the encoding of XML documents” on page 364

#### RELATED REFERENCES

“Locales and code pages that are supported” on page 183

---

## Handling exceptions that the XML parser finds

If the exception code that the XML parser passes in XML-CODE is within a certain range, you can handle the exception event with a processing procedure.

To handle an exception event in your processing procedure, do these steps:

1. Check the contents of XML-CODE.
2. Handle the exception as appropriate.
3. Set XML-CODE to zero to indicate that you handled the exception.
4. Return control to the parser. The exception condition then no longer exists.

You can handle exceptions this way only if the exception code that is passed in XML-CODE is within one of the following ranges, which indicate that an encoding conflict was detected:

- 50-99
- 100,001-165,535
- 200,001-265,535

You can do limited handling of exceptions for which the exception code passed in XML-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 be useful as a way of discovering multiple errors in the document.) At the end of parsing after an exception in this range, control is passed to the statement that you specify in the ON EXCEPTION phrase, if any, otherwise to the end of the XML PARSE statement. The special register XML-CODE contains the code for the most recent exception that the parser detected.

For all other exceptions, the parser signals no further events, and passes control to the statement that you specify in the ON EXCEPTION phrase. In this case, XML-CODE contains the original exception number even if you set XML-CODE to zero 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 that you specify in the ON EXCEPTION phrase. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement.

If you return control to the parser with XML-CODE set to a nonzero value that is different from the original exception code, the results are undefined.

If no unhandled exceptions occur before the end of parsing, control is passed to the statement that you specify in the NOT ON EXCEPTION phrase (normal end of parsing). If you do not code a NOT ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. The special register XML-CODE contains zero.

#### RELATED CONCEPTS

“How the XML parser handles errors”

“The content of XML-CODE” on page 361

#### RELATED TASKS

“Writing procedures to process XML” on page 357

“Understanding the encoding of XML documents” on page 364

“Handling conflicts in code pages” on page 370

#### RELATED REFERENCES

“XML PARSE exceptions that allow continuation” on page 625

“XML PARSE exceptions that do not allow continuation” on page 629

XML-CODE (*COBOL for Windows Language Reference*)

## 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 provides the following information in special registers:





- XML-EVENT contains 'EXCEPTION'.
- XML-CODE contains a numeric exception code.
- XML-TEXT (for XML documents in an alphanumeric data item) or XML-NTEXT (for XML documents in a national data item) contains the document text up to and including the point where the exception was detected.

You might be able to handle the exception in your processing procedure and continue parsing if the numeric 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. 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) either is zero length or contains only the encoding declaration value from the document.

Exceptions in the range 1-49 are fatal errors according to the XML specification. Therefore the parser does not continue normal parsing even if you handle the exception. However, the parser does continue scanning for further errors until it reaches the end of the document or encounters an error that does not allow continuation. For these exceptions, the parser does not signal any further normal events except the END-OF-DOCUMENT event.

Use the following figure to understand the flow of control between the parser and your processing procedure. The figure illustrates how you can handle certain exceptions and can use XML-CODE to identify the exceptions. The off-page connectors (for example, ) connect the multiple charts in this information. In particular,  connects to the chart “XML CCSID exception flow control” on page 370. Within this figure,  and  serve both as off-page and on-page connectors.

### **Control flow for XML exceptions**








## Handling conflicts in code pages

Exception events in which the document item is alphanumeric and the exception code in XML-CODE is between 100,001 and 165,535 or between 200,001 and 265,535 indicate that the code page of the document (as specified by its encoding declaration) conflicts with the external code-page information.

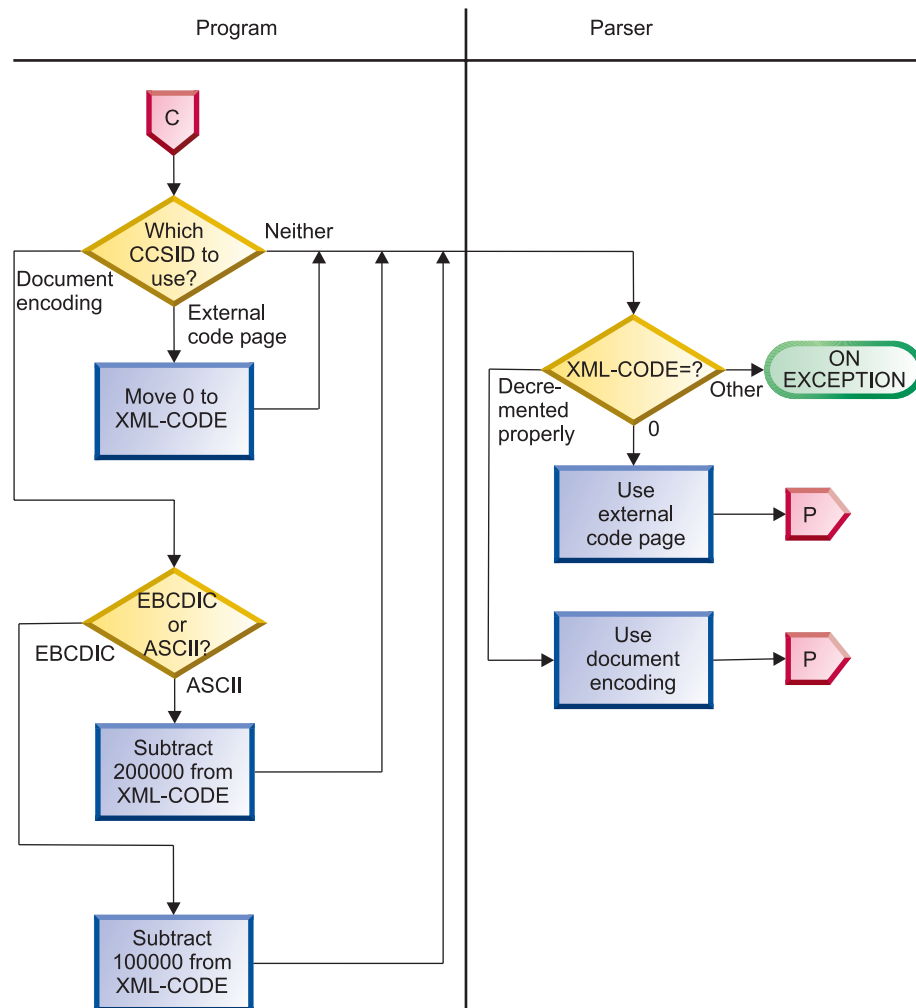
In this special case, you can choose to parse with the code page of the document by subtracting 100,000 or 200,000 from the value in XML-CODE (depending on whether it is an EBCDIC code page or ASCII code page, respectively). For instance, if XML-CODE contains 101,140, the code page of the document is 1140. Alternatively, you can choose to parse with 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 a code-page 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 document 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 where 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 to the exception event.

The following figure illustrates these actions. The off-page connectors (for example, ) connect the multiple charts in this information. In particular,  in the following figure connects to “Control flow between XML parser and program, showing XML-CODE usage” on page 362, and  connects from “Control flow for XML exceptions” on page 368.

### XML CCSID exception flow control



In this figure, *CCSID* (coded character set identifier) is a value from 1 to 65,536 that denotes the code page.

#### RELATED CONCEPTS

“How the XML parser handles errors” on page 367

#### RELATED TASKS

“Understanding the encoding of XML documents” on page 364

“Handling exceptions that the XML parser finds” on page 366

#### RELATED REFERENCES

“XML PARSE exceptions that allow continuation” on page 625

“XML PARSE exceptions that do not allow continuation” on page 629

XML-CODE (*COBOL for Windows Language Reference*)

## Terminating XML parsing

You can terminate parsing deliberately by setting XML-CODE to -1 in your processing procedure before returning to the parser from any normal XML event (that is, not an EXCEPTION event). You can use this technique when you have seen enough of the document or have detected some irregularity in the document that precludes further meaningful processing.

In this case, the parser does not signal any further events although an exception condition exists. Therefore, control returns to the ON EXCEPTION phrase if specified. In the imperative statement of the ON EXCEPTION phrase, you can test whether XML-CODE is -1, which indicates that you terminated parsing deliberately. If you do not specify an ON EXCEPTION phrase, control returns to the end of the XML PARSE statement.

You can also terminate parsing after any XML exception event by returning to the parser without changing 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 exception number.

#### RELATED CONCEPTS

“How the XML parser handles errors” on page 367

#### RELATED TASKS

“Handling exceptions that the XML parser finds” on page 366

#### RELATED REFERENCES

XML-CODE (*COBOL for Windows Language Reference*)

---

## Chapter 23. Producing XML output

You can produce XML output from a COBOL program by using the XML GENERATE statement. In the XML GENERATE statement, you can also identify a field to receive a count of the number of characters of XML output generated, and a statement to receive control if an exception occurs.

To produce XML output, use:

- The XML GENERATE statement to identify the source and target data items, count field, and ON EXCEPTION statement
- The special register XML-CODE 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”

“Enhancing XML output” on page 379

“Controlling the encoding of generated XML output” on page 383

“Handling errors in generating XML output” on page 384

---

## Generating XML output

To transform COBOL data to XML, use the XML GENERATE statement.

```
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 eight 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 declare 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).

The receiving identifier must be national if the external code pages are multibyte code pages or the XML output will contain any data from the COBOL source record that has any of the following characteristics:

- Is of class national or class DBCS
- Has a multibyte name (that is, is a data item whose name contains multibyte characters)
- Is an alphanumeric item that contains multibyte characters

If the receiving data item is alphanumeric, the code pages indicated by the compile-time locale and the runtime locale must be identical.

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. Do not specify the RENAMES clause in the data description of that data item.

If the source data item is an alphanumeric group item or a national group item, the source data item is processed with group semantics, not as an elementary item. Any groups that are subordinate to the source data item are also processed with group semantics.

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

There must be at least one elementary data item that is not ignored when you generate XML. For the data items that are not ignored, ensure that the identifier that you transform to XML satisfies these conditions when you declare it in the DATA DIVISION:

- Each elementary data item is either an index data item or belongs to one of these classes:
  - Alphabetic
  - Alphanumeric
  - DBCS
  - Numeric
  - National

That is, no elementary data item is described with the USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE phrase.

- Each data-name other than FILLER is unique within the immediately containing group, if any.
- Any multibyte data-names, when converted to Unicode, are legal as names in the XML specification, version 1.0.
- The data item or items do not specify the DATE FORMAT clause, or the DATEPROC compiler option is not in effect.

An XML declaration is not generated. No 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 positions that are filled during generation of the XML output. Declare the count field as an integer data item that does not have the symbol P in its PICTURE string. You can use the count field and reference modification 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.

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:

- Special register XML-CODE 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, special register XML-CODE 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”

#### RELATED TASKS

“Controlling the encoding of generated XML output” on page 383

“Handling errors in generating XML output” on page 384

#### RELATED REFERENCES

“Locales and code pages that are supported” on page 183

Classes and categories of data (*COBOL for Windows Language Reference*)

XML GENERATE statement (*COBOL for Windows Language Reference*)

Operation of XML GENERATE (*COBOL for Windows Language Reference*)

Extensible Markup Language (XML)

## 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 in XML elements 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
 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 element pic x(30).
 05 indent pic x(20).
 05 buffer pic x(100).
 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 posd
 Move 1 to pose
 Xml parse doc processing procedure p
 Goback.
 p.
 Evaluate xml-event
 When 'START-OF-ELEMENT'
 If element not = space
 If depth > 1
 Display indent(1:2 * depth - 2) buffer(1:pose - 1)
 Else
 Display buffer(1:pose - 1)
 End-if
 End-if
 Move xml-text to element
 Add 1 to depth
 Move 1 to pose
 String '<' xml-text '>' delimited by size into buffer
 with pointer pose
 Move pose to posd
 When 'CONTENT-CHARACTERS'
 String xml-text delimited by size into buffer
 with pointer posd
 When 'CONTENT-CHARACTER'
 String xml-text delimited by size into buffer
 with pointer posd
 When 'END-OF-ELEMENT'
 Move space to element
 String '</' xml-text '>' delimited by size into buffer

```

```

 with pointer posd
 If depth > 1
 Display indent(1:2 * depth - 2) buffer(1:posd - 1)
 Else
 Display buffer(1:posd - 1)
 End-if
 Subtract 1 from depth
 Move 1 to posd
 When other
 Continue
 End-evaluate
.
End program Pretty.

```

## Output from program XGFX

```

<purchaseOrder>
 <orderDate>1999-10-20</orderDate>
 <shipTo>
 <country>US</country>
 <name>Alice Smith</name>
 <street>123 Maple Street</street>
 <city>Mill Valley</city>
 <state>CA</state>
 <zip>90952</zip>
 </shipTo>
 <billTo>
 <country>US</country>
 <name>Robert Smith</name>
 <street>8 Oak Avenue</street>
 <city>Old Town</city>
 <state>PA</state>
 <zip>95819</zip>
 </billTo>
 <orderComment>Hurry, my lawn is going wild!</orderComment>
 <items>
 <item>
 <partNum>872-AA</partNum>
 <productName>Lawnmower</productName>
 <quantity>1</quantity>
 <USPrice>148.95</USPrice>
 <shipDate> </shipDate>
 <itemComment>Confirm this is electric</itemComment>
 </item>
 </items>
 <items>
 <item>
 <partNum>926-AA</partNum>
 <productName>Baby Monitor</productName>
 <quantity>1</quantity>
 <USPrice>39.98</USPrice>
 <shipDate>1999-05-21</shipDate>
 <itemComment> </itemComment>
 </item>
 </items>
</purchaseOrder>

```

### RELATED TASKS

Chapter 22, “Processing XML input,” on page 349

### RELATED REFERENCES

Operation of XML GENERATE (*COBOL for Windows 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 definition of the data is not of the required data type. Perhaps only the redefinitions (which are ignored by the XML GENERATE statement) have the appropriate format.
- The required data items are nested too deeply within irrelevant subordinate groups. The XML output should be “flattened” rather than hierarchical as it would be by default.
- The required data items are broken up into too many components, and should be output as the content of the containing group.
- The group item contains the required information but in the wrong order.

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.

An alternative approach that has some advantages is to provide a redefinition of the original group data item, and to generate the XML output from that redefinition. To do so, start from the original set of data descriptions, and make these changes:

- Exclude elementary data items from the generated XML either by renaming them to FILLER or by deleting their names.
- Provide more meaningful and appropriate names for the selected elementary items and for the group items that contain them.
- Remove unneeded intermediate group items to flatten the hierarchy.
- Specify different data types to obtain the desired trimming behavior.
- Choose a different order for the output by using a sequence of XML GENERATE statements.

The safest way to accomplish these changes is to use another copy of the original declarations accompanied by one or more REPLACE compiler-directing statements.

“Example: enhancing XML output” on page 380

You might also find when you generate an XML document that some of the element names and element values contain hyphens. You might want to convert the hyphens in the element names to underscores without changing the hyphens that are in the element values. The example that is referenced below shows a way to do so.

“Example: converting hyphens in element names to underscores” on page 382

## Example: enhancing XML output

The following example shows how you can improve 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 X(02).
 25 CDR-LIFE-BP-DASH PIC X(02).
 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(02).
 25 CDR-LIFE-QTY-SPACE1 PIC X(02).
 25 FILLER PIC X(02).
 20 CDR-LIFE-BASE-QTY-ED REDEFINES
 CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.
 20 CDR-LIFE-BASE-VALUE PIC X(15).
 20 CDR-LIFE-BASE-VALUE-ED REDEFINES
 CDR-LIFE-BASE-VALUE
 PIC $(4),$$,$$9.99.
15 CDR-LIFE-BASE-TOT-VALUE-LINE.
 20 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 375), 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>$2</CDR-LIFE-BP-SPACE>
 <CDR-LIFE-BP-DASH>3.</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>1</CDR-LIFE-QTY-SPACE>
 <CDR-LIFE-QTY-DASH>23</CDR-LIFE-QTY-DASH>
 <CDR-LIFE-QTY-SPACE1>.0</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>$3</CDR-LIFE-BP-SPACE>
 <CDR-LIFE-BP-DASH>4.</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> 2</CDR-LIFE-QTY-SPACE>
 <CDR-LIFE-QTY-DASH>34</CDR-LIFE-QTY-DASH>
 <CDR-LIFE-QTY-SPACE1>.0</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-LINE>
 <CDR-LIFE-BASE-TOT-VALUE>Very high!</CDR-LIFE-BASE-TOT-VALUE>
</CDR-LIFE-BASE-TOT-VALUE-LINE>
</CDR-LIFE-BASE-VALUES-BOX>

```

This generated XML suffers from several problems:

- The element names are long and not very meaningful.
- 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 redefining the storage as follows:

```

1 BaseValues redefines CDR-LIFE-BASE-VALUES-BOX.
2 BaseValueDate pic x(8).
2 BaseValueLine occurs 2 times.
3 Description pic x(15).
3 pic x(9).
3 BaseDate pic x(8).
3 BasePrice pic x(6) justified.
3 BaseQuantity pic x(14) justified.
3 BaseValue pic x(15) justified.
2 TotalValue pic x(15).

```

The result of generating and formatting XML from the set of definitions of the data values shown above is more usable:

```

<BaseValues>
 <BaseValueDate>01/02/03</BaseValueDate>
 <BaseValueLine>
 <Description>First</Description>
 <BaseDate>01/01/01</BaseDate>
 <BasePrice>$23.00</BasePrice>
 <BaseQuantity>123.000</BaseQuantity>
 <BaseValue>$765.00</BaseValue>
 </BaseValueLine>
 <BaseValueLine>
 <Description>Second</Description>
 <BaseDate>02/02/02</BaseDate>
 <BasePrice>$34.00</BasePrice>
 <BaseQuantity>234.000</BaseQuantity>
 <BaseValue>$654.00</BaseValue>
 </BaseValueLine>
 <TotalValue>Very high!</TotalValue>
</BaseValues>

```

You can redefine the original data definition directly, as shown above. However, it is generally safer to use the original definition but to modify it suitably using the text-manipulation capabilities of the compiler. An example is shown in the REPLACE compiler-directing statement below. This REPLACE statement might appear complicated, but it has the advantage of being self-maintaining if the original data definitions are modified.

```
replace ==CDR-LIFE-BASE-VALUES-BOX== by
 ==BaseValues redefines CDR-LIFE-BASE-VALUES-BOX==
 ==CDR-LIFE-BASE-VAL-DATE== by ==BaseValueDate==
 ==CDR-LIFE-BASE-VALUE-LINE== by ==BaseValueLine==
 ==20 CDR-LIFE-BASE-DESC.== by ====
 ==CDR-LIFE-BASE-DESC1== by ==Description==
 ==CDR-LIFE-BASE-LIT== by ====
 ==CDR-LIFE-BASE-DTE== by ==BaseDate==
 ==20 CDR-LIFE-BASE-PRICE.== by ====
 ==25 CDR-LIFE-BP-SPACE PIC X(02).== by ====
 ==25 CDR-LIFE-BP-DASH PIC X(02).== by ====
 ==25 CDR-LIFE-BP-SPACE1 PIC X(02).== by ====
 ==CDR-LIFE-BASE-PRICE-ED== by ==BasePrice==
 ==REDEFINES CDR-LIFE-BASE-PRICE PIC $$$$.== by
 ==pic x(6) justified.==
 ==20 CDR-LIFE-BASE-QTY.
 25 CDR-LIFE-QTY-SPACE PIC X(08).
 25 CDR-LIFE-QTY-DASH PIC X(02).
 25 CDR-LIFE-QTY-SPACE1 PIC X(02).
 25 FILLER PIC X(02).== by ====
 ==CDR-LIFE-BASE-QTY-ED== by ==BaseQuantity==
 ==REDEFINES CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.== by
 ==pic x(14) justified.==
 ==CDR-LIFE-BASE-VALUE-ED== by ==BaseValue==
 ==20 CDR-LIFE-BASE-VALUE PIC X(15).== by ====
 ==REDEFINES CDR-LIFE-BASE-VALUE PIC $(4),$$,$$9.99.==
 by ==pic x(15) justified.==
 ==CDR-LIFE-BASE-TOT-VALUE-LINE. 20== by ====
 ==CDR-LIFE-BASE-TOT-VALUE== by ==TotalValue==.
```

The result of this REPLACE statement followed by a second instance of the original set of definitions is similar to the suggested redefinition of group item BaseValues shown above. This REPLACE statement illustrates a variety of techniques for eliminating unwanted definitions and for modifying the definitions that should be retained. Use whichever technique is appropriate for your situation.

#### RELATED REFERENCES

Operation of XML GENERATE (*COBOL for Windows Language Reference*)

REPLACE statement (*COBOL for Windows Language Reference*)

## Example: converting hyphens in element names to underscores

When you generate an XML document from a data structure whose items have data-names that contain hyphens, the generated XML has element names that contain hyphens. This example shows a way to convert the hyphens in the element names without changing hyphens that occur in the element values.

```
1 Customer-Record.
2 Customer-Number pic 9(9).
2 First-Name pic x(10).
2 Last-Name pic x(20).
```

When the data structure above 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 375), the result is as follows:

```

<Customer-Record>
 <Customer-Number>12345</Customer-Number>
 <First-Name>John</First-Name>
 <Last-Name>Smith-Jones</Last-Name>
</Customer-Record>

```

The element names contain hyphens, and the content of the element Last-Name also contains a hyphen.

Assuming that this XML document is the content of data item `xmlDoc`, and that `charcnt` is set to the length of this XML document, you can change all the hyphens in the element names to underscores but leave the element values unchanged by using the following code:

```

1 xmlDoc pic x(16384).
1 charcnt comp-5 pic 9(5).
1 pos comp-5 pic 9(5).
1 tagstate comp-5 pic 9 value zero.
. . .
dash-to-underscore.
 perform varying pos from 1 by 1
 until pos > charcnt
 if xmlDoc(pos:1) = '<'
 move 1 to tagstate
 end-if
 if tagstate = 1 and xmlDoc(pos:1) = '-'
 move '_' to xmlDoc(pos:1)
 else
 if xmlDoc(pos:1) = '>'
 move 0 to tagstate
 end-if
 end-if
 end-perform.

```

The revised XML document in data item `xmlDoc` has underscores instead of hyphens in the element names, as shown below:

```

<Customer_Record>
 <Customer_Number>12345</Customer_Number>
 <First_Name>John</First_Name>
 <Last_Name>Smith-Jones</Last_Name>
</Customer_Record>

```

## 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 XML output.

**Table 49. Encoding of generated XML output**

If you define the receiving XML identifier as:	The generated XML output is encoded in:
Native alphanumeric	The code page indicated by the locale in effect
Host alphanumeric	The EBCDIC code page in effect <sup>2</sup>
National	UTF-16 little-endian (UTF-16LE, CCSID 1202) <sup>1</sup>
<ol style="list-style-type: none"> <li>1. A <i>byte order mark</i> is not generated.</li> <li>2. You can set the 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 locale.</li> </ol>	

For details about how data items are converted to XML and how the XML element 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 179  
“Setting environment variables” on page 193

#### RELATED REFERENCES

“CHAR” on page 230  
Operation of XML GENERATE (*COBOL for Windows Language Reference*)

---

## Handling errors in generating XML output

When an error is detected during generation of XML output, an exception condition exists. You can write code to check the special register XML-CODE, 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 special register XML-CODE 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 positions that are filled during XML generation. If you define the XML output as national, the count is in national character positions (UTF-16 character encoding units); otherwise the count is in bytes.
2. Use the count field with reference modification to refer to the substring of the receiving data item that contains the generated XML output.  
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 634



---

## Part 6. Developing object-oriented programs

<b>Chapter 24. Writing object-oriented programs</b>	387	Defining a factory method . . . . .	420
Example: accounts. . . . .	388	Hiding a factory or static method . . . . .	421
Subclasses . . . . .	389	Invoking factory or static methods . . . . .	421
Defining a class . . . . .	390	Example: defining a factory (with methods) . . . . .	422
CLASS-ID paragraph for defining a class . . . . .	392	Account class . . . . .	423
REPOSITORY paragraph for defining a class . . . . .	392	CheckingAccount class (subclass of Account) . . . . .	424
Example: external class-names and Java . . . . .		Check class . . . . .	426
packages . . . . .	393	TestAccounts client program . . . . .	426
WORKING-STORAGE SECTION for defining . . . . .		Output produced by the TestAccounts client . . . . .	427
class instance data. . . . .	394	program . . . . .	427
Example: defining a class . . . . .	395	Wrapping procedure-oriented COBOL programs . . . . .	427
Defining a class instance method. . . . .	395	Structuring OO applications . . . . .	428
METHOD-ID paragraph for defining a class . . . . .		Examples: COBOL applications that you can run . . . . .	
instance method . . . . .	396	using the java command . . . . .	428
INPUT-OUTPUT SECTION for defining a class . . . . .		Displaying a message . . . . .	428
instance method . . . . .	397	Echoing the input strings . . . . .	429
DATA DIVISION for defining a class instance . . . . .			
method . . . . .	397	<b>Chapter 25. Communicating with Java methods</b>	431
PROCEDURE DIVISION for defining a class . . . . .		Accessing JNI services . . . . .	431
instance method . . . . .	398	Handling Java exceptions . . . . .	432
Overriding an instance method . . . . .	399	Example: handling Java exceptions . . . . .	433
Overloading an instance method . . . . .	400	Managing local and global references . . . . .	434
Coding attribute (get and set) methods. . . . .	401	Deleting, saving, and freeing local references . . . . .	434
Example: coding a get method . . . . .	401	Java access controls . . . . .	435
Example: defining a method . . . . .	402	Sharing data with Java . . . . .	435
Account class . . . . .	402	Coding interoperable data types in COBOL and . . . . .	
Check class . . . . .	403	Java . . . . .	436
Defining a client . . . . .	403	Declaring arrays and strings for Java . . . . .	437
REPOSITORY paragraph for defining a client . . . . .	404	Manipulating Java arrays . . . . .	438
DATA DIVISION for defining a client . . . . .	405	Example: processing a Java int array . . . . .	440
Choosing LOCAL-STORAGE or . . . . .		Manipulating Java strings . . . . .	441
WORKING-STORAGE . . . . .	406		
Comparing and setting object references . . . . .	406		
Invoking methods (INVOKE) . . . . .	407		
USING phrase for passing arguments . . . . .	408		
Example: passing conforming object-reference . . . . .			
arguments from a COBOL client . . . . .	409		
RETURNING phrase for obtaining a returned . . . . .			
value . . . . .	411		
Invoking overridden superclass methods . . . . .	411		
Creating and initializing instances of classes . . . . .	412		
Instantiating Java classes . . . . .	412		
Instantiating COBOL classes . . . . .	413		
Freeing instances of classes. . . . .	413		
Example: defining a client . . . . .	414		
Defining a subclass . . . . .	414		
CLASS-ID paragraph for defining a subclass . . . . .	415		
REPOSITORY paragraph for defining a subclass . . . . .	416		
WORKING-STORAGE SECTION for defining . . . . .			
subclass instance data . . . . .	416		
Defining a subclass instance method . . . . .	417		
Example: defining a subclass (with methods) . . . . .	417		
CheckingAccount class (subclass of Account) . . . . .	417		
Defining a factory section . . . . .	418		
WORKING-STORAGE SECTION for defining . . . . .			
factory data . . . . .	419		



---

## Chapter 24. Writing object-oriented programs

When you write an object-oriented (OO) program, you have to determine what classes you need and the methods and data that the classes need to do their work.

OO programs are based on *objects* (entities that encapsulate state and behavior) and their classes, methods, and data. A *class* is a template that defines the state and the capabilities of an object. Usually a program creates and works with multiple *object instances* (or simply, *instances*) of a class, that is, multiple objects that are members of that class. The state of each instance is stored in data known as *instance data*, and the capabilities of each instance are called *instance methods*. A class can define data that is shared by all instances of the class, known as *factory* or *static* data, and methods that are supported independently of any object instance, known as *factory* or *static* methods.

Using COBOL for Windows, you can:

- Define classes, with methods and data implemented in COBOL.
- Create instances of Java and COBOL classes.
- Invoke methods on Java and COBOL objects.
- Write classes that inherit from Java classes or other COBOL classes.
- Define and invoke overloaded methods.

In COBOL for Windows programs, you can call the services provided by the Java Native Interface (JNI) to obtain Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language.

In COBOL for Windows classes, you can code CALL statements to interface with procedural COBOL programs. Thus COBOL class definition syntax can be especially useful for writing *wrapper* classes for procedural COBOL logic, enabling existing COBOL code to be accessed from Java.

Java code can create instances of COBOL classes, invoke methods of these classes, and can extend COBOL classes.

### Restrictions:

- COBOL class definitions and methods cannot contain EXEC SQL statements and cannot be compiled using the SQL compiler option.
- COBOL class definitions and methods cannot contain EXEC CICS statements, and cannot be run in a CICS environment. They cannot be compiled using the CICS compiler option.

“Example: accounts” on page 388

### RELATED TASKS

“Defining a class” on page 390

“Defining a class instance method” on page 395

“Defining a client” on page 403

“Defining a subclass” on page 414

“Defining a factory section” on page 418

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

## Example: accounts

Consider the example of a bank in which customers can open accounts and make deposits to and withdrawals from their accounts. You could represent an account by a general-purpose class, called `Account`. Because there are many customers, multiple instances of the `Account` class could exist simultaneously.

After you determine the classes that you need, the next step is to determine the methods that the classes need to do their work. An `Account` class must provide the following services:

- Open the account.
- Get the current balance.
- Deposit to the account.
- Withdraw from the account.
- Report account status.

The following methods for an `Account` class meet those needs:

**init**     Open an account and assign it an account number.

**getBalance**

Return the current balance of the account.

**credit**   Deposit a given sum to the account.

**debit**    Withdraw a given sum from the account.

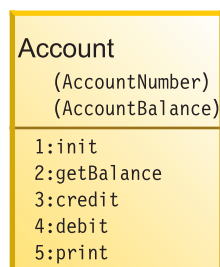
**print**    Display account number and account balance.

As you design an `Account` class and its methods, you discover the need for the class to keep some instance data. Typically, an `Account` object needs the following instance data:

- Account number
- Account balance
- Customer information: name, address, home phone, work phone, social security number, and so forth

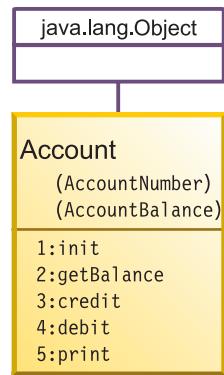
To keep the example simple, however, it is assumed that the account number and account balance are the only instance data that the `Account` class needs.

Diagrams are helpful when you design classes and methods. The following diagram depicts a first attempt at a design of the `Account` class:



The words in parentheses in the diagrams are the names of the instance data, and the words that follow a number and colon are the names of the instance methods.

The structure below shows how the classes relate to each other, and is known as the *inheritance hierarchy*. The Account class inherits directly from the class `java.lang.Object`.



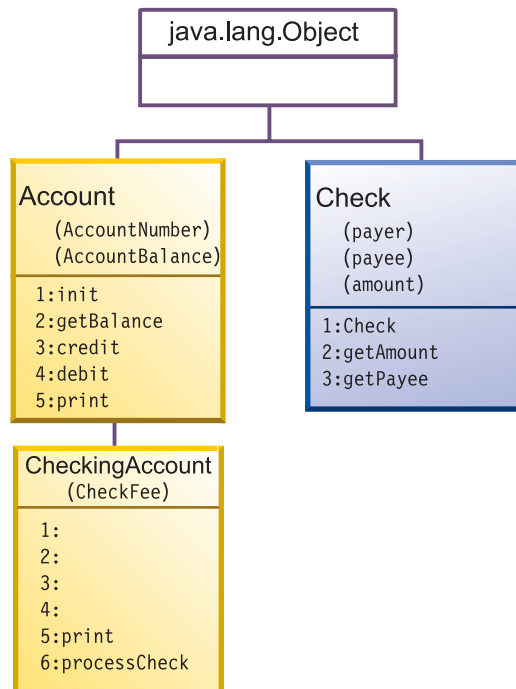
## Subclasses

In the account example, `Account` is a general-purpose class. However, a bank could have many different types of accounts: checking accounts, savings accounts, mortgage loans, and so forth, all of which have all the general characteristics of accounts but could have additional characteristics not shared by all types of accounts.

For example, a `CheckingAccount` class could have, in addition to the account number and account balance that all accounts have, a check fee that applies to each check written on the account. A `CheckingAccount` class also needs a method to process checks (that is, to read the amount, debit the payer, credit the payee, and so forth). So it makes sense to define `CheckingAccount` as a subclass of `Account`, and to define in the subclass the additional instance data and instance methods that the subclass needs.

As you design the `CheckingAccount` class, you discover the need for a class that models checks. An instance of class `Check` needs, at a minimum, instance data for payer, payee, and the check amount.

Many additional classes (and database and transaction-processing logic) would need to be designed in a real-world OO account system, but have been omitted to keep the example simple. The updated inheritance diagram is shown below.



A number and colon with no method-name following them indicate that the method with that number is inherited from the superclass.

**Multiple inheritance:** You cannot use *multiple inheritance* in OO COBOL applications. All classes that you define must have exactly one parent, and `java.lang.Object` must be at the root of every inheritance hierarchy. The class structure of any object-oriented system defined in an OO COBOL application is thus a tree.

“Example: defining a method” on page 402

#### RELATED TASKS

“Defining a class”

“Defining a class instance method” on page 395

“Defining a subclass” on page 414

## Defining a class

A COBOL class definition consists of an IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, followed by an optional factory definition and optional object definition, followed by an END CLASS marker.

Table 50. Structure of class definitions

Section	Purpose	Syntax
IDENTIFICATION DIVISION (required)	Name the class. Provide inheritance information for it.	“CLASS-ID paragraph for defining a class” on page 392 (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)

Table 50. Structure of class definitions (continued)

Section	Purpose	Syntax
ENVIRONMENT DIVISION (required)	Describe the computing environment. Relate class-names used within the class definition to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) “REPOSITORY paragraph for defining a class” on page 392 (required) SOURCE-COMPUTER paragraph (optional) OBJECT-COMPUTER paragraph (optional) SPECIAL-NAMES paragraph (optional)
Factory definition (optional)	Define data to be shared by all instances of the class, and methods supported independently of any object instance.	IDENTIFICATION DIVISION. FACTORY. DATA DIVISION. WORKING-STORAGE SECTION. * (Factory data here) PROCEDURE DIVISION. * (Factory methods here) END FACTORY.
Object definition (optional)	Define instance data and instance methods.	IDENTIFICATION DIVISION. OBJECT. DATA DIVISION. WORKING-STORAGE SECTION. * (Instance data here) PROCEDURE DIVISION. * (Instance methods here) END OBJECT.

If you specify the SOURCE-COMPUTER, OBJECT-COMPUTER, or SPECIAL-NAMES paragraphs in a class CONFIGURATION SECTION, they apply to the entire class definition including all methods that the class introduces.

A class CONFIGURATION SECTION can consist of the same entries as a program CONFIGURATION SECTION, except that a class CONFIGURATION SECTION cannot contain an INPUT-OUTPUT SECTION. You define an INPUT-OUTPUT SECTION only in the individual methods that require it rather than defining it at the class level.

As shown above, you define instance data and methods in the DATA DIVISION and PROCEDURE DIVISION, respectively, within the OBJECT paragraph of the class definition. In classes that require data and methods that are to be associated with the class itself rather than with individual object instances, define a separate DATA DIVISION and PROCEDURE DIVISION within the FACTORY paragraph of the class definition.

Each COBOL class definition must be in a separate source file.

“Example: defining a class” on page 395

#### RELATED TASKS

“WORKING-STORAGE SECTION for defining class instance data” on page 394

“Defining a class instance method” on page 395

“Defining a subclass” on page 414

“Defining a factory section” on page 418

“Describing the computing environment” on page 7

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

#### RELATED REFERENCES

COBOL class definition structure (*COBOL for Windows Language Reference*)

## CLASS-ID paragraph for defining a class

Use the CLASS-ID paragraph in the IDENTIFICATION DIVISION to name a class and provide inheritance information for it.

Identification Division.	<b>Required</b>
Class-id. Account inherits Base.	<b>Required</b>

Use the CLASS-ID paragraph to identify these classes:

- The class that you are defining (Account in the example above).
- The immediate superclass from which the class that you are defining inherits its characteristics. The superclass can be implemented in Java or COBOL.

In the example above, inherits Base indicates that the Account class inherits methods and data from the class known within the class definition as Base. It is recommended that you use the name Base in your OO COBOL programs to refer to java.lang.Object.

A class-name must use single-byte characters and must conform to the normal rules of formation for a COBOL user-defined word.

Use the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION to associate the superclass name (Base in the example) with the name of the superclass as it is known externally (java.lang.Object for Base). You can optionally also specify the name of the class that you are defining (Account in the example) in the REPOSITORY paragraph and associate it with its corresponding external class-name.

You must derive all classes directly or indirectly from the java.lang.Object class.

### RELATED TASKS

"REPOSITORY paragraph for defining a class"

### RELATED REFERENCES

CLASS-ID paragraph (*COBOL for Windows Language Reference*)

User-defined words (*COBOL for Windows Language Reference*)

## REPOSITORY paragraph for defining a class

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them within a class definition, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive and must conform to Java rules of formation. For example, in the Account class definition you might code this:

Environment Division.	<b>Required</b>
Configuration Section.	<b>Required</b>
Repository.	<b>Required</b>
Class Base is "java.lang.Object"	<b>Required</b>
Class Account is "Account".	<b>Optional</b>

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Base and Account within the class definition are java.lang.Object and Account, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the class definition. For example:



- Base
- A superclass from which the class that you are defining inherits
- The classes that you reference in methods within the class definition

In a REPOSITORY paragraph entry, you must specify the external class-name if the name contains non-COBOL characters. You must also specify the external class-name for any referenced class that is part of a Java *package*. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (.), followed by the simple name of the Java class. For example, the Object class is part of the java.lang package, so specify its external name as java.lang.Object as shown above.

An external class-name that you specify in the REPOSITORY paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a REPOSITORY paragraph entry, the external class-name is formed from the class-name in the following manner:

- The class-name is converted to uppercase.
- Each hyphen is changed to zero.
- The first character, if a digit, is changed:
  - 1-9 are changed to A-I.
  - 0 is changed to J.

In the example above, class Account is known externally as Account (in mixed case) because the external name is spelled using mixed case.

You can optionally include in the REPOSITORY paragraph an entry for the class that you are defining (Account in this example). You must include an entry for the class that you are defining if the external class-name contains non-COBOL characters, or to specify a fully package-qualified class-name if the class is to be part of a Java package.

“Example: external class-names and Java packages”

#### RELATED TASKS

“Declaring arrays and strings for Java” on page 437

#### RELATED REFERENCES

REPOSITORY paragraph (*COBOL for Windows Language Reference*)

Identifiers (*The Java Language Specification*)

Packages (*The Java Language Specification*)

### Example: external class-names and Java packages

The following example shows how external class-names are determined from entries in a REPOSITORY paragraph.

Environment division.

Configuration section.

Repository.

Class Employee is "com.acme.Employee"

Class JavaException is "java.lang.Exception"

Class Orders.

The local class-names (the class-names as used within the class definition), the Java packages that contain the classes, and the associated external class-names are as shown in the table below.

Local class-name	Java package	External class-name
Employee	com.acme	com.acme.Employee
JavaException	java.lang	java.lang.Exception
Orders	(unnamed)	ORDERS

The external class-name (the name after the class-name and optional IS in the REPOSITORY paragraph entry) is composed of the fully qualified name of the package (if any) followed by a period, followed by the simple name of the class.

#### RELATED TASKS

“REPOSITORY paragraph for defining a class” on page 392

#### RELATED REFERENCES

REPOSITORY paragraph (*COBOL for Windows Language Reference*)

## WORKING-STORAGE SECTION for defining class instance data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the OBJECT paragraph to describe the *instance data* that a COBOL class needs, that is, the data to be allocated for each instance of the class.

The OBJECT keyword, which you must immediately precede with an IDENTIFICATION DIVISION declaration, indicates the beginning of the definitions of the instance data and instance methods for the class. For example, the definition of the instance data for the Account class might look like this:

```
Identification division.
Object.
 Data division.
 Working-storage section.
 01 AccountNumber pic 9(6).
 01 AccountBalance pic S9(9) value zero.
 . . .
End Object.
```

The instance data is allocated when an object instance is created, and exists until garbage collection of the instance by the Java run time.

You can initialize simple instance data by using VALUE clauses as shown above. You can initialize more complex instance data by coding customized methods to create and initialize instances of classes.

COBOL instance data is equivalent to Java private nonstatic member data. No other class or subclass (nor factory method in the same class, if any) can reference COBOL instance data directly. Instance data is global to all instance methods that the OBJECT paragraph defines. If you want to make instance data accessible from outside the OBJECT paragraph, define attribute (get or set) instance methods for doing so.

The syntax of the WORKING-STORAGE SECTION for instance data declaration is generally the same as in a program, with these exceptions:

- You cannot use the EXTERNAL attribute.
- You can use the GLOBAL attribute, but it has no effect.

#### RELATED TASKS

“Creating and initializing instances of classes” on page 412

“Freeing instances of classes” on page 413

“Defining a factory method” on page 420

“Coding attribute (get and set) methods” on page 401

## Example: defining a class

The following example shows a first attempt at the definition of the Account class, excluding method definitions.

```

cbl thread,pgmname(longmixed)
Identification Division.
Class-id. Account inherits Base.
Environment Division.
Configuration section.
Repository.
 Class Base is "java.lang.Object"
 Class Account is "Account".
*
Identification division.
Object.
Data division.
Working-storage section.
01 AccountNumber pic 9(6).
01 AccountBalance pic S9(9) value zero.
*
Procedure Division.
*
* (Instance method definitions here)
*
End Object.
*
End class Account.
```

#### RELATED TASKS

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

“Defining a client” on page 403

## Defining a class instance method

Define COBOL *instance methods* in the PROCEDURE DIVISION of the OBJECT paragraph of a class definition. An instance method defines an operation that is supported for each object instance of a class.

A COBOL instance method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker.

**Table 51. Structure of instance method definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a method.	“METHOD-ID paragraph for defining a class instance method” on page 396 (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)

Table 51. Structure of instance method definitions (continued)

Division	Purpose	Syntax
ENVIRONMENT (optional)	Relate the file-names used in a method to the corresponding file-names known to the operating system.	"INPUT-OUTPUT SECTION for defining a class instance method" on page 397 (optional)
DATA (optional)	Define external files. Allocate a copy of the data.	"DATA DIVISION for defining a class instance method" on page 397 (optional)
PROCEDURE (optional)	Code the executable statements to complete the service provided by the method.	"PROCEDURE DIVISION for defining a class instance method" on page 398 (optional)

**Definition:** The *signature* of a method consists of the name of the method and the number and type of its formal parameters. (You define the formal parameters of a COBOL method in the USING phrase of the method's PROCEDURE DIVISION header.)

Within a class definition, you do not need to make each method-name unique, but you do need to give each method a unique signature. (You *overload* methods by giving them the same name but a different signature.)

COBOL instance methods are equivalent to Java public nonstatic methods.

"Example: defining a method" on page 402

#### RELATED TASKS

"PROCEDURE DIVISION for defining a class instance method" on page 398

"Overloading an instance method" on page 400

"Overriding an instance method" on page 399

"Invoking methods (INVOKE)" on page 407

"Defining a subclass instance method" on page 417

"Defining a factory method" on page 420

## METHOD-ID paragraph for defining a class instance method

Use the METHOD-ID paragraph to name an instance method. Immediately precede the METHOD-ID paragraph with an IDENTIFICATION DIVISION declaration to indicate the beginning of the method definition.

For example, the definition of the credit method in the Account class begins like this:

```
Identification Division.
Method-id. "credit".
```

Code the method-name as an alphanumeric or national literal. The method-name is processed in a case-sensitive manner and must conform to the rules of formation for a Java method-name.

Other Java or COBOL methods or programs (that is, clients) use the method-name to invoke a method.

#### RELATED TASKS

“Invoking methods (INVOKE)” on page 407

“Using national data (Unicode) in COBOL” on page 160

#### RELATED REFERENCES

Meaning of method names (*The Java Language Specification*)

Identifiers (*The Java Language Specification*)

METHOD-ID paragraph (*COBOL for Windows Language Reference*)

## INPUT-OUTPUT SECTION for defining a class instance method

The ENVIRONMENT DIVISION of an instance method can have only one section, the INPUT-OUTPUT SECTION. This section relates the file-names used in a method definition to the corresponding file-names as they are known to the operating system.

For example, if the Account class defined a method that read information from a file, the Account class might have an INPUT-OUTPUT SECTION that is coded like this:

```
Environment Division.
Input-Output Section.
File-Control.
 Select account-file Assign AcctFile.
```

The syntax for the INPUT-OUTPUT SECTION of a method is the same as the syntax for the INPUT-OUTPUT SECTION of a program.

#### RELATED TASKS

“Describing the computing environment” on page 7

#### RELATED REFERENCES

INPUT-OUTPUT section (*COBOL for Windows Language Reference*)

## DATA DIVISION for defining a class instance method

The DATA DIVISION of an instance method consists of any of the following four sections: FILE SECTION, LOCAL-STORAGE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION.

### FILE SECTION

The same as a program FILE SECTION, except that a method FILE SECTION can define EXTERNAL files only.

### LOCAL-STORAGE SECTION

A separate copy of the LOCAL-STORAGE data is allocated for each invocation of the method, and is freed on return from the method. The method LOCAL-STORAGE SECTION is similar to a program LOCAL-STORAGE SECTION.

If you specify the VALUE clause on a data item, the item is initialized to that value on each invocation of the method.

### WORKING-STORAGE SECTION

A single copy of the WORKING-STORAGE data is allocated. The data persists in its last-used state until the run unit ends. The same copy of the data is used whenever the method is invoked, regardless of the invoking object or thread. The method WORKING-STORAGE SECTION is similar to a program WORKING-STORAGE SECTION.

If you specify the VALUE clause on a data item, the item is initialized to that value on the first invocation of the method. You can specify the EXTERNAL clause for the data items.

## LINKAGE SECTION

The same as a program LINKAGE SECTION.

If you define a data item with the same name in both the DATA DIVISION of an instance method and the DATA DIVISION of the OBJECT paragraph, a reference in the method to that data-name refers only to the method data item. The method DATA DIVISION takes precedence.

## RELATED TASKS

"Describing the data" on page 11

"Sharing data by using the EXTERNAL clause" on page 478

## RELATED REFERENCES

DATA DIVISION overview (*COBOL for Windows Language Reference*)

## PROCEDURE DIVISION for defining a class instance method

Code the executable statements to implement the service that an instance method provides in the PROCEDURE DIVISION of the instance method.

You can code most COBOL statements in the PROCEDURE DIVISION of a method that you can code in the PROCEDURE DIVISION of a program. You cannot, however, code the following statements in a method:

- ENTRY
- EXIT PROGRAM
- The following obsolete elements of Standard COBOL 85:
  - ALTER
  - GOTO without a specified procedure-name
  - SEGMENT-LIMIT
  - USE FOR DEBUGGING

You can code the EXIT METHOD or GOBACK statement in an instance method to return control to the invoking client. Both statements have the same effect. If you specify the RETURNING phrase upon invocation of the method, the EXIT METHOD or GOBACK statement returns the value of the data item to the invoking client.

An implicit EXIT METHOD is generated as the last statement in the PROCEDURE DIVISION of each method.

You can specify STOP RUN in a method; doing so terminates the entire run unit including all threads executing within it.

You must terminate a method definition with an END METHOD marker. For example, the following statement marks the end of the credit method:

End method "credit".

**USING phrase for obtaining passed arguments:** Specify the formal parameters to a method, if any, in the USING phrase of the method's PROCEDURE DIVISION header. You must specify that the arguments are passed BY VALUE. Define each parameter as a level-01 or level-77 item in the method's LINKAGE SECTION. The data type of each parameter must be one of the types that are interoperable with Java.

**RETURNING phrase for returning a value:** Specify the data item to be returned as the method result, if any, in the RETURNING phrase of the method's PROCEDURE

DIVISION header. Define the data item as a level-01 or level-77 item in the method's LINKAGE SECTION. The data type of the return value must be one of the types that are interoperable with Java.

#### RELATED TASKS

"Coding interoperable data types in COBOL and Java" on page 436

"Overriding an instance method"

"Overloading an instance method" on page 400

"Comparing and setting object references" on page 406

"Invoking methods (INVOKE)" on page 407

Chapter 13, "Compiling, linking, and running OO applications," on page 219

#### RELATED REFERENCES

"THREAD" on page 265

The procedure division header (*COBOL for Windows Language Reference*)

## Overriding an instance method

An instance method that is defined in a subclass is said to *override* an inherited instance method that would otherwise be accessible in the subclass if the two methods have the same signature.

To override a superclass instance method `m1` in a COBOL subclass, define an instance method `m1` in the subclass that has the same name and whose PROCEDURE DIVISION USING phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes `m1` on an instance of the subclass, the subclass method rather than the superclass method is invoked.

For example, the `Account` class defines a method `debit` whose LINKAGE SECTION and PROCEDURE DIVISION header look like this:

```
Linkage section.
01 inDebit pic S9(9) binary.
Procedure Division using by value inDebit.
```

If you define a `CheckingAccount` subclass and want it to have a `debit` method that overrides the `debit` method defined in the `Account` superclass, define the subclass method with exactly one input parameter also specified as `pic S9(9) binary`. If a client invokes `debit` using an object reference to a `CheckingAccount` instance, the `CheckingAccount` `debit` method (rather than the `debit` method in the `Account` superclass) is invoked.

The presence or absence of a method return value and the data type of the return value used in the PROCEDURE DIVISION RETURNING phrase (if any) must be identical in the subclass instance method and the overridden superclass instance method.

An instance method must not override a factory method in a COBOL superclass nor a static method in a Java superclass.

"Example: defining a method" on page 402

#### RELATED TASKS

"PROCEDURE DIVISION for defining a class instance method" on page 398

"Coding interoperable data types in COBOL and Java" on page 436

"Invoking methods (INVOKE)" on page 407



“Invoking overridden superclass methods” on page 411

“Defining a subclass” on page 414

“Hiding a factory or static method” on page 421

#### RELATED REFERENCES

Inheritance, overriding, and hiding (*The Java Language Specification*)

## Overloading an instance method

Two methods that are supported in a class (whether defined in the class or inherited from a superclass) are said to be *overloaded* if they have the same name but different signatures.

You overload methods when you want to enable clients to invoke different versions of a method, for example, to initialize data using different sets of parameters.

To overload a method, define a method whose PROCEDURE DIVISION USING phrase (if any) has a different number or type of formal parameters than an identically named method that is supported in the same class. For example, the Account class defines an instance method `init` that has exactly one formal parameter. The LINKAGE SECTION and PROCEDURE DIVISION header of the `init` method look like this:

Linkage section.

```
01 inAccountNumber pic S9(9) binary.
```

```
Procedure Division using by value inAccountNumber.
```

Clients invoke this method to initialize an Account instance with a given account number (and a default account balance of zero) by passing exactly one argument that matches the data type of `inAccountNumber`.

But the Account class could define, for example, a second instance method `init` that has an additional formal parameter that allows the opening account balance to also be specified. The LINKAGE SECTION and PROCEDURE DIVISION header of this `init` method could look like this:

Linkage section.

```
01 inAccountNumber pic S9(9) binary.
```

```
01 inBalance pic S9(9) binary.
```

```
Procedure Division using by value inAccountNumber
 inBalance.
```

Clients could invoke either `init` method by passing arguments that match the signature of the desired method.

The presence or absence of a method return value does not have to be consistent in overloaded methods, and the data type of the return value given in the PROCEDURE DIVISION RETURNING phrase (if any) does not have to be identical in overloaded methods.

You can overload factory methods in exactly the same way that you overload instance methods.

The rules for overloaded method definition and resolution of overloaded method invocations are based on the corresponding rules for Java.

#### RELATED TASKS

“Invoking methods (INVOKE)” on page 407

“Defining a factory method” on page 420



## Coding attribute (get and set) methods

You can provide access to an instance variable *X* from outside the class in which *X* is defined by coding accessor (get) and mutator (set) methods for *X*.

Instance variables in COBOL are *private*. The class that defines instance variables fully encapsulates them, and only the instance methods defined in the same OBJECT paragraph can access them directly. Normally a well-designed object-oriented application does not need to access instance variables from outside the class.

COBOL does not directly support the concept of a *public* instance variable as defined in Java and other object-oriented languages, nor the concept of a class attribute as defined by CORBA. (A CORBA *attribute* is an instance variable that has an automatically generated get method for accessing the value of the variable, and an automatically generated set method for modifying the value of the variable if the variable is not read-only.)

“Example: coding a get method”

## RELATED TASKS

“WORKING-STORAGE SECTION for defining class instance data” on page 394

“Processing the data” on page 16

### Example: coding a get method

The following example shows the definition in the Account class of an instance method, *getBalance*, to return the value of the instance variable *AccountBalance* to a client. *getBalance* and *AccountBalance* are defined in the OBJECT paragraph of the Account class definition.

```

Identification Division.
Class-id. Account inherits Base.
* (ENVIRONMENT DIVISION not shown)
* (FACTORY paragraph not shown)
*
Identification division.
Object.
Data division.
Working-storage section.
01 AccountBalance pic S9(9) value zero.
* (Other instance data not shown)
*
Procedure Division.
*
Identification Division.
Method-id. "getBalance".
Data division.
Linkage section.
01 outBalance pic S9(9) binary.
*
Procedure Division returning outBalance.
Move AccountBalance to outBalance.
End method "getBalance".
*
* (Other instance methods not shown)
End Object.
*
End class Account.
```

## Example: defining a method

The following example adds to the previous example the instance method definitions of the Account class, and shows the definition of the Java Check class.

(The previous example was “Example: defining a class” on page 395.)

### Account class

```
cb1 thread,pgmname(longmixed)
Identification Division.
Class-id. Account inherits Base.
Environment Division.
Configuration section.
Repository.
 Class Base is "java.lang.Object"
 Class Account is "Account".
*
* (FACTORY paragraph not shown)
*
Identification division.
Object.
Data division.
Working-storage section.
01 AccountNumber pic 9(6).
01 AccountBalance pic S9(9) value zero.
*
Procedure Division.
*
* init method to initialize the account:
Identification Division.
Method-id. "init".
Data division.
Linkage section.
01 inAccountNumber pic S9(9) binary.
Procedure Division using by value inAccountNumber.
 Move inAccountNumber to AccountNumber.
End method "init".
*
* getBalance method to return the account balance:
Identification Division.
Method-id. "getBalance".
Data division.
Linkage section.
01 outBalance pic S9(9) binary.
Procedure Division returning outBalance.
 Move AccountBalance to outBalance.
End method "getBalance".
*
* credit method to deposit to the account:
Identification Division.
Method-id. "credit".
Data division.
Linkage section.
01 inCredit pic S9(9) binary.
Procedure Division using by value inCredit.
 Add inCredit to AccountBalance.
End method "credit".
*
* debit method to withdraw from the account:
Identification Division.
Method-id. "debit".
Data division.
Linkage section.
01 inDebit pic S9(9) binary.
Procedure Division using by value inDebit.
 Subtract inDebit from AccountBalance.
```

```

 End method "debit".
*
* print method to display formatted account number and balance:
Identification Division.
Method-id. "print".
Data division.
Local-storage section.
01 PrintableAccountNumber pic ZZZZZ999999.
01 PrintableAccountBalance pic $$$,$$$,$$9CR.
Procedure Division.
 Move AccountNumber to PrintableAccountNumber
 Move AccountBalance to PrintableAccountBalance
 Display " Account: " PrintableAccountNumber
 Display " Balance: " PrintableAccountBalance.
 End method "print".
*
End Object.
*
End class Account.

```

## Check class

```

/**
 * A Java class for check information
 */
public class Check {
 private CheckingAccount payer;
 private Account payee;
 private int amount;

 public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
 payer=inPayer;
 payee=inPayee;
 amount=inAmount;
 }

 public int getAmount() {
 return amount;
 }

 public Account getPayee() {
 return payee;
 }
}

```

### RELATED TASKS

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

---

## Defining a client

A program or method that requests services from one or more methods in a class is called a *client* of that class.

In a COBOL or Java client, you can:

- Create object instances of Java and COBOL classes.
- Invoke instance methods on Java and COBOL objects.
- Invoke COBOL factory methods and Java static methods.

In a COBOL client, you can also call services provided by the Java Native Interface (JNI).

A COBOL client program consists of the usual four divisions:

Table 52. Structure of COBOL clients

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a client.	Code as usual, except that a client program must be: <ul style="list-style-type: none"> <li>• Thread-enabled (compiled with the THREAD option, and conforming to the coding guidelines for threaded applications)</li> </ul>
ENVIRONMENT (required)	Describe the computing environment. Relate class-names used in the client to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) "REPOSITORY paragraph for defining a client" (required)
DATA (optional)	Describe the data that the client needs.	"DATA DIVISION for defining a client" on page 405 (optional)
PROCEDURE (optional)	Create instances of classes, manipulate object reference data items, and invoke methods.	Code using INVOKE, IF, and SET statements.

"Example: defining a client" on page 414

#### RELATED TASKS

Chapter 13, "Compiling, linking, and running OO applications," on page 219

Chapter 30, "Preparing COBOL programs for multithreading," on page 497

Chapter 25, "Communicating with Java methods," on page 431

"Coding interoperable data types in COBOL and Java" on page 436

"Creating and initializing instances of classes" on page 412

"Comparing and setting object references" on page 406

"Invoking methods (INVOKE)" on page 407

"Invoking factory or static methods" on page 421

#### RELATED REFERENCES

"THREAD" on page 265

## REPOSITORY paragraph for defining a client

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them in a COBOL client, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive, and must conform to Java rules of formation. For example, in a client program that uses the Account and Check classes you might code this:

```

Environment division. Required
Configuration section. Required
 Source-Computer.
 Object-Computer.
Repository. Required
 Class Account is "Account"
 Class Check is "Check".

```

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Account and Check within the client are Account and Check, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the client. In a REPOSITORY paragraph entry, you must specify the external class-name if the name contains non-COBOL characters.

You must specify the external class-name for any referenced class that is part of a Java package. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (.), followed by the simple name of the Java class.

An external class-name that you specify in the REPOSITORY paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a REPOSITORY paragraph entry, the external class-name is formed from the class-name in the same manner as it is when an external class-name is not included in a REPOSITORY paragraph entry in a class definition. In the example above, class Account and class Check are known externally as Account and Check (in mixed case), respectively, because the external names are spelled using mixed case.

The SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES paragraphs of the CONFIGURATION SECTION are optional.

#### RELATED TASKS

“REPOSITORY paragraph for defining a class” on page 392

#### RELATED REFERENCES

REPOSITORY paragraph (*COBOL for Windows Language Reference*)

Identifiers (*The Java Language Specification*)

Packages (*The Java Language Specification*)

## DATA DIVISION for defining a client

You can use any of the sections of the DATA DIVISION to describe the data that the client needs.

Data Division.

Local-storage section.

```
01 anAccount usage object reference Account.
01 aCheckingAccount usage object reference CheckingAccount.
01 aCheck usage object reference Check.
01 payee usage object reference Account.
. . .
```

Because a client references classes, it needs one or more special data items called *object references*, that is, references to instances of those classes. All requests to instance methods require an object reference to an instance of a class in which the method is supported (that is, either defined or available by inheritance). You code object references to refer to instances of Java classes using the same syntax as you use to refer to instances of COBOL classes. In the example above, the phrase usage object reference indicates an object reference data item.

All four object references in the code above are called *typed* object references because a class-name appears after the OBJECT REFERENCE phrase. A typed object

reference can refer only to an instance of the class named in the OBJECT REFERENCE phrase or to one of its subclasses. Thus anAccount can refer to instances of the Account class or one of its subclasses, but cannot refer to instances of any other class. Similarly, aCheck can refer only to instances of the Check class or any subclasses that it might have.

Another type of object reference, not shown above, does not have a class-name after the OBJECT REFERENCE phrase. Such a reference is called a *universal* object reference, which means that it can refer to instances of any class. Avoid coding universal object references, because they are interoperable with Java in only very limited circumstances (when used in the RETURNING phrase of the INVOKE *class-name* NEW . . . statement).

You must define, in the REPOSITORY paragraph of the CONFIGURATION SECTION, class-names that you use in the OBJECT REFERENCE phrase.

#### RELATED TASKS

“Choosing LOCAL-STORAGE or WORKING-STORAGE”

“Coding interoperable data types in COBOL and Java” on page 436

“Invoking methods (INVOKE)” on page 407

“REPOSITORY paragraph for defining a client” on page 404

#### RELATED REFERENCES

RETURNING phrase (*COBOL for Windows Language Reference*)

### Choosing LOCAL-STORAGE or WORKING-STORAGE

You can in general use the WORKING-STORAGE SECTION to define working data that a client program needs. However, if the program could simultaneously run on multiple threads, you might instead want to define the data in the LOCAL-STORAGE SECTION.

Each thread has access to a separate copy of LOCAL-STORAGE data but shares access to a single copy of WORKING-STORAGE data. If you define the data in the WORKING-STORAGE SECTION, you need to synchronize access to the data or ensure that no two threads can access it simultaneously.

#### RELATED TASKS

Chapter 30, “Preparing COBOL programs for multithreading,” on page 497

## Comparing and setting object references

You can compare object references by coding conditional statements or a call to the JNI service IsSameObject, and you can set object references by using the SET statement.

For example, code either IF statement below to check whether the object reference anAccount refers to no object instance:

```
If anAccount = Null . . .
If anAccount = Nulls . . .
```

You can code a call to IsSameObject to check whether two object references, object1 and object2, refer to the same object instance or whether each refers to no object instance. To ensure that the arguments and return value are interoperable with Java and to establish addressability to the callable service, code the following data definitions and statements before the call to IsSameObject:

Local-storage Section.

```
. . .
01 is-same Pic X.
 88 is-same-false Value X'00'.
 88 is-same-true Value X'01' Through X'FF'.
```

Linkage Section.

Copy JNI.

Procedure Division.

```
Set Address Of JNIEnv To JNIEnvPtr
Set Address Of JNINativeInterface To JNIEnv
Call IsSameObject Using By Value JNIEnvPtr object1 object2
 Returning is-same
If is-same-true . . .
```

Within a method you can check whether an object reference refers to the object instance on which the method was invoked by coding a call to `IsSameObject` that compares the object reference and `SELF`.

You can instead invoke the Java `equals` method (inherited from `java.lang.Object`) to determine whether two object references refer to the same object instance.

You can make an object reference refer to no object instance by using the `SET` statement. For example:

```
Set anAccount To Null.
```

You can also make one object reference refer to the same instance as another object reference does by using the `SET` statement. For example:

```
Set anotherAccount To anAccount.
```

This `SET` statement causes `anotherAccount` to refer to the same object instance as `anAccount` does. If the receiver (`anotherAccount`) is a universal object reference, the sender (`anAccount`) can be either a universal or a typed object reference. If the receiver is a typed object reference, the sender must be a typed object reference bound to the same class as the receiver or to one of its subclasses.

Within a method you can make an object reference refer to the object instance on which the method was invoked by setting it to `SELF`. For example:

```
Set anAccount To Self.
```

#### RELATED TASKS

“Coding interoperable data types in COBOL and Java” on page 436

“Accessing JNI services” on page 431

#### RELATED REFERENCES

`IsSameObject` (*The Java Native Interface*)

## Invoking methods (INVOKE)

In a Java client, you can create object instances of classes that were implemented in COBOL and invoke methods on those objects using standard Java syntax. In a COBOL client, you can invoke methods that are defined in Java or COBOL classes by coding the `INVOKE` statement.

```
Invoke Account "createAccount"
 using by value 123456
 returning anAccount
Invoke anAccount "credit" using by value 500.
```

The first example INVOKE statement above uses the class-name Account to invoke a method called createAccount. This method must be either defined or inherited in the Account class, and must be one of the following types:

- A Java static method
- A COBOL factory method

The phrase using by value 123456 indicates that 123456 is an input argument to the method, and is passed by value. The input argument 123456 and the returned data item anAccount must conform to the definition of the formal parameters and return type, respectively, of the (possibly overloaded) createAccount method.

The second INVOKE statement uses the returned object reference anAccount to invoke the instance method credit, which is defined in the Account class. The input argument 500 must conform to the definition of the formal parameters of the (possibly overloaded) credit method.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time matches the method-name in the signature of the target method. The method-name must be an alphanumeric or national literal or a category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

When you code an INVOKE statement using an object reference (as in the second example statement above), the statement begins with one of the following two forms:

```
Invoke objRef "literal-name" . . .
Invoke objRef identifier-name . . .
```

When the method-name is an identifier, you must define the object reference (objRef) as USAGE OBJECT REFERENCE with no specified type, that is, as a universal object reference.

If an invoked method is not supported in the class to which the object reference refers, a severe error condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

You can use the optional scope terminator END-INVOKE with the INVOKE statement.

The INVOKE statement does not set the RETURN-CODE special register.

#### RELATED TASKS

“USING phrase for passing arguments”  
“RETURNING phrase for obtaining a returned value” on page 411  
“PROCEDURE DIVISION for defining a class instance method” on page 398  
“Coding interoperable data types in COBOL and Java” on page 436  
“Invoking overridden superclass methods” on page 411  
“Invoking factory or static methods” on page 421

#### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

### USING phrase for passing arguments

If you pass arguments to a method, specify the arguments in the USING phrase of the INVOKE statement. Code the data type of each argument so that it conforms to the type of the corresponding formal parameter in the intended target method.



Table 53. Conformance of arguments in a COBOL client

Programming language of the target method	Is the argument an object reference?	Then code the DATA DIVISION definition of the argument as:	Restriction
COBOL	No	The same as the definition of the corresponding formal parameter	
Java	No	Interoperable with the corresponding Java parameter	
COBOL or Java	Yes	An object reference that is typed to the same class as the corresponding parameter in the target method	In a COBOL client (unlike in a Java client), the class of an argument cannot be a subclass of the class of the corresponding parameter.

See the example referenced below for a way to make an object-reference argument conform to the type of a corresponding formal parameter by using the SET statement or the REDEFINES clause.

“Example: passing conforming object-reference arguments from a COBOL client”

If the target method is overloaded, the data types of the arguments are used to select from among the methods that have the same name.

You must specify that the arguments are passed BY VALUE. In other words, the arguments are not affected by any change to the corresponding formal parameters in the invoked method.

The data type of each argument must be one of the types that are interoperable with Java.

#### RELATED TASKS

“PROCEDURE DIVISION for defining a class instance method” on page 398

“Overloading an instance method” on page 400

“Coding interoperable data types in COBOL and Java” on page 436

“Passing data” on page 467

#### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

SET statement (*COBOL for Windows Language Reference*)

REDEFINES clause (*COBOL for Windows Language Reference*)

### Example: passing conforming object-reference arguments from a COBOL client

The following example shows a way to make an object-reference argument in a COBOL client conform to the expected class of the corresponding formal parameter in an invoked method.

Class C defines a method M that has one parameter, a reference to an object of class java.lang.Object:

```

. . .
Class-id. C inherits Base.
. . .
Repository.
 Class Base is "java.lang.Object"
 Class JavaObject is "java.lang.Object".
Identification division.
Factory.
. . .
Procedure Division.
Identification Division.
Method-id. "M".
Data division.
Linkage section.
01 obj object reference JavaObject.
Procedure Division using by value obj.
. . .

```

To invoke method M, a COBOL client must pass an argument that is a reference to an object of class `java.lang.Object`. The client below defines a data item `aString`, which cannot be passed as an argument to M because `aString` is a reference to an object of class `java.lang.String`. The client first uses a SET statement to assign `aString` to a data item, `anObj`, that is a reference to an object of class `java.lang.Object`. (This SET statement is legal because `java.lang.String` is a subclass of `java.lang.Object`.) The client then passes `anObj` as the argument to M.

```

. . .
Repository.
 Class jstring is "java.lang.String"
 Class JavaObject is "java.lang.Object".
Data division.
Local-storage section.
01 aString object reference jstring.
01 anObj object reference JavaObject.
*
Procedure division.
. . . (statements here assign a value to aString)
Set anObj to aString
Invoke C "M"
using by value anObj

```

Instead of using a SET statement to obtain `anObj` as a reference to an object of class `java.lang.Object`, the client could define `aString` and `anObj` with the REDEFINES clause as follows:

```

. . .
01 aString object reference jstring.
01 anObj redefines aString object reference JavaObject.

```

After the client assigns a value to data item `aString` (that is, a valid reference to an object of class `java.lang.String`), `anObj` can be passed as the argument to M.

#### RELATED TASKS

“Coding interoperable data types in COBOL and Java” on page 436

“PROCEDURE DIVISION for defining a class instance method” on page 398

#### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

SET statement (*COBOL for Windows Language Reference*)

REDEFINES clause (*COBOL for Windows Language Reference*)

## RETURNING phrase for obtaining a returned value

If a data item is to be returned as the method result, specify the item in the RETURNING phrase of the INVOKE statement. Define the returned item in the DATA DIVISION of the client.

The item that you specify in the RETURNING phrase of the INVOKE statement must conform to the type returned by the target method, as shown in the table below.

Table 54. Conformance of the returned data item in a COBOL client

Programming language of the target method	Is the returned item an object reference?	Then code the DATA DIVISION definition of the returned item as:
COBOL	No	The same as the definition of the RETURNING item in the target method
Java	No	Interoperable with the returned Java data item
COBOL or Java	Yes	An object reference that is typed to the same class as the object reference that is returned by the target method

In all cases, the data type of the returned value must be one of the types that are interoperable with Java.

### RELATED TASKS

“Coding interoperable data types in COBOL and Java” on page 436

### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

## Invoking overridden superclass methods

Sometimes within a class you need to invoke an overridden superclass method instead of invoking a method that has the same signature and is defined in the current class.

For example, suppose that the CheckingAccount class overrides the debit instance method defined in its immediate superclass, Account. You could invoke the Account debit method within a method in the CheckingAccount class by coding this statement:

Invoke Super "debit" Using By Value amount.

You would define amount as PIC S9(9) BINARY to match the signature of the debit methods.

The CheckingAccount class overrides the print method that is defined in the Account class. Because the print method has no formal parameters, a method in the CheckingAccount class could invoke the superclass print method with this statement:

Invoke Super "print".

The keyword SUPER indicates that you want to invoke a superclass method rather than a method in the current class. (SUPER is an implicit reference to the object used in the invocation of the currently executing method.)

“Example: accounts” on page 388

#### RELATED TASKS

“Overriding an instance method” on page 399

#### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

## Creating and initializing instances of classes

Before you can use the instance methods that are defined in a Java or COBOL class, you must first create an instance of the class.

To create a new instance of class *class-name* and to obtain a reference *object-reference* to the created object, code a statement of the following form, where *object-reference* is defined in the DATA DIVISION of the client:

```
INVOKE class-name NEW . . . RETURNING object-reference
```

When you code the INVOKE . . . NEW statement within a method, and the use of the returned object reference is not limited to the duration of the method invocation, you must convert the returned object reference to a global reference by calling the JNI service `NewGlobalRef`:

```
Call NewGlobalRef using by value JNIEnvPtr object-reference
returning object-reference
```

If you do not call `NewGlobalRef`, the returned object reference is only a local reference, which means that it is automatically freed after the method returns.

#### RELATED TASKS

“Instantiating Java classes”

“Instantiating COBOL classes” on page 413

“Accessing JNI services” on page 431

“Managing local and global references” on page 434

“DATA DIVISION for defining a client” on page 405

“Invoking methods (INVOKE)” on page 407

“Coding interoperable data types in COBOL and Java” on page 436

#### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

## Instantiating Java classes

To instantiate a Java class, invoke any parameterized constructor that the class supports by coding the USING phrase in the INVOKE . . . NEW statement immediately before the RETURNING phrase, passing BY VALUE the number and types of arguments that match the signature of the constructor.

The data type of each argument must be one of the types that are interoperable with Java. To invoke the default (parameterless) constructor, omit the USING phrase.

For example, to create an instance of the `Check` class, initialize its instance data, and obtain reference `aCheck` to the `Check` instance created, you could code this statement in a COBOL client:

```
Invoke Check New
 using by value aCheckingAccount, payee, 125
 returning aCheck
```

#### RELATED TASKS

“Invoking methods (INVOKE)” on page 407

“Coding interoperable data types in COBOL and Java” on page 436

#### RELATED REFERENCES

VALUE clause (*COBOL for Windows Language Reference*)

INVOKE statement (*COBOL for Windows Language Reference*)

### Instantiating COBOL classes

To instantiate a COBOL class, you can specify either a typed or universal object reference in the RETURNING phrase of the INVOKE . . . NEW statement. However, you cannot code the USING phrase: the instance data is initialized as specified in the VALUE clauses in the class definition.

Thus the INVOKE . . . NEW statement is useful for instantiating COBOL classes that have only simple instance data. For example, the following statement creates an instance of the Account class, initializes the instance data as specified in VALUE clauses in the WORKING-STORAGE SECTION of the OBJECT paragraph of the Account class definition, and provides reference outAccount to the new instance:

```
Invoke Account New returning outAccount
```

To make it possible to initialize COBOL instance data that cannot be initialized using VALUE clauses alone, when designing a COBOL class you must define a parameterized creation method in the FACTORY paragraph and a parameterized initialization method in the OBJECT paragraph:

1. In the parameterized factory creation method, do these steps:
  - a. Code INVOKE *class-name* NEW RETURNING *objectRef* to create an instance of *class-name* and to give initial values to the instance data items that have VALUE clauses.
  - b. Invoke the parameterized initialization method on the instance (*objectRef*), passing BY VALUE the arguments that were supplied to the factory method.
2. In the initialization method, code logic to complete the instance data initialization using the values supplied through the formal parameters.

To create an instance of the COBOL class and properly initialize it, the client invokes the parameterized factory method, passing BY VALUE the desired arguments. The object reference returned to the client is a local reference. If the client code is within a method, and the use of the returned object reference is not limited to the duration of that method, the client code must convert the returned object reference to a global reference by calling the JNI service NewGlobalRef.

“Example: defining a factory (with methods)” on page 422

#### RELATED TASKS

“Accessing JNI services” on page 431

“Managing local and global references” on page 434

“Invoking methods (INVOKE)” on page 407

“Defining a factory section” on page 418

#### RELATED REFERENCES

VALUE clause (*COBOL for Windows Language Reference*)

INVOKE statement (*COBOL for Windows Language Reference*)

### Freeing instances of classes

You do not need to take any action to free individual object instances of any class. No syntax is available for doing so. The Java runtime system automatically performs *garbage collection*, that is, it reclaims the memory for objects that are no longer in use.

There could be times, however, when you need to explicitly free local or global references to objects within a native COBOL client in order to permit garbage collection of the referenced objects to occur.

#### RELATED TASKS

“Managing local and global references” on page 434

## Example: defining a client

The following example shows a small client program of the Account class.

The program does this:

- Invokes a factory method `createAccount` to create an Account instance with a default balance of zero
- Invokes the instance method `credit` to deposit \$500 to the new account
- Invokes the instance method `print` to display the account status

(The Account class was shown in “Example: defining a method” on page 402.)

```
cb1 thread,pgmname(longmixed)
Identification division.
Program-id. "TestAccounts" recursive.
Environment division.
Configuration section.
Repository.
 Class Account is "Account".
Data Division.
* Working data is declared in LOCAL-STORAGE instead of
* WORKING-STORAGE so that each thread has its own copy:
Local-storage section.
01 anAccount usage object reference Account.
*
Procedure division.
Test-Account-section.
 Display "Test Account class"
* Create account 123456 with 0 balance:
 Invoke Account "createAccount"
 using by value 123456
 returning anAccount
* Deposit 500 to the account:
 Invoke anAccount "credit" using by value 500
 Invoke anAccount "print"
 Display space
*
 Stop Run.
End program "TestAccounts".
```

“Example: defining a factory (with methods)” on page 422

#### RELATED TASKS

“Defining a factory method” on page 420

“Invoking factory or static methods” on page 421

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

---

## Defining a subclass

You can make a class (called a *subclass*, derived class, or child class) a specialization of another class (called a *superclass*, base class, or parent class).

A subclass inherits the methods and instance data of its superclasses, and is related to its superclasses by an *is-a* relationship. For example, if subclass P inherits from

superclass Q, and subclass Q inherits from superclass S, then an instance of P is an instance of Q and also (by transitivity) an instance of S. An instance of P inherits the methods and data of Q and S.

Using subclasses has several advantages:

- Reuse of code: Through inheritance, a subclass can reuse methods that already exist in a superclass.
- Specialization: In a subclass you can add new methods to handle cases that the superclass does not handle. You can also add new data items that the superclass does not need.
- Change in action: A subclass can override a method that it inherits from a superclass by defining a method of the same signature as that in the superclass. When you override a method, you might make only a few minor changes or completely change what the method does.

**Restriction:** You cannot use *multiple inheritance* in your COBOL programs. Each COBOL class that you define must have exactly one immediate superclass that is implemented in Java or COBOL, and each class must be derived directly or indirectly from `java.lang.Object`. The semantics of inheritance are as defined by Java.

The structure and syntax of a subclass definition are identical to those of a class definition: Define instance data and methods in the DATA DIVISION and PROCEDURE DIVISION, respectively, within the OBJECT paragraph of the subclass definition. In subclasses that require data and methods that are to be associated with the subclass itself rather than with individual object instances, define a separate DATA DIVISION and PROCEDURE DIVISION within the FACTORY paragraph of the subclass definition.

COBOL instance data is private. A subclass can access the instance data of a COBOL superclass only if the superclass defines attribute (get or set) instance methods for doing so.

“Example: accounts” on page 388

“Example: defining a subclass (with methods)” on page 417

#### RELATED TASKS

“Defining a class” on page 390

“Overriding an instance method” on page 399

“Coding attribute (get and set) methods” on page 401

“Defining a subclass instance method” on page 417

“Defining a factory section” on page 418

#### RELATED REFERENCES

Inheritance, overriding, and hiding (*The Java Language Specification*)

COBOL class definition structure (*COBOL for Windows Language Reference*)

## CLASS-ID paragraph for defining a subclass

Use the CLASS-ID paragraph to name the subclass and indicate from which immediate Java or COBOL superclass it inherits its characteristics.

Identification Division.

**Required**

Class-id. CheckingAccount inherits Account. **Required**

In the example above, CheckingAccount is the subclass being defined.

CheckingAccount inherits all the methods of the class known within the subclass



definition as Account. CheckingAccount methods can access Account instance data only if the Account class provides attribute (get or set) methods for doing so.

You must specify the name of the immediate superclass in the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION. You can optionally associate the superclass name with the name of the class as it is known externally. You can also specify the name of the subclass that you are defining (here, CheckingAccount) in the REPOSITORY paragraph and associate it with its corresponding external class-name.

**RELATED TASKS**

“CLASS-ID paragraph for defining a class” on page 392

“Coding attribute (get and set) methods” on page 401

“REPOSITORY paragraph for defining a subclass”

**REPOSITORY paragraph for defining a subclass**

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them within a subclass definition, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

For example, in the CheckingAccount subclass definition, these REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as CheckingAccount, Check, and Account within the subclass definition are CheckingAccount, Check, and Account, respectively.

Environment Division.		<b>Required</b>
Configuration Section.		<b>Required</b>
Repository.		<b>Required</b>
Class CheckingAccount	is "CheckingAccount"	<b>Optional</b>
Class Check	is "Check"	<b>Required</b>
Class Account	is "Account".	<b>Required</b>

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the subclass definition. For example:

- A user-defined superclass from which the subclass that you are defining inherits
- The classes that you reference in methods within the subclass definition

The rules for coding REPOSITORY paragraph entries in a subclass are identical to those for coding REPOSITORY paragraph entries in a class.

**RELATED TASKS**

“REPOSITORY paragraph for defining a class” on page 392

**RELATED REFERENCES**

REPOSITORY paragraph (*COBOL for Windows Language Reference*)

**WORKING-STORAGE SECTION for defining subclass instance data**

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the subclass OBJECT paragraph to describe any instance data that the subclass needs in addition to the instance data defined in its superclasses. Use the same syntax that you use to define instance data in a class.

For example, the definition of the instance data for the CheckingAccount subclass of the Account class might look like this:



```

Identification division.
Object.
 Data division.
 Working-storage section.
 01 CheckFee pic S9(9) value 1.
 .
 .
End Object.

```

#### RELATED TASKS

“WORKING-STORAGE SECTION for defining class instance data” on page 394

## Defining a subclass instance method

A subclass inherits the methods of its superclasses. In a subclass definition, you can override any instance method that the subclass inherits by defining an instance method with the same signature as the inherited method. You can also define new methods that the subclass needs.

The structure and syntax of a subclass instance method are identical to those of a class instance method. Define subclass instance methods in the PROCEDURE DIVISION of the OBJECT paragraph of the subclass definition.

“Example: defining a subclass (with methods)”

#### RELATED TASKS

“Defining a class instance method” on page 395

“Overriding an instance method” on page 399

“Overloading an instance method” on page 400

## Example: defining a subclass (with methods)

The following example shows the instance method definitions for the CheckingAccount subclass of the Account class.

The processCheck method invokes the Java instance methods getAmount and getPayee of the Check class to get the check data. It invokes the credit and debit instance methods inherited from the Account class to credit the payee and debit the payer of the check.

The print method overrides the print instance method defined in the Account class. It invokes the overridden print method to display account status, and also displays the check fee. CheckFee is an instance data item defined in the subclass.

(The Account class was shown in “Example: defining a method” on page 402.)

### CheckingAccount class (subclass of Account)

```

cbl thread,pgmname(longmixed)
Identification Division.
Class-id. CheckingAccount inherits Account.
Environment Division.
Configuration section.
Repository.
 Class CheckingAccount is "CheckingAccount"
 Class Check is "Check"
 Class Account is "Account".
*
* (FACTORY paragraph not shown)
*
Identification division.
Object.

```

```

Data division.
Working-storage section.
01 CheckFee pic S9(9) value 1.
Procedure Division.
*
* processCheck method to get the check amount and payee,
* add the check fee, and invoke inherited methods debit
* to debit the payer and credit to credit the payee:
Identification Division.
Method-id. "processCheck".
Data division.
Local-storage section.
01 amount pic S9(9) binary.
01 payee usage object reference Account.
Linkage section.
01 aCheck usage object reference Check.
*
* Procedure Division using by value aCheck.
* Invoke aCheck "getAmount" returning amount
* Invoke aCheck "getPayee" returning payee
* Invoke payee "credit" using by value amount
* Add checkFee to amount
* Invoke self "debit" using by value amount.
End method "processCheck".
*
* print method override to display account status:
Identification Division.
Method-id. "print".
Data division.
Local-storage section.
01 printableFee pic $$,$$$,$$9.
Procedure Division.
* Invoke super "print"
* Move CheckFee to printableFee
* Display " Check fee: " printableFee.
End method "print".
*
End Object.
*
End class CheckingAccount.

```

#### RELATED TASKS

Chapter 13, “Compiling, linking, and running OO applications,” on page 219  
 “Invoking methods (INVOKE)” on page 407  
 “Overriding an instance method” on page 399  
 “Invoking overridden superclass methods” on page 411

---

## Defining a factory section

Use the **FACTORY** paragraph in a class definition to define data and methods that are to be associated with the class itself rather than with individual object instances.

COBOL *factory data* is equivalent to Java private static data. A single copy of the data is instantiated for the class and is shared by all object instances of the class. You most commonly use factory data when you want to gather data from all the instances of a class. For example, you could define a factory data item to keep a running total of the number of instances of the class that are created.

COBOL *factory methods* are equivalent to Java public static methods. The methods are supported by the class independently of any object instance. You most

commonly use factory methods to customize object creation when you cannot use VALUE clauses alone to initialize instance data.

By contrast, you use the OBJECT paragraph in a class definition to define data that is created for each object instance of the class, and methods that are supported for each object instance of the class.

A factory definition consists of three divisions, followed by an END FACTORY statement:

*Table 55. Structure of factory definitions*

Division	Purpose	Syntax
IDENTIFICATION (required)	Identify the start of the factory definition.	IDENTIFICATION DIVISION. FACTORY.
DATA (optional)	Describe data that is allocated once for the class (as opposed to data allocated for each instance of a class).	“WORKING-STORAGE SECTION for defining factory data” (optional)
PROCEDURE (optional)	Define factory methods.	Factory method definitions: “Defining a factory method” on page 420

“Example: defining a factory (with methods)” on page 422

#### RELATED TASKS

“Defining a class” on page 390

“Instantiating COBOL classes” on page 413

“Wrapping procedure-oriented COBOL programs” on page 427

“Structuring OO applications” on page 428

## WORKING-STORAGE SECTION for defining factory data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the FACTORY paragraph to describe the *factory data* that a COBOL class needs, that is, statically allocated data to be shared by all object instances of the class.

The FACTORY keyword, which you must immediately precede with an IDENTIFICATION DIVISION declaration, indicates the beginning of the definitions of the factory data and factory methods for the class. For example, the definition of the factory data for the Account class might look like this:

```
Identification division.
Factory.
Data division.
Working-storage section.
01 NumberOfAccounts pic 9(6) value zero.
 . . .
End Factory.
```

You can initialize simple factory data by using VALUE clauses as shown above.

COBOL factory data is equivalent to Java private static data. No other class or subclass (nor instance method in the same class, if any) can reference COBOL factory data directly. Factory data is global to all factory methods that the FACTORY paragraph defines. If you want to make factory data accessible from outside the FACTORY paragraph, define factory attribute (get or set) methods for doing so.

#### RELATED TASKS

“Coding attribute (get and set) methods” on page 401

“Instantiating COBOL classes” on page 413

## Defining a factory method

Define COBOL *factory methods* in the PROCEDURE DIVISION of the FACTORY paragraph of a class definition. A factory method defines an operation that is supported by a class independently of any object instance of the class. COBOL factory methods are equivalent to Java public static methods.

You typically define factory methods for classes whose instances require complex initialization, that is, to values that you cannot assign by using VALUE clauses alone. Within a factory method you can invoke instance methods to initialize the instance data. A factory method cannot directly access instance data.

You can code factory attribute (get and set) methods to make factory data accessible from outside the FACTORY paragraph, for example, to make the data accessible from instance methods in the same class or from a client program. For example, the Account class could define a factory method getNumberOfAccounts to return the current tally of the number of accounts.

You can use factory methods to wrap procedure-oriented COBOL programs so that they are accessible from Java programs. You can code a factory method called main to enable you to run an OO application by using the java command, and to structure your applications in keeping with standard Java practice. See the related tasks for details.

In defining factory methods, you use the same syntax that you use to define instance methods. A COBOL factory method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker:

**Table 56. Structure of factory method definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Same as for a class instance method	Same as for a class instance method (required)
ENVIRONMENT (optional)	Same as for a class instance method	Same as for a class instance method
DATA (optional)	Same as for a class instance method	Same as for a class instance method
PROCEDURE (optional)	Same as for a class instance method	Same as for a class instance method

Within a class definition, you do not need to make each factory method-name unique, but you do need to give each factory method a unique signature. You can overload factory methods in exactly the same way that you overload instance methods. For example, the CheckingAccount subclass provides two versions of the factory method createCheckingAccount: one that initializes the account to have a default balance of zero, and one that allows the opening balance to be passed in. Clients can invoke either createCheckingAccount method by passing arguments that match the signature of the intended method.

If you define a data item with the same name in both the DATA DIVISION of a factory method and the DATA DIVISION of the FACTORY paragraph, a reference in the method to that data-name refers only to the method data item. The method DATA DIVISION takes precedence.

“Example: defining a factory (with methods)” on page 422

#### RELATED TASKS

“Structuring OO applications” on page 428

“Wrapping procedure-oriented COBOL programs” on page 427

“Instantiating COBOL classes” on page 413

“Defining a class instance method” on page 395

“Coding attribute (get and set) methods” on page 401

“Overloading an instance method” on page 400

“Hiding a factory or static method”

“Invoking factory or static methods”

## Hiding a factory or static method

A factory method defined in a subclass is said to *hide* an inherited COBOL or Java method that would otherwise be accessible in the subclass if the two methods have the same signature.

To hide a superclass factory method f1 in a COBOL subclass, define a factory method f1 in the subclass that has the same name and whose PROCEDURE DIVISION USING phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes f1 using the subclass name, the subclass method rather than the superclass method is invoked.

The presence or absence of a method return value and the data type of the return value used in the PROCEDURE DIVISION RETURNING phrase (if any) must be identical in the subclass factory method and the hidden superclass method.

A factory method must not hide an instance method in a Java or COBOL superclass.

“Example: defining a factory (with methods)” on page 422

#### RELATED TASKS

“Coding interoperable data types in COBOL and Java” on page 436

“Overriding an instance method” on page 399

“Invoking methods (INVOKE)” on page 407

#### RELATED REFERENCES

Inheritance, overriding, and hiding (*The Java Language Specification*)

The procedure division header (*COBOL for Windows Language Reference*)

## Invoking factory or static methods

To invoke a COBOL factory method or Java static method in a COBOL method or client program, code the class-name as the first operand of the INVOKE statement.

For example, a client program could invoke one of the overloaded CheckingAccount factory methods called createCheckingAccount to create a checking account with account number 777777 and an opening balance of \$300 by coding this statement:

```
Invoke CheckingAccount "createCheckingAccount"
 using by value 777777 300
 returning aCheckingAccount
```

To invoke a factory method from within the same class in which you define the factory method, you also use the class-name as the first operand in the INVOKE statement.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time is the method-name. The method-name must be an alphanumeric or national literal or a category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

If an invoked method is not supported in the class that you name in the INVOKE statement, a severe error condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

The conformance requirements for passing arguments to a COBOL factory method or Java static method in the USING phrase, and receiving a return value in the RETURNING phrase, are the same as those for invoking instance methods.

“Example: defining a factory (with methods)”

#### RELATED TASKS

“Invoking methods (INVOKE)” on page 407

“Using national data (Unicode) in COBOL” on page 160

“Coding interoperable data types in COBOL and Java” on page 436

#### RELATED REFERENCES

INVOKE statement (*COBOL for Windows Language Reference*)

## Example: defining a factory (with methods)

The following example updates the previous examples to show the definition of factory data and methods.

These updates are shown:

- The Account class adds factory data and a parameterized factory method, createAccount, which allows an Account instance to be created using an account number that is passed in.
- The CheckingAccount subclass adds factory data and an overloaded parameterized factory method, createCheckingAccount. One implementation of createCheckingAccount initializes the account with a default balance of zero, and the other allows the opening balance to be passed in. Clients can invoke either method by passing arguments that match the signature of the desired method.
- The TestAccounts client invokes the services provided by the factory methods of the Account and CheckingAccount classes, and instantiates the Java Check class.
- The output from the TestAccounts client program is shown.

(The previous examples were “Example: defining a method” on page 402, “Example: defining a client” on page 414, and “Example: defining a subclass (with methods)” on page 417.)

You can also find the complete source code for this example in the samples subdirectory of the COBOL install directory. (The Windows environment variable RDZvrINSTDIR, where *v* is the version number and *r* is the release number of

Rational Developer for System z, specifies the location of the COBOL install directory.) You can use the makefile in the samples subdirectory to compile and link the code.

## Account class

```

cbl thread,pgmname(longmixed),lib
Identification Division.
Class-id. Account inherits Base.
Environment Division.
Configuration section.
Repository.
 Class Base is "java.lang.Object"
 Class Account is "Account".
*
Identification division.
Factory.
 Data division.
 Working-storage section.
 01 NumberOfAccounts pic 9(6) value zero.
*
Procedure Division.
*
* createAccount method to create a new Account
* instance, then invoke the OBJECT paragraph's init
* method on the instance to initialize its instance data:
Identification Division.
Method-id. "createAccount".
Data division.
Linkage section.
 01 inAccountNumber pic S9(6) binary.
 01 outAccount object reference Account.
* Facilitate access to JNI services:
 Copy JNI.
 Procedure Division using by value inAccountNumber
 returning outAccount.
* Establish addressability to JNI environment structure:
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNINativeInterface to JNIEnv
 Invoke Account New returning outAccount
 Invoke outAccount "init" using by value inAccountNumber
 Add 1 to NumberOfAccounts.
 End method "createAccount".
*
End Factory.
*
Identification division.
Object.
Data division.
Working-storage section.
 01 AccountNumber pic 9(6).
 01 AccountBalance pic S9(9) value zero.
*
Procedure Division.
*
* init method to initialize the account:
Identification Division.
Method-id. "init".
Data division.
Linkage section.
 01 inAccountNumber pic S9(9) binary.
 Procedure Division using by value inAccountNumber.
 Move inAccountNumber to AccountNumber.
 End method "init".
*
* getBalance method to return the account balance:
Identification Division.

```

```

Method-id. "getBalance".
Data division.
Linkage section.
01 outBalance pic S9(9) binary.
Procedure Division returning outBalance.
 Move AccountBalance to outBalance.
End method "getBalance".
*
* credit method to deposit to the account:
Identification Division.
Method-id. "credit".
Data division.
Linkage section.
01 inCredit pic S9(9) binary.
Procedure Division using by value inCredit.
 Add inCredit to AccountBalance.
End method "credit".
*
* debit method to withdraw from the account:
Identification Division.
Method-id. "debit".
Data division.
Linkage section.
01 inDebit pic S9(9) binary.
Procedure Division using by value inDebit.
 Subtract inDebit from AccountBalance.
End method "debit".
*
* print method to display formatted account number and balance:
Identification Division.
Method-id. "print".
Data division.
Local-storage section.
01 PrintableAccountNumber pic ZZZZZ999999.
01 PrintableAccountBalance pic $$$,$$$,$$9CR.
Procedure Division.
 Move AccountNumber to PrintableAccountNumber
 Move AccountBalance to PrintableAccountBalance
 Display " Account: " PrintableAccountNumber
 Display " Balance: " PrintableAccountBalance.
End method "print".
*
End Object.
*
End class Account.

```

## CheckingAccount class (subclass of Account)

```

cbl thread,pgmname(longmixed),lib
Identification Division.
Class-id. CheckingAccount inherits Account.
Environment Division.
Configuration section.
Repository.
 Class CheckingAccount is "CheckingAccount"
 Class Check is "Check"
 Class Account is "Account".
*
Identification division.
Factory.
Data division.
Working-storage section.
01 NumberOfCheckingAccounts pic 9(6) value zero.
*
Procedure Division.
*
* createCheckingAccount overloaded method to create a new
* CheckingAccount instance with a default balance, invoke

```



```

* inherited instance method init to initialize the account
* number, and increment factory data tally of checking accounts:
Identification Division.
Method-id. "createCheckingAccount".
Data division.
Linkage section.
01 inAccountNumber pic S9(6) binary.
01 outCheckingAccount object reference CheckingAccount.
* Facilitate access to JNI services:
 Copy JNI.
 Procedure Division using by value inAccountNumber
 returning outCheckingAccount.
* Establish addressability to JNI environment structure:
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNINativeInterface to JNIEnv
 Invoke CheckingAccount New returning outCheckingAccount
 Invoke outCheckingAccount "init"
 using by value inAccountNumber
 Add 1 to NumberOfCheckingAccounts.
End method "createCheckingAccount".

*
* createCheckingAccount overloaded method to create a new
* CheckingAccount instance, invoke inherited instance methods
* init to initialize the account number and credit to set the
* balance, and increment factory data tally of checking accounts:
Identification Division.
Method-id. "createCheckingAccount".
Data division.
Linkage section.
01 inAccountNumber pic S9(6) binary.
01 inInitialBalance pic S9(9) binary.
01 outCheckingAccount object reference CheckingAccount.
 Copy JNI.
 Procedure Division using by value inAccountNumber
 inInitialBalance
 returning outCheckingAccount.
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNINativeInterface to JNIEnv
 Invoke CheckingAccount New returning outCheckingAccount
 Invoke outCheckingAccount "init"
 using by value inAccountNumber
 Invoke outCheckingAccount "credit"
 using by value inInitialBalance
 Add 1 to NumberOfCheckingAccounts.
End method "createCheckingAccount".

*
End Factory.
*
Identification division.
Object.
Data division.
Working-storage section.
01 CheckFee pic S9(9) value 1.
Procedure Division.

*
* processCheck method to get the check amount and payee,
* add the check fee, and invoke inherited methods debit
* to debit the payer and credit to credit the payee:
Identification Division.
Method-id. "processCheck".
Data division.
Local-storage section.
01 amount pic S9(9) binary.
01 payee usage object reference Account.
Linkage section.
01 aCheck usage object reference Check.
Procedure Division using by value aCheck.

```

```

 Invoke aCheck "getAmount" returning amount
 Invoke aCheck "getPayee" returning payee
 Invoke payee "credit" using by value amount
 Add checkFee to amount
 Invoke self "debit" using by value amount.
 End method "processCheck".
*
* print method override to display account status:
 Identification Division.
 Method-id. "print".
 Data division.
 Local-storage section.
 01 printableFee pic $,$,$,$,$9.
 Procedure Division.
 Invoke super "print"
 Move CheckFee to printableFee
 Display " Check fee: " printableFee.
 End method "print".
*
 End Object.
*
 End class CheckingAccount.

```

## Check class

```

/**
 * A Java class for check information
 */
public class Check {
 private CheckingAccount payer;
 private Account payee;
 private int amount;

 public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
 payer=inPayer;
 payee=inPayee;
 amount=inAmount;
 }

 public int getAmount() {
 return amount;
 }

 public Account getPayee() {
 return payee;
 }
}

```

## TestAccounts client program

```

cbl thread,pgmname(longmixed)
Identification division.
Program-id. "TestAccounts" recursive.
Environment division.
Configuration section.
Repository.
 Class Account is "Account"
 Class CheckingAccount is "CheckingAccount"
 Class Check is "Check".
Data Division.
* Working data is declared in Local-storage
* so that each thread has its own copy:
 Local-storage section.
 01 anAccount usage object reference Account.
 01 aCheckingAccount usage object reference CheckingAccount.
 01 aCheck usage object reference Check.
 01 payee usage object reference Account.
*

```

```

Procedure division.
Test-Account-section.
 Display "Test Account class"
* Create account 123456 with 0 balance:
 Invoke Account "createAccount"
 using by value 123456
 returning anAccount
* Deposit 500 to the account:
 Invoke anAccount "credit" using by value 500
 Invoke anAccount "print"
 Display space
*
 Display "Test CheckingAccount class"
* Create checking account 777777 with balance of 300:
 Invoke CheckingAccount "createCheckingAccount"
 using by value 777777 300
 returning aCheckingAccount
* Set account 123456 as the payee:
 Set payee to anAccount
* Initialize check for 125 to be paid by account 777777 to payee:
 Invoke Check New
 using by value aCheckingAccount, payee, 125
 returning aCheck
* Debit the payer, and credit the payee:
 Invoke aCheckingAccount "processCheck"
 using by value aCheck
 Invoke aCheckingAccount "print"
 Invoke anAccount "print"
*
 Stop Run.
End program "TestAccounts".

```

### Output produced by the TestAccounts client program

```

Test Account class
Account: 123456
Balance: $500

Test CheckingAccount class
Account: 777777
Balance: $174
Check fee: $1
Account: 123456
Balance: $625

```

#### RELATED TASKS

“Creating and initializing instances of classes” on page 412

“Defining a factory method” on page 420

“Invoking factory or static methods” on page 421

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

---

## Wrapping procedure-oriented COBOL programs

A *wrapper* is a class that provides an interface between object-oriented code and procedure-oriented code. Factory methods provide a convenient means for writing wrappers for existing procedural COBOL code to make it accessible from Java programs.

To wrap COBOL code, do these steps:

1. Create a simple COBOL class that contains a FACTORY paragraph.
2. In the FACTORY paragraph, code a factory method that uses a CALL statement to call the procedural program.

A Java program can invoke the factory method by using a static method invocation expression, thus invoking the COBOL procedural program.

#### RELATED TASKS

“Defining a class” on page 390

“Defining a factory section” on page 418

“Defining a factory method” on page 420

---

## Structuring OO applications

You can structure applications that use object-oriented COBOL syntax in one of three ways.

An OO application can begin with:

- A COBOL program, which can have any name.

You can run the program by specifying the name of the executable module at the command prompt.

- A Java class definition that contains a method called `main`. Declare `main` as `public`, `static`, and `void`, with a single parameter of type `String[]`.

You can run the application with the `java` command, specifying the name of the class that contains `main` and zero or more strings as command-line arguments.

- A COBOL class definition that contains a factory method called `main`. Declare `main` with no `RETURNING` phrase and a single `USING` parameter, an object reference to a class that is an array with elements of type `java.lang.String`. (Thus `main` is in effect `public`, `static`, and `void`, with a single parameter of type `String[]`.)

You can run the application with the `java` command, specifying the name of the class that contains `main` and zero or more strings as command-line arguments.

Structure an OO application this way if you want to:

- Run the application by using the `java` command.
- Run the application in an environment where applications must start with the `main` method of a Java class file.
- Follow standard Java programming practice.

“Examples: COBOL applications that you can run using the `java` command”

#### RELATED TASKS

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

“Running programs” on page 217

“Defining a factory method” on page 420

“Declaring arrays and strings for Java” on page 437

## Examples: COBOL applications that you can run using the `java` command

The following examples show COBOL class definitions that contain a factory method called `main`.

In each case, `main` has no `RETURNING` phrase and has a single `USING` parameter, an object reference to a class that is an array with elements of type `java.lang.String`. You can run these applications by using the `java` command.

### Displaying a message

```
cb1 thread
Identification Division.
Class-id. CBLmain inherits Base.
```

```

Environment Division.
Configuration section.
Repository.
 Class Base is "java.lang.Object"
 Class stringArray is "jobjectArray:java.lang.String"
 Class CBLmain is "CBLmain".
*
Identification Division.
Factory.
 Procedure division.
*
 Identification Division.
 Method-id. "main".
 Data division.
 Linkage section.
 01 SA usage object reference stringArray.
 Procedure division using by value SA.
 Display " >> COBOL main method entered"
 .
 End method "main".
End factory.
End class CBLmain.

```

## Echoing the input strings

```

cbl thread,lib,ssrange
Identification Division.
Class-id. Echo inherits Base.
Environment Division.
Configuration section.
Repository.
 Class Base is "java.lang.Object"
 Class stringArray is "jobjectArray:java.lang.String"
 Class jstring is "java.lang.String"
 Class Echo is "Echo".
*
Identification Division.
Factory.
 Procedure division.
*
 Identification Division.
 Method-id. "main".
 Data division.
 Local-storage section.
 01 SAlen pic s9(9) binary.
 01 I pic s9(9) binary.
 01 SAelement object reference jstring.
 01 SAelementlen pic s9(9) binary.
 01 P pointer.
 Linkage section.
 01 SA object reference stringArray.
 01 Sbuffer pic N(65535).
 Copy "JNI.cpy" suppress.
 Procedure division using by value SA.
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNINativeInterface to JNIEnv
 Call GetArrayLength using by value JNIEnvPtr SA
 returning SAlen
 Display "Input string array length: " SAlen
 Display "Input strings:"
 Perform varying I from 0 by 1 until I = SAlen
 Call GetObjectArrayElement
 using by value JNIEnvPtr SA I
 returning SAelement
 Call GetStringLength
 using by value JNIEnvPtr SAelement
 returning SAelementlen
 Call GetStringChars

```

```

 using by value JNIEnvPtr SAelement 0
 returning P
 Set address of Sbuffer to P
 Display function display-of(Sbuffer(1:SAelementlen))
 Call ReleaseStringChars
 using by value JNIEnvPtr SAelement P
 End-perform
.
End method "main".
End factory.
End class Echo.

```

#### RELATED TASKS

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

“Defining a factory method” on page 420

Chapter 25, “Communicating with Java methods,” on page 431

---

## Chapter 25. Communicating with Java methods

To achieve interlanguage interoperability with Java, you need to follow certain rules and guidelines for using services in the Java Native Interface (JNI), coding data types, and compiling COBOL programs.

You can invoke methods that are written in Java from COBOL programs, and you can invoke methods that are written in COBOL from Java programs. You need to code COBOL object-oriented language for basic Java object capabilities. For additional Java capabilities, you can call JNI services.

Because Java programs might be multithreaded and use asynchronous signals, compile COBOL programs with the `THREAD` option.

### RELATED TASKS

"Using national data (Unicode) in COBOL" on page 160

"Accessing JNI services"

"Sharing data with Java" on page 435

Chapter 24, "Writing object-oriented programs," on page 387

Chapter 13, "Compiling, linking, and running OO applications," on page 219

Chapter 30, "Preparing COBOL programs for multithreading," on page 497

### RELATED REFERENCES

Java 2 Enterprise Edition Developer's Guide

---

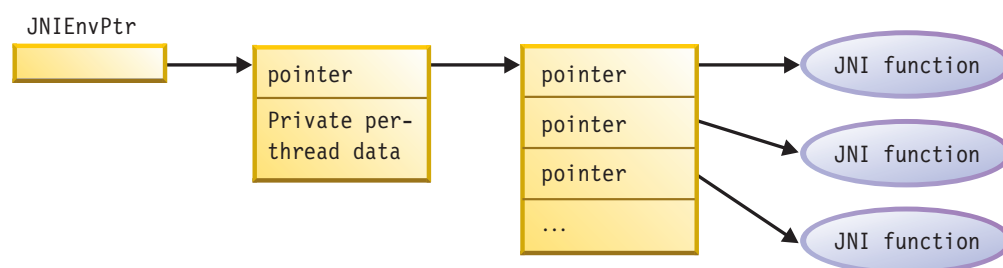
## Accessing JNI services

The Java Native Interface (JNI) provides many callable services that you can use when you develop applications that mix COBOL and Java. To facilitate access to these services, copy `JNI.cpy` into the `LINKAGE SECTION` of your COBOL program.

The `JNI.cpy` copybook contains these definitions:

- COBOL data definitions that correspond to the Java JNI types
- `JNINativeInterface`, the JNI environment structure that contains function pointers for accessing the callable service functions

You obtain the JNI environment structure by two levels of indirection from the JNI environment pointer, as the following illustration shows:



Use the special register `JNIEnvPtr` to reference the JNI environment pointer to obtain the address for the JNI environment structure. `JNIEnvPtr` is implicitly defined as `USAGE POINTER`; do not use it as a receiving data item. Before you

reference the contents of the JNI environment structure, you must code the following statements to establish its addressability:

```
Linkage section.
COPY JNI
.
.
.
Procedure division.
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNINativeInterface to JNIEnv
 .
 .
 .
```

The code above sets the addresses of the following items:

- JNIEnv, a pointer data item that JNI.cpy provides. JNIEnvPtr is the COBOL special register that contains the environment pointer.
- JNINativeInterface, the COBOL group structure that JNI.cpy contains. This structure maps the JNI environment structure, which contains an array of function pointers for the JNI callable services.

After you code the statements above, you can access the JNI callable services with CALL statements that reference the function pointers. You can pass the JNIEnvPtr special register as the first argument to the services that require the environment pointer, as shown in the following example:

```
01 InputArrayObj usage object reference jlongArray.
01 ArrayLen pic S9(9) comp-5.
.
.
.
 Call GetArrayLength using by value JNIEnvPtr InputArrayObj
 returning ArrayLen
```

**Important:** Pass all arguments to the JNI callable services by value. To be interoperable, the arguments must be in native format (for example, by being declared with the NATIVE clause on the data description entry).

Some JNI callable services require a Java class-object reference as an argument. To obtain a reference to the class object that is associated with a class, use one of the following JNI callable services:

- GetObjectClass
- FindClass

**Restriction:** The JNI environment pointer is thread specific. Do not pass it from one thread to another.

#### RELATED TASKS

“Managing local and global references” on page 434

“Handling Java exceptions”

“Coding interoperable data types in COBOL and Java” on page 436

“Defining a client” on page 403

#### RELATED REFERENCES

Appendix G, “JNI.cpy,” on page 635

The Java Native Interface

## Handling Java exceptions

Use JNI services to throw and catch Java exceptions.

**Throwing an exception:** Use one of the following services to throw a Java exception from a COBOL method:

- Throw



- ThrowNew

You must make the thrown object an instance of a subclass of `java.lang.Throwable`.

The Java virtual machine (JVM) does not recognize and process the thrown exception until the method that contains the call has completed and returned to the JVM.

**Catching an exception:** After you invoke a method that might have thrown a Java exception, you can do these steps:

1. Test whether an exception occurred.
2. If an exception occurred, process the exception.
3. Clear the exception, if clearing is appropriate.

Use the following JNI services:

- ExceptionOccurred
- ExceptionCheck
- ExceptionDescribe
- ExceptionClear

To do error analysis, use the methods supported by the exception object that is returned. This object is an instance of the `java.lang.Throwable` class.

“Example: handling Java exceptions”

### Example: handling Java exceptions

The following example shows the use of JNI services for catching an exception from Java and the use of the `PrintStackTrace` method of `java.lang.Throwable` for error analysis.

```
Repository.
 Class JavaException is "java.lang.Exception".
. . .
Local-storage section.
 01 ex usage object reference JavaException.
Linkage section.
COPY "JNI.cpy".
. . .
Procedure division.
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNINativeInterface to JNIEnv
. . .
 Invoke anObj "someMethod"
 Perform ErrorCheck
. . .
ErrorCheck.
 Call ExceptionOccurred
 using by value JNIEnvPtr
 returning ex
 If ex not = null then
 Call ExceptionClear using by value JNIEnvPtr
 Display "Caught an unexpected exception"
 Invoke ex "printStackTrace"
 Stop run
 End-if
```

## Managing local and global references

The Java virtual machine tracks the object references that you use in native methods, such as COBOL methods. This tracking ensures that the objects are not prematurely released during garbage collection.

There are two classes of such references:

### Local references

Local references are valid only while the method that you invoke runs. Automatic freeing of the local references occurs after the native method returns.

### Global references

Global references remain valid until you explicitly delete them. You can create global references from local references by using the JNI service `NewGlobalRef`.

The following object references are always local:

- Object references that are received as method parameters
- Object references that are returned as the method RETURNING value from a method invocation
- Object references that are returned by a call to a JNI function
- Object references that you create by using the `INVOKE . . . NEW` statement

You can pass either a local reference or a global reference as an object reference argument to a JNI service.

You can code methods to return either local or global references as RETURNING values. However, in either case, the reference that is received by the invoking program is a local reference.

You can pass either local or global references as USING arguments in a method invocation. However, in either case, the reference that is received by the invoked method is a local reference.

Local references are valid only in the thread in which you create them. Do not pass them from one thread to another.

### RELATED TASKS

“Accessing JNI services” on page 431

“Deleting, saving, and freeing local references”

## Deleting, saving, and freeing local references

You can manually delete local references at any point within a method. Save local references only in object references that you define in the LOCAL-STORAGE SECTION of a method.

Use a SET statement to convert a local reference to a global reference if you want to save a reference in any of these data items:

- An object instance variable
- A factory variable
- A data item in the WORKING-STORAGE SECTION of a method

Otherwise, an error occurs. These storage areas persist when a method returns; therefore a local reference is no longer valid.

In most cases you can rely on the automatic freeing of local references that occurs when a method returns. However, in some cases you should explicitly free a local reference within a method by using the JNI service `DeleteLocalRef`. Here are two situations where explicit freeing is appropriate:

- In a method you access a large object, thereby creating a local reference to the object. After extensive computations, the method returns. Free the large object if you do not need it for the additional computations, because the local reference prevents the object from being released during garbage collection.
- You create a large number of local references in a method, but do not use all of them at the same time. Because the Java virtual machine requires space to keep track of each local reference, you should free those that you no longer need. Freeing the local references helps prevent the system from running out of memory.

For example, in a COBOL method you loop through a large array of objects, retrieve the elements as local references, and operate on one element at each iteration. You can free the local reference to the array element after each iteration.

Use the following callable services to manage local references and global references.

**Table 57. JNI services for local and global references**

Service	Input arguments	Return value	Purpose
<code>NewGlobalRef</code>	<ul style="list-style-type: none"> <li>• The JNI environment pointer</li> <li>• A local or global object reference</li> </ul>	The global reference, or NULL if the system is out of memory	To create a new global reference to the object that the input object reference refers to
<code>DeleteGlobalRef</code>	<ul style="list-style-type: none"> <li>• The JNI environment pointer</li> <li>• A global object reference</li> </ul>	None	To delete a global reference to the object that the input object reference refers to
<code>DeleteLocalRef</code>	<ul style="list-style-type: none"> <li>• The JNI environment pointer</li> <li>• A local object reference</li> </ul>	None	To delete a local reference to the object that the input object reference refers to

#### RELATED TASKS

“Accessing JNI services” on page 431

## Java access controls

The Java access modifiers `protected` and `private` are not enforced when you use the Java Native Interface. Therefore a COBOL program could invoke a `protected` or `private` Java method that is not invocable from a Java client. This usage is not recommended.

## Sharing data with Java

You can share the COBOL data types that have Java equivalents. (Some COBOL data types have Java equivalents, but others do not.)

Share data items with Java in these ways:

- Pass them as arguments in the `USING` phrase of an `INVOKE` statement.
- Receive them as parameters in the `USING` phrase from a Java method.

- Receive them as the RETURNING value in an INVOKE statement.
- Return them as the value in the RETURNING phrase of the PROCEDURE DIVISION header in a COBOL method.

To pass or receive arrays and strings, declare them as object references:

- Declare an array as an object reference that contains an instance of one of the special array classes.
- Declare a string as an object reference that contains an instance of the jstring class.

#### RELATED TASKS

“Coding interoperable data types in COBOL and Java”

“Declaring arrays and strings for Java” on page 437

“Manipulating Java arrays” on page 438

“Manipulating Java strings” on page 441

“Invoking methods (INVOKE)” on page 407

Chapter 28, “Sharing data,” on page 467

## Coding interoperable data types in COBOL and Java

Your COBOL program can use only certain data types when communicating with Java.

Table 58. Interoperable data types in COBOL and Java

Primitive Java data type	Corresponding COBOL data type
boolean <sup>1</sup>	PIC X followed by exactly two condition-names of this form: <i>level-number data-name</i> PIC X. 88 <i>data-name-false</i> value X'00'. 88 <i>data-name-true</i> value X'01' through X'FF'.
byte <sup>1</sup>	Single-byte alphanumeric: PIC X or PIC A
short <sup>3</sup>	USAGE BINARY, COMP, COMP-4, or COMP-5, with PICTURE clause of the form S9( <i>n</i> ), where 1<= <i>n</i> <=4
int <sup>3</sup>	USAGE BINARY, COMP, COMP-4, or COMP-5, with PICTURE clause of the form S9( <i>n</i> ), where 5<= <i>n</i> <=9
long <sup>3</sup>	USAGE BINARY, COMP, COMP-4, or COMP-5, with PICTURE clause of the form S9( <i>n</i> ), where 10<= <i>n</i> <=18
float <sup>2</sup>	USAGE COMP-1
double <sup>2</sup>	USAGE COMP-2
char	Single-character elementary national: PIC N USAGE NATIONAL. (Cannot be a national group.)
class types (object references)	USAGE OBJECT REFERENCE <i>class-name</i>

**Table 58. Interoperable data types in COBOL and Java** (continued)

Primitive Java data type	Corresponding COBOL data type
<ol style="list-style-type: none"> <li>1. You must distinguish boolean from byte, because they each correspond to PIC X. PIC X is interpreted as boolean only when you define an argument or a parameter with the two condition-names as shown. Otherwise, a PIC X data item is interpreted as the Java byte type. A single-byte alphanumeric item can contain either EBCDIC or native content.</li> <li>2. Java floating-point data is represented in IEEE floating point. Floating-point data items that you pass as arguments in an INVOKE statement or receive as parameters from a Java method must be in the native format. Therefore if you compile using the -host option of the cob2 command or using the FLOAT(S390) option, each floating-point data item that you pass to or receive from a Java method must be declared with the NATIVE phrase in its data description.</li> <li>3. Binary data items that you pass as arguments in an INVOKE statement or receive as parameters from a Java method must be in the native format, for example, by being declared with the NATIVE phrase in their data description. Binary data items are interoperable with Java only if they are in the native format.</li> </ol>	

#### RELATED TASKS

“Using national data (Unicode) in COBOL” on page 160

#### RELATED REFERENCES

“cob2 options” on page 206

“BINARY” on page 229

“CHAR” on page 230

“FLOAT” on page 247

## Declaring arrays and strings for Java

When you communicate with Java, declare arrays by using the special array classes, and declare strings by using jstring. Code the COBOL data types shown in the table below.

**Table 59. Interoperable arrays and strings in COBOL and Java**

Java data type	Corresponding COBOL data type
boolean[ ]	object reference jbooleanArray
byte[ ]	object reference jbyteArray
short[ ]	object reference jshortArray
int[ ]	object reference jintArray
long[ ]	object reference jlongArray
char[ ]	object reference jcharArray
Object[ ]	object reference jobjectArray
String	object reference jstring

To use one of these classes for interoperability with Java, you must code an entry in the REPOSITORY paragraph. For example:

Configuration section.

Repository.

Class jbooleanArray is "jbooleanArray".

The REPOSITORY paragraph entry for an object array type must specify an external class-name in one of these forms:

```
"jobjectArray"
"jobjectArray:external-classname-2"
```

In the first case, the REPOSITORY entry specifies an array class in which the elements of the array are objects of type `java.lang.Object`. In the second case, the REPOSITORY entry specifies an array class in which the elements of the array are objects of type `external-classname-2`. Code a colon as the separator between the specification of the `jobjectArray` type and the external class-name of the array elements.

The following example shows both cases. In the example, `oa` defines an array of elements that are objects of type `java.lang.Object`. `aDepartment` defines an array of elements that are objects of type `com.acme.Employee`.

```
Environment Division.
Configuration Section.
Repository.
 Class jobjectArray is "jobjectArray"
 Class Employee is "com.acme.Employee"
 Class Department is "jobjectArray:com.acme.Employee".
 . . .
Linkage section.
01 oa usage object reference jobjectArray.
01 aDepartment usage object reference Department.
 . . .
Procedure division using by value aDepartment.
 . . .
```

“Examples: COBOL applications that you can run using the `java` command” on page 428

The following Java array types are currently not supported for interoperation with COBOL programs.

*Table 60. Noninteroperable array types in COBOL and Java*

Java data type	Corresponding COBOL data type
<code>float[ ]</code>	object reference <code>jfloatArray</code>
<code>double[ ]</code>	object reference <code>jdoubleArray</code>

#### RELATED TASKS

“REPOSITORY paragraph for defining a class” on page 392

## Manipulating Java arrays

To represent an array in a COBOL program, code a group item that contains a single elementary item that is of the data type that corresponds to the Java type of the array. Specify an `OCCURS` or `OCCURS DEPENDING ON` clause that is appropriate for the array.

For example, the following code specifies a structure to receive 500 or fewer integer values from a `jlongArray` object:

```
01 longArray.
 02 X pic S9(10) comp-5 occurs 1 to 500 times depending on N.
```

To operate on objects of the special Java-array classes, call the services that the JNI provides. You can use services to access and set individual elements of an array and for the following purposes, using the services cited:

Table 61. JNI array services

Service	Input arguments	Return value	Purpose
GetArrayLength	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The array object reference</li> </ul>	The array length as a binary fullword integer	To get the number of elements in a Java array object
NewBooleanArray, NewByteArray, NewCharArray, NewShortArray, NewIntArray, NewLongArray	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The number of elements in the array, as a binary fullword integer</li> </ul>	The array object reference, or NULL if the array cannot be constructed	To create a new Java array object
GetBooleanArrayElements, GetByteArrayElements, GetCharArrayElements, GetShortArrayElements, GetIntArrayElements, GetLongArrayElements	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The array object reference</li> <li>A pointer to a boolean item. If the pointer is not null, the boolean item is set to true if a copy of the array elements was made. If a copy was made, the corresponding ReleasexxxArrayElements service must be called if changes are to be written back to the array object.</li> </ul>	A pointer to the storage buffer	To extract the array elements from a Java array into a storage buffer. The services return a pointer to the storage buffer, which you can use as the address of a COBOL group data item defined in the LINKAGE SECTION.
ReleaseBooleanArrayElements, ReleaseByteArrayElements, ReleaseCharArrayElements, ReleaseShortArrayElements, ReleaseIntArrayElements, ReleaseLongArrayElements	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The array object reference</li> <li>A pointer to the storage buffer</li> <li>The release mode, as a binary fullword integer. See Java JNI documentation for details. (Recommendation: Specify 0 to copy back the array content and free the storage buffer.)</li> </ul>	None; the storage for the array is released.	To release the storage buffer that contains elements that have been extracted from a Java array, and conditionally map the updated array values back into the array object
NewObjectArray	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The number of elements in the array, as a binary fullword integer</li> <li>An object reference for the array element class</li> <li>An object reference for the initial element value. All array elements are set to this value.</li> </ul>	The array object reference, or NULL if the array cannot be constructed <sup>1</sup>	To create a new Java object array
GetObjectArrayElement	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The array object reference</li> <li>An array element index, as a binary fullword integer using origin zero</li> </ul>	An object reference <sup>2</sup>	To return the element at a given index within an object array
SetObjectArrayElement	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>The array object reference</li> <li>The array element index, as a binary fullword integer using origin zero</li> <li>The object reference for the new value</li> </ul>	None <sup>3</sup>	To set an element within an object array

Table 61. JNI array services (continued)

Service	Input arguments	Return value	Purpose
<ol style="list-style-type: none"> <li>1. NewObjectArray throws an exception if the system runs out of memory.</li> <li>2. GetObjectArrayElement throws an exception if the index is not valid.</li> <li>3. SetObjectArrayElement throws an exception if the index is not valid or if the new value is not a subclass of the element class of the array.</li> </ol>			

“Examples: COBOL applications that you can run using the java command” on page 428

“Example: processing a Java int array”

#### RELATED TASKS

“Coding interoperable data types in COBOL and Java” on page 436

“Declaring arrays and strings for Java” on page 437

“Accessing JNI services” on page 431

### Example: processing a Java int array

The following example shows the use of the Java-array classes and JNI services to process a Java array in COBOL.

```

cbl lib,thread
Identification division.
Class-id. OOARRAY inherits Base.
Environment division.
Configuration section.
Repository.
 Class Base is "java.lang.Object"
 Class jintArray is "jintArray".
Identification division.
Object.
Procedure division.
 Identification division.
 Method-id. "ProcessArray".
 Data Division.
 Local-storage section.
 01 intArrayPtr pointer.
 01 intArrayLen pic S9(9) comp-5.
 Linkage section.
 COPY JNI.
 01 inIntArrayObj usage object reference jintArray.
 01 intArrayGroup.
 02 X pic S9(9) comp-5
 occurs 1 to 1000 times depending on intArrayLen.
 Procedure division using by value inIntArrayObj.
 Set address of JNIEnv to JNIEnvPtr
 Set address of JNIInterface to JNIEnv

 Call GetArrayLength
 using by value JNIEnvPtr inIntArrayObj
 returning intArrayLen
 Call GetIntArrayElements
 using by value JNIEnvPtr inIntArrayObj 0
 returning IntArrayPtr
 Set address of intArrayGroup to intArrayPtr

* . . . process the array elements X(I) . . .

 Call ReleaseIntArrayElements
 using by value JNIEnvPtr inIntArrayObj intArrayPtr 0.
 End method "ProcessArray".
End Object.
End class OOARRAY.
```



## Manipulating Java strings

COBOL represents Java String data in Unicode. To represent a Java String in a COBOL program, declare the string as an object reference of the `jstring` class. Then use JNI services to set or extract COBOL national (Unicode) or UTF-8 data from the object.

**Services for Unicode:** Use the following standard services to convert between `jstring` object references and COBOL `USAGE NATIONAL` data items. Use these services for applications that you intend to be portable between the workstation and the mainframe. Access these services by using function pointers in the `JNINativeInterface` environment structure.

Table 62. Services that convert between `jstring` references and national data

Service	Input arguments	Return value
<code>NewString</code> <sup>1</sup>	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>A pointer to a Unicode string, such as a COBOL national data item</li> <li>The number of characters in the string; binary fullword</li> </ul>	<code>jstring</code> object reference
<code>GetStringLength</code>	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>A <code>jstring</code> object reference</li> </ul>	The number of Unicode characters in the <code>jstring</code> object reference; binary fullword
<code>GetStringChars</code> <sup>1</sup>	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>A <code>jstring</code> object reference</li> <li>A pointer to a boolean data item, or NULL</li> </ul>	<ul style="list-style-type: none"> <li>A pointer to the array of Unicode characters extracted from the <code>jstring</code> object, or NULL if the operation fails. The pointer is valid until it is released with <code>ReleaseStringChars</code>.</li> <li>When the pointer to the boolean data item is not null, the boolean value is set to true if a copy is made of the string and to false if no copy is made.</li> </ul>
<code>ReleaseStringChars</code>	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>A <code>jstring</code> object reference</li> <li>A pointer to the array of Unicode characters that was returned from <code>GetStringChars</code></li> </ul>	None; the storage for the array is released.
1. This service throws an exception if the system runs out of memory.		

**Services for UTF-8:** You can use the following services, an extension of the JNI, to convert between `jstring` object references and UTF-8 strings. Use these services in programs that do not need to be portable to the mainframe. Access these services by using function pointers in the JNI environment structure `JNINativeInterface`.

Table 63. Services that convert between `jstring` references and UTF-8 data

Service	Input arguments	Return value
<code>NewStringUTF</code> <sup>1</sup>	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>A pointer to a null-terminated UTF-8 string</li> </ul>	<code>jstring</code> object reference, or NULL if the string cannot be constructed
<code>GetStringUTFLength</code>	<ul style="list-style-type: none"> <li>The JNI environment pointer</li> <li>A <code>jstring</code> object reference</li> </ul>	The number of bytes needed to represent the string in UTF-8 format; binary fullword

Table 63. Services that convert between jstring references and UTF-8 data (continued)

Service	Input arguments	Return value
GetStringUTFChars <sup>1</sup>	<ul style="list-style-type: none"> <li>• The JNI environment pointer</li> <li>• A jstring object reference</li> <li>• A pointer to a boolean data item, or NULL</li> </ul>	<ul style="list-style-type: none"> <li>• Pointer to an array of UTF-8 characters extracted from the jstring object, or NULL if the operation fails. The pointer is valid until it is released with ReleaseStringUTFChars.</li> <li>• When the pointer to the boolean data item is not null, the boolean value is set to true if a copy is made of the string and to false if no copy is made.</li> </ul>
ReleaseStringUTFChars	<ul style="list-style-type: none"> <li>• The JNI environment pointer</li> <li>• A jstring object reference</li> <li>• A pointer to the UTF-8 string that was derived from the jstring argument by using GetStringUTFChars</li> </ul>	None; the storage for the UTF-8 string is released.
1. This service throws an exception if the system runs out of memory.		

#### RELATED TASKS

“Accessing JNI services” on page 431

“Coding interoperable data types in COBOL and Java” on page 436

“Declaring arrays and strings for Java” on page 437

“Using national data (Unicode) in COBOL” on page 160

Chapter 13, “Compiling, linking, and running OO applications,” on page 219

---

## Part 7. Working with more complex applications

### Chapter 26. Porting applications between platforms . . . . . 445

Getting mainframe applications to compile . . . . .	445
Getting mainframe applications to run: overview . . . . .	447
Fixing differences caused by data representations . . . . .	447
Handling differences in ASCII SBCS and EBCDIC SBCS characters . . . . .	447
Handling differences in native and nonnative formats . . . . .	448
Handling differences in IEEE and hexadecimal data . . . . .	448
Handling differences in ASCII DBCS and EBCDIC DBCS strings . . . . .	449
Fixing environment differences that affect portability . . . . .	450
Fixing differences caused by language elements . . . . .	450
Writing code to run on the mainframe . . . . .	451
Writing applications that are portable between the Windows-based and AIX workstations . . . . .	451

### Chapter 27. Using subprograms . . . . . 453

Main programs, subprograms, and calls . . . . .	453
Ending and reentering main programs or subprograms . . . . .	454
Calling nested COBOL programs . . . . .	454
Nested programs . . . . .	455
Example: structure of nested programs . . . . .	456
Scope of names . . . . .	457
Local names . . . . .	457
Global names . . . . .	457
Searches for name declarations . . . . .	457
Calling nonnested COBOL programs . . . . .	458
CALL identifier and CALL literal . . . . .	458
Call interface conventions . . . . .	459
CDECL . . . . .	459
OPTLINK . . . . .	460
SYSTEM . . . . .	461
Calling between COBOL and C/C++ programs . . . . .	462
Initializing environments . . . . .	462
Passing data between COBOL and C/C++ . . . . .	462
Setting linkage conventions for COBOL and C/C++ . . . . .	463
Collapsing stack frames and terminating run units or processes . . . . .	463
COBOL and C/C++ data types . . . . .	464
Example: COBOL program calling C/C++ DLL . . . . .	465
Making recursive calls . . . . .	466

### Chapter 28. Sharing data . . . . . 467

Passing data . . . . .	467
Describing arguments in the calling program . . . . .	469
Describing parameters in the called program . . . . .	469
Testing for OMITTED arguments . . . . .	470
Coding the LINKAGE SECTION . . . . .	470

Coding the PROCEDURE DIVISION for passing arguments . . . . .	471
Grouping data to be passed . . . . .	471
Handling null-terminated strings . . . . .	472
Using pointers to process a chained list . . . . .	472
Example: using pointers to process a chained list . . . . .	473
Using procedure and function pointers . . . . .	475
Dealing with a Windows restriction . . . . .	476
Coding multiple entry points . . . . .	477
Passing return-code information . . . . .	477
Understanding the RETURN-CODE special register . . . . .	477
Using PROCEDURE DIVISION RETURNING . . . . .	478
Specifying CALL . . . . .	478
Specifying CALL . . . . .	478
Sharing data by using the EXTERNAL clause . . . . .	478
Sharing files between programs (external files) . . . . .	479
Example: using external files . . . . .	479
Input-output using external files . . . . .	480
Using command-line arguments . . . . .	482
Example: command-line arguments . . . . .	482

### Chapter 29. Building dynamic link libraries . . . 485

Static linking and dynamic linking . . . . .	485
How the linker resolves references to DLLs . . . . .	486
Creating DLLs . . . . .	486
Example: DLL source file and related files . . . . .	487
Module definition file . . . . .	488
Resolving DLL references at run time . . . . .	488
Resolving DLL references at link time . . . . .	488
Compiling and linking the DLL . . . . .	489
Creating module definition files . . . . .	489
Reserved words for module statements . . . . .	490
Summary of module statements . . . . .	490
BASE . . . . .	491
DESCRIPTION . . . . .	491
EXPORTS . . . . .	492
HEAPSIZE . . . . .	493
LIBRARY . . . . .	493
NAME . . . . .	494
STACKSIZE . . . . .	494
STUB . . . . .	494
VERSION . . . . .	495

### Chapter 30. Preparing COBOL programs for multithreading . . . . . 497

Multithreading . . . . .	497
Working with language elements with multithreading . . . . .	498
Working with elements that have run-unit scope . . . . .	499
Working with elements that have program invocation instance scope . . . . .	499
Scope of COBOL language elements with multithreading . . . . .	500
Choosing THREAD to support multithreading . . . . .	500

Transferring control to multithreaded programs	501
Ending multithreaded programs . . . . .	501
Handling COBOL limitations with multithreading	502
Example: using COBOL in a multithreaded environment. . . . .	502
Source code for thr cob.c . . . . .	502
Source code for subd.cbl. . . . .	504
Source code for sube.cbl. . . . .	504

### **Chapter 31. Preinitializing the COBOL runtime environment . . . . .**

Initializing persistent COBOL environment . . . . .	507
Terminating preinitialized COBOL environment	508
Example: preinitializing the COBOL environment	509

### **Chapter 32. Processing two-digit-year dates 513**

Millennium language extensions (MLE) . . . . .	514
Principles and objectives of these extensions . . . . .	514
Resolving date-related logic problems . . . . .	515
Using a century window . . . . .	516
Example: century window . . . . .	517
Using internal bridging . . . . .	517
Example: internal bridging . . . . .	518
Moving to full field expansion. . . . .	518
Example: converting files to expanded date form . . . . .	519
Using year-first, year-only, and year-last date fields	520
Compatible dates . . . . .	521
Example: comparing year-first date fields . . . . .	522
Using other date formats . . . . .	522
Example: isolating the year. . . . .	523
Manipulating literals as dates . . . . .	523
Assumed century window . . . . .	524
Treatment of nondates . . . . .	525
Using sign conditions . . . . .	526
Performing arithmetic on date fields. . . . .	526
Allowing for overflow from windowed date fields . . . . .	527
Specifying the order of evaluation . . . . .	528
Controlling date processing explicitly . . . . .	528
Using DATEVAL . . . . .	529
Using UNDATE . . . . .	529
Example: DATEVAL . . . . .	530
Example: UNDATE . . . . .	530
Analyzing and avoiding date-related diagnostic messages . . . . .	530
Avoiding problems in processing dates . . . . .	532
Avoiding problems with packed-decimal fields	532
Moving from expanded to windowed date fields	533

---

## Chapter 26. Porting applications between platforms

Your Windows-based workstation has a different hardware and operating-system architecture than IBM mainframes or AIX workstations. Because of these differences, some problems can arise as you move COBOL programs between these environments.

The related information describes some of the differences between development platforms, and provides instructions to help you minimize portability problems.

### RELATED TASKS

“Getting mainframe applications to compile”

“Getting mainframe applications to run: overview” on page 447

“Writing code to run on the mainframe” on page 451

“Writing applications that are portable between the Windows-based and AIX workstations” on page 451

### RELATED REFERENCES

Appendix A, “Summary of differences with host COBOL,” on page 561

---

## Getting mainframe applications to compile

If you move programs to the Windows-based workstation from the mainframe and compile them in the new environment, you need to choose the right compiler options and allow for language features of mainframe COBOL. You can also use the COPY statement to help port programs.

**Choosing the right compiler options:** One mainframe COBOL compiler option is not applicable under COBOL for Windows, and is treated as a comment. It might yield unpredictable results. The compiler flags it with a W-level message:

**NOADV** Programs that require the use of NOADV are sensitive to device control characters and almost certainly are not portable. If your program relies on NOADV, revise the program such that language specification does not assume a printer control character as the first character of the level-01 record for the file.

**Allowing for language features of mainframe COBOL:** The following language features are valid in mainframe COBOL but can create errors or unpredictable results in compilation with COBOL for Windows. Where possible, the following table provides a solution to the potential problem.

Table 64. Language features that differ from mainframe COBOL

Language feature on mainframe	COBOL for Windows behavior	Solution or restriction
ACCEPT and DISPLAY statements	Determines the targets of DISPLAY or ACCEPT statements by checking COBOL environment variables	If your mainframe program expects host ddnames as the targets of ACCEPT or DISPLAY statements, define these targets by using equivalent environment variables with values set to appropriate file-names.

Table 64. Language features that differ from mainframe COBOL (continued)

Language feature on mainframe	COBOL for Windows behavior	Solution or restriction
ASSIGN clause	Uses a different syntax and mapping to the system file-name based on <i>assignment-name</i>	See ASSIGN clause ( <i>COBOL for Windows Language Reference</i> ).
CALL statement	Does not support file-name as a CALL argument	
CLOSE statement	Treats phrases FOR REMOVAL, WITH NO REWIND, and UNIT/REEL as comments	Avoid use of these phrases in portable programs.
LABEL RECORD clause	Treats phrases LABEL RECORD IS <i>data-name</i> , USE. . . AFTER. . . LABEL PROCEDURE, and GO TO MORE-LABELS as errors	You cannot port programs that depend on the user-label processing that z/OS QSAM supports.
PROCEDURE-POINTER data item	Is 4 bytes (8 bytes on the mainframe)	
RERUN clause	Treats RERUN clause as a comment	
SHIFT-IN, SHIFT-OUT special registers	Puts out an E-level message when encountering these registers unless the CHAR(EBCDIC) compiler option is in effect	Use the CHAR(EBCDIC) compiler option.
SORT-CONTROL special register	Follows the file-naming conventions for the system file-name identified by this register	Be aware of differences in naming conventions between the Windows workstation and the mainframe.
STOP RUN	Not supported in a multithreaded program	Replace STOP RUN with a call to the C exit() function
WRITE statement	Ignores ADVANCING phrase if you specify WRITE. . . ADVANCING with the environment-names C01-C12 or S01-S05	

**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 REFERENCES

“Runtime environment variables” on page 196

Appendix A, “Summary of differences with host COBOL,” on page 561

COPY statement (*COBOL for Windows Language Reference*)

---

## Getting mainframe applications to run: overview

After you download a source program and successfully compile it on the Windows-based workstation, the next step is to run the program. In many cases, you can get the same results as on the mainframe 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.

### RELATED TASKS

“Fixing differences caused by data representations”

“Fixing environment differences that affect portability” on page 450

“Fixing differences caused by language elements” on page 450

## 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.

COBOL stores signed packed-decimal data in the same manner on both the mainframe and the workstation. However, external-decimal, floating-point, binary, and unsigned packed-decimal data are by default represented differently. 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.

Most programs behave the same on the workstation and the mainframe regardless of the data representation.

### RELATED TASKS

“Handling differences in ASCII SBCS and EBCDIC SBCS characters”

“Handling differences in native and nonnative formats” on page 448

“Handling differences in IEEE and hexadecimal data” on page 448

“Handling differences in ASCII DBCS and EBCDIC DBCS strings” on page 449

### RELATED REFERENCES

“Data representation” on page 561

## 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.

The Windows-based workstation uses the ASCII character set, and the mainframe uses the EBCDIC character set. Therefore, most characters have a different hexadecimal value, as shown in the following table.

**Table 65. 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 66. ASCII comparisons contrasted with EBCDIC**

Comparison	Evaluation if ASCII	Evaluation if EBCDIC
'a' < 'A'	False	True
'A' < '1'	False	True
$x \geq '0'$	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 230

"COLLSEQ" on page 233

### Handling differences in native and nonnative formats

On a workstation that has Intel hardware architecture, binary integers are held in a form that is byte reversed when compared to the form in which they are held on the mainframe.

The mainframe representation is known as *big-endian*: the most significant digit of the number is stored at the lowest address. The Intel representation is known as *little-endian*: the least significant digit of the number is stored at the lowest address.

For most programs, this difference should create no problems. However, if a program depends on the byte-by-byte encoding values that an integer has, you should be aware of potential logic errors.

For programs that use mainframe binary data and rely on the internal representation of integer values, you should compile with the BINARY(S390) compiler option. For such programs, you should avoid the USAGE COMP-5 type, which is treated as the native binary data format regardless of whether the BINARY(S390) option is in effect.

"Examples: numeric data and internal representation" on page 46

#### RELATED REFERENCES

"BINARY" on page 229

Appendix A, "Summary of differences with host COBOL," on page 561

### 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(S390) compiler option.



The Windows-based workstation represents floating-point data using the IEEE format. The mainframe uses the zSeries 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 67. 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, zSeries floating-point representation makes a program run more slowly because the software must simulate the semantics of zSeries hardware instructions. This is a consideration especially if the FLOAT(S390) compiler option is in effect and a program has a large number of floating-point calculations.

“Examples: numeric data and internal representation” on page 46

#### RELATED REFERENCES

“FLOAT” on page 247

### Handling differences in ASCII DBCS and EBCDIC DBCS strings

To obtain mainframe behavior for alphanumeric data items that contain DBCS characters, use the CHAR(EBCDIC) compiler option and the S0SI compiler option. 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, mainframe double-byte character strings (containing EBCDIC DBCS characters) are enclosed in shift codes, and Windows-based workstation multibyte character strings (containing ASCII DBCS characters) are not enclosed in shift codes. The hexadecimal values used to represent the same characters are also different.

In DBCS data items, mainframe 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 workstation 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 230

“SOSI” on page 260

## Fixing environment differences that affect portability

Differences in file-names and control codes between workstation and mainframe platforms can affect the portability of your programs.

File naming conventions on the Windows-based workstation are very different from those on the mainframe. This difference can affect portability if you use file-names in your COBOL source programs. The following file-name, for example, is valid on the Windows-based workstation but not on the mainframe:

```
\users\joesmith\programs\cobol\myfile.cbl
```

Some characters that have no particular meaning on the mainframe are interpreted as control characters on the Windows-based workstation. 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 the Windows-based workstation as they do on the mainframe. However, be aware of the differences in file-status values used in I/O processing.

If your program responds to status key data items, you should be concerned with two issues, depending on whether the program is written to respond to the first or the second status key:

- 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 the Windows-based workstation that it has on the mainframe. 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 the Windows-based workstation than the VSAM file system does on the mainframe. If your program relies on the interpretation of the second file status data item, it is probably not portable.

#### RELATED TASKS

“Using file status keys” on page 148

“Using file system status codes” on page 150

---

## Writing code to run on the mainframe

You can use IBM COBOL for Windows to write new applications, taking advantage of the productivity gains and increased flexibility of using your Windows-based workstation. However, you need to avoid using language features that are not supported by IBM mainframe COBOL.

IBM COBOL for Windows supports several language features not supported by the IBM mainframe COBOL compilers. As you write code on the Windows-based workstation that is intended to run on the mainframe, avoid using these features:

- Code-page names as arguments to the DISPLAY-OF or NATIONAL-OF intrinsic function
- ASSIGN USING *data-name*
- READ statement using PREVIOUS phrase
- START statement using <, <=, or NOT > in the KEY phrase
- >>CALLINTERFACE compiler directive

Several compiler options are not available with the mainframe compilers. Do not use any of the following options in your source code if you intend to port the code to the mainframe:

- BINARY(NATIVE)
- CALLINT
- CHAR(NATIVE)
- ENTRYINT
- FLOAT(NATIVE)
- PROBE

Be aware of the difference in file-naming conventions between Windows and other 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* and *library-name*)

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 for execution in a multithreaded environment.

---

## Writing applications that are portable between the Windows-based and AIX workstations

The Windows and AIX language support is almost identical. However, there are differences between the Windows and the AIX platforms that you should keep in mind.

When developing applications to be portable between the Windows-based and the AIX workstations, consider these items:

- Hard-coded file-names in source programs can lead to problems. Instead of hard-coding the names, use mnemonic names so that you can compile a program without having to change the source code. In particular, consider how you refer to files in the following language elements:
  - ACCEPT and DISPLAY statement
  - ASSIGN clause
  - COPY (*text-name* and *library-name*) statement
- Windows represents integers in *little-endian* format. AIX maintains integers in *big-endian* format. Therefore, if your Windows COBOL program depends on the internal representation of an integer, the program is probably not portable to AIX. Avoid writing programs that rely on such internal representation. If your program requires manipulating the internal representation of Windows-format integers, use the BINARY(S390) compiler option and avoid using USAGE COMP-5 data items.
- Windows represents class national data items in little-endian format. AIX represents class national data items in big-endian format. Therefore, if your COBOL program depends on the internal representation of such data items, the program is probably not portable between COBOL for Windows and COBOL for AIX. Avoid writing programs that depend on such internal representation.  
If your program does not depend on the internal representation of class national data items, your program is probably portable between COBOL for Windows and COBOL for AIX, but you must convert file data to the representation of the target platform.

#### RELATED REFERENCES

“BINARY” on page 229

---

## Chapter 27. 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”

### RELATED TASKS

“Ending and reentering main programs or subprograms” on page 454

“Calling nested COBOL programs” on page 454

“Calling nonnested COBOL programs” on page 458

“Calling between COBOL and C/C++ programs” on page 462

“Making recursive calls” on page 466

### RELATED REFERENCES

“Call interface conventions” on page 459

---

## 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 454

“Calling nested COBOL programs” on page 454

“Calling nonnested COBOL programs” on page 458

“Calling between COBOL and C/C++ programs” on page 462

“Making recursive calls” on page 466

---

## Ending and reentering main programs or subprograms

Whether a program is left in its last-used state or its initial state, and to what 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. 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 with 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 where 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 attribute 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 13

### RELATED TASKS

“Calling nested COBOL programs”

“Making recursive calls” on page 466

---

## 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.
- The linkage that is generated for nested programs is always OPTLINK. If you are calling a nested program using a CALL *identifier* statement, ensure that the OPTLINK linkage convention is used for that CALL statement by specifying the CALLINT(OPTLINK) compiler option or the >>CALLINT OPTLINK compiler directive.

#### RELATED CONCEPTS

"Nested programs"

#### RELATED REFERENCES

"Scope of names" on page 457

"OPTLINK" on page 460

"CALLINT" on page 229

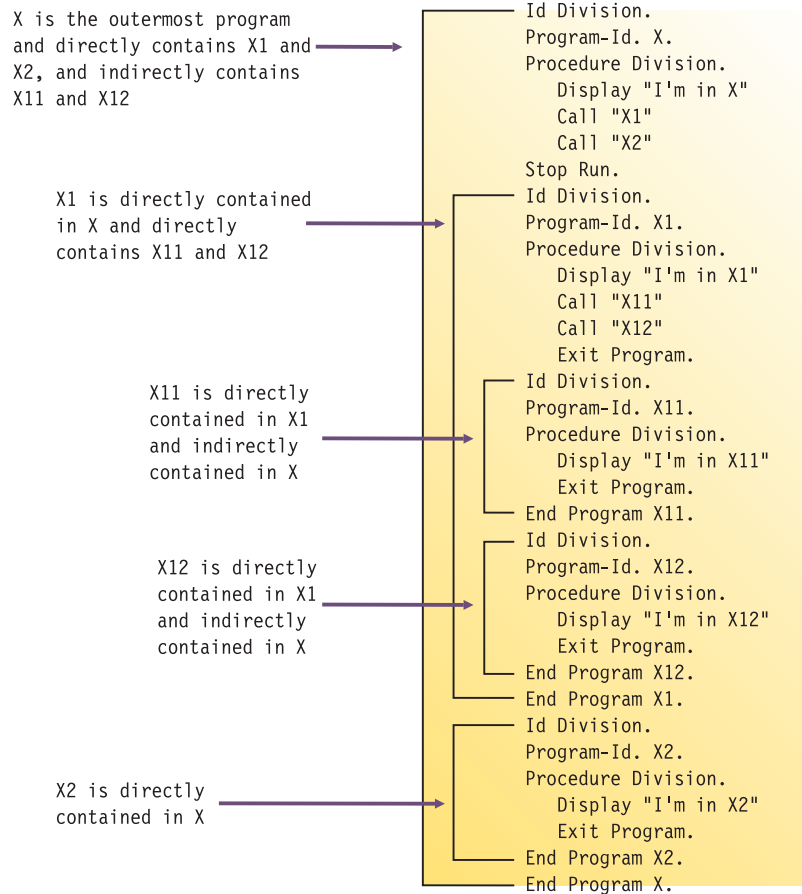
## Nested programs

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, 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 the application.
- Nested programs let you 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”

#### RELATED TASKS

“Calling nested COBOL programs” on page 454

#### RELATED REFERENCES

“Scope of names” on page 457

## Example: structure of nested programs

The following example shows a nested structure with some nested programs that are identified as **COMMON**.

```

Program-Id. A.
Program-Id. A1.
Program-Id. A11.
Program-Id. A111.
End Program A111.
End Program A11.
Program-Id. A12 is Common.
End Program A12.
End Program A1.
Program-Id. A2 is Common.
End Program A2.
Program-Id. A3 is Common.
End Program A13.
End Program A.

```



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 DLL (*dynamic linking*). COBOL for Windows also provides for runtime resolution of a target subprogram from a DLL.

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* when the DYNAM option is in effect.

### RELATED CONCEPTS

“CALL identifier and CALL literal”

“Static linking and dynamic linking” on page 485

### RELATED REFERENCES

“DYNAM” on page 238

CALL statement (*COBOL for Windows 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 DLL 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 DLL 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. When 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 (.EXE or .DLL files) in an application must not statically call the same nonnested subprogram.

#### RELATED CONCEPTS

"Static linking and dynamic linking" on page 485

#### RELATED TASKS

"Calling nonnested COBOL programs" on page 458

"Creating DLLs" on page 486

#### RELATED REFERENCES

"DYNAM" on page 238

CALL statement (*COBOL for Windows Language Reference*)

---

## Call interface conventions

The call interface convention that you use determines the order of passed parameters on the stack, the methods for sending data to a program, how returned data is received from a program, and methods of name decoration.

You can use any of the call interface conventions shown in the related references below with your COBOL programs.

#### RELATED REFERENCES

"CDECL"

"OPTLINK" on page 460

"SYSTEM" on page 461

## CDECL

The CDECL call interface convention is used by many Windows C and C++ systems. With CDECL, all general-purpose registers are preserved except for EAX, ECX, and EDX.

The following rules apply to CDECL:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The calling program removes the parameters from the stack.
- Floating-point values are returned in ST(0), the top register of the floating-point register stack. All programs that return non-floating-point values return them in EAX, except for the special case of returning items less than or equal to 4 bytes in size. Programs that return non-floating-point values return them as shown in the table below.

**Table 68. Location of returned non-floating-point items with CDECL**

Size of item in bytes	Value returned in	Comment
8	EAX-EDX pair	
5-7	EAX	The address for return values is passed as a hidden parameter to EAX.
4	EAX	
3	EAX	The address for return values is passed as a hidden parameter to EAX.
2	AX	
1	AL	

For programs that return items that are 5, 6, 7, or more than 8 bytes in size, the address for the return values is passed as a hidden parameter, and the address is passed back in EAX.

- Program-names are decorated with an underscore prefix when they appear in object modules. For example, a program named fred in the source program will appear as \_fred in the object.

When you build export or import lists in module definition files, use the decorated version of the name. If you use undecorated names in the module definition file, you must give the object files to the ILIB utility along with the module definition file. ILIB will use the object files to determine how each name ended up after decoration.

#### RELATED TASKS

“Calling between COBOL and C/C++ programs” on page 462  
“Creating module definition files” on page 489

#### RELATED REFERENCES

“CALLINT” on page 229  
“ENTRYINT” on page 239  
Chapter 15, “Compiler-directing statements,” on page 273

## OPTLINK

The OPTLINK call interface convention provides a faster call than the SYSTEM convention.

The following rules apply to OPTLINK:

- Program-names are prefixed with a question-mark character (?).
- Parameters are pushed from right to left onto the stack.
- The caller removes the parameters from the stack.

The registers are used as follows:

- The general-purpose registers EBP, EBX, EDI, and ESI are preserved across the call.
- The general-purpose registers EAX, EDX, and ECX are not preserved across the call.
- Floating-point registers are not preserved across the call.
- The three conforming parameters that are lexically leftmost (*conforming parameters* are 2-byte and 4-byte binary items, and all pointer types) are passed in EAX, EDX, and ECX, respectively.
- Up to four floating-point parameters (the lexically first four) are passed in extended-precision (80-bit) format in the floating-point register stack. All other parameters are passed on the runtime stack.
- Space for the parameters in registers is allocated on the stack, but the parameters are not copied into that space.
- Conforming return values, with the exception of 64-bit binary items, are returned in EAX. In the case of 64-bit binary items, the most significant 32 bits are returned in EDX and the least significant 32 bits are returned in EAX.
- Floating-point return values are returned in extended-precision format in the topmost register of the floating-point stack.
- Calls that return aggregates pass a hidden first parameter that is the address of a storage area determined by the caller. This area becomes the returned value. The

hidden pointer parameter is always considered nonconforming, and is not passed in a register. The called program must load it into EAX before returning.

RELATED REFERENCES

- “CALLINT” on page 229
- “ENTRYINT” on page 239
- Chapter 15, “Compiler-directing statements,” on page 273

## SYSTEM

The SYSTEM call interface convention is the standard calling convention for the Microsoft Windows application programming interface. Unlike CDECL, the called program, instead of the calling program, cleans the stack.

The following rules apply to SYSTEM:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in lexical right-to-left order.
- The called program removes the parameters from the stack.
- Floating-point values are returned in ST(0), the top register of the floating-point register stack. Programs that return non-floating-point values return them as shown in the table below.

Table 69. Location of returned non-floating-point items with SYSTEM

Size of item in bytes	Value returned in	Comment
8	EAX-EDX pair	
5-7	EAX	The address for return values is passed as a hidden parameter in EAX.
4	EAX	
3	EAX	The address for return values is passed as a hidden parameter in EAX.
2	AX	
1	AL	

For programs that return items that are 5, 6, 7, or more than 8 bytes in size, the address for the return values is passed as a hidden parameter and the address is passed back in EAX.

- Program-names are decorated with an underscore prefix, and a suffix which consists of an at character (@) followed by the number of bytes of parameters (in decimal). Parameters of fewer than 4 bytes are rounded up to 4 bytes. Structure sizes are also rounded up to a multiple of 4 bytes.

When building export or import lists in module definition files, use the decorated version of the name. If you use undecorated names in the module definition file, you must give the object files to the ILIB utility along with the module definition file. ILIB will use the object files to determine how each name ended up after decoration.

RELATED TASKS

- “Dealing with a Windows restriction” on page 476
- “Coding multiple entry points” on page 477
- “Creating module definition files” on page 489

#### RELATED REFERENCES

"CALLINT" on page 229

"ENTRYINT" on page 239

Chapter 15, "Compiler-directing statements," on page 273

"Linker errors in program-names" on page 214

---

## 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. The rules and guidelines referenced below describe how to perform these interlanguage calls.

Unqualified references to "C/C++" in the referenced sections are to Microsoft Visual C++ for Windows.

#### RELATED TASKS

"Initializing environments"

"Passing data between COBOL and C/C++"

"Setting linkage conventions for COBOL and C/C++" on page 463

"Collapsing stack frames and terminating run units or processes" on page 463

## Initializing environments

To efficiently make repeated calls from C/C++ to COBOL subprograms, you must preinitialize the COBOL environment before calling the subprograms.

#### RELATED CONCEPTS

Chapter 31, "Preinitializing the COBOL runtime environment," on page 507

## 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
- OBJECT REFERENCE
- POINTER
- PROCEDURE-POINTER

In C/C++, you can call a program with a varying number of parameters by using the `va_start`, `va_arg`, and `va_end` macros to manage the variable aspect of the

parameter list. You need to know how the called program determines the end of the parameter list, however. For example, some programs look for a null pointer to signify the end of the list. COBOL does not terminate the parameter list with a null pointer, but you can supply a null pointer by passing `BY VALUE 0` as the last argument.

#### RELATED TASKS

Chapter 28, “Sharing data,” on page 467

#### RELATED REFERENCES

“COBOL and C/C++ data types” on page 464

## Setting linkage conventions for COBOL and C/C++

C/C++ and COBOL use different default linkage conventions. For calls between COBOL and C/C++ programs, the linkage convention must be consistent between the called and calling program. You can set the linkage convention for COBOL programs by using COBOL compiler directives or compiler options.

Use the `>>CALLINT CDECL` compiler directive or `CALLINT(CDECL)` compiler option for COBOL programs that call Microsoft Visual C++ for Windows functions.

Use the compiler directive when you want to change the linkage convention for a particular call rather than for the entire program.

Use the `ENTRYINT(CDECL)` compiler option for COBOL programs that are called by Microsoft Visual C++ for Windows functions. This option sets the linkage convention to that of the `CDECL` linkage convention of Microsoft Visual C++ for Windows. Use the `ENTRYINT(OTLINK)` compiler option for COBOL programs that are called by IBM C/C++ for Windows functions.

#### RELATED REFERENCES

Chapter 15, “Compiler-directing statements,” on page 273

“Call interface conventions” on page 459

“COBOL and C/C++ data types” on page 464

“CALLINT” on page 229

“ENTRYINT” on page 239

## 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 Windows 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 295

## 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 70. COBOL and C/C++ data types*

C/C++ data types	COBOL data types
<code>wchar_t</code>	USAGE NATIONAL (PICTURE N)
<code>char</code>	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
<code>char(n)</code>	PICTURE X(n)
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 462



## Example: COBOL program calling C/C++ DLL

The following example shows a COBOL program that calls a C/C++ dynamic link library (DLL) 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/C++.
- COBOL supports calling programs with mixed-case names.
- You can call a C/C++ subprogram that is in a DLL.

Do the following steps to create and run the example:

1. Create the file sub.c from the following source. This will be your DLL.

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

BOOL WINAPI DllMain(HINSTANCE hinstDLL,DWORD,LPVOID);
_declspec (dllexport) int Sub(void);

BOOL WINAPI DllMain (
 HANDLE hModule,
 DWORD dwFunction,
 LPVOID lpNot)
{
 return TRUE;
}

_declspec (dllexport) int Sub(void)
{
 printf ("Hello from sub.\n");
 return 0;
}
```

2. Create a module definition file named sub.def that has the following statements. This definition file will make Sub an external entry point.

```
LIBRARY Sub

EXPORTS
 Sub
```

3. Compile the DLL by using the following command. The result will be a file named sub.dll that will be called by COBOL.

```
cl /EHsc /LD /DLL /DEF:sub.def sub.c
```

4. Create a COBOL program file driver.cbl with the following source to call sub.dll:

```
cb1 CALLINT(CDECL),PGMNAME(MIXED),LIST
 Identification Division.
 Program-Id. "Driver".
 Data Division.
 Working-Storage Section.
 77 rc pic 9(8) usage binary.
 Procedure Division.
 Display "Hello World, from Driver"
```

```
Call "Sub" returning rc
Display "Sub Returned to Driver"
```

```
Goback.
```

5. Compile the COBOL program and link it to the DLL by using the following command:

```
cob2 -g -v driver.cbl -IMP:sub.lib
```

6. Run the sample by issuing the command driver.

#### RELATED TASKS

Chapter 29, "Building dynamic link libraries," on page 485

#### RELATED REFERENCES

"cob2 options" on page 206

"CDECL" on page 459

CALL statement (*COBOL for Windows 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 either code the RECURSIVE clause in the PROGRAM-ID paragraph of the recursively called program or specify the THREAD compiler option. If you try to recursively call a COBOL program that does not either specify the THREAD compiler option or have the RECURSIVE clause in the PROGRAM-ID paragraph, the run unit will end abnormally.

#### RELATED TASKS

"Identifying a program as recursive" on page 6

#### RELATED REFERENCES

"THREAD" on page 265

PROGRAM-ID paragraph (*COBOL for Windows Language Reference*)

---

## Chapter 28. Sharing data

When 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

"Passing data"

"Coding the LINKAGE SECTION" on page 470

"Coding the PROCEDURE DIVISION for passing arguments" on page 471

"Using procedure and function pointers" on page 475

"Coding multiple entry points" on page 477

"Passing return-code information" on page 477

"Specifying CALL . . . RETURNING" on page 478

"Sharing data by using the EXTERNAL clause" on page 478

"Sharing files between programs (external files)" on page 479

"Using command-line arguments" on page 482

"Sharing data with Java" on page 435

---

## 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 71. Methods for passing data in the CALL statement

Code	Purpose	Comments
CALL . . . BY REFERENCE <i>identifier</i>	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.
CALL . . . BY REFERENCE ADDRESS OF <i>identifier</i>	To pass the address of <i>identifier</i> to a called program, where <i>identifier</i> 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 <i>identifier</i>	To pass a copy of the address of <i>identifier</i> to a called program	Any changes to the copy of the address will not affect the address of <i>identifier</i> , but changes to <i>identifier</i> using the copy of the address will cause changes to <i>identifier</i> .
CALL . . . BY CONTENT <i>identifier</i>	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 <i>literal</i>	To pass a copy of a literal value to a called program	
CALL . . . BY CONTENT LENGTH OF <i>identifier</i>	To pass a copy of the length of a data item	The calling program passes the length of the <i>identifier</i> 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 <i>identifier</i>	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 <i>literal</i>	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 <i>identifier</i>	To pass the address of <i>identifier</i> 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 <i>identifier</i> , but changes to <i>identifier</i> using the copy of the address will cause changes to <i>identifier</i> .
CALL . . . RETURNING	To call a C/C++ function with a function return value	

#### RELATED TASKS

"Describing arguments in the calling program" on page 469  
 "Describing parameters in the called program" on page 469  
 "Testing for OMITTED arguments" on page 470  
 "Specifying CALL . . . RETURNING" on page 478  
 "Sharing data by using the EXTERNAL clause" on page 478  
 "Sharing files between programs (external files)" on page 479  
 "Sharing data with Java" on page 435

#### RELATED REFERENCES

CALL statement (COBOL for Windows Language Reference)

The USING phrase (*COBOL for Windows Language Reference*)  
INVOKE statement (*COBOL for Windows 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 highest 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 470

“Coding the PROCEDURE DIVISION for passing arguments” on page 471

### RELATED REFERENCES

The USING phrase (*COBOL for Windows 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 if the calling program was compiled with the OPTIMIZE compiler option.

### RELATED TASKS

“Coding the LINKAGE SECTION” on page 470

### RELATED REFERENCES

The USING phrase (*COBOL for Windows 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 RPARAM1, RPARAM2, RPARAM3.
```

```
 If Address Of RPARAM2 = Null Then
```

```
 Display 'No 2nd argument was passed this time'
```

```
 Else
```

```
 Perform Process-Param-2
```

```
 End-If
```

### RELATED REFERENCES

CALL statement (*COBOL for Windows Language Reference*)

The USING phrase (*COBOL for Windows Language Reference*)

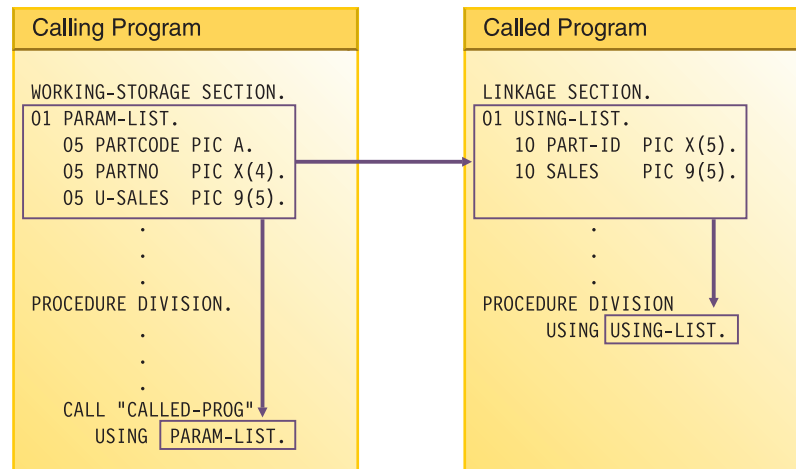
---

## 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 Windows Language Reference*)

The USING phrase (*COBOL for Windows Language Reference*)

CALL statement (*COBOL for Windows 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 470

#### RELATED REFERENCES

CALL statement (*COBOL for Windows Language Reference*)

## Handling null-terminated strings

COBOL supports null-terminated strings when you use string-handling verbs 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 Windows 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 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”

#### RELATED TASKS

“Coding the LINKAGE SECTION” on page 470

“Coding the PROCEDURE DIVISION for passing arguments” on page 471

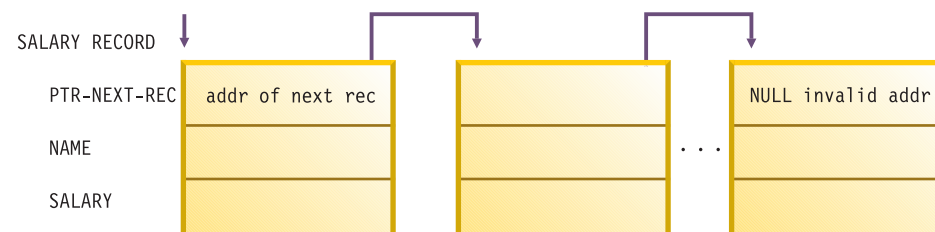
#### RELATED REFERENCES

SET statement (*COBOL for Windows 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 logic of 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 CUMULATE 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, using 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 472

---

## 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, these 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

PROCEDURE-POINTER phrase (*COBOL for Windows Language Reference*)

SET statement (*COBOL for Windows Language Reference*)

## Dealing with a Windows restriction

In general, you cannot use SYSTEM(STDCALL) linkage for programs that are called by means of a procedure pointer or a function pointer if the calls have any arguments. This restriction is due to the associated convention for forming names (known as *name decoration*).

With STDCALL linkage, the name is formed by appending to the entry name the number of bytes in the parameter list. For example, a program named abc that passes an argument by reference and a 4-byte integer by value has 8 bytes in the parameter list; the resulting name is \_abc@8. You cannot set a procedure pointer or a function pointer to the address of an entry point because there is no syntactical way to specify the arguments that are to be passed to the entry point in the SET statement; the generated name will have '0' as the number of bytes in the parameter list. The link fails because of unresolved external references when the entry point has arguments.

Use the CALLINT compiler directive to ensure that calls using a procedure pointer or a function pointer to programs with arguments use the OPTLINK convention. For example:

```
CBL
 IDENTIFICATION DIVISION.
 PROGRAM-ID. XC.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 PP1 PROCEDURE-POINTER.
 01 HW PIC X(12).
 PROCEDURE DIVISION USING XA.
* Use OPTLINK linkage:
 >>CALLINT OPTLINK
 SET PP1 TO ENTRY "X".
* Restore default linkage:
 >>CALLINT
 MOVE "Hello World." to HW
 DISPLAY "Calling X."
* Use OPTLINK linkage:
 >>CALLINT OPTLINK
 CALL PP1 USING HW.
* Restore default linkage:
 >>CALLINT
 GOBACK.
 END PROGRAM XC.

* Use OPTLINK linkage:
 CBL ENTRYINT(OPTLINK)
 IDENTIFICATION DIVISION.
 PROGRAM-ID. X.
 DATA DIVISION.
 LINKAGE SECTION.
 01 XA PIC 9(9).
 PROCEDURE DIVISION USING XA.
 DISPLAY XA.
 GOBACK.
 END PROGRAM X.
```

Without using the CALLINT compiler directives and the CALLINT compiler option, you would have an unresolved reference to \_X@0 when you link.

If the called program is in C or PL/I and uses the STDCALL interface, use the pragma statement in the called program to form the name without STDCALL name decoration.

#### RELATED REFERENCES

Chapter 15, “Compiler-directing statements,” on page 273  
“Call interface conventions” on page 459

---

## Coding multiple entry points

You cannot always use the `SYSTEM(STDCALL)` convention for calling programs that have multiple entry points (`PROCEDURE DIVISION USING . . .` and `ENTRY xxx USING . . .`).

If the number of parameters in each entry point is not the same or if the caller does not pass the number of arguments that the called entry point expects, the `STDSCALL` convention causes unpredictable results. The `STDSCALL` convention requires the called program to clean up the stack, where the calling program placed the arguments. Because the called program has no way to determine how many arguments were passed to it, it uses the expected number of arguments. When this number is not the same as the number passed, the called program cannot clean up the stack correctly.

Because you cannot use a common exit point for programs that have multiple entry points, the fact that the different entry points have a different number of arguments also makes it impossible to determine how to clean up the stack correctly.

**Data type:** Because `STDSCALL` linkage uses 4 bytes on the stack for each argument, differences in data type are immaterial.

#### RELATED REFERENCES

“SYSTEM” on page 461

---

## Passing return-code information

Use the `RETURN-CODE` special register to pass return codes between programs. (Methods do not return information in the `RETURN-CODE` special register, but they can check the register after a call to a program.)

You can also use the `RETURNING` phrase in the `PROCEDURE DIVISION` header of a method to return information to an invoking program or method. If you use `PROCEDURE DIVISION . . . RETURNING` with `CALL . . . RETURNING`, the `RETURN-CODE` register will not be set.

## Understanding 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.

You cannot set the RETURN-CODE special register by using the INVOKE statement.

## 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 you 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 allow separately compiled programs and methods (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 group item or subordinate item that is EXTERNAL.

In the run unit, any COBOL program or method 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 it 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.

---

## Sharing files between programs (external files)

To enable separately compiled programs or methods 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 on the level-01 data definition for the file status field.

Using an external file has these benefits:

- Even though 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”

### RELATED TASKS

“Using data in input and output operations” on page 12

### RELATED REFERENCES

`EXTERNAL` clause (*COBOL for Windows 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 table below 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 "eflopeno"
 Call "eflwrite"
 Call "eflclose"
 Call "eflopeni"
 Call "eflread"
 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 "eflclose"
 Goback.
End Program ef1.
Identification Division.
Program-Id.
 eflopeno.
*
* 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 eflopeno.
Identification Division.
Program-Id.
 eflwrite.

```



```

*
* 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 eflwrite.
Identification Division.
Program-Id.
 eflopeni.
*
* 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 efl
 If efs1 Not = 0
 Display "file status " efs1 " on open input"
 Stop Run
End-If
Goback.
End Program eflopeni.
Identification Division.
Program-Id.
 eflread.
*
* 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 efl
 If efs1 Not = 0
 Display "file status " efs1 " on read"
 Stop Run
End-If
Goback.
End Program eflread.

```

```

Identification Division.
Program-Id.
 eflclose.
*
* 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 efl
 If efs1 Not = 0
 Display "file status " efs1 " on close"
 Stop Run
 End-If
 Goback.
End Program eflclose.

```

---

## Using command-line arguments

You can pass arguments to a main program on the command line. The operating system calls main programs with a string that contains the arguments.

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.

If you specify the `-host` option, Windows calls all main programs with an EBCDIC string that contains the command-line arguments. The length of the string is in *big-endian* format.

“Example: command-line arguments”

### RELATED TASKS

“Manipulating null-terminated strings” on page 98

“Handling null-terminated strings” on page 472

### RELATED REFERENCES

“cob2 options” on page 206

## Example: command-line arguments

This example shows how to read the command-line arguments.

```

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 testarg.cbl
```

```
testarg 95
```

Then the resulting output is:

```

parm-len=002
parm-string='95'
case ninety-five

```



---

## Chapter 29. Building dynamic link libraries

You can build a *dynamic link library (DLL)* as a library of one or more frequently used functions. The DLL can provide those function as needed to calling programs.

A *DLL* in COBOL terms is a collection of outermost programs. Outermost programs might contain nested programs, but programs external to a DLL can call only the outermost programs (known as *entry points*) in that DLL. Just as you can compile and link several COBOL programs together as a single executable (.EXE), you can link one or more compiled outermost COBOL programs together to create a DLL.

Because outermost programs in the DLL are part of a library of programs, each program in a DLL (even if there is only one such program) is referred to as a *subprogram*.

### RELATED CONCEPTS

“Static linking and dynamic linking”

“How the linker resolves references to DLLs” on page 486

### RELATED TASKS

“Creating DLLs” on page 486

---

## Static linking and dynamic linking

By using linking, you can have a program call another program that is not contained in the source code of the calling program. Before or during execution, the object module of the calling program is linked with the object module of the called program.

*Static linking* occurs when a calling program is linked to a called program in 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 result of statically linking programs is an .EXE file or dynamic link library (DLL) subprogram that contains the executable code for multiple programs. This file includes both the calling program and the called program.

The primary advantage of static linking is that you can create self-contained, independent programs. In other words, the executable program consists of one part (the .EXE file) that you need to keep track of. Static linking has these disadvantages:

- Linked external programs are built into the executable files, making these files larger.
- You cannot change the behavior of executable files without relinking them.
- External called programs cannot be shared, requiring that duplicate copies of programs be loaded in memory if more than one calling program needs to access them.

To overcome these disadvantages, use dynamic linking.

*Dynamic linking* lets several programs use a single copy of an executable module. The executable module is separate from the programs that use it. You can build several subprograms into a DLL. Calling programs can use these subprograms as if they were part of the executable code of the calling program. You can change the dynamically linked subprograms without recompiling or relinking the calling program.

DLLs are typically used to provide common functions for a number of programs. For example, you can use DLLs to implement subprogram packages, subsystems, and interfaces to other programs, or to create object-oriented class libraries.

You can dynamically link files with the supplied runtime DLLs and with your own COBOL DLLs.

**RELATED CONCEPTS**

"CALL identifier and CALL literal" on page 458

"How the linker resolves references to DLLs"

**RELATED TASKS**

"Creating DLLs"

---

## How the linker resolves references to DLLs

When you compile a program, the compiler generates an object module for the code in the program. If you use any subprograms (*functions* in C, *subroutines* in other languages) that are in an external object module, the compiler adds an external program reference to the program's object module.

The linker resolves these external references. If it finds a reference to external subprograms in an import library or in a module definition file of a DLL, the code for the external subprogram is in a DLL. To resolve an external reference to a DLL, the linker adds information to the executable file that tells the loader where to find the DLL code when the executable file is loaded.

The DLLs that you reference can be created to load when the executable that calls them is loaded (preload) or to load when they are first referenced (load on call). However, the linker does not resolve all references to DLLs by COBOL CALL statements. With the DYNAM compiler option in effect, COBOL resolves CALL *identifier* and CALL *literal* when these calls are executed.

**RELATED CONCEPTS**

"Static linking and dynamic linking" on page 485

**RELATED TASKS**

"Creating DLLs"

---

## Creating DLLs

A DLL is built using compiled source code and a module definition (.DEF) file or export (.EXP) file.

Do the followings steps to create and use a DLL:

1. Write the source code for a DLL subprogram the way you write any other COBOL source program.
2. Construct a .DEF file or .LIB file for compiling and linking the DLL.

You can use a *module definition file*, which is a text file that describes the names, attributes, exports, and other characteristics of a program or DLL. In it you use the EXPORTS statement to list all the subprograms in the DLL that can be called by a program or by another DLL.

If you provide a .LIB file but not a module definition file, cob2 creates the .DEF file. Otherwise, to link a DLL, you must provide to cob2 a .DEF file; cob2 will then generate an import (.IMP) file and an export (.EXP) file. An *import file* tells the linker where to find the DLL subprograms used by your program. An *export file* is a binary file that tells the linker what parts the DLL exports.

3. Code the call to the DLL in your program.

Your COBOL program can make a call to a user-defined identifier rather than to a literal DLL subprogram name. Use this type of call when the name of the target subprogram is not known until run time.

Rather than using calls that are resolved at run time, you can use COBOL CALL *literal*. By default, the linker resolves these calls. However, the setting of the DYNAM compiler option determines whether the linker resolves these calls:

- If the setting is DYNAM, the call is treated the same as CALL *identifier* and is resolved at run time.
- If the setting is NODYNAM, the linker resolves CALL *literal*. With call resolution by the linker, calls using CALL *literal* to subprograms in a DLL do not cause the object code of the DLL to be included in the executable module for your main program. With CALL *literal* and NODYNAM, you can also statically link. To make such a call to an entry point in a DLL, use an import library. Unlike calls that result in runtime resolution, with link-time resolution you can have multiple entry points (outermost programs) in the DLL called.

4. Compile and link your DLL.

Use cob2 to compile your source files and create a DLL. When you use cob2 to compile and link the DLL, specify the names of all the DLL source files and the name of the module definition file. The name of the first source file is used as the name of the DLL unless you use the -dll option (for example, -dll:TEST).

“Example: DLL source file and related files”

RELATED CONCEPTS

“CALL identifier and CALL literal” on page 458

“Static linking and dynamic linking” on page 485

“How the linker resolves references to DLLs” on page 486

RELATED TASKS

“Creating module definition files” on page 489

RELATED REFERENCES

“DYNAM” on page 238

## Example: DLL source file and related files

The following COBOL source code is a simple example of a DLL source subprogram. When compiled, a DLL can contain numerous outermost programs, each of which is considered a subprogram within the DLL.

In the following example, the DLL contains only one subprogram. When another program calls the subprogram named MYDLL that is in the DLL named MYDLL.DLL and this subprogram runs, it will display the text MYDLL Entered on the computer screen.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. MYDLL.
PROCEDURE DIVISION.
 DISPLAY "MYDLL Entered".
 EXIT PROGRAM.

```

## Module definition file

The .DEF file for the compiled MYDLL.CBL source program illustrates the statements most frequently used in module definition files that build DLLs.

```

;*****
;* MYDLL.DEF *
;* Description: Provides the module definition *
;* for MYDLL.DLL, a simple COBOL DLL *
;*****
LIBRARY MYDLL INITINSTANCE TERMINSTANCE
PROTMODE
DATA MULTIPLE READWRITE LOADONCALL NONSHARED
CODE LOADONCALL EXECUTEREAD
EXPORTS
 MYDLL

```

## Resolving DLL references at run time

The following COBOL source program calls MYDLL in MYDLL.DLL:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. RTDLLRES.
*
* THIS PROGRAM USES CALL identifier to call a subprogram
* NAMED MYDLL in a DLL. IT REQUIRES A DLL
* NAMED MYDLL.DLL.
*
*
DATA DIVISION.
WORKING-STORAGE SECTION.
77 CALLNAM PIC IS X(8).
PROCEDURE DIVISION.
 DISPLAY "Start sample program RTDLLRES".
 MOVE "MYDLL" TO CALLNAM.
 CALL CALLNAM.
 DISPLAY "RTDLLRES successful".
 STOP RUN.

```

In this example, the following statement is a reference to the single subprogram MYDLL of the DLL MYDLL.DLL:

```
MOVE "MYDLL" TO CALLNAM.
```

This DLL must be in a directory defined by the COBPATH environment variable.

## Resolving DLL references at link time

The following program is identical to RTDLLRES.CBL, except that the call is made directly to a symbol exported in a .DEF file rather than to a user-defined word:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. LTDLLRES.
*
* THIS PROGRAM CALLS A SUBPROGRAM CALLED MYDLL WHICH
* RESOLVES AT LINK-TIME. THE DLL IN WHICH "MYDLL"
* IS FOUND IS NOT REQUIRED TO BE NAMED MYDLL.DLL
*
PROCEDURE DIVISION.
 DISPLAY "Start sample program LTDLLRES".
 CALL "MYDLL"
 DISPLAY "LTDLLRES successful".
 STOP RUN.

```



In this example, the call to MYDLL is not a direct reference to MYDLL.DLL (although, in this case, the DLL happens to have the same name). The call is to the symbolic name MYDLL, which was exported in the MYDLL.DEF file. MYDLL, in turn, is the name of the only COBOL subprogram in MYDLL.DLL. When LTDLLRES is built using cob2, the linker will resolve the call to MYDLL in MYDLL.DLL.

## Compiling and linking the DLL

Use cob2 to compile and link the DLL and main executable programs. The DLL and programs that call it must be compiled in separate steps.

Either of the following cob2 commands will build the DLL named MYDLL.DLL:

```
cob2 mydll.cb1 mydll.def
cob2 mydll.cb1 -dll:mydll
```

The following cob2 command will build the program RTDLLRES.EXE, which calls the DLL subprogram MYDLL with runtime resolution:

```
cob2 rtdllres.cb1
```

The following cob2 command will build the program LTDLLRES.EXE, which calls the DLL subprogram MYDLL with link-time resolution:

```
cob2 ltdllres.cb1 mydll.lib
```

---

## Creating module definition files

A module definition file contains one or more module statements. Use these statements to define attributes of your executable output file and of code and data segments in the file, and to identify data and functions that are imported into or exported from the file.

If you provide a .LIB file but not a module definition file, cob2 creates the .DEF file. Otherwise, to link a DLL, you must provide to cob2 a .DEF file. cob2 will then generate an .IMP file and an .EXP file.

You can use module definition files when:

- You create a DLL.
- You need to define attributes of the executable output file more precisely than you can with options alone. For example, to define library initialization and termination behavior, use the LIBRARY statement.
- You need to define segment attributes more precisely than you can with options alone.

When you create a module definition file:

- Use a NAME or LIBRARY statement to define the type of executable output you want. You can use only one of these statements, and it must precede all other statements in the module definition file.
- Begin comments with a semicolon (;). The linker ignores lines in the file that begin with a semicolon, and any portion of a line that follows a semicolon.
- Enter all module definition keywords (such as NAME, LIBRARY, and IOPL) in uppercase letters.
- Do not use a reserved word as a text parameter to a statement. For example, you cannot use the LIBRARY statement to name a library SHARED, because SHARED is a keyword.

#### RELATED REFERENCES

“Reserved words for module statements”

“Summary of module statements”

## Reserved words for module statements

You cannot use the following words as text parameters to a module statement. The words are either module definition keywords or reserved by the linker.

For example, you cannot use these words as the names of functions defined with the EXPORTS statement or to name a stub file with the STUB statement.

Although you should always enter module definition keywords in uppercase letters, the mixed-case and lowercase forms of these words are also reserved. For example, CONTIGUOUS, Contiguous, and contiguous are reserved even though only the uppercase form is shown below.

ALIAS	INITGLOBAL	PRELOAD
BASE	INITINSTANCE	PRIVATE
CLASS	INVALID	PROTECT
CODE	LIBRARY	PROTMODE
CONFORMING	LOADONCALL	PURE
CONSTANT	LONGNAMES	READONLY
CONTIGUOUS	MAXVAL	READWRITE
DATA	MIXED1632	REALMODE
DECORATED	MOVABLE	RESIDENT
DESCRIPTION	MOVEABLE	RESIDENTNAME
DEVICE	MULTIPLES	ROBASE
DEV386	NAME	SECTIONS
DISCARDABLE	NEWFILES	SEGMENTS
DOS4	NODATA	SHARED
DYNAMIC	NOEXPANDDOWN	SINGLE
EXECUTE	NOIOPL	STACKSIZE
EXECUTEONLY	NONAME	STUB
EXECUTE-ONLY	NONCONFORMING	SWAPPABLE
EXECUTEREAD	NONDISCARDABLE	SYSBASE
EXETYPE	NONE	TERMGLOBAL
EXPANDDOWN	NONPERMANENT	TERMINSTANCE
EXPORTS	NONSHARED	UNKNOWN
FIXED	NOTWINDOWCOMPAT	VERSION
HEAPSIZE	OBJECTS	VIRTUAL
HUGE	OLD	VIRTUAL DEVICE
IOPL	ORDER	WINDOWAPI
IMPORTS	OS2	WINDOWCOMPAT
IMPURE	PERMANENT	WINDOWS
INCLUDE	PHYSICAL DEVICE	WRITE

## Summary of module statements

You can use linker module statements as shown below to create module definition files.

The defaults for NONE|SINGLE|MULTIPLE, SHARED|NONSHARED, INITGLOBAL|INITINSTANCE, and TERMGLOBAL|TERMINSTANCE are described in the detailed description of that option.

Table 72. Summary of linker module statements

Statement	Description	Parameters
“BASE” on page 491	Set preferred loading address.	Loading address

Table 72. Summary of linker module statements (continued)

Statement	Description	Parameters
"DESCRIPTION"	Describe the executable.	Descriptive text
"EXPORTS" on page 492	Define exported functions and data.	Entry name Internal name Ordinal position DECORATED CONSTANT Parameter size
"HEAPSIZE" on page 493	Specify local heap size.	Virtual stack size Initial physical memory
"LIBRARY" on page 493	Identify output as dynamic link library (DLL).	Library name Loading address
"NAME" on page 494	Identify output as executable (EXE).	Application name Loading address
"STACKSIZE" on page 494	Specify local stack size.	Virtual stack size Initial physical memory
"STUB" on page 494	Add DOS executable file to module.	File name to add
"VERSION" on page 495	Add string to executable.	Version number to add

## BASE

Use the BASE statement to specify the preferred load address for the first load segment of the module.

### BASE statement syntax

```
►►—BASE=address—◄◄
```

This statement has the same effect as the /BASE linker option. If you specify both the statement and the option, the statement value overrides the option value.

## DESCRIPTION

Use the DESCRIPTION statement to insert the specified text into the .EXE or .DLL file that you are creating.

### DESCRIPTION statement syntax

```
►►—DESCRIPTION—'text'—◄◄
```

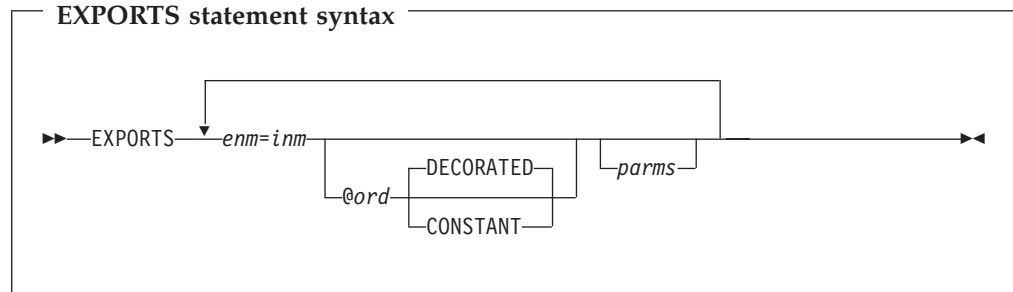
The DESCRIPTION statement is useful for embedding source control or copyright information. The inserted text must be a one-line string enclosed in single quotation marks.

In the following example, the linker inserts the text Template Program into the .EXE or .DLL file if the following line is in a .DEF file:

```
DESCRIPTION 'Template Program'
```

## EXPORTS

Use the EXPORT statement to define the names and attributes of data and functions exported from the DLL that you are creating, and of functions that run with I/O hardware privilege.



Provide export definitions for functions and data in your DLL that you want to make available to other .EXE or .DLL files. Data and functions that are not exported can only be accessed within the DLL.

The EXPORTS keyword marks the beginning of the export definitions. Enter each definition on a separate line. You can provide the following information for each export:

- enm** The entry name of the data construct or function, which is the name other files use to access it. Always provide an entry name for each export. It is strongly recommended that you decorate each entry name, to ensure that the correct function is linked in.
- inn** The internal name of the data construct or function, which is its actual name as it appears within the DLL. If specified, the internal name must be decorated. If you do not specify an internal name, the linker assumes it is the same as *enm*.
- ord** The data construct or function's ordinal position in the module definition table. If you provide the ordinal position, the data construct or function can be referenced either by its entry name or by the ordinal. It is faster to access by ordinal positions, and might save space.

You can specify only one of these two values:

### DECORATED

(Default) Indicates that the name should be left as is. No decoration is applied to the function-name.

### CONSTANT

Indicates that the export entity is a data item, not a function.

- parms** The total size of the function's parameters as measured in words (bytes divided by two). This field is required only if the function runs with I/O privilege. When a function with I/O privilege is called, the operating system consults *parms* to determine how many words to copy from the caller's stack to the stack of the I/O-privileged function.

The following example defines three exported functions:

- SampleRead
- StringIn

- CharTest

#### EXPORTS

```
SampleRead = read2bin @8
StringIn = str1 @4
CharTest 6
```

The first two functions can be accessed either by their exported names or by an ordinal number. In the module's source code, these functions are actually defined as `read2bin` and `str1`, respectively. The last function runs with I/O privilege and so has *parms* (the total size of the parameters) defined for it: 6 words.

## HEAPSIZE

Use the `HEAPSIZE` statement to define the size of the application's local heap in bytes. You can enter any positive integer for the heap size.

#### HEAPSIZE statement syntax

```
►►—HEAPSIZE—reserve—┐—————►►
 └─, commit—┘
```

#### *reserve*

Indicates the total virtual address space reserved.

#### *commit*

Sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, although execution time might be slower.

Values specified by the `/HEAP` linker option take precedence over the `HEAPSIZE` statement.

In the following example, the linker sets the local heap to 4000 bytes if the following line is in a `.DEF` file:

```
HEAPSIZE 4000
```

## LIBRARY

Use the `LIBRARY` statement to identify the output file as a DLL, and to optionally define the name, library module initialization, and library module termination.

#### LIBRARY statement syntax

```
►►—LIBRARY—libname—►►
```

You can also identify the output file as a DLL by using the `/DLL` option.

If you use the `LIBRARY` statement in your `.DEF` file, it must be the first statement, and you cannot use the `NAME` statement.

If you specify both the `BASE` parameter in the `LIBRARY` statement and the `BASE` statement, the `BASE` statement takes precedence.

The following example assigns the name `calendar` to the DLL:  
`LIBRARY calendar`

## NAME

Use the `NAME` statement to define the name of the executable file.

### NAME statement syntax

►► `NAME appname` ◄◄

You can also identify the output file as an `.EXE` file by using the `/EXEC` option.

If you use the `NAME` statement in your `.DEF` file, it must be the first statement, and you cannot use the `LIBRARY` statement.

The following example assigns the name `calendar` to the executable program:  
`NAME calendar`

## STACKSIZE

Use `STACKSIZE` to set the stack size of your program in bytes. The size must be an even number, from 0 to 0xFffffffe. If you specify an odd number, it is rounded up to the next even number.

### STACKSIZE statement syntax

►► `STACKSIZE reserve [, commit]` ◄◄

#### *reserve*

Indicates the total virtual address space reserved.

#### *commit*

Sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, although execution time might be slower.

If your program generates a stack-overflow message, use the `STACKSIZE` statement to increase the size of the stack. If your program uses the stack very little, you can save some space by decreasing the stack size.

The `STACKSIZE` statement is equivalent to the `/STACK` linker option. If you specify both the statement and the option, the statement value overrides the option value.

The following example allocates 4 KB of local stack space:  
`STACKSIZE 4096`

## STUB

Use the `STUB` statement to add a DOS `.EXE` file to the beginning of the `.EXE` or `.DLL` file that you are creating.

#### STUB statement syntax

►►—STUB—'*filename*'——◄◄

The linker searches for the file-name you specify as the stub in the following order:

1. In the directory you specify, or in the current directory if you did not specify a path
2. In the directories listed in the PATH environment variable

The stub function will be invoked whenever your .EXE or .DLL file is run under DOS. Typically, the stub displays the message that the program cannot run in DOS mode, and ends the program.

If you do not use the STUB statement, the linker adds its own standard stub for this purpose.

The following example adds the DOS .EXE file STOPIT.EXE to the beginning of the file that you are creating:

```
STUB 'STOPIT.EXE'
```

## VERSION

Use the VERSION statement to add a version number to the header of the run file.

#### VERSION statement syntax

►►—VERSION—'*file number*'——◄◄

The following example adds the text “VERSION 2.3” to the executable.

```
VERSION '2.3'
```





---

## Chapter 30. Preparing COBOL programs for multithreading

Although you cannot initiate or manage program threads in a COBOL program, you can prepare a COBOL program to run in a multithreaded environment. You can run COBOL programs in multiple threads within a process.

There is no explicit COBOL language to use for multithreaded execution; rather, you compile with the THREAD compiler option.

After you compile COBOL programs using the THREAD compiler option, other applications can call these COBOL programs in such a way that the programs run in multiple threads within a process or as multiple program invocation instances within a thread. Therefore, COBOL programs can run in multithreaded environments such as MQ applications.

“Example: using COBOL in a multithreaded environment” on page 502

### RELATED CONCEPTS

“Multithreading”

### RELATED TASKS

“Working with language elements with multithreading” on page 498

“Choosing THREAD to support multithreading” on page 500

“Transferring control to multithreaded programs” on page 501

“Ending multithreaded programs” on page 501

“Handling COBOL limitations with multithreading” on page 502

### RELATED REFERENCES

“THREAD” on page 265

PROGRAM-ID paragraph (*COBOL for Windows Language Reference*)

---

## Multithreading

To use COBOL support for multithreading, you need to understand how processes, threads, run units, and program invocation instances relate to each other.

The operating system and multithreaded applications can handle execution flow within a *process*, which is the course of events when all or part of a program runs. Programs that run within a process can share resources. Processes can be manipulated. For example, they can have a high or low priority in terms of the amount of time that the system devotes to running the process.

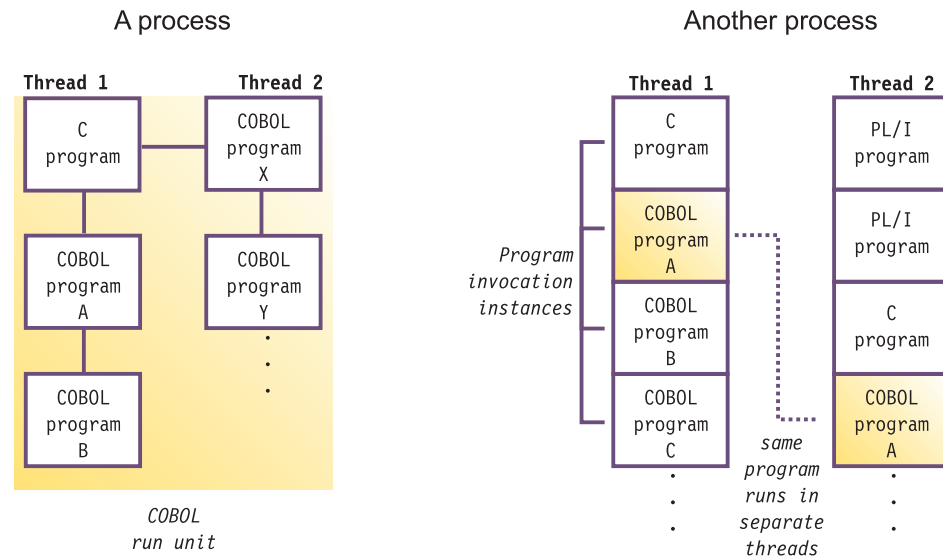
Within a process, an application can initiate one or more *threads*, each of which is a stream of computer instructions that controls that thread. A multithreaded process begins with one stream of instructions (one thread) and can later create other instruction streams to perform tasks. These multiple threads can run concurrently. Within a thread, control is transferred between executing programs.

In a multithreaded environment, a COBOL *run unit* is the portion of the process that includes threads that have actively executing COBOL programs. The COBOL run unit continues until no COBOL program is active in the execution stack for any of the threads. For example, a called COBOL program contains a GOBACK

statement and returns control to a C program. Within the run unit, COBOL programs can call non-COBOL programs, and vice versa.

Within a thread, control is transferred between separate COBOL and non-COBOL programs. For example, a COBOL program can call another COBOL program or a C program. Each separately called program is a *program invocation instance*. Program invocation instances of a particular program can exist in multiple threads within a given process.

The following illustration shows these relationships between processes, threads, run units, and program invocation instances.



Do not confuse multiprocessing or multithreading with *multitasking*, a term that is generally used to describe the external behavior of applications. In multitasking, the operating system appears to be running more than one application simultaneously. Multitasking is not related to multithreading as implemented in COBOL.

"Example: using COBOL in a multithreaded environment" on page 502

#### RELATED TASKS

"Working with language elements with multithreading"

"Choosing THREAD to support multithreading" on page 500

"Transferring control to multithreaded programs" on page 501

"Ending multithreaded programs" on page 501

"Handling COBOL limitations with multithreading" on page 502

#### RELATED REFERENCES

"THREAD" on page 265

---

## Working with language elements with multithreading

Because your COBOL programs can run as separate threads within a process, a language element can be interpreted in two different scopes: run-unit scope, or program invocation instance scope. These two types of scope are important in determining where an item can be referenced and how long the item persists in storage.

**Run-unit scope**

While the COBOL run unit runs, the language element persists and is available to other programs within the thread.

**Program invocation instance scope**

The language element persists only within a particular instance of a program invocation.

An item can be referenced from the scope in which it was declared or from its containing scope. For example, if a data item has run-unit scope, any instance of a program invocation in the run unit can reference the data item.

An item persists in storage only as long as the item in which it is declared persists. For example, if a data item has program invocation instance scope, it remains in storage only while that instance is running.

**RELATED TASKS**

“Working with elements that have run-unit scope”

“Working with elements that have program invocation instance scope”

**RELATED REFERENCES**

“Scope of COBOL language elements with multithreading” on page 500

## **Working with elements that have run-unit scope**

If you have resources that have run-unit scope (such as GLOBAL data declared in WORKING-STORAGE), you must synchronize access to that data from multiple threads by using logic in the application.

You can take one or both of the following actions:

- Structure the application so that you do not simultaneously access from multiple threads resources that have run-unit scope.
- If you are going to access resources simultaneously from separate threads, synchronize access by using facilities provided by C or by platform functions.

If you have resources that have run-unit scope and you want them to be isolated within an individual program invocation instance (for example, programs with individual copies of data), define the data in the LOCAL-STORAGE SECTION. The data will then have the scope of the program invocation instance.

## **Working with elements that have program invocation instance scope**

With these language elements, storage is allocated for each instance of a program invocation. Therefore, even if a program is called multiple times among multiple threads, each time it is called it is allocated separate storage.

For example, if program X is called in two or more threads, each instance of X that is called gets its own set of resources, such as storage.

Because the storage associated with these language elements has the scope of a program invocation instance, data is protected from access across threads. You do not have to concern yourself with synchronizing access to data. However, this data cannot be shared between invocations of programs unless it is explicitly passed.

## Scope of COBOL language elements with multithreading

The following table summarizes the scope of various COBOL language elements.

*Table 73. Scope of COBOL language elements with multithreading*

Language element	Can be referenced from:	Lifetime same as:
ADDRESS-OF special register	Same as associated record	Program invocation instance
Files	Run unit	Run unit
Index data	Program	Program invocation instance
LENGTH of special register	Same as associated identifier	Same as associated identifier
LINAGE-COUNTER special register	Run unit	Run unit
LINKAGE-SECTION data	Run unit	Based on scope of underlying data
LOCAL-STORAGE data	Within the thread	Program invocation instance
RETURN-CODE	Run unit	Program invocation instance
SORT-CONTROL, SORT-CORE-SIZE, SORT-RETURN, TALLY special registers	Run unit	Program invocation instance
WHEN-COMPILED special register	Run unit	Run unit
WORKING-STORAGE data	Run unit	Run unit

---

## Choosing THREAD to support multithreading

Use the THREAD compiler option for multithreading support. Use THREAD if your program will be called in more than one thread in a single process by an application (such as MQ applications). However, THREAD might adversely affect performance because of the serialization logic that is automatically generated.

In order to run COBOL programs in more than one thread, you must compile all of the COBOL programs in the run unit with the THREAD compiler option. You cannot mix programs compiled with THREAD and compiled with NOTHREAD in the same run unit.

Use the THREAD option when you compile object-oriented (OO) clients and classes.

You must use the THREAD option for CICS TXSeries applications.

**Language restrictions:** When you use the THREAD option, you cannot use certain language elements. For details, see the related reference below.

**Recursion:** When you compile a program with the THREAD compiler option, you can call the program recursively in a threaded or nonthreaded environment. This recursive capability is available regardless of whether you specified the RECURSIVE phrase in the PROGRAM-ID paragraph.

### RELATED TASKS

“Sharing data in recursive or multithreaded programs” on page 16

“Compiling OO applications” on page 219  
“Compiling and running CICS programs” on page 330

#### RELATED REFERENCES

“THREAD” on page 265

---

## Transferring control to multithreaded programs

When you write COBOL programs for a multithreaded environment, choose appropriate program linkage statements.

As in single-threaded environments, a called program is in its initial state when it is first called within a run unit and when it is first called after a CANCEL to the called program.

In general, it is recommended that the programs that initiate and manage multiple threads use the COBOL preinitialization interface.

If your program initiates multiple COBOL threads (for example your C program calls COBOL programs to carry out the input and output of data), do not assume that the COBOL programs will clean up their environment, such as releasing storage no longer needed. In particular, do not assume that files will be automatically closed. You should preinitialize the COBOL environment so that your application can control the COBOL cleanup.

#### RELATED CONCEPTS

Chapter 31, “Preinitializing the COBOL runtime environment,” on page 507

#### RELATED TASKS

“Ending multithreaded programs”

“Ending and reentering main programs or subprograms” on page 454

---

## Ending multithreaded programs

You can end a multithreaded program by using GOBACK or EXIT PROGRAM.

Use GOBACK to return to the caller of the program. When you use GOBACK from the first program in a thread, the thread is terminated.

Use EXIT PROGRAM as you would GOBACK, except from a main program where it has no effect.

When the COBOL environment is not preinitialized, and the COBOL run time can determine that there are no other COBOL programs active in the run unit, the COBOL process for terminating a run unit (including closing all open COBOL files) is performed upon the GOBACK from the first program of this thread. This determination can be made if all COBOL programs that are called within the run unit have returned to their callers through GOBACK or EXIT PROGRAM. This determination cannot be made under certain conditions such as the following:

- A thread that has one or more active COBOL programs was terminated (for example, because of an exception or by means of pthread\_exit).
- A longjmp was executed and resulted in collapsing active COBOL programs in the invocation stack.

There is no COBOL function that effectively does a STOP RUN in a threaded environment. If you need this behavior, consider calling the C exit function from your COBOL program and using `_iwzCOBOLTerm` after the runtime termination exit.

**RELATED TASKS**

“Ending and reentering main programs or subprograms” on page 454

---

## Handling COBOL limitations with multithreading

Some COBOL applications depend on subsystems or other applications. In a multithreaded environment, these dependencies and others result in some limitations on COBOL programs.

In general, you must synchronize access to resources that are visible to the application within a run unit. Exceptions to this requirement are `DISPLAY` and `ACCEPT`, which you can use from multiple threads; all synchronization is provided for these by the runtime environment.

**DB2:** You can run a DB2 application in multiple threads. However, you must provide any needed synchronization for accessing DB2 data.

**SORT and MERGE:** SORT and MERGE should be active in only one thread at a time. However, the COBOL runtime environment does not enforce this restriction. The application must therefore do so.

**RELATED TASKS**

“Making recursive calls” on page 466

---

## Example: using COBOL in a multithreaded environment

This multithreading example consists of a C main program and two COBOL programs.

**thrcob.c**

A C main program that creates two COBOL threads, waits for them to finish, then exits

**subd.cbl**

A COBOL program that is run by the thread created by `thrcob.c`

**sube.cbl**

A second COBOL program that is run by the thread created by `thrcob.c`

To create and run the multithreading example, enter the following commands at a command prompt:

- To compile `thrcob.c`: `cl /MT /c thrcob.c`
- To compile `subd.cbl`: `cob2 -qthread -c -dll subd.cbl`
- To compile `sube.cbl`: `cob2 -qthread -c -dll sube.cbl`
- To link the two programs in a DLL: `cob2 -qthread -dll subd.obj sube.obj`
- To link the executable `thrcob.exe`: `cob2 thrcob.obj subd.lib iwzrlibm.lib`
- To run the program `thrcob`: `thrcob`

### Source code for `thrcob.c`

```
#define LINKAGE __stdcall
#include <windows.h>
#include <stdio.h>
```

```

#include <setjmp.h>
#include <stdlib.h>
#pragma handler(SUBD)
#pragma handler(SUBE)

typedef int (LINKAGE *PRN) (long *);

long done;
jmp_buf Jmpbuf;

extern void _iwzCOBOLInit(int fcode, PFN StopFun, int *err_code, void *StopArg);
extern void _iwzCOBOLTerm(int fcode, int *err_code);

extern unsigned long LINAGE SUBD(void *);
extern unsigned long LINAGE SUBE(void *);

int LINKAGE StopFun(long *stoparg)
{
 printf("inside StopFun. Got stoparg = %d\n", *stoparg);
 *stoparg = 123;
 longjmp(Jmpbuf,1);
}

long StopArg = 0;

void LINKAGE testrc(int rc, const char *s)
{
 if (rc != 0){
 printf("%s: Fatal error rc=%d\n",s,rc);
 exit(-1);
 }
}

void LINKAGE pgmy(void)
{
 int rc;
 int parm1, parm2;

 DWORD t1, t2;
 HANDLE hThread1;
 HANDLE hThread2;

 parm1 = 20;
 parm2 = 10;
 _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
 printf("_iwzCOBOLInit got %d\n",rc);

 hThread1 = CreateThread(
 NULL, // no security attributes
 0, // use default stack size
 SUBD, // thread function
 &parm1, // argument to thread function
 0, // use default creation flags
 &t1); // returns the thread identifier

 // Check the return value for success.
 if (hThread1 == NULL)
 exit(-1);

 testrc(rc,"create 1");

 hThread1 = CreateThread(
 NULL, // no security attributes
 0, // use default stack size
 SUBE, // thread function
 &parm2, // argument to thread function
 0, // use default creation flags
 &t2);
}

```

```

 &t2);
// Check the return value for success.
if (hThread2 == NULL)
 exit(-1);

testrc(rc,"create 2");

printf("threads are %x and %x\n",t1, t2);

WaitForSingleObject(hThread2, INFINITE);

WaitForSingleObject(hThread1, INFINITE);

CloseHandle(hThread1);
CloseHandle(hThread2);

printf("test gets done = %d \n",done);
_iwzCOBOLTerm(1, &rc);
printf("_iwzCOBOLTerm expects rc=0, got rc=%d\n",rc);
}

main(char *argv,int argc)
{
 if (setjmp(Jmpbuf) ==0) (
 pgmy();
 })

```

## Source code for subd.cbl

```

PROCESS PGMNAME(MIXED)
 IDENTIFICATION DIVISION
 PROGRAM-ID. "SUBD".
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL NAMES.
 DECIMAL-POINT IS COMMA.
 INPUT-OUTPUT SECTION.
 DATA DIVISION.
 FILE SECTION.
 Working-Storage SECTION.

 Local-Storage Section.
 01 n2 pic 9(8) comp-5 value 0.

 Linkage Section.
 01 n1 pic 9(8) comp-5.

 PROCEDURE DIVISION using by Reference n1.
 Display "In SUBD "

 perform n1 times
 compute n2 = n2 + 1
 Display "From Thread 1: " n2
 CALL "Sleep" Using by value 1000
 end-perform

 GOBACK.

```

## Source code for sube.cbl

```

PROCESS PGMNAME(MIXED)
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "SUBE".
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.

```



DECIMAL-POINT IS COMMA.  
INPUT-OUTPUT SECTION.  
DATA DIVISION.  
FILE SECTION.  
Working-Storage SECTION.

Local-Storage Section.  
01 n2 pic 9(8) comp-5 value 0.

Linkage Section.  
01 n1 pic 9(8) comp-5.

PROCEDURE DIVISION using by reference n1.

    perform n1 times  
        compute n2 = n2 + 1  
        Display "From Thread 2: " n2  
\*Go to sleep for 3/4 sec.  
    CALL "Sleep" Using by value 750  
end-perform  
  
GOBACK.



---

## Chapter 31. Preinitializing the COBOL runtime environment

*Preinitialization* allows an application to 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 the 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 faster.

Creating and taking down the COBOL runtime environment repeatedly involves overhead and can slow down your application.

Use preinitialization services for multilanguage applications where non-COBOL programs need to use a COBOL program in its last-used state. For example, a file can 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 DLL 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 509

### RELATED TASKS

“Initializing persistent COBOL environment”

“Terminating preinitialized COBOL environment” on page 508

---

## Initializing persistent COBOL environment

Use the following interface to initialize a persistent COBOL environment.

### CALL *init\_routine* syntax

►►—CALL—*init\_routine*(*function\_code*,*routine*,*error\_code*,*token*)—►►

**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` (using OPTLINK linkage convention), or `_IwzCOBOLInit` (using STDCALL linkage convention)

### *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.

### *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()`. This routine is invoked with the SYSTEM linkage convention.

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"

#### RELATED REFERENCES

"Call interface conventions" on page 459

---

## Terminating preinitialized COBOL environment

Use the following interface to terminate the preinitialized persistent COBOL environment.

### CALL *term\_routine* syntax

►►CALL—*term\_routine*(*function\_code*,*error\_code*)◄◄

**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` (using OPTLINK linkage convention), or `_IwzCOBOLTerm` (using STDCALL linkage convention)

### *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 prior to the invocation of the run-unit exit routine.

**If 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”

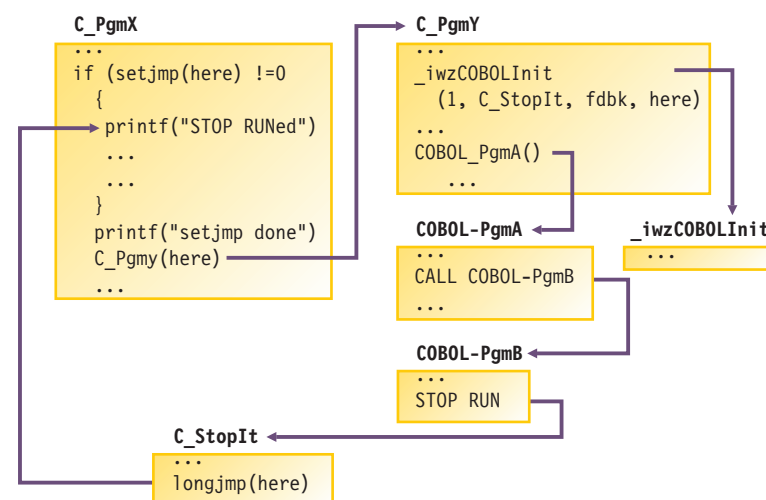
#### RELATED REFERENCES

“Call interface conventions” on page 459

---

## 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 XI0 several times. The first call to XI0 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 window:

```
cob2 -c xio.cbl
cl testinit.c xio.obj
testinit
```

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=0000000009
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 _AIX
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 XI0(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 32. 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 514

### RELATED TASKS

"Resolving date-related logic problems" on page 515

"Using year-first, year-only, and year-last date fields" on page 520

"Manipulating literals as dates" on page 523

"Performing arithmetic on date fields" on page 526

"Controlling date processing explicitly" on page 528

"Analyzing and avoiding date-related diagnostic messages" on page 530

"Avoiding problems in processing dates" on page 532

### RELATED REFERENCES

"DATEPROC" on page 237

"YEARWINDOW" on page 271

DATE FORMAT clause (*COBOL for Windows Language Reference*)

---

## Millennium language extensions (MLE)

The term *millennium language extensions* (MLE) refers to the features of COBOL for Windows 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. Add this clause to items in the DATA DIVISION to identify date fields and to specify the location of the year component within the date.

There are several restrictions on use of the DATE FORMAT clause; for example, you cannot specify it for items that have USAGE NATIONAL. See the related references below for details.

- The reinterpretation as a date field of the function return value for the following intrinsic functions:
  - DATE-OF-INTEGERS
  - DATE-TO-YYYYMMDD
  - DAY-OF-INTEGERS
  - 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”

### RELATED REFERENCES

“DATEPROC” on page 237

“YEARWINDOW” on page 271

Restrictions on using date fields (*COBOL for Windows 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 74. 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 517

“Example: internal bridging” on page 518

“Example: converting files to expanded date form” on page 519

### RELATED TASKS

“Using a century window”

“Using internal bridging” on page 517

“Moving to full field expansion” on page 518

## 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, with a century window of 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 2008, then the century window is 1958-2057; in 2009 it automatically becomes 1959-2058, 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”

#### RELATED REFERENCES

“DATEPROC” on page 237

“YEARWINDOW” on page 271

DATE FORMAT clause (*COBOL for Windows Language Reference*)

Restrictions on using date fields (*COBOL for Windows Language Reference*)

### Example: century window

The following example shows (in bold) how to modify a program with the DATE FORMAT clause to use the automatic date windowing capability.

```
CBL LIB,QUOTE,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 on 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 080102 (January 2, 2008) and Date-Returned contains 071231 (December 31, 2007), Date-Returned is less than (earlier than) Date-Due-Back, so 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”

#### RELATED TASKS

“Using a century window” on page 516

“Performing arithmetic on date fields” on page 526

“Moving to full field expansion”

### 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”

### 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 LIB,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 INFIL
 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

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.

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.



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 522

#### RELATED CONCEPTS

“Compatible dates”

#### RELATED TASKS

“Using other date formats” on page 522

## 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 XXXXY to another XXXXY date field, but not to a date field that has a format of XXXYYYY.

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” on page 523

## 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 230108 (January 23, 2008), then Next-Review-Date will contain 230109 (January 23, 2009) 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 desired 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 a level-88 condition-name associated with it, 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 081220.
```

If the century window is 1950-2049, and the contents of Date-Due are 081220 (representing December 20, 2008), then the first condition below evaluates to true, but the second condition evaluates to false:

```
If Date-Target. . .
If Date-Due = 081220
```

The literal 081220 is treated as a nondate; therefore it is windowed against the assumed century window of 1900-1999, and represents December 20, 1908. But where the literal is specified in the VALUE clause of an 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 it 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 will be 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 (081220 "YYYYXX")
If Date-Due = Function DATEVAL (20081220 "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 on 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(-68)).

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"

"Treatment of nondates" on page 525

#### RELATED TASKS

"Controlling date processing explicitly" on page 528

## 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 desired 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 528

## 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 525

### RELATED TASKS

"Controlling date processing explicitly" on page 528

### RELATED REFERENCES

"DATEPROC" on page 237

---

## 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”

“Specifying the order of evaluation” on page 528

## 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 517

## 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 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”

“Using UNDATE”

## **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 530

## **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 530

## 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 Windows 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 Windows 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 528

Analyzing date-related diagnostic messages (*COBOL Millennium Language Extensions Guide*)

#### RELATED REFERENCES

Restrictions on using date fields (*COBOL for Windows 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”

“Moving from expanded to windowed date fields” on page 533

## 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 8. Improving performance and productivity

<b>Chapter 33. Tuning your program . . . . .</b>	<b>537</b>
Using an optimal programming style . . . . .	537
Using structured programming . . . . .	538
Factoring expressions. . . . .	538
Using symbolic constants . . . . .	538
Grouping constant computations . . . . .	539
Grouping duplicate computations . . . . .	539
Choosing efficient data types . . . . .	539
Choosing efficient computational data items . . . . .	540
Using consistent data types. . . . .	540
Making arithmetic expressions efficient. . . . .	540
Making exponentiations efficient . . . . .	541
Handling tables efficiently . . . . .	541
Optimization of table references . . . . .	542
Optimization of constant and variable items . . . . .	543
Optimization of duplicate items . . . . .	543
Optimization of variable-length items . . . . .	543
Comparison of direct and relative indexing . . . . .	544
Optimizing your code . . . . .	544
Optimization . . . . .	544
Contained program procedure integration . . . . .	545
Choosing compiler features to enhance performance. . . . .	545
Performance-related compiler options . . . . .	546
Evaluating performance . . . . .	548
 <b>Chapter 34. Simplifying coding. . . . .</b>	 <b>549</b>
Eliminating repetitive coding . . . . .	549
Example: using the COPY statement. . . . .	550
Manipulating dates and times . . . . .	551
Getting feedback from date and time callable services . . . . .	551
Handling conditions from date and time callable services . . . . .	552
Example: manipulating dates . . . . .	552
Example: formatting dates for output . . . . .	552
Feedback token. . . . .	553
Picture character terms and strings . . . . .	555
Example: date-and-time picture strings . . . . .	556
Century window . . . . .	557
Example: querying and changing the century window . . . . .	557





---

## Chapter 33. Tuning your program

When a program is comprehensible, you can assess its performance. A program that has a tangled control flow is difficult to understand and maintain. The tangled control flow also inhibits the optimization of the code.

Therefore, before you try to improve the performance directly, you need to assess certain aspects of your program:

1. Examine the underlying algorithms for your program. For top performance, a sound algorithm is essential. For example, a sophisticated algorithm for sorting a million items can be hundreds of thousands times faster than a simple algorithm.
2. Look at the data structures. They should be appropriate for the algorithm. When your program frequently accesses data, reduce the number of steps needed to access the data wherever possible.
3. After you have improved the algorithm and data structures, look at other details of the COBOL source code that affect performance.

You can write programs that result in better generated code sequences and use system services better. These areas affect program performance:

- Coding techniques. These include using a programming style that helps the optimizer, choosing efficient data types, and handling tables efficiently.
- Optimization. You can optimize your code by using the OPTIMIZE compiler option.
- Compiler options and USE FOR DEBUGGING ON ALL PROCEDURES. Certain compiler options and language affect the efficiency of your program.
- Runtime environment. Carefully consider your choice of runtime options and other runtime considerations that control how your compiled program runs.
- Running under CICS. Convert instances of EXEC CICS LINK to CALL to improve transaction response time.

### RELATED CONCEPTS

"Optimization" on page 544

### RELATED TASKS

"Using an optimal programming style"

"Choosing efficient data types" on page 539

"Handling tables efficiently" on page 541

"Optimizing your code" on page 544

"Choosing compiler features to enhance performance" on page 545

### RELATED REFERENCES

"Performance-related compiler options" on page 546

Chapter 17, "Runtime options," on page 293

---

## 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"

"Factoring expressions"

"Using symbolic constants"

"Grouping constant computations" on page 539

"Grouping duplicate computations" on page 539

## 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 statement
- 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

V1 \* V2 \* V3 \* (C1 \* C2 \* C3)  
C1 + C2 + C3 + V1 + V2 + V3

### Less efficient

V1 \* V2 \* V3 \* C1 \* C2 \* C3  
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 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.

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 43

#### RELATED TASKS

“Choosing efficient computational data items”

“Using consistent data types”

“Making arithmetic expressions efficient”

“Making exponentiations efficient” on page 541

## 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.

Declare 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"

#### RELATED TASKS

"Referring to an item in a table" on page 64

"Choosing efficient data types" on page 539

#### RELATED REFERENCES

"SSRANGE" on page 263

## 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:

$$\text{comp\_s1} * \text{d1} + \text{comp\_s2} * \text{d2} + \text{comp\_s3} * \text{d3} + \text{base\_address}$$

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”

#### RELATED TASKS

“Handling tables efficiently” on page 541

---

## 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.

You might also want to use this compiler option during development if a program is used frequently without recompilation. However, the overhead for OPTIMIZE might outweigh its benefits if you recompile frequently, 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 the OPTIMIZE option and compare the generated code. (Use the LIST compiler option to request the assembler listing of the generated code.)

#### RELATED CONCEPTS

“Optimization”

#### RELATED REFERENCES

“LIST” on page 249

“OPTIMIZE” on page 254

## 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.



- Simplify the compiled code for a CALL statement to a contained (nested) program. Where possible, the optimizer places the statements inline, eliminating the need for linkage code. This optimization is known as *procedure integration*. If procedure integration cannot be done, the optimizer uses the simplest linkage possible (perhaps as few as two instructions) to get to and from the called program.
- 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.)

### Contained program procedure integration

In contained program procedure integration, the contained program code replaces a CALL to a contained program. The resulting program runs faster without the overhead of CALL linkage and with more linear control flow.

**Program size:** If several CALL statements call contained programs and these programs replace each such statement, the containing program can become large. The optimizer limits this increase to no more than 50 percent, after which it no longer integrates the programs. The optimizer then chooses the next best optimization for the CALL statement. The linkage overhead can be as few as two instructions.

**Unreachable code:** As a result of this integration, one contained program might be repeated several times. As further optimization proceeds on each copy of the program, portions might be found to be unreachable, depending on the context into which the code was copied.

#### RELATED CONCEPTS

“Optimization of table references” on page 542

#### RELATED REFERENCES

“OPTIMIZE” on page 254

---

## 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 544

#### RELATED TASKS

“Optimizing your code” on page 544

“Getting listings” on page 307

#### RELATED REFERENCES

“Performance-related compiler options”

## Performance-related compiler options

In the table below you can see a brief description of the purpose of each option, its performance advantages and disadvantages, and usage notes where applicable.

**Table 75. Performance-related compiler options**

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
ARITH(EXTEND) See “ARITH” on page 228.	To increase the maximum number of digits allowed for decimal numbers	In general, none	ARITH(EXTEND) causes some degradation in performance for all decimal data types due to larger intermediate results.	The amount of degradation that you experience depends directly on the amount of decimal data that you use.
“DYNAM” on page 238	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 254)	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) (see “OPTIMIZE” on page 254)	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.

Table 75. Performance-related compiler options (continued)

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
NOSSRANGE (see “SSRANGE” on page 263)	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.	None	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.
NOTEST (see “TEST” on page 264)	To avoid the additional object code that would be produced to take full advantage of the Debug Perspective of Rational Developer for System z.	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 in the executable, a slight performance degradation might occur because the larger executable will take longer to load and might increase paging.	None	TEST forces the NOOPTIMIZE compiler option into effect. For production runs, using NOTEST is recommended.
TRUNC(OPT) (see “TRUNC” on page 266)	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, though its performance was improved in COBOL for OS/390 & VM V2R2.	TRUNC(STD) conforms to Standard COBOL 85, 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 544

#### RELATED TASKS

“Generating a list of compiler error messages” on page 204

“Choosing compiler features to enhance performance” on page 545

“Handling tables efficiently” on page 541

#### RELATED REFERENCES

“Sign representation of zoned and packed-decimal data” on page 50

## 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 76. Performance-tuning worksheet*

Compiler option	Consideration	Yes?
DYNAM	Do you use NODYNAM? Consider the performance tradeoffs.	
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 545

#### RELATED REFERENCES

“Performance-related compiler options” on page 546

---

## Chapter 34. 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 timestamps 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 53

“Eliminating repetitive coding”

“Converting data items (intrinsic functions)” on page 104

“Evaluating data items (intrinsic functions)” on page 107

“Manipulating dates and times” on page 551

---

## Eliminating repetitive coding

Use the COPY statement in any program division and at any code sequence level to include stored source statements in a program. 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 semicolon (;) 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.

Use a command such as the following example to set an environment variable that defines COPYLIB at compile time:

```
SET COPYLIB=D:\CPYFILES\COBCOPY
```

**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 itself cannot 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”

#### RELATED REFERENCES

Chapter 15, “Compiler-directing statements,” on page 273

## 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 549

#### RELATED REFERENCES

Chapter 15, “Compiler-directing statements,” on page 273

---

## 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. Therefore, either compile the program using the CALLINT(SYSTEM) compiler option (the default) or use the >>CALLINTERFACE SYSTEM compiler-directing statement.

"Example: manipulating dates" on page 552

### RELATED CONCEPTS

Appendix E, "Date and time callable services," on page 585

### RELATED TASKS

"Getting feedback from date and time callable services"

"Handling conditions from date and time callable services" on page 552

### RELATED REFERENCES

"Feedback token" on page 553

"Picture character terms and strings" on page 555

"CALLINT" on page 229

Chapter 15, "Compiler-directing statements," on page 273

CALL statement (*COBOL for Windows 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”

#### RELATED REFERENCES

“Feedback token” on page 553

## Handling conditions from date and time callable services

Condition handling by COBOL for Windows is significantly different from that provided by IBM Language Environment® on the host. COBOL for Windows 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 553

## 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 require extensive coding using previous versions of COBOL. 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: *
```



```

* WWWWWWWW, MMMMMMM DD, YYYY *
*
* For example: MONDAY, OCTOBER 20, 2008 *
*

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 " WWWWWWWWZ, MMMMMMMZ 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 Windows 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 copybooks that are provided define the condition tokens. The file CEEIGZCT.CPY contains the definitions of the condition tokens to use if you use native data formats in your program. The file CEEIGZCT.EBC contains the copybook with the host data formats; to use that file, rename it to CEEIGZCT.CPY. These files are in the Samples\cee directory.

The condition tokens in the files are equivalent to those provided by Language Environment, except that for native data formats, the character representations are in ASCII instead of EBCDIC, and the bytes within binary fields are reversed.

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 I, "Runtime messages," on page 707

## Picture character terms and strings

You use picture strings (templates that indicate the format of the input data or the desired format of the output data) for several of the date and time callable services.

Table 77. Picture character terms and strings

Picture terms	Explanations	Valid values	Notes
Y YY YYY ZYY YYYY	One-digit year Two-digit year Three-digit year Three-digit year within era Four-digit year	0-9 00-99 000-999 1-999 1582-9999	Y valid for output only. YY assumes range set by CEEScen. YYY/ZYY used with <JJJJ>, <CCCC>, and <CCCCCCCC>.
<JJJJ>	Japanese Era name in Kanji characters with UTF-16 hexadecimal encoding	<i>Heisei</i> (NX'5E736210') <i>Showa</i> (NX'662D548C') <i>Taisho</i> (NX'59276B63') <i>Meiji</i> (NX'660E6CBB')	Affects YY field: if <JJJJ> is specified, YY means the year within Japanese Era. For example, 1988 equals Showa 63.
MM ZM	Two-digit month One- or two-digit month	01-12 1-12	For output, leading zero suppressed. For input, ZM treated as MM.
RRRR RRRZ	Roman numeral month	<i>Ibbb-XIIb</i> (left justified)	For input, source string is folded to uppercase. For output, uppercase only. I=Jan, II=Feb, ..., XII=Dec.
MMM Mmm MMMM...M Mmmm...m MMMMMMMMMZ Mmmmmmmmmz	Three-char month, uppercase Three-char month, mixed case Three-20 char mo., uppercase Three-20 char mo., mixed case Trailing blanks suppressed Trailing blanks suppressed	JAN-DEC Jan-Dec JANUARY <b>bb</b> -DECEMBER <b>b</b> January <b>bb</b> -December <b>b</b> JANUARY-DECEMBER January-December	For input, source string always folded to uppercase. For output, M generates uppercase and m generates lowercase. Output is padded with blanks ( <i>b</i> ) (unless Z specified) or truncated to match the number of Ms, up to 20.
DD ZD DDD	Two-digit day of month One- or two-digit day of mo. Day of year (Julian day)	01-31 1-31 001-366	For output, leading zero is always suppressed. For input, ZD treated as DD.
HH ZH	Two-digit hour One- or two-digit hour	00-23 0-23	For output, leading zero suppressed. For input, ZH treated as HH. If AP specified, valid values are 01-12.
MI	Minute	00-59	
SS	Second	00-59	
9 99 999	Tenths of a second Hundredths of a second Thousandths of a second	0-9 00-99 000-999	No rounding
AP ap A.P. a.p.	AM/PM indicator	AM or PM am or pm A.M. or P.M. a.m. or p.m.	AP affects HH/ZH field. For input, source string always folded to uppercase. For output, AP generates uppercase and ap generates lowercase.
W WWW Www WWW...W Www...w WWWWWWWWZ WWWWWWWWwz	One-char day-of-week Three-char day, uppercase Three-char day, mixed case Three-20 char day, uppercase Three-20 char day, mixed case Trailing blanks suppressed Trailing blanks suppressed	S, M, T, W, T, F, S SUN-SAT Sun-Sat SUNDAY <b>bbb</b> -SATURDAY <b>b</b> Sunday <b>bbb</b> -Saturday <b>b</b> SUNDAY-SATURDAY Sunday-Saturday	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.

Table 77. 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.
<b>Note:</b> Blank characters are indicated by the symbol <i>b</i> .			

The following table defines Japanese Eras used by date and time services when <JJJJ> is specified.

Table 78. 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'660E6CBB'	01-45
1912-07-30	Taisho	NX'59276B63'	01-15
1926-12-25	Showa	NX'662D548C'	01-64
1989-01-08	Heisei	NX'5E736210'	01-999 (01 = 1989)

"Example: date-and-time picture strings"

## Example: date-and-time picture strings

These are examples of picture strings that are recognized by the date and time services.

Table 79. Examples of date-and-time picture strings

Picture strings	Examples	Comments
YYMMDD YYYYMMDD	880516 19880516	
YYYY-MM-DD	1988-05-16	1988-5-16 would also be valid input.
<JJJJ> YY.MM.DD	<i>Showa</i> 63.05.16	<i>Showa</i> is a Japanese Era name. <i>Showa</i> 63 equals 1988.
MMDDYY MM/DD/YY ZM/ZD/YY MM/DD/YYYY MM/DD/Y	050688 05/06/88 5/6/88 05/06/1988 05/06/8	One-digit year format (Y) is valid for output only.
DD.MM.YY DD-RRRR-YY DD MMM YY DD Mmmmmmmmm YY ZD Mmmmmmmmmz YY Mmmmmmmmmz ZD, YYYY ZDMMMMMMmzYY	09.06.88 09-VI -88 09 JUN 88 09 June 88 9 June 88 June 9, 1988 9JUNE88	Z suppresses zeros and blanks.

Table 79. Examples of date-and-time picture strings (continued)

Picture strings	Examples	Comments
YY.DDD YYDDD YYYY/DDD	88.137 88137 1988/137	Julian date
YYMMDDHHMISS YYYYMMDDHHMISS YYYY-MM-DD HH:MI:SS.999 WWW, ZM/ZD/YY HH:MI AP Wwwwwwwwwz, DD Mmm YYYY, ZH:MI AP	880516204229 19880516204229 1988-05-16 20:42:29.046 MON, 5/16/88 08:42 PM Monday, 16 May 1988, 8:42 PM	Timestamp—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.
<b>Note:</b> 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 2008 for example, the 100 years that span from 1928 to 2027 are the default century window for the date and time callable services. In 2008, years 28 through 99 are recognized as 1928-1999, and years 00 through 27 are recognized as 2000-2027.



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 01/31/28. The two-digit year 28 would be interpreted as 1928 if the century window described above were in effect. The CEEQCEN 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”

### 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 E, "Date and time callable services," on page 585

---

## Part 9. Appendixes





---

## Appendix A. Summary of differences with host COBOL

IBM COBOL for Windows 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 26, “Porting applications between platforms,” on page 445

### RELATED REFERENCES

“Compiler options”

“Data representation”

“Environment variables” on page 563

“File specification” on page 564

“Interlanguage communication (ILC)” on page 564

“Input and output” on page 564

“Runtime options” on page 564

“Source code line size” on page 565

“Language elements” on page 565

---

## Compiler options

COBOL for Windows treats the following compiler options as comments: ADV, AWO, BUFSIZE, CODEPAGE, DATA, DECK, DBCS, FASTSRT, FLAGMIG, INTDATE, LANGUAGE, NAME, OUTDD, and RENT (the compiler always produces reentrant code). These options are flagged with I-level messages.

COBOL for Windows treats the NOADV compiler option as a comment. It is flagged with a W-level message. If you specify this option, the application might yield unexpected results when you use a WRITE statement with the ADVANCING phrase.

For OO COBOL applications, you can specify the -host option of the cob2 command, any setting of the BINARY, CHAR, or FLOAT compiler options, or both. However, you must specify any binary data items and floating-point items that are used as method arguments or parameters, and any arguments to JNI services, in native format (for example, by using the NATIVE phrase on the data description entry).

### RELATED REFERENCES

“CHAR” on page 230

---

## Data representation

The representation of data can differ between IBM host COBOL and IBM COBOL for Windows.

### Binary data

COBOL for Windows handles binary data types based on the setting of the BINARY compiler option.

Do not specify the zSeries formats of binary data 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.

## 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` phrase. COBOL for Windows processes the sign representation of zoned decimal data consistently with the processing that occurs on the host when the compiler option `NUMPROC(NOPFD)` is in effect.

## Packed-decimal data

Sign representation for unsigned packed-decimal numbers is different between COBOL for Windows and host COBOL. COBOL for Windows always uses a sign nibble of `x'C'` for unsigned packed-decimal numbers. Host 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 Windows 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($390)` compiler option to indicate that display floating-point data items are in the zSeries data representation (hexadecimal) as opposed to the native (IEEE) format.

Do not specify the zSeries 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

COBOL for Windows uses UTF-16 little-endian format for national data. Host 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 zSeries 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 Windows, 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 Windows-based 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 Windows 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 179

"Fixing differences caused by data representations" on page 447

### RELATED REFERENCES

"CHAR" on page 230

"SOSI" on page 260

---

## Environment variables

COBOL for Windows recognizes several environment variables, as listed below.

- *assignment-name*
- COBMSGS
- COBPATH
- COBRTOPT
- DB2DBDFT
- EBCDIC\_CODEPAGE
- LANG
- LC\_ALL
- LC\_COLLATE
- LC\_CTYPE
- LC\_MESSAGES
- LC\_TIME
- LIB
- LOCPATH
- NLSPATH
- TMP
- TZ

---

## File specification

COBOL for Windows treats all files as single-volume files. All other file specifications are treated as comments. This change affects the following items: REEL, UNIT, MULTIPLE FILE TAPE clause, and CLOSE. . .UNIT/REEL.

---

## Interlanguage communication (ILC)

ILC is available with C/C++ and PL/I programs.

These are the differences in ILC behavior on the Windows workstation compared to using ILC on the host with Language Environment:

- There are differences in termination behavior when a COBOL STOP RUN, a C `exit()`, or a PL/I STOP is used.
- There is no coordinated condition handling on the workstation. Avoid using a C `longjmp()` that crosses COBOL programs.
- On the host, the first program that is invoked within the process and that is enabled for Language Environment is considered to be the “main” program. On Windows, 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 31, “Preinitializing the COBOL runtime environment,” on page 507

---

## Input and output

COBOL for Windows supports input and output for sequential, relative, and indexed files by using the STL file system and Btrieve file system. Line-sequential input and output is supported by using the native byte stream file support of the platform.

Sizes and values are different for the file status *data-name* returned from the file system.

COBOL for Windows does not provide direct support for tape drives or diskette drives.

---

## Runtime options

COBOL for Windows does not recognize the following host runtime options, and treats them as not valid: AIXBLD, ALL31, CBLPSHPOP, CBLQDA, COUNTRY, HEAP, MSGFILE, NATLANG, SIMVRD, and STACK.

On the host, you can use the STORAGE runtime option to initialize COBOL WORKING-STORAGE. With COBOL for Windows, use the WSCLEAR compiler option.

### RELATED REFERENCES

“WSCLEAR” on page 269

---

## Source code line size

A COBOL source line can contain fewer than 72 characters. A line ends on column 72 or where a new-line control character is found.

---

## Language elements

The following table lists the language elements that are different between the host COBOL and COBOL for Windows compilers.

Many host COBOL clauses and phrases are syntax checked but have no effect on the execution of the program under COBOL for Windows. These clauses and phrases should have minimal effect on existing applications that you download. COBOL for Windows recognizes and processes most host COBOL language syntax even if the language element has no functional effect.

*Table 80. Language differences between Enterprise COBOL for z/OS and COBOL for Windows*

Language element	Implementation
ACCEPT statement	Under COBOL for Windows, <i>environment-name</i> and the associated environment-variable value, if set, determines file identification.
APPLY WRITE-ONLY clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
ASSIGN clause	Different syntax for <i>assignment-name</i> . ASSIGN. . . USING <i>data-name</i> is not supported under host COBOL.
BLOCK CONTAINS clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
CALL statement	A file-name as a CALL argument is not supported under COBOL for Windows.
CLOSE statement	The following phrases are syntax checked, but have no effect on the execution of the program under COBOL for Windows: FOR REMOVAL, WITH NO REWIND, and UNIT/REEL.
CODE-SET clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
DATA RECORDS clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
DISPLAY statement	Under COBOL for Windows, <i>environment-name</i> and the associated environment-variable value, if set, determines file identification.
File status <i>data-name-1</i>	Some values and meanings for file status 9x are different under host COBOL than under COBOL for Windows.
File status <i>data-name-8</i>	The format and values are different depending on the platform and the file system.
LABEL RECORD clause	LABEL RECORD IS <i>data-name</i> , USE. . . AFTER. . . LABEL PROCEDURE, and GO TO MORE-LABELS are syntax checked, but have no effect on the execution of the program under COBOL for Windows. These language elements are processed by the compiler; however, the user label declaratives are not called at run time.
MULTIPLE FILE TAPE	Syntax checked, but has no effect on the execution of the program under COBOL for Windows. On the Windows-based workstation, all files are treated as single volume files.
OPEN statement	The following phrases are syntax checked, but have no effect on the execution of the program under COBOL for Windows: REVERSED and WITH NO REWIND.
PASSWORD clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows

Table 80. Language differences between Enterprise COBOL for z/OS and COBOL for Windows (continued)

Language element	Implementation
POINTER, PROCEDURE-POINTER, and FUNCTION-POINTER data items	Under host COBOL, POINTER and FUNCTION-POINTER data items are defined as 4 bytes; a PROCEDURE-POINTER data item is defined as 8 bytes. Under COBOL for Windows, the size of these data items is 4 bytes.
READ. . .PREVIOUS	Under COBOL for Windows only, allows you to read the previous record for relative or indexed files with DYNAMIC access mode
RECORD CONTAINS clause	The RECORD CONTAINS <i>n</i> CHARACTERS clause is accepted with one exception: RECORD CONTAINS 0 CHARACTERS is syntax checked, but has no effect on the execution of the program under COBOL for Windows.
RECORDING MODE clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows 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 under COBOL for Windows
RESERVE clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
SAME AREA clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
SAME SORT clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
SORT-CONTROL special register	The 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 under COBOL for Windows. Values in this special register are not used.
SORT-MESSAGE special register	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
SORT-MODE-SIZE special register	Syntax checked, but has no effect on the execution of the program under COBOL for Windows. Values in this special register are not used.
SORT MERGE AREA clause	Syntax checked, but has no effect on the execution of the program under COBOL for Windows
START. . .	Under COBOL for Windows, the following relational operators are allowed: IS LESS THAN, IS <, IS NOT GREATER THAN, IS NOT >, IS LESS THAN OR EQUAL TO, IS <=
WRITE statement	Under COBOL for Windows, if you specify WRITE. . .ADVANCING with environment names C01 through C12 or S01 through S05, one line is advanced.
Names known to the platform environment	The following names are identified differently: <i>program-name</i> , <i>text-name</i> , <i>library-name</i> , <i>assignment-name</i> , file-name in the SORT-CONTROL special register, <i>basis-name</i> , DISPLAY or ACCEPT target identification, and system-dependent names.

---

## Appendix B. zSeries host data format considerations

The following are considerations, restrictions, and limitations that apply to the use of zSeries host data internal representation. The BINARY, CHAR, and FLOAT compiler options determine whether zSeries 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

CICS allows you to specify various data conversion choices at various places and at various granularities. For example, client CICS translator option specifications on the server for different resources (file, EIBLK, COMMAREA, transient data queue, and so forth). Your use of host or native data representation depends on such selections. Refer to the appropriate CICS documentation for specific information about how such choices can best be made.

There is no zSeries host data format support for COBOL programs that are translated by the separate or integrated CICS translator and run on TXSeries.

---

### Date and time callable services

You can use the date and time callable services with the zSeries host data format internal representations. All of the parameters passed to the callable services must be in zSeries 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 Windows workstation and the zSeries host, it is possible if FLOAT(HEX) 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 81. Maximum floating-point values*

Data type	Maximum workstation value	Maximum zSeries 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 zSeries host data format compiler options can be used with DB2 programs.

---

## Object-oriented syntax for Java interoperability

The zSeries format of binary and floating-point data items must not be specified for arguments or returning items in INVOKE statements.

---

## MQ applications

The zSeries host data format compiler options should not be used with MQ 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 COLLSEQ(EBCDIC), CHAR(EBCDIC), BINARY(S390), and FLOAT(S390) to process EBCDIC character data, big-endian binary data, and hexadecimal floating-point data that is acquired from these files.

---

## SORT

| All of the zSeries host data formats except DBCS (USAGE DISPLAY-1) can be used as sort keys.

### RELATED CONCEPTS

“Formats for numeric data” on page 43

### RELATED TASKS

“Coding interoperable data types in COBOL and Java” on page 436

### RELATED REFERENCES

Chapter 14, “Compiler options,” on page 225



---

## 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 571

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 VisualAge® COBOL:

- 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 floating-point results. (For COBOL for Windows, which runs with 80-bit extended-precision IEEE floating point, the accuracy of extended-precision floating-point results is substantially reduced as compared with the 128-bit extended-precision floating point that is used with Enterprise COBOL for z/OS.)
- 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 43

"Fixed-point contrasted with floating-point arithmetic" on page 56

#### RELATED REFERENCES

"Fixed-point data and intermediate results" on page 571

"Floating-point data and intermediate results" on page 576

"Arithmetic expressions in nonarithmetic statements" on page 577

"ARITH" on page 228

---

## 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 Windows 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 53

### RELATED REFERENCES

“Terminology used for intermediate results” on page 570

---

## 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 -	$(i1 \text{ or } i2) + 1$ , 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
		<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 $i + d_{max}$	Number of places carried for $ir$
<31 or =31	Any value	Any value	$i$ integer and $d$ decimal places
>31	< $d_{max}$ or = $d_{max}$	Any value	31- $d$ integer and $d$ decimal places
	> $d_{max}$	<31 or =31	$i$ integer and 31- $i$ decimal places
		>31	31- $d_{max}$ integer and $d_{max}$ 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 $ir$ .
=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”

#### RELATED REFERENCES

“Terminology used for intermediate results” on page 570

“Truncated intermediate results”

“Binary data and intermediate results” on page 574

“Floating-point data and intermediate results” on page 576

“Intrinsic functions evaluated in fixed-point arithmetic” on page 574

“ARITH” on page 228

SIZE ERROR phrases (*COBOL for Windows 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 570

“Fixed-point data and intermediate results” on page 571

## 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 43

**RELATED REFERENCES**

“Fixed-point data and intermediate results” on page 571

“ARITH” on page 228

## 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 571

“ARITH” on page 228

---

## 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	<i>Inner-dmax</i>	Digit precision of function result
DATE-OF-INTEG	0	8
DATE-TO-YYYYMMDD	0	8
DAY-OF-INTEG	0	7
DAY-TO-YYYYDDD	0	7
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( <i>i1 i2</i> )
ORD	n/a	3
ORD-MAX		9
ORD-MIN		9
YEAR-TO-YYYY	0	4

Function	<i>Inner-dmax</i>	Digit precision of function result
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 570

"Fixed-point data and intermediate results" on page 571

"Floating-point data and intermediate results"

"ARITH" on page 228

---

## 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 (80-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 570

"ARITH" on page 228

---

## 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,  $d_{max}$  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  $d_{max}$  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 56

#### RELATED REFERENCES

“Terminology used for intermediate results” on page 570

“Fixed-point data and intermediate results” on page 571

“Floating-point data and intermediate results” on page 576

IF statement (*COBOL for Windows Language Reference*)

EVALUATE statement (*COBOL for Windows Language Reference*)

Conditional expressions (*COBOL for Windows Language Reference*)

---

## Appendix D. Complex OCCURS DEPENDING ON

Several types of complex OCCURS DEPENDING ON (*complex ODO*) are possible. Complex ODO is supported as an extension to Standard COBOL 85.

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”

### RELATED TASKS

“Preventing index errors when changing ODO object value” on page 581

“Preventing overlay when adding elements to a variable table” on page 581

### RELATED REFERENCES

“Effects of change in ODO object value” on page 580

OCCURS DEPENDING ON clause (*COBOL for Windows 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”

“Preventing overlay when adding elements to a variable table”

## 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 579) 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 Windows 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 the following:

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:

VARY-FIELD-1(1)										
VARY-FIELD-1(2)										
VARY-FIELD-1(3)										
VARY-FIELD-2(1)										
VARY-FIELD-2(2)										
VARY-FIELD-2(3)										

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:

VARY-FIELD-1(1)										
VARY-FIELD-1(2)										
VARY-FIELD-1(3)										
VARY-FIELD-1(4)										
VARY-FIELD-2(1)										
VARY-FIELD-2(2)										
VARY-FIELD-2(3)										

Note that the fourth element of VARY-FIELD-1 did not overlay the first element of VARY-FIELD-2.





---

## Appendix E. Date and time callable services

With the date and time callable services, you can get the current local time and date in several formats and 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 08 for 2008.

*Table 82. Date and time callable services*

Callable service	Description
CEECBLDY ("CEECBLDY—convert date to COBOL integer format" on page 587)	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 ("CEEDATE—convert Lilian date to character format" on page 590)	Converts dates in the Lilian format back to character values.
CEEDATM ("CEEDATM—convert seconds to character timestamp" on page 594)	Converts number of seconds to character timestamp.
CEEDAYS ("CEEDAYS—convert date to Lilian format" on page 597)	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 ("CEEDYWK—calculate day of week from Lilian date" on page 601)	Provides day of week calculation.
CEEGMT ("CEEGMT—get current Greenwich Mean Time" on page 603)	Gets current Greenwich Mean Time (date and time).
CEEGMTO ("CEEGMTO—get offset from Greenwich Mean Time to local time" on page 605)	Gets difference between Greenwich Mean Time and local time.
CEEISEC ("CEEISEC—convert integers to seconds" on page 607)	Converts binary year, month, day, hour, second, and millisecond to a number representing the number of seconds since 00:00:00 15 October 1582.
CEELOCT ("CEELOCT—get current local date or time" on page 609)	Gets current date and time.
CEEQCEN ("CEEQCEN—query the century window" on page 611)	Queries the callable services century window.
CEESCEN ("CEESCEN—set the century window" on page 613)	Sets the callable services century window.
CEESECI ("CEESECI—convert seconds to integers" on page 615)	Converts a number representing the number of seconds since 00:00:00 15 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond.
CEESECS ("CEESECS—convert timestamp to seconds" on page 618)	Converts character timestamps (a date and time) to the number of seconds since 00:00:00 15 October 1582.

**Table 82. Date and time callable services** *(continued)*

Callable service	Description
CEEUTC (“CEEUTC—get coordinated universal time” on page 622)	Same as CEEGMT.
IGZEDT4 (“IGZEDT4—get current date” on page 622)	Returns the current date with a four-digit year in the form YYYYMMDD.

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 83. Date and time intrinsic functions**

Intrinsic function	Description
CURRENT-DATE	Current date and time and difference from Greenwich mean time
DATE-OF-INTEGER <sup>1</sup>	Standard date equivalent (YYYYMMDD) of integer date
DATE-TO-YYYYMMDD <sup>1</sup>	Standard date equivalent (YYYYMMDD) of integer date with a windowed year, according to the specified 100-year interval
DATEVAL <sup>1</sup>	Date field equivalent of integer or alphanumeric date
DAY-OF-INTEGER <sup>1</sup>	Julian date equivalent (YYYYDDD) of integer date
DAY-TO-YYYYDDD <sup>1</sup>	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 <sup>1</sup>	Nondate equivalent of integer or alphanumeric date field
YEAR-TO-YYYY <sup>1</sup>	Expanded year equivalent (YYYY) of windowed year, according to the specified 100-year interval
YEARWINDOW <sup>1</sup>	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 552

#### RELATED REFERENCES

“Feedback token” on page 553

CALL statement (*COBOL for Windows Language Reference*)

Function definitions (*COBOL for Windows Language Reference*)

---

## CEEGBLDY—convert date to COBOL integer format

CEEGBLDY converts a string that represents a date into the number of days since 31 December 1600. Use CEEGBLDY 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 CEEGBLDY syntax

```
►►CALL"CEEGBLDY"USINGinput_char_date,picture_string,►
►output_Integer_date,fc.◄◄
```

#### *input\_char\_date* (input)

A halfword length-prefixed character string that represents a date or timestamp, 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 CEEGBLDY 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*, CEEGBLDY 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*, CEEGBLDY 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 CEEGBLDY would 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 CEEGBLDY 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 CEEGBLDY 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 CEEGBLDY 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 84. 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-INTEG and DAY-OF-INTEG 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 2008, all two-digit years represent dates between 1928 and 2027, inclusive. You can change this default range by using the CEESCEN callable service.

### Example

CBL LIB

```

** **
** 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 Mmmmmmmmmmmmmz YY'
 TO Vstring-text of PICSTR.

** Call CEECBLDY to convert input date to a **
** COBOL integer date **

 CALL 'CEECBLY' USING CHRDATE, PICSTR,
 INTEGER, FC.

** If CEECBLY 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 'CEEBLY failed with msg '
 Msg-No of FC UPON CONSOLE
 STOP RUN
 END-IF.
*
 GOBACK.

```

#### RELATED REFERENCES

"Picture character terms and strings" on page 555

## 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 2008/04/23.

#### CALL CEEDATE syntax

```

►►—CALL—"CEEDATE"—USING—input_Lilian_date,—picture_string,—————►
►—output_char_date,—fc.—————►◄

```

#### *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 desired 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 85. 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**

CBL LIB

```

** **
** 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 **
** desired format of the output from CEEDATE, **
** and the picture string's length. **

MOVE 23 TO Vstring-length OF PICSTR.
MOVE 'ZD Mmmmmmmmmmmmmz 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 YYMM YY-MM YYMMDD YYYYMMDD YYYY-MM-DD YYYY-ZM-ZD <JJJJ> YY.MM.DD	98 9805 98-05 980516 19980516 1998-05-16 1998-5-16 <i>Showa</i> 63.05.16 (in a DBCS string)
148139	MM MMDD MM/DD MMDDYY MM/DD/YYYY ZM/DD/YYYY	05 0517 05/17 051798 05/17/1998 5/17/1998
148140	DD DDMM DDMMYY DD.MM.YY DD.MM.YYYY DD Mmm YYYY	18 1805 180598 18.05.98 18.05.1998 18 May 1998
148141	DDD YYDDD YY.DDD YYYY.DDD	140 98140 98.140 1998.140
148142	YY/MM/DD HH:MI:SS.99 YYYY/ZM/ZD ZH:MI AP	98/05/20 00:00:00.00 1998/5/20 0:00 AM
148143	WWW., MMM DD, YYYY Www., Mmm DD, YYYY  Wwwwwwwwww, Mmmmmmmmm DD, YYYY  Wwwwwwwwwz, Mmmmmmmmmz DD, YYYY	SAT., MAY 21, 1998 Sat., May 21, 1998  Saturday, May 21, 1998  Saturday, May 21, 1998

“Example: date-and-time picture strings” on page 556

## CEEDATM—convert seconds to character timestamp

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 timestamp such as 1988/07/26 20:37:00.

### CALL CEEDATM syntax

```
►►CALL—"CEEDATM"—USING—input_seconds,—picture_string,—►
►output_timestamp,—fc.—◄◄
```

#### *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 ( $24 \times 60 \times 60 + 01$ ). 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 desired 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 (/) appear 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 86. CEEDATM symbolic conditions

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	—	The service completed successfully.
CEE2E9	3	2505	The <i>input_seconds</i> value in a call to CEEDATM or CEESECI was not within the supported range.

Table 86. CEEDATM symbolic conditions (continued)

Symbolic feedback code	Severity	Message number	Message text
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 timestamp 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 timestamp to number of seconds.

### Example

CBL LIB

```

** **
** Function: CEEDATM - convert seconds to **
** character timestamp **
** **
** 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 timestamp 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 desired format of the output. **

 MOVE 35 TO Vstring-length OF PICSTR.
 MOVE 'ZD Mmmmmmmmmmmmmmmz YYYY at HH:MI:SS'
 TO Vstring-text OF PICSTR.

** Call CEEDATM to convert Lilian seconds to **
** a character timestamp **

 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 table shows the sample output of CEEDATM.

input_seconds	picture_string	output_timestamp
12,799,191,601.000	YYMMDD HH:MI:SS YY-MM-DD YYMMDDHHMISS YY-MM-DD HH:MI:SS  YYYY-MM-DD HH:MI:SS AP	880516 19:00:01 88-05-16 880516190001 88-05-16 19:00:01  1988-05-16 07:00:01 PM
12,799,191,661.986	DD Mmm YY DD MMM YY HH:MM  WWW, MMM DD, YYYY ZH:MI AP  Wwwwwwwwz, ZM/ZD/YY HH:MI:SS.99	16 May 88 16 MAY 88 19:01  MON, MAY 16, 1988 7:01 PM  Monday, 5/16/88 19:01:01.98
12,799,191,662.009	YYYY YY Y MM ZM RRRR MMM Mmm Mmmmmmmmm Mmmmmmmmmz DD ZD DDD HH ZH MI SS 99 999 AP WWW Www Wwwwwwww Wwwwwwwwz	1988 88 8 05 5 V MAY May May May 16 16 137 19 19 01 02 00 009 PM MON Mon Monday Monday

“Example: date-and-time picture strings” on page 556

#### RELATED REFERENCES

“Picture character terms and strings” on page 555

---

## 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

```
►CALL—"CEEDAYS"—USING—input_char_date,—picture_string,—
►output_Lilian_date,—fc.—◄
```

#### *input\_char\_date* (input)

A halfword length-prefixed character string that represents a date or timestamp, 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 87. 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.
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 2008, all two-digit years represent dates between 1928 and 2027, 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

```

CBL LIB

** **
** 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 Mmmmmmmmmmmmmmmz 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 556



## 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

```
►►CALL—"CEEDYWK"—USING—input_Lilian_date,—output_day_no,—fc.—◄◄
```

#### *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 88. CEEDYWK 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.

### Example

CBL LIB

```

**
** 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

```

►►—CALL—"CEEGMT"—USING—output_GMT_Lilian,—output_GMT_seconds,—fc.—◄◄

```

#### *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 ( $24 \times 60 \times 60 + 01$ ). 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 89. 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 timestamp.
- 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

CBL LIB

```

** **
** 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 193

## 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

```

▶▶CALL—"CEEGMTO"—USING—offset_hours,—offset_minutes,—————▶
▶—offset_seconds,—fc.—————▶▶

```

#### *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 Savings 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 90. 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 timestamp.
- 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

```

CBL LIB

** **
** 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. IGZTGMT0.

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-CBLGMT0.
 CALL 'CEEGMT0' 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 'CEEGMT0 failed with msg '

```

```

Msg-No of FC UPON CONSOLE
STOP RUN
END-IF.

GOBACK.

```

#### RELATED TASKS

“Setting environment variables” on page 193

#### RELATED REFERENCES

CEEGMT (“CEEGMT—get current Greenwich Mean Time” on page 603)

“Runtime environment variables” on page 196

---

## 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

```

▶▶CALL—"CEEISEC"—USING—input_year,—input_months,—input_day,————▶
▶—input_hours,—input_minutes,—input_seconds,—input_milliseconds,————▶
▶—output_seconds,—fc.————▶▶

```

#### *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 ( $24 \times 60 \times 60 + 01$ ). 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 91. 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**

CBL LIB

```

** **
** 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

```
►►CALL—"CEELOCT"—USING—output_Lilian,—output_seconds,—►
►output_Gregorian,—fc.—◄◄
```

#### *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 ( $24 \times 60 \times 60 + 01$ ). 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 92. 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

CBL LIB

```

**
** Function: Call CEEOCT to get current
** local time
**
** In this example, a call is made to CEEOCT
** 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
** YYYYMMDDMISS999). 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 'CEEOCT' USING LILIAN, SECONDS,
 GREGORN, FC.

** If CEEOCT runs successfully, display
** Gregorian character string
**

IF CEE000 of FC THEN
 DISPLAY 'Local Time is ' GREGORN
ELSE
 DISPLAY 'CEEOCT 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

►►CALL—"CEEQCEN"—USING—*century\_start*,—*fc*.◀◀

#### *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 2008, 80 indicates that all two-digit years lie within the 100-year window between 1928 and 2027, inclusive.

#### *fc* (output)

A 12-byte feedback code (optional) that indicates the result of this service.

Table 93. CEEQCEN symbolic conditions

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	—	The service completed successfully.

### Example

CBL LIB

```

** **
** 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

```

▶▶—CALL—"CEESCEN"—USING—century_start,—fc.—◀◀

```

#### *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 2008, therefore, all two-digit years are assumed to represent dates between 1928 and 2027, inclusive.

#### *fc* (output)

A 12-byte feedback code (optional) that indicates the result of this service.

Table 94. 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.

Table 94. CEESCEN symbolic conditions (continued)

Symbolic feedback code	Severity	Message number	Message text
CEE2F5	3	2533	The value passed to CEESCEN was not between 0 and 100.

### Example

CBL LIB

```

** **
** 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

```

▶▶—CALL—"CEESECI"—USING—input_seconds,—output_year,—output_month,—▶▶
▶—output_day,—output_hours,—output_minutes,—output_seconds,—▶▶
▶—output_milliseconds,—fc.—▶▶

```

#### *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 ( $24 \times 60 \times 60 + 01$ ). 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 95. 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**

CBL LIB

```

** **
** 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 timestamp 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 timestamp to seconds

CEESECS converts a string that represents a timestamp 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 timestamps.

### CALL CEESECS syntax

```

►►CALL—"CEESECS"—USING—input_timestamp,—picture_string,—►
►—output_seconds,—fc.—►►

```

#### *input\_timestamp* (input)

A halfword length-prefixed character string that represents a date or timestamp 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 timestamp 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 timestamp, *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 96. 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.
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.

Table 96. CEESECS 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.
CEE2ET	3	2525	CEESECS detected nonnumeric data in a numeric field, or the timestamp 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 2008, all two-digit years represent dates between 1928 and 2027, inclusive. You can change this range by using the callable service CEESSEN.

### Example

```

CBL LIB

** **
** Function: Call CEESECS to convert **
** timestamp to number of seconds **
** **
** In this example, calls are made to CEESECS **
** to convert two timestamps to the number of **
** seconds since 00:00:00 14 October 1582. **
** The Lilian seconds for the earlier **
** timestamp are then subtracted from the **
** Lilian seconds for the later timestamp **
** 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 timestamp and a picture string
** describing the format of the timestamp
** 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 timestamp
** 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 timestamp and a picture string
** describing the format of the timestamp 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 timestamp
** 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 timestamps

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 556

#### RELATED REFERENCES

“Picture character terms and strings” on page 555

---

## CEEUTC—get coordinated universal time

CEEUTC is identical to CEEGMT.

#### RELATED REFERENCES

CEEGMT (“CEEGMT—get current Greenwich Mean Time” on page 603)

---

## IGZEDT4—get current date

IGZEDT4 returns the current date with a four-digit year in the form YYYYMMDD.

#### CALL IGZEDT4 syntax

```

▶▶—CALL—"IGZEDT4"—USING—output_char_date.—————▶▶

```

#### *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

```

CBL LIB

** 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 F. XML reference material

This information describes the XML exception codes that the XML parser and the XML GENERATE statement return in special register XML-CODE. This information also documents which well-formedness constraints from the *XML specification* that the parser checks.

### RELATED REFERENCES

“XML PARSE exceptions that allow continuation”

“XML PARSE exceptions that do not allow continuation” on page 629

“XML GENERATE exceptions” on page 634

“XML conformance” on page 632

XML specification

---

### XML PARSE exceptions that allow continuation

The following table shows the exception codes that are associated with the XML EXCEPTION event and that the XML parser returns in special register XML-CODE when the parser can continue processing the XML data.

That is, the code is within one of the following ranges:

- 1-99
- 100,001-165,535
- 200,001-265,535

The table describes each exception and the actions that the parser takes when you request that it continue after the exception. Some of the descriptions use the following terms:

- *Basic document encoding*
- *Document encoding declaration*
- *External ASCII code page*
- *External EBCDIC code page*

For definitions of the terms, see the related task below about understanding the encoding of XML documents.

**Table 97. XML PARSE exceptions that allow continuation**

Code	Description	Parser action on continuation
1	The parser found an invalid character while scanning white space 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.
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.

Table 97. XML PARSE exceptions that allow continuation (continued)

Code	Description	Parser action on continuation
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.
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.

Table 97. XML PARSE exceptions that allow continuation (continued)

Code	Description	Parser action on continuation
12	A processing instruction target name was 'xml' in lowercase, uppercase, or mixed case.	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 &#xddd;).	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 &#ddd;).	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.
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 basic 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 the external EBCDIC code page.
71	The basic 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.

Table 97. **XML PARSE exceptions that allow continuation** (continued)

Code	Description	Parser action on continuation
72	The basic 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 basic 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 basic 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 basic 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 basic 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 1252 (MS Windows Latin 1).
83	The basic 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 1252 (MS Windows Latin 1).
92	The document data item was alphanumeric, but the basic document encoding was Unicode UTF-16.	The parser uses code page 1202 (Unicode UTF-16).
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

“The content of XML-CODE” on page 361

#### RELATED TASKS

“Understanding the encoding of XML documents” on page 364

“Handling exceptions that the XML parser finds” on page 366

---

## XML PARSE exceptions that do not allow continuation

With these XML exceptions, no further events are returned from the parser even if you set XML-CODE to zero and return control to the parser after processing the exception.

Control is passed to the statement that you specify in the ON EXCEPTION phrase, or to the end of the XML PARSE statement if you did not code an ON EXCEPTION phrase.

*Table 98. XML PARSE exceptions that do not allow continuation*

Code	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.
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.

**Table 98. XML PARSE exceptions that do not allow continuation** *(continued)*

Code	Description
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, <code>'_'</code> , or <code>':'</code> .
125	The first character of the first attribute name of an element was not a letter, <code>'_'</code> , or <code>':'</code> .
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than <code>'='</code> 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, <code>'_'</code> , or <code>':'</code> .
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a <code>'&gt;'</code> following the <code>'/'</code> .
133	The first character of an element end tag name was not a letter, <code>'_'</code> , or <code>':'</code> .
134	An element end tag name was not terminated by a <code>'&gt;'</code> .
135	The first character of an element name was not a letter, <code>'_'</code> , or <code>':'</code> .
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, <code>'_'</code> , or <code>':'</code> .
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 <code>'?&gt;'</code> .
141	The parser found an invalid character following <code>'&amp;'</code> in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	<code>'version'</code> in the XML declaration was not followed by <code>'='</code> .
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	<code>'encoding'</code> in the XML declaration was not followed by <code>'='</code> .
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.

**Table 98. XML PARSE exceptions that do not allow continuation** *(continued)*

Code	Description
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.
315	The basic document encoding was UTF-16 big-endian, which the parser does not support on this platform.
316	The basic 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 basic document encoding was UTF-8, which the parser does not support.
320	The document data item was national, but the basic document encoding was EBCDIC.
321	The document data item was national, but the basic document encoding was ASCII.
322	The document data item was a native alphanumeric data item, but the basic document encoding was EBCDIC.
323	The document data item was a host alphanumeric data item, but the basic document encoding was ASCII.
500-599	Internal error. Please report the error to your service representative.

#### RELATED CONCEPTS

"The content of XML-CODE" on page 361

#### RELATED TASKS

"Handling exceptions that the XML parser finds" on page 366



---

## XML conformance

The XML parser that is included in COBOL for Windows is not a conforming XML processor according to the definition in the *XML specification*. It 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, it 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-DESCRIPTOR XML event, which allows the application to perform these actions if required.

The parser optionally allows programs to 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 given 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 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](http://www.w3.org/TR/REC-xml)). All the annotations are non-normative and are shown in *italic*.

Copyright 1994-2002 W3C<sup>(R)</sup> (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](http://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](http://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 349

#### RELATED REFERENCES

Extensible Markup Language (XML)

2.8 Prolog and document type declaration (*XML specification*)

---

## XML GENERATE exceptions

One of several exception codes might be returned in special register XML-CODE 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 99. XML GENERATE exceptions

Code	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 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 name.
403	The value of an OCCURS DEPENDING ON variable exceeded 16,777,215.
410	The external code page is not supported for conversion to Unicode.
411	The external code page is not a supported single-byte EBCDIC code page.
412	The external code page is not a supported single-byte ASCII code page.
413	The document data item was alphanumeric, but the runtime locale was not consistent with the compile-time locale.
600-699	Internal error. Report the error to your service representative.

#### RELATED TASKS

“Handling errors in generating XML output” on page 384

---

## Appendix G. JNI.cpy

This listing shows the copybook JNI.cpy, which you can use to access the Java Native Interface (JNI) services from your COBOL programs.

JNI.cpy contains sample COBOL data definitions that correspond to the Java JNI types, and contains JNINativeInterface, the JNI environment structure that contains function pointers for accessing the JNI callable services.

JNI.cpy is in the include subdirectory of the COBOL install directory. (The location of the COBOL install directory is specified in the Windows environment variable RDZvrINSTDIR, where *v* is the version number and *r* is the release number of Rational Developer for System z.) JNI.cpy is analogous to the header file jni.h that C programmers use to access the JNI.

```

* COBOL declarations for Java native method interoperation *
* *
* To use the Java Native Interface callable services from a *
* COBOL program: *
* 1) Use a COPY statement to include this file into the *
* the Linkage Section of the program, e.g. *
* Linkage Section. *
* Copy JNI *
* 2) Code the following statements at the beginning of the *
* Procedure Division: *
* Set address of JNIEnv to JNIEnvPtr *
* Set address of JNINativeInterface to JNIEnv *

*
* Sample JNI type definitions in COBOL
*
*01 jboolean1 pic X.
* 88 jboolean1-true value X'01' through X'FF'.
* 88 jboolean1-false value X'00'.
*
*01 jbyte1 pic X.
*
*01 jchar1 pic N usage national.
*
*01 jshort1 pic s9(4) comp-5.
*01 jint1 pic s9(9) comp-5.
*01 jlong1 pic s9(18) comp-5.
*
*01 jfloat1 comp-1.
*01 jdouble1 comp-2.
*
*01 jobject1 object reference.
*01 jclass1 object reference.
*01 jstring1 object reference jstring.
*01 jarray1 object reference jarray.
*
*01 jbooleanArray1 object reference jbooleanArray.
*01 jbyteArray1 object reference jbyteArray.
*01 jcharArray1 object reference jcharArray.
*01 jshortArray1 object reference jshortArray.
*01 jintArray1 object reference jintArray.
*01 jlongArray1 object reference jlongArray.
*01 jfloatArray1 object reference floatArray.
*01 jdoubleArray1 object reference jdoubleArray.
*01 jobjectArray1 object reference jobjectArray.
```

```

* Possible return values for JNI functions.
01 JNI-RC pic S9(9) comp-5.
* success
 88 JNI-OK value 0.
* unknown error
 88 JNI-ERR value -1.
* thread detached from the VM
 88 JNI-EDETACHED value -2.
* JNI version error
 88 JNI-EVERSION value -3.
* not enough memory
 88 JNI-ENOMEM value -4.
* VM already created
 88 JNI-EEXIST value -5.
* invalid arguments
 88 JNI-EINVAL value -6.

* Used in ReleaseScalarArrayElements
01 releaseMode pic s9(9) comp-5.
 88 JNI-COMMIT value 1.
 88 JNI-ABORT value 2.

01 JNIenv pointer.

* JNI Native Method Interface - environment structure.
01 JNINativeInterface.
 02 pointer.
 02 pointer.
 02 pointer.
 02 pointer.
 02 GetVersion function-pointer.
 02 DefineClass function-pointer.
 02 FindClass function-pointer.
 02 FromReflectedMethod function-pointer.
 02 FromReflectedField function-pointer.
 02 ToReflectedMethod function-pointer.
 02 GetSuperclass function-pointer.
 02 IsAssignableFrom function-pointer.
 02 ToReflectedField function-pointer.
 02 Throw function-pointer.
 02 ThrowNew function-pointer.
 02 ExceptionOccurred function-pointer.
 02 ExceptionDescribe function-pointer.
 02 ExceptionClear function-pointer.
 02 FatalError function-pointer.
 02 PushLocalFrame function-pointer.
 02 PopLocalFrame function-pointer.
 02 NewGlobalRef function-pointer.
 02 DeleteGlobalRef function-pointer.
 02 DeleteLocalRef function-pointer.
 02 IsSameObject function-pointer.
 02 NewLocalRef function-pointer.
 02 EnsureLocalCapacity function-pointer.
 02 AllocObject function-pointer.
 02 NewObject function-pointer.
 02 NewObjectV function-pointer.
 02 NewObjectA function-pointer.
 02 GetObjectClass function-pointer.
 02 IsInstanceOf function-pointer.
 02 GetMethodID function-pointer.
 02 CallObjectMethod function-pointer.
 02 CallObjectMethodV function-pointer.
 02 CallObjectMethodA function-pointer.
 02 CallBooleanMethod function-pointer.
 02 CallBooleanMethodV function-pointer.

```

02 CallBooleanMethodA	function-pointer.
02 CallByteMethod	function-pointer.
02 CallByteMethodV	function-pointer.
02 CallByteMethodA	function-pointer.
02 CallCharMethod	function-pointer.
02 CallCharMethodV	function-pointer.
02 CallCharMethodA	function-pointer.
02 CallShortMethod	function-pointer.
02 CallShortMethodV	function-pointer.
02 CallShortMethodA	function-pointer.
02 CallIntMethod	function-pointer.
02 CallIntMethodV	function-pointer.
02 CallIntMethodA	function-pointer.
02 CallLongMethod	function-pointer.
02 CallLongMethodV	function-pointer.
02 CallLongMethodA	function-pointer.
02 CallFloatMethod	function-pointer.
02 CallFloatMethodV	function-pointer.
02 CallFloatMethodA	function-pointer.
02 CallDoubleMethod	function-pointer.
02 CallDoubleMethodV	function-pointer.
02 CallDoubleMethodA	function-pointer.
02 CallVoidMethod	function-pointer.
02 CallVoidMethodV	function-pointer.
02 CallVoidMethodA	function-pointer.
02 CallNonvirtualObjectMethod	function-pointer.
02 CallNonvirtualObjectMethodV	function-pointer.
02 CallNonvirtualObjectMethodA	function-pointer.
02 CallNonvirtualBooleanMethod	function-pointer.
02 CallNonvirtualBooleanMethodV	function-pointer.
02 CallNonvirtualBooleanMethodA	function-pointer.
02 CallNonvirtualByteMethod	function-pointer.
02 CallNonvirtualByteMethodV	function-pointer.
02 CallNonvirtualByteMethodA	function-pointer.
02 CallNonvirtualCharMethod	function-pointer.
02 CallNonvirtualCharMethodV	function-pointer.
02 CallNonvirtualCharMethodA	function-pointer.
02 CallNonvirtualShortMethod	function-pointer.
02 CallNonvirtualShortMethodV	function-pointer.
02 CallNonvirtualShortMethodA	function-pointer.
02 CallNonvirtualIntMethod	function-pointer.
02 CallNonvirtualIntMethodV	function-pointer.
02 CallNonvirtualIntMethodA	function-pointer.
02 CallNonvirtualLongMethod	function-pointer.
02 CallNonvirtualLongMethodV	function-pointer.
02 CallNonvirtualLongMethodA	function-pointer.
02 CallNonvirtualFloatMethod	function-pointer.
02 CallNonvirtualFloatMethodV	function-pointer.
02 CallNonvirtualFloatMethodA	function-pointer.
02 CallNonvirtualDoubleMethod	function-pointer.
02 CallNonvirtualDoubleMethodV	function-pointer.
02 CallNonvirtualDoubleMethodA	function-pointer.
02 CallNonvirtualVoidMethod	function-pointer.
02 CallNonvirtualVoidMethodV	function-pointer.
02 CallNonvirtualVoidMethodA	function-pointer.
02 GetFieldID	function-pointer.
02 GetObjectField	function-pointer.
02 GetBooleanField	function-pointer.
02 GetByteField	function-pointer.
02 GetCharField	function-pointer.
02 GetShortField	function-pointer.
02 GetIntField	function-pointer.
02 GetLongField	function-pointer.
02 GetFloatField	function-pointer.
02 GetDoubleField	function-pointer.
02 SetObjectField	function-pointer.
02 SetBooleanField	function-pointer.

02 SetByteField	function-pointer.
02 SetCharField	function-pointer.
02 SetShortField	function-pointer.
02 SetIntField	function-pointer.
02 SetLongField	function-pointer.
02 SetFloatField	function-pointer.
02 SetDoubleField	function-pointer.
02 GetStaticMethodID	function-pointer.
02 CallStaticObjectMethod	function-pointer.
02 CallStaticObjectMethodV	function-pointer.
02 CallStaticObjectMethodA	function-pointer.
02 CallStaticBooleanMethod	function-pointer.
02 CallStaticBooleanMethodV	function-pointer.
02 CallStaticBooleanMethodA	function-pointer.
02 CallStaticByteMethod	function-pointer.
02 CallStaticByteMethodV	function-pointer.
02 CallStaticByteMethodA	function-pointer.
02 CallStaticCharMethod	function-pointer.
02 CallStaticCharMethodV	function-pointer.
02 CallStaticCharMethodA	function-pointer.
02 CallStaticShortMethod	function-pointer.
02 CallStaticShortMethodV	function-pointer.
02 CallStaticShortMethodA	function-pointer.
02 CallStaticIntMethod	function-pointer.
02 CallStaticIntMethodV	function-pointer.
02 CallStaticIntMethodA	function-pointer.
02 CallStaticLongMethod	function-pointer.
02 CallStaticLongMethodV	function-pointer.
02 CallStaticLongMethodA	function-pointer.
02 CallStaticFloatMethod	function-pointer.
02 CallStaticFloatMethodV	function-pointer.
02 CallStaticFloatMethodA	function-pointer.
02 CallStaticDoubleMethod	function-pointer.
02 CallStaticDoubleMethodV	function-pointer.
02 CallStaticDoubleMethodA	function-pointer.
02 CallStaticVoidMethod	function-pointer.
02 CallStaticVoidMethodV	function-pointer.
02 CallStaticVoidMethodA	function-pointer.
02 GetStaticFieldID	function-pointer.
02 GetStaticObjectField	function-pointer.
02 GetStaticBooleanField	function-pointer.
02 GetStaticByteField	function-pointer.
02 GetStaticCharField	function-pointer.
02 GetStaticShortField	function-pointer.
02 GetStaticIntField	function-pointer.
02 GetStaticLongField	function-pointer.
02 GetStaticFloatField	function-pointer.
02 GetStaticDoubleField	function-pointer.
02 SetStaticObjectField	function-pointer.
02 SetStaticBooleanField	function-pointer.
02 SetStaticByteField	function-pointer.
02 SetStaticCharField	function-pointer.
02 SetStaticShortField	function-pointer.
02 SetStaticIntField	function-pointer.
02 SetStaticLongField	function-pointer.
02 SetStaticFloatField	function-pointer.
02 SetStaticDoubleField	function-pointer.
02 NewString	function-pointer.
02 GetStringLength	function-pointer.
02 GetStringChars	function-pointer.
02 ReleaseStringChars	function-pointer.
02 NewStringUTF	function-pointer.
02 GetStringUTFLength	function-pointer.
02 GetStringUTFChars	function-pointer.
02 ReleaseStringUTFChars	function-pointer.
02 GetArrayLength	function-pointer.
02 NewObjectArray	function-pointer.

02	GetObjectArrayElement	function-pointer.
02	SetObjectArrayElement	function-pointer.
02	NewBooleanArray	function-pointer.
02	NewByteArray	function-pointer.
02	NewCharArray	function-pointer.
02	NewShortArray	function-pointer.
02	NewIntArray	function-pointer.
02	NewLongArray	function-pointer.
02	NewFloatArray	function-pointer.
02	NewDoubleArray	function-pointer.
02	GetBooleanArrayElements	function-pointer.
02	GetByteArrayElements	function-pointer.
02	GetCharArrayElements	function-pointer.
02	GetShortArrayElements	function-pointer.
02	GetIntArrayElements	function-pointer.
02	GetLongArrayElements	function-pointer.
02	GetFloatArrayElements	function-pointer.
02	GetDoubleArrayElements	function-pointer.
02	ReleaseBooleanArrayElements	function-pointer.
02	ReleaseByteArrayElements	function-pointer.
02	ReleaseCharArrayElements	function-pointer.
02	ReleaseShortArrayElements	function-pointer.
02	ReleaseIntArrayElements	function-pointer.
02	ReleaseLongArrayElements	function-pointer.
02	ReleaseFloatArrayElements	function-pointer.
02	ReleaseDoubleArrayElements	function-pointer.
02	GetBooleanArrayRegion	function-pointer.
02	GetByteArrayRegion	function-pointer.
02	GetCharArrayRegion	function-pointer.
02	GetShortArrayRegion	function-pointer.
02	GetIntArrayRegion	function-pointer.
02	GetLongArrayRegion	function-pointer.
02	GetFloatArrayRegion	function-pointer.
02	GetDoubleArrayRegion	function-pointer.
02	SetBooleanArrayRegion	function-pointer.
02	SetByteArrayRegion	function-pointer.
02	SetCharArrayRegion	function-pointer.
02	SetShortArrayRegion	function-pointer.
02	SetIntArrayRegion	function-pointer.
02	SetLongArrayRegion	function-pointer.
02	SetFloatArrayRegion	function-pointer.
02	SetDoubleArrayRegion	function-pointer.
02	RegisterNatives	function-pointer.
02	UnregisterNatives	function-pointer.
02	MonitorEnter	function-pointer.
02	MonitorExit	function-pointer.
02	GetJavaVM	function-pointer.
02	GetStringRegion	function-pointer.
02	GetStringUTFRegion	function-pointer.
02	GetPrimitiveArrayCritical	function-pointer.
02	ReleasePrimitiveArrayCritical	function-pointer.
02	GetStringCritical	function-pointer.
02	ReleaseStringCritical	function-pointer.
02	NewWeakGlobalRef	function-pointer.
02	DeleteWeakGlobalRef	function-pointer.
02	ExceptionCheck	function-pointer.

#### RELATED TASKS

“Compiling OO applications” on page 219

“Accessing JNI services” on page 431





---

## Appendix H. COBOL SYSADATA file contents

When you use the ADATA compiler option, the compiler produces a file that contains program data. You can use this file instead of the compiler listing to extract information about the program. For example, you can extract information about the program for symbolic debugging tools or cross-reference tools.

“Example: SYSADATA” on page 643

### RELATED REFERENCES

“ADATA” on page 227

“Existing compiler options affecting the SYSADATA file”

“SYSADATA record types” on page 642

“SYSADATA record descriptions” on page 644

---

## Existing compiler options affecting the SYSADATA file

Several compiler options could affect the contents of the SYSADATA file.

### COMPILE

NOCOMPILE(W|E|S) might stop compilation prematurely, resulting in the loss of specific messages.

**EVENTS** (For compatibility) EVENTS has the same result as the ADATA option (that is, when either ADATA or EVENTS is in effect, the SYSADATA file is produced).

**EXIT** INEXIT prohibits identification of the compilation source file.

**TEST** TEST causes additional object text records to be created that also affect the contents of the SYSADATA file.

**NUM** NUM causes the compiler to use the contents of columns 1-6 in the source records for line numbering, rather than using generated sequence numbers. Any invalid (nonnumeric) or out-of-sequence numbers are replaced with a number one higher than that of the previous record.

The following SYSADATA fields contain line numbers whose contents differ depending on the NUM|NONUM setting:

Type	Field	Record
0020	AE_LINE	External Symbol record
0030	ATOK_LINE	Token record
0032	AF_STMT	Source Error record
0038	AS_STMT	Source record
0039	AS_REP_EXP_SLIN	COPY REPLACING record
0039	AS_REP_EXP_ELIN	COPY REPLACING record
0042	ASY_STMT	Symbol record
0044	AX_DEFN	Symbol Cross Reference record
0044	AX_STMT	Symbol Cross Reference record
0046	AN_STMT	Nested Program record

The Type 0038 Source record contains two fields that relate to line numbers and record numbers:

- AS\_STMT contains the compiler line number in both the NUM and NONUM cases.
- AS\_CUR\_REC# contains the physical source record number.

These two fields can always be used to correlate the compiler line numbers, used in all the above fields, with physical source record numbers.

The remaining compiler options have no direct effect on the SYSADATA file, but might trigger generation of additional error messages associated with the specific option, such as FLAGSAA, FLAGSTD, or SSRANGE.

“Example: SYSADATA” on page 643

#### RELATED REFERENCES

“SYSADATA record types”

“COMPILE” on page 235

“EXIT” on page 240

“NUMBER” on page 253

“TEST” on page 264

---

## SYSADATA record types

The SYSADATA file contains records classified into different record types. Each type of record provides information about the COBOL program being compiled.

Each record consists of two parts:

- A 12-byte header section, which has the same structure for all record types, and contains the record code that identifies the type of record
- A variable-length data section, which varies by record type

*Table 100. SYSADATA record types*

Record type	What it does
“Job identification record - X'0000'” on page 646	Provides information about the environment used to process the source data
“ADATA identification record - X'0001'” on page 647	Provides common information about the records in the SYSADATA file
“Compilation unit start/end record - X'0002'” on page 647	Marks the beginning and ending of compilation units in a source file
“Options record - X'0010'” on page 647	Describes the compiler options used for the compilation
“External symbol record - X'0020'” on page 657	Describes all external names in the program, definitions, and references
“Parse tree record - X'0024'” on page 657	Defines a node in the parse tree of the program
“Token record - X'0030'” on page 672	Defines a source token
“Source error record - X'0032'” on page 685	Describes errors in source program statements
“Source record - X'0038'” on page 686	Describes a single source line
“COPY REPLACING record - X'0039'” on page 686	Describes an instance of text replacement as a result of a match of COPY. . .REPLACING <i>operand-1</i> with text in the copybook

Table 100. **SYSADATA record types** (continued)

Record type	What it does
"Symbol record - X'0042'" on page 687	Describes a single symbol defined in the program. There is one symbol record for each symbol defined in the program.
"Symbol cross-reference record - X'0044'" on page 700	Describes references to a single symbol
"Nested program record - X'0046'" on page 701	Describes the name and nesting level of a program
"Library record - X'0060'" on page 702	Describes the library files and members used from each library
"Statistics record - X'0090'" on page 702	Describes the statistics about the compilation
"EVENTS record - X'0120'" on page 703	EVENTS records provide compatibility with COBOL/370™. The record format is identical with that in COBOL/370, with the addition of the standard ADATA header at the beginning of the record and a field indicating the length of the EVENTS record data.

"Example: SYSADATA"

## Example: SYSADATA

The following sample shows part of the listing of a COBOL program. If this COBOL program were compiled with the ADATA option, the records produced in the associated data file would be in the sequence shown in the table below.

000001	IDENTIFICATION DIVISION.	AD000020
000002	PROGRAM-ID. AD04202.	AD000030
000003	ENVIRONMENT DIVISION.	AD000040
000004	DATA DIVISION.	AD000050
000005	WORKING-STORAGE SECTION.	AD000060
000006	77 COMP3-FLD2 pic S9(3)v9.	AD000070
000007	PROCEDURE DIVISION.	AD000080
000008	STOP RUN.	

Type	Description
X'0120'	EVENTS Timestamp record
X'0120'	EVENTS Processor record
X'0120'	EVENTS File-ID record
X'0120'	EVENTS Program record
X'0001'	ADATA Identification record
X'0000'	Job Identification record
X'0010'	Options record
X'0038'	Source record for statement 1
X'0038'	Source record for statement 2
X'0038'	Source record for statement 3
X'0038'	Source record for statement 4
X'0038'	Source record for statement 5
X'0038'	Source record for statement 6
X'0038'	Source record for statement 7

Type	Description
X'0038'	Source record for statement 8
X'0020'	External Symbol record for AD04202
X'0044'	Symbol Cross Reference record for STOP
X'0044'	Symbol Cross Reference record for COMP3-FLD2
X'0044'	Symbol Cross Reference record for AD04202
X'0042'	Symbol record for AD04202
X'0042'	Symbol record for COMP3-FLD2
X'0090'	Statistics record
X'0120'	EVENTS FileEnd record

#### RELATED REFERENCES

"SYSADATA record descriptions"

---

## SYSADATA record descriptions

The formats of the records written to the associated data file are shown in the related references below.

In the fields described in each of the record types, these symbols occur:

- C** Indicates character (EBCDIC or ASCII) data
- H** Indicates 2-byte binary integer data
- F** Indicates 4-byte binary integer data
- A** Indicates 4-byte binary integer address and offset data
- X** Indicates hexadecimal (bit) data or 1-byte binary integer data

No boundary alignments are implied by any data type, and the implied lengths above might be changed by the presence of a length indicator (*Ln*). All integer data is in big-endian or little-endian format depending on the indicator bit in the header flag byte. *Big-endian* format means that bit 0 is always the most significant bit and bit *n* is the least significant bit. *Little-endian* refers to "byte-reversed" integers as seen on Intel processors.

All undefined fields and unused values are reserved.

#### RELATED REFERENCES

- "Common header section" on page 645
- "Job identification record - X'0000'" on page 646
- "ADATA identification record - X'0001'" on page 647
- "Compilation unit start/end record - X'0002'" on page 647
- "Options record - X'0010'" on page 647
- "External symbol record - X'0020'" on page 657
- "Parse tree record - X'0024'" on page 657
- "Token record - X'0030'" on page 672
- "Source error record - X'0032'" on page 685
- "Source record - X'0038'" on page 686
- "COPY REPLACING record - X'0039'" on page 686
- "Symbol record - X'0042'" on page 687
- "Symbol cross-reference record - X'0044'" on page 700

"Nested program record - X'0046'" on page 701  
 "Library record - X'0060'" on page 702  
 "Statistics record - X'0090'" on page 702  
 "EVENTS record - X'0120'" on page 703

## Common header section

The table below shows the format of the header section that is common for all record types. For MVS™ and VSE, each record is preceded by a 4-byte RDW (record-descriptor word) that is normally used only by access methods and stripped off by download utilities.

**Table 101. SYSADATA common header section**

Field	Size	Description
Language code	XL1	16 High Level Assembler 17 COBOL on all platforms 40 PL/I on supported platforms
Record type	HL2	The record type, which can be one of the following: X'0000' Job Identification record <sup>1</sup> X'0001' ADATA Identification record X'0002' Compilation unit start/end record X'0010' Options record <sup>1</sup> X'0020' External Symbol record X'0024' Parse Tree record X'0030' Token record X'0032' Source Error record X'0038' Source record X'0039' COPY REPLACING record X'0042' Symbol record X'0044' Symbol Cross-Reference record X'0046' Nested Program record X'0060' Library record X'0090' Statistics record <sup>1</sup> X'0120' EVENTS record
Associated data architecture level	XL1	3 Definition level for the header structure
Flag	XL1	.... ..1. ADATA record integers are in little-endian (Intel) format .... ....1 This record is continued in the next record 1111 11.. Reserved for future use
Associated data record edition level	XL1	Used to indicate a new format for a specific record type, usually 0

Table 101. SYSADATA common header section (continued)

Field	Size	Description
Reserved	CL4	Reserved for future use
Associated data field length	HL2	The length in bytes of the data following the header
1. When a batch compilation (sequence of programs) is run with the ADATA option, there will be multiple Job Identification, Options, and Statistics records for each compilation.		

The mapping of the 12-byte header does not include the area used for the variable-length record-descriptor word required by the access method on MVS and VSE.

## Job identification record - X'0000'

The following table shows the contents of the job identification record.

Table 102. SYSADATA job identification record

Field	Size	Description
Date	CL8	The date of the compilation in the format YYYYMMDD
Time	CL4	The time of the compilation in the format HHMM
Product number	CL8	The product number of the compiler that produced the associated data file
Product version	CL8	The version number of the product that produced the associated data file, in the form V.R.M
PTF level	CL8	The PTF level number of the product that produced the associated data file. (This field is blank if the PTF number is not available.)
System ID	CL24	The system identification of the system on which the compilation was run
Job name	CL8	The MVS job name of the compilation job
Step name	CL8	The MVS step name of the compilation step
Proc step	CL8	The MVS procedure step name of the compilation procedure
Number of input files <sup>1</sup>	HL2	The number of input files recorded in this record.  The following group of seven fields will occur <i>n</i> times depending on the value in this field.
...Input file number	HL2	The assigned sequence number of the file
...Input file name length	HL2	The length of the following input file name
...Volume serial number length	HL2	The length of the volume serial number
...Member name length	HL2	The length of the member name
...Input file name	CL( <i>n</i> )	The name of the input file for the compilation
...Volume serial number	CL( <i>n</i> )	The volume serial number of the (first) volume on which the input file resides
...Member name	CL( <i>n</i> )	Where applicable, the name of the member in the input file

Table 102. **SYSADATA job identification record** (continued)

Field	Size	Description
1. Where the number of input files would exceed the record size for the associated data file, the record is continued on the next record. The current number of input files (for that record) is stored in the record, and the record is written to the associated data file. The next record contains the rest of the input files. The count of the number of input files is a count for the current record.		

## ADATA identification record - X'0001'

The following table shows the contents of the ADATA identification record.

Table 103. **ADATA identification record**

Field	Size	Description
Time (binary)	XL8	Universal Time (UT) as a binary number of microseconds since midnight Greenwich Mean Time, with the low-order bit representing 1 microsecond. This time can be used as a time-zone-independent time stamp.  On Windows and AIX systems, only bytes 5-8 of the field are used as a fullword binary field that contains the time.
CCSID <sup>1</sup>	XL2	Coded Character Set Identifier
Character-set flags	XL1	X'80' EBCDIC (IBM-037) X'40' ASCII (IBM-1252)
Code-page name length	XL2	Length of the code-page name that follows
Code-page name	CL( <i>n</i> )	Name of the code page
1. The appropriate CCS flag will always be set. If the CCSID is set to nonzero, the code-page name length will be zero. If the CCSID is set to zero, the code-page name length will be nonzero and the code-page name will be present.		

## Compilation unit start/end record - X'0002'

The following table shows the contents of the compilation unit start/end record.

Table 104. **SYSADATA compilation unit start/end record**

Field	Size	Description
Type	HL2	Compilation unit type, which can be one of the following:  X'0000' Start compilation unit X'0001' End compilation unit
Reserved	CL2	Reserved for future use
Reserved	FL4	Reserved for future use

## Options record - X'0010'

The following table shows the contents of the options record.

Table 105. SYSADATA options record

Field	Size	Description
Option byte 0	XL1	<b>1111 1111</b> Reserved for future use
Option byte 1	XL1	<b>1... ....</b> Bit 1 = DECK, Bit 0 = NODECK  <b>.1.. ....</b> Bit 1 = ADATA, Bit 0 = NOADATA  <b>..1. ....</b> Bit 1 = COLLSEQ(EBCDIC), Bit 0 = COLLSEQ(LOCALE BINARY) (Windows and AIX only)  <b>...1 ....</b> Bit 1 = SEPOBJ, Bit 0 = NOSEPOBJ (Windows and AIX only)  <b>.... 1...</b> Bit 1 = NAME, Bit 0 = NONAME  <b>.... .1..</b> Bit 1 = OBJECT, Bit 0 = NOOBJECT  <b>.... ..1.</b> Bit 1 = SQL, Bit 0 = NOSQL  <b>.... ...1</b> Bit 1 = CICS, Bit 0 = NOCICS
Option byte 2	XL1	<b>1... ....</b> Bit 1 = OFFSET, Bit 0 = NOOFFSET (host only)  <b>.1.. ....</b> Bit 1 = MAP, Bit 0 = NOMAP  <b>..1. ....</b> Bit 1 = LIST, Bit 0 = NOLIST  <b>...1 ....</b> Bit 1 = DBCSXREF, Bit 0 = NODBCSXREF  <b>.... 1...</b> Bit 1 = XREF(SHORT), Bit 0 = not XREF(SHORT). This flag should be used in combination with the flag at bit 7. XREF(FULL) is indicated by this flag being off and the flag at bit 7 being on.  <b>.... .1..</b> Bit 1 = SOURCE, Bit 0 = NOSOURCE  <b>.... ..1.</b> Bit 1 = VBREF, Bit 0 = NOVBREF  <b>.... ...1</b> Bit 1 = XREF, Bit 0 = not XREF. See also flag at bit 4 above.



Table 105. SYSADATA options record (continued)

Field	Size	Description
Option byte 3	XL1	<p><b>1... ....</b> Bit 1 = FLAG imbedded diagnostics level specified (a value <i>y</i> is specified as in FLAG(<i>x,y</i>))</p> <p><b>.1... ....</b> Bit 1 = FLAGSTD, Bit 0 = NOFLAGSTD</p> <p><b>..1. ....</b> Bit 1 = NUM, Bit 0 = NONUM</p> <p><b>...1 ....</b> Bit 1 = SEQUENCE, Bit 0 = NOSEQUENCE</p> <p><b>.... 1...</b> Bit 1 = SOSI, Bit 0 = NOSOSI (Windows and AIX only)</p> <p><b>.... .1..</b> Bit 1 = NSYMBOL(NATIONAL), Bit 0 = NSYMBOL(DBCS)</p> <p><b>.... ..1.</b> Bit 1 = PROFILE, Bit 0 = NOPROFILE (AIX only)</p> <p><b>.... ...1</b> Bit 1 = WORD, Bit 0 = NOWORD</p>
Option byte 4	XL1	<p><b>1... ....</b> Bit 1 = ADV, Bit 0 = NOADV</p> <p><b>.1... ....</b> Bit 1 = APOST, Bit 0 = QUOTE</p> <p><b>..1. ....</b> Bit 1 = DYNAM, Bit 0 = NODYNAM</p> <p><b>...1 ....</b> Bit 1 = AWO, Bit 0 = NOAWO</p> <p><b>.... 1...</b> Bit 1 = RMODE specified, Bit 0 = RMODE(AUTO)</p> <p><b>.... .1..</b> Bit 1 = RENT, Bit 0 = NORENT</p> <p><b>.... ..1.</b> Bit 1 = RES: this flag will always be set on for COBOL.</p> <p><b>.... ...1</b> Bit 1 = RMODE(24), Bit 0 = RMODE(ANY)</p>

Table 105. SYSADATA options record (continued)

Field	Size	Description
Option byte 5	XL1	<p><b>1... ....</b> Reserved for compatibility</p> <p><b>.1.. ....</b> Bit 1 = OPT, Bit 0 = NOOPT</p> <p><b>..1. ....</b> Bit 1 = LIB, Bit 0 = NOLIB</p> <p><b>...1 ....</b> Bit 1 = DBCS, Bit 0 = NODBCS</p> <p><b>.... 1...</b> Bit 1 = OPT(FULL), Bit 0 = not OPT(FULL)</p> <p><b>.... .1..</b> Bit 1 = SSRANGE, Bit 0 = NOSSRANGE</p> <p><b>.... ..1.</b> Bit 1 = TEST, Bit 0 = NOTEST</p> <p><b>.... ...1</b> Bit 1 = PROBE, Bit 0 = NOPROBE (Windows only)</p>
Option byte 6	XL1	<p><b>..1. ....</b> Bit 1 = NUMPROC(PFD), Bit 0 = NUMPROC(NOPFD)</p> <p><b>...1 ....</b> Bit 1 = NUMCLS(ALT), Bit 0 = NUMCLS(PRIM)</p> <p><b>.... .1..</b> Bit 1 = BINARY(S390), Bit 0 = BINARY(NATIVE) (Windows and AIX only)</p> <p><b>.... ..1.</b> Bit 1 = TRUNC(STD), Bit 0 = TRUNC(OPT)</p> <p><b>.... ...1</b> Bit 1 = ZWB, Bit 0 = NOZWB</p> <p><b>11.. 1...</b> Reserved for future use</p>
Option byte 7	XL1	<p><b>1... ....</b> Bit 1 = ALLOWCBL, Bit 0 = NOALLOWCBL</p> <p><b>.1.. ....</b> Bit 1 = TERM, Bit 0 = NOTERM</p> <p><b>..1. ....</b> Bit 1 = DUMP, Bit 0 = NODUMP</p> <p><b>...1 11..</b> Reserved</p> <p><b>.... ..1.</b> Bit 1 = CURRENCY, Bit 0 = NOCURRENCY</p> <p><b>.... ...1</b> Reserved</p>
Option byte 8	XL1	<p><b>1111 1111</b> Reserved for future use</p>

Table 105. SYSADATA options record (continued)

Field	Size	Description
Option byte 9	XL1	<p><b>1... ....</b> Bit 1 = DATA(24), Bit 0 = DATA(31)</p> <p><b>.1... ....</b> Bit 1 = FASTSRT, Bit 0 = NOFASTSRT</p> <p><b>..1. ....</b> Bit 1 = SIZE(MAX), Bit 0 = SIZE(<i>nnnn</i>) or SIZE(<i>nnnn</i>K)</p> <p><b>.... .1..</b> Bit 1 = THREAD, Bit 0 = NOTHREAD</p> <p><b>...1 1.11</b> Reserved for future use</p>
Option byte A	XL1	<p><b>1111 1111</b> Reserved for future use</p>
Option byte B	XL1	<p><b>1111 1111</b> Reserved for future use</p>
Option byte C	XL1	<p><b>1... ....</b> Bit 1 = NCOLLSEQ(LOCALE) (Windows and AIX only)</p> <p><b>.1... ....</b> Reserved for future use</p> <p><b>..1. ....</b> Bit 1 = INTDATE(LILIAN), Bit 0 = INTDATE(ANSI)</p> <p><b>...1 ....</b> Bit 1 = NCOLLSEQ(BINARY) (Windows and AIX only)</p> <p><b>.... 1...</b> Bit 1 = CHAR(EBCDIC), Bit 0 = CHAR(NATIVE) (Windows and AIX only)</p> <p><b>.... .1..</b> Bit 1 = FLOAT(HEX), Bit 0 = FLOAT(NATIVE) (Windows and AIX only)</p> <p><b>.... ..1.</b> Bit 1 = COLLSEQ(BINARY) (Windows and AIX only)</p> <p><b>.... ...1</b> Bit 1 = COLLSEQ(LOCALE) (Windows and AIX only)</p>

Table 105. SYSADATA options record (continued)

Field	Size	Description
Option byte D	XL1	<p><b>1... ....</b> Bit 1 = DLL Bit 0 = NODLL (host only)</p> <p><b>.1.. ....</b> Bit 1 = EXPORTALL, Bit 0 = NOEXPORTALL (host only)</p> <p><b>..1. ....</b> Bit 1 = CODEPAGE (host only)</p> <p><b>...1 ....</b> Bit 1 = DATEPROC, Bit 0 = NODATEPROC</p> <p><b>.... 1...</b> Bit 1 = DATEPROC(FLAG), Bit 0 = DATEPROC(NOFLAG)</p> <p><b>.... .1..</b> Bit 1 = YEARWINDOW</p> <p><b>.... ..1.</b> Bit 1 = WSCLEAR, Bit 0 = NOWSCLEAR (Windows and AIX only)</p> <p><b>.... ...1</b> Bit 1 = BEOPT, Bit 0 = NOBEOPT (Windows and AIX only)</p>
Option byte E	XL1	<p><b>1... ....</b> Bit 1 = DATEPROC(TRIG), Bit 0 = DATEPROC(NOTRIG)</p> <p><b>.1.. ....</b> Bit 1 = DIAGTRUNC, Bit 0 = NODIAGTRUNC</p> <p><b>.... .1..</b> Bit 1 = LSTFILE(UTF-8), Bit 0 = LSTFILE(LOCALE) (Windows and AIX only)</p> <p><b>..11 1.11</b> Reserved for future use</p>
Option byte F	XL1	<p><b>1111 1111</b> Reserved for future use</p>
Flag level	XL1	<p><b>X'00'</b> Flag(I)</p> <p><b>X'04'</b> Flag(W)</p> <p><b>X'08'</b> Flag(E)</p> <p><b>X'0C'</b> Flag(S)</p> <p><b>X'10'</b> Flag(U)</p> <p><b>X'FF'</b> Noflag</p>

Table 105. SYSADATA options record (continued)

Field	Size	Description
Imbedded diagnostic level	XL1	<b>X'00'</b> Flag(I) <b>X'04'</b> Flag(W) <b>X'08'</b> Flag(E) <b>X'0C'</b> Flag(S) <b>X'10'</b> Flag(U) <b>X'FF'</b> Noflag
FLAGSTD (FIPS) specification	XL1	<b>1... ....</b> Minimum <b>.1.. ....</b> Intermediate <b>..1. ....</b> High <b>...1 ....</b> IBM extensions <b>.... 1...</b> Level-1 segmentation <b>.... .1..</b> Level-2 segmentation <b>.... ..1.</b> Debugging <b>.... ...1</b> Obsolete
Reserved for flagging	XL1	<b>1111 1111</b> Reserved for future use
Compiler mode	XL1	<b>X'00'</b> Unconditional Nocompile, Nocompile(I) <b>X'04'</b> Nocompile(W) <b>X'08'</b> Nocompile(E) <b>X'0C'</b> Nocompile(S) <b>X'FF'</b> Compile
Space value	CL1	
Data for 3-valued options	XL1	<b>1... ....</b> NAME(ALIAS) specified <b>.1.. ....</b> NUMPROC(MIG) specified <b>..1. ....</b> TRUNC(BIN) specified <b>...1 1111</b> Reserved for future use

Table 105. **SYSADATA options record** (continued)

Field	Size	Description
TEST(hook,sym,sep) suboptions (host only)	XL1	<b>1... ....</b> TEST(ALL,x)  <b>.1.. ....</b> TEST(NONE,x)  <b>..1. ....</b> TEST(STMT,x)  <b>...1 ....</b> TEST(PATH,x)  <b>.... 1...</b> TEST(BLOCK,x)  <b>.... .1..</b> TEST(x,SYM)  <b>.... ..1.</b> Bit 1 = SEPARATE, Bit 0 = NOSEPARATE  <b>.... ...1</b> Reserved for TEST suboptions
OUTDD name length	HL2	Length of OUTDD name
RWT ID Length	HL2	Length of Reserved Word Table identifier
LVLINFO	CL4	User-specified LVLINFO data
PGMNAME suboptions	XL1	<b>1... ....</b> Bit 1 = PGMNAME(COMPAT)  <b>.1.. ....</b> Bit 1 = PGMNAME(LONGUPPER)  <b>..1. ....</b> Bit 1 = PGMNAME(LONGMIXED)  <b>...1 1111</b> Reserved for future use
Entry interface suboptions	XL1	<b>1... ....</b> Bit 1 = EntryInterface(System) (Windows only)  <b>.1.. ....</b> Bit 1 = EntryInterface(OptLink) (Windows only)  <b>..11 1111</b> Reserved for future use

Table 105. **SYSADATA options record** (continued)

Field	Size	Description
CallInterface suboptions	XL1	<b>1... ....</b> Bit 1 = CallInterface(System) (Windows and AIX only)  <b>.1... ....</b> Bit 1 = CallInterface(OptLink) (Windows only)  <b>...1 ....</b> Bit 1 = CallInterface(Cdecl) (Windows only)  <b>.... 1...</b> Bit 1 = CallInterface(System(Desc)) (Windows and AIX only)  <b>..1. .111</b> Reserved for future use
ARITH suboption	XL1	<b>1... ....</b> Bit 1 = ARITH(COMPAT)  <b>.1... ....</b> Bit 1 = ARITH(EXTEND)  <b>11 1111</b> Reserved for future use
DBCS Req	FL4	DBCS XREF storage requirement
DBCS ORDPGM length	HL2	Length of name of DBCS Ordering Program
DBCS ENCTBL length	HL2	Length of name of DBCS Encode Table
DBCS ORD TYPE	CL2	DBCS Ordering type
Reserved	CL6	Reserved for future use
Converted SO	CL1	Converted SO hexadecimal value
Converted SI	CL1	Converted SI hexadecimal value
Language id	CL2	This field holds the two-character abbreviation (one of EN, UE, JA, or JP) from the LANGUAGE option.
Reserved	CL8	Reserved for future use
INEXIT name length	HL2	Length of SYSIN user-exit name
PRTEXT name length	HL2	Length of SYSPRINT user-exit name
LIBEXIT name length	HL2	Length of 'Library' user-exit name
ADEXIT name length	HL2	Length of ADATA user-exit name
CURROPT	CL5	CURRENCY option value
Reserved	CL1	Reserved for future use
YEARWINDOW	HL2	YEARWINDOW option value
CODEPAGE	HL2	CODEPAGE CCSID option value
Reserved	CL50	Reserved for future use
LINECNT	HL2	LINECOUNT value
Reserved	CL2	Reserved for future use
BUFSIZE	FL4	BUFSIZE option value
Size value	FL4	SIZE option value

Table 105. **SYSADATA options record** (continued)

Field	Size	Description
Reserved	FL4	Reserved for future use
Phase residence bits byte 1	XL1	<p><b>1...</b> .... Bit 1 = IGYCLIBR in user region</p> <p><b>.1..</b> .... Bit 1 = IGYCSCAN in user region</p> <p><b>..1.</b> .... Bit 1 = IGYCDSCN in user region</p> <p><b>...1</b> .... Bit 1 = IGYCGROU in user region</p> <p><b>.... 1...</b> Bit 1 = IGYCPSCN in user region</p> <p><b>.... .1..</b> Bit 1 = IGYCPANA in user region</p> <p><b>.... ..1.</b> Bit 1 = IGYCFGEN in user region</p> <p><b>.... ...1</b> Bit 1 = IGYCPGEN in user region</p>
Phase residence bits byte 2	XL1	<p><b>1...</b> .... Bit 1 = IGYCOPTM in user region</p> <p><b>.1..</b> .... Bit 1 = IGYCLSTR in user region</p> <p><b>..1.</b> .... Bit 1 = IGYCXREF in user region</p> <p><b>...1</b> .... Bit 1 = IGYCDMAP in user region</p> <p><b>.... 1...</b> Bit 1 = IGYCASM1 in user region</p> <p><b>.... .1..</b> Bit 1 = IGYCASM2 in user region</p> <p><b>.... ..1.</b> Bit 1 = IGYCDIAG in user region</p> <p><b>.... ...1</b> Reserved for future use</p>
Phase residence bits bytes 3 and 4	XL2	Reserved
Reserved	CL8	Reserved for future use
OUTDD name	CL( <i>n</i> )	OUTDD name
RWT	CL( <i>n</i> )	Reserved word table identifier
DBCS ORDPGM	CL( <i>n</i> )	DBCS Ordering program name
DBCS ENCTBL	CL( <i>n</i> )	DBCS Encode table name
INEXIT name	CL( <i>n</i> )	SYSIN user-exit name
PRTEXIT name	CL( <i>n</i> )	SYSPRINT user-exit name
LIBEXIT name	CL( <i>n</i> )	'Library' user-exit name



Table 105. SYSADATA options record (continued)

Field	Size	Description
ADEXIT name	CL( <i>n</i> )	ADATA user-exit name

## External symbol record - X'0020'

The following table shows the contents of the external symbol record.

Table 106. SYSADATA external symbol record

Field	Size	Description
Section type	XL1	<b>X'00'</b> PROGRAM-ID name (main entry point name)
		<b>X'01'</b> ENTRY name (secondary entry point name)
		<b>X'02'</b> External reference (referenced external entry point)
		<b>X'04'</b> N/A for COBOL
		<b>X'05'</b> N/A for COBOL
		<b>X'06'</b> N/A for COBOL
		<b>X'0A'</b> N/A for COBOL
		<b>X'12'</b> Internal reference (referenced internal subprogram)
		<b>X'C0'</b> External class-name (OO COBOL class definition)
		<b>X'C1'</b> METHOD-ID name (OO COBOL method definition)
		<b>X'C6'</b> Method reference (OO COBOL method reference)
		<b>X'FF'</b> N/A for COBOL
Types X'12', X'C0', X'C1' and X'C6' are for COBOL only.		
Flags	XL1	N/A for COBOL
Reserved	HL2	Reserved for future use
Symbol-ID	FL4	Symbol-ID of program that contains the reference (only for types x'02' and x'12')
Line number	FL4	Line number of statement that contains the reference (only for types x'02' and x'12')
Section length	FL4	N/A for COBOL
LD ID	FL4	N/A for COBOL
Reserved	CL8	Reserved for future use
External name length	HL2	Number of characters in the external name
Alias name length	HL2	N/A for COBOL
External name	CL( <i>n</i> )	The external name
Alias section name	CL( <i>n</i> )	N/A for COBOL

## Parse tree record - X'0024'

The following table shows the contents of the parse tree record.

Table 107. SYSADATA parse tree record

Field	Size	Description
Node number	FL4	The node number generated by the compiler, starting at 1
Node type	HL2	The type of the node: <b>001</b> Program <b>002</b> Class <b>003</b> Method
		<b>101</b> Identification Division <b>102</b> Environment Division <b>103</b> Data Division <b>104</b> Procedure Division <b>105</b> End Program/Method/Class
		<b>201</b> Declaratives body <b>202</b> Nondeclaratives body
		<b>301</b> Section <b>302</b> Procedure section
		<b>401</b> Paragraph <b>402</b> Procedure paragraph
		<b>501</b> Sentence <b>502</b> File definition <b>503</b> Sort file definition <b>504</b> Program-name <b>505</b> Program attribute <b>508</b> ENVIRONMENT DIVISION clause <b>509</b> CLASS attribute <b>510</b> METHOD attribute <b>511</b> USE statement
		<b>601</b> Statement <b>602</b> Data description clause <b>603</b> Data entry <b>604</b> File description clause <b>605</b> Data entry name <b>606</b> Data entry level <b>607</b> EXEC entry

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		<b>701</b> EVALUATE subject phrase <b>702</b> EVALUATE WHEN phrase <b>703</b> EVALUATE WHEN OTHER phrase <b>704</b> SEARCH WHEN phrase <b>705</b> INSPECT CONVERTING phrase <b>706</b> INSPECT REPLACING phrase <b>707</b> INSPECT TALLYING phrase <b>708</b> PERFORM UNTIL phrase <b>709</b> PERFORM VARYING phrase <b>710</b> PERFORM AFTER phrase <b>711</b> Statement block <b>712</b> Scope terminator <b>713</b> INITIALIZE REPLACING phrase <b>714</b> EXEC CICS Command <b>720</b> DATA DIVISION phrase
		<b>801</b> Phrase <b>802</b> ON phrase <b>803</b> NOT phrase <b>804</b> THEN phrase <b>805</b> ELSE phrase <b>806</b> Condition <b>807</b> Expression <b>808</b> Relative indexing <b>809</b> EXEC CICS Option <b>810</b> Reserved word <b>811</b> INITIALIZE REPLACING category

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		<b>901</b> Section or paragraph name <b>902</b> Identifier <b>903</b> Alphabet-name <b>904</b> Class-name <b>905</b> Condition-name <b>906</b> File-name <b>907</b> Index-name <b>908</b> Mnemonic-name <b>910</b> Symbolic-character <b>911</b> Literal <b>912</b> Function identifier <b>913</b> Data-name <b>914</b> Special register <b>915</b> Procedure reference <b>916</b> Arithmetic operator <b>917</b> All Procedures <b>918</b> INITIALIZE literal (no tokens) <b>919</b> ALL literal or figcon <b>920</b> Keyword class test name <b>921</b> Reserved word at identifier level <b>922</b> Unary operator <b>923</b> Relational operator
		<b>1001</b> Subscript <b>1002</b> Reference modification
Node subtype	HL2	The subtype of the node.  For Section type: <b>0001</b> CONFIGURATION Section <b>0002</b> INPUT-OUTPUT Section <b>0003</b> FILE Section <b>0004</b> WORKING-STORAGE Section <b>0005</b> LINKAGE Section <b>0006</b> LOCAL-STORAGE Section <b>0007</b> REPOSITORY Section

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For Paragraph type:
		0001 PROGRAM-ID paragraph
		0002 AUTHOR paragraph
		0003 INSTALLATION paragraph
		0004 DATE-WRITTEN paragraph
		0005 SECURITY paragraph
		0006 SOURCE-COMPUTER paragraph
		0007 OBJECT-COMPUTER paragraph
		0008 SPECIAL-NAMES paragraph
		0009 FILE-CONTROL paragraph
		0010 I-O-CONTROL paragraph
		0011 DATE-COMPILED paragraph
		0012 CLASS-ID paragraph
		0013 METHOD-ID paragraph
		0014 REPOSITORY paragraph
		For Environment Division clause type:
		0001 WITH DEBUGGING MODE
		0002 MEMORY-SIZE
		0003 SEGMENT-LIMIT
		0004 CURRENCY-SIGN
		0005 DECIMAL POINT
		0006 PROGRAM COLLATING SEQUENCE
		0007 ALPHABET
		0008 SYMBOLIC-CHARACTER
		0009 CLASS
		0010 ENVIRONMENT NAME
		0011 SELECT

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For Data description clause type:
		<b>0001</b> BLANK WHEN ZERO
		<b>0002</b> DATA-NAME OR FILLER
		<b>0003</b> JUSTIFIED
		<b>0004</b> OCCURS
		<b>0005</b> PICTURE
		<b>0006</b> REDEFINES
		<b>0007</b> RENAMES
		<b>0008</b> SIGN
		<b>0009</b> SYNCHRONIZED
		<b>0010</b> USAGE
		<b>0011</b> VALUE
		<b>0023</b> GLOBAL
		<b>0024</b> EXTERNAL

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For File description clause type:
		0001 FILE STATUS
		0002 ORGANIZATION
		0003 ACCESS MODE
		0004 RECORD KEY
		0005 ASSIGN
		0006 RELATIVE KEY
		0007 PASSWORD
		0008 PROCESSING MODE
		0009 RECORD DELIMITER
		0010 PADDING CHARACTER
		0011 BLOCK CONTAINS
		0012 RECORD CONTAINS
		0013 LABEL RECORDS
		0014 VALUE OF
		0015 DATA RECORDS
		0016 LINAGE
		0017 ALTERNATE KEY
		0018 LINES AT TOP
		0019 LINES AT BOTTOM
		0020 CODE-SET
		0021 RECORDING MODE
		0022 RESERVE
		0023 GLOBAL
		0024 EXTERNAL
		0025 LOCK

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For Statement type:
		0002 NEXT SENTENCE
		0003 ACCEPT
		0004 ADD
		0005 ALTER
		0006 CALL
		0007 CANCEL
		0008 CLOSE
		0009 COMPUTE
		0010 CONTINUE
		0011 DELETE
		0012 DISPLAY
		0013 DIVIDE (INTO)
		0113 DIVIDE (BY)
		0014 ENTER
		0015 ENTRY
		0016 EVALUATE
		0017 EXIT
		0018 GO
		0019 GOBACK
		0020 IF
		0021 INITIALIZE
		0022 INSPECT



Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		0023 INVOKE
		0024 MERGE
		0025 MOVE
		0026 MULTIPLY
		0027 OPEN
		0028 PERFORM
		0029 READ
		0030 READY
		0031 RELEASE
		0032 RESET
		0033 RETURN
		0034 REWRITE
		0035 SEARCH
		0036 SERVICE
		0037 SET
		0038 SORT
		0039 START
		0040 STOP
		0041 STRING
		0042 SUBTRACT
		0043 UNSTRING
		0044 EXEC SQL
		0144 EXEC CICS
		0045 WRITE
		0046 XML

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For Phrase type:
		0001 INTO
		0002 DELIMITED
		0003 INITIALIZE. .REPLACING
		0004 INSPECT. .ALL
		0005 INSPECT. .LEADING
		0006 SET. .TO
		0007 SET. .UP
		0008 SET. .DOWN
		0009 PERFORM. .TIMES
		0010 DIVIDE. .REMAINDER
		0011 INSPECT. .FIRST
		0012 SEARCH. .VARYING
		0013 MORE-LABELS
		0014 SEARCH ALL
		0015 SEARCH. .AT END
		0016 SEARCH. .TEST INDEX
		0017 GLOBAL
		0018 LABEL
		0019 DEBUGGING
		0020 SEQUENCE
		0021 Reserved for future use
		0022 Reserved for future use
		0023 Reserved for future use
		0024 TALLYING
		0025 Reserved for future use
		0026 ON SIZE ERROR
		0027 ON OVERFLOW
		0028 ON ERROR
		0029 AT END
		0030 INVALID KEY

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		0031 END-OF-PAGE
		0032 USING
		0033 BEFORE
		0034 AFTER
		0035 EXCEPTION
		0036 CORRESPONDING
		0037 Reserved for future use
		0038 RETURNING
		0039 GIVING
		0040 THROUGH
		0041 KEY
		0042 DELIMITER
		0043 POINTER
		0044 COUNT
		0045 METHOD
		0046 PROGRAM
		0047 INPUT
		0048 OUTPUT
		0049 I-O
		0050 EXTEND
		0051 RELOAD
		0052 ASCENDING
		0053 DESCENDING
		0054 DUPLICATES
		0055 NATIVE (USAGE)
		0056 INDEXED
		0057 FROM
		0058 FOOTING
		0059 LINES AT BOTTOM
		0060 LINES AT TOP

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For Function identifier type:
		0001 COS
		0002 LOG
		0003 MAX
		0004 MIN
		0005 MOD
		0006 ORD
		0007 REM
		0008 SIN
		0009 SUM
		0010 TAN
		0011 ACOS
		0012 ASIN
		0013 ATAN
		0014 CHAR
		0015 MEAN
		0016 SQRT
		0017 LOG10
		0018 RANGE
		0019 LENGTH
		0020 MEDIAN
		0021 NUMVAL
		0022 RANDOM
		0023 ANNUITY
		0024 INTEGER
		0025 ORD-MAX
		0026 ORD-MIN
		0027 REVERSE
		0028 MIDRANGE
		0029 NUMVAL-C
		0030 VARIANCE
		0031 FACTORIAL
		0032 LOWER-CASE

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		<b>0033</b> UPPER-CASE <b>0034</b> CURRENT-DATE <b>0035</b> INTEGER-PART <b>0036</b> PRESENT-VALUE <b>0037</b> WHEN-COMPILED <b>0038</b> DAY-OF-INTEGERS <b>0039</b> INTEGER-OF-DAY <b>0040</b> DATE-OF-INTEGERS <b>0041</b> INTEGER-OF-DATE <b>0042</b> STANDARD-DEVIATION <b>0043</b> YEAR-TO-YYYY <b>0044</b> DAY-TO-YYYYDDD <b>0045</b> DATE-TO-YYYYMMDD <b>0046</b> UNDATE <b>0047</b> DATEVAL <b>0048</b> YEARWINDOW <b>0049</b> DISPLAY-OF <b>0050</b> NATIONAL-OF
		For Special Register type: <b>0001</b> ADDRESS OF <b>0002</b> LENGTH OF
		For Keyword Class Test Name type: <b>0001</b> ALPHABETIC <b>0002</b> ALPHABETIC-LOWER <b>0003</b> ALPHABETIC-UPPER <b>0004</b> DBCS <b>0005</b> KANJI <b>0006</b> NUMERIC <b>0007</b> NEGATIVE <b>0008</b> POSITIVE <b>0009</b> ZERO
		For Reserved Word type: <b>0001</b> TRUE <b>0002</b> FALSE <b>0003</b> ANY <b>0004</b> THRU

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		For Identifier, Data-name, Index-name, Condition-name or Mnemonic-name type: <b>0001</b> REFERENCED <b>0002</b> CHANGED <b>0003</b> REFERENCED & CHANGED
		For Initialize literal type: <b>0001</b> ALPHABETIC <b>0002</b> ALPHANUMERIC <b>0003</b> NUMERIC <b>0004</b> ALPHANUMERIC-EDITED <b>0005</b> NUMERIC-EDITED <b>0006</b> DBCS/EGCS <b>0007</b> NATIONAL <b>0008</b> NATIONAL-EDITED
		For Procedure-name type: <b>0001</b> SECTION <b>0002</b> PARAGRAPH
		For Reserved word at identifier level type: <b>0001</b> ROUNDED <b>0002</b> TRUE <b>0003</b> ON <b>0004</b> OFF <b>0005</b> SIZE <b>0006</b> DATE <b>0007</b> DAY <b>0008</b> DAY-OF-WEEK <b>0009</b> TIME <b>0010</b> WHEN-COMPILED <b>0011</b> PAGE <b>0012</b> DATE YYYYMMDD <b>0013</b> DAY YYYYDDD

Table 107. SYSADATA parse tree record (continued)

Field	Size	Description
		<p>For Arithmetic Operator type:</p> <p><b>0001</b> PLUS</p> <p><b>0002</b> MINUS</p> <p><b>0003</b> TIMES</p> <p><b>0004</b> DIVIDE</p> <p><b>0005</b> DIVIDE REMAINDER</p> <p><b>0006</b> EXPONENTIATE</p> <p><b>0007</b> NEGATE</p>
		<p>For Relational Operator type:</p> <p><b>0008</b> LESS</p> <p><b>0009</b> LESS OR EQUAL</p> <p><b>0010</b> EQUAL</p> <p><b>0011</b> NOT EQUAL</p> <p><b>0012</b> GREATER</p> <p><b>0013</b> GREATER OR EQUAL</p> <p><b>0014</b> AND</p> <p><b>0015</b> OR</p> <p><b>0016</b> CLASS CONDITION</p> <p><b>0017</b> NOT CLASS CONDITION</p>
Parent node number	FL4	The node number of the parent of the node
Left sibling node number	FL4	The node number of the left sibling of the node, if any. If none, the value is zero.
Symbol Id	FL4	<p>The Symbol Id of the node, if it is a user-name of one of the following types:</p> <ul style="list-style-type: none"> <li>• Data entry</li> <li>• Identifier</li> <li>• File-name</li> <li>• Index-name</li> <li>• Procedure-name</li> <li>• Condition-name</li> <li>• Mnemonic-name</li> </ul> <p>This value corresponds to the Symbol ID in a Symbol (Type 42) record, except for procedure-names where it corresponds to the Paragraph ID.</p> <p>For all other node types this value is zero.</p>
Section Symbol Id	FL4	<p>The Symbol Id of the section containing the node, if it is a qualified paragraph-name reference. This value corresponds to the Section ID in a Symbol (Type 42) record.</p> <p>For all other node types this value is zero.</p>
First token number	FL4	The number of the first token associated with the node

Table 107. **SYSADATA parse tree record** (continued)

Field	Size	Description
Last token number	FL4	The number of the last token associated with the node
Reserved	FL4	Reserved for future use
Flags	CL1	Information about the node:  X'80'    Reserved X'40'    Generated node, no tokens
Reserved	CL3	Reserved for future use

## Token record - X'0030'

The compiler does not generate token records for any lines that are treated as comment lines, which include, but are not limited to, items in the following list.

- Comment lines, which are lines that have an asterisk (\*) or a slash (/) in column 7
- The following compiler-directing statements:
  - \*CBL (\*CONTROL)
  - BASIS
  - >>CALLINT
  - COPY
  - DELETE
  - EJECT
  - INSERT
  - REPLACE
  - SKIP1
  - SKIP2
  - SKIP3
  - TITLE
- Debugging lines, which are lines that have a D in column 7, if WITH DEBUGGING MODE is not specified

Table 108. **SYSADATA token record**

Field	Size	Description
Token number	FL4	The token number within the source file generated by the compiler, starting at 1. Any copybooks have already been included in the source.



Table 108. SYSADATA token record (continued)

Field	Size	Description
Token code	HL2	<p>The type of token (user-name, literal, reserved word, and so forth).</p> <p>For reserved words, the compiler reserved-word table values are used.</p> <p>For PICTURE strings, the special code 0000 is used.</p> <p>For each piece (other than the last) of a continued token, the special code 3333 is used.</p> <p>Otherwise, the following codes are used:</p> <p>0001 ACCEPT</p> <p>0002 ADD</p> <p>0003 ALTER</p> <p>0004 CALL</p> <p>0005 CANCEL</p> <p>0007 CLOSE</p> <p>0009 COMPUTE</p> <p>0011 DELETE</p> <p>0013 DISPLAY</p> <p>0014 DIVIDE</p> <p>0017 READY</p> <p>0018 END-PERFORM</p> <p>0019 ENTER</p> <p>0020 ENTRY</p> <p>0021 EXIT</p> <p>0022 EXEC</p> <p>EXECUTE</p> <p>0023 GO</p> <p>0024 IF</p> <p>0025 INITIALIZE</p> <p>0026 INVOKE</p> <p>0027 INSPECT</p> <p>0028 MERGE</p> <p>0029 MOVE</p>

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0030 MULTIPLY
		0031 OPEN
		0032 PERFORM
		0033 READ
		0035 RELEASE
		0036 RETURN
		0037 REWRITE
		0038 SEARCH
		0040 SET
		0041 SORT
		0042 START
		0043 STOP
		0044 STRING
		0045 SUBTRACT
		0048 UNSTRING
		0049 USE
		0050 WRITE
		0051 CONTINUE
		0052 END-ADD
		0053 END-CALL
		0054 END-COMPUTE
		0055 END-DELETE
		0056 END-DIVIDE
		0057 END-EVALUATE
		0058 END-IF
		0059 END-MULTIPLY
		0060 END-READ
		0061 END-RETURN
		0062 END-REWRITE
		0063 END-SEARCH
		0064 END-START
		0065 END-STRING
		0066 END-SUBTRACT
		0067 END-UNSTRING
		0068 END-WRITE
		0069 GOBACK

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0070 EVALUATE
		0071 RESET
		0072 SERVICE
		0073 END-INVOKE
		0074 END-EXEC
		0075 XML
		0076 END-XML
		0099 FOREIGN-VERB
		0101 DATA-NAME
		0105 DASHED-NUM
		0106 DECIMAL
		0107 DIV-SIGN
		0108 EQ
		0109 EXPONENTIATION
		0110 GT
		0111 INTEGER
		0112 LT
		0113 LPAREN
		0114 MINUS-SIGN
		0115 MULT-SIGN
		0116 NONUMLIT
		0117 PERIOD
		0118 PLUS-SIGN
		0121 RPAREN
		0122 SIGNED-INTEGERS
		0123 QUID
		0124 COLON
		0125 IEOF
		0126 EGCS-LIT
		0127 COMMA-SPACE
		0128 SEMICOLON-SPACE
		0129 PROCEDURE-NAME
		0130 FLT-POINT-LIT
		0131 Language Environment

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0132 GE
		0133 IDREF
		0134 EXPREF
		0136 CICS
		0137 NEW
		0138 NATIONAL-LIT
		0200 ADDRESS
		0201 ADVANCING
		0202 AFTER
		0203 ALL
		0204 ALPHABETIC
		0205 ALPHANUMERIC
		0206 ANY
		0207 AND
		0208 ALPHANUMERIC-EDITED
		0209 BEFORE
		0210 BEGINNING
		0211 FUNCTION
		0212 CONTENT
		0213 CORR
		CORRESPONDING
		0214 DAY
		0215 DATE
		0216 DEBUG-CONTENTS
		0217 DEBUG-ITEM
		0218 DEBUG-LINE
		0219 DEBUG-NAME
		0220 DEBUG-SUB-1
		0221 DEBUG-SUB-2
		0222 DEBUG-SUB-3
		0223 DELIMITED
		0224 DELIMITER
		0225 DOWN

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0226 NUMERIC-EDITED
		0227 XML-EVENT
		0228 END-OF-PAGE
		EOP
		0229 EQUAL
		0230 ERROR
		0231 XML-NTEXT
		0232 EXCEPTION
		0233 EXTEND
		0234 FIRST
		0235 FROM
		0236 GIVING
		0237 GREATER
		0238 I-O
		0239 IN
		0240 INITIAL
		0241 INTO
		0242 INVALID
		0243 SQL
		0244 LESS
		0245 LINAGE-COUNTER
		0246 XML-TEXT
		0247 LOCK
		0248 GENERATE
		0249 NEGATIVE
		0250 NEXT
		0251 NO
		0252 NOT
		0253 NUMERIC
		0254 KANJI
		0255 OR
		0256 OTHER
		0257 OVERFLOW
		0258 PAGE
		0259 CONVERTING

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0260 POINTER
		0261 POSITIVE
		0262 DBCS
		0263 PROCEDURES
		0264 PROCEED
		0265 REFERENCES
		0266 DAY-OF-WEEK
		0267 REMAINDER
		0268 REMOVAL
		0269 REPLACING
		0270 REVERSED
		0271 REWIND
		0272 ROUNDED
		0273 RUN
		0274 SENTENCE
		0275 STANDARD
		0276 RETURN-CODE
		SORT-CORE-SIZE
		SORT-FILE-SIZE
		SORT-MESSAGE
		SORT-MODE-SIZE
		SORT-RETURN
		TALLY
		XML-CODE
		0277 TALLYING
		0278 SUM
		0279 TEST
		0280 THAN
		0281 UNTIL
		0282 UP
		0283 UPON
		0284 VARYING
		0285 RELOAD
		0286 TRUE

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0287 THEN
		0288 RETURNING
		0289 ELSE
		0290 SELF
		0291 SUPER
		0292 WHEN-COMPILED
		0293 ENDING
		0294 FALSE
		0295 REFERENCE
		0296 NATIONAL-EDITED
		0297 COM-REG
		0298 ALPHABETIC-LOWER
		0299 ALPHABETIC-UPPER
		0301 REDEFINES
		0302 OCCURS
		0303 SYNC
		SYNCHRONIZED
		0304 MORE-LABELS
		0305 JUST
		JUSTIFIED
		0306 SHIFT-IN
		0307 BLANK
		0308 VALUE
		0309 COMP
		COMPUTATIONAL
		0310 COMP-1
		COMPUTATIONAL-1
		0311 COMP-3
		COMPUTATIONAL-3
		0312 COMP-2
		COMPUTATIONAL-2
		0313 COMP-4
		COMPUTATIONAL-4
		0314 DISPLAY-1
		0315 SHIFT-OUT

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0316 INDEX
		0317 USAGE
		0318 SIGN
		0319 LEADING
		0320 SEPARATE
		0321 INDEXED
		0322 LEFT
		0323 RIGHT
		0324 PIC
		PICTURE
		0325 VALUES
		0326 GLOBAL
		0327 EXTERNAL
		0328 BINARY
		0329 PACKED-DECIMAL
		0330 EGCS
		0331 PROCEDURE-POINTER
		0332 COMP-5
		COMPUTATIONAL-5
		0333 FUNCTION-POINTER
		0334 TYPE
		0335 JNIENVPTR
		0336 NATIONAL
		0337 GROUP-USAGE
		0401 HIGH-VALUE
		HIGH-VALUES
		0402 LOW-VALUE
		LOW-VALUES
		0403 QUOTE
		QUOTES
		0404 SPACE
		SPACES
		0405 ZERO



Table 108. SYSADATA token record (continued)

Field	Size	Description
		0406 ZEROES
		ZEROS
		0407 NULL
		NULLS
		0501 BLOCK
		0502 BOTTOM
		0505 CHARACTER
		0506 CODE
		0507 CODE-SET
		0514 FILLER
		0516 FOOTING
		0520 LABEL
		0521 LENGTH
		0524 LINAGE
		0526 OMITTED
		0531 RENAMES
		0543 TOP
		0545 TRAILING
		0549 RECORDING
		0601 INHERITS
		0603 RECURSIVE
		0701 ACCESS
		0702 ALSO
		0703 ALTERNATE
		0704 AREA
		AREAS
		0705 ASSIGN
		0707 COLLATING
		0708 COMMA
		0709 CURRENCY
		0710 CLASS
		0711 DECIMAL-POINT
		0712 DUPLICATES
		0713 DYNAMIC
		0714 EVERY

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0716 MEMORY
		0717 MODE
		0718 MODULES
		0719 MULTIPLE
		0720 NATIVE
		0721 OFF
		0722 OPTIONAL
		0723 ORGANIZATION
		0724 POSITION
		0725 PROGRAM
		0726 RANDOM
		0727 RELATIVE
		0728 RERUN
		0729 RESERVE
		0730 SAME
		0731 SEGMENT-LIMIT
		0732 SELECT
		0733 SEQUENCE
		0734 SEQUENTIAL
		0736 SORT-MERGE
		0737 STANDARD-1
		0738 TAPE
		0739 WORDS
		0740 PROCESSING
		0741 APPLY
		0742 WRITE-ONLY
		0743 COMMON
		0744 ALPHABET
		0745 PADDING
		0746 SYMBOLIC
		0747 STANDARD-2
		0748 OVERRIDE
		0750 PASSWORD

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0801 ARE
		IS
		0802 ASCENDING
		0803 AT
		0804 BY
		0805 CHARACTERS
		0806 CONTAINS
		0808 COUNT
		0809 DEBUGGING
		0810 DEPENDING
		0811 DESCENDING
		0812 DIVISION
		0814 FOR
		0815 ORDER
		0816 INPUT
		0817 REPLACE
		0818 KEY
		0819 LINE
		LINES
		0821 OF
		0822 ON
		0823 OUTPUT
		0825 RECORD
		0826 RECORDS
		0827 REEL
		0828 SECTION
		0829 SIZE
		0830 STATUS
		0831 THROUGH
		THRU
		0832 TIME
		0833 TIMES
		0834 TO
		0836 UNIT

Table 108. SYSADATA token record (continued)

Field	Size	Description
		0837 USING
		0838 WHEN
		0839 WITH
		0901 PROCEDURE
		0902 DECLARATIVES
		0903 END
		1001 DATA
		1002 FILE
		1003 FD
		1004 SD
		1005 WORKING-STORAGE
		1006 LOCAL-STORAGE
		1007 LINKAGE
		1101 ENVIRONMENT
		1102 CONFIGURATION
		1103 SOURCE-COMPUTER
		1104 OBJECT-COMPUTER
		1105 SPECIAL-NAMES
		1106 REPOSITORY
		1107 INPUT-OUTPUT
		1108 FILE-CONTROL
		1109 I-O-CONTROL
		1201 ID
		IDENTIFICATION
		1202 PROGRAM-ID
		1203 AUTHOR
		1204 INSTALLATION
		1205 DATE-WRITTEN
		1206 DATE-COMPILED
		1207 SECURITY
		1208 CLASS-ID
		1209 METHOD-ID
		1210 METHOD
		1211 FACTORY

Table 108. SYSADATA token record (continued)

Field	Size	Description
		<b>1212</b> OBJECT <b>2020</b> TRACE <b>3000</b> DATADEF <b>3001</b> F-NAME <b>3002</b> UPSI-SWITCH <b>3003</b> CONDNAME <b>3004</b> CONDVAR <b>3005</b> BLOB <b>3006</b> CLOB <b>3007</b> DBCLOB <b>3008</b> BLOB-LOCATOR <b>3009</b> CLOB-LOCATOR <b>3010</b> DBCLOB-LOCATOR <b>3011</b> BLOB-FILE <b>3012</b> CLOB-FILE <b>3013</b> DBCLOB-FILE <b>3014</b> DFHRESP <b>5001</b> PARSE <b>5002</b> AUTOMATIC <b>5003</b> PREVIOUS <b>9999</b> COBOL
Token length	HL2	The length of the token
Token column	FL4	The starting column number of the token on the source line
Token line	FL4	The line number of the token
Flags	CL1	Information about the token: <b>X'80'</b> Token is continued <b>X'40'</b> Last piece of continued token  Note that for PICTURE strings, even if the source token is continued, there will be only one Token record generated. It will have a token code of 0000, the token column and line of the first piece, the length of the complete string, no continuation flags set, and the token text of the complete string.
Reserved	CL7	Reserved for future use
Token text	CL( <i>n</i> )	The actual token string

## Source error record - X'0032'

The following table shows the contents of the source error record.

Table 109. SYSADATA source error record

Field	Size	Description
Statement number	FL4	The statement number of the statement in error
Error identifier	CL16	The error message identifier (left-justified and padded with blanks)
Error severity	HL2	The severity of the error
Error message length	HL2	The length of the error message text
Line position	XL1	The line position indicator provided in FIPS messages
Reserved	CL7	Reserved for future use
Error message	CL( <i>n</i> )	The error message text

## Source record - X'0038'

The following table shows the contents of the source record.

Table 110. SYSADATA source record

Field	Size	Description
Line number	FL4	The listing line number of the source record
Input record number	FL4	The input source record number in the current input file
Primary file number	HL2	The input file's assigned sequence number if this record is from the primary input file. (Refer to the Input file <i>n</i> field in the Job identification record).
Library file number	HL2	The library input file's assigned sequence number if this record is from a COPY   BASIS input file. (Refer to the Member File ID <i>n</i> field in the Library record.)
Reserved	CL8	Reserved for future use
Parent record number	FL4	The parent source record number. This will be the record number of the COPY   BASIS statement.
Parent primary file number	HL2	The parent file's assigned sequence number if the parent of this record is from the primary input file. (Refer to the Input file <i>n</i> field in the Job Identification Record.)
Parent library assigned file number	HL2	The parent library file's assigned sequence number if this record's parent is from a COPY   BASIS input file. (Refer to the COPY / BASIS Member File ID <i>n</i> field in the Library record.)
Reserved	CL8	Reserved for future use
Length of source record	HL2	The length of the actual source record following
Reserved	CL10	Reserved for future use
Source record	CL( <i>n</i> )	

## COPY REPLACING record - X'0039'

One COPY REPLACING type record will be emitted each time a REPLACING action takes place. That is, whenever *operand-1* of the REPLACING phrase is matched with text in the copybook, a COPY REPLACING TEXT record will be written.

The following table shows the contents of the COPY REPLACING record.

Table 111. **SYSADATA COPY REPLACING** record

Field	Size	Description
Starting line number of replaced string	FL4	The listing line number of the start of the text that resulted from REPLACING
Starting column number of replaced string	FL4	The listing column number of the start of the text that resulted from REPLACING
Ending line number of replaced string	FL4	The listing line number of the end of the text that resulted from REPLACING
Ending column number of replaced string	FL4	The listing column number of the end of the text that resulted from REPLACING
Starting line number of original string	FL4	The source file line number of the start of the text that was changed by REPLACING
Starting column number of original string	FL4	The source file column number of the start of the text that was changed by REPLACING
Ending line number of original string	FL4	The source file line number of the end of the text that was changed by REPLACING
Ending column number of original string	FL4	The source file column number of the end of the text that was changed by REPLACING

## Symbol record - X'0042'

The following table shows the contents of the symbol record.

Table 112. **SYSADATA symbol** record

Field	Size	Description
Symbol ID	FL4	Unique ID of symbol
Line number	FL4	The listing line number of the source record in which the symbol is defined or declared
Level	XL1	True level-number of symbol (or relative level-number of a data item within a structure). For COBOL, this can be in the range 01-49, 66 (for RENAMEs items), 77, or 88 (for condition items).
Qualification indicator	XL1	<b>X'00'</b> Unique name; no qualification needed. <b>X'01'</b> This data item needs qualification. The name is not unique within the program. This field applies only when this data item is <i>not</i> the level-01 name.

Table 112. **SYSADATA symbol record** (continued)

Field	Size	Description
Symbol type	XL1	<p><b>X'68'</b> Class-name (Class-ID)</p> <p><b>X'58'</b> Method-name</p> <p><b>X'40'</b> Data-name</p> <p><b>X'20'</b> Procedure-name</p> <p><b>X'10'</b> Mnemonic-name</p> <p><b>X'08'</b> Program-name</p> <p><b>X'81'</b> Reserved</p> <p>The following are ORed into the above types, when applicable:</p> <p><b>X'04'</b> External</p> <p><b>X'02'</b> Global</p>
Symbol attribute	XL1	<p><b>X'01'</b> Numeric</p> <p><b>X'02'</b> Elementary character of one of these classes:</p> <ul style="list-style-type: none"> <li>• Alphabetic</li> <li>• Alphanumeric</li> <li>• DBCS</li> <li>• National</li> </ul> <p><b>X'03'</b> Group</p> <p><b>X'04'</b> Pointer</p> <p><b>X'05'</b> Index data item</p> <p><b>X'06'</b> Index-name</p> <p><b>X'07'</b> Condition</p> <p><b>X'0F'</b> File</p> <p><b>X'10'</b> Sort file</p> <p><b>X'17'</b> Class-name (repository)</p> <p><b>X'18'</b> Object reference</p>



Table 112. SYSADATA symbol record (continued)

Field	Size	Description
Clauses	XL1	<p>Clauses specified in symbol definition.</p> <p>For symbols that have a symbol attribute of Numeric (X'01'), Elementary character (X'02'), Group (X'03'), Pointer (X'04'), Index data item (X'05'), or Object reference (X'18'):</p> <p><b>1... ....</b> Value</p> <p><b>.1.. ....</b> Indexed</p> <p><b>..1. ....</b> Redefines</p> <p><b>...1 ....</b> Renames</p> <p><b>.... 1...</b> Occurs</p> <p><b>.... .1..</b> Has Occurs keys</p> <p><b>.... ..1.</b> Occurs Depending On</p> <p><b>.... ...1</b> Occurs in parent</p> <p>For both file types:</p> <p><b>1... ....</b> Select</p> <p><b>.1.. ....</b> Assign</p> <p><b>..1. ....</b> Rerun</p> <p><b>...1 ....</b> Same area</p> <p><b>.... 1...</b> Same record area</p> <p><b>.... .1..</b> Recording mode</p> <p><b>.... ..1.</b> Reserved</p> <p><b>.... ...1</b> Record</p>

Table 112. SYSADATA symbol record (continued)

Field	Size	Description
		For mnemonic-name symbols:
	01	CSP
	02	C01
	03	C02
	04	C03
	05	C04
	06	C05
	07	C06
	08	C07
	09	C08
	10	C09
	11	C10
	12	C11
	13	C12
	14	S01
	15	S02
	16	S03
	17	S04
	18	S05
	19	CONSOLE
	20	SYSIN   SYSIPT
	22	SYSOUT   SYSLST   SYSLIST
	24	SYSPUNCH   SYSPCH
	26	UPSI-0
	27	UPSI-1
	28	UPSI-2
	29	UPSI-3
	30	UPSI-4
	31	UPSI-5
	32	UPSI-6
	33	UPSI-7
	34	AFP-5A

Table 112. **SYSADATA symbol record** (continued)

Field	Size	Description
Data flags 1	XL1	<p>For both file types, and for symbols that have a symbol attribute of Numeric (X'01'), Elementary character (X'02'), Group (X'03'), Pointer (X'04'), Index data item (X'05'), or Object reference (X'18'):</p> <p><b>1... ....</b> Redefined</p> <p><b>.1.. ....</b> Renamed</p> <p><b>..1. ....</b> Synchronized</p> <p><b>...1 ....</b> Implicitly redefined</p> <p><b>.... 1...</b> Date field</p> <p><b>.... .1..</b> Implicit redefines</p> <p><b>.... ..1.</b> FILLER</p> <p><b>.... ...1</b> Level 77</p>

Table 112. **SYSADATA symbol record** (continued)

Field	Size	Description
Data flags 2	XL1	<p>For symbols that have a symbol attribute of Numeric (X'01'):</p> <p><b>1... ....</b> Binary</p> <p><b>.1.. ....</b> External floating point (of USAGE DISPLAY or USAGE NATIONAL)</p> <p><b>..1. ....</b> Internal floating point</p> <p><b>...1 ....</b> Packed</p> <p><b>.... 1...</b> External decimal (of USAGE DISPLAY or USAGE NATIONAL)</p> <p><b>.... .1..</b> Scaled negative</p> <p><b>.... ..1.</b> Numeric edited (of USAGE DISPLAY or USAGE NATIONAL)</p> <p><b>.... ...1</b> Reserved for future use</p> <p>For symbols that have a symbol attribute of Elementary character (X'02') or Group (X'03'):</p> <p><b>1... ....</b> Alphabetic</p> <p><b>.1.. ....</b> Alphanumeric</p> <p><b>..1. ....</b> Alphanumeric edited</p> <p><b>...1 ....</b> Group contains its own ODO object</p> <p><b>.... 1...</b> DBCS item</p> <p><b>.... .1..</b> Group variable length</p> <p><b>.... ..1.</b> EGCS item</p> <p><b>.... ...1</b> EGCS edited</p>

Table 112. SYSADATA symbol record (continued)

Field	Size	Description
		<p>For both file types:</p> <p><b>1...</b> .... Object of ODO in record</p> <p><b>.1..</b> .... Subject of ODO in record</p> <p><b>..1.</b> .... Sequential access</p> <p><b>...1</b> .... Random access</p> <p><b>.... 1...</b> Dynamic access</p> <p><b>.... .1..</b> Locate mode</p> <p><b>.... ..1.</b> Record area</p> <p><b>.... ...1</b> Reserved for future use</p> <p>Field will be zero for all other data types.</p>
Data flags 3	XL1	<p>For both file types:</p> <p><b>1...</b> .... All records are the same length</p> <p><b>.1..</b> .... Fixed length</p> <p><b>..1.</b> .... Variable length</p> <p><b>...1</b> .... Undefined</p> <p><b>.... 1...</b> Spanned</p> <p><b>.... .1..</b> Blocked</p> <p><b>.... ..1.</b> Apply write only</p> <p><b>.... ...1</b> Same sort merge area</p> <p>Field will be zero for all other data types.</p>

Table 112. SYSADATA symbol record (continued)

Field	Size	Description
File organization	XL1	<p>For both file types:</p> <p>1... .... QSAM</p> <p>.1... .... ASCII</p> <p>..1. .... Standard label</p> <p>...1 .... User label</p> <p>.... 1... VSAM sequential</p> <p>.... .1.. VSAM indexed</p> <p>.... ..1.. VSAM relative</p> <p>.... ...1 Line sequential</p> <p>Field will be zero for all other data types.</p>
USAGE clause	FL1	<p>X'00' USAGE IS DISPLAY</p> <p>X'01' USAGE IS COMP-1</p> <p>X'02' USAGE IS COMP-2</p> <p>X'03' USAGE IS PACKED-DECIMAL or USAGE IS COMP-3</p> <p>X'04' USAGE IS BINARY, USAGE IS COMP, or USAGE IS COMP-4</p> <p>X'05' USAGE IS DISPLAY-1</p> <p>X'06' USAGE IS POINTER</p> <p>X'07' USAGE IS INDEX</p> <p>X'08' USAGE IS PROCEDURE-POINTER</p> <p>X'09' USAGE IS OBJECT-REFERENCE</p> <p>X'0B' NATIONAL</p> <p>X'0A' FUNCTION-POINTER</p>
Sign clause	FL1	<p>X'00' No SIGN clause</p> <p>X'01' SIGN IS LEADING</p> <p>X'02' SIGN IS LEADING SEPARATE CHARACTER</p> <p>X'03' SIGN IS TRAILING</p> <p>X'04' SIGN IS TRAILING SEPARATE CHARACTER</p>
Indicators	FL1	<p>X'01' Has JUSTIFIED clause. Right-justified attribute is in effect.</p> <p>X'02' Has BLANK WHEN ZERO clause.</p>

Table 112. **SYSADATA symbol record** (continued)

Field	Size	Description
Size	FL4	The size of this data item. The actual number of bytes this item occupies in storage. If a DBCS item, the number is in bytes, not characters. For variable-length items, this field will reflect the maximum size of storage reserved for this item by the compiler. Also known as the "Length attribute."
Precision	FL1	The precision of a fixed or float data item
Scale	FL1	The scale factor of a fixed data item. This is the number of digits to the right of the decimal point.

Table 112. **SYSADATA** symbol record (continued)

Field	Size	Description
Base locator type	FL1	For host:
		01 Base Locator File
		02 Base Locator Working-Storage
		03 Base Locator Linkage Section
		05 Base Locator special regs
		07 Indexed by variable
		09 COMREG special reg
		10 UPSI switch
		13 Base Locator for Varloc items
		14 Base Locator for Extern data
		15 Base Locator alphanumeric FUNC
		16 Base Locator alphanumeric EVAL
		17 Base Locator for Object data
		19 Base Locator for Local-Storage
		20 Factory data
		21 XML-TEXT and XML-NTEXT
		For Windows and AIX:
		01 Base Locator File
		02 Base Locator Linkage Section
		03 Base Locator for Varloc items
		04 Base Locator for Extern data
		05 Base Locator for Object data
		06 XML-TEXT and XML-NTEXT
		10 Base Locator Working-Storage
		11 Base Locator special regs
		12 Base Locator alphanumeric FUNC
		13 Base Locator alphanumeric EVAL
		14 Indexed by variable
		16 COMREG special reg
		17 UPSI switch
		18 Factory data
		22 Base Locator for Local-Storage



Table 112. SYSADATA symbol record (continued)

Field	Size	Description
Date format	FL1	<p>Date format:</p> <p>01 YY</p> <p>02 YYXX</p> <p>03 YYYYYX</p> <p>04 YYYYYX</p> <p>05 YYYY</p> <p>06 YYYYXX</p> <p>07 YYYYYXXX</p> <p>08 YYYYYXXX</p> <p>09 YYX</p> <p>10 YYYYYX</p> <p>22 XYY</p> <p>23 XXXYY</p> <p>24 XXXYY</p> <p>26 XYYYY</p> <p>27 XXXXXXX</p> <p>28 XXXXXXX</p> <p>29 XY</p> <p>30 XYYY</p>
Data flags 4	XL1	<p>For symbols that have a symbol attribute of Numeric (X'01'):</p> <p>1... .. Numeric national</p> <p>For symbols that have a symbol attribute of Elementary character (X'02'):</p> <p>1... .. National</p> <p>.1... .. National edited</p> <p>For symbols that have a symbol attribute of Group (X'03'):</p> <p>1... .. Group-Usage National</p>
Reserved	FL3	Reserved for future use

Table 112. **SYSADATA symbol record** (continued)

Field	Size	Description
Addressing information	FL4	For host, the Base Locator number and displacement:  <b>Bits 0-4</b> Unused  <b>Bits 5-19</b> Base Locator (BL) number  <b>Bits 20-31</b> Displacement off Base Locator  For Windows and AIX, the W-code SymId.
Structure displacement	AL4	Offset of symbol within structure. This offset is set to 0 for variably located items.
Parent displacement	AL4	Byte offset from immediate parent of the item being defined.
Parent ID	FL4	The symbol ID of the immediate parent of the item being defined.
Redefined ID	FL4	The symbol ID of the data item that this item redefines, if applicable.
Start-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the starting COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
End-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the ending COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
Program-name symbol ID	FL4	ID of the program-name of the program or the class-name of the class where this symbol is defined.
OCCURS minimum	FL4	Minimum value for OCCURS
Paragraph ID		Proc-name ID for a paragraph-name
OCCURS maximum	FL4	Maximum value for OCCURS
Section ID		Proc-name ID for a section-name
Dimensions	FL4	Number of dimensions
Reserved	CL12	Reserved for future use
Value pairs count	HL2	Count of value pairs
Symbol name length	HL2	Number of characters in the symbol name
Picture data length for data-name or Assignment-name length for file-name	HL2	Number of characters in the picture data; zero if symbol has no associated PICTURE clause. (Length of the PICTURE field.) Length represents the field as it is found in the source input. This length does not represent the expanded field for PICTURE items that contain a replication factor. The maximum COBOL length for a PICTURE string is 50 bytes. Zero in this field indicates no PICTURE specified.  Number of characters in the external file-name if this is a file-name. This is the DD name part of the assignment-name. Zero if file-name and ASSIGN USING specified.

Table 112. SYSADATA symbol record (continued)

Field	Size	Description
Initial Value length for data-name	HL2	Number of characters in the symbol value; zero if symbol has no initial value
External class-name length for CLASS-ID		Number of characters in the external class-name for CLASS-ID
ODO symbol name ID for data-name	FL4	If data-name, ID of the ODO symbol name; zero if ODO not specified
ID of ASSIGN data-name if file-name		If file-name, Symbol-ID for ASSIGN USING data-name; zero if ASSIGN TO specified
Keys count	HL2	The number of keys defined
Index count	HL2	Count of Index symbol IDs; zero if none specified
Symbol name	CL( <i>n</i> )	
Picture data string for data-name or Assignment-name for file-name	CL( <i>n</i> )	The PICTURE character string <i>exactly</i> as the user types it in. The character string includes all symbols, parentheses, and replication factor.  The external file-name if this is a file-name. This is the DD name part of the assignment-name.
Index ID list	( <i>n</i> )FL4	ID of each index symbol name
Keys	( <i>n</i> )XL8	This field contains data describing keys specified for an array. The following three fields are repeated as many times as specified in the 'Keys count' field.
...Key Sequence	FL1	Ascending or descending indicator.  <b>X'00'</b> DESCENDING  <b>X'01'</b> ASCENDING
...Filler	CL3	Reserved
...Key ID	FL4	The symbol ID of the data item that is the key field in the array
Initial Value data for data-name  External class-name for CLASS-ID	CL( <i>n</i> )	This field contains the data specified in the INITIAL VALUE clause for this symbol. The following four subfields are repeated according to the count in the 'Value pairs count' field. The total length of the data in this field is contained in the 'Initial value length' field.  The external class-name for CLASS-ID.
...1st value length	HL2	Length of first value
...1st value data	CL( <i>n</i> )	1st value.  This field contains the literal (or figurative constant) as it is specified in the VALUE clause in the source file. It includes any beginning and ending delimiters, embedded quotation marks, and SHIFT IN and SHIFT OUT characters. If the literal spans multiple lines, the lines are concatenated into one long string. If a figurative constant is specified, this field contains the actual reserved word, not the value associated with that word.
...2nd value length	HL2	Length of second value, zero if not a THRU value pair

Table 112. **SYSADATA symbol record** (continued)

Field	Size	Description
...2nd value data	CL( <i>n</i> )	2nd value.  This field contains the literal (or figurative constant) as it is specified in the VALUE clause in the source file. It includes any beginning and ending delimiters, embedded quotation marks, and SHIFT IN and SHIFT OUT characters. If the literal spans multiple lines, the lines are concatenated into one long string. If a figurative constant is specified, this field contains the actual reserved word, not the value associated with that word.

## Symbol cross-reference record - X'0044'

The following table shows the contents of the symbol cross-reference record.

Table 113. **SYSADATA symbol cross-reference record**

Field	Size	Description
Symbol length	HL2	The length of the symbol
Statement definition	FL4	The statement number where the symbol is defined or declared  <b>For VERB XREF only:</b>  Verb count - total number of references to this verb.
Number of references <sup>1</sup>	HL2	The number of references in this record to the symbol following
Cross-reference type	XL1	X'01' Program X'02' Procedure X'03' Verb X'04' Symbol or data-name X'05' Method X'06' Class
Reserved	CL7	Reserved for future use
Symbol name	CL( <i>n</i> )	The symbol. Variable length.

Table 113. **SYSADATA symbol cross-reference record** (continued)

Field	Size	Description
...Reference flag	CL1	<p>For symbol or data-name references:</p> <p><b>C' '</b> Blank means reference only</p> <p><b>C'M'</b> Modification reference flag</p> <p>For Procedure type symbol references:</p> <p><b>C'A'</b> ALTER (procedure-name)</p> <p><b>C'D'</b> GO TO (procedure-name) DEPENDING ON</p> <p><b>C'E'</b> End of range of (PERFORM) through (procedure-name)</p> <p><b>C'G'</b> GO TO (procedure-name)</p> <p><b>C'P'</b> PERFORM (procedure-name)</p> <p><b>C'T'</b> (ALTER) TO PROCEED TO (procedure-name)</p> <p><b>C'U'</b> Use for debugging (procedure-name)</p>
...Statement number	XL4	The statement number on which the symbol or verb is referenced
<p>1. The reference flag field and the statement number field occur as many times as the number of references field dictates. For example, if there is a value of 10 in the number of references field, there will be 10 occurrences of the reference flag and statement number pair for data-name, procedure, or program symbols, or 10 occurrences of the statement number for verbs.</p> <p>Where the number of references would exceed the record size for the SYSADATA file, the record is continued on the next record. The continuation flag is set in the common header section of the record.</p>		

## Nested program record - X'0046'

The following table shows the contents of the nested program record.

Table 114. **SYSADATA nested program record**

Field	Size	Description
Statement definition	FL4	The statement number where the symbol is defined or declared
Nesting level	XL1	Program nesting level
Program attributes	XL1	<p><b>1... ....</b> Initial</p> <p><b>.1.. ....</b> Common</p> <p><b>..1. ....</b> PROCEDURE DIVISION using</p> <p><b>...1 1111</b> Reserved for future use</p>
Reserved	XL1	Reserved for future use
Program-name length	XL1	Length of the following field
Program-name	CL( <i>n</i> )	The program-name

---

## Library record - X'0060'

The following table shows the contents of the SYSADATA library record.

*Table 115. SYSADATA library record*

Field	Size	Description
Number of members <sup>1</sup>	HL2	Count of the number of COPY/INCLUDE code members described in this record
Library name length	HL2	The length of the library name
Library volume length	HL2	The length of the library volume ID
Concatenation number	XL2	Concatenation number of the library
Library ddname length	HL2	The length of the library ddname
Reserved	CL4	Reserved for future use
Library name	CL( <i>n</i> )	The name of the library from which the COPY/INCLUDE member was retrieved
Library volume	CL( <i>n</i> )	The volume identification of the volume where the library resides
Library ddname	CL( <i>n</i> )	The ddname (or equivalent) used for this library
...COPY/BASIS member file ID <sup>2</sup>	HL2	The library file ID of the name following
...COPY/BASIS name length	HL2	The length of the name following
...COPY/BASIS name	CL( <i>n</i> )	The name of the COPY/BASIS member that has been used
<ol style="list-style-type: none"><li>1. If 10 COPY members are retrieved from a library, the "Number of members" field will contain 10 and there will be 10 occurrences of the "COPY/BASIS member file ID" field, the "COPY/BASIS name length" field, and the "COPY/BASIS name" field.</li><li>2. If COPY/BASIS members are retrieved from different libraries, a library record is written to the SYSADATA file for each unique library.</li></ol>		

---

## Statistics record - X'0090'

The following table shows the contents of the statistics record.

*Table 116. SYSADATA statistics record*

Field	Size	Description
Source records	FL4	The number of source records processed
DATA DIVISION statements	FL4	The number of data division statements processed
PROCEDURE DIVISION statements	FL4	The number of procedure division statements processed
Compilation number	HL2	Batch compilation number
Error severity	XL1	The highest error message severity

Table 116. **SYSADATA statistics record** (continued)

Field	Size	Description
Flags	XL1	<b>1... ....</b> End of Job indicator  <b>.1... ....</b> Class definition indicator  <b>..11 1111</b> Reserved for future use
EOJ severity	XL1	The maximum return code for the compile job
Program-name length	XL1	The length of the program-name
Program-name	CL( <i>n</i> )	Program-name

## EVENTS record - X'0120'

Events records are included in the ADATA file to provide compatibility with previous levels of the compiler.

Events records are of the following types:

- Timestamp
- Processor
- File end
- Program
- File ID
- Error

Table 117. **SYSADATA EVENTS TIMESTAMP record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type TIMESTAMP record	CL12	C'TIMESTAMP'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Date	XL8	YYYYMMDD
Hour	XL2	HH
Minutes	XL2	MI
Seconds	XL2	SS

Table 118. **SYSADATA EVENTS PROCESSOR record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)

Table 118. **SYSADATA EVENTS PROCESSOR** record layout (continued)

Field	Size	Description
EVENTS record type PROCESSOR record	CL9	C'PROCESSOR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Line-class indicator	XL1	

Table 119. **SYSADATA EVENTS FILE END** record layout

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE END record	CL7	C'FILEEND'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input file ID	XL1	
Blank separator	CL1	
Expansion indicator	XL1	

Table 120. **SYSADATA EVENTS PROGRAM** record layout

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type PROGRAM record	CL7	C'PROGRAM'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Program input record number	XL1	

Table 121. **SYSADATA EVENTS FILE ID** record layout

Field	Size	Description
Header	CL12	Standard ADATA record header



Table 121. **SYSADATA EVENTS FILE ID record layout** (continued)

Field	Size	Description
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE ID record	CL7	C'FILEID'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Reference indicator	XL1	
Blank separator	CL1	
Source file name length	H2	
Blank separator	CL1	
Source file name	CL(n)	

Table 122. **SYSADATA EVENTS ERROR record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type ERROR record	CL5	C'ERROR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Annot class	XL1	Annot-class message placement
Blank separator	CL1	
Error input record number	XL10	
Blank separator	CL1	
Error start line number	XL10	
Blank separator	CL1	
Error token start number	XL1	Column number of error token start
Blank separator	CL1	
Error end line number	XL10	
Blank separator	CL1	

Table 122. **SYSADATA EVENTS ERROR** record layout (continued)

Field	Size	Description
Error token end number	XL1	Column number of error token end
Blank separator	CL1	
Error message ID number	XL9	
Blank separator	CL1	
Error message severity code	XL1	
Blank separator	CL1	
Error message severity level number	XL2	
Blank separator	CL1	
Error message length	HL3	
Blank separator	CL1	
Error message text	CL(n)	

---

## Appendix I. Runtime messages

Messages for COBOL for Windows 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 123. Runtime messages**

Message number	Message text
“IWZ006S” on page 714	The reference to table <i>table-name</i> by verb number <i>verb-number</i> on line <i>line-number</i> addressed an area outside the region of the table.
“IWZ007S” on page 715	The reference to variable-length group <i>group-name</i> by verb number <i>verb-number</i> on line <i>line-number</i> addressed an area outside the maximum defined length of the group.
“IWZ012I” on page 715	Invalid run unit termination occurred while sort or merge is running.
“IWZ013S” on page 715	Sort or merge requested while sort or merge is running in a different thread.
“IWZ026W” on page 715	The SORT-RETURN special register was never referenced, but the current content indicated the sort or merge operation in program <i>program-name</i> on line number <i>line-number</i> was unsuccessful. The sort or merge return code was <i>return code</i> .
“IWZ029S” on page 716	Argument-1 for function <i>function-name</i> in program <i>program-name</i> at line <i>line-number</i> was less than zero.
“IWZ030S” on page 716	Argument-2 for function <i>function-name</i> in program <i>program-name</i> at line <i>line-number</i> was not a positive integer.

Table 123. Runtime messages (continued)

"IWZ036W" on page 716	Truncation of high order digit positions occurred in program <i>program-name</i> on line number <i>line-number</i> .
"IWZ037I" on page 716	The flow of control in program <i>program-name</i> proceeded beyond the last line of the program. Control returned to the caller of the program <i>program-name</i> .
"IWZ038S" on page 717	A reference modification length value of <i>reference-modification-value</i> on line <i>line-number</i> which was not equal to 1 was found in a reference to data item <i>data-item</i> .
"IWZ039S" on page 717	An invalid overpunched sign was detected.
"IWZ040S" on page 718	An invalid separate sign was detected.
"IWZ045S" on page 718	Unable to invoke method <i>method-name</i> on line number <i>line number</i> in program <i>program-name</i> .
"IWZ047S" on page 718	Unable to invoke method <i>method-name</i> on line number <i>line number</i> in class <i>class-name</i> .
"IWZ048W" on page 718	A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.
"IWZ049W" on page 719	A zero base was raised to a zero power in an exponentiation expression. The result was set to one.
"IWZ050S" on page 719	A zero base was raised to a negative power in an exponentiation expression.
"IWZ051W" on page 719	No significant digits remain in a fixed-point exponentiation operation in program <i>program-name</i> due to excessive decimal positions specified in the operands or receivers.
"IWZ053S" on page 720	An overflow occurred on conversion to floating point.
"IWZ054S" on page 720	A floating-point exception occurred.
"IWZ055W" on page 720	An underflow occurred on conversion to floating point. The result was set to zero.
"IWZ058S" on page 720	Exponent overflow occurred.
"IWZ059W" on page 721	An exponent with more than nine digits was truncated.
"IWZ060W" on page 721	Truncation of high order digit positions occurred.
"IWZ061S" on page 721	Division by zero occurred.
"IWZ063S" on page 721	An invalid sign was detected in a numeric edited sending field in <i>program-name</i> on line number <i>line-number</i> .
"IWZ064S" on page 722	A recursive call to active program <i>program-name</i> in compilation unit <i>compilation-unit</i> was attempted.
"IWZ065I" on page 722	A CANCEL of active program <i>program-name</i> in compilation unit <i>compilation-unit</i> was attempted.
"IWZ066S" on page 722	The length of external data record <i>data-record</i> in program <i>program-name</i> did not match the existing length of the record.

Table 123. Runtime messages (continued)

"IWZ071S" on page 722	ALL subscripted table reference to table <i>table-name</i> by verb number <i>verb-number</i> on line <i>line-number</i> had an ALL subscript specified for an OCCURS DEPENDING ON dimension, and the object was less than or equal to 0.
"IWZ072S" on page 723	A reference modification start position value of <i>reference-modification-value</i> on line <i>line-number</i> referenced an area outside the region of data item <i>data-item</i> .
"IWZ073S" on page 723	A nonpositive reference modification length value of <i>reference-modification-value</i> on line <i>line-number</i> was found in a reference to data item <i>data-item</i> .
"IWZ074S" on page 723	A reference modification start position value of <i>reference-modification-value</i> and length value of <i>length</i> on line <i>line-number</i> caused reference to be made beyond the rightmost character of data item <i>data-item</i> .
"IWZ075S" on page 724	Inconsistencies were found in EXTERNAL file <i>file-name</i> in program <i>program-name</i> . The following file attributes did not match those of the established external file: <i>attribute-1 attribute-2 attribute-3 attribute-4 attribute-5 attribute-6 attribute-7</i> .
"IWZ076W" on page 724	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 724	The lengths of the INSPECT data items were not equal. The shorter length was used.
"IWZ078S" on page 724	ALL subscripted table reference to table <i>table-name</i> by verb number <i>verb-number</i> on line <i>line-number</i> will exceed the upper bound of the table.
"IWZ096C" on page 725	Dynamic call of program <i>program-name</i> failed. Message variants include: <ul style="list-style-type: none"> <li>• A load of module <i>module-name</i> failed with an error code of <i>error-code</i>.</li> <li>• A load of module <i>module-name</i> failed with a return code of <i>return-code</i>.</li> <li>• Dynamic call of program <i>program-name</i> failed. Insufficient resources.</li> <li>• Dynamic call of program <i>program-name</i> failed. COBPATH not found in environment.</li> <li>• Dynamic call of program <i>program-name</i> failed. Entry <i>entry-name</i> not found.</li> <li>• Dynamic call failed. The name of the target program does not contain any valid characters.</li> <li>• Dynamic call of program <i>program-name</i> failed. The load module <i>load-module</i> could not be found in the directories identified in the COBPATH environment variable.</li> </ul>
"IWZ097S" on page 725	Argument-1 for function <i>function-name</i> contained no digits.
"IWZ100S" on page 725	Argument-1 for function <i>function-name</i> was less than or equal to -1.
"IWZ103S" on page 726	Argument-1 for function <i>function-name</i> was less than zero or greater than 99.
"IWZ104S" on page 726	Argument-1 for function <i>function-name</i> was less than zero or greater than 99999.

Table 123. Runtime messages (continued)

"IWZ105S" on page 726	Argument-1 for function <i>function-name</i> was less than zero or greater than 999999.
"IWZ151S" on page 726	Argument-1 for function <i>function-name</i> contained more than 18 digits.
"IWZ152S" on page 726	Invalid character <i>character</i> was found in column <i>column-number</i> in argument-1 for function <i>function-name</i> .
"IWZ155S" on page 727	Invalid character <i>character</i> was found in column <i>column-number</i> in argument-2 for function <i>function-name</i> .
"IWZ156S" on page 727	Argument-1 for function <i>function-name</i> was less than zero or greater than 28.
"IWZ157S" on page 727	The length of Argument-1 for function <i>function-name</i> was not equal to 1.
"IWZ158S" on page 727	Argument-1 for function <i>function-name</i> was less than zero or greater than 29.
"IWZ159S" on page 728	Argument-1 for function <i>function-name</i> was less than 1 or greater than 3067671.
"IWZ160S" on page 728	Argument-1 for function <i>function-name</i> was less than 16010101 or greater than 99991231.
"IWZ161S" on page 728	Argument-1 for function <i>function-name</i> was less than 1601001 or greater than 9999365.
"IWZ162S" on page 728	Argument-1 for function <i>function-name</i> was less than 1 or greater than the number of positions in the program collating sequence.
"IWZ163S" on page 728	Argument-1 for function <i>function-name</i> was less than zero.
"IWZ165S" on page 729	A reference modification start position value of <i>start-position-value</i> on line <i>line number</i> referenced an area outside the region of the function result of <i>function-result</i> .
"IWZ166S" on page 729	A nonpositive reference modification length value of <i>length</i> on line <i>line-number</i> was found in a reference to the function result of <i>function-result</i> .
"IWZ167S" on page 729	A reference modification start position value of <i>start-position</i> and length value of <i>length</i> on line <i>line-number</i> caused reference to be made beyond the rightmost character of the function result of <i>function-result</i> .
"IWZ168W" on page 729	SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.
"IWZ170S" on page 730	Illegal data type for DISPLAY operand.
"IWZ171I" on page 730	<i>string-name</i> is not a valid runtime option.
"IWZ172I" on page 730	The string <i>string-name</i> is not a valid suboption of the runtime option <i>option-name</i> .
"IWZ173I" on page 731	The suboption string <i>string-name</i> of the runtime option <i>option-name</i> must be <i>number</i> characters long. The default will be used.
"IWZ174I" on page 731	The suboption string <i>string-name</i> of the runtime option <i>option-name</i> contains one or more invalid characters. The default will be used.
"IWZ175S" on page 731	There is no support for routine <i>routine-name</i> on this system.

Table 123. Runtime messages (continued)

"IWZ176S" on page 731	Argument-1 for function <i>function-name</i> was greater than <i>decimal-value</i> .
"IWZ177S" on page 731	Argument-2 for function <i>function-name</i> was equal to <i>decimal-value</i> .
"IWZ178S" on page 732	Argument-1 for function <i>function-name</i> was less than or equal to <i>decimal-value</i> .
"IWZ179S" on page 732	Argument-1 for function <i>function-name</i> was less than <i>decimal-value</i> .
"IWZ180S" on page 732	Argument-1 for function <i>function-name</i> was not an integer.
"IWZ181I" on page 732	An invalid character was found in the numeric string <i>string</i> of the runtime option <i>option-name</i> . The default will be used.
"IWZ182I" on page 732	The number <i>number</i> of the runtime option <i>option-name</i> exceeded the range of <i>min-range</i> to <i>max-range</i> . The default will be used.
"IWZ183S" on page 733	The function name in _IWZCOBOLInit did a return.
"IWZ200S" on page 733	Error detected during I/O operation for file <i>file-name</i> . File status is: <i>file-status</i> .
"IWZ200S" on page 733	STOP or ACCEPT failed with an I/O error, <i>error-code</i> . The run unit is terminated.
"IWZ203W" on page 733	The code page in effect is not a DBCS code page.
"IWZ204W" on page 734	An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.
"IWZ221S" on page 734	ICU converter for code page, <i>codepage value</i> , can not be opened.
"IWZ222S" on page 734	Data conversion via ICU failed with error code <i>error code value</i> .
"IWZ223S" on page 734	Close of ICU converter failed with error code <i>error code value</i> .
"IWZ224S" on page 735	ICU collator for the locale value, <i>locale value</i> , can not be opened. The error code is <i>error code value</i> .
"IWZ225S" on page 735	Unicode case mapping function via ICU failed with error code <i>error code value</i> .
"IWZ230W" on page 735	The conversion table for the current codeset, <i>ASCII codeset-id</i> , to the EBCDIC codeset, <i>EBCDIC codeset-id</i> , is not available. The default ASCII to EBCDIC conversion table will be used.
"IWZ230W" on page 735	The EBCDIC code page specified, <i>EBCDIC codepage</i> , is not consistent with the locale <i>locale</i> , but will be used as requested.
"IWZ230W" on page 736	The EBCDIC code page specified, <i>EBCDIC codepage</i> , is not supported. The default EBCDIC code page, <i>EBCDIC codepage</i> , will be used.
"IWZ230S" on page 736	The EBCDIC conversion table cannot be opened.
"IWZ230S" on page 736	The EBCDIC conversion table cannot be built.
"IWZ230S" on page 736	The main program was compiled with both the -host flag and the CHAR(NATIVE) option, which are not compatible.

Table 123. Runtime messages (continued)

"IWZ231S" on page 737	Query of current locale setting failed.
"IWZ232W" on page 737	<p>Message variants include:</p> <ul style="list-style-type: none"> <li>• An error occurred during the conversion of data item <i>data-name</i> to EBCDIC in program <i>program-name</i> on line number <i>decimal-value</i>.</li> <li>• An error occurred during the conversion of data item <i>data-name</i> to ASCII in program <i>program-name</i> on line number <i>decimal-value</i>.</li> <li>• An error occurred during the conversion to EBCDIC for data item <i>data-name</i> in program <i>program-name</i> on line number <i>decimal-value</i>.</li> <li>• An error occurred during the conversion to ASCII for data item <i>data-name</i> in program <i>program-name</i> on line number <i>decimal-value</i>.</li> <li>• An error occurred during the conversion from ASCII to EBCDIC in program <i>program-name</i> on line number <i>decimal-value</i>.</li> <li>• An error occurred during the conversion from EBCDIC to ASCII in program <i>program-name</i> on line number <i>decimal-value</i>.</li> </ul>
"IWZ240S" on page 738	The base year for program <i>program-name</i> was outside the valid range of 1900 through 1999. The sliding window value <i>window-value</i> resulted in a base year of <i>base-year</i> .
"IWZ241S" on page 738	The current year was outside the 100-year window, <i>year-start</i> through <i>year-end</i> , for program <i>program-name</i> .
"IWZ242S" on page 738	There was an invalid attempt to start an XML PARSE statement.
"IWZ243S" on page 739	There was an invalid attempt to end an XML PARSE statement.
"IWZ250S" on page 739	Internal error: the call to JNI_GetCreatedJavaVMs returned an error, return code <i>nn</i> .
"IWZ251S" on page 739	Internal error: <i>n</i> active Java VMs were detected, when only 1 was expected.
"IWZ253S" on page 739	More than <i>nn</i> JVM initialization options was specified.
"IWZ254S" on page 739	Internal error: the call to JNI_CreateJavaVM returned an error.
"IWZ255S" on page 740	Internal error: the call to GetEnv returned code <i>nn</i> because the current thread is not attached to the JVM.
"IWZ256S" on page 740	Internal error: the call to GetEnv returned code <i>nn</i> because the JVM version is not supported.
"IWZ257S" on page 740	Internal error: the call to GetEnv returned the unrecognized return code <i>nn</i> .
"IWZ258S" on page 740	Internal error: GetByteArrayElements was unable to acquire a pointer to the instance data.
"IWZ259S" on page 741	Unable to acquire a direct pointer to the instance data. The installed JVM does not support pinned byte arrays.
"IWZ258S" on page 740	The Java class <i>name</i> could not be found.
"IWZ813S" on page 741	Insufficient storage was available to satisfy a get storage request.



Table 123. Runtime messages (continued)

"IWZ901S" on page 741	<p>Message variants include:</p> <ul style="list-style-type: none"> <li>• Program exits due to severe or critical error.</li> <li>• Program exits: more than ERRCOUNT errors occurred.</li> </ul>
"IWZ902S" on page 742	The system detected a decimal-divide exception.
"IWZ903S" on page 742	The system detected a data exception.
"IWZ907S" on page 742	<p>Message variants include:</p> <ul style="list-style-type: none"> <li>• Insufficient storage.</li> <li>• Insufficient storage. Cannot get <i>number-bytes</i> bytes of space for <i>storage</i>.</li> </ul>
"IWZ993W" on page 742	Insufficient storage. Cannot find space for message <i>message-number</i> .
"IWZ994W" on page 743	Cannot find message <i>message-number</i> in message-catalog.
"IWZ995C" on page 743	<p>Message variants include:</p> <ul style="list-style-type: none"> <li>• <i>System exception</i> signal received while executing routine <i>routine-name</i> at offset <i>0xoffset-value</i>.</li> <li>• <i>System exception</i> signal received while executing code at location <i>0xoffset-value</i>.</li> <li>• <i>System exception</i> signal received. The location could not be determined.</li> </ul>
"IWZ2502S" on page 743	The UTC/GMT was not available from the system.
"IWZ2503S" on page 743	The offset from UTC/GMT to local time was not available from the system.
"IWZ2505S" on page 744	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
"IWZ2506S" on page 744	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 744	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
"IWZ2508S" on page 745	The date value passed to CEEDAYS or CEESECS was invalid.
"IWZ2509S" on page 745	The era passed to CEEDAYS or CEESECS was not recognized.
"IWZ2510S" on page 745	The hours value in a call to CEEISEC or CEESECS was not recognized.
"IWZ2511S" on page 746	The day parameter passed in a CEEISEC call was invalid for year and month specified.
"IWZ2512S" on page 746	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.
"IWZ2513S" on page 746	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
"IWZ2514S" on page 747	The year value passed in a CEEISEC call was not within the supported range.

Table 123. Runtime messages (continued)

"IWZ2515S" on page 747	The milliseconds value in a CEEISEC call was not recognized.
"IWZ2516S" on page 747	The minutes value in a CEEISEC call was not recognized.
"IWZ2517S" on page 747	The month value in a CEEISEC call was not recognized.
"IWZ2518S" on page 748	An invalid picture string was specified in a call to a date/time service.
"IWZ2519S" on page 748	The seconds value in a CEEISEC call was not recognized.
"IWZ2520S" on page 748	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
"IWZ2521S" on page 749	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.
"IWZ2522S" on page 749	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 749	CEESECS detected nonnumeric data in a numeric field, or the timestamp string did not match the picture string.
"IWZ2526S" on page 750	The date string returned by CEEDATE was truncated.
"IWZ2527S" on page 750	The timestamp string returned by CEEDATM was truncated.
"IWZ2531S" on page 750	The local time was not available from the system.
"IWZ2533S" on page 750	The value passed to CEESCEN was not between 0 and 100.
"IWZ2534W" on page 751	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 141 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 below 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:** The application was terminated.

#### 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.

#### IWZ045S

Unable to invoke method *method-name* on line number *line number* in program *program-name*.

**Explanation:** The specific method is not supported for the class of the current object reference.

**Programmer response:** Check the indicated line number in the program to ensure that the class of the current object reference supports the method being invoked.

**System action:** The application was terminated.

#### IWZ047S

Unable to invoke method *method-name* on line number *line number* in class *class-name*.

**Explanation:** The specific method is not supported for the class of the current object reference.

**Programmer response:** Check the indicated line number in the class to ensure that the class of the current object reference supports the method being invoked.

**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.

#### IWZ051W

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:** No system action was taken.

#### 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 below 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 which must match between definitions of the same external file.

**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

Dynamic call of program *program-name* failed. Message variants include:

- A load of module *module-name* failed with an error code of *error-code*.
- 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 last-error code value set by LoadLibrary.

**Programmer response:** Check that COBPATH is defined. Check that the module exists: Windows has graphical interfaces for showing directories and files. You can also use the dir command. 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, for example, to build a DLL on Windows, the -dll option must be used.

**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 <i>function-name</i> was less than 1 or greater than 3067671.
---------------------------------------------------------------------------------------

**Explanation:** The input argument to DATE-OF-INTEGERS or DAY-OF-INTEGERS 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 <i>function-name</i> was less than 16010101 or greater than 99991231.
-----------------------------------------------------------------------------------------------

**Explanation:** The input argument to function INTEGERS-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 <i>function-name</i> was less than 1601001 or greater than 9999365.
---------------------------------------------------------------------------------------------

**Explanation:** The input argument to function INTEGERS-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 <i>function-name</i> 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 <i>function-name</i> 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 SET 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.

#### 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 PROCEDURE-POINTER
- Data items defined with USAGE IS OBJECT REFERENCE
- 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 <i>file-name</i> . File status is: <i>file-status</i> .
------------------------------------------------------------------------------------------------------

**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, <i>error-code</i> . 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.

#### IWZ203W

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-943

Country or region	Code page
Korea	
People's Republic of China (Simplified)	
Taiwan (Traditional)	IBM-950

**Note:** The code pages listed above might not be supported for a specific version or release of that platform.

**System Action:** No system action was taken.

#### IWZ204W

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:** No system action was taken.

#### IWZ221S

The ICU converter for the code page, <i>codepage value</i> , can not be opened. The error code is <i>error code value</i> .
-----------------------------------------------------------------------------------------------------------------------------

**Explanation:** The ICU converter to convert between the code page and UTF-16 cannot be opened.

**Programmer response:** Verify that the code-page value is valid according to Code page names (*COBOL for Windows Language Reference*). 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 <i>error code value</i> .
--------------------------------------------------------------------------

**Explanation:** The data conversion through ICU failed.

**Programmer response:** Contact your IBM representative.

**System action:** The application was terminated.

#### IWZ223S

Close of ICU converter failed with error code <i>error code value</i> .
-------------------------------------------------------------------------

**Explanation:** The close of an ICU converter failed.

**Programmer response:** Contact your IBM representative.

**System action:** The application was terminated.

#### 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 via ICU failed with error code *error code 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 set, *ASCII codeset-id*, to the EBCDIC code set, *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 and EBCDIC code page IBM-037.

**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, <i>EBCDIC codepage</i> , is not supported. The default EBCDIC code page, <i>EBCDIC codepage</i> , 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 <code>-host</code> flag and the <code>CHAR(NATIVE)</code> 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 variables:

##### LOCPATH

This environment variable should include the IBMCOBOL\LOCALE directory.

##### LANG

This should be set to the name of one of the directories located in the IBMCOBW\LOCALE directory. The default value is en\_US.

**System action:** The application was terminated.

#### IWZ232W

Message variants include:
---------------------------

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• An error occurred during the conversion of data item <i>data-name</i> to EBCDIC in program <i>program-name</i> on line number <i>decimal-value</i>.</li><li>• An error occurred during the conversion of data item <i>data-name</i> to ASCII in program <i>program-name</i> on line number <i>decimal-value</i>.</li><li>• An error occurred during the conversion to EBCDIC for data item <i>data-name</i> in program <i>program-name</i> on line number <i>decimal-value</i>.</li><li>• An error occurred during the conversion to ASCII for data item <i>data-name</i> in program <i>program-name</i> on line number <i>decimal-value</i>.</li><li>• An error occurred during the conversion from ASCII to EBCDIC in program <i>program-name</i> on line number <i>decimal-value</i>.</li><li>• An error occurred during the conversion from EBCDIC to ASCII in program <i>program-name</i> on line number <i>decimal-value</i>.</li></ul> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**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.

#### IWZ250S

Internal error: the call to JNI\_GetCreatedJavaVMs returned an error, return code *nn*.

**Explanation:** The call to the JNI\_GetCreatedJavaVMs function returned an error with a return code of *nn*.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ251S

Internal error: *n* active Java VMs were detected, when only 1 was expected.

**Explanation:** Multiple active Java VMs were detected, when only one Java VM was expected.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ253S

More than *nn* JVM initialization options was specified.

**Explanation:** In the COBJVMINITOPTIONS environment variable, you specified more Java initialization options than the maximum allowable number of options. The limit is 256.

**Programmer response:** Specify fewer options in the COBJVMINITOPTIONS environment variable.

**System action:** Application execution is terminated.

#### IWZ254S

**Internal error: the call to JNI\_CreateJavaVM returned an error.**

**Explanation:** The call to JNI\_CreateJavaVM returned an error.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ255S

**Internal error: the call to GetEnv returned code *nn* because the current thread is not attached to the JVM.**

**Explanation:** The call to GetEnv returned code *nn* because the current thread is not attached to the JVM.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ256S

**Internal error: the call to GetEnv returned code *nn* because the JVM version is not supported.**

**Explanation:** The call to GetEnv returned code *nn* because the JVM version is not supported.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ257S

**Internal error: the call to GetEnv returned the unrecognized return code *nn*.**

**Explanation:** The call to GetEnv returned the unrecognized return code *nn*.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ258S

**Internal error: GetByteArrayElements was unable to acquire a pointer to the instance data.**

**Explanation:** The GetByteArrayElements service was unable to acquire a pointer to the instance data.

**Programmer response:** Contact your IBM representative.

**System action:** Application execution is terminated.

#### IWZ259S

Unable to acquire a direct pointer to instance data. The installed JVM does not support pinned byte arrays.

**Explanation:** You are running with an unsupported JVM, such as Sun 1.4.1, that doesn't support pinned byte arrays. For more information on pinning, see The Java Native Interface.

**Programmer response:** Run a supported JVM.

**System action:** Application execution is terminated.

#### IWZ260S

The Java class *name* could not be found.

**Explanation:** Your program references a class name that is not defined or specified in the CLASSPATH environment variable.

**Programmer response:** Check the program that references *name*, and either correct the reference or provide the missing *name.class*.

**System action:** Application execution 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 0xoffset-value.*
- *system exception signal received while executing code at location 0xoffset-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 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 timestamp 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 timestamp string returned by CEEDATM was truncated.

**Explanation:** In a CEEDATM call, the output string was not large enough to contain the formatted timestamp value.

**Programmer response:** Verify that the output string data item is large enough to contain the entire formatted timestamp. 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 CEEOCT 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, MMMMMM, WWW, Wwwww, 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 569

**RELATED TASKS**

"Setting environment variables" on page 193

"Generating a list of compiler error messages" on page 204

**RELATED REFERENCES**

"Locales and code pages that are supported" on page 183





---

## 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 Corporation  
J46A/G4  
555 Bailey Avenue  
San Jose, CA 95141-1099  
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:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, 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 publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM  
The IBM logo  
ibm.com  
AIX  
CICS  
COBOL/370  
Database 2  
DB2  
DB2 Universal Database  
Language Environment  
MVS  
OS/390  
Rational  
System z  
TXSeries  
VisualAge  
WebSphere  
z/OS  
zSeries

Intel and Pentium are registered trademarks of Intel Corporation in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. 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 company, product, or service names may be the trademarks or service marks of others.

---

## 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.

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*
- *ANSI X3.172-2002, American National Standard Dictionary for Information Systems*

American National Standard definitions are preceded by an asterisk (\*).

This glossary includes definitions developed by Sun Microsystems, Inc. for their Java and J2EE glossaries. When Sun is the source of a definition, that is indicated.

### 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.

\* **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.

**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 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 or INVOKE statement, used for passing values to a called program or an invoked method.

\* **arithmetic expression.** An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

\* **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).

**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 present.
- 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 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
- Unicode UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

**big-endian.** The default format that the mainframe and the AIX 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*.

**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.

**Btrieve file system.** A key-indexed record management system that allows applications to manage records by key value, sequential access method, or random access method. Btrieve is IBM's name for the Pervasive.SQL file system, a separate product available from Pervasive Software. IBM COBOL for Windows supports COBOL sequential and indexed file input-output language through the Btrieve file system.

**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. (Sun)

## C

**callable services.** In Language Environment, a set of services that a COBOL program can invoke by using the conventional Language Environment-defined call interface. All programs that share the Language Environment conventions can use these services.

**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.

**cataloged procedure.** A set of job control statements that are placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors in coding JCL.

**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 definition.** The COBOL source unit that defines a class.

**class hierarchy.** A tree-like structure that shows relationships among object classes. It places one class at the top and one or more layers of classes below it. Synonymous with *inheritance hierarchy*.

\* **class identification entry.** An entry in the CLASS-ID paragraph of the IDENTIFICATION DIVISION; this entry contains clauses that specify the class-name and assign selected attributes to the class definition.

**class-name (object-oriented).** The name of an object-oriented COBOL class definition.

\* **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.

**class object.** The runtime object that represents a class.

\* **clause.** An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

**client.** In object-oriented programming, a program or method that requests services from one or more methods in a class.

\* **COBOL character set.** The set of characters used in writing COBOL syntax. The complete COBOL character set consists of the characters listed below:

Character	Meaning
0,1, . . . ,9	Digit
A,B, . . . ,Z	Uppercase letter
a,b, . . . ,z	Lowercase letter
	Space
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (virgule, slash)
=	Equal sign
\$	Currency sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol
:	Colon

\* **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 can be any combination of characters from the character set of the computer.

\* **comment line.** A source program line represented by an asterisk (\*) 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. The 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 printing the comment.

\* **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 YYYYXX, and the other has YYYY.
  - 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 YYYY.
- **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 YYYY is compatible with:
    - Another windowed date field with DATE FORMAT YYYY
    - An expanded date field with DATE FORMAT YYYYXX

\* **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.

\* **compile time.** The time at which COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

**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 Windows has one compiler directive, CALLINTERFACE. You can code CALLINTERFACE directives within a program to use specific interface conventions for specific CALL statements.

\* **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 Sun Microsystems, Inc.'s architecture for creating components.

\* **computer-name.** A system-name that identifies the computer where the program is to be compiled or run.

**condition.** 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.** A status of a program at run time for which a truth value can be determined. When used in these language specifications 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 and class definitions.

**CONSOLE.** A COBOL environment-name associated with the operator console.

**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.

## 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 or method that describes the data to be processed by the program or method: 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:

- 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 Windows Language Reference*).

The term *date field* refers to both *expanded date field* and *windowed date field*. See also *nodate*.



**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 Windows Language Reference*).

**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.

\* **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.

**DLL.** See *dynamic link library (DLL)*.

**DLL application.** An application that references imported programs, functions, or variables.

**DLL linkage.** A CALL in a program that has been compiled with the DLL and NODYNAM options; the CALL resolves to an exported name in a separate module, or to an INVOKE of a method that is defined in a separate module.

**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 definition (DTD).** The grammar for a class of XML documents. See *XML type definition*.

**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.

\* **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.

**dynamic link library (DLL).** A file that contains executable code and data that are bound to a program at load time or run time, rather than during linking. Several applications can share the code and data in a DLL simultaneously. Although a DLL is not part of the executable (.EXE) file for a program, it can be required for an executable file to run properly.

\* **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.

**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 (virgule, 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.

**encapsulation.** [In object-oriented programming, the technique that is used to hide the inherent details of an object. The object provides an interface that queries and manipulates the data without exposing its underlying structure. Synonymous with *information hiding*.

**encoding unit.** See *character encoding unit*.

**end class marker.** A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:  
END CLASS *class-name*.

**end method marker.** A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker is:  
END METHOD *method-name*.

\* **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, class definition, or method definition. 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.

**EXE.** See *executable file (EXE)*.

**executable file (EXE).** A file that contains programs or commands that perform operations or actions to be taken.

**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 Standard COBOL 85.

**external code page.** For ASCII 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

**factory data.** Data that is allocated once for a class and shared by all instances of the class. Factory data is declared in the WORKING-STORAGE SECTION of the DATA DIVISION in the FACTORY paragraph of the class definition, and is equivalent to Java private static data.

**factory method.** A method that is supported by a class independently of an object instance. Factory methods are declared in the FACTORY paragraph of the class definition, and are equivalent to Java public static methods. They are typically used to customize the creation of objects.

\* **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 present, 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 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.

\* **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 file label or data-set label that precedes the data records on a unit of recording media. (2) Synonym for *beginning-of-file label*.

**hide.** To redefine a factory or static method (inherited from a parent class) in a subclass.

**hierarchical file system.** A collection of files and directories that are organized in a hierarchical structure and can be accessed by using z/OS UNIX System Services.

\* **high-order end.** The leftmost character of a string of characters.

# I

**IBM COBOL extension.** COBOL syntax and semantics supported by IBM compilers in addition to those described in Standard COBOL 85.

**ICU.** See *International Components for Unicode (ICU)*.

**IDENTIFICATION DIVISION.** One of the four main component parts of a COBOL program, class definition, or method definition. The IDENTIFICATION DIVISION identifies the program name, class name, or method name. 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.

**IGZCBSO.** The COBOL for Windows bootstrap routine. It must be link-edited with any module that contains a COBOL for Windows program.

\* **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.

**inheritance.** A mechanism for using the implementation of a class as the basis for another class. By definition, the inheriting class conforms to the inherited classes. COBOL for Windows does not support *multiple inheritance*; a subclass has exactly one immediate superclass.

**inheritance hierarchy.** See *class hierarchy*.

\* **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 or method 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.

**instance data.** Data that defines the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION in the OBJECT paragraph of the class definition. The state of an object also includes the state of the instance variables introduced by classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

\* **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 allows the application developer to 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 Windows.

\* **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 phrase 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.

**is-a.** A relationship that characterizes classes and subclasses in an inheritance hierarchy. Subclasses that have an is-a relationship to a class inherit from that class.

**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 Sun Microsystems, Inc. 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. (Sun)

**Java Native Interface (JNI).** A programming interface that allows Java code that runs inside a Java virtual machine (JVM) to 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.

**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 reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

**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 an executable file.

**LINKAGE SECTION.** The section in the DATA DIVISION of the called program or invoked method that describes data items available from the calling program or invoking method. Both the calling program or invoking method and the called program or invoked method 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.

## 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.

**method.** Procedural code that defines an operation supported by an object and that is executed by an INVOKE statement on that object.

\* **method definition.** The COBOL source code that defines a method.

\* **method identification entry.** An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION; this entry contains a clause that specifies the method-name.

**method invocation.** A communication from one object to another that requests the receiving object to execute a method.

**method-name.** The name of an object-oriented operation. When used to invoke the method, the name can be an alphanumeric or national literal or a category alphanumeric or category national data item. When used in the METHOD-ID paragraph to define the method, the name must be an alphanumeric or national literal.

\* **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 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.

**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 position.** See *character position*.

**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 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.

**native method.** A Java method with an implementation that is written in another programming language, such as COBOL.

\* **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:

- 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. It 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.** An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior. Each object in the class is said to be an instance of the class.

**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 instance.** See *object*.

\* **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-oriented programming.** A programming approach based on the concepts of encapsulation and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates

on the data objects that comprise the problem and how they are manipulated, not on how something is accomplished.

**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 or class definition. Where there is no danger of ambiguity, the word *program* can be used in place of *object program*.

**object reference.** A value that identifies an instance of a class. If the class is not specified, the object reference is universal and can apply to instances of any class.

\* **object time.** The time at which an object program is executed. Synonymous with *run time*.

\* **obsolete element.** A COBOL language element in Standard COBOL 85 that was deleted from Standard COBOL 2002.

**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 PICS9.
 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 present each time the object program is run. The object program causes an interrogation for the presence or absence of the file.

\* **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.

**overload.** To define a method with the same name as another method that is available in the same class, but with a different signature. See also *signature*.

**override.** To redefine an instance method (inherited from a parent class) in a subclass.

## P

**package.** A group of related Java classes, which can be imported individually or as a whole.

**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 n 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)  
CLASS-ID. (Class IDENTIFICATION DIVISION)  
METHOD-ID. (Method 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 or Class  
CONFIGURATION SECTION)  
FILE-CONTROL.  
I-O-CONTROL.

\* **paragraph-name.** A user-defined word that identifies and begins a paragraph in the PROCEDURE DIVISION.

**parameter.** (1) Data passed between a calling program and a called program. (2) A data element in the USING phrase of a method invocation. Arguments provide additional information that the invoked method can use to perform the requested operation.

**Pervasive.SQL file system.** A relational database file system that can be accessed from the Btrieve interface. See *Btrieve file system*.

\* **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.

**preinitialization.** The initialization of the COBOL runtime environment in preparation for multiple calls from programs, especially non-COBOL programs. The environment is not terminated until an explicit termination.

\* **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.

**private.** As applied to factory data or instance data, accessible only by methods of the class that defines the data.

\* **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 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 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

\* **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-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 a 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 allows more than one user to share a single copy of a load module. The COBOL for Windows compiler always produces reentrant code.

\* **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, nonnumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, nonnumeric 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.

**ring.** In the COBOL editor, a set of files that are available for editing so that you can easily move between them.

**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, including the full Standard COBOL 85 sequential I/O language and all of the extensions described in *COBOL for Windows Language Reference*, unless exceptions are explicitly noted. An RSD file supports all COBOL data types in fixed-length records, can be edited by most file editors, and can be read by programs written in other languages.

\* **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 or INVOKE 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.

**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.

\* **77-level-description-entry.** A data description entry that describes a noncontiguous data item with the level-number 77.

\* **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.** (1) The name of an operation and its parameters. (2) The name of a method and the number and types of its formal 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.** Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

\* **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 or a class definition. 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 (virgule, slash)
=	Equal sign
\$	Currency sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol
:	Colon

**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.

**Standard COBOL 85.** 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*

\* **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 and PL/I. This system supports sequential, relative, and indexed files, including the full Standard COBOL 85 input and output language and all of the extensions described in *COBOL for Windows Language Reference*, unless exceptions are explicitly noted.

**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.

\* **subclass.** A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheritor or inheriting class; the superclass is the inheritee or inherited class.

\* **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.

\* **superclass.** A class that is inherited by another class. See also *subclass*.

**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.

\* **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 nonnumeric 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 nonnumeric 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.

**token highlighting.** In the COBOL editor, a feature that enables you to view the token types of the programming language in different colors and fonts. This feature makes the structure of the program more obvious. You use the Token Attributes window to customize the appearance of the types of tokens.

**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 file or data-set 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.

**typed object reference.** A data-name that can refer only to an object of a specified class or any of its subclasses.

## 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 Windows supports Unicode using UTF-16 in little-endian format as the representation for the national data type.

**unit.** A module of direct access, the dimensions of which are determined by IBM.

**universal object reference.** A data-name that can refer to an object of any class.



\* **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.

\* **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 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.

## 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, 08 could be interpreted as 2008. 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 used by end users 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 allows programs to 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 type definition.** 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.

## Y

**year field expansion.** Explicitly expanding date fields that contain two-digit years to contain four-digit years in files and databases and then using 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).

---

## List of resources

---

### COBOL for Windows

*Language Reference*, SC23-8560

*Programming Guide*, SC23-8559

---

### Related publications

#### COBOL

*COBOL Millennium Language Extensions Guide*, GC26-9266

#### DB2 Universal Database

*Application Development Guide: Building and Running Applications*, SC09-4825

*Application Development Guide: Programming Client Applications*, SC09-4826

*Application Development Guide: Programming Server Applications*, SC09-4827

*Command Reference*, SC09-4828

*SQL Reference Volume 1*, SC09-4844

*SQL Reference Volume 2*, SC09-4845

#### Java

*The Java Language Specification, Second Edition*, by Gosling et al., [java.sun.com/docs/books/jls/second\\_edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html)

*The Java Native Interface*, [java.sun.com/j2se/1.3/docs/guide/jni/index.html](http://java.sun.com/j2se/1.3/docs/guide/jni/index.html)

*The Java 2 Enterprise Edition Developer's Guide*, [java.sun.com/j2ee/sdk\\_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html](http://java.sun.com/j2ee/sdk_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html)

*Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201

#### TXSeries for Multiplatforms V5.1

*CICS Administration Guide for Windows Systems*, SC09-4456

*CICS Administration Reference*, SC09-4459

*CICS Application Programming Guide*, SC09-4460

*CICS Application Programming Reference*, SC09-4461

*CICS Problem Determination Guide*, SC09-4465

*Planning and Installation Guide for Windows Systems*, SC09-4451

#### TXSeries for Multiplatforms V6.1

*CICS Administration Guide*, SC34-6746

*CICS Administration Reference*, SC34-6641

*CICS Application Programming Guide*, SC34-6634

*CICS Application Programming Reference*, SC34-6640

*CICS Problem Determination Guide*, SC34-6636

*CICS Installation Guide*, SC34-6632

#### Unicode and character representation

*Unicode*, [www.unicode.org/](http://www.unicode.org/)

*Character Data Representation Architecture: Reference and Registry*, SC09-2190

#### WebSphere® Application Server for z/OS

*Applications*, SA22-7959

#### XML

*Extensible Markup Language (XML)*, [www.w3.org/XML/](http://www.w3.org/XML/)

*XML specification*, [www.w3.org/TR/REC-xml/](http://www.w3.org/TR/REC-xml/)

#### Other

*Btrieve Programmer's Guide*



---

# Index

## Special characters

- `_iwbzGetCCSID`: convert code-page ID to CCSID
  - example 189
  - syntax 189
- `_iwbzGetLocaleCP`: get locale and EBCDIC code-page values
  - example 189
  - syntax 188
- `-? cob2 option` 208
- `-# cob2 option` 208
- `-b cob2 option` 207
- `-c cob2 option` 206
- `-cmain cob2 option` 207
- `-comprc_ok cob2 option` 206
- `-dll cob2 option` 206
- `-g cob2 option`
  - for debugging 208
- `-h cob2 option` 208
- `-host cob2 option`
  - effect on command-line arguments 482
  - effect on compiler options 206
  - for host data format 206
  - with OO applications 561
- `-I cob2 option`
  - searching copybooks 207
- `-imp cob2 option`, specifying import library with 207
- `-main cob2 option`
  - specifying main program 208
- `-q cob2 option` 207
- `-s cob2 option`
  - specifying stack space 208
- `-v cob2 option` 208
- `? cob2 option` 208
- `/HEAP cob2 option`, specifying heap space with 208
- `.adt file` 227
- `.asm file` 249
- `.CBL file extension` 209
- `.DEF as linker parameter` 209
- `.DLL as linker parameter` 209
- `.EXE as linker parameter` 209
- `.EXP as linker parameter` 209
- `.exp file`, creating with cob2 206
- `.IMP as linker parameter` 209
- `.LIB as linker parameter` 209
- `.lib file`, creating with cob2 206
- `.LST file extension` 205
- `.MAP as linker parameter` 210
- `.OBJ as linker parameter` 210
- `*CBL statement` 273
- `*CONTROL statement` 273
- `>>CALLINT` compiler directive
  - for calling C/C++ 463
- `>>CALLINT statement`
  - description 273
  - example of ensuring OPTLINK 476

## Numerics

- 16-MB line
- performance options 546

## A

- abends, using ERRCOUNT runtime option to induce 294
- ACCEPT statement
  - assigning input 34
  - under CICS 328
  - using in GUI applications 199
- ACCEPT statement, environment variables used in 199
- accessibility of this document xiii
- accessing files using environment variables 196
- active locale 179
- ADATA compiler option 227
- adding external program reference to object module 486
- adding records to files
  - overview 129
  - randomly or dynamically 130
  - sequentially 129
- adding stub files 494
- ADDRESS special register, CALL statement 468
- addresses
  - incrementing 473
  - NULL value 472
  - passing between programs 472
  - passing entry-point addresses 475
- ADEXIT suboption of EXIT compiler option 244
- ADEXIT suboption of EXIT option 241
- AIX, porting to 451
- ALL subscript
  - examples 78
  - processing table elements iteratively 78
  - table elements as function arguments 54
- allocating heap space 208
- allocating stack space 208
- ALPHABET clause, establishing collating sequence with 8
- alphabetic data
  - comparing to national 174
  - MOVE statement with 32
- alphanumeric comparison 86
- alphanumeric data
  - comparing
    - effect of collating sequence 186
    - effect of ZWB 271
    - to national 174
  - converting
    - to national with MOVE 168
    - to national with NATIONAL-OF 169

- alphanumeric data (*continued*)
  - MOVE statement with 32
- alphanumeric date fields,
  - contracting 533
- alphanumeric group item
  - a group without GROUP-USAGE NATIONAL 25
  - definition 24
- alphanumeric literals
  - control characters within 26
  - description 25
  - with multibyte content 176
- alphanumeric-edited data
  - initializing
    - example 29
    - using INITIALIZE 68
  - MOVE statement with 32
- alternate collating sequence
  - choosing 139
  - example 9
- alternate file system
  - file system ID 197
  - using environment variables 197
- alternate index
  - definition 120
- ANNUITY intrinsic function 56
- APOST compiler option 257
- applications, porting
  - differences between platforms 445
  - language differences 445
  - mainframe to workstation
    - choosing compiler options 445
    - running mainframe applications on the workstation 447
  - using COPY to isolate platform-specific code 446
  - Windows to AIX 451
  - workstation to mainframe
    - workstation-only language features 451
    - workstation-only names 451
- arguments
  - describing in calling program 469
  - passing BY VALUE 469
  - specifying OMITTED 470
  - testing for OMITTED arguments 470
  - to main program 482
- ARITH compiler option
  - description 228
- arithmetic
  - calculation on dates
    - convert date to COBOL integer format (CEECLDY) 587
    - convert date to Lilian format (CEEDAYS) 597
    - convert timestamp to number of seconds (CEESECS) 618
    - get current Greenwich Mean Time (CEEGMT) 603
- COMPUTE statement simpler to code 52

- arithmetic (*continued*)
  - error handling 146
  - with intrinsic functions 53
- arithmetic comparisons 57
- arithmetic evaluation
  - conversions and precision 49
  - data format conversion 49
  - examples 56, 58
  - fixed-point contrasted with floating-point 56
  - intermediate results 569
  - performance tips 539
  - precedence 53, 571
  - precision 569
- arithmetic expression
  - as reference modifier 102
  - description of 53
  - in nonarithmetic statement 577
  - in parentheses 53
  - with MLE 526
- arithmetic operation
  - with MLE 525, 526
- arrays
  - COBOL 37
  - Java
    - declaring 437
    - manipulating 438
- ASCII
  - converting to EBCDIC 107
  - multibyte portability 449
  - SBCS portability 447
- asm file 249
- assembler
  - from LIST option 544
  - programs
    - listing of 249, 544
- assembler language programs
  - debugging 318
- ASSIGN clause 10
- assigning values 27
- assignment-name, environment variable 196
- assumed century window for nondates 524
- AT END (end-of-file) 147
- attribute methods 401
- ATTRIBUTE-CHARACTER XML event 353
- ATTRIBUTE-CHARACTERS XML event 353
- ATTRIBUTE-NAME XML event 353
- ATTRIBUTE-NATIONAL-CHARACTER XML event 353
- avoiding coding errors 537

## B

- backward branches, avoid 538
- Base class
  - equating to java.lang.Object 392
  - using for java.lang.Object 392
- base locator 312
- BASE statement 491
- basic document encoding 364
- batch compilation 258
- batch debugging, activating 294

- big-endian
  - definition 45
  - format for data representation 229
  - representation of integers 448, 452
  - representation of national characters 452
- big-endian, converting to little-endian 159
- BINARY compiler option 229
- binary data item
  - byte reversal 45
  - general description 44
  - intermediate results 574
  - synonyms 42
  - using efficiently 44, 540
- binary data, data representation 229
- binary search
  - description 77
  - example 77
- BLANK WHEN ZERO clause
  - coded for numeric data 161
  - example with numeric-edited data 41
- branch, implicit 90
- Btrieve files
  - file status indicators 149
  - identifying 113
  - processing 115
- BY CONTENT 467
- BY REFERENCE 467
- BY VALUE
  - description 467
  - restrictions 469
  - valid data types 469
- byte order mark 159
- byte-reversed integers, effect on portability 448

## C

- C/C++
  - and COBOL 462
  - communicating with COBOL 462
  - data types, correspondence with COBOL 464
  - DLL called from COBOL, example 465
  - linkage for calls from COBOL 463
  - multiple calls to a COBOL program 462
  - variable parameter list 462
- C/C++ calling convention 230
- call interface conventions
  - C/C++ 230
  - CDECL 459
  - indicating with CALLINT 230
  - indicating with ENTRYINT 239
  - OPTLINK 460
  - overview 459
  - STDCALL 461
  - SYSTEM 461
  - with ODBC 345
- CALL statement
  - BY CONTENT 467
  - BY REFERENCE 467
  - BY VALUE
    - description 467

- CALL statement (*continued*)
  - BY VALUE (*continued*)
    - restrictions 469
  - CALL identifier 458
  - CALL literal 458
  - effect of CALLINT option 229
  - exception condition 152
  - for error handling 152
  - handling of program-name in 255
  - overflow condition 152
  - RETURNING 478
  - to invoke date and time services 551
  - USING 469
  - with DYNAM 238
  - with ON EXCEPTION 152
  - with ON OVERFLOW 19, 152
- callable services
  - \_iwbzGetCCSID: convert code-page ID to CCSID 189
  - \_iwbzGetLocaleCP: get locale and EBCDIC code-page values 188
  - CEECLDLY: convert date to COBOL integer format 587
  - CEEDATE: convert Lilian date to character format 590
  - CEEDATM: convert seconds to character timestamp 594
  - CEEDAYS: convert date to Lilian format 597
  - CEEDYWK: calculate day of week from Lilian date 601
  - CEEGMT: get current Greenwich Mean Time 603
  - CEEGMTO: get offset from Greenwich Mean Time 605
  - CEEISEC: convert integers to seconds 607
  - CEELOCT: get current local time 609
  - CEEQCEN: query the century window 611
  - CEESCEN: set the century window 613
  - CEESECI: convert seconds to integers 615
  - CEESECS: convert timestamp to number of seconds 618
  - CEEUTC: get Coordinated Universal Time 622
  - IGZEDT4: get current date with four-digit year 622
- calling conventions, overview 459
- CALLINT compiler option
  - description 229
  - for calling C/C++ 463
  - OPTLINK with CALL identifier to nested programs 454
- CALLINT statement
  - description 273
  - example of ensuring OPTLINK 476
- calls
  - dynamic 458
  - exception condition 152
  - LINKAGE SECTION 470
  - OMITTED arguments 470
  - overflow condition 152
  - passing arguments 469
  - passing data 467



- calls (*continued*)
  - receiving parameters 469
  - recursive 466
  - static 458
  - to date and time services 551
  - to JNI services 431
- CANCEL statement
  - handling of program-name in 255
- case structure, EVALUATE statement
  - for 83
- CBL file extension 209
- CBL statement
  - description 273
  - specifying compiler options 203
- CCSID
  - conflict in XML documents 370
  - definition 159
  - of PARSE statement 351
  - of XML documents 351, 364, 365
- CDECL interface convention
  - description 459
  - specified with CALLINT 230
- CDECL suboption of CALLINT compiler option 230
- CEECEBLDY: convert date to COBOL integer format
  - example 587
  - syntax 587
- CEEDATE: convert Lilian date to character format
  - example 591
  - syntax 590
  - table of sample output 593
- CEEDATM: convert seconds to character timestamp
  - CEESECI 615
  - example 595
  - syntax 594
  - table of sample output 596
- CEEDAYS: convert date to Lilian format
  - example 599
  - syntax 597
- CEEDYWK: calculate day of week from Lilian date
  - example 601
  - syntax 601
- CEEGMT: get current Greenwich Mean Time
  - example 604
  - syntax 603
- CEEGMTO: get offset from Greenwich Mean Time
  - example 606
  - syntax 605
- CEEISEC: convert integers to seconds
  - example 608
  - syntax 607
- CEELOCT: get current local time
  - example 610
  - syntax 609
- CEEQCEN: query the century window
  - example 612
  - syntax 611
- CEESCEN: set the century window
  - example 614
  - syntax 613
- CEESECI: convert seconds to integers
  - example 616
  - syntax 615
- CEESECS: convert timestamp to number of seconds
  - example 620
  - syntax 618
- century window
  - assumed for nondates 524
  - CEECEBLDY 589
  - CEEDAYS 599
  - CEEQCEN 611
  - CEESCEN 613
  - CEESECS 620
  - example of querying and changing 557
  - fixed 516
  - overview 557
  - sliding 516
- chained-list processing
  - example 473
  - overview 472
- changing
  - characters to numbers 105
  - file-name 10
  - title on source listing 6
- CHAR compiler option
  - description 230
  - multibyte portability 449
  - SBCS portability 447
- CHAR intrinsic function, example 108
- character set, definition 159
- character timestamp
  - converting Lilian seconds to (CEEDATM) 594
  - example 595
  - converting to COBOL integer format (CEECEBLDY) 587
  - example 589
  - converting to Lilian seconds (CEESECS) 618
  - example 618
- CHECK runtime option 293
  - performance considerations 546
  - reference modification 101
- checking errors, flagging at run time 293
- checking for valid data
  - conditional expressions 86
  - numeric 51
- Chinese GB 18030 data
  - processing 171
- CICS
  - coding programs 327
  - commands relevant to COBOL
    - cicsmap 327
    - cicstcl 327
  - compiler options 330
  - debugging programs 332
  - developing programs for 327
  - DFHCOMMAREA parameter for dynamic calls 329
  - DFHEIBLK parameter for dynamic calls 329
  - dynamic calls under 329
  - dynamic link libraries 329
  - host data format not supported 328
- CICS (*continued*)
  - integrated translator
    - advantages 331
    - invoking 327
    - overview 331
  - portability considerations 328
  - restrictions
    - DYNAM compiler option 239
    - nested programs 327
    - OO programs 327, 387
    - overview 327
    - preinitialization 507
    - separate translator 331
  - runtime options 330
  - separate translator
    - invoking 327
    - restrictions 331
  - system date, getting 328
  - TRAP runtime option, effect of 295
- CICS compiler option
  - description 232
  - effect of cicstcl -p command 331
  - enables integrated translator 331
  - multioption interaction 227
  - specifying suboptions 233
- cicsmap command 327
- cicstcl command
  - p flag for integrated translator 331
  - translate, compile, and link CICS programs 327
- class
  - defining 390
  - definition of 387
  - factory data 419
  - instance data 394
  - instantiating
    - COBOL 413
    - Java 412
  - name
    - external 393, 405
    - in a program 392
  - object, obtaining reference with JNI 432
  - user-defined 10
- class condition
  - testing
    - for DBCS 177
    - for Kanji 177
    - for numeric 51
    - overview 86
    - validating data 299
- CLASSPATH environment variable
  - description 197
  - specifying location of Java classes 221
- client
  - defining 403
  - definition of 403
- cob2 command
  - command-line argument format 482
  - description 201
  - examples of compiling 202
  - examples of linking 211
  - file extensions supported 209
  - for compiling OO applications 219
  - for linking OO applications 220

- cob2 command (*continued*)
  - options
    - b to pass options to linker 210
    - host 206, 482, 561
    - description 206
- COBCPYEXT environment variable 195
- COBJVMINIOPTIONS environment variable
  - description 197
  - specifying JVM options 223
- COBLSTDIR environment variable 195
- COBMSG environment variable 197
- COBOL
  - and C/C++ 462
  - and Java 431
  - compiling 219
  - linking 220
  - running 221
  - structuring applications 428
  - calling C/C++ DLL, example 465
  - data types, correspondence with C/C++ 464
  - linkage for calling C/C++ 463
  - object-oriented
    - compiling 219
    - linking 220
    - running 221
  - parameter list for calling C/C++ 462
- COBOL client
  - example 422
  - example of passing object references 409
- COBOL for Windows
  - runtime messages 707
  - setting environment variables for 194
- COBOL terms 23
- COBOPT environment variable 195
- COBPATH environment variable
  - CICS dynamic calls 329
  - description 195, 197
- COBRTOPT environment variable 197
- code
  - copy 549
  - optimized 544
- code page
  - accessing 189
  - ASCII 180
  - conflict in XML documents 370
  - definition 159
  - EBCDIC 180
  - euro currency support 59
  - for alphabetic data item 180
  - for alphanumeric data item 180
  - for DBCS data item 180
  - for national data item 181
  - overriding 170
  - querying 189
  - specifying 365
  - system default 182
  - using characters from 180
  - valid 183
- code point, definition 159
- coded character set
  - definition 159
  - in XML documents 365
- coding
  - class definition 390

- coding (*continued*)
  - clients 403
  - condition tests 87
  - constructor methods 420
  - DATA DIVISION 11
  - decisions 81
  - efficiently 537
  - ENVIRONMENT DIVISION 7
  - EVALUATE statement 83
  - factory definition 418
  - factory methods 420
  - file input/output 119
  - for files
    - example 124
    - overview 123
  - IDENTIFICATION DIVISION 5
  - IF statement 81
  - input/output
    - example 124
    - overview 123
  - instance methods 395, 417
  - interoperable data types with Java 436
  - loops 89
  - OO programs
    - overview 387
  - PROCEDURE DIVISION 16
  - programs to run under CICS
    - overview 327
    - system date, getting 328
  - programs to run under DB2
    - overview 321
  - restrictions under CICS 327
  - simplifying 549
  - SQL statements 322
  - subclasses
    - example 417
    - overview 414
  - tables 61
  - techniques 11, 537
  - test conditions 87
- collating sequence
  - alphanumeric 186
  - alternate
    - choosing 139
    - example 9
  - ASCII 8
  - binary for national keys 139
  - binary for national sort or merge keys 187
  - COLLSEQ effect on alphanumeric and DBCS operands 233
  - controlling 185
  - DBCS 186
  - EBCDIC 8
  - HIGH-VALUE 8
  - intrinsic functions and 188
  - ISO 7-bit code 8
  - LOW-VALUE 8
  - MERGE 8, 140
  - national 187
  - NATIVE 8
  - NCOLLSEQ effect on national operands 252
  - nonnumeric comparisons 8
  - ordinal position of a character 107
  - portability considerations 448

- collating sequence (*continued*)
  - SEARCH ALL 8
  - SORT 8, 140
  - specifying 8
  - STANDARD-1 8
  - STANDARD-2 8
  - symbolic characters in the 9
- COLLATING SEQUENCE phrase
  - does not apply to national keys 139
  - effect on sort and merge keys 186
  - overrides PROGRAM COLLATING SEQUENCE clause 8, 140
  - portability considerations 448
  - use in SORT or MERGE 140
- COLLSEQ compiler option
  - description 233
  - effect on alphanumeric collating sequence 185
  - effect on DBCS collating sequence 187
  - portability considerations 448
- columns in tables 61
- command file for setting environment variables 194
- command prompt, defining environment variables 193
- command-line arguments
  - example 482
  - using 482
- COMMENT XML event 353
- COMMON attribute 6, 454
- COMP (COMPUTATIONAL) 44
- COMP-1 (COMPUTATIONAL-1)
  - format 46
  - performance tips 540
- COMP-2 (COMPUTATIONAL-2)
  - format 46
  - performance tips 540
- COMP-3 (COMPUTATIONAL-3) 46
- COMP-4 (COMPUTATIONAL-4) 44
- COMP-5 (COMPUTATIONAL-5) 45
- comparing data items
  - alphanumeric
    - effect of collating sequence 186
    - effect of COLLSEQ 233
  - date fields 520
  - DBCS
    - effect of collating sequence 186
    - effect of COLLSEQ 233
    - literals 176
    - to alphanumeric groups 187
    - to national 187
  - national
    - effect of collating sequence 187
    - effect of NCOLLSEQ 173
    - overview 172
    - to alphabetic, alphanumeric, or DBCS 174
    - to alphanumeric groups 174
    - to numeric 173
    - two operands 172
  - object references 406
  - zoned decimal and alphanumeric, effect of ZWB 271
- compatibility mode 39, 569
- compatible dates
  - in comparisons 520



- compatible dates (*continued*)
  - with MLE 521
- compilation
  - statistics 309
- COMPILE compiler option
  - description 235
  - use NOCOMPILE to find syntax errors 302
- compile-time considerations
  - cob2 command options 206
  - compiler-directed errors 205
  - compiling programs 206
  - compiling programs without linking 206
  - display cob2 help 208
  - display compile and link steps 208
  - error messages
    - determining what severity level to produce 245
    - severity levels 204
  - executing compile and link steps after display 208
- compiler
  - calculation of intermediate results 570
  - date-related messages, analyzing 530
  - generating list of error messages 204
  - invoking 201
  - limits
    - DATA DIVISION 11
  - messages
    - choosing severity to be flagged 303
    - determining what severity level to produce 245
    - embedding in source listing 303
    - severity levels 204
- compiler listings
  - getting 307
  - specifying output directory 195
- compiler options
  - abbreviations 225
  - ADATA 227
  - APOST 257
  - ARITH 228
  - BINARY 229
  - CALLINT 229
  - CHAR 230
  - CICS 232
  - COLLSEQ 233
  - COMPILE 235
  - conflicting 226
  - CURRENCY 236
  - DATEPROC 237
  - DIAGTRUNC 238
  - DYNAM 238, 546
  - ENTRYINT 239
  - EXIT 240
  - FLAG 245, 303
  - FLAGSTD 246
  - FLOAT 247
  - for CICS 330
  - for debugging
    - overview 301
    - THREAD restriction 300
  - for portability 445
  - LIB 248
- compiler options (*continued*)
  - LINECOUNT 249
  - LIST 249, 307
  - LSTFILE 250
  - MAP 250, 306, 307
  - MDECK 251
  - NCOLLSEQ 252
  - NOCOMPILE 302
  - NSYMBOL 253
  - NUMBER 253, 308
  - on compiler invocation 309
  - OPTIMIZE 254, 544, 546
  - PGMNAME 255
  - PROBE 256
  - QUOTE 257
  - SEPOBJ 258
  - SEQUENCE 259
  - SIZE 260
  - SOSI 260
  - SOURCE 261, 307
  - SPACE 262
  - specifying
    - cob2 command 201
    - environment variable 195
    - in command or batch file 203
    - using COBOPT 195
  - specifying with PROCESS (CBL) 203
  - SQL
    - coding suboptions 323
    - description 262
  - SSRANGE 263, 302, 546
  - status 309
  - table of 225
  - TERMINAL 264
  - TEST 264, 306, 546
  - THREAD
    - debugging restriction 300
    - description 265
    - performance 546
  - TRUNC 266, 546
  - VBREF 269, 307
  - WSCLEAR 269
  - XREF 269, 305
  - YEARWINDOW 271
  - ZWB 271
- compiler-directing statements
  - description 273
  - overview 19
- compiling
  - DLLs, example 489
  - OO applications
    - cob2 command 219
    - example 221
- compiling and linking
  - OO applications
    - cob2 command 220
    - example 221
- completion code
  - merge 141
  - sort 141
- complex OCCURS DEPENDING ON
  - basic forms of 579
  - complex ODO item 579
  - variably located data item 579
  - variably located group 579
- computation
  - arithmetic data items 540
- computation (*continued*)
  - constant data items 539
  - duplicate 539
  - of indexes 66
  - of subscripts 542
  - COMPUTATIONAL (COMP) 44
  - COMPUTATIONAL-1 (COMP-1)
    - format 46
    - performance tips 540
  - COMPUTATIONAL-2 (COMP-2)
    - format 46
    - performance tips 540
  - COMPUTATIONAL-3 (COMP-3)
    - date fields, potential problems 532
    - description 46
  - COMPUTATIONAL-4 (COMP-4) 44
  - COMPUTATIONAL-5 (COMP-5) 45
  - COMPUTE statement
    - assigning arithmetic results 34
    - simpler to code 52
  - computer, describing 7
  - concatenating data items (STRING) 93
  - condition handling
    - date and time services and 552
    - effect of ERRCOUNT 294
  - condition testing 87
  - condition-name 523
  - conditional expression
    - EVALUATE statement 81
    - IF statement 81
    - PERFORM statement 91
  - conditional statement
    - overview 18
    - with NOT phrase 18
    - with object references 406
  - CONFIGURATION SECTION 7
  - conflicting compiler options 226
  - conformance requirements
    - example of passing object references in INVOKE 409
    - RETURNING phrase of INVOKE 411
    - USING phrase of INVOKE 408
  - constants
    - computations 539
    - data items 538
    - definition 26
    - figurative, definition 26
  - contained program integration 545
  - CONTENT-CHARACTER XML
    - event 353
  - CONTENT-CHARACTERS XML
    - event 353
  - CONTENT-NATIONAL-CHARACTER XML
    - event 353
  - continuation
    - of program 146
    - syntax checking 235
  - CONTINUE statement 81
  - contracting alphanumeric dates 533
  - control
    - in nested programs 454
    - program flow 81
    - transfer 453
  - CONTROL statement 273
  - convert character format to Lilian date (CEEDAYS) 597

- convert Lilian date to character format (CEEDATE) 590
- converting data items
  - between code pages 107
  - between data formats 49
  - precision 49
  - reversing order of characters 105
  - to alphanumeric
    - with DISPLAY 35
    - with DISPLAY-OF 169
  - to Chinese GB 18030 from national 171
  - to integers with INTEGER, INTEGER-PART 102
  - to national
    - from Chinese GB 18030 171
    - from UTF-8 171
    - with ACCEPT 35
    - with MOVE 168
    - with NATIONAL-OF 169
  - to numbers with NUMVAL, NUMVAL-C 105
  - to uppercase or lowercase
    - with INSPECT 104
    - with intrinsic functions 105
  - to UTF-8 from national 171
  - with INSPECT 103
  - with intrinsic functions 104
- converting files to expanded date form, example 519
- CONVERTING phrase (INSPECT), example 104
- coprocessor, DB2
  - overview 321
  - using SQL INCLUDE with 322
- copy code, obtaining from user-supplied module 241
- copy libraries
  - example 550
- COPY name
  - file extensions searched 195
- COPY statement
  - description 275
  - example 550
  - nested 243, 549
  - uses for portability 446
- copybook 275
  - ODBC3D.CPY example 343
  - ODBC3EG.CPY 339
  - ODBC3P.CPY 340
  - search rules 275
  - using 549
  - with ODBC 338
- copybooks
  - library-name environment variable 195
  - searching for 207
  - specifying search paths with SYSLIB 196
  - with ODBC 338
- COUNT IN phrase
  - UNSTRING 95
  - XML GENERATE 384
- counting
  - characters (INSPECT) 103
  - generated XML characters 374

- creating
  - objects 412
  - variable-length tables 72
- cross-reference
  - data and procedure-names 305
  - embedded 307
  - list 269
  - program-name 315
  - special definition symbols 316
  - verb list 269
  - verbs 307
- cultural conventions, definition 179
- CURRENCY compiler option 236
- currency signs
  - euro 59
  - hexadecimal literals 59
  - multiple-character 59
  - using 59
- CURRENT-DATE intrinsic function
  - example 55
  - under CICS 329
- customizing
  - setting environment variables 193

## D

- data
  - concatenating (STRING) 93
  - efficient execution 537
  - format conversion 49
  - format, numeric types 42
  - grouping 471
  - incompatible 51
  - naming 12
  - numeric 39
  - passing 467
  - record size 12
  - splitting (UNSTRING) 95
  - validating 51
- data and procedure-name cross-reference, description 305
- data areas, dynamic 238
- DATA compiler option
  - performance considerations 546
- data definition 311
- data definition attribute codes 311
- data description entry 12
- DATA DIVISION
  - client 405
  - coding 11
  - description 11
  - factory data 419
  - factory method 421
  - FD entry 11
  - FILE SECTION 11
  - GROUP-USAGE NATIONAL clause 62
  - instance data 394, 416
  - instance method 397
  - limits 11
  - LINKAGE SECTION 15
  - listing 307
  - mapping of items 250, 307
  - OCCURS clause 61
  - OCCURS DEPENDING ON (ODO) clause 72
  - REDEFINES clause 69

- DATA DIVISION (*continued*)
  - restrictions 11
  - USAGE clause at the group level 25
  - USAGE IS INDEX clause 66
  - USAGE NATIONAL clause at the group level 164
  - WORKING-STORAGE SECTION 11
- data item
  - coding Java types 435
  - common, in subprogram linkage 469
  - concatenating (STRING) 93
  - converting characters (INSPECT) 103
  - converting characters to numbers 105
  - converting to uppercase or lowercase 105
  - converting with intrinsic functions 104
  - counting characters (INSPECT) 103
  - elementary, definition 24
  - evaluating with intrinsic functions 107
  - finding the smallest or largest item 108
  - group, definition 24
  - index, referring to table elements with 64
  - initializing, examples of 28
  - numeric 39
  - reference modification 99
  - referring to a substring 99
  - replacing characters (INSPECT) 103
  - reversing characters 105
  - splitting (UNSTRING) 95
  - unused 254
  - variably located 579
- data manipulation
  - character data 93
- DATA RECORDS clause 12
- data representation
  - compiler option affecting 229
  - portability 447
- data types, correspondence between COBOL and C/C++ 464
- data-name
  - cross-reference 314
  - in MAP listing 311
- date and time
  - format
    - converting from character format to COBOL integer format (CEECLDY) 587
    - converting from character format to Lilian format (CEEDAYS) 597
    - converting from integers to seconds (CEEISEC) 607
    - converting from Lilian format to character format (CEEDATE) 590
    - converting from seconds to character timestamp (CEEDATM) 594
    - converting from seconds to integers (CEESECI) 615
    - converting from timestamp to number of seconds (CEESECS) 618

- date and time (*continued*)
  - getting date and time (CEELOCT) 609
  - intrinsic functions 586
  - services
    - CEEGBLDY: convert date to COBOL integer format 587
    - CEEDATE: convert Lilian date to character format 590
    - CEEDATM: convert seconds to character timestamp 594
    - CEEDAYS: convert date to Lilian format 597
    - CEEDYWK: calculate day of week from Lilian date 601
    - CEEGMT: get current Greenwich Mean Time 603
    - CEEGMTO: get offset from Greenwich Mean Time 605
    - CEEISEC: convert integers to seconds 607
    - CEELOCT: get current local time 609
    - CEEQCEN: query the century window 611, 612
    - CEESCEN: set the century window 613
    - CEESECI: convert seconds to integers 615
    - CEESECS: convert timestamp to number of seconds 618
    - CEEUTC: get Coordinated Universal Time 622
    - condition feedback 553
    - condition handling 552
    - examples of using 552
    - feedback code 551
    - invoking with a CALL statement 551
    - list of 585
    - overview 585
    - performing calculations with 552
    - picture strings 555
    - return code 551
    - RETURN-CODE special register 551
    - syntax 618
  - date arithmetic 527
  - date comparisons 520
  - date field expansion
    - advantages 516
    - description 518
  - date fields, potential problems with 532
  - DATE FORMAT clause
    - cannot use with national data 514
    - use for automatic date recognition 513
  - date information, formatting 198
  - date operations
    - finding date of compilation 111
    - intrinsic functions for 36
  - date processing with internal bridges, advantages 516
  - date windowing
    - advantages 516
    - example 517, 523
    - how to control 528
- date windowing (*continued*)
  - MLE approach 516
  - when not supported 522
- DATE-COMPILED paragraph 5
- DATE-OF-INTEGER intrinsic function 55
- DATEPROC compiler option
  - analyzing warning-level messages 530
  - description 237
- DATEVAL intrinsic function
  - example 530
  - using 529
- day of week, calculating with CEEDYWK 601
- DB2
  - bind file name 324
  - coding considerations 321
  - coprocessor
    - overview 321
    - using SQL INCLUDE with 322
  - ignored options 324
  - options 323
  - package name 324
  - precompiler requires NODYNAM 321
  - SQL statements
    - coding 322
    - overview 321
    - return codes 323
    - SQL INCLUDE 322
    - using binary data in 323
- DB2DBDFT environment variable 195, 323
- DB2INCLUDE environment variable 322
- DB2PATH environment variable
  - description 195
  - interaction with SYSLIB 322
- DBCS comparison 86
- DBCS data
  - comparing
    - effect of collating sequence 186
    - literals 176
    - to alphanumeric groups 187
    - to national 174, 187
  - converting
    - to national, overview 177
  - declaring 175
  - encoding 167
  - literals
    - comparing 176
    - description 26
    - maximum length 176
    - using 176
  - MOVE statement with 32
  - testing for 177
- DEBUG runtime option 294
- debugging
  - activating batch features 294
  - assembler 318
  - CICS programs 332
  - compiler options for
    - overview 301
    - THREAD restriction 300
  - idebug command 318
  - overview 297
  - producing symbolic information 208
- debugging (*continued*)
  - runtime options for 300
  - user exits 317
  - using COBOL language features 297
  - using the debugger 306
- debugging, language features
  - class test 299
  - debugging lines 300
  - debugging statements 300
  - declaratives 299
  - file status keys 298
  - INITIALIZE statements 299
  - scope terminators 298
  - SET statements 299
  - WITH DEBUGGING MODE clause 300
- declarative procedures
  - EXCEPTION/ERROR 148
  - USE FOR DEBUGGING 299
- DEF extension as linker parameter 209
- defining files
  - example 124
  - overview 123
- deleting records from files 130
- delimited scope statement
  - description of 18
  - nested 20
- depth in tables 63
- DESC suboption of CALLINT compiler option 230
- description file 215
- DESCRIPTION statement 491
- DESCRIPTOR suboption of CALLINT compiler option 230
- DFHCOMMAREA parameter
  - use in CICS dynamic calls 329
- DFHEIBLK parameter
  - use in CICS dynamic calls 329
- diagnostics, program 309
- DIAGTRUNC compiler option 238
- differences from host COBOL 561
- direct-access
  - direct indexing 66
- directories
  - adding a path to 207
  - for error listing file 205
  - for linker search 211
  - specifying paths for DLLs 197
- DISPLAY (USAGE IS)
  - encoding 167
  - external decimal 43
  - floating point 44
- display floating-point data (USAGE DISPLAY) 44
- DISPLAY statement
  - displaying data values 35
  - using in debugging 297
  - using in GUI applications 199
- DISPLAY statement, environment variables used in 199
- DISPLAY-1 (USAGE IS)
  - encoding 167
- DISPLAY-OF intrinsic function
  - example with Chinese data 171
  - example with Greek data 170
  - example with UTF-8 data 171

- DISPLAY-OF intrinsic function *(continued)*
  - using 169
  - with XML documents 365
- DLL extension as linker parameter 209
- DLL files
  - redistributing 217
  - setting directory path 197
- do loop 91
- do-until 91
- do-while 91
- document encoding declaration 364
- DOCUMENT-TYPE-DECLARATION
  - XML event 353
- documentation of program 7
- DOS, running under 494
- dumps, TRAP(OFF) side effect 295
- duplicate computations, grouping 539
- DYNAM compiler option
  - description 238
  - effect on CALL literal 458
  - effect on DLL resolution 487
  - performance considerations 546
- dynamic calls
  - cannot use for DB2 APIs 321
  - under CICS 329
- dynamic link libraries
  - dll cob2 option 206
  - building 485
  - CALL identifier, example 488
  - CALL literal, overview 487
  - CICS considerations 329
  - compiling and linking using cob2 487
  - compiling, example 489
  - creating 206
  - creating a module definition file
    - example 488
    - overview 487
  - creating DLL source files
    - example 486, 487
  - creating for OO 220
  - files, setting directory path 197
  - for Java interoperability 220
  - function 485
  - linking, example 489
  - overview of dynamic linking 486
  - purpose 486
  - resolving references 486
  - subprograms and outermost programs 485
  - using with Java interoperability 221
  - using with OO 221
- dynamic linking
  - advantages 486
  - definition 458
  - overview 486
  - resolving DLL references 486
- dynamic loading, requirements for 197

## E

- E-level error message 204, 303
- EBCDIC
  - converting to ASCII 107
  - multibyte portability 449
  - SBCS portability 447

- EBCDIC\_CODEPAGE environment variable
  - setting 197
  - valid values 183
- efficiency of coding 537
- embedded cross-reference
  - description 307
  - example 315
- embedded error messages 303
- embedded MAP summary 306, 311
- embedded SQL
  - advantages of 333
  - comparison with ODBC 333
- encoding
  - controlling in XML output 383
  - description 167
  - language characters 159
  - of XML documents 364
- encoding declaration
  - preferable to omit 365
  - specifying 366
- ENCODING-DECLARATION XML event 354
- END-OF-CDATA-SECTION XML event 354
- END-OF-DOCUMENT XML event 354
- END-OF-ELEMENT XML event 354
- end-of-file phrase (AT END) 147
- enhancing XML output
  - example of converting hyphens in element names to underscores 382
  - example of modifying data definitions 380
  - rationale and techniques 379
- entry point
  - alternate in ENTRY statement 475
  - definition 485
  - ENTRYINT for indicating call convention 239
  - function-pointer data item 475
  - multiple, restriction 477
  - passing addresses of 475
  - procedure-pointer data item 475
- ENTRY statement
  - for alternate entry points 475
  - handling of program-name in 255
- ENTRYINT compiler option
  - description 239
  - for calls from C/C++ 463
- environment differences, zSeries and the workstation 450
- ENVIRONMENT DIVISION
  - class 392
  - client 404
  - collating sequence coding 8
- CONFIGURATION SECTION 7
  - description 7
- INPUT-OUTPUT SECTION 7
  - instance method 397
  - subclass 416
- environment variables
  - accessing files with 196
  - assignment-name 196
  - CLASSPATH
    - description 197
    - specifying location of Java classes 221

- environment variables *(continued)*
  - COBCPYEXT 195
  - COBJVMINITOPTIONS
    - description 197
    - specifying JVM options 223
  - COBLSTDIR 195
  - COBMSG 197
  - COBOPT 195
  - COBPATH
    - CICS dynamic calls 329
    - description 195, 197
  - COBRTOPT 197
  - compiler 195
  - DB2DBDFT 195
  - DB2PATH
    - description 195
    - interaction with SYSLIB 322
  - definition 193
  - EBCDIC\_CODEPAGE 197
  - ILINK 209
  - LANG 198
  - LC\_ALL 181, 198
  - LC\_COLLATE 181, 198
  - LC\_CTYPE 181, 198
  - LC\_MESSAGES 181, 198
  - LC\_TIME 181, 198
  - LIB 196, 211
  - library-name 195, 275
  - linker 196
  - LOCPATH 198
  - NLSPATH 198
  - PATH
    - description 199
    - specifying location for COBOL classes 221
  - RDZvrINSTDIR 217
  - runtime 196
  - search-order precedence 193
  - setting
    - for COBOL for Windows 194
    - locale 181
    - SET command 193
    - System Properties window 193
  - SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 199
  - SYSLIB 196
    - specifying location of JNL.cpy 219
  - TEMPMEM 196
  - text-name 195, 275
  - TMP 200
  - TZ 200
- environment-name 7
- environment, preinitializing
  - example 509
  - for C/C++ program 462
  - overview 507
- ERRCOUNT runtime option 294
- ERRMSG, for generating list of error messages 204
- error
  - arithmetic 146
  - compiler options, conflicting 226
  - flagging at run time 293
  - handling 145
  - message table
    - example using indexing 71

error (*continued*)  
  message table (*continued*)  
    example using subscripting 70  
error messages  
  compiler  
    choosing severity to be flagged 303  
    correcting source 203  
    determining what severity level to produce 245  
    embedding in source listing 303  
    format 205  
    generating a list of 204  
    location in listing 205  
    severity levels 204  
  compiler-directed 205  
  runtime  
    format 707  
    incomplete or abbreviated 217  
    list of 707  
  setting national language 198  
euro currency sign 59  
EVALUATE statement  
  case structure 83  
  coding 83  
  contrasted with nested IFs 84, 85  
  example that tests several conditions 85  
  example with multiple WHEN phrases 84  
  example with THRU phrase 84  
  performance 84  
  structured programming 538  
  testing multiple values, example 88, 89  
  use to test multiple conditions 81  
evaluating data item contents  
  class test  
    for numeric 51  
    overview 86  
  INSPECT statement 103  
  intrinsic functions 107  
example  
  \_iwzGetCCSID: convert code-page ID to CCSID 189  
  \_iwzGetLocaleCP: get locale and EBCDIC code-page values 189  
examples  
  CEEGBLDY: convert date to COBOL integer format 589  
  CEEDATE: convert Lilian date to character format 591  
  CEEDATM: convert seconds to character format 595  
  CEEDAYS: convert date to Lilian format 599  
  CEEDYWK: calculate day of week from Lilian date 601  
  CEEGMT: get current GMT 604  
  CEEGMTO: get offset from Greenwich Mean Time 606  
  CEEISEC: convert integers to seconds 608  
  CEELOCT: get current local time 610  
  CEEQCEN: query century window 612  
  CEESCEN: set century window 614

examples (*continued*)  
  CEESECI: convert seconds to integers 616  
  CEESECS: convert timestamp to number of seconds 620  
  IGZEDT4: get current date with four-digit year 622  
exception condition  
  CALL 152  
  XML GENERATE 384  
  XML PARSE 367  
exception handling  
  with Java 432  
EXCEPTION XML event 354  
EXCEPTION/ERROR declarative  
  description 148  
  file status key 149  
exceptions, intercepting 295  
EXE extension as linker parameter 209  
EXIT compiler option  
  character string formats 241  
  description 240  
exit modules  
  called for SYSADATA data set 244  
  debugging 317  
  loading and invoking 243  
  when used in place of library-name 243  
  when used in place of SYSLIB 243  
  when used in place of SYSPRINT 244  
EXIT PROGRAM statement  
  in main program 454  
  in subprogram 454  
EXP extension as linker parameter 209  
exp file, creating with cob2 206  
explicit scope terminator 19  
exponentiation  
  evaluated in fixed-point arithmetic 572  
  evaluated in floating-point arithmetic 577  
  performance tips 541  
EXPORTS statement 492  
  listing callable subprograms 487  
extended mode 39, 569  
external class-name 393, 405  
EXTERNAL clause  
  example for files 479  
  for data items 478  
  for sharing files 12, 479  
external code page, definition 364  
external data  
  sharing 478  
external decimal data  
  national 43  
  zoned 43  
external file 479  
external floating-point data  
  display 44  
  national 44  
external program reference added to module 486

## F

factoring expressions 538

factory data  
  defining 419  
  definition of 387  
  making it accessible 419  
  private 419  
factory definition, coding 418  
factory methods  
  defining 420  
  definition of 387  
  hiding 421  
  invoking 421  
  using to wrap procedural programs 427  
FACTORY paragraph  
  factory data 419  
  factory methods 420  
factory section, defining 418  
FD (file description) entry 12  
feedback token  
  date and time services and 553  
figurative constants  
  definition 26  
  HIGH-VALUE restriction 162  
  national-character 162  
file access mode  
  dynamic 121  
  for indexed files 120  
  for line-sequential files 120  
  for relative files 121  
  for sequential files 119  
  random 121  
  sequential 121  
  summary table of 119  
file conversion  
  with millennium language extensions 519  
file description (FD) entry 12  
file extensions  
  as linker parameters 209  
  for error messages listing 205  
file organization  
  indexed 120  
  line-sequential 120  
  overview 119  
  relative 121  
  sequential 119  
file position indicator 125, 128  
FILE SECTION  
  DATA RECORDS clause 12  
  description 12  
  EXTERNAL clause 12  
  FD entry 12  
  GLOBAL clause 12  
  RECORD CONTAINS clause 12  
  record description 12  
  RECORD IS VARYING 12  
  RECORDING MODE clause 12  
  VALUE OF 12  
FILE STATUS clause  
  example 151  
  file loading 129  
  using 148  
  with status code  
    example 150  
    overview 150  
file status code  
  02 128



- file status code (*continued*)
  - 39 150
  - 92 130
  - using 147
- file status key
  - Btrieve files 149
  - checking for I/O errors 148
  - checking for successful OPEN 148, 149
  - error handling 298
  - setting up 122
  - used with status code
    - example 150
    - overview 150
- FILE-CONTROL paragraph, example 7
- file-system support
  - Btrieve 294
  - Pervasive.SQL 294
  - RSD 294
  - STL 294
  - using FILESYS runtime option 294
- files
  - accessing using environment
    - variables 196
  - adding records to 129
  - associating program files to external files 7
  - Btrieve 113
  - changing name 10
  - COBOL coding
    - example 124
    - overview 123
  - comparison of file organizations 119
  - deleting records from 130
  - describing 12
  - extensions supported by cob2 209
  - external 479
  - file position indicator 125, 128
  - identifying 113
  - identifying to the operating system 10
  - linker 212
  - multiple, compiling 201
  - opening 125
  - passed to compiler or linker 209
  - Pervasive.SQL 113
  - processing
    - Btrieve files 115
    - Pervasive.SQL files 115
    - RSD files 115
    - STL files 115
  - reading records from 128
  - replacing records in 130
  - RSD 113
  - size limits 122
  - STL 113
  - TRAP runtime option, effect of 295
  - updating records in 131
  - usage explanation 10
- FILESYS runtime option 294
- fixed century window 516
- fixed-point arithmetic
  - comparisons 57
  - evaluation 56
  - example evaluations 58
  - exponentiation 572

- fixed-point data
  - binary 44
  - conversions and precision 49
  - conversions between fixed- and floating-point 49
  - external decimal 43
  - intermediate results 571
  - packed-decimal 46
  - planning use of 539
- FLAG compiler option
  - compiler output 304
  - description 245
  - using 303
- flags and switches 87
- FLAGSTD compiler option 246
- FLOAT compiler option 247
- floating-point arithmetic
  - comparisons 57
  - evaluation 56
  - example evaluations 58
  - exponentiation 577
- floating-point data
  - conversions and precision 49
  - conversions between fixed- and floating-point 49
  - external 44
  - intermediate results 576
  - internal
    - format 46
    - performance tips 540
  - performance considerations 449
  - planning use of 539
  - portability 449
- four-digit years 555
- freeing object instances 413
- full date field expansion,
  - advantages 516
- function-pointer data item
  - addressing JNI services 635
  - definition 475
  - entry address for entry point 475
  - passing parameters to callable services 475
- functions
  - storing those frequently used 485

## G

- garbage collection 413
- GB 18030 data
  - converting to or from national 171
- processing 171
- generating XML output
  - example 375
  - overview 373
- get and set methods 401
- GETMAIN, saving address of 241
- GLOBAL clause for files 12, 15
- global names 457
- GOBACK statement
  - in main program 454
  - in subprogram 454
- Greenwich Mean Time (GMT)
  - getting offset to local time (CEEGMTO) 605
  - return Lilian date and Lilian seconds (CEEGMT) 603

- Gregorian character string
  - returning local time as a (CEELOCT) 609
  - example 610
- group item
  - cannot subordinate alphanumeric group within national group 165
  - comparing to national data 174
  - definition 24
  - for defining tables 61
  - group move contrasted with elementary move 33, 165
  - initializing
    - using a VALUE clause 70
    - using INITIALIZE 30, 67
  - MOVE statement with 33
  - passing as an argument 471
  - treated as a group item
    - example with INITIALIZE 68
    - in INITIALIZE 31
  - variably located 579
- group move contrasted with elementary move 33, 165
- GROUP-USAGE NATIONAL clause
  - communicating with Java 436
  - defining a national group 164
  - defining tables 62
  - example of declaring a national group 24
  - initializing a national group 31
- grouping data to pass as an argument 471

## H

- header on listing 6
- heap space, allocating with cob2 command 208
- heap, defining size of 493
- HEAPSIZE statement 493
- help files
  - setting national language 198
  - specifying path name 198
- hexadecimal
  - portability 449
- hexadecimal literals
  - as currency sign 59
  - national
    - description 26
    - using 161
- hiding factory methods 421

## I

- I-level message 204, 303
- idebug command, example 318
- IDENTIFICATION DIVISION
  - class 392
  - CLASS-ID paragraph 392, 415
  - client 403
  - coding 5
  - DATE-COMPILED paragraph 5
  - listing header example 6
  - method 396
  - PROGRAM-ID paragraph 5
  - required paragraphs 5

- IDENTIFICATION DIVISION (*continued*)
  - subclass 415
  - TITLE statement 6
- IEEE
  - portability 449
- IF statement
  - coding 81
  - nested 82
  - use EVALUATE instead for multiple conditions 82
  - with null branch 81
- IGZEDT4: get current date with four-digit year 622
- ILIB command
  - creating and maintaining libraries 211
- ILINK command
  - invoking the linker 211
- ILINK environment variable
  - example 212
  - for setting linker options 209
- IMP extension as linker parameter 209
- imperative statement, list 18
- implicit scope terminator 19
- import library, specifying with cob2 207
- incompatible data 51
- incrementing addresses 473
- index
  - assigning a value to 66
  - computation of element displacement, example 64
  - creating with OCCURS INDEXED BY clause 66
  - definition 64
  - incrementing or decrementing 66
  - initializing 66
  - range checking 302
  - referencing other tables with 66
- index data item
  - cannot use as subscript or index 66
  - creating with USAGE IS INDEX clause 66
- indexed file organization 120
- indexed files
  - file access mode 120
- indexing
  - computation of element displacement, example 64
  - definition 64
  - example 71
  - preferred to subscripting 541
  - tables 66
- INEXIT suboption of EXIT option 241, 243
- inheritance hierarchy, definition of 389
- INITIAL attribute 454
  - effect on nested programs 6
  - setting programs to initial state 6
- INITIALIZE statement
  - examples 28
  - loading group values 30
  - loading national group values 31
  - loading table values 67
  - REPLACING phrase 67
  - using for debugging 299
- initializing
  - a group item
    - using a VALUE clause 70
    - using INITIALIZE 30, 67
  - a national group item
    - using a VALUE clause 70
    - using INITIALIZE 31, 68
  - a structure using INITIALIZE 30
  - a table
    - all occurrences of an element 70
    - at the group level 70
    - each item individually 69
    - using INITIALIZE 67
    - using PERFORM VARYING 91
  - examples 28
  - instance data 412
  - the runtime environment
    - example 509
    - overview 507
  - variable-length group 75
- inline PERFORM
  - example 90
  - overview 90
- input
  - from files 113
  - overview 119
- input procedure
  - coding 136
  - example 140
  - requires RELEASE or RELEASE FROM 136
  - restrictions 137
- INPUT-OUTPUT SECTION 7
- input/output
  - checking for errors 148
  - coding
    - example 124
    - overview 123
  - GUI applications 199
  - introduction 113
  - logic flow after error 146
- input/output coding
  - AT END (end-of-file) phrase 147
  - checking for successful operation 148
  - checking status code
    - example 150
    - overview 150
  - error handling techniques 146
  - EXCEPTION/ERROR
    - declaratives 148
- INSPECT statement
  - examples 103
  - using 103
- inspecting data (INSPECT) 103
- instance
  - creating 412
  - definition of 387
  - deleting 413
- instance data
  - defining 394, 416
  - definition of 387
  - initializing 412
  - making it accessible 401
  - private 394
- instance methods
  - defining 395, 417
  - definition of 387
- instance methods (*continued*)
  - invoking overridden 411
  - overloading 400
  - overriding 399
- INTEGER intrinsic function, example 102
- INTEGER-OF-DATE intrinsic function 55
- INTEGER-PART intrinsic function 102
- integers
  - converting Lilian seconds to (CEESECI) 615
- integrated CICS translator
  - advantages 331
  - cicstcl -p command 331
  - invoking 327
  - overview 331
- interlanguage communication
  - between COBOL and C/C++ 462
  - between COBOL and Java 431
- intermediate results 569
- internal bridges
  - advantages 516
  - example 518
  - for date processing 517
- internal floating-point data (COMP-1, COMP-2) 46
- interoperable data types with Java 436
- intrinsic functions
  - as reference modifiers 102
  - collating sequence, effect of 188
  - compatibility with CEELOCT 609
  - converting alphanumeric data items with 104
  - converting national data items with 104
  - date and time 586
- DATEVAL
  - example 530
  - using 529
- evaluating data items 107
- example of
  - ANNUITY 56
  - CHAR 108
  - CURRENT-DATE 55
  - DISPLAY-OF 170
  - INTEGER 102
  - INTEGER-OF-DATE 55
  - LENGTH 55, 109, 110
  - LOG 56
  - LOWER-CASE 105
  - MAX 55, 78, 108, 109
  - MEAN 56
  - MEDIAN 56, 78
  - MIN 102
  - NATIONAL-OF 170
  - NUMVAL 105
  - NUMVAL-C 55, 105
  - ORD 107
  - ORD-MAX 78, 108
  - PRESENT-VALUE 55
  - RANGE 56, 78
  - REM 56
  - REVERSE 105
  - SQRT 56
  - SUM 78
  - UPPER-CASE 105

- intrinsic functions (*continued*)
  - example of (*continued*)
    - WHEN-COMPILED 111
  - finding date of compilation 111
  - finding largest or smallest item 108
  - finding length of data items 110
  - intermediate results 574, 577
  - introduction to 36
  - nesting 37
  - numeric functions
    - examples of 54
    - integer, floating-point, mixed 53
    - nested 54
    - special registers as arguments 54
    - table elements as arguments 54
    - uses for 53
  - processing table elements 78
  - UNDATE
    - example 530
    - using 529
- INVALID KEY phrase
  - example 151
- INVOKE statement
  - RETURNING phrase 411
  - USING phrase 408
  - using to create objects 412
  - using to invoke methods 407
  - with ON EXCEPTION 408, 422
  - with PROCEDURE DIVISION
    - RETURNING 477
- invoking
  - compiler and linker 201
  - date and time services 551
  - factory or static methods 421
  - instance methods 407
- iwzGetSortErrno, obtaining sort or merge
  - error number with 141

## J

- Java
  - and COBOL 431
    - compiling 219
    - linking 220
    - running 221
    - structuring applications 428
  - array classes 436
  - arrays
    - declaring 437
    - example 440
    - manipulating 438
  - boolean array 437
  - boolean type 436
  - byte array 437
  - byte type 436
  - char array 437
  - char type 436
  - class types 436
  - double array 438
  - double type 436
  - example
    - exception handling 433
    - processing an array 440
  - exception
    - catching 433
    - example 433
    - handling 432
- Java (*continued*)
  - exception (*continued*)
    - throwing 432
  - float array 438
  - float type 436
  - global references
    - JNI services for 435
      - managing 434
      - object 434
      - passing 434
  - int array 437
  - int type 436
  - interoperability 431
  - interoperable data types, coding 436
  - jstring class 436
  - local references
    - deleting 434
    - freeing 435
    - JNI services for 435
      - managing 434
      - object 434
      - passing 434
      - per multithreading 434
      - saving 434
  - long array 437
  - long type 436
  - methods
    - access control 435
    - object array 437
    - running with COBOL 221
    - sharing data with 435
    - short array 437
    - short type 436
    - string array 437
    - strings
      - declaring 437
      - manipulating 441
    - supported releases 221
  - Java virtual machine
    - exceptions 433
    - initializing 222
    - object references 434
  - java.lang.Object
    - referring to as Base 392
  - javac command 219
  - JNI
    - accessing services 431
    - comparing object references 406
    - converting local references to
      - global 412
    - environment structure 431
      - addressability for 431
    - exception handling services 432
    - Java array services 438
    - Java string services 441
    - obtaining class object reference 432
    - restrictions when using 432
    - Unicode services 441
    - UTF-8 services 441
  - JNI.cpy
    - for compiling 219
    - for JNINativeInterface 431
    - listing 635
  - JNIEnvPtr special register 431
  - JNINativeInterface
    - environment structure 431
  - JNI.cpy 431

- jstring Java class 436

## K

- Kanji comparison 86
- Kanji data, testing for 177
- keys
  - for binary search 77
  - for merging
    - default 186
    - defining 139
    - overview 134
  - for sorting
    - default 186
    - defining 139
    - overview 133
  - permissible data types
    - in MERGE statement 139
    - in OCCURS clause 62
    - in SORT statement 139
  - to specify order of table elements 62

## L

- LANG environment variable 198
- language features for debugging
  - DISPLAY statements 297
- largest or smallest item, finding 108
- last-used state
  - subprograms with EXIT PROGRAM
    - or GOBACK 454
- LC\_ALL 181
- LC\_ALL environment variable 198
- LC\_COLLATE 181
- LC\_COLLATE environment variable 198
- LC\_CTYPE 181
- LC\_CTYPE environment variable 198
- LC\_MESSAGES 181
- LC\_MESSAGES environment
  - variable 198
- LC\_TIME 181
- LC\_TIME environment variable 198
- LENGTH intrinsic function 107
  - compared with LENGTH OF special
    - register 110
  - example 55, 110
  - variable-length results 109
  - with national data 110
- length of data items, finding 110
- LENGTH OF special register
  - passing 468
  - using 110
- level-88 item
  - conditional expressions 86
  - for windowed date fields 523
  - restriction 524
  - setting switches off, example 89
  - setting switches on, example 88
  - switches and flags 87
  - testing multiple values, example 88
  - testing single values, example 87
- level-number 311
- LIB compiler option 248
- LIB environment variable 196
- LIB extension as linker parameter 209
- lib file, creating with cob2 206



- LIBEXIT suboption of EXIT option 241, 243
- LIBRARY statement 493
- library text
  - specifying path for 195, 275
- library-name
  - alternative if not specified 207
  - specifying path for library text 195, 275
- library-name, when not used 243
- Lilian date
  - calculate day of week from (CEEDYWK) 601
  - convert date to (CEEDAYS) 597
  - convert date to COBOL integer format (CEECLDY) 587
  - convert output\_seconds to (CEEISEC) 608
  - convert to character format (CEEDATE) 590
  - get current local date or time as a (CEELOCT) 609
  - get GMT as a (CEEGMT) 603
  - using as input to CEESECI 616
- limits of the compiler
  - DATA DIVISION 11
  - file input-output 122
  - user data 11
- line number 310
- line-sequential files
  - file access mode 120
  - limitations 122
  - organization 120
- LINECOUNT compiler option 249
- linkage conventions
  - C/C++ called from COBOL 463
  - COBOL called from C/C++ 463
  - compiler directive CALLINT for 273
  - compiler option CALLINT for 229
  - differences between COBOL and C/C++ 463
  - OPTLINK for calls to nested programs 454
  - overview 459
- LINKAGE SECTION
  - coding 470
  - for describing parameters 469
  - with recursive calls 16
  - with the THREAD option 16
- linkages, data 464
- linked-list processing, example 473
- linker
  - errors 213
  - errors in program-names 214
  - files 212
  - files passed to 209
  - invoking 201
  - LIB environment variable 211
  - object-file compatibility 212
  - options
    - list of 277
    - specifying 210
  - parameters 209
  - passing information to 207
  - resolving references to DLLs 486
  - return codes 214
  - search path 211
- linker (*continued*)
  - search rules 212
  - setting options with ILINK environment variable 209
- linker options
  - /? 278
  - /ALIGNADDR 278
  - /ALIGNFILE 279
  - /BASE 279
  - /CODE 280
  - /DATA 280
  - /DBGPACK 281
  - /DEBUG 281
  - /DEFAULTLIBRARYSEARCH 281
  - /DLL 282
  - /ENTRY 282
  - /EXECUTABLE 283
  - /EXTDICTIONARY 283
  - /FIXED 284
  - /FORCE 284
  - /HEAP 285
  - /HELP 285
  - /INCLUDE 285
  - /INFORMATION 285
  - /LINENUMBERS 286
  - /LOGO 286
  - /MAP 287
  - /OUT 287
  - /PMTYP 288
  - /SECTION 288
  - /SEGMENTS 289
  - /STACK 290
  - /STUB 290
  - /SUBSYSTEM 291
  - /VERBOSE 291
  - /VERSION 292
  - example of overriding 212
  - list of valid options 277
  - setting 209
  - specifying 210
- linker response file, echoing contents 286
- linking
  - DLLs, example 489
  - dynamic 486
  - from the command line 211
  - ILINK environment variable for
    - setting options 209
  - options for 277
  - programs 208
  - specifying options 210
  - static 485
  - using the compiler 210
- linking OO applications
  - cob2 command 220
  - example 221
- LIST compiler option
  - description 249
  - getting output 307
- list of resources 779
- listings
  - compiler options affecting 225
  - data and procedure-name cross-reference 305
  - embedded error messages 303
  - generating a short listing 307
  - line numbers, user-supplied 308
- listings (*continued*)
  - sorted cross-reference of
    - program-names 315
    - terms used in MAP output 312
- literals
  - alphanumeric
    - control characters within 26
    - description 25
    - with multibyte content 176
  - DBCS
    - description 26
    - maximum length 176
    - using 176
  - definition 25
  - hexadecimal
    - using 161
  - national
    - description 26
    - using 161
  - numeric 26
  - using 25
- little-endian
  - definition 45
  - format for data representation 229
  - representation of integers 448, 452
  - representation of national characters 452
- little-endian, converting to
  - big-endian 159
- load address 491
- load segment 491
- loading a table dynamically 67
- local names 457
- local references, converting to global 412
- local time
  - getting (CEELOCT) 609
- LOCAL-STORAGE SECTION
  - client 405, 406
  - comparison with WORKING-STORAGE
    - example 14
    - OO client 406
    - overview 13
- locale
  - accessing 188
  - and messages 182
  - cultural conventions, definition 179
  - default 182
  - definition 179
  - effect of COLLSEQ compiler option 185
  - effect of PROGRAM COLLATING SEQUENCE 185
  - list of supported values 183
  - locale-based collating 185
  - querying 188
  - shown in listing 305, 309
  - specifying 198
  - value syntax 181
- locale information database
  - specifying search path name 198
- LOCPATH environment variable 198
- LOG intrinsic function 56
- loops
  - coding 89
  - conditional 91
  - do 91

- loops (*continued*)
  - in a table 91
  - performed an explicit number of times 91
- LOWER-CASE intrinsic function 105
- lowercase, converting to 105
- LST file extension 205
- LSTFILE compiler option 250

## M

- main program
  - and subprograms 453
  - arguments to 482
  - specifying with cob2 207, 208
- makefile 215
- MAP compiler option
  - description 250
  - embedded MAP summary 307
  - example 310, 313
  - nested program map 307
    - example 313
  - symbols used in output 312
  - terms used in output 312
  - using 306, 307
- MAP extension as linker parameter 210
- mapping of DATA DIVISION items 307
- mathematics
  - intrinsic functions 54, 56
- MAX intrinsic function
  - example table calculation 78
  - example with functions 55
  - using 108
- maximum
  - file size 122
  - record size 122
- MDECK compiler option
  - description 251
  - multioption interaction 227
- MEAN intrinsic function
  - example statistics calculation 56
  - example table calculation 78
- MEDIAN intrinsic function
  - example statistics calculation 56
  - example table calculation 78
- memory-protection attributes of segments 288
- merge
  - alternate collating sequence 139
  - completion code 141
  - criteria 139
  - description 133
  - determining success 141
  - diagnostic message 141
  - error number
    - list of possible values 141
    - obtaining with iwzGetSortErrno 141
  - files, describing 134
  - keys
    - default 186
    - defining 139
    - overview 134
  - process 133
  - terminating 144

- MERGE statement
  - ASCENDING|DESCENDING KEY phrase 139
  - COLLATING SEQUENCE phrase 8, 140
  - description 138
  - GIVING phrase 138
  - overview 133
  - USING phrase 138
- MERGE work files 200
- message catalogs
  - specifying path name 198
- messages
  - compiler
    - choosing severity to be flagged 303
    - date-related 530
    - determining what severity level to produce 245
    - embedding in source listing 303
    - generating a list of 204
    - millennium language extensions 530
    - severity levels 204
  - compiler-directed 205
  - national language support 182
  - runtime
    - format 707
    - incomplete or abbreviated 217
    - list of 707
  - setting national language 198
  - specifying file for 197
  - TRAP(OFF) side effect 295
- METHOD-ID paragraph 396
- methods
  - constructor 420
  - factory 420
  - hiding factory 421
  - instance 395, 417
  - invoking 407, 421
  - invoking superclass 411
  - Java access control 435
  - obtaining passed arguments 398
  - overloading 400
  - overriding 399, 421
- PROCEDURE DIVISION
  - RETURNING 478
  - returning a value from 398
  - signature 396
- millennium language extensions
  - assumed century window 524
  - compatible dates 521
  - compiler options affecting 225
  - concepts 514
  - date windowing 513
  - DATEPROC compiler option 237
  - nondates 525
  - objectives 515
  - principles 514
  - YEARWINDOW compiler option 271
- MIN intrinsic function
  - example 102
  - using 108
- MIXED suboption of PGMNAME 256
- MLE 514
- mnemonic-name
  - SPECIAL-NAMES paragraph 7

- module definition files
  - contents 487
  - creating 487, 489
  - module statements 490
  - reserved words 490
  - rules 489
  - when to use 489
- module export files
  - creating 206, 486
  - export file, creating with cob2 206
- module statements 490
- modules, exit
  - loading and invoking 243
- MOVE statement
  - assigning arithmetic results 34
  - converting to national data 168
  - CORRESPONDING 33
  - effect of ODO on lengths of sending and receiving items 73
  - group move contrasted with elementary move 33, 165
  - with elementary receiving items 32
  - with group receiving items 32
  - with national items 32
- MQ applications 497
- multiple currency signs
  - example 60
  - using 59
- multiple inheritance, not permitted 390, 415
- multiple thread environment, running in 265
- multitasking, definition 498
- multithread environment
  - requirements 256
- multithreading
  - choosing data section in an OO client 406
  - control transfer 501
  - ending programs 501
  - example 502
  - limitations 502
  - overview 497
  - preinitializing 501
  - preparing COBOL programs for 497
  - recursion 500
  - scope of language elements
    - program invocation instance 499
    - run unit 499
    - summary table 500
  - synchronizing access to resources 502
  - terminology 497
  - THREAD compiler option
    - restrictions with 265
    - when to choose 500

## N

- N delimiter for national or DBCS literals 26
- name declaration
  - searching for 457
- name decoration
  - definition 214
  - with CDECL interface convention 460

- name decoration (*continued*)
  - with SYSTEM interface
    - convention 461, 476
- NAME statement 494
- naming
  - programs 5
- NATIONAL (USAGE IS)
  - external decimal 43
  - floating point 44
- national comparison 86
- national data
  - cannot use with DATE FORMAT
    - clause 514
  - communicating with Java 436
  - comparing
    - effect of collating sequence 187
    - effect of NCOLLSEQ 173
    - overview 172
    - to alphabetic, alphanumeric, or DBCS 174
    - to alphanumeric groups 174
    - to numeric 173
    - two operands 172
  - concatenating (STRING) 93
  - converting
    - from alphanumeric or DBCS with NATIONAL-OF 169
    - from alphanumeric, DBCS, or integer with MOVE 168
    - overview 167
    - to alphanumeric with DISPLAY-OF 169
    - to numbers with NUMVAL, NUMVAL-C 105
    - to or from Chinese GB 18030 171
    - to or from Greek alphanumeric, example 170
    - to or from UTF-8 171
    - to uppercase or lowercase 105
    - with INSPECT 103
  - defining 160
  - encoding in XML documents 364
  - evaluating with intrinsic functions 107
  - external decimal 43
  - external floating-point 44
  - figurative constants 162
  - finding the smallest or largest item 108
  - in conditional expressions 172
  - in generated XML documents 373
  - in keys
    - in MERGE statement 139
    - in OCCURS clause 62
    - in SORT statement 139
  - initializing, example of 29
  - input with ACCEPT 35
  - inspecting (INSPECT) 103
  - LENGTH intrinsic function and 110
  - LENGTH OF special register 110
  - literals
    - using 161
  - MOVE statement with 32, 167
  - NSYMBOL compiler option if no USAGE clause 161
  - output with DISPLAY 35
  - reference modification of 100
- national data (*continued*)
  - reversing characters 105
  - specifying 160
  - splitting (UNSTRING) 96
  - VALUE clause with alphanumeric literal, example 109
- national decimal data (USAGE NATIONAL)
  - defining 163
  - example 39
  - format 43
  - initializing, example of 29
- national floating-point data (USAGE NATIONAL)
  - defining 163
  - definition 44
- national group item
  - advantages over alphanumeric groups 163
  - can contain only national data 24, 165
  - communicating with Java 436
  - contrasted with USAGE NATIONAL group 25
  - defining 164
  - example 24
  - for defining tables 62
  - in generated XML documents 373
  - initializing
    - using a VALUE clause 70
    - using INITIALIZE 31, 68
  - LENGTH intrinsic function and 110
  - MOVE statement with 33
  - overview 163
  - passing as an argument 471
  - treated as a group item
    - example with INITIALIZE 166
    - in INITIALIZE 31
    - in MOVE CORRESPONDING 33
    - in XML GENERATE 374
    - summary 166
  - treated as an elementary item
    - example with MOVE 33
    - in most cases 24, 163
  - using
    - as an elementary item 165
    - overview 164
  - VALUE clause with alphanumeric literal, example 70
- national language support
  - messages 182
- national language support (NLS)
  - accessing locale and code-page values 188
  - collating sequence 185
  - DBCS 175
  - locale 179
  - locale-based collating 185
  - processing data 155
  - setting the locale 179
  - specifying locale and code page 198
- national literals
  - description 26
  - using 161
- national-edited data
  - defining 161
  - editing symbols 161
- national-edited data (*continued*)
  - initializing
    - example 29
    - using INITIALIZE 68
  - MOVE statement with 32
  - PICTURE clause 161
- NATIONAL-OF intrinsic function
  - example with Chinese data 171
  - example with Greek data 170
  - example with UTF-8 data 171
  - using 169
  - with XML documents 365
- native format
  - host option effect on command-line arguments 482
  - BINARY option 229
  - CHAR option 231
  - COMP-5 data 229
  - FLOAT option 247
  - for JNI service arguments 561
  - for method arguments 561
  - NATIVE phrase 229
  - portability considerations 448
- NCOLLSEQ compiler option
  - description 252
  - effect on national collating sequence 185, 187
  - effect on national comparisons 173
  - effect on sort and merge keys 139
- nested COPY statement 243, 549
- nested delimited scope statements 20
- nested IF statement
  - coding 82
  - CONTINUE statement 81
  - EVALUATE statement preferred 82
  - with null branches 81
- nested intrinsic functions 54
- nested program integration 545
- nested program map
  - description 307
  - example 313
- nested programs
  - calling 454
  - description 455
  - guidelines 454
  - map 307, 313
  - scope of names 457
  - transfer of control 454
  - use OPTLINK for CALL identifier 454
- nesting level
  - program 310, 314
  - statement 310
- NLSPATH environment variable 198
- NMAKE command
  - description 215
  - on the command line 215
  - with a command file 216
  - with a description file 216
- NOCOMPILER compiler option
  - use to find syntax errors 302
- NODESC suboption of CALLINT compiler option 230
- NODESCRIPTOR suboption of CALLINT compiler option 230
- nondates with MLE 525

- NOSSRANGE compiler option
  - effect on checking errors 293
- Notices 753
- NSYMBOL compiler option
  - description 253
  - effect on N literals 26
  - for DBCS literals 161
  - for national data items 161
  - for national literals 161
- null branch 81
- null-terminated strings
  - example 99
  - handling 472
  - manipulating 98
- NUMBER compiler option
  - description 253
  - for debugging 308
- numeric class test
  - checking for valid data 51
- numeric comparison 86
- numeric data
  - binary
    - byte reversal of 45
    - USAGE BINARY 44
    - USAGE COMPUTATIONAL (COMP) 44
    - USAGE COMPUTATIONAL-4 (COMP-4) 44
    - USAGE COMPUTATIONAL-5 (COMP-5) 45
  - can compare algebraic values regardless of USAGE 174
  - comparing to national 173
  - converting
    - between fixed- and floating-point 49
    - precision 49
    - to national with MOVE 168
  - defining 39
  - display floating-point (USAGE DISPLAY) 44
  - editing symbols 41
  - external decimal
    - USAGE DISPLAY 43
    - USAGE NATIONAL 43
  - external floating-point
    - USAGE DISPLAY 44
    - USAGE NATIONAL 44
  - internal floating-point
    - USAGE COMPUTATIONAL-1 (COMP-1) 46
    - USAGE COMPUTATIONAL-2 (COMP-2) 46
  - national decimal (USAGE NATIONAL) 43
  - national floating-point (USAGE NATIONAL) 44
  - packed-decimal
    - sign representation 50
    - USAGE COMPUTATIONAL-3 (COMP-3) 46
    - USAGE PACKED-DECIMAL 46
  - PICTURE clause 39, 41
  - storage formats 42
  - USAGE DISPLAY 39
  - USAGE NATIONAL 39

- numeric data (*continued*)
  - zoned decimal (USAGE DISPLAY)
    - format 43
    - sign representation 50
- numeric intrinsic functions
  - example of
    - ANNUITY 56
    - CURRENT-DATE 55
    - INTEGER 102
    - INTEGER-OF-DATE 55
    - LENGTH 55, 109
    - LOG 56
    - MAX 55, 78, 108, 109
    - MEAN 56
    - MEDIAN 56, 78
    - MIN 102
    - NUMVAL 105
    - NUMVAL-C 55, 105
    - ORD 107
    - ORD-MAX 78
    - PRESENT-VALUE 55
    - RANGE 56, 78
    - REM 56
    - SQRT 56
    - SUM 78
  - integer, floating-point, mixed 53
  - nested 54
  - special registers as arguments 54
  - table elements as arguments 54
  - uses for 53
- numeric literals, description 26
- numeric-edited data
  - BLANK WHEN ZERO clause
    - coding with numeric data 161
    - example 41
  - defining 161
  - editing symbols 41
  - initializing
    - examples 30
    - using INITIALIZE 68
  - PICTURE clause 41
  - USAGE DISPLAY
    - displaying 41
    - initializing, example of 30
  - USAGE NATIONAL
    - displaying 41
    - initializing, example of 30
- NUMVAL intrinsic function
  - description 105
- NUMVAL-C intrinsic function
  - description 105
  - example 55
- NX delimiter for national literals 26

## O

- OBJ extension as linker parameter 210
- object
  - creating 412
  - definition of 387
  - deleting 413
- object code
  - compatibility 212
  - controlling 225
  - generating 235
- object instances, definition of 387

- object module, adding external program
  - reference to 486
- OBJECT paragraph
  - instance data 394, 416
  - instance methods 395
- object references
  - comparing 406
  - converting from local to global 412
  - example of passing 409
  - setting 407
  - typed 405
  - universal 406
- OBJECT-COMPUTER paragraph 7
- object-file compatibility 212
- object-oriented COBOL
  - communicating with Java 436
  - compiling 219
  - linking
    - example 221
    - overview 220
  - preparing applications 220
  - restrictions
    - CICS 387
    - EXEC CICS statements 387
    - EXEC SQL statements 387
    - SQL compiler option 387
  - running 221
  - writing OO programs 387
- objectives of millennium language
  - extensions 515
- OCCURS clause
  - ASCENDING | DESCENDING KEY phrase
    - example 77
    - needed for binary search 77
    - specify order of table elements 62
  - cannot use in a level-01 item 61
  - for defining table elements 61
  - for defining tables 61
  - INDEXED BY phrase for creating indexes 66
  - nested for creating multidimensional tables 62
- OCCURS DEPENDING ON (ODO) clause
  - complex 579
  - for creating variable-length tables 72
  - initializing ODO elements 75
  - ODO object 72
  - ODO subject 72
  - optimization 542
  - simple 72
- OCCURS INDEXED BY clause, creating indexes with 66
- ODBC
  - accessing return values 337
  - advantages of 333
  - background 334
  - CALL interface convention 345
  - comparison with embedded SQL 333
  - driver manager 334
  - embedded SQL 333
  - error messages 345
  - installing and configuring the drivers 334
  - mapping of C data types 334
  - overview 333

- ODBC (*continued*)
  - passing a COBOL pointer to 335
  - supplied copybooks 338
    - example 339
    - ODBC3D.CPY example 343
    - ODBC3P.CPY 340
  - using APIs from COBOL 334
- ODO object 72
- ODO subject 72
- OMITTED parameters 551
- OMITTED phrase for omitting arguments 470
- ON EXCEPTION phrase
  - INVOKE statement 408, 422
- ON SIZE ERROR
  - with windowed date fields 527
- OPEN operation code 243
- OPEN statement
  - file availability 125
  - file status key 148
- opening files
  - overview 125
  - using environment variables 196
- optimization
  - avoid ALTER statement 538
  - avoid backward branches 538
  - BINARY data items 540
  - consistent data 540
  - constant computations 539
  - constant data items 538
  - contained program integration 545
  - duplicate computations 539
  - effect of compiler options on 545
  - effect on parameter passing 469
  - effect on performance 537
  - factor expressions 538
  - index computations 542
  - indexing 541
  - nested program integration 545
  - OCCURS DEPENDING ON 542
  - out-of-line PERFORM 538
  - packed-decimal data items 540
  - performance implications 542
  - structured programming 537
  - subscript computations 542
  - subscripting 541
  - table elements 541
  - top-down programming 538
  - unused data items 254
- OPTIMIZE compiler option
  - description 254
  - effect on parameter passing 469
  - effect on performance 544
  - performance considerations 546
  - using 544
- optimizer
  - overview 544
- OPTLINK interface convention
  - description 460
  - example of ensuring its use 476
  - specified with CALLINT 230
- OPTLINK suboption of CALLINT
  - compiler option 230
- ORD intrinsic function, example 107
- ORD-MAX intrinsic function
  - example table calculation 78
  - using 108

- ORD-MIN intrinsic function 108
- order of evaluation
  - arithmetic operators 53, 571
  - compiler options 227
- ordinal position of data construct 492
- ordinal position of function 492
- out-of-line PERFORM 90
- outermost programs in DLLs 485
- output
  - overview 119
  - to files 113
- output procedure
  - coding 137
  - example 140
  - requires RETURN or RETURN INTO 137
  - restrictions 137
- overflow condition
  - CALL 152
  - joining and splitting strings 145
  - UNSTRING 95
- overloading instance methods 400
- overriding
  - factory methods 421
  - instance methods 399
- overriding linker options, example 212

## P

- packed-decimal data item
  - date fields, potential problems 532
  - description 46
  - sign representation 50
  - synonym 42
  - using efficiently 46, 540
- paragraph
  - grouping 92
  - introduction 17
- parameter list
  - address of with INEXIT 243
  - for ADEXIT 244
  - for PRTEXTIT 244
- parameters
  - describing in called program 469
  - in main program 482
- parsing XML documents
  - description 351
  - overview 349
- passing data between programs
  - addresses 472
  - arguments in calling program 469
  - BY CONTENT 467
  - BY REFERENCE 467
  - BY VALUE
    - overview 467
    - restrictions 469
  - EXTERNAL data 478
  - in the RETURN-CODE special register 477
  - JNI services 432
  - OMITTED arguments 470
  - parameters in called program 469
  - with Java 435
- PATH environment variable
  - description 199
  - specifying location for COBOL classes 221

- path name
  - for copybook search 207, 275
  - library text 195, 275
  - multiple, specifying 195, 275
  - search-order precedence 193
  - specifying for catalogs and help files 198
  - specifying for executable programs 199
  - specifying for locale information database 198
  - specifying with LIB compiler option 248
- PERFORM statement
  - coding loops 89
  - for a table
    - example using indexing 71
    - example using subscripting 70
  - for changing an index 66
  - inline 90
  - out-of-line 90
  - performed an explicit number of times 91
  - TEST AFTER 91
  - TEST BEFORE 91
  - THRU 92
  - TIMES 91
  - UNTIL 91
  - VARYING 91
  - VARYING WITH TEST AFTER 91
  - WITH TEST AFTER . . . UNTIL 91
  - WITH TEST BEFORE . . . UNTIL 91
- performance
  - arithmetic evaluations 539
  - arithmetic expressions 540
  - CICS environment 537
  - coding 537
  - coding tables 541
  - compiler option
    - DYNAM 546
    - FLOAT 449
    - OPTIMIZE 544, 546
    - SSRANGE 546
    - TEST 546
    - THREAD 266, 546
    - TRUNC 266, 546
  - consistent data types 540
  - data usage 540
  - effect of compiler options on 545
  - exponentiations 541
  - OCCURS DEPENDING ON 542
  - optimizer
    - overview 544
  - order of WHEN phrases in EVALUATE 84
  - out-of-line PERFORM compared with inline 90
  - programming style 537
  - runtime considerations 537
  - table handling 542
  - table searching
    - binary compared with serial 75
    - improving serial search 76
  - using the TEMPMEM environment variable 196
  - variable subscript data format 65
  - worksheet 548



- performing calculations
  - date and time services 552
- period as scope terminator 19
- Pervasive.SQL files
  - identifying 113
  - processing 115
- PGMNAME compiler option 255
- physical
  - record 12
- PICTURE clause
  - cannot use for internal floating point 40
  - determining symbol used 236
  - incompatible data 51
  - N for national data 160
  - national-edited data 161
  - numeric data 39
  - numeric-edited data 161
  - Z for zero suppression 41
- picture strings
  - date and time services and 555
- platform differences 450
- pointer data item
  - description 37
  - incrementing addresses with 473
  - NULL value 472
  - used to pass addresses 472
  - used to process chained list 472, 473
- porting applications
  - CICS 328
  - differences between platforms 445
  - effect of separate sign 40
  - environment differences 450
  - file-status keys 450
  - language differences 445
  - mainframe to workstation
    - choosing compiler options 445
    - running mainframe applications on the workstation 447
  - multibyte 449
  - multitasking 451
  - overview 445
  - SBCS 447
  - using COPY to isolate
    - platform-specific code 446
  - Windows to AIX 451
  - workstation to mainframe
    - workstation-only compiler options 451
    - workstation-only language features 451
    - workstation-only names 451
- precedence
  - arithmetic operators 53, 571
- preinitializing the COBOL environment
  - example 509
  - for C/C++ program 462
  - initialization 507
  - overview 507
  - restriction under CICS 507
  - termination 508
  - with multithreading 501
- PRESENT-VALUE intrinsic function 55
- printer files 120
- PROBE compiler option 256
- procedure and data-name cross-reference,
  - description 305

- PROCEDURE DIVISION
  - client 404
  - description 16
  - in subprograms 471
  - instance method 398
  - RETURNING
    - methods, use of 478
    - to return a value 16
  - statements
    - compiler-directing 19
    - conditional 18
    - delimited scope 18
    - imperative 18
  - terminology 16
  - USING
    - BY VALUE 471
    - to receive parameters 16, 469
- procedure-pointer data item
  - definition 475
  - entry address for entry point 475
  - passing parameters to callable services 475
  - rules for using 475
  - SET statement and 475
  - SYSTEM interface convention 476
  - Windows restriction 476
- process
  - definition 497
  - terminating 463
- PROCESS (CBL) statement
  - conflicting options in 226
  - description 273
  - specifying compiler options 203
- processing
  - chained lists
    - example 473
    - overview 472
  - tables
    - example using indexing 71
    - example using subscripting 70
- PROCESSING-INSTRUCTION-DATA
  - XML event 354
- PROCESSING-INSTRUCTION-TARGET
  - XML event 354
- producing XML output 373
- program
  - attribute codes 314
  - decisions
    - EVALUATE statement 81
    - IF statement 81
    - loops 91
    - PERFORM statement 91
    - switches and flags 87
  - diagnostics 309
  - entry points, call convention 239
  - limitations 537
  - main 453
  - nesting level 310
  - source code samples
    - definition file 488
    - dynamic link library 486
  - statistics 309
  - structure 5
  - subprogram 453
- PROGRAM COLLATING SEQUENCE
  - clause
    - COLLSEQ interaction 234

- PROGRAM COLLATING SEQUENCE
  - clause (*continued*)
    - does not affect national or DBCS operands 8
    - effect on alphanumeric comparisons 186
    - establishing collating sequence 8
    - no effect on DBCS comparisons 187
    - no effect on national comparisons 187
    - overridden by COLLATING SEQUENCE phrase 8
    - overrides default collating sequence 139
- PROGRAM-ID paragraph
  - coding 5
  - COMMON attribute 6
  - INITIAL attribute 6
- program-names
  - cross-reference 315
  - handling of case 255
  - specifying 5
- programs, running 217
- PRTEXIT suboption of EXIT option 241, 244

## Q

- QUOTE compiler option 257

## R

- railroad track diagrams, how to read xiv
- RANGE intrinsic function
  - example statistics calculation 56
  - example table calculation 78
- RDZvrINSTDIR environment
  - variable 217
- reading records from files
  - dynamically 128
  - randomly 128
  - sequentially 128
- record
  - description 12
  - format 119
  - size limits 122
  - TRAP runtime option, effect of 295
- RECORD CONTAINS clause
  - FILE SECTION entry 12
- RECORDING MODE clause
  - QSAM files 12
- recursive calls
  - and the LINKAGE SECTION 16
  - coding 466
  - identifying 6
- REDEFINES clause, making a record into
  - a table using 69
- redistributing COBOL for Windows
  - DLLs 217
- reentrant code 561
- reference modification
  - example 101
  - intrinsic functions 99
  - national data 100
  - out-of-range values 101
  - tables 65, 100

- reference modifier
  - arithmetic expression as 102
  - intrinsic function as, example 102
  - variables as 100
- relate items to system-names 7
- relation condition 86
- relative files
  - file access mode 121
  - organization 121
- RELEASE FROM statement
  - compared to RELEASE 136
  - example 136
- RELEASE statement
  - compared to RELEASE FROM 136
  - with SORT 136
- REM intrinsic function 56
- replacing
  - data items (INSPECT) 103
  - records in files 130
- REPLACING phrase (INSPECT), example 103
- REPOSITORY paragraph
  - class 392
  - client 404
  - coding 7
  - subclass 416
- representation
  - data 51
  - sign 50
- resolving references to DLLs 486
- restrictions
  - CICS
    - overview 327
    - separate translator 331
  - input/output procedures 137
  - OO programs 387
  - subscripting 65
- return code
  - compiler 204
  - feedback code from date and time services 551
  - files
    - example 150
    - overview 150
  - from DB2 SQL statements 323
  - RETURN-CODE special register 477, 551
- return codes, linker 214
- RETURN statement
  - required in output procedure 137
  - with INTO phrase 137
- RETURN-CODE special register
  - not set by INVOKE 408
  - passing data between programs 477
  - sharing return codes between programs 477
  - value after call to date and time service 551
- RETURNING phrase
  - CALL statement 478
  - INVOKE statement 411
  - methods, use of 478
  - PROCEDURE DIVISION header 398
- REVERSE intrinsic function 105
- reversing characters 105
- RMODE compiler option
  - performance considerations 546

- ROUNDED phrase 570
- rows in tables 63
- RSD file system 118
- RSD files
  - identifying 113
  - limitations 122
  - processing 115
- run time
  - arguments 482
  - changing file-name 10
  - differences between platforms 447
  - messages 707
  - performance considerations 537
- run unit
  - role in multithreading 497
  - terminating 463
- running OO applications 221
- running programs 217
- running under DOS 494
- runtime environment, preinitializing
  - example 509
  - overview 507
- runtime messages
  - format 707
  - incomplete or abbreviated 217
  - list of 707
  - setting file for language 197
  - setting national language 198
- runtime options
  - CHECK 293
  - CHECK(OFF) 546
  - DEBUG 294, 300
  - ERRCOUNT 294
  - FILESYS 294
  - for CICS 330
  - overview 293
  - specifying 197
  - TRAP 295
    - ON SIZE ERROR 146
  - UPSI 296

## S

- S-level error message 204, 303
- scope of names
  - effect of multithreading 498
  - global 457
  - local 457
  - program invocation instance 499
  - run unit 499
  - summary table 500
- scope terminator
  - aids in debugging 298
  - explicit 18, 19
  - implicit 19
- SD (sort description) entry, example 135
- SEARCH ALL statement
  - binary search 77
  - example 77
  - for changing an index 66
  - table must be ordered 77
- search rules for linker 212
  - example 213
- SEARCH statement
  - example 76
  - for changing an index 66
- SEARCH statement (*continued*)
  - nesting to search more than one level of a table 76
  - serial search 75
- searching
  - for name declarations 457
  - tables
    - binary search 77
    - overview 75
    - performance 75
    - serial search 75
- section
  - declarative 20
  - description of 17
  - grouping 92
- SELECT clause
  - vary input-output file 10
- SELECT OPTIONAL 125
- SELF 407
- sentence, definition 17
- separate CICS translator
  - invoking 327
  - restrictions 331
- separate sign
  - portability 40
  - printing 40
  - required for signed national decimal 40
- SEPOBJ compiler option 258
- SEQUENCE compiler option 259
- sequential files
  - file access mode 119
  - organization 119
- sequential search
  - description 75
  - example 76
- serial search
  - description 75
  - example 76
- SET command
  - defining environment variables 193
  - path search order 193
- SET condition-name TO TRUE statement
  - example 90, 92
  - switches and flags 88
- SET statement
  - for changing an index 66
  - for changing index data items 66
  - for object references 407
  - for procedure-pointer data items 475
  - for setting a condition, example 88
  - handling of program-name in 255
  - using for debugging 299
- setting
  - index data items 66
  - indexes 66
  - linker options 209
  - switches and flags 88
- sharing
  - data
    - between separately compiled programs 478
    - coding the LINKAGE SECTION 470
    - from another program 15
    - in recursive or multithreaded programs 16

- sharing (*continued*)
  - data (*continued*)
    - in separately compiled programs 15
    - overview 467
    - parameter-passing mechanisms 467
    - passing arguments to a method 408
  - PROCEDURE DIVISION header 471
  - RETURN-CODE special register 477
  - returning a value from a method 411
  - scope of names 457
  - with Java 435
- files
  - scope of names 457
  - using EXTERNAL clause 12, 479
  - using GLOBAL clause 12
- short listing, example 308
- sign condition
  - testing sign of numeric operand 86
  - using in date processing 526
- SIGN IS SEPARATE clause
  - portability 40
  - printing 40
  - required for signed national decimal data 40
- sign representation 50
- signature
  - definition of 396
  - must be unique 396
- SIZE compiler option 260
- sliding century window 516
- sort
  - alternate collating sequence 139
  - completion code 141
  - criteria 139
  - description 133
  - determining success 141
  - diagnostic message 141
  - error number
    - list of possible values 141
    - obtaining with iwzGetSortErrno 141
  - files, describing 134
  - input procedures
    - coding 136
    - example 140
  - keys
    - default 186
    - defining 139
    - overview 133
  - output procedures
    - coding 137
    - example 140
  - process 133
  - restrictions on input/output procedures 137
  - terminating 144
- SORT statement
  - ASCENDING | DESCENDING KEY phrase 139
  - COLLATING SEQUENCE phrase 8, 140
- SORT statement (*continued*)
  - description 138
  - GIVING phrase 138
  - overview 133
  - USING phrase 138
- SORT work files 200
- SORT-RETURN special register
  - determining sort or merge success 141
  - terminating sort or merge 144
- SOSI compiler option
  - description 260
  - multibyte portability 449
- SOURCE and NUMBER output, example 310
- source code
  - line number 310, 311, 314
  - listing, description 307
- SOURCE compiler option 261, 307
- SOURCE-COMPUTER paragraph 7
- SPACE compiler option 262
- special feature specification 7
- special register
  - ADDRESS 468
  - arguments in intrinsic functions 54
  - JNIEnvPtr 431
  - LENGTH OF 110, 468
  - RETURN-CODE 477
  - SORT-RETURN
    - determining sort or merge success 141
    - terminating sort or merge 144
  - WHEN-COMPILED 111
  - XML-CODE 357
  - XML-EVENT 357
  - XML-NTEXT 358
  - XML-TEXT 357
- SPECIAL-NAMES paragraph
  - coding 7
- splitting data items (UNSTRING) 95
- SQL compiler option
  - coding 323
  - description 262
  - multioption interaction 227
  - restriction in OO programs 387
- SQL statements
  - coding 322
  - return codes 323
- SQL INCLUDE 322
- use for DB2 services 321
- using binary data in 323
- SQLCA
  - declare for programs that use SQL statements 322
  - return codes from DB2 323
- SQRT intrinsic function 56
- SSRANGE compiler option
  - description 263
  - performance considerations 546
  - reference modification 101
  - turn off by using CHECK(OFF) runtime option 546
  - using 302
- stack frames, collapsing 463
- stack probes, generating 256
- stack space, allocating with cob2 command 208
- STACKSIZE statement 494
- STANDALONE-DECLARATION XML event 354
- Standard COBOL 85
  - definition xiii
- START-OF-CDATA-SECTION XML event 354
- START-OF-DOCUMENT XML event 354
- START-OF-ELEMENT XML event 354
- statement
  - compiler-directing 19
  - conditional 18
  - definition 17
  - delimited scope 18
  - explicit scope terminator 19
  - imperative 18
  - implicit scope terminator 19
- statement nesting level 310
- static data, definition of 387
- static linking
  - advantages 485
  - definition 458
  - disadvantages 485
  - overview 485
- static methods
  - definition of 387
  - invoking 421
- statistics intrinsic functions 56
- status code, files
  - example 150
  - overview 150
- STDCALL interface convention
  - argument and parameter list must match 214
  - name decoration with 476
  - restriction
    - multiple entry points 477
    - programs with arguments 476
  - specified with CALLINT 230
  - specified with ENTRYINT 239
- STL file system
  - description 115
  - return codes 116
- STL files
  - identifying 113
  - limitations 122
  - processing 115
- STOP RUN statement
  - in main program 454
  - in subprogram 454
- storage
  - for arguments 469
  - mapping 307
  - stack 256
- storing frequently used functions 485
- stride, table 542
- STRING statement
  - example 94
  - overflow condition 145
  - using 93
- strings
  - handling 93
  - Java
    - declaring 437
    - manipulating 441
    - null-terminated 472



- structure, initializing using
  - INITIALIZE 30
- structured programming 538
- structuring OO applications 428
- STUB statement 494
- subclass
  - coding
    - example 417
    - overview 414
  - instance data 416
- subprogram
  - and main program 453
  - definition 467
  - description 453
  - linkage
    - common data items 469
  - PROCEDURE DIVISION in 471
- subprograms
  - in DLLs 485
  - using 453
- subscript
  - computations 543
  - definition 64
  - literal, example 64
  - range checking 302
  - variable, example 64
- subscripting
  - definition 64
  - example 70
  - literal, example 64
  - reference modification 65
  - relative 65
  - restrictions 65
  - use data-name or literal 65
  - variable, example 64
- substitution character 162
- substrings
  - of table elements 100
  - reference modification of 99
- SUM intrinsic function, example table
  - calculation 78
- SUPER 411
- switch-status condition 86
- switches and flags
  - defining 87
  - description 87
  - resetting 88
  - setting switches off, example 89
  - setting switches on, example 88
  - testing multiple values, example 88
  - testing single values, example 87
- SYMBOLIC CHARACTERS clause 9
- symbolic constant 538
- symbols used in MAP output 312
- syntax diagrams, how to read xiv
- syntax errors
  - finding with NOCOMPILE compiler
    - option 302
- SYSADATA
  - output 227
- SYSADATA file
  - example 643
  - file contents 641
  - record descriptions 644
  - record types 642
- SYSADATA records
  - exit module called 244

- SYSADATA records (*continued*)
  - supplying modules 240
- SYSIN
  - supplying alternative modules 240
- SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH environment variables 199
- SYSLIB
  - supplying alternative modules 240
  - when not used 243
- SYSLIB environment variable 196
  - specifying location of JNL.cpy 219
- SYSPRINT
  - supplying alternative modules 240
  - when not used 244
- system date
  - under CICS 328
- SYSTEM interface convention
  - argument and parameter list must match 214
  - description 461
  - name decoration with 476
  - restriction
    - multiple entry points 477
    - programs with arguments 476
  - specified with CALLINT 230
  - specified with ENTRYINT 239
- System Properties window, defining
  - environment variables 193
- SYSTEM suboption of CALLINT compiler
  - option 230
- system-name 7
- SYSTEM data set
  - sending messages to 264

## T

- table
  - assigning values to 69
  - columns 61
  - compare to array 37
  - defining with OCCURS clause 61
  - definition 61
  - depth 63
  - description 37
  - dynamically loading 67
  - efficient coding 541, 542
  - elements 61
  - identical element specifications 541
  - index, definition 64
  - initializing
    - all occurrences of an element 70
    - at the group level 70
    - each item individually 69
    - using INITIALIZE 67
    - using PERFORM VARYING 91
  - loading values in 67
  - looping through 91
  - multidimensional 62
  - one-dimensional 61
  - processing with intrinsic
    - functions 78
  - redefining a record as 69
  - reference modification 65
  - referencing substrings of
    - elements 100
  - referencing with indexes, example 64
- table (*continued*)
  - referencing with subscripts,
    - example 64
  - referring to elements 64
  - rows 63
  - searching
    - binary 77
    - overview 75
    - performance 75
    - sequential 75
    - serial 75
  - stride computation 542
  - subscript, definition 64
  - three-dimensional 63
  - two-dimensional 63
  - variable-length
    - creating 72
    - example of loading 74
    - initializing 75
    - preventing overlay in 581
- TALLYING phrase (INSPECT),
  - example 103
- TEMPMEM environment variable 196
- temporary work-file location
  - specifying with TMP 200
- TERMINAL compiler option 264
- terminal, sending messages to 264
- terms used in MAP output 312
- test
  - conditions 91
  - data 86
  - numeric operand 86
  - UPSI switch 86
- TEST AFTER 91
- TEST BEFORE 91
- TEST compiler option
  - description 264
  - multioption interaction 227
  - performance considerations 546
  - use for debugging 306
- THREAD compiler option
  - and the LINKAGE SECTION 16
  - description 265
  - for Java interoperability 219
  - for OO COBOL 219
  - performance considerations 546
- thread environment requirements 256
- threading
  - and preinitialization 501
  - control transfer 501
  - ending programs 501
- three-digit years 555
- time information, formatting 198
- time-zone information
  - specifying with TZ 200
- time, getting local (CEELOCT) 609
- timestamp 594, 618
- TITLE statement
  - controlling header on listing 6
- TMP environment variable 200
- top-down programming
  - constructs to avoid 538
- trademarks 754
- transferring control
  - between COBOL programs 454
  - called program 453
  - calling program 453

- transferring control (*continued*)
  - main and subprograms 453
  - nested programs 455
- transforming COBOL data to XML
  - example 375
  - overview 373
- translating CICS into COBOL 327
- TRAP runtime option
  - description 295
  - ON SIZE ERROR 146
- TRUNC compiler option
  - description 266
  - performance considerations 546
- tuning considerations, performance 545, 546
- two-digit years
  - querying within 100-year range (CEEQCEN) 611
    - example 612
  - setting within 100-year range (CEESCEN) 613
    - example 614
  - valid values for 555
- typed object references 405
- TZ environment variable 200

## U

- U-level error message 204, 303
- UNDATE intrinsic function
  - example 530
  - using 529
- Unicode
  - description 159
  - encoding 167
  - JNI services 441
  - processing data 155
- universal object references 406
- UNKNOWN-REFERENCE-IN-ATTRIBUTE XML event 354
- UNKNOWN-REFERENCE-IN-CONTENT XML event 354
- UNSTRING statement
  - example 96
  - overflow condition 145
  - using 95
- UPPER suboption of PGMNAME 256
- UPPER-CASE intrinsic function 105
- uppercase, converting to 105
- UPSI runtime options 296
- UPSI switches, setting 296
- USAGE clause
  - at the group level 25
  - incompatible data 51
  - INDEX phrase, creating index data items with 66
  - NATIONAL phrase at the group level 164
  - OBJECT REFERENCE 405
- USE FOR DEBUGGING declarative 294
- USE FOR DEBUGGING declaratives 299
- user-defined condition 86
- user-exit work area 241
- USING phrase
  - INVOKE statement 408
  - PROCEDURE DIVISION header 398, 471

- UTF-16
  - definition 159
  - encoding for national data 159
- UTF-8
  - converting to or from national 171
  - definition 159
  - encoding for ASCII invariant characters 159
  - JNI services 441
  - processing data items 171

## V

- valid data
  - numeric 51
- VALUE clause
  - alphanumeric literal with national data, example 109
  - alphanumeric literal with national group, example 70
  - assigning table values
    - at the group level 70
    - to each item individually 69
    - to each occurrence of an element 70
  - assigning to a variable-length group 75
  - cannot use for external floating point 44
  - initializing internal floating-point literals 40
  - large literals with COMP-5 45
  - large, with TRUNC(BIN) 267
- VALUE IS NULL 472
- VALUE OF clause 12
- variable
  - as reference modifier 100
  - definition 23
- variable parameter list for C/C++ 462
- variable-length records
  - OCCURS DEPENDING ON (ODO) clause 542
- variable-length table
  - assigning values to 75
  - creating 72
  - example 73
  - example of loading 74
  - preventing overlay in 581
- variables, environment
  - assignment-name 196
  - CLASSPATH 197
  - COBCPYEXT 195
  - COBJVMINITOPTIONS 197
  - COBLSTDIR 195
  - COBMSGs 197
  - COBOPT 195
  - COBPATH
    - CICS dynamic calls 329
    - description 195, 197
  - COBRTOPT 197
  - compiler 195
  - definition 193
  - EBCDIC\_CODEPAGE 197
  - ILINK 209
  - LANG 198
  - LC\_ALL 198
  - LC\_COLLATE 198

variables, environment (*continued*)

- LC\_CTYPE 198
- LC\_MESSAGES 198
- LC\_TIME 198
- LIB 196
- library-name 195, 275
- linker 196
- LOCPATH 198
- NLSPATH 198
- PATH 199
- runtime 196
- setting
  - for COBOL for Windows 194
  - locale 181
  - SET command 193
  - System Properties window 193
- SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 199
- SYSLIB 196
- TEMPMEM 196
- text-name 195, 275
- TMP 200
- TZ 200
- variably located data item 579
- variably located group 579
- VBREF compiler option
  - description 269
  - output example 316
  - using 307
- verb cross-reference listing
  - description 307
- verbs used in program 307
- VERSION statement 495
- VERSION-INFORMATION XML event 355

## W

- W-level message 204, 303
- WHEN phrase
  - EVALUATE statement 83
  - SEARCH ALL statement 77
  - SEARCH statement 75
- WHEN-COMPILED intrinsic function 111
- WHEN-COMPILED special register 111
- windowed date fields
  - contracting 533
- WITH DEBUGGING MODE clause 294
  - for debugging lines 300
  - for debugging statements 300
- WITH POINTER phrase
  - STRING 93
  - UNSTRING 95
- WORKING-STORAGE SECTION
  - client 405, 406
  - comparison with LOCAL-STORAGE example 14
  - OO client 406
  - overview 13
  - factory data 419
  - initializing 269
  - instance data 394, 416
  - instance method 397
  - multithreading considerations 406

- workstation and workstation COBOL
  - differences from host 561
- wrapper, definition of 427
- wrapping procedure-oriented
  - programs 427
- writing compatible code 451
- WSCLEAR compiler option 269

## X

- X delimiter for control characters in
  - alphanumeric literals 26
- XML declaration
  - restriction 352
  - specifying encoding declaration 366
- XML document
  - accessing 351
  - basic document encoding 364
  - coded character sets 365
  - controlling the encoding of 383
  - document encoding declaration 364
  - effect of CHAR(EBCDIC) 351
  - encoding 364
  - enhancing
    - example of converting hyphens in
      - element names to
        - underscores 382
    - example of modifying data
      - definitions 380
    - rationale and techniques 379
  - external code page 364
  - generating
    - example 375
    - overview 373
  - handling parsing exceptions 366
  - national language 364
  - parser 349
  - parsing
    - description 351
    - example 359
  - processing 349
  - specifying code page 365
  - XML declaration 352
- XML event
  - ATTRIBUTE-CHARACTER 353
  - ATTRIBUTE-CHARACTERS 353
  - ATTRIBUTE-NAME 353
  - ATTRIBUTE-NATIONAL-CHARACTER 353
  - COMMENT 353
  - CONTENT-CHARACTER 353
  - CONTENT-CHARACTERS 353
  - CONTENT-NATIONAL-CHARACTER 353
  - DOCUMENT-TYPE-DECLARATION 353
  - ENCODING-DECLARATION 354
  - END-OF-CDATA-SECTION 354
  - END-OF-DOCUMENT 354
  - END-OF-ELEMENT 354
  - EXCEPTION 354
  - PROCESSING-INSTRUCTION-DATA 354
  - PROCESSING-INSTRUCTION-TARGET 354
  - STANDALONE-DECLARATION 354
  - START-OF-CDATA-SECTION 354

- XML event (*continued*)
  - START-OF-DOCUMENT 354
  - START-OF-ELEMENT 354
  - UNKNOWN-REFERENCE-IN-ATTRIBUTE 354
  - UNKNOWN-REFERENCE-IN-CONTENT 354
  - VERSION-INFORMATION 355
- XML events
  - description 349
  - processing 352
  - processing procedure 351
- XML exception codes
  - for generating 634
  - for parsing
    - handleable 625
    - not handleable 629
- XML GENERATE statement
  - COUNT IN 384
  - NOT ON EXCEPTION 375
  - ON EXCEPTION 384
- XML generation
  - counting generated characters 374
  - description 373
  - enhancing output
    - example of converting hyphens in
      - element names to
        - underscores 382
    - example of modifying data
      - definitions 380
    - rationale and techniques 379
  - example 375
  - handling errors 384
  - ignored data items 374
  - overview 373
- XML output
  - controlling the encoding of 383
  - enhancing
    - example of converting hyphens in
      - element names to
        - underscores 382
    - example of modifying data
      - definitions 380
    - rationale and techniques 379
  - generating
    - example 375
    - overview 373
- XML PARSE statement
  - NOT ON EXCEPTION 367
  - ON EXCEPTION 367
  - overview 349
  - using 351
- XML parser
  - conformance 632
  - error handling 367
  - overview 349
- XML parsing
  - control flow with processing
    - procedure 361
  - description 351
  - effect of CHAR(EBCDIC) 351
  - example 355
  - handling CCSID conflicts 370
  - handling code-page conflicts 370
  - handling exceptions 366
  - overview 349
  - special registers 357

- XML parsing (*continued*)
  - terminating 371
  - XML declaration 352
- XML processing procedure
  - control flow with parser 361
  - error with EXIT PROGRAM or GOBACK 358
  - example 359
  - handling parsing exceptions 366
  - restriction on XML PARSE 358
  - specifying 351
  - using special registers 357
  - with code-page conflicts 370
  - writing 357
- XML-CODE special register
  - content 361
  - control flow between parser and
    - processing procedure 361
  - description 357
  - exception codes for generating 634
  - exception codes for parsing
    - encoding conflicts 366
    - handleable 625
    - not handleable 629
  - terminating parsing with 371
  - using in generating 375
  - using in parsing 349
  - with code-page conflicts 370
  - with generating exceptions 384
  - with parsing exceptions 367
- XML-EVENT special register
  - content 352
  - description 357
  - example of using 355
  - using 349, 352
  - with parsing exceptions 367
- XML-NTEXT special register
  - content 363
  - description 358
  - using 349
  - with parsing exceptions 367
- XML-TEXT special register
  - content 363
  - description 357
  - encoding 358
  - using 349
  - with parsing exceptions 367
- XREF compiler option
  - description 269
  - finding data- and
    - procedure-names 305
- XREF output
  - data-name cross-references 314
  - program-name cross-references 315

## Y

- year field expansion 518
- year windowing
  - advantages 516
  - how to control 528
  - MLE approach 516
  - when not supported 522
- year-last date fields 520
- YEARWINDOW compiler option
  - description 271

## Z

- zero comparison 526
- zero suppression
  - example of BLANK WHEN ZERO clause 41
  - PICTURE symbol Z 41
- zoned decimal data (USAGE DISPLAY)
  - effect of ZWB on comparison to alphanumeric 271
  - example 39
  - format 43
  - sign representation 50
- zSeries host data format
  - considerations 567
- ZWB compiler option 271

---

## Readers' Comments — We'd Like to Hear from You

COBOL for Windows  
Programming Guide  
Version 7.5

Publication No. SC23-8559-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Send your comments to the address on the reverse side of this form.

If you would like a response from IBM, please fill in the following information:

\_\_\_\_\_  
Name

\_\_\_\_\_  
Address

\_\_\_\_\_  
Company or Organization

\_\_\_\_\_  
Phone No.

\_\_\_\_\_  
E-mail address



Cut or Fold  
Along Line

Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation  
Reader Comments  
DTX/E269  
555 Bailey Avenue  
San Jose, CA  
United States of America 95141-9989



Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold  
Along Line





Program Number: 5724-T07

Printed in USA

SC23-8559-00

